

Isabelle/HOL — Higher-Order Logic

September 11, 2023

Contents

1	Loading the code generator and related modules	32
2	The basis of Higher-Order Logic	33
2.1	Primitive logic	34
2.1.1	Core syntax	34
2.1.2	Defined connectives and quantifiers	35
2.1.3	Additional concrete syntax	35
2.1.4	Axioms and basic definitions	37
2.2	Fundamental rules	38
2.2.1	Equality	38
2.2.2	Congruence rules for application	38
2.2.3	Equality of booleans – iff	39
2.2.4	True (1)	39
2.2.5	Universal quantifier (1)	39
2.2.6	False	39
2.2.7	Negation	40
2.2.8	Implication	40
2.2.9	Disjunction (1)	41
2.2.10	Derivation of <i>iffI</i>	41
2.2.11	True (2)	41
2.2.12	Universal quantifier (2)	42
2.2.13	Existential quantifier	42
2.2.14	Conjunction	42
2.2.15	Disjunction (2)	42
2.2.16	Classical logic	43
2.2.17	Unique existence	43
2.2.18	Classical intro rules for disjunction and existential quantifiers	44
2.2.19	Intuitionistic Reasoning	45
2.2.20	Atomizing meta-level connectives	46
2.2.21	Atomizing elimination rules	47

2.3	Package setup	47
2.3.1	Sledgehammer setup	48
2.3.2	Classical Reasoner setup	48
2.3.3	THE: definite description operator	50
2.3.4	Simplifier	51
2.3.5	Generic cases and induction	62
2.3.6	Coherent logic	65
2.3.7	Reorienting equalities	65
2.4	Other simple lemmas and lemma duplicates	66
2.5	Basic ML bindings	67
3	<i>NO-MATCH simproc</i>	69
3.1	Code generator setup	70
3.1.1	Generic code generator preprocessor setup	70
3.1.2	Equality	71
3.1.3	Generic code generator foundation	71
3.1.4	Generic code generator target languages	73
3.1.5	Evaluation and normalization by evaluation	74
3.2	Counterexample Search Units	75
3.2.1	Quickcheck	75
3.2.2	Nitpick setup	75
3.3	Preprocessing for the predicate compiler	75
3.4	Legacy tactics and ML bindings	75
4	Abstract orderings	76
4.1	Abstract ordering	76
4.2	Syntactic orders	80
4.3	Quasi orders	80
4.4	Partial orders	82
4.5	Linear (total) orders	84
4.6	Reasoning tools setup	86
4.7	Bounded quantifiers	90
4.8	Transitivity reasoning	92
4.9	min and max – fundamental	98
4.10	(Unique) top and bottom elements	98
4.11	Dense orders	100
4.12	Wellorders	101
4.13	Order on <i>bool</i>	103
4.14	Order on $- \Rightarrow -$	104
4.15	Order on unary and binary predicates	106
4.16	Name duplicates	107

5	Groups, also combined with orderings	108
5.1	Dynamic facts	108
5.2	Abstract structures	108
5.3	Generic operations	111
5.4	Semigroups and Monoids	112
5.5	Groups	116
5.6	(Partially) Ordered Groups	120
5.7	Support for reasoning about signs	122
5.8	Canonically ordered monoids	135
5.9	Tools setup	137
6	Abstract lattices	138
6.1	Abstract semilattice	138
6.2	Syntactic infimum and supremum operations	141
6.3	Concrete lattices	141
6.3.1	Intro and elim rules	142
6.3.2	Equational laws	143
6.3.3	Strict order	146
6.4	Distributive lattices	146
6.5	Bounded lattices	147
6.6	<i>min/max</i> as special case of lattice	149
6.7	Uniqueness of inf and sup	150
6.8	Lattice on $- \Rightarrow -$	151
7	Boolean Algebras	152
7.1	Abstract boolean algebra	152
7.1.1	Complement	153
7.1.2	Conjunction	154
7.1.3	Disjunction	155
7.1.4	De Morgan's Laws	155
7.2	Symmetric Difference	156
7.3	Type classes	158
7.4	Lattice on <i>bool</i>	160
7.5	Lattice on unary and binary predicates	161
7.6	Simproc setup	162
8	Set theory for higher-order logic	164
8.1	Sets as predicates	164
8.2	Subsets and bounded quantifiers	166
8.3	Basic operations	172
8.3.1	Subsets	172
8.3.2	Equality	173
8.3.3	The empty set	174
8.3.4	The universal set – UNIV	174

8.3.5	The Powerset operator – Pow	175
8.3.6	Set complement	176
8.3.7	Binary intersection	176
8.3.8	Binary union	176
8.3.9	Set difference	177
8.3.10	Augmenting a set – <i>insert</i>	178
8.3.11	Singletons, using insert	179
8.3.12	Image of a set under a function	180
8.3.13	Some rules with <i>if</i>	183
8.4	Further operations and lemmas	184
8.4.1	The “proper subset” relation	184
8.4.2	Derived rules involving subsets.	185
8.4.3	Equalities involving union, intersection, inclusion, etc.	186
8.4.4	Monotonicity of various operations	196
8.4.5	Inverse image of a function	197
8.4.6	Singleton sets	199
8.4.7	Getting the contents of a singleton set	199
8.4.8	Monad operation	199
8.4.9	Operations for execution	200
9	HOL type definitions	204
10	Notions about functions	206
10.1	The Identity Function <i>id</i>	207
10.2	The Composition Operator $f \circ g$	207
10.3	The Forward Composition Operator <i>fcomp</i>	208
10.4	Mapping functions	209
10.5	Injectivity and Bijectivity	209
10.5.1	Inj/surj/bij of Algebraic Operations	221
10.6	Function Updating	223
10.7	<i>override-on</i>	225
10.8	Inversion of injective functions	225
10.9	Monotonicity	226
10.9.1	Specializations For <i>ord</i> Type Class And More	227
10.9.2	Least value operator	234
10.10	Setup	234
10.10.1	Proof tools	234
10.10.2	Functorial structure of types	235
11	Complete lattices	235
11.1	Syntactic infimum and supremum operations	235
11.2	Abstract complete lattices	237
11.3	Complete lattice on <i>bool</i>	248
11.4	Complete lattice on $- \Rightarrow -$	248

11.5	Complete lattice on unary and binary predicates	249
11.6	Complete lattice on - <i>set</i>	251
11.6.1	Inter	251
11.6.2	Intersections of families	252
11.6.3	Union	254
11.6.4	Unions of families	256
11.6.5	Distributive laws	259
11.7	Injections and bijections	260
11.7.1	Complement	261
11.7.2	Miniscoping and maxiscoping	262
12	Wrapping Existing Freely Generated Type's Constructors	263
13	Knaster-Tarski Fixpoint Theorem and inductive definitions	264
13.1	Least fixed points	264
13.2	General induction rules for least fixed points	265
13.3	Greatest fixed points	267
13.4	Coinduction rules for greatest fixed points	268
13.5	Even Stronger Coinduction Rule, by Martin Coen	269
13.6	Rules for fixed point calculus	270
13.7	Inductive predicates and sets	272
13.8	The Schroeder-Bernstein Theorem	273
13.9	Inductive datatypes and primitive recursion	274
14	Cartesian products	275
14.1	<i>bool</i> is a datatype	275
14.2	The <i>unit</i> type	276
14.3	The product type	279
14.3.1	Type definition	279
14.3.2	Tuple syntax	280
14.3.3	Code generator setup	282
14.3.4	Fundamental operations and properties	283
14.3.5	Derived operations	290
14.4	Simproc for rewriting a set comprehension into a pointfree expression	300
14.5	Lemmas about disjointness	301
14.6	Inductively defined sets	301
14.7	Legacy theorem bindings and duplicates	302
15	The Disjoint Sum of Two Types	302
15.1	Construction of the sum type and its basic abstract operations	302
15.2	Projections	305
15.3	The Disjoint Sum of Sets	306

16 Rings	307
16.1 Semirings and rings	307
16.2 Abstract divisibility	310
16.3 Towards integral domains	316
16.4 (Partial) Division	321
16.5 Quotient and remainder in integral domains	343
16.6 Interlude: basic tool support for algebraic and arithmetic calculations	345
16.7 Ordered semirings and rings	345
16.8 Dioids	363
17 Natural numbers	364
17.1 Type <i>ind</i>	364
17.2 Type <i>nat</i>	364
17.3 Arithmetic operators	368
17.3.1 Addition	370
17.3.2 Difference	370
17.3.3 Multiplication	371
17.4 Orders on <i>nat</i>	372
17.4.1 Operation definition	372
17.4.2 Introduction properties	374
17.4.3 Elimination properties	374
17.4.4 Inductive (?) properties	375
17.4.5 Monotonicity of Addition	379
17.4.6 <i>min</i> and <i>max</i>	381
17.4.7 Additional theorems about (\leq)	383
17.4.8 More results about difference	388
17.4.9 Monotonicity of multiplication	390
17.5 Natural operation of natural numbers on functions	393
17.6 Kleene iteration	396
17.7 Embedding of the naturals into any <i>semiring-1: of-nat</i>	398
17.8 The set of natural numbers	402
17.9 Further arithmetic facts concerning the natural numbers	403
17.9.1 Greatest operator	411
17.10 Monotonicity of <i>funpow</i>	411
17.11 The divides relation on <i>nat</i>	412
17.12 Aliasses	414
17.13 Size of a datatype value	415
17.14 Code module namespace	415
18 Fields	416
18.1 Division rings	416
18.2 Fields	422
18.3 Ordered fields	427

19 Relations – as sets of pairs, and binary predicates	443
19.1 Fundamental	444
19.1.1 Relations as sets of pairs	444
19.1.2 Conversions between set and predicate relations	444
19.2 Properties of relations	446
19.2.1 Reflexivity	446
19.2.2 Irreflexivity	448
19.2.3 Asymmetry	450
19.2.4 Symmetry	451
19.2.5 Antisymmetry	453
19.2.6 Transitivity	455
19.2.7 Totality	458
19.2.8 Single valued relations	459
19.3 Relation operations	460
19.3.1 The identity relation	460
19.3.2 Diagonal: identity over a set	461
19.3.3 Composition	462
19.3.4 Converse	464
19.3.5 Domain, range and field	468
19.3.6 Image of a set under a relation	471
19.3.7 Inverse image	473
19.3.8 Powerset	473
20 Finite sets	474
20.1 Predicate for finite sets	474
20.1.1 Choice principles	475
20.1.2 Finite sets are the images of initial segments of natural numbers	475
20.2 Finiteness and common set operations	477
20.3 Further induction rules on finite sets	486
20.4 Class <i>finite</i>	488
20.5 A basic fold functional for finite sets	489
20.5.1 From <i>fold-graph</i> to <i>fold</i>	491
20.5.2 Liftings to <i>comp-fun-commute-on</i> etc.	497
20.5.3 <i>UNIV</i> as carrier set	498
20.5.4 Expressing set operations via <i>fold</i>	499
20.5.5 Expressing relation operations via <i>fold</i>	504
20.6 Locales as mini-packages for fold operations	506
20.6.1 The natural case	506
20.6.2 With idempotency	507
20.6.3 <i>UNIV</i> as the carrier set	507
20.7 Finite cardinality	508
20.7.1 Cardinality of image	520
20.7.2 Pigeonhole Principles	522

20.7.3	Cardinality of sums	523
20.8	Minimal and maximal elements of finite sets	524
20.8.1	Finite orders	530
20.8.2	Relating injectivity and surjectivity	531
20.9	Infinite Sets	532
20.10	The finite powerset operator	535
21	Reflexive and Transitive closure of a relation	535
21.1	Reflexive closure	537
21.2	Reflexive-transitive closure	537
21.3	Transitive closure	542
21.4	Symmetric closure	550
21.5	The power operation on relations	551
21.6	Bounded transitive closure	560
21.7	Acyclic relations	561
21.8	Setup of transitivity reasoner	562
22	Well-founded Recursion	563
22.1	Basic Definitions	564
22.2	Basic Results	565
22.2.1	Minimal-element characterization of well-foundedness	566
22.2.2	Well-foundedness of transitive closure	566
22.2.3	Well-foundedness of image	569
22.3	Well-Foundedness Results for Unions	569
22.4	Well-Foundedness of Composition	572
22.5	Acyclic relations	573
22.5.1	Wellfoundedness of finite acyclic relations	574
22.6	<i>nat</i> is well-founded	574
22.7	Accessible Part	576
22.8	Tools for building wellfounded relations	578
22.8.1	Conversion to a known well-founded relation	579
22.8.2	Measure functions into <i>nat</i>	579
22.8.3	Lexicographic combinations	579
22.8.4	Bounded increase must terminate	584
23	Well-Founded Recursion Combinator	584
23.0.1	Well-founded recursion via genuine fixpoints	586
23.1	Wellfoundedness of <i>same-fst</i>	586
24	Orders as Relations	587
24.1	Orders on a set	587
24.2	Orders on the field	588
24.3	Relations given by a predicate and the field	590
24.4	Orders on a type	590

24.5	Order-like relations	591
24.5.1	Auxiliaries	591
24.5.2	The upper and lower bounds operators	592
24.6	Variations on Well-Founded Relations	595
24.6.1	Characterizations of well-foundedness	595
24.6.2	Characterizations of well-foundedness	598
25	Hilbert's Epsilon-Operator and the Axiom of Choice	599
25.1	Hilbert's epsilon	599
25.2	Hilbert's Epsilon-operator	600
25.3	Axiom of Choice, Proved Using the Description Operator	601
25.4	Function Inverse	602
25.5	Other Consequences of Hilbert's Epsilon	612
25.6	An aside: bounded accessible part	614
25.7	More on injections, bijections, and inverses	616
25.8	Specification package – Hilbertized version	618
25.9	Complete Distributive Lattices – Properties depending on Hilbert Choice	619
26	Zorn's Lemma and the Well-ordering Theorem	625
26.1	Zorn's Lemma for the Subset Relation	625
26.1.1	Results that do not require an order	625
26.1.2	Hausdorff's Maximum Principle	631
26.1.3	Results for the proper subset relation	631
26.1.4	Zorn's lemma	632
26.2	Zorn's Lemma for Partial Orders	633
26.3	Other variants of Zorn's Lemma	634
26.4	The Well Ordering Theorem	637
27	Well-Order Relations as Needed by Bounded Natural Func- tors	643
27.1	Auxiliaries	644
27.2	Well-founded induction and recursion adapted to non-strict well-order relations	644
27.3	The notions of maximum, minimum, supremum, successor and order filter	645
27.3.1	Properties of max2	646
27.3.2	Existence and uniqueness for isMinim and well-definedness of minim	647
27.3.3	Properties of minim	648
27.3.4	Properties of successor	648
27.3.5	Properties of order filters	650
27.3.6	Other properties	652

28 Well-Order Embeddings as Needed by Bounded Natural Functors	654
28.1 Auxiliaries	654
28.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions	655
28.3 Given any two well-orders, one can be embedded in the other	663
28.4 Uniqueness of embeddings	670
28.5 More properties of embeddings, strict embeddings and iso- morphisms	672
29 Constructions on Wellorders as Needed by Bounded Natural Functors	678
29.1 Restriction to a set	678
29.2 Order filters versus restrictions and embeddings	680
29.3 The strict inclusion on proper ofilters is well-founded	683
29.4 Ordering the well-orders by existence of embeddings	684
29.5 $< o$ is well-founded	694
29.6 Copy via direct images	695
29.7 Bounded square	698
30 Cardinal-Order Relations as Needed by Bounded Natural Functors	711
30.1 Cardinal orders	711
30.2 Cardinal of a set	713
30.3 Cardinals versus set operations on arbitrary sets	717
30.4 Cardinals versus set operations involving infinite sets	726
30.5 The cardinal ω and the finite cardinals	732
30.5.1 First as well-orders	733
30.5.2 Then as cardinals	734
30.6 The successor of a cardinal	735
30.7 Regular cardinals	742
30.8 Others	744
30.9 Regular vs. stable cardinals	745
31 Cardinal Arithmetic as Needed by Bounded Natural Functors	750
31.1 Zero	751
31.2 (In)finite cardinals	752
31.3 Binary sum	753
31.4 One	755
31.5 Two	755
31.6 Family sum	756
31.7 Product	756
31.8 Exponentiation	759

32 Function Definition Base	767
33 Definition of Bounded Natural Functors	768
34 Composition of Bounded Natural Functors	773
35 Registration of Basic Types as Bounded Natural Functors	778
36 Shared Fixpoint Operations on Bounded Natural Functors	784
37 Least Fixpoint (Datatype) Operation on Bounded Natural Functors	790
38 Equivalence Relations in Higher-Order Set Theory	795
38.1 Equivalence relations – set version	795
38.2 Equivalence classes	796
38.3 Quotients	796
38.4 Refinement of one equivalence relation WRT another	798
38.5 Defining unary operations upon equivalence classes	798
38.6 Defining binary operations upon equivalence classes	799
38.7 Quotients and finiteness	801
38.8 Projection	802
38.9 Equivalence relations – predicate version	803
38.10 Equivalence closure	805
39 MESON Proof Method	809
39.1 Negation Normal Form	809
39.2 Pulling out the existential quantifiers	810
39.3 Lemmas for Forward Proof	811
39.4 Clausification helper	811
39.5 Skolemization helpers	812
39.6 Meson package	813
40 Automatic Theorem Provers (ATPs)	813
40.1 ATP problems and proofs	813
40.2 Higher-order reasoning helpers	813
40.3 Basic connection between ATPs and HOL	816
41 Metis Proof Method	816
41.1 Literal selection and lambda-lifting helpers	816
41.2 Metis package	817
42 Generic theorem transfer using relations	817
42.1 Relator for function space	817
42.2 Transfer method	818

42.3	Predicates on relations, i.e. “class constraints”	819
42.4	Properties of relators	823
42.5	Transfer rules	825
42.6	<i>of-bool</i> and <i>of-nat</i>	831
43	Lifting package	831
43.1	Function map	831
43.2	Quotient Predicate	832
43.3	Quotient composition	835
43.4	Respects predicate	835
43.5	Domains	841
43.6	ML setup	842
44	Definition of Quotient Types	843
44.1	Quotient Predicate	843
44.2	lemmas for regularisation of ball and bex	847
44.3	Bounded abstraction	849
44.4	<i>Bex1-rel</i> quantifier	851
44.5	Various respects and preserve lemmas	852
44.6	Quotient composition	854
44.7	Quotient3 to Quotient	856
44.8	ML setup	856
44.9	Methods / Interface	858
45	Lifting of BNFs	859
46	Binary Numerals	862
46.1	The <i>num</i> type	862
46.2	N numeral operations	863
46.3	Binary numerals	866
46.4	Class-specific numeral rules	868
46.4.1	Structures with addition: class <i>numeral</i>	868
46.4.2	Structures with negation: class <i>neg-numeral</i>	869
46.4.3	Structures with multiplication: class <i>semiring-numeral</i>	872
46.4.4	Structures with a zero: class <i>semiring-1</i>	873
46.4.5	Equality: class <i>semiring-char-0</i>	873
46.4.6	Comparisons: class <i>linordered-nonzero-semiring</i>	874
46.4.7	Multiplication and negation: class <i>ring-1</i>	876
46.4.8	Equality using <i>iszero</i> for rings with non-zero characteristic	877
46.4.9	Equality and negation: class <i>ring-char-0</i>	878
46.4.10	Structures with negation and order: class <i>linordered-idom</i>	879
46.4.11	Natural numbers	882
46.5	Particular lemmas concerning $2::'a$	886

46.6	N numeral equations as default simplification rules	886
46.6.1	Special Simplification for Constants	886
46.6.2	Optional Simplification Rules Involving Constants	888
46.7	Setting up simprocs	889
46.7.1	Simplification of arithmetic operations on integer constants	889
46.7.2	Simplification of arithmetic when nested to the right	891
46.8	Code module namespace	891
46.9	Printing of evaluated natural numbers as numerals	891
46.10	More on auxiliary conversion	892
47	Exponentiation	893
47.1	Powers for Arbitrary Monoids	893
47.2	Exponentiation on ordered types	901
47.3	Miscellaneous rules	910
47.4	Exponentiation for the Natural Numbers	911
47.4.1	Cardinality of the Powerset	913
47.5	Code generator tweak	914
48	Big sum and product over finite (non-empty) sets	914
48.1	Generic monoid operation over a set	914
48.1.1	Standard sum or product indexed by a finite set	914
48.1.2	HOL Light variant: sum/product indexed by the non-neutral subset	925
48.2	Generalized summation over a set	927
48.2.1	Properties in more restricted classes of structures	928
48.2.2	Cardinality as special case of <i>sum</i>	936
48.2.3	Cardinality of products	942
48.3	Generalized product over a set	942
48.3.1	Properties in more restricted classes of structures	943
49	Chain-complete partial orders and their fixpoints	950
49.1	Chains	950
49.2	Chain-complete partial orders	950
49.3	Transfinite iteration of a function	951
49.4	Fixpoint combinator	952
49.5	Fixpoint induction	953
50	Datatype option	957
50.0.1	Operations	958
50.1	Transfer rules for the Transfer package	963
50.1.1	Interaction with finite sets	964
50.1.2	Code generator setup	964

51 Partial Function Definitions	965
51.1 Axiomatic setup	966
51.2 Flat interpretation: tailrec and option	970
52 Reconstructing external resolution proofs for propositional logic	975
53 Function Definitions and Termination Proofs	975
53.1 Definitions with default value	975
53.2 Measure functions	977
53.3 Congruence rules	978
53.4 Simp rules for termination proofs	978
53.5 Decomposition	978
53.6 Reduction pairs	978
53.7 Concrete orders for SCNP termination proofs	979
53.8 Yet another induction principle on the natural numbers	981
53.9 Tool setup	982
54 The Integers as Equivalence Classes over Pairs of Natural Numbers	982
54.1 Definition of integers as a quotient type	982
54.2 Integers form a commutative ring	982
54.3 Integers are totally ordered	984
54.4 Ordering properties of arithmetic operations	984
54.5 Embedding of the Integers into any <i>ring-1: of-int</i>	986
54.6 Magnitude of an Integer, as a Natural Number: <i>nat</i>	992
54.7 Lemmas about the Function <i>of-nat</i> and Orderings	996
54.8 Cases and induction	997
54.8.1 Binary comparisons	999
54.8.2 Comparisons, for Ordered Rings	999
54.9 The Set of Integers	1000
54.10 <i>sum</i> and <i>prod</i>	1003
54.11 Setting up simplification procedures	1004
54.12 More Inequality Reasoning	1005
54.13 The functions <i>nat</i> and <i>int</i>	1006
54.14 Induction principles for int	1008
54.15 Intermediate value theorems	1010
54.16 Products and 1, by T. M. Rasmussen	1011
54.17 The divides relation	1012
54.18 Powers with integer exponents	1017
54.19 Finiteness of intervals	1024
54.20 Configuration of the code generator	1024
54.21 Duplicates	1028

55 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets	1028
55.1 Generic lattice operations over a set	1028
55.1.1 Without neutral element	1028
55.1.2 With neutral element	1032
55.2 Lattice operations on finite sets	1034
55.3 Infimum and Supremum over non-empty sets	1035
55.4 Minimum and Maximum over non-empty sets	1037
55.5 Arg Min	1046
55.6 Arg Max	1048
56 Division in euclidean (semi)rings	1050
56.1 Euclidean (semi)rings with explicit division and remainder . .	1050
56.2 Euclidean (semi)rings with cancel rules	1053
56.3 Uniquely determined division	1062
56.4 Division on <i>nat</i>	1067
56.5 Division on <i>int</i>	1084
56.5.1 Basic instantiation	1084
56.5.2 Algebraic foundations	1085
56.5.3 Basic conversions	1086
56.5.4 Euclidean division	1088
56.5.5 Trivial reduction steps	1090
56.5.6 More uniqueness rules	1091
56.5.7 Laws for unary minus	1092
56.5.8 Borders	1093
56.5.9 Splitting Rules for div and mod	1094
56.5.10 Algebraic rewrites	1095
56.5.11 Distributive laws for conversions.	1097
56.5.12 Monotonicity in the First Argument (Dividend)	1097
56.5.13 Monotonicity in the Second Argument (Divisor)	1098
56.5.14 Quotients of Signs	1098
56.5.15 Further properties	1102
56.5.16 Computing <i>div</i> and <i>mod</i> by shifting	1103
56.6 Code generation	1105
57 Parity in rings and semirings	1106
57.1 Ring structures with parity and <i>even/odd</i> predicates	1106
57.2 Instance for <i>nat</i>	1110
57.3 Parity and powers	1113
57.4 Instance for <i>int</i>	1116
57.5 Special case: euclidean rings containing the natural numbers	1117
57.6 Generic symbolic computations	1126
57.6.1 Computation by simplification	1132
57.7 Computing congruences modulo 2^q	1134

58 Combination and Cancellation Simprocs for Numeral Expressions	1135
59 Semiring normalization	1141
60 Groebner bases	1145
60.1 Groebner Bases	1145
61 Set intervals	1147
61.1 Various equivalences	1149
61.2 Logical Equivalences for Set Inclusion and Equality	1149
61.3 Two-sided intervals	1150
61.3.1 Emptiness, singletons, subset	1151
61.4 Infinite intervals	1158
61.4.1 Intersection	1160
61.5 Intervals of natural numbers	1161
61.5.1 The Constant <i>lessThan</i>	1161
61.5.2 The Constant <i>greaterThan</i>	1162
61.5.3 The Constant <i>atLeast</i>	1162
61.5.4 The Constant <i>atMost</i>	1162
61.5.5 The Constant <i>atLeastLessThan</i>	1162
61.5.6 The Constant <i>atLeastAtMost</i>	1163
61.5.7 Intervals of nats with <i>Suc</i>	1163
61.5.8 Intervals and numerals	1164
61.5.9 Image	1164
61.5.10 Finiteness	1171
61.5.11 Proving Inclusions and Equalities between Unions	1173
61.5.12 Cardinality	1174
61.6 Intervals of integers	1177
61.6.1 Finiteness	1177
61.6.2 Cardinality	1178
61.7 Lemmas useful with the summation operator <i>sum</i>	1180
61.7.1 Disjoint Unions	1180
61.7.2 Disjoint Intersections	1181
61.7.3 Some Differences	1182
61.7.4 Some Subset Conditions	1182
61.8 Generic big monoid operation over intervals	1182
61.9 Summation indexed over intervals	1185
61.9.1 Shifting bounds	1191
61.9.2 Telescoping	1192
61.9.3 The formula for geometric sums	1193
61.9.4 Geometric progressions	1195
61.9.5 The formulae for arithmetic sums	1196
61.9.6 Division remainder	1198

61.10	Products indexed over intervals	1198
61.11	Efficient folding over intervals	1199
62	Decision Procedure for Presburger Arithmetic	1200
62.1	The $-\infty$ and $+\infty$ Properties	1200
62.2	The A and B sets	1203
62.3	Cooper's Theorem $-\infty$ and $+\infty$ Version	1206
62.3.1	First some trivial facts about periodic sets or predicates	1206
62.3.2	The $-\infty$ Version	1207
62.3.3	The $+\infty$ Version	1208
62.4	Nice facts about division by $4::'a$	1212
62.5	Try0	1213
63	Misc lemmas on division, to be sorted out finally	1213
64	Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2	1216
64.1	A skolemization tactic and proof method	1216
64.2	Triggers for quantifier instantiation	1217
64.3	Higher-order encoding	1217
64.4	Normalization	1218
64.5	Integer division and modulo for Z3	1219
64.6	Extra theorems for veriT reconstruction	1219
64.7	Setup	1228
64.8	Configuration	1229
64.9	General configuration options	1229
64.10	Certificates	1230
64.11	Tracing	1231
64.12	Schematic rules for Z3 proof reconstruction	1231
65	Sledgehammer: Isabelle-ATP Linkup	1233
66	Setup for Lifting/Transfer for the set type	1234
66.1	Relator and predicator properties	1234
66.2	Quotient theorem for the Lifting package	1236
66.3	Transfer rules for the Transfer package	1236
66.3.1	Unconditional transfer rules	1236
66.3.2	Rules requiring bi-unique, bi-total or right-total rela- tions	1238
67	The datatype of finite lists	1242
67.1	Basic list processing functions	1243
67.1.1	List comprehension	1250
67.1.2	\square and $(\#)$	1258
67.1.3	<i>length</i>	1259

67.1.4	@ – append	1260
67.1.5	<i>map</i>	1263
67.1.6	<i>rev</i>	1266
67.1.7	<i>set</i>	1267
67.1.8	<i>concat</i>	1271
67.1.9	<i>filter</i>	1273
67.1.10	List partitioning	1276
67.1.11	(!)	1277
67.1.12	<i>list-update</i>	1280
67.1.13	<i>last</i> and <i>butlast</i>	1281
67.1.14	<i>take</i> and <i>drop</i>	1284
67.1.15	<i>takeWhile</i> and <i>dropWhile</i>	1290
67.1.16	<i>zip</i>	1295
67.1.17	<i>list-all2</i>	1300
67.1.18	<i>List.product</i> and <i>product-lists</i>	1304
67.1.19	<i>fold</i> with natural argument order	1305
67.1.20	Fold variants: <i>foldr</i> and <i>foldl</i>	1308
67.1.21	<i>upt</i>	1309
67.1.22	<i>upto</i> : interval-list on <i>int</i>	1312
67.1.23	<i>successively</i>	1313
67.1.24	<i>distinct</i> and <i>remdups</i> and <i>remdups-adj</i>	1315
67.2	<i>distinct-adj</i>	1327
67.2.1	<i>insert</i>	1328
67.2.2	<i>List.union</i>	1329
67.2.3	<i>find</i>	1329
67.2.4	<i>count-list</i>	1330
67.2.5	<i>List.extract</i>	1331
67.2.6	<i>remove1</i>	1331
67.2.7	<i>removeAll</i>	1332
67.2.8	<i>replicate</i>	1333
67.2.9	<i>enumerate</i>	1338
67.2.10	<i>rotate1</i> and <i>rotate</i>	1339
67.2.11	<i>nths</i> — a generalization of (!) to sets	1342
67.2.12	<i>subseqs</i> and <i>List.n-lists</i>	1344
67.2.13	<i>splice</i>	1345
67.2.14	<i>shuffles</i>	1346
67.2.15	Transpose	1349
67.2.16	<i>min</i> and <i>arg-min</i>	1351
67.2.17	(In)finiteness	1351
67.3	Sorting	1354
67.3.1	<i>sorted-wrt</i>	1354
67.3.2	<i>sorted</i>	1356
67.3.3	Sorting functions	1360
67.3.4	<i>transpose</i> on sorted lists	1365

67.3.5	<i>sorted-key-list-of-set</i>	1368
67.3.6	<i>lists</i> : the list-forming operator over sets	1374
67.3.7	Inductive definition for membership	1375
67.3.8	Lists as Cartesian products	1376
67.4	Relations on Lists	1376
67.4.1	Length Lexicographic Ordering	1376
67.4.2	Lexicographic Ordering	1381
67.4.3	Lexicographic combination of measure functions	1389
67.4.4	Lifting Relations to Lists: one element	1390
67.4.5	Lifting Relations to Lists: all elements	1393
67.5	Size function	1396
67.6	Monad operation	1396
67.7	Code generation	1397
67.7.1	Counterparts for set-related operations	1397
67.7.2	Optimizing by rewriting	1401
67.7.3	Pretty lists	1403
67.7.4	Use convenient predefined operations	1405
67.7.5	Implementation of sets by lists	1405
67.8	Setup for Lifting/Transfer	1408
67.8.1	Transfer rules for the Transfer package	1408
68	Sum and product over lists	1413
68.1	List summation	1414
68.2	Horner sums	1420
68.3	Further facts about <i>List.n-lists</i>	1423
68.4	Tools setup	1424
68.5	List product	1424
69	Bit operations in suitable algebraic structures	1425
69.1	Abstract bit structures	1425
69.2	Bit operations	1440
69.3	Instance <i>int</i>	1455
69.4	Instance <i>nat</i>	1473
69.5	Common algebraic structure	1479
69.6	Symbolic computations on numeral expressions	1483
69.7	More properties	1493
69.8	Bit concatenation	1494
69.9	Taking bits with sign propagation	1495
69.10	Horner sums	1502
69.11	Symbolic computations for code generation	1504
69.12	Key ideas of bit operations	1506

70	Numeric types for code generation onto target language numerals only	1507
70.1	Type of target language integers	1508
70.2	Code theorems for target language integers	1516
70.3	Serializer setup for target language integers	1523
70.4	Type of target language naturals	1525
70.5	Inductive representation of target language naturals	1531
70.6	Code refinement for target language naturals	1532
71	A HOL random engine	1534
71.1	Auxiliary functions	1534
71.2	Random seeds	1535
71.3	Base selectors	1535
71.4	<i>ML</i> interface	1537
72	Maps	1538
72.1	<i>empty</i>	1540
72.2	<i>map-upd</i>	1540
72.3	<i>map-of</i>	1541
72.4	<i>map-option</i> related	1544
72.5	<i>map-comp</i> related	1544
72.6	<i>++</i>	1544
72.7	<i>restrict-map</i>	1546
72.8	<i>map-upds</i>	1547
72.9	<i>dom</i>	1549
72.10	<i>ran</i>	1552
72.11	<i>graph</i>	1553
72.12	<i>map-le</i>	1554
72.13	Various	1556
73	Finite types as explicit enumerations	1558
73.1	Class <i>enum</i>	1558
73.2	Implementations using <i>enum</i>	1559
73.2.1	Unbounded operations and quantifiers	1559
73.2.2	An executable choice operator	1559
73.2.3	Equality and order on functions	1560
73.2.4	Operations on relations	1561
73.2.5	Bounded accessible part	1561
73.3	Default instances for <i>enum</i>	1563
73.4	Small finite types	1568
73.5	Closing up	1582

74 Character and string types	1582
74.1 Strings as list of bytes	1582
74.1.1 Bytes as datatype	1582
74.2 Strings as dedicated type for target language code generation	1589
74.2.1 Logical specification	1589
74.2.2 Syntactic representation	1591
74.2.3 Operations	1593
74.2.4 Executable conversions	1595
74.2.5 Technical code generation setup	1596
74.2.6 Code generation utility	1599
74.2.7 Finally	1600
75 Reflecting Pure types into HOL	1600
76 Predicates as enumerations	1602
76.1 The type of predicate enumerations (a monad)	1602
76.2 Emptiness check and definite choice	1606
76.3 Derived operations	1608
76.4 Implementation	1610
77 Lazy sequences	1617
77.1 Type of lazy sequences	1617
77.2 Code setup	1620
77.3 Generator Sequences	1621
77.3.1 General lazy sequence operation	1621
77.3.2 Small lazy typeclasses	1621
77.4 With Hit Bound Value	1622
78 Depth-Limited Sequences with failure element	1624
78.1 Depth-Limited Sequence	1624
78.2 Positive Depth-Limited Sequence	1625
78.3 Negative Depth-Limited Sequence	1626
78.4 Negation	1627
79 Term evaluation using the generic code generator	1628
79.1 Term representation	1628
79.1.1 Terms and class <i>term-of</i>	1628
79.1.2 Syntax	1629
79.2 Tools setup and evaluation	1629
79.3 Dedicated <i>term-of</i> instances	1630
79.4 Generic reification	1631
79.5 Diagnostic	1631

80 A simple counterexample generator performing random testing	1631
80.1 Catching Match exceptions	1632
80.2 The <i>random</i> class	1632
80.3 Fundamental and numeric types	1632
80.4 Complex generators	1635
80.5 Deriving random generators for datatypes	1637
80.6 Code setup	1637
81 The Random-Predicate Monad	1637
82 Various kind of sequences inside the random monad	1639
83 A simple counterexample generator performing exhaustive testing	1642
83.1 Basic operations for exhaustive generators	1643
83.2 Exhaustive generator type classes	1643
83.2.1 A smarter enumeration scheme for functions over finite datatypes	1649
83.3 Bounded universal quantifiers	1656
83.4 Fast exhaustive combinators	1656
83.5 Continuation passing style functions as plus monad	1656
83.6 Defining generators for any first-order data type	1658
83.7 Defining generators for abstract types	1658
84 A compiler for predicates defined by introduction rules	1659
84.1 Set membership as a generator predicate	1659
85 Counterexample generator performing narrowing-based testing	1661
85.1 Counterexample generator	1661
85.1.1 Code generation setup	1661
85.1.2 Narrowing's deep representation of types and terms	1662
85.1.3 From narrowing's deep representation of terms to <i>HOL.Code-Evaluation's</i> terms	1662
85.1.4 Auxiliary functions for Narrowing	1662
85.1.5 Narrowing's basic operations	1663
85.1.6 Narrowing generator type class	1664
85.1.7 class <i>is-testable</i>	1664
85.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way	1665
85.1.9 Setting up the counterexample generator	1665
85.2 Narrowing for sets	1666
85.3 Narrowing for integers	1666

85.4	The <i>find-unused-assms</i> command	1668
85.5	Closing up	1668
86	Program extraction for HOL	1669
86.1	Setup	1669
86.2	Type of extracted program	1669
86.3	Realizability	1670
86.4	Computational content of basic inference rules	1672
87	Extensible records with structural subtyping	1677
87.1	Introduction	1677
87.2	Operators and lemmas for types isomorphic to tuples	1678
87.3	Logical infrastructure for records	1679
87.4	Concrete record syntax	1685
87.5	Record package	1686
88	Greatest common divisor and least common multiple	1686
88.1	Abstract bounded quasi semilattices as common foundation	1686
88.2	Abstract GCD and LCM	1688
88.3	An aside: GCD and LCM on finite sets for incomplete gcd rings	1707
88.4	Coprimality	1710
88.5	GCD and LCM for multiplicative normalisation functions	1720
88.6	GCD and LCM on <i>nat</i> and <i>int</i>	1721
88.7	Bezout's theorem	1730
88.8	LCM properties on <i>nat</i> and <i>int</i>	1735
88.9	The complete divisibility lattice on <i>nat</i> and <i>int</i>	1736
88.9.1	Setwise GCD and LCM for integers	1739
88.10	GCD and LCM on <i>integer</i>	1741
88.11	Characteristic of a semiring	1744
89	Nitpick: Yet Another Counterexample Generator for Isabelle/HOL	1746
90	Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors	1751
90.1	Equivalence relations, quotients, and Hilbert's choice	1756
91	Filters on predicates	1758
91.1	Filters	1758
91.1.1	Eventually	1758
91.2	Frequently as dual to eventually	1761
91.2.1	Finer-than relation	1763
91.2.2	Map function for filters	1770
91.2.3	Contravariant map function for filters	1771

91.2.4	Standard filters	1774
91.2.5	Order filters	1775
91.3	Sequentially	1777
91.4	Increasing finite subsets	1778
91.5	The cofinite filter	1779
91.5.1	Product of filters	1780
91.6	Limits	1785
91.7	Limits to <i>at-top</i> and <i>at-bot</i>	1789
91.8	Setup 'a filter for lifting and transfer	1792
92	Conditionally-complete Lattices	1803
92.1	Preorders	1804
92.2	Lattices	1806
92.3	Conditionally complete lattices	1808
92.4	Complete lattices	1813
92.5	Instances	1815
93	Factorial Function, Rising Factorials	1823
93.1	Factorial Function	1823
93.2	Pochhammer's symbol: generalized rising factorial	1826
93.3	Misc	1831
94	Binomial Coefficients and Binomial Theorem	1832
94.1	Binomial coefficients	1832
94.2	The binomial theorem (courtesy of Tobias Nipkow):	1835
94.3	Generalized binomial coefficients	1839
94.3.1	Summation on the upper index	1849
94.4	More on Binomial Coefficients	1856
94.5	Executable code	1862
95	Main HOL	1862
95.1	Namespace cleanup	1863
95.2	Syntax cleanup	1863
95.3	Lattice syntax	1863
96	Archimedean Fields, Floor and Ceiling Functions	1864
96.1	Class of Archimedean fields	1866
96.2	Existence and uniqueness of floor function	1867
96.3	Floor function	1868
96.4	Ceiling function	1873
96.4.1	Ceiling with numerals.	1874
96.4.2	Addition and subtraction of integers.	1876
96.5	Negation	1877
96.6	Natural numbers	1877

96.7	Frac Function	1878
96.8	Rounding to the nearest integer	1879
97	Rational numbers	1881
97.1	Rational numbers as quotient	1881
97.1.1	Construction of the type of rational numbers	1881
97.1.2	Representation and basic operations	1882
97.1.3	Function <i>normalize</i>	1886
97.1.4	Various	1890
97.1.5	The ordered field of rational numbers	1890
97.1.6	Rationals are an Archimedean field	1893
97.2	Linear arithmetic setup	1894
97.3	Embedding from Rationals to other Fields	1894
97.4	The Set of Rational Numbers	1897
97.5	Implementation of rational numbers as pairs of integers	1899
97.6	Setup for Nitpick	1904
97.7	Float syntax	1904
97.8	Hiding implementation details	1905
98	Development of the Reals using Cauchy Sequences	1905
98.1	Preliminary lemmas	1905
98.2	Sequences that converge to zero	1906
98.3	Cauchy sequences	1908
98.4	Equivalence relation on Cauchy sequences	1912
98.5	The field of real numbers	1913
98.6	Positive reals	1916
98.7	Completeness	1919
98.8	Supremum of a set of reals	1923
98.9	Hiding implementation details	1924
98.10	Embedding numbers into the Reals	1924
98.11	Embedding the Naturals into the Reals	1926
98.12	The Archimedean Property of the Reals	1927
98.13	Rationals	1928
98.14	Density of the Rational Reals in the Reals	1930
98.15	Numerals and Arithmetic	1931
98.16	Simprules combining $x + y$ and 0	1931
98.17	Lemmas about powers	1932
98.18	Density of the Reals	1932
98.19	Archimedean properties and useful consequences	1932
98.20	Floor and Ceiling Functions from the Reals to the Integers	1935
98.21	Exponentiation with floor	1938
98.22	Implementation of rational real numbers	1938
98.23	Setup for Nitpick	1941
98.24	Setup for SMT	1941

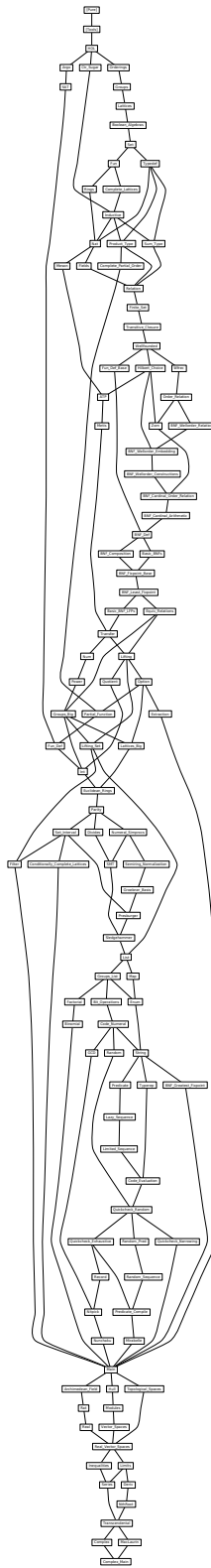
98.25	Setup for Argo	1943
99	Topological Spaces	1943
99.1	Topological space	1943
99.2	Hausdorff and other separation properties	1946
99.3	Generators for topologies	1948
99.4	Order topologies	1948
99.5	Setup some topologies	1950
99.5.1	Boolean is an order topology	1950
99.5.2	Topological filters	1952
99.5.3	Tendsto	1958
99.5.4	Rules about <i>Lim</i>	1965
99.6	Limits on sequences	1968
99.7	Monotone sequences and subsequences	1969
99.7.1	Definition of subsequence.	1969
99.7.2	Subsequence (alternative definition, (e.g. Hoskins)	1970
99.7.3	Increasing and Decreasing Series	1976
99.8	First countable topologies	1976
99.9	Function limit at a point	1980
99.9.1	Relation of <i>LIM</i> and <i>LIMSEQ</i>	1982
99.10	Continuity	1985
99.10.1	Continuity on a set	1985
99.10.2	Continuity at a point	1990
99.10.3	Open-cover compactness	1996
99.11	Finite intersection property	1997
99.12	Connectedness	2001
100	Linear Continuum Topologies	2006
100.1	Intermediate Value Theorem	2010
100.2	Uniform spaces	2014
100.2.1	Totally bounded sets	2016
100.2.2	Cauchy filter	2016
100.2.3	Uniformly continuous functions	2018
101	Product Topology	2019
101.1	Product is a topological space	2019
101.1.1	Continuity of operations	2022
101.1.2	Connectedness of products	2024
101.1.3	Separation axioms	2025
101.2	A generic notion of the convex, affine, conic hull, or closed "hull".	2026
102	Modules	2028
102.1	Locale for additive functions	2028

103	Subspace	2030
104	Span: subspace generated by a set	2031
105	Dependent and independent sets	2036
106	Representation of a vector on a specific basis	2038
107	Vector Spaces	2049
108	Vector Spaces and Algebras over the Reals	2084
108.1	Real vector spaces	2084
108.2	Embedding of the Reals into any <i>real-algebra-1: of-real</i> . . .	2089
108.3	The Set of Real Numbers	2093
108.4	Ordered real vector spaces	2094
108.5	Real normed vector spaces	2099
108.6	Metric spaces	2108
108.7	Class instances for real numbers	2111
108.8	Extra type constraints	2113
108.9	Sign function	2113
108.10	Bounded Linear and Bilinear Operators	2115
108.11	Filters and Limits on Metric Space	2121
108.11.1	Limits of Sequences	2124
108.11.2	Limits of Functions	2125
108.12	Complete metric spaces	2127
108.13	Cauchy sequences	2127
108.13.1	Cauchy Sequences are Convergent	2132
108.14	The set of real numbers is a complete metric space	2133
109	Limits on Real Vector Spaces	2137
109.1	Filter going to infinity norm	2137
109.1.1	Boundedness	2138
109.1.2	Bounded Sequences	2139
109.1.3	A Few More Equivalence Theorems for Boundedness	2141
109.1.4	Upper Bounds and Lubs of Bounded Sequences	2142
109.1.5	Polynomial function extremal theorem, from HOL Light	2144
109.2	Convergence to Zero	2146
109.2.1	Distance and norms	2149
109.3	Topological Monoid	2151
109.3.1	Topological group	2152
109.3.2	Linear operators and multiplication	2154
109.3.3	Inverse and division	2160
109.4	Relate <i>at</i> , <i>at-left</i> and <i>at-right</i>	2168
109.5	Floor and Ceiling	2181
109.6	Limits of Sequences	2182

109.7	Convergence on sequences	2186
109.8	More about <i>filterlim</i> (thanks to Wenda Li)	2191
109.9	Power Sequences	2194
109.10	Limits of Functions	2195
109.11	Continuity	2197
109.12	Uniform Continuity	2198
109.13	Nested Intervals and Bisection – Needed for Compactness	2200
109.14	Boundedness of continuous functions	2203
110	Infinite Series	2206
110.1	Definition of infinite summability	2206
110.2	Infinite summability on topological monoids	2207
110.3	Infinite summability on ordered, topological monoids	2210
110.4	Infinite summability on topological monoids	2212
110.5	Infinite summability on real normed vector spaces	2214
110.6	Infinite summability on real normed algebras	2217
110.7	Infinite summability on real normed fields	2218
110.8	Telescoping	2220
110.9	Infinite summability on Banach spaces	2221
110.10	The Ratio Test	2224
110.11	Cauchy Product Formula	2225
110.12	Series on <i>reals</i>	2227
111	Differentiation	2235
111.1	Frechet derivative	2235
111.1.1	Limit transformation for derivatives	2240
111.2	Continuity	2241
111.3	Composition	2242
111.4	Uniqueness	2248
111.5	Differentiability predicate	2249
111.6	Vector derivative	2253
111.7	Derivatives	2255
111.8	Local extrema	2263
111.9	Rolle’s Theorem	2266
111.10	Mean Value Theorem	2269
111.10.1	A function is constant if its derivative is 0 over an interval.	2270
111.10.2	A function with positive derivative is increasing	2273
111.11	Generalized Mean Value Theorem	2277
111.12	L’Hopitals rule	2278

112	Nth Roots of Real Numbers	2287
112.1	Existence of Nth Root	2287
112.2	Nth Root	2288
112.3	Square Root	2294
112.4	Square Root of Sum of Squares	2299
113	Power Series, Transcendental Functions etc.	2303
113.1	More facts about binomial coefficients	2305
113.2	Properties of Power Series	2308
113.3	Alternating series test / Leibniz formula	2312
113.4	Term-by-Term Differentiability of Power Series	2316
113.5	The Derivative of a Power Series Has the Same Radius of Convergence	2321
113.6	Derivability of power series	2325
113.7	Exponential Function	2331
113.7.1	Properties of the Exponential Function	2333
113.7.2	Properties of the Exponential Function on Reals	2336
113.8	Natural Logarithm	2339
113.8.1	A couple of simple bounds	2353
113.9	The general logarithm	2354
113.10	Sine and Cosine	2371
113.11	Properties of Sine and Cosine	2375
113.12	Deriving the Addition Formulas	2375
113.13	The Constant Pi	2380
113.14	More Corollaries about Sine and Cosine	2395
113.15	Tangent	2399
113.16	Cotangent	2407
113.17	Inverse Trigonometric Functions	2409
113.18	Prove Totality of the Trigonometric Functions	2418
113.19	Machin's formula	2424
113.20	Introducing the inverse tangent power series	2425
113.21	Existence of Polar Coordinates	2435
113.22	Basics about polynomial functions: products, extremal be- haviour and root counts	2436
113.23	Hyperbolic functions	2442
113.23.1	More specific properties of the real functions	2447
113.23.2	Limits	2452
113.23.3	Properties of the inverse hyperbolic functions	2453
113.24	Simprocs for root and power literals	2459
114	Complex Numbers: Rectangular and Polar Representation	2463
114.1	Addition and Subtraction	2464
114.2	Multiplication and Division	2464
114.3	Scalar Multiplication	2465

114.4	Numerals, Arithmetic, and Embedding from \mathbb{R}	2466
114.5	The Complex Number i	2468
114.6	Vector Norm	2470
114.7	Absolute value	2472
114.8	Completeness of the Complexes	2472
114.9	Complex Conjugation	2474
114.1	Basic Lemmas	2478
114.1	Polar Form for Complex Numbers	2480
114.11.1	$\cos \theta + i \sin \theta$	2480
114.11.2	$(\cos \theta + i \sin \theta)^2$	2482
114.11.3	Complex exponential	2483
114.11.4	Complex argument	2484
114.1	Complex n-th roots	2486
114.1	Square root of complex numbers	2490
115	MacLaurin and Taylor Series	2493
115.1	Maclaurin's Theorem with Lagrange Form of Remainder	2493
115.2	More Convenient "Bidirectional" Version.	2497
115.3	Version for Exponential Function	2499
115.4	Version for Sine Function	2500
115.5	Maclaurin Expansion for Cosine Function	2501
116	Taylor series	2504
117	Comprehensive Complex Theory	2506



1 Loading the code generator and related modules

```

theory Code-Generator
imports Pure
keywords
  print-codeproc code-thms code-deps :: diag and
  export-code code-identifier code-printing code-reserved
  code-monad code-reflect :: thy-decl and
  checking and
  datatypes functions module-name file file-prefix
  constant type-constructor type-class class-relation class-instance code-module
  :: quasi-command
begin

ML-file <~~/src/Tools/cache-io.ML>
ML-file <~~/src/Tools/Code/code-preproc.ML>
ML-file <~~/src/Tools/Code/code-symbol.ML>
ML-file <~~/src/Tools/Code/code-thingol.ML>
ML-file <~~/src/Tools/Code/code-simp.ML>
ML-file <~~/src/Tools/Code/code-printer.ML>
ML-file <~~/src/Tools/Code/code-target.ML>
ML-file <~~/src/Tools/Code/code-namespace.ML>
ML-file <~~/src/Tools/Code/code-ml.ML>
ML-file <~~/src/Tools/Code/code-haskell.ML>
ML-file <~~/src/Tools/Code/code-scala.ML>

code-datatype TYPE('a::{})

definition holds :: prop where
  holds  $\equiv ((\lambda x::prop. x) \equiv (\lambda x. x))$ 

lemma holds: PROP holds
  by (unfold holds-def) (rule reflexive)

code-datatype holds

lemma implies-code [code]:
  (PROP holds  $\implies$  PROP P)  $\equiv$  PROP P
  (PROP P  $\implies$  PROP holds)  $\equiv$  PROP holds
proof –
  show (PROP holds  $\implies$  PROP P)  $\equiv$  PROP P
  proof
    assume PROP holds  $\implies$  PROP P
    then show PROP P using holds .
  next
    assume PROP P
    then show PROP P .
  qed
next

```



```

  show (PROP P  $\implies$  PROP holds)  $\equiv$  PROP holds
    by rule (rule holds)+
qed

```

```
ML-file <~~/src/Tools/Code/code-runtime.ML>
```

```
ML-file <~~/src/Tools/nbe.ML>
```

```
hide-const (open) holds
```

```
end
```

2 The basis of Higher-Order Logic

```
theory HOL
```

```
imports Pure Tools.Code-Generator
```

```
keywords
```

```
  try solve-direct quickcheck print-coercions print-claset
```

```
  print-induct-rules :: diag and
```

```
  quickcheck-params :: thy-decl
```

```
abbrevs ?< =  $\exists_{\leq 1}$ 
```

```
begin
```

```
ML-file <~~/src/Tools/misc-legacy.ML>
```

```
ML-file <~~/src/Tools/try.ML>
```

```
ML-file <~~/src/Tools/quickcheck.ML>
```

```
ML-file <~~/src/Tools/solve-direct.ML>
```

```
ML-file <~~/src/Tools/IsaPlanner/zipper.ML>
```

```
ML-file <~~/src/Tools/IsaPlanner/isand.ML>
```

```
ML-file <~~/src/Tools/IsaPlanner/rw-inst.ML>
```

```
ML-file <~~/src/Provers/hypsubst.ML>
```

```
ML-file <~~/src/Provers/splitter.ML>
```

```
ML-file <~~/src/Provers/classical.ML>
```

```
ML-file <~~/src/Provers/blast.ML>
```

```
ML-file <~~/src/Provers/clasimp.ML>
```

```
ML-file <~~/src/Tools/eqsubst.ML>
```

```
ML-file <~~/src/Provers/quantifier1.ML>
```

```
ML-file <~~/src/Tools/atomize-elim.ML>
```

```
ML-file <~~/src/Tools/cong-tac.ML>
```

```
ML-file <~~/src/Tools/intuitionistic.ML> setup <Intuitionistic.method-setup binding <iprover>>
```

```
ML-file <~~/src/Tools/project-rule.ML>
```

```
ML-file <~~/src/Tools/subtyping.ML>
```

```
ML-file <~~/src/Tools/case-product.ML>
```

```
ML <Plugin-Name.declare-setup binding <extraction>>
```

```
ML <
```

```
  Plugin-Name.declare-setup binding <quickcheck-random>;
```

```

Plugin-Name.declare-setup binding <quickcheck-exhaustive>;
Plugin-Name.declare-setup binding <quickcheck-bounded-forall>;
Plugin-Name.declare-setup binding <quickcheck-full-exhaustive>;
Plugin-Name.declare-setup binding <quickcheck-narrowing>;
>
ML <
Plugin-Name.define-setup binding <quickcheck>
[plugin <quickcheck-exhaustive>,
plugin <quickcheck-random>,
plugin <quickcheck-bounded-forall>,
plugin <quickcheck-full-exhaustive>,
plugin <quickcheck-narrowing>]
>

```

2.1 Primitive logic

The definition of the logic is based on Mike Gordon’s technical report [2] that describes the first implementation of HOL. However, there are a number of differences. In particular, we start with the definite description operator and introduce Hilbert’s ε operator only much later. Moreover, axiom $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)$ is derived from the other axioms. The fact that this axiom is derivable was first noticed by Bruno Barras (for Mike Gordon’s line of HOL systems) and later independently by Alexander Maletzky (for Isabelle/HOL).

2.1.1 Core syntax

```

setup <Aclass.class-axiomatization (binding <type>, [])>
default-sort type
setup <Object-Logic.add-base-sort sort <type>>

setup <Proofterm.set-preproc (Proof-Rewrite-Rules.standard-preproc [])>

axiomatization where fun-arity: OFCLASS('a  $\Rightarrow$  'b, type-class)
instance fun :: (type, type) type by (rule fun-arity)

axiomatization where itself-arity: OFCLASS('a itself, type-class)
instance itself :: (type) type by (rule itself-arity)

typedecl bool

judgment Trueprop :: bool  $\Rightarrow$  prop ((-) 5)

axiomatization implies :: [bool, bool]  $\Rightarrow$  bool (infixr  $\longrightarrow$  25)
and eq :: ['a, 'a]  $\Rightarrow$  bool
and The :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a

notation (input)

```

eq (**infixl** = 50)
notation (**output**)
eq (**infix** = 50)

The input syntax for *eq* is more permissive than the output syntax because of the large amount of material that relies on *infixl*.

2.1.2 Defined connectives and quantifiers

definition *True* :: *bool*
where *True* $\equiv ((\lambda x::bool. x) = (\lambda x. x))$

definition *All* :: (*a* \Rightarrow *bool*) \Rightarrow *bool* (**binder** \forall 10)
where *All* *P* $\equiv (P = (\lambda x. True))$

definition *Ex* :: (*a* \Rightarrow *bool*) \Rightarrow *bool* (**binder** \exists 10)
where *Ex* *P* $\equiv \forall Q. (\forall x. P x \longrightarrow Q) \longrightarrow Q$

definition *False* :: *bool*
where *False* $\equiv (\forall P. P)$

definition *Not* :: *bool* \Rightarrow *bool* (\neg - [40] 40)
where *not-def*: $\neg P \equiv P \longrightarrow False$

definition *conj* :: [*bool*, *bool*] \Rightarrow *bool* (**infixr** \wedge 35)
where *and-def*: $P \wedge Q \equiv \forall R. (P \longrightarrow Q \longrightarrow R) \longrightarrow R$

definition *disj* :: [*bool*, *bool*] \Rightarrow *bool* (**infixr** \vee 30)
where *or-def*: $P \vee Q \equiv \forall R. (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

definition *Uniq* :: (*a* \Rightarrow *bool*) \Rightarrow *bool*
where *Uniq* *P* $\equiv (\forall x y. P x \longrightarrow P y \longrightarrow y = x)$

definition *Ex1* :: (*a* \Rightarrow *bool*) \Rightarrow *bool*
where *Ex1* *P* $\equiv \exists x. P x \wedge (\forall y. P y \longrightarrow y = x)$

2.1.3 Additional concrete syntax

syntax (*ASCII*) *-Uniq* :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ((4?< -./ -) [0, 10] 10)

syntax *-Uniq* :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ((2 $\exists_{\leq 1}$ -./ -) [0, 10] 10)

translations $\exists_{\leq 1} x. P \equiv CONST Uniq (\lambda x. P)$

print-translation \langle

[*Syntax-Trans.preserve-binder-abs-tr'* **const-syntax** $\langle Uniq \rangle$ **syntax-const** $\langle -Uniq \rangle$]
 \rangle — to avoid eta-contraction of body

syntax (*ASCII*)

-Ex1 :: *pttrn* \Rightarrow *bool* \Rightarrow *bool* ((3EX! -./ -) [0, 10] 10)

syntax (*input*)

-Ex1 :: *pttrn* ⇒ *bool* ⇒ *bool* (($\exists!$! -./ -) [0, 10] 10)
syntax *-Ex1* :: *pttrn* ⇒ *bool* ⇒ *bool* ((\exists ! -./ -) [0, 10] 10)
translations $\exists!x. P \equiv \text{CONST } Ex1 (\lambda x. P)$

print-translation ‹

[*Syntax-Trans.preserve-binder-abs-tr'* **const-syntax** ‹*Ex1*› **syntax-const** ‹*-Ex1*›]
› — to avoid eta-contraction of body

syntax

-Not-Ex :: *idts* ⇒ *bool* ⇒ *bool* ((\exists ! -./ -) [0, 10] 10)
-Not-Ex1 :: *pttrn* ⇒ *bool* ⇒ *bool* ((\exists ! -./ -) [0, 10] 10)
translations
 $\exists x. P \equiv \neg (\exists x. \neg P)$
 $\exists!x. P \equiv \neg (\exists!x. \neg P)$

abbreviation *not-equal* :: [*'a*, *'a*] ⇒ *bool* (**infix** ≠ 50)

where $x \neq y \equiv \neg (x = y)$

notation (*ASCII*)

Not (\sim - [40] 40) **and**
conj (**infixr** & 35) **and**
disj (**infixr** | 30) **and**
implies (**infixr** --> 25) **and**
not-equal (**infix** ≈ = 50)

abbreviation (*iff*)

iff :: [*bool*, *bool*] ⇒ *bool* (**infixr** ‹‹ 25)
where $A \longleftrightarrow B \equiv A = B$

syntax *-The* :: [*pttrn*, *bool*] ⇒ *'a* ((\exists THE -./ -) [0, 10] 10)

translations *THE* *x. P* ≡ *CONST The* ($\lambda x. P$)

print-translation ‹

[(**const-syntax** ‹*The*›, *fn - => fn* [*Abs abs*] =>
let val (*x*, *t*) = *Syntax-Trans.atomic-abs-tr'* *abs*
in *Syntax.const syntax-const* ‹*-The*› \$ *x* \$ *t end*)]
› — To avoid eta-contraction of body

nonterminal *letbinds* **and** *letbind*

syntax

-bind :: [*pttrn*, *'a*] ⇒ *letbind* (($2-$ =/ -) 10)
:: *letbind* ⇒ *letbinds* (-)
-binds :: [*letbind*, *letbinds*] ⇒ *letbinds* (-;/ -)
-Let :: [*letbinds*, *'a*] ⇒ *'a* ((*let* (-)/ *in* (-)) [0, 10] 10)

nonterminal *case-syn* **and** *cases-syn*

syntax

$-case-syntax :: ['a, cases-syn] \Rightarrow 'b \ ((case - of / -) 10)$
 $-case1 :: ['a, 'b] \Rightarrow case-syn \ ((2- \Rightarrow / -) 10)$
 $:: case-syn \Rightarrow cases-syn \ (-)$
 $-case2 :: [case-syn, cases-syn] \Rightarrow cases-syn \ (- / | -)$
syntax (ASCII)
 $-case1 :: ['a, 'b] \Rightarrow case-syn \ ((2- \Rightarrow / -) 10)$

notation (ASCII)
All (**binder** ALL 10) **and**
Ex (**binder** EX 10)

notation (input)
All (**binder** ! 10) **and**
Ex (**binder** ? 10)

2.1.4 Axioms and basic definitions

axiomatization where

$refl: t = (t::'a)$ **and**
 $subst: s = t \Longrightarrow P s \Longrightarrow P t$ **and**
 $ext: (\bigwedge x::'a. (f x ::'b) = g x) \Longrightarrow (\lambda x. f x) = (\lambda x. g x)$

— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL **and**

$the-eq-trivial: (THE x. x = a) = (a::'a)$

axiomatization where

$impI: (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$ **and**
 $mp: \llbracket P \longrightarrow Q; P \rrbracket \Longrightarrow Q$ **and**

$True-or-False: (P = True) \vee (P = False)$

definition $If :: bool \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \ ((if (-) / then (-) / else (-)) [0, 0, 10] 10)$
where $If P x y \equiv (THE z::'a. (P = True \longrightarrow z = x) \wedge (P = False \longrightarrow z = y))$

definition $Let :: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$
where $Let s f \equiv f s$

translations

$-Let (-binds b bs) e \equiv -Let b (-Let bs e)$
 $let x = a in e \equiv CONST Let a (\lambda x. e)$

axiomatization $undefined :: 'a$

class $default = fixes default :: 'a$

2.2 Fundamental rules

2.2.1 Equality

lemma *sym*: $s = t \implies t = s$
by (*erule subst*) (*rule refl*)

lemma *ssubst*: $t = s \implies P s \implies P t$
by (*drule sym*) (*erule subst*)

lemma *trans*: $\llbracket r = s; s = t \rrbracket \implies r = t$
by (*erule subst*)

lemma *trans-sym* [*Pure.elim?*]: $r = s \implies t = s \implies r = t$
by (*rule trans* [*OF - sym*])

lemma *meta-eq-to-obj-eq*:
assumes $A \equiv B$
shows $A = B$
unfolding *assms* **by** (*rule refl*)

Useful with *erule* for proving equalities from known equalities.

lemma *box-equals*: $\llbracket a = b; a = c; b = d \rrbracket \implies c = d$
by (*iprover intro: sym trans*)

For calculational reasoning:

lemma *forw-subst*: $a = b \implies P b \implies P a$
by (*rule ssubst*)

lemma *back-subst*: $P a \implies a = b \implies P b$
by (*rule subst*)

2.2.2 Congruence rules for application

Similar to *AP-THM* in Gordon’s HOL.

lemma *fun-cong*: $(f :: 'a \Rightarrow 'b) = g \implies f x = g x$
by (*iprover intro: refl elim: subst*)

Similar to *AP-TERM* in Gordon’s HOL and FOL’s *subst-context*.

lemma *arg-cong*: $x = y \implies f x = f y$
by (*iprover intro: refl elim: subst*)

lemma *arg-cong2*: $\llbracket a = b; c = d \rrbracket \implies f a c = f b d$
by (*iprover intro: refl elim: subst*)

lemma *cong*: $\llbracket f = g; (x::'a) = y \rrbracket \implies f x = g y$
by (*iprover intro: refl elim: subst*)

ML $\langle \text{fun cong-tac ctxt} = \text{Cong-Tac.cong-tac ctxt} @ \{ \text{thm cong} \} \rangle$

2.2.3 Equality of booleans – iff

lemma *iffD2*: $\llbracket P = Q; Q \rrbracket \Longrightarrow P$
by (*erule ssubst*)

lemma *rev-iffD2*: $\llbracket Q; P = Q \rrbracket \Longrightarrow P$
by (*erule iffD2*)

lemma *iffD1*: $Q = P \Longrightarrow Q \Longrightarrow P$
by (*drule sym*) (*rule iffD2*)

lemma *rev-iffD1*: $Q \Longrightarrow Q = P \Longrightarrow P$
by (*drule sym*) (*rule rev-iffD2*)

lemma *iffE*:
assumes *major*: $P = Q$
and *minor*: $\llbracket P \longrightarrow Q; Q \longrightarrow P \rrbracket \Longrightarrow R$
shows R
by (*iprover intro: minor impI major [THEN iffD2] major [THEN iffD1]*)

2.2.4 True (1)

lemma *TrueI*: *True*
unfolding *True-def* **by** (*rule refl*)

lemma *eqTrueE*: $P = \text{True} \Longrightarrow P$
by (*erule iffD2*) (*rule TrueI*)

2.2.5 Universal quantifier (1)

lemma *spec*: $\forall x::'a. P x \Longrightarrow P x$
unfolding *All-def* **by** (*iprover intro: eqTrueE fun-cong*)

lemma *allE*:
assumes *major*: $\forall x. P x$ **and** *minor*: $P x \Longrightarrow R$
shows R
by (*iprover intro: minor major [THEN spec]*)

lemma *all-dupE*:
assumes *major*: $\forall x. P x$ **and** *minor*: $\llbracket P x; \forall x. P x \rrbracket \Longrightarrow R$
shows R
by (*iprover intro: minor major major [THEN spec]*)

2.2.6 False

Depends upon *spec*; it is impossible to do propositional logic before quantifiers!

lemma *FalseE*: $\text{False} \Longrightarrow P$
unfolding *False-def* **by** (*erule spec*)

lemma *False-neq-True*: $False = True \implies P$
by (*erule eqTrueE [THEN FalseE]*)

2.2.7 Negation

lemma *notI*:
assumes $P \implies False$
shows $\neg P$
unfolding *not-def* **by** (*iprover intro: impI assms*)

lemma *False-not-True*: $False \neq True$
by (*iprover intro: notI elim: False-neq-True*)

lemma *True-not-False*: $True \neq False$
by (*iprover intro: notI dest: sym elim: False-neq-True*)

lemma *notE*: $[\neg P; P] \implies R$
unfolding *not-def*
by (*iprover intro: mp [THEN FalseE]*)

2.2.8 Implication

lemma *impE*:
assumes $P \longrightarrow Q$ P $Q \implies R$
shows R
by (*iprover intro: assms mp*)

Reduces Q to $P \longrightarrow Q$, allowing substitution in P .

lemma *rev-mp*: $[[P; P \longrightarrow Q] \implies Q$
by (*rule mp*)

lemma *contrapos-nn*:
assumes *major*: $\neg Q$
and *minor*: $P \implies Q$
shows $\neg P$
by (*iprover intro: notI minor major [THEN notE]*)

Not used at all, but we already have the other 3 combinations.

lemma *contrapos-pn*:
assumes *major*: Q
and *minor*: $P \implies \neg Q$
shows $\neg P$
by (*iprover intro: notI minor major notE*)

lemma *not-sym*: $t \neq s \implies s \neq t$
by (*erule contrapos-nn*) (*erule sym*)

lemma *eq-neq-eq-imp-neq*: $[[x = a; a \neq b; b = y] \implies x \neq y$
by (*erule subst, erule ssubst, assumption*)

2.2.9 Disjunction (1)

lemma *disjE*:
 assumes *major*: $P \vee Q$
 and *minorP*: $P \implies R$
 and *minorQ*: $Q \implies R$
 shows R
 by (iprover intro: *minorP minorQ impI*
major [unfolded or-def, THEN spec, THEN mp, THEN mp])

2.2.10 Derivation of *iffI*

In an intuitionistic version of HOL *iffI* needs to be an axiom.

lemma *iffI*:
 assumes $P \implies Q$ and $Q \implies P$
 shows $P = Q$
 proof (rule *disjE*[OF *True-or-False*[of P]])
 assume 1: $P = \text{True}$
 note $Q = \text{assms}(1)$ [OF *eqTrueE*[OF *this*]]
 from 1 show ?thesis
 proof (rule *subst*)
 from *True-or-False*[of Q] show $\text{True} = Q$
 proof (rule *disjE*)
 assume $Q = \text{True}$
 thus ?thesis by (rule *sym*)
 next
 assume $Q = \text{False}$
 with Q have False by (rule *rev-iffD1*)
 thus ?thesis by (rule *FalseE*)
 qed
 qed
 next
 assume 2: $P = \text{False}$
 thus ?thesis
 proof (rule *subst*)
 from *True-or-False*[of Q] show $\text{False} = Q$
 proof (rule *disjE*)
 assume $Q = \text{True}$
 from 2 *assms*(2)[OF *eqTrueE*[OF *this*]] have False by (rule *iffD1*)
 thus ?thesis by (rule *FalseE*)
 next
 assume $Q = \text{False}$
 thus ?thesis by (rule *sym*)
 qed
 qed
 qed

2.2.11 True (2)

lemma *eqTrueI*: $P \implies P = \text{True}$

by (*iprover intro: iffI TrueI*)

2.2.12 Universal quantifier (2)

lemma *allI*:

assumes $\bigwedge x::'a. P\ x$

shows $\forall x. P\ x$

unfolding *All-def* by (*iprover intro: ext eqTrueI assms*)

2.2.13 Existential quantifier

lemma *exI*: $P\ x \implies \exists x::'a. P\ x$

unfolding *Ex-def* by (*iprover intro: allI allE impI mp*)

lemma *exE*:

assumes *major*: $\exists x::'a. P\ x$

and *minor*: $\bigwedge x. P\ x \implies Q$

shows Q

by (*rule major [unfolded Ex-def, THEN spec, THEN mp]*) (*iprover intro: impI [THEN allI] minor*)

2.2.14 Conjunction

lemma *conjI*: $\llbracket P; Q \rrbracket \implies P \wedge Q$

unfolding *and-def* by (*iprover intro: impI [THEN allI] mp*)

lemma *conjunct1*: $\llbracket P \wedge Q \rrbracket \implies P$

unfolding *and-def* by (*iprover intro: impI dest: spec mp*)

lemma *conjunct2*: $\llbracket P \wedge Q \rrbracket \implies Q$

unfolding *and-def* by (*iprover intro: impI dest: spec mp*)

lemma *conjE*:

assumes *major*: $P \wedge Q$

and *minor*: $\llbracket P; Q \rrbracket \implies R$

shows R

proof (*rule minor*)

show P by (*rule major [THEN conjunct1]*)

show Q by (*rule major [THEN conjunct2]*)

qed

lemma *context-conjI*:

assumes $P\ P \implies Q$

shows $P \wedge Q$

by (*iprover intro: conjI assms*)

2.2.15 Disjunction (2)

lemma *disjI1*: $P \implies P \vee Q$

unfolding *or-def* by (*iprover intro: allI impI mp*)

lemma *disjI2*: $Q \implies P \vee Q$
unfolding *or-def* **by** (*iprover intro: allI impI mp*)

2.2.16 Classical logic

lemma *classical*:
assumes $\neg P \implies P$
shows P
proof (*rule True-or-False [THEN disjE]*)
show P **if** $P = \text{True}$
using *that* **by** (*iprover intro: eqTrueE*)
show P **if** $P = \text{False}$
proof (*intro notI assms*)
assume P
with *that* **show** False
by (*iprover elim: subst*)
qed
qed

lemmas *ccontr* = *FalseE [THEN classical]*

notE with premises exchanged; it discharges $\neg R$ so that it can be used to make elimination rules.

lemma *rev-notE*:
assumes *premp*: P
and *premntot*: $\neg R \implies \neg P$
shows R
by (*iprover intro: ccontr notE [OF premntot premp]*)

Double negation law.

lemma *notnotD*: $\neg\neg P \implies P$
by (*iprover intro: ccontr notE*)

lemma *contrapos-pp*:
assumes *p1*: Q
and *p2*: $\neg P \implies \neg Q$
shows P
by (*iprover intro: classical p1 p2 notE*)

2.2.17 Unique existence

lemma *Uniq-I* [*intro?*]:
assumes $\bigwedge x y. \llbracket P x; P y \rrbracket \implies y = x$
shows *Uniq* P
unfolding *Uniq-def* **by** (*iprover intro: assms allI impI*)

lemma *Uniq-D* [*dest?*]: $\llbracket \text{Uniq } P; P a; P b \rrbracket \implies a=b$
unfolding *Uniq-def* **by** (*iprover dest: spec mp*)

lemma *ex1I*:
assumes $P\ a\ \wedge x. P\ x \implies x = a$
shows $\exists!x. P\ x$
unfolding *Ex1-def* **by** (*iprover intro: assms exI conjI allI impI*)

Sometimes easier to use: the premises have no shared variables. Safe!

lemma *ex-ex1I*:
assumes *ex-prem*: $\exists x. P\ x$
and *eq*: $\wedge x\ y. \llbracket P\ x; P\ y \rrbracket \implies x = y$
shows $\exists!x. P\ x$
by (*iprover intro: ex-prem [THEN exE] ex1I eq*)

lemma *ex1E*:
assumes *major*: $\exists!x. P\ x$ **and** *minor*: $\wedge x. \llbracket P\ x; \forall y. P\ y \longrightarrow y = x \rrbracket \implies R$
shows R
proof (*rule major [unfolded Ex1-def, THEN exE]*)
show $\wedge x. P\ x \wedge (\forall y. P\ y \longrightarrow y = x) \implies R$
by (*iprover intro: minor elim: conjE*)
qed

lemma *ex1-implies-ex*: $\exists!x. P\ x \implies \exists x. P\ x$
by (*iprover intro: exI elim: ex1E*)

2.2.18 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes $\neg Q \implies P$
shows $P \vee Q$
by (*rule classical*) (*iprover intro: assms disjI1 disjI2 notI elim: notE*)

lemma *excluded-middle*: $\neg P \vee P$
by (*iprover intro: disjCI*)

case distinction as a natural deduction rule. Note that $\neg P$ is the second case, not the first.

lemma *case-split* [*case-names True False*]:
assumes $P \implies Q\ \neg P \implies Q$
shows Q
using *excluded-middle* [*of P*]
by (*iprover intro: assms elim: disjE*)

Classical implies (\longrightarrow) elimination.

lemma *impCE*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $\neg P \implies R\ Q \implies R$
shows R
using *excluded-middle* [*of P*]

by (*iprover intro: minor major [THEN mp] elim: disjE*)+

This version of \longrightarrow elimination works on Q before P . It works best for those cases in which P holds "almost everywhere". Can't install as default: would break old proofs.

lemma *impCE'*:
 assumes *major*: $P \longrightarrow Q$
 and *minor*: $Q \Longrightarrow R \neg P \Longrightarrow R$
 shows R
 using *assms* by (*elim impCE*)

Classical \longleftrightarrow elimination.

lemma *iffCE*:
 assumes *major*: $P = Q$
 and *minor*: $\llbracket P; Q \rrbracket \Longrightarrow R \llbracket \neg P; \neg Q \rrbracket \Longrightarrow R$
 shows R
 by (*rule major [THEN iffE]*) (*iprover intro: minor elim: impCE notE*)

lemma *exCI*:
 assumes $\forall x. \neg P x \Longrightarrow P a$
 shows $\exists x. P x$
 by (*rule ccontr*) (*iprover intro: assms exI allI notI notE [of $\exists x. P x$]*)

2.2.19 Intuitionistic Reasoning

lemma *impE'*:
 assumes *1*: $P \longrightarrow Q$
 and *2*: $Q \Longrightarrow R$
 and *3*: $P \longrightarrow Q \Longrightarrow P$
 shows R
proof –
 from *3* and *1* have P .
 with *1* have Q by (*rule impE*)
 with *2* show R .
qed

lemma *allE'*:
 assumes *1*: $\forall x. P x$
 and *2*: $P x \Longrightarrow \forall x. P x \Longrightarrow Q$
 shows Q
proof –
 from *1* have $P x$ by (*rule spec*)
 from *this* and *1* show Q by (*rule 2*)
qed

lemma *notE'*:
 assumes *1*: $\neg P$
 and *2*: $\neg P \Longrightarrow P$
 shows R

proof –
 from 2 and 1 have P .
 with 1 show R by (rule notE)
qed

lemma *TrueE*: $True \implies P \implies P$.
lemma *notFalseE*: $\neg False \implies P \implies P$.

lemmas [*Pure.elim!*] = *disjE iffE FalseE conjE exE TrueE notFalseE*
 and [*Pure.intro!*] = *iffI conjI impI TrueI notI allI refl*
 and [*Pure.elim 2*] = *allE notE' impE'*
 and [*Pure.intro*] = *exI disjI2 disjI1*

lemmas [*trans*] = *trans*
 and [*sym*] = *sym not-sym*
 and [*Pure.elim?*] = *iffD1 iffD2 impE*

2.2.20 Atomizing meta-level connectives

axiomatization where

eq-reflection: $x = y \implies x \equiv y$ — admissible axiom

lemma *atomize-all* [*atomize*]: $(\bigwedge x. P x) \equiv Trueprop (\forall x. P x)$

proof
 assume $\bigwedge x. P x$
 then show $\forall x. P x$..
next
 assume $\forall x. P x$
 then show $\bigwedge x. P x$ by (rule allE)
qed

lemma *atomize-imp* [*atomize*]: $(A \implies B) \equiv Trueprop (A \longrightarrow B)$

proof
 assume $r: A \implies B$
 show $A \longrightarrow B$ by (rule impI) (rule r)
next
 assume $A \longrightarrow B$ and A
 then show B by (rule mp)
qed

lemma *atomize-not*: $(A \implies False) \equiv Trueprop (\neg A)$

proof
 assume $r: A \implies False$
 show $\neg A$ by (rule notI) (rule r)
next
 assume $\neg A$ and A
 then show $False$ by (rule notE)
qed

```

lemma atomize-eq [atomize, code]:  $(x \equiv y) \equiv \text{Trueprop } (x = y)$ 
proof
  assume  $x \equiv y$ 
  show  $x = y$  by (unfold  $\langle x \equiv y \rangle$ ) (rule refl)
next
  assume  $x = y$ 
  then show  $x \equiv y$  by (rule eq-reflection)
qed

```

```

lemma atomize-conj [atomize]:  $(A \ \&\&\& \ B) \equiv \text{Trueprop } (A \wedge B)$ 
proof
  assume conj:  $A \ \&\&\& \ B$ 
  show  $A \wedge B$ 
  proof (rule conjI)
    from conj show  $A$  by (rule conjunctionD1)
    from conj show  $B$  by (rule conjunctionD2)
  qed
next
  assume conj:  $A \wedge B$ 
  show  $A \ \&\&\& \ B$ 
  proof –
    from conj show  $A$  ..
    from conj show  $B$  ..
  qed
qed

```

```

lemmas [symmetric, rulify] = atomize-all atomize-imp
and [symmetric, defn] = atomize-all atomize-imp atomize-eq

```

2.2.21 Atomizing elimination rules

```

lemma atomize-exL[atomize-elim]:  $(\bigwedge x. P \ x \implies Q) \equiv ((\exists x. P \ x) \implies Q)$ 
by (rule equal-intr-rule) iprover+

```

```

lemma atomize-conjL[atomize-elim]:  $(A \implies B \implies C) \equiv (A \wedge B \implies C)$ 
by (rule equal-intr-rule) iprover+

```

```

lemma atomize-disjL[atomize-elim]:  $((A \implies C) \implies (B \implies C) \implies C) \equiv ((A \vee B \implies C) \implies C)$ 
by (rule equal-intr-rule) iprover+

```

```

lemma atomize-elimL[atomize-elim]:  $(\bigwedge B. (A \implies B) \implies B) \equiv \text{Trueprop } A \ ..$ 

```

2.3 Package setup

```

ML-file  $\langle \text{Tools/hologic.ML} \rangle$ 

```

```

ML-file  $\langle \text{Tools/rewrite-hol-proof.ML} \rangle$ 

```

```

setup  $\langle \text{Proofterm.set-preproc } (\text{Proof-Rewrite-Rules.standard-preproc Rewrite-HOL-Proof.rews}) \rangle$ 

```

2.3.1 Sledgehammer setup

Theorems blacklisted to Sledgehammer. These theorems typically produce clauses that are prolific (match too many equality or membership literals) and relate to seldom-used facts. Some duplicate other rules.

named-theorems *no-atp theorems that should be filtered out by Sledgehammer*

2.3.2 Classical Reasoner setup

lemma *imp-elim*: $P \longrightarrow Q \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by *(rule classical) iprover*

lemma *swap*: $\neg P \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow R$
by *(rule classical) iprover*

lemma *thin-refl*: $\llbracket x = x; PROP W \rrbracket \Longrightarrow PROP W$.

ML <

```
structure Hypsubst = Hypsubst
(
  val dest-eq = HOLogic.dest-eq
  val dest-Trueprop = HOLogic.dest-Trueprop
  val dest-imp = HOLogic.dest-imp
  val eq-reflection = @{thm eq-reflection}
  val rev-eq-reflection = @{thm meta-eq-to-obj-eq}
  val imp-intr = @{thm impI}
  val rev-mp = @{thm rev-mp}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl};
);
open Hypsubst;

structure Classical = Classical
(
  val imp-elim = @{thm imp-elim}
  val not-elim = @{thm notE}
  val swap = @{thm swap}
  val classical = @{thm classical}
  val sizef = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
);

structure Basic-Classical: BASIC-CLASSICAL = Classical;
open Basic-Classical;
>
```

setup <

*(*prevent substitution on bool*)*


```

let
  fun non-bool-eq (const-name <HOL.eq>, Type (-, [T, -])) = T <> typ <bool>
    | non-bool-eq - = false;
  fun hyp-subst-tac' ctxt =
    SUBGOAL (fn (goal, i) =>
      if Term.exists-Const non-bool-eq goal
      then Hypsubst.hyp-subst-tac ctxt i
      else no-tac);
in
  Context-Rules.addSWrapper (fn ctxt => fn tac => hyp-subst-tac' ctxt ORELSE'
tac)
end
>

```

```

declare iffI [intro!]
and notI [intro!]
and impI [intro!]
and disjCI [intro!]
and conjI [intro!]
and TrueI [intro!]
and refl [intro!]

```

```

declare iffCE [elim!]
and FalseE [elim!]
and impCE [elim!]
and disjE [elim!]
and conjE [elim!]

```

```

declare ex-ex1I [intro!]
and allI [intro!]
and exI [intro]

```

```

declare exE [elim!]
  allE [elim]

```

ML <val HOL-cs = claset-of **context**>

```

lemma contrapos-np:  $\neg Q \implies (\neg P \implies Q) \implies P$ 
by (erule swap)

```

```

declare ex-ex1I [rule del, intro! 2]
and ex1I [intro]

```

```

declare ext [intro]

```

```

lemmas [intro?] = ext
and [elim?] = ex1-implies-ex

```

Better than *ex1E* for classical reasoner: needs no quantifier duplication!

lemma *alt-ex1E* [*elim!*]:
assumes *major*: $\exists!x. P x$
and *minor*: $\bigwedge x. \llbracket P x; \forall y y'. P y \wedge P y' \longrightarrow y = y' \rrbracket \Longrightarrow R$
shows R
proof (*rule ex1E* [*OF major minor*])
show $\forall y y'. P y \wedge P y' \longrightarrow y = y'$ **if** $P x$ **and** $\S: \forall y. P y \longrightarrow y = x$ **for** x
using $\langle P x \rangle \S \S$ **by** *fast*
qed *assumption*

And again using *Uniq*

lemma *alt-ex1E'*:
assumes $\exists!x. P x \wedge x. \llbracket P x; \exists_{\leq 1}x. P x \rrbracket \Longrightarrow R$
shows R
using *assms unfolding Uniq-def* **by** *fast*

lemma *ex1-iff-ex-Uniq*: $(\exists!x. P x) \longleftrightarrow (\exists x. P x) \wedge (\exists_{\leq 1}x. P x)$
unfolding *Uniq-def* **by** *fast*

ML \langle

```

structure Blast = Blast
(
  structure Classical = Classical
  val Trueprop-const = dest-Const Const  $\langle$ Trueprop $\rangle$ 
  val equality-name = const-name  $\langle$ HOL.eq $\rangle$ 
  val not-name = const-name  $\langle$ Not $\rangle$ 
  val notE = @{thm notE}
  val ccontr = @{thm ccontr}
  val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
);
val blast-tac = Blast.blast-tac;

```

2.3.3 THE: definite description operator

lemma *the-equality* [*intro*]:
assumes $P a$
and $\bigwedge x. P x \Longrightarrow x = a$
shows $(THE x. P x) = a$
by (*blast intro: assms trans* [*OF arg-cong* [**where** $f=The$] *the-eq-trivial*])

lemma *theI*:
assumes $P a$
and $\bigwedge x. P x \Longrightarrow x = a$
shows $P (THE x. P x)$
by (*iprover intro: assms the-equality* [*THEN ssubst*])

lemma *theI'*: $\exists!x. P x \Longrightarrow P (THE x. P x)$
by (*blast intro: theI*)

Easier to apply than *theI*: only one occurrence of *P*.

lemma *theI2*:

assumes $P\ a\ \wedge x. P\ x \implies x = a\ \wedge x. P\ x \implies Q\ x$
shows $Q\ (THE\ x. P\ x)$
by (*iprover intro: assms theI*)

lemma *the1I2*:

assumes $\exists!x. P\ x\ \wedge x. P\ x \implies Q\ x$
shows $Q\ (THE\ x. P\ x)$
by (*iprover intro: assms(2) theI2[where P=P and Q=Q] ex1E[OF assms(1)]*)
elim: allE impE

lemma *the1-equality* [*elim?*]: $[\exists!x. P\ x; P\ a] \implies (THE\ x. P\ x) = a$
by *blast*

lemma *the1-equality'*: $[\exists_{\leq 1}x. P\ x; P\ a] \implies (THE\ x. P\ x) = a$
unfolding *Uniq-def* **by** *blast*

lemma *the-sym-eq-trivial*: $(THE\ y. x = y) = x$
by *blast*

2.3.4 Simplifier

lemma *eta-contract-eq*: $(\lambda s. f\ s) = f\ ..$

lemma *subst-all*:

$\langle \wedge x. x = a \implies PROP\ P\ x \rangle \equiv PROP\ P\ a$
 $\langle \wedge x. a = x \implies PROP\ P\ x \rangle \equiv PROP\ P\ a$

proof –

show $\langle \wedge x. x = a \implies PROP\ P\ x \rangle \equiv PROP\ P\ a$

proof (*rule equal-intr-rule*)

assume *: $\langle \wedge x. x = a \implies PROP\ P\ x \rangle$

show $\langle PROP\ P\ a \rangle$

by (*rule **) (*rule refl*)

next

fix *x*

assume $\langle PROP\ P\ a \rangle$ **and** $\langle x = a \rangle$

from $\langle x = a \rangle$ **have** $\langle x \equiv a \rangle$

by (*rule eq-reflection*)

with $\langle PROP\ P\ a \rangle$ **show** $\langle PROP\ P\ x \rangle$

by *simp*

qed

show $\langle \wedge x. a = x \implies PROP\ P\ x \rangle \equiv PROP\ P\ a$

proof (*rule equal-intr-rule*)

assume *: $\langle \wedge x. a = x \implies PROP\ P\ x \rangle$

show $\langle PROP\ P\ a \rangle$

by (*rule **) (*rule refl*)

next

fix *x*

assume $\langle PROP P a \rangle$ **and** $\langle a = x \rangle$
from $\langle a = x \rangle$ **have** $\langle a \equiv x \rangle$
by (rule eq-reflection)
with $\langle PROP P a \rangle$ **show** $\langle PROP P x \rangle$
by simp
qed
qed

lemma simp-thms:

shows not-not: $(\neg \neg P) = P$
and Not-eq-iff: $((\neg P) = (\neg Q)) = (P = Q)$
and
 $(P \neq Q) = (P = (\neg Q))$
 $(P \vee \neg P) = True \quad (\neg P \vee P) = True$
 $(x = x) = True$
and not-True-eq-False [code]: $(\neg True) = False$
and not-False-eq-True [code]: $(\neg False) = True$
and
 $(\neg P) \neq P \quad P \neq (\neg P)$
 $(True = P) = P$
and eq-True: $(P = True) = P$
and $(False = P) = (\neg P)$
and eq-False: $(P = False) = (\neg P)$
and
 $(True \longrightarrow P) = P \quad (False \longrightarrow P) = True$
 $(P \longrightarrow True) = True \quad (P \longrightarrow P) = True$
 $(P \longrightarrow False) = (\neg P) \quad (P \longrightarrow \neg P) = (\neg P)$
 $(P \wedge True) = P \quad (True \wedge P) = P$
 $(P \wedge False) = False \quad (False \wedge P) = False$
 $(P \wedge P) = P \quad (P \wedge (P \wedge Q)) = (P \wedge Q)$
 $(P \wedge \neg P) = False \quad (\neg P \wedge P) = False$
 $(P \vee True) = True \quad (True \vee P) = True$
 $(P \vee False) = P \quad (False \vee P) = P$
 $(P \vee P) = P \quad (P \vee (P \vee Q)) = (P \vee Q)$ **and**
 $(\forall x. P) = P \quad (\exists x. P) = P \quad \exists x. x = t \quad \exists x. t = x$
and
 $\bigwedge P. (\exists x. x = t \wedge P x) = P t$
 $\bigwedge P. (\exists x. t = x \wedge P x) = P t$
 $\bigwedge P. (\forall x. x = t \longrightarrow P x) = P t$
 $\bigwedge P. (\forall x. t = x \longrightarrow P x) = P t$
 $(\forall x. x \neq t) = False \quad (\forall x. t \neq x) = False$
by (blast, blast, blast, blast, blast, iprover+)

lemma disj-absorb: $A \vee A \longleftrightarrow A$
by blast

lemma disj-left-absorb: $A \vee (A \vee B) \longleftrightarrow A \vee B$
by blast

lemma *conj-absorb*: $A \wedge A \longleftrightarrow A$
by *blast*

lemma *conj-left-absorb*: $A \wedge (A \wedge B) \longleftrightarrow A \wedge B$
by *blast*

lemma *eq-ac*:
shows *eq-commute*: $a = b \longleftrightarrow b = a$
and *iff-left-commute*: $(P \longleftrightarrow (Q \longleftrightarrow R)) \longleftrightarrow (Q \longleftrightarrow (P \longleftrightarrow R))$
and *iff-assoc*: $((P \longleftrightarrow Q) \longleftrightarrow R) \longleftrightarrow (P \longleftrightarrow (Q \longleftrightarrow R))$
by (*iprover*, *blast+*)

lemma *neq-commute*: $a \neq b \longleftrightarrow b \neq a$ **by** *iprover*

lemma *conj-comms*:
shows *conj-commute*: $P \wedge Q \longleftrightarrow Q \wedge P$
and *conj-left-commute*: $P \wedge (Q \wedge R) \longleftrightarrow Q \wedge (P \wedge R)$ **by** *iprover+*
lemma *conj-assoc*: $(P \wedge Q) \wedge R \longleftrightarrow P \wedge (Q \wedge R)$ **by** *iprover*

lemmas *conj-ac* = *conj-commute conj-left-commute conj-assoc*

lemma *disj-comms*:
shows *disj-commute*: $P \vee Q \longleftrightarrow Q \vee P$
and *disj-left-commute*: $P \vee (Q \vee R) \longleftrightarrow Q \vee (P \vee R)$ **by** *iprover+*
lemma *disj-assoc*: $(P \vee Q) \vee R \longleftrightarrow P \vee (Q \vee R)$ **by** *iprover*

lemmas *disj-ac* = *disj-commute disj-left-commute disj-assoc*

lemma *conj-disj-distribL*: $P \wedge (Q \vee R) \longleftrightarrow P \wedge Q \vee P \wedge R$ **by** *iprover*
lemma *conj-disj-distribR*: $(P \vee Q) \wedge R \longleftrightarrow P \wedge R \vee Q \wedge R$ **by** *iprover*

lemma *disj-conj-distribL*: $P \vee (Q \wedge R) \longleftrightarrow (P \vee Q) \wedge (P \vee R)$ **by** *iprover*
lemma *disj-conj-distribR*: $(P \wedge Q) \vee R \longleftrightarrow (P \vee R) \wedge (Q \vee R)$ **by** *iprover*

lemma *imp-conjR*: $(P \longrightarrow (Q \wedge R)) = ((P \longrightarrow Q) \wedge (P \longrightarrow R))$ **by** *iprover*

lemma *imp-conjL*: $((P \wedge Q) \longrightarrow R) = (P \longrightarrow (Q \longrightarrow R))$ **by** *iprover*

lemma *imp-disjL*: $((P \vee Q) \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ **by** *iprover*

These two are specialized, but *imp-disj-not1* is useful in *Auth/Yahalom*.

lemma *imp-disj-not1*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg Q \longrightarrow P \longrightarrow R)$ **by** *blast*

lemma *imp-disj-not2*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg R \longrightarrow P \longrightarrow Q)$ **by** *blast*

lemma *imp-disj1*: $((P \longrightarrow Q) \vee R) \longleftrightarrow (P \longrightarrow Q \vee R)$ **by** *blast*

lemma *imp-disj2*: $(Q \vee (P \longrightarrow R)) \longleftrightarrow (P \longrightarrow Q \vee R)$ **by** *blast*

lemma *imp-cong*: $(P = P') \Longrightarrow (P' \Longrightarrow (Q = Q')) \Longrightarrow ((P \longrightarrow Q) \longleftrightarrow (P' \longrightarrow Q'))$

by *iprover*

lemma *de-Morgan-disj*: $\neg (P \vee Q) \longleftrightarrow \neg P \wedge \neg Q$ **by** *iprover*
lemma *de-Morgan-conj*: $\neg (P \wedge Q) \longleftrightarrow \neg P \vee \neg Q$ **by** *blast*
lemma *not-imp*: $\neg (P \longrightarrow Q) \longleftrightarrow P \wedge \neg Q$ **by** *blast*
lemma *not-iff*: $P \neq Q \longleftrightarrow (P \longleftrightarrow \neg Q)$ **by** *blast*
lemma *disj-not1*: $\neg P \vee Q \longleftrightarrow (P \longrightarrow Q)$ **by** *blast*
lemma *disj-not2*: $P \vee \neg Q \longleftrightarrow (Q \longrightarrow P)$ **by** *blast* — changes orientation :-(
lemma *imp-conv-disj*: $(P \longrightarrow Q) \longleftrightarrow (\neg P) \vee Q$ **by** *blast*
lemma *disj-imp*: $P \vee Q \longleftrightarrow \neg P \longrightarrow Q$ **by** *blast*

lemma *iff-conv-conj-imp*: $(P \longleftrightarrow Q) \longleftrightarrow (P \longrightarrow Q) \wedge (Q \longrightarrow P)$ **by** *iprover*

lemma *cases-simp*: $(P \longrightarrow Q) \wedge (\neg P \longrightarrow Q) \longleftrightarrow Q$
— Avoids duplication of subgoals after *if-split*, when the true and false
— cases boil down to the same thing.
by *blast*

lemma *not-all*: $\neg (\forall x. P x) \longleftrightarrow (\exists x. \neg P x)$ **by** *blast*
lemma *imp-all*: $((\forall x. P x) \longrightarrow Q) \longleftrightarrow (\exists x. P x \longrightarrow Q)$ **by** *blast*
lemma *not-ex*: $\neg (\exists x. P x) \longleftrightarrow (\forall x. \neg P x)$ **by** *iprover*
lemma *imp-ex*: $((\exists x. P x) \longrightarrow Q) \longleftrightarrow (\forall x. P x \longrightarrow Q)$ **by** *iprover*
lemma *all-not-ex*: $(\forall x. P x) \longleftrightarrow \neg (\exists x. \neg P x)$ **by** *blast*

declare *All-def* [*no-atp*]

lemma *ex-disj-distrib*: $(\exists x. P x \vee Q x) \longleftrightarrow (\exists x. P x) \vee (\exists x. Q x)$ **by** *iprover*
lemma *all-conj-distrib*: $(\forall x. P x \wedge Q x) \longleftrightarrow (\forall x. P x) \wedge (\forall x. Q x)$ **by** *iprover*

The \wedge congruence rule: not included by default! May slow rewrite proofs down by as much as 50%

lemma *conj-cong*: $P = P' \Longrightarrow (P' \Longrightarrow Q = Q') \Longrightarrow (P \wedge Q) = (P' \wedge Q')$
by *iprover*

lemma *rev-conj-cong*: $Q = Q' \Longrightarrow (Q' \Longrightarrow P = P') \Longrightarrow (P \wedge Q) = (P' \wedge Q')$
by *iprover*

The $|$ congruence rule: not included by default!

lemma *disj-cong*: $P = P' \Longrightarrow (\neg P' \Longrightarrow Q = Q') \Longrightarrow (P \vee Q) = (P' \vee Q')$
by *blast*

if-then-else rules

lemma *if-True* [*code*]: $(\text{if True then } x \text{ else } y) = x$
unfolding *If-def* **by** *blast*

lemma *if-False* [*code*]: $(\text{if False then } x \text{ else } y) = y$
unfolding *If-def* **by** *blast*

lemma *if-P*: $P \Longrightarrow (\text{if } P \text{ then } x \text{ else } y) = x$

unfolding *If-def* by *blast*

lemma *if-not-P*: $\neg P \implies (\text{if } P \text{ then } x \text{ else } y) = y$
unfolding *If-def* by *blast*

lemma *if-split*: $P (\text{if } Q \text{ then } x \text{ else } y) = ((Q \longrightarrow P x) \wedge (\neg Q \longrightarrow P y))$

proof (*rule case-split* [of *Q*])

show *?thesis* if *Q*

using *that* by (*simplesubst if-P*) *blast+*

show *?thesis* if $\neg Q$

using *that* by (*simplesubst if-not-P*) *blast+*

qed

lemma *if-split-asm*: $P (\text{if } Q \text{ then } x \text{ else } y) = (\neg ((Q \wedge \neg P x) \vee (\neg Q \wedge \neg P y)))$
by (*simplesubst if-split*) *blast*

lemmas *if-splits* [*no-atp*] = *if-split if-split-asm*

lemma *if-cancel*: $(\text{if } c \text{ then } x \text{ else } x) = x$
by (*simplesubst if-split*) *blast*

lemma *if-eq-cancel*: $(\text{if } x = y \text{ then } y \text{ else } x) = x$
by (*simplesubst if-split*) *blast*

lemma *if-bool-eq-conj*: $(\text{if } P \text{ then } Q \text{ else } R) = ((P \longrightarrow Q) \wedge (\neg P \longrightarrow R))$
 — This form is useful for expanding *ifs* on the RIGHT of the \implies symbol.
by (*rule if-split*)

lemma *if-bool-eq-disj*: $(\text{if } P \text{ then } Q \text{ else } R) = ((P \wedge Q) \vee (\neg P \wedge R))$
 — And this form is useful for expanding *ifs* on the LEFT.
by (*simplesubst if-split*) *blast*

lemma *Eq-TrueI*: $P \implies P \equiv \text{True}$ **unfolding** *atomize-eq* **by** *iprover*

lemma *Eq-FalseI*: $\neg P \implies P \equiv \text{False}$ **unfolding** *atomize-eq* **by** *iprover*

let rules for *simproc*

lemma *Let-folded*: $f x \equiv g x \implies \text{Let } x f \equiv \text{Let } x g$
by (*unfold Let-def*)

lemma *Let-unfold*: $f x \equiv g \implies \text{Let } x f \equiv g$
by (*unfold Let-def*)

The following copy of the implication operator is useful for fine-tuning congruence rules. It instructs the simplifier to simplify its premise.

definition *simp-implies* :: *prop* \Rightarrow *prop* \Rightarrow *prop* (**infixr** =*simp=>* 1)
where *simp-implies* \equiv (\implies)

lemma *simp-impliesI*:

assumes PQ : $(PROP\ P \implies PROP\ Q)$
shows $PROP\ P =_{simp}=> PROP\ Q$
unfolding *simp-implies-def*
by (*iprover intro: PQ*)

lemma *simp-impliesE*:
assumes PQ : $PROP\ P =_{simp}=> PROP\ Q$
and P : $PROP\ P$
and QR : $PROP\ Q \implies PROP\ R$
shows $PROP\ R$
by (*iprover intro: QR P PQ [unfolded simp-implies-def]*)

lemma *simp-implies-cong*:
assumes PP' : $PROP\ P \equiv PROP\ P'$
and $P'QQ'$: $PROP\ P' \implies (PROP\ Q \equiv PROP\ Q')$
shows $(PROP\ P =_{simp}=> PROP\ Q) \equiv (PROP\ P' =_{simp}=> PROP\ Q')$
unfolding *simp-implies-def*
proof (*rule equal-intr-rule*)
assume PQ : $PROP\ P \implies PROP\ Q$
and P' : $PROP\ P'$
from PP' [*symmetric*] **and** P' **have** $PROP\ P$
by (*rule equal-elim-rule1*)
then have $PROP\ Q$ **by** (*rule PQ*)
with $P'QQ'$ [*OF P'*] **show** $PROP\ Q'$ **by** (*rule equal-elim-rule1*)
next
assume $P'Q'$: $PROP\ P' \implies PROP\ Q'$
and P : $PROP\ P$
from PP' **and** P **have** P' : $PROP\ P'$ **by** (*rule equal-elim-rule1*)
then have $PROP\ Q'$ **by** (*rule P'Q'*)
with $P'QQ'$ [*OF P', symmetric*] **show** $PROP\ Q$
by (*rule equal-elim-rule1*)
qed

lemma *uncurry*:
assumes $P \longrightarrow Q \longrightarrow R$
shows $P \wedge Q \longrightarrow R$
using *assms* **by** *blast*

lemma *iff-allI*:
assumes $\bigwedge x. P\ x = Q\ x$
shows $(\forall x. P\ x) = (\forall x. Q\ x)$
using *assms* **by** *blast*

lemma *iff-exI*:
assumes $\bigwedge x. P\ x = Q\ x$
shows $(\exists x. P\ x) = (\exists x. Q\ x)$
using *assms* **by** *blast*

lemma *all-comm*: $(\forall x\ y. P\ x\ y) = (\forall y\ x. P\ x\ y)$

by *blast*

lemma *ex-comm*: $(\exists x y. P x y) = (\exists y x. P x y)$
by *blast*

ML-file $\langle \text{Tools/simpdata.ML} \rangle$

ML $\langle \text{open Simpdata} \rangle$

setup \langle
 map-theory-simpset (*put-simpset* *HOL-basic-ss*) # \rangle
 Simplifier.method-setup *Splitter.split-modifiers*
 \rangle

simproc-setup *defined-Ex* $(\exists x. P x) = \langle K \text{ Quantifier1.rearrange-Ex} \rangle$

simproc-setup *defined-All* $(\forall x. P x) = \langle K \text{ Quantifier1.rearrange-All} \rangle$

simproc-setup *defined-all* $(\bigwedge x. \text{PROP } P x) = \langle K \text{ Quantifier1.rearrange-all} \rangle$

Simproc for proving $(y = x) \equiv \text{False}$ from premise $\neg (x = y)$:

simproc-setup *neq* $(x = y) = \langle$
 let
 val *neq-to-EQ-False* = $\text{@}\{thm \text{ not-sym}\} \text{ RS } \text{@}\{thm \text{ Eq-FalseI}\};$
 fun *is-neq* *eq lhs rhs thm* =
 (*case* *Thm.prop-of thm of*
 - $\$ (\text{Not } \$ (eq' \$ l' \$ r')) \Rightarrow$
 Not = HOLogic.Not andalso eq' = eq andalso
 r' aconv lhs andalso l' aconv rhs
 | $\Rightarrow \text{false}$);
 fun *proc ss ct* =
 (*case* *Thm.term-of ct of*
 eq $\$ \text{lhs} \$ \text{rhs} \Rightarrow$
 (*case* *find-first* (*is-neq* *eq lhs rhs*) (*Simplifier.premis-of ss*) *of*
 SOME *thm* $\Rightarrow \text{SOME } (thm \text{ RS } neq-to-EQ-False)$
 | *NONE* $\Rightarrow \text{NONE}$)
 | $\Rightarrow \text{NONE}$);
 in *K proc* *end*
 \rangle

simproc-setup *let-simp* $(\text{Let } x f) = \langle$

let
 fun *count-loose* (*Bound* *i*) *k* = *if* *i* $\geq k$ *then* 1 *else* 0
 | *count-loose* (*s* $\$$ *t*) *k* = *count-loose* *s* *k* + *count-loose* *t* *k*
 | *count-loose* (*Abs* $(-, -, t)$) *k* = *count-loose* *t* (*k* + 1)
 | *count-loose* - - = 0;
 fun *is-trivial-let* (*Const* (*const-name* $\langle \text{Let} \rangle$, -) $\$$ *x* $\$$ *t*) =
 (*case* *t of*
 Abs $(-, -, t')$ $\Rightarrow \text{count-loose } t' 0 \leq 1$
 | $\Rightarrow \text{true}$);
 in
 K (*fn* *ctxt* \Rightarrow *fn* *ct* \Rightarrow

```

if is-trivial-let (Thm.term-of ct)
then SOME @{thm Let-def} (*no or one occurrence of bound variable*)
else
  let (*Norbert Schirmer's case*)
    val t = Thm.term-of ct;
    val (t', ctxt') = yield-singleton (Variable.import-terms false) t ctxt;
  in
    Option.map (hd o Variable.export ctxt' ctxt o single)
      (case t' of Const (const-name <Let>, -) $ x $ f => (* x and f are already
in normal form *))
      if is-Free x orelse is-Bound x orelse is-Const x
      then SOME @{thm Let-def}
      else
        let
          val n = case f of (Abs (x, -, -)) => x | - => x;
          val cx = Thm.cterm-of ctxt x;
          val xT = Thm.typ-of-cterm cx;
          val cf = Thm.cterm-of ctxt f;
          val fx-g = Simplifier.rewrite ctxt (Thm.apply cf cx);
          val (- $ - $ g) = Thm.prop-of fx-g;
          val g' = abstract-over (x, g);
          val abs-g' = Abs (n, xT, g');
        in
          if g aconv g' then
            let
              val rl =
                infer-instantiate ctxt [((f, 0), cf), ((x, 0), cx)] @{thm Let-unfold};
              in SOME (rl OF [fx-g]) end
            else if (Envir.beta-eta-contract f) aconv (Envir.beta-eta-contract
abs-g')
            then NONE (*avoid identity conversion*)
            else
              let
                val g'x = abs-g' $ x;
                val g-g'x = Thm.symmetric (Thm.beta-conversion false
(Thm.cterm-of ctxt g'x));
              in
                val rl =
                  @{thm Let-folded} |> infer-instantiate ctxt
                    [((f, 0), Thm.cterm-of ctxt f),
                     ((x, 0), cx),
                     ((g, 0), Thm.cterm-of ctxt abs-g')];
                in SOME (rl OF [Thm.transitive fx-g g-g'x]) end
              end
            | - => NONE)
          end
        end
      end
    >

```

lemma *True-implies-equals*: ($\text{True} \implies \text{PROP } P$) \equiv $\text{PROP } P$

proof

```

assume True  $\implies$  PROP P
from this [OF TrueI] show PROP P .
next
  assume PROP P
  then show PROP P .
qed

```

lemma *implies-True-equals*: $(PROP P \implies True) \equiv \text{Trueprop True}$
by *standard* (intro TrueI)

lemma *False-implies-equals*: $(False \implies P) \equiv \text{Trueprop True}$
by *standard simp-all*

lemma *implies-False-swap*:

```

 $(False \implies PROP P \implies PROP Q) \equiv (PROP P \implies False \implies PROP Q)$ 
by (rule swap-prems-eq)

```

ML \langle

```

fun eliminate-false-implies ct =

```

```

  let

```

```

    val (prems, concl) = Logic.strip-horn (Thm.term-of ct)

```

```

    fun go n =

```

```

      if n > 1 then

```

```

        Conv.rewr-conv @ {thm Pure.swap-prems-eq}

```

```

        then-conv Conv.arg-conv (go (n - 1))

```

```

        then-conv Conv.rewr-conv @ {thm HOL.implies-True-equals}

```

```

      else

```

```

        Conv.rewr-conv @ {thm HOL.False-implies-equals}

```

```

    in

```

```

      case concl of

```

```

        Const (@{const-name HOL.Trueprop}, -) $ - => SOME (go (length prems))

```

```

      ct)

```

```

        | - => NONE

```

```

      end

```

```

     $\rangle$ 

```

simproc-setup *eliminate-false-implies* $(False \implies PROP P) = \langle K (K \text{ eliminate-false-implies}) \rangle$

lemma *ex-simps*:

```

 $\bigwedge P Q. (\exists x. P x \wedge Q) = ((\exists x. P x) \wedge Q)$ 

```

```

 $\bigwedge P Q. (\exists x. P \wedge Q x) = (P \wedge (\exists x. Q x))$ 

```

```

 $\bigwedge P Q. (\exists x. P x \vee Q) = ((\exists x. P x) \vee Q)$ 

```

```

 $\bigwedge P Q. (\exists x. P \vee Q x) = (P \vee (\exists x. Q x))$ 

```

```

 $\bigwedge P Q. (\exists x. P x \longrightarrow Q) = ((\forall x. P x) \longrightarrow Q)$ 

```

```

 $\bigwedge P Q. (\exists x. P \longrightarrow Q x) = (P \longrightarrow (\exists x. Q x))$ 

```

— Miniscoping: pushing in existential quantifiers.

by (*iprover* | *blast*)+

lemma *all-simps*:

$\bigwedge P Q. (\forall x. P x \wedge Q) = ((\forall x. P x) \wedge Q)$
 $\bigwedge P Q. (\forall x. P \wedge Q x) = (P \wedge (\forall x. Q x))$
 $\bigwedge P Q. (\forall x. P x \vee Q) = ((\forall x. P x) \vee Q)$
 $\bigwedge P Q. (\forall x. P \vee Q x) = (P \vee (\forall x. Q x))$
 $\bigwedge P Q. (\forall x. P x \longrightarrow Q) = ((\exists x. P x) \longrightarrow Q)$
 $\bigwedge P Q. (\forall x. P \longrightarrow Q x) = (P \longrightarrow (\forall x. Q x))$
 — Miniscoping: pushing in universal quantifiers.
 by (*iprover* | *blast*)+

lemmas [*simp*] =

triv-forall-equality — prunes params
True-implies-equals implies-True-equals — prune *True* in asms
False-implies-equals — prune *False* in asms
if-True
if-False
if-cancel
if-eq-cancel

imp-disjL — In general it seems wrong to add distributive laws by default: they might cause exponential blow-up. But *imp-disjL* has been in for a while and cannot be removed without affecting existing proofs. Moreover, rewriting by $(P \vee Q \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ might be justified on the grounds that it allows simplification of *R* in the two cases.

conj-assoc
disj-assoc
de-Morgan-conj
de-Morgan-disj
imp-disj1
imp-disj2
not-imp
disj-not1
not-all
not-ex
cases-simp
the-eq-trivial
the-sym-eq-trivial
ex-simps
all-simps
simp-thms
subst-all

lemmas [*cong*] = *imp-cong* *simp-implies-cong*

lemmas [*split*] = *if-split*

ML $\langle \text{val } \text{HOL-ss} = \text{simpset-of } \text{context} \rangle$

Simplifies *x* assuming *c* and *y* assuming $\neg c$.

lemma *if-cong*:
assumes $b = c$
and $c \implies x = u$
and $\neg c \implies y = v$
shows $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$
using *assms* **by** *simp*

Prevents simplification of x and y : faster and allows the execution of functional programs.

lemma *if-weak-cong* [*cong*]:
assumes $b = c$
shows $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$
using *assms* **by** (*rule arg-cong*)

Prevents simplification of t : much faster

lemma *let-weak-cong*:
assumes $a = b$
shows $(\text{let } x = a \text{ in } t \ x) = (\text{let } x = b \text{ in } t \ x)$
using *assms* **by** (*rule arg-cong*)

To tidy up the result of a *simproc*. Only the RHS will be simplified.

lemma *eq-cong2*:
assumes $u = u'$
shows $(t \equiv u) \equiv (t \equiv u')$
using *assms* **by** *simp*

lemma *if-distrib*: $f (\text{if } c \text{ then } x \text{ else } y) = (\text{if } c \text{ then } f \ x \text{ else } f \ y)$
by *simp*

lemma *if-distribR*: $(\text{if } b \text{ then } f \text{ else } g) \ x = (\text{if } b \text{ then } f \ x \text{ else } g \ x)$
by *simp*

lemma *all-if-distrib*: $(\forall x. \text{if } x = a \text{ then } P \ x \text{ else } Q \ x) \longleftrightarrow P \ a \wedge (\forall x. x \neq a \longrightarrow Q \ x)$
by *auto*

lemma *ex-if-distrib*: $(\exists x. \text{if } x = a \text{ then } P \ x \text{ else } Q \ x) \longleftrightarrow P \ a \vee (\exists x. x \neq a \wedge Q \ x)$
by *auto*

lemma *if-if-eq-conj*: $(\text{if } P \text{ then if } Q \text{ then } x \text{ else } y \text{ else } y) = (\text{if } P \wedge Q \text{ then } x \text{ else } y)$
by *simp*

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma *fun-eq-iff*: $f = g \longleftrightarrow (\forall x. f \ x = g \ x)$
by *auto*

2.3.5 Generic cases and induction

Rule projections:

```
ML <
structure Project-Rule = Project-Rule
(
  val conjunct1 = @{thm conjunct1}
  val conjunct2 = @{thm conjunct2}
  val mp = @{thm mp}
);
>
```

```
context
begin
```

qualified definition $induct-forall\ P \equiv \forall x. P\ x$

qualified definition $induct-implies\ A\ B \equiv A \longrightarrow B$

qualified definition $induct-equal\ x\ y \equiv x = y$

qualified definition $induct-conj\ A\ B \equiv A \wedge B$

qualified definition $induct-true \equiv True$

qualified definition $induct-false \equiv False$

lemma $induct-forall-eq: (\bigwedge x. P\ x) \equiv Trueprop\ (induct-forall\ (\lambda x. P\ x))$
by (*unfold atomize-all induct-forall-def*)

lemma $induct-implies-eq: (A \implies B) \equiv Trueprop\ (induct-implies\ A\ B)$
by (*unfold atomize-imp induct-implies-def*)

lemma $induct-equal-eq: (x \equiv y) \equiv Trueprop\ (induct-equal\ x\ y)$
by (*unfold atomize-eq induct-equal-def*)

lemma $induct-conj-eq: (A \ \&\&\ B) \equiv Trueprop\ (induct-conj\ A\ B)$
by (*unfold atomize-conj induct-conj-def*)

lemmas $induct-atomize' = induct-forall-eq\ induct-implies-eq\ induct-conj-eq$

lemmas $induct-atomize = induct-atomize'\ induct-equal-eq$

lemmas $induct-rulify'\ [symmetric] = induct-atomize'$

lemmas $induct-rulify\ [symmetric] = induct-atomize$

lemmas $induct-rulify-fallback =$

induct-forall-def induct-implies-def induct-equal-def induct-conj-def
induct-true-def induct-false-def

lemma $induct-forall-conj: induct-forall\ (\lambda x. induct-conj\ (A\ x)\ (B\ x)) =$
 $induct-conj\ (induct-forall\ A)\ (induct-forall\ B)$
by (*unfold induct-forall-def induct-conj-def*) *iprover*

lemma $induct-implies-conj: induct-implies\ C\ (induct-conj\ A\ B) =$
 $induct-conj\ (induct-implies\ C\ A)\ (induct-implies\ C\ B)$
by (*unfold induct-implies-def induct-conj-def*) *iprover*

lemma *induct-conj-curry*: $(\text{induct-conj } A \ B \implies \text{PROP } C) \equiv (A \implies B \implies \text{PROP } C)$

proof

assume *r*: *induct-conj* *A B* \implies *PROP C*

assume *ab*: *A B*

show *PROP C* **by** (*rule r*) (*simp add: induct-conj-def ab*)

next

assume *r*: *A* \implies *B* \implies *PROP C*

assume *ab*: *induct-conj A B*

show *PROP C* **by** (*rule r*) (*simp-all add: ab [unfolded induct-conj-def]*)

qed

lemmas *induct-conj* = *induct-forall-conj induct-implies-conj induct-conj-curry*

lemma *induct-trueI*: *induct-true*

by (*simp add: induct-true-def*)

Method setup.

ML-file $\langle \sim \sim / \text{src} / \text{Tools} / \text{induct.ML} \rangle$

ML \langle

structure *Induct* = *Induct*

$($

val *cases-default* = $\@ \{ \text{thm case-split} \}$

val *atomize* = $\@ \{ \text{thms induct-atomize} \}$

val *rulify* = $\@ \{ \text{thms induct-rulify}' \}$

val *rulify-fallback* = $\@ \{ \text{thms induct-rulify-fallback} \}$

val *equal-def* = $\@ \{ \text{thm induct-equal-def} \}$

fun *dest-def* (*Const* (**const-name** $\langle \text{induct-equal} \rangle$, -) \$ *t* \$ *u*) = *SOME* (*t*, *u*)

| *dest-def* - = *NONE*

fun *trivial-tac ctxt* = *match-tac ctxt* $\@ \{ \text{thms induct-trueI} \}$

$)$

\rangle

ML-file $\langle \sim \sim / \text{src} / \text{Tools} / \text{induction.ML} \rangle$

declaration \langle

fn $- \implies$ *Induct.map-simpset* (*fn* *ss* \implies *ss*

addsimprocs

[*Simplifier.make-simproc* **context** *swap-induct-false*

{*lhss* = [**term** $\langle \text{induct-false} \implies \text{PROP } P \implies \text{PROP } Q \rangle$],

proc = *fn* $- \implies$ *fn* $- \implies$ *fn* *ct* \implies

(*case* *Thm.term-of ct* of

- \$ (*P as* - \$ **Const**- $\langle \text{induct-false} \rangle$) \$ (- \$ *Q* \$ -) \implies

if *P* $\langle \rangle$ *Q* then *SOME* *Drule.swap-prems-eq* else *NONE*

| $- \implies$ *NONE*)}],

Simplifier.make-simproc **context** *induct-equal-conj-curry*

{*lhss* = [**term** $\langle \text{induct-conj } P \ Q \implies \text{PROP } R \rangle$],

proc = *fn* $- \implies$ *fn* $- \implies$ *fn* *ct* \implies

```

(case Thm.term-of ct of
- $ (- $ P) $ - =>
  let
    fun is-conj Const-⟨induct-conj for P Q⟩ =
      is-conj P andalso is-conj Q
      | is-conj Const-⟨induct-equal - for - -⟩ = true
      | is-conj Const-⟨induct-true⟩ = true
      | is-conj Const-⟨induct-false⟩ = true
      | is-conj - = false
    in if is-conj P then SOME @{thm induct-conj-curry} else NONE end
  | - => NONE))]]
|> Simplifier.set-mksimps (fn ctxt =>
  Simpdata.mksimps Simpdata.mksimps-pairs ctxt #>
  map (rewrite-rule ctxt (map Thm.symmetric @{thms induct-rulify-fallback}))))))

```

Pre-simplification of induction and cases rules

lemma [induct-simp]: $(\bigwedge x. \text{induct-equal } x \ t \implies \text{PROP } P \ x) \equiv \text{PROP } P \ t$

unfolding induct-equal-def

proof

assume $r: \bigwedge x. x = t \implies \text{PROP } P \ x$

show $\text{PROP } P \ t$ **by** (rule r [OF refl])

next

fix x

assume $\text{PROP } P \ t \ x = t$

then show $\text{PROP } P \ x$ **by** simp

qed

lemma [induct-simp]: $(\bigwedge x. \text{induct-equal } t \ x \implies \text{PROP } P \ x) \equiv \text{PROP } P \ t$

unfolding induct-equal-def

proof

assume $r: \bigwedge x. t = x \implies \text{PROP } P \ x$

show $\text{PROP } P \ t$ **by** (rule r [OF refl])

next

fix x

assume $\text{PROP } P \ t \ t = x$

then show $\text{PROP } P \ x$ **by** simp

qed

lemma [induct-simp]: $(\text{induct-false} \implies P) \equiv \text{Trueprop induct-true}$

unfolding induct-false-def induct-true-def

by (iprover intro: equal-intr-rule)

lemma [induct-simp]: $(\text{induct-true} \implies \text{PROP } P) \equiv \text{PROP } P$

unfolding induct-true-def

proof

assume $\text{True} \implies \text{PROP } P$

then show $\text{PROP } P$ **using** TrueI .

next


```

assume PROP P
then show PROP P .
qed

```

```

lemma [induct-simp]: (PROP P  $\implies$  induct-true)  $\equiv$  Trueprop induct-true
unfolding induct-true-def
by (iprover intro: equal-intr-rule)

```

```

lemma [induct-simp]: ( $\bigwedge x::'a::\{\}$ . induct-true)  $\equiv$  Trueprop induct-true
unfolding induct-true-def
by (iprover intro: equal-intr-rule)

```

```

lemma [induct-simp]: induct-implies induct-true P  $\equiv$  P
by (simp add: induct-implies-def induct-true-def)

```

```

lemma [induct-simp]:  $x = x \longleftrightarrow$  True
by (rule simp-thms)

```

end

ML-file $\langle \sim\sim / \text{src} / \text{Tools} / \text{induct-tacs.ML} \rangle$

2.3.6 Coherent logic

ML-file $\langle \sim\sim / \text{src} / \text{Tools} / \text{coherent.ML} \rangle$

```

ML  $\langle$ 
  structure Coherent = Coherent
  (
    val atomize-elimL = @{thm atomize-elimL};
    val atomize-exL = @{thm atomize-exL};
    val atomize-conjL = @{thm atomize-conjL};
    val atomize-disjL = @{thm atomize-disjL};
    val operator-names = [const-name  $\langle$ HOL.disj $\rangle$ , const-name  $\langle$ HOL.conj $\rangle$ , const-name  $\langle$ Ex $\rangle$ ];
  );
 $\rangle$ 

```

2.3.7 Reorienting equalities

```

ML  $\langle$ 
  signature REORIENT-PROC =
  sig
    val add : (term  $\rightarrow$  bool)  $\rightarrow$  theory  $\rightarrow$  theory
    val proc : Proof.context  $\rightarrow$  cterm  $\rightarrow$  thm option
  end;

```

```

structure Reorient-Proc : REORIENT-PROC =
struct
  structure Data = Theory-Data
  (
    type T = ((term  $\rightarrow$  bool) * stamp) list;
  )

```

```

    val empty = [];
    fun merge data : T = Library.merge (eq-snd (op =)) data;
  );
  fun add m = Data.map (cons (m, stamp ()));
  fun matches thy t = exists (fn (m, -) => m t) (Data.get thy);

  val meta-reorient = @{thm eq-commute [THEN eq-reflection]};
  fun proc ctxt ct =
    let
      val thy = Proof-Context.theory-of ctxt;
    in
      case Thm.term-of ct of
        (- $ t $ u) => if matches thy u then NONE else SOME meta-reorient
      | - => NONE
    end;
  end;
  >

```

2.4 Other simple lemmas and lemma duplicates

lemma *eq-iff-swap*: $(x = y \longleftrightarrow P) \Longrightarrow (y = x \longleftrightarrow P)$
by *blast*

lemma *all-cong1*: $(\bigwedge x. P x = P' x) \Longrightarrow (\forall x. P x) = (\forall x. P' x)$
by *auto*

lemma *ex-cong1*: $(\bigwedge x. P x = P' x) \Longrightarrow (\exists x. P x) = (\exists x. P' x)$
by *auto*

lemma *all-cong*: $(\bigwedge x. Q x \Longrightarrow P x = P' x) \Longrightarrow (\forall x. Q x \longrightarrow P x) = (\forall x. Q x \longrightarrow P' x)$
by *auto*

lemma *ex-cong*: $(\bigwedge x. Q x \Longrightarrow P x = P' x) \Longrightarrow (\exists x. Q x \wedge P x) = (\exists x. Q x \wedge P' x)$
by *auto*

lemma *ex1-eq* [*iff*]: $\exists!x. x = t \iff \exists!x. t = x$
by *blast+*

lemma *choice-eq*: $(\forall x. \exists!y. P x y) = (\exists!f. \forall x. P x (f x))$ (**is** *?lhs = ?rhs*)

proof (*intro iffI allI*)

assume *L*: *?lhs*

then have \S : $\forall x. P x (THE y. P x y)$

by (*best intro: theI'*)

show *?rhs*

by (*rule exII*) (*use L § in <fast+>*)

next

fix *x*

```

assume R: ?rhs
then obtain f where f:  $\forall x. P\ x\ (f\ x)$  and f1:  $\bigwedge y. (\forall x. P\ x\ (y\ x)) \implies y = f$ 
  by (blast elim: ex1E)
show  $\exists! y. P\ x\ y$ 
proof (rule ex1I)
  show  $P\ x\ (f\ x)$ 
    using f by blast
  show  $y = f\ x$  if  $P\ x\ y$  for y
  proof –
    have  $P\ z\ (if\ z = x\ then\ y\ else\ f\ z)$  for z
      using f that by (auto split: if-split)
    with f1 [of  $\lambda z. if\ z = x\ then\ y\ else\ f\ z$ ] f
    show ?thesis
      by (auto simp add: split: if-split-asm dest: fun-cong)
  qed
qed
qed

```

lemmas *eq-sym-conv = eq-commute*

lemma *nnf-simps*:

```

 $(\neg (P \wedge Q)) = (\neg P \vee \neg Q)$ 
 $(\neg (P \vee Q)) = (\neg P \wedge \neg Q)$ 
 $(P \longrightarrow Q) = (\neg P \vee Q)$ 
 $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))$ 
 $(\neg (P = Q)) = ((P \wedge \neg Q) \vee (\neg P \wedge Q))$ 
 $(\neg \neg P) = P$ 
by blast+

```

2.5 Basic ML bindings

ML \langle

```

val FalseE = @{thm FalseE}
val Let-def = @{thm Let-def}
val TrueI = @{thm TrueI}
val allE = @{thm allE}
val allI = @{thm allI}
val all-dupE = @{thm all-dupE}
val arg-cong = @{thm arg-cong}
val box-equals = @{thm box-equals}
val ccontr = @{thm ccontr}
val classical = @{thm classical}
val conjE = @{thm conjE}
val conjI = @{thm conjI}
val conjunct1 = @{thm conjunct1}
val conjunct2 = @{thm conjunct2}
val disjCI = @{thm disjCI}
val disjE = @{thm disjE}
val disjI1 = @{thm disjI1}

```

```

val disjI2 = @{thm disjI2}
val eq-reflection = @{thm eq-reflection}
val ex1E = @{thm ex1E}
val ex1I = @{thm ex1I}
val ex1-implies-ex = @{thm ex1-implies-ex}
val exE = @{thm exE}
val exI = @{thm exI}
val excluded-middle = @{thm excluded-middle}
val ext = @{thm ext}
val fun-cong = @{thm fun-cong}
val iffD1 = @{thm iffD1}
val iffD2 = @{thm iffD2}
val iffI = @{thm iffI}
val impE = @{thm impE}
val impI = @{thm impI}
val meta-eq-to-obj-eq = @{thm meta-eq-to-obj-eq}
val mp = @{thm mp}
val notE = @{thm notE}
val notI = @{thm notI}
val not-all = @{thm not-all}
val not-ex = @{thm not-ex}
val not-iff = @{thm not-iff}
val not-not = @{thm not-not}
val not-sym = @{thm not-sym}
val refl = @{thm refl}
val rev-mp = @{thm rev-mp}
val spec = @{thm spec}
val ssubst = @{thm ssubst}
val subst = @{thm subst}
val sym = @{thm sym}
val trans = @{thm trans}
>

```

locale *cnf*

begin

lemma *clause2raw-notE*: $\llbracket P; \neg P \rrbracket \Longrightarrow \text{False}$ **by** *auto*

lemma *clause2raw-not-disj*: $\llbracket \neg P; \neg Q \rrbracket \Longrightarrow \neg (P \vee Q)$ **by** *auto*

lemma *clause2raw-not-not*: $P \Longrightarrow \neg \neg P$ **by** *auto*

lemma *iff-refl*: $(P::\text{bool}) = P$ **by** *auto*

lemma *iff-trans*: $\llbracket (P::\text{bool}) = Q; Q = R \rrbracket \Longrightarrow P = R$ **by** *auto*

lemma *conj-cong*: $\llbracket P = P'; Q = Q' \rrbracket \Longrightarrow (P \wedge Q) = (P' \wedge Q')$ **by** *auto*

lemma *disj-cong*: $\llbracket P = P'; Q = Q' \rrbracket \Longrightarrow (P \vee Q) = (P' \vee Q')$ **by** *auto*

lemma *make-nnf-imp*: $\llbracket (\neg P) = P'; Q = Q' \rrbracket \Longrightarrow (P \longrightarrow Q) = (P' \vee Q')$ **by** *auto*

lemma *make-nnf-iff*: $\llbracket P = P'; (\neg P) = NP; Q = Q'; (\neg Q) = NQ \rrbracket \Longrightarrow (P = Q) = ((P' \vee NQ) \wedge (NP \vee Q'))$ **by** *auto*

lemma *make-nnf-not-false*: $(\neg \text{False}) = \text{True}$ **by** *auto*
lemma *make-nnf-not-true*: $(\neg \text{True}) = \text{False}$ **by** *auto*
lemma *make-nnf-not-conj*: $[[(\neg P) = P'; (\neg Q) = Q']] \implies (\neg(P \wedge Q)) = (P' \vee Q')$ **by** *auto*
lemma *make-nnf-not-disj*: $[[(\neg P) = P'; (\neg Q) = Q']] \implies (\neg(P \vee Q)) = (P' \wedge Q')$ **by** *auto*
lemma *make-nnf-not-imp*: $[[P = P'; (\neg Q) = Q']] \implies (\neg(P \longrightarrow Q)) = (P' \wedge Q')$ **by** *auto*
lemma *make-nnf-not-iff*: $[[P = P'; (\neg P) = NP; Q = Q'; (\neg Q) = NQ]] \implies (\neg(P = Q)) = ((P' \vee Q') \wedge (NP \vee NQ))$ **by** *auto*
lemma *make-nnf-not-not*: $P = P' \implies (\neg\neg P) = P'$ **by** *auto*

lemma *simp-TF-conj-True-l*: $[[P = \text{True}; Q = Q']] \implies (P \wedge Q) = Q'$ **by** *auto*
lemma *simp-TF-conj-True-r*: $[[P = P'; Q = \text{True}]] \implies (P \wedge Q) = P'$ **by** *auto*
lemma *simp-TF-conj-False-l*: $P = \text{False} \implies (P \wedge Q) = \text{False}$ **by** *auto*
lemma *simp-TF-conj-False-r*: $Q = \text{False} \implies (P \wedge Q) = \text{False}$ **by** *auto*
lemma *simp-TF-disj-True-l*: $P = \text{True} \implies (P \vee Q) = \text{True}$ **by** *auto*
lemma *simp-TF-disj-True-r*: $Q = \text{True} \implies (P \vee Q) = \text{True}$ **by** *auto*
lemma *simp-TF-disj-False-l*: $[[P = \text{False}; Q = Q']] \implies (P \vee Q) = Q'$ **by** *auto*
lemma *simp-TF-disj-False-r*: $[[P = P'; Q = \text{False}]] \implies (P \vee Q) = P'$ **by** *auto*

lemma *make-cnfn-disj-conj-l*: $[[(P \vee R) = PR; (Q \vee R) = QR]] \implies ((P \wedge Q) \vee R) = (PR \wedge QR)$ **by** *auto*
lemma *make-cnfn-disj-conj-r*: $[[(P \vee Q) = PQ; (P \vee R) = PR]] \implies (P \vee (Q \wedge R)) = (PQ \wedge PR)$ **by** *auto*

lemma *make-cnfn-disj-ex-l*: $((\exists (x::\text{bool}). P x) \vee Q) = (\exists x. P x \vee Q)$ **by** *auto*
lemma *make-cnfn-disj-ex-r*: $(P \vee (\exists (x::\text{bool}). Q x)) = (\exists x. P \vee Q x)$ **by** *auto*
lemma *make-cnfn-newlit*: $(P \vee Q) = (\exists x. (P \vee x) \wedge (Q \vee \neg x))$ **by** *auto*
lemma *make-cnfn-ex-cong*: $(\forall (x::\text{bool}). P x = Q x) \implies (\exists x. P x) = (\exists x. Q x)$ **by** *auto*

lemma *weakening-thm*: $[[P; Q]] \implies Q$ **by** *auto*

lemma *cnftac-eq-imp*: $[[P = Q; P]] \implies Q$ **by** *auto*

end

ML-file $\langle \text{Tools}/\text{cnf}.ML \rangle$

3 NO-MATCH simproc

The simplification procedure can be used to avoid simplification of terms of a certain form.

definition *NO-MATCH* :: $'a \Rightarrow 'b \Rightarrow \text{bool}$
where *NO-MATCH* *pat val* $\equiv \text{True}$

lemma *NO-MATCH-cong*[*cong*]: *NO-MATCH pat val* = *NO-MATCH pat val*

```

by (rule refl)

declare [[coercion-args NO-MATCH - -]]

simproc-setup NO-MATCH (NO-MATCH pat val) = ⟨K (fn ctxt => fn ct =>
  let
    val thy = Proof-Context.theory-of ctxt
    val dest-binop = Term.dest-comb #> apfst (Term.dest-comb #> snd)
    val m = Pattern.matches thy (dest-binop (Thm.term-of ct))
    in if m then NONE else SOME @{thm NO-MATCH-def} end)
  ⟩

```

This setup ensures that a rewrite rule of the form *NO-MATCH pat val* $\implies t$ is only applied, if the pattern *pat* does not match the value *val*.

Tagging a premise of a simp rule with ASSUMPTION forces the simplifier not to simplify the argument and to solve it by an assumption.

```

definition ASSUMPTION :: bool  $\implies$  bool
  where ASSUMPTION A  $\equiv$  A

```

```

lemma ASSUMPTION-cong[cong]: ASSUMPTION A = ASSUMPTION A
  by (rule refl)

```

```

lemma ASSUMPTION-I: A  $\implies$  ASSUMPTION A
  by (simp add: ASSUMPTION-def)

```

```

lemma ASSUMPTION-D: ASSUMPTION A  $\implies$  A
  by (simp add: ASSUMPTION-def)

```

```

setup ⟨
  let
    val asm-sol = mk-solver ASSUMPTION (fn ctxt =>
      resolve-tac ctxt [@{thm ASSUMPTION-I}] THEN'
      resolve-tac ctxt (Simplifier.premis-of ctxt))
    in
      map-theory-simpset (fn ctxt => Simplifier.addSolver (ctxt,asm-sol))
    end
  ⟩

```

3.1 Code generator setup

3.1.1 Generic code generator preprocessor setup

```

lemma conj-left-cong: P  $\longleftrightarrow$  Q  $\implies$  P  $\wedge$  R  $\longleftrightarrow$  Q  $\wedge$  R
  by (fact arg-cong)

```

```

lemma disj-left-cong: P  $\longleftrightarrow$  Q  $\implies$  P  $\vee$  R  $\longleftrightarrow$  Q  $\vee$  R
  by (fact arg-cong)

```

```

setup ⟨

```

```

Code-Preproc.map-pre (put-simpset HOL-basic-ss) #>
Code-Preproc.map-post (put-simpset HOL-basic-ss) #>
Code-Simp.map-ss (put-simpset HOL-basic-ss #>
Simplifier.add-cong @ {thm conj-left-cong} #>
Simplifier.add-cong @ {thm disj-left-cong})
>

```

3.1.2 Equality

```

class equal =
  fixes equal :: 'a ⇒ 'a ⇒ bool
  assumes equal-eq: equal x y ⟷ x = y
begin

lemma equal: equal = (=)
  by (rule ext equal-eq)+

lemma equal-refl: equal x x ⟷ True
  unfolding equal by (rule iffI TrueI refl)+

lemma eq-equal: (=) ≡ equal
  by (rule eq-reflection) (rule ext, rule ext, rule sym, rule equal-eq)

end

declare eq-equal [symmetric, code-post]
declare eq-equal [code]

setup <
  Code-Preproc.map-pre (fn ctxt =>
    ctxt addsimprocs
      [Simplifier.make-simproc context equal
        {lhs = [term <HOL.eq>],
          proc = fn - => fn - => fn ct =>
            (case Thm.term-of ct of
              Const (-, Type (type-name <fun>, [Type -, -])) => SOME @ {thm eq-equal}
              | - => NONE)}}]
  >

```

3.1.3 Generic code generator foundation

Datatype *bool*

code-datatype *True False*

```

lemma [code]:
  shows False ∧ P ⟷ False
  and True ∧ P ⟷ P
  and P ∧ False ⟷ False
  and P ∧ True ⟷ P

```

by *simp-all*

```
lemma [code]:
  shows  $False \vee P \longleftrightarrow P$ 
    and  $True \vee P \longleftrightarrow True$ 
    and  $P \vee False \longleftrightarrow P$ 
    and  $P \vee True \longleftrightarrow True$ 
  by simp-all
```

```
lemma [code]:
  shows  $(False \longrightarrow P) \longleftrightarrow True$ 
    and  $(True \longrightarrow P) \longleftrightarrow P$ 
    and  $(P \longrightarrow False) \longleftrightarrow \neg P$ 
    and  $(P \longrightarrow True) \longleftrightarrow True$ 
  by simp-all
```

More about *prop*

```
lemma [code nbe]:
  shows  $(True \Longrightarrow PROP Q) \equiv PROP Q$ 
    and  $(PROP Q \Longrightarrow True) \equiv Trueprop True$ 
    and  $(P \Longrightarrow R) \equiv Trueprop (P \longrightarrow R)$ 
  by (auto intro!: equal-intr-rule)
```

```
lemma Trueprop-code [code]:  $Trueprop True \equiv Code-Generator.holds$ 
  by (auto intro!: equal-intr-rule holds)
```

```
declare Trueprop-code [symmetric, code-post]
```

Equality

```
declare simp-thms(6) [code nbe]
```

```
instantiation itself :: (type) equal
begin
```

```
definition equal-itself :: 'a itself  $\Rightarrow$  'a itself  $\Rightarrow$  bool
  where equal-itself x y  $\longleftrightarrow x = y$ 
```

```
instance
  by standard (fact equal-itself-def)
```

end

```
lemma equal-itself-code [code]:  $equal\ TYPE('a)\ TYPE('a) \longleftrightarrow True$ 
  by (simp add: equal)
```

```
setup  $\langle Sign.add-const-constraint\ (\mathbf{const-name}\ \langle equal \rangle,\ SOME\ \mathbf{typ}\ \langle 'a::type \Rightarrow 'a \Rightarrow bool \rangle) \rangle$ 
```

```
lemma equal-alias-cert:  $OFCLASS('a,\ equal-class) \equiv (((=) :: 'a \Rightarrow 'a \Rightarrow bool) \equiv$ 
```



```

equal)
  (is ?ofclass  $\equiv$  ?equal)
proof
  assume PROP ?ofclass
  show PROP ?equal
  by (tactic  $\langle$ ALLGOALS (resolve-tac context [Thm.unconstrainT @{thm eq-equal}]) $\rangle$ )
    (fact  $\langle$ PROP ?ofclass $\rangle$ )
next
  assume PROP ?equal
  show PROP ?ofclass proof
  qed (simp add:  $\langle$ PROP ?equal $\rangle$ )
qed

```

```

setup  $\langle$ Sign.add-const-constraint (const-name  $\langle$ equal $\rangle$ , SOME typ  $\langle$ 'a::equal  $\Rightarrow$  'a
 $\Rightarrow$  bool $\rangle$ ) $\rangle$ 

```

```

setup  $\langle$ Nbe.add-const-alias @{thm equal-alias-cert} $\rangle$ 

```

Cases

```

lemma Let-case-cert:
  assumes CASE  $\equiv$  ( $\lambda x$ . Let x f)
  shows CASE x  $\equiv$  f x
  using assms by simp-all

```

```

setup  $\langle$ 
  Code.declare-case-global @{thm Let-case-cert} #>
  Code.declare-undefined-global const-name  $\langle$ undefined $\rangle$ 
 $\rangle$ 

```

```

declare [[code abort: undefined]]

```

3.1.4 Generic code generator target languages

```

type bool

```

```

code-printing

```

```

  type-constructor bool  $\rightarrow$ 
    (SML) bool and (OCaml) bool and (Haskell) Bool and (Scala) Boolean
| constant True  $\rightarrow$ 
    (SML) true and (OCaml) true and (Haskell) True and (Scala) true
| constant False  $\rightarrow$ 
    (SML) false and (OCaml) false and (Haskell) False and (Scala) false

```

```

code-reserved SML

```

```

  bool true false

```

```

code-reserved OCaml

```

```

  bool

```

```

code-reserved Scala

```

Boolean

code-printing

```

constant Not  $\rightarrow$ 
  (SML) not and (OCaml) not and (Haskell) not and (Scala) ! -
| constant HOL.conj  $\rightarrow$ 
  (SML) infixl 1 andalso and (OCaml) infixl 3 && and (Haskell) infixr 3 &&
and (Scala) infixl 3 &&
| constant HOL.disj  $\rightarrow$ 
  (SML) infixl 0 orelse and (OCaml) infixl 2 || and (Haskell) infixl 2 || and
(Scala) infixl 1 ||
| constant HOL.implies  $\rightarrow$ 
  (SML) !(if (-) then (-) else true)
  and (OCaml) !(if (-) then (-) else true)
  and (Haskell) !(if (-) then (-) else True)
  and (Scala) !(if ((-)) then (-) else true)
| constant If  $\rightarrow$ 
  (SML) !(if (-) then (-) else (-))
  and (OCaml) !(if (-) then (-) else (-))
  and (Haskell) !(if (-) then (-) else (-))
  and (Scala) !(if ((-)) then (-) else (-))

```

code-reserved SML

not

code-reserved OCaml

not

code-identifier

code-module *Pure* \rightarrow

(*SML*) *HOL* **and** (*OCaml*) *HOL* **and** (*Haskell*) *HOL* **and** (*Scala*) *HOL*

Using built-in Haskell equality.

code-printing

```

type-class equal  $\rightarrow$  (Haskell) Eq
| constant HOL.equal  $\rightarrow$  (Haskell) infix 4 ==
| constant HOL.eq  $\rightarrow$  (Haskell) infix 4 ==

```

undefined

code-printing

```

constant undefined  $\rightarrow$ 
  (SML) !(raise/ Fail/ undefined)
  and (OCaml) failwith/ undefined
  and (Haskell) error/ undefined
  and (Scala) !sys.error(undefined)

```

3.1.5 Evaluation and normalization by evaluation

method-setup *eval* = \langle

```

let
  fun eval-tac ctxt =
    let val conv = Code-Runtime.dynamic-holds-conv
    in
      CONVERSION (Conv.params-conv ~ 1 (Conv.concl-conv ~ 1 o conv) ctxt)
    THEN'
      resolve-tac ctxt [TrueI]
    end
  in
    Scan.succeed (SIMPLE-METHOD' o eval-tac)
  end
› solve goal by evaluation

method-setup normalization = ‹
  Scan.succeed (fn ctxt =>
    SIMPLE-METHOD'
      (CHANGED-PROP o
        (CONVERSION (Nbe.dynamic-conv ctxt)
          THEN-ALL-NEW (TRY o resolve-tac ctxt [TrueI])))
  )
› solve goal by normalization

```

3.2 Counterexample Search Units

3.2.1 Quickcheck

quickcheck-params [size = 5, iterations = 50]

3.2.2 Nitpick setup

named-theorems *nitpick-unfold* alternative definitions of constants as needed by *Nitpick*

and *nitpick-simp* equational specification of constants as needed by *Nitpick*
and *nitpick-psimp* partial equational specification of constants as needed by *Nitpick*
and *nitpick-choice-spec* choice specification of constants as needed by *Nitpick*

declare *if-bool-eq-conj* [*nitpick-unfold*, *no-atp*]
and *if-bool-eq-disj* [*no-atp*]

3.3 Preprocessing for the predicate compiler

named-theorems *code-pred-def* alternative definitions of constants for the *Predicate Compiler*

and *code-pred-inline* inlining definitions for the *Predicate Compiler*
and *code-pred-simp* simplification rules for the optimisations in the *Predicate Compiler*

3.4 Legacy tactics and ML bindings

ML ‹
 (* combination of (spec RS spec RS ...(j times) ... spec RS mp) *)

```

local
  fun wrong-prem (Const (const-name ⟨All⟩, -) $ Abs (-, -, t)) = wrong-prem t
    | wrong-prem (Bound -) = true
    | wrong-prem - = false;
  val filter-right = filter (not o wrong-prem o HOLogic.dest-Trueprop o hd o
Thm.premsof);
  fun simp i = funpow i (fn m => filter-right ([spec] RL m)) [mp];
in
  fun simp-tac ctxt j = EVERY' [dresolve-tac ctxt (simp j), assume-tac ctxt];
end;

local
  val nnf-ss =
    simpset-of (put-simpset HOL-basic-ss context addsimps @{thms simp-thms
nnf-simps});
in
  fun nnf-conv ctxt = Simplifier.rewrite (put-simpset nnf-ss ctxt);
end
>

```

hide-const (open) eq equal

end

4 Abstract orderings

```

theory Orderings
imports HOL
keywords print-orders :: diag
begin

```

4.1 Abstract ordering

```

locale partial-preordering =
  fixes less-eq :: ⟨'a ⇒ 'a ⇒ bool⟩ (infix <≤> 50)
  assumes refl: ⟨a ≤ a⟩ — not iff: makes problems due to multiple (dual) inter-
pretations
  and trans: ⟨a ≤ b ⇒ b ≤ c ⇒ a ≤ c⟩

locale preordering = partial-preordering +
  fixes less :: ⟨'a ⇒ 'a ⇒ bool⟩ (infix <<> 50)
  assumes strict-iff-not: ⟨a < b ⟷ a ≤ b ∧ ¬ b ≤ a⟩
begin

```

```

lemma strict-implies-order:
  ⟨a < b ⇒ a ≤ b⟩
by (simp add: strict-iff-not)

```

lemma irrefl: — not *iff*: makes problems due to multiple (dual) interpretations

$\langle \neg a < a \rangle$
by (*simp add: strict-iff-not*)

lemma *asym*:
 $\langle a < b \implies b < a \implies \text{False} \rangle$
by (*auto simp add: strict-iff-not*)

lemma *strict-trans1*:
 $\langle a \leq b \implies b < c \implies a < c \rangle$
by (*auto simp add: strict-iff-not intro: trans*)

lemma *strict-trans2*:
 $\langle a < b \implies b \leq c \implies a < c \rangle$
by (*auto simp add: strict-iff-not intro: trans*)

lemma *strict-trans*:
 $\langle a < b \implies b < c \implies a < c \rangle$
by (*auto intro: strict-trans1 strict-implies-order*)

end

lemma *preordering-strictI*: — Alternative introduction rule with bias towards strict order

fixes *less-eq* (**infix** $\langle \leq \rangle$ 50)
and *less* (**infix** $\langle < \rangle$ 50)
assumes *less-eq-less*: $\langle \bigwedge a b. a \leq b \longleftrightarrow a < b \vee a = b \rangle$
assumes *asym*: $\langle \bigwedge a b. a < b \implies \neg b < a \rangle$
assumes *irrefl*: $\langle \bigwedge a. \neg a < a \rangle$
assumes *trans*: $\langle \bigwedge a b c. a < b \implies b < c \implies a < c \rangle$
shows $\langle \text{preordering } (\leq) (\langle) \rangle$
proof
fix *a b*
show $\langle a < b \longleftrightarrow a \leq b \wedge \neg b \leq a \rangle$
by (*auto simp add: less-eq-less asym irrefl*)
next
fix *a*
show $\langle a \leq a \rangle$
by (*auto simp add: less-eq-less*)
next
fix *a b c*
assume $\langle a \leq b \rangle$ **and** $\langle b \leq c \rangle$ **then show** $\langle a \leq c \rangle$
by (*auto simp add: less-eq-less intro: trans*)
qed

lemma *preordering-dualI*:
fixes *less-eq* (**infix** $\langle \leq \rangle$ 50)
and *less* (**infix** $\langle < \rangle$ 50)
assumes $\langle \text{preordering } (\lambda a b. b \leq a) (\lambda a b. b < a) \rangle$
shows $\langle \text{preordering } (\leq) (\langle) \rangle$

proof –

from *assms* **interpret** *preordering* $\langle \lambda a b. b \leq a \rangle \langle \lambda a b. b < a \rangle$.

show *?thesis*

by *standard (auto simp: strict-iff-not refl intro: trans)*

qed

locale *ordering* = *partial-preordering* +

fixes *less* :: $\langle 'a \Rightarrow 'a \Rightarrow \text{bool} \rangle$ (**infix** $\langle < \rangle$ 50)

assumes *strict-iff-order*: $\langle a < b \longleftrightarrow a \leq b \wedge a \neq b \rangle$

assumes *antisym*: $\langle a \leq b \Longrightarrow b \leq a \Longrightarrow a = b \rangle$

begin

sublocale *preordering* $\langle (\leq) \rangle \langle (<) \rangle$

proof

show $\langle a < b \longleftrightarrow a \leq b \wedge \neg b \leq a \rangle$ **for** *a b*

by (*auto simp add: strict-iff-order intro: antisym*)

qed

lemma *strict-implies-not-eq*:

$\langle a < b \Longrightarrow a \neq b \rangle$

by (*simp add: strict-iff-order*)

lemma *not-eq-order-implies-strict*:

$\langle a \neq b \Longrightarrow a \leq b \Longrightarrow a < b \rangle$

by (*simp add: strict-iff-order*)

lemma *order-iff-strict*:

$\langle a \leq b \longleftrightarrow a < b \vee a = b \rangle$

by (*auto simp add: strict-iff-order refl*)

lemma *eq-iff*: $\langle a = b \longleftrightarrow a \leq b \wedge b \leq a \rangle$

by (*auto simp add: refl intro: antisym*)

end

lemma *ordering-strictI*: — Alternative introduction rule with bias towards strict order

fixes *less-eq* (**infix** $\langle \leq \rangle$ 50)

and *less* (**infix** $\langle < \rangle$ 50)

assumes *less-eq-less*: $\langle \bigwedge a b. a \leq b \longleftrightarrow a < b \vee a = b \rangle$

assumes *asym*: $\langle \bigwedge a b. a < b \Longrightarrow \neg b < a \rangle$

assumes *irrefl*: $\langle \bigwedge a. \neg a < a \rangle$

assumes *trans*: $\langle \bigwedge a b c. a < b \Longrightarrow b < c \Longrightarrow a < c \rangle$

shows $\langle \text{ordering } (\leq) (<) \rangle$

proof

fix *a b*

show $\langle a < b \longleftrightarrow a \leq b \wedge a \neq b \rangle$

by (*auto simp add: less-eq-less asym irrefl*)

next

```

fix a
  show  $\langle a \leq a \rangle$ 
    by (auto simp add: less-eq-less)
next
  fix a b c
    assume  $\langle a \leq b \rangle$  and  $\langle b \leq c \rangle$  then show  $\langle a \leq c \rangle$ 
      by (auto simp add: less-eq-less intro: trans)
next
  fix a b
    assume  $\langle a \leq b \rangle$  and  $\langle b \leq a \rangle$  then show  $\langle a = b \rangle$ 
      by (auto simp add: less-eq-less asym)
qed

lemma ordering-dualI:
  fixes less-eq (infix  $\langle \leq \rangle$  50)
    and less (infix  $\langle < \rangle$  50)
  assumes  $\langle \text{ordering } (\lambda a b. b \leq a) (\lambda a b. b < a) \rangle$ 
  shows  $\langle \text{ordering } (\leq) (<) \rangle$ 
proof –
  from assms interpret ordering  $\langle \lambda a b. b \leq a \rangle \langle \lambda a b. b < a \rangle$  .
  show ?thesis
    by standard (auto simp: strict-iff-order refl intro: antisym trans)
qed

locale ordering-top = ordering +
  fixes top ::  $\langle 'a \rangle$  ( $\langle \top \rangle$ )
  assumes extremum [simp]:  $\langle a \leq \top \rangle$ 
begin

lemma extremum-uniqueI:
   $\langle \top \leq a \implies a = \top \rangle$ 
  by (rule antisym) auto

lemma extremum-unique:
   $\langle \top \leq a \iff a = \top \rangle$ 
  by (auto intro: antisym)

lemma extremum-strict [simp]:
   $\langle \neg (\top < a) \rangle$ 
  using extremum [of a] by (auto simp add: order-iff-strict intro: asym irrefl)

lemma not-eq-extremum:
   $\langle a \neq \top \iff a < \top \rangle$ 
  by (auto simp add: order-iff-strict intro: not-eq-order-implies-strict extremum)

end

```

4.2 Syntactic orders

```

class ord =
  fixes less-eq :: 'a ⇒ 'a ⇒ bool
    and less :: 'a ⇒ 'a ⇒ bool
begin

notation
  less-eq ('(≤)') and
  less-eq ((-/ ≤ -) [51, 51] 50) and
  less ('(<)') and
  less ((-/ < -) [51, 51] 50)

abbreviation (input)
  greater-eq (infix ≥ 50)
  where x ≥ y ≡ y ≤ x

abbreviation (input)
  greater (infix > 50)
  where x > y ≡ y < x

notation (ASCII)
  less-eq ('(<=)') and
  less-eq ((-/ <= -) [51, 51] 50)

notation (input)
  greater-eq (infix >= 50)

end

```

4.3 Quasi orders

```

class preorder = ord +
  assumes less-le-not-le: x < y ↔ x ≤ y ∧ ¬ (y ≤ x)
  and order-refl [iff]: x ≤ x
  and order-trans: x ≤ y ⇒ y ≤ z ⇒ x ≤ z
begin

sublocale order: preordering less-eq less + dual-order: preordering greater-eq greater
proof -
  interpret preordering less-eq less
    by standard (auto intro: order-trans simp add: less-le-not-le)
  show ⟨preordering less-eq less⟩
    by (fact preordering-axioms)
  then show ⟨preordering greater-eq greater⟩
    by (rule preordering-dualI)
qed

```

Reflexivity.

lemma *eq-refl*: $x = y \implies x \leq y$

— This form is useful with the classical reasoner.

by (*erule ssubst*) (*rule order-refl*)

lemma *less-irrefl* [*iff*]: $\neg x < x$

by (*simp add: less-le-not-le*)

lemma *less-imp-le*: $x < y \implies x \leq y$

by (*simp add: less-le-not-le*)

Asymmetry.

lemma *less-not-sym*: $x < y \implies \neg (y < x)$

by (*simp add: less-le-not-le*)

lemma *less-asy*: $x < y \implies (\neg P \implies y < x) \implies P$

by (*drule less-not-sym, erule contrapos-np*) *simp*

Transitivity.

lemma *less-trans*: $x < y \implies y < z \implies x < z$

by (*auto simp add: less-le-not-le intro: order-trans*)

lemma *le-less-trans*: $x \leq y \implies y < z \implies x < z$

by (*auto simp add: less-le-not-le intro: order-trans*)

lemma *less-le-trans*: $x < y \implies y \leq z \implies x < z$

by (*auto simp add: less-le-not-le intro: order-trans*)

Useful for simplification, but too risky to include by default.

lemma *less-imp-not-less*: $x < y \implies (\neg y < x) \longleftrightarrow \text{True}$

by (*blast elim: less-asy*)

lemma *less-imp-triv*: $x < y \implies (y < x \longrightarrow P) \longleftrightarrow \text{True}$

by (*blast elim: less-asy*)

Transitivity rules for calculational reasoning

lemma *less-asy'*: $a < b \implies b < a \implies P$

by (*rule less-asy*)

Dual order

lemma *dual-preorder*:

$\langle \text{class.preorder } (\geq) (>) \rangle$

by *standard* (*auto simp add: less-le-not-le intro: order-trans*)

end

lemma *preordering-preorderI*:

$\langle \text{class.preorder } (\leq) (<) \rangle$ **if** $\langle \text{preordering } (\leq) (<) \rangle$

for *less-eq* (**infix** $\langle \leq \rangle$ 50) **and** *less* (**infix** $\langle < \rangle$ 50)

proof —

from *that interpret preordering* $\langle(\leq)\rangle \langle(<)\rangle$.
show *?thesis*
by *standard (auto simp add: strict-iff-not refl intro: trans)*
qed

4.4 Partial orders

class *order = preorder +*
assumes *order-antisym: $x \leq y \implies y \leq x \implies x = y$*
begin

lemma *less-le: $x < y \iff x \leq y \wedge x \neq y$*
by *(auto simp add: less-le-not-le intro: order-antisym)*

sublocale *order: ordering less-eq less + dual-order: ordering greater-eq greater*
proof –
interpret *ordering less-eq less*
by *standard (auto intro: order-antisym order-trans simp add: less-le)*
show *ordering less-eq less*
by *(fact ordering-axioms)*
then show *ordering greater-eq greater*
by *(rule ordering-dualI)*
qed

Reflexivity.

lemma *le-less: $x \leq y \iff x < y \vee x = y$*
– NOT suitable for iff, since it can cause PROOF FAILED.
by *(fact order.order-iff-strict)*

lemma *le-imp-less-or-eq: $x \leq y \implies x < y \vee x = y$*
by *(simp add: less-le)*

Useful for simplification, but too risky to include by default.

lemma *less-imp-not-eq: $x < y \implies (x = y) \iff False$*
by *auto*

lemma *less-imp-not-eq2: $x < y \implies (y = x) \iff False$*
by *auto*

Transitivity rules for calculational reasoning

lemma *neq-le-trans: $a \neq b \implies a \leq b \implies a < b$*
by *(fact order.not-eq-order-implies-strict)*

lemma *le-neq-trans: $a \leq b \implies a \neq b \implies a < b$*
by *(rule order.not-eq-order-implies-strict)*

Asymmetry.

lemma *order-eq-iff: $x = y \iff x \leq y \wedge y \leq x$*
by *(fact order.eq-iff)*

lemma *antisym-conv*: $y \leq x \implies x \leq y \longleftrightarrow x = y$
by (*simp add: order.eq-iff*)

lemma *less-imp-neq*: $x < y \implies x \neq y$
by (*fact order.strict-implies-not-eq*)

lemma *antisym-conv1*: $\neg x < y \implies x \leq y \longleftrightarrow x = y$
by (*simp add: local.le-less*)

lemma *antisym-conv2*: $x \leq y \implies \neg x < y \longleftrightarrow x = y$
by (*simp add: local.less-le*)

lemma *leD*: $y \leq x \implies \neg x < y$
by (*auto simp: less-le order.antisym*)

Least value operator

definition (*in ord*)
Least :: $('a \Rightarrow \text{bool}) \Rightarrow 'a$ (**binder** *LEAST* 10) **where**
Least $P = (\text{THE } x. P x \wedge (\forall y. P y \longrightarrow x \leq y))$

lemma *Least-equality*:
assumes $P x$
and $\bigwedge y. P y \implies x \leq y$
shows $\text{Least } P = x$
unfolding *Least-def* **by** (*rule the-equality*)
(blast intro: assms order.antisym)+

lemma *LeastI2-order*:
assumes $P x$
and $\bigwedge y. P y \implies x \leq y$
and $\bigwedge x. P x \implies \forall y. P y \longrightarrow x \leq y \implies Q x$
shows $Q (\text{Least } P)$
unfolding *Least-def* **by** (*rule theI2*)
(blast intro: assms order.antisym)+

lemma *Least-ex1*:
assumes $\exists! x. P x \wedge (\forall y. P y \longrightarrow x \leq y)$
shows $\text{Least } P = x$ **and** *Least1-le*: $P z \implies \text{Least } P \leq z$
using *theI[OF assms]*
unfolding *Least-def*
by *auto*

Greatest value operator

definition *Greatest* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a$ (**binder** *GREATEST* 10) **where**
Greatest $P = (\text{THE } x. P x \wedge (\forall y. P y \longrightarrow x \geq y))$

lemma *GreatestI2-order*:
 $\llbracket P x;$

$$\begin{aligned} & \bigwedge y. P y \implies x \geq y; \\ & \bigwedge x. [P x; \forall y. P y \longrightarrow x \geq y] \implies Q x] \\ & \implies Q \text{ (Greatest } P\text{)} \end{aligned}$$

unfolding *Greatest-def*
by (rule *theI2*) (blast intro: *order.antisym*)+

lemma *Greatest-equality*:
 $[P x; \bigwedge y. P y \implies x \geq y] \implies \text{Greatest } P = x$
unfolding *Greatest-def*
by (rule *the-equality*) (blast intro: *order.antisym*)+

end

lemma *ordering-orderI*:
fixes *less-eq* (**infix** \leq 50)
and *less* (**infix** $<$ 50)
assumes *ordering less-eq less*
shows *class.order less-eq less*
proof –
from *assms interpret ordering less-eq less* .
show *?thesis*
by *standard* (auto intro: *antisym trans simp add: refl strict-iff-order*)
qed

lemma *order-strictI*:
fixes *less* (**infix** $<$ 50)
and *less-eq* (**infix** \leq 50)
assumes $\bigwedge a b. a \leq b \longleftrightarrow a < b \vee a = b$
assumes $\bigwedge a b. a < b \implies \neg b < a$
assumes $\bigwedge a. \neg a < a$
assumes $\bigwedge a b c. a < b \implies b < c \implies a < c$
shows *class.order less-eq less*
by (rule *ordering-orderI*) (rule *ordering-strictI*, (fact *assms*))+

context *order*
begin

Dual order

lemma *dual-order*:
class.order (\geq) ($>$)
using *dual-order.ordering-axioms* **by** (rule *ordering-orderI*)

end

4.5 Linear (total) orders

class *linorder* = *order* +
assumes *linear*: $x \leq y \vee y \leq x$
begin

lemma *less-linear*: $x < y \vee x = y \vee y < x$
unfolding *less-le* **using** *less-le linear* **by** *blast*

lemma *le-less-linear*: $x \leq y \vee y < x$
by (*simp add: le-less less-linear*)

lemma *le-cases* [*case-names le ge*]:
 $(x \leq y \implies P) \implies (y \leq x \implies P) \implies P$
using *linear* **by** *blast*

lemma (**in** *linorder*) *le-cases3*:
 $\llbracket x \leq y; y \leq z \rrbracket \implies P; \llbracket y \leq x; x \leq z \rrbracket \implies P; \llbracket x \leq z; z \leq y \rrbracket \implies P;$
 $\llbracket z \leq y; y \leq x \rrbracket \implies P; \llbracket y \leq z; z \leq x \rrbracket \implies P; \llbracket z \leq x; x \leq y \rrbracket \implies P \implies P$
by (*blast intro: le-cases*)

lemma *linorder-cases* [*case-names less equal greater*]:
 $(x < y \implies P) \implies (x = y \implies P) \implies (y < x \implies P) \implies P$
using *less-linear* **by** *blast*

lemma *linorder-wlog*[*case-names le sym*]:
 $(\bigwedge a b. a \leq b \implies P a b) \implies (\bigwedge a b. P b a \implies P a b) \implies P a b$
by (*cases rule: le-cases[of a b]*) *blast+*

lemma *not-less*: $\neg x < y \longleftrightarrow y \leq x$
unfolding *less-le*
using *linear* **by** (*blast intro: order.antisym*)

lemma *not-less-iff-gr-or-eq*: $\neg(x < y) \longleftrightarrow (x > y \vee x = y)$
by (*auto simp add: not-less le-less*)

lemma *not-le*: $\neg x \leq y \longleftrightarrow y < x$
unfolding *less-le*
using *linear* **by** (*blast intro: order.antisym*)

lemma *neq-iff*: $x \neq y \longleftrightarrow x < y \vee y < x$
by (*cut-tac x = x and y = y in less-linear, auto*)

lemma *neqE*: $x \neq y \implies (x < y \implies R) \implies (y < x \implies R) \implies R$
by (*simp add: neq-iff*) *blast*

lemma *antisym-conv3*: $\neg y < x \implies \neg x < y \longleftrightarrow x = y$
by (*blast intro: order.antisym dest: not-less [THEN iffD1]*)

lemma *leI*: $\neg x < y \implies y \leq x$
unfolding *not-less* .

lemma *not-le-imp-less*: $\neg y \leq x \implies x < y$
unfolding *not-le* .

lemma *linorder-less-wlog*[*case-names less refl sym*]:
 $\llbracket \bigwedge a b. a < b \implies P a b; \bigwedge a. P a a; \bigwedge a b. P b a \implies P a b \rrbracket \implies P a b$
using *antisym-conv3* **by** *blast*

Dual order

lemma *dual-linorder*:
class.linorder (\geq) ($>$)
by (*rule class.linorder.intro*, *rule dual-order*) (*unfold-locales*, *rule linear*)

end

Alternative introduction rule with bias towards strict order

lemma *linorder-strictI*:
fixes *less-eq* (**infix** \leq 50)
and *less* (**infix** $<$ 50)
assumes *class.order less-eq less*
assumes *trichotomy*: $\bigwedge a b. a < b \vee a = b \vee b < a$
shows *class.linorder less-eq less*
proof –
interpret *order less-eq less*
by (*fact* \langle *class.order less-eq less* \rangle)
show *?thesis*
proof
fix *a b*
show $a \leq b \vee b \leq a$
using *trichotomy* **by** (*auto simp add: le-less*)
qed
qed

4.6 Reasoning tools setup

ML-file \langle ~/src/Provers/order-procedure.ML \rangle

ML-file \langle ~/src/Provers/order-tac.ML \rangle

ML \langle

structure Logic-Signature : LOGIC-SIGNATURE = struct

val mk-Trueprop = HOLogic.mk-Trueprop
val dest-Trueprop = HOLogic.dest-Trueprop
val Trueprop-conv = HOLogic.Trueprop-conv
val Not = HOLogic.Not
val conj = HOLogic.conj
val disj = HOLogic.disj

val notI = @{thm notI}
val ccontr = @{thm ccontr}
val conjI = @{thm conjI}
val conjE = @{thm conjE}
val disjE = @{thm disjE}

```

val not-not-conv = Conv.rewr-conv @ { thm eq-reflection [OF not-not] }
val de-Morgan-conj-conv = Conv.rewr-conv @ { thm eq-reflection [OF de-Morgan-conj] }
val de-Morgan-disj-conv = Conv.rewr-conv @ { thm eq-reflection [OF de-Morgan-disj] }
val conj-disj-distribL-conv = Conv.rewr-conv @ { thm eq-reflection [OF conj-disj-distribL] }
val conj-disj-distribR-conv = Conv.rewr-conv @ { thm eq-reflection [OF conj-disj-distribR] }
end

```

```

structure HOL-Base-Order-Tac = Base-Order-Tac(
  structure Logic-Sig = Logic-Signature;
  (* Exclude types with specialised solvers. *)
  val excluded-types = [HOLLogic.natT, HOLLogic.intT, HOLLogic.realT]
)

```

```

structure HOL-Order-Tac = Order-Tac(structure Base-Tac = HOL-Base-Order-Tac)

```

```

fun print-orders ctxt0 =
  let
    val ctxt = Config.put show-sorts true ctxt0
    val orders = HOL-Order-Tac.Data.get (Context.Proof ctxt)
    fun pretty-term t = Pretty.block
      [Pretty.quote (Syntax.pretty-term ctxt t), Pretty.brk 1,
       Pretty.str ::, Pretty.brk 1,
       Pretty.quote (Syntax.pretty-typ ctxt (type-of t)), Pretty.brk 1]
    fun pretty-order ({kind = kind, ops = ops, ...}, -) =
      Pretty.block ([Pretty.str (@{make-string} kind), Pretty.str :, Pretty.brk 1]
        @ map pretty-term ops)
  in
    Pretty.writeln (Pretty.big-list order structures: (map pretty-order orders))
  end

val - =
  Outer-Syntax.command command-keyword ⟨print-orders⟩
  print order structures available to order reasoner
  (Scan.succeed (Toplevel.keep (print-orders o Toplevel.context-of)))
⟩

```

```

method-setup order = ⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (HOL-Order-Tac.tac [] ctxt))
⟩ partial and linear order reasoner

```

The method *order* allows one to use the order tactic located in `../Provers/order_tac.ML` in a standalone fashion.

The tactic rearranges the goal to prove *False*, then retrieves order literals of partial and linear orders (i.e. $x = y$, $x \leq y$, $x < y$, and their negated versions) from the premises and finally tries to derive a contradiction. Its main use case is as a solver to *simp* (see below), where it e.g. solves premises of conditional rewrite rules.

The tactic has two configuration attributes that control its behaviour:

- *order-trace* toggles tracing for the solver.
- *order-split-limit* limits the number of order literals of the form $\neg x < y$ that are passed to the tactic. This is helpful since these literals lead to case splitting and thus exponential runtime. This only applies to partial orders.

We setup the solver for HOL with the structure `HOL_Order_Tac` here but the prover is agnostic to the object logic. It is possible to register orders with the prover using the functions `HOL_Order_Tac.declare_order` and `HOL_Order_Tac.declare_linorder`, which we do below for the type classes *order* and *linorder*. If possible, one should instantiate these type classes instead of registering new orders with the solver. One can also interpret the type class locales *order* and *linorder*. An example can be seen in `Library/Sublist.thy`, which contains e.g. the prefix order on lists.

The diagnostic command **print-orders** shows all orders known to the tactic in the current context.

Declarations to set up transitivity reasoner of partial and linear orders.

```
context order
begin
```

```
lemma nless-le:  $(\neg a < b) \longleftrightarrow (\neg a \leq b) \vee a = b$ 
using local.dual-order.order-iff-strict by blast
```

```
local-setup <
  HOL-Order-Tac.declare-order {
    ops = {eq = @{term <(=) :: 'a ⇒ 'a ⇒ bool>}, le = @{term <(≤)>}, lt = @{term
<(<)>}},
    thms = {trans = @{thm order-trans}, refl = @{thm order-refl}, eqD1 = @{thm
eq-refl},
            eqD2 = @{thm eq-refl[OF sym]}, antisym = @{thm order-antisym}, contr
= @{thm notE}},
    conv-thms = {less-le = @{thm eq-reflection[OF less-le]},
                 nless-le = @{thm eq-reflection[OF nless-le]}}
  }
>
```

```
end
```

```
context linorder
begin
```

```
lemma nle-le:  $(\neg a \leq b) \longleftrightarrow b \leq a \wedge b \neq a$ 
```


using *not-le less-le* **by** *simp*

local-setup <

```

  HOL-Order-Tac.declare-linorder {
    ops = {eq = @{term <(=) :: 'a ⇒ 'a ⇒ bool>}, le = @{term <(≤)>}, lt = @{term
    <(<)>}},
    thms = {trans = @{thm order-trans}, refl = @{thm order-refl}, eqD1 = @{thm
    eq-refl},
            eqD2 = @{thm eq-refl[OF sym]}, antisym = @{thm order-antisym}, contr
    = @{thm notE}},
    conv-thms = {less-le = @{thm eq-reflection[OF less-le]},
                 nless-le = @{thm eq-reflection[OF not-less]},
                 nle-le = @{thm eq-reflection[OF nle-le]}}
  }
>

```

end

setup <

```

  map-theory-simpset (fn ctxt0 => ctxt0 addSolver
  mk-solver partial and linear orders (fn ctxt => HOL-Order-Tac.tac (Simplifier.premis-of
  ctxt) ctxt))
>

```

ML <

local

```

  fun prp t thm = Thm.prop-of thm = t; (* FIXME proper aconv! ? *)
  in

```

```

  fun antisym-le-simproc ctxt ct =

```

```

    (case Thm.term-of ct of
     (le as Const (-, T)) $ r $ s =>
      (let
        val prems = Simplifier.premis-of ctxt;
        val less = Const (const-name <less>, T);
        val t = HOLogic.mk-Trueprop(le $ s $ r);
      in
        (case find-first (prp t) prems of
         NONE =>
          let val t = HOLogic.mk-Trueprop(HOLogic.Not $ (less $ r $ s)) in
            (case find-first (prp t) prems of
             NONE => NONE
             | SOME thm => SOME(mk-meta-eq(thm RS @{thm antisym-conv1})))
          end
         | SOME thm => SOME (mk-meta-eq (thm RS @{thm order-class.antisym-conv})))
        end handle THM - => NONE)
    | - => NONE);

```

```

  fun antisym-less-simproc ctxt ct =

```

```

(case Thm.term-of ct of
  NotC $ ((less as Const(-, T)) $ r $ s) =>
  (let
    val prems = Simplifier.prems-of ctxt;
    val le = Const (const-name <less-eq>, T);
    val t = HOLogic.mk-Trueprop(le $ r $ s);
  in
    (case find-first (prp t) prems of
      NONE =>
        let val t = HOLogic.mk-Trueprop (NotC $ (less $ s $ r)) in
          (case find-first (prp t) prems of
            NONE => NONE
          | SOME thm => SOME (mk-meta-eq(thm RS @ {thm linorder-class.antisym-conv3})))
        end
      | SOME thm => SOME (mk-meta-eq (thm RS @ {thm antisym-conv2})))
    end handle THM - => NONE)
  | - => NONE);
end;
>

```

simproc-setup *antisym-le* (($x::'a::order$) \leq y) = K *antisym-le-simproc*

simproc-setup *antisym-less* (\neg ($x::'a::linorder$) $<$ y) = K *antisym-less-simproc*

4.7 Bounded quantifiers

syntax (ASCII)

```

-All-less :: [idt, 'a, bool] => bool  (( $\exists$ ALL -<-./ -) [0, 0, 10] 10)
-Ex-less  :: [idt, 'a, bool] => bool  (( $\exists$ EX  -<-./ -) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool  (( $\exists$ ALL -<= -./ -) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool  (( $\exists$ EX  -<= -./ -) [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool  (( $\exists$ ALL ->-./ -) [0, 0, 10] 10)
-Ex-greater  :: [idt, 'a, bool] => bool  (( $\exists$ EX  ->-./ -) [0, 0, 10] 10)
-All-greater-eq :: [idt, 'a, bool] => bool  (( $\exists$ ALL ->= -./ -) [0, 0, 10] 10)
-Ex-greater-eq :: [idt, 'a, bool] => bool  (( $\exists$ EX  ->= -./ -) [0, 0, 10] 10)

-All-neq :: [idt, 'a, bool] => bool  (( $\exists$ ALL -~=-./ -) [0, 0, 10] 10)
-Ex-neq  :: [idt, 'a, bool] => bool  (( $\exists$ EX  -~=-./ -) [0, 0, 10] 10)

```

syntax

```

-All-less :: [idt, 'a, bool] => bool  (( $\exists$ V -<-./ -) [0, 0, 10] 10)
-Ex-less  :: [idt, 'a, bool] => bool  (( $\exists$ E  -<-./ -) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool  (( $\exists$ V -<= -./ -) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool  (( $\exists$ E  -<= -./ -) [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool  (( $\exists$ V ->-./ -) [0, 0, 10] 10)
-Ex-greater  :: [idt, 'a, bool] => bool  (( $\exists$ E  ->-./ -) [0, 0, 10] 10)
-All-greater-eq :: [idt, 'a, bool] => bool  (( $\exists$ V ->= -./ -) [0, 0, 10] 10)

```

-*Ex-greater-eq* :: [idt, 'a, bool] => bool (($\exists \geq$ -./ -) [0, 0, 10] 10)

-*All-neq* :: [idt, 'a, bool] => bool (($\exists \neq$ -./ -) [0, 0, 10] 10)

-*Ex-neq* :: [idt, 'a, bool] => bool (($\exists \neq$ -./ -) [0, 0, 10] 10)

syntax (input)

-*All-less* :: [idt, 'a, bool] => bool (($\exists ! <$ -./ -) [0, 0, 10] 10)

-*Ex-less* :: [idt, 'a, bool] => bool (($\exists ? <$ -./ -) [0, 0, 10] 10)

-*All-less-eq* :: [idt, 'a, bool] => bool (($\exists ! \leq$ -./ -) [0, 0, 10] 10)

-*Ex-less-eq* :: [idt, 'a, bool] => bool (($\exists ? \leq$ -./ -) [0, 0, 10] 10)

-*All-neq* :: [idt, 'a, bool] => bool (($\exists ! \neq$ -./ -) [0, 0, 10] 10)

-*Ex-neq* :: [idt, 'a, bool] => bool (($\exists ? \neq$ -./ -) [0, 0, 10] 10)

translations

$\forall x < y. P \rightarrow \forall x. x < y \rightarrow P$

$\exists x < y. P \rightarrow \exists x. x < y \wedge P$

$\forall x \leq y. P \rightarrow \forall x. x \leq y \rightarrow P$

$\exists x \leq y. P \rightarrow \exists x. x \leq y \wedge P$

$\forall x > y. P \rightarrow \forall x. x > y \rightarrow P$

$\exists x > y. P \rightarrow \exists x. x > y \wedge P$

$\forall x \geq y. P \rightarrow \forall x. x \geq y \rightarrow P$

$\exists x \geq y. P \rightarrow \exists x. x \geq y \wedge P$

$\forall x \neq y. P \rightarrow \forall x. x \neq y \rightarrow P$

$\exists x \neq y. P \rightarrow \exists x. x \neq y \wedge P$

print-translation <

let

val *All-binder* = *Mixfix.binder-name const-syntax* <*All*>;

val *Ex-binder* = *Mixfix.binder-name const-syntax* <*Ex*>;

val *impl* = *const-syntax* <*HOL.implies*>;

val *conj* = *const-syntax* <*HOL.conj*>;

val *less* = *const-syntax* <*less*>;

val *less-eq* = *const-syntax* <*less-eq*>;

val *trans* =

[((*All-binder*, *impl*, *less*),
 (*syntax-const* <*-All-less*>, *syntax-const* <*-All-greater*>)),
 ((*All-binder*, *impl*, *less-eq*),
 (*syntax-const* <*-All-less-eq*>, *syntax-const* <*-All-greater-eq*>)),
 ((*Ex-binder*, *conj*, *less*),
 (*syntax-const* <*-Ex-less*>, *syntax-const* <*-Ex-greater*>)),
 ((*Ex-binder*, *conj*, *less-eq*),
 (*syntax-const* <*-Ex-less-eq*>, *syntax-const* <*-Ex-greater-eq*>))];

fun *matches-bound* v t =

(case t of
 Const (*syntax-const* <*-bound*>, -) \$ Free (v', -) => v = v'
 | - => false);

fun *contains-var* v = *Term.exists-subterm* (fn Free (x, -) => x = v | - => false);

```

fun mk x c n P = Syntax.const c $ Syntax-Trans.mark-bound-body x $ n $ P;

fun tr' q = (q, fn - =>
  (fn [Const (syntax-const <-bound>, -) $ Free (v, T),
    Const (c, -) $ (Const (d, -) $ t $ u) $ P] =>
  (case AList.lookup (=) trans (q, c, d) of
    NONE => raise Match
  | SOME (l, g) =>
    if matches-bound v t andalso not (contains-var v u) then mk (v, T) l u P
    else if matches-bound v u andalso not (contains-var v t) then mk (v, T)
g t P
    else raise Match)
  | - => raise Match));
in [tr' All-binder, tr' Ex-binder] end
>

```

4.8 Transitivity reasoning

context *ord*
begin

lemma *ord-le-eq-trans*: $a \leq b \implies b = c \implies a \leq c$
by (*rule subst*)

lemma *ord-eq-le-trans*: $a = b \implies b \leq c \implies a \leq c$
by (*rule ssubst*)

lemma *ord-less-eq-trans*: $a < b \implies b = c \implies a < c$
by (*rule subst*)

lemma *ord-eq-less-trans*: $a = b \implies b < c \implies a < c$
by (*rule ssubst*)

end

lemma *order-less-subst2*: $(a::'a::order) < b \implies f b < (c::'c::order) \implies$
 $(!!x y. x < y \implies f x < f y) \implies f a < c$

proof –

assume $r: !!x y. x < y \implies f x < f y$
assume $a < b$ **hence** $f a < f b$ **by** (*rule r*)
also assume $f b < c$
finally (*less-trans*) **show** *?thesis* .

qed

lemma *order-less-subst1*: $(a::'a::order) < f b \implies (b::'b::order) < c \implies$
 $(!!x y. x < y \implies f x < f y) \implies a < f c$

proof –

assume $r: !!x y. x < y \implies f x < f y$
assume $a < f b$

also assume $b < c$ hence $f b < f c$ by (rule r)
 finally (less-trans) show ?thesis .
 qed

lemma order-le-less-subst2: $(a::'a::order) \leq b \implies f b < (c::'c::order) \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies f a < c$
 proof -
 assume $r: !!x y. x \leq y \implies f x \leq f y$
 assume $a \leq b$ hence $f a \leq f b$ by (rule r)
 also assume $f b < c$
 finally (le-less-trans) show ?thesis .
 qed

lemma order-le-less-subst1: $(a::'a::order) \leq f b \implies (b::'b::order) < c \implies$
 $(!!x y. x < y \implies f x < f y) \implies a < f c$
 proof -
 assume $r: !!x y. x < y \implies f x < f y$
 assume $a \leq f b$
 also assume $b < c$ hence $f b < f c$ by (rule r)
 finally (le-less-trans) show ?thesis .
 qed

lemma order-less-le-subst2: $(a::'a::order) < b \implies f b \leq (c::'c::order) \implies$
 $(!!x y. x < y \implies f x < f y) \implies f a < c$
 proof -
 assume $r: !!x y. x < y \implies f x < f y$
 assume $a < b$ hence $f a < f b$ by (rule r)
 also assume $f b \leq c$
 finally (less-le-trans) show ?thesis .
 qed

lemma order-less-le-subst1: $(a::'a::order) < f b \implies (b::'b::order) \leq c \implies$
 $(!!x y. x < y \implies f x < f y) \implies a < f c$
 proof -
 assume $r: !!x y. x < y \implies f x < f y$
 assume $a < f b$
 also assume $b \leq c$ hence $f b \leq f c$ by (rule r)
 finally (less-le-trans) show ?thesis .
 qed

lemma order-subst1: $(a::'a::order) \leq f b \implies (b::'b::order) \leq c \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies a \leq f c$
 proof -
 assume $r: !!x y. x \leq y \implies f x \leq f y$
 assume $a \leq f b$
 also assume $b \leq c$ hence $f b \leq f c$ by (rule r)
 finally (order-trans) show ?thesis .
 qed

lemma *order-subst2*: $(a::'a::order) \leq b \implies f b \leq (c::'c::order) \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies f a \leq c$

proof –

assume $r: !!x y. x \leq y \implies f x \leq f y$
 assume $a \leq b$ hence $f a \leq f b$ by (rule r)
 also assume $f b \leq c$
 finally (*order-trans*) show ?thesis .

qed

lemma *ord-le-eq-subst*: $a \leq b \implies f b = c \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies f a \leq c$

proof –

assume $r: !!x y. x \leq y \implies f x \leq f y$
 assume $a \leq b$ hence $f a \leq f b$ by (rule r)
 also assume $f b = c$
 finally (*ord-le-eq-trans*) show ?thesis .

qed

lemma *ord-eq-le-subst*: $a = f b \implies b \leq c \implies$
 $(!!x y. x \leq y \implies f x \leq f y) \implies a \leq f c$

proof –

assume $r: !!x y. x \leq y \implies f x \leq f y$
 assume $a = f b$
 also assume $b \leq c$ hence $f b \leq f c$ by (rule r)
 finally (*ord-eq-le-trans*) show ?thesis .

qed

lemma *ord-less-eq-subst*: $a < b \implies f b = c \implies$
 $(!!x y. x < y \implies f x < f y) \implies f a < c$

proof –

assume $r: !!x y. x < y \implies f x < f y$
 assume $a < b$ hence $f a < f b$ by (rule r)
 also assume $f b = c$
 finally (*ord-less-eq-trans*) show ?thesis .

qed

lemma *ord-eq-less-subst*: $a = f b \implies b < c \implies$
 $(!!x y. x < y \implies f x < f y) \implies a < f c$

proof –

assume $r: !!x y. x < y \implies f x < f y$
 assume $a = f b$
 also assume $b < c$ hence $f b < f c$ by (rule r)
 finally (*ord-eq-less-trans*) show ?thesis .

qed

Note that this list of rules is in reverse order of priorities.

lemmas [*trans*] =
order-less-subst2
order-less-subst1

order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp

lemmas (in *order*) [*trans*] =
neq-le-trans
le-neq-trans

lemmas (in *preorder*) [*trans*] =
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans

lemmas (in *order*) [*trans*] =
order.antisym

lemmas (in *ord*) [*trans*] =
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans

lemmas [*trans*] =
trans

lemmas *order-trans-rules* =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst

ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp
neq-le-trans
le-neq-trans
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans
order.antisym
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans
trans

These support proving chains of decreasing inequalities $a \geq b \geq c \dots$ in Isar proofs.

lemma *xt1* [*no-atp*]:

$a = b \implies b > c \implies a > c$
 $a > b \implies b = c \implies a > c$
 $a = b \implies b \geq c \implies a \geq c$
 $a \geq b \implies b = c \implies a \geq c$
 $(x::'a::order) \geq y \implies y \geq x \implies x = y$
 $(x::'a::order) \geq y \implies y \geq z \implies x \geq z$
 $(x::'a::order) > y \implies y \geq z \implies x > z$
 $(x::'a::order) \geq y \implies y > z \implies x > z$
 $(a::'a::order) > b \implies b > a \implies P$
 $(x::'a::order) > y \implies y > z \implies x > z$
 $(a::'a::order) \geq b \implies a \neq b \implies a > b$
 $(a::'a::order) \neq b \implies a \geq b \implies a > b$
 $a = f b \implies b > c \implies (\bigwedge x y. x > y \implies f x > f y) \implies a > f c$
 $a > b \implies f b = c \implies (\bigwedge x y. x > y \implies f x > f y) \implies f a > c$
 $a = f b \implies b \geq c \implies (\bigwedge x y. x \geq y \implies f x \geq f y) \implies a \geq f c$
 $a \geq b \implies f b = c \implies (\bigwedge x y. x \geq y \implies f x \geq f y) \implies f a \geq c$
by *auto*

lemma *xt2* [*no-atp*]:

assumes $(a::'a::order) \geq f b$
and $b \geq c$
and $\bigwedge x y. x \geq y \implies f x \geq f y$
shows $a \geq f c$
using *assms* **by** *force*

lemma *xt3* [*no-atp*]:

assumes $(a::'a::order) \geq b$
and $(f\ b::'b::order) \geq c$
and $\bigwedge x\ y. x \geq y \implies f\ x \geq f\ y$
shows $f\ a \geq c$
using *assms* **by** *force*

lemma *xt4* [*no-atp*]:
assumes $(a::'a::order) > f\ b$
and $(b::'b::order) \geq c$
and $\bigwedge x\ y. x \geq y \implies f\ x \geq f\ y$
shows $a > f\ c$
using *assms* **by** *force*

lemma *xt5* [*no-atp*]:
assumes $(a::'a::order) > b$
and $(f\ b::'b::order) \geq c$
and $\bigwedge x\ y. x > y \implies f\ x > f\ y$
shows $f\ a > c$
using *assms* **by** *force*

lemma *xt6* [*no-atp*]:
assumes $(a::'a::order) \geq f\ b$
and $b > c$
and $\bigwedge x\ y. x > y \implies f\ x > f\ y$
shows $a > f\ c$
using *assms* **by** *force*

lemma *xt7* [*no-atp*]:
assumes $(a::'a::order) \geq b$
and $(f\ b::'b::order) > c$
and $\bigwedge x\ y. x \geq y \implies f\ x \geq f\ y$
shows $f\ a > c$
using *assms* **by** *force*

lemma *xt8* [*no-atp*]:
assumes $(a::'a::order) > f\ b$
and $(b::'b::order) > c$
and $\bigwedge x\ y. x > y \implies f\ x > f\ y$
shows $a > f\ c$
using *assms* **by** *force*

lemma *xt9* [*no-atp*]:
assumes $(a::'a::order) > b$
and $(f\ b::'b::order) > c$
and $\bigwedge x\ y. x > y \implies f\ x > f\ y$
shows $f\ a > c$
using *assms* **by** *force*

lemmas *xtrans* = *xt1* *xt2* *xt3* *xt4* *xt5* *xt6* *xt7* *xt8* *xt9*

4.9 min and max – fundamental

definition (in *ord*) $\text{min} :: 'a \Rightarrow 'a \Rightarrow 'a$ where
 $\text{min } a \ b = (\text{if } a \leq b \text{ then } a \text{ else } b)$

definition (in *ord*) $\text{max} :: 'a \Rightarrow 'a \Rightarrow 'a$ where
 $\text{max } a \ b = (\text{if } a \leq b \text{ then } b \text{ else } a)$

lemma *min-absorb1*: $x \leq y \implies \text{min } x \ y = x$
 by (*simp add: min-def*)

lemma *max-absorb2*: $x \leq y \implies \text{max } x \ y = y$
 by (*simp add: max-def*)

lemma *min-absorb2*: $(y::'a::\text{order}) \leq x \implies \text{min } x \ y = y$
 by (*simp add: min-def*)

lemma *max-absorb1*: $(y::'a::\text{order}) \leq x \implies \text{max } x \ y = x$
 by (*simp add: max-def*)

lemma *max-min-same* [*simp*]:
 fixes $x \ y :: 'a :: \text{linorder}$
 shows $\text{max } x \ (\text{min } x \ y) = x \ \text{max } (\text{min } x \ y) \ x = x \ \text{max } (\text{min } x \ y) \ y = y \ \text{max } y$
 $(\text{min } x \ y) = y$
 by(*auto simp add: max-def min-def*)

4.10 (Unique) top and bottom elements

class *bot* =
 fixes $\text{bot} :: 'a \ (\perp)$

class *order-bot* = *order* + *bot* +
 assumes *bot-least*: $\perp \leq a$
begin

sublocale *bot*: *ordering-top greater-eq greater bot*
 by *standard (fact bot-least)*

lemma *le-bot*:
 $a \leq \perp \implies a = \perp$
 by (*fact bot.extremum-uniqueI*)

lemma *bot-unique*:
 $a \leq \perp \longleftrightarrow a = \perp$
 by (*fact bot.extremum-unique*)

lemma *not-less-bot*:
 $\neg a < \perp$
 by (*fact bot.extremum-strict*)

```

lemma bot-less:
   $a \neq \perp \iff \perp < a$ 
  by (fact bot.not-eq-extremum)

lemma max-bot[simp]:  $\max \text{ bot } x = x$ 
by(simp add: max-def bot-unique)

lemma max-bot2[simp]:  $\max x \text{ bot} = x$ 
by(simp add: max-def bot-unique)

lemma min-bot[simp]:  $\min \text{ bot } x = \text{bot}$ 
by(simp add: min-def bot-unique)

lemma min-bot2[simp]:  $\min x \text{ bot} = \text{bot}$ 
by(simp add: min-def bot-unique)

end

class top =
  fixes top :: 'a ( $\top$ )

class order-top = order + top +
  assumes top-greatest:  $a \leq \top$ 
begin

sublocale top: ordering-top less-eq less top
  by standard (fact top-greatest)

lemma top-le:
   $\top \leq a \implies a = \top$ 
  by (fact top.extremum-uniqueI)

lemma top-unique:
   $\top \leq a \iff a = \top$ 
  by (fact top.extremum-unique)

lemma not-top-less:
   $\neg \top < a$ 
  by (fact top.extremum-strict)

lemma less-top:
   $a \neq \top \iff a < \top$ 
  by (fact top.not-eq-extremum)

lemma max-top[simp]:  $\max \text{ top } x = \text{top}$ 
by(simp add: max-def top-unique)

lemma max-top2[simp]:  $\max x \text{ top} = \text{top}$ 
by(simp add: max-def top-unique)

```

lemma *min-top[simp]*: $\text{min top } x = x$
by(*simp add: min-def top-unique*)

lemma *min-top2[simp]*: $\text{min } x \text{ top} = x$
by(*simp add: min-def top-unique*)

end

4.11 Dense orders

class *dense-order* = *order* +
assumes *dense*: $x < y \implies (\exists z. x < z \wedge z < y)$

class *dense-linorder* = *linorder* + *dense-order*
begin

lemma *dense-le*:
fixes $y z :: 'a$
assumes $\bigwedge x. x < y \implies x \leq z$
shows $y \leq z$
proof (*rule ccontr*)
assume $\neg ?thesis$
hence $z < y$ **by** *simp*
from *dense[OF this]*
obtain x **where** $x < y$ **and** $z < x$ **by** *safe*
moreover **have** $x \leq z$ **using** *assms[OF ‹x < y›]*.
ultimately show *False* **by** *auto*
qed

lemma *dense-le-bounded*:
fixes $x y z :: 'a$
assumes $x < y$
assumes *: $\bigwedge w. [x < w ; w < y] \implies w \leq z$
shows $y \leq z$
proof (*rule dense-le*)
fix w **assume** $w < y$
from *dense[OF ‹x < y›]* **obtain** u **where** $x < u$ $u < y$ **by** *safe*
from *linear[of u w]*
show $w \leq z$
proof (*rule disjE*)
assume $u \leq w$
from *less-le-trans[OF ‹x < u› ‹u ≤ w› ‹w < y›]*
show $w \leq z$ **by** (*rule **)
next
assume $w \leq u$
from $w \leq u$ **[OF ‹x < u› ‹u < y›]*
show $w \leq z$ **by** (*rule order-trans*)
qed

qed

lemma *dense-ge*:

fixes $y z :: 'a$

assumes $\bigwedge x. z < x \implies y \leq x$

shows $y \leq z$

proof (rule *ccontr*)

assume $\neg ?thesis$

hence $z < y$ by *simp*

from *dense*[*OF this*]

obtain x where $x < y$ and $z < x$ by *safe*

moreover have $y \leq x$ using *assms*[*OF <z < x>*].

ultimately show *False* by *auto*

qed

lemma *dense-ge-bounded*:

fixes $x y z :: 'a$

assumes $z < x$

assumes *: $\bigwedge w. [z < w ; w < x] \implies y \leq w$

shows $y \leq z$

proof (rule *dense-ge*)

fix w assume $z < w$

from *dense*[*OF <z < x>*] obtain u where $z < u < x$ by *safe*

from *linear*[*of u w*]

show $y \leq w$

proof (rule *disjE*)

assume $w \leq u$

from $\langle z < w \rangle$ *le-less-trans*[*OF <w ≤ u> <u < x>*]

show $y \leq w$ by (rule *)

next

assume $u \leq w$

from * [*OF <z < u> <u < x>*] $\langle u \leq w \rangle$

show $y \leq w$ by (rule *order-trans*)

qed

qed

end

class *no-top* = *order* +

assumes *gt-ex*: $\exists y. x < y$

class *no-bot* = *order* +

assumes *lt-ex*: $\exists y. y < x$

class *unbounded-dense-linorder* = *dense-linorder* + *no-top* + *no-bot*

4.12 Wellorders

class *wellorder* = *linorder* +

assumes *less-induct* [*case-names less*]: $(\bigwedge x. (\bigwedge y. y < x \implies P y) \implies P x) \implies P a$

begin

lemma *wellorder-Least-lemma*:

fixes $k :: 'a$

assumes $P k$

shows *LeastI*: $P (LEAST x. P x)$ **and** *Least-le*: $(LEAST x. P x) \leq k$

proof –

have $P (LEAST x. P x) \wedge (LEAST x. P x) \leq k$

using *assms* **proof** (*induct k rule: less-induct*)

case (*less x*) **then have** $P x$ **by** *simp*

show *?case* **proof** (*rule classical*)

assume *assm*: $\neg (P (LEAST a. P a) \wedge (LEAST a. P a) \leq k)$

have $\bigwedge y. P y \implies x \leq y$

proof (*rule classical*)

fix y

assume $P y$ **and** $\neg x \leq y$

with *less* **have** $P (LEAST a. P a)$ **and** $(LEAST a. P a) \leq k$

by (*auto simp add: not-le*)

with *assm* **have** $x < (LEAST a. P a)$ **and** $(LEAST a. P a) \leq k$

by *auto*

then show $x \leq y$ **by** *auto*

qed

with $\langle P x \rangle$ **have** *Least*: $(LEAST a. P a) = x$

by (*rule Least-equality*)

with $\langle P x \rangle$ **show** *?thesis* **by** *simp*

qed

qed

then show $P (LEAST x. P x)$ **and** $(LEAST x. P x) \leq k$ **by** *auto*

qed

— The following 3 lemmas are due to Brian Huffman

lemma *LeastI-ex*: $\exists x. P x \implies P (Least P)$

by (*erule exE*) (*erule LeastI*)

lemma *LeastI2*:

$P a \implies (\bigwedge x. P x \implies Q x) \implies Q (Least P)$

by (*blast intro: LeastI*)

lemma *LeastI2-ex*:

$\exists a. P a \implies (\bigwedge x. P x \implies Q x) \implies Q (Least P)$

by (*blast intro: LeastI-ex*)

lemma *LeastI2-wellorder*:

assumes $P a$

and $\bigwedge a. [\![P a; \forall b. P b \longrightarrow a \leq b]\!] \implies Q a$

shows $Q (Least P)$

proof (*rule LeastI2-order*)

```

  show  $P$  (Least  $P$ ) using  $\langle P \ a \rangle$  by (rule LeastI)
next
  fix  $y$  assume  $P \ y$  thus  $\text{Least } P \leq y$  by (rule Least-le)
next
  fix  $x$  assume  $P \ x \ \forall y. P \ y \longrightarrow x \leq y$  thus  $Q \ x$  by (rule assms(2))
qed

```

```

lemma LeastI2-wellorder-ex:
  assumes  $\exists x. P \ x$ 
  and  $\bigwedge a. \llbracket P \ a; \forall b. P \ b \longrightarrow a \leq b \rrbracket \Longrightarrow Q \ a$ 
  shows  $Q$  (Least  $P$ )
using assms by clarify (blast intro!: LeastI2-wellorder)

```

```

lemma not-less-Least:  $k < (\text{LEAST } x. P \ x) \Longrightarrow \neg P \ k$ 
apply (simp add: not-le [symmetric])
apply (erule contrapos-nn)
apply (erule Least-le)
done

```

```

lemma exists-least-iff:  $(\exists n. P \ n) \longleftrightarrow (\exists n. P \ n \wedge (\forall m < n. \neg P \ m))$  (is ?lhs
 $\longleftrightarrow$  ?rhs)

```

```

proof
  assume ?rhs thus ?lhs by blast
next
  assume  $H$ : ?lhs then obtain  $n$  where  $n: P \ n$  by blast
  let ?x = Least  $P$ 
  { fix  $m$  assume  $m: m < ?x$ 
    from not-less-Least[OF  $m$ ] have  $\neg P \ m$  . }
  with LeastI-ex[OF  $H$ ] show ?rhs by blast
qed

```

end

4.13 Order on *bool*

```

instantiation bool :: {order-bot, order-top, linorder}
begin

```

```

definition
  le-bool-def [simp]:  $P \leq Q \longleftrightarrow P \longrightarrow Q$ 

```

```

definition
  [simp]:  $(P::\text{bool}) < Q \longleftrightarrow \neg P \wedge Q$ 

```

```

definition
  [simp]:  $\perp \longleftrightarrow \text{False}$ 

```

```

definition
  [simp]:  $\top \longleftrightarrow \text{True}$ 

```

```

instance proof
qed auto

end

lemma le-boolI:  $(P \implies Q) \implies P \leq Q$ 
  by simp

lemma le-boolI':  $P \longrightarrow Q \implies P \leq Q$ 
  by simp

lemma le-boolE:  $P \leq Q \implies P \implies (Q \implies R) \implies R$ 
  by simp

lemma le-boolD:  $P \leq Q \implies P \longrightarrow Q$ 
  by simp

lemma bot-boolE:  $\perp \implies P$ 
  by simp

lemma top-boolI:  $\top$ 
  by simp

lemma [code]:
  False  $\leq$  b  $\longleftrightarrow$  True
  True  $\leq$  b  $\longleftrightarrow$  b
  False  $<$  b  $\longleftrightarrow$  b
  True  $<$  b  $\longleftrightarrow$  False
  by simp-all

4.14 Order on  $- \implies -$ 

instantiation fun :: (type, ord) ord
begin

definition
  le-fun-def:  $f \leq g \longleftrightarrow (\forall x. f\ x \leq g\ x)$ 

definition
   $(f :: 'a \implies 'b) < g \longleftrightarrow f \leq g \wedge \neg (g \leq f)$ 

instance ..

end

instance fun :: (type, preorder) preorder proof
qed (auto simp add: le-fun-def less-fun-def
  intro: order-trans order.antisym)

```



```
instance fun :: (type, order) order proof
qed (auto simp add: le-fun-def intro: order.antisym)
```

```
instantiation fun :: (type, bot) bot
begin
```

```
definition
  ⊥ = (λx. ⊥)
```

```
instance ..
```

```
end
```

```
instantiation fun :: (type, order-bot) order-bot
begin
```

```
lemma bot-apply [simp, code]:
  ⊥ x = ⊥
  by (simp add: bot-fun-def)
```

```
instance proof
qed (simp add: le-fun-def)
```

```
end
```

```
instantiation fun :: (type, top) top
begin
```

```
definition
  [no-atp]: ⊤ = (λx. ⊤)
```

```
instance ..
```

```
end
```

```
instantiation fun :: (type, order-top) order-top
begin
```

```
lemma top-apply [simp, code]:
  ⊤ x = ⊤
  by (simp add: top-fun-def)
```

```
instance proof
qed (simp add: le-fun-def)
```

```
end
```

```
lemma le-funI: (λx. f x ≤ g x) ⇒ f ≤ g
```

unfolding *le-fun-def* **by** *simp*

lemma *le-funE*: $f \leq g \implies (f x \leq g x \implies P) \implies P$
unfolding *le-fun-def* **by** *simp*

lemma *le-funD*: $f \leq g \implies f x \leq g x$
by (*rule le-funE*)

4.15 Order on unary and binary predicates

lemma *predicate1I*:
assumes $PQ: \bigwedge x. P x \implies Q x$
shows $P \leq Q$
apply (*rule le-funI*)
apply (*rule le-boolI*)
apply (*rule PQ*)
apply *assumption*
done

lemma *predicate1D*:
 $P \leq Q \implies P x \implies Q x$
apply (*erule le-funE*)
apply (*erule le-boolE*)
apply *assumption+*
done

lemma *rev-predicate1D*:
 $P x \implies P \leq Q \implies Q x$
by (*rule predicate1D*)

lemma *predicate2I*:
assumes $PQ: \bigwedge x y. P x y \implies Q x y$
shows $P \leq Q$
apply (*rule le-funI*)
apply (*rule le-boolI*)
apply (*rule PQ*)
apply *assumption*
done

lemma *predicate2D*:
 $P \leq Q \implies P x y \implies Q x y$
apply (*erule le-funE*)
apply (*erule le-boolE*)
apply *assumption+*
done

lemma *rev-predicate2D*:
 $P x y \implies P \leq Q \implies Q x y$
by (*rule predicate2D*)

lemma *bot1E* [*no-atp*]: $\perp x \implies P$
by (*simp add: bot-fun-def*)

lemma *bot2E*: $\perp x y \implies P$
by (*simp add: bot-fun-def*)

lemma *top1I*: $\top x$
by (*simp add: top-fun-def*)

lemma *top2I*: $\top x y$
by (*simp add: top-fun-def*)

4.16 Name duplicates

lemmas *antisym* = *order.antisym*
lemmas *eq-iff* = *order.eq-iff*

lemmas *order-eq-refl* = *preorder-class.eq-refl*
lemmas *order-less-irrefl* = *preorder-class.less-irrefl*
lemmas *order-less-imp-le* = *preorder-class.less-imp-le*
lemmas *order-less-not-sym* = *preorder-class.less-not-sym*
lemmas *order-less-asym* = *preorder-class.less-asym*
lemmas *order-less-trans* = *preorder-class.less-trans*
lemmas *order-le-less-trans* = *preorder-class.le-less-trans*
lemmas *order-less-le-trans* = *preorder-class.less-le-trans*
lemmas *order-less-imp-not-less* = *preorder-class.less-imp-not-less*
lemmas *order-less-imp-triv* = *preorder-class.less-imp-triv*
lemmas *order-less-asym'* = *preorder-class.less-asym'*

lemmas *order-less-le* = *order-class.less-le*
lemmas *order-le-less* = *order-class.le-less*
lemmas *order-le-imp-less-or-eq* = *order-class.le-imp-less-or-eq*
lemmas *order-less-imp-not-eq* = *order-class.less-imp-not-eq*
lemmas *order-less-imp-not-eq2* = *order-class.less-imp-not-eq2*
lemmas *order-neq-le-trans* = *order-class.neq-le-trans*
lemmas *order-le-neq-trans* = *order-class.le-neq-trans*
lemmas *order-eq-iff* = *order-class.order.eq-iff*
lemmas *order-antisym-conv* = *order-class.antisym-conv*

lemmas *linorder-linear* = *linorder-class.linear*
lemmas *linorder-less-linear* = *linorder-class.less-linear*
lemmas *linorder-le-less-linear* = *linorder-class.le-less-linear*
lemmas *linorder-le-cases* = *linorder-class.le-cases*
lemmas *linorder-not-less* = *linorder-class.not-less*
lemmas *linorder-not-le* = *linorder-class.not-le*
lemmas *linorder-neq-iff* = *linorder-class.neq-iff*
lemmas *linorder-neqE* = *linorder-class.neqE*

end

5 Groups, also combined with orderings

```
theory Groups
  imports Orderings
begin
```

5.1 Dynamic facts

```
named-theorems ac-simps associativity and commutativity simplification rules
  and algebra-simps algebra simplification rules for rings
  and algebra-split-simps algebra simplification rules for rings, with potential goal
  splitting
  and field-simps algebra simplification rules for fields
  and field-split-simps algebra simplification rules for fields, with potential goal
  splitting
```

The rewrites accumulated in *algebra-simps* deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by *field-simps*.

Facts in *field-simps* multiply with denominators in (in)equations if they can be proved to be non-zero (for equations) or positive/negative (for inequalities). Can be too aggressive and is therefore separate from the more benign *algebra-simps*.

Collections *algebra-split-simps* and *field-split-simps* correspond to *algebra-simps* and *field-simps* but contain more aggressive rules that may lead to goal splitting.

5.2 Abstract structures

These locales provide basic structures for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

```
locale semigroup =
  fixes f :: 'a ⇒ 'a ⇒ 'a (infixl * 70)
  assumes assoc [ac-simps]: a * b * c = a * (b * c)
```

```
locale abel-semigroup = semigroup +
  assumes commute [ac-simps]: a * b = b * a
begin
```

lemma *left-commute* [*ac-simps*]: $b * (a * c) = a * (b * c)$

proof –

have $(b * a) * c = (a * b) * c$

by (*simp only: commute*)

then show *?thesis*

by (*simp only: assoc*)

qed

end

locale *monoid* = *semigroup* +

fixes $z :: 'a$ (**1**)

assumes *left-neutral* [*simp*]: $\mathbf{1} * a = a$

assumes *right-neutral* [*simp*]: $a * \mathbf{1} = a$

locale *comm-monoid* = *abel-semigroup* +

fixes $z :: 'a$ (**1**)

assumes *comm-neutral*: $a * \mathbf{1} = a$

begin

sublocale *monoid*

by *standard* (*simp-all add: commute comm-neutral*)

end

locale *group* = *semigroup* +

fixes $z :: 'a$ (**1**)

fixes *inverse* :: $'a \Rightarrow 'a$

assumes *group-left-neutral*: $\mathbf{1} * a = a$

assumes *left-inverse* [*simp*]: *inverse* $a * a = \mathbf{1}$

begin

lemma *left-cancel*: $a * b = a * c \longleftrightarrow b = c$

proof

assume $a * b = a * c$

then have *inverse* $a * (a * b) = \textit{inverse}$ $a * (a * c)$ **by** *simp*

then have $(\textit{inverse}$ $a * a) * b = (\textit{inverse}$ $a * a) * c$

by (*simp only: assoc*)

then show $b = c$ **by** (*simp add: group-left-neutral*)

qed *simp*

sublocale *monoid*

proof

fix a

have *inverse* $a * a = \mathbf{1}$ **by** *simp*

then have *inverse* $a * (a * \mathbf{1}) = \textit{inverse}$ $a * a$

by (*simp add: group-left-neutral assoc [symmetric]*)

with *left-cancel* **show** $a * \mathbf{1} = a$

by (*simp only: left-cancel*)
 qed (*fact group-left-neutral*)

lemma *inverse-unique*:
 assumes $a * b = \mathbf{1}$
 shows $\text{inverse } a = b$
proof –
 from *assms* have $\text{inverse } a * (a * b) = \text{inverse } a$
 by *simp*
 then show *?thesis*
 by (*simp add: assoc [symmetric]*)
 qed

lemma *inverse-neutral* [*simp*]: $\text{inverse } \mathbf{1} = \mathbf{1}$
 by (*rule inverse-unique*) *simp*

lemma *inverse-inverse* [*simp*]: $\text{inverse } (\text{inverse } a) = a$
 by (*rule inverse-unique*) *simp*

lemma *right-inverse* [*simp*]: $a * \text{inverse } a = \mathbf{1}$
proof –
 have $a * \text{inverse } a = \text{inverse } (\text{inverse } a) * \text{inverse } a$
 by *simp*
 also have $\dots = \mathbf{1}$
 by (*rule left-inverse*)
 then show *?thesis* by *simp*
 qed

lemma *inverse-distrib-swap*: $\text{inverse } (a * b) = \text{inverse } b * \text{inverse } a$
proof (*rule inverse-unique*)
 have $a * b * (\text{inverse } b * \text{inverse } a) =$
 $a * (b * \text{inverse } b) * \text{inverse } a$
 by (*simp only: assoc*)
 also have $\dots = \mathbf{1}$
 by *simp*
 finally show $a * b * (\text{inverse } b * \text{inverse } a) = \mathbf{1}$.
 qed

lemma *right-cancel*: $b * a = c * a \iff b = c$
proof
 assume $b * a = c * a$
 then have $b * a * \text{inverse } a = c * a * \text{inverse } a$
 by *simp*
 then show $b = c$
 by (*simp add: assoc*)
 qed *simp*

end

5.3 Generic operations

```

class zero =
  fixes zero :: 'a (0)

class one =
  fixes one :: 'a (1)

hide-const (open) zero one

lemma Let-0 [simp]: Let 0 f = f 0
  unfolding Let-def ..

lemma Let-1 [simp]: Let 1 f = f 1
  unfolding Let-def ..

setup <
  Reorient-Proc.add
  (fn Const(const-name <Groups.zero>, -) => true
   | Const(const-name <Groups.one>, -) => true
   | - => false)
>

simproc-setup reorient-zero (0 = x) = <K Reorient-Proc.proc>
simproc-setup reorient-one (1 = x) = <K Reorient-Proc.proc>

typed-print-translation <
  let
    fun tr' c = (c, fn ctxt => fn T => fn ts =>
      if null ts andalso Printer.type-emphasis ctxt T then
        Syntax.const syntax-const <-constrain> $ Syntax.const c $
          Syntax-Phases.term-of-typ ctxt T
      else raise Match);
    in map tr' [const-syntax <Groups.one>, const-syntax <Groups.zero>] end
  > — show types that are presumably too general

class plus =
  fixes plus :: 'a ⇒ 'a ⇒ 'a (infixl + 65)

class minus =
  fixes minus :: 'a ⇒ 'a ⇒ 'a (infixl - 65)

class uminus =
  fixes uminus :: 'a ⇒ 'a (- - [81] 80)

class times =
  fixes times :: 'a ⇒ 'a ⇒ 'a (infixl * 70)

```

5.4 Semigroups and Monoids

```

class semigroup-add = plus +
  assumes add-assoc [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
     $(a + b) + c = a + (b + c)$ 
begin

sublocale add: semigroup plus
  by standard (fact add-assoc)

end

hide-fact add-assoc

class ab-semigroup-add = semigroup-add +
  assumes add-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
     $a + b = b + a$ 
begin

sublocale add: ab-semigroup plus
  by standard (fact add-commute)

declare add.left-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]

lemmas add-ac = add.assoc add.commute add.left-commute

end

hide-fact add-commute

lemmas add-ac = add.assoc add.commute add.left-commute

class semigroup-mult = times +
  assumes mult-assoc [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
     $(a * b) * c = a * (b * c)$ 
begin

sublocale mult: semigroup times
  by standard (fact mult-assoc)

end

hide-fact mult-assoc

class ab-semigroup-mult = semigroup-mult +
  assumes mult-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
     $a * b = b * a$ 
begin

sublocale mult: ab-semigroup times

```



```

    by standard (fact mult-commute)

declare mult.left-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]

lemmas mult-ac = mult.assoc mult.commute mult.left-commute

end

hide-fact mult-commute

lemmas mult-ac = mult.assoc mult.commute mult.left-commute

class monoid-add = zero + semigroup-add +
  assumes add-0-left:  $0 + a = a$ 
  and add-0-right:  $a + 0 = a$ 
begin

sublocale add: monoid plus 0
  by standard (fact add-0-left add-0-right)+

end

lemma zero-reorient:  $0 = x \longleftrightarrow x = 0$ 
  by (fact eq-commute)

class comm-monoid-add = zero + ab-semigroup-add +
  assumes add-0:  $0 + a = a$ 
begin

subclass monoid-add
  by standard (simp-all add: add-0 add.commute [of - 0])

sublocale add: comm-monoid plus 0
  by standard (simp add: ac-simps)

end

class monoid-mult = one + semigroup-mult +
  assumes mult-1-left:  $1 * a = a$ 
  and mult-1-right:  $a * 1 = a$ 
begin

sublocale mult: monoid times 1
  by standard (fact mult-1-left mult-1-right)+

end

lemma one-reorient:  $1 = x \longleftrightarrow x = 1$ 
  by (fact eq-commute)

```

```

class comm-monoid-mult = one + ab-semigroup-mult +
  assumes mult-1: 1 * a = a
begin

subclass monoid-mult
  by standard (simp-all add: mult-1 mult.commute [of - 1])

sublocale mult: comm-monoid times 1
  by standard (simp add: ac-simps)

end

class cancel-semigroup-add = semigroup-add +
  assumes add-left-imp-eq: a + b = a + c  $\implies$  b = c
  assumes add-right-imp-eq: b + a = c + a  $\implies$  b = c
begin

lemma add-left-cancel [simp]: a + b = a + c  $\longleftrightarrow$  b = c
  by (blast dest: add-left-imp-eq)

lemma add-right-cancel [simp]: b + a = c + a  $\longleftrightarrow$  b = c
  by (blast dest: add-right-imp-eq)

end

class cancel-ab-semigroup-add = ab-semigroup-add + minus +
  assumes add-diff-cancel-left' [simp]: (a + b) - a = b
  assumes diff-diff-add [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
    a - b - c = a - (b + c)
begin

lemma add-diff-cancel-right' [simp]: (a + b) - b = a
  using add-diff-cancel-left' [of b a] by (simp add: ac-simps)

subclass cancel-semigroup-add
proof
  fix a b c :: 'a
  assume a + b = a + c
  then have a + b - a = a + c - a
    by simp
  then show b = c
    by simp
next
  fix a b c :: 'a
  assume b + a = c + a
  then have b + a - a = c + a - a
    by simp
  then show b = c

```

by *simp*
qed

lemma *add-diff-cancel-left* [*simp*]: $(c + a) - (c + b) = a - b$
unfolding *diff-diff-add* [*symmetric*] by *simp*

lemma *add-diff-cancel-right* [*simp*]: $(a + c) - (b + c) = a - b$
using *add-diff-cancel-left* [*symmetric*] by (*simp add: ac-simps*)

lemma *diff-right-commute*: $a - c - b = a - b - c$
by (*simp add: diff-diff-add add commute*)

end

class *cancel-comm-monoid-add* = *cancel-ab-semigroup-add* + *comm-monoid-add*
begin

lemma *diff-zero* [*simp*]: $a - 0 = a$
using *add-diff-cancel-right'* [*of a 0*] by *simp*

lemma *diff-cancel* [*simp*]: $a - a = 0$
proof –
have $(a + 0) - (a + 0) = 0$
by (*simp only: add-diff-cancel-left diff-zero*)
then show *?thesis* by *simp*
qed

lemma *add-implies-diff*:
assumes $c + b = a$
shows $c = a - b$
proof –
from *assms* have $(b + c) - (b + 0) = a - b$
by (*simp add: add commute*)
then show $c = a - b$ by *simp*
qed

lemma *add-cancel-right-right* [*simp*]: $a = a + b \longleftrightarrow b = 0$
(is *?P* \longleftrightarrow *?Q*)
proof
assume *?Q*
then show *?P* by *simp*
next
assume *?P*
then have $a - a = a + b - a$ by *simp*
then show *?Q* by *simp*
qed

lemma *add-cancel-right-left* [*simp*]: $a = b + a \longleftrightarrow b = 0$
using *add-cancel-right-right* [*of a b*] by (*simp add: ac-simps*)

lemma *add-cancel-left-right* [*simp*]: $a + b = a \leftrightarrow b = 0$
by (*auto dest: sym*)

lemma *add-cancel-left-left* [*simp*]: $b + a = a \leftrightarrow b = 0$
by (*auto dest: sym*)

end

class *comm-monoid-diff* = *cancel-comm-monoid-add* +
assumes *zero-diff* [*simp*]: $0 - a = 0$
begin

lemma *diff-add-zero* [*simp*]: $a - (a + b) = 0$

proof –

have $a - (a + b) = (a + 0) - (a + b)$

by *simp*

also have $\dots = 0$

by (*simp only: add-diff-cancel-left zero-diff*)

finally show *?thesis* .

qed

end

5.5 Groups

class *group-add* = *minus* + *uminus* + *monoid-add* +
assumes *left-minus*: $- a + a = 0$
assumes *add-uminus-conv-diff* [*simp*]: $a + (- b) = a - b$
begin

lemma *diff-conv-add-uminus*: $a - b = a + (- b)$

by *simp*

sublocale *add: group plus 0 uminus*

by *standard (simp-all add: left-minus)*

lemma *minus-unique*: $a + b = 0 \implies - a = b$

by (*fact add.inverse-unique*)

lemma *minus-zero*: $- 0 = 0$

by (*fact add.inverse-neutral*)

lemma *minus-minus*: $- (- a) = a$

by (*fact add.inverse-inverse*)

lemma *right-minus*: $a + - a = 0$

by (*fact add.right-inverse*)

lemma *diff-self* [*simp*]: $a - a = 0$
using *right-minus* [*of a*] **by** *simp*

subclass *cancel-semigroup-add*
by *standard* (*simp-all* *add: add.left-cancel add.right-cancel*)

lemma *minus-add-cancel* [*simp*]: $- a + (a + b) = b$
by (*simp* *add: add.assoc* [*symmetric*])

lemma *add-minus-cancel* [*simp*]: $a + (- a + b) = b$
by (*simp* *add: add.assoc* [*symmetric*])

lemma *diff-add-cancel* [*simp*]: $a - b + b = a$
by (*simp* *only: diff-conv-add-uminus add.assoc*) *simp*

lemma *add-diff-cancel* [*simp*]: $a + b - b = a$
by (*simp* *only: diff-conv-add-uminus add.assoc*) *simp*

lemma *minus-add*: $-(a + b) = - b + - a$
by (*fact* *add.inverse-distrib-swap*)

lemma *right-minus-eq* [*simp*]: $a - b = 0 \longleftrightarrow a = b$
proof
assume $a - b = 0$
have $a = (a - b) + b$ **by** (*simp* *add: add.assoc*)
also have $\dots = b$ **using** $\langle a - b = 0 \rangle$ **by** *simp*
finally show $a = b$.

next
assume $a = b$
then show $a - b = 0$ **by** *simp*

qed

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$
by (*fact* *right-minus-eq* [*symmetric*])

lemma *diff-0* [*simp*]: $0 - a = - a$
by (*simp* *only: diff-conv-add-uminus add-0-left*)

lemma *diff-0-right* [*simp*]: $a - 0 = a$
by (*simp* *only: diff-conv-add-uminus minus-zero add-0-right*)

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + b$
by (*simp* *only: diff-conv-add-uminus minus-minus*)

lemma *neg-equal-iff-equal* [*simp*]: $- a = - b \longleftrightarrow a = b$
proof
assume $- a = - b$
then have $- (- a) = - (- b)$ **by** *simp*
then show $a = b$ **by** *simp*

next
assume $a = b$
then show $- a = - b$ **by** *simp*
qed

lemma *neg-equal-0-iff-equal* [*simp*]: $- a = 0 \longleftrightarrow a = 0$
by (*subst neg-equal-iff-equal [symmetric]*) *simp*

lemma *neg-0-equal-iff-equal* [*simp*]: $0 = - a \longleftrightarrow 0 = a$
by (*subst neg-equal-iff-equal [symmetric]*) *simp*

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*: $a = - b \longleftrightarrow b = - a$
proof –
have $- (- a) = - b \longleftrightarrow - a = b$
by (*rule neg-equal-iff-equal*)
then show *?thesis*
by (*simp add: eq-commute*)
qed

lemma *minus-equation-iff*: $- a = b \longleftrightarrow - b = a$
proof –
have $- a = - (- b) \longleftrightarrow a = - b$
by (*rule neg-equal-iff-equal*)
then show *?thesis*
by (*simp add: eq-commute*)
qed

lemma *eq-neg-iff-add-eq-0*: $a = - b \longleftrightarrow a + b = 0$
proof
assume $a = - b$
then show $a + b = 0$ **by** *simp*
next
assume $a + b = 0$
moreover have $a + (b + - b) = (a + b) + - b$
by (*simp only: add.assoc*)
ultimately show $a = - b$
by *simp*
qed

lemma *add-eq-0-iff2*: $a + b = 0 \longleftrightarrow a = - b$
by (*fact eq-neg-iff-add-eq-0 [symmetric]*)

lemma *neg-eq-iff-add-eq-0*: $- a = b \longleftrightarrow a + b = 0$
by (*auto simp add: add-eq-0-iff2*)

lemma *add-eq-0-iff*: $a + b = 0 \longleftrightarrow b = - a$
by (*auto simp add: neg-eq-iff-add-eq-0 [symmetric]*)

lemma *minus-diff-eq* [*simp*]: $-(a - b) = b - a$
by (*simp only: neg-eq-iff-add-eq-0 diff-conv-add-uminus add.assoc minus-add-cancel*)
simp

lemma *add-diff-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a + (b - c) = (a + b) - c$
by (*simp only: diff-conv-add-uminus add.assoc*)

lemma *diff-add-eq-diff-diff-swap*: $a - (b + c) = a - c - b$
by (*simp only: diff-conv-add-uminus add.assoc minus-add*)

lemma *diff-eq-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a - b = c \iff a = c + b$
by *auto*

lemma *eq-diff-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a = c - b \iff a + b = c$
by *auto*

lemma *diff-diff-eq2* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a - (b - c) = (a + c) - b$
by (*simp only: diff-conv-add-uminus add.assoc*) *simp*

lemma *diff-eq-diff-eq*: $a - b = c - d \implies a = b \iff c = d$
by (*simp only: eq-iff-diff-eq-0 [of a b] eq-iff-diff-eq-0 [of c d]*)

end

class *ab-group-add* = *minus + uminus + comm-monoid-add +*
assumes *ab-left-minus*: $-a + a = 0$
assumes *ab-diff-conv-add-uminus*: $a - b = a + (-b)$
begin

subclass *group-add*
by *standard (simp-all add: ab-left-minus ab-diff-conv-add-uminus)*

subclass *cancel-comm-monoid-add*
proof
fix *a b c* :: 'a
have $b + a - a = b$
by *simp*
then show $a + b - a = b$
by (*simp add: ac-simps*)
show $a - b - c = a - (b + c)$
by (*simp add: algebra-simps*)
qed

lemma *uminus-add-conv-diff* [*simp*]: $-a + b = b - a$
by (*simp add: add.commute*)

lemma *minus-add-distrib* [*simp*]: $-(a + b) = -a + -b$
by (*simp add: algebra-simps*)

lemma *diff-add-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $(a - b) + c = (a + c) - b$
by (*simp add: algebra-simps*)

lemma *minus-diff-commute*:
 $-b - a = -a - b$
by (*simp only: diff-conv-add-uminus add.commute*)

end

5.6 (Partially) Ordered Groups

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society, 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press, 1963

Most of the used notions can also be looked up in

- <http://www.mathworld.com> by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

class *ordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
assumes *add-left-mono*: $a \leq b \implies c + a \leq c + b$
begin

lemma *add-right-mono*: $a \leq b \implies a + c \leq b + c$
by (*simp add: add.commute [of - c] add-left-mono*)

non-strict, in both arguments

lemma *add-mono*: $a \leq b \implies c \leq d \implies a + c \leq b + d$
by (*simp add: add.commute add-left-mono add-right-mono [THEN order-trans]*)

end

Strict monotonicity in both arguments

class *strict-ordered-ab-semigroup-add* = *ordered-ab-semigroup-add* +
assumes *add-strict-mono*: $a < b \implies c < d \implies a + c < b + d$

class *ordered-cancel-ab-semigroup-add* =
ordered-ab-semigroup-add + *cancel-ab-semigroup-add*

begin

lemma *add-strict-left-mono*: $a < b \implies c + a < c + b$
by (*auto simp add: less-le add-left-mono*)

lemma *add-strict-right-mono*: $a < b \implies a + c < b + c$
by (*simp add: add.commute [of - c] add-strict-left-mono*)

subclass *strict-ordered-ab-semigroup-add*

proof

show $\bigwedge a b c d. \llbracket a < b; c < d \rrbracket \implies a + c < b + d$

by (*iprover intro: add-strict-left-mono add-strict-right-mono less-trans*)

qed

lemma *add-less-le-mono*: $a < b \implies c \leq d \implies a + c < b + d$
by (*iprover intro: add-left-mono add-strict-right-mono less-le-trans*)

lemma *add-le-less-mono*: $a \leq b \implies c < d \implies a + c < b + d$
by (*iprover intro: add-strict-left-mono add-right-mono less-le-trans*)

end

class *ordered-ab-semigroup-add-imp-le* = *ordered-cancel-ab-semigroup-add* +
assumes *add-le-imp-le-left*: $c + a \leq c + b \implies a \leq b$

begin

lemma *add-less-imp-less-left*:

assumes *less*: $c + a < c + b$

shows $a < b$

proof –

from *less* **have** *le*: $c + a \leq c + b$

by (*simp add: order-le-less*)

have $a \leq b$

using *add-le-imp-le-left* [*OF le*] .

moreover **have** $a \neq b$

proof (*rule ccontr*)

assume $\neg ?thesis$

then **have** $a = b$ **by** *simp*

then **have** $c + a = c + b$ **by** *simp*

with *less* **show** *False* **by** *simp*

qed

ultimately **show** $a < b$

by (*simp add: order-le-less*)

qed

lemma *add-less-imp-less-right*: $a + c < b + c \implies a < b$
by (*rule add-less-imp-less-left [of c]*) (*simp add: add.commute*)

lemma *add-less-cancel-left* [*simp*]: $c + a < c + b \longleftrightarrow a < b$

by (blast intro: add-less-imp-less-left add-strict-left-mono)

lemma *add-less-cancel-right* [simp]: $a + c < b + c \longleftrightarrow a < b$
 by (blast intro: add-less-imp-less-right add-strict-right-mono)

lemma *add-le-cancel-left* [simp]: $c + a \leq c + b \longleftrightarrow a \leq b$
 by (auto simp: dest: add-le-imp-le-left add-left-mono)

lemma *add-le-cancel-right* [simp]: $a + c \leq b + c \longleftrightarrow a \leq b$
 by (simp add: add.commute [of a c] add.commute [of b c])

lemma *add-le-imp-le-right*: $a + c \leq b + c \implies a \leq b$
 by simp

lemma *max-add-distrib-left*: $\max x y + z = \max (x + z) (y + z)$
 unfolding max-def by auto

lemma *min-add-distrib-left*: $\min x y + z = \min (x + z) (y + z)$
 unfolding min-def by auto

lemma *max-add-distrib-right*: $x + \max y z = \max (x + y) (x + z)$
 unfolding max-def by auto

lemma *min-add-distrib-right*: $x + \min y z = \min (x + y) (x + z)$
 unfolding min-def by auto

end

5.7 Support for reasoning about signs

class *ordered-comm-monoid-add* = *comm-monoid-add* + *ordered-ab-semigroup-add*
begin

lemma *add-nonneg-nonneg* [simp]: $0 \leq a \implies 0 \leq b \implies 0 \leq a + b$
 using add-mono[of 0 a 0 b] by simp

lemma *add-nonpos-nonpos*: $a \leq 0 \implies b \leq 0 \implies a + b \leq 0$
 using add-mono[of a 0 b 0] by simp

lemma *add-nonneg-eq-0-iff*: $0 \leq x \implies 0 \leq y \implies x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
 using add-left-mono[of 0 y x] add-right-mono[of 0 x y] by auto

lemma *add-nonpos-eq-0-iff*: $x \leq 0 \implies y \leq 0 \implies x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
 using add-left-mono[of y 0 x] add-right-mono[of x 0 y] by auto

lemma *add-increasing*: $0 \leq a \implies b \leq c \implies b \leq a + c$
 using add-mono [of 0 a b c] by simp

lemma *add-increasing2*: $0 \leq c \implies b \leq a \implies b \leq a + c$

```

    by (simp add: add-increasing add.commute [of a])

lemma add-decreasing:  $a \leq 0 \implies c \leq b \implies a + c \leq b$ 
  using add-mono [of a 0 c b] by simp

lemma add-decreasing2:  $c \leq 0 \implies a \leq b \implies a + c \leq b$ 
  using add-mono[of a b c 0] by simp

lemma add-pos-nonneg:  $0 < a \implies 0 \leq b \implies 0 < a + b$ 
  using less-le-trans[of 0 a a + b] by (simp add: add-increasing2)

lemma add-pos-pos:  $0 < a \implies 0 < b \implies 0 < a + b$ 
  by (intro add-pos-nonneg less-imp-le)

lemma add-nonneg-pos:  $0 \leq a \implies 0 < b \implies 0 < a + b$ 
  using add-pos-nonneg[of b a] by (simp add: add-commute)

lemma add-neg-nonpos:  $a < 0 \implies b \leq 0 \implies a + b < 0$ 
  using le-less-trans[of a + b a 0] by (simp add: add-decreasing2)

lemma add-neg-neg:  $a < 0 \implies b < 0 \implies a + b < 0$ 
  by (intro add-neg-nonpos less-imp-le)

lemma add-nonpos-neg:  $a \leq 0 \implies b < 0 \implies a + b < 0$ 
  using add-neg-nonpos[of b a] by (simp add: add-commute)

lemmas add-sign-intros =
  add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
  add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

end

class strict-ordered-comm-monoid-add = comm-monoid-add + strict-ordered-ab-semigroup-add
begin

lemma pos-add-strict:  $0 < a \implies b < c \implies b < a + c$ 
  using add-strict-mono [of 0 a b c] by simp

end

class ordered-cancel-comm-monoid-add = ordered-comm-monoid-add + cancel-ab-semigroup-add
begin

subclass ordered-cancel-ab-semigroup-add ..
subclass strict-ordered-comm-monoid-add ..

lemma add-strict-increasing:  $0 < a \implies b \leq c \implies b < a + c$ 
  using add-less-le-mono [of 0 a b c] by simp

```

lemma *add-strict-increasing2*: $0 \leq a \implies b < c \implies b < a + c$
using *add-le-less-mono* [of 0 a b c] **by** *simp*

end

class *ordered-ab-semigroup-monoid-add-imp-le* = *monoid-add* + *ordered-ab-semigroup-add-imp-le*
begin

lemma *add-less-same-cancel1* [*simp*]: $b + a < b \longleftrightarrow a < 0$
using *add-less-cancel-left* [of - - 0] **by** *simp*

lemma *add-less-same-cancel2* [*simp*]: $a + b < b \longleftrightarrow a < 0$
using *add-less-cancel-right* [of - - 0] **by** *simp*

lemma *less-add-same-cancel1* [*simp*]: $a < a + b \longleftrightarrow 0 < b$
using *add-less-cancel-left* [of - 0] **by** *simp*

lemma *less-add-same-cancel2* [*simp*]: $a < b + a \longleftrightarrow 0 < b$
using *add-less-cancel-right* [of 0] **by** *simp*

lemma *add-le-same-cancel1* [*simp*]: $b + a \leq b \longleftrightarrow a \leq 0$
using *add-le-cancel-left* [of - - 0] **by** *simp*

lemma *add-le-same-cancel2* [*simp*]: $a + b \leq b \longleftrightarrow a \leq 0$
using *add-le-cancel-right* [of - - 0] **by** *simp*

lemma *le-add-same-cancel1* [*simp*]: $a \leq a + b \longleftrightarrow 0 \leq b$
using *add-le-cancel-left* [of - 0] **by** *simp*

lemma *le-add-same-cancel2* [*simp*]: $a \leq b + a \longleftrightarrow 0 \leq b$
using *add-le-cancel-right* [of 0] **by** *simp*

subclass *cancel-comm-monoid-add*
by *standard auto*

subclass *ordered-cancel-comm-monoid-add*
by *standard*

end

class *ordered-ab-group-add* = *ab-group-add* + *ordered-ab-semigroup-add*
begin

subclass *ordered-cancel-ab-semigroup-add* ..

subclass *ordered-ab-semigroup-monoid-add-imp-le*

proof

fix *a b c* :: 'a
assume $c + a \leq c + b$

then have $(-c) + (c + a) \leq (-c) + (c + b)$
by (*rule add-left-mono*)
then have $((-c) + c) + a \leq ((-c) + c) + b$
by (*simp only: add.assoc*)
then show $a \leq b$ **by** *simp*
qed

lemma *max-diff-distrib-left*: $\max x y - z = \max (x - z) (y - z)$
using *max-add-distrib-left* [*of x y - z*] **by** *simp*

lemma *min-diff-distrib-left*: $\min x y - z = \min (x - z) (y - z)$
using *min-add-distrib-left* [*of x y - z*] **by** *simp*

lemma *le-imp-neg-le*:
assumes $a \leq b$
shows $-b \leq -a$
proof –
from *assms* **have** $-a + a \leq -a + b$
by (*rule add-left-mono*)
then have $0 \leq -a + b$
by *simp*
then have $0 + (-b) \leq (-a + b) + (-b)$
by (*rule add-right-mono*)
then show *?thesis*
by (*simp add: algebra-simps*)
qed

lemma *neg-le-iff-le* [*simp*]: $-b \leq -a \longleftrightarrow a \leq b$
proof
assume $-b \leq -a$
then have $-(-a) \leq -(-b)$
by (*rule le-imp-neg-le*)
then show $a \leq b$
by *simp*
next
assume $a \leq b$
then show $-b \leq -a$
by (*rule le-imp-neg-le*)
qed

lemma *neg-le-0-iff-le* [*simp*]: $-a \leq 0 \longleftrightarrow 0 \leq a$
by (*subst neg-le-iff-le* [*symmetric*]) *simp*

lemma *neg-0-le-iff-le* [*simp*]: $0 \leq -a \longleftrightarrow a \leq 0$
by (*subst neg-le-iff-le* [*symmetric*]) *simp*

lemma *neg-less-iff-less* [*simp*]: $-b < -a \longleftrightarrow a < b$
by (*auto simp add: less-le*)

lemma *neg-less-0-iff-less* [*simp*]: $- a < 0 \iff 0 < a$
by (*subst neg-less-iff-less [symmetric]*) *simp*

lemma *neg-0-less-iff-less* [*simp*]: $0 < - a \iff a < 0$
by (*subst neg-less-iff-less [symmetric]*) *simp*

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \iff b < - a$
proof –
have $- (- a) < - b \iff b < - a$
by (*rule neg-less-iff-less*)
then show *?thesis* **by** *simp*
qed

lemma *minus-less-iff*: $- a < b \iff - b < a$
proof –
have $- a < - (- b) \iff - b < a$
by (*rule neg-less-iff-less*)
then show *?thesis* **by** *simp*
qed

lemma *le-minus-iff*: $a \leq - b \iff b \leq - a$
by (*auto simp: order.order-iff-strict less-minus-iff*)

lemma *minus-le-iff*: $- a \leq b \iff - b \leq a$
by (*auto simp add: le-less minus-less-iff*)

lemma *diff-less-0-iff-less* [*simp*]: $a - b < 0 \iff a < b$
proof –
have $a - b < 0 \iff a + (- b) < b + (- b)$
by *simp*
also have $\dots \iff a < b$
by (*simp only: add-less-cancel-right*)
finally show *?thesis* .
qed

lemmas *less-iff-diff-less-0 = diff-less-0-iff-less* [*symmetric*]

lemma *diff-less-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a - b < c \iff a < c + b$
proof (*subst less-iff-diff-less-0 [of a]*)
show $(a - b < c) = (a - (c + b) < 0)$
by (*simp add: algebra-simps less-iff-diff-less-0 [of - c]*)
qed

lemma *less-diff-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a < c - b \iff a + b < c$
proof (*subst less-iff-diff-less-0 [of a + b]*)
show $(a < c - b) = (a + b - c < 0)$

by (*simp add: algebra-simps less-iff-diff-less-0 [of a]*)
 qed

lemma *diff-gt-0-iff-gt* [*simp*]: $a - b > 0 \iff a > b$
 by (*simp add: less-diff-eq*)

lemma *diff-le-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a - b \leq c \iff a \leq c + b$
 by (*auto simp add: le-less diff-less-eq*)

lemma *le-diff-eq* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps*]:
 $a \leq c - b \iff a + b \leq c$
 by (*auto simp add: le-less less-diff-eq*)

lemma *diff-le-0-iff-le* [*simp*]: $a - b \leq 0 \iff a \leq b$
 by (*simp add: algebra-simps*)

lemmas *le-iff-diff-le-0 = diff-le-0-iff-le* [*symmetric*]

lemma *diff-ge-0-iff-ge* [*simp*]: $a - b \geq 0 \iff a \geq b$
 by (*simp add: le-diff-eq*)

lemma *diff-eq-diff-less*: $a - b = c - d \implies a < b \iff c < d$
 by (*auto simp only: less-iff-diff-less-0 [of a b] less-iff-diff-less-0 [of c d]*)

lemma *diff-eq-diff-less-eq*: $a - b = c - d \implies a \leq b \iff c \leq d$
 by (*auto simp only: le-iff-diff-le-0 [of a b] le-iff-diff-le-0 [of c d]*)

lemma *diff-mono*: $a \leq b \implies d \leq c \implies a - c \leq b - d$
 by (*simp add: field-simps add-mono*)

lemma *diff-left-mono*: $b \leq a \implies c - a \leq c - b$
 by (*simp add: field-simps*)

lemma *diff-right-mono*: $a \leq b \implies a - c \leq b - c$
 by (*simp add: field-simps*)

lemma *diff-strict-mono*: $a < b \implies d < c \implies a - c < b - d$
 by (*simp add: field-simps add-strict-mono*)

lemma *diff-strict-left-mono*: $b < a \implies c - a < c - b$
 by (*simp add: field-simps*)

lemma *diff-strict-right-mono*: $a < b \implies a - c < b - c$
 by (*simp add: field-simps*)

end

locale *group-cancel*

begin

lemma *add1*: $(A::'a::\text{comm-monoid-add}) \equiv k + a \implies A + b \equiv k + (a + b)$
by (*simp only: ac-simps*)

lemma *add2*: $(B::'a::\text{comm-monoid-add}) \equiv k + b \implies a + B \equiv k + (a + b)$
by (*simp only: ac-simps*)

lemma *sub1*: $(A::'a::\text{ab-group-add}) \equiv k + a \implies A - b \equiv k + (a - b)$
by (*simp only: add-diff-eq*)

lemma *sub2*: $(B::'a::\text{ab-group-add}) \equiv k + b \implies a - B \equiv -k + (a - b)$
by (*simp only: minus-add diff-conv-add-uminus ac-simps*)

lemma *neg1*: $(A::'a::\text{ab-group-add}) \equiv k + a \implies -A \equiv -k + -a$
by (*simp only: minus-add-distrib*)

lemma *rule0*: $(a::'a::\text{comm-monoid-add}) \equiv a + 0$
by (*simp only: add-0-right*)

end

ML-file $\langle \text{Tools/group-cancel.ML} \rangle$

simproc-setup *group-cancel-add* $(a + b::'a::\text{ab-group-add}) =$
 $\langle \text{fn phi} \Rightarrow \text{fn ss} \Rightarrow \text{try Group-Cancel.cancel-add-conv} \rangle$

simproc-setup *group-cancel-diff* $(a - b::'a::\text{ab-group-add}) =$
 $\langle \text{fn phi} \Rightarrow \text{fn ss} \Rightarrow \text{try Group-Cancel.cancel-diff-conv} \rangle$

simproc-setup *group-cancel-eq* $(a = (b::'a::\text{ab-group-add})) =$
 $\langle \text{fn phi} \Rightarrow \text{fn ss} \Rightarrow \text{try Group-Cancel.cancel-eq-conv} \rangle$

simproc-setup *group-cancel-le* $(a \leq (b::'a::\text{ordered-ab-group-add})) =$
 $\langle \text{fn phi} \Rightarrow \text{fn ss} \Rightarrow \text{try Group-Cancel.cancel-le-conv} \rangle$

simproc-setup *group-cancel-less* $(a < (b::'a::\text{ordered-ab-group-add})) =$
 $\langle \text{fn phi} \Rightarrow \text{fn ss} \Rightarrow \text{try Group-Cancel.cancel-less-conv} \rangle$

class *linordered-ab-semigroup-add* =
linorder + *ordered-ab-semigroup-add*

class *linordered-cancel-ab-semigroup-add* =
linorder + *ordered-cancel-ab-semigroup-add*

begin

subclass *linordered-ab-semigroup-add* ..

subclass *ordered-ab-semigroup-add-imp-le*


```

proof
  fix  $a\ b\ c :: 'a$ 
  assume  $le1: c + a \leq c + b$ 
  show  $a \leq b$ 
  proof (rule ccontr)
    assume  $*$ :  $\neg ?thesis$ 
    then have  $b \leq a$  by (simp add: linorder-not-le)
    then have  $c + b \leq c + a$  by (rule add-left-mono)
    then have  $c + a = c + b$ 
      using  $le1$  by (iprover intro: order.antisym)
    then have  $a = b$ 
      by simp
    with  $*$  show False
      by (simp add: linorder-not-le [symmetric])
  qed
qed

end

class linordered-ab-group-add = linorder + ordered-ab-group-add
begin

subclass linordered-cancel-ab-semigroup-add ..

lemma equal-neg-zero [simp]:  $a = - a \longleftrightarrow a = 0$ 
proof
  assume  $a = 0$ 
  then show  $a = - a$  by simp
next
  assume  $A: a = - a$ 
  show  $a = 0$ 
  proof (cases  $0 \leq a$ )
    case True
      with  $A$  have  $0 \leq - a$  by auto
      with le-minus-iff have  $a \leq 0$  by simp
      with True show  $?thesis$  by (auto intro: order-trans)
    next
      case False
      then have  $B: a \leq 0$  by auto
      with  $A$  have  $- a \leq 0$  by auto
      with  $B$  show  $?thesis$  by (auto intro: order-trans)
  qed
qed

lemma neg-equal-zero [simp]:  $- a = a \longleftrightarrow a = 0$ 
  by (auto dest: sym)

lemma neg-less-eq-nonneg [simp]:  $- a \leq a \longleftrightarrow 0 \leq a$ 
proof

```

```

assume *:  $- a \leq a$ 
show  $0 \leq a$ 
proof (rule classical)
  assume  $\neg ?thesis$ 
  then have  $a < 0$  by auto
  with * have  $- a < 0$  by (rule le-less-trans)
  then show ?thesis by auto
qed
next
  assume *:  $0 \leq a$ 
  then have  $- a \leq 0$  by (simp add: minus-le-iff)
  from this * show  $- a \leq a$  by (rule order-trans)
qed

```

```

lemma neg-less-pos [simp]:  $- a < a \longleftrightarrow 0 < a$ 
  by (auto simp add: less-le)

```

```

lemma less-eq-neg-nonpos [simp]:  $a \leq - a \longleftrightarrow a \leq 0$ 
  using neg-less-eq-nonneg [of - a] by simp

```

```

lemma less-neg-neg [simp]:  $a < - a \longleftrightarrow a < 0$ 
  using neg-less-pos [of - a] by simp

```

```

lemma double-zero [simp]:  $a + a = 0 \longleftrightarrow a = 0$ 
proof
  assume  $a + a = 0$ 
  then have  $a: - a = a$  by (rule minus-unique)
  then show  $a = 0$  by (simp only: neg-equal-zero)
next
  assume  $a = 0$ 
  then show  $a + a = 0$  by simp
qed

```

```

lemma double-zero-sym [simp]:  $0 = a + a \longleftrightarrow a = 0$ 
  using double-zero [of a] by (simp only: eq-commute)

```

```

lemma zero-less-double-add-iff-zero-less-single-add [simp]:  $0 < a + a \longleftrightarrow 0 < a$ 
proof
  assume  $0 < a + a$ 
  then have  $0 - a < a$  by (simp only: diff-less-eq)
  then have  $- a < a$  by simp
  then show  $0 < a$  by simp
next
  assume  $0 < a$ 
  with this have  $0 + 0 < a + a$ 
    by (rule add-strict-mono)
  then show  $0 < a + a$  by simp
qed

```

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: $0 \leq a + a \longleftrightarrow 0 \leq a$
by (*auto simp add: le-less*)

lemma *double-add-less-zero-iff-single-add-less-zero* [*simp*]: $a + a < 0 \longleftrightarrow a < 0$
proof –
have $\neg a + a < 0 \longleftrightarrow \neg a < 0$
by (*simp add: not-less*)
then show *?thesis* **by** *simp*
qed

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: $a + a \leq 0 \longleftrightarrow a \leq 0$
proof –
have $\neg a + a \leq 0 \longleftrightarrow \neg a \leq 0$
by (*simp add: not-le*)
then show *?thesis* **by** *simp*
qed

lemma *minus-max-eq-min*: $-\max x y = \min (-x) (-y)$
by (*auto simp add: max-def min-def*)

lemma *minus-min-eq-max*: $-\min x y = \max (-x) (-y)$
by (*auto simp add: max-def min-def*)

end

class *abs* =
fixes *abs* :: 'a \Rightarrow 'a (*|-*)

class *sgn* =
fixes *sgn* :: 'a \Rightarrow 'a

class *ordered-ab-group-add-abs* = *ordered-ab-group-add* + *abs* +
assumes *abs-ge-zero* [*simp*]: $|a| \geq 0$
and *abs-ge-self*: $a \leq |a|$
and *abs-leI*: $a \leq b \implies -a \leq b \implies |a| \leq b$
and *abs-minus-cancel* [*simp*]: $|-a| = |a|$
and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$
begin

lemma *abs-minus-le-zero*: $-|a| \leq 0$
unfolding *neg-le-0-iff-le* **by** *simp*

lemma *abs-of-nonneg* [*simp*]:
assumes *nonneg*: $0 \leq a$
shows $|a| = a$
proof (*rule order.antisym*)
show $a \leq |a|$ **by** (*rule abs-ge-self*)
from *nonneg le-imp-neg-le* **have** $-a \leq 0$ **by** *simp*
from *this nonneg* **have** $-a \leq a$ **by** (*rule order-trans*)

then show $|a| \leq a$ **by** (*auto intro: abs-leI*)
qed

lemma *abs-idempotent* [*simp*]: $||a|| = |a|$
by (*rule order.antisym*) (*auto intro!: abs-ge-self abs-leI order-trans [of - |a| 0 |a|]*)

lemma *abs-eq-0* [*simp*]: $|a| = 0 \longleftrightarrow a = 0$

proof –
have $|a| = 0 \implies a = 0$
proof (*rule order.antisym*)
assume *zero*: $|a| = 0$
with *abs-ge-self* **show** $a \leq 0$ **by** *auto*
from *zero* **have** $|-a| = 0$ **by** *simp*
with *abs-ge-self* [*of - a*] **have** $-a \leq 0$ **by** *auto*
with *neg-le-0-iff-le* **show** $0 \leq a$ **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *abs-zero* [*simp*]: $|0| = 0$
by *simp*

lemma *abs-0-eq* [*simp*]: $0 = |a| \longleftrightarrow a = 0$
proof –
have $0 = |a| \longleftrightarrow |a| = 0$ **by** (*simp only: eq-acc*)
then show *?thesis* **by** *simp*
qed

lemma *abs-le-zero-iff* [*simp*]: $|a| \leq 0 \longleftrightarrow a = 0$
proof
assume $|a| \leq 0$
then have $|a| = 0$ **by** (*rule order.antisym*) *simp*
then show $a = 0$ **by** *simp*
next
assume $a = 0$
then show $|a| \leq 0$ **by** *simp*
qed

lemma *abs-le-self-iff* [*simp*]: $|a| \leq a \longleftrightarrow 0 \leq a$
proof –
have $0 \leq |a|$
using *abs-ge-zero* **by** *blast*
then have $|a| \leq a \implies 0 \leq a$
using *order.trans* **by** *blast*
then show *?thesis*
using *abs-of-nonneg eq-refl* **by** *blast*
qed

lemma *zero-less-abs-iff* [simp]: $0 < |a| \longleftrightarrow a \neq 0$
 by (*simp add: less-le*)

lemma *abs-not-less-zero* [simp]: $\neg |a| < 0$
proof –
 have $x \leq y \implies \neg y < x$ for $x y$ by *auto*
 then show *?thesis* by *simp*
qed

lemma *abs-ge-minus-self*: $- a \leq |a|$
proof –
 have $- a \leq |-a|$ by (*rule abs-ge-self*)
 then show *?thesis* by *simp*
qed

lemma *abs-minus-commute*: $|a - b| = |b - a|$
proof –
 have $|a - b| = |-(a - b)|$
 by (*simp only: abs-minus-cancel*)
 also have $\dots = |b - a|$ by *simp*
 finally show *?thesis* .
qed

lemma *abs-of-pos*: $0 < a \implies |a| = a$
 by (*rule abs-of-nonneg*) (*rule less-imp-le*)

lemma *abs-of-nonpos* [simp]:
 assumes $a \leq 0$
 shows $|a| = - a$
proof –
 let $?b = - a$
 have $- ?b \leq 0 \implies |- ?b| = - (- ?b)$
 unfolding *abs-minus-cancel* [of $?b$]
 unfolding *neg-le-0-iff-le* [of $?b$]
 unfolding *minus-minus* by (*erule abs-of-nonneg*)
 then show *?thesis* using *assms* by *auto*
qed

lemma *abs-of-neg*: $a < 0 \implies |a| = - a$
 by (*rule abs-of-nonpos*) (*rule less-imp-le*)

lemma *abs-le-D1*: $|a| \leq b \implies a \leq b$
 using *abs-ge-self* by (*blast intro: order-trans*)

lemma *abs-le-D2*: $|a| \leq b \implies - a \leq b$
 using *abs-le-D1* [of $- a$] by *simp*

lemma *abs-le-iff*: $|a| \leq b \longleftrightarrow a \leq b \wedge - a \leq b$
 by (*blast intro: abs-leI dest: abs-le-D1 abs-le-D2*)

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$

proof –

have $|a| = |b + (a - b)|$
 by (*simp add: algebra-simps*)
 then have $|a| \leq |b| + |a - b|$
 by (*simp add: abs-triangle-ineq*)
 then show *?thesis*
 by (*simp add: algebra-simps*)

qed

lemma *abs-triangle-ineq2-sym*: $|a| - |b| \leq |b - a|$

by (*simp only: abs-minus-commute [of b] abs-triangle-ineq2*)

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$

by (*simp add: abs-le-iff abs-triangle-ineq2 abs-triangle-ineq2-sym*)

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$

proof –

have $|a - b| = |a + - b|$
 by (*simp add: algebra-simps*)
 also have $\dots \leq |a| + |- b|$
 by (*rule abs-triangle-ineq*)
 finally show *?thesis* **by** *simp*

qed

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$

proof –

have $|a + b - (c + d)| = |(a - c) + (b - d)|$
 by (*simp add: algebra-simps*)
 also have $\dots \leq |a - c| + |b - d|$
 by (*rule abs-triangle-ineq*)
 finally show *?thesis* .

qed

lemma *abs-add-abs [simp]*: $||a| + |b|| = |a| + |b|$

(*is ?L = ?R*)

proof (*rule order.antisym*)

show $?L \geq ?R$ **by** (*rule abs-ge-self*)
 have $?L \leq ||a| + |b||$ **by** (*rule abs-triangle-ineq*)
 also have $\dots = ?R$ **by** *simp*
 finally show $?L \leq ?R$.

qed

end

lemma *dense-eq0-I*:

fixes $x::'a::\{dense-linorder, ordered-ab-group-add-abs\}$

assumes $\bigwedge e. 0 < e \implies |x| \leq e$

```

shows  $x = 0$ 
proof (cases  $|x| = 0$ )
  case False
  then have  $|x| > 0$ 
    by simp
  then obtain  $z$  where  $0 < z & z < |x|$ 
    using dense by force
  then show ?thesis
    using assms by (simp flip: not-less)
qed auto

```

```

hide-fact (open) ab-diff-conv-add-uminus add-0 mult-1 ab-left-minus

```

```

lemmas add-0 = add-0-left
lemmas mult-1 = mult-1-left
lemmas ab-left-minus = left-minus
lemmas diff-diff-eq = diff-diff-add

```

5.8 Canonically ordered monoids

Canonically ordered monoids are never groups.

```

class canonically-ordered-monoid-add = comm-monoid-add + order +
  assumes le-iff-add:  $a \leq b \longleftrightarrow (\exists c. b = a + c)$ 
begin

```

```

lemma zero-le[simp]:  $0 \leq x$ 
  by (auto simp: le-iff-add)

```

```

lemma le-zero-eq[simp]:  $n \leq 0 \longleftrightarrow n = 0$ 
  by (auto intro: order.antisym)

```

```

lemma not-less-zero[simp]:  $\neg n < 0$ 
  by (auto simp: less-le)

```

```

lemma zero-less-iff-neq-zero:  $0 < n \longleftrightarrow n \neq 0$ 
  by (auto simp: less-le)

```

This theorem is useful with *blast*

```

lemma gr-zeroI:  $(n = 0 \implies \text{False}) \implies 0 < n$ 
  by (rule zero-less-iff-neq-zero[THEN iffD2] iprover)

```

```

lemma not-gr-zero[simp]:  $\neg 0 < n \longleftrightarrow n = 0$ 
  by (simp add: zero-less-iff-neq-zero)

```

```

subclass ordered-comm-monoid-add
  proof qed (auto simp: le-iff-add add-ac)

```

```

lemma gr-implies-not-zero:  $m < n \implies n \neq 0$ 
  by auto

```

lemma *add-eq-0-iff-both-eq-0[simp]*: $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*intro add-nonneg-eq-0-iff zero-le*)

lemma *zero-eq-add-iff-both-eq-0[simp]*: $0 = x + y \longleftrightarrow x = 0 \wedge y = 0$
using *add-eq-0-iff-both-eq-0[of x y]* **unfolding** *eq-commute[of 0]* .

lemma *less-eqE*:
assumes $\langle a \leq b \rangle$
obtains *c* **where** $\langle b = a + c \rangle$
using *assms* **by** (*auto simp add: le-iff-add*)

lemma *lessE*:
assumes $\langle a < b \rangle$
obtains *c* **where** $\langle b = a + c \rangle$ **and** $\langle c \neq 0 \rangle$
proof –
from *assms* **have** $\langle a \leq b \rangle \langle a \neq b \rangle$
by *simp-all*
from $\langle a \leq b \rangle$ **obtain** *c* **where** $\langle b = a + c \rangle$
by (*rule less-eqE*)
moreover **have** $\langle c \neq 0 \rangle$ **using** $\langle a \neq b \rangle \langle b = a + c \rangle$
by *auto*
ultimately show *?thesis*
by (*rule that*)

qed

lemmas *zero-order = zero-le le-zero-eq not-less-zero zero-less-iff-neq-zero not-gr-zero*
 — This should be attributed with [*iff*], but then *blast* fails in *Set*.

end

class *ordered-cancel-comm-monoid-diff* =
canonically-ordered-monoid-add + comm-monoid-diff + ordered-ab-semigroup-add-imp-le
begin

context
fixes $a\ b :: 'a$
assumes $le: a \leq b$
begin

lemma *add-diff-inverse*: $a + (b - a) = b$
using *le* **by** (*auto simp add: le-iff-add*)

lemma *add-diff-assoc*: $c + (b - a) = c + b - a$
using *le* **by** (*auto simp add: le-iff-add add.left-commute [of c]*)

lemma *add-diff-assoc2*: $b - a + c = b + c - a$
using *le* **by** (*auto simp add: le-iff-add add.assoc*)


```

lemma diff-add-assoc:  $c + b - a = c + (b - a)$ 
  using le by (simp add: add.commute add-diff-assoc)

lemma diff-add-assoc2:  $b + c - a = b - a + c$ 
  using le by (simp add: add.commute add-diff-assoc)

lemma diff-diff-right:  $c - (b - a) = c + a - b$ 
  by (simp add: add-diff-inverse add-diff-cancel-left [of a c b - a, symmetric]
add.commute)

lemma diff-add:  $b - a + a = b$ 
  by (simp add: add.commute add-diff-inverse)

lemma le-add-diff:  $c \leq b + c - a$ 
  by (auto simp add: add.commute diff-add-assoc2 le-iff-add)

lemma le-imp-diff-is-add:  $a \leq b \implies b - a = c \iff b = c + a$ 
  by (auto simp add: add.commute add-diff-inverse)

lemma le-diff-conv2:  $c \leq b - a \iff c + a \leq b$ 
  (is ?P \iff ?Q)
proof
  assume ?P
  then have  $c + a \leq b - a + a$ 
    by (rule add-right-mono)
  then show ?Q
    by (simp add: add-diff-inverse add.commute)
next
  assume ?Q
  then have  $a + c \leq a + (b - a)$ 
    by (simp add: add-diff-inverse add.commute)
  then show ?P by simp
qed

end

end

```

5.9 Tools setup

```

lemma add-mono-thms-linordered-semiring:
  fixes i j k :: 'a::ordered-ab-semigroup-add
  shows  $i \leq j \wedge k \leq l \implies i + k \leq j + l$ 
    and  $i = j \wedge k \leq l \implies i + k \leq j + l$ 
    and  $i \leq j \wedge k = l \implies i + k \leq j + l$ 
    and  $i = j \wedge k = l \implies i + k = j + l$ 
  by (rule add-mono, clarify)+

```

```

lemma add-mono-thms-linordered-field:

```

```

fixes  $i\ j\ k :: 'a::\text{ordered-cancel-ab-semigroup-add}$ 
shows  $i < j \wedge k = l \implies i + k < j + l$ 
  and  $i = j \wedge k < l \implies i + k < j + l$ 
  and  $i < j \wedge k \leq l \implies i + k < j + l$ 
  and  $i \leq j \wedge k < l \implies i + k < j + l$ 
  and  $i < j \wedge k < l \implies i + k < j + l$ 
by (auto intro: add-strict-right-mono add-strict-left-mono
      add-less-le-mono add-le-less-mono add-strict-mono)

code-identifier
code-module Groups  $\rightarrow$  (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

```

6 Abstract lattices

```

theory Lattices
imports Groups
begin

```

6.1 Abstract semilattice

These locales provide a basic structure for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

```

locale semilattice = abel-semigroup +
  assumes idem [simp]:  $a * a = a$ 
begin

```

```

lemma left-idem [simp]:  $a * (a * b) = a * b$ 
  by (simp add: assoc [symmetric])

```

```

lemma right-idem [simp]:  $(a * b) * b = a * b$ 
  by (simp add: assoc)

```

```

end

```

```

locale semilattice-neutr = semilattice + comm-monoid

```

```

locale semilattice-order = semilattice +
  fixes less-eq ::  $'a \Rightarrow 'a \Rightarrow \text{bool}$  (infix  $\leq$  50)
  and less ::  $'a \Rightarrow 'a \Rightarrow \text{bool}$  (infix  $<$  50)
  assumes order-iff:  $a \leq b \longleftrightarrow a = a * b$ 
  and strict-order-iff:  $a < b \longleftrightarrow a = a * b \wedge a \neq b$ 
begin

```

```

lemma orderI:  $a = a * b \implies a \leq b$ 
  by (simp add: order-iff)

```

```

lemma orderE:
  assumes  $a \leq b$ 
  obtains  $a = a * b$ 
  using assms by (unfold order-iff)

sublocale ordering less-eq less
proof
  show  $a < b \longleftrightarrow a \leq b \wedge a \neq b$  for  $a\ b$ 
    by (simp add: order-iff strict-order-iff)
next
  show  $a \leq a$  for  $a$ 
    by (simp add: order-iff)
next
  fix  $a\ b$ 
  assume  $a \leq b\ b \leq a$ 
  then have  $a = a * b\ a * b = b$ 
    by (simp-all add: order-iff commute)
  then show  $a = b$  by simp
next
  fix  $a\ b\ c$ 
  assume  $a \leq b\ b \leq c$ 
  then have  $a = a * b\ b = b * c$ 
    by (simp-all add: order-iff commute)
  then have  $a = a * (b * c)$ 
    by simp
  then have  $a = (a * b) * c$ 
    by (simp add: assoc)
  with  $\langle a = a * b \rangle$  [symmetric] have  $a = a * c$  by simp
  then show  $a \leq c$  by (rule orderI)
qed

lemma cobounded1 [simp]:  $a * b \leq a$ 
  by (simp add: order-iff commute)

lemma cobounded2 [simp]:  $a * b \leq b$ 
  by (simp add: order-iff)

lemma boundedI:
  assumes  $a \leq b$  and  $a \leq c$ 
  shows  $a \leq b * c$ 
proof (rule orderI)
  from assms obtain  $a * b = a$  and  $a * c = a$ 
    by (auto elim!: orderE)
  then show  $a = a * (b * c)$ 
    by (simp add: assoc [symmetric])
qed

lemma boundedE:

```

```

assumes  $a \leq b * c$ 
obtains  $a \leq b$  and  $a \leq c$ 
using assms by (blast intro: trans cobounded1 cobounded2)

lemma bounded-iff [simp]:  $a \leq b * c \longleftrightarrow a \leq b \wedge a \leq c$ 
by (blast intro: boundedI elim: boundedE)

lemma strict-boundedE:
assumes  $a < b * c$ 
obtains  $a < b$  and  $a < c$ 
using assms by (auto simp add: commute strict-iff-order elim: orderE intro!: that)+

lemma coboundedI1:  $a \leq c \implies a * b \leq c$ 
by (rule trans) auto

lemma coboundedI2:  $b \leq c \implies a * b \leq c$ 
by (rule trans) auto

lemma strict-coboundedI1:  $a < c \implies a * b < c$ 
using irrefl
by (auto intro: not-eq-order-implies-strict coboundedI1 strict-implies-order elim: strict-boundedE)

lemma strict-coboundedI2:  $b < c \implies a * b < c$ 
using strict-coboundedI1 [of b c a] by (simp add: commute)

lemma mono:  $a \leq c \implies b \leq d \implies a * b \leq c * d$ 
by (blast intro: boundedI coboundedI1 coboundedI2)

lemma absorb1:  $a \leq b \implies a * b = a$ 
by (rule antisym) (auto simp: refl)

lemma absorb2:  $b \leq a \implies a * b = b$ 
by (rule antisym) (auto simp: refl)

lemma absorb3:  $a < b \implies a * b = a$ 
by (rule absorb1) (rule strict-implies-order)

lemma absorb4:  $b < a \implies a * b = b$ 
by (rule absorb2) (rule strict-implies-order)

lemma absorb-iff1:  $a \leq b \longleftrightarrow a * b = a$ 
using order-iff by auto

lemma absorb-iff2:  $b \leq a \longleftrightarrow a * b = b$ 
using order-iff by (auto simp add: commute)

end

```

locale *semilattice-neutr-order* = *semilattice-neutr* + *semilattice-order*
begin

sublocale *ordering-top less-eq less 1*
by *standard (simp add: order-iff)*

lemma *eq-neutr-iff [simp]*: $\langle a * b = \mathbf{1} \longleftrightarrow a = \mathbf{1} \wedge b = \mathbf{1} \rangle$
by *(simp add: eq-iff)*

lemma *neutr-eq-iff [simp]*: $\langle \mathbf{1} = a * b \longleftrightarrow a = \mathbf{1} \wedge b = \mathbf{1} \rangle$
by *(simp add: eq-iff)*

end

Interpretations for boolean operators

interpretation *conj: semilattice-neutr* $\langle (\wedge) \rangle$ *True*
by *standard auto*

interpretation *disj: semilattice-neutr* $\langle (\vee) \rangle$ *False*
by *standard auto*

declare *conj-assoc [ac-simps del] disj-assoc [ac-simps del]* — already simp by default

6.2 Syntactic infimum and supremum operations

class *inf* =
fixes *inf* :: 'a \Rightarrow 'a \Rightarrow 'a (**infixl** \sqcap 70)

class *sup* =
fixes *sup* :: 'a \Rightarrow 'a \Rightarrow 'a (**infixl** \sqcup 65)

6.3 Concrete lattices

class *semilattice-inf* = *order* + *inf* +
assumes *inf-le1 [simp]*: $x \sqcap y \leq x$
and *inf-le2 [simp]*: $x \sqcap y \leq y$
and *inf-greatest*: $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$

class *semilattice-sup* = *order* + *sup* +
assumes *sup-ge1 [simp]*: $x \leq x \sqcup y$
and *sup-ge2 [simp]*: $y \leq x \sqcup y$
and *sup-least*: $y \leq x \Longrightarrow z \leq x \Longrightarrow y \sqcup z \leq x$
begin

Dual lattice.

lemma *dual-semilattice: class.semilattice-inf sup greater-eq greater*
by *(rule class.semilattice-inf.intro, rule dual-order)*

(*unfold-locales, simp-all add: sup-least*)

end

class *lattice* = *semilattice-inf* + *semilattice-sup*

6.3.1 Intro and elim rules

context *semilattice-inf*

begin

lemma *le-infI1*: $a \leq x \implies a \sqcap b \leq x$
by (*rule order-trans*) *auto*

lemma *le-infI2*: $b \leq x \implies a \sqcap b \leq x$
by (*rule order-trans*) *auto*

lemma *le-infI*: $x \leq a \implies x \leq b \implies x \leq a \sqcap b$
by (*fact inf-greatest*)

lemma *le-infE*: $x \leq a \sqcap b \implies (x \leq a \implies x \leq b \implies P) \implies P$
by (*blast intro: order-trans inf-le1 inf-le2*)

lemma *le-inf-iff*: $x \leq y \sqcap z \iff x \leq y \wedge x \leq z$
by (*blast intro: le-infI elim: le-infE*)

lemma *le-iff-inf*: $x \leq y \iff x \sqcap y = x$
by (*auto intro: le-infI1 order.antisym dest: order.eq-iff [THEN iffD1] simp add: le-inf-iff*)

lemma *inf-mono*: $a \leq c \implies b \leq d \implies a \sqcap b \leq c \sqcap d$
by (*fast intro: inf-greatest le-infI1 le-infI2*)

end

context *semilattice-sup*

begin

lemma *le-supI1*: $x \leq a \implies x \leq a \sqcup b$
by (*rule order-trans*) *auto*

lemma *le-supI2*: $x \leq b \implies x \leq a \sqcup b$
by (*rule order-trans*) *auto*

lemma *le-supI*: $a \leq x \implies b \leq x \implies a \sqcup b \leq x$
by (*fact sup-least*)

lemma *le-supE*: $a \sqcup b \leq x \implies (a \leq x \implies b \leq x \implies P) \implies P$
by (*blast intro: order-trans sup-ge1 sup-ge2*)

lemma *le-sup-iff*: $x \sqcup y \leq z \iff x \leq z \wedge y \leq z$

by (*blast intro: le-supI elim: le-supE*)

lemma *le-iff-sup*: $x \leq y \iff x \sqcup y = y$

by (*auto intro: le-supI2 order.antisym dest: order.eq-iff [THEN iffD1] simp add: le-sup-iff*)

lemma *sup-mono*: $a \leq c \implies b \leq d \implies a \sqcup b \leq c \sqcup d$

by (*fast intro: sup-least le-supI1 le-supI2*)

end

6.3.2 Equational laws

context *semilattice-inf*

begin

sublocale *inf: semilattice inf*

proof

fix $a b c$

show $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$

by (*rule order.antisym*) (*auto intro: le-infI1 le-infI2 simp add: le-inf-iff*)

show $a \sqcap b = b \sqcap a$

by (*rule order.antisym*) (*auto simp add: le-inf-iff*)

show $a \sqcap a = a$

by (*rule order.antisym*) (*auto simp add: le-inf-iff*)

qed

sublocale *inf: semilattice-order inf less-eq less*

by *standard* (*auto simp add: le-iff-inf less-le*)

lemma *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$

by (*fact inf.assoc*)

lemma *inf-commute*: $(x \sqcap y) = (y \sqcap x)$

by (*fact inf.commute*)

lemma *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$

by (*fact inf.left-commute*)

lemma *inf-idem*: $x \sqcap x = x$

by (*fact inf.idem*)

lemma *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$

by (*fact inf.left-idem*)

lemma *inf-right-idem*: $(x \sqcap y) \sqcap y = x \sqcap y$

by (*fact inf.right-idem*)

```

lemma inf-absorb1:  $x \leq y \implies x \sqcap y = x$ 
  by (rule order.antisym) auto

lemma inf-absorb2:  $y \leq x \implies x \sqcap y = y$ 
  by (rule order.antisym) auto

lemmas inf-aci = inf-commute inf-assoc inf-left-commute inf-left-idem

end

context semilattice-sup
begin

sublocale sup: semilattice sup
proof
  fix a b c
  show  $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$ 
    by (rule order.antisym) (auto intro: le-supI1 le-supI2 simp add: le-sup-iff)
  show  $a \sqcup b = b \sqcup a$ 
    by (rule order.antisym) (auto simp add: le-sup-iff)
  show  $a \sqcup a = a$ 
    by (rule order.antisym) (auto simp add: le-sup-iff)
qed

sublocale sup: semilattice-order sup greater-eq greater
  by standard (auto simp add: le-iff-sup sup.commute less-le)

lemma sup-assoc:  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ 
  by (fact sup.assoc)

lemma sup-commute:  $(x \sqcup y) = (y \sqcup x)$ 
  by (fact sup.commute)

lemma sup-left-commute:  $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$ 
  by (fact sup.left-commute)

lemma sup-idem:  $x \sqcup x = x$ 
  by (fact sup.idem)

lemma sup-left-idem [simp]:  $x \sqcup (x \sqcup y) = x \sqcup y$ 
  by (fact sup.left-idem)

lemma sup-absorb1:  $y \leq x \implies x \sqcup y = x$ 
  by (rule order.antisym) auto

lemma sup-absorb2:  $x \leq y \implies x \sqcup y = y$ 
  by (rule order.antisym) auto

```


lemmas *sup-aci* = *sup-commute sup-assoc sup-left-commute sup-left-idem*

end

context *lattice*

begin

lemma *dual-lattice*: *class.lattice sup* (\geq) ($>$) *inf*

by (*rule class.lattice.intro,*
rule dual-semilattice,
rule class.semilattice-sup.intro,
rule dual-order)
(unfold-locales, auto)

lemma *inf-sup-absorb* [*simp*]: $x \sqcap (x \sqcup y) = x$

by (*blast intro: order.antisym inf-le1 inf-greatest sup-ge1*)

lemma *sup-inf-absorb* [*simp*]: $x \sqcup (x \sqcap y) = x$

by (*blast intro: order.antisym sup-ge1 sup-least inf-le1*)

lemmas *inf-sup-aci* = *inf-aci sup-aci*

lemmas *inf-sup-ord* = *inf-le1 inf-le2 sup-ge1 sup-ge2*

Towards distributivity.

lemma *distrib-sup-le*: $x \sqcup (y \sqcap z) \leq (x \sqcup y) \sqcap (x \sqcup z)$

by (*auto intro: le-infI1 le-infI2 le-supI1 le-supI2*)

lemma *distrib-inf-le*: $(x \sqcap y) \sqcup (x \sqcap z) \leq x \sqcap (y \sqcup z)$

by (*auto intro: le-infI1 le-infI2 le-supI1 le-supI2*)

If you have one of them, you have them all.

lemma *distrib-imp1*:

assumes *distrib*: $\bigwedge x y z. x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

shows $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

proof –

have $x \sqcup (y \sqcap z) = (x \sqcup (x \sqcap z)) \sqcup (y \sqcap z)$

by *simp*

also have $\dots = x \sqcup (z \sqcap (x \sqcup y))$

by (*simp add: distrib inf-commute sup-assoc del: sup-inf-absorb*)

also have $\dots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$

by (*simp add: inf-commute*)

also have $\dots = (x \sqcup y) \sqcap (x \sqcup z)$ **by** (*simp add: distrib*)

finally show *?thesis* .

qed

lemma *distrib-imp2*:

assumes *distrib*: $\bigwedge x y z. x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

proof–
have $x \sqcap (y \sqcup z) = (x \sqcap (x \sqcup z)) \sqcap (y \sqcup z)$
by *simp*
also have $\dots = x \sqcap (z \sqcup (x \sqcap y))$
by (*simp add: distrib sup-commute inf-assoc del: inf-sup-absorb*)
also have $\dots = ((x \sqcap y) \sqcup x) \sqcap ((x \sqcap y) \sqcup z)$
by (*simp add: sup-commute*)
also have $\dots = (x \sqcap y) \sqcup (x \sqcap z)$ **by** (*simp add:distrib*)
finally show *?thesis* .
qed
end

6.3.3 Strict order

context *semilattice-inf*
begin

lemma *less-infI1*: $a < x \implies a \sqcap b < x$
by (*auto simp add: less-le inf-absorb1 intro: le-infI1*)

lemma *less-infI2*: $b < x \implies a \sqcap b < x$
by (*auto simp add: less-le inf-absorb2 intro: le-infI2*)

end

context *semilattice-sup*
begin

lemma *less-supI1*: $x < a \implies x < a \sqcup b$
using *dual-semilattice*
by (*rule semilattice-inf.less-infI1*)

lemma *less-supI2*: $x < b \implies x < a \sqcup b$
using *dual-semilattice*
by (*rule semilattice-inf.less-infI2*)

end

6.4 Distributive lattices

class *distrib-lattice* = *lattice* +
assumes *sup-inf-distrib1*: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

context *distrib-lattice*
begin

lemma *sup-inf-distrib2*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by (*simp add: sup-commute sup-inf-distrib1*)

lemma *inf-sup-distrib1*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
by (*rule distrib-imp2 [OF sup-inf-distrib1]*)

lemma *inf-sup-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
by (*simp add: inf-commute inf-sup-distrib1*)

lemma *dual-distrib-lattice*: *class.distrib-lattice sup* (\geq) ($>$) *inf*
by (*rule class.distrib-lattice.intro, rule dual-lattice*)
(*unfold-locales, fact inf-sup-distrib1*)

lemmas *sup-inf-distrib* = *sup-inf-distrib1 sup-inf-distrib2*

lemmas *inf-sup-distrib* = *inf-sup-distrib1 inf-sup-distrib2*

lemmas *distrib* = *sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2*

end

6.5 Bounded lattices

class *bounded-semilattice-inf-top* = *semilattice-inf* + *order-top*
begin

sublocale *inf-top*: *semilattice-neutr inf top*
+ *inf-top: semilattice-neutr-order inf top less-eq less*
proof

show $x \sqcap \top = x$ **for** x
by (*rule inf-absorb1*) *simp*
qed

lemma *inf-top-left*: $\top \sqcap x = x$
by (*fact inf-top.left-neutral*)

lemma *inf-top-right*: $x \sqcap \top = x$
by (*fact inf-top.right-neutral*)

lemma *inf-eq-top-iff*: $x \sqcap y = \top \iff x = \top \wedge y = \top$
by (*fact inf-top.eq-neutr-iff*)

lemma *top-eq-inf-iff*: $\top = x \sqcap y \iff x = \top \wedge y = \top$
by (*fact inf-top.neutr-eq-iff*)

end

class *bounded-semilattice-sup-bot* = *semilattice-sup* + *order-bot*
begin

sublocale *sup-bot*: *semilattice-neutr sup bot*
+ *sup-bot: semilattice-neutr-order sup bot greater-eq greater*

proof

show $x \sqcup \perp = x$ **for** x
by (rule *sup-absorb1*) *simp*

qed

lemma *sup-bot-left*: $\perp \sqcup x = x$
by (fact *sup-bot.left-neutral*)

lemma *sup-bot-right*: $x \sqcup \perp = x$
by (fact *sup-bot.right-neutral*)

lemma *sup-eq-bot-iff*: $x \sqcup y = \perp \longleftrightarrow x = \perp \wedge y = \perp$
by (fact *sup-bot.eq-neutr-iff*)

lemma *bot-eq-sup-iff*: $\perp = x \sqcup y \longleftrightarrow x = \perp \wedge y = \perp$
by (fact *sup-bot.neutr-eq-iff*)

end

class *bounded-lattice-bot* = *lattice* + *order-bot*
begin

subclass *bounded-semilattice-sup-bot* ..

lemma *inf-bot-left* [*simp*]: $\perp \sqcap x = \perp$
by (rule *inf-absorb1*) *simp*

lemma *inf-bot-right* [*simp*]: $x \sqcap \perp = \perp$
by (rule *inf-absorb2*) *simp*

end

class *bounded-lattice-top* = *lattice* + *order-top*
begin

subclass *bounded-semilattice-inf-top* ..

lemma *sup-top-left* [*simp*]: $\top \sqcup x = \top$
by (rule *sup-absorb1*) *simp*

lemma *sup-top-right* [*simp*]: $x \sqcup \top = \top$
by (rule *sup-absorb2*) *simp*

end

class *bounded-lattice* = *lattice* + *order-bot* + *order-top*
begin

subclass *bounded-lattice-bot* ..

subclass *bounded-lattice-top* ..

lemma *dual-bounded-lattice*: *class.bounded-lattice sup greater-eq greater inf $\top \perp$*
by *unfold-locales (auto simp add: less-le-not-le)*

end

6.6 *min/max* as special case of lattice

context *linorder*

begin

sublocale *min*: *semilattice-order min less-eq less*
 + *max*: *semilattice-order max greater-eq greater*
by *standard (auto simp add: min-def max-def)*

declare *min.absorb1* [*simp*] *min.absorb2* [*simp*]
min.absorb3 [*simp*] *min.absorb4* [*simp*]
max.absorb1 [*simp*] *max.absorb2* [*simp*]
max.absorb3 [*simp*] *max.absorb4* [*simp*]

lemma *min-le-iff-disj*: $\min x y \leq z \longleftrightarrow x \leq z \vee y \leq z$
unfolding *min-def* **using** *linear* **by** (*auto intro: order-trans*)

lemma *le-max-iff-disj*: $z \leq \max x y \longleftrightarrow z \leq x \vee z \leq y$
unfolding *max-def* **using** *linear* **by** (*auto intro: order-trans*)

lemma *min-less-iff-disj*: $\min x y < z \longleftrightarrow x < z \vee y < z$
unfolding *min-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

lemma *less-max-iff-disj*: $z < \max x y \longleftrightarrow z < x \vee z < y$
unfolding *max-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

lemma *min-less-iff-conj* [*simp*]: $z < \min x y \longleftrightarrow z < x \wedge z < y$
unfolding *min-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

lemma *max-less-iff-conj* [*simp*]: $\max x y < z \longleftrightarrow x < z \wedge y < z$
unfolding *max-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

lemma *min-max-distrib1*: $\min (\max b c) a = \max (\min b a) (\min c a)$
by (*auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym*)

lemma *min-max-distrib2*: $\min a (\max b c) = \max (\min a b) (\min a c)$
by (*auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym*)

lemma *max-min-distrib1*: $\max (\min b c) a = \min (\max b a) (\max c a)$
by (*auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro:*

antisym)

lemma *max-min-distrib2*: $\max a (\min b c) = \min (\max a b) (\max a c)$

by (*auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym*)

lemmas *min-max-distrib1 = min-max-distrib1 min-max-distrib2 max-min-distrib1 max-min-distrib2*

lemma *split-min* [*no-atp*]: $P (\min i j) \longleftrightarrow (i \leq j \longrightarrow P i) \wedge (\neg i \leq j \longrightarrow P j)$

by (*simp add: min-def*)

lemma *split-max* [*no-atp*]: $P (\max i j) \longleftrightarrow (i \leq j \longrightarrow P j) \wedge (\neg i \leq j \longrightarrow P i)$

by (*simp add: max-def*)

lemma *split-min-lin* [*no-atp*]:

$\langle P (\min a b) \longleftrightarrow (b = a \longrightarrow P a) \wedge (a < b \longrightarrow P a) \wedge (b < a \longrightarrow P b) \rangle$

by (*cases a b rule: linorder-cases*) *auto*

lemma *split-max-lin* [*no-atp*]:

$\langle P (\max a b) \longleftrightarrow (b = a \longrightarrow P a) \wedge (a < b \longrightarrow P b) \wedge (b < a \longrightarrow P a) \rangle$

by (*cases a b rule: linorder-cases*) *auto*

end

lemma *inf-min*: $\text{inf} = (\text{min} :: 'a::\{\text{semilattice-inf}, \text{linorder}\} \Rightarrow 'a \Rightarrow 'a)$

by (*auto intro: antisym simp add: min-def fun-eq-iff*)

lemma *sup-max*: $\text{sup} = (\text{max} :: 'a::\{\text{semilattice-sup}, \text{linorder}\} \Rightarrow 'a \Rightarrow 'a)$

by (*auto intro: antisym simp add: max-def fun-eq-iff*)

6.7 Uniqueness of inf and sup

lemma (*in semilattice-inf*) *inf-unique*:

fixes *f* (*infixl* Δ 70)

assumes *le1*: $\bigwedge x y. x \Delta y \leq x$

and *le2*: $\bigwedge x y. x \Delta y \leq y$

and *greatest*: $\bigwedge x y z. x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \Delta z$

shows $x \sqcap y = x \Delta y$

proof (*rule order.antisym*)

show $x \Delta y \leq x \sqcap y$

by (*rule le-infI*) (*rule le1, rule le2*)

have *leI*: $\bigwedge x y z. x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \Delta z$

by (*blast intro: greatest*)

show $x \sqcap y \leq x \Delta y$

by (*rule leI*) *simp-all*

qed

lemma (*in semilattice-sup*) *sup-unique*:

```

fixes f (infixl  $\nabla$  70)
assumes ge1 [simp]:  $\bigwedge x y. x \leq x \nabla y$ 
  and ge2:  $\bigwedge x y. y \leq x \nabla y$ 
  and least:  $\bigwedge x y z. y \leq x \implies z \leq x \implies y \nabla z \leq x$ 
shows  $x \sqcup y = x \nabla y$ 
proof (rule order.antisym)
  show  $x \sqcup y \leq x \nabla y$ 
    by (rule le-supI) (rule ge1, rule ge2)
  have leI:  $\bigwedge x y z. x \leq z \implies y \leq z \implies x \nabla y \leq z$ 
    by (blast intro: least)
  show  $x \nabla y \leq x \sqcup y$ 
    by (rule leI) simp-all
qed

```

6.8 Lattice on $- \Rightarrow -$

```

instantiation fun :: (type, semilattice-sup) semilattice-sup
begin

```

```

definition  $f \sqcup g = (\lambda x. f x \sqcup g x)$ 

```

```

lemma sup-apply [simp, code]:  $(f \sqcup g) x = f x \sqcup g x$ 
  by (simp add: sup-fun-def)

```

```

instance
  by standard (simp-all add: le-fun-def)

```

```

end

```

```

instantiation fun :: (type, semilattice-inf) semilattice-inf
begin

```

```

definition  $f \sqcap g = (\lambda x. f x \sqcap g x)$ 

```

```

lemma inf-apply [simp, code]:  $(f \sqcap g) x = f x \sqcap g x$ 
  by (simp add: inf-fun-def)

```

```

instance by standard (simp-all add: le-fun-def)

```

```

end

```

```

instance fun :: (type, lattice) lattice ..

```

```

instance fun :: (type, distrib-lattice) distrib-lattice
  by standard (rule ext, simp add: sup-inf-distrib1)

```

```

instance fun :: (type, bounded-lattice) bounded-lattice ..

```

```

instantiation fun :: (type, uminus) uminus

```

begin

definition *fun-Compl-def*: $- A = (\lambda x. - A x)$

lemma *uminus-apply* [*simp*, *code*]: $(- A) x = - (A x)$
by (*simp add: fun-Compl-def*)

instance ..

end

instantiation *fun* :: (*type*, *minus*) *minus*
begin

definition *fun-diff-def*: $A - B = (\lambda x. A x - B x)$

lemma *minus-apply* [*simp*, *code*]: $(A - B) x = A x - B x$
by (*simp add: fun-diff-def*)

instance ..

end

end

7 Boolean Algebras

theory *Boolean-Algebras*
imports *Lattices*
begin

7.1 Abstract boolean algebra

locale *abstract-boolean-algebra* = *conj: abel-semigroup* $\langle(\sqcap)\rangle$ + *disj: abel-semigroup* $\langle(\sqcup)\rangle$

for *conj* :: $\langle'a \Rightarrow 'a \Rightarrow 'a\rangle$ (**infixr** $\langle(\sqcap)\rangle$ 70)
and *disj* :: $\langle'a \Rightarrow 'a \Rightarrow 'a\rangle$ (**infixr** $\langle(\sqcup)\rangle$ 65) +
fixes *compl* :: $\langle'a \Rightarrow 'a\rangle$ ($\langle-\ \rightarrow$ [81] 80)
and *zero* :: $\langle'a\rangle$ ($\langle\mathbf{0}\rangle$)
and *one* :: $\langle'a\rangle$ ($\langle\mathbf{1}\rangle$)

assumes *conj-disj-distrib*: $\langle x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z) \rangle$
and *disj-conj-distrib*: $\langle x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z) \rangle$
and *conj-one-right*: $\langle x \sqcap \mathbf{1} = x \rangle$
and *disj-zero-right*: $\langle x \sqcup \mathbf{0} = x \rangle$
and *conj-cancel-right* [*simp*]: $\langle x \sqcap - x = \mathbf{0} \rangle$
and *disj-cancel-right* [*simp*]: $\langle x \sqcup - x = \mathbf{1} \rangle$

begin

sublocale *conj: semilattice-neutr* $\langle(\sqcap)\rangle$ $\langle\mathbf{1}\rangle$

proof

show $x \sqcap \mathbf{1} = x$ **for** x

by (*fact conj-one-right*)

show $x \sqcap x = x$ **for** x

proof –

have $x \sqcap x = (x \sqcap x) \sqcup \mathbf{0}$

by (*simp add: disj-zero-right*)

also have $\dots = (x \sqcap x) \sqcup (x \sqcap - x)$

by *simp*

also have $\dots = x \sqcap (x \sqcup - x)$

by (*simp only: conj-disj-distrib*)

also have $\dots = x \sqcap \mathbf{1}$

by *simp*

also have $\dots = x$

by (*simp add: conj-one-right*)

finally show *?thesis* .

qed

qed

sublocale *disj: semilattice-neutr* $\langle (\sqcup) \rangle \langle \mathbf{0} \rangle$

proof

show $x \sqcup \mathbf{0} = x$ **for** x

by (*fact disj-zero-right*)

show $x \sqcup x = x$ **for** x

proof –

have $x \sqcup x = (x \sqcup x) \sqcap \mathbf{1}$

by *simp*

also have $\dots = (x \sqcup x) \sqcap (x \sqcup - x)$

by *simp*

also have $\dots = x \sqcup (x \sqcap - x)$

by (*simp only: disj-conj-distrib*)

also have $\dots = x \sqcup \mathbf{0}$

by *simp*

also have $\dots = x$

by (*simp add: disj-zero-right*)

finally show *?thesis* .

qed

qed

7.1.1 Complement

lemma *complement-unique*:

assumes 1: $a \sqcap x = \mathbf{0}$

assumes 2: $a \sqcup x = \mathbf{1}$

assumes 3: $a \sqcap y = \mathbf{0}$

assumes 4: $a \sqcup y = \mathbf{1}$

shows $x = y$

proof –

from 1 3 **have** $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$

```

  by simp
  then have  $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$ 
    by (simp add: ac-simps)
  then have  $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$ 
    by (simp add: conj-disj-distrib)
  with 2 4 have  $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$ 
    by simp
  then show  $x = y$ 
    by simp
qed

```

lemma *compl-unique*: $x \sqcap y = \mathbf{0} \implies x \sqcup y = \mathbf{1} \implies \neg x = y$
 by (rule complement-unique [OF conj-cancel-right disj-cancel-right])

lemma *double-compl* [simp]: $\neg(\neg x) = x$
proof (rule compl-unique)
 show $\neg x \sqcap x = \mathbf{0}$
 by (simp only: conj-cancel-right conj commute)
 show $\neg x \sqcup x = \mathbf{1}$
 by (simp only: disj-cancel-right disj commute)
qed

lemma *compl-eq-compl-iff* [simp]:
 $\langle \neg x = \neg y \longleftrightarrow x = y \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)
proof
 assume $\langle ?Q \rangle$
 then show $?P$ by simp
next
 assume $\langle ?P \rangle$
 then have $\langle \neg(\neg x) = \neg(\neg y) \rangle$
 by simp
 then show $?Q$
 by simp
qed

7.1.2 Conjunction

lemma *conj-zero-right* [simp]: $x \sqcap \mathbf{0} = \mathbf{0}$
 using conj.left-idem conj-cancel-right by fastforce

lemma *compl-one* [simp]: $\neg \mathbf{1} = \mathbf{0}$
 by (rule compl-unique [OF conj-zero-right disj-zero-right])

lemma *conj-zero-left* [simp]: $\mathbf{0} \sqcap x = \mathbf{0}$
 by (subst conj commute) (rule conj-zero-right)

lemma *conj-cancel-left* [simp]: $\neg x \sqcap x = \mathbf{0}$
 by (subst conj commute) (rule conj-cancel-right)

lemma *conj-disj-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
 by (*simp only: conj commute conj-disj-distrib*)

lemmas *conj-disj-distrib* = *conj-disj-distrib conj-disj-distrib2*

7.1.3 Disjunction

context
begin

interpretation *dual: abstract-boolean-algebra* $\langle(\sqcup)\rangle \langle(\sqcap)\rangle$ *compl* $\langle\mathbf{1}\rangle \langle\mathbf{0}\rangle$
apply *standard*
 apply (*rule disj-conj-distrib*)
 apply (*rule conj-disj-distrib*)
 apply *simp-all*
done

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
 by (*fact dual.conj-zero-right*)

lemma *compl-zero* [*simp*]: $-\mathbf{0} = \mathbf{1}$
 by (*fact dual.compl-one*)

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
 by (*fact dual.conj-zero-left*)

lemma *disj-cancel-left* [*simp*]: $-\ x \sqcup x = \mathbf{1}$
 by (*fact dual.conj-cancel-left*)

lemma *disj-conj-distrib2*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
 by (*fact dual.conj-disj-distrib2*)

lemmas *disj-conj-distrib* = *disj-conj-distrib disj-conj-distrib2*

end

7.1.4 De Morgan’s Laws

lemma *de-Morgan-conj* [*simp*]: $-(x \sqcap y) = -x \sqcup -y$

proof (*rule compl-unique*)

have $(x \sqcap y) \sqcap (-x \sqcup -y) = ((x \sqcap y) \sqcap -x) \sqcup ((x \sqcap y) \sqcap -y)$
 by (*rule conj-disj-distrib*)

also have $\dots = (y \sqcap (x \sqcap -x)) \sqcup (x \sqcap (y \sqcap -y))$
 by (*simp only: ac-simps*)

finally show $(x \sqcap y) \sqcap (-x \sqcup -y) = \mathbf{0}$

 by (*simp only: conj-cancel-right conj-zero-right disj-zero-right*)

next

have $(x \sqcap y) \sqcup (-x \sqcup -y) = (x \sqcup (-x \sqcup -y)) \sqcap (y \sqcup (-x \sqcup -y))$
 by (*rule disj-conj-distrib2*)

also have $\dots = (-y \sqcup (x \sqcup -x)) \sqcap (-x \sqcup (y \sqcup -y))$

```

    by (simp only: ac-simps)
    finally show  $(x \sqcap y) \sqcup (-x \sqcup -y) = \mathbf{1}$ 
    by (simp only: disj-cancel-right disj-one-right conj-one-right)
qed

context
begin

interpretation dual: abstract-boolean-algebra  $\langle(\sqcup)\rangle \langle(\sqcap)\rangle$  compl  $\langle\mathbf{1}\rangle \langle\mathbf{0}\rangle$ 
  apply standard
    apply (rule disj-conj-distrib)
    apply (rule conj-disj-distrib)
    apply simp-all
  done

lemma de-Morgan-disj [simp]:  $-(x \sqcup y) = -x \sqcap -y$ 
  by (fact dual.de-Morgan-conj)

end

end

```

7.2 Symmetric Difference

```

locale abstract-boolean-algebra-sym-diff = abstract-boolean-algebra +
  fixes xor ::  $\langle'a \Rightarrow 'a \Rightarrow 'a\rangle$  (infixr  $\langle\ominus\rangle$  65)
  assumes xor-def :  $\langle x \ominus y = (x \sqcap -y) \sqcup (-x \sqcap y)\rangle$ 
begin

sublocale xor: comm-monoid xor  $\langle\mathbf{0}\rangle$ 
proof
  fix x y z :: 'a
  let ?t =  $(x \sqcap y \sqcap z) \sqcup (x \sqcap -y \sqcap -z) \sqcup (-x \sqcap y \sqcap -z) \sqcup (-x \sqcap -y \sqcap z)$ 
  have  $?t \sqcup (z \sqcap x \sqcap -x) \sqcup (z \sqcap y \sqcap -y) = ?t \sqcup (x \sqcap y \sqcap -y) \sqcup (x \sqcap z \sqcap -z)$ 
  by (simp only: conj-cancel-right conj-zero-right)
  then show  $(x \ominus y) \ominus z = x \ominus (y \ominus z)$ 
  by (simp only: xor-def de-Morgan-disj de-Morgan-conj double-compl)
  (simp only: conj-disj-distrib conj-ac ac-simps)
  show  $x \ominus y = y \ominus x$ 
  by (simp only: xor-def ac-simps)
  show  $x \ominus \mathbf{0} = x$ 
  by (simp add: xor-def)
qed

lemma xor-def2:
   $\langle x \ominus y = (x \sqcup y) \sqcap (-x \sqcup -y)\rangle$ 
proof -

```

note *xor-def* [of $x\ y$]
also have $\langle x \sqcap - y \sqcup - x \sqcap y = ((x \sqcup - x) \sqcap (- y \sqcup - x)) \sqcap (x \sqcup y) \sqcap (- y \sqcup y) \rangle$
by (*simp add: ac-simps disj-conj-distrib*)
also have $\langle \dots = (x \sqcup y) \sqcap (- x \sqcup - y) \rangle$
by (*simp add: ac-simps*)
finally show *?thesis* .
qed

lemma *xor-one-right* [*simp*]: $x \ominus \mathbf{1} = - x$
by (*simp only: xor-def compl-one conj-zero-right conj-one-right disj.left-neutral*)

lemma *xor-one-left* [*simp*]: $\mathbf{1} \ominus x = - x$
using *xor-one-right* [of x] **by** (*simp add: ac-simps*)

lemma *xor-self* [*simp*]: $x \ominus x = \mathbf{0}$
by (*simp only: xor-def conj-cancel-right conj-cancel-left disj-zero-right*)

lemma *xor-left-self* [*simp*]: $x \ominus (x \ominus y) = y$
by (*simp only: xor.assoc [symmetric] xor-self xor.left-neutral*)

lemma *xor-compl-left* [*simp*]: $- x \ominus y = - (x \ominus y)$
by (*simp add: ac-simps flip: xor-one-left*)

lemma *xor-compl-right* [*simp*]: $x \ominus - y = - (x \ominus y)$
using *xor commute xor-compl-left* **by** *auto*

lemma *xor-cancel-right* [*simp*]: $x \ominus - x = \mathbf{1}$
by (*simp only: xor-compl-right xor-self compl-zero*)

lemma *xor-cancel-left* [*simp*]: $- x \ominus x = \mathbf{1}$
by (*simp only: xor-compl-left xor-self compl-zero*)

lemma *conj-xor-distrib*: $x \sqcap (y \ominus z) = (x \sqcap y) \ominus (x \sqcap z)$

proof –

have $*$: $(x \sqcap y \sqcap - z) \sqcup (x \sqcap - y \sqcap z) =$
 $(y \sqcap x \sqcap - x) \sqcup (z \sqcap x \sqcap - x) \sqcup (x \sqcap y \sqcap - z) \sqcup (x \sqcap - y \sqcap z)$

by (*simp only: conj-cancel-right conj-zero-right disj.left-neutral*)

then show $x \sqcap (y \ominus z) = (x \sqcap y) \ominus (x \sqcap z)$

by (*simp (no-asm-use) only:*

xor-def de-Morgan-disj de-Morgan-conj double-compl
conj-disj-distrib ac-simps)

qed

lemma *conj-xor-distrib2*: $(y \ominus z) \sqcap x = (y \sqcap x) \ominus (z \sqcap x)$
by (*simp add: conj commute conj-xor-distrib*)

lemmas *conj-xor-distrib = conj-xor-distrib conj-xor-distrib2*

end

7.3 Type classes

```
class boolean-algebra = distrib-lattice + bounded-lattice + minus + uminus +
  assumes inf-compl-bot:  $\langle x \sqcap - x = \perp \rangle$ 
  and sup-compl-top:  $\langle x \sqcup - x = \top \rangle$ 
  assumes diff-eq:  $\langle x - y = x \sqcap - y \rangle$ 
begin
```

```
sublocale boolean-algebra: abstract-boolean-algebra  $\langle (\sqcap) \rangle \langle (\sqcup) \rangle$  uminus  $\perp \top$ 
  apply standard
  apply (rule inf-sup-distrib1)
  apply (rule sup-inf-distrib1)
  apply (simp-all add: ac-simps inf-compl-bot sup-compl-top)
done
```

```
lemma compl-inf-bot:  $- x \sqcap x = \perp$ 
  by (fact boolean-algebra.conj-cancel-left)
```

```
lemma compl-sup-top:  $- x \sqcup x = \top$ 
  by (fact boolean-algebra.disj-cancel-left)
```

```
lemma compl-unique:
  assumes  $x \sqcap y = \perp$ 
  and  $x \sqcup y = \top$ 
  shows  $- x = y$ 
  using assms by (rule boolean-algebra.compl-unique)
```

```
lemma double-compl:  $- (- x) = x$ 
  by (fact boolean-algebra.double-compl)
```

```
lemma compl-eq-compl-iff:  $- x = - y \iff x = y$ 
  by (fact boolean-algebra.compl-eq-compl-iff)
```

```
lemma compl-bot-eq:  $- \perp = \top$ 
  by (fact boolean-algebra.compl-zero)
```

```
lemma compl-top-eq:  $- \top = \perp$ 
  by (fact boolean-algebra.compl-one)
```

```
lemma compl-inf:  $- (x \sqcap y) = - x \sqcup - y$ 
  by (fact boolean-algebra.de-Morgan-conj)
```

```
lemma compl-sup:  $- (x \sqcup y) = - x \sqcap - y$ 
  by (fact boolean-algebra.de-Morgan-disj)
```

```
lemma compl-mono:
  assumes  $x \leq y$ 
```

shows $-y \leq -x$
proof –
 from *assms* have $x \sqcup y = y$ by (*simp only: le-iff-sup*)
 then have $-(x \sqcup y) = -y$ by *simp*
 then have $-x \sqcap -y = -y$ by *simp*
 then have $-y \sqcap -x = -y$ by (*simp only: inf-commute*)
 then show *?thesis* by (*simp only: le-iff-inf*)
qed

lemma *compl-le-compl-iff* [*simp*]: $-x \leq -y \longleftrightarrow y \leq x$
 by (*auto dest: compl-mono*)

lemma *compl-le-swap1*:
 assumes $y \leq -x$
 shows $x \leq -y$
proof –
 from *assms* have $-(-x) \leq -y$ by (*simp only: compl-le-compl-iff*)
 then show *?thesis* by *simp*
qed

lemma *compl-le-swap2*:
 assumes $-y \leq x$
 shows $-x \leq y$
proof –
 from *assms* have $-x \leq -(-y)$ by (*simp only: compl-le-compl-iff*)
 then show *?thesis* by *simp*
qed

lemma *compl-less-compl-iff* [*simp*]: $-x < -y \longleftrightarrow y < x$
 by (*auto simp add: less-le*)

lemma *compl-less-swap1*:
 assumes $y < -x$
 shows $x < -y$
proof –
 from *assms* have $-(-x) < -y$ by (*simp only: compl-less-compl-iff*)
 then show *?thesis* by *simp*
qed

lemma *compl-less-swap2*:
 assumes $-y < x$
 shows $-x < y$
proof –
 from *assms* have $-x < -(-y)$
 by (*simp only: compl-less-compl-iff*)
 then show *?thesis* by *simp*
qed

lemma *sup-cancel-left1*: $\langle x \sqcup a \sqcup (-x \sqcup b) = \top \rangle$

by (simp add: ac-simps)

lemma *sup-cancel-left2*: $\langle - x \sqcup a \sqcup (x \sqcup b) = \top \rangle$
 by (simp add: ac-simps)

lemma *inf-cancel-left1*: $\langle x \sqcap a \sqcap (- x \sqcap b) = \perp \rangle$
 by (simp add: ac-simps)

lemma *inf-cancel-left2*: $\langle - x \sqcap a \sqcap (x \sqcap b) = \perp \rangle$
 by (simp add: ac-simps)

lemma *sup-compl-top-left1* [simp]: $\langle - x \sqcup (x \sqcup y) = \top \rangle$
 by (simp add: sup-assoc [symmetric])

lemma *sup-compl-top-left2* [simp]: $\langle x \sqcup (- x \sqcup y) = \top \rangle$
 using *sup-compl-top-left1* [of $- x y$] by simp

lemma *inf-compl-bot-left1* [simp]: $\langle - x \sqcap (x \sqcap y) = \perp \rangle$
 by (simp add: inf-assoc [symmetric])

lemma *inf-compl-bot-left2* [simp]: $\langle x \sqcap (- x \sqcap y) = \perp \rangle$
 using *inf-compl-bot-left1* [of $- x y$] by simp

lemma *inf-compl-bot-right* [simp]: $\langle x \sqcap (y \sqcap - x) = \perp \rangle$
 by (subst *inf-left-commute*) simp

end

7.4 Lattice on *bool*

instantiation *bool* :: *boolean-algebra*
begin

definition *bool-Compl-def* [simp]: *uminus* = *Not*

definition *bool-diff-def* [simp]: $A - B \longleftrightarrow A \wedge \neg B$

definition [simp]: $P \sqcap Q \longleftrightarrow P \wedge Q$

definition [simp]: $P \sqcup Q \longleftrightarrow P \vee Q$

instance by *standard auto*

end

lemma *sup-boolI1*: $P \Longrightarrow P \sqcup Q$
 by *simp*

lemma *sup-boolI2*: $Q \Longrightarrow P \sqcup Q$

by *simp*

lemma *sup-boolE*: $P \sqcup Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by *auto*

instance *fun* :: (*type*, *boolean-algebra*) *boolean-algebra*
by *standard* (rule *ext*, *simp-all* add: *inf-compl-bot sup-compl-top diff-eq*)⁺

7.5 Lattice on unary and binary predicates

lemma *inf1I*: $A x \Longrightarrow B x \Longrightarrow (A \sqcap B) x$
by (*simp* add: *inf-fun-def*)

lemma *inf2I*: $A x y \Longrightarrow B x y \Longrightarrow (A \sqcap B) x y$
by (*simp* add: *inf-fun-def*)

lemma *inf1E*: $(A \sqcap B) x \Longrightarrow (A x \Longrightarrow B x \Longrightarrow P) \Longrightarrow P$
by (*simp* add: *inf-fun-def*)

lemma *inf2E*: $(A \sqcap B) x y \Longrightarrow (A x y \Longrightarrow B x y \Longrightarrow P) \Longrightarrow P$
by (*simp* add: *inf-fun-def*)

lemma *inf1D1*: $(A \sqcap B) x \Longrightarrow A x$
by (*rule inf1E*)

lemma *inf2D1*: $(A \sqcap B) x y \Longrightarrow A x y$
by (*rule inf2E*)

lemma *inf1D2*: $(A \sqcap B) x \Longrightarrow B x$
by (*rule inf1E*)

lemma *inf2D2*: $(A \sqcap B) x y \Longrightarrow B x y$
by (*rule inf2E*)

lemma *sup1I1*: $A x \Longrightarrow (A \sqcup B) x$
by (*simp* add: *sup-fun-def*)

lemma *sup2I1*: $A x y \Longrightarrow (A \sqcup B) x y$
by (*simp* add: *sup-fun-def*)

lemma *sup1I2*: $B x \Longrightarrow (A \sqcup B) x$
by (*simp* add: *sup-fun-def*)

lemma *sup2I2*: $B x y \Longrightarrow (A \sqcup B) x y$
by (*simp* add: *sup-fun-def*)

lemma *sup1E*: $(A \sqcup B) x \Longrightarrow (A x \Longrightarrow P) \Longrightarrow (B x \Longrightarrow P) \Longrightarrow P$
by (*simp* add: *sup-fun-def*) *iprover*

lemma *sup2E*: $(A \sqcup B) x y \implies (A x y \implies P) \implies (B x y \implies P) \implies P$
 by (*simp add: sup-fun-def*) *iprover*

Classical introduction rule: no commitment to A vs B .

lemma *sup1CI*: $(\neg B x \implies A x) \implies (A \sqcup B) x$
 by (*auto simp add: sup-fun-def*)

lemma *sup2CI*: $(\neg B x y \implies A x y) \implies (A \sqcup B) x y$
 by (*auto simp add: sup-fun-def*)

7.6 Simproc setup

locale *boolean-algebra-cancel*
begin

lemma *sup1*: $(A::'a::\text{semilattice-sup}) \equiv \text{sup } k a \implies \text{sup } A b \equiv \text{sup } k (\text{sup } a b)$
 by (*simp only: ac-simps*)

lemma *sup2*: $(B::'a::\text{semilattice-sup}) \equiv \text{sup } k b \implies \text{sup } a B \equiv \text{sup } k (\text{sup } a b)$
 by (*simp only: ac-simps*)

lemma *sup0*: $(a::'a::\text{bounded-semilattice-sup-bot}) \equiv \text{sup } a \text{ bot}$
 by *simp*

lemma *inf1*: $(A::'a::\text{semilattice-inf}) \equiv \text{inf } k a \implies \text{inf } A b \equiv \text{inf } k (\text{inf } a b)$
 by (*simp only: ac-simps*)

lemma *inf2*: $(B::'a::\text{semilattice-inf}) \equiv \text{inf } k b \implies \text{inf } a B \equiv \text{inf } k (\text{inf } a b)$
 by (*simp only: ac-simps*)

lemma *inf0*: $(a::'a::\text{bounded-semilattice-inf-top}) \equiv \text{inf } a \text{ top}$
 by *simp*

end

ML-file $\langle \text{Tools/boolean-algebra-cancel.ML} \rangle$

simproc-setup *boolean-algebra-cancel-sup* ($\text{sup } a b::'a::\text{boolean-algebra}$) =
 $\langle K (K (\text{try Boolean-Algebra-Cancel.cancel-sup-conv})) \rangle$

simproc-setup *boolean-algebra-cancel-inf* ($\text{inf } a b::'a::\text{boolean-algebra}$) =
 $\langle K (K (\text{try Boolean-Algebra-Cancel.cancel-inf-conv})) \rangle$

context *boolean-algebra*
begin

lemma *shunt1*: $(x \sqcap y \leq z) \iff (x \leq -y \sqcup z)$
proof

```

assume  $x \sqcap y \leq z$ 
hence  $-y \sqcup (x \sqcap y) \leq -y \sqcup z$ 
using sup.mono by blast
hence  $-y \sqcup x \leq -y \sqcup z$ 
by (simp add: sup-inf-distrib1)
thus  $x \leq -y \sqcup z$ 
by simp
next
assume  $x \leq -y \sqcup z$ 
hence  $x \sqcap y \leq (-y \sqcup z) \sqcap y$ 
using inf-mono by auto
thus  $x \sqcap y \leq z$ 
using inf.boundedE inf-sup-distrib2 by auto
qed

lemma shunt2:  $(x \sqcap -y \leq z) \longleftrightarrow (x \leq y \sqcup z)$ 
by (simp add: shunt1)

lemma inf-shunt:  $(x \sqcap \bar{y} = \perp) \longleftrightarrow (x \leq -y)$ 
by (simp add: order.eq-iff shunt1)

lemma sup-shunt:  $(x \sqcup y = \top) \longleftrightarrow (-x \leq y)$ 
using inf-shunt [of  $\langle -x \rangle \langle -y \rangle$ , symmetric]
by (simp flip: compl-sup compl-top-eq)

lemma diff-shunt-var:  $(x - y = \perp) \longleftrightarrow (x \leq y)$ 
by (simp add: diff-eq inf-shunt)

lemma sup-neg-inf:
 $\langle p \leq q \sqcup r \longleftrightarrow p \sqcap -q \leq r \rangle$  (is  $\langle ?P \longleftrightarrow ?Q \rangle$ )
proof
assume  $?P$ 
then have  $\langle p \sqcap -q \leq (q \sqcup r) \sqcap -q \rangle$ 
by (rule inf-mono) simp
then show  $?Q$ 
by (simp add: inf-sup-distrib2)
next
assume  $?Q$ 
then have  $\langle p \sqcap -q \sqcup q \leq r \sqcup q \rangle$ 
by (rule sup-mono) simp
then show  $?P$ 
by (simp add: sup-inf-distrib ac-simps)
qed

end

end

```

8 Set theory for higher-order logic

```
theory Set
  imports Lattices Boolean-Algebras
begin
```

8.1 Sets as predicates

```
typedecl 'a set
```

```
axiomatization Collect :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a set — comprehension
and member :: 'a  $\Rightarrow$  'a set  $\Rightarrow$  bool — membership
where mem-Collect-eq [iff, code-unfold]: member a (Collect P) = P a
and Collect-mem-eq [simp]: Collect ( $\lambda x.$  member x A) = A
```

notation

```
member ('( $\in$ ')) and
member ((-/  $\in$  -) [51, 51] 50)
```

abbreviation not-member

```
where not-member x A  $\equiv$   $\neg$  (x  $\in$  A) — non-membership
```

notation

```
not-member ('( $\notin$ ')) and
not-member ((-/  $\notin$  -) [51, 51] 50)
```

notation (ASCII)

```
member ('(:')) and
member ((-/ : -) [51, 51] 50) and
not-member ('( $\sim$ :')) and
not-member ((-/  $\sim$ : -) [51, 51] 50)
```

Set comprehensions

syntax

```
-Coll :: ptrn  $\Rightarrow$  bool  $\Rightarrow$  'a set ((1{-/ -}))
```

translations

```
{x. P}  $\equiv$  CONST Collect ( $\lambda x.$  P)
```

syntax (ASCII)

```
-Collect :: ptrn  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  'a set ((1{(-/: -)/ -}))
```

syntax

```
-Collect :: ptrn  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  'a set ((1{(-/  $\in$  -)/ -}))
```

translations

```
{p:A. P}  $\rightarrow$  CONST Collect ( $\lambda p.$  p  $\in$  A  $\wedge$  P)
```

lemma CollectI: P a \Longrightarrow a \in {x. P x}

by simp

lemma CollectD: a \in {x. P x} \Longrightarrow P a

by simp

lemma *Collect-cong*: $(\bigwedge x. P x = Q x) \implies \{x. P x\} = \{x. Q x\}$
by *simp*

Simproc for pulling $x = t$ in $\{x. \dots \wedge x = t \wedge \dots\}$ to the front (and similarly for $t = x$):

simproc-setup *defined-Collect* ($\{x. P x \wedge Q x\} = \langle$
 $K (Quantifier1.rearrange-Collect$
 $(fn ctxt =>$
 $resolve-tac ctxt @\{thms Collect-cong\} 1 THEN$
 $resolve-tac ctxt @\{thms iffI\} 1 THEN$
 $ALLGOALS$
 $(EVERY' [REPEAT-DETERM o eresolve-tac ctxt @\{thms conjE\},$
 $DEPTH-SOLVE-1 o (assume-tac ctxt ORELSE' resolve-tac ctxt @\{thms$
 $conjI\}]))))$
 \rangle

lemmas *CollectE = CollectD [elim-format]*

lemma *set-eqI*:
assumes $\bigwedge x. x \in A \longleftrightarrow x \in B$
shows $A = B$
proof –
from *assms* **have** $\{x. x \in A\} = \{x. x \in B\}$
by *simp*
then show *?thesis* **by** *simp*
qed

lemma *set-eq-iff*: $A = B \longleftrightarrow (\forall x. x \in A \longleftrightarrow x \in B)$
by (*auto intro:set-eqI*)

lemma *Collect-eqI*:
assumes $\bigwedge x. P x = Q x$
shows $Collect P = Collect Q$
using *assms* **by** (*auto intro: set-eqI*)

Lifting of predicate class instances

instantiation *set* :: (*type*) *boolean-algebra*
begin

definition *less-eq-set*
where $A \leq B \longleftrightarrow (\lambda x. member x A) \leq (\lambda x. member x B)$

definition *less-set*
where $A < B \longleftrightarrow (\lambda x. member x A) < (\lambda x. member x B)$

definition *inf-set*
where $A \sqcap B = Collect ((\lambda x. member x A) \sqcap (\lambda x. member x B))$

definition *sup-set*

where $A \sqcup B = \text{Collect } ((\lambda x. \text{member } x A) \sqcup (\lambda x. \text{member } x B))$

definition *bot-set*

where $\perp = \text{Collect } \perp$

definition *top-set*

where $\top = \text{Collect } \top$

definition *uminus-set*

where $- A = \text{Collect } (- (\lambda x. \text{member } x A))$

definition *minus-set*

where $A - B = \text{Collect } ((\lambda x. \text{member } x A) - (\lambda x. \text{member } x B))$

instance

by *standard*

(*simp-all add: less-eq-set-def less-set-def inf-set-def sup-set-def
bot-set-def top-set-def uminus-set-def minus-set-def
less-le-not-le sup-inf-distrib1 diff-eq set-eqI fun-eq-iff
del: inf-apply sup-apply bot-apply top-apply minus-apply uminus-apply*)

end

Set enumerations

abbreviation *empty* :: 'a set ({})

where {} \equiv bot

definition *insert* :: 'a \Rightarrow 'a set \Rightarrow 'a set

where *insert-compr*: $\text{insert } a B = \{x. x = a \vee x \in B\}$

syntax

-Finset :: args \Rightarrow 'a set ({}(-))

translations

$\{x, xs\} \equiv \text{CONST } \text{insert } x \{xs\}$

$\{x\} \equiv \text{CONST } \text{insert } x \{\}$

8.2 Subsets and bounded quantifiers

abbreviation *subset* :: 'a set \Rightarrow 'a set \Rightarrow bool

where *subset* \equiv less

abbreviation *subset-eq* :: 'a set \Rightarrow 'a set \Rightarrow bool

where *subset-eq* \equiv less-eq

notation

subset (('(\subset ')) **and**

subset ((-/ \subset -) [51, 51] 50) **and**

subset-eq (('(\subseteq ')) **and**

subset-eq ((-/ \subseteq -) [51, 51] 50)

abbreviation (*input*)

$supset :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$ **where**
 $supset \equiv greater$

abbreviation (*input*)

$supset-eq :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$ **where**
 $supset-eq \equiv greater-eq$

notation

$supset$ ((\supset)) **and**
 $supset$ ($(-/ \supset -)$ [51, 51] 50) **and**
 $supset-eq$ ((\supseteq)) **and**
 $supset-eq$ ($(-/ \supseteq -)$ [51, 51] 50)

notation (*ASCII output*)

$subset$ ($(<')$) **and**
 $subset$ ($(-/ < -)$ [51, 51] 50) **and**
 $subset-eq$ ((\leq')) **and**
 $subset-eq$ ($(-/ \leq -)$ [51, 51] 50)

definition $Ball :: 'a\ set \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$

where $Ball\ A\ P \longleftrightarrow (\forall x. x \in A \longrightarrow P\ x)$ — bounded universal quantifiers

definition $Bex :: 'a\ set \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$

where $Bex\ A\ P \longleftrightarrow (\exists x. x \in A \wedge P\ x)$ — bounded existential quantifiers

syntax (*ASCII*)

$-Ball$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists ALL\ (-/:-)/ -) [0, 0, 10] 10)$
 $-Bex$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists EX\ (-/:-)/ -) [0, 0, 10] 10)$
 $-Bex1$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists EX!\ (-/:-)/ -) [0, 0, 10] 10)$
 $-Bleast$ $::\ id \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a$ $((\exists LEAST\ (-/:-)/ -) [0, 0, 10] 10)$

syntax (*input*)

$-Ball$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists! (-/:-)/ -) [0, 0, 10] 10)$
 $-Bex$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists? (-/:-)/ -) [0, 0, 10] 10)$
 $-Bex1$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists?! (-/:-)/ -) [0, 0, 10] 10)$

syntax

$-Ball$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists \forall (-/\in-)/ -) [0, 0, 10] 10)$
 $-Bex$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists \exists (-/\in-)/ -) [0, 0, 10] 10)$
 $-Bex1$ $::\ ptrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((\exists \exists! (-/\in-)/ -) [0, 0, 10] 10)$
 $-Bleast$ $::\ id \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a$ $((\exists LEAST (-/\in-)/ -) [0, 0, 10] 10)$

translations

$\forall x \in A. P \Rightarrow CONST\ Ball\ A\ (\lambda x. P)$
 $\exists x \in A. P \Rightarrow CONST\ Bex\ A\ (\lambda x. P)$
 $\exists! x \in A. P \Rightarrow \exists! x. x \in A \wedge P$
 $LEAST\ x:A. P \Rightarrow LEAST\ x. x \in A \wedge P$

syntax (ASCII output)

```

-setlessAll :: [idt, 'a, bool] => bool (( $\exists$ ALL -<-./ -) [0, 0, 10] 10)
-setlessEx  :: [idt, 'a, bool] => bool (( $\exists$ EX -<-./ -) [0, 0, 10] 10)
-settleAll  :: [idt, 'a, bool] => bool (( $\exists$ ALL -<=.-./ -) [0, 0, 10] 10)
-settleEx   :: [idt, 'a, bool] => bool (( $\exists$ EX -<=.-./ -) [0, 0, 10] 10)
-settleEx1  :: [idt, 'a, bool] => bool (( $\exists$ EX! -<=.-./ -) [0, 0, 10] 10)

```

syntax

```

-setlessAll :: [idt, 'a, bool] => bool (( $\exists$  $\forall$ -C-./ -) [0, 0, 10] 10)
-setlessEx  :: [idt, 'a, bool] => bool (( $\exists$  $\exists$ -C-./ -) [0, 0, 10] 10)
-settleAll  :: [idt, 'a, bool] => bool (( $\exists$  $\forall$ - $\subseteq$ -./ -) [0, 0, 10] 10)
-settleEx   :: [idt, 'a, bool] => bool (( $\exists$  $\exists$ - $\subseteq$ -./ -) [0, 0, 10] 10)
-settleEx1  :: [idt, 'a, bool] => bool (( $\exists$  $\exists$ !- $\subseteq$ -./ -) [0, 0, 10] 10)

```

translations

```

 $\forall A \subset B. P \rightarrow \forall A. A \subset B \rightarrow P$ 
 $\exists A \subset B. P \rightarrow \exists A. A \subset B \wedge P$ 
 $\forall A \subseteq B. P \rightarrow \forall A. A \subseteq B \rightarrow P$ 
 $\exists A \subseteq B. P \rightarrow \exists A. A \subseteq B \wedge P$ 
 $\exists !A \subseteq B. P \rightarrow \exists !A. A \subseteq B \wedge P$ 

```

print-translation <

let

```

val All-binder = Mixfix.binder-name const-syntax <All>;
val Ex-binder  = Mixfix.binder-name const-syntax <Ex>;
val impl       = const-syntax <HOL.implies>;
val conj       = const-syntax <HOL.conj>;
val sbset      = const-syntax <subset>;
val sbset-eq   = const-syntax <subset-eq>;

```

val trans =

```

[[ (All-binder, impl, sbset), syntax-const <-setlessAll>,
  (All-binder, impl, sbset-eq), syntax-const <-settleAll>,
  (Ex-binder, conj, sbset), syntax-const <-setlessEx>,
  (Ex-binder, conj, sbset-eq), syntax-const <-settleEx> ];

```

fun mk v (v', T) c n P =

```

if v = v' and also not (Term.exists-subterm (fn Free (x, -) => x = v | - =>
false) n)

```

```

then Syntax.const c $ Syntax-Trans.mark-bound-body (v', T) $ n $ P
else raise Match;

```

fun tr' q = (q, fn - =>

```

(fn [Const (syntax-const <-bound>, -) $ Free (v, Type (type-name <set>, -)),
    Const (c, -) $
      (Const (d, -) $ (Const (syntax-const <-bound>, -) $ Free (v', T)) $ n)

```

\$ P] =>

```

(case AList.lookup (=) trans (q, c, d) of

```



```

      NONE => raise Match
      | SOME l => mk v (v', T) l n P
      | - => raise Match));
in
  [tr' All-binder, tr' Ex-binder]
end
>

```

Translate between $\{e \mid x1 \dots xn. P\}$ and $\{u. \exists x1 \dots xn. u = e \wedge P\}$; $\{y. \exists x1 \dots xn. y = e \wedge P\}$ is only translated if $[0..n] \subseteq \text{bvs } e$.

syntax

```
-Setcompr :: 'a => idts => bool => 'a set ((1{- |./-/-})
```

parse-translation <

```

let
  val ex-tr = snd (Syntax-Trans.mk-binder-tr (EX, const-syntax <Ex>));

  fun nvars (Const (syntax-const <-idts>, -) $ - $ idts) = nvars idts + 1
    | nvars - = 1;

  fun setcompr-tr ctxt [e, idts, b] =
    let
      val eq = Syntax.const const-syntax <HOL.eq> $ Bound (nvars idts) $ e;
      val P = Syntax.const const-syntax <HOL.conj> $ eq $ b;
      val exP = ex-tr ctxt [idts, P];
    in Syntax.const const-syntax <Collect> $ absdummy dummyT exP end;

  in [(syntax-const <-Setcompr>, setcompr-tr)] end
>

```

print-translation <

```

[Syntax-Trans.preserve-binder-abs2-tr' const-syntax <Ball> syntax-const <-Ball>,
 Syntax-Trans.preserve-binder-abs2-tr' const-syntax <Bex> syntax-const <-Bex>]
> — to avoid eta-contraction of body

```

print-translation <

```

let
  val ex-tr' = snd (Syntax-Trans.mk-binder-tr' (const-syntax <Ex>, DUMMY));

  fun setcompr-tr' ctxt [Abs (abs as (-, -, P))] =
    let
      fun check (Const (const-syntax <Ex>, -) $ Abs (-, -, P), n) = check (P, n +
1)
        | check (Const (const-syntax <HOL.conj>, -) $
          (Const (const-syntax <HOL.eq>, -) $ Bound m $ e) $ P, n) =
          n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
          subset (=) (0 upto (n - 1), add-loose-bnos (e, 0, []))
        | check - = false;
    in

```

```

fun tr' (- $ abs) =
  let val - $ idts $ (- $ (- $ - $ e) $ Q) = ex-tr' ctxt [abs]
      in Syntax.const syntax-const <-Setcompr> $ e $ idts $ Q end;
in
  if check (P, 0) then tr' P
  else
    let
      val (x as - $ Free(xN, -), t) = Syntax-Trans.atomic-abs-tr' abs;
      val M = Syntax.const syntax-const <-Coll> $ x $ t;
    in
      case t of
        Const (const-syntax <HOL.conj>, -) $
          (Const (const-syntax <Set.member>, -) $
            (Const (syntax-const <-bound>, -) $ Free (yN, -)) $ A) $ P =>
          if xN = yN then Syntax.const syntax-const <-Collect> $ x $ A $ P else
M
      | - => M
    end
  end;
in [(const-syntax <Collect>, setcompr-tr')] end
>

```

```

simproc-setup defined-Bex ( $\exists x \in A. P x \wedge Q x$ ) = <
  K (Quantifier1.rearrange-Bex (fn ctxt => unfold-tac ctxt @ {thms Bex-def}))
>

```

```

simproc-setup defined-All ( $\forall x \in A. P x \longrightarrow Q x$ ) = <
  K (Quantifier1.rearrange-Ball (fn ctxt => unfold-tac ctxt @ {thms Ball-def}))
>

```

```

lemma ballI [intro!]: ( $\bigwedge x. x \in A \implies P x$ )  $\implies \forall x \in A. P x$ 
  by (simp add: Ball-def)

```

```

lemmas strip = impI allI ballI

```

```

lemma bspec [dest?]:  $\forall x \in A. P x \implies x \in A \implies P x$ 
  by (simp add: Ball-def)

```

Gives better instantiation for bound:

```

setup <
  map-theory-claset (fn ctxt =>
    ctxt addbefore (bspec, fn ctxt' => dresolve-tac ctxt' @ {thms bspec} THEN'
      assume-tac ctxt'))
>

```

```

ML <
  structure Simpdata =
  struct
    open Simpdata;

```

```

  val mksimps-pairs = [(const-name <Ball>, @{thms bspec})] @ mksimps-pairs;
end;

```

```

open Simpdata;
>

```

declaration <fn - => *Simplifier.map-ss* (*Simplifier.set-mksimps* (*mksimps mk-simps-pairs*))>

lemma *ballE* [*elim*]: $\forall x \in A. P x \implies (P x \implies Q) \implies (x \notin A \implies Q) \implies Q$
unfolding *Ball-def* **by** *blast*

lemma *bexI* [*intro*]: $P x \implies x \in A \implies \exists x \in A. P x$
— Normally the best argument order: $P x$ constrains the choice of $x \in A$.
unfolding *Bex-def* **by** *blast*

lemma *rev-bexI* [*intro?*]: $x \in A \implies P x \implies \exists x \in A. P x$
— The best argument order when there is only one $x \in A$.
unfolding *Bex-def* **by** *blast*

lemma *bexCI*: $(\forall x \in A. \neg P x \implies P a) \implies a \in A \implies \exists x \in A. P x$
unfolding *Bex-def* **by** *blast*

lemma *bexE* [*elim!*]: $\exists x \in A. P x \implies (\bigwedge x. x \in A \implies P x \implies Q) \implies Q$
unfolding *Bex-def* **by** *blast*

lemma *ball-triv* [*simp*]: $(\forall x \in A. P) \longleftrightarrow ((\exists x. x \in A) \longrightarrow P)$
— trivial rewrite rule.
by (*simp add: Ball-def*)

lemma *bex-triv* [*simp*]: $(\exists x \in A. P) \longleftrightarrow ((\exists x. x \in A) \wedge P)$
— Dual form for existentials.
by (*simp add: Bex-def*)

lemma *bex-triv-one-point1* [*simp*]: $(\exists x \in A. x = a) \longleftrightarrow a \in A$
by *blast*

lemma *bex-triv-one-point2* [*simp*]: $(\exists x \in A. a = x) \longleftrightarrow a \in A$
by *blast*

lemma *bex-one-point1* [*simp*]: $(\exists x \in A. x = a \wedge P x) \longleftrightarrow a \in A \wedge P a$
by *blast*

lemma *bex-one-point2* [*simp*]: $(\exists x \in A. a = x \wedge P x) \longleftrightarrow a \in A \wedge P a$
by *blast*

lemma *ball-one-point1* [*simp*]: $(\forall x \in A. x = a \longrightarrow P x) \longleftrightarrow (a \in A \longrightarrow P a)$
by *blast*

lemma *ball-one-point2* [*simp*]: $(\forall x \in A. a = x \longrightarrow P x) \longleftrightarrow (a \in A \longrightarrow P a)$
by *blast*

lemma *ball-conj-distrib*: $(\forall x \in A. P x \wedge Q x) \longleftrightarrow (\forall x \in A. P x) \wedge (\forall x \in A. Q x)$
by *blast*

lemma *bex-disj-distrib*: $(\exists x \in A. P x \vee Q x) \longleftrightarrow (\exists x \in A. P x) \vee (\exists x \in A. Q x)$
by *blast*

Congruence rules

lemma *ball-cong*:
 $\llbracket A = B; \bigwedge x. x \in B \Longrightarrow P x \longleftrightarrow Q x \rrbracket \Longrightarrow$
 $(\forall x \in A. P x) \longleftrightarrow (\forall x \in B. Q x)$
by (*simp add: Ball-def*)

lemma *ball-cong-simp* [*cong*]:
 $\llbracket A = B; \bigwedge x. x \in B =_{\text{simp}} \Longrightarrow P x \longleftrightarrow Q x \rrbracket \Longrightarrow$
 $(\forall x \in A. P x) \longleftrightarrow (\forall x \in B. Q x)$
by (*simp add: simp-implies-def Ball-def*)

lemma *bex-cong*:
 $\llbracket A = B; \bigwedge x. x \in B \Longrightarrow P x \longleftrightarrow Q x \rrbracket \Longrightarrow$
 $(\exists x \in A. P x) \longleftrightarrow (\exists x \in B. Q x)$
by (*simp add: Bex-def cong: conj-cong*)

lemma *bex-cong-simp* [*cong*]:
 $\llbracket A = B; \bigwedge x. x \in B =_{\text{simp}} \Longrightarrow P x \longleftrightarrow Q x \rrbracket \Longrightarrow$
 $(\exists x \in A. P x) \longleftrightarrow (\exists x \in B. Q x)$
by (*simp add: simp-implies-def Bex-def cong: conj-cong*)

lemma *bex1-def*: $(\exists ! x \in X. P x) \longleftrightarrow (\exists x \in X. P x) \wedge (\forall x \in X. \forall y \in X. P x \longrightarrow P y \longrightarrow x = y)$
by *auto*

8.3 Basic operations

8.3.1 Subsets

lemma *subsetI* [*intro!*]: $(\bigwedge x. x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
by (*simp add: less-eq-set-def le-fun-def*)

Map the type *'a set* \Rightarrow *anything* to just *'a*; for overloading constants whose first argument has type *'a set*.

lemma *subsetD* [*elim, intro?*]: $A \subseteq B \Longrightarrow c \in A \Longrightarrow c \in B$
by (*simp add: less-eq-set-def le-fun-def*)
 — Rule in Modus Ponens style.

lemma *rev-subsetD* [*intro?, no-atp*]: $c \in A \Longrightarrow A \subseteq B \Longrightarrow c \in B$

— The same, with reversed premises for use with *erule* – cf. $\llbracket ?P; ?P \longrightarrow ?Q \rrbracket \Longrightarrow ?Q$.

by (*rule subsetD*)

lemma *subsetCE* [*elim,no-atp*]: $A \subseteq B \Longrightarrow (c \notin A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$

— Classical elimination rule.

by (*auto simp add: less-eq-set-def le-fun-def*)

lemma *subset-eq*: $A \subseteq B \longleftrightarrow (\forall x \in A. x \in B)$

by *blast*

lemma *contra-subsetD* [*no-atp*]: $A \subseteq B \Longrightarrow c \notin B \Longrightarrow c \notin A$

by *blast*

lemma *subset-refl*: $A \subseteq A$

by (*fact order-refl*)

lemma *subset-trans*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow A \subseteq C$

by (*fact order-trans*)

lemma *subset-not-subset-eq* [*code*]: $A \subset B \longleftrightarrow A \subseteq B \wedge \neg B \subseteq A$

by (*fact less-le-not-le*)

lemma *eq-mem-trans*: $a = b \Longrightarrow b \in A \Longrightarrow a \in A$

by *simp*

lemmas *basic-trans-rules* [*trans*] =

order-trans-rules rev-subsetD subsetD eq-mem-trans

8.3.2 Equality

lemma *subset-antisym* [*intro!*]: $A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow A = B$

— Anti-symmetry of the subset relation.

by (*iprover intro: set-eqI subsetD*)

Equality rules from ZF set theory – are they appropriate here?

lemma *equalityD1*: $A = B \Longrightarrow A \subseteq B$

by *simp*

lemma *equalityD2*: $A = B \Longrightarrow B \subseteq A$

by *simp*

Be careful when adding this to the claset as *subset-empty* is in the simpset:

$A = \{\}$ goes to $\{\} \subseteq A$ and $A \subseteq \{\}$ and then back to $A = \{\}$!

lemma *equalityE*: $A = B \Longrightarrow (A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow P) \Longrightarrow P$

by *simp*

lemma *equalityCE* [*elim*]: $A = B \implies (c \in A \implies c \in B \implies P) \implies (c \notin A \implies c \notin B \implies P) \implies P$
by *blast*

lemma *eqset-imp-iff*: $A = B \implies x \in A \longleftrightarrow x \in B$
by *simp*

lemma *eqelem-imp-iff*: $x = y \implies x \in A \longleftrightarrow y \in A$
by *simp*

8.3.3 The empty set

lemma *empty-def*: $\{\} = \{x. \text{False}\}$
by (*simp add: bot-set-def bot-fun-def*)

lemma *empty-iff* [*simp*]: $c \in \{\} \longleftrightarrow \text{False}$
by (*simp add: empty-def*)

lemma *emptyE* [*elim!*]: $a \in \{\} \implies P$
by *simp*

lemma *empty-subsetI* [*iff*]: $\{\} \subseteq A$
 — One effect is to delete the ASSUMPTION $\{\} \subseteq A$
by *blast*

lemma *equalsOI*: $(\bigwedge y. y \in A \implies \text{False}) \implies A = \{\}$
by *blast*

lemma *equalsOD*: $A = \{\} \implies a \notin A$
 — Use for reasoning about disjointness: $A \cap B = \{\}$
by *blast*

lemma *ball-empty* [*simp*]: $\text{Ball } \{\} P \longleftrightarrow \text{True}$
by (*simp add: Ball-def*)

lemma *bex-empty* [*simp*]: $\text{Bex } \{\} P \longleftrightarrow \text{False}$
by (*simp add: Bex-def*)

8.3.4 The universal set – UNIV

abbreviation *UNIV* :: 'a set
where *UNIV* \equiv *top*

lemma *UNIV-def*: $\text{UNIV} = \{x. \text{True}\}$
by (*simp add: top-set-def top-fun-def*)

lemma *UNIV-I* [*simp*]: $x \in \text{UNIV}$
by (*simp add: UNIV-def*)

declare *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

lemma *UNIV-witness* [*intro?*]: $\exists x. x \in UNIV$
by *simp*

lemma *subset-UNIV*: $A \subseteq UNIV$
by (*fact top-greatest*)

Eta-contracting these two rules (to remove P) causes them to be ignored because of their interaction with congruence rules.

lemma *ball-UNIV* [*simp*]: $Ball\ UNIV\ P \longleftrightarrow All\ P$
by (*simp add: Ball-def*)

lemma *bex-UNIV* [*simp*]: $Bex\ UNIV\ P \longleftrightarrow Ex\ P$
by (*simp add: Bex-def*)

lemma *UNIV-eq-I*: $(\bigwedge x. x \in A) \Longrightarrow UNIV = A$
by *auto*

lemma *UNIV-not-empty* [*iff*]: $UNIV \neq \{\}$
by (*blast elim: equalityE*)

lemma *empty-not-UNIV* [*simp*]: $\{\} \neq UNIV$
by *blast*

8.3.5 The Powerset operator – Pow

definition *Pow* :: $'a\ set \Rightarrow 'a\ set\ set$
where *Pow-def*: $Pow\ A = \{B. B \subseteq A\}$

lemma *Pow-iff* [*iff*]: $A \in Pow\ B \longleftrightarrow A \subseteq B$
by (*simp add: Pow-def*)

lemma *PowI*: $A \subseteq B \Longrightarrow A \in Pow\ B$
by (*simp add: Pow-def*)

lemma *PowD*: $A \in Pow\ B \Longrightarrow A \subseteq B$
by (*simp add: Pow-def*)

lemma *Pow-bottom*: $\{\} \in Pow\ B$
by *simp*

lemma *Pow-top*: $A \in Pow\ A$
by *simp*

lemma *Pow-not-empty*: $Pow\ A \neq \{\}$
using *Pow-top* **by** *blast*

8.3.6 Set complement

lemma *Compl-iff* [*simp*]: $c \in - A \longleftrightarrow c \notin A$
 by (*simp add: fun-Compl-def uminus-set-def*)

lemma *ComplI* [*intro!*]: $(c \in A \Longrightarrow \text{False}) \Longrightarrow c \in - A$
 by (*simp add: fun-Compl-def uminus-set-def*) *blast*

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile ...

lemma *ComplD* [*dest!*]: $c \in - A \Longrightarrow c \notin A$
 by *simp*

lemmas *ComplE* = *ComplD* [*elim-format*]

lemma *Compl-eq*: $- A = \{x. \neg x \in A\}$
 by *blast*

8.3.7 Binary intersection

abbreviation *inter* :: 'a set \Rightarrow 'a set \Rightarrow 'a set (**infixl** \cap 70)
 where $(\cap) \equiv \text{inf}$

notation (*ASCII*)
inter (**infixl** *Int* 70)

lemma *Int-def*: $A \cap B = \{x. x \in A \wedge x \in B\}$
 by (*simp add: inf-set-def inf-fun-def*)

lemma *Int-iff* [*simp*]: $c \in A \cap B \longleftrightarrow c \in A \wedge c \in B$
 unfolding *Int-def* by *blast*

lemma *IntI* [*intro!*]: $c \in A \Longrightarrow c \in B \Longrightarrow c \in A \cap B$
 by *simp*

lemma *IntD1*: $c \in A \cap B \Longrightarrow c \in A$
 by *simp*

lemma *IntD2*: $c \in A \cap B \Longrightarrow c \in B$
 by *simp*

lemma *IntE* [*elim!*]: $c \in A \cap B \Longrightarrow (c \in A \Longrightarrow c \in B \Longrightarrow P) \Longrightarrow P$
 by *simp*

8.3.8 Binary union

abbreviation *union* :: 'a set \Rightarrow 'a set \Rightarrow 'a set (**infixl** \cup 65)
 where *union* $\equiv \text{sup}$

notation (*ASCII*)
union (**infixl** *Un 65*)

lemma *Un-def*: $A \cup B = \{x. x \in A \vee x \in B\}$
by (*simp add: sup-set-def sup-fun-def*)

lemma *Un-iff* [*simp*]: $c \in A \cup B \longleftrightarrow c \in A \vee c \in B$
unfolding *Un-def* **by** *blast*

lemma *UnI1* [*elim?*]: $c \in A \Longrightarrow c \in A \cup B$
by *simp*

lemma *UnI2* [*elim?*]: $c \in B \Longrightarrow c \in A \cup B$
by *simp*

Classical introduction rule: no commitment to A vs. B .

lemma *UnCI* [*intro!*]: $(c \notin B \Longrightarrow c \in A) \Longrightarrow c \in A \cup B$
by *auto*

lemma *UnE* [*elim!*]: $c \in A \cup B \Longrightarrow (c \in A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$
unfolding *Un-def* **by** *blast*

lemma *insert-def*: $\text{insert } a \ B = \{x. x = a\} \cup B$
by (*simp add: insert-compr Un-def*)

8.3.9 Set difference

lemma *Diff-iff* [*simp*]: $c \in A - B \longleftrightarrow c \in A \wedge c \notin B$
by (*simp add: minus-set-def fun-diff-def*)

lemma *DiffI* [*intro!*]: $c \in A \Longrightarrow c \notin B \Longrightarrow c \in A - B$
by *simp*

lemma *DiffD1*: $c \in A - B \Longrightarrow c \in A$
by *simp*

lemma *DiffD2*: $c \in A - B \Longrightarrow c \in B \Longrightarrow P$
by *simp*

lemma *DiffE* [*elim!*]: $c \in A - B \Longrightarrow (c \in A \Longrightarrow c \notin B \Longrightarrow P) \Longrightarrow P$
by *simp*

lemma *set-diff-eq*: $A - B = \{x. x \in A \wedge x \notin B\}$
by *blast*

lemma *Compl-eq-Diff-UNIV*: $- A = (UNIV - A)$
by *blast*

abbreviation *sym-diff* :: 'a set \Rightarrow 'a set \Rightarrow 'a set **where**
sym-diff A B $\equiv ((A - B) \cup (B - A))$

8.3.10 Augmenting a set – insert

lemma *insert-iff* [*simp*]: $a \in \text{insert } b \ A \longleftrightarrow a = b \vee a \in A$
unfolding *insert-def* **by** *blast*

lemma *insertI1*: $a \in \text{insert } a \ B$
by *simp*

lemma *insertI2*: $a \in B \Longrightarrow a \in \text{insert } b \ B$
by *simp*

lemma *insertE* [*elim!*]: $a \in \text{insert } b \ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a \in A \Longrightarrow P) \Longrightarrow P$
unfolding *insert-def* **by** *blast*

lemma *insertCI* [*intro!*]: $(a \notin B \Longrightarrow a = b) \Longrightarrow a \in \text{insert } b \ B$
 — Classical introduction rule.
by *auto*

lemma *subset-insert-iff*: $A \subseteq \text{insert } x \ B \longleftrightarrow (\text{if } x \in A \text{ then } A - \{x\} \subseteq B \text{ else } A \subseteq B)$
by *auto*

lemma *set-insert*:
assumes $x \in A$
obtains B **where** $A = \text{insert } x \ B$ **and** $x \notin B$

proof

show $A = \text{insert } x \ (A - \{x\})$ **using** *assms* **by** *blast*

show $x \notin A - \{x\}$ **by** *blast*

qed

lemma *insert-ident*: $x \notin A \Longrightarrow x \notin B \Longrightarrow \text{insert } x \ A = \text{insert } x \ B \longleftrightarrow A = B$
by *auto*

lemma *insert-eq-iff*:

assumes $a \notin A \ b \notin B$

shows $\text{insert } a \ A = \text{insert } b \ B \longleftrightarrow$

(if $a = b$ then $A = B$ else $\exists C. A = \text{insert } b \ C \wedge b \notin C \wedge B = \text{insert } a \ C \wedge a \notin C$)

(is $?L \longleftrightarrow ?R$)

proof

show $?R$ **if** $?L$

proof (*cases* $a = b$)

case *True*

with *assms* $\langle ?L \rangle$ **show** $?R$

by (*simp add: insert-ident*)

```

next
  case False
  let  $?C = A - \{b\}$ 
  have  $A = \text{insert } b \ ?C \wedge b \notin ?C \wedge B = \text{insert } a \ ?C \wedge a \notin ?C$ 
    using assms  $\langle ?L \rangle \langle a \neq b \rangle$  by auto
  then show  $?R$  using  $\langle a \neq b \rangle$  by auto
qed
show  $?L$  if  $?R$ 
  using that by (auto split: if-splits)
qed

```

lemma *insert-UNIV*[*simp*]: $\text{insert } x \ UNIV = UNIV$
 by *auto*

8.3.11 Singletons, using insert

lemma *singletonI* [*intro!*]: $a \in \{a\}$
 — Redundant? But unlike *insertCI*, it proves the subgoal immediately!
 by (*rule insertI1*)

lemma *singletonD* [*dest!*]: $b \in \{a\} \implies b = a$
 by *blast*

lemmas *singletonE* = *singletonD* [*elim-format*]

lemma *singleton-iff*: $b \in \{a\} \longleftrightarrow b = a$
 by *blast*

lemma *singleton-inject* [*dest!*]: $\{a\} = \{b\} \implies a = b$
 by *blast*

lemma *singleton-insert-inj-eq* [*iff*]: $\{b\} = \text{insert } a \ A \longleftrightarrow a = b \wedge A \subseteq \{b\}$
 by *blast*

lemma *singleton-insert-inj-eq'* [*iff*]: $\text{insert } a \ A = \{b\} \longleftrightarrow a = b \wedge A \subseteq \{b\}$
 by *blast*

lemma *subset-singletonD*: $A \subseteq \{x\} \implies A = \{\} \vee A = \{x\}$
 by *fast*

lemma *subset-singleton-iff*: $X \subseteq \{a\} \longleftrightarrow X = \{\} \vee X = \{a\}$
 by *blast*

lemma *subset-singleton-iff-Uniq*: $(\exists a. A \subseteq \{a\}) \longleftrightarrow (\exists_{\leq 1} x. x \in A)$
 unfolding *Uniq-def* by *blast*

lemma *singleton-conv* [*simp*]: $\{x. x = a\} = \{a\}$
 by *blast*

lemma *singleton-conv2* [*simp*]: $\{x. a = x\} = \{a\}$
by *blast*

lemma *Diff-single-insert*: $A - \{x\} \subseteq B \implies A \subseteq \text{insert } x B$
by *blast*

lemma *subset-Diff-insert*: $A \subseteq B - \text{insert } x C \iff A \subseteq B - C \wedge x \notin A$
by *blast*

lemma *doubleton-eq-iff*: $\{a, b\} = \{c, d\} \iff a = c \wedge b = d \vee a = d \wedge b = c$
by (*blast elim: equalityE*)

lemma *Un-singleton-iff*: $A \cup B = \{x\} \iff A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
by *auto*

lemma *singleton-Un-iff*: $\{x\} = A \cup B \iff A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
by *auto*

8.3.12 Image of a set under a function

Frequently b does not have the syntactic form of $f x$.

definition *image* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set}$ (**infixr** ‘90)
where $f \text{ ‘ } A = \{y. \exists x \in A. y = f x\}$

lemma *image-eqI* [*simp, intro*]: $b = f x \implies x \in A \implies b \in f \text{ ‘ } A$
unfolding *image-def* **by** *blast*

lemma *imageI*: $x \in A \implies f x \in f \text{ ‘ } A$
by (*rule image-eqI*) (*rule refl*)

lemma *rev-image-eqI*: $x \in A \implies b = f x \implies b \in f \text{ ‘ } A$
— This version’s more effective when we already have the required x .
by (*rule image-eqI*)

lemma *imageE* [*elim!*]:
assumes $b \in (\lambda x. f x) \text{ ‘ } A$ — The eta-expansion gives variable-name preservation.
obtains x **where** $b = f x$ **and** $x \in A$
using *assms* **unfolding** *image-def* **by** *blast*

lemma *Compr-image-eq*: $\{x \in f \text{ ‘ } A. P x\} = f \text{ ‘ } \{x \in A. P (f x)\}$
by *auto*

lemma *image-Un*: $f \text{ ‘ } (A \cup B) = f \text{ ‘ } A \cup f \text{ ‘ } B$
by *blast*

lemma *image-iff*: $z \in f \text{ ‘ } A \iff (\exists x \in A. z = f x)$
by *blast*

lemma *image-subsetI*: $(\bigwedge x. x \in A \implies f x \in B) \implies f ' A \subseteq B$

— Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.

by *blast*

lemma *image-subset-iff*: $f ' A \subseteq B \longleftrightarrow (\forall x \in A. f x \in B)$

— This rewrite rule would confuse users if made default.

by *blast*

lemma *subset-imageE*:

assumes $B \subseteq f ' A$

obtains C where $C \subseteq A$ and $B = f ' C$

proof —

from *assms* have $B = f ' \{a \in A. f a \in B\}$ by *fast*

moreover have $\{a \in A. f a \in B\} \subseteq A$ by *blast*

ultimately show *thesis* by (*blast intro: that*)

qed

lemma *subset-image-iff*: $B \subseteq f ' A \longleftrightarrow (\exists AA \subseteq A. B = f ' AA)$

by (*blast elim: subset-imageE*)

lemma *image-ident* [*simp*]: $(\lambda x. x) ' Y = Y$

by *blast*

lemma *image-empty* [*simp*]: $f ' \{\} = \{\}$

by *blast*

lemma *image-insert* [*simp*]: $f ' \text{insert } a B = \text{insert } (f a) (f ' B)$

by *blast*

lemma *image-constant*: $x \in A \implies (\lambda x. c) ' A = \{c\}$

by *auto*

lemma *image-constant-conv*: $(\lambda x. c) ' A = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \{c\})$

by *auto*

lemma *image-image*: $f ' (g ' A) = (\lambda x. f (g x)) ' A$

by *blast*

lemma *insert-image* [*simp*]: $x \in A \implies \text{insert } (f x) (f ' A) = f ' A$

by *blast*

lemma *image-is-empty* [*iff*]: $f ' A = \{\} \longleftrightarrow A = \{\}$

by *blast*

lemma *empty-is-image* [*iff*]: $\{\} = f ' A \longleftrightarrow A = \{\}$

by *blast*

lemma *image-Collect*: $f \text{ ‘ } \{x. P x\} = \{f x \mid x. P x\}$

— NOT suitable as a default simp rule: the RHS isn’t simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.

by *blast*

lemma *if-image-distrib* [*simp*]:

$(\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) \text{ ‘ } S = f \text{ ‘ } (S \cap \{x. P x\}) \cup g \text{ ‘ } (S \cap \{x. \neg P x\})$

by *auto*

lemma *image-cong*:

$f \text{ ‘ } M = g \text{ ‘ } N \text{ if } M = N \wedge x. x \in N \implies f x = g x$

using *that* **by** (*simp add: image-def*)

lemma *image-cong-simp* [*cong*]:

$f \text{ ‘ } M = g \text{ ‘ } N \text{ if } M = N \wedge x. x \in N = \text{simp} \implies f x = g x$

using *that image-cong* [*of M N f g*] **by** (*simp add: simp-implies-def*)

lemma *image-Int-subset*: $f \text{ ‘ } (A \cap B) \subseteq f \text{ ‘ } A \cap f \text{ ‘ } B$

by *blast*

lemma *image-diff-subset*: $f \text{ ‘ } A - f \text{ ‘ } B \subseteq f \text{ ‘ } (A - B)$

by *blast*

lemma *Setcompr-eq-image*: $\{f x \mid x. x \in A\} = f \text{ ‘ } A$

by *blast*

lemma *setcompr-eq-image*: $\{f x \mid x. P x\} = f \text{ ‘ } \{x. P x\}$

by *auto*

lemma *ball-imageD*: $\forall x \in f \text{ ‘ } A. P x \implies \forall x \in A. P (f x)$

by *simp*

lemma *bex-imageD*: $\exists x \in f \text{ ‘ } A. P x \implies \exists x \in A. P (f x)$

by *auto*

lemma *image-add-0* [*simp*]: $(+) (0::'a::\text{comm-monoid-add}) \text{ ‘ } S = S$

by *auto*

theorem *Cantors-theorem*: $\nexists f. f \text{ ‘ } A = \text{Pow } A$

proof

assume $\exists f. f \text{ ‘ } A = \text{Pow } A$

then obtain f **where** $f: f \text{ ‘ } A = \text{Pow } A$..

let $?X = \{a \in A. a \notin f a\}$

have $?X \in \text{Pow } A$ **by** *blast*

then have $?X \in f \text{ ‘ } A$ **by** (*simp only: f*)

then obtain x **where** $x \in A$ **and** $f x = ?X$ **by** *blast*

then show *False* **by** *blast*

qed

Range of a function – just an abbreviation for image!

abbreviation $range :: ('a \Rightarrow 'b) \Rightarrow 'b\ set$ — of function
where $range\ f \equiv f\ 'UNIV$

lemma $range\ eqI: b = f\ x \Longrightarrow b \in range\ f$
by $simp$

lemma $rangeI: f\ x \in range\ f$
by $simp$

lemma $rangeE$ [$elim?$]: $b \in range\ (\lambda x. f\ x) \Longrightarrow (\bigwedge x. b = f\ x \Longrightarrow P) \Longrightarrow P$
by ($rule\ imageE$)

lemma $range\ subsetD: range\ f \subseteq B \Longrightarrow f\ i \in B$
by $blast$

lemma $full\ SetCompr\ eq: \{u. \exists x. u = f\ x\} = range\ f$
by $auto$

lemma $range\ composition: range\ (\lambda x. f\ (g\ x)) = f\ 'range\ g$
by $auto$

lemma $range\ constant$ [$simp$]: $range\ (\lambda -. x) = \{x\}$
by ($simp\ add: image\ constant$)

lemma $range\ eq\ singletonD: range\ f = \{a\} \Longrightarrow f\ x = a$
by $auto$

8.3.13 Some rules with *if*

Elimination of $\{x. \dots \wedge x = t \wedge \dots\}$.

lemma $Collect\ conv\ if: \{x. x = a \wedge P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$
by $auto$

lemma $Collect\ conv\ if2: \{x. a = x \wedge P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$
by $auto$

Rewrite rules for boolean case-splitting: faster than *if-split* [$split$].

lemma $if\ split\ eq1: (if\ Q\ then\ x\ else\ y) = b \longleftrightarrow (Q \longrightarrow x = b) \wedge (\neg Q \longrightarrow y = b)$
by ($rule\ if\ split$)

lemma $if\ split\ eq2: a = (if\ Q\ then\ x\ else\ y) \longleftrightarrow (Q \longrightarrow a = x) \wedge (\neg Q \longrightarrow a = y)$
by ($rule\ if\ split$)

Split ifs on either side of the membership relation. Not for [$simp$] – can cause goals to blow up!

lemma *if-split-mem1*: $(\text{if } Q \text{ then } x \text{ else } y) \in b \longleftrightarrow (Q \longrightarrow x \in b) \wedge (\neg Q \longrightarrow y \in b)$

by (*rule if-split*)

lemma *if-split-mem2*: $(a \in (\text{if } Q \text{ then } x \text{ else } y)) \longleftrightarrow (Q \longrightarrow a \in x) \wedge (\neg Q \longrightarrow a \in y)$

by (*rule if-split [where $P = \lambda S. a \in S$]*)

lemmas *split-ifs = if-bool-eq-conj if-split-eq1 if-split-eq2 if-split-mem1 if-split-mem2*

8.4 Further operations and lemmas

8.4.1 The “proper subset” relation

lemma *psubsetI [intro!]*: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$

unfolding *less-le* **by** *blast*

lemma *psubsetE [elim!]*: $A \subset B \Longrightarrow (A \subseteq B \Longrightarrow \neg B \subseteq A \Longrightarrow R) \Longrightarrow R$

unfolding *less-le* **by** *blast*

lemma *psubset-insert-iff*:

$A \subset \text{insert } x \ B \longleftrightarrow (\text{if } x \in B \text{ then } A \subset B \text{ else if } x \in A \text{ then } A - \{x\} \subset B \text{ else } A \subseteq B)$

by (*auto simp add: less-le subset-insert-iff*)

lemma *psubset-eq*: $A \subset B \longleftrightarrow A \subseteq B \wedge A \neq B$

by (*simp only: less-le*)

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$

by (*simp add: psubset-eq*)

lemma *psubset-trans*: $A \subset B \Longrightarrow B \subset C \Longrightarrow A \subset C$

unfolding *less-le* **by** (*auto dest: subset-antisym*)

lemma *psubsetD*: $A \subset B \Longrightarrow c \in A \Longrightarrow c \in B$

unfolding *less-le* **by** (*auto dest: subsetD*)

lemma *psubset-subset-trans*: $A \subset B \Longrightarrow B \subseteq C \Longrightarrow A \subset C$

by (*auto simp add: psubset-eq*)

lemma *subset-psubset-trans*: $A \subseteq B \Longrightarrow B \subset C \Longrightarrow A \subset C$

by (*auto simp add: psubset-eq*)

lemma *psubset-imp-ex-mem*: $A \subset B \Longrightarrow \exists b. b \in B - A$

unfolding *less-le* **by** *blast*

lemma *atomize-ball*: $(\bigwedge x. x \in A \Longrightarrow P x) \equiv \text{Trueprop } (\forall x \in A. P x)$

by (*simp only: Ball-def atomize-all atomize-imp*)

lemmas [*symmetric, rulify*] = *atomize-ball*

and [*symmetric, defn*] = *atomize-ball*

lemma *image-Pow-mono*: $f \text{ ' } A \subseteq B \implies \text{image } f \text{ ' } \text{Pow } A \subseteq \text{Pow } B$
by *blast*

lemma *image-Pow-surj*: $f \text{ ' } A = B \implies \text{image } f \text{ ' } \text{Pow } A = \text{Pow } B$
by (*blast elim: subset-imageE*)

8.4.2 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq \text{insert } a \ B$
by (*rule subsetI*) (*erule insertI2*)

lemma *subset-insertI2*: $A \subseteq B \implies A \subseteq \text{insert } b \ B$
by *blast*

lemma *subset-insert*: $x \notin A \implies A \subseteq \text{insert } x \ B \longleftrightarrow A \subseteq B$
by *blast*

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
by (*fact sup-ge1*)

lemma *Un-upper2*: $B \subseteq A \cup B$
by (*fact sup-ge2*)

lemma *Un-least*: $A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C$
by (*fact sup-least*)

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
by (*fact inf-le1*)

lemma *Int-lower2*: $A \cap B \subseteq B$
by (*fact inf-le2*)

lemma *Int-greatest*: $C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B$
by (*fact inf-greatest*)

Set difference.

lemma *Diff-subset[simp]*: $A - B \subseteq A$
by *blast*

lemma *Diff-subset-conv*: $A - B \subseteq C \longleftrightarrow A \subseteq B \cup C$
by *blast*

8.4.3 Equalities involving union, intersection, inclusion, etc.

$\{\}$.

lemma *Collect-const* [*simp*]: $\{s. P\} = (\text{if } P \text{ then } \text{UNIV} \text{ else } \{\})$
 — supersedes *Collect-False-empty*
 by *auto*

lemma *subset-empty* [*simp*]: $A \subseteq \{\} \longleftrightarrow A = \{\}$
 by (*fact bot-unique*)

lemma *not-psubset-empty* [*iff*]: $\neg (A < \{\})$
 by (*fact not-less-bot*)

lemma *Collect-subset* [*simp*]: $\{x \in A. P\ x\} \subseteq A$ by *auto*

lemma *Collect-empty-eq* [*simp*]: $\text{Collect } P = \{\} \longleftrightarrow (\forall x. \neg P\ x)$
 by *blast*

lemma *empty-Collect-eq* [*simp*]: $\{\} = \text{Collect } P \longleftrightarrow (\forall x. \neg P\ x)$
 by *blast*

lemma *Collect-neg-eq*: $\{x. \neg P\ x\} = - \{x. P\ x\}$
 by *blast*

lemma *Collect-disj-eq*: $\{x. P\ x \vee Q\ x\} = \{x. P\ x\} \cup \{x. Q\ x\}$
 by *blast*

lemma *Collect-imp-eq*: $\{x. P\ x \longrightarrow Q\ x\} = - \{x. P\ x\} \cup \{x. Q\ x\}$
 by *blast*

lemma *Collect-conj-eq*: $\{x. P\ x \wedge Q\ x\} = \{x. P\ x\} \cap \{x. Q\ x\}$
 by *blast*

lemma *Collect-mono-iff*: $\text{Collect } P \subseteq \text{Collect } Q \longleftrightarrow (\forall x. P\ x \longrightarrow Q\ x)$
 by *blast*

insert.

lemma *insert-is-Un*: $\text{insert } a\ A = \{a\} \cup A$
 — NOT SUITABLE FOR REWRITING since $\{a\} \equiv \text{insert } a\ \{\}$
 by *blast*

lemma *insert-not-empty* [*simp*]: $\text{insert } a\ A \neq \{\}$
and *empty-not-insert* [*simp*]: $\{\} \neq \text{insert } a\ A$
 by *blast+*

lemma *insert-absorb*: $a \in A \implies \text{insert } a\ A = A$
 — [*simp*] causes recursive calls when there are nested inserts
 — with *quadratic* running time
 by *blast*

lemma *insert-absorb2* [*simp*]: $insert\ x\ (insert\ x\ A) = insert\ x\ A$
by *blast*

lemma *insert-commute*: $insert\ x\ (insert\ y\ A) = insert\ y\ (insert\ x\ A)$
by *blast*

lemma *insert-subset* [*simp*]: $insert\ x\ A \subseteq B \longleftrightarrow x \in B \wedge A \subseteq B$
by *blast*

lemma *mk-disjoint-insert*: $a \in A \implies \exists B. A = insert\ a\ B \wedge a \notin B$
— use new B rather than $A - \{a\}$ to avoid infinite unfolding
by (*rule exI* [**where** $x = A - \{a\}$]) *blast*

lemma *insert-Collect*: $insert\ a\ (Collect\ P) = \{u. u \neq a \longrightarrow P\ u\}$
by *auto*

lemma *insert-inter-insert* [*simp*]: $insert\ a\ A \cap insert\ a\ B = insert\ a\ (A \cap B)$
by *blast*

lemma *insert-disjoint* [*simp*]:
 $insert\ a\ A \cap B = \{\} \longleftrightarrow a \notin B \wedge A \cap B = \{\}$
 $\{\} = insert\ a\ A \cap B \longleftrightarrow a \notin B \wedge \{\} = A \cap B$
by *auto*

lemma *disjoint-insert* [*simp*]:
 $B \cap insert\ a\ A = \{\} \longleftrightarrow a \notin B \wedge B \cap A = \{\}$
 $\{\} = A \cap insert\ b\ B \longleftrightarrow b \notin A \wedge \{\} = A \cap B$
by *auto*

Int

lemma *Int-absorb*: $A \cap A = A$
by (*fact inf-idem*)

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$
by (*fact inf-left-idem*)

lemma *Int-commute*: $A \cap B = B \cap A$
by (*fact inf-commute*)

lemma *Int-left-commute*: $A \cap (B \cap C) = B \cap (A \cap C)$
by (*fact inf-left-commute*)

lemma *Int-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
by (*fact inf-assoc*)

lemmas *Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute*
— Intersection is an AC-operator

lemma *Int-absorb1*: $B \subseteq A \implies A \cap B = B$
by (*fact inf-absorb2*)

lemma *Int-absorb2*: $A \subseteq B \implies A \cap B = A$
by (*fact inf-absorb1*)

lemma *Int-empty-left*: $\{\} \cap B = \{\}$
by (*fact inf-bot-left*)

lemma *Int-empty-right*: $A \cap \{\} = \{\}$
by (*fact inf-bot-right*)

lemma *disjoint-eq-subset-Compl*: $A \cap B = \{\} \iff A \subseteq - B$
by *blast*

lemma *disjoint-iff*: $A \cap B = \{\} \iff (\forall x. x \in A \longrightarrow x \notin B)$
by *blast*

lemma *disjoint-iff-not-equal*: $A \cap B = \{\} \iff (\forall x \in A. \forall y \in B. x \neq y)$
by *blast*

lemma *Int-UNIV-left*: $UNIV \cap B = B$
by (*fact inf-top-left*)

lemma *Int-UNIV-right*: $A \cap UNIV = A$
by (*fact inf-top-right*)

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by (*fact inf-sup-distrib1*)

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
by (*fact inf-sup-distrib2*)

lemma *Int-UNIV [simp]*: $A \cap B = UNIV \iff A = UNIV \wedge B = UNIV$
by (*fact inf-eq-top-iff*)

lemma *Int-subset-iff [simp]*: $C \subseteq A \cap B \iff C \subseteq A \wedge C \subseteq B$
by (*fact le-inf-iff*)

lemma *Int-Collect*: $x \in A \cap \{x. P x\} \iff x \in A \wedge P x$
by *blast*

Un.

lemma *Un-absorb*: $A \cup A = A$
by (*fact sup-idem*)

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
by (*fact sup-left-idem*)

lemma *Un-commute*: $A \cup B = B \cup A$
by (*fact sup-commute*)

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
by (*fact sup-left-commute*)

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
by (*fact sup-assoc*)

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*
 — Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B \implies A \cup B = B$
by (*fact sup-absorb2*)

lemma *Un-absorb2*: $B \subseteq A \implies A \cup B = A$
by (*fact sup-absorb1*)

lemma *Un-empty-left*: $\{\} \cup B = B$
by (*fact sup-bot-left*)

lemma *Un-empty-right*: $A \cup \{\} = A$
by (*fact sup-bot-right*)

lemma *Un-UNIV-left*: $UNIV \cup B = UNIV$
by (*fact sup-top-left*)

lemma *Un-UNIV-right*: $A \cup UNIV = UNIV$
by (*fact sup-top-right*)

lemma *Un-insert-left [simp]*: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
by *blast*

lemma *Un-insert-right [simp]*: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
by *blast*

lemma *Int-insert-left*: $(insert\ a\ B) \cap C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
by *auto*

lemma *Int-insert-left-if0 [simp]*: $a \notin C \implies (insert\ a\ B) \cap C = B \cap C$
by *auto*

lemma *Int-insert-left-if1 [simp]*: $a \in C \implies (insert\ a\ B) \cap C = insert\ a\ (B \cap C)$
by *auto*

lemma *Int-insert-right*: $A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
by *auto*

lemma *Int-insert-right-if0* [*simp*]: $a \notin A \implies A \cap (\text{insert } a \ B) = A \cap B$
by *auto*

lemma *Int-insert-right-if1* [*simp*]: $a \in A \implies A \cap (\text{insert } a \ B) = \text{insert } a \ (A \cap B)$
by *auto*

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by (*fact sup-inf-distrib1*)

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
by (*fact sup-inf-distrib2*)

lemma *Un-Int-crazy*: $(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
by *blast*

lemma *subset-Un-eq*: $A \subseteq B \longleftrightarrow A \cup B = B$
by (*fact le-iff-sup*)

lemma *Un-empty* [*iff*]: $A \cup B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
by (*fact sup-eq-bot-iff*)

lemma *Un-subset-iff* [*simp*]: $A \cup B \subseteq C \longleftrightarrow A \subseteq C \wedge B \subseteq C$
by (*fact le-sup-iff*)

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
by *blast*

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
by *blast*

lemma *subset-UnE*:

assumes $C \subseteq A \cup B$

obtains $A' \ B'$ **where** $A' \subseteq A \ B' \subseteq B \ C = A' \cup B'$

proof

show $C \cap A \subseteq A \ C \cap B \subseteq B \ C = (C \cap A) \cup (C \cap B)$

using *assms* **by** *blast+*

qed

lemma *Un-Int-eq* [*simp*]: $(S \cup T) \cap S = S \ (S \cup T) \cap T = T \ S \cap (S \cup T) = S$
 $T \cap (S \cup T) = T$
by *auto*

lemma *Int-Un-eq* [*simp*]: $(S \cap T) \cup S = S \ (S \cap T) \cup T = T \ S \cup (S \cap T) = S$
 $T \cup (S \cap T) = T$
by *auto*

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap - A = \{\}$
by (*fact inf-compl-bot*)

lemma *Compl-disjoint2* [*simp*]: $- A \cap A = \{\}$
by (*fact compl-inf-bot*)

lemma *Compl-partition*: $A \cup - A = UNIV$
by (*fact sup-compl-top*)

lemma *Compl-partition2*: $- A \cup A = UNIV$
by (*fact compl-sup-top*)

lemma *double-complement*: $- (-A) = A$ **for** $A :: 'a \text{ set}$
by (*fact double-compl*)

lemma *Compl-Un*: $- (A \cup B) = (- A) \cap (- B)$
by (*fact compl-sup*)

lemma *Compl-Int*: $- (A \cap B) = (- A) \cup (- B)$
by (*fact compl-inf*)

lemma *subset-Compl-self-eq*: $A \subseteq - A \longleftrightarrow A = \{\}$
by *blast*

lemma *Un-Int-assoc-eq*: $(A \cap B) \cup C = A \cap (B \cup C) \longleftrightarrow C \subseteq A$
— Halmos, Naive Set Theory, page 16.
by *blast*

lemma *Compl-UNIV-eq*: $- UNIV = \{\}$
by (*fact compl-top-eq*)

lemma *Compl-empty-eq*: $- \{\} = UNIV$
by (*fact compl-bot-eq*)

lemma *Compl-subset-Compl-iff* [*iff*]: $- A \subseteq - B \longleftrightarrow B \subseteq A$
by (*fact compl-le-compl-iff*)

lemma *Compl-eq-Compl-iff* [*iff*]: $- A = - B \longleftrightarrow A = B$
for $A B :: 'a \text{ set}$
by (*fact compl-eq-compl-iff*)

lemma *Compl-insert*: $- \text{insert } x A = (- A) - \{x\}$
by *blast*

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

lemma *ball-Un*: $(\forall x \in A \cup B. P x) \longleftrightarrow (\forall x \in A. P x) \wedge (\forall x \in B. P x)$
by *blast*

lemma *beX-Un*: $(\exists x \in A \cup B. P x) \longleftrightarrow (\exists x \in A. P x) \vee (\exists x \in B. P x)$
by *blast*

Set difference.

lemma *Diff-eq*: $A - B = A \cap (- B)$
by *blast*

lemma *Diff-eq-empty-iff* [*simp*]: $A - B = \{\} \longleftrightarrow A \subseteq B$
by *blast*

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
by *blast*

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - B$
for $A B :: 'a \text{ set}$
by *blast*

lemma *Diff-triv*: $A \cap B = \{\} \implies A - B = A$
by (*blast elim: equalityE*)

lemma *empty-Diff* [*simp*]: $\{\} - A = \{\}$
by *blast*

lemma *Diff-empty* [*simp*]: $A - \{\} = A$
by *blast*

lemma *Diff-UNIV* [*simp*]: $A - UNIV = \{\}$
by *blast*

lemma *Diff-insert0* [*simp*]: $x \notin A \implies A - \text{insert } x B = A - B$
by *blast*

lemma *Diff-insert*: $A - \text{insert } a B = A - B - \{a\}$
 — NOT SUITABLE FOR REWRITING since $\{a\} \equiv \text{insert } a 0$
by *blast*

lemma *Diff-insert2*: $A - \text{insert } a B = A - \{a\} - B$
 — NOT SUITABLE FOR REWRITING since $\{a\} \equiv \text{insert } a 0$
by *blast*

lemma *insert-Diff-if*: $\text{insert } x A - B = (\text{if } x \in B \text{ then } A - B \text{ else } \text{insert } x (A - B))$
by *auto*

lemma *insert-Diff1* [*simp*]: $x \in B \implies \text{insert } x A - B = A - B$
by *blast*

lemma *insert-Diff-single*[simp]: $\text{insert } a (A - \{a\}) = \text{insert } a A$
by *blast*

lemma *insert-Diff*: $a \in A \implies \text{insert } a (A - \{a\}) = A$
by *blast*

lemma *Diff-insert-absorb*: $x \notin A \implies (\text{insert } x A) - \{x\} = A$
by *auto*

lemma *Diff-disjoint* [simp]: $A \cap (B - A) = \{\}$
by *blast*

lemma *Diff-partition*: $A \subseteq B \implies A \cup (B - A) = B$
by *blast*

lemma *double-diff*: $A \subseteq B \implies B \subseteq C \implies B - (C - A) = A$
by *blast*

lemma *Un-Diff-cancel* [simp]: $A \cup (B - A) = A \cup B$
by *blast*

lemma *Un-Diff-cancel2* [simp]: $(B - A) \cup A = B \cup A$
by *blast*

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$
by *blast*

lemma *Diff-Int*: $A - (B \cap C) = (A - B) \cup (A - C)$
by *blast*

lemma *Diff-Diff-Int*: $A - (A - B) = A \cap B$
by *blast*

lemma *Un-Diff*: $(A \cup B) - C = (A - C) \cup (B - C)$
by *blast*

lemma *Int-Diff*: $(A \cap B) - C = A \cap (B - C)$
by *blast*

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$
by *blast*

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
by *blast*

lemma *Diff-Compl* [simp]: $A - (- B) = A \cap B$
by *auto*

lemma *Compl-Diff-eq* [simp]: $\neg (A - B) = \neg A \cup B$
by *blast*

lemma *subset-Compl-singleton* [simp]: $A \subseteq \neg \{b\} \longleftrightarrow b \notin A$
by *blast*

Quantification over type *bool*.

lemma *bool-induct*: $P \text{ True} \Longrightarrow P \text{ False} \Longrightarrow P x$
by (*cases x*) *auto*

lemma *all-bool-eq*: $(\forall b. P b) \longleftrightarrow P \text{ True} \wedge P \text{ False}$
by (*auto intro: bool-induct*)

lemma *bool-contrapos*: $P x \Longrightarrow \neg P \text{ False} \Longrightarrow P \text{ True}$
by (*cases x*) *auto*

lemma *ex-bool-eq*: $(\exists b. P b) \longleftrightarrow P \text{ True} \vee P \text{ False}$
by (*auto intro: bool-contrapos*)

lemma *UNIV-bool*: $\text{UNIV} = \{\text{False}, \text{True}\}$
by (*auto intro: bool-induct*)

Pow

lemma *Pow-empty* [simp]: $\text{Pow } \{\} = \{\{\}\}$
by (*auto simp add: Pow-def*)

lemma *Pow-singleton-iff* [simp]: $\text{Pow } X = \{Y\} \longleftrightarrow X = \{\} \wedge Y = \{\}$
by *blast*

lemma *Pow-insert*: $\text{Pow } (\text{insert } a \ A) = \text{Pow } A \cup (\text{insert } a \ \text{'Pow } A)$
by (*blast intro: image-eqI [where ?x = u - {a} for u]*)

lemma *Pow-Compl*: $\text{Pow } (\neg A) = \{\neg B \mid B. A \in \text{Pow } B\}$
by (*blast intro: exI [where ?x = - u for u]*)

lemma *Pow-UNIV* [simp]: $\text{Pow } \text{UNIV} = \text{UNIV}$
by *blast*

lemma *Un-Pow-subset*: $\text{Pow } A \cup \text{Pow } B \subseteq \text{Pow } (A \cup B)$
by *blast*

lemma *Pow-Int-eq* [simp]: $\text{Pow } (A \cap B) = \text{Pow } A \cap \text{Pow } B$
by *blast*

Miscellany.

lemma *Int-Diff-disjoint*: $A \cap B \cap (A - B) = \{\}$
by *blast*

lemma *Int-Diff-Un*: $A \cap B \cup (A - B) = A$

by *blast*

lemma *set-eq-subset*: $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$

by *blast*

lemma *subset-iff*: $A \subseteq B \longleftrightarrow (\forall t. t \in A \longrightarrow t \in B)$

by *blast*

lemma *subset-iff-psubset-eq*: $A \subseteq B \longleftrightarrow A \subset B \vee A = B$

unfolding *less-le* by *blast*

lemma *all-not-in-conv* [*simp*]: $(\forall x. x \notin A) \longleftrightarrow A = \{\}$

by *blast*

lemma *ex-in-conv*: $(\exists x. x \in A) \longleftrightarrow A \neq \{\}$

by *blast*

lemma *ball-simps* [*simp, no-atp*]:

$\bigwedge A P Q. (\forall x \in A. P x \vee Q) \longleftrightarrow ((\forall x \in A. P x) \vee Q)$

$\bigwedge A P Q. (\forall x \in A. P \vee Q x) \longleftrightarrow (P \vee (\forall x \in A. Q x))$

$\bigwedge A P Q. (\forall x \in A. P \longrightarrow Q x) \longleftrightarrow (P \longrightarrow (\forall x \in A. Q x))$

$\bigwedge A P Q. (\forall x \in A. P x \longrightarrow Q) \longleftrightarrow ((\exists x \in A. P x) \longrightarrow Q)$

$\bigwedge P. (\forall x \in \{\}. P x) \longleftrightarrow \text{True}$

$\bigwedge P. (\forall x \in \text{UNIV}. P x) \longleftrightarrow (\forall x. P x)$

$\bigwedge a B P. (\forall x \in \text{insert } a B. P x) \longleftrightarrow (P a \wedge (\forall x \in B. P x))$

$\bigwedge P Q. (\forall x \in \text{Collect } Q. P x) \longleftrightarrow (\forall x. Q x \longrightarrow P x)$

$\bigwedge A P f. (\forall x \in f'A. P x) \longleftrightarrow (\forall x \in A. P (f x))$

$\bigwedge A P. (\neg (\forall x \in A. P x)) \longleftrightarrow (\exists x \in A. \neg P x)$

by *auto*

lemma *bex-simps* [*simp, no-atp*]:

$\bigwedge A P Q. (\exists x \in A. P x \wedge Q) \longleftrightarrow ((\exists x \in A. P x) \wedge Q)$

$\bigwedge A P Q. (\exists x \in A. P \wedge Q x) \longleftrightarrow (P \wedge (\exists x \in A. Q x))$

$\bigwedge P. (\exists x \in \{\}. P x) \longleftrightarrow \text{False}$

$\bigwedge P. (\exists x \in \text{UNIV}. P x) \longleftrightarrow (\exists x. P x)$

$\bigwedge a B P. (\exists x \in \text{insert } a B. P x) \longleftrightarrow (P a \vee (\exists x \in B. P x))$

$\bigwedge P Q. (\exists x \in \text{Collect } Q. P x) \longleftrightarrow (\exists x. Q x \wedge P x)$

$\bigwedge A P f. (\exists x \in f'A. P x) \longleftrightarrow (\exists x \in A. P (f x))$

$\bigwedge A P. (\neg (\exists x \in A. P x)) \longleftrightarrow (\forall x \in A. \neg P x)$

by *auto*

lemma *ex-image-cong-iff* [*simp, no-atp*]:

$(\exists x. x \in f'A) \longleftrightarrow A \neq \{\} \quad (\exists x. x \in f'A \wedge P x) \longleftrightarrow (\exists x \in A. P (f x))$

by *auto*

8.4.4 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \implies f \text{ ' } A \subseteq f \text{ ' } B$
 by *blast*

lemma *Pow-mono*: $A \subseteq B \implies \text{Pow } A \subseteq \text{Pow } B$
 by *blast*

lemma *insert-mono*: $C \subseteq D \implies \text{insert } a \ C \subseteq \text{insert } a \ D$
 by *blast*

lemma *Un-mono*: $A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D$
 by (*fact sup-mono*)

lemma *Int-mono*: $A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D$
 by (*fact inf-mono*)

lemma *Diff-mono*: $A \subseteq C \implies D \subseteq B \implies A - B \subseteq C - D$
 by *blast*

lemma *Compl-anti-mono*: $A \subseteq B \implies - B \subseteq - A$
 by (*fact compl-mono*)

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B \implies x \in A \longrightarrow x \in B$
 by (*rule impI*) (*erule subsetD*)

lemma *conj-mono*: $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \wedge P2) \longrightarrow (Q1 \wedge Q2)$
 by *iprover*

lemma *disj-mono*: $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \vee P2) \longrightarrow (Q1 \vee Q2)$
 by *iprover*

lemma *imp-mono*: $Q1 \longrightarrow P1 \implies P2 \longrightarrow Q2 \implies (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2)$
 by *iprover*

lemma *imp-refl*: $P \longrightarrow P ..$

lemma *not-mono*: $Q \longrightarrow P \implies \neg P \longrightarrow \neg Q$
 by *iprover*

lemma *ex-mono*: $(\bigwedge x. P x \longrightarrow Q x) \implies (\exists x. P x) \longrightarrow (\exists x. Q x)$
 by *iprover*

lemma *all-mono*: $(\bigwedge x. P x \longrightarrow Q x) \implies (\forall x. P x) \longrightarrow (\forall x. Q x)$
 by *iprover*

lemma *Collect-mono*: $(\bigwedge x. P x \longrightarrow Q x) \implies \text{Collect } P \subseteq \text{Collect } Q$

by *blast*

lemma *Int-Collect-mono*: $A \subseteq B \implies (\bigwedge x. x \in A \implies P x \longrightarrow Q x) \implies A \cap \text{Collect } P \subseteq B \cap \text{Collect } Q$

by *blast*

lemmas *basic-monos* =

subset-refl imp-refl disj-mono conj-mono ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b \implies c = d \implies b \longrightarrow d \implies a \longrightarrow c$

by *iprover*

8.4.5 Inverse image of a function

definition *vimage* :: $('a \Rightarrow 'b) \Rightarrow 'b \text{ set} \Rightarrow 'a \text{ set}$ (**infixr** -' 90)

where $f -' B \equiv \{x. f x \in B\}$

lemma *vimage-eq [simp]*: $a \in f -' B \longleftrightarrow f a \in B$

unfolding *vimage-def* by *blast*

lemma *vimage-singleton-eq*: $a \in f -' \{b\} \longleftrightarrow f a = b$

by *simp*

lemma *vimageI [intro]*: $f a = b \implies b \in B \implies a \in f -' B$

unfolding *vimage-def* by *blast*

lemma *vimageI2*: $f a \in A \implies a \in f -' A$

unfolding *vimage-def* by *fast*

lemma *vimageE [elim!]*: $a \in f -' B \implies (\bigwedge x. f a = x \implies x \in B \implies P) \implies P$

unfolding *vimage-def* by *blast*

lemma *vimageD*: $a \in f -' A \implies f a \in A$

unfolding *vimage-def* by *fast*

lemma *vimage-empty [simp]*: $f -' \{\} = \{\}$

by *blast*

lemma *vimage-Compl*: $f -' (- A) = - (f -' A)$

by *blast*

lemma *vimage-Un [simp]*: $f -' (A \cup B) = (f -' A) \cup (f -' B)$

by *blast*

lemma *vimage-Int [simp]*: $f -' (A \cap B) = (f -' A) \cap (f -' B)$

by *fast*

lemma *vimage-Collect-eq [simp]*: $f -' \text{Collect } P = \{y. P (f y)\}$

by *blast*

lemma *vimage-Collect*: $(\bigwedge x. P (f x) = Q x) \implies f -' (Collect P) = Collect Q$
by *blast*

lemma *vimage-insert*: $f -' (insert a B) = (f -' \{a\}) \cup (f -' B)$
 — NOT suitable for rewriting because of the recurrence of $\{a\}$.
by *blast*

lemma *vimage-Diff*: $f -' (A - B) = (f -' A) - (f -' B)$
by *blast*

lemma *vimage-UNIV [simp]*: $f -' UNIV = UNIV$
by *blast*

lemma *vimage-mono*: $A \subseteq B \implies f -' A \subseteq f -' B$
 — monotonicity
by *blast*

lemma *vimage-image-eq*: $f -' (f ' A) = \{y. \exists x \in A. f x = f y\}$
by (*blast intro: sym*)

lemma *image-vimage-subset*: $f ' (f -' A) \subseteq A$
by *blast*

lemma *image-vimage-eq [simp]*: $f ' (f -' A) = A \cap range f$
by *blast*

lemma *image-subset-iff-subset-vimage*: $f ' A \subseteq B \longleftrightarrow A \subseteq f -' B$
by *blast*

lemma *subset-vimage-iff*: $A \subseteq f -' B \longleftrightarrow (\forall x \in A. f x \in B)$
by *auto*

lemma *vimage-const [simp]*: $((\lambda x. c) -' A) = (if c \in A then UNIV else \{\})$
by *auto*

lemma *vimage-if [simp]*: $((\lambda x. if x \in B then c else d) -' A) =$
 $(if c \in A then (if d \in A then UNIV else B)$
 $else if d \in A then - B else \{\})$
by (*auto simp add: vimage-def*)

lemma *vimage-inter-cong*: $(\bigwedge w. w \in S \implies f w = g w) \implies f -' y \cap S = g -' y \cap S$
by *auto*

lemma *vimage-ident [simp]*: $(\lambda x. x) -' Y = Y$
by *blast*

8.4.6 Singleton sets

definition *is-singleton* :: 'a set \Rightarrow bool
 where *is-singleton* A \longleftrightarrow ($\exists x. A = \{x\}$)

lemma *is-singletonI* [*simp*, *intro!*]: *is-singleton* {x}
unfolding *is-singleton-def* **by** *simp*

lemma *is-singletonI'*: $A \neq \{\}$ \Longrightarrow ($\bigwedge x y. x \in A \Longrightarrow y \in A \Longrightarrow x = y$) \Longrightarrow
is-singleton A
unfolding *is-singleton-def* **by** *blast*

lemma *is-singletonE*: *is-singleton* A \Longrightarrow ($\bigwedge x. A = \{x\} \Longrightarrow P$) \Longrightarrow P
unfolding *is-singleton-def* **by** *blast*

8.4.7 Getting the contents of a singleton set

definition *the-elem* :: 'a set \Rightarrow 'a
 where *the-elem* X = (*THE* x. X = {x})

lemma *the-elem-eq* [*simp*]: *the-elem* {x} = x
by (*simp add: the-elem-def*)

lemma *is-singleton-the-elem*: *is-singleton* A \longleftrightarrow A = {*the-elem* A}
by (*auto simp: is-singleton-def*)

lemma *the-elem-image-unique*:
assumes $A \neq \{\}$
and *: $\bigwedge y. y \in A \Longrightarrow f y = f x$
shows *the-elem* (f 'A) = f x
unfolding *the-elem-def*
proof (*rule the1-equality*)
from $\langle A \neq \{\} \rangle$ **obtain** y **where** $y \in A$ **by** *auto*
with * **have** $f x = f y$ **by** *simp*
with $\langle y \in A \rangle$ **have** $f x \in f 'A$ **by** *blast*
with * **show** $f 'A = \{f x\}$ **by** *auto*
then show $\exists! x. f 'A = \{x\}$ **by** *auto*
qed

8.4.8 Monad operation

definition *bind* :: 'a set \Rightarrow ('a \Rightarrow 'b set) \Rightarrow 'b set
 where *bind* A f = {x. $\exists B \in f'A. x \in B$ }

hide-const (**open**) *bind*

lemma *bind-bind*: *Set.bind* (*Set.bind* A B) C = *Set.bind* A ($\lambda x. \text{Set.bind } (B x)$ C)
for A :: 'a set
by (*auto simp: bind-def*)

lemma *empty-bind* [*simp*]: $Set.bind \{\} f = \{\}$
 by (*simp add: bind-def*)

lemma *nonempty-bind-const*: $A \neq \{\} \implies Set.bind A (\lambda-. B) = B$
 by (*auto simp: bind-def*)

lemma *bind-const*: $Set.bind A (\lambda-. B) = (if A = \{\} then \{\} else B)$
 by (*auto simp: bind-def*)

lemma *bind-singleton-conv-image*: $Set.bind A (\lambda x. \{f x\}) = f ` A$
 by (*auto simp: bind-def*)

8.4.9 Operations for execution

definition *is-empty* :: $'a set \Rightarrow bool$
 where [*code-abbrev*]: $is_empty A \longleftrightarrow A = \{\}$

hide-const (**open**) *is-empty*

definition *remove* :: $'a \Rightarrow 'a set \Rightarrow 'a set$
 where [*code-abbrev*]: $remove x A = A - \{x\}$

hide-const (**open**) *remove*

lemma *member-remove* [*simp*]: $x \in Set.remove y A \longleftrightarrow x \in A \wedge x \neq y$
 by (*simp add: remove-def*)

definition *filter* :: $('a \Rightarrow bool) \Rightarrow 'a set \Rightarrow 'a set$
 where [*code-abbrev*]: $filter P A = \{a \in A. P a\}$

hide-const (**open**) *filter*

lemma *member-filter* [*simp*]: $x \in Set.filter P A \longleftrightarrow x \in A \wedge P x$
 by (*simp add: filter-def*)

instantiation *set* :: (*equal*) *equal*
 begin

definition *HOL.equal* $A B \longleftrightarrow A \subseteq B \wedge B \subseteq A$

instance by *standard* (*auto simp add: equal-set-def*)

end

Misc

definition *pairwise* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a set \Rightarrow bool$
 where $pairwise R S \longleftrightarrow (\forall x \in S. \forall y \in S. x \neq y \longrightarrow R x y)$

lemma *pairwise-alt*: *pairwise* $R S \longleftrightarrow (\forall x \in S. \forall y \in S - \{x\}. R x y)$
by (*auto simp add: pairwise-def*)

lemma *pairwise-trivial* [*simp*]: *pairwise* $(\lambda i j. j \neq i) I$
by (*auto simp: pairwise-def*)

lemma *pairwiseI* [*intro?*]:
pairwise $R S$ **if** $\bigwedge x y. x \in S \implies y \in S \implies x \neq y \implies R x y$
using that by (*simp add: pairwise-def*)

lemma *pairwiseD*:
 $R x y$ **and** $R y x$
if *pairwise* $R S$ $x \in S$ **and** $y \in S$ **and** $x \neq y$
using that by (*simp-all add: pairwise-def*)

lemma *pairwise-empty* [*simp*]: *pairwise* $P \{\}$
by (*simp add: pairwise-def*)

lemma *pairwise-singleton* [*simp*]: *pairwise* $P \{A\}$
by (*simp add: pairwise-def*)

lemma *pairwise-insert*:
pairwise r (*insert* $x s$) $\longleftrightarrow (\forall y. y \in s \wedge y \neq x \longrightarrow r x y \wedge r y x) \wedge$ *pairwise* $r s$
by (*force simp: pairwise-def*)

lemma *pairwise-subset*: *pairwise* $P S \implies T \subseteq S \implies$ *pairwise* $P T$
by (*force simp: pairwise-def*)

lemma *pairwise-mono*: \llbracket *pairwise* $P A; \bigwedge x y. P x y \implies Q x y; B \subseteq A \rrbracket \implies$ *pairwise* $Q B$
by (*fastforce simp: pairwise-def*)

lemma *pairwise-imageI*:
pairwise $P (f \text{' } A)$
if $\bigwedge x y. x \in A \implies y \in A \implies x \neq y \implies f x \neq f y \implies P (f x) (f y)$
using that by (*auto intro: pairwiseI*)

lemma *pairwise-image*: *pairwise* $r (f \text{' } s) \longleftrightarrow$ *pairwise* $(\lambda x y. (f x \neq f y) \longrightarrow r (f x) (f y)) s$
by (*force simp: pairwise-def*)

definition *disjnt* :: 'a set \Rightarrow 'a set \Rightarrow bool
where *disjnt* $A B \longleftrightarrow A \cap B = \{\}$

lemma *disjnt-self-iff-empty* [*simp*]: *disjnt* $S S \longleftrightarrow S = \{\}$
by (*auto simp: disjnt-def*)

lemma *disjnt-iff*: *disjnt* $A B \longleftrightarrow (\forall x. \neg (x \in A \wedge x \in B))$
by (*force simp: disjnt-def*)

lemma *disjnt-sym*: $disjnt\ A\ B \implies disjnt\ B\ A$
using *disjnt-iff* **by** *blast*

lemma *disjnt-empty1* [*simp*]: $disjnt\ \{\}\ A$ **and** *disjnt-empty2* [*simp*]: $disjnt\ A\ \{\}$
by (*auto simp: disjnt-def*)

lemma *disjnt-insert1* [*simp*]: $disjnt\ (insert\ a\ X)\ Y \longleftrightarrow a \notin Y \wedge disjnt\ X\ Y$
by (*simp add: disjnt-def*)

lemma *disjnt-insert2* [*simp*]: $disjnt\ Y\ (insert\ a\ X) \longleftrightarrow a \notin Y \wedge disjnt\ Y\ X$
by (*simp add: disjnt-def*)

lemma *disjnt-subset1* : $\llbracket disjnt\ X\ Y; Z \subseteq X \rrbracket \implies disjnt\ Z\ Y$
by (*auto simp: disjnt-def*)

lemma *disjnt-subset2* : $\llbracket disjnt\ X\ Y; Z \subseteq Y \rrbracket \implies disjnt\ X\ Z$
by (*auto simp: disjnt-def*)

lemma *disjnt-Un1* [*simp*]: $disjnt\ (A \cup B)\ C \longleftrightarrow disjnt\ A\ C \wedge disjnt\ B\ C$
by (*auto simp: disjnt-def*)

lemma *disjnt-Un2* [*simp*]: $disjnt\ C\ (A \cup B) \longleftrightarrow disjnt\ C\ A \wedge disjnt\ C\ B$
by (*auto simp: disjnt-def*)

lemma *disjnt-Diff1*: $disjnt\ (X - Y)\ (U - V)$ **and** *disjnt-Diff2*: $disjnt\ (U - V)\ (X - Y)$
if $X \subseteq V$
using *that* **by** (*auto simp: disjnt-def*)

lemma *disjoint-image-subset*: $\llbracket pairwise\ disjnt\ \mathcal{A}; \bigwedge X. X \in \mathcal{A} \implies f\ X \subseteq X \rrbracket \implies$
 $pairwise\ disjnt\ (f\ \mathcal{A})$
unfolding *disjnt-def pairwise-def* **by** *fast*

lemma *pairwise-disjnt-iff*: $pairwise\ disjnt\ \mathcal{A} \longleftrightarrow (\forall x. \exists_{\leq 1} X. X \in \mathcal{A} \wedge x \in X)$
by (*auto simp: Uniq-def disjnt-iff pairwise-def*)

lemma *disjnt-insert*:
 $\langle disjnt\ (insert\ x\ M)\ N \rangle$ **if** $\langle x \notin N \rangle$ $\langle disjnt\ M\ N \rangle$
using *that* **by** (*simp add: disjnt-def*)

lemma *Int-emptyI*: $(\bigwedge x. x \in A \implies x \in B \implies False) \implies A \cap B = \{\}$
by *blast*

lemma *in-image-insert-iff*:
assumes $\bigwedge C. C \in B \implies x \notin C$
shows $A \in insert\ x\ B \longleftrightarrow x \in A \wedge A - \{x\} \in B$ (**is** $?P \longleftrightarrow ?Q$)
proof
assume $?P$ **then show** $?Q$
using *assms* **by** *auto*

```

next
  assume ?Q
  then have  $x \in A$  and  $A - \{x\} \in B$ 
    by simp-all
  from  $\langle A - \{x\} \in B \rangle$  have  $\text{insert } x (A - \{x\}) \in \text{insert } x \text{ ` } B$ 
    by (rule imageI)
  also from  $\langle x \in A \rangle$ 
  have  $\text{insert } x (A - \{x\}) = A$ 
    by auto
  finally show ?P .
qed

```

hide-const (open) *member not-member*

lemmas *equalityI = subset-antisym*

lemmas *set-mp = subsetD*

lemmas *set-rev-mp = rev-subsetD*

ML <

```

val Ball-def = @{thm Ball-def}
val Bex-def = @{thm Bex-def}
val CollectD = @{thm CollectD}
val CollectE = @{thm CollectE}
val CollectI = @{thm CollectI}
val Collect-conj-eq = @{thm Collect-conj-eq}
val Collect-mem-eq = @{thm Collect-mem-eq}
val IntD1 = @{thm IntD1}
val IntD2 = @{thm IntD2}
val IntE = @{thm IntE}
val IntI = @{thm IntI}
val Int-Collect = @{thm Int-Collect}
val UNIV-I = @{thm UNIV-I}
val UNIV-witness = @{thm UNIV-witness}
val UnE = @{thm UnE}
val UnI1 = @{thm UnI1}
val UnI2 = @{thm UnI2}
val ballE = @{thm ballE}
val ballI = @{thm ballI}
val bexCI = @{thm bexCI}
val bexE = @{thm bexE}
val bexI = @{thm bexI}
val bex-triv = @{thm bex-triv}
val bspec = @{thm bspec}
val contra-subsetD = @{thm contra-subsetD}
val equalityCE = @{thm equalityCE}
val equalityD1 = @{thm equalityD1}
val equalityD2 = @{thm equalityD2}
val equalityE = @{thm equalityE}
val equalityI = @{thm equalityI}

```

```

val imageE = @{thm imageE}
val imageI = @{thm imageI}
val image-Un = @{thm image-Un}
val image-insert = @{thm image-insert}
val insert-commute = @{thm insert-commute}
val insert-iff = @{thm insert-iff}
val mem-Collect-eq = @{thm mem-Collect-eq}
val rangeE = @{thm rangeE}
val rangeI = @{thm rangeI}
val range-eqI = @{thm range-eqI}
val subsetCE = @{thm subsetCE}
val subsetD = @{thm subsetD}
val subsetI = @{thm subsetI}
val subset-refl = @{thm subset-refl}
val subset-trans = @{thm subset-trans}
val vimageD = @{thm vimageD}
val vimageE = @{thm vimageE}
val vimageI = @{thm vimageI}
val vimageI2 = @{thm vimageI2}
val vimage-Collect = @{thm vimage-Collect}
val vimage-Int = @{thm vimage-Int}
val vimage-Un = @{thm vimage-Un}
>

```

end

9 HOL type definitions

```

theory Typedef
imports Set
keywords
  typedef :: thy-goal-defn and
  morphisms :: quasi-command
begin

locale type-definition =
  fixes Rep and Abs and A
  assumes Rep: Rep x ∈ A
    and Rep-inverse: Abs (Rep x) = x
    and Abs-inverse: y ∈ A ⇒ Rep (Abs y) = y
  — This will be axiomatized for each typedef!
begin

lemma Rep-inject: Rep x = Rep y ⟷ x = y
proof
  assume Rep x = Rep y
  then have Abs (Rep x) = Abs (Rep y) by (simp only:)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  moreover have Abs (Rep y) = y by (rule Rep-inverse)

```

ultimately show $x = y$ by *simp*
 next
 assume $x = y$
 then show $\text{Rep } x = \text{Rep } y$ by (*simp only*:)
 qed

lemma *Abs-inject*:
 assumes $x \in A$ and $y \in A$
 shows $\text{Abs } x = \text{Abs } y \longleftrightarrow x = y$
 proof
 assume $\text{Abs } x = \text{Abs } y$
 then have $\text{Rep } (\text{Abs } x) = \text{Rep } (\text{Abs } y)$ by (*simp only*:)
 moreover from $\langle x \in A \rangle$ have $\text{Rep } (\text{Abs } x) = x$ by (*rule Abs-inverse*)
 moreover from $\langle y \in A \rangle$ have $\text{Rep } (\text{Abs } y) = y$ by (*rule Abs-inverse*)
 ultimately show $x = y$ by *simp*
 next
 assume $x = y$
 then show $\text{Abs } x = \text{Abs } y$ by (*simp only*:)
 qed

lemma *Rep-cases* [*cases set*]:
 assumes $y \in A$
 and *hyp*: $\bigwedge x. y = \text{Rep } x \implies P$
 shows P
 proof (*rule hyp*)
 from $\langle y \in A \rangle$ have $\text{Rep } (\text{Abs } y) = y$ by (*rule Abs-inverse*)
 then show $y = \text{Rep } (\text{Abs } y)$..
 qed

lemma *Abs-cases* [*cases type*]:
 assumes *r*: $\bigwedge y. x = \text{Abs } y \implies y \in A \implies P$
 shows P
 proof (*rule r*)
 have $\text{Abs } (\text{Rep } x) = x$ by (*rule Rep-inverse*)
 then show $x = \text{Abs } (\text{Rep } x)$..
 show $\text{Rep } x \in A$ by (*rule Rep*)
 qed

lemma *Rep-induct* [*induct set*]:
 assumes *y*: $y \in A$
 and *hyp*: $\bigwedge x. P (\text{Rep } x)$
 shows $P y$
 proof –
 have $P (\text{Rep } (\text{Abs } y))$ by (*rule hyp*)
 moreover from *y* have $\text{Rep } (\text{Abs } y) = y$ by (*rule Abs-inverse*)
 ultimately show $P y$ by *simp*
 qed

lemma *Abs-induct* [*induct type*]:

```

  assumes r:  $\bigwedge y. y \in A \implies P (Abs\ y)$ 
  shows  $P\ x$ 
proof -
  have  $Rep\ x \in A$  by (rule Rep)
  then have  $P (Abs (Rep\ x))$  by (rule r)
  moreover have  $Abs (Rep\ x) = x$  by (rule Rep-inverse)
  ultimately show  $P\ x$  by simp
qed

lemma Rep-range: range Rep = A
proof
  show  $range\ Rep \subseteq A$  using Rep by (auto simp add: image-def)
  show  $A \subseteq range\ Rep$ 
  proof
    fix x assume  $x \in A$ 
    then have  $x = Rep (Abs\ x)$  by (rule Abs-inverse [symmetric])
    then show  $x \in range\ Rep$  by (rule range-eqI)
  qed
qed

lemma Abs-image:  $Abs\ ` A = UNIV$ 
proof
  show  $Abs\ ` A \subseteq UNIV$  by (rule subset-UNIV)
  show  $UNIV \subseteq Abs\ ` A$ 
  proof
    fix x
    have  $x = Abs (Rep\ x)$  by (rule Rep-inverse [symmetric])
    moreover have  $Rep\ x \in A$  by (rule Rep)
    ultimately show  $x \in Abs\ ` A$  by (rule image-eqI)
  qed
qed

end

ML-file <Tools/typedef.ML>

end

```

10 Notions about functions

```

theory Fun
  imports Set
  keywords functor :: thy-goal-defn
begin

lemma apply-inverse:  $f\ x = u \implies (\bigwedge x. P\ x \implies g\ (f\ x) = x) \implies P\ x \implies x = g\ u$ 
  by auto

```

Uniqueness, so NOT the axiom of choice.

lemma *uniq-choice*: $\forall x. \exists!y. Q\ x\ y \implies \exists f. \forall x. Q\ x\ (f\ x)$
by (*force intro: theI'*)

lemma *b-uniq-choice*: $\forall x \in S. \exists!y. Q\ x\ y \implies \exists f. \forall x \in S. Q\ x\ (f\ x)$
by (*force intro: theI'*)

10.1 The Identity Function *id*

definition *id* :: $'a \Rightarrow 'a$
where *id* = $(\lambda x. x)$

lemma *id-apply* [*simp*]: $id\ x = x$
by (*simp add: id-def*)

lemma *image-id* [*simp*]: $image\ id = id$
by (*simp add: id-def fun-eq-iff*)

lemma *vimage-id* [*simp*]: $vimage\ id = id$
by (*simp add: id-def fun-eq-iff*)

lemma *eq-id-iff*: $(\forall x. f\ x = x) \longleftrightarrow f = id$
by *auto*

code-printing

constant *id* \rightarrow (*Haskell*) *id*

10.2 The Composition Operator $f \circ g$

definition *comp* :: $('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (*infixl* \circ 55)
where $f \circ g = (\lambda x. f\ (g\ x))$

notation (*ASCII*)
comp (*infixl* \circ 55)

lemma *comp-apply* [*simp*]: $(f \circ g)\ x = f\ (g\ x)$
by (*simp add: comp-def*)

lemma *comp-assoc*: $(f \circ g) \circ h = f \circ (g \circ h)$
by (*simp add: fun-eq-iff*)

lemma *id-comp* [*simp*]: $id \circ g = g$
by (*simp add: fun-eq-iff*)

lemma *comp-id* [*simp*]: $f \circ id = f$
by (*simp add: fun-eq-iff*)

lemma *comp-eq-dest*: $a \circ b = c \circ d \implies a\ (b\ v) = c\ (d\ v)$
by (*simp add: fun-eq-iff*)

lemma *comp-eq-elim*: $a \circ b = c \circ d \implies ((\bigwedge v. a (b v) = c (d v)) \implies R) \implies R$
by (*simp add: fun-eq-iff*)

lemma *comp-eq-dest-lhs*: $a \circ b = c \implies a (b v) = c v$
by *clarsimp*

lemma *comp-eq-id-dest*: $a \circ b = id \circ c \implies a (b v) = c v$
by *clarsimp*

lemma *image-comp*: $f \text{ ` } (g \text{ ` } r) = (f \circ g) \text{ ` } r$
by *auto*

lemma *vimage-comp*: $f \text{ -` } (g \text{ -` } x) = (g \circ f) \text{ -` } x$
by *auto*

lemma *image-eq-imp-comp*: $f \text{ ` } A = g \text{ ` } B \implies (h \circ f) \text{ ` } A = (h \circ g) \text{ ` } B$
by (*auto simp: comp-def elim!: equalityE*)

lemma *image-bind*: $f \text{ ` } (Set.bind A g) = Set.bind A ((\text{ ` } f \circ g)$
by (*auto simp add: Set.bind-def*)

lemma *bind-image*: $Set.bind (f \text{ ` } A) g = Set.bind A (g \circ f)$
by (*auto simp add: Set.bind-def*)

lemma (**in** *group-add*) *minus-comp-minus* [*simp*]: $uminus \circ uminus = id$
by (*simp add: fun-eq-iff*)

lemma (**in** *boolean-algebra*) *minus-comp-minus* [*simp*]: $uminus \circ uminus = id$
by (*simp add: fun-eq-iff*)

code-printing

constant *comp* \rightarrow (*SML*) **infixl** 5 *o* and (*Haskell*) **infixr** 9 .

10.3 The Forward Composition Operator *fcomp*

definition *fcomp* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$ (**infixl** $\circ>$ 60)
where $f \circ> g = (\lambda x. g (f x))$

lemma *fcomp-apply* [*simp*]: $(f \circ> g) x = g (f x)$
by (*simp add: fcomp-def*)

lemma *fcomp-assoc*: $(f \circ> g) \circ> h = f \circ> (g \circ> h)$
by (*simp add: fcomp-def*)

lemma *id-fcomp* [*simp*]: $id \circ> g = g$
by (*simp add: fcomp-def*)

lemma *fcomp-id* [*simp*]: $f \circ> id = f$

by (simp add: fcomp-def)

lemma fcomp-comp: fcomp f g = comp g f
by (simp add: ext)

code-printing

constant fcomp \rightarrow (Eval) **infixl** 1 #>

no-notation fcomp (**infixl** \circ > 60)

10.4 Mapping functions

definition map-fun :: ('c \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'd
where map-fun f g h = g \circ h \circ f

lemma map-fun-apply [simp]: map-fun f g h x = g (h (f x))
by (simp add: map-fun-def)

10.5 Injectivity and Bijectivity

definition inj-on :: ('a \Rightarrow 'b) \Rightarrow 'a set \Rightarrow bool — injective
where inj-on f A \longleftrightarrow ($\forall x \in A. \forall y \in A. f x = f y \longrightarrow x = y$)

definition bij-betw :: ('a \Rightarrow 'b) \Rightarrow 'a set \Rightarrow 'b set \Rightarrow bool — bijective
where bij-betw f A B \longleftrightarrow inj-on f A \wedge f 'A = B

A common special case: functions injective, surjective or bijective over the entire domain type.

abbreviation inj :: ('a \Rightarrow 'b) \Rightarrow bool
where inj f \equiv inj-on f UNIV

abbreviation surj :: ('a \Rightarrow 'b) \Rightarrow bool
where surj f \equiv range f = UNIV

translations — The negated case:
 \neg CONST surj f \leftarrow CONST range f \neq CONST UNIV

abbreviation bij :: ('a \Rightarrow 'b) \Rightarrow bool
where bij f \equiv bij-betw f UNIV UNIV

lemma inj-def: inj f \longleftrightarrow ($\forall x y. f x = f y \longrightarrow x = y$)
unfolding inj-on-def by blast

lemma injI: ($\bigwedge x y. f x = f y \Longrightarrow x = y$) \Longrightarrow inj f
unfolding inj-def by blast

theorem range-ex1-eq: inj f \Longrightarrow b \in range f \longleftrightarrow ($\exists !x. b = f x$)
unfolding inj-def by blast

lemma injD: $\text{inj } f \implies f x = f y \implies x = y$
by (*simp add: inj-def*)

lemma inj-on-eq-iff: $\text{inj-on } f A \implies x \in A \implies y \in A \implies f x = f y \longleftrightarrow x = y$
by (*auto simp: inj-on-def*)

lemma inj-on-cong: $(\bigwedge a. a \in A \implies f a = g a) \implies \text{inj-on } f A \longleftrightarrow \text{inj-on } g A$
by (*auto simp: inj-on-def*)

lemma image-strict-mono: $\text{inj-on } f B \implies A \subset B \implies f ' A \subset f ' B$
unfolding inj-on-def by blast

lemma inj-compose: $\text{inj } f \implies \text{inj } g \implies \text{inj } (f \circ g)$
by (*simp add: inj-def*)

lemma inj-fun: $\text{inj } f \implies \text{inj } (\lambda x y. f x)$
by (*simp add: inj-def fun-eq-iff*)

lemma inj-eq: $\text{inj } f \implies f x = f y \longleftrightarrow x = y$
by (*simp add: inj-on-eq-iff*)

lemma inj-on-iff-Uniq: $\text{inj-on } f A \longleftrightarrow (\forall x \in A. \exists_{\leq 1} y. y \in A \wedge f x = f y)$
by (*auto simp: Uniq-def inj-on-def*)

lemma inj-on-id[*simp*]: $\text{inj-on } \text{id } A$
by (*simp add: inj-on-def*)

lemma inj-on-id2[*simp*]: $\text{inj-on } (\lambda x. x) A$
by (*simp add: inj-on-def*)

lemma inj-on-Int: $\text{inj-on } f A \vee \text{inj-on } f B \implies \text{inj-on } f (A \cap B)$
unfolding inj-on-def by blast

lemma surj-id: $\text{surj } \text{id}$
by simp

lemma bij-id[*simp*]: $\text{bij } \text{id}$
by (*simp add: bij-betw-def*)

lemma bij-uminus: $\text{bij } (\text{uminus} :: 'a \Rightarrow 'a::\text{group-add})$
unfolding bij-betw-def inj-on-def
by (*force intro: minus-minus [symmetric]*)

lemma bij-betwE: $\text{bij-betw } f A B \implies \forall a \in A. f a \in B$
unfolding bij-betw-def by auto

lemma inj-onI [intro?]: $(\bigwedge x y. x \in A \implies y \in A \implies f x = f y \implies x = y) \implies \text{inj-on } f A$
by (*simp add: inj-on-def*)

For those frequent proofs by contradiction

lemma *inj-onCI*: $(\bigwedge x y. x \in A \implies y \in A \implies f x = f y \implies x \neq y \implies \text{False})$
 $\implies \text{inj-on } f A$
by (*force simp: inj-on-def*)

lemma *inj-on-inverseI*: $(\bigwedge x. x \in A \implies g (f x) = x) \implies \text{inj-on } f A$
by (*auto dest: arg-cong [of concl: g] simp add: inj-on-def*)

lemma *inj-onD*: $\text{inj-on } f A \implies f x = f y \implies x \in A \implies y \in A \implies x = y$
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-subset*:
assumes *inj-on* $f A$
and $B \subseteq A$
shows *inj-on* $f B$
proof (*rule inj-onI*)
fix $a b$
assume $a \in B$ **and** $b \in B$
with *assms* **have** $a \in A$ **and** $b \in A$
by *auto*
moreover **assume** $f a = f b$
ultimately **show** $a = b$
using *assms* **by** (*auto dest: inj-onD*)
qed

lemma *comp-inj-on*: $\text{inj-on } f A \implies \text{inj-on } g (f^{-1} A) \implies \text{inj-on } (g \circ f) A$
by (*simp add: comp-def inj-on-def*)

lemma *inj-on-imageI*: $\text{inj-on } (g \circ f) A \implies \text{inj-on } g (f^{-1} A)$
by (*auto simp add: inj-on-def*)

lemma *inj-on-image-iff*:
 $\forall x \in A. \forall y \in A. g (f x) = g (f y) \iff g x = g y \implies \text{inj-on } f A \implies \text{inj-on } g (f^{-1} A) \iff \text{inj-on } g A$
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-contrad*: $\text{inj-on } f A \implies x \neq y \implies x \in A \implies y \in A \implies f x \neq f y$
unfolding *inj-on-def* **by** *blast*

lemma *inj-singleton [simp]*: $\text{inj-on } (\lambda x. \{x\}) A$
by (*simp add: inj-on-def*)

lemma *inj-on-empty[iff]*: $\text{inj-on } f \{\}$
by (*simp add: inj-on-def*)

lemma *subset-inj-on*: $\text{inj-on } f B \implies A \subseteq B \implies \text{inj-on } f A$
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-Un*: $\text{inj-on } f (A \cup B) \iff \text{inj-on } f A \wedge \text{inj-on } f B \wedge f^{-1} (A - B) \cap$

$f \text{ ' } (B - A) = \{\}$

unfolding *inj-on-def* **by** (*blast intro: sym*)

lemma *inj-on-insert [iff]*: $\text{inj-on } f \text{ (insert } a \text{ } A) \longleftrightarrow \text{inj-on } f \text{ } A \wedge f \text{ } a \notin f \text{ ' } (A - \{a\})$

unfolding *inj-on-def* **by** (*blast intro: sym*)

lemma *inj-on-diff*: $\text{inj-on } f \text{ } A \Longrightarrow \text{inj-on } f \text{ } (A - B)$

unfolding *inj-on-def* **by** *blast*

lemma *comp-inj-on-iff*: $\text{inj-on } f \text{ } A \Longrightarrow \text{inj-on } f' \text{ } (f \text{ ' } A) \longleftrightarrow \text{inj-on } (f' \circ f) \text{ } A$

by (*auto simp: comp-inj-on inj-on-def*)

lemma *inj-on-imageI2*: $\text{inj-on } (f' \circ f) \text{ } A \Longrightarrow \text{inj-on } f \text{ } A$

by (*auto simp: comp-inj-on inj-on-def*)

lemma *inj-img-insertE*:

assumes *inj-on* $f \text{ } A$

assumes $x \notin B$

and $\text{insert } x \text{ } B = f \text{ ' } A$

obtains $x' \notin A'$ **where** $x' \notin A'$ **and** $A = \text{insert } x' \text{ } A'$ **and** $x = f \text{ ' } x'$ **and** $B = f \text{ ' } A'$

proof –

from *assms* **have** $x \in f \text{ ' } A$ **by** *auto*

then obtain x' **where** $x' \in A$ $x = f \text{ ' } x'$ **by** *auto*

then have $A: A = \text{insert } x' \text{ } (A - \{x'\})$ **by** *auto*

with *assms* $*$ **have** $B: B = f \text{ ' } (A - \{x'\})$ **by** (*auto dest: inj-on-contrad*)

have $x' \notin A - \{x'\}$ **by** *simp*

from *this* $A \langle x = f \text{ ' } x' \rangle B$ **show** *?thesis ..*

qed

lemma *linorder-inj-onI*:

fixes $A :: 'a :: \text{order set}$

assumes $ne: \bigwedge x y. \llbracket x < y; x \in A; y \in A \rrbracket \Longrightarrow f \text{ ' } x \neq f \text{ ' } y$ **and** $lin: \bigwedge x y. \llbracket x \in A; y \in A \rrbracket \Longrightarrow x \leq y \vee y \leq x$

shows *inj-on* $f \text{ } A$

proof (*rule inj-onI*)

fix $x \ y$

assume $eq: f \text{ ' } x = f \text{ ' } y$ **and** $x \in A \ y \in A$

then show $x = y$

using *lin* [*of* $x \ y$] *ne* **by** (*force simp: dual-order.order-iff-strict*)

qed

lemma *linorder-inj-onI'*:

fixes $A :: 'a :: \text{linorder set}$

assumes $\bigwedge i \ j. i \in A \Longrightarrow j \in A \Longrightarrow i < j \Longrightarrow f \text{ ' } i \neq f \text{ ' } j$

shows *inj-on* $f \text{ } A$

by (*intro linorder-inj-onI*) (*auto simp add: assms*)

lemma *linorder-injI*:

assumes $\bigwedge x y::'a::\text{linorder}. x < y \implies f x \neq f y$

shows *inj f*

— Courtesy of Stephan Merz

using *assms* **by** (*simp add: linorder-inj-onI'*)

lemma *inj-on-image-Pow*: $\text{inj-on } f A \implies \text{inj-on } (\text{image } f) (\text{Pow } A)$

unfolding *Pow-def inj-on-def* **by** *blast*

lemma *bij-betw-image-Pow*: $\text{bij-betw } f A B \implies \text{bij-betw } (\text{image } f) (\text{Pow } A) (\text{Pow } B)$

by (*auto simp add: bij-betw-def inj-on-image-Pow image-Pow-surj*)

lemma *surj-def*: $\text{surj } f \iff (\forall y. \exists x. y = f x)$

by *auto*

lemma *surjI*:

assumes $\bigwedge x. g (f x) = x$

shows *surj g*

using *assms* [*symmetric*] **by** *auto*

lemma *surjD*: $\text{surj } f \implies \exists x. y = f x$

by (*simp add: surj-def*)

lemma *surjE*: $\text{surj } f \implies (\bigwedge x. y = f x \implies C) \implies C$

by (*simp add: surj-def*) *blast*

lemma *comp-surj*: $\text{surj } f \implies \text{surj } g \implies \text{surj } (g \circ f)$

using *image-comp* [*of g f UNIV*] **by** *simp*

lemma *bij-betw-imageI*: $\text{inj-on } f A \implies f ' A = B \implies \text{bij-betw } f A B$

unfolding *bij-betw-def* **by** *clarify*

lemma *bij-betw-imp-surj-on*: $\text{bij-betw } f A B \implies f ' A = B$

unfolding *bij-betw-def* **by** *clarify*

lemma *bij-betw-imp-surj*: $\text{bij-betw } f A \text{ UNIV} \implies \text{surj } f$

unfolding *bij-betw-def* **by** *auto*

lemma *bij-betw-empty1*: $\text{bij-betw } f \{\} A \implies A = \{\}$

unfolding *bij-betw-def* **by** *blast*

lemma *bij-betw-empty2*: $\text{bij-betw } f A \{\} \implies A = \{\}$

unfolding *bij-betw-def* **by** *blast*

lemma *inj-on-imp-bij-betw*: $\text{inj-on } f A \implies \text{bij-betw } f A (f ' A)$

unfolding *bij-betw-def* **by** *simp*

lemma *bij-betw-DiffI*:

assumes $\text{bij-betw } f \ A \ B \ \text{bij-betw } f \ C \ D \ C \subseteq A \ D \subseteq B$
shows $\text{bij-betw } f \ (A - C) \ (B - D)$
using *assms* **unfolding** *bij-betw-def inj-on-def* **by** *auto*

lemma *bij-betw-singleton-iff* [*simp*]: $\text{bij-betw } f \ \{x\} \ \{y\} \longleftrightarrow f \ x = y$
by (*auto simp: bij-betw-def*)

lemma *bij-betw-singletonI* [*intro*]: $f \ x = y \implies \text{bij-betw } f \ \{x\} \ \{y\}$
by *auto*

lemma *bij-betw-apply*: $\llbracket \text{bij-betw } f \ A \ B; a \in A \rrbracket \implies f \ a \in B$
unfolding *bij-betw-def* **by** *auto*

lemma *bij-def*: $\text{bij } f \longleftrightarrow \text{inj } f \wedge \text{surj } f$
by (*rule bij-betw-def*)

lemma *bijI*: $\text{inj } f \implies \text{surj } f \implies \text{bij } f$
by (*rule bij-betw-imageI*)

lemma *bij-is-inj*: $\text{bij } f \implies \text{inj } f$
by (*simp add: bij-def*)

lemma *bij-is-surj*: $\text{bij } f \implies \text{surj } f$
by (*simp add: bij-def*)

lemma *bij-betw-imp-inj-on*: $\text{bij-betw } f \ A \ B \implies \text{inj-on } f \ A$
by (*simp add: bij-betw-def*)

lemma *bij-betw-trans*: $\text{bij-betw } f \ A \ B \implies \text{bij-betw } g \ B \ C \implies \text{bij-betw } (g \circ f) \ A \ C$
by (*auto simp add: bij-betw-def comp-inj-on*)

lemma *bij-comp*: $\text{bij } f \implies \text{bij } g \implies \text{bij } (g \circ f)$
by (*rule bij-betw-trans*)

lemma *bij-betw-comp-iff*: $\text{bij-betw } f \ A \ A' \implies \text{bij-betw } f' \ A' \ A'' \longleftrightarrow \text{bij-betw } (f' \circ f) \ A \ A''$
by (*auto simp add: bij-betw-def inj-on-def*)

lemma *bij-betw-comp-iff2*:
assumes *bij*: $\text{bij-betw } f' \ A' \ A''$
and *img*: $f \ ' \ A \leq A'$
shows $\text{bij-betw } f \ A \ A' \longleftrightarrow \text{bij-betw } (f' \circ f) \ A \ A''$ (**is** *?L* \longleftrightarrow *?R*)
proof
assume *?L*
then show *?R*
using *assms* **by** (*auto simp add: bij-betw-comp-iff*)
next
assume ***: *?R*
have $\text{inj-on } (f' \circ f) \ A \implies \text{inj-on } f \ A$

```

    using inj-on-imageI2 by blast
  moreover have  $A' \subseteq f' A$ 
  proof
    fix a'
    assume **:  $a' \in A'$ 
    with bij have  $f' a' \in A''$ 
      unfolding bij-betw-def by auto
    with * obtain a where  $1: a \in A \wedge f' (f a) = f' a'$ 
      unfolding bij-betw-def by force
    with img have  $f a \in A'$  by auto
    with bij ** 1 have  $f a = a'$ 
      unfolding bij-betw-def inj-on-def by auto
    with 1 show  $a' \in f' A$  by auto
  qed
  ultimately show ?L
    using img * by (auto simp add: bij-betw-def)
qed

lemma bij-betw-inv:
  assumes bij-betw f A B
  shows  $\exists g. \text{bij-betw } g B A$ 
  proof -
    have i: inj-on f A and s:  $f' A = B$ 
      using assms by (auto simp: bij-betw-def)
    let ?P =  $\lambda b. a \in A \wedge f a = b$ 
    let ?g =  $\lambda b. \text{The } (?P b)$ 
    have g:  $?g b = a$  if  $P: ?P b a$  for  $a b$ 
      proof -
        from that s have ex1:  $\exists a. ?P b a$  by blast
        then have uex1:  $\exists! a. ?P b a$  by (blast dest: inj-onD[OF i])
        then show ?thesis
          using the1-equality[OF uex1, OF P] P by simp
      qed
    have inj-on ?g B
      proof (rule inj-onI)
        fix x y
        assume  $x \in B \ y \in B \ ?g x = ?g y$ 
        from s  $\langle x \in B \rangle$  obtain a1 where  $a1: ?P x a1$  by blast
        from s  $\langle y \in B \rangle$  obtain a2 where  $a2: ?P y a2$  by blast
        from g [OF a1] a1 g [OF a2] a2  $\langle ?g x = ?g y \rangle$  show  $x = y$  by simp
      qed
    moreover have  $?g' B = A$ 
      proof safe
        fix b
        assume  $b \in B$ 
        with s obtain a where  $P: ?P b a$  by blast
        with g [OF P] show  $?g b \in A$  by auto
      next
        fix a

```

assume $a \in A$
with s **obtain** b **where** $P: ?P\ b\ a$ **by** *blast*
with s **have** $b \in B$ **by** *blast*
with $g[OF\ P]$ **have** $\exists b \in B. a = ?g\ b$ **by** *blast*
then show $a \in ?g\ 'B$
by *auto*
qed
ultimately show *?thesis*
by (*auto simp: bij-betw-def*)
qed

lemma *bij-betw-cong*: $(\bigwedge a. a \in A \implies f\ a = g\ a) \implies \text{bij-betw}\ f\ A\ A' = \text{bij-betw}\ g\ A\ A'$
unfolding *bij-betw-def inj-on-def* **by** *safe force+*

lemma *bij-betw-id[intro, simp]*: *bij-betw id A A*
unfolding *bij-betw-def id-def* **by** *auto*

lemma *bij-betw-id-iff*: *bij-betw id A B \longleftrightarrow A = B*
by (*auto simp add: bij-betw-def*)

lemma *bij-betw-combine*:
bij-betw f A B \implies bij-betw f C D \implies B \cap D = {} \implies bij-betw f (A \cup C) (B \cup D)
unfolding *bij-betw-def inj-on-Un image-Un* **by** *auto*

lemma *bij-betw-subset*: *bij-betw f A A' \implies B \subseteq A \implies f ' B = B' \implies bij-betw f B B'*
by (*auto simp add: bij-betw-def inj-on-def*)

lemma *bij-betw-ball*: *bij-betw f A B \implies ($\forall b \in B. \text{phi}\ b$) = ($\forall a \in A. \text{phi}\ (f\ a)$)*
unfolding *bij-betw-def inj-on-def* **by** *blast*

lemma *bij-pointE*:
assumes *bij f*
obtains x **where** $y = f\ x$ **and** $\bigwedge x'. y = f\ x' \implies x' = x$
proof –
from *assms* **have** *inj f* **by** (*rule bij-is-inj*)
moreover from *assms* **have** *surj f* **by** (*rule bij-is-surj*)
then have $y \in \text{range}\ f$ **by** *simp*
ultimately have $\exists! x. y = f\ x$ **by** (*simp add: range-ex1-eq*)
with that show *thesis* **by** *blast*
qed

lemma *bij-iff*:
 $\langle \text{bij}\ f \longleftrightarrow (\forall x. \exists! y. f\ y = x) \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)
proof
assume *?P*
then have $\langle \text{inj}\ f \rangle \langle \text{surj}\ f \rangle$


```

  by (simp-all add: bij-def)
show ?Q
proof
  fix y
  from ⟨surj f⟩ obtain x where ⟨y = f x⟩
  by (auto simp add: surj-def)
  with ⟨inj f⟩ show ⟨∃!x. f x = y⟩
  by (auto simp add: inj-def)
qed
next
assume ?Q
then have ⟨inj f⟩
  by (auto simp add: inj-def)
moreover have ⟨∃x. y = f x⟩ for y
proof -
  from ⟨?Q⟩ obtain x where ⟨f x = y⟩
  by blast
  then have ⟨y = f x⟩
  by simp
  then show ?thesis ..
qed
then have ⟨surj f⟩
  by (auto simp add: surj-def)
ultimately show ?P
  by (rule bijI)
qed

```

lemma *bij-betw-partition*:

```

⟨bij-betw f A B⟩
if ⟨bij-betw f (A ∪ C) (B ∪ D)⟩ ⟨bij-betw f C D⟩ ⟨A ∩ C = {}⟩ ⟨B ∩ D = {}⟩
proof -
  from that have ⟨inj-on f (A ∪ C)⟩ ⟨inj-on f C⟩ ⟨f ‘ (A ∪ C) = B ∪ D⟩ ⟨f ‘ C
= D⟩
  by (simp-all add: bij-betw-def)
  then have ⟨inj-on f A⟩ and ⟨f ‘ (A - C) ∩ f ‘ (C - A) = {}⟩
  by (simp-all add: inj-on-Un)
  with ⟨A ∩ C = {}⟩ have ⟨f ‘ A ∩ f ‘ C = {}⟩
  by auto
  with ⟨f ‘ (A ∪ C) = B ∪ D⟩ ⟨f ‘ C = D⟩ ⟨B ∩ D = {}⟩
  have ⟨f ‘ A = B⟩
  by blast
  with ⟨inj-on f A⟩ show ?thesis
  by (simp add: bij-betw-def)
qed

```

lemma *surj-image-vimage-eq*: $\text{surj } f \implies f '(f -' A) = A$
 by *simp*

lemma *surj-vimage-empty*:

assumes *surj f*
shows $f - ' A = \{\} \longleftrightarrow A = \{\}$
using *surj-image-vimage-eq* [*OF* \langle surj *f* \rangle , of *A*]
by (*intro iffI*) *fastforce+*

lemma *inj-vimage-image-eq*: $\text{inj } f \implies f - ' (f ' A) = A$
unfolding *inj-def* **by** *blast*

lemma *vimage-subsetD*: $\text{surj } f \implies f - ' B \subseteq A \implies B \subseteq f ' A$
by (*blast intro: sym*)

lemma *vimage-subsetI*: $\text{inj } f \implies B \subseteq f ' A \implies f - ' B \subseteq A$
unfolding *inj-def* **by** *blast*

lemma *vimage-subset-eq*: $\text{bij } f \implies f - ' B \subseteq A \longleftrightarrow B \subseteq f ' A$
unfolding *bij-def* **by** (*blast del: subsetI intro: vimage-subsetI vimage-subsetD*)

lemma *inj-on-image-eq-iff*: $\text{inj-on } f C \implies A \subseteq C \implies B \subseteq C \implies f ' A = f ' B$
 $\longleftrightarrow A = B$
by (*fastforce simp: inj-on-def*)

lemma *inj-on-Un-image-eq-iff*: $\text{inj-on } f (A \cup B) \implies f ' A = f ' B \longleftrightarrow A = B$
by (*erule inj-on-image-eq-iff*) *simp-all*

lemma *inj-on-image-Int*: $\text{inj-on } f C \implies A \subseteq C \implies B \subseteq C \implies f ' (A \cap B) = f ' A \cap f ' B$
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-image-set-diff*: $\text{inj-on } f C \implies A - B \subseteq C \implies B \subseteq C \implies f ' (A - B) = f ' A - f ' B$
unfolding *inj-on-def* **by** *blast*

lemma *image-Int*: $\text{inj } f \implies f ' (A \cap B) = f ' A \cap f ' B$
unfolding *inj-def* **by** *blast*

lemma *image-set-diff*: $\text{inj } f \implies f ' (A - B) = f ' A - f ' B$
unfolding *inj-def* **by** *blast*

lemma *inj-on-image-mem-iff*: $\text{inj-on } f B \implies a \in B \implies A \subseteq B \implies f a \in f ' A$
 $\longleftrightarrow a \in A$
by (*auto simp: inj-on-def*)

lemma *inj-image-mem-iff*: $\text{inj } f \implies f a \in f ' A \longleftrightarrow a \in A$
by (*blast dest: injD*)

lemma *inj-image-subset-iff*: $\text{inj } f \implies f ' A \subseteq f ' B \longleftrightarrow A \subseteq B$
by (*blast dest: injD*)

lemma *inj-image-eq-iff*: $\text{inj } f \implies f ' A = f ' B \longleftrightarrow A = B$

by (*blast dest: injD*)

lemma *surj-Compl-image-subset*: $\text{surj } f \implies - (f \text{ ' } A) \subseteq f \text{ ' } (- A)$
by *auto*

lemma *inj-image-Compl-subset*: $\text{inj } f \implies f \text{ ' } (- A) \subseteq - (f \text{ ' } A)$
by (*auto simp: inj-def*)

lemma *bij-image-Compl-eq*: $\text{bij } f \implies f \text{ ' } (- A) = - (f \text{ ' } A)$
by (*simp add: bij-def inj-image-Compl-subset surj-Compl-image-subset equalityI*)

lemma *inj-vimage-singleton*: $\text{inj } f \implies f \text{ ' } \{a\} \subseteq \{\text{THE } x. f x = a\}$
— The inverse image of a singleton under an injective function is included in a singleton.

by (*simp add: inj-def*) (*blast intro: the-equality [symmetric]*)

lemma *inj-on-vimage-singleton*: $\text{inj-on } f A \implies f \text{ ' } \{a\} \cap A \subseteq \{\text{THE } x. x \in A \wedge f x = a\}$

by (*auto simp add: inj-on-def intro: the-equality [symmetric]*)

lemma *bij-betw-byWitness*:

assumes *left*: $\forall a \in A. f' (f a) = a$
and *right*: $\forall a' \in A'. f (f' a') = a'$
and $f \text{ ' } A \subseteq A'$
and *img2*: $f' \text{ ' } A' \subseteq A$
shows *bij-betw* $f A A'$
using *assms*
unfolding *bij-betw-def inj-on-def*

proof *safe*

fix $a b$

assume $a \in A \ b \in A$

with *left* have $a = f' (f a) \wedge b = f' (f b)$ by *simp*

moreover assume $f a = f b$

ultimately show $a = b$ by *simp*

next

fix a' assume *: $a' \in A'$

with *img2* have $f' a' \in A$ by *blast*

moreover from * *right* have $a' = f (f' a')$ by *simp*

ultimately show $a' \in f \text{ ' } A$ by *blast*

qed

corollary *notIn-Un-bij-betw*:

assumes $b \notin A$

and $f b \notin A'$

and *bij-betw* $f A A'$

shows *bij-betw* $f (A \cup \{b\}) (A' \cup \{f b\})$

proof —

have *bij-betw* $f \{b\} \{f b\}$

unfolding *bij-betw-def inj-on-def* by *simp*

```

with assms show ?thesis
  using bij-betw-combine[of f A A' {b} {f b}] by blast
qed

lemma notIn-Un-bij-betw3:
  assumes  $b \notin A$ 
    and  $f b \notin A'$ 
  shows  $\text{bij-betw } f A A' = \text{bij-betw } f (A \cup \{b\}) (A' \cup \{f b\})$ 
proof
  assume  $\text{bij-betw } f A A'$ 
  then show  $\text{bij-betw } f (A \cup \{b\}) (A' \cup \{f b\})$ 
    using assms notIn-Un-bij-betw [of  $b A f A'$ ] by blast
next
  assume *:  $\text{bij-betw } f (A \cup \{b\}) (A' \cup \{f b\})$ 
  have  $f' A = A'$ 
  proof safe
    fix  $a$ 
    assume **:  $a \in A$ 
    then have  $f a \in A' \cup \{f b\}$ 
      using * unfolding bij-betw-def by blast
    moreover
    have False if  $f a = f b$ 
    proof –
      have  $a = b$ 
      using * ** that unfolding bij-betw-def inj-on-def by blast
      with  $\langle b \notin A \rangle$  ** show ?thesis by blast
    qed
    ultimately show  $f a \in A'$  by blast
  next
  fix  $a'$ 
  assume **:  $a' \in A'$ 
  then have  $a' \in f' (A \cup \{b\})$ 
    using * by (auto simp add: bij-betw-def)
  then obtain  $a$  where  $1: a \in A \cup \{b\} \wedge f a = a'$  by blast
  moreover
  have False if  $a = b$  using  $1$  **  $\langle f b \notin A' \rangle$  that by blast
  ultimately have  $a \in A$  by blast
  with  $1$  show  $a' \in f' A$  by blast
  qed
  then show  $\text{bij-betw } f A A'$ 
    using * bij-betw-subset[of  $f A \cup \{b\} - A$ ] by blast
qed

lemma inj-on-disjoint-Un:
  assumes inj-on  $f A$  and inj-on  $g B$ 
  and  $f' A \cap g' B = \{\}$ 
  shows inj-on  $(\lambda x. \text{if } x \in A \text{ then } f x \text{ else } g x) (A \cup B)$ 
  using assms by (simp add: inj-on-def disjoint-iff) (blast)

```

lemma *bij-betw-disjoint-Un*:
assumes *bij-betw f A C* **and** *bij-betw g B D*
and $A \cap B = \{\}$
and $C \cap D = \{\}$
shows *bij-betw* $(\lambda x. \text{if } x \in A \text{ then } f x \text{ else } g x)$ $(A \cup B)$ $(C \cup D)$
using *assms* **by** *(auto simp: inj-on-disjoint-Un bij-betw-def)*

lemma *involuntary-imp-bij*:
 $\langle \text{bij } f \rangle$ **if** $\langle \bigwedge x. f (f x) = x \rangle$
proof *(rule bijI)*
from *that* **show** $\langle \text{surj } f \rangle$
by *(rule surjI)*
show $\langle \text{inj } f \rangle$
proof *(rule injI)*
fix $x y$
assume $\langle f x = f y \rangle$
then have $\langle f (f x) = f (f y) \rangle$
by *simp*
then show $\langle x = y \rangle$
by *(simp add: that)*
qed
qed

10.5.1 Inj/surj/bij of Algebraic Operations

context *cancel-semigroup-add*
begin

lemma *inj-on-add [simp]*:
inj-on $((+) a) A$
by *(rule inj-onI) simp*

lemma *inj-on-add' [simp]*:
inj-on $(\lambda b. b + a) A$
by *(rule inj-onI) simp*

lemma *bij-betw-add [simp]*:
bij-betw $((+) a) A B \longleftrightarrow (+) a ' A = B$
by *(simp add: bij-betw-def)*

end

context *group-add*
begin

lemma *diff-left-imp-eq: a - b = a - c \implies b = c*
unfolding *add-uminus-conv-diff [symmetric]*
by *(drule local.add-left-imp-eq) simp*

lemma *inj-uminus*[*simp*, *intro*]: *inj-on uminus A*
by (*auto intro!*: *inj-onI*)

lemma *surj-uminus*[*simp*]: *surj uminus*
using *surjI minus-minus* **by** *blast*

lemma *surj-plus* [*simp*]:
surj ((+) a)
proof (*standard, simp, standard, simp*)
fix *x*
have $x = a + (-a + x)$ **by** (*simp add: add.assoc*)
thus $x \in \text{range } ((+) a)$ **by** *blast*
qed

lemma *surj-plus-right* [*simp*]:
surj ($\lambda b. b+a$)
proof (*standard, simp, standard, simp*)
fix *b* **show** $b \in \text{range } (\lambda b. b+a)$
using *diff-add-cancel*[*of b a, symmetric*] **by** *blast*
qed

lemma *inj-on-diff-left* [*simp*]:
 $\langle \text{inj-on } ((-) a) A \rangle$
by (*auto intro: inj-onI dest!: diff-left-imp-eq*)

lemma *inj-on-diff-right* [*simp*]:
 $\langle \text{inj-on } (\lambda b. b - a) A \rangle$
by (*auto intro: inj-onI simp add: algebra-simps*)

lemma *surj-diff* [*simp*]:
surj ((-) a)
proof (*standard, simp, standard, simp*)
fix *x*
have $x = a - (-x + a)$ **by** (*simp add: algebra-simps*)
thus $x \in \text{range } ((-) a)$ **by** *blast*
qed

lemma *surj-diff-right* [*simp*]:
surj ($\lambda x. x - a$)
proof (*standard, simp, standard, simp*)
fix *x*
have $x = x + a - a$ **by** *simp*
thus $x \in \text{range } (\lambda x. x - a)$ **by** *fast*
qed

lemma *shows* *bij-plus*: *bij ((+) a)* **and** *bij-plus-right*: *bij ($\lambda x. x + a$)*
and *bij-uminus*: *bij uminus*
and *bij-diff*: *bij ((-) a)* **and** *bij-diff-right*: *bij ($\lambda x. x - a$)*
by(*simp-all add: bij-def*)

lemma *translation-subtract-Compl:*

$$(\lambda x. x - a) \text{ ' } (- t) = - ((\lambda x. x - a) \text{ ' } t)$$

by(*rule bij-image-Compl-eq*)

$$(auto simp add: bij-def surj-def inj-def diff-eq-eq intro!: add-diff-cancel[symmetric])$$

lemma *translation-diff:*

$$(+ a \text{ ' } (s - t) = ((+) a \text{ ' } s) - ((+) a \text{ ' } t)$$

by *auto*

lemma *translation-subtract-diff:*

$$(\lambda x. x - a) \text{ ' } (s - t) = ((\lambda x. x - a) \text{ ' } s) - ((\lambda x. x - a) \text{ ' } t)$$

by(*rule image-set-diff*)(*simp add: inj-on-def diff-eq-eq*)

lemma *translation-Int:*

$$(+ a \text{ ' } (s \cap t) = ((+) a \text{ ' } s) \cap ((+) a \text{ ' } t)$$

by *auto*

lemma *translation-subtract-Int:*

$$(\lambda x. x - a) \text{ ' } (s \cap t) = ((\lambda x. x - a) \text{ ' } s) \cap ((\lambda x. x - a) \text{ ' } t)$$

by(*rule image-Int*)(*simp add: inj-on-def diff-eq-eq*)

end

context *ab-group-add*

begin

lemma *translation-Compl:*

$$(+ a \text{ ' } (- t) = - ((+) a \text{ ' } t)$$

proof (*rule set-eqI*)

fix *b*

$$\text{show } b \in (+ a \text{ ' } (- t) \longleftrightarrow b \in - (+ a \text{ ' } t)$$

$$\text{by } (auto simp: image-iff algebra-simps intro!: bezI [of - b - a])$$

qed

end

10.6 Function Updating

definition *fun-upd* :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)

$$\text{where } \text{fun-upd } f \ a \ b = (\lambda x. \text{if } x = a \text{ then } b \text{ else } f \ x)$$

nonterminal *updbinds* and *updbind*

syntax

$$\text{-updbind} :: 'a \Rightarrow 'a \Rightarrow \text{updbind} \quad ((2- := / -))$$

$$:: \text{updbind} \Rightarrow \text{updbinds} \quad (-)$$

$$\text{-updbinds} :: \text{updbind} \Rightarrow \text{updbinds} \Rightarrow \text{updbinds} \ (-, / -)$$

$-Update :: 'a \Rightarrow updbinds \Rightarrow 'a \quad (-/'((-) [1000, 0] 900)$

translations

$-Update f (-updbinds b bs) \Rightarrow -Update (-Update f b) bs$
 $f(x:=y) \Rightarrow CONST fun-upd f x y$

lemma *fun-upd-idem-iff*: $f(x:=y) = f \longleftrightarrow f x = y$

unfolding *fun-upd-def*

apply *safe*

apply (*erule subst*)

apply *auto*

done

lemma *fun-upd-idem*: $f x = y \Longrightarrow f(x := y) = f$

by (*simp only: fun-upd-idem-iff*)

lemma *fun-upd-triv [iff]*: $f(x := f x) = f$

by (*simp only: fun-upd-idem*)

lemma *fun-upd-apply [simp]*: $(f(x := y)) z = (if z = x then y else f z)$

by (*simp add: fun-upd-def*)

lemma *fun-upd-same*: $(f(x := y)) x = y$

by *simp*

lemma *fun-upd-other*: $z \neq x \Longrightarrow (f(x := y)) z = f z$

by *simp*

lemma *fun-upd-upd [simp]*: $f(x := y, x := z) = f(x := z)$

by (*simp add: fun-eq-iff*)

lemma *fun-upd-twist*: $a \neq c \Longrightarrow (m(a := b))(c := d) = (m(c := d))(a := b)$

by *auto*

lemma *inj-on-fun-updI*: $inj-on f A \Longrightarrow y \notin f ' A \Longrightarrow inj-on (f(x := y)) A$

by (*auto simp: inj-on-def*)

lemma *fun-upd-image*: $f(x := y) ' A = (if x \in A then insert y (f ' (A - \{x\})) else f ' A)$

by *auto*

lemma *fun-upd-comp*: $f \circ (g(x := y)) = (f \circ g)(x := f y)$

by *auto*

lemma *fun-upd-eqD*: $f(x := y) = g(x := z) \Longrightarrow y = z$

by (*simp add: fun-eq-iff split: if-split-asm*)

10.7 *override-on*

definition *override-on* :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ 'a ⇒ 'b
where *override-on* f g A = (λa. if a ∈ A then g a else f a)

lemma *override-on-emptyset*[simp]: *override-on* f g {} = f
by (simp add: *override-on-def*)

lemma *override-on-apply-notin*[simp]: a ∉ A ⇒ (override-on f g A) a = f a
by (simp add: *override-on-def*)

lemma *override-on-apply-in*[simp]: a ∈ A ⇒ (override-on f g A) a = g a
by (simp add: *override-on-def*)

lemma *override-on-insert*: *override-on* f g (insert x X) = (override-on f g X)(x:=g x)
by (simp add: *override-on-def fun-eq-iff*)

lemma *override-on-insert'*: *override-on* f g (insert x X) = (override-on (f(x:=g x)) g X)
by (simp add: *override-on-def fun-eq-iff*)

10.8 Inversion of injective functions

definition *the-inv-into* :: 'a set ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a)
where *the-inv-into* A f = (λx. THE y. y ∈ A ∧ f y = x)

lemma *the-inv-into-f-f*: inj-on f A ⇒ x ∈ A ⇒ *the-inv-into* A f (f x) = x
unfolding *the-inv-into-def inj-on-def* **by** blast

lemma *f-the-inv-into-f*: inj-on f A ⇒ y ∈ f ' A ⇒ f (*the-inv-into* A f y) = y
unfolding *the-inv-into-def*
by (rule *the1I2*; blast dest: *inj-onD*)

lemma *f-the-inv-into-f-bij-betw*:
bij-betw f A B ⇒ (*bij-betw* f A B ⇒ x ∈ B) ⇒ f (*the-inv-into* A f x) = x
unfolding *bij-betw-def* **by** (blast intro: *f-the-inv-into-f*)

lemma *the-inv-into-into*: inj-on f A ⇒ x ∈ f ' A ⇒ A ⊆ B ⇒ *the-inv-into* A f x ∈ B
unfolding *the-inv-into-def*
by (rule *the1I2*; blast dest: *inj-onD*)

lemma *the-inv-into-onto* [simp]: inj-on f A ⇒ *the-inv-into* A f ' (f ' A) = A
by (fast intro: *the-inv-into-into the-inv-into-f-f [symmetric]*)

lemma *the-inv-into-f-eq*: inj-on f A ⇒ f x = y ⇒ x ∈ A ⇒ *the-inv-into* A f y = x
by (force simp add: *the-inv-into-f-f*)

lemma *the-inv-into-comp*:

$inj\text{-}on\ f\ (g\ 'A) \implies inj\text{-}on\ g\ A \implies x \in f\ 'g\ 'A \implies$
 $the\text{-}inv\text{-}into\ A\ (f \circ g)\ x = (the\text{-}inv\text{-}into\ A\ g \circ the\text{-}inv\text{-}into\ (g\ 'A)\ f)\ x$
apply (*rule the-inv-into-f-eq*)
apply (*fast intro: comp-inj-on*)
apply (*simp add: f-the-inv-into-f the-inv-into-into*)
apply (*simp add: the-inv-into-into*)
done

lemma *inj-on-the-inv-into*: $inj\text{-}on\ f\ A \implies inj\text{-}on\ (the\text{-}inv\text{-}into\ A\ f)\ (f\ 'A)$
by (*auto intro: inj-onI simp: the-inv-into-f-f*)

lemma *bij-betw-the-inv-into*: $bij\text{-}betw\ f\ A\ B \implies bij\text{-}betw\ (the\text{-}inv\text{-}into\ A\ f)\ B\ A$
by (*auto simp add: bij-betw-def inj-on-the-inv-into the-inv-into-into*)

lemma *bij-betw-iff-bijections*:

$bij\text{-}betw\ f\ A\ B \longleftrightarrow (\exists g. (\forall x \in A. f\ x \in B \wedge g(f\ x) = x) \wedge (\forall y \in B. g\ y \in A \wedge$
 $f(g\ y) = y))$
(is ?lhs = ?rhs)

proof

show $?lhs \implies ?rhs$
by (*auto simp: bij-betw-def f-the-inv-into-f the-inv-into-f-f the-inv-into-into*
 $exI[where\ ?x=the\text{-}inv\text{-}into\ A\ f]$)

next

show $?rhs \implies ?lhs$
by (*force intro: bij-betw-byWitness*)

qed

abbreviation *the-inv* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$
where $the\text{-}inv\ f \equiv the\text{-}inv\text{-}into\ UNIV\ f$

lemma *the-inv-f-f*: $the\text{-}inv\ f\ (f\ x) = x$ **if** $inj\ f$
using *that UNIV-I* **by** (*rule the-inv-into-f-f*)

10.9 Monotonicity

definition *monotone-on* :: $'a\ set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a$
 $\Rightarrow 'b) \Rightarrow bool$

where $monotone\text{-}on\ A\ orda\ ordb\ f \longleftrightarrow (\forall x \in A. \forall y \in A. orda\ x\ y \longrightarrow ordb\ (f\ x)$
 $(f\ y))$

abbreviation *monotone* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow$
 $bool$

where $monotone \equiv monotone\text{-}on\ UNIV$

lemma *monotone-def[no-atp]*: $monotone\ orda\ ordb\ f \longleftrightarrow (\forall x\ y. orda\ x\ y \longrightarrow ordb$
 $(f\ x)\ (f\ y))$

by (*simp add: monotone-on-def*)

Lemma *monotone-def* is provided for backward compatibility.

lemma *monotone-onI*:

$(\bigwedge x y. x \in A \implies y \in A \implies \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f \ y)) \implies \text{monotone-on } A$
 $\text{orda } \text{ordb } f$
by (*simp add: monotone-on-def*)

lemma *monotoneI[intro?]*: $(\bigwedge x y. \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f \ y)) \implies \text{monotone}$
 $\text{orda } \text{ordb } f$

by (*rule monotone-onI*)

lemma *monotone-onD*:

$\text{monotone-on } A \ \text{orda } \text{ordb } f \implies x \in A \implies y \in A \implies \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f$
 $y)$
by (*simp add: monotone-on-def*)

lemma *monotoneD[dest?]*: $\text{monotone } \text{orda } \text{ordb } f \implies \text{orda } x \ y \implies \text{ordb } (f \ x) \ (f \ y)$
by (*rule monotone-onD[of UNIV, simplified]*)

lemma *monotone-on-subset*: $\text{monotone-on } A \ \text{orda } \text{ordb } f \implies B \subseteq A \implies \text{mono-}$
 $\text{tone-on } B \ \text{orda } \text{ordb } f$

by (*auto intro: monotone-onI dest: monotone-onD*)

lemma *monotone-on-empty[simp]*: $\text{monotone-on } \{\}$ $\text{orda } \text{ordb } f$

by (*auto intro: monotone-onI dest: monotone-onD*)

lemma *monotone-on-o*:

assumes

mono-f: $\text{monotone-on } A \ \text{orda } \text{ordb } f$ **and**

mono-g: $\text{monotone-on } B \ \text{ordc } \text{orda } g$ **and**

$g \ ' \ B \subseteq A$

shows $\text{monotone-on } B \ \text{ordc } \text{ordb } (f \circ g)$

proof (*rule monotone-onI*)

fix $x \ y$ **assume** $x \in B$ **and** $y \in B$ **and** $\text{ordc } x \ y$

hence $\text{orda } (g \ x) \ (g \ y)$

by (*rule mono-g[THEN monotone-onD]*)

moreover from $\langle g \ ' \ B \subseteq A \rangle \langle x \in B \rangle \langle y \in B \rangle$ **have** $g \ x \in A$ **and** $g \ y \in A$

unfolding *image-subset-iff* **by** *simp-all*

ultimately show $\text{ordb } ((f \circ g) \ x) \ ((f \circ g) \ y)$

using *mono-f[THEN monotone-onD]* **by** *simp*

qed

10.9.1 Specializations For *ord* Type Class And More

context *ord* **begin**

abbreviation *mono-on* :: $'a \ \text{set} \Rightarrow ('a \Rightarrow 'b :: \text{ord}) \Rightarrow \text{bool}$

where $\text{mono-on } A \equiv \text{monotone-on } A \ (\leq) \ (\leq)$

abbreviation *strict-mono-on* :: $'a \ \text{set} \Rightarrow ('a \Rightarrow 'b :: \text{ord}) \Rightarrow \text{bool}$

where $\text{strict-mono-on } A \equiv \text{monotone-on } A \ (<) \ (<)$

abbreviation *antimono-on* :: 'a set \Rightarrow ('a \Rightarrow 'b :: ord) \Rightarrow bool

where *antimono-on* A \equiv *monotone-on* A (\leq) (\geq)

abbreviation *strict-antimono-on* :: 'a set \Rightarrow ('a \Rightarrow 'b :: ord) \Rightarrow bool

where *strict-antimono-on* A \equiv *monotone-on* A ($<$) ($>$)

lemma *mono-on-def[no-atp]*: *mono-on* A f \longleftrightarrow (\forall r s. r \in A \wedge s \in A \wedge r \leq s \longrightarrow f r \leq f s)

by (*auto simp add: monotone-on-def*)

lemma *strict-mono-on-def[no-atp]*:

strict-mono-on A f \longleftrightarrow (\forall r s. r \in A \wedge s \in A \wedge r $<$ s \longrightarrow f r $<$ f s)

by (*auto simp add: monotone-on-def*)

Lemmas *mono-on-def* and *strict-mono-on-def* are provided for backward compatibility.

lemma *mono-onI*:

(\bigwedge r s. r \in A \Longrightarrow s \in A \Longrightarrow r \leq s \Longrightarrow f r \leq f s) \Longrightarrow *mono-on* A f

by (*rule monotone-onI*)

lemma *strict-mono-onI*:

(\bigwedge r s. r \in A \Longrightarrow s \in A \Longrightarrow r $<$ s \Longrightarrow f r $<$ f s) \Longrightarrow *strict-mono-on* A f

by (*rule monotone-onI*)

lemma *mono-onD*: [*mono-on* A f; r \in A; s \in A; r \leq s] \Longrightarrow f r \leq f s

by (*rule monotone-onD*)

lemma *strict-mono-onD*: [*strict-mono-on* A f; r \in A; s \in A; r $<$ s] \Longrightarrow f r $<$ f s

by (*rule monotone-onD*)

lemma *mono-on-subset*: *mono-on* A f \Longrightarrow B \subseteq A \Longrightarrow *mono-on* B f

by (*rule monotone-on-subset*)

end

lemma *mono-on-greaterD*:

assumes *mono-on* A g x \in A y \in A g x $>$ (g (y:::linorder) :: -:: linorder)

shows x $>$ y

proof (*rule ccontr*)

assume \neg x $>$ y

hence x \leq y **by** (*simp add: not-less*)

from *assms*(1–3) **and this have** g x \leq g y **by** (*rule mono-onD*)

with *assms*(4) **show** False **by** *simp*

qed

context *order* **begin**

abbreviation *mono* :: ('a \Rightarrow 'b::order) \Rightarrow bool

where $mono \equiv mono\text{-on UNIV}$

abbreviation $strict\text{-mono} :: ('a \Rightarrow 'b::order) \Rightarrow bool$
where $strict\text{-mono} \equiv strict\text{-mono-on UNIV}$

abbreviation $antimono :: ('a \Rightarrow 'b::order) \Rightarrow bool$
where $antimono \equiv monotone (\leq) (\lambda x y. y \leq x)$

lemma $mono\text{-def}[no\text{-atp}]$: $mono f \longleftrightarrow (\forall x y. x \leq y \longrightarrow f x \leq f y)$
by ($simp\ add: monotone\text{-on}\text{-def}$)

lemma $strict\text{-mono}\text{-def}[no\text{-atp}]$: $strict\text{-mono} f \longleftrightarrow (\forall x y. x < y \longrightarrow f x < f y)$
by ($simp\ add: monotone\text{-on}\text{-def}$)

lemma $antimono\text{-def}[no\text{-atp}]$: $antimono f \longleftrightarrow (\forall x y. x \leq y \longrightarrow f x \geq f y)$
by ($simp\ add: monotone\text{-on}\text{-def}$)

Lemmas $mono\text{-def}$, $strict\text{-mono}\text{-def}$, and $antimono\text{-def}$ are provided for backward compatibility.

lemma $monoI$ [$intro?$]: $(\bigwedge x y. x \leq y \Longrightarrow f x \leq f y) \Longrightarrow mono f$
by ($rule\ monotoneI$)

lemma $strict\text{-mono}I$ [$intro?$]: $(\bigwedge x y. x < y \Longrightarrow f x < f y) \Longrightarrow strict\text{-mono} f$
by ($rule\ monotoneI$)

lemma $antimonoI$ [$intro?$]: $(\bigwedge x y. x \leq y \Longrightarrow f x \geq f y) \Longrightarrow antimono f$
by ($rule\ monotoneI$)

lemma $monoD$ [$dest?$]: $mono f \Longrightarrow x \leq y \Longrightarrow f x \leq f y$
by ($rule\ monotoneD$)

lemma $strict\text{-mono}D$ [$dest?$]: $strict\text{-mono} f \Longrightarrow x < y \Longrightarrow f x < f y$
by ($rule\ monotoneD$)

lemma $antimonoD$ [$dest?$]: $antimono f \Longrightarrow x \leq y \Longrightarrow f x \geq f y$
by ($rule\ monotoneD$)

lemma $monoE$:

assumes $mono f$
assumes $x \leq y$
obtains $f x \leq f y$

proof

from $assms$ **show** $f x \leq f y$ **by** ($simp\ add: mono\text{-def}$)

qed

lemma $antimonoE$:

fixes $f :: 'a \Rightarrow 'b::order$
assumes $antimono f$
assumes $x \leq y$

```

  obtains  $f x \geq f y$ 
proof
  from assms show  $f x \geq f y$  by (simp add: antimono-def)
qed

```

```

lemma mono-imp-mono-on:  $\text{mono } f \implies \text{mono-on } A f$ 
  by (rule monotone-on-subset[OF - subset-UNIV])

```

```

lemma strict-mono-mono [dest?]:
  assumes strict-mono  $f$ 
  shows mono  $f$ 
proof (rule monoI)
  fix  $x y$ 
  assume  $x \leq y$ 
  show  $f x \leq f y$ 
  proof (cases  $x = y$ )
    case True then show ?thesis by simp
  next
    case False with  $\langle x < y \rangle$  have  $x < y$  by simp
    with assms strict-monoD have  $f x < f y$  by auto
    then show ?thesis by simp
  qed

```

```

  qed
qed

```

```

end

```

```

context linorder begin

```

```

lemma mono-invE:
  fixes  $f :: 'a \Rightarrow 'b::order$ 
  assumes mono  $f$ 
  assumes  $f x < f y$ 
  obtains  $x \leq y$ 
proof
  show  $x \leq y$ 
  proof (rule ccontr)
    assume  $\neg x \leq y$ 
    then have  $y \leq x$  by simp
    with  $\langle \text{mono } f \rangle$  obtain  $f y \leq f x$  by (rule monoE)
    with  $\langle f x < f y \rangle$  show False by simp
  qed
qed

```

```

lemma mono-strict-invE:
  fixes  $f :: 'a \Rightarrow 'b::order$ 
  assumes mono  $f$ 
  assumes  $f x < f y$ 
  obtains  $x < y$ 

```

```

proof
  show  $x < y$ 
  proof (rule ccontr)
    assume  $\neg x < y$ 
    then have  $y \leq x$  by simp
    with  $\langle \text{mono } f \rangle$  obtain  $f y \leq f x$  by (rule monoE)
    with  $\langle f x < f y \rangle$  show False by simp
  qed
qed

lemma strict-mono-eq:
  assumes strict-mono  $f$ 
  shows  $f x = f y \longleftrightarrow x = y$ 
proof
  assume  $f x = f y$ 
  show  $x = y$  proof (cases  $x y$  rule: linorder-cases)
    case less with assms strict-monoD have  $f x < f y$  by auto
    with  $\langle f x = f y \rangle$  show ?thesis by simp
  next
    case equal then show ?thesis .
  next
    case greater with assms strict-monoD have  $f y < f x$  by auto
    with  $\langle f x = f y \rangle$  show ?thesis by simp
  qed
qed simp

lemma strict-mono-less-eq:
  assumes strict-mono  $f$ 
  shows  $f x \leq f y \longleftrightarrow x \leq y$ 
proof
  assume  $x \leq y$ 
  with assms strict-mono-mono monoD show  $f x \leq f y$  by auto
next
  assume  $f x \leq f y$ 
  show  $x \leq y$  proof (rule ccontr)
    assume  $\neg x \leq y$  then have  $y < x$  by simp
    with assms strict-monoD have  $f y < f x$  by auto
    with  $\langle f x \leq f y \rangle$  show False by simp
  qed
qed

lemma strict-mono-less:
  assumes strict-mono  $f$ 
  shows  $f x < f y \longleftrightarrow x < y$ 
  using assms
  by (auto simp add: less-le Orderings.less-le strict-mono-eq strict-mono-less-eq)

end

```

lemma *strict-mono-inv*:
fixes $f :: ('a::linorder) \Rightarrow ('b::linorder)$
assumes *strict-mono f and surj f and inv*: $\bigwedge x. g (f x) = x$
shows *strict-mono g*
proof
fix $x y :: 'b$ **assume** $x < y$
from $\langle \text{surj } f \rangle$ **obtain** $x' y'$ **where** $[\text{simp}]: x = f x' \ y = f y'$ **by** *blast*
with $\langle x < y \rangle$ **and** $\langle \text{strict-mono } f \rangle$ **have** $x' < y'$ **by** $(\text{simp add: strict-mono-less})$
with *inv* **show** $g x < g y$ **by** *simp*
qed

lemma *strict-mono-on-imp-inj-on*:
assumes *strict-mono-on A* $(f :: (- :: linorder) \Rightarrow (- :: preorder))$
shows *inj-on f A*
proof (rule inj-onI)
fix $x y$ **assume** $x \in A \ y \in A \ f x = f y$
thus $x = y$
by $(\text{cases } x \ y \ \text{rule: linorder-cases})$
 $(\text{auto dest: strict-mono-onD}[OF \ \text{assms}, \ \text{of } x \ y] \ \text{strict-mono-onD}[OF \ \text{assms}, \ \text{of } y \ x])$
qed

lemma *strict-mono-on-leD*:
assumes *strict-mono-on A* $(f :: (- :: linorder) \Rightarrow - :: preorder) \ x \in A \ y \in A \ x \leq y$
shows $f x \leq f y$
proof $(\text{cases } x = y)$
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with *assms* **have** $f x < f y$
using *strict-mono-onD* $[OF \ \text{assms}(1)]$ **by** *simp*
then show *?thesis* **by** $(\text{rule less-imp-le})$
qed

lemma *strict-mono-on-eqD*:
fixes $f :: (- :: linorder) \Rightarrow (- :: preorder)$
assumes *strict-mono-on A* $f f x = f y \ x \in A \ y \in A$
shows $y = x$
using *assms* **by** $(\text{cases rule: linorder-cases}) \ (\text{auto dest: strict-mono-onD})$

lemma *strict-mono-on-imp-mono-on*:
strict-mono-on A $(f :: (- :: linorder) \Rightarrow - :: preorder) \Longrightarrow \text{mono-on } A \ f$
by $(\text{rule mono-onI, rule strict-mono-on-leD})$

lemma *mono-imp-strict-mono*:
fixes $f :: 'a::order \Rightarrow 'b::order$
shows $\llbracket \text{mono-on } S \ f; \ \text{inj-on } f \ S \rrbracket \Longrightarrow \text{strict-mono-on } S \ f$

by (auto simp add: monotone-on-def order-less-le inj-on-eq-iff)

lemma *strict-mono-iff-mono*:

fixes $f :: 'a::\text{linorder} \Rightarrow 'b::\text{order}$

shows $\text{strict-mono-on } S f \longleftrightarrow \text{mono-on } S f \wedge \text{inj-on } f S$

proof

show $\text{strict-mono-on } S f \Longrightarrow \text{mono-on } S f \wedge \text{inj-on } f S$

by (simp add: strict-mono-on-imp-inj-on strict-mono-on-imp-mono-on)

qed (auto intro: mono-imp-strict-mono)

lemma *antimono-imp-strict-antimono*:

fixes $f :: 'a::\text{order} \Rightarrow 'b::\text{order}$

shows $\llbracket \text{antimono-on } S f; \text{inj-on } f S \rrbracket \Longrightarrow \text{strict-antimono-on } S f$

by (auto simp add: monotone-on-def order-less-le inj-on-eq-iff)

lemma *strict-antimono-iff-antimono*:

fixes $f :: 'a::\text{linorder} \Rightarrow 'b::\text{order}$

shows $\text{strict-antimono-on } S f \longleftrightarrow \text{antimono-on } S f \wedge \text{inj-on } f S$

proof

show $\text{strict-antimono-on } S f \Longrightarrow \text{antimono-on } S f \wedge \text{inj-on } f S$

by (force simp add: monotone-on-def intro: linorder-inj-onI)

qed (auto intro: antimono-imp-strict-antimono)

lemma *mono-compose*: $\text{mono } Q \Longrightarrow \text{mono } (\lambda i x. Q i (f x))$

unfolding *mono-def le-fun-def* by auto

lemma *mono-add*:

fixes $a :: 'a::\text{ordered-ab-semigroup-add}$

shows $\text{mono } ((+) a)$

by (simp add: add-left-mono monoI)

lemma (in *semilattice-inf*) *mono-inf*: $\text{mono } f \Longrightarrow f (A \sqcap B) \leq f A \sqcap f B$

for $f :: 'a \Rightarrow 'b::\text{semilattice-inf}$

by (auto simp add: mono-def intro: Lattices.inf-greatest)

lemma (in *semilattice-sup*) *mono-sup*: $\text{mono } f \Longrightarrow f A \sqcup f B \leq f (A \sqcup B)$

for $f :: 'a \Rightarrow 'b::\text{semilattice-sup}$

by (auto simp add: mono-def intro: Lattices.sup-least)

lemma (in *linorder*) *min-of-mono*: $\text{mono } f \Longrightarrow \min (f m) (f n) = f (\min m n)$

by (auto simp: mono-def Orderings.min-def min-def intro: Orderings.antisym)

lemma (in *linorder*) *max-of-mono*: $\text{mono } f \Longrightarrow \max (f m) (f n) = f (\max m n)$

by (auto simp: mono-def Orderings.max-def max-def intro: Orderings.antisym)

lemma (in *linorder*)

max-of-antimono: $\text{antimono } f \Longrightarrow \max (f x) (f y) = f (\min x y)$ **and**

min-of-antimono: $\text{antimono } f \Longrightarrow \min (f x) (f y) = f (\max x y)$

by (auto simp: antimono-def Orderings.max-def max-def Orderings.min-def min-def)

intro!: *antisym*)

lemma (*in linorder*) *strict-mono-imp-inj-on*: *strict-mono* $f \implies$ *inj-on* $f A$
by (*auto intro!*: *inj-onI dest*: *strict-mono-eq*)

lemma *mono-Int*: *mono* $f \implies f (A \cap B) \subseteq f A \cap f B$
by (*fact mono-inf*)

lemma *mono-Un*: *mono* $f \implies f A \cup f B \subseteq f (A \cup B)$
by (*fact mono-sup*)

10.9.2 Least value operator

lemma *Least-mono*: *mono* $f \implies \exists x \in S. \forall y \in S. x \leq y \implies (LEAST y. y \in f ' S)$
 $= f (LEAST x. x \in S)$
for $f :: 'a::order \Rightarrow 'b::order$
 — Courtesy of Stephan Merz
apply *clarify*
apply (*erule-tac* $P = \lambda x. x \in S$ **in** *LeastI2-order*)
apply *fast*
apply (*rule LeastI2-order*)
apply (*auto elim*: *monoD intro!*: *order-antisym*)
done

10.10 Setup

10.10.1 Proof tools

Simplify terms of the form $f(\dots, x:=y, \dots, x:=z, \dots)$ to $f(\dots, x:=z, \dots)$

simproc-setup *fun-upd2* ($f(v := w, x := y)$) = \langle
let
fun *gen-fun-upd* *NONE* $T - - = NONE$
 | *gen-fun-upd* (*SOME* f) $T x y = SOME (Const (\mathbf{const-name} \langle \mathit{fun-upd} \rangle, T)$
 $\$ f \$ x \$ y)$
fun *dest-fun-T1* (*Type* $(-, T :: Ts)$) = T
fun *find-double* (t *as* $Const (\mathbf{const-name} \langle \mathit{fun-upd} \rangle, T)$) $\$ f \$ x \$ y =$
let
fun *find* ($Const (\mathbf{const-name} \langle \mathit{fun-upd} \rangle, T)$) $\$ g \$ v \$ w =$
if v *aconv* x *then* *SOME* g *else* *gen-fun-upd* (*find* g) $T v w$
 | *find* $t = NONE$
in (*dest-fun-T1* T , *gen-fun-upd* (*find* f) $T x y$) *end*

val *ss* = *simpset-of* **context**

fun *proc* *ctxt* *ct* =
let
val $t = Thm.term-of$ ct
in
 (*case find-double* t *of*

```

      (T, NONE) => NONE
    | (T, SOME rhs) =>
      SOME (Goal.prove ctxt [] [] (Logic.mk-equals (t, rhs))
        (fn - =>
          resolve-tac ctxt [eq-reflection] 1 THEN
          resolve-tac ctxt @{{thms ext}} 1 THEN
          simp-tac (put-simpset ss ctxt 1)))
    end
  in K proc end
>

```

10.10.2 Functorial structure of types

ML-file *<Tools/functor.ML>*

```

functor map-fun: map-fun
  by (simp-all add: fun-eq-iff)

functor vimage
  by (simp-all add: fun-eq-iff vimage-comp)

```

Legacy theorem names

```

lemmas o-def = comp-def
lemmas o-apply = comp-apply
lemmas o-assoc = comp-assoc [symmetric]
lemmas id-o = id-comp
lemmas o-id = comp-id
lemmas o-eq-dest = comp-eq-dest
lemmas o-eq-elim = comp-eq-elim
lemmas o-eq-dest-lhs = comp-eq-dest-lhs
lemmas o-eq-id-dest = comp-eq-id-dest

```

end

11 Complete lattices

```

theory Complete-Lattices
  imports Fun
begin

```

11.1 Syntactic infimum and supremum operations

```

class Inf =
  fixes Inf :: 'a set => 'a (⌊ - [900] 900)

class Sup =
  fixes Sup :: 'a set => 'a (⌋ - [900] 900)

syntax

```

$-INF1 \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad ((\exists INF \text{-./ -}) [0, 10] 10)$
 $-INF \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad ((\exists INF \text{-\in-./ -}) [0, 0, 10] 10)$
 $-SUP1 \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad ((\exists SUP \text{-./ -}) [0, 10] 10)$
 $-SUP \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad ((\exists SUP \text{-\in-./ -}) [0, 0, 10] 10)$

syntax

$-INF1 \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad ((\exists \sqcap \text{-./ -}) [0, 10] 10)$
 $-INF \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad ((\exists \sqcap \text{-\in-./ -}) [0, 0, 10] 10)$
 $-SUP1 \quad :: \text{pttrns} \Rightarrow 'b \Rightarrow 'b \quad ((\exists \sqcup \text{-./ -}) [0, 10] 10)$
 $-SUP \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \quad ((\exists \sqcup \text{-\in-./ -}) [0, 0, 10] 10)$

translations

$\sqcap x y. f \quad \equiv \sqcap x. \sqcap y. f$
 $\sqcap x. f \quad \equiv \sqcap (\text{CONST range } (\lambda x. f))$
 $\sqcap x \in A. f \quad \equiv \text{CONST Inf } ((\lambda x. f) ' A)$
 $\sqcup x y. f \quad \equiv \sqcup x. \sqcup y. f$
 $\sqcup x. f \quad \equiv \sqcup (\text{CONST range } (\lambda x. f))$
 $\sqcup x \in A. f \quad \equiv \text{CONST Sup } ((\lambda x. f) ' A)$

context *Inf***begin**

lemma *INF-image*: $\sqcap (g ' f ' A) = \sqcap ((g \circ f) ' A)$
by (*simp add: image-comp*)

lemma *INF-identity-eq* [*simp*]: $(\sqcap x \in A. x) = \sqcap A$
by *simp*

lemma *INF-id-eq* [*simp*]: $\sqcap (\text{id} ' A) = \sqcap A$
by *simp*

lemma *INF-cong*: $A = B \implies (\bigwedge x. x \in B \implies C x = D x) \implies \sqcap (C ' A) = \sqcap (D ' B)$
by (*simp add: image-def*)

lemma *INF-cong-simp*:

$A = B \implies (\bigwedge x. x \in B = \text{simp} \implies C x = D x) \implies \sqcap (C ' A) = \sqcap (D ' B)$
unfolding *simp-implies-def* **by** (*fact INF-cong*)

end**context** *Sup***begin**

lemma *SUP-image*: $\sqcup (g ' f ' A) = \sqcup ((g \circ f) ' A)$
by(*fact Inf.INF-image*)

lemma *SUP-identity-eq* [*simp*]: $(\sqcup x \in A. x) = \sqcup A$
by(*fact Inf.INF-identity-eq*)

lemma *SUP-id-eq* [*simp*]: $\bigsqcup (id \text{ ' } A) = \bigsqcup A$
by (*fact Inf.INF-id-eq*)

lemma *SUP-cong*: $A = B \implies (\bigwedge x. x \in B \implies C x = D x) \implies \bigsqcup (C \text{ ' } A) = \bigsqcup (D \text{ ' } B)$
by (*fact Inf.INF-cong*)

lemma *SUP-cong-simp*:
 $A = B \implies (\bigwedge x. x \in B = \text{simp} \implies C x = D x) \implies \bigsqcup (C \text{ ' } A) = \bigsqcup (D \text{ ' } B)$
by (*fact Inf.INF-cong-simp*)

end

11.2 Abstract complete lattices

A complete lattice always has a bottom and a top, so we include them into the following type class, along with assumptions that define bottom and top in terms of infimum and supremum.

```
class complete-lattice = lattice + Inf + Sup + bot + top +
  assumes Inf-lower:  $x \in A \implies \prod A \leq x$ 
  and Inf-greatest:  $(\bigwedge x. x \in A \implies z \leq x) \implies z \leq \prod A$ 
  and Sup-upper:  $x \in A \implies x \leq \bigsqcup A$ 
  and Sup-least:  $(\bigwedge x. x \in A \implies x \leq z) \implies \bigsqcup A \leq z$ 
  and Inf-empty [simp]:  $\prod \{\} = \top$ 
  and Sup-empty [simp]:  $\bigsqcup \{\} = \perp$ 
begin

subclass bounded-lattice
proof
  fix a
  show  $\perp \leq a$ 
  by (auto intro: Sup-least simp only: Sup-empty [symmetric])
  show  $a \leq \top$ 
  by (auto intro: Inf-greatest simp only: Inf-empty [symmetric])
qed
```

lemma *dual-complete-lattice*: *class.complete-lattice* *Sup* *Inf* *sup* (\geq) ($>$) *inf* \top \perp
by (*auto intro!: class.complete-lattice.intro dual-lattice*)
 (*unfold-locales, (fact Inf-empty Sup-empty Sup-upper Sup-least Inf-lower Inf-greatest)*+))

end

context *complete-lattice*
begin

lemma *Sup-eqI*:
 $(\bigwedge y. y \in A \implies y \leq x) \implies (\bigwedge y. (\bigwedge z. z \in A \implies z \leq y) \implies x \leq y) \implies \bigsqcup A = x$

by (*blast intro: order.antisym Sup-least Sup-upper*)

lemma *Inf-eqI*:

$(\bigwedge i. i \in A \implies x \leq i) \implies (\bigwedge y. (\bigwedge i. i \in A \implies y \leq i) \implies y \leq x) \implies \bigcap A = x$
by (*blast intro: order.antisym Inf-greatest Inf-lower*)

lemma *SUP-eqI*:

$(\bigwedge i. i \in A \implies f i \leq x) \implies (\bigwedge y. (\bigwedge i. i \in A \implies f i \leq y) \implies x \leq y) \implies$
 $(\bigsqcup_{i \in A} f i) = x$
using *Sup-eqI [of f ‘ A x]* by *auto*

lemma *INF-eqI*:

$(\bigwedge i. i \in A \implies x \leq f i) \implies (\bigwedge y. (\bigwedge i. i \in A \implies f i \geq y) \implies x \geq y) \implies$
 $(\bigcap_{i \in A} f i) = x$
using *Inf-eqI [of f ‘ A x]* by *auto*

lemma *INF-lower*: $i \in A \implies (\bigcap_{i \in A} f i) \leq f i$

using *Inf-lower [of - f ‘ A]* by *simp*

lemma *INF-greatest*: $(\bigwedge i. i \in A \implies u \leq f i) \implies u \leq (\bigcap_{i \in A} f i)$

using *Inf-greatest [of f ‘ A]* by *auto*

lemma *SUP-upper*: $i \in A \implies f i \leq (\bigsqcup_{i \in A} f i)$

using *Sup-upper [of - f ‘ A]* by *simp*

lemma *SUP-least*: $(\bigwedge i. i \in A \implies f i \leq u) \implies (\bigsqcup_{i \in A} f i) \leq u$

using *Sup-least [of f ‘ A]* by *auto*

lemma *Inf-lower2*: $u \in A \implies u \leq v \implies \bigcap A \leq v$

using *Inf-lower [of u A]* by *auto*

lemma *INF-lower2*: $i \in A \implies f i \leq u \implies (\bigcap_{i \in A} f i) \leq u$

using *INF-lower [of i A f]* by *auto*

lemma *Sup-upper2*: $u \in A \implies v \leq u \implies v \leq \bigsqcup A$

using *Sup-upper [of u A]* by *auto*

lemma *SUP-upper2*: $i \in A \implies u \leq f i \implies u \leq (\bigsqcup_{i \in A} f i)$

using *SUP-upper [of i A f]* by *auto*

lemma *le-Inf-iff*: $b \leq \bigcap A \iff (\forall a \in A. b \leq a)$

by (*auto intro: Inf-greatest dest: Inf-lower*)

lemma *le-INF-iff*: $u \leq (\bigcap_{i \in A} f i) \iff (\forall i \in A. u \leq f i)$

using *le-Inf-iff [of - f ‘ A]* by *simp*

lemma *Sup-le-iff*: $\bigsqcup A \leq b \iff (\forall a \in A. a \leq b)$

by (*auto intro: Sup-least dest: Sup-upper*)

lemma *SUP-le-iff*: $(\bigsqcup_{i \in A}. f\ i) \leq u \iff (\forall i \in A. f\ i \leq u)$
using *Sup-le-iff* [*of f ‘ A*] **by** *simp*

lemma *Inf-insert* [*simp*]: $\prod(\text{insert } a\ A) = a \sqcap \prod A$
by (*auto intro: le-infI le-infI1 le-infI2 order.antisym Inf-greatest Inf-lower*)

lemma *INF-insert*: $(\prod_{x \in \text{insert } a\ A}. f\ x) = f\ a \sqcap \prod(f\ ‘\ A)$
by *simp*

lemma *Sup-insert* [*simp*]: $\bigsqcup(\text{insert } a\ A) = a \sqcup \bigsqcup A$
by (*auto intro: le-supI le-supI1 le-supI2 order.antisym Sup-least Sup-upper*)

lemma *SUP-insert*: $(\bigsqcup_{x \in \text{insert } a\ A}. f\ x) = f\ a \sqcup \bigsqcup(f\ ‘\ A)$
by *simp*

lemma *INF-empty*: $(\prod_{x \in \{\}}. f\ x) = \top$
by *simp*

lemma *SUP-empty*: $(\bigsqcup_{x \in \{\}}. f\ x) = \perp$
by *simp*

lemma *Inf-UNIV* [*simp*]: $\prod UNIV = \perp$
by (*auto intro!: order.antisym Inf-lower*)

lemma *Sup-UNIV* [*simp*]: $\bigsqcup UNIV = \top$
by (*auto intro!: order.antisym Sup-upper*)

lemma *Inf-eq-Sup*: $\prod A = \bigsqcup \{b. \forall a \in A. b \leq a\}$
by (*auto intro: order.antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Sup-eq-Inf*: $\bigsqcup A = \prod \{b. \forall a \in A. a \leq b\}$
by (*auto intro: order.antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Inf-superset-mono*: $B \subseteq A \implies \prod A \leq \prod B$
by (*auto intro: Inf-greatest Inf-lower*)

lemma *Sup-subset-mono*: $A \subseteq B \implies \bigsqcup A \leq \bigsqcup B$
by (*auto intro: Sup-least Sup-upper*)

lemma *Inf-mono*:
assumes $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$
shows $\prod A \leq \prod B$
proof (*rule Inf-greatest*)
fix *b* **assume** $b \in B$
with *assms* **obtain** *a* **where** $a \in A$ **and** $a \leq b$ **by** *blast*
from $\langle a \in A \rangle$ **have** $\prod A \leq a$ **by** (*rule Inf-lower*)
with $\langle a \leq b \rangle$ **show** $\prod A \leq b$ **by** *auto*
qed

lemma *INF-mono*: $(\bigwedge m. m \in B \implies \exists n \in A. f\ n \leq g\ m) \implies (\prod n \in A. f\ n) \leq (\prod n \in B. g\ n)$

using *Inf-mono* [of $g\ ' B\ f\ ' A$] **by** *auto*

lemma *INF-mono'*: $(\bigwedge x. f\ x \leq g\ x) \implies (\prod x \in A. f\ x) \leq (\prod x \in A. g\ x)$

by (*rule INF-mono*) *auto*

lemma *Sup-mono*:

assumes $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$

shows $\bigsqcup A \leq \bigsqcup B$

proof (*rule Sup-least*)

fix a **assume** $a \in A$

with *assms* **obtain** b **where** $b \in B$ **and** $a \leq b$ **by** *blast*

from $\langle b \in B \rangle$ **have** $b \leq \bigsqcup B$ **by** (*rule Sup-upper*)

with $\langle a \leq b \rangle$ **show** $a \leq \bigsqcup B$ **by** *auto*

qed

lemma *SUP-mono*: $(\bigwedge n. n \in A \implies \exists m \in B. f\ n \leq g\ m) \implies (\bigsqcup n \in A. f\ n) \leq (\bigsqcup n \in B. g\ n)$

using *Sup-mono* [of $f\ ' A\ g\ ' B$] **by** *auto*

lemma *SUP-mono'*: $(\bigwedge x. f\ x \leq g\ x) \implies (\bigsqcup x \in A. f\ x) \leq (\bigsqcup x \in A. g\ x)$

by (*rule SUP-mono*) *auto*

lemma *INF-superset-mono*: $B \subseteq A \implies (\bigwedge x. x \in B \implies f\ x \leq g\ x) \implies (\prod x \in A. f\ x) \leq (\prod x \in B. g\ x)$

— The last inclusion is POSITIVE!

by (*blast intro: INF-mono dest: subsetD*)

lemma *SUP-subset-mono*: $A \subseteq B \implies (\bigwedge x. x \in A \implies f\ x \leq g\ x) \implies (\bigsqcup x \in A. f\ x) \leq (\bigsqcup x \in B. g\ x)$

by (*blast intro: SUP-mono dest: subsetD*)

lemma *Inf-less-eq*:

assumes $\bigwedge v. v \in A \implies v \leq u$

and $A \neq \{\}$

shows $\prod A \leq u$

proof —

from $\langle A \neq \{\} \rangle$ **obtain** v **where** $v \in A$ **by** *blast*

moreover from $\langle v \in A \rangle$ *assms*(1) **have** $v \leq u$ **by** *blast*

ultimately show *?thesis* **by** (*rule Inf-lower2*)

qed

lemma *less-eq-Sup*:

assumes $\bigwedge v. v \in A \implies u \leq v$

and $A \neq \{\}$

shows $u \leq \bigsqcup A$

proof —

from $\langle A \neq \{\} \rangle$ **obtain** v **where** $v \in A$ **by** *blast*

moreover from $\langle v \in A \rangle$ *assms(1)* have $u \leq v$ by *blast*
ultimately show *?thesis* by (rule *Sup-upper2*)
qed

lemma *INF-eq*:

assumes $\bigwedge i. i \in A \implies \exists j \in B. f i \geq g j$

and $\bigwedge j. j \in B \implies \exists i \in A. g j \geq f i$

shows $\bigcap (f \text{ ` } A) = \bigcap (g \text{ ` } B)$

by (intro *order.antisym INF-greatest*) (blast intro: *INF-lower2 dest: assms*)+

lemma *SUP-eq*:

assumes $\bigwedge i. i \in A \implies \exists j \in B. f i \leq g j$

and $\bigwedge j. j \in B \implies \exists i \in A. g j \leq f i$

shows $\bigcup (f \text{ ` } A) = \bigcup (g \text{ ` } B)$

by (intro *order.antisym SUP-least*) (blast intro: *SUP-upper2 dest: assms*)+

lemma *less-eq-Inf-inter*: $\bigcap A \sqcup \bigcap B \leq \bigcap (A \cap B)$

by (auto intro: *Inf-greatest Inf-lower*)

lemma *Sup-inter-less-eq*: $\bigcup (A \cap B) \leq \bigcup A \cap \bigcup B$

by (auto intro: *Sup-least Sup-upper*)

lemma *Inf-union-distrib*: $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$

by (rule *order.antisym*) (auto intro: *Inf-greatest Inf-lower le-infI1 le-infI2*)

lemma *INF-union*: $(\bigcap i \in A \cup B. M i) = (\bigcap i \in A. M i) \cap (\bigcap i \in B. M i)$

by (auto intro!: *order.antisym INF-mono intro: le-infI1 le-infI2 INF-greatest INF-lower*)

lemma *Sup-union-distrib*: $\bigcup (A \cup B) = \bigcup A \sqcup \bigcup B$

by (rule *order.antisym*) (auto intro: *Sup-least Sup-upper le-supI1 le-supI2*)

lemma *SUP-union*: $(\bigcup i \in A \cup B. M i) = (\bigcup i \in A. M i) \sqcup (\bigcup i \in B. M i)$

by (auto intro!: *order.antisym SUP-mono intro: le-supI1 le-supI2 SUP-least SUP-upper*)

lemma *INF-inf-distrib*: $(\bigcap a \in A. f a) \cap (\bigcap a \in A. g a) = (\bigcap a \in A. f a \cap g a)$

by (rule *order.antisym*) (rule *INF-greatest*, auto intro: *le-infI1 le-infI2 INF-lower INF-mono*)

lemma *SUP-sup-distrib*: $(\bigcup a \in A. f a) \sqcup (\bigcup a \in A. g a) = (\bigcup a \in A. f a \sqcup g a)$

(is *?L = ?R*)

proof (rule *order.antisym*)

show *?L ≤ ?R*

by (auto intro: *le-supI1 le-supI2 SUP-upper SUP-mono*)

show *?R ≤ ?L*

by (rule *SUP-least*) (auto intro: *le-supI1 le-supI2 SUP-upper*)

qed

lemma *Inf-top-conv* [*simp*]:

$\prod A = \top \longleftrightarrow (\forall x \in A. x = \top)$
 $\top = \prod A \longleftrightarrow (\forall x \in A. x = \top)$
proof –
show $\prod A = \top \longleftrightarrow (\forall x \in A. x = \top)$
proof
assume $\forall x \in A. x = \top$
then have $A = \{\} \vee A = \{\top\}$ **by** *auto*
then show $\prod A = \top$ **by** *auto*
next
assume $\prod A = \top$
show $\forall x \in A. x = \top$
proof (*rule ccontr*)
assume $\neg (\forall x \in A. x = \top)$
then obtain x **where** $x \in A$ **and** $x \neq \top$ **by** *blast*
then obtain B **where** $A = \text{insert } x B$ **by** *blast*
with $\langle \prod A = \top \rangle \langle x \neq \top \rangle$ **show** *False* **by** *simp*
qed
qed
then show $\top = \prod A \longleftrightarrow (\forall x \in A. x = \top)$ **by** *auto*
qed

lemma *INF-top-conv* [*simp*]:
 $(\prod x \in A. B x) = \top \longleftrightarrow (\forall x \in A. B x = \top)$
 $\top = (\prod x \in A. B x) \longleftrightarrow (\forall x \in A. B x = \top)$
using *Inf-top-conv* [*of B ' A*] **by** *simp-all*

lemma *Sup-bot-conv* [*simp*]:
 $\bigsqcup A = \perp \longleftrightarrow (\forall x \in A. x = \perp)$
 $\perp = \bigsqcup A \longleftrightarrow (\forall x \in A. x = \perp)$
using *dual-complete-lattice*
by (*rule complete-lattice.Inf-top-conv*)+

lemma *SUP-bot-conv* [*simp*]:
 $(\bigsqcup x \in A. B x) = \perp \longleftrightarrow (\forall x \in A. B x = \perp)$
 $\perp = (\bigsqcup x \in A. B x) \longleftrightarrow (\forall x \in A. B x = \perp)$
using *Sup-bot-conv* [*of B ' A*] **by** *simp-all*

lemma *INF-constant*: $(\prod y \in A. c) = (\text{if } A = \{\} \text{ then } \top \text{ else } c)$
by (*auto intro: order.antisym INF-lower INF-greatest*)

lemma *SUP-constant*: $(\bigsqcup y \in A. c) = (\text{if } A = \{\} \text{ then } \perp \text{ else } c)$
by (*auto intro: order.antisym SUP-upper SUP-least*)

lemma *INF-const* [*simp*]: $A \neq \{\} \implies (\prod i \in A. f) = f$
by (*simp add: INF-constant*)

lemma *SUP-const* [*simp*]: $A \neq \{\} \implies (\bigsqcup i \in A. f) = f$
by (*simp add: SUP-constant*)

lemma *INF-top* [*simp*]: $(\prod_{x \in A}. \top) = \top$
by (*cases* $A = \{\}$) *simp-all*

lemma *SUP-bot* [*simp*]: $(\bigsqcup_{x \in A}. \perp) = \perp$
by (*cases* $A = \{\}$) *simp-all*

lemma *INF-commute*: $(\prod_{i \in A}. \prod_{j \in B}. f\ i\ j) = (\prod_{j \in B}. \prod_{i \in A}. f\ i\ j)$
by (*iprover intro: INF-lower INF-greatest order-trans order.antisym*)

lemma *SUP-commute*: $(\bigsqcup_{i \in A}. \bigsqcup_{j \in B}. f\ i\ j) = (\bigsqcup_{j \in B}. \bigsqcup_{i \in A}. f\ i\ j)$
by (*iprover intro: SUP-upper SUP-least order-trans order.antisym*)

lemma *INF-absorb*:

assumes $k \in I$

shows $A\ k \sqcap (\prod_{i \in I}. A\ i) = (\prod_{i \in I}. A\ i)$

proof –

from *assms* **obtain** J **where** $I = \text{insert } k\ J$ **by** *blast*

then show *?thesis* **by** *simp*

qed

lemma *SUP-absorb*:

assumes $k \in I$

shows $A\ k \sqcup (\bigsqcup_{i \in I}. A\ i) = (\bigsqcup_{i \in I}. A\ i)$

proof –

from *assms* **obtain** J **where** $I = \text{insert } k\ J$ **by** *blast*

then show *?thesis* **by** *simp*

qed

lemma *INF-inf-const1*: $I \neq \{\} \implies (\prod_{i \in I}. \text{inf } x\ (f\ i)) = \text{inf } x\ (\prod_{i \in I}. f\ i)$
by (*intro order.antisym INF-greatest inf-mono order-refl INF-lower*)
(auto intro: INF-lower2 le-infI2 intro!: INF-mono)

lemma *INF-inf-const2*: $I \neq \{\} \implies (\prod_{i \in I}. \text{inf } (f\ i)\ x) = \text{inf } (\prod_{i \in I}. f\ i)\ x$
using *INF-inf-const1* [*of I x f*] **by** (*simp add: inf-commute*)

lemma *less-INF-D*:

assumes $y < (\prod_{i \in A}. f\ i)$ $i \in A$

shows $y < f\ i$

proof –

note $\langle y < (\prod_{i \in A}. f\ i) \rangle$

also have $(\prod_{i \in A}. f\ i) \leq f\ i$ **using** $\langle i \in A \rangle$

by (*rule INF-lower*)

finally show $y < f\ i$.

qed

lemma *SUP-lessD*:

assumes $(\bigsqcup_{i \in A}. f\ i) < y$ $i \in A$

shows $f\ i < y$

proof –

have $f i \leq (\bigsqcup_{i \in A} f i)$
using $\langle i \in A \rangle$ **by** (rule *SUP-upper*)
also note $\langle (\bigsqcup_{i \in A} f i) < y \rangle$
finally show $f i < y$.
qed

lemma *INF-UNIV-bool-expand*: $(\prod b. A b) = A \text{ True} \sqcap A \text{ False}$
by (simp add: *UNIV-bool inf-commute*)

lemma *SUP-UNIV-bool-expand*: $(\bigsqcup b. A b) = A \text{ True} \sqcup A \text{ False}$
by (simp add: *UNIV-bool sup-commute*)

lemma *Inf-le-Sup*: $A \neq \{\} \implies \text{Inf } A \leq \text{Sup } A$
by (blast intro: *Sup-upper2 Inf-lower ex-in-conv*)

lemma *INF-le-SUP*: $A \neq \{\} \implies \prod (f ' A) \leq \bigsqcup (f ' A)$
using *Inf-le-Sup [of f ' A]* **by** simp

lemma *INF-eq-const*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies f i = x) \implies \prod (f ' I) = x$
by (auto intro: *INF-eqI*)

lemma *SUP-eq-const*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies f i = x) \implies \bigsqcup (f ' I) = x$
by (auto intro: *SUP-eqI*)

lemma *INF-eq-iff*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies f i \leq c) \implies \prod (f ' I) = c \longleftrightarrow$
 $(\forall i \in I. f i = c)$
by (auto intro: *INF-eq-const INF-lower order.antisym*)

lemma *SUP-eq-iff*: $I \neq \{\} \implies (\bigwedge i. i \in I \implies c \leq f i) \implies \bigsqcup (f ' I) = c \longleftrightarrow$
 $(\forall i \in I. f i = c)$
by (auto intro: *SUP-eq-const SUP-upper order.antisym*)

end

context *complete-lattice*

begin

lemma *Sup-Inf-le*: $\text{Sup} (\text{Inf} ' \{f ' A \mid f . (\forall Y \in A . f Y \in Y)\}) \leq \text{Inf} (\text{Sup} ' A)$
by (rule *SUP-least, clarify, rule INF-greatest, simp add: INF-lower2 Sup-upper*)
end

class *complete-distrib-lattice* = *complete-lattice* +

assumes *Inf-Sup-le*: $\text{Inf} (\text{Sup} ' A) \leq \text{Sup} (\text{Inf} ' \{f ' A \mid f . (\forall Y \in A . f Y \in Y)\})$
begin

lemma *Inf-Sup*: $\text{Inf} (\text{Sup} ' A) = \text{Sup} (\text{Inf} ' \{f ' A \mid f . (\forall Y \in A . f Y \in Y)\})$
by (rule *order.antisym, rule Inf-Sup-le, rule Sup-Inf-le*)

subclass *distrib-lattice*

```

proof
  fix a b c
  show  $a \sqcup b \sqcap c = (a \sqcup b) \sqcap (a \sqcup c)$ 
  proof (rule order.antisym, simp-all, safe)
    show  $b \sqcap c \leq a \sqcup b$ 
      by (rule le-infI1, simp)
    show  $b \sqcap c \leq a \sqcup c$ 
      by (rule le-infI2, simp)
    have [simp]:  $a \sqcap c \leq a \sqcup b \sqcap c$ 
      by (rule le-infI1, simp)
    have [simp]:  $b \sqcap a \leq a \sqcup b \sqcap c$ 
      by (rule le-infI2, simp)
    have  $\sqcap (Sup \text{ ' } \{\{a, b\}, \{a, c\}\}) =$ 
       $\sqcup (Inf \text{ ' } \{f \text{ ' } \{\{a, b\}, \{a, c\}\} \mid f. \forall Y \in \{\{a, b\}, \{a, c\}\}. f Y \in Y\})$ 
      by (rule Inf-Sup)
    from this show  $(a \sqcup b) \sqcap (a \sqcup c) \leq a \sqcup b \sqcap c$ 
      apply simp
      by (rule SUP-least, safe, simp-all)
  qed
qed
end

context complete-lattice
begin
context
  fixes f :: 'a  $\Rightarrow$  'b::complete-lattice
  assumes mono f
begin

lemma mono-Inf:  $f (\sqcap A) \leq (\sqcap x \in A. f x)$ 
  using  $\langle mono f \rangle$  by (auto intro: complete-lattice-class.INF-greatest Inf-lower dest: monoD)

lemma mono-Sup:  $(\sqcup x \in A. f x) \leq f (\sqcup A)$ 
  using  $\langle mono f \rangle$  by (auto intro: complete-lattice-class.SUP-least Sup-upper dest: monoD)

lemma mono-INF:  $f (\sqcap i \in I. A i) \leq (\sqcap x \in I. f (A x))$ 
  by (intro complete-lattice-class.INF-greatest monoD[OF  $\langle mono f \rangle$ ] INF-lower)

lemma mono-SUP:  $(\sqcup x \in I. f (A x)) \leq f (\sqcup i \in I. A i)$ 
  by (intro complete-lattice-class.SUP-least monoD[OF  $\langle mono f \rangle$ ] SUP-upper)

end

end

class complete-boolean-algebra = boolean-algebra + complete-distrib-lattice
begin

```

lemma *uminus-Inf*: $-(\prod A) = \sqcup(\text{uminus } ' A)$
proof (*rule order.antisym*)
 show $-\prod A \leq \sqcup(\text{uminus } ' A)$
 by (*rule compl-le-swap2, rule Inf-greatest, rule compl-le-swap2, rule Sup-upper*)
simp
 show $\sqcup(\text{uminus } ' A) \leq -\prod A$
 by (*rule Sup-least, rule compl-le-swap1, rule Inf-lower*) *auto*
qed

lemma *uminus-INF*: $-(\prod x \in A. B x) = (\sqcup x \in A. - B x)$
by (*simp add: uminus-Inf image-image*)

lemma *uminus-Sup*: $-(\sqcup A) = \prod(\text{uminus } ' A)$
proof –
 have $\sqcup A = -\prod(\text{uminus } ' A)$
 by (*simp add: image-image uminus-INF*)
 then show *?thesis* **by** *simp*
qed

lemma *uminus-SUP*: $-(\sqcup x \in A. B x) = (\prod x \in A. - B x)$
by (*simp add: uminus-Sup image-image*)

end

class *complete-linorder* = *linorder* + *complete-lattice*
begin

lemma *dual-complete-linorder*:
 class.complete-linorder *Sup Inf sup* (\geq) ($>$) *inf* $\top \perp$
by (*rule class.complete-linorder.intro, rule dual-complete-lattice, rule dual-linorder*)

lemma *complete-linorder-inf-min*: *inf* = *min*
by (*auto intro: order.antisym simp add: min-def fun-eq-iff*)

lemma *complete-linorder-sup-max*: *sup* = *max*
by (*auto intro: order.antisym simp add: max-def fun-eq-iff*)

lemma *Inf-less-iff*: $\prod S < a \longleftrightarrow (\exists x \in S. x < a)$
by (*simp add: not-le [symmetric] le-Inf-iff*)

lemma *INF-less-iff*: $(\prod i \in A. f i) < a \longleftrightarrow (\exists x \in A. f x < a)$
by (*simp add: Inf-less-iff [of f ' A]*)

lemma *less-Sup-iff*: $a < \sqcup S \longleftrightarrow (\exists x \in S. a < x)$
by (*simp add: not-le [symmetric] Sup-le-iff*)

lemma *less-SUP-iff*: $a < (\sqcup i \in A. f i) \longleftrightarrow (\exists x \in A. a < f x)$
by (*simp add: less-Sup-iff [of f ' A]*)

lemma *Sup-eq-top-iff* [simp]: $\bigsqcup A = \top \iff (\forall x < \top. \exists i \in A. x < i)$
proof
 assume *: $\bigsqcup A = \top$
 show $(\forall x < \top. \exists i \in A. x < i)$
 unfolding * [symmetric]
 proof (intro allI impI)
 fix x
 assume $x < \bigsqcup A$
 then show $\exists i \in A. x < i$
 by (simp add: less-Sup-iff)
 qed
 next
 assume *: $\forall x < \top. \exists i \in A. x < i$
 show $\bigsqcup A = \top$
 proof (rule ccontr)
 assume $\bigsqcup A \neq \top$
 with top-greatest [of $\bigsqcup A$] have $\bigsqcup A < \top$
 unfolding le-less by auto
 with * have $\bigsqcup A < \bigsqcup A$
 unfolding less-Sup-iff by auto
 then show False by auto
 qed
 qed

lemma *SUP-eq-top-iff* [simp]: $(\bigsqcup i \in A. f i) = \top \iff (\forall x < \top. \exists i \in A. x < f i)$
 using *Sup-eq-top-iff* [of $f \text{ ' } A$] by simp

lemma *Inf-eq-bot-iff* [simp]: $\bigsqcap A = \perp \iff (\forall x > \perp. \exists i \in A. i < x)$
 using *dual-complete-linorder*
 by (rule complete-linorder.Sup-eq-top-iff)

lemma *INF-eq-bot-iff* [simp]: $(\bigsqcap i \in A. f i) = \perp \iff (\forall x > \perp. \exists i \in A. f i < x)$
 using *Inf-eq-bot-iff* [of $f \text{ ' } A$] by simp

lemma *Inf-le-iff*: $\bigsqcap A \leq x \iff (\forall y > x. \exists a \in A. y > a)$
proof safe
 fix y
 assume $x \geq \bigsqcap A$ $y > x$
 then have $y > \bigsqcap A$ by auto
 then show $\exists a \in A. y > a$
 unfolding *Inf-less-iff* .
 qed (auto elim!: allE[*of* - $\bigsqcap A$] simp add: not-le[symmetric] *Inf-lower*)

lemma *INF-le-iff*: $\bigsqcap (f \text{ ' } A) \leq x \iff (\forall y > x. \exists i \in A. y > f i)$
 using *Inf-le-iff* [of $f \text{ ' } A$] by simp

lemma *le-Sup-iff*: $x \leq \bigsqcup A \iff (\forall y < x. \exists a \in A. y < a)$
proof safe

```

fix y
assume  $x \leq \bigsqcup A$   $y < x$ 
then have  $y < \bigsqcup A$  by auto
then show  $\exists a \in A. y < a$ 
  unfolding less-Sup-iff .
qed (auto elim!: allE[of - \bigsqcup A] simp add: not-le[symmetric] Sup-upper)

```

```

lemma le-SUP-iff:  $x \leq \bigsqcup (f \text{ ' } A) \longleftrightarrow (\forall y < x. \exists i \in A. y < f i)$ 
  using le-Sup-iff [of - f \text{ ' } A] by simp

```

end

11.3 Complete lattice on *bool*

```

instantiation bool :: complete-lattice
begin

```

```

definition [simp, code]:  $\sqcap A \longleftrightarrow \text{False} \notin A$ 

```

```

definition [simp, code]:  $\sqcup A \longleftrightarrow \text{True} \in A$ 

```

instance

```

  by standard (auto intro: bool-induct)

```

end

```

lemma not-False-in-image-Ball [simp]:  $\text{False} \notin P \text{ ' } A \longleftrightarrow \text{Ball } A P$ 
  by auto

```

```

lemma True-in-image-Bex [simp]:  $\text{True} \in P \text{ ' } A \longleftrightarrow \text{Bex } A P$ 
  by auto

```

```

lemma INF-bool-eq [simp]:  $(\lambda A f. \sqcap (f \text{ ' } A)) = \text{Ball}$ 
  by (simp add: fun-eq-iff)

```

```

lemma SUP-bool-eq [simp]:  $(\lambda A f. \sqcup (f \text{ ' } A)) = \text{Bex}$ 
  by (simp add: fun-eq-iff)

```

```

instance bool :: complete-boolean-algebra
  by (standard, fastforce)

```

11.4 Complete lattice on $- \Rightarrow -$

```

instantiation fun :: (type, Inf) Inf
begin

```

```

definition  $\sqcap A = (\lambda x. \sqcap f \in A. f x)$ 

```

```

lemma Inf-apply [simp, code]:  $(\sqcap A) x = (\sqcap f \in A. f x)$ 
  by (simp add: Inf-fun-def)

```


instance ..

end

instantiation *fun* :: (*type*, *Sup*) *Sup*
begin

definition $\sqcup A = (\lambda x. \sqcup f \in A. f x)$

lemma *Sup-apply* [*simp*, *code*]: $(\sqcup A) x = (\sqcup f \in A. f x)$
by (*simp add: Sup-fun-def*)

instance ..

end

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

instance

by *standard* (*auto simp add: le-fun-def intro: INF-lower INF-greatest SUP-upper SUP-least*)

end

lemma *INF-apply* [*simp*]: $(\prod y \in A. f y) x = (\prod y \in A. f y x)$
by (*simp add: image-comp*)

lemma *SUP-apply* [*simp*]: $(\sqcup y \in A. f y) x = (\sqcup y \in A. f y x)$
by (*simp add: image-comp*)

11.5 Complete lattice on unary and binary predicates

lemma *Inf1-I*: $(\bigwedge P. P \in A \implies P a) \implies (\prod A) a$
by *auto*

lemma *INF1-I*: $(\bigwedge x. x \in A \implies B x b) \implies (\prod x \in A. B x) b$
by *simp*

lemma *INF2-I*: $(\bigwedge x. x \in A \implies B x b c) \implies (\prod x \in A. B x) b c$
by *simp*

lemma *Inf2-I*: $(\bigwedge r. r \in A \implies r a b) \implies (\prod A) a b$
by *auto*

lemma *Inf1-D*: $(\prod A) a \implies P \in A \implies P a$
by *auto*

lemma *INF1-D*: $(\prod x \in A. B x) b \implies a \in A \implies B a b$
by *simp*

lemma *Inf2-D*: $(\prod A) a b \implies r \in A \implies r a b$
by *auto*

lemma *INF2-D*: $(\prod x \in A. B x) b c \implies a \in A \implies B a b c$
by *simp*

lemma *Inf1-E*:
assumes $(\prod A) a$
obtains $P a \mid P \notin A$
using *assms* **by** *auto*

lemma *INF1-E*:
assumes $(\prod x \in A. B x) b$
obtains $B a b \mid a \notin A$
using *assms* **by** *auto*

lemma *Inf2-E*:
assumes $(\prod A) a b$
obtains $r a b \mid r \notin A$
using *assms* **by** *auto*

lemma *INF2-E*:
assumes $(\prod x \in A. B x) b c$
obtains $B a b c \mid a \notin A$
using *assms* **by** *auto*

lemma *Sup1-I*: $P \in A \implies P a \implies (\bigsqcup A) a$
by *auto*

lemma *SUP1-I*: $a \in A \implies B a b \implies (\bigsqcup x \in A. B x) b$
by *auto*

lemma *Sup2-I*: $r \in A \implies r a b \implies (\bigsqcup A) a b$
by *auto*

lemma *SUP2-I*: $a \in A \implies B a b c \implies (\bigsqcup x \in A. B x) b c$
by *auto*

lemma *Sup1-E*:
assumes $(\bigsqcup A) a$
obtains P **where** $P \in A$ **and** $P a$
using *assms* **by** *auto*

lemma *SUP1-E*:
assumes $(\bigsqcup x \in A. B x) b$
obtains x **where** $x \in A$ **and** $B x b$

using *assms* by *auto*

lemma *Sup2-E*:
 assumes $(\bigsqcup A) a b$
 obtains *r* where $r \in A r a b$
 using *assms* by *auto*

lemma *SUP2-E*:
 assumes $(\bigsqcup_{x \in A} B x) b c$
 obtains *x* where $x \in A B x b c$
 using *assms* by *auto*

11.6 Complete lattice on - set

instantiation *set* :: (type) complete-lattice
begin

definition $\sqcap A = \{x. \sqcap((\lambda B. x \in B) ' A)\}$

definition $\sqcup A = \{x. \sqcup((\lambda B. x \in B) ' A)\}$

instance
 by *standard* (*auto simp add: less-eq-set-def Inf-set-def Sup-set-def le-fun-def*)

end

11.6.1 Inter

abbreviation *Inter* :: 'a set set \Rightarrow 'a set (\cap)
 where $\cap S \equiv \sqcap S$

lemma *Inter-eq*: $\cap A = \{x. \forall B \in A. x \in B\}$

proof (*rule set-eqI*)

fix *x*

have $(\forall Q \in \{P. \exists B \in A. P \longleftrightarrow x \in B\}. Q) \longleftrightarrow (\forall B \in A. x \in B)$

by *auto*

then show $x \in \cap A \longleftrightarrow x \in \{x. \forall B \in A. x \in B\}$

by (*simp add: Inf-set-def image-def*)

qed

lemma *Inter-iff* [*simp*]: $A \in \cap C \longleftrightarrow (\forall X \in C. A \in X)$
 by (*unfold Inter-eq*) *blast*

lemma *InterI* [*intro!*]: $(\bigwedge X. X \in C \Longrightarrow A \in X) \Longrightarrow A \in \cap C$
 by (*simp add: Inter-eq*)

A “destruct” rule – every *X* in *C* contains *A* as an element, but $A \in X$ can hold when $X \in C$ does not! This rule is analogous to *spec*.

lemma *InterD* [*elim*, *Pure.elim*]: $A \in \cap C \Longrightarrow X \in C \Longrightarrow A \in X$

by *auto*

lemma *InterE* [*elim*]: $A \in \bigcap C \implies (X \notin C \implies R) \implies (A \in X \implies R) \implies R$
 — “Classical” elimination rule – does not require proving $X \in C$.
unfolding *Inter-eq* by *blast*

lemma *Inter-lower*: $B \in A \implies \bigcap A \subseteq B$
 by (*fact Inf-lower*)

lemma *Inter-subset*: $(\bigwedge X. X \in A \implies X \subseteq B) \implies A \neq \{\} \implies \bigcap A \subseteq B$
 by (*fact Inf-less-eq*)

lemma *Inter-greatest*: $(\bigwedge X. X \in A \implies C \subseteq X) \implies C \subseteq \bigcap A$
 by (*fact Inf-greatest*)

lemma *Inter-empty*: $\bigcap \{\} = UNIV$
 by (*fact Inf-empty*)

lemma *Inter-UNIV*: $\bigcap UNIV = \{\}$
 by (*fact Inf-UNIV*)

lemma *Inter-insert*: $\bigcap (\text{insert } a \ B) = a \cap \bigcap B$
 by (*fact Inf-insert*)

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$
 by (*fact less-eq-Inf-inter*)

lemma *Inter-Un-distrib*: $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$
 by (*fact Inf-union-distrib*)

lemma *Inter-UNIV-conv* [*simp*]:
 $\bigcap A = UNIV \iff (\forall x \in A. x = UNIV)$
 $UNIV = \bigcap A \iff (\forall x \in A. x = UNIV)$
 by (*fact Inf-top-conv*)⁺

lemma *Inter-anti-mono*: $B \subseteq A \implies \bigcap A \subseteq \bigcap B$
 by (*fact Inf-superset-mono*)

11.6.2 Intersections of families

syntax (*ASCII*)

-*INTER1* :: *pttrns* \implies 'b set \implies 'b set ((*3INT* -./ -) [0, 10] 10)
 -*INTER* :: *pttrn* \implies 'a set \implies 'b set \implies 'b set ((*3INT* -:./ -) [0, 0, 10] 10)

syntax

-*INTER1* :: *pttrns* \implies 'b set \implies 'b set ((*3* \bigcap -./ -) [0, 10] 10)
 -*INTER* :: *pttrn* \implies 'a set \implies 'b set \implies 'b set ((*3* \bigcap - \in -./ -) [0, 0, 10] 10)

syntax (*latex output*)

-*INTER1* :: $pttrns \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ $((\exists \bigcap (\langle \text{unbreakable} \rangle -) / -) [0, 10]$
 $10)$

-*INTER* :: $pttrn \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ $((\exists \bigcap (\langle \text{unbreakable} \rangle - \in -) / -) [0,$
 $0, 10] 10)$

translations

$\bigcap x y. f \equiv \bigcap x. \bigcap y. f$
 $\bigcap x. f \equiv \bigcap (\text{CONST range } (\lambda x. f))$
 $\bigcap x \in A. f \equiv \text{CONST Inter } ((\lambda x. f) ' A)$

lemma *INTER-eq*: $(\bigcap x \in A. B x) = \{y. \forall x \in A. y \in B x\}$
by (*auto intro!*: *INF-eqI*)

lemma *INT-iff* [*simp*]: $b \in (\bigcap x \in A. B x) \longleftrightarrow (\forall x \in A. b \in B x)$
using *Inter-iff* [*of - B ' A*] **by** *simp*

lemma *INT-I* [*intro!*]: $(\bigwedge x. x \in A \Longrightarrow b \in B x) \Longrightarrow b \in (\bigcap x \in A. B x)$
by *auto*

lemma *INT-D* [*elim, Pure.elim*]: $b \in (\bigcap x \in A. B x) \Longrightarrow a \in A \Longrightarrow b \in B a$
by *auto*

lemma *INT-E* [*elim*]: $b \in (\bigcap x \in A. B x) \Longrightarrow (b \in B a \Longrightarrow R) \Longrightarrow (a \notin A \Longrightarrow R) \Longrightarrow R$
 — "Classical" elimination – by the Excluded Middle on $a \in A$.
by *auto*

lemma *Collect-ball-eq*: $\{x. \forall y \in A. P x y\} = (\bigcap y \in A. \{x. P x y\})$
by *blast*

lemma *Collect-all-eq*: $\{x. \forall y. P x y\} = (\bigcap y. \{x. P x y\})$
by *blast*

lemma *INT-lower*: $a \in A \Longrightarrow (\bigcap x \in A. B x) \subseteq B a$
by (*fact INF-lower*)

lemma *INT-greatest*: $(\bigwedge x. x \in A \Longrightarrow C \subseteq B x) \Longrightarrow C \subseteq (\bigcap x \in A. B x)$
by (*fact INF-greatest*)

lemma *INT-empty*: $(\bigcap x \in \{\}. B x) = \text{UNIV}$
by (*fact INF-empty*)

lemma *INT-absorb*: $k \in I \Longrightarrow A k \cap (\bigcap i \in I. A i) = (\bigcap i \in I. A i)$
by (*fact INF-absorb*)

lemma *INT-subset-iff*: $B \subseteq (\bigcap i \in I. A i) \longleftrightarrow (\forall i \in I. B \subseteq A i)$
by (*fact le-INF-iff*)

lemma *INT-insert* [*simp*]: $(\bigcap x \in \text{insert } a \text{ } A. B x) = B a \cap \bigcap (B ' A)$

by (fact *INF-insert*)

lemma *INT-Un*: $(\bigcap i \in A \cup B. M i) = (\bigcap i \in A. M i) \cap (\bigcap i \in B. M i)$
by (fact *INF-union*)

lemma *INT-insert-distrib*: $u \in A \implies (\bigcap x \in A. \text{insert } a (B x)) = \text{insert } a (\bigcap x \in A. B x)$
by *blast*

lemma *INT-constant* [*simp*]: $(\bigcap y \in A. c) = (\text{if } A = \{\} \text{ then } UNIV \text{ else } c)$
by (fact *INF-constant*)

lemma *INTER-UNIV-conv*:
 $(UNIV = (\bigcap x \in A. B x)) = (\forall x \in A. B x = UNIV)$
 $((\bigcap x \in A. B x) = UNIV) = (\forall x \in A. B x = UNIV)$
by (fact *INF-top-conv*)⁺

lemma *INT-bool-eq*: $(\bigcap b. A b) = A \text{ True} \cap A \text{ False}$
by (fact *INF-UNIV-bool-expand*)

lemma *INT-anti-mono*: $A \subseteq B \implies (\bigwedge x. x \in A \implies f x \subseteq g x) \implies (\bigcap x \in B. f x) \subseteq (\bigcap x \in A. g x)$
— The last inclusion is POSITIVE!
by (fact *INF-superset-mono*)

lemma *Pow-INT-eq*: $\text{Pow } (\bigcap x \in A. B x) = (\bigcap x \in A. \text{Pow } (B x))$
by *blast*

lemma *image-INT*: $f - ' (\bigcap x \in A. B x) = (\bigcap x \in A. f - ' B x)$
by *blast*

11.6.3 Union

abbreviation *Union* :: 'a set set \Rightarrow 'a set (\bigcup)
where $\bigcup S \equiv \bigsqcup S$

lemma *Union-eq*: $\bigcup A = \{x. \exists B \in A. x \in B\}$

proof (rule *set-eqI*)

fix x

have $(\exists Q \in \{P. \exists B \in A. P \longleftrightarrow x \in B\}. Q) \longleftrightarrow (\exists B \in A. x \in B)$

by *auto*

then show $x \in \bigcup A \longleftrightarrow x \in \{x. \exists B \in A. x \in B\}$

by (*simp add: Sup-set-def image-def*)

qed

lemma *Union-iff* [*simp*]: $A \in \bigcup C \longleftrightarrow (\exists X \in C. A \in X)$
by (*unfold Union-eq blast*)

lemma *UnionI* [*intro*]: $X \in C \implies A \in X \implies A \in \bigcup C$

— The order of the premises presupposes that C is rigid; A may be flexible.
by *auto*

lemma *UnionE* [*elim!*]: $A \in \bigcup C \implies (\bigwedge X. A \in X \implies X \in C \implies R) \implies R$
by *auto*

lemma *Union-upper*: $B \in A \implies B \subseteq \bigcup A$
by (*fact Sup-upper*)

lemma *Union-least*: $(\bigwedge X. X \in A \implies X \subseteq C) \implies \bigcup A \subseteq C$
by (*fact Sup-least*)

lemma *Union-empty*: $\bigcup \{\} = \{\}$
by (*fact Sup-empty*)

lemma *Union-UNIV*: $\bigcup UNIV = UNIV$
by (*fact Sup-UNIV*)

lemma *Union-insert*: $\bigcup (\text{insert } a \ B) = a \cup \bigcup B$
by (*fact Sup-insert*)

lemma *Union-Un-distrib* [*simp*]: $\bigcup (A \cup B) = \bigcup A \cup \bigcup B$
by (*fact Sup-union-distrib*)

lemma *Union-Int-subset*: $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$
by (*fact Sup-inter-less-eq*)

lemma *Union-empty-conv*: $(\bigcup A = \{\}) \longleftrightarrow (\forall x \in A. x = \{\})$
by (*fact Sup-bot-conv*)

lemma *empty-Union-conv*: $(\{\} = \bigcup A) \longleftrightarrow (\forall x \in A. x = \{\})$
by (*fact Sup-bot-conv*)

lemma *subset-Pow-Union*: $A \subseteq \text{Pow } (\bigcup A)$
by *blast*

lemma *Union-Pow-eq* [*simp*]: $\bigcup (\text{Pow } A) = A$
by *blast*

lemma *Union-mono*: $A \subseteq B \implies \bigcup A \subseteq \bigcup B$
by (*fact Sup-subset-mono*)

lemma *Union-subsetI*: $(\bigwedge x. x \in A \implies \exists y. y \in B \wedge x \subseteq y) \implies \bigcup A \subseteq \bigcup B$
by *blast*

lemma *disjnt-inj-on-iff*:
 $\llbracket \text{inj-on } f \ (\bigcup \mathcal{A}); X \in \mathcal{A}; Y \in \mathcal{A} \rrbracket \implies \text{disjnt } (f \text{ ‘ } X) \ (f \text{ ‘ } Y) \longleftrightarrow \text{disjnt } X \ Y$
unfolding *disjnt-def*
by *safe* (use *inj-on-eq-iff* **in** $\langle \text{fastforce+} \rangle$)

lemma *disjnt-Union1* [*simp*]: $\text{disjnt } (\bigcup \mathcal{A}) B \longleftrightarrow (\forall A \in \mathcal{A}. \text{disjnt } A B)$
by (*auto simp: disjnt-def*)

lemma *disjnt-Union2* [*simp*]: $\text{disjnt } B (\bigcup \mathcal{A}) \longleftrightarrow (\forall A \in \mathcal{A}. \text{disjnt } B A)$
by (*auto simp: disjnt-def*)

11.6.4 Unions of families

syntax (*ASCII*)

-*UNION1* :: $\text{pttrns} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ (($\exists \text{UN} \text{-./ -}$) [$0, 10$] 10)
 -*UNION* :: $\text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ (($\exists \text{UN} \text{-:./ -}$) [$0, 0, 10$] 10)

syntax

-*UNION1* :: $\text{pttrns} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ (($\exists \bigcup \text{-./ -}$) [$0, 10$] 10)
 -*UNION* :: $\text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ (($\exists \bigcup \text{-\in./ -}$) [$0, 0, 10$] 10)

syntax (*latex output*)

-*UNION1* :: $\text{pttrns} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ (($\exists \bigcup (\langle \text{unbreakable} \rangle \text{-./ -})$) [$0, 10$] 10)
 -*UNION* :: $\text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ (($\exists \bigcup (\langle \text{unbreakable} \rangle \text{-\in./ -})$) [$0, 0, 10$] 10)

translations

$\bigcup x y. f \Rightarrow \bigcup x. \bigcup y. f$
 $\bigcup x. f \Rightarrow \bigcup (\text{CONST range } (\lambda x. f))$
 $\bigcup x \in A. f \Rightarrow \text{CONST Union } ((\lambda x. f) ' A)$

Note the difference between ordinary syntax of indexed unions and intersections (e.g. $\bigcup_{a_1 \in A_1}. B$) and their \LaTeX rendition: $\bigcup_{a_1 \in A_1} B$.

lemma *disjoint-UN-iff*: $\text{disjnt } A (\bigcup i \in I. B i) \longleftrightarrow (\forall i \in I. \text{disjnt } A (B i))$
by (*auto simp: disjnt-def*)

lemma *UNION-eq*: $(\bigcup x \in A. B x) = \{y. \exists x \in A. y \in B x\}$
by (*auto intro!: SUP-eqI*)

lemma *bind-UNION* [*code*]: $\text{Set.bind } A f = \bigcup (f ' A)$
by (*simp add: bind-def UNION-eq*)

lemma *member-bind* [*simp*]: $x \in \text{Set.bind } A f \longleftrightarrow x \in \bigcup (f ' A)$
by (*simp add: bind-UNION*)

lemma *Union-SetCompr-eq*: $\bigcup \{f x \mid x. P x\} = \{a. \exists x. P x \wedge a \in f x\}$
by *blast*

lemma *UN-iff* [*simp*]: $b \in (\bigcup x \in A. B x) \longleftrightarrow (\exists x \in A. b \in B x)$
using *Union-iff* [*of - B ' A*] **by** *simp*

lemma *UN-I* [*intro*]: $a \in A \Longrightarrow b \in B a \Longrightarrow b \in (\bigcup x \in A. B x)$

— The order of the premises presupposes that A is rigid; b may be flexible.
by *auto*

lemma *UN-E* [*elim!*]: $b \in (\bigcup x \in A. B x) \implies (\bigwedge x. x \in A \implies b \in B x \implies R) \implies R$

by *auto*

lemma *UN-upper*: $a \in A \implies B a \subseteq (\bigcup x \in A. B x)$
by (*fact SUP-upper*)

lemma *UN-least*: $(\bigwedge x. x \in A \implies B x \subseteq C) \implies (\bigcup x \in A. B x) \subseteq C$
by (*fact SUP-least*)

lemma *Collect-bex-eq*: $\{x. \exists y \in A. P x y\} = (\bigcup y \in A. \{x. P x y\})$
by *blast*

lemma *UN-insert-distrib*: $u \in A \implies (\bigcup x \in A. \text{insert } a (B x)) = \text{insert } a (\bigcup x \in A. B x)$
by *blast*

lemma *UN-empty*: $(\bigcup x \in \{\}. B x) = \{\}$
by (*fact SUP-empty*)

lemma *UN-empty2*: $(\bigcup x \in A. \{\}) = \{\}$
by (*fact SUP-bot*)

lemma *UN-absorb*: $k \in I \implies A k \cup (\bigcup i \in I. A i) = (\bigcup i \in I. A i)$
by (*fact SUP-absorb*)

lemma *UN-insert* [*simp*]: $(\bigcup x \in \text{insert } a A. B x) = B a \cup \bigcup (B \text{ ` } A)$
by (*fact SUP-insert*)

lemma *UN-Un* [*simp*]: $(\bigcup i \in A \cup B. M i) = (\bigcup i \in A. M i) \cup (\bigcup i \in B. M i)$
by (*fact SUP-union*)

lemma *UN-UN-flatten*: $(\bigcup x \in (\bigcup y \in A. B y). C x) = (\bigcup y \in A. \bigcup x \in B y. C x)$
by *blast*

lemma *UN-subset-iff*: $((\bigcup i \in I. A i) \subseteq B) = (\forall i \in I. A i \subseteq B)$
by (*fact SUP-le-iff*)

lemma *UN-constant* [*simp*]: $(\bigcup y \in A. c) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } c)$
by (*fact SUP-constant*)

lemma *UNION-singleton-eq-range*: $(\bigcup x \in A. \{f x\}) = f \text{ ` } A$
by *blast*

lemma *image-Union*: $f \text{ ` } \bigcup S = (\bigcup x \in S. f \text{ ` } x)$
by *blast*

lemma *UNION-empty-conv*:

$$\begin{aligned} \{\} &= (\bigcup_{x \in A}. B\ x) \longleftrightarrow (\forall x \in A. B\ x = \{\}) \\ (\bigcup_{x \in A}. B\ x) = \{\} &\longleftrightarrow (\forall x \in A. B\ x = \{\}) \\ \text{by } &(\text{fact SUP-bot-conv})+ \end{aligned}$$

lemma *Collect-ex-eq*: $\{x. \exists y. P\ x\ y\} = (\bigcup y. \{x. P\ x\ y\})$
by *blast*

lemma *ball-UN*: $(\forall z \in \bigcup (B\ ' A). P\ z) \longleftrightarrow (\forall x \in A. \forall z \in B\ x. P\ z)$
by *blast*

lemma *beX-UN*: $(\exists z \in \bigcup (B\ ' A). P\ z) \longleftrightarrow (\exists x \in A. \exists z \in B\ x. P\ z)$
by *blast*

lemma *Un-eq-UN*: $A \cup B = (\bigcup b. \text{if } b \text{ then } A \text{ else } B)$
by *safe (auto simp add: if-split-mem2)*

lemma *UN-bool-eq*: $(\bigcup b. A\ b) = (A\ \text{True} \cup A\ \text{False})$
by *(fact SUP-UNIV-bool-expand)*

lemma *UN-Pow-subset*: $(\bigcup_{x \in A}. \text{Pow}\ (B\ x)) \subseteq \text{Pow}\ (\bigcup_{x \in A}. B\ x)$
by *blast*

lemma *UN-mono*:
 $A \subseteq B \implies (\bigwedge x. x \in A \implies f\ x \subseteq g\ x) \implies$
 $(\bigcup_{x \in A}. f\ x) \subseteq (\bigcup_{x \in B}. g\ x)$
by *(fact SUP-subset-mono)*

lemma *vimage-Union*: $f\ -' (\bigcup A) = (\bigcup X \in A. f\ -' X)$
by *blast*

lemma *vimage-UN*: $f\ -' (\bigcup_{x \in A}. B\ x) = (\bigcup_{x \in A}. f\ -' B\ x)$
by *blast*

lemma *vimage-eq-UN*: $f\ -' B = (\bigcup_{y \in B}. f\ -' \{y\})$
— NOT suitable for rewriting
by *blast*

lemma *image-UN*: $f\ ' \bigcup (B\ ' A) = (\bigcup_{x \in A}. f\ ' B\ x)$
by *blast*

lemma *UN-singleton [simp]*: $(\bigcup_{x \in A}. \{x\}) = A$
by *blast*

lemma *inj-on-image*: $\text{inj-on } f\ (\bigcup A) \implies \text{inj-on } ((\cdot) f)\ A$
unfolding *inj-on-def* by *blast*

11.6.5 Distributive laws

lemma *Int-Union*: $A \cap \bigcup B = \bigcup_{C \in B} (A \cap C)$
 by *blast*

lemma *Un-Inter*: $A \cup \bigcap B = \bigcap_{C \in B} (A \cup C)$
 by *blast*

lemma *Int-Union2*: $\bigcup B \cap A = \bigcup_{C \in B} (C \cap A)$
 by *blast*

lemma *INT-Int-distrib*: $(\bigcap_{i \in I} (A \ i \cap B \ i)) = (\bigcap_{i \in I} (A \ i)) \cap (\bigcap_{i \in I} (B \ i))$
 by *(rule sym) (rule INF-inf-distrib)*

lemma *UN-Un-distrib*: $(\bigcup_{i \in I} (A \ i \cup B \ i)) = (\bigcup_{i \in I} (A \ i)) \cup (\bigcup_{i \in I} (B \ i))$
 by *(rule sym) (rule SUP-sup-distrib)*

lemma *Int-Inter-image*: $(\bigcap_{x \in C} (A \ x \cap B \ x)) = \bigcap (A \ ' C) \cap \bigcap (B \ ' C)$
 by *(simp add: INT-Int-distrib)*

lemma *Int-Inter-eq*: $A \cap \bigcap \mathcal{B} = (\text{if } \mathcal{B} = \{\} \text{ then } A \text{ else } (\bigcap_{B \in \mathcal{B}} (A \cap B)))$
 $\bigcap \mathcal{B} \cap A = (\text{if } \mathcal{B} = \{\} \text{ then } A \text{ else } (\bigcap_{B \in \mathcal{B}} (B \cap A)))$
 by *auto*

lemma *Un-Union-image*: $(\bigcup_{x \in C} (A \ x \cup B \ x)) = \bigcup (A \ ' C) \cup \bigcup (B \ ' C)$
 — Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
 — Union of a family of unions
 by *(simp add: UN-Un-distrib)*

lemma *Un-INT-distrib*: $B \cup (\bigcap_{i \in I} (A \ i)) = (\bigcap_{i \in I} (B \cup A \ i))$
 by *blast*

lemma *Int-UN-distrib*: $B \cap (\bigcup_{i \in I} (A \ i)) = (\bigcup_{i \in I} (B \cap A \ i))$
 — Halmos, Naive Set Theory, page 35.
 by *blast*

lemma *Int-UN-distrib2*: $(\bigcup_{i \in I} (A \ i)) \cap (\bigcup_{j \in J} (B \ j)) = (\bigcup_{i \in I} \bigcup_{j \in J} (A \ i \cap B \ j))$
 by *blast*

lemma *Un-INT-distrib2*: $(\bigcap_{i \in I} (A \ i)) \cup (\bigcap_{j \in J} (B \ j)) = (\bigcap_{i \in I} \bigcap_{j \in J} (A \ i \cup B \ j))$
 by *blast*

lemma *Union-disjoint*: $(\bigcup C \cap A = \{\}) \longleftrightarrow (\forall B \in C. B \cap A = \{\})$
 by *blast*

lemma *SUP-UNION*: $(\bigsqcup_{x \in (\bigcup y \in A. g \ y)}. f \ x) = (\bigsqcup_{y \in A} \bigsqcup_{x \in g \ y} f \ x :: - :: \text{complete-lattice})$
 by *(rule order-antisym) (blast intro: SUP-least SUP-upper2)+*

11.7 Injections and bijections

lemma *inj-on-Inter*: $S \neq \{\}$ $\implies (\bigwedge A. A \in S \implies \text{inj-on } f \ A) \implies \text{inj-on } f \ (\bigcap S)$
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-INTER*: $I \neq \{\}$ $\implies (\bigwedge i. i \in I \implies \text{inj-on } f \ (A \ i)) \implies \text{inj-on } f \ (\bigcap_{i \in I. A \ i})$
unfolding *inj-on-def* **by** *safe simp*

lemma *inj-on-UNION-chain*:

assumes *chain*: $\bigwedge i \ j. i \in I \implies j \in I \implies A \ i \leq A \ j \vee A \ j \leq A \ i$
and *inj*: $\bigwedge i. i \in I \implies \text{inj-on } f \ (A \ i)$
shows $\text{inj-on } f \ (\bigcup_{i \in I. A \ i})$

proof –

have $x = y$
if *: $i \in I \ j \in I$
and **: $x \in A \ i \ y \in A \ j$
and ***: $f \ x = f \ y$
for $i \ j \ x \ y$
using *chain* [*OF* *]

proof

assume $A \ i \leq A \ j$
with ** **have** $x \in A \ j$ **by** *auto*
with *inj* * ** *** **show** *?thesis*
by (*auto simp add: inj-on-def*)

next

assume $A \ j \leq A \ i$
with ** **have** $y \in A \ i$ **by** *auto*
with *inj* * ** *** **show** *?thesis*
by (*auto simp add: inj-on-def*)

qed

then show *?thesis*

by (*unfold inj-on-def UNION-eq*) *auto*

qed

lemma *bij-betw-UNION-chain*:

assumes *chain*: $\bigwedge i \ j. i \in I \implies j \in I \implies A \ i \leq A \ j \vee A \ j \leq A \ i$
and *bij*: $\bigwedge i. i \in I \implies \text{bij-betw } f \ (A \ i) \ (A' \ i)$
shows $\text{bij-betw } f \ (\bigcup_{i \in I. A \ i}) \ (\bigcup_{i \in I. A' \ i})$
unfolding *bij-betw-def*

proof *safe*

have $\bigwedge i. i \in I \implies \text{inj-on } f \ (A \ i)$
using *bij* *bij-betw-def*[*of f*] **by** *auto*
then show $\text{inj-on } f \ (\bigcup (A \ ' \ I))$
using *chain inj-on-UNION-chain*[*of I A f*] **by** *auto*

next

fix $i \ x$
assume *: $i \in I \ x \in A \ i$
with *bij* **have** $f \ x \in A' \ i$
by (*auto simp: bij-betw-def*)

with * show $f x \in \bigcup (A' \text{ ' } I)$ **by** *blast*
next
fix $i x'$
assume *: $i \in I \ x' \in A' \ i$
with *bij* **have** $\exists x \in A \ i. \ x' = f x$
unfolding *bij-betw-def* **by** *blast*
with * have $\exists j \in I. \ \exists x \in A \ j. \ x' = f x$
by *blast*
then show $x' \in f \text{ ' } \bigcup (A' \text{ ' } I)$
by *blast*
qed

lemma *image-INT*: $\text{inj-on } f \ C \Longrightarrow \forall x \in A. \ B \ x \subseteq C \Longrightarrow j \in A \Longrightarrow f \text{ ' } (\bigcap (B \text{ ' } A))$
 $= (\bigcap x \in A. \ f \text{ ' } B \ x)$
by (*auto simp add: inj-on-def*) *blast*

lemma *bij-image-INT*: $\text{bij } f \Longrightarrow f \text{ ' } (\bigcap (B \text{ ' } A)) = (\bigcap x \in A. \ f \text{ ' } B \ x)$
by (*auto simp: bij-def inj-def surj-def*) *blast*

lemma *UNION-fun-upd*: $\bigcup (A(i := B) \text{ ' } J) = \bigcup (A \text{ ' } (J - \{i\})) \cup (\text{if } i \in J \text{ then } B \text{ else } \{\})$
by (*auto simp add: set-eq-iff*)

lemma *bij-betw-Pow*:
assumes *bij-betw* $f \ A \ B$
shows *bij-betw* $(\text{image } f) \ (\text{Pow } A) \ (\text{Pow } B)$
proof –
from *assms* **have** *inj-on* $f \ A$
by (*rule bij-betw-imp-inj-on*)
then have *inj-on* $f \ (\bigcup (\text{Pow } A))$
by *simp*
then have *inj-on* $(\text{image } f) \ (\text{Pow } A)$
by (*rule inj-on-image*)
then have *bij-betw* $(\text{image } f) \ (\text{Pow } A) \ (\text{image } f \text{ ' } \text{Pow } A)$
by (*rule inj-on-imp-bij-betw*)
moreover from *assms* **have** $f \text{ ' } A = B$
by (*rule bij-betw-imp-surj-on*)
then have $\text{image } f \text{ ' } \text{Pow } A = \text{Pow } B$
by (*rule image-Pow-surj*)
ultimately show *?thesis* **by** *simp*
qed

11.7.1 Complement

lemma *Compl-INT* [*simp*]: $\neg (\bigcap x \in A. \ B \ x) = (\bigcup x \in A. \ \neg B \ x)$
by *blast*

lemma *Compl-UN* [*simp*]: $\neg (\bigcup x \in A. \ B \ x) = (\bigcap x \in A. \ \neg B \ x)$

by *blast*

11.7.2 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps [simp]*:

$$\begin{aligned} \bigwedge a B C. (\bigcup x \in C. \text{insert } a (B x)) &= (\text{if } C = \{\} \text{ then } \{\} \text{ else insert } a (\bigcup x \in C. B x)) \\ \bigwedge A B C. (\bigcup x \in C. A x \cup B) &= ((\text{if } C = \{\} \text{ then } \{\} \text{ else } (\bigcup x \in C. A x) \cup B) \\ \bigwedge A B C. (\bigcup x \in C. A \cup B x) &= ((\text{if } C = \{\} \text{ then } \{\} \text{ else } A \cup (\bigcup x \in C. B x)) \\ \bigwedge A B C. (\bigcup x \in C. A x \cap B) &= ((\bigcup x \in C. A x) \cap B) \\ \bigwedge A B C. (\bigcup x \in C. A \cap B x) &= (A \cap (\bigcup x \in C. B x)) \\ \bigwedge A B C. (\bigcup x \in C. A x - B) &= ((\bigcup x \in C. A x) - B) \\ \bigwedge A B C. (\bigcup x \in C. A - B x) &= (A - (\bigcap x \in C. B x)) \\ \bigwedge A B. (\bigcup x \in \bigcup A. B x) &= (\bigcup y \in A. \bigcup x \in y. B x) \\ \bigwedge A B C. (\bigcup z \in (\bigcup (B ' A)). C z) &= (\bigcup x \in A. \bigcup z \in B x. C z) \\ \bigwedge A B f. (\bigcup x \in f' A. B x) &= (\bigcup a \in A. B (f a)) \end{aligned}$$

by *auto*

lemma *INT-simps [simp]*:

$$\begin{aligned} \bigwedge A B C. (\bigcap x \in C. A x \cap B) &= (\text{if } C = \{\} \text{ then UNIV else } (\bigcap x \in C. A x) \cap B) \\ \bigwedge A B C. (\bigcap x \in C. A \cap B x) &= (\text{if } C = \{\} \text{ then UNIV else } A \cap (\bigcap x \in C. B x)) \\ \bigwedge A B C. (\bigcap x \in C. A x - B) &= (\text{if } C = \{\} \text{ then UNIV else } (\bigcap x \in C. A x) - B) \\ \bigwedge A B C. (\bigcap x \in C. A - B x) &= (\text{if } C = \{\} \text{ then UNIV else } A - (\bigcup x \in C. B x)) \\ \bigwedge a B C. (\bigcap x \in C. \text{insert } a (B x)) &= \text{insert } a (\bigcap x \in C. B x) \\ \bigwedge A B C. (\bigcap x \in C. A x \cup B) &= ((\bigcap x \in C. A x) \cup B) \\ \bigwedge A B C. (\bigcap x \in C. A \cup B x) &= (A \cup (\bigcap x \in C. B x)) \\ \bigwedge A B. (\bigcap x \in \bigcup A. B x) &= (\bigcap y \in A. \bigcap x \in y. B x) \\ \bigwedge A B C. (\bigcap z \in (\bigcup (B ' A)). C z) &= (\bigcap x \in A. \bigcap z \in B x. C z) \\ \bigwedge A B f. (\bigcap x \in f' A. B x) &= (\bigcap a \in A. B (f a)) \end{aligned}$$

by *auto*

lemma *UN-ball-bex-simps [simp]*:

$$\begin{aligned} \bigwedge A P. (\forall x \in \bigcup A. P x) &\longleftrightarrow (\forall y \in A. \forall x \in y. P x) \\ \bigwedge A B P. (\forall x \in (\bigcup (B ' A)). P x) &= (\forall a \in A. \forall x \in B a. P x) \\ \bigwedge A P. (\exists x \in \bigcup A. P x) &\longleftrightarrow (\exists y \in A. \exists x \in y. P x) \\ \bigwedge A B P. (\exists x \in (\bigcup (B ' A)). P x) &\longleftrightarrow (\exists a \in A. \exists x \in B a. P x) \end{aligned}$$

by *auto*

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:

$$\begin{aligned} \bigwedge a B C. \text{insert } a (\bigcup x \in C. B x) &= (\text{if } C = \{\} \text{ then } \{a\} \text{ else } (\bigcup x \in C. \text{insert } a (B x))) \\ \bigwedge A B C. (\bigcup x \in C. A x) \cup B &= (\text{if } C = \{\} \text{ then } B \text{ else } (\bigcup x \in C. A x \cup B)) \\ \bigwedge A B C. A \cup (\bigcup x \in C. B x) &= (\text{if } C = \{\} \text{ then } A \text{ else } (\bigcup x \in C. A \cup B x)) \\ \bigwedge A B C. ((\bigcup x \in C. A x) \cap B) &= (\bigcup x \in C. A x \cap B) \\ \bigwedge A B C. (A \cap (\bigcup x \in C. B x)) &= (\bigcup x \in C. A \cap B x) \\ \bigwedge A B C. ((\bigcup x \in C. A x) - B) &= (\bigcup x \in C. A x - B) \end{aligned}$$

$$\begin{aligned}
\bigwedge A B C. (A - (\bigcap x \in C. B x)) &= (\bigcup x \in C. A - B x) \\
\bigwedge A B. (\bigcup y \in A. \bigcup x \in y. B x) &= (\bigcup x \in \bigcup A. B x) \\
\bigwedge A B C. (\bigcup x \in A. \bigcup z \in B x. C z) &= (\bigcup z \in (\bigcup (B \text{ ' } A)). C z) \\
\bigwedge A B f. (\bigcup a \in A. B (f a)) &= (\bigcup x \in f \text{ ' } A. B x)
\end{aligned}$$

by *auto*

lemma *INT-extend-simps*:

$$\begin{aligned}
\bigwedge A B C. (\bigcap x \in C. A x) \cap B &= (\text{if } C = \{\} \text{ then } B \text{ else } (\bigcap x \in C. A x \cap B)) \\
\bigwedge A B C. A \cap (\bigcap x \in C. B x) &= (\text{if } C = \{\} \text{ then } A \text{ else } (\bigcap x \in C. A \cap B x)) \\
\bigwedge A B C. (\bigcap x \in C. A x) - B &= (\text{if } C = \{\} \text{ then } \text{UNIV} - B \text{ else } (\bigcap x \in C. A x - B)) \\
\bigwedge A B C. A - (\bigcup x \in C. B x) &= (\text{if } C = \{\} \text{ then } A \text{ else } (\bigcap x \in C. A - B x)) \\
\bigwedge a B C. \text{insert } a (\bigcap x \in C. B x) &= (\bigcap x \in C. \text{insert } a (B x)) \\
\bigwedge A B C. ((\bigcap x \in C. A x) \cup B) &= (\bigcap x \in C. A x \cup B) \\
\bigwedge A B C. A \cup (\bigcap x \in C. B x) &= (\bigcap x \in C. A \cup B x) \\
\bigwedge A B. (\bigcap y \in A. \bigcap x \in y. B x) &= (\bigcap x \in \bigcup A. B x) \\
\bigwedge A B C. (\bigcap x \in A. \bigcap z \in B x. C z) &= (\bigcap z \in (\bigcup (B \text{ ' } A)). C z) \\
\bigwedge A B f. (\bigcap a \in A. B (f a)) &= (\bigcap x \in f \text{ ' } A. B x)
\end{aligned}$$

by *auto*

Finally

lemmas *mem-simps* =

insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [*simp*] above.

end

12 Wrapping Existing Freely Generated Type’s Constructors

theory *Ctr-Sugar*

imports *HOL*

keywords

print-case-translations :: *diag* **and**

free-constructors :: *thy-goal*

begin

consts

case-guard :: *bool* \Rightarrow *'a* \Rightarrow (*'a* \Rightarrow *'b*) \Rightarrow *'b*

case-nil :: *'a* \Rightarrow *'b*

case-cons :: (*'a* \Rightarrow *'b*) \Rightarrow (*'a* \Rightarrow *'b*) \Rightarrow *'a* \Rightarrow *'b*

case-elem :: *'a* \Rightarrow *'b* \Rightarrow *'a* \Rightarrow *'b*

case-abs :: (*'c* \Rightarrow *'b*) \Rightarrow *'b*

declare [[*coercion-args case-guard* - + -]]

declare [[*coercion-args case-cons* - -]]

declare [[*coercion-args case-abs* -]]

declare $[[\text{coercion-args case-elem - +}]]$

ML-file $\langle \text{Tools/Ctr-Sugar/case-translation.ML} \rangle$

lemma *iffI-np*: $[[x \implies \neg y; \neg x \implies y]] \implies \neg x \longleftrightarrow y$
by (*erule iffI*) (*erule contrapos-pn*)

lemma *iff-contradict*:

$\neg P \implies P \longleftrightarrow Q \implies Q \implies R$

$\neg Q \implies P \longleftrightarrow Q \implies P \implies R$

by *blast+*

ML-file $\langle \text{Tools/Ctr-Sugar/ctr-sugar-util.ML} \rangle$

ML-file $\langle \text{Tools/Ctr-Sugar/ctr-sugar-tactics.ML} \rangle$

ML-file $\langle \text{Tools/Ctr-Sugar/ctr-sugar-code.ML} \rangle$

ML-file $\langle \text{Tools/Ctr-Sugar/ctr-sugar.ML} \rangle$

Coinduction method that avoids some boilerplate compared with coinduct.

ML-file $\langle \text{Tools/coinduction.ML} \rangle$

end

13 Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*

imports *Complete-Lattices Ctr-Sugar*

keywords

inductive coinductive inductive-cases inductive-simps :: thy-defn and

monos and

print-inductives :: diag and

old-rep-datatype :: thy-goal and

primrec :: thy-defn

begin

13.1 Least fixed points

context *complete-lattice*

begin

definition *lfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$

where $\text{lfp } f = \text{Inf } \{u. f u \leq u\}$

lemma *lfp-lowerbound*: $f A \leq A \implies \text{lfp } f \leq A$

unfolding *lfp-def* **by** (*rule Inf-lower*) *simp*

lemma *lfp-greatest*: $(\bigwedge u. f u \leq u \implies A \leq u) \implies A \leq \text{lfp } f$

unfolding *lfp-def* **by** (*rule Inf-greatest*) *simp*

end

lemma *lfp-fixpoint*:

assumes *mono f*

shows $f (lfp f) = lfp f$

unfolding *lfp-def*

proof (*rule order-antisym*)

let $?H = \{u. f u \leq u\}$

let $?a = \bigcap ?H$

show $f ?a \leq ?a$

proof (*rule Inf-greatest*)

fix x

assume $x \in ?H$

then have $?a \leq x$ by (*rule Inf-lower*)

with $\langle mono f \rangle$ have $f ?a \leq f x$..

also from $\langle x \in ?H \rangle$ have $f x \leq x$..

finally show $f ?a \leq x$.

qed

show $?a \leq f ?a$

proof (*rule Inf-lower*)

from $\langle mono f \rangle$ and $\langle f ?a \leq ?a \rangle$ have $f (f ?a) \leq f ?a$..

then show $f ?a \in ?H$..

qed

qed

lemma *lfp-unfold*: $mono f \implies lfp f = f (lfp f)$

by (*rule lfp-fixpoint [symmetric]*)

lemma *lfp-const*: $lfp (\lambda x. t) = t$

by (*rule lfp-unfold*) (*simp add: mono-def*)

lemma *lfp-eqI*: $mono F \implies F x = x \implies (\bigwedge z. F z = z \implies x \leq z) \implies lfp F = x$

by (*rule antisym*) (*simp-all add: lfp-lowerbound lfp-unfold[symmetric]*)

13.2 General induction rules for least fixed points

lemma *lfp-ordinal-induct* [*case-names mono step union*]:

fixes $f :: 'a::complete-lattice \Rightarrow 'a$

assumes *mono*: $mono f$

and *P-f*: $\bigwedge S. P S \implies S \leq lfp f \implies P (f S)$

and *P-Union*: $\bigwedge M. \forall S \in M. P S \implies P (Sup M)$

shows $P (lfp f)$

proof –

let $?M = \{S. S \leq lfp f \wedge P S\}$

from *P-Union* have $P (Sup ?M)$ by *simp*

also have $Sup ?M = lfp f$

proof (*rule antisym*)

show $Sup ?M \leq lfp f$

```

    by (blast intro: Sup-least)
  then have  $f (Sup ?M) \leq f (lfp f)$ 
    by (rule mono [THEN monoD])
  then have  $f (Sup ?M) \leq lfp f$ 
    using mono [THEN lfp-unfold] by simp
  then have  $f (Sup ?M) \in ?M$ 
    using P-Union by simp (intro P-f Sup-least, auto)
  then have  $f (Sup ?M) \leq Sup ?M$ 
    by (rule Sup-upper)
  then show  $lfp f \leq Sup ?M$ 
    by (rule lfp-lowerbound)
qed
finally show ?thesis .
qed

```

```

theorem lfp-induct:
  assumes mono: mono f
    and ind:  $f (inf (lfp f) P) \leq P$ 
  shows  $lfp f \leq P$ 
proof (induct rule: lfp-ordinal-induct)
  case mono
  show ?case by fact
next
  case (step S)
  then show ?case
    by (intro order-trans[OF - ind] monoD[OF mono]) auto
next
  case (union M)
  then show ?case
    by (auto intro: Sup-least)
qed

```

```

lemma lfp-induct-set:
  assumes lfp:  $a \in lfp f$ 
    and mono: mono f
    and hyp:  $\bigwedge x. x \in f (lfp f \cap \{x. P x\}) \implies P x$ 
  shows  $P a$ 
  by (rule lfp-induct [THEN subsetD, THEN CollectD, OF mono - lfp]) (auto intro: hyp)

```

```

lemma lfp-ordinal-induct-set:
  assumes mono: mono f
    and P-f:  $\bigwedge S. P S \implies P (f S)$ 
    and P-Union:  $\bigwedge M. \forall S \in M. P S \implies P (\bigcup M)$ 
  shows  $P (lfp f)$ 
  using assms by (rule lfp-ordinal-induct)

```

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding.

```

lemma def-lfp-unfold:  $h \equiv lfp f \implies mono f \implies h = f h$ 

```

by (auto intro!: lfp-unfold)

lemma *def-lfp-induct*: $A \equiv \text{lfp } f \implies \text{mono } f \implies f (\text{inf } A P) \leq P \implies A \leq P$
 by (blast intro: lfp-induct)

lemma *def-lfp-induct-set*:

$A \equiv \text{lfp } f \implies \text{mono } f \implies a \in A \implies (\bigwedge x. x \in f (A \cap \{x. P x\}) \implies P x) \implies P a$

by (blast intro: lfp-induct-set)

Monotonicity of *lfp*!

lemma *lfp-mono*: $(\bigwedge Z. f Z \leq g Z) \implies \text{lfp } f \leq \text{lfp } g$

by (rule lfp-lowerbound [THEN lfp-greatest]) (blast intro: order-trans)

13.3 Greatest fixed points

context *complete-lattice*

begin

definition *gfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$

where $\text{gfp } f = \text{Sup } \{u. u \leq f u\}$

lemma *gfp-upperbound*: $X \leq f X \implies X \leq \text{gfp } f$

by (auto simp add: gfp-def intro: Sup-upper)

lemma *gfp-least*: $(\bigwedge u. u \leq f u \implies u \leq X) \implies \text{gfp } f \leq X$

by (auto simp add: gfp-def intro: Sup-least)

end

lemma *lfp-le-gfp*: $\text{mono } f \implies \text{lfp } f \leq \text{gfp } f$

by (rule gfp-upperbound) (simp add: lfp-fixpoint)

lemma *gfp-fixpoint*:

assumes *mono f*

shows $f (\text{gfp } f) = \text{gfp } f$

unfolding *gfp-def*

proof (rule order-antisym)

let $?H = \{u. u \leq f u\}$

let $?a = \bigsqcup ?H$

show $?a \leq f ?a$

proof (rule Sup-least)

fix x

assume $x \in ?H$

then have $x \leq f x$..

also from $\langle x \in ?H \rangle$ have $x \leq ?a$ by (rule Sup-upper)

with $\langle \text{mono } f \rangle$ have $f x \leq f ?a$..

finally show $x \leq f ?a$.

qed

```

show  $f ?a \leq ?a$ 
proof (rule Sup-upper)
  from  $\langle \text{mono } f \rangle$  and  $\langle ?a \leq f ?a \rangle$  have  $f ?a \leq f (f ?a)$  ..
  then show  $f ?a \in ?H$  ..
qed
qed

```

```

lemma gfp-unfold:  $\text{mono } f \implies \text{gfp } f = f (\text{gfp } f)$ 
  by (rule gfp-fixpoint [symmetric])

```

```

lemma gfp-const:  $\text{gfp } (\lambda x. t) = t$ 
  by (rule gfp-unfold) (simp add: mono-def)

```

```

lemma gfp-eqI:  $\text{mono } F \implies F x = x \implies (\bigwedge z. F z = z \implies z \leq x) \implies \text{gfp } F = x$ 
  by (rule antisym) (simp-all add: gfp-upperbound gfp-unfold[symmetric])

```

13.4 Coinduction rules for greatest fixed points

Weak version.

```

lemma weak-coinduct:  $a \in X \implies X \subseteq f X \implies a \in \text{gfp } f$ 
  by (rule gfp-upperbound [THEN subsetD]) auto

```

```

lemma weak-coinduct-image:  $a \in X \implies g'X \subseteq f (g'X) \implies g a \in \text{gfp } f$ 
  apply (erule gfp-upperbound [THEN subsetD])
  apply (erule imageI)
  done

```

```

lemma coinduct-lemma:  $X \leq f (\text{sup } X (\text{gfp } f)) \implies \text{mono } f \implies \text{sup } X (\text{gfp } f) \leq f (\text{sup } X (\text{gfp } f))$ 
  apply (frule gfp-unfold [THEN eq-refl])
  apply (drule mono-sup)
  apply (rule le-supI)
  apply assumption
  apply (rule order-trans)
  apply (rule order-trans)
  apply assumption
  apply (rule sup-ge2)
  apply assumption
  done

```

Strong version, thanks to Coen and Frost.

```

lemma coinduct-set:  $\text{mono } f \implies a \in X \implies X \subseteq f (X \cup \text{gfp } f) \implies a \in \text{gfp } f$ 
  by (rule weak-coinduct[rotated], rule coinduct-lemma) blast+

```

```

lemma gfp-fun-UnI2:  $\text{mono } f \implies a \in \text{gfp } f \implies a \in f (X \cup \text{gfp } f)$ 
  by (blast dest: gfp-fixpoint mono-Un)

```

```

lemma gfp-ordinal-induct[case-names mono step union]:

```

```

fixes  $f :: 'a::\text{complete-lattice} \Rightarrow 'a$ 
assumes  $\text{mono}: \text{mono } f$ 
  and  $P\text{-}f: \bigwedge S. P S \Longrightarrow \text{gfp } f \leq S \Longrightarrow P (f S)$ 
  and  $P\text{-}Union: \bigwedge M. \forall S \in M. P S \Longrightarrow P (\text{Inf } M)$ 
shows  $P (\text{gfp } f)$ 
proof –
  let  $?M = \{S. \text{gfp } f \leq S \wedge P S\}$ 
  from  $P\text{-}Union$  have  $P (\text{Inf } ?M)$  by  $\text{simp}$ 
  also have  $\text{Inf } ?M = \text{gfp } f$ 
  proof ( $\text{rule antisym}$ )
    show  $\text{gfp } f \leq \text{Inf } ?M$ 
      by ( $\text{blast intro: Inf-greatest}$ )
    then have  $f (\text{gfp } f) \leq f (\text{Inf } ?M)$ 
      by ( $\text{rule mono [THEN monoD]}$ )
    then have  $\text{gfp } f \leq f (\text{Inf } ?M)$ 
      using  $\text{mono [THEN gfp-unfold]}$  by  $\text{simp}$ 
    then have  $f (\text{Inf } ?M) \in ?M$ 
      using  $P\text{-}Union$  by  $\text{simp (intro P-f Inf-greatest, auto)}$ 
    then have  $\text{Inf } ?M \leq f (\text{Inf } ?M)$ 
      by ( $\text{rule Inf-lower}$ )
    then show  $\text{Inf } ?M \leq \text{gfp } f$ 
      by ( $\text{rule gfp-upperbound}$ )
  qed
  finally show  $?thesis$  .
qed

```

```

lemma coinduct:
  assumes  $\text{mono}: \text{mono } f$ 
    and  $\text{ind}: X \leq f (\text{sup } X (\text{gfp } f))$ 
  shows  $X \leq \text{gfp } f$ 
proof ( $\text{induct rule: gfp-ordinal-induct}$ )
  case  $\text{mono}$ 
    then show  $?case$  by  $\text{fact}$ 
next
  case ( $\text{step } S$ )
    then show  $?case$ 
      by ( $\text{intro order-trans[OF ind -] monoD[OF mono]} \text{ auto}$ )
next
  case ( $\text{union } M$ )
    then show  $?case$ 
      by ( $\text{auto intro: mono Inf-greatest}$ )
qed

```

13.5 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition $X \subseteq f X$ to one expressed using both lfp and gfp

```

lemma coinduct3-mono-lemma:  $\text{mono } f \Longrightarrow \text{mono } (\lambda x. f x \cup X \cup B)$ 
  by ( $\text{iprover intro: subset-refl monoI Un-mono monoD}$ )

```

lemma *coinduct3-lemma*:

$X \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f)) \implies \text{mono } f \implies$
 $\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f) \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f))$
apply (*rule subset-trans*)
apply (*erule coinduct3-mono-lemma [THEN lfp-unfold [THEN eq-refl]]*)
apply (*rule Un-least [THEN Un-least]*)
apply (*rule subset-refl, assumption*)
apply (*rule gfp-unfold [THEN equalityD1, THEN subset-trans], assumption*)
apply (*rule monoD, assumption*)
apply (*subst coinduct3-mono-lemma [THEN lfp-unfold], auto*)
done

lemma *coinduct3*: $\text{mono } f \implies a \in X \implies X \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup \text{gfp } f)) \implies$
 $a \in \text{gfp } f$

apply (*rule coinduct3-lemma [THEN [2] weak-coinduct]*)
apply (*rule coinduct3-mono-lemma [THEN lfp-unfold, THEN ssubst]*)
apply *simp-all*
done

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding.

lemma *def-gfp-unfold*: $A \equiv \text{gfp } f \implies \text{mono } f \implies A = f A$
by (*auto intro!: gfp-unfold*)

lemma *def-coinduct*: $A \equiv \text{gfp } f \implies \text{mono } f \implies X \leq f (\text{sup } X A) \implies X \leq A$
by (*iprover intro!: coinduct*)

lemma *def-coinduct-set*: $A \equiv \text{gfp } f \implies \text{mono } f \implies a \in X \implies X \subseteq f (X \cup A) \implies a \in A$
by (*auto intro!: coinduct-set*)

lemma *def-Collect-coinduct*:

$A \equiv \text{gfp } (\lambda w. \text{Collect } (P w)) \implies \text{mono } (\lambda w. \text{Collect } (P w)) \implies a \in X \implies$
 $(\bigwedge z. z \in X \implies P (X \cup A) z) \implies a \in A$
by (*erule def-coinduct-set*) *auto*

lemma *def-coinduct3*: $A \equiv \text{gfp } f \implies \text{mono } f \implies a \in X \implies X \subseteq f (\text{lfp } (\lambda x. f x \cup X \cup A)) \implies a \in A$
by (*auto intro!: coinduct3*)

Monotonicity of *gfp*!

lemma *gfp-mono*: $(\bigwedge Z. f Z \leq g Z) \implies \text{gfp } f \leq \text{gfp } g$
by (*rule gfp-upperbound [THEN gfp-least]*) (*blast intro: order-trans*)

13.6 Rules for fixed point calculus

lemma *lfp-rolling*:

assumes *mono g mono f*
shows $g (\text{lfp } (\lambda x. f (g x))) = \text{lfp } (\lambda x. g (f x))$
proof (*rule antisym*)

```

have *: mono ( $\lambda x. f (g x)$ )
  using assms by (auto simp: mono-def)
show  $\text{lfp } (\lambda x. g (f x)) \leq g (\text{lfp } (\lambda x. f (g x)))$ 
  by (rule lfp-lowerbound) (simp add: lfp-unfold[OF *, symmetric])
show  $g (\text{lfp } (\lambda x. f (g x))) \leq \text{lfp } (\lambda x. g (f x))$ 
proof (rule lfp-greatest)
  fix u
  assume u:  $g (f u) \leq u$ 
  then have  $g (\text{lfp } (\lambda x. f (g x))) \leq g (f u)$ 
    by (intro assms[THEN monoD] lfp-lowerbound)
  with u show  $g (\text{lfp } (\lambda x. f (g x))) \leq u$ 
    by auto
qed
qed

```

```

lemma lfp-lfp:
  assumes f:  $\bigwedge x y w z. x \leq y \implies w \leq z \implies f x w \leq f y z$ 
  shows  $\text{lfp } (\lambda x. \text{lfp } (f x)) = \text{lfp } (\lambda x. f x x)$ 
proof (rule antisym)
  have *: mono ( $\lambda x. f x x$ )
    by (blast intro: monoI f)
  show  $\text{lfp } (\lambda x. \text{lfp } (f x)) \leq \text{lfp } (\lambda x. f x x)$ 
    by (intro lfp-lowerbound) (simp add: lfp-unfold[OF *, symmetric])
  show  $\text{lfp } (\lambda x. \text{lfp } (f x)) \geq \text{lfp } (\lambda x. f x x)$  (is  $?F \geq -$ )
proof (intro lfp-lowerbound)
  have *:  $?F = \text{lfp } (f ?F)$ 
    by (rule lfp-unfold) (blast intro: monoI lfp-mono f)
  also have  $\dots = f ?F (\text{lfp } (f ?F))$ 
    by (rule lfp-unfold) (blast intro: monoI lfp-mono f)
  finally show  $f ?F ?F \leq ?F$ 
    by (simp add: *[symmetric])
qed
qed

```

```

lemma gfp-rolling:
  assumes mono g mono f
  shows  $g (\text{gfp } (\lambda x. f (g x))) = \text{gfp } (\lambda x. g (f x))$ 
proof (rule antisym)
  have *: mono ( $\lambda x. f (g x)$ )
    using assms by (auto simp: mono-def)
  show  $g (\text{gfp } (\lambda x. f (g x))) \leq \text{gfp } (\lambda x. g (f x))$ 
    by (rule gfp-upperbound) (simp add: gfp-unfold[OF *, symmetric])
  show  $\text{gfp } (\lambda x. g (f x)) \leq g (\text{gfp } (\lambda x. f (g x)))$ 
proof (rule gfp-least)
  fix u
  assume u:  $u \leq g (f u)$ 
  then have  $g (f u) \leq g (\text{gfp } (\lambda x. f (g x)))$ 
    by (intro assms[THEN monoD] gfp-upperbound)
  with u show  $u \leq g (\text{gfp } (\lambda x. f (g x)))$ 

```

```

    by auto
  qed
qed

lemma gfp-gfp:
  assumes f:  $\bigwedge x y w z. x \leq y \implies w \leq z \implies f x w \leq f y z$ 
  shows  $\text{gfp } (\lambda x. \text{gfp } (f x)) = \text{gfp } (\lambda x. f x x)$ 
proof (rule antisym)
  have *:  $\text{mono } (\lambda x. f x x)$ 
    by (blast intro: monoI f)
  show  $\text{gfp } (\lambda x. f x x) \leq \text{gfp } (\lambda x. \text{gfp } (f x))$ 
    by (intro gfp-upperbound) (simp add: gfp-unfold[OF *, symmetric])
  show  $\text{gfp } (\lambda x. \text{gfp } (f x)) \leq \text{gfp } (\lambda x. f x x)$  (is ?F  $\leq$  -)
  proof (intro gfp-upperbound)
    have *:  $?F = \text{gfp } (f ?F)$ 
      by (rule gfp-unfold) (blast intro: monoI gfp-mono f)
    also have  $\dots = f ?F (\text{gfp } (f ?F))$ 
      by (rule gfp-unfold) (blast intro: monoI gfp-mono f)
    finally show  $?F \leq f ?F ?F$ 
      by (simp add: *[symmetric])
  qed
qed

```

13.7 Inductive predicates and sets

Package setup.

```

lemmas basic-monos =
  subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  Collect-mono in-mono vimage-mono

```

```

lemma le-rel-bool-arg-iff:  $X \leq Y \iff X \text{ False} \leq Y \text{ False} \wedge X \text{ True} \leq Y \text{ True}$ 
  unfolding le-fun-def le-bool-def using bool-induct by auto

```

```

lemma imp-conj-iff:  $((P \longrightarrow Q) \wedge P) = (P \wedge Q)$ 
  by blast

```

```

lemma meta-fun-cong:  $P \equiv Q \implies P a \equiv Q a$ 
  by auto

```

ML-file $\langle \text{Tools/inductive.ML} \rangle$

```

lemmas [mono] =
  imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  imp-mono not-mono
  Ball-def Bex-def
  induct-rulify-fallback

```


13.8 The Schroeder-Bernstein Theorem

See also:

- `$ISABELLE_HOME/src/HOL/ex/Set_Theory.thy`
- <http://planetmath.org/proofofschroederbernsteintheoremingtarskiknastertheorem>
- Springer LNCS 828 (cover page)

theorem *Schroeder-Bernstein*:

fixes $f :: 'a \Rightarrow 'b$ **and** $g :: 'b \Rightarrow 'a$
and $A :: 'a$ set **and** $B :: 'b$ set
assumes $inj1: inj\text{-on } f\ A$ **and** $sub1: f\ 'A \subseteq B$
and $inj2: inj\text{-on } g\ B$ **and** $sub2: g\ 'B \subseteq A$
shows $\exists h. bij\text{-betw } h\ A\ B$

proof (rule *exI*, rule *bij-betw-imageI*)

define X **where** $X = lfp\ (\lambda X. A - (g\ '(B - (f\ 'X))))$
define g' **where** $g' = the\text{-inv-into } (B - (f\ 'X))\ g$
let $?h = \lambda z. if\ z \in X\ then\ f\ z\ else\ g'\ z$

have $X: X = A - (g\ '(B - (f\ 'X)))$
unfolding $X\text{-def}$ **by** (rule *lfp-unfold*) (blast *intro: monoI*)
then have $X\text{-compl}: A - X = g\ '(B - (f\ 'X))$
using $sub2$ **by** *blast*

from $inj2$ **have** $inj2': inj\text{-on } g\ (B - (f\ 'X))$
by (rule *inj-on-subset*) *auto*
with $X\text{-compl}$ **have** $*$: $g'\ '(A - X) = B - (f\ 'X)$
by (*simp add: g'-def*)

from X **have** $X\text{-sub}: X \subseteq A$ **by** *auto*
from $X\ sub1$ **have** $fX\text{-sub}: f\ 'X \subseteq B$ **by** *auto*

show $?h\ 'A = B$

proof –

from $X\text{-sub}$ **have** $?h\ 'A = ?h\ '(X \cup (A - X))$ **by** *auto*
also have $\dots = ?h\ 'X \cup ?h\ '(A - X)$ **by** (*simp only: image-Un*)
also have $?h\ 'X = f\ 'X$ **by** *auto*
also from $*$ **have** $?h\ '(A - X) = B - (f\ 'X)$ **by** *auto*
also from $fX\text{-sub}$ **have** $f\ 'X \cup (B - f\ 'X) = B$ **by** *blast*
finally show *?thesis* .

qed

show $inj\text{-on } ?h\ A$

proof –

from $inj1\ X\text{-sub}$ **have** $on\text{-}X: inj\text{-on } f\ X$
by (rule *subset-inj-on*)

have $on\text{-}X\text{-compl}: inj\text{-on } g'\ (A - X)$

```

unfolding g'-def X-compl
by (rule inj-on-the-inv-into) (rule inj2')

have impossible: False if eq: f a = g' b and a: a ∈ X and b: b ∈ A - X for a
b
proof -
  from a have fa: f a ∈ f ' X by (rule imageI)
  from b have g' b ∈ g' ' (A - X) by (rule imageI)
  with * have g' b ∈ - (f ' X) by simp
  with eq fa show False by simp
qed

show ?thesis
proof (rule inj-onI)
  fix a b
  assume h: ?h a = ?h b
  assume a ∈ A and b ∈ A
  then consider a ∈ X b ∈ X | a ∈ A - X b ∈ A - X
  | a ∈ X b ∈ A - X | a ∈ A - X b ∈ X
  by blast
  then show a = b
  proof cases
    case 1
    with h on-X show ?thesis by (simp add: inj-on-eq-iff)
  next
    case 2
    with h on-X-compl show ?thesis by (simp add: inj-on-eq-iff)
  next
    case 3
    with h impossible [of a b] have False by simp
    then show ?thesis ..
  next
    case 4
    with h impossible [of b a] have False by simp
    then show ?thesis ..
  qed
qed
qed
qed

```

13.9 Inductive datatypes and primitive recursion

Package setup.

ML-file \langle Tools/Old-Datatype/old-datatype-aux.ML \rangle

ML-file \langle Tools/Old-Datatype/old-datatype-prop.ML \rangle

ML-file \langle Tools/Old-Datatype/old-datatype-data.ML \rangle

ML-file \langle Tools/Old-Datatype/old-rep-datatype.ML \rangle

ML-file \langle Tools/Old-Datatype/old-datatype-codegen.ML \rangle

ML-file \langle Tools/BNF/bnf-fp-rec-sugar-util.ML \rangle

ML-file \langle Tools/Old-Datatype/old-primrec.ML \rangle

ML-file \langle Tools/BNF/bnf-lfp-rec-sugar.ML \rangle

Lambda-abstractions with pattern matching:

syntax (ASCII)

-lam-pats-syntax :: *cases-syn* \Rightarrow 'a \Rightarrow 'b ((%-) 10)

syntax

-lam-pats-syntax :: *cases-syn* \Rightarrow 'a \Rightarrow 'b ((λ -) 10)

parse-translation \langle

let

fun fun-tr ctxt [cs] =

let

val x = Syntax.free (fst (Name.variant x (Term.declare-term-frees cs
Name.context)));

val ft = Case-Translation.case-tr true ctxt [x, cs];

in lambda x ft end

in [(syntax-const \langle -lam-pats-syntax \rangle , fun-tr)] end

\rangle

end

14 Cartesian products

theory *Product-Type*

imports *Typedef Inductive Fun*

keywords *inductive-set coinductive-set :: thy-defn*

begin

14.1 *bool* is a datatype

free-constructors (*discs-sels*) *case-bool* **for** *True* | *False*

by *auto*

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

setup \langle *Sign.mandatory-path old* \rangle

old-rep-datatype *True False* **by** (*auto intro: bool-induct*)

setup \langle *Sign.parent-path* \rangle

But erase the prefix for properties that are not generated by *free-constructors*.

setup \langle *Sign.mandatory-path bool* \rangle

lemmas *induct* = *old.bool.induct*

lemmas *inducts* = *old.bool.inducts*

lemmas *rec* = *old.bool.rec*

lemmas *simps* = *bool.distinct bool.case bool.rec*

```

setup ‹Sign.parent-path›

declare case-split [cases type: bool]
  — prefer plain propositional version

lemma [code]: HOL.equal False P  $\longleftrightarrow$   $\neg P$ 
and [code]: HOL.equal True P  $\longleftrightarrow$   $P$ 
and [code]: HOL.equal P False  $\longleftrightarrow$   $\neg P$ 
and [code]: HOL.equal P True  $\longleftrightarrow$   $P$ 
and [code nbe]: HOL.equal P P  $\longleftrightarrow$  True
by (simp-all add: equal)

lemma If-case-cert:
assumes CASE  $\equiv$  ( $\lambda b. \text{If } b \text{ } f \text{ } g$ )
shows (CASE True  $\equiv$  f) &&& (CASE False  $\equiv$  g)
using assms by simp-all

setup ‹Code.declare-case-global @{thm If-case-cert}›

code-printing
constant HOL.equal :: bool  $\Rightarrow$  bool  $\Rightarrow$  bool  $\rightarrow$  (Haskell) infix 4 ==
| class-instance bool :: equal  $\rightarrow$  (Haskell) –

```

14.2 The *unit* type

```

typedef unit = {True}
by auto

definition Unity :: unit ('())
where () = Abs-unit True

lemma unit-eq [no-atp]: u = ()
by (induct u) (simp add: Unity-def)

Simplification procedure for unit-eq. Cannot use this rule directly — it
loops!

simproc-setup unit-eq (x::unit) = ‹
  K (K (fn ct =>
    if HOLogic.is-unit (Thm.term-of ct) then NONE
    else SOME (mk-meta-eq @{thm unit-eq}))))
›

free-constructors case-unit for ()
by auto

```

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

```

setup ‹Sign.mandatory-path old›

old-rep-datatype () by simp

```

```
setup <Sign.parent-path>
```

But erase the prefix for properties that are not generated by *free-constructors*.

```
setup <Sign.mandatory-path unit>
```

```
lemmas induct = old.unit.induct
```

```
lemmas inducts = old.unit.inducts
```

```
lemmas rec = old.unit.rec
```

```
lemmas_simps = unit.case unit.rec
```

```
setup <Sign.parent-path>
```

```
lemma unit-all-eq1: ( $\bigwedge x::unit. PROP P x$ )  $\equiv$  PROP P ()
  by simp
```

```
lemma unit-all-eq2: ( $\bigwedge x::unit. PROP P$ )  $\equiv$  PROP P
  by (rule triv-forall-equality)
```

This rewrite counters the effect of *simproc unit-eq* on $\lambda u::unit. f u$, replacing it by f rather than by $\lambda u. f ()$.

```
lemma unit-abs-eta-conv [simp]: ( $\lambda u::unit. f ()$ ) = f
  by (rule ext) simp
```

```
lemma UNIV-unit: UNIV = {()}
  by auto
```

```
instantiation unit :: default
begin
```

```
definition default = ()
```

```
instance ..
```

```
end
```

```
instantiation unit :: {complete-boolean-algebra, complete-linorder, wellorder}
begin
```

```
definition less-eq-unit :: unit  $\Rightarrow$  unit  $\Rightarrow$  bool
  where (-::unit)  $\leq$  -  $\longleftrightarrow$  True
```

```
lemma less-eq-unit [iff]:  $u \leq v$  for  $u v :: unit$ 
  by (simp add: less-eq-unit-def)
```

```
definition less-unit :: unit  $\Rightarrow$  unit  $\Rightarrow$  bool
  where (-::unit) < -  $\longleftrightarrow$  False
```

```
lemma less-unit [iff]:  $\neg u < v$  for  $u v :: unit$ 
```

by (*simp-all add: less-eq-unit-def less-unit-def*)

definition *bot-unit* :: *unit*
 where [*code-unfold*]: $\perp = ()$

definition *top-unit* :: *unit*
 where [*code-unfold*]: $\top = ()$

definition *inf-unit* :: *unit* \Rightarrow *unit* \Rightarrow *unit*
 where [*simp*]: $- \sqcap - = ()$

definition *sup-unit* :: *unit* \Rightarrow *unit* \Rightarrow *unit*
 where [*simp*]: $- \sqcup - = ()$

definition *Inf-unit* :: *unit set* \Rightarrow *unit*
 where [*simp*]: $\sqcap - = ()$

definition *Sup-unit* :: *unit set* \Rightarrow *unit*
 where [*simp*]: $\sqcup - = ()$

definition *uminus-unit* :: *unit* \Rightarrow *unit*
 where [*simp*]: $- - = ()$

declare *less-eq-unit-def* [*abs-def*, *code-unfold*]
less-unit-def [*abs-def*, *code-unfold*]
inf-unit-def [*abs-def*, *code-unfold*]
sup-unit-def [*abs-def*, *code-unfold*]
Inf-unit-def [*abs-def*, *code-unfold*]
Sup-unit-def [*abs-def*, *code-unfold*]
uminus-unit-def [*abs-def*, *code-unfold*]

instance
 by *intro-classes auto*

end

lemma [*code*]: *HOL.equal* *u v* \longleftrightarrow *True* **for** *u v* :: *unit*
unfolding *equal unit-eq* [*of u*] *unit-eq* [*of v*] **by** (*rule iffI TrueI refl*)+

code-printing
type-constructor *unit* \rightarrow
 (*SML*) *unit*
and (*OCaml*) *unit*
and (*Haskell*) $()$
and (*Scala*) *Unit*
 | **constant** *Unity* \rightarrow
 (*SML*) $()$
and (*OCaml*) $()$
and (*Haskell*) $()$

```

and (Scala) ()
| class-instance unit :: equal  $\rightarrow$ 
  (Haskell) –
| constant HOL.equal :: unit  $\Rightarrow$  unit  $\Rightarrow$  bool  $\rightarrow$ 
  (Haskell) infix 4 ==

```

code-reserved *SML*
unit

code-reserved *OCaml*
unit

code-reserved *Scala*
Unit

14.3 The product type

14.3.1 Type definition

definition *Pair-Rep* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$
where *Pair-Rep* *a b* = $(\lambda x y. x = a \wedge y = b)$

definition *prod* = $\{f. \exists a b. f = \text{Pair-Rep } (a::'a) (b::'b)\}$

typedef $('a, 'b)$ *prod* $((- \times / -) [21, 20] 20) = \text{prod} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \text{ set}$
unfolding *prod-def* **by** *auto*

type-notation (*ASCII*)
prod (**infixr** * 20)

definition *Pair* :: $'a \Rightarrow 'b \Rightarrow 'a \times 'b$
where *Pair* *a b* = *Abs-prod* (*Pair-Rep* *a b*)

lemma *prod-cases*: $(\bigwedge a b. P (\text{Pair } a b)) \Longrightarrow P p$
by (*cases p*) (*auto simp add: prod-def Pair-def Pair-Rep-def*)

free-constructors *case-prod* **for** *Pair fst snd*

proof –

fix *P* :: *bool* **and** *p* :: $'a \times 'b$

show $(\bigwedge x1 x2. p = \text{Pair } x1 x2 \Longrightarrow P) \Longrightarrow P$

by (*cases p*) (*auto simp add: prod-def Pair-def Pair-Rep-def*)

next

fix *a c* :: $'a$ **and** *b d* :: $'b$

have *Pair-Rep* *a b* = *Pair-Rep* *c d* $\longleftrightarrow a = c \wedge b = d$

by (*auto simp add: Pair-Rep-def fun-eq-iff*)

moreover have *Pair-Rep* *a b* $\in \text{prod}$ **and** *Pair-Rep* *c d* $\in \text{prod}$

by (*auto simp add: prod-def*)

ultimately show *Pair* *a b* = *Pair* *c d* $\longleftrightarrow a = c \wedge b = d$

by (*simp add: Pair-def Abs-prod-inject*)

qed

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

setup \langle Sign.mandatory-path old \rangle

old-rep-datatype *Pair*

by (*erule prod-cases*) (*rule prod.inject*)

setup \langle Sign.parent-path \rangle

But erase the prefix for properties that are not generated by *free-constructors*.

setup \langle Sign.mandatory-path prod \rangle

declare *old.prod.inject* [*iff del*]

lemmas *induct* = *old.prod.induct*

lemmas *inducts* = *old.prod.inducts*

lemmas *rec* = *old.prod.rec*

lemmas *simps* = *prod.inject prod.case prod.rec*

setup \langle Sign.parent-path \rangle

declare *prod.case* [*nitpick-simp del*]

declare *old.prod.case-cong-weak* [*cong del*]

declare *prod.case-eq-if* [*mono*]

declare *prod.split* [*no-atp*]

declare *prod.split-asm* [*no-atp*]

prod.split could be declared as [*split*] done after the Splitter has been speeded up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

14.3.2 Tuple syntax

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminal *tuple-args* and *patterns*

syntax

-tuple :: 'a \Rightarrow tuple-args \Rightarrow 'a \times 'b ((1'(-, / -')))

-tuple-arg :: 'a \Rightarrow tuple-args (-)

-tuple-args :: 'a \Rightarrow tuple-args \Rightarrow tuple-args (-, / -)

-pattern :: pttrn \Rightarrow patterns \Rightarrow pttrn (('(-, / -'))

 :: pttrn \Rightarrow patterns (-)

-patterns :: pttrn \Rightarrow patterns \Rightarrow patterns (-, / -)

-unit :: pttrn (('()'))

translations

(*x*, *y*) \equiv *CONST Pair x y*

-pattern *x y* \equiv *CONST Pair x y*

-patterns *x y* \equiv *CONST Pair x y*

-tuple *x* (-tuple-args *y z*) \equiv -tuple *x* (-tuple-arg (-tuple *y z*))

$\lambda(x, y, zs). b \equiv$ *CONST case-prod* ($\lambda x (y, zs). b$)

$\lambda(x, y). b \equiv \text{CONST case-prod } (\lambda x y. b)$
 $\text{-abs } (\text{CONST Pair } x y) t \rightarrow \lambda(x, y). t$

— This rule accommodates tuples in *case* $C \dots (x, y) \dots \Rightarrow \dots$: The (x, y) is parsed as *Pair* $x y$ because it is *logic*, not *pttrn*.

$\lambda(). b \equiv \text{CONST case-unit } b$
 $\text{-abs } (\text{CONST Unity}) t \rightarrow \lambda(). t$

print *case-prod* f as *case-prod* f and *case-prod* f as *case-prod* f

typed-print-translation \langle

```

  let
    fun case-prod-guess-names-tr' T [Abs (x, -, Abs -)] = raise Match
      | case-prod-guess-names-tr' T [Abs (x, xT, t)] =
        (case (head-of t) of
          Const (const-syntax  $\langle$ case-prod $\rangle$ , -) => raise Match
        | - =>
          let
            val (- :: yT :: -) = binder-types (domain-type T) handle Bind => raise
Match;
            val (y, t') = Syntax-Trans.atomic-abs-tr' (y, yT, incr-boundvars 1 t $
Bound 0);
            val (x', t'') = Syntax-Trans.atomic-abs-tr' (x, xT, t');
          in
            Syntax.const syntax-const  $\langle$ -abs $\rangle$  $
              (Syntax.const syntax-const  $\langle$ -pattern $\rangle$  $ x' $ y) $ t''
          end)
        | case-prod-guess-names-tr' T [t] =
          (case head-of t of
            Const (const-syntax  $\langle$ case-prod $\rangle$ , -) => raise Match
          | - =>
            let
              val (xT :: yT :: -) = binder-types (domain-type T) handle Bind =>
raise Match;
              val (y, t') =
                Syntax-Trans.atomic-abs-tr' (y, yT, incr-boundvars 2 t $ Bound 1 $
Bound 0);
              val (x', t'') = Syntax-Trans.atomic-abs-tr' (x, xT, t');
            in
              Syntax.const syntax-const  $\langle$ -abs $\rangle$  $
                (Syntax.const syntax-const  $\langle$ -pattern $\rangle$  $ x' $ y) $ t''
            end)
          | case-prod-guess-names-tr' - - = raise Match;
    in [(const-syntax  $\langle$ case-prod $\rangle$ , K case-prod-guess-names-tr')] end
   $\rangle$ 

```

Reconstruct pattern from (nested) *case-prods*, avoiding eta-contraction of body; required for enclosing "let", if "let" does not avoid eta-contraction, which has been observed to occur.

print-translation \langle

let

```

fun case-prod-tr' [Abs (x, T, t as (Abs abs))] =
  (* case-prod ( $\lambda x y. t$ )  $\Rightarrow \lambda(x, y) t *$ )
  let
    val (y, t') = Syntax-Trans.atomic-abs-tr' abs;
    val (x', t'') = Syntax-Trans.atomic-abs-tr' (x, T, t');
  in
    Syntax.const syntax-const <-abs> $
      (Syntax.const syntax-const <-pattern> $ x' $ y) $ t''
  end
| case-prod-tr' [Abs (x, T, (s as Const (const-syntax <case-prod>, -) $ t))] =
  (* case-prod ( $\lambda x. (case-prod (\lambda y z. t))$ )  $\Rightarrow \lambda(x, y, z). t *$ )
  let
    val Const (syntax-const <-abs>, -) $
      (Const (syntax-const <-pattern>, -) $ y $ z) $ t' =
      case-prod-tr' [t];
    val (x', t'') = Syntax-Trans.atomic-abs-tr' (x, T, t');
  in
    Syntax.const syntax-const <-abs> $
      (Syntax.const syntax-const <-pattern> $ x' $
        (Syntax.const syntax-const <-patterns> $ y $ z)) $ t''
  end
| case-prod-tr' [Const (const-syntax <case-prod>, -) $ t] =
  (* case-prod ( $case-prod (\lambda x y z. t)$ )  $\Rightarrow \lambda((x, y), z). t *$ )
  case-prod-tr' [(case-prod-tr' [t])]
  (* inner case-prod-tr' creates next pattern *)
| case-prod-tr' [Const (syntax-const <-abs>, -) $ x-y $ Abs abs] =
  (* case-prod ( $\lambda pptrn z. t$ )  $\Rightarrow \lambda(pptrn, z). t *$ )
  let val (z, t) = Syntax-Trans.atomic-abs-tr' abs in
    Syntax.const syntax-const <-abs> $
      (Syntax.const syntax-const <-pattern> $ x-y $ z) $ t
  end
| case-prod-tr' - = raise Match;
in [(const-syntax <case-prod>, K case-prod-tr')] end
>

```

14.3.3 Code generator setup

code-printing

```

type-constructor prod  $\rightarrow$ 
  (SML) infix 2 *
  and (OCaml) infix 2 *
  and (Haskell) !((-),/ (-))
  and (Scala) ((-),/ (-))
| constant Pair  $\rightarrow$ 
  (SML) !((-),/ (-))
  and (OCaml) !((-),/ (-))
  and (Haskell) !((-),/ (-))
  and (Scala) !((-),/ (-))
| class-instance prod :: equal  $\rightarrow$ 

```

```

(Haskell) –
| constant HOL.equal :: 'a × 'b ⇒ 'a × 'b ⇒ bool ↪
(Haskell) infix 4 ==
| constant fst ↪ (Haskell) fst
| constant snd ↪ (Haskell) snd

```

14.3.4 Fundamental operations and properties

lemma *Pair-inject*: $(a, b) = (a', b') \implies (a = a' \implies b = b' \implies R) \implies R$
by *simp*

lemma *surj-pair* [*simp*]: $\exists x y. p = (x, y)$
by (*cases p*) *simp*

lemma *fst-eqD*: $\text{fst } (x, y) = a \implies x = a$
by *simp*

lemma *snd-eqD*: $\text{snd } (x, y) = a \implies y = a$
by *simp*

lemma *case-prod-unfold* [*nitpick-unfold*]: $\text{case-prod} = (\lambda c p. c (\text{fst } p) (\text{snd } p))$
by (*simp add: fun-eq-iff split: prod.split*)

lemma *case-prod-conv* [*simp, code*]: $\text{case } (a, b) \text{ of } (c, d) \Rightarrow f c d = f a b$
by (*fact prod.case*)

lemmas *surjective-pairing* = *prod.collapse* [*symmetric*]

lemma *prod-eq-iff*: $s = t \iff \text{fst } s = \text{fst } t \wedge \text{snd } s = \text{snd } t$
by (*cases s, cases t*) *simp*

lemma *prod-eqI* [*intro?*]: $\text{fst } p = \text{fst } q \implies \text{snd } p = \text{snd } q \implies p = q$
by (*simp add: prod-eq-iff*)

lemma *case-prodI*: $f a b \implies \text{case } (a, b) \text{ of } (c, d) \Rightarrow f c d$
by (*rule prod.case [THEN iffD2]*)

lemma *case-prodD*: $\text{case } (a, b) \text{ of } (c, d) \Rightarrow f c d \implies f a b$
by (*rule prod.case [THEN iffD1]*)

lemma *case-prod-Pair* [*simp*]: $\text{case-prod } \text{Pair} = \text{id}$
by (*simp add: fun-eq-iff split: prod.split*)

lemma *case-prod-eta*: $(\lambda(x, y). f (x, y)) = f$
— Subsumes the old *split-Pair* when *f* is the identity function.
by (*simp add: fun-eq-iff split: prod.split*)

lemma *case-prod-comp*: $(\text{case } x \text{ of } (a, b) \Rightarrow (f \circ g) a b) = f (g (fst x)) (snd x)$
by $(\text{cases } x) \text{ simp}$

lemma *The-case-prod*: $The (case\text{-}prod P) = (THE xy. P (fst xy) (snd xy))$
by $(\text{simp add: case-prod-unfold})$

lemma *cond-case-prod-eta*: $(\bigwedge x y. f x y = g (x, y)) \Longrightarrow (\lambda(x, y). f x y) = g$
by $(\text{simp add: case-prod-eta})$

lemma *split-paired-all [no-atp]*: $(\bigwedge x. PROP P x) \equiv (\bigwedge a b. PROP P (a, b))$

proof

fix $a b$

assume $\bigwedge x. PROP P x$

then show $PROP P (a, b)$.

next

fix x

assume $\bigwedge a b. PROP P (a, b)$

from $\langle PROP P (fst x, snd x) \rangle$ **show** $PROP P x$ **by** *simp*

qed

The rule *split-paired-all* does not work with the Simplifier because it also affects premises in congruence rules, where this can lead to premises of the form $\bigwedge a b. \dots = ?P(a, b)$ which cannot be solved by reflexivity.

lemmas *split-tupled-all = split-paired-all unit-all-eq2*

ML \langle

(replace parameters of product type by individual component parameters *)*

local (filtering with exists-paired-all is an essential optimization *)*

fun *exists-paired-all* $(\text{Const } (\mathbf{const_name} \langle \text{Pure.all} \rangle, -) \$ \text{Abs } (-, T, t)) =$
 $\text{can } HOLogic.dest\text{-}prodT T \text{ orelse } \text{exists-paired-all } t$

$| \text{exists-paired-all } (t \$ u) = \text{exists-paired-all } t \text{ orelse } \text{exists-paired-all } u$

$| \text{exists-paired-all } (\text{Abs } (-, -, t)) = \text{exists-paired-all } t$

$| \text{exists-paired-all } - = \text{false};$

val *ss* =

simpset-of

$(\text{put-simpset } HOL\text{-}basic\text{-}ss \ \mathbf{context}$

$\text{addsimps } [\text{@}\{thm \ \text{split-paired-all}\}, \text{@}\{thm \ \text{unit-all-eq2}\}, \text{@}\{thm \ \text{unit-abs-eta-conv}\}]$

$\text{addsimprocs } [\mathbf{simproc} \ \langle \text{unit-eq} \rangle];$

in

fun *split-all-tac* *ctxt* = *SUBGOAL* $(fn (t, i) =>$

$\text{if } \text{exists-paired-all } t \text{ then } \text{safe-full-simp-tac } (\text{put-simpset } ss \ \text{ctxt}) \ i \ \text{else } \text{no-tac};$

fun *unsafe-split-all-tac* *ctxt* = *SUBGOAL* $(fn (t, i) =>$

$\text{if } \text{exists-paired-all } t \text{ then } \text{full-simp-tac } (\text{put-simpset } ss \ \text{ctxt}) \ i \ \text{else } \text{no-tac};$

fun *split-all* *ctxt* *th* =

$\text{if } \text{exists-paired-all } (\text{Thm.prop-of } th)$

$\text{then } \text{full-simplify } (\text{put-simpset } ss \ \text{ctxt}) \ th \ \text{else } th;$

end;

›

setup ‹*map-theory-claset* (fn *ctxt* => *ctxt* *addSbefore* (*split-all-tac*, *split-all-tac*))›

lemma *split-paired-All* [*simp*, *no-atp*]: $(\forall x. P x) \longleftrightarrow (\forall a b. P (a, b))$
 — [*iff*] is not a good idea because it makes *blast* loop
by *fast*

lemma *split-paired-Ex* [*simp*, *no-atp*]: $(\exists x. P x) \longleftrightarrow (\exists a b. P (a, b))$
by *fast*

lemma *split-paired-The* [*no-atp*]: $(THE x. P x) = (THE (a, b). P (a, b))$
 — Can’t be added to *simpset*: loops!
by (*simp add: case-prod-eta*)

Simplification procedure for *cond-case-prod-eta*. Using *case-prod-eta* as a rewrite rule is not general enough, and using *cond-case-prod-eta* directly would render some existing proofs very inefficient; similarly for *prod.case-eq-if*.

ML ‹

local

```

val cond-case-prod-eta-ss =
  simpset-of (put-simpset HOL-basic-ss context addsimps @{thms cond-case-prod-eta});
fun Pair-pat k 0 (Bound m) = (m = k)
  | Pair-pat k i (Const (const_name ‹Pair›, -) $ Bound m $ t) =
    i > 0 andalso m = k + i andalso Pair-pat k (i - 1) t
  | Pair-pat - - - = false;
fun no-args k i (Abs (-, -, t)) = no-args (k + 1) i t
  | no-args k i (t $ u) = no-args k i t andalso no-args k i u
  | no-args k i (Bound m) = m < k orelse m > k + i
  | no-args - - - = true;
fun split-pat tp i (Abs (-, -, t)) = if tp 0 i t then SOME (i, t) else NONE
  | split-pat tp i (Const (const_name ‹case-prod›, -) $ Abs (-, -, t)) = split-pat
tp (i + 1) t
  | split-pat tp i - = NONE;
fun metaeq ctxt lhs rhs = mk-meta-eq (Goal.prove ctxt [] []
  (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs)))
  (K (simp-tac (put-simpset cond-case-prod-eta-ss ctxt) 1)));

fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k + 1) i t
  | beta-term-pat k i (t $ u) =
  Pair-pat k i (t $ u) orelse (beta-term-pat k i t andalso beta-term-pat k i u)
  | beta-term-pat k i t = no-args k i t;
fun eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
  | eta-term-pat - - - = false;
fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
  | subst arg k i (t $ u) =
  if Pair-pat k i (t $ u) then incr-boundvars k arg
  else (subst arg k i t $ subst arg k i u)
  | subst arg k i t = t;

```

```

in
  fun beta-proc ctxt (s as Const (const-name ⟨case-prod⟩, -) $ Abs (-, -, t) $ arg)
  =
    (case split-pat beta-term-pat 1 t of
      SOME (i, f) => SOME (metaeq ctxt s (subst arg 0 i f))
    | NONE => NONE)
  | beta-proc - - = NONE;
  fun eta-proc ctxt (s as Const (const-name ⟨case-prod⟩, -) $ Abs (-, -, t)) =
    (case split-pat eta-term-pat 1 t of
      SOME (-, ft) => SOME (metaeq ctxt s (let val f $ - = ft in f end))
    | NONE => NONE)
  | eta-proc - - = NONE;
end;
>
simproc-setup case-prod-beta (case-prod f z) =
  ⟨K (fn ctxt => fn ct => beta-proc ctxt (Thm.term-of ct))⟩
simproc-setup case-prod-eta (case-prod f) =
  ⟨K (fn ctxt => fn ct => eta-proc ctxt (Thm.term-of ct))⟩

lemma case-prod-beta': (λ(x,y). f x y) = (λx. f (fst x) (snd x))
  by (auto simp: fun-eq-iff)

```

case-prod used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *prod.split* as rewrite.

lemma case-prodI2:

```

  ∧p. (∧a b. p = (a, b) ⇒ c a b) ⇒ case p of (a, b) ⇒ c a b
  by (simp add: split-tupled-all)

```

lemma case-prodI2':

```

  ∧p. (∧a b. (a, b) = p ⇒ c a b x) ⇒ (case p of (a, b) ⇒ c a b) x
  by (simp add: split-tupled-all)

```

lemma case-prodE [elim!]:

```

  (case p of (a, b) ⇒ c a b) ⇒ (∧x y. p = (x, y) ⇒ c x y ⇒ Q) ⇒ Q
  by (induct p) simp

```

lemma case-prodE' [elim!]:

```

  (case p of (a, b) ⇒ c a b) z ⇒ (∧x y. p = (x, y) ⇒ c x y z ⇒ Q) ⇒ Q
  by (induct p) simp

```

lemma case-prodE2:

```

  assumes q: Q (case z of (a, b) ⇒ P a b)
  and r: ∧x y. z = (x, y) ⇒ Q (P x y) ⇒ R
  shows R

```

proof (rule r)

```

  show z = (fst z, snd z) by simp
  then show Q (P (fst z) (snd z))

```

using q **by** (*simp add: case-prod-unfold*)
qed

lemma *case-prodD'*: (*case* (a, b) *of* (c, d) $\Rightarrow R\ c\ d$) $c \Longrightarrow R\ a\ b\ c$
by *simp*

lemma *mem-case-prodI*: $z \in c\ a\ b \Longrightarrow z \in (\text{case } (a, b) \text{ of } (d, e) \Rightarrow c\ d\ e)$
by *simp*

lemma *mem-case-prodI2* [*intro!*]:
 $\bigwedge p. (\bigwedge a\ b. p = (a, b) \Longrightarrow z \in c\ a\ b) \Longrightarrow z \in (\text{case } p \text{ of } (a, b) \Rightarrow c\ a\ b)$
by (*simp only: split-tupled-all*) *simp*

declare *mem-case-prodI* [*intro!*] — postponed to maintain traditional declaration order!

declare *case-prodI2'* [*intro!*] — postponed to maintain traditional declaration order!

declare *case-prodI2* [*intro!*] — postponed to maintain traditional declaration order!

declare *case-prodI* [*intro!*] — postponed to maintain traditional declaration order!

lemma *mem-case-prodE* [*elim!*]:
assumes $z \in \text{case-prod } c\ p$
obtains $x\ y$ **where** $p = (x, y)$ **and** $z \in c\ x\ y$
using *assms* **by** (*rule case-prodE2*)

ML \langle
local (** filtering with exists-p-split is an essential optimization **)
fun *exists-p-split* (*Const* (**const-name** $\langle \text{case-prod} \rangle, -$) $\$ - \$$ (*Const* (**const-name** $\langle \text{Pair} \rangle, -$) $\$ - \$$))
 $= \text{true}$
 $\quad | \text{exists-p-split } (t\ \$\ u) = \text{exists-p-split } t\ \text{orelse } \text{exists-p-split } u$
 $\quad | \text{exists-p-split } (\text{Abs } (-, -, t)) = \text{exists-p-split } t$
 $\quad | \text{exists-p-split } - = \text{false};$
in
 $\text{split-conv-tac } \text{ctxt} = \text{SUBGOAL } (\text{fn } (t, i) \Rightarrow$
 $\quad \text{if } \text{exists-p-split } t$
 $\quad \text{then } \text{safe-full-simp-tac } (\text{put-simpset } \text{HOL-basic-ss } \text{ctxt } \text{addsimps } @\{\text{thms } \text{case-prod-conv}\})$
 $\quad i$
 $\quad \text{else } \text{no-tac});$
 $\text{end};$
 \rangle

setup $\langle \text{map-theory-claset } (\text{fn } \text{ctxt} \Rightarrow \text{ctxt } \text{addSbefore } (\text{split-conv-tac}, \text{split-conv-tac})) \rangle$

lemma *split-eta-SetCompr* [*simp, no-atp*]: $(\lambda u. \exists x\ y. u = (x, y) \wedge P\ (x, y)) = P$
by (*rule ext*) *fast*

lemma *split-eta-SetCompr2* [*simp, no-atp*]: $(\lambda u. \exists x\ y. u = (x, y) \wedge P\ x\ y) = \text{case-prod } P$

by (rule ext) fast

lemma *split-part* [simp]: $(\lambda(a,b). P \wedge Q a b) = (\lambda ab. P \wedge \text{case-prod } Q ab)$
 — Allows simplifications of nested splits in case of independent predicates.
 by (rule ext) blast

lemma *split-comp-eq*:
 fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$
 and $g :: 'd \Rightarrow 'a$
 shows $(\lambda u. f (g (fst u)) (snd u)) = \text{case-prod } (\lambda x. f (g x))$
 by (rule ext) auto

lemma *pair-imageI* [intro]: $(a, b) \in A \Longrightarrow f a b \in (\lambda(a, b). f a b) ` A$
 by (rule image-eqI [where $x = (a, b)$]) auto

lemma *Collect-const-case-prod*[simp]: $\{(a,b). P\} = (\text{if } P \text{ then UNIV else }\{\})$
 by auto

lemma *The-split-eq* [simp]: $(\text{THE } (x',y'). x = x' \wedge y = y') = (x, y)$
 by blast

lemma *case-prod-beta*: $\text{case-prod } f p = f (fst p) (snd p)$
 by (fact prod.case-eq-if)

lemma *prod-cases3* [cases type]:
 obtains (fields) $a b c$ where $y = (a, b, c)$
proof (cases y)
 case (Pair a b)
 with that show ?thesis
 by (cases b) blast
qed

lemma *prod-induct3* [case-names fields, induct type]:
 $(\bigwedge a b c. P (a, b, c)) \Longrightarrow P x$
 by (cases x) blast

lemma *prod-cases4* [cases type]:
 obtains (fields) $a b c d$ where $y = (a, b, c, d)$
proof (cases y)
 case (fields a b c)
 with that show ?thesis
 by (cases c) blast
qed

lemma *prod-induct4* [case-names fields, induct type]:

$(\bigwedge a\ b\ c\ d. P\ (a, b, c, d)) \implies P\ x$
by (cases x) blast

lemma *prod-cases5* [cases type]:
obtains (fields) a b c d e **where** $y = (a, b, c, d, e)$
proof (cases y)
case (fields a b c d)
with that show ?thesis
by (cases d) blast
qed

lemma *prod-induct5* [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e. P\ (a, b, c, d, e)) \implies P\ x$
by (cases x) blast

lemma *prod-cases6* [cases type]:
obtains (fields) a b c d e f **where** $y = (a, b, c, d, e, f)$
proof (cases y)
case (fields a b c d e)
with that show ?thesis
by (cases e) blast
qed

lemma *prod-induct6* [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f. P\ (a, b, c, d, e, f)) \implies P\ x$
by (cases x) blast

lemma *prod-cases7* [cases type]:
obtains (fields) a b c d e f g **where** $y = (a, b, c, d, e, f, g)$
proof (cases y)
case (fields a b c d e f)
with that show ?thesis
by (cases f) blast
qed

lemma *prod-induct7* [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g. P\ (a, b, c, d, e, f, g)) \implies P\ x$
by (cases x) blast

definition *internal-case-prod* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
where *internal-case-prod* \equiv *case-prod*

lemma *internal-case-prod-conv*: *internal-case-prod* c (a, b) = c a b
by (simp only: *internal-case-prod-def case-prod-conv*)

ML-file <Tools/split-rule.ML>

hide-const *internal-case-prod*

14.3.5 Derived operations

definition $curry :: ('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$
where $curry = (\lambda c x y. c (x, y))$

lemma $curry\text{-}conv$ [*simp*, *code*]: $curry f a b = f (a, b)$
by (*simp add: curry-def*)

lemma $curryI$ [*intro!*]: $f (a, b) \Longrightarrow curry f a b$
by (*simp add: curry-def*)

lemma $curryD$ [*dest!*]: $curry f a b \Longrightarrow f (a, b)$
by (*simp add: curry-def*)

lemma $curryE$: $curry f a b \Longrightarrow (f (a, b) \Longrightarrow Q) \Longrightarrow Q$
by (*simp add: curry-def*)

lemma $curry\text{-}case\text{-}prod$ [*simp*]: $curry (case\text{-}prod f) = f$
by (*simp add: curry-def case-prod-unfold*)

lemma $case\text{-}prod\text{-}curry$ [*simp*]: $case\text{-}prod (curry f) = f$
by (*simp add: curry-def case-prod-unfold*)

lemma $curry\text{-}K$: $curry (\lambda x. c) = (\lambda x y. c)$
by (*simp add: fun-eq-iff*)

The composition-uncurry combinator.

definition $scomp :: ('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (**infixl** $\circ\rightarrow$ 60)
where $f \circ\rightarrow g = (\lambda x. case\text{-}prod g (f x))$

no-notation $scomp$ (**infixl** $\circ\rightarrow$ 60)

bundle $state\text{-}combinator\text{-}syntax$
begin

notation $fcomp$ (**infixl** $\circ>$ 60)

notation $scomp$ (**infixl** $\circ\rightarrow$ 60)

end

context

includes $state\text{-}combinator\text{-}syntax$

begin

lemma $scomp\text{-}unfold$: $(\circ\rightarrow) = (\lambda f g x. g (fst (f x)) (snd (f x)))$
by (*simp add: fun-eq-iff scomp-def case-prod-unfold*)

lemma $scomp\text{-}apply$ [*simp*]: $(f \circ\rightarrow g) x = case\text{-}prod g (f x)$
by (*simp add: scomp-unfold case-prod-unfold*)

lemma *Pair-scomp*: $\text{Pair } x \circ\rightarrow f = f x$
by (*simp add: fun-eq-iff*)

lemma *scomp-Pair*: $x \circ\rightarrow \text{Pair} = x$
by (*simp add: fun-eq-iff*)

lemma *scomp-scomp*: $(f \circ\rightarrow g) \circ\rightarrow h = f \circ\rightarrow (\lambda x. g x \circ\rightarrow h)$
by (*simp add: fun-eq-iff scomp-unfold*)

lemma *scomp-fcomp*: $(f \circ\rightarrow g) \circ> h = f \circ\rightarrow (\lambda x. g x \circ> h)$
by (*simp add: fun-eq-iff scomp-unfold fcomp-def*)

lemma *fcomp-scomp*: $(f \circ> g) \circ\rightarrow h = f \circ> (g \circ\rightarrow h)$
by (*simp add: fun-eq-iff scomp-unfold*)

end

code-printing

constant *scomp* \rightarrow (*Eval*) **infixl** 3 $\#-\rightarrow$

map-prod — action of the product functor upon functions.

definition *map-prod* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'd$
where *map-prod* $f g = (\lambda(x, y). (f x, g y))$

lemma *map-prod-simp* [*simp, code*]: $\text{map-prod } f g (a, b) = (f a, g b)$
by (*simp add: map-prod-def*)

functor *map-prod*: *map-prod*
by (*auto simp add: split-paired-all*)

lemma *fst-map-prod* [*simp*]: $\text{fst } (\text{map-prod } f g x) = f (\text{fst } x)$
by (*cases x*) *simp-all*

lemma *snd-map-prod* [*simp*]: $\text{snd } (\text{map-prod } f g x) = g (\text{snd } x)$
by (*cases x*) *simp-all*

lemma *fst-comp-map-prod* [*simp*]: $\text{fst} \circ \text{map-prod } f g = f \circ \text{fst}$
by (*rule ext*) *simp-all*

lemma *snd-comp-map-prod* [*simp*]: $\text{snd} \circ \text{map-prod } f g = g \circ \text{snd}$
by (*rule ext*) *simp-all*

lemma *map-prod-compose*: $\text{map-prod } (f1 \circ f2) (g1 \circ g2) = (\text{map-prod } f1 g1 \circ \text{map-prod } f2 g2)$
by (*rule ext*) (*simp add: map-prod.compositionality comp-def*)

lemma *map-prod-ident* [*simp*]: $\text{map-prod } (\lambda x. x) (\lambda y. y) = (\lambda z. z)$
by (*rule ext*) (*simp add: map-prod.identity*)

lemma *map-prod-imageI* [*intro*]: $(a, b) \in R \implies (f a, g b) \in \text{map-prod } f g \text{ ' } R$
by (*rule image-eqI*) *simp-all*

lemma *prod-fun-imageE* [*elim!*]:
assumes *major*: $c \in \text{map-prod } f g \text{ ' } R$
and cases: $\bigwedge x y. c = (f x, g y) \implies (x, y) \in R \implies P$
shows P
proof (*rule major [THEN imageE]*)
fix x
assume $c = \text{map-prod } f g x x \in R$
then show P
using cases by (*cases x*) *simp*
qed

definition *apfst* :: $('a \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'b$
where $\text{apfst } f = \text{map-prod } f \text{ id}$

definition *apsnd* :: $('b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'c$
where $\text{apsnd } f = \text{map-prod id } f$

lemma *apfst-conv* [*simp*, *code*]: $\text{apfst } f (x, y) = (f x, y)$
by (*simp add: apfst-def*)

lemma *apsnd-conv* [*simp*, *code*]: $\text{apsnd } f (x, y) = (x, f y)$
by (*simp add: apsnd-def*)

lemma *fst-apfst* [*simp*]: $\text{fst } (\text{apfst } f x) = f (\text{fst } x)$
by (*cases x*) *simp*

lemma *fst-comp-apfst* [*simp*]: $\text{fst} \circ \text{apfst } f = f \circ \text{fst}$
by (*simp add: fun-eq-iff*)

lemma *fst-apsnd* [*simp*]: $\text{fst } (\text{apsnd } f x) = \text{fst } x$
by (*cases x*) *simp*

lemma *fst-comp-apsnd* [*simp*]: $\text{fst} \circ \text{apsnd } f = \text{fst}$
by (*simp add: fun-eq-iff*)

lemma *snd-apfst* [*simp*]: $\text{snd } (\text{apfst } f x) = \text{snd } x$
by (*cases x*) *simp*

lemma *snd-comp-apfst* [*simp*]: $\text{snd} \circ \text{apfst } f = \text{snd}$
by (*simp add: fun-eq-iff*)

lemma *snd-apsnd* [*simp*]: $\text{snd } (\text{apsnd } f x) = f (\text{snd } x)$
by (*cases x*) *simp*

lemma *snd-comp-apsnd* [*simp*]: $\text{snd} \circ \text{apsnd } f = f \circ \text{snd}$
by (*simp add: fun-eq-iff*)

lemma *apfst-compose*: $apfst\ f\ (apfst\ g\ x) = apfst\ (f\ \circ\ g)\ x$
by (*cases x*) *simp*

lemma *apsnd-compose*: $apsnd\ f\ (apsnd\ g\ x) = apsnd\ (f\ \circ\ g)\ x$
by (*cases x*) *simp*

lemma *apfst-apsnd [simp]*: $apfst\ f\ (apsnd\ g\ x) = (f\ (fst\ x),\ g\ (snd\ x))$
by (*cases x*) *simp*

lemma *apsnd-apfst [simp]*: $apsnd\ f\ (apfst\ g\ x) = (g\ (fst\ x),\ f\ (snd\ x))$
by (*cases x*) *simp*

lemma *apfst-id [simp]*: $apfst\ id = id$
by (*simp add: fun-eq-iff*)

lemma *apsnd-id [simp]*: $apsnd\ id = id$
by (*simp add: fun-eq-iff*)

lemma *apfst-eq-conv [simp]*: $apfst\ f\ x = apfst\ g\ x \longleftrightarrow f\ (fst\ x) = g\ (fst\ x)$
by (*cases x*) *simp*

lemma *apsnd-eq-conv [simp]*: $apsnd\ f\ x = apsnd\ g\ x \longleftrightarrow f\ (snd\ x) = g\ (snd\ x)$
by (*cases x*) *simp*

lemma *apsnd-apfst-commute*: $apsnd\ f\ (apfst\ g\ p) = apfst\ g\ (apsnd\ f\ p)$
by *simp*

context
begin

local-setup \langle *Local-Theory.map-background-naming* (*Name-Space.mandatory-path prod*) \rangle

definition *swap* :: $'a \times 'b \Rightarrow 'b \times 'a$
where $swap\ p = (snd\ p,\ fst\ p)$

end

lemma *swap-simp [simp]*: $prod.swap\ (x,\ y) = (y,\ x)$
by (*simp add: prod.swap-def*)

lemma *swap-swap [simp]*: $prod.swap\ (prod.swap\ p) = p$
by (*cases p*) *simp*

lemma *swap-comp-swap [simp]*: $prod.swap \circ prod.swap = id$
by (*simp add: fun-eq-iff*)

lemma *pair-in-swap-image [simp]*: $(y,\ x) \in prod.swap\ ` A \longleftrightarrow (x,\ y) \in A$

by (*auto intro!*: *image-eqI*)

lemma *inj-swap* [*simp*]: *inj-on prod.swap A*
by (*rule inj-onI*) *auto*

lemma *swap-inj-on*: *inj-on* ($\lambda(i, j). (j, i)$) *A*
by (*rule inj-onI*) *auto*

lemma *surj-swap* [*simp*]: *surj prod.swap*
by (*rule surjI* [*of - prod.swap*]) *simp*

lemma *bij-swap* [*simp*]: *bij prod.swap*
by (*simp add: bij-def*)

lemma *case-swap* [*simp*]: (*case prod.swap p of* (y, x) $\Rightarrow f x y$) = (*case p of* (x, y) $\Rightarrow f x y$)
by (*cases p*) *simp*

lemma *fst-swap* [*simp*]: *fst (prod.swap x) = snd x*
by (*cases x*) *simp*

lemma *snd-swap* [*simp*]: *snd (prod.swap x) = fst x*
by (*cases x*) *simp*

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: *'a set* \Rightarrow *'b set* \Rightarrow (*'a* \times *'b*) *set*
where *Sigma A B* $\equiv \bigcup x \in A. \bigcup y \in B x. \{Pair x y\}$

abbreviation *Times* :: *'a set* \Rightarrow *'b set* \Rightarrow (*'a* \times *'b*) *set* (**infixr** \times 80)
where *A* \times *B* \equiv *Sigma A* ($\lambda-. B$)

hide-const (**open**) *Times*

bundle *no-Set-Product-syntax* **begin**
no-notation *Product-Type.Times* (**infixr** \times 80)
end
bundle *Set-Product-syntax* **begin**
notation *Product-Type.Times* (**infixr** \times 80)
end

syntax
-Sigma :: *pttrn* \Rightarrow *'a set* \Rightarrow *'b set* \Rightarrow (*'a* \times *'b*) *set* ((*3SIGMA* *-:/ -*) [0, 0, 10]
10)

translations
SIGMA *x:A. B* \equiv *CONST Sigma A* ($\lambda x. B$)

lemma *SigmaI* [*intro!*]: $a \in A \Longrightarrow b \in B \Longrightarrow (a, b) \in$ *Sigma A B*
unfolding *Sigma-def* **by** *blast*

lemma *SigmaE* [*elim!*]: $c \in \text{Sigma } A \ B \Longrightarrow (\bigwedge x \ y. x \in A \Longrightarrow y \in B \ x \Longrightarrow c = (x, y) \Longrightarrow P) \Longrightarrow P$

— The general elimination rule.

unfolding *Sigma-def* **by** *blast*

Elimination of $(a, b) \in A \times B$ – introduces no eigenvariables.

lemma *SigmaD1*: $(a, b) \in \text{Sigma } A \ B \Longrightarrow a \in A$

by *blast*

lemma *SigmaD2*: $(a, b) \in \text{Sigma } A \ B \Longrightarrow b \in B \ a$

by *blast*

lemma *SigmaE2*: $(a, b) \in \text{Sigma } A \ B \Longrightarrow (a \in A \Longrightarrow b \in B \ a \Longrightarrow P) \Longrightarrow P$

by *blast*

lemma *Sigma-cong*: $A = B \Longrightarrow (\bigwedge x. x \in B \Longrightarrow C \ x = D \ x) \Longrightarrow (\text{SIGMA } x:A. C \ x) = (\text{SIGMA } x:B. D \ x)$

by *auto*

lemma *Sigma-mono*: $A \subseteq C \Longrightarrow (\bigwedge x. x \in A \Longrightarrow B \ x \subseteq D \ x) \Longrightarrow \text{Sigma } A \ B \subseteq \text{Sigma } C \ D$

by *blast*

lemma *Sigma-empty1* [*simp*]: $\text{Sigma } \{\} \ B = \{\}$

by *blast*

lemma *Sigma-empty2* [*simp*]: $A \times \{\} = \{\}$

by *blast*

lemma *UNIV-Times-UNIV* [*simp*]: $\text{UNIV} \times \text{UNIV} = \text{UNIV}$

by *auto*

lemma *Compl-Times-UNIV1* [*simp*]: $-(\text{UNIV} \times A) = \text{UNIV} \times (-A)$

by *auto*

lemma *Compl-Times-UNIV2* [*simp*]: $-(A \times \text{UNIV}) = (-A) \times \text{UNIV}$

by *auto*

lemma *mem-Sigma-iff* [*iff*]: $(a, b) \in \text{Sigma } A \ B \longleftrightarrow a \in A \wedge b \in B \ a$

by *blast*

lemma *mem-Times-iff*: $x \in A \times B \longleftrightarrow \text{fst } x \in A \wedge \text{snd } x \in B$

by (*induct x*) *simp*

lemma *Sigma-empty-iff*: $(\text{SIGMA } i:I. X \ i) = \{\} \longleftrightarrow (\forall i \in I. X \ i = \{\})$

by *auto*

lemma *Times-subset-cancel2*: $x \in C \Longrightarrow A \times C \subseteq B \times C \longleftrightarrow A \subseteq B$

by *blast*

lemma *Times-eq-cancel2*: $x \in C \implies A \times C = B \times C \longleftrightarrow A = B$
by (*blast elim: equalityE*)

lemma *Collect-case-prod-Sigma*: $\{(x, y). P\ x \wedge Q\ x\ y\} = (\text{SIGMA } x:\text{Collect } P.\ \text{Collect } (Q\ x))$
by *blast*

lemma *Collect-case-prod [simp]*: $\{(a, b). P\ a \wedge Q\ b\} = \text{Collect } P \times \text{Collect } Q$
by (*fact Collect-case-prod-Sigma*)

lemma *Collect-case-prodD*: $x \in \text{Collect } (\text{case-prod } A) \implies A\ (fst\ x)\ (snd\ x)$
by *auto*

lemma *Collect-case-prod-mono*: $A \leq B \implies \text{Collect } (\text{case-prod } A) \subseteq \text{Collect } (\text{case-prod } B)$
by *auto (auto elim!: le-funE)*

lemma *Collect-split-mono-strong*:
 $X = fst\ 'A \implies Y = snd\ 'A \implies \forall a \in X. \forall b \in Y. P\ a\ b \longrightarrow Q\ a\ b$
 $\implies A \subseteq \text{Collect } (\text{case-prod } P) \implies A \subseteq \text{Collect } (\text{case-prod } Q)$
by *fastforce*

lemma *UN-Times-distrib*: $(\bigcup (a, b) \in A \times B. E\ a \times F\ b) = \bigcup (E\ 'A) \times \bigcup (F\ 'B)$
— Suggested by Pierre Chartier
by *blast*

lemma *split-paired-Ball-Sigma [simp, no-atp]*: $(\forall z \in \text{Sigma } A\ B. P\ z) \longleftrightarrow (\forall x \in A. \forall y \in B\ x. P\ (x, y))$
by *blast*

lemma *split-paired-Bex-Sigma [simp, no-atp]*: $(\exists z \in \text{Sigma } A\ B. P\ z) \longleftrightarrow (\exists x \in A. \exists y \in B\ x. P\ (x, y))$
by *blast*

lemma *Sigma-Un-distrib1*: $\text{Sigma } (I \cup J)\ C = \text{Sigma } I\ C \cup \text{Sigma } J\ C$
by *blast*

lemma *Sigma-Un-distrib2*: $(\text{SIGMA } i:I. A\ i \cup B\ i) = \text{Sigma } I\ A \cup \text{Sigma } I\ B$
by *blast*

lemma *Sigma-Int-distrib1*: $\text{Sigma } (I \cap J)\ C = \text{Sigma } I\ C \cap \text{Sigma } J\ C$
by *blast*

lemma *Sigma-Int-distrib2*: $(\text{SIGMA } i:I. A\ i \cap B\ i) = \text{Sigma } I\ A \cap \text{Sigma } I\ B$
by *blast*

lemma *Sigma-Diff-distrib1*: $\text{Sigma } (I - J)\ C = \text{Sigma } I\ C - \text{Sigma } J\ C$
by *blast*

lemma *Sigma-Diff-distrib2*: $(\text{SIGMA } i:I. A \ i - B \ i) = \text{Sigma } I \ A - \text{Sigma } I \ B$
by *blast*

lemma *Sigma-Union*: $\text{Sigma } (\bigcup X) \ B = (\bigcup A \in X. \ \text{Sigma } A \ B)$
by *blast*

lemma *Pair-vimage-Sigma*: $\text{Pair } x - ' \ \text{Sigma } A \ f = (\text{if } x \in A \ \text{then } f \ x \ \text{else } \{\})$
by *auto*

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A \cup B) \times C = A \times C \cup B \times C$
by (*fact Sigma-Un-distrib1*)

lemma *Times-Int-distrib1*: $(A \cap B) \times C = A \times C \cap B \times C$
by (*fact Sigma-Int-distrib1*)

lemma *Times-Diff-distrib1*: $(A - B) \times C = A \times C - B \times C$
by (*fact Sigma-Diff-distrib1*)

lemma *Times-empty [simp]*: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
by *auto*

lemma *times-subset-iff*: $A \times C \subseteq B \times D \longleftrightarrow A = \{\} \vee C = \{\} \vee A \subseteq B \wedge C \subseteq D$
by *blast*

lemma *times-eq-iff*: $A \times B = C \times D \longleftrightarrow A = C \wedge B = D \vee (A = \{\} \vee B = \{\}) \wedge (C = \{\} \vee D = \{\})$
by *auto*

lemma *fst-image-times [simp]*: $\text{fst } ' (A \times B) = (\text{if } B = \{\} \ \text{then } \{\} \ \text{else } A)$
by *force*

lemma *snd-image-times [simp]*: $\text{snd } ' (A \times B) = (\text{if } A = \{\} \ \text{then } \{\} \ \text{else } B)$
by *force*

lemma *fst-image-Sigma*: $\text{fst } ' (\text{Sigma } A \ B) = \{x \in A. \ B(x) \neq \{\}\}$
by *force*

lemma *snd-image-Sigma*: $\text{snd } ' (\text{Sigma } A \ B) = (\bigcup x \in A. \ B \ x)$
by *force*

lemma *vimage-fst*: $\text{fst } - ' \ A = A \times \text{UNIV}$
by *auto*

lemma *vimage-snd*: $\text{snd } - ' \ A = \text{UNIV} \times A$
by *auto*

lemma *insert-Times-insert* [simp]:

$$\text{insert } a \ A \times \text{insert } b \ B = \text{insert } (a,b) \ (A \times \text{insert } b \ B \cup \{a\} \times B)$$

by *blast*

lemma *vimage-Times*: $f \text{ --' } (A \times B) = (fst \circ f) \text{ --' } A \cap (snd \circ f) \text{ --' } B$

proof (*rule set-eqI*)

$$\text{show } x \in f \text{ --' } (A \times B) \longleftrightarrow x \in (fst \circ f) \text{ --' } A \cap (snd \circ f) \text{ --' } B \text{ for } x$$

by (*cases f x*) (*auto split: prod.split*)

qed

lemma *Times-Int-Times*: $A \times B \cap C \times D = (A \cap C) \times (B \cap D)$

by *auto*

lemma *image-paired-Times*:

$$(\lambda(x,y). (f \ x, g \ y)) \text{ --' } (A \times B) = (f \text{ --' } A) \times (g \text{ --' } B)$$

by *auto*

lemma *product-swap*: $\text{prod.swap} \text{ --' } (A \times B) = B \times A$

by (*auto simp add: set-eq-iff*)

lemma *swap-product*: $(\lambda(i, j). (j, i)) \text{ --' } (A \times B) = B \times A$

by (*auto simp add: set-eq-iff*)

lemma *image-split-eq-Sigma*: $(\lambda x. (f \ x, g \ x)) \text{ --' } A = \text{Sigma } (f \text{ --' } A) \ (\lambda x. g \text{ --' } (f \text{ --' } \{x\} \cap A))$

proof (*safe intro! imageI*)

fix *a b*

assume *: $a \in A \ b \in A$ **and** $eq: f \ a = f \ b$

show $(f \ b, g \ a) \in (\lambda x. (f \ x, g \ x)) \text{ --' } A$

using * *eq[symmetric]* **by** *auto*

qed *simp-all*

lemma *subset-fst-snd*: $A \subseteq (fst \text{ --' } A \times snd \text{ --' } A)$

by *force*

lemma *inj-on-afst* [simp]: $\text{inj-on } (afst \ f) \ (A \times UNIV) \longleftrightarrow \text{inj-on } f \ A$

by (*auto simp add: inj-on-def*)

lemma *inj-afst* [simp]: $\text{inj } (afst \ f) \longleftrightarrow \text{inj } f$

using *inj-on-afst[of f UNIV]* **by** *simp*

lemma *inj-on-apsnd* [simp]: $\text{inj-on } (apsnd \ f) \ (UNIV \times A) \longleftrightarrow \text{inj-on } f \ A$

by (*auto simp add: inj-on-def*)

lemma *inj-apsnd* [simp]: $\text{inj } (apsnd \ f) \longleftrightarrow \text{inj } f$

using *inj-on-apsnd[of f UNIV]* **by** *simp*

context

begin

qualified definition $product :: 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \times 'b)\ set$
where [*code-abbrev*]: $product\ A\ B = A \times B$

lemma *member-product*: $x \in Product\text{-}Type.product\ A\ B \longleftrightarrow x \in A \times B$
by (*simp add: product-def*)

end

The following *map-prod* lemmas are due to Joachim Breitner:

lemma *map-prod-inj-on*:

assumes *inj-on f A*

and *inj-on g B*

shows *inj-on (map-prod f g) (A × B)*

proof (*rule inj-onI*)

fix $x :: 'a \times 'c$

fix $y :: 'a \times 'c$

assume $x \in A \times B$

then have $fst\ x \in A$ **and** $snd\ x \in B$ **by** *auto*

assume $y \in A \times B$

then have $fst\ y \in A$ **and** $snd\ y \in B$ **by** *auto*

assume $map\text{-}prod\ f\ g\ x = map\text{-}prod\ f\ g\ y$

then have $fst\ (map\text{-}prod\ f\ g\ x) = fst\ (map\text{-}prod\ f\ g\ y)$ **by** *auto*

then have $f\ (fst\ x) = f\ (fst\ y)$ **by** (*cases x, cases y auto*)

with $\langle inj\text{-}on\ f\ A \rangle$ **and** $\langle fst\ x \in A \rangle$ **and** $\langle fst\ y \in A \rangle$ **have** $fst\ x = fst\ y$

by (*auto dest: inj-onD*)

moreover from $\langle map\text{-}prod\ f\ g\ x = map\text{-}prod\ f\ g\ y \rangle$

have $snd\ (map\text{-}prod\ f\ g\ x) = snd\ (map\text{-}prod\ f\ g\ y)$ **by** *auto*

then have $g\ (snd\ x) = g\ (snd\ y)$ **by** (*cases x, cases y auto*)

with $\langle inj\text{-}on\ g\ B \rangle$ **and** $\langle snd\ x \in B \rangle$ **and** $\langle snd\ y \in B \rangle$ **have** $snd\ x = snd\ y$

by (*auto dest: inj-onD*)

ultimately show $x = y$ **by** (*rule prod-eqI*)

qed

lemma *map-prod-surj*:

fixes $f :: 'a \Rightarrow 'b$

and $g :: 'c \Rightarrow 'd$

assumes *surj f* **and** *surj g*

shows *surj (map-prod f g)*

unfolding *surj-def*

proof

fix $y :: 'b \times 'd$

from $\langle surj\ f \rangle$ **obtain** a **where** $fst\ y = f\ a$

by (*auto elim: surjE*)

moreover

from $\langle surj\ g \rangle$ **obtain** b **where** $snd\ y = g\ b$

by (*auto elim: surjE*)

ultimately have $(fst\ y, snd\ y) = map\text{-}prod\ f\ g\ (a, b)$

by *auto*

```

then show  $\exists x. y = \text{map-prod } f \ g \ x$ 
  by auto
qed

```

lemma *map-prod-surj-on*:

```

assumes  $f \ 'A = A'$  and  $g \ 'B = B'$ 
shows  $\text{map-prod } f \ g \ '(A \times B) = A' \times B'$ 
unfolding image-def
proof (rule set-eqI, rule iffI)
  fix  $x :: 'a \times 'c$ 
  assume  $x \in \{y :: 'a \times 'c. \exists x :: 'b \times 'd \in A \times B. y = \text{map-prod } f \ g \ x\}$ 
  then obtain  $y$  where  $y \in A \times B$  and  $x = \text{map-prod } f \ g \ y$ 
    by blast
  from  $\langle \text{image } f \ A = A' \rangle$  and  $\langle y \in A \times B \rangle$  have  $f \ (\text{fst } y) \in A'$ 
    by auto
  moreover from  $\langle \text{image } g \ B = B' \rangle$  and  $\langle y \in A \times B \rangle$  have  $g \ (\text{snd } y) \in B'$ 
    by auto
  ultimately have  $(f \ (\text{fst } y), g \ (\text{snd } y)) \in (A' \times B')$ 
    by auto
  with  $\langle x = \text{map-prod } f \ g \ y \rangle$  show  $x \in A' \times B'$ 
    by (cases  $y$ ) auto
next
  fix  $x :: 'a \times 'c$ 
  assume  $x \in A' \times B'$ 
  then have  $\text{fst } x \in A'$  and  $\text{snd } x \in B'$ 
    by auto
  from  $\langle \text{image } f \ A = A' \rangle$  and  $\langle \text{fst } x \in A' \rangle$  have  $\text{fst } x \in \text{image } f \ A$ 
    by auto
  then obtain  $a$  where  $a \in A$  and  $\text{fst } x = f \ a$ 
    by (rule imageE)
  moreover from  $\langle \text{image } g \ B = B' \rangle$  and  $\langle \text{snd } x \in B' \rangle$  obtain  $b$  where  $b \in B$ 
and  $\text{snd } x = g \ b$ 
    by auto
  ultimately have  $(\text{fst } x, \text{snd } x) = \text{map-prod } f \ g \ (a, b)$ 
    by auto
  moreover from  $\langle a \in A \rangle$  and  $\langle b \in B \rangle$  have  $(a, b) \in A \times B$ 
    by auto
  ultimately have  $\exists y \in A \times B. x = \text{map-prod } f \ g \ y$ 
    by auto
  then show  $x \in \{x. \exists y \in A \times B. x = \text{map-prod } f \ g \ y\}$ 
    by auto
qed

```

14.4 Simproc for rewriting a set comprehension into a point-free expression

ML-file $\langle \text{Tools}/\text{set-comprehension-pointfree.ML} \rangle$

setup \langle


```

      else NONE)
    | - => NONE)
  end
  | - => NONE))
>

```

ML-file $\langle Tools/inductive-set.ML \rangle$

14.7 Legacy theorem bindings and duplicates

```

lemmas fst-conv = prod.sel(1)
lemmas snd-conv = prod.sel(2)
lemmas split-def = case-prod-unfold
lemmas split-beta' = case-prod-beta'
lemmas split-beta = prod.case-eq-if
lemmas split-conv = case-prod-conv
lemmas split = case-prod-conv

```

hide-const (open) prod

end

15 The Disjoint Sum of Two Types

```

theory Sum-Type
  imports Typedef Inductive Fun
begin

```

15.1 Construction of the sum type and its basic abstract operations

```

definition Inl-Rep :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  bool  $\Rightarrow$  bool
  where Inl-Rep a x y p  $\longleftrightarrow$  x = a  $\wedge$  p

```

```

definition Inr-Rep :: 'b  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  bool  $\Rightarrow$  bool
  where Inr-Rep b x y p  $\longleftrightarrow$  y = b  $\wedge$   $\neg$  p

```

```

definition sum = {f. ( $\exists$  a. f = Inl-Rep (a::'a))  $\vee$  ( $\exists$  b. f = Inr-Rep (b::'b))}

```

```

typedef ('a, 'b) sum (infixr + 10) = sum :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool  $\Rightarrow$  bool) set
  unfolding sum-def by auto

```

```

lemma Inl-RepI: Inl-Rep a  $\in$  sum
  by (auto simp add: sum-def)

```

```

lemma Inr-RepI: Inr-Rep b  $\in$  sum
  by (auto simp add: sum-def)

```

```

lemma inj-on-Abs-sum: A  $\subseteq$  sum  $\implies$  inj-on Abs-sum A

```

by (rule inj-on-inverseI, rule Abs-sum-inverse) auto

lemma *Inl-Rep-inject*: inj-on Inl-Rep A

proof (rule inj-onI)

show $\bigwedge a c. \text{Inl-Rep } a = \text{Inl-Rep } c \implies a = c$

by (auto simp add: Inl-Rep-def fun-eq-iff)

qed

lemma *Inr-Rep-inject*: inj-on Inr-Rep A

proof (rule inj-onI)

show $\bigwedge b d. \text{Inr-Rep } b = \text{Inr-Rep } d \implies b = d$

by (auto simp add: Inr-Rep-def fun-eq-iff)

qed

lemma *Inl-Rep-not-Inr-Rep*: Inl-Rep a \neq Inr-Rep b

by (auto simp add: Inl-Rep-def Inr-Rep-def fun-eq-iff)

definition *Inl* :: 'a \Rightarrow 'a + 'b

where *Inl* = Abs-sum \circ Inl-Rep

definition *Inr* :: 'b \Rightarrow 'a + 'b

where *Inr* = Abs-sum \circ Inr-Rep

lemma *inj-Inl* [simp]: inj-on Inl A

by (auto simp add: Inl-def intro!: comp-inj-on Inl-Rep-inject inj-on-Abs-sum Inl-RepI)

lemma *Inl-inject*: Inl x = Inl y \implies x = y

using inj-Inl by (rule injD)

lemma *inj-Inr* [simp]: inj-on Inr A

by (auto simp add: Inr-def intro!: comp-inj-on Inr-Rep-inject inj-on-Abs-sum Inr-RepI)

lemma *Inr-inject*: Inr x = Inr y \implies x = y

using inj-Inr by (rule injD)

lemma *Inl-not-Inr*: Inl a \neq Inr b

proof –

have {Inl-Rep a, Inr-Rep b} \subseteq sum

using Inl-RepI [of a] Inr-RepI [of b] by auto

with inj-on-Abs-sum have inj-on Abs-sum {Inl-Rep a, Inr-Rep b} .

with Inl-Rep-not-Inr-Rep [of a b] inj-on-contrad have Abs-sum (Inl-Rep a) \neq Abs-sum (Inr-Rep b)

by auto

then show ?thesis

by (simp add: Inl-def Inr-def)

qed

lemma *Inr-not-Inl*: $Inr\ b \neq Inl\ a$
using *Inl-not-Inr* **by** (*rule not-sym*)

lemma *sumE*:
assumes $\bigwedge x::'a. s = Inl\ x \implies P$
and $\bigwedge y::'b. s = Inr\ y \implies P$
shows P
proof (*rule Abs-sum-cases [of s]*)
fix f
assume $s = Abs\text{-}sum\ f$ **and** $f \in sum$
with *assms* **show** P
by (*auto simp add: sum-def Inl-def Inr-def*)
qed

free-constructors *case-sum* **for**
isl: Inl projl
| *Inr projr*
by (*erule sumE, assumption*) (*auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr*)

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

setup $\langle Sign.mandatory\text{-}path\ old \rangle$

old-rep-datatype *Inl Inr*
proof –
fix P
fix $s :: 'a + 'b$
assume $x: \bigwedge x::'a. P (Inl\ x)$ **and** $y: \bigwedge y::'b. P (Inr\ y)$
then show $P\ s$ **by** (*auto intro: sumE [of s]*)
qed (*auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr*)

setup $\langle Sign.parent\text{-}path \rangle$

But erase the prefix for properties that are not generated by *free-constructors*.

setup $\langle Sign.mandatory\text{-}path\ sum \rangle$

declare
old.sum.inject[*iff del*]
old.sum.distinct(1)[*simp del, induct-simp del*]

lemmas *induct* = *old.sum.induct*
lemmas *inducts* = *old.sum.inducts*
lemmas *rec* = *old.sum.rec*
lemmas *simps* = *sum.inject sum.distinct sum.case sum.rec*

setup $\langle Sign.parent\text{-}path \rangle$

primrec *map-sum* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a + 'b \Rightarrow 'c + 'd$
where
map-sum $f1\ f2\ (Inl\ a) = Inl\ (f1\ a)$

| $\text{map-sum } f1 \ f2 \ (\text{Inr } a) = \text{Inr } (f2 \ a)$

functor map-sum : map-sum

proof –

show $\text{map-sum } f \ g \ \circ \ \text{map-sum } h \ i = \text{map-sum } (f \ \circ \ h) \ (g \ \circ \ i)$ **for** $f \ g \ h \ i$

proof

show $(\text{map-sum } f \ g \ \circ \ \text{map-sum } h \ i) \ s = \text{map-sum } (f \ \circ \ h) \ (g \ \circ \ i) \ s$ **for** s

by $(\text{cases } s) \ \text{simp-all}$

qed

show $\text{map-sum } \text{id} \ \text{id} = \text{id}$

proof

show $\text{map-sum } \text{id} \ \text{id} \ s = \text{id} \ s$ **for** s

by $(\text{cases } s) \ \text{simp-all}$

qed

qed

lemma split-sum-all : $(\forall x. P \ x) \longleftrightarrow (\forall x. P \ (\text{Inl } x)) \wedge (\forall x. P \ (\text{Inr } x))$

by $(\text{auto intro: sum.induct})$

lemma split-sum-ex : $(\exists x. P \ x) \longleftrightarrow (\exists x. P \ (\text{Inl } x)) \vee (\exists x. P \ (\text{Inr } x))$

using split-sum-all [of $\lambda x. \neg P \ x$] **by** blast

15.2 Projections

lemma case-sum-KK [simp]: $\text{case-sum } (\lambda x. a) \ (\lambda x. a) = (\lambda x. a)$

by $(\text{rule ext}) \ (\text{simp split: sum.split})$

lemma surjective-sum : $\text{case-sum } (\lambda x::'a. f \ (\text{Inl } x)) \ (\lambda y::'b. f \ (\text{Inr } y)) = f$

proof

fix $s :: 'a + 'b$

show $(\text{case } s \ \text{of } \text{Inl } (x::'a) \Rightarrow f \ (\text{Inl } x) \mid \text{Inr } (y::'b) \Rightarrow f \ (\text{Inr } y)) = f \ s$

by $(\text{cases } s) \ \text{simp-all}$

qed

lemma case-sum-inject :

assumes a : $\text{case-sum } f1 \ f2 = \text{case-sum } g1 \ g2$

and r : $f1 = g1 \Longrightarrow f2 = g2 \Longrightarrow P$

shows P

proof $(\text{rule } r)$

show $f1 = g1$

proof

fix $x :: 'a$

from a **have** $\text{case-sum } f1 \ f2 \ (\text{Inl } x) = \text{case-sum } g1 \ g2 \ (\text{Inl } x)$ **by** simp

then show $f1 \ x = g1 \ x$ **by** simp

qed

show $f2 = g2$

proof

fix $y :: 'b$

from a **have** $\text{case-sum } f1 \ f2 \ (\text{Inr } y) = \text{case-sum } g1 \ g2 \ (\text{Inr } y)$ **by** simp

then show $f^2 y = g^2 y$ **by** *simp*
qed
qed

primrec *Suml* :: $('a \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$
where $Suml\ f\ (Inl\ x) = f\ x$

primrec *Sumr* :: $('b \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$
where $Sumr\ f\ (Inr\ x) = f\ x$

lemma *Suml-inject*:
assumes $Suml\ f = Suml\ g$
shows $f = g$
proof
fix $x :: 'a$
let $?s = Inl\ x :: 'a + 'b$
from *assms* **have** $Suml\ f\ ?s = Suml\ g\ ?s$ **by** *simp*
then show $f\ x = g\ x$ **by** *simp*
qed

lemma *Sumr-inject*:
assumes $Sumr\ f = Sumr\ g$
shows $f = g$
proof
fix $x :: 'b$
let $?s = Inr\ x :: 'a + 'b$
from *assms* **have** $Sumr\ f\ ?s = Sumr\ g\ ?s$ **by** *simp*
then show $f\ x = g\ x$ **by** *simp*
qed

15.3 The Disjoint Sum of Sets

definition *Plus* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a + 'b)\ set$ (**infixr** $<+>$ 65)
where $A <+> B = Inl\ 'A \cup Inr\ 'B$

hide-const (**open**) *Plus* — Valuable identifier

lemma *InlI* [*intro!*]: $a \in A \Longrightarrow Inl\ a \in A <+> B$
by (*simp add: Plus-def*)

lemma *InrI* [*intro!*]: $b \in B \Longrightarrow Inr\ b \in A <+> B$
by (*simp add: Plus-def*)

Exhaustion rule for sums, a degenerate form of induction

lemma *PlusE* [*elim!*]:
 $u \in A <+> B \Longrightarrow (\bigwedge x. x \in A \Longrightarrow u = Inl\ x \Longrightarrow P) \Longrightarrow (\bigwedge y. y \in B \Longrightarrow u = Inr\ y \Longrightarrow P) \Longrightarrow P$
by (*auto simp add: Plus-def*)

lemma *Plus-eq-empty-conv* [*simp*]: $A <+> B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
by *auto*

lemma *UNIV-Plus-UNIV* [*simp*]: $UNIV <+> UNIV = UNIV$
proof (*rule set-eqI*)
fix $u :: 'a + 'b$
show $u \in UNIV <+> UNIV \longleftrightarrow u \in UNIV$ **by** (*cases u*) *auto*
qed

lemma *UNIV-sum*: $UNIV = Inl \text{ ' } UNIV \cup Inr \text{ ' } UNIV$
proof –
have $x \in \text{range } Inl$ **if** $x \notin \text{range } Inr$ **for** $x :: 'a + 'b$
using *that* **by** (*cases x*) *simp-all*
then show *?thesis* **by** *auto*
qed

hide-const (**open**) *Suml Sumr sum*

end

16 Rings

theory *Rings*
imports *Groups Set Fun*
begin

16.1 Semirings and rings

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
assumes *distrib-right* [*algebra-simps, algebra-split-simps*]: $(a + b) * c = a * c + b * c$
assumes *distrib-left* [*algebra-simps, algebra-split-simps*]: $a * (b + c) = a * b + a * c$
begin

For the *combine-numerals* *simproc*

lemma *combine-common-factor*: $a * e + (b * e + c) = (a + b) * e + c$
by (*simp add: distrib-right ac-simps*)

end

class *mult-zero* = *times* + *zero* +
assumes *mult-zero-left* [*simp*]: $0 * a = 0$
assumes *mult-zero-right* [*simp*]: $a * 0 = 0$
begin

lemma *mult-not-zero*: $a * b \neq 0 \implies a \neq 0 \wedge b \neq 0$
by *auto*

end

class *semiring-0* = *semiring* + *comm-monoid-add* + *mult-zero*

class *semiring-0-cancel* = *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0*

proof

fix *a* :: 'a

have $0 * a + 0 * a = 0 * a + 0$

by (*simp add: distrib-right [symmetric]*)

then show $0 * a = 0$

by (*simp only: add-left-cancel*)

have $a * 0 + a * 0 = a * 0 + 0$

by (*simp add: distrib-left [symmetric]*)

then show $a * 0 = 0$

by (*simp only: add-left-cancel*)

qed

end

class *comm-semiring* = *ab-semigroup-add* + *ab-semigroup-mult* +
assumes *distrib*: $(a + b) * c = a * c + b * c$

begin

subclass *semiring*

proof

fix *a b c* :: 'a

show $(a + b) * c = a * c + b * c$

by (*simp add: distrib*)

have $a * (b + c) = (b + c) * a$

by (*simp add: ac-simps*)

also have $\dots = b * a + c * a$

by (*simp only: distrib*)

also have $\dots = a * b + a * c$

by (*simp add: ac-simps*)

finally show $a * (b + c) = a * b + a * c$

by *blast*

qed

end

class *comm-semiring-0* = *comm-semiring* + *comm-monoid-add* + *mult-zero*
begin

subclass *semiring-0* ..

end

```

class comm-semiring-0-cancel = comm-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..

subclass comm-semiring-0 ..

end

class zero-neq-one = zero + one +
  assumes zero-neq-one [simp]:  $0 \neq 1$ 
begin

lemma one-neq-zero [simp]:  $1 \neq 0$ 
  by (rule not-sym) (rule zero-neq-one)

definition of-bool :: bool  $\Rightarrow$  'a
  where of-bool p = (if p then 1 else 0)

lemma of-bool-eq [simp, code]:
  of-bool False = 0
  of-bool True = 1
  by (simp-all add: of-bool-def)

lemma of-bool-eq-iff: of-bool p = of-bool q  $\longleftrightarrow$  p = q
  by (simp add: of-bool-def)

lemma split-of-bool [split]:  $P$  (of-bool p)  $\longleftrightarrow$  (p  $\longrightarrow$   $P$  1)  $\wedge$  ( $\neg$  p  $\longrightarrow$   $P$  0)
  by (cases p) simp-all

lemma split-of-bool-asm:  $P$  (of-bool p)  $\longleftrightarrow$   $\neg$  (p  $\wedge$   $\neg$   $P$  1  $\vee$   $\neg$  p  $\wedge$   $\neg$   $P$  0)
  by (cases p) simp-all

lemma of-bool-eq-0-iff [simp]:
   $\langle$  of-bool P = 0  $\longleftrightarrow$   $\neg$  P  $\rangle$ 
  by simp

lemma of-bool-eq-1-iff [simp]:
   $\langle$  of-bool P = 1  $\longleftrightarrow$  P  $\rangle$ 
  by simp

end

class semiring-1 = zero-neq-one + semiring-0 + monoid-mult
begin

lemma of-bool-conj:
  of-bool (P  $\wedge$  Q) = of-bool P * of-bool Q

```

by *auto*

end

lemma *lambda-zero*: $(\lambda h::'a::mult-zero. 0) = (*) 0$
by *auto*

lemma *lambda-one*: $(\lambda x::'a::monoid-mult. x) = (*) 1$
by *auto*

16.2 Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** *dvd* 50)
where $b \text{ dvd } a \iff (\exists k. a = b * k)$

lemma *dvdI* [*intro?*]: $a = b * k \implies b \text{ dvd } a$
unfolding *dvd-def* ..

lemma *dvdE* [*elim*]: $b \text{ dvd } a \implies (\bigwedge k. a = b * k \implies P) \implies P$
unfolding *dvd-def* **by** *blast*

end

context *comm-monoid-mult*
begin

subclass *dvd* .

lemma *dvd-refl* [*simp*]: $a \text{ dvd } a$
proof
show $a = a * 1$ **by** *simp*
qed

lemma *dvd-trans* [*trans*]:
assumes $a \text{ dvd } b$ **and** $b \text{ dvd } c$
shows $a \text{ dvd } c$

proof –

from *assms* **obtain** v **where** $b = a * v$
by *auto*

moreover from *assms* **obtain** w **where** $c = b * w$
by *auto*

ultimately have $c = a * (v * w)$
by (*simp add: mult.assoc*)

then show *?thesis* ..

qed

lemma *subset-divisors-dvd*: $\{c. c \text{ dvd } a\} \subseteq \{c. c \text{ dvd } b\} \longleftrightarrow a \text{ dvd } b$
by (*auto simp add: subset-iff intro: dvd-trans*)

lemma *strict-subset-divisors-dvd*: $\{c. c \text{ dvd } a\} \subset \{c. c \text{ dvd } b\} \longleftrightarrow a \text{ dvd } b \wedge \neg b \text{ dvd } a$
by (*auto simp add: subset-iff intro: dvd-trans*)

lemma *one-dvd* [*simp*]: $1 \text{ dvd } a$
by (*auto intro: dvdI*)

lemma *dvd-mult* [*simp*]: $a \text{ dvd } (b * c)$ **if** $a \text{ dvd } c$
using *that* **by** (*auto intro: mult.left-commute dvdI*)

lemma *dvd-mult2* [*simp*]: $a \text{ dvd } (b * c)$ **if** $a \text{ dvd } b$
using *that* *dvd-mult* [*of a b c*] **by** (*simp add: ac-simps*)

lemma *dvd-triv-right* [*simp*]: $a \text{ dvd } b * a$
by (*rule dvd-mult*) (*rule dvd-refl*)

lemma *dvd-triv-left* [*simp*]: $a \text{ dvd } a * b$
by (*rule dvd-mult2*) (*rule dvd-refl*)

lemma *mult-dvd-mono*:

assumes $a \text{ dvd } b$

and $c \text{ dvd } d$

shows $a * c \text{ dvd } b * d$

proof –

from $\langle a \text{ dvd } b \rangle$ **obtain** b' **where** $b = a * b'$..

moreover from $\langle c \text{ dvd } d \rangle$ **obtain** d' **where** $d = c * d'$..

ultimately have $b * d = (a * c) * (b' * d')$

by (*simp add: ac-simps*)

then show *?thesis* ..

qed

lemma *dvd-mult-left*: $a * b \text{ dvd } c \implies a \text{ dvd } c$
by (*simp add: dvd-def mult.assoc*) *blast*

lemma *dvd-mult-right*: $a * b \text{ dvd } c \implies b \text{ dvd } c$
using *dvd-mult-left* [*of b a c*] **by** (*simp add: ac-simps*)

end

class *comm-semiring-1* = *zero-neg-one* + *comm-semiring-0* + *comm-monoid-mult*
begin

subclass *semiring-1* ..

lemma *dvd-0-left-iff* [*simp*]: $0 \text{ dvd } a \longleftrightarrow a = 0$
by *auto*

```

lemma dvd-0-right [iff]: a dvd 0
proof
  show  $0 = a * 0$  by simp
qed

lemma dvd-0-left:  $0 \text{ dvd } a \implies a = 0$ 
  by simp

lemma dvd-add [simp]:
  assumes a dvd b and a dvd c
  shows a dvd (b + c)
proof –
  from  $\langle a \text{ dvd } b \rangle$  obtain  $b'$  where  $b = a * b'$  ..
  moreover from  $\langle a \text{ dvd } c \rangle$  obtain  $c'$  where  $c = a * c'$  ..
  ultimately have  $b + c = a * (b' + c')$ 
    by (simp add: distrib-left)
  then show ?thesis ..
qed

end

class semiring-1-cancel = semiring + cancel-comm-monoid-add
  + zero-neq-one + monoid-mult
begin

subclass semiring-0-cancel ..

subclass semiring-1 ..

end

class comm-semiring-1-cancel =
  comm-semiring + cancel-comm-monoid-add + zero-neq-one + comm-monoid-mult
  +
  assumes right-diff-distrib' [algebra-simps, algebra-split-simps]:
     $a * (b - c) = a * b - a * c$ 
begin

subclass semiring-1-cancel ..
subclass comm-semiring-0-cancel ..
subclass comm-semiring-1 ..

lemma left-diff-distrib' [algebra-simps, algebra-split-simps]:
   $(b - c) * a = b * a - c * a$ 
  by (simp add: algebra-simps)

lemma dvd-add-times-triv-left-iff [simp]:  $a \text{ dvd } c * a + b \iff a \text{ dvd } b$ 
proof –

```



```

have a dvd a * c + b  $\longleftrightarrow$  a dvd b (is ?P  $\longleftrightarrow$  ?Q)
proof
  assume ?Q
  then show ?P by simp
next
  assume ?P
  then obtain d where a * c + b = a * d ..
  then have a * c + b - a * c = a * d - a * c by simp
  then have b = a * d - a * c by simp
  then have b = a * (d - c) by (simp add: algebra-simps)
  then show ?Q ..
qed
then show a dvd c * a + b  $\longleftrightarrow$  a dvd b by (simp add: ac-simps)
qed

lemma dvd-add-times-triv-right-iff [simp]: a dvd b + c * a  $\longleftrightarrow$  a dvd b
  using dvd-add-times-triv-left-iff [of a c b] by (simp add: ac-simps)

lemma dvd-add-triv-left-iff [simp]: a dvd a + b  $\longleftrightarrow$  a dvd b
  using dvd-add-times-triv-left-iff [of a 1 b] by simp

lemma dvd-add-triv-right-iff [simp]: a dvd b + a  $\longleftrightarrow$  a dvd b
  using dvd-add-times-triv-right-iff [of a b 1] by simp

lemma dvd-add-right-iff:
  assumes a dvd b
  shows a dvd b + c  $\longleftrightarrow$  a dvd c (is ?P  $\longleftrightarrow$  ?Q)
proof
  assume ?P
  then obtain d where b + c = a * d ..
  moreover from ⟨a dvd b⟩ obtain e where b = a * e ..
  ultimately have a * e + c = a * d by simp
  then have a * e + c - a * e = a * d - a * e by simp
  then have c = a * d - a * e by simp
  then have c = a * (d - e) by (simp add: algebra-simps)
  then show ?Q ..
next
  assume ?Q
  with assms show ?P by simp
qed

lemma dvd-add-left-iff: a dvd c  $\implies$  a dvd b + c  $\longleftrightarrow$  a dvd b
  using dvd-add-right-iff [of a c b] by (simp add: ac-simps)

end

class ring = semiring + ab-group-add
begin

```

subclass *semiring-0-cancel* ..

Distribution rules

lemma *minus-mult-left*: $-(a * b) = -a * b$
by (*rule minus-unique*) (*simp add: distrib-right [symmetric]*)

lemma *minus-mult-right*: $-(a * b) = a * -b$
by (*rule minus-unique*) (*simp add: distrib-left [symmetric]*)

Extract signs from products

lemmas *mult-minus-left* [*simp*] = *minus-mult-left* [*symmetric*]

lemmas *mult-minus-right* [*simp*] = *minus-mult-right* [*symmetric*]

lemma *minus-mult-minus* [*simp*]: $-a * -b = a * b$
by *simp*

lemma *minus-mult-commute*: $-a * b = a * -b$
by *simp*

lemma *right-diff-distrib* [*algebra-simps, algebra-split-simps*]:
 $a * (b - c) = a * b - a * c$
using *distrib-left* [*of a b -c*] **by** *simp*

lemma *left-diff-distrib* [*algebra-simps, algebra-split-simps*]:
 $(a - b) * c = a * c - b * c$
using *distrib-right* [*of a - b c*] **by** *simp*

lemmas *ring-distrib* = *distrib-left distrib-right left-diff-distrib right-diff-distrib*

lemma *eq-add-iff1*: $a * e + c = b * e + d \iff (a - b) * e + c = d$
by (*simp add: algebra-simps*)

lemma *eq-add-iff2*: $a * e + c = b * e + d \iff c = (b - a) * e + d$
by (*simp add: algebra-simps*)

end

lemmas *ring-distrib* = *distrib-left distrib-right left-diff-distrib right-diff-distrib*

class *comm-ring* = *comm-semiring + ab-group-add*
begin

subclass *ring* ..

subclass *comm-semiring-0-cancel* ..

lemma *square-diff-square-factored*: $x * x - y * y = (x + y) * (x - y)$
by (*simp add: algebra-simps*)

end

```

class ring-1 = ring + zero-neq-one + monoid-mult
begin

subclass semiring-1-cancel ..

lemma of-bool-not-iff:
  ⟨of-bool (¬ P) = 1 - of-bool P⟩
  by simp

lemma square-diff-one-factored:  $x * x - 1 = (x + 1) * (x - 1)$ 
  by (simp add: algebra-simps)

end

class comm-ring-1 = comm-ring + zero-neq-one + comm-monoid-mult
begin

subclass ring-1 ..
subclass comm-semiring-1-cancel
  by standard (simp add: algebra-simps)

lemma dvd-minus-iff [simp]:  $x \text{ dvd } - y \iff x \text{ dvd } y$ 
proof
  assume  $x \text{ dvd } - y$ 
  then have  $x \text{ dvd } - 1 * - y$  by (rule dvd-mult)
  then show  $x \text{ dvd } y$  by simp
next
  assume  $x \text{ dvd } y$ 
  then have  $x \text{ dvd } - 1 * y$  by (rule dvd-mult)
  then show  $x \text{ dvd } - y$  by simp
qed

lemma minus-dvd-iff [simp]:  $- x \text{ dvd } y \iff x \text{ dvd } y$ 
proof
  assume  $- x \text{ dvd } y$ 
  then obtain  $k$  where  $y = - x * k$  ..
  then have  $y = x * - k$  by simp
  then show  $x \text{ dvd } y$  ..
next
  assume  $x \text{ dvd } y$ 
  then obtain  $k$  where  $y = x * k$  ..
  then have  $y = - x * - k$  by simp
  then show  $- x \text{ dvd } y$  ..
qed

lemma dvd-diff [simp]:  $x \text{ dvd } y \implies x \text{ dvd } z \implies x \text{ dvd } (y - z)$ 
  using dvd-add [of  $x \ y - z$ ] by simp

```

end

16.3 Towards integral domains

```
class semiring-no-zero-divisors = semiring-0 +
  assumes no-zero-divisors:  $a \neq 0 \implies b \neq 0 \implies a * b \neq 0$ 
begin
```

```
lemma divisors-zero:
  assumes  $a * b = 0$ 
  shows  $a = 0 \vee b = 0$ 
proof (rule classical)
  assume  $\neg ?thesis$ 
  then have  $a \neq 0$  and  $b \neq 0$  by auto
  with no-zero-divisors have  $a * b \neq 0$  by blast
  with assms show ?thesis by simp
qed
```

```
lemma mult-eq-0-iff [simp]:  $a * b = 0 \iff a = 0 \vee b = 0$ 
proof (cases  $a = 0 \vee b = 0$ )
  case False
  then have  $a \neq 0$  and  $b \neq 0$  by auto
  then show ?thesis using no-zero-divisors by simp
next
  case True
  then show ?thesis by auto
qed
```

end

```
class semiring-1-no-zero-divisors = semiring-1 + semiring-no-zero-divisors
```

```
class semiring-no-zero-divisors-cancel = semiring-no-zero-divisors +
  assumes mult-cancel-right [simp]:  $a * c = b * c \iff c = 0 \vee a = b$ 
  and mult-cancel-left [simp]:  $c * a = c * b \iff c = 0 \vee a = b$ 
begin
```

```
lemma mult-left-cancel:  $c \neq 0 \implies c * a = c * b \iff a = b$ 
  by simp
```

```
lemma mult-right-cancel:  $c \neq 0 \implies a * c = b * c \iff a = b$ 
  by simp
```

end

```
class ring-no-zero-divisors = ring + semiring-no-zero-divisors
begin
```

```
subclass semiring-no-zero-divisors-cancel
```

```

proof
  fix  $a\ b\ c$ 
  have  $a * c = b * c \longleftrightarrow (a - b) * c = 0$ 
    by (simp add: algebra-simps)
  also have  $\dots \longleftrightarrow c = 0 \vee a = b$ 
    by auto
  finally show  $a * c = b * c \longleftrightarrow c = 0 \vee a = b$  .
  have  $c * a = c * b \longleftrightarrow c * (a - b) = 0$ 
    by (simp add: algebra-simps)
  also have  $\dots \longleftrightarrow c = 0 \vee a = b$ 
    by auto
  finally show  $c * a = c * b \longleftrightarrow c = 0 \vee a = b$  .
qed

end

class ring-1-no-zero-divisors = ring-1 + ring-no-zero-divisors
begin

subclass semiring-1-no-zero-divisors ..

lemma square-eq-1-iff:  $x * x = 1 \longleftrightarrow x = 1 \vee x = - 1$ 
proof -
  have  $(x - 1) * (x + 1) = x * x - 1$ 
    by (simp add: algebra-simps)
  then have  $x * x = 1 \longleftrightarrow (x - 1) * (x + 1) = 0$ 
    by simp
  then show ?thesis
    by (simp add: eq-neg-iff-add-eq-0)
qed

lemma mult-cancel-right1 [simp]:  $c = b * c \longleftrightarrow c = 0 \vee b = 1$ 
  using mult-cancel-right [of 1 c b] by auto

lemma mult-cancel-right2 [simp]:  $a * c = c \longleftrightarrow c = 0 \vee a = 1$ 
  using mult-cancel-right [of a c 1] by simp

lemma mult-cancel-left1 [simp]:  $c = c * b \longleftrightarrow c = 0 \vee b = 1$ 
  using mult-cancel-left [of c 1 b] by force

lemma mult-cancel-left2 [simp]:  $c * a = c \longleftrightarrow c = 0 \vee a = 1$ 
  using mult-cancel-left [of c a 1] by simp

end

class semidom = comm-semiring-1-cancel + semiring-no-zero-divisors
begin

subclass semiring-1-no-zero-divisors ..

```

end

class *idom* = *comm-ring-1* + *semiring-no-zero-divisors*
begin

subclass *semidom* ..

subclass *ring-1-no-zero-divisors* ..

lemma *dvd-mult-cancel-right* [*simp*]:

$a * c \text{ dvd } b * c \longleftrightarrow c = 0 \vee a \text{ dvd } b$

proof –

have $a * c \text{ dvd } b * c \longleftrightarrow (\exists k. b * c = (a * k) * c)$

by (*auto simp add: ac-simps*)

also have $(\exists k. b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \text{ dvd } b$

by *auto*

finally show *?thesis* .

qed

lemma *dvd-mult-cancel-left* [*simp*]:

$c * a \text{ dvd } c * b \longleftrightarrow c = 0 \vee a \text{ dvd } b$

using *dvd-mult-cancel-right* [*of a c b*] **by** (*simp add: ac-simps*)

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow a = b \vee a = - b$

proof

assume $a * a = b * b$

then have $(a - b) * (a + b) = 0$

by (*simp add: algebra-simps*)

then show $a = b \vee a = - b$

by (*simp add: eq-neg-iff-add-eq-0*)

next

assume $a = b \vee a = - b$

then show $a * a = b * b$ **by** *auto*

qed

lemma *inj-mult-left* [*simp*]: $\langle \text{inj } ((*) a) \longleftrightarrow a \neq 0 \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)

proof

assume *?P*

show *?Q*

proof

assume $\langle a = 0 \rangle$

with $\langle ?P \rangle$ **have** $\text{inj } ((*) 0)$

by *simp*

moreover have $0 * 0 = 0 * 1$

by *simp*

ultimately have $0 = 1$

by (*rule injD*)

then show *False*

```

    by simp
  qed
next
  assume ?Q then show ?P
    by (auto intro: injI)
  qed
end

class idom-abs-sgn = idom + abs + sgn +
  assumes sgn-mult-abs:  $sgn\ a * |a| = a$ 
  and sgn-sgn [simp]:  $sgn\ (sgn\ a) = sgn\ a$ 
  and abs-abs [simp]:  $||a|| = |a|$ 
  and abs-0 [simp]:  $|0| = 0$ 
  and sgn-0 [simp]:  $sgn\ 0 = 0$ 
  and sgn-1 [simp]:  $sgn\ 1 = 1$ 
  and sgn-minus-1:  $sgn\ (-\ 1) = -\ 1$ 
  and sgn-mult:  $sgn\ (a * b) = sgn\ a * sgn\ b$ 
begin

lemma sgn-eq-0-iff:
   $sgn\ a = 0 \longleftrightarrow a = 0$ 
proof -
  { assume  $sgn\ a = 0$ 
    then have  $sgn\ a * |a| = 0$ 
      by simp
    then have  $a = 0$ 
      by (simp add: sgn-mult-abs)
  } then show ?thesis
    by auto
qed

lemma abs-eq-0-iff:
   $|a| = 0 \longleftrightarrow a = 0$ 
proof -
  { assume  $|a| = 0$ 
    then have  $sgn\ a * |a| = 0$ 
      by simp
    then have  $a = 0$ 
      by (simp add: sgn-mult-abs)
  } then show ?thesis
    by auto
qed

lemma abs-mult-sgn:
   $|a| * sgn\ a = a$ 
  using sgn-mult-abs [of a] by (simp add: ac-simps)

lemma abs-1 [simp]:

```

$|1| = 1$
using *sgn-mult-abs* [of 1] **by** *simp*

lemma *sgn-abs* [*simp*]:
 $|sgn\ a| = of\ bool\ (a \neq 0)$
using *sgn-mult-abs* [of *sgn a*] *mult-cancel-left* [of *sgn a* | *sgn a* | 1]
by (*auto simp add: sgn-eq-0-iff*)

lemma *abs-sgn* [*simp*]:
 $sgn\ |a| = of\ bool\ (a \neq 0)$
using *sgn-mult-abs* [of |*a*|] *mult-cancel-right* [of *sgn* |*a*| |*a*| 1]
by (*auto simp add: abs-eq-0-iff*)

lemma *abs-mult*:
 $|a * b| = |a| * |b|$
proof (*cases a = 0 ∨ b = 0*)
case *True*
then show *?thesis*
by *auto*
next
case *False*
then have $*$: $sgn\ (a * b) \neq 0$
by (*simp add: sgn-eq-0-iff*)
from *abs-mult-sgn* [of $a * b$] *abs-mult-sgn* [of *a*] *abs-mult-sgn* [of *b*]
have $|a * b| * sgn\ (a * b) = |a| * sgn\ a * |b| * sgn\ b$
by (*simp add: ac-simps*)
then have $|a * b| * sgn\ (a * b) = |a| * |b| * sgn\ (a * b)$
by (*simp add: sgn-mult ac-simps*)
with $*$ **show** *?thesis*
by *simp*
qed

lemma *sgn-minus* [*simp*]:
 $sgn\ (-\ a) = -\ sgn\ a$
proof –
from *sgn-minus-1* **have** $sgn\ (-\ 1 * a) = -\ 1 * sgn\ a$
by (*simp only: sgn-mult*)
then show *?thesis*
by *simp*
qed

lemma *abs-minus* [*simp*]:
 $|-\ a| = |a|$
proof –
have [*simp*]: $|-\ 1| = 1$
using *sgn-mult-abs* [of $- 1$] **by** *simp*
then have $|-\ 1 * a| = 1 * |a|$
by (*simp only: abs-mult*)
then show *?thesis*


```

    by simp
  qed

```

```

end

```

16.4 (Partial) Division

```

class divide =

```

```

  fixes divide :: 'a ⇒ 'a ⇒ 'a (infixl div 70)

```

```

  setup ⟨Sign.add-const-constraint (const-name ⟨divide⟩, SOME typ ⟨'a ⇒ 'a ⇒ 'a⟩)⟩

```

```

  context semiring

```

```

  begin

```

```

  lemma [field-simps, field-split-simps]:

```

```

    shows distrib-left-NO-MATCH: NO-MATCH (x div y) a ⇒ a * (b + c) = a *
    b + a * c

```

```

    and distrib-right-NO-MATCH: NO-MATCH (x div y) c ⇒ (a + b) * c = a
    * c + b * c

```

```

    by (rule distrib-left distrib-right)+

```

```

  end

```

```

  context ring

```

```

  begin

```

```

  lemma [field-simps, field-split-simps]:

```

```

    shows left-diff-distrib-NO-MATCH: NO-MATCH (x div y) c ⇒ (a - b) * c =
    a * c - b * c

```

```

    and right-diff-distrib-NO-MATCH: NO-MATCH (x div y) a ⇒ a * (b - c)
    = a * b - a * c

```

```

    by (rule left-diff-distrib right-diff-distrib)+

```

```

  end

```

```

  setup ⟨Sign.add-const-constraint (const-name ⟨divide⟩, SOME typ ⟨'a::divide ⇒ 'a ⇒ 'a⟩)⟩

```

Algebraic classes with division

```

class semidom-divide = semidom + divide +

```

```

  assumes nonzero-mult-div-cancel-right [simp]: b ≠ 0 ⇒ (a * b) div b = a

```

```

  assumes div-by-0 [simp]: a div 0 = 0

```

```

begin

```

```

lemma nonzero-mult-div-cancel-left [simp]: a ≠ 0 ⇒ (a * b) div a = b

```

```

  using nonzero-mult-div-cancel-right [of a b] by (simp add: ac-simps)

```

```

subclass semiring-no-zero-divisors-cancel
proof
  show *:  $a * c = b * c \longleftrightarrow c = 0 \vee a = b$  for  $a\ b\ c$ 
  proof (cases  $c = 0$ )
    case True
      then show ?thesis by simp
    next
      case False
      have  $a = b$  if  $a * c = b * c$ 
      proof -
        from that have  $a * c\ \text{div}\ c = b * c\ \text{div}\ c$ 
        by simp
        with False show ?thesis
        by simp
      qed
      then show ?thesis by auto
    qed
  show  $c * a = c * b \longleftrightarrow c = 0 \vee a = b$  for  $a\ b\ c$ 
  using * [of  $a\ c\ b$ ] by (simp add: ac-simps)
qed

lemma div-self [simp]:  $a \neq 0 \implies a\ \text{div}\ a = 1$ 
  using nonzero-mult-div-cancel-left [of  $a\ 1$ ] by simp

lemma div-0 [simp]:  $0\ \text{div}\ a = 0$ 
proof (cases  $a = 0$ )
  case True
    then show ?thesis by simp
  next
    case False
    then have  $a * 0\ \text{div}\ a = 0$ 
    by (rule nonzero-mult-div-cancel-left)
    then show ?thesis by simp
qed

lemma div-by-1 [simp]:  $a\ \text{div}\ 1 = a$ 
  using nonzero-mult-div-cancel-left [of  $1\ a$ ] by simp

lemma dvd-div-eq-0-iff:
  assumes  $b\ \text{dvd}\ a$ 
  shows  $a\ \text{div}\ b = 0 \longleftrightarrow a = 0$ 
  using assms by (elim dvdE, cases  $b = 0$ ) simp-all

lemma dvd-div-eq-cancel:
   $a\ \text{div}\ c = b\ \text{div}\ c \implies c\ \text{dvd}\ a \implies c\ \text{dvd}\ b \implies a = b$ 
  by (elim dvdE, cases  $c = 0$ ) simp-all

lemma dvd-div-eq-iff:
   $c\ \text{dvd}\ a \implies c\ \text{dvd}\ b \implies a\ \text{div}\ c = b\ \text{div}\ c \longleftrightarrow a = b$ 

```

by (*elim dvdE*, *cases c = 0*) *simp-all*

lemma *inj-on-mult*:

inj-on $((*) a) A$ if $a \neq 0$

proof (*rule inj-onI*)

fix $b c$

assume $a * b = a * c$

then have $a * b \text{ div } a = a * c \text{ div } a$

by (*simp only*.)

with that show $b = c$

by *simp*

qed

end

class *idom-divide* = *idom* + *semidom-divide*

begin

lemma *dvd-neg-div*:

assumes $b \text{ dvd } a$

shows $- a \text{ div } b = - (a \text{ div } b)$

proof (*cases b = 0*)

case *True*

then show *?thesis* by *simp*

next

case *False*

from *assms* **obtain** c **where** $a = b * c$..

then have $- a \text{ div } b = (b * - c) \text{ div } b$

by *simp*

from *False* **also have** $\dots = - c$

by (*rule nonzero-mult-div-cancel-left*)

with *False* $\langle a = b * c \rangle$ **show** *?thesis*

by *simp*

qed

lemma *dvd-div-neg*:

assumes $b \text{ dvd } a$

shows $a \text{ div } - b = - (a \text{ div } b)$

proof (*cases b = 0*)

case *True*

then show *?thesis* by *simp*

next

case *False*

then have $- b \neq 0$

by *simp*

from *assms* **obtain** c **where** $a = b * c$..

then have $a \text{ div } - b = (- b * - c) \text{ div } - b$

by *simp*

from $\langle - b \neq 0 \rangle$ **also have** $\dots = - c$

```

    by (rule nonzero-mult-div-cancel-left)
  with False  $\langle a = b * c \rangle$  show ?thesis
    by simp
qed

```

```
end
```

```
class algebraic-semidom = semidom-divide
begin

```

Class *algebraic-semidom* enriches a integral domain by notions from algebra, like units in a ring. It is a separate class to avoid spoiling fields with notions which are degenerated there.

```
lemma dvd-times-left-cancel-iff [simp]:
```

```

  assumes  $a \neq 0$ 
  shows  $a * b \text{ dvd } a * c \iff b \text{ dvd } c$ 
    (is ?lhs  $\iff$  ?rhs)

```

```
proof
```

```

  assume ?lhs
  then obtain d where  $a * c = a * b * d$  ..
  with assms have  $c = b * d$  by (simp add: ac-simps)
  then show ?rhs ..

```

```
next
```

```

  assume ?rhs
  then obtain d where  $c = b * d$  ..
  then have  $a * c = a * b * d$  by (simp add: ac-simps)
  then show ?lhs ..

```

```
qed
```

```
lemma dvd-times-right-cancel-iff [simp]:
```

```

  assumes  $a \neq 0$ 
  shows  $b * a \text{ dvd } c * a \iff b \text{ dvd } c$ 
  using dvd-times-left-cancel-iff [of a b c] assms by (simp add: ac-simps)

```

```
lemma div-dvd-iff-mult:
```

```

  assumes  $b \neq 0$  and  $b \text{ dvd } a$ 
  shows  $a \text{ div } b \text{ dvd } c \iff a \text{ dvd } c * b$ 

```

```
proof -
```

```

  from  $\langle b \text{ dvd } a \rangle$  obtain d where  $a = b * d$  ..
  with  $\langle b \neq 0 \rangle$  show ?thesis by (simp add: ac-simps)

```

```
qed
```

```
lemma dvd-div-iff-mult:
```

```

  assumes  $c \neq 0$  and  $c \text{ dvd } b$ 
  shows  $a \text{ dvd } b \text{ div } c \iff a * c \text{ dvd } b$ 

```

```
proof -
```

```

  from  $\langle c \text{ dvd } b \rangle$  obtain d where  $b = c * d$  ..
  with  $\langle c \neq 0 \rangle$  show ?thesis by (simp add: mult.commute [of a])

```

```
qed
```

lemma *div-dvd-div* [*simp*]:
 assumes $a \text{ dvd } b$ and $a \text{ dvd } c$
 shows $b \text{ div } a \text{ dvd } c \text{ div } a \longleftrightarrow b \text{ dvd } c$
proof (*cases* $a = 0$)
 case *True*
 with *assms* show *?thesis* by *simp*
next
 case *False*
 moreover from *assms* obtain $k \ l$ where $b = a * k$ and $c = a * l$
 by *blast*
 ultimately show *?thesis* by *simp*
qed

lemma *div-add* [*simp*]:
 assumes $c \text{ dvd } a$ and $c \text{ dvd } b$
 shows $(a + b) \text{ div } c = a \text{ div } c + b \text{ div } c$
proof (*cases* $c = 0$)
 case *True*
 then show *?thesis* by *simp*
next
 case *False*
 moreover from *assms* obtain $k \ l$ where $a = c * k$ and $b = c * l$
 by *blast*
 moreover have $c * k + c * l = c * (k + l)$
 by (*simp add: algebra-simps*)
 ultimately show *?thesis*
 by *simp*
qed

lemma *div-mult-div-if-dvd*:
 assumes $b \text{ dvd } a$ and $d \text{ dvd } c$
 shows $(a \text{ div } b) * (c \text{ div } d) = (a * c) \text{ div } (b * d)$
proof (*cases* $b = 0 \vee c = 0$)
 case *True*
 with *assms* show *?thesis* by *auto*
next
 case *False*
 moreover from *assms* obtain $k \ l$ where $a = b * k$ and $c = d * l$
 by *blast*
 moreover have $b * k * (d * l) \text{ div } (b * d) = (b * d) * (k * l) \text{ div } (b * d)$
 by (*simp add: ac-simps*)
 ultimately show *?thesis* by *simp*
qed

lemma *dvd-div-eq-mult*:
 assumes $a \neq 0$ and $a \text{ dvd } b$
 shows $b \text{ div } a = c \longleftrightarrow b = c * a$
 (*is ?lhs* \longleftrightarrow *?rhs*)

proof
 assume $?rhs$
 then show $?lhs$ by (simp add: *assms*)
next
 assume $?lhs$
 then have $b \text{ div } a * a = c * a$ by *simp*
 moreover from *assms* have $b \text{ div } a * a = b$
 by (auto simp add: *ac-simps*)
 ultimately show $?rhs$ by *simp*
qed

lemma *dvd-div-mult-self* [*simp*]: $a \text{ dvd } b \implies b \text{ div } a * a = b$
 by (cases $a = 0$) (auto simp add: *ac-simps*)

lemma *dvd-mult-div-cancel* [*simp*]: $a \text{ dvd } b \implies a * (b \text{ div } a) = b$
 using *dvd-div-mult-self* [of $a \ b$] by (simp add: *ac-simps*)

lemma *div-mult-swap*:
 assumes $c \text{ dvd } b$
 shows $a * (b \text{ div } c) = (a * b) \text{ div } c$
proof (cases $c = 0$)
 case *True*
 then show $?thesis$ by *simp*
next
 case *False*
 from *assms* obtain d where $b = c * d$..
 moreover from *False* have $a * \text{divide } (d * c) \ c = ((a * d) * c) \text{ div } c$
 by *simp*
 ultimately show $?thesis$ by (simp add: *ac-simps*)
qed

lemma *dvd-div-mult*: $c \text{ dvd } b \implies b \text{ div } c * a = (b * a) \text{ div } c$
 using *div-mult-swap* [of $c \ b \ a$] by (simp add: *ac-simps*)

lemma *dvd-div-mult2-eq*:
 assumes $b * c \text{ dvd } a$
 shows $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$
proof –
 from *assms* obtain k where $a = b * c * k$..
 then show $?thesis$
 by (cases $b = 0 \vee c = 0$) (auto, simp add: *ac-simps*)
qed

lemma *dvd-div-div-eq-mult*:
 assumes $a \neq 0 \ c \neq 0$ and $a \text{ dvd } b \ c \text{ dvd } d$
 shows $b \text{ div } a = d \text{ div } c \iff b * c = a * d$
 (is $?lhs \iff ?rhs$)
proof –
 from *assms* have $a * c \neq 0$ by *simp*

then have $?lhs \longleftrightarrow b \text{ div } a * (a * c) = d \text{ div } c * (a * c)$
by *simp*
also have $\dots \longleftrightarrow (a * (b \text{ div } a)) * c = (c * (d \text{ div } c)) * a$
by (*simp add: ac-simps*)
also have $\dots \longleftrightarrow (a * b \text{ div } a) * c = (c * d \text{ div } c) * a$
using *assms* **by** (*simp add: div-mult-swap*)
also have $\dots \longleftrightarrow ?rhs$
using *assms* **by** (*simp add: ac-simps*)
finally show *?thesis* .
qed

lemma *dvd-mult-imp-div*:
assumes $a * c \text{ dvd } b$
shows $a \text{ dvd } b \text{ div } c$
proof (*cases c = 0*)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
from $\langle a * c \text{ dvd } b \rangle$ **obtain** d **where** $b = a * c * d$..
with *False* **show** *?thesis*
by (*simp add: mult.commute [of a] mult.assoc*)
qed

lemma *div-div-eq-right*:
assumes $c \text{ dvd } b \text{ } b \text{ dvd } a$
shows $a \text{ div } (b \text{ div } c) = a \text{ div } b * c$
proof (*cases c = 0 \vee b = 0*)
case *True*
then show *?thesis*
by *auto*
next
case *False*
from *assms* **obtain** $r \ s$ **where** $b = c * r$ **and** $a = c * r * s$
by *blast*
moreover with *False* **have** $r \neq 0$
by *auto*
ultimately show *?thesis* **using** *False*
by *simp (simp add: mult.commute [of - r] mult.assoc mult.commute [of c])*
qed

lemma *div-div-div-same*:
assumes $d \text{ dvd } b \text{ } b \text{ dvd } a$
shows $(a \text{ div } d) \text{ div } (b \text{ div } d) = a \text{ div } b$
proof (*cases b = 0 \vee d = 0*)
case *True*
with *assms* **show** *?thesis*
by *auto*
next
case *False*

from *assms* **obtain** $r\ s$
where $a = d * r * s$ **and** $b = d * r$
by *blast*
with *False* **show** *?thesis*
by *simp* (*simp add: ac-simps*)
qed

Units: invertible elements in a ring

abbreviation *is-unit* :: ' $a \Rightarrow bool$
where *is-unit* $a \equiv a\ dvd\ 1$

lemma *not-is-unit-0* [*simp*]: $\neg\ is\ unit\ 0$
by *simp*

lemma *unit-imp-dvd* [*dest*]: $is\ unit\ b \Longrightarrow b\ dvd\ a$
by (*rule dvd-trans [of - 1]*) *simp-all*

lemma *unit-dvdE*:
assumes *is-unit* a
obtains c **where** $a \neq 0$ **and** $b = a * c$
proof –
from *assms* **have** $a\ dvd\ b$ **by** *auto*
then **obtain** c **where** $b = a * c$..
moreover **from** *assms* **have** $a \neq 0$ **by** *auto*
ultimately **show** *thesis* **using** *that* **by** *blast*
qed

lemma *dvd-unit-imp-unit*: $a\ dvd\ b \Longrightarrow is\ unit\ b \Longrightarrow is\ unit\ a$
by (*rule dvd-trans*)

lemma *unit-div-1-unit* [*simp, intro*]:
assumes *is-unit* a
shows *is-unit* ($1\ div\ a$)
proof –
from *assms* **have** $1 = 1\ div\ a * a$ **by** *simp*
then **show** *is-unit* ($1\ div\ a$) **by** (*rule dvdI*)
qed

lemma *is-unitE* [*elim?*]:
assumes *is-unit* a
obtains b **where** $a \neq 0$ **and** $b \neq 0$
and *is-unit* b **and** $1\ div\ a = b$ **and** $1\ div\ b = a$
and $a * b = 1$ **and** $c\ div\ a = c * b$
proof (*rule that*)
define b **where** $b = 1\ div\ a$
then **show** $1\ div\ a = b$ **by** *simp*
from *assms* *b-def* **show** *is-unit* b **by** *simp*
with *assms* **show** $a \neq 0$ **and** $b \neq 0$ **by** *auto*
from *assms* *b-def* **show** $a * b = 1$ **by** *simp*

then have $1 = a * b$..
with $b\text{-def } \langle b \neq 0 \rangle$ **show** $1 \text{ div } b = a$ **by** *simp*
from *assms* **have** $a \text{ dvd } c$..
then obtain d **where** $c = a * d$..
with $\langle a \neq 0 \rangle \langle a * b = 1 \rangle$ **show** $c \text{ div } a = c * b$
by (*simp add: mult.assoc mult.left-commute [of a]*)
qed

lemma *unit-prod* [*intro*]: $is\text{-unit } a \implies is\text{-unit } b \implies is\text{-unit } (a * b)$
by (*subst mult-1-left [of 1, symmetric]*) (*rule mult-dvd-mono*)

lemma *is-unit-mult-iff*: $is\text{-unit } (a * b) \longleftrightarrow is\text{-unit } a \wedge is\text{-unit } b$
by (*auto dest: dvd-mult-left dvd-mult-right*)

lemma *unit-div* [*intro*]: $is\text{-unit } a \implies is\text{-unit } b \implies is\text{-unit } (a \text{ div } b)$
by (*erule is-unitE [of b a]*) (*simp add: ac-simps unit-prod*)

lemma *mult-unit-dvd-iff*:
assumes $is\text{-unit } b$
shows $a * b \text{ dvd } c \longleftrightarrow a \text{ dvd } c$
proof
assume $a * b \text{ dvd } c$
with *assms* **show** $a \text{ dvd } c$
by (*simp add: dvd-mult-left*)
next
assume $a \text{ dvd } c$
then obtain k **where** $c = a * k$..
with *assms* **have** $c = (a * b) * (1 \text{ div } b * k)$
by (*simp add: mult-ac*)
then show $a * b \text{ dvd } c$ **by** (*rule dvdI*)
qed

lemma *mult-unit-dvd-iff'*: $is\text{-unit } a \implies (a * b) \text{ dvd } c \longleftrightarrow b \text{ dvd } c$
using *mult-unit-dvd-iff [of a b c]* **by** (*simp add: ac-simps*)

lemma *dvd-mult-unit-iff*:
assumes $is\text{-unit } b$
shows $a \text{ dvd } c * b \longleftrightarrow a \text{ dvd } c$
proof
assume $a \text{ dvd } c * b$
with *assms* **have** $c * b \text{ dvd } c * (b * (1 \text{ div } b))$
by (*subst mult-assoc [symmetric]*) *simp*
also from *assms* **have** $b * (1 \text{ div } b) = 1$
by (*rule is-unitE*) *simp*
finally have $c * b \text{ dvd } c$ **by** *simp*
with $\langle a \text{ dvd } c * b \rangle$ **show** $a \text{ dvd } c$ **by** (*rule dvd-trans*)
next
assume $a \text{ dvd } c$
then show $a \text{ dvd } c * b$ **by** *simp*

qed

lemma *dvd-mult-unit-iff'*: $is\text{-unit } b \implies a \text{ dvd } b * c \longleftrightarrow a \text{ dvd } c$
using *dvd-mult-unit-iff* [of $b \ a \ c$] **by** (*simp add: ac-simps*)

lemma *div-unit-dvd-iff*: $is\text{-unit } b \implies a \text{ div } b \text{ dvd } c \longleftrightarrow a \text{ dvd } c$
by (*erule is-unitE* [of $- \ a$]) (*auto simp add: mult-unit-dvd-iff*)

lemma *dvd-div-unit-iff*: $is\text{-unit } b \implies a \text{ dvd } c \text{ div } b \longleftrightarrow a \text{ dvd } c$
by (*erule is-unitE* [of $- \ c$]) (*simp add: dvd-mult-unit-iff*)

lemmas *unit-dvd-iff = mult-unit-dvd-iff mult-unit-dvd-iff'*
dvd-mult-unit-iff dvd-mult-unit-iff'
div-unit-dvd-iff dvd-div-unit-iff

lemma *unit-mult-div-div* [*simp*]: $is\text{-unit } a \implies b * (1 \text{ div } a) = b \text{ div } a$
by (*erule is-unitE* [of $- \ b$]) *simp*

lemma *unit-div-mult-self* [*simp*]: $is\text{-unit } a \implies b \text{ div } a * a = b$
by (*rule dvd-div-mult-self*) *auto*

lemma *unit-div-1-div-1* [*simp*]: $is\text{-unit } a \implies 1 \text{ div } (1 \text{ div } a) = a$
by (*erule is-unitE*) *simp*

lemma *unit-div-mult-swap*: $is\text{-unit } c \implies a * (b \text{ div } c) = (a * b) \text{ div } c$
by (*erule unit-dvdE* [of $- \ b$]) (*simp add: mult.left-commute* [of $- \ c$])

lemma *unit-div-commute*: $is\text{-unit } b \implies (a \text{ div } b) * c = (a * c) \text{ div } b$
using *unit-div-mult-swap* [of $b \ c \ a$] **by** (*simp add: ac-simps*)

lemma *unit-eq-div1*: $is\text{-unit } b \implies a \text{ div } b = c \longleftrightarrow a = c * b$
by (*auto elim: is-unitE*)

lemma *unit-eq-div2*: $is\text{-unit } b \implies a = c \text{ div } b \longleftrightarrow a * b = c$
using *unit-eq-div1* [of $b \ c \ a$] **by** *auto*

lemma *unit-mult-left-cancel*: $is\text{-unit } a \implies a * b = a * c \longleftrightarrow b = c$
using *mult-cancel-left* [of $a \ b \ c$] **by** *auto*

lemma *unit-mult-right-cancel*: $is\text{-unit } a \implies b * a = c * a \longleftrightarrow b = c$
using *unit-mult-left-cancel* [of $a \ b \ c$] **by** (*auto simp add: ac-simps*)

lemma *unit-div-cancel*:

assumes *is-unit a*

shows $b \text{ div } a = c \text{ div } a \longleftrightarrow b = c$

proof –

from *assms* **have** $is\text{-unit } (1 \text{ div } a)$ **by** *simp*

then **have** $b * (1 \text{ div } a) = c * (1 \text{ div } a) \longleftrightarrow b = c$

by (*rule unit-mult-right-cancel*)

with *assms* **show** *?thesis* **by** *simp*
qed

lemma *is-unit-div-mult2-eq*:
assumes *is-unit b* **and** *is-unit c*
shows $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$
proof –
from *assms* **have** *is-unit (b * c)*
by (*simp add: unit-prod*)
then have $b * c \text{ dvd } a$
by (*rule unit-imp-dvd*)
then show *?thesis*
by (*rule dvd-div-mult2-eq*)
qed

lemma *is-unit-div-mult-cancel-left*:
assumes $a \neq 0$ **and** *is-unit b*
shows $a \text{ div } (a * b) = 1 \text{ div } b$
proof –
from *assms* **have** $a \text{ div } (a * b) = a \text{ div } a \text{ div } b$
by (*simp add: mult-unit-dvd-iff dvd-div-mult2-eq*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *is-unit-div-mult-cancel-right*:
assumes $a \neq 0$ **and** *is-unit b*
shows $a \text{ div } (b * a) = 1 \text{ div } b$
using *assms is-unit-div-mult-cancel-left [of a b]* **by** (*simp add: ac-simps*)

lemma *unit-div-eq-0-iff*:
assumes *is-unit b*
shows $a \text{ div } b = 0 \iff a = 0$
using *assms* **by** (*simp add: dvd-div-eq-0-iff unit-imp-dvd*)

lemma *div-mult-unit2*:
 $is-unit c \implies b \text{ dvd } a \implies a \text{ div } (b * c) = a \text{ div } b \text{ div } c$
by (*rule dvd-div-mult2-eq*) (*simp-all add: mult-unit-dvd-iff*)

Coprimality

definition *coprime* :: $'a \Rightarrow 'a \Rightarrow bool$
where $coprime\ a\ b \iff (\forall c. c \text{ dvd } a \longrightarrow c \text{ dvd } b \longrightarrow is-unit\ c)$

lemma *coprimeI*:
assumes $\bigwedge c. c \text{ dvd } a \implies c \text{ dvd } b \implies is-unit\ c$
shows $coprime\ a\ b$
using *assms* **by** (*auto simp: coprime-def*)

lemma *not-coprimeI*:
assumes $c \text{ dvd } a$ **and** $c \text{ dvd } b$ **and** $\neg is-unit\ c$

shows $\neg \text{coprime } a \ b$
using *assms* **by** (*auto simp: coprime-def*)

lemma *coprime-common-divisor*:
is-unit c **if** *coprime* $a \ b$ **and** $c \ \text{dvd} \ a$ **and** $c \ \text{dvd} \ b$
using *that* **by** (*auto simp: coprime-def*)

lemma *not-coprimeE*:
assumes $\neg \text{coprime } a \ b$
obtains c **where** $c \ \text{dvd} \ a$ **and** $c \ \text{dvd} \ b$ **and** $\neg \text{is-unit } c$
using *assms* **by** (*auto simp: coprime-def*)

lemma *coprime-imp-coprime*:
coprime $a \ b$ **if** *coprime* $c \ d$
and $\bigwedge e. \neg \text{is-unit } e \implies e \ \text{dvd} \ a \implies e \ \text{dvd} \ b \implies e \ \text{dvd} \ c$
and $\bigwedge e. \neg \text{is-unit } e \implies e \ \text{dvd} \ a \implies e \ \text{dvd} \ b \implies e \ \text{dvd} \ d$
proof (*rule coprimeI*)
fix e
assume $e \ \text{dvd} \ a$ **and** $e \ \text{dvd} \ b$
with *that* **have** $e \ \text{dvd} \ c$ **and** $e \ \text{dvd} \ d$
by (*auto intro: dvd-trans*)
with $\langle \text{coprime } c \ d \rangle$ **show** *is-unit* e
by (*rule coprime-common-divisor*)
qed

lemma *coprime-divisors*:
coprime $a \ b$ **if** $a \ \text{dvd} \ c \ b \ \text{dvd} \ d$ **and** *coprime* $c \ d$
using $\langle \text{coprime } c \ d \rangle$ **proof** (*rule coprime-imp-coprime*)
fix e
assume $e \ \text{dvd} \ a$ **then show** $e \ \text{dvd} \ c$
using $\langle a \ \text{dvd} \ c \rangle$ **by** (*rule dvd-trans*)
assume $e \ \text{dvd} \ b$ **then show** $e \ \text{dvd} \ d$
using $\langle b \ \text{dvd} \ d \rangle$ **by** (*rule dvd-trans*)
qed

lemma *coprime-self* [*simp*]:
coprime $a \ a \longleftrightarrow \text{is-unit } a$ (**is** $?P \longleftrightarrow ?Q$)
proof
assume $?P$
then show $?Q$
by (*rule coprime-common-divisor*) *simp-all*
next
assume $?Q$
show $?P$
by (*rule coprimeI*) (*erule dvd-unit-imp-unit, rule* $\langle ?Q \rangle$)
qed

lemma *coprime-commute* [*ac-simps*]:
coprime $b \ a \longleftrightarrow \text{coprime } a \ b$

unfolding *coprime-def* **by** *auto*

lemma *is-unit-left-imp-coprime*:

coprime a b **if** *is-unit a*

proof (*rule coprimeI*)

fix *c*

assume *c dvd a*

with that show *is-unit c*

by (*auto intro: dvd-unit-imp-unit*)

qed

lemma *is-unit-right-imp-coprime*:

coprime a b **if** *is-unit b*

using *that is-unit-left-imp-coprime [of b a]* **by** (*simp add: ac-simps*)

lemma *coprime-1-left [simp]*:

coprime 1 a

by (*rule coprimeI*)

lemma *coprime-1-right [simp]*:

coprime a 1

by (*rule coprimeI*)

lemma *coprime-0-left-iff [simp]*:

coprime 0 a \longleftrightarrow *is-unit a*

by (*auto intro: coprimeI dvd-unit-imp-unit coprime-common-divisor [of 0 a a]*)

lemma *coprime-0-right-iff [simp]*:

coprime a 0 \longleftrightarrow *is-unit a*

using *coprime-0-left-iff [of a]* **by** (*simp add: ac-simps*)

lemma *coprime-mult-self-left-iff [simp]*:

*coprime (c * a) (c * b)* \longleftrightarrow *is-unit c* \wedge *coprime a b*

by (*auto intro: coprime-common-divisor*)

(*rule coprimeI, auto intro: coprime-common-divisor simp add: dvd-mult-unit-iff'*)**+**

lemma *coprime-mult-self-right-iff [simp]*:

*coprime (a * c) (b * c)* \longleftrightarrow *is-unit c* \wedge *coprime a b*

using *coprime-mult-self-left-iff [of c a b]* **by** (*simp add: ac-simps*)

lemma *coprime-absorb-left*:

assumes *x dvd y*

shows *coprime x y* \longleftrightarrow *is-unit x*

using *assms coprime-common-divisor is-unit-left-imp-coprime* **by** *auto*

lemma *coprime-absorb-right*:

assumes *y dvd x*

shows *coprime x y* \longleftrightarrow *is-unit y*

using *assms coprime-common-divisor is-unit-right-imp-coprime* **by** *auto*

end

class *unit-factor* =
fixes *unit-factor* :: 'a \Rightarrow 'a

class *semidom-divide-unit-factor* = *semidom-divide* + *unit-factor* +
assumes *unit-factor-0* [*simp*]: *unit-factor* 0 = 0
and *is-unit-unit-factor*: a dvd 1 \implies *unit-factor* a = a
and *unit-factor-is-unit*: a \neq 0 \implies *unit-factor* a dvd 1
and *unit-factor-mult-unit-left*: a dvd 1 \implies *unit-factor* (a * b) = a * *unit-factor* b

— This fine-grained hierarchy will later on allow lean normalization of polynomials

begin

lemma *unit-factor-mult-unit-right*: a dvd 1 \implies *unit-factor* (b * a) = *unit-factor* b * a

using *unit-factor-mult-unit-left*[of a b] **by** (*simp* add: *mult-ac*)

lemmas [*simp*] = *unit-factor-mult-unit-left* *unit-factor-mult-unit-right*

end

class *normalization-semidom* = *algebraic-semidom* + *semidom-divide-unit-factor* +

fixes *normalize* :: 'a \Rightarrow 'a

assumes *unit-factor-mult-normalize* [*simp*]: *unit-factor* a * *normalize* a = a

and *normalize-0* [*simp*]: *normalize* 0 = 0

begin

Class *normalization-semidom* cultivates the idea that each integral domain can be split into equivalence classes whose representants are associated, i.e. divide each other. *normalize* specifies a canonical representant for each equivalence class. The rationale behind this is that it is easier to reason about equality than equivalences, hence we prefer to think about equality of normalized values rather than associated elements.

declare *unit-factor-is-unit* [*iff*]

lemma *unit-factor-dvd* [*simp*]: a \neq 0 \implies *unit-factor* a dvd b

by (*rule* *unit-imp-dvd*) *simp*

lemma *unit-factor-self* [*simp*]: *unit-factor* a dvd a

by (*cases* a = 0) *simp-all*

lemma *normalize-mult-unit-factor* [*simp*]: *normalize* a * *unit-factor* a = a

using *unit-factor-mult-normalize* [of a] **by** (*simp* add: *ac-simps*)

lemma *normalize-eq-0-iff* [*simp*]: *normalize* a = 0 \iff a = 0

(**is** ?lhs \iff ?rhs)

proof

assume *?lhs*

moreover have *unit-factor a * normalize a = a* **by** *simp*

ultimately show *?rhs* **by** *simp*

next

assume *?rhs*

then show *?lhs* **by** *simp*

qed

lemma *unit-factor-eq-0-iff* [*simp*]: *unit-factor a = 0* \longleftrightarrow *a = 0*
 (**is** *?lhs* \longleftrightarrow *?rhs*)

proof

assume *?lhs*

moreover have *unit-factor a * normalize a = a* **by** *simp*

ultimately show *?rhs* **by** *simp*

next

assume *?rhs*

then show *?lhs* **by** *simp*

qed

lemma *div-unit-factor* [*simp*]: *a div unit-factor a = normalize a*

proof (*cases a = 0*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

then have *unit-factor a \neq 0*

by *simp*

with *nonzero-mult-div-cancel-left*

have *unit-factor a * normalize a div unit-factor a = normalize a*

by *blast*

then show *?thesis* **by** *simp*

qed

lemma *normalize-div* [*simp*]: *normalize a div a = 1 div unit-factor a*

proof (*cases a = 0*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

have *normalize a div a = normalize a div (unit-factor a * normalize a)*

by *simp*

also have $\dots = 1 \text{ div unit-factor } a$

using *False* **by** (*subst is-unit-div-mult-cancel-right*) *simp-all*

finally show *?thesis* .

qed

lemma *is-unit-normalize*:

assumes *is-unit a*

shows $normalize\ a = 1$
proof –
from *assms* **have** $unit\text{-}factor\ a = a$
by (*rule is-unit-unit-factor*)
moreover from *assms* **have** $a \neq 0$
by *auto*
moreover have $normalize\ a = a\ div\ unit\text{-}factor\ a$
by *simp*
ultimately show *?thesis*
by *simp*
qed

lemma *unit-factor-1* [*simp*]: $unit\text{-}factor\ 1 = 1$
by (*rule is-unit-unit-factor*) *simp*

lemma *normalize-1* [*simp*]: $normalize\ 1 = 1$
by (*rule is-unit-normalize*) *simp*

lemma *normalize-1-iff*: $normalize\ a = 1 \longleftrightarrow is\text{-}unit\ a$
(is *?lhs* \longleftrightarrow *?rhs***)**

proof
assume *?rhs*
then show *?lhs* **by** (*rule is-unit-normalize*)
next
assume *?lhs*
then have $unit\text{-}factor\ a * normalize\ a = unit\text{-}factor\ a * 1$
by *simp*
then have $unit\text{-}factor\ a = a$
by *simp*
moreover
from $\langle ?lhs \rangle$ **have** $a \neq 0$ **by** *auto*
then have $is\text{-}unit\ (unit\text{-}factor\ a)$ **by** *simp*
ultimately show *?rhs* **by** *simp*
qed

lemma *div-normalize* [*simp*]: $a\ div\ normalize\ a = unit\text{-}factor\ a$

proof (*cases a = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $normalize\ a \neq 0$ **by** *simp*
with *nonzero-mult-div-cancel-right*
have $unit\text{-}factor\ a * normalize\ a\ div\ normalize\ a = unit\text{-}factor\ a$ **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *mult-one-div-unit-factor* [*simp*]: $a * (1\ div\ unit\text{-}factor\ b) = a\ div\ unit\text{-}factor\ b$

by (cases $b = 0$) simp-all

lemma *inv-unit-factor-eq-0-iff* [simp]:

$1 \text{ div unit-factor } a = 0 \iff a = 0$

(is ?lhs \iff ?rhs)

proof

assume ?lhs

then have $a * (1 \text{ div unit-factor } a) = a * 0$

by simp

then show ?rhs

by simp

next

assume ?rhs

then show ?lhs by simp

qed

lemma *unit-factor-idem* [simp]: $\text{unit-factor } (\text{unit-factor } a) = \text{unit-factor } a$

by (cases $a = 0$) (auto intro: is-unit-unit-factor)

lemma *normalize-unit-factor* [simp]: $a \neq 0 \implies \text{normalize } (\text{unit-factor } a) = 1$

by (rule is-unit-normalize) simp

lemma *normalize-mult-unit-left* [simp]:

assumes $a \text{ dvd } 1$

shows $\text{normalize } (a * b) = \text{normalize } b$

proof (cases $b = 0$)

case False

have $a * \text{unit-factor } b * \text{normalize } (a * b) = \text{unit-factor } (a * b) * \text{normalize } (a * b)$

using assms by (subst unit-factor-mult-unit-left) auto

also have $\dots = a * b$ by simp

also have $b = \text{unit-factor } b * \text{normalize } b$ by simp

hence $a * b = a * \text{unit-factor } b * \text{normalize } b$

by (simp only: mult-ac)

finally show ?thesis

using assms False by auto

qed auto

lemma *normalize-mult-unit-right* [simp]:

assumes $b \text{ dvd } 1$

shows $\text{normalize } (a * b) = \text{normalize } a$

using assms by (subst mult.commute) auto

lemma *normalize-idem* [simp]: $\text{normalize } (\text{normalize } a) = \text{normalize } a$

proof (cases $a = 0$)

case False

have $\text{normalize } a = \text{normalize } (\text{unit-factor } a * \text{normalize } a)$

by simp

also from False have $\dots = \text{normalize } (\text{normalize } a)$

by (subst normalize-mult-unit-left) auto
 finally show ?thesis ..
 qed auto

lemma unit-factor-normalize [simp]:
 assumes $a \neq 0$
 shows unit-factor (normalize a) = 1
proof –
 from assms have *: normalize a $\neq 0$
 by simp
 have unit-factor (normalize a) * normalize (normalize a) = normalize a
 by (simp only: unit-factor-mult-normalize)
 then have unit-factor (normalize a) * normalize a = normalize a
 by simp
 with * have unit-factor (normalize a) * normalize a div normalize a = normalize
 a div normalize a
 by simp
 with * show ?thesis
 by simp
 qed

lemma normalize-dvd-iff [simp]: normalize a dvd b \longleftrightarrow a dvd b
proof –
 have normalize a dvd b \longleftrightarrow unit-factor a * normalize a dvd b
 using mult-unit-dvd-iff [of unit-factor a normalize a b]
 by (cases a = 0) simp-all
 then show ?thesis by simp
 qed

lemma dvd-normalize-iff [simp]: a dvd normalize b \longleftrightarrow a dvd b
proof –
 have a dvd normalize b \longleftrightarrow a dvd normalize b * unit-factor b
 using dvd-mult-unit-iff [of unit-factor b a normalize b]
 by (cases b = 0) simp-all
 then show ?thesis by simp
 qed

lemma normalize-idem-imp-unit-factor-eq:
 assumes normalize a = a
 shows unit-factor a = of-bool (a $\neq 0$)
proof (cases a = 0)
 case True
 then show ?thesis
 by simp
 next
 case False
 then show ?thesis
 using assms unit-factor-normalize [of a] by simp
 qed

lemma *normalize-idem-imp-is-unit-iff*:
assumes *normalize a = a*
shows *is-unit a \longleftrightarrow a = 1*
using *assms* **by** (*cases a = 0*) (*auto dest: is-unit-normalize*)

lemma *coprime-normalize-left-iff* [*simp*]:
coprime (normalize a) b \longleftrightarrow coprime a b
by (*rule iffI*; *rule coprimeI*) (*auto intro: coprime-common-divisor*)

lemma *coprime-normalize-right-iff* [*simp*]:
coprime a (normalize b) \longleftrightarrow coprime a b
using *coprime-normalize-left-iff* [*of b a*] **by** (*simp add: ac-simps*)

We avoid an explicit definition of associated elements but prefer explicit normalisation instead. In theory we could define an abbreviation like *associated a b = (normalize a = normalize b)* but this is counterproductive without suggestive infix syntax, which we do not want to sacrifice for this purpose here.

lemma *associatedI*:
assumes *a dvd b and b dvd a*
shows *normalize a = normalize b*
proof (*cases a = 0 \vee b = 0*)
case *True*
with *assms* **show** *?thesis* **by** *auto*
next
case *False*
from *$\langle a \text{ dvd } b \rangle$* **obtain** *c* **where** *b = a * c ..*
moreover from *$\langle b \text{ dvd } a \rangle$* **obtain** *d* **where** *a = b * d ..*
ultimately have *b * 1 = b * (c * d)*
by (*simp add: ac-simps*)
with *False* **have** *1 = c * d*
unfolding *mult-cancel-left* **by** *simp*
then have *is-unit c and is-unit d*
by *auto*
with *a b* **show** *?thesis*
by (*simp add: is-unit-normalize*)
qed

lemma *associatedD1*: *normalize a = normalize b \implies a dvd b*
using *dvd-normalize-iff* [*of - b, symmetric*] *normalize-dvd-iff* [*of a -, symmetric*]
by *simp*

lemma *associatedD2*: *normalize a = normalize b \implies b dvd a*
using *dvd-normalize-iff* [*of - a, symmetric*] *normalize-dvd-iff* [*of b -, symmetric*]
by *simp*

lemma *associated-unit*: *normalize a = normalize b \implies is-unit a \implies is-unit b*
using *dvd-unit-imp-unit* **by** (*auto dest!: associatedD1 associatedD2*)

lemma *associated-iff-dvd*: $normalize\ a = normalize\ b \iff a\ dvd\ b \wedge b\ dvd\ a$
(is ?lhs \iff ?rhs)

proof
assume *?rhs*
then show *?lhs* **by** (*auto intro!*: *associatedI*)

next
assume *?lhs*
then have *unit-factor a * normalize a = unit-factor a * normalize b*
by *simp*
then have **; normalize b * unit-factor a = a*
by (*simp add: ac-simps*)
show *?rhs*
proof (*cases a = 0 \vee b = 0*)
case *True*
with *<?lhs>* **show** *?thesis* **by** *auto*

next
case *False*
then have *b dvd normalize b * unit-factor a* **and** *normalize b * unit-factor a*
dvd b
by (*simp-all add: mult-unit-dvd-iff dvd-mult-unit-iff*)
with *** **show** *?thesis* **by** *simp*

qed
qed

lemma *associated-eqI*:
assumes *a dvd b* **and** *b dvd a*
assumes *normalize a = a* **and** *normalize b = b*
shows *a = b*

proof –
from *assms* **have** *normalize a = normalize b*
unfolding *associated-iff-dvd* **by** *simp*
with *<normalize a = a>* **have** *a = normalize b*
by *simp*
with *<normalize b = b>* **show** *a = b*
by *simp*

qed

lemma *normalize-unit-factor-eqI*:
assumes *normalize a = normalize b*
and *unit-factor a = unit-factor b*
shows *a = b*

proof –
from *assms* **have** *unit-factor a * normalize a = unit-factor b * normalize b*
by *simp*
then show *?thesis*
by *simp*

qed

lemma *normalize-mult-normalize-left* [simp]: $\text{normalize} (\text{normalize } a * b) = \text{normalize} (a * b)$

by (rule *associated-eqI*) (auto intro!: *mult-dvd-mono*)

lemma *normalize-mult-normalize-right* [simp]: $\text{normalize} (a * \text{normalize } b) = \text{normalize} (a * b)$

by (rule *associated-eqI*) (auto intro!: *mult-dvd-mono*)

end

class *normalization-semidom-multiplicative* = *normalization-semidom* +
assumes *unit-factor-mult*: $\text{unit-factor} (a * b) = \text{unit-factor } a * \text{unit-factor } b$
begin

lemma *normalize-mult*: $\text{normalize} (a * b) = \text{normalize } a * \text{normalize } b$

proof (cases $a = 0 \vee b = 0$)

case *True*

then show *?thesis* **by** *auto*

next

case *False*

have $\text{unit-factor} (a * b) * \text{normalize} (a * b) = a * b$

by (rule *unit-factor-mult-normalize*)

then have $\text{normalize} (a * b) = a * b \text{ div } \text{unit-factor} (a * b)$

by *simp*

also have $\dots = a * b \text{ div } \text{unit-factor} (b * a)$

by (*simp add: ac-simps*)

also have $\dots = a * b \text{ div } \text{unit-factor } b \text{ div } \text{unit-factor } a$

using *False* **by** (*simp add: unit-factor-mult is-unit-div-mult2-eq [symmetric]*)

also have $\dots = a * (b \text{ div } \text{unit-factor } b) \text{ div } \text{unit-factor } a$

using *False* **by** (*subst unit-div-mult-swap simp-all*)

also have $\dots = \text{normalize } a * \text{normalize } b$

using *False*

by (*simp add: mult.commute [of a] mult.commute [of normalize a] unit-div-mult-swap [symmetric]*)

finally show *?thesis* .

qed

lemma *dvd-unit-factor-div*:

assumes $b \text{ dvd } a$

shows $\text{unit-factor} (a \text{ div } b) = \text{unit-factor } a \text{ div } \text{unit-factor } b$

proof –

from *assms* **have** $a = a \text{ div } b * b$

by *simp*

then have $\text{unit-factor } a = \text{unit-factor} (a \text{ div } b * b)$

by *simp*

then show *?thesis*

by (cases $b = 0$) (*simp-all add: unit-factor-mult*)

qed

```

lemma dvd-normalize-div:
  assumes  $b \text{ dvd } a$ 
  shows  $\text{normalize } (a \text{ div } b) = \text{normalize } a \text{ div } \text{normalize } b$ 
proof –
  from assms have  $a = a \text{ div } b * b$ 
    by simp
  then have  $\text{normalize } a = \text{normalize } (a \text{ div } b * b)$ 
    by simp
  then show ?thesis
    by (cases  $b = 0$ ) (simp-all add: normalize-mult)
qed

end

```

Syntactic division remainder operator

```

class modulo = dvd + divide +
  fixes modulo :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl mod 70)

```

Arbitrary quotient and remainder partitions

```

class semiring-modulo = comm-semiring-1-cancel + divide + modulo +
  assumes div-mult-mod-eq:  $a \text{ div } b * b + a \text{ mod } b = a$ 
begin

```

```

lemma mod-div-decomp:
  fixes  $a \ b$ 
  obtains  $q \ r$  where  $q = a \text{ div } b$  and  $r = a \text{ mod } b$ 
    and  $a = q * b + r$ 
proof –
  from div-mult-mod-eq have  $a = a \text{ div } b * b + a \text{ mod } b$  by simp
  moreover have  $a \text{ div } b = a \text{ div } b$  ..
  moreover have  $a \text{ mod } b = a \text{ mod } b$  ..
  note that ultimately show thesis by blast
qed

```

```

lemma mult-div-mod-eq:  $b * (a \text{ div } b) + a \text{ mod } b = a$ 
  using div-mult-mod-eq [of a b] by (simp add: ac-simps)

```

```

lemma mod-div-mult-eq:  $a \text{ mod } b + a \text{ div } b * b = a$ 
  using div-mult-mod-eq [of a b] by (simp add: ac-simps)

```

```

lemma mod-mult-div-eq:  $a \text{ mod } b + b * (a \text{ div } b) = a$ 
  using div-mult-mod-eq [of a b] by (simp add: ac-simps)

```

```

lemma minus-div-mult-eq-mod:  $a - a \text{ div } b * b = a \text{ mod } b$ 
  by (rule add-implies-diff [symmetric]) (fact mod-div-mult-eq)

```

```

lemma minus-mult-div-eq-mod:  $a - b * (a \text{ div } b) = a \text{ mod } b$ 
  by (rule add-implies-diff [symmetric]) (fact mod-mult-div-eq)

```

lemma *minus-mod-eq-div-mult*: $a - a \text{ mod } b = a \text{ div } b * b$
by (*rule add-implies-diff* [*symmetric*]) (*fact div-mult-mod-eq*)

lemma *minus-mod-eq-mult-div*: $a - a \text{ mod } b = b * (a \text{ div } b)$
by (*rule add-implies-diff* [*symmetric*]) (*fact mult-div-mod-eq*)

lemma *mod-0-imp-dvd* [*dest!*]:

b dvd a **if** $a \text{ mod } b = 0$

proof –

have *b dvd (a div b) * b* **by** *simp*

also have $(a \text{ div } b) * b = a$

using *div-mult-mod-eq* [*of a b*] **by** (*simp add: that*)

finally show *?thesis* .

qed

lemma [*nitpick-unfold*]:

$a \text{ mod } b = a - a \text{ div } b * b$

by (*fact minus-div-mult-eq-mod* [*symmetric*])

end

16.5 Quotient and remainder in integral domains

class *semidom-modulo* = *algebraic-semidom* + *semiring-modulo*

begin

lemma *mod-0* [*simp*]: $0 \text{ mod } a = 0$

using *div-mult-mod-eq* [*of 0 a*] **by** *simp*

lemma *mod-by-0* [*simp*]: $a \text{ mod } 0 = a$

using *div-mult-mod-eq* [*of a 0*] **by** *simp*

lemma *mod-by-1* [*simp*]:

$a \text{ mod } 1 = 0$

proof –

from *div-mult-mod-eq* [*of a one*] *div-by-1* **have** $a + a \text{ mod } 1 = a$ **by** *simp*

then have $a + a \text{ mod } 1 = a + 0$ **by** *simp*

then show *?thesis* **by** (*rule add-left-imp-eq*)

qed

lemma *mod-self* [*simp*]:

$a \text{ mod } a = 0$

using *div-mult-mod-eq* [*of a a*] **by** *simp*

lemma *dvd-imp-mod-0* [*simp*]:

$b \text{ mod } a = 0$ **if** *a dvd b*

using *that minus-div-mult-eq-mod* [*of b a*] **by** *simp*

lemma *mod-eq-0-iff-dvd*:

$a \bmod b = 0 \iff b \text{ dvd } a$

by (*auto intro: mod-0-imp-dvd*)

lemma *dvd-eq-mod-eq-0* [*nitpick-unfold, code*]:

$a \text{ dvd } b \iff b \bmod a = 0$

by (*simp add: mod-eq-0-iff-dvd*)

lemma *dvd-mod-iff*:

assumes $c \text{ dvd } b$

shows $c \text{ dvd } a \bmod b \iff c \text{ dvd } a$

proof –

from *assms* **have** $(c \text{ dvd } a \bmod b) \iff (c \text{ dvd } ((a \text{ div } b) * b + a \bmod b))$

by (*simp add: dvd-add-right-iff*)

also have $(a \text{ div } b) * b + a \bmod b = a$

using *div-mult-mod-eq* [*of a b*] **by** *simp*

finally show *?thesis* .

qed

lemma *dvd-mod-imp-dvd*:

assumes $c \text{ dvd } a \bmod b$ **and** $c \text{ dvd } b$

shows $c \text{ dvd } a$

using *assms dvd-mod-iff* [*of c b a*] **by** *simp*

lemma *dvd-minus-mod* [*simp*]:

$b \text{ dvd } a - a \bmod b$

by (*simp add: minus-mod-eq-div-mult*)

lemma *cancel-div-mod-rules*:

$((a \text{ div } b) * b + a \bmod b) + c = a + c$

$(b * (a \text{ div } b) + a \bmod b) + c = a + c$

by (*simp-all add: div-mult-mod-eq mult-div-mod-eq*)

end

class *idom-modulo* = *idom* + *semidom-modulo*

begin

subclass *idom-divide* ..

lemma *div-diff* [*simp*]:

$c \text{ dvd } a \implies c \text{ dvd } b \implies (a - b) \text{ div } c = a \text{ div } c - b \text{ div } c$

using *div-add* [*of - - b*] **by** (*simp add: dvd-neg-div*)

end

16.6 Interlude: basic tool support for algebraic and arithmetic calculations

named-theorems *arith arith facts* -- only ground formulas

ML-file $\langle \text{Tools/arith-data.ML} \rangle$

ML-file $\langle \sim / \text{src/Provers/Arith/cancel-div-mod.ML} \rangle$

ML \langle

structure Cancel-Div-Mod-Ring = Cancel-Div-Mod

$($

val div-name = const-name $\langle \text{divide} \rangle$;

val mod-name = const-name $\langle \text{modulo} \rangle$;

val mk-binop = HLogic.mk-binop;

val mk-sum = Arith-Data.mk-sum;

val dest-sum = Arith-Data.dest-sum;

val div-mod-eqs = map mk-meta-eq $\@ \{ \text{thms cancel-div-mod-rules} \}$;

val prove-eq-sums = Arith-Data.prove-conv2 all-tac $(\text{Arith-Data.simp-all-tac}$

$\@ \{ \text{thms diff-conv-add-uminus add-0-left add-0-right ac-simps} \})$

$)$

\rangle

simproc-setup *cancel-div-mod-int* $((a::'a::\text{semidom-modulo}) + b) =$

$\langle K \text{ Cancel-Div-Mod-Ring.proc} \rangle$

16.7 Ordered semirings and rings

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society, 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press, 1963

Most of the used notions can also be looked up in

- <http://www.mathworld.com> by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

class *ordered-semiring* = *semiring* + *ordered-comm-monoid-add* +

assumes *mult-left-mono*: $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

assumes *mult-right-mono*: $a \leq b \implies 0 \leq c \implies a * c \leq b * c$

begin

lemma *mult-mono*: $a \leq b \implies c \leq d \implies 0 \leq b \implies 0 \leq c \implies a * c \leq b * d$

```

apply (erule (1) mult-right-mono [THEN order-trans])
apply (erule (1) mult-left-mono)
done

lemma mult-mono':  $a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \implies a * c \leq b * d$ 
by (rule mult-mono) (fast intro: order-trans)+

end

lemma mono-mult:
  fixes a :: 'a::ordered-semiring
  shows  $a \geq 0 \implies \text{mono } ((* a)$ 
  by (simp add: mono-def mult-left-mono)

class ordered-semiring-0 = semiring-0 + ordered-semiring
begin

lemma mult-nonneg-nonneg [simp]:  $0 \leq a \implies 0 \leq b \implies 0 \leq a * b$ 
using mult-left-mono [of 0 b a] by simp

lemma mult-nonneg-nonpos:  $0 \leq a \implies b \leq 0 \implies a * b \leq 0$ 
using mult-left-mono [of b 0 a] by simp

lemma mult-nonpos-nonneg:  $a \leq 0 \implies 0 \leq b \implies a * b \leq 0$ 
using mult-right-mono [of a 0 b] by simp

Legacy – use mult-nonpos-nonneg.

lemma mult-nonneg-nonpos2:  $0 \leq a \implies b \leq 0 \implies b * a \leq 0$ 
by (drule mult-right-mono [of b 0]) auto

lemma split-mult-neg-le:  $(0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b) \implies a * b \leq 0$ 
by (auto simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)

end

class ordered-cancel-semiring = ordered-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..

subclass ordered-semiring-0 ..

end

class linordered-semiring = ordered-semiring + linordered-cancel-ab-semigroup-add
begin

subclass ordered-cancel-semiring ..

```

subclass *ordered-cancel-comm-monoid-add* ..

subclass *ordered-ab-semigroup-monoid-add-imp-le* ..

lemma *mult-left-less-imp-less*: $c * a < c * b \implies 0 \leq c \implies a < b$
by (*force simp add: mult-left-mono not-le [symmetric]*)

lemma *mult-right-less-imp-less*: $a * c < b * c \implies 0 \leq c \implies a < b$
by (*force simp add: mult-right-mono not-le [symmetric]*)

end

class *zero-less-one* = *order* + *zero* + *one* +
assumes *zero-less-one [simp]*: $0 < 1$
begin

subclass *zero-neg-one*
by *standard (simp add: less-imp-neg)*

lemma *zero-le-one [simp]*:
 $\langle 0 \leq 1 \rangle$ **by** (*rule less-imp-le*) *simp*

end

class *linordered-semiring-1* = *linordered-semiring* + *semiring-1* + *zero-less-one*
begin

lemma *convex-bound-le*:
assumes $x \leq a$ $y \leq a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
shows $u * x + v * y \leq a$

proof –

from *assms* **have** $u * x + v * y \leq u * a + v * a$

by (*simp add: add-mono mult-left-mono*)

with *assms* **show** *?thesis*

unfolding *distrib-right[symmetric]* **by** *simp*

qed

end

class *linordered-semiring-strict* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add*
+

assumes *mult-strict-left-mono*: $a < b \implies 0 < c \implies c * a < c * b$

assumes *mult-strict-right-mono*: $a < b \implies 0 < c \implies a * c < b * c$

begin

subclass *semiring-0-cancel* ..

subclass *linordered-semiring*

proof

```

fix a b c :: 'a
assume *: a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
from * show a * c ≤ b * c
  unfolding le-less
  using mult-strict-right-mono by (cases c = 0) auto
qed

```

```

lemma mult-left-le-imp-le: c * a ≤ c * b ⇒ 0 < c ⇒ a ≤ b
  by (auto simp add: mult-strict-left-mono -not-less [symmetric])

```

```

lemma mult-right-le-imp-le: a * c ≤ b * c ⇒ 0 < c ⇒ a ≤ b
  by (auto simp add: mult-strict-right-mono not-less [symmetric])

```

```

lemma mult-pos-pos[simp]: 0 < a ⇒ 0 < b ⇒ 0 < a * b
  using mult-strict-left-mono [of 0 b a] by simp

```

```

lemma mult-pos-neg: 0 < a ⇒ b < 0 ⇒ a * b < 0
  using mult-strict-left-mono [of b 0 a] by simp

```

```

lemma mult-neg-pos: a < 0 ⇒ 0 < b ⇒ a * b < 0
  using mult-strict-right-mono [of a 0 b] by simp

```

Legacy – use *mult-neg-pos*.

```

lemma mult-pos-neg2: 0 < a ⇒ b < 0 ⇒ b * a < 0
  by (drule mult-strict-right-mono [of b 0]) auto

```

```

lemma zero-less-mult-pos:
  assumes 0 < a * b 0 < a shows 0 < b
proof (cases b ≤ 0)
  case True
  then show ?thesis
    using assms by (auto simp: le-less dest: less-not-sym mult-pos-neg [of a b])
qed (auto simp add: le-less not-less)

```

```

lemma zero-less-mult-pos2:
  assumes 0 < b * a 0 < a shows 0 < b
proof (cases b ≤ 0)
  case True
  then show ?thesis
    using assms by (auto simp: le-less dest: less-not-sym mult-pos-neg2 [of a b])
qed (auto simp add: le-less not-less)

```

Strict monotonicity in both arguments

```

lemma mult-strict-mono:
  assumes a < b c < d 0 < b 0 ≤ c

```

```

shows  $a * c < b * d$ 
proof (cases  $c = 0$ )
  case True
    with assms show ?thesis
      by simp
  next
    case False
      with assms have  $a * c < b * c$ 
        by (simp add: mult-strict-right-mono [OF ‹a < b›])
      also have  $\dots < b * d$ 
        by (simp add: assms mult-strict-left-mono)
      finally show ?thesis .
qed

```

This weaker variant has more natural premises

```

lemma mult-strict-mono':
  assumes  $a < b$  and  $c < d$  and  $0 \leq a$  and  $0 \leq c$ 
  shows  $a * c < b * d$ 
  using assms by (auto simp add: mult-strict-mono)

```

```

lemma mult-less-le-imp-less:
  assumes  $a < b$  and  $c \leq d$  and  $0 \leq a$  and  $0 < c$ 
  shows  $a * c < b * d$ 
proof –
  have  $a * c < b * c$ 
    by (simp add: assms mult-strict-right-mono)
  also have  $\dots \leq b * d$ 
    by (intro mult-left-mono) (use assms in auto)
  finally show ?thesis .
qed

```

```

lemma mult-le-less-imp-less:
  assumes  $a \leq b$  and  $c < d$  and  $0 < a$  and  $0 \leq c$ 
  shows  $a * c < b * d$ 
proof –
  have  $a * c \leq b * c$ 
    by (simp add: assms mult-right-mono)
  also have  $\dots < b * d$ 
    by (intro mult-strict-left-mono) (use assms in auto)
  finally show ?thesis .
qed

```

end

```

class linordered-semiring-1-strict = linordered-semiring-strict + semiring-1 + zero-less-one
begin

```

```

subclass linordered-semiring-1 ..

```

lemma *convex-bound-lt*:

assumes $x < a$ $y < a$ $0 \leq u$ $0 \leq v$ $u + v = 1$

shows $u * x + v * y < a$

proof –

from *assms* **have** $u * x + v * y < u * a + v * a$

by (*cases* $u = 0$) (*auto intro!*: *add-less-le-mono* *mult-strict-left-mono* *mult-left-mono*)

with *assms* **show** *?thesis*

unfolding *distrib-right[symmetric]* **by** *simp*

qed

end

class *ordered-comm-semiring* = *comm-semiring-0* + *ordered-ab-semigroup-add* +

assumes *comm-mult-left-mono*: $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

begin

subclass *ordered-semiring*

proof

fix a b c :: *'a*

assume $a \leq b$ $0 \leq c$

then show $c * a \leq c * b$ **by** (*rule comm-mult-left-mono*)

then show $a * c \leq b * c$ **by** (*simp only: mult.commute*)

qed

end

class *ordered-cancel-comm-semiring* = *ordered-comm-semiring* + *cancel-comm-monoid-add*

begin

subclass *comm-semiring-0-cancel* ..

subclass *ordered-comm-semiring* ..

subclass *ordered-cancel-semiring* ..

end

class *linordered-comm-semiring-strict* = *comm-semiring-0* + *linordered-cancel-ab-semigroup-add*

+

assumes *comm-mult-strict-left-mono*: $a < b \implies 0 < c \implies c * a < c * b$

begin

subclass *linordered-semiring-strict*

proof

fix a b c :: *'a*

assume $a < b$ $0 < c$

then show $c * a < c * b$

by (*rule comm-mult-strict-left-mono*)

then show $a * c < b * c$

by (*simp only: mult.commute*)

qed

```

subclass ordered-cancel-comm-semiring
proof
  fix a b c :: 'a
  assume a ≤ b 0 ≤ c
  then show c * a ≤ c * b
    unfolding le-less
    using mult-strict-left-mono by (cases c = 0) auto
qed

end

class ordered-ring = ring + ordered-cancel-semiring
begin

subclass ordered-ab-group-add ..

lemma less-add-iff1: a * e + c < b * e + d ↔ (a - b) * e + c < d
  by (simp add: algebra-simps)

lemma less-add-iff2: a * e + c < b * e + d ↔ c < (b - a) * e + d
  by (simp add: algebra-simps)

lemma le-add-iff1: a * e + c ≤ b * e + d ↔ (a - b) * e + c ≤ d
  by (simp add: algebra-simps)

lemma le-add-iff2: a * e + c ≤ b * e + d ↔ c ≤ (b - a) * e + d
  by (simp add: algebra-simps)

lemma mult-left-mono-neg: b ≤ a ⇒ c ≤ 0 ⇒ c * a ≤ c * b
  by (auto dest: mult-left-mono [of - - - c])

lemma mult-right-mono-neg: b ≤ a ⇒ c ≤ 0 ⇒ a * c ≤ b * c
  by (auto dest: mult-right-mono [of - - - c])

lemma mult-nonpos-nonpos: a ≤ 0 ⇒ b ≤ 0 ⇒ 0 ≤ a * b
  using mult-right-mono-neg [of a 0 b] by simp

lemma split-mult-pos-le: (0 ≤ a ∧ 0 ≤ b) ∨ (a ≤ 0 ∧ b ≤ 0) ⇒ 0 ≤ a * b
  by (auto simp add: mult-nonpos-nonpos)

end

class abs-if = minus + uminus + ord + zero + abs +
  assumes abs-if: |a| = (if a < 0 then - a else a)

class linordered-ring = ring + linordered-semiring + linordered-ab-group-add +
  abs-if
begin

```

```

subclass ordered-ring ..

subclass ordered-ab-group-add-abs
proof
  fix a b
  show  $|a + b| \leq |a| + |b|$ 
    by (auto simp add: abs-if not-le not-less algebra-simps
        simp del: add commute dest: add-neg-neg add-nonneg-nonneg)
qed (auto simp: abs-if)

lemma zero-le-square [simp]:  $0 \leq a * a$ 
  using linear [of 0 a] by (auto simp add: mult-nonpos-nonpos)

lemma not-square-less-zero [simp]:  $\neg (a * a < 0)$ 
  by (simp add: not-less)

proposition abs-eq-iff:  $|x| = |y| \iff x = y \vee x = -y$ 
  by (auto simp add: abs-if split: if-split-asm)

lemma abs-eq-iff':
   $|a| = b \iff b \geq 0 \wedge (a = b \vee a = -b)$ 
  by (cases a  $\geq 0$ ) auto

lemma eq-abs-iff':
   $a = |b| \iff a \geq 0 \wedge (b = a \vee b = -a)$ 
  using abs-eq-iff' [of b a] by auto

lemma sum-squares-ge-zero:  $0 \leq x * x + y * y$ 
  by (intro add-nonneg-nonneg zero-le-square)

lemma not-sum-squares-lt-zero:  $\neg x * x + y * y < 0$ 
  by (simp add: not-less sum-squares-ge-zero)

end

class linordered-ring-strict = ring + linordered-semiring-strict
  + ordered-ab-group-add + abs-if
begin

subclass linordered-ring ..

lemma mult-strict-left-mono-neg:  $b < a \implies c < 0 \implies c * a < c * b$ 
  using mult-strict-left-mono [of b a - c] by simp

lemma mult-strict-right-mono-neg:  $b < a \implies c < 0 \implies a * c < b * c$ 
  using mult-strict-right-mono [of b a - c] by simp

lemma mult-neg-neg:  $a < 0 \implies b < 0 \implies 0 < a * b$ 

```



```

using mult-strict-right-mono-neg [of a 0 b] by simp

subclass ring-no-zero-divisors
proof
  fix a b
  assume a ≠ 0
  then have a: a < 0 ∨ 0 < a by (simp add: neq-iff)
  assume b ≠ 0
  then have b: b < 0 ∨ 0 < b by (simp add: neq-iff)
  have a * b < 0 ∨ 0 < a * b
  proof (cases a < 0)
    case True
    show ?thesis
    proof (cases b < 0)
      case True
      with ⟨a < 0⟩ show ?thesis by (auto dest: mult-neg-neg)
    next
      case False
      with b have 0 < b by auto
      with ⟨a < 0⟩ show ?thesis by (auto dest: mult-strict-right-mono)
    qed
  next
  case False
  with a have 0 < a by auto
  show ?thesis
  proof (cases b < 0)
    case True
    with ⟨0 < a⟩ show ?thesis
    by (auto dest: mult-strict-right-mono-neg)
  next
  case False
  with b have 0 < b by auto
  with ⟨0 < a⟩ show ?thesis by auto
  qed
  then show a * b ≠ 0
  by (simp add: neq-iff)
qed

lemma zero-less-mult-iff [algebra-split-simps, field-split-simps]:
  0 < a * b ⟷ 0 < a ∧ 0 < b ∨ a < 0 ∧ b < 0
  by (cases a 0 b 0 rule: linorder-cases[case-product linorder-cases])
  (auto simp add: mult-neg-neg not-less le-less dest: zero-less-mult-pos zero-less-mult-pos2)

lemma zero-le-mult-iff [algebra-split-simps, field-split-simps]:
  0 ≤ a * b ⟷ 0 ≤ a ∧ 0 ≤ b ∨ a ≤ 0 ∧ b ≤ 0
  by (auto simp add: eq-commute [of 0] le-less not-less zero-less-mult-iff)

lemma mult-less-0-iff [algebra-split-simps, field-split-simps]:

```

$a * b < 0 \iff 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
using *zero-less-mult-iff* [of $- a b$] **by** *auto*

lemma *mult-le-0-iff* [*algebra-split-simps*, *field-split-simps*]:
 $a * b \leq 0 \iff 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
using *zero-le-mult-iff* [of $- a b$] **by** *auto*

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations \leq and equality.

These “disjunction” versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

lemma *mult-less-cancel-right-disj*: $a * c < b * c \iff 0 < c \wedge a < b \vee c < 0 \wedge b < a$

proof (*cases c = 0*)
case *False*
show *?thesis* (**is** *?lhs* \iff *?rhs*)
proof
assume *?lhs*
then have $c < 0 \implies b < a \ c > 0 \implies b > a$
by (*auto simp flip: not-le intro: mult-right-mono mult-right-mono-neg*)
with *False* **show** *?rhs*
by (*auto simp add: neq-iff*)
next
assume *?rhs*
with *False* **show** *?lhs*
by (*auto simp add: mult-strict-right-mono mult-strict-right-mono-neg*)
qed
qed *auto*

lemma *mult-less-cancel-left-disj*: $c * a < c * b \iff 0 < c \wedge a < b \vee c < 0 \wedge b < a$

proof (*cases c = 0*)
case *False*
show *?thesis* (**is** *?lhs* \iff *?rhs*)
proof
assume *?lhs*
then have $c < 0 \implies b < a \ c > 0 \implies b > a$
by (*auto simp flip: not-le intro: mult-left-mono mult-left-mono-neg*)
with *False* **show** *?rhs*
by (*auto simp add: neq-iff*)
next
assume *?rhs*
with *False* **show** *?lhs*
by (*auto simp add: mult-strict-left-mono mult-strict-left-mono-neg*)
qed
qed *auto*

The “conjunction of implication” lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*: $a * c < b * c \leftrightarrow (0 \leq c \rightarrow a < b) \wedge (c \leq 0 \rightarrow b < a)$

using *mult-less-cancel-right-disj* [of *a c b*] **by** *auto*

lemma *mult-less-cancel-left*: $c * a < c * b \leftrightarrow (0 \leq c \rightarrow a < b) \wedge (c \leq 0 \rightarrow b < a)$

using *mult-less-cancel-left-disj* [of *c a b*] **by** *auto*

lemma *mult-le-cancel-right*: $a * c \leq b * c \leftrightarrow (0 < c \rightarrow a \leq b) \wedge (c < 0 \rightarrow b \leq a)$

by (*simp add: not-less [symmetric] mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*: $c * a \leq c * b \leftrightarrow (0 < c \rightarrow a \leq b) \wedge (c < 0 \rightarrow b \leq a)$

by (*simp add: not-less [symmetric] mult-less-cancel-left-disj*)

lemma *mult-le-cancel-left-pos*: $0 < c \implies c * a \leq c * b \leftrightarrow a \leq b$

by (*auto simp: mult-le-cancel-left*)

lemma *mult-le-cancel-left-neg*: $c < 0 \implies c * a \leq c * b \leftrightarrow b \leq a$

by (*auto simp: mult-le-cancel-left*)

lemma *mult-less-cancel-left-pos*: $0 < c \implies c * a < c * b \leftrightarrow a < b$

by (*auto simp: mult-less-cancel-left*)

lemma *mult-less-cancel-left-neg*: $c < 0 \implies c * a < c * b \leftrightarrow b < a$

by (*auto simp: mult-less-cancel-left*)

end

lemmas *mult-sign-intros* =
mult-nonneg-nonneg mult-nonneg-nonpos
mult-nonpos-nonneg mult-nonpos-nonpos
mult-pos-pos mult-pos-neg
mult-neg-pos mult-neg-neg

class *ordered-comm-ring* = *comm-ring* + *ordered-comm-semiring*
begin

subclass *ordered-ring* ..
subclass *ordered-cancel-comm-semiring* ..

end

class *linordered-nonzero-semiring* = *ordered-comm-semiring* + *monoid-mult* + *linorder*
+ *zero-less-one* +

assumes *add-mono1*: $a < b \implies a + 1 < b + 1$

begin

```

subclass zero-neq-one
  by standard

subclass comm-semiring-1
  by standard (rule mult-1-left)

lemma zero-le-one [simp]:  $0 \leq 1$ 
  by (rule zero-less-one [THEN less-imp-le])

lemma not-one-le-zero [simp]:  $\neg 1 \leq 0$ 
  by (simp add: not-le)

lemma not-one-less-zero [simp]:  $\neg 1 < 0$ 
  by (simp add: not-less)

lemma of-bool-less-eq-iff [simp]:
   $\langle \text{of-bool } P \leq \text{of-bool } Q \longleftrightarrow (P \longrightarrow Q) \rangle$ 
  by auto

lemma of-bool-less-iff [simp]:
   $\langle \text{of-bool } P < \text{of-bool } Q \longleftrightarrow \neg P \wedge Q \rangle$ 
  by auto

lemma mult-left-le:  $c \leq 1 \implies 0 \leq a \implies a * c \leq a$ 
  using mult-left-mono[of c 1 a] by simp

lemma mult-le-one:  $a \leq 1 \implies 0 \leq b \implies b \leq 1 \implies a * b \leq 1$ 
  using mult-mono[of a 1 b 1] by simp

lemma zero-less-two:  $0 < 1 + 1$ 
  using add-pos-pos[OF zero-less-one zero-less-one] .

end

class linordered-semidom = semidom + linordered-comm-semiring-strict + zero-less-one
+
  assumes le-add-diff-inverse2 [simp]:  $b \leq a \implies a - b + b = a$ 
begin

subclass linordered-nonzero-semiring
proof
  show  $a + 1 < b + 1$  if  $a < b$  for  $a$   $b$ 
  proof (rule ccontr)
    assume  $\neg a + 1 < b + 1$ 
    moreover with that have  $a + 1 < b + 1$ 
      by simp
    ultimately show False
      by contradiction
  qed

```

qed

lemma *zero-less-eq-of-bool* [simp]:
 $\langle 0 \leq \text{of-bool } P \rangle$
 by *simp*

lemma *zero-less-of-bool-iff* [simp]:
 $\langle 0 < \text{of-bool } P \iff P \rangle$
 by *simp*

lemma *of-bool-less-eq-one* [simp]:
 $\langle \text{of-bool } P \leq 1 \rangle$
 by *simp*

lemma *of-bool-less-one-iff* [simp]:
 $\langle \text{of-bool } P < 1 \iff \neg P \rangle$
 by *simp*

lemma *of-bool-or-iff* [simp]:
 $\langle \text{of-bool } (P \vee Q) = \max (\text{of-bool } P) (\text{of-bool } Q) \rangle$
 by (*simp add: max-def*)

Addition is the inverse of subtraction.

lemma *le-add-diff-inverse* [simp]: $b \leq a \implies b + (a - b) = a$
 by (*frule le-add-diff-inverse2*) (*simp add: add.commute*)

lemma *add-diff-inverse*: $\neg a < b \implies b + (a - b) = a$
 by *simp*

lemma *add-le-imp-le-diff*:
 assumes $i + k \leq n$ shows $i \leq n - k$
proof –
 have $n - (i + k) + i + k = n$
 by (*simp add: assms add.assoc*)
 with *assms add-implies-diff* have $i + k \leq n - k + k$
 by *fastforce*
 then show *?thesis*
 by *simp*
 qed

lemma *add-le-add-imp-diff-le*:
 assumes 1: $i + k \leq n$
 and 2: $n \leq j + k$
 shows $i + k \leq n \implies n \leq j + k \implies n - k \leq j$
proof –
 have $n - (i + k) + i + k = n$
 using 1 by (*simp add: add.assoc*)
 moreover have $n - k = n - k - i + i$
 using 1 by (*simp add: add-le-imp-le-diff*)

```

ultimately show ?thesis
  using 2 add-le-imp-le-diff [of n-k k j + k]
  by (simp add: add commute diff-diff-add)
qed

lemma less-1-mult:  $1 < m \implies 1 < n \implies 1 < m * n$ 
  using mult-strict-mono [of 1 m 1 n] by (simp add: less-trans [OF zero-less-one])

end

class linordered-idom = comm-ring-1 + linordered-comm-semiring-strict +
  ordered-ab-group-add + abs-if + sgn +
  assumes sgn-if:  $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -\ 1)$ 
begin

subclass linordered-ring-strict ..

subclass linordered-semiring-1-strict
proof
  have  $0 \leq 1 * 1$ 
    by (fact zero-le-square)
  then show  $0 < 1$ 
    by (simp add: le-less)
qed

subclass ordered-comm-ring ..
subclass idom ..

subclass linordered-semidom
  by standard simp

subclass idom-abs-sgn
  by standard
  (auto simp add: sgn-if abs-if zero-less-mult-iff)

lemma abs-bool-eq [simp]:
   $\langle |of\ bool\ P| = of\ bool\ P \rangle$ 
  by simp

lemma linorder-neqE-linordered-idom:
  assumes  $x \neq y$ 
  obtains  $x < y \mid y < x$ 
  using assms by (rule neqE)

These cancellation simp rules also produce two cases when the comparison
is a goal.

lemma mult-le-cancel-right1:  $c \leq b * c \iff (0 < c \implies 1 \leq b) \wedge (c < 0 \implies b \leq 1)$ 
  using mult-le-cancel-right [of 1 c b] by simp

```

lemma *mult-le-cancel-right2*: $a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$

using *mult-le-cancel-right* [of a c 1] **by** *simp*

lemma *mult-le-cancel-left1*: $c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$

using *mult-le-cancel-left* [of c 1 b] **by** *simp*

lemma *mult-le-cancel-left2*: $c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$

using *mult-le-cancel-left* [of c a 1] **by** *simp*

lemma *mult-less-cancel-right1*: $c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

using *mult-less-cancel-right* [of 1 c b] **by** *simp*

lemma *mult-less-cancel-right2*: $a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$

using *mult-less-cancel-right* [of a c 1] **by** *simp*

lemma *mult-less-cancel-left1*: $c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

using *mult-less-cancel-left* [of c 1 b] **by** *simp*

lemma *mult-less-cancel-left2*: $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$

using *mult-less-cancel-left* [of c a 1] **by** *simp*

lemma *sgn-0-0*: $\text{sgn } a = 0 \longleftrightarrow a = 0$

by (*fact sgn-eq-0-iff*)

lemma *sgn-1-pos*: $\text{sgn } a = 1 \longleftrightarrow a > 0$

unfolding *sgn-if* **by** *simp*

lemma *sgn-1-neg*: $\text{sgn } a = -1 \longleftrightarrow a < 0$

unfolding *sgn-if* **by** *auto*

lemma *sgn-pos* [*simp*]: $0 < a \implies \text{sgn } a = 1$

by (*simp only: sgn-1-pos*)

lemma *sgn-neg* [*simp*]: $a < 0 \implies \text{sgn } a = -1$

by (*simp only: sgn-1-neg*)

lemma *abs-sgn*: $|k| = k * \text{sgn } k$

unfolding *sgn-if abs-if* **by** *auto*

lemma *sgn-greater* [*simp*]: $0 < \text{sgn } a \longleftrightarrow 0 < a$

unfolding *sgn-if* **by** *auto*

lemma *sgn-less* [*simp*]: $\text{sgn } a < 0 \iff a < 0$
unfolding *sgn-if* **by** *auto*

lemma *abs-sgn-eq-1* [*simp*]:
 $a \neq 0 \implies |\text{sgn } a| = 1$
by *simp*

lemma *abs-sgn-eq*: $|\text{sgn } a| = (\text{if } a = 0 \text{ then } 0 \text{ else } 1)$
by (*simp add: sgn-if*)

lemma *sgn-mult-self-eq* [*simp*]:
 $\text{sgn } a * \text{sgn } a = \text{of-bool } (a \neq 0)$
by (*cases a > 0*) *simp-all*

lemma *left-sgn-mult-self-eq* [*simp*]:
 $\langle \text{sgn } a * (\text{sgn } a * b) = \text{of-bool } (a \neq 0) * b \rangle$
by (*simp flip: mult.assoc*)

lemma *abs-mult-self-eq* [*simp*]:
 $|a| * |a| = a * a$
by (*cases a > 0*) *simp-all*

lemma *same-sgn-sgn-add*:
 $\text{sgn } (a + b) = \text{sgn } a$ **if** $\text{sgn } b = \text{sgn } a$
proof (*cases a 0 rule: linorder-cases*)
case *equal*
with *that* **show** *?thesis*
by *simp*

next
case *less*
with *that* **have** $b < 0$
by (*simp add: sgn-1-neg*)
with $\langle a < 0 \rangle$ **have** $a + b < 0$
by (*rule add-neg-neg*)
with $\langle a < 0 \rangle$ **show** *?thesis*
by *simp*

next
case *greater*
with *that* **have** $b > 0$
by (*simp add: sgn-1-pos*)
with $\langle a > 0 \rangle$ **have** $a + b > 0$
by (*rule add-pos-pos*)
with $\langle a > 0 \rangle$ **show** *?thesis*
by *simp*

qed

lemma *same-sgn-abs-add*:
 $|a + b| = |a| + |b|$ **if** $\text{sgn } b = \text{sgn } a$

proof –

have $a + b = \text{sgn } a * |a| + \text{sgn } b * |b|$
by (*simp add: sgn-mult-abs*)
also have $\dots = \text{sgn } a * (|a| + |b|)$
using that by (*simp add: algebra-simps*)
finally show *?thesis*
by (*auto simp add: abs-mult*)

qed

lemma *sgn-not-eq-imp*:

$\text{sgn } a = - \text{sgn } b$ **if** $\text{sgn } b \neq \text{sgn } a$ **and** $\text{sgn } a \neq 0$ **and** $\text{sgn } b \neq 0$
using that by (*cases a < 0*) (*auto simp add: sgn-0-0 sgn-1-pos sgn-1-neg*)

lemma *abs-dvd-iff* [*simp*]: $|m| \text{ dvd } k \longleftrightarrow m \text{ dvd } k$

by (*simp add: abs-if*)

lemma *dvd-abs-iff* [*simp*]: $m \text{ dvd } |k| \longleftrightarrow m \text{ dvd } k$

by (*simp add: abs-if*)

lemma *dvd-if-abs-eq*: $|l| = |k| \implies l \text{ dvd } k$

by (*subst abs-dvd-iff [symmetric]*) *simp*

The following lemmas can be proven in more general structures, but are dangerous as simp rules in absence of $(- ?a = ?a) = (?a = (0::'a))$, $(- ?a < ?a) = ((0::'a) < ?a)$, $(- ?a \leq ?a) = ((0::'a) \leq ?a)$.

lemma *equation-minus-iff-1* [*simp, no-atp*]: $1 = - a \longleftrightarrow a = - 1$

by (*fact equation-minus-iff*)

lemma *minus-equation-iff-1* [*simp, no-atp*]: $- a = 1 \longleftrightarrow a = - 1$

by (*subst minus-equation-iff, auto*)

lemma *le-minus-iff-1* [*simp, no-atp*]: $1 \leq - b \longleftrightarrow b \leq - 1$

by (*fact le-minus-iff*)

lemma *minus-le-iff-1* [*simp, no-atp*]: $- a \leq 1 \longleftrightarrow - 1 \leq a$

by (*fact minus-le-iff*)

lemma *less-minus-iff-1* [*simp, no-atp*]: $1 < - b \longleftrightarrow b < - 1$

by (*fact less-minus-iff*)

lemma *minus-less-iff-1* [*simp, no-atp*]: $- a < 1 \longleftrightarrow - 1 < a$

by (*fact minus-less-iff*)

lemma *add-less-zeroD*:

shows $x+y < 0 \implies x < 0 \vee y < 0$

by (*auto simp: not-less intro: le-less-trans [of - x+y]*)

Is this really better than just rewriting with *abs-if*?

lemma *abs-split* [*no-atp*]: $\langle P |a| \longleftrightarrow (0 \leq a \longrightarrow P a) \wedge (a < 0 \longrightarrow P (- a)) \rangle$

by (force dest: order-less-le-trans simp add: abs-if linorder-not-less)

end

Reasoning about inequalities with division

context *linordered-semidom*
begin

lemma *less-add-one*: $a < a + 1$
proof –
 have $a + 0 < a + 1$
 by (blast intro: zero-less-one add-strict-left-mono)
 then show ?thesis by simp
qed

end

context *linordered-idom*
begin

lemma *mult-right-le-one-le*: $0 \leq x \implies 0 \leq y \implies y \leq 1 \implies x * y \leq x$
by (rule mult-left-le)

lemma *mult-left-le-one-le*: $0 \leq x \implies 0 \leq y \implies y \leq 1 \implies y * x \leq x$
by (auto simp add: mult-le-cancel-right2)

end

Absolute Value

context *linordered-idom*
begin

lemma *mult-sgn-abs*: $\text{sgn } x * |x| = x$
by (fact sgn-mult-abs)

lemma *abs-one*: $|1| = 1$
by (fact abs-1)

end

class *ordered-ring-abs* = *ordered-ring* + *ordered-ab-group-add-abs* +
 assumes *abs-eq-mult*:
 $(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \implies |a * b| = |a| * |b|$

context *linordered-idom*
begin

subclass *ordered-ring-abs*
by standard (auto simp: abs-if not-less mult-less-0-iff)

```

lemma abs-mult-self:  $|a| * |a| = a * a$ 
  by (fact abs-mult-self-eq)

lemma abs-mult-less:
  assumes ac:  $|a| < c$ 
    and bd:  $|b| < d$ 
  shows  $|a| * |b| < c * d$ 
proof –
  from ac have  $0 < c$ 
    by (blast intro: le-less-trans abs-ge-zero)
  with bd show ?thesis by (simp add: ac mult-strict-mono)
qed

lemma abs-less-iff:  $|a| < b \longleftrightarrow a < b \wedge - a < b$ 
  by (simp add: less-le abs-le-iff) (auto simp add: abs-iff)

lemma abs-mult-pos:  $0 \leq x \implies |y| * x = |y * x|$ 
  by (simp add: abs-mult)

lemma abs-mult-pos':  $0 \leq x \implies x * |y| = |x * y|$ 
  by (simp add: abs-mult)

lemma abs-diff-less-iff:  $|x - a| < r \longleftrightarrow a - r < x \wedge x < a + r$ 
  by (auto simp add: diff-less-eq ac-simps abs-less-iff)

lemma abs-diff-le-iff:  $|x - a| \leq r \longleftrightarrow a - r \leq x \wedge x \leq a + r$ 
  by (auto simp add: diff-le-eq ac-simps abs-le-iff)

lemma abs-add-one-gt-zero:  $0 < 1 + |x|$ 
  by (auto simp: abs-if not-less intro: zero-less-one add-strict-increasing less-trans)

end

16.8 Dioids

Dioids are the alternative extensions of semirings, a semiring can either be
a ring or a dioid but never both.

class dioid = semiring-1 + canonically-ordered-monoid-add
begin

subclass ordered-semiring
  by standard (auto simp: le-iff-add distrib-left distrib-right)

end

hide-fact (open) comm-mult-left-mono comm-mult-strict-left-mono distrib

```

```

code-identifier
  code-module Rings  $\rightarrow$  (SML) Arith and (OCaml) Arith and (Haskell) Arith
end

```

17 Natural numbers

```

theory Nat
imports Inductive Typedef Fun Rings
begin

```

17.1 Type *ind*

```

typedecl ind

```

```

axiomatization Zero-Rep :: ind and Suc-Rep :: ind  $\Rightarrow$  ind
  — The axiom of infinity in 2 parts:
  where Suc-Rep-inject: Suc-Rep  $x = \text{Suc-Rep } y \Rightarrow x = y$ 
  and Suc-Rep-not-Zero-Rep: Suc-Rep  $x \neq \text{Zero-Rep}$ 

```

17.2 Type *nat*

Type definition

```

inductive Nat :: ind  $\Rightarrow$  bool
  where
    Zero-RepI: Nat Zero-Rep
  | Suc-RepI: Nat  $i \Rightarrow \text{Nat } (\text{Suc-Rep } i)$ 

```

```

typedef nat = {n. Nat  $n$ }
morphisms Rep-Nat Abs-Nat
using Nat.Zero-RepI by auto

```

```

lemma Nat-Rep-Nat: Nat (Rep-Nat  $n$ )
  using Rep-Nat by simp

```

```

lemma Nat-Abs-Nat-inverse: Nat  $n \Rightarrow \text{Rep-Nat } (\text{Abs-Nat } n) = n$ 
  using Abs-Nat-inverse by simp

```

```

lemma Nat-Abs-Nat-inject: Nat  $n \Rightarrow \text{Nat } m \Rightarrow \text{Abs-Nat } n = \text{Abs-Nat } m \iff n = m$ 
  using Abs-Nat-inject by simp

```

```

instantiation nat :: zero
begin

```

```

definition Zero-nat-def:  $0 = \text{Abs-Nat } \text{Zero-Rep}$ 

```

```

instance ..

```

end

definition $Suc :: nat \Rightarrow nat$
where $Suc\ n = Abs-Nat\ (Suc-Rep\ (Rep-Nat\ n))$

lemma $Suc-not-Zero: Suc\ m \neq 0$
by (*simp add: Zero-nat-def Suc-def Suc-RepI Zero-RepI*
Nat-Abs-Nat-inject Suc-Rep-not-Zero-Rep Nat-Rep-Nat)

lemma $Zero-not-Suc: 0 \neq Suc\ m$
by (*rule not-sym*) (*rule Suc-not-Zero*)

lemma $Suc-Rep-inject': Suc-Rep\ x = Suc-Rep\ y \longleftrightarrow x = y$
by (*rule iffI, rule Suc-Rep-inject*) *simp-all*

lemma $nat-induct0$:
assumes $P\ 0$ **and** $\bigwedge n. P\ n \implies P\ (Suc\ n)$
shows $P\ n$
proof –
have $P\ (Abs-Nat\ (Rep-Nat\ n))$
using *assms unfolding Zero-nat-def Suc-def*
by (*iprover intro: Nat-Rep-Nat [THEN Nat.induct] elim: Nat-Abs-Nat-inverse*
[THEN subst])
then show *?thesis*
by (*simp add: Rep-Nat-inverse*)
qed

free-constructors $case-nat$ **for** $0 :: nat \mid Suc\ pred$
where $pred\ (0 :: nat) = (0 :: nat)$
proof *atomize-elim*
fix n
show $n = 0 \vee (\exists m. n = Suc\ m)$
by (*induction n rule: nat-induct0*) *auto*
next
fix $n\ m$
show $(Suc\ n = Suc\ m) = (n = m)$
by (*simp add: Suc-def Nat-Abs-Nat-inject Nat-Rep-Nat Suc-RepI Suc-Rep-inject'*
Rep-Nat-inject)
next
fix n
show $0 \neq Suc\ n$
by (*simp add: Suc-not-Zero*)
qed

— Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.
setup $\langle Sign.mandatory-path\ old \rangle$

old-rep-datatype $0 :: \text{nat } \text{Suc}$
by (*erule nat-induct0*) *auto*

setup $\langle \text{Sign.parent-path} \rangle$

— But erase the prefix for properties that are not generated by *free-constructors*.

setup $\langle \text{Sign.mandatory-path nat} \rangle$

declare $\text{old.nat.inject}[\text{iff del}]$
and $\text{old.nat.distinct}(1)[\text{simp del, induct-simp del}]$

lemmas $\text{induct} = \text{old.nat.induct}$

lemmas $\text{inducts} = \text{old.nat.inducts}$

lemmas $\text{rec} = \text{old.nat.rec}$

lemmas $\text{simps} = \text{nat.inject nat.distinct nat.case nat.rec}$

setup $\langle \text{Sign.parent-path} \rangle$

abbreviation $\text{rec-nat} :: 'a \Rightarrow (\text{nat} \Rightarrow 'a \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a$
where $\text{rec-nat} \equiv \text{old.rec-nat}$

declare $\text{nat.sel}[\text{code del}]$

hide-const (**open**) Nat.pred — hide everything related to the selector

hide-fact

nat.case-eq-if

nat.collapse

nat.expand

nat.sel

nat.exhaust-sel

nat.split-sel

nat.split-sel-asm

lemma nat-exhaust [*case-names 0 Suc, cases type: nat*]:
 $(y = 0 \Longrightarrow P) \Longrightarrow (\bigwedge \text{nat. } y = \text{Suc nat} \Longrightarrow P) \Longrightarrow P$
— for backward compatibility – names of variables differ
by (*rule old.nat.exhaust*)

lemma nat-induct [*case-names 0 Suc, induct type: nat*]:
fixes n
assumes $P\ 0$ **and** $\bigwedge n. P\ n \Longrightarrow P\ (\text{Suc } n)$
shows $P\ n$
— for backward compatibility – names of variables differ
using *assms* **by** (*rule nat.induct*)

hide-fact

nat-exhaust

nat-induct0

```

ML <
val nat-basic-lfp-sugar =
  let
    val ctr-sugar = the (Ctr-Sugar.ctr-sugar-of-global theory type-name <nat>);
    val recx = Logic.varify-types-global term <rec-nat>;
    val C = body-type (fastype-of recx);
  in
    {T = HOLogic.natT, fp-res-index = 0, C = C, fun-arg-Tsss = [], [[HOLogic.natT,
C]]},
    ctr-sugar = ctr-sugar, recx = recx, rec-thms = @{thms nat.rec}}
  end;
>

```

```

setup <
let
  fun basic-lfp-sugars-of - [typ <nat>] - - ctxt =
    ([], [0], [nat-basic-lfp-sugar], [], [], [], TrueI (*dummy*), [], false, ctxt)
  | basic-lfp-sugars-of bs arg-Ts callers callssss ctxt =
    BNF-LFP-Rec-Sugar.default-basic-lfp-sugars-of bs arg-Ts callers callssss ctxt;
in
  BNF-LFP-Rec-Sugar.register-lfp-rec-extension
  {nested-simps = [], special-endgame-tac = K (K (K (K no-tac))), is-new-datatype
= K (K true),
  basic-lfp-sugars-of = basic-lfp-sugars-of, rewrite-nested-rec-call = NONE}
end
>

```

Injectiveness and distinctness lemmas

```

lemma inj-Suc [simp]:
  inj-on Suc N
  by (simp add: inj-on-def)

```

```

lemma bij-betw-Suc [simp]:
  bij-betw Suc M N  $\longleftrightarrow$  Suc ‘ M = N
  by (simp add: bij-betw-def)

```

```

lemma Suc-neq-Zero: Suc m = 0  $\implies$  R
  by (rule notE) (rule Suc-not-Zero)

```

```

lemma Zero-neq-Suc: 0 = Suc m  $\implies$  R
  by (rule Suc-neq-Zero) (erule sym)

```

```

lemma Suc-inject: Suc x = Suc y  $\implies$  x = y
  by (rule inj-Suc [THEN injD])

```

```

lemma n-not-Suc-n: n  $\neq$  Suc n
  by (induct n) simp-all

```

```

lemma Suc-n-not-n: Suc n  $\neq$  n

```

by (rule not-sym) (rule n-not-Suc-n)

A special form of induction for reasoning about $m < n$ and $m - n$.

```

lemma diff-induct:
  assumes  $\bigwedge x. P\ x\ 0$ 
    and  $\bigwedge y. P\ 0\ (Suc\ y)$ 
    and  $\bigwedge x\ y. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y)$ 
  shows  $P\ m\ n$ 
proof (induct n arbitrary: m)
  case 0
  show ?case by (rule assms(1))
next
  case (Suc n)
  show ?case
  proof (induct m)
  case 0
  show ?case by (rule assms(2))
  next
  case (Suc m)
  from  $\langle P\ m\ n \rangle$  show ?case by (rule assms(3))
qed
qed

```

17.3 Arithmetic operators

```

instantiation nat :: comm-monoid-diff
begin

```

```

primrec plus-nat
  where
    add-0:  $0 + n = (n::nat)$ 
    | add-Suc:  $Suc\ m + n = Suc\ (m + n)$ 

```

```

lemma add-0-right [simp]:  $m + 0 = m$ 
for  $m :: nat$ 
by (induct m) simp-all

```

```

lemma add-Suc-right [simp]:  $m + Suc\ n = Suc\ (m + n)$ 
by (induct m) simp-all

```

```

declare add-0 [code]

```

```

lemma add-Suc-shift [code]:  $Suc\ m + n = m + Suc\ n$ 
by simp

```

```

primrec minus-nat
  where
    diff-0 [code]:  $m - 0 = (m::nat)$ 
    | diff-Suc:  $m - Suc\ n = (case\ m - n\ of\ 0 \Rightarrow 0\ |\ Suc\ k \Rightarrow k)$ 

```



```

declare diff-Suc [simp del]

lemma diff-0-eq-0 [simp, code]:  $0 - n = 0$ 
  for  $n :: \text{nat}$ 
  by (induct n) (simp-all add: diff-Suc)

lemma diff-Suc-Suc [simp, code]:  $\text{Suc } m - \text{Suc } n = m - n$ 
  by (induct n) (simp-all add: diff-Suc)

instance
proof
  fix  $n\ m\ q :: \text{nat}$ 
  show  $(n + m) + q = n + (m + q)$  by (induct n) simp-all
  show  $n + m = m + n$  by (induct n) simp-all
  show  $m + n - m = n$  by (induct m) simp-all
  show  $n - m - q = n - (m + q)$  by (induct q) (simp-all add: diff-Suc)
  show  $0 + n = n$  by simp
  show  $0 - n = 0$  by simp
qed

end

hide-fact (open) add-0 add-0-right diff-0

instantiation  $\text{nat} :: \text{comm-semiring-1-cancel}$ 
begin

definition One-nat-def [simp]:  $1 = \text{Suc } 0$ 

primrec times-nat
  where
    mult-0:  $0 * n = (0 :: \text{nat})$ 
    | mult-Suc:  $\text{Suc } m * n = n + (m * n)$ 

lemma mult-0-right [simp]:  $m * 0 = 0$ 
  for  $m :: \text{nat}$ 
  by (induct m) simp-all

lemma mult-Suc-right [simp]:  $m * \text{Suc } n = m + (m * n)$ 
  by (induct m) (simp-all add: add.left-commute)

lemma add-mult-distrib:  $(m + n) * k = (m * k) + (n * k)$ 
  for  $m\ n\ k :: \text{nat}$ 
  by (induct m) (simp-all add: add.assoc)

instance
proof
  fix  $k\ n\ m\ q :: \text{nat}$ 

```

```

show 0 ≠ (1::nat)
  by simp
show 1 * n = n
  by simp
show n * m = m * n
  by (induct n) simp-all
show (n * m) * q = n * (m * q)
  by (induct n) (simp-all add: add-mult-distrib)
show (n + m) * q = n * q + m * q
  by (rule add-mult-distrib)
show k * (m - n) = (k * m) - (k * n)
  by (induct m n rule: diff-induct) simp-all
qed

end

```

17.3.1 Addition

Reasoning about $m + 0 = 0$, etc.

```

lemma add-is-0 [iff]: m + n = 0 ↔ m = 0 ∧ n = 0
  for m n :: nat
  by (cases m) simp-all

```

```

lemma add-is-1: m + n = Suc 0 ↔ m = Suc 0 ∧ n = 0 ∨ m = 0 ∧ n = Suc 0
  by (cases m) simp-all

```

```

lemma one-is-add: Suc 0 = m + n ↔ m = Suc 0 ∧ n = 0 ∨ m = 0 ∧ n =
Suc 0
  by (rule trans, rule eq-commute, rule add-is-1)

```

```

lemma add-eq-self-zero: m + n = m ⇒ n = 0
  for m n :: nat
  by (induct m) simp-all

```

```

lemma plus-1-eq-Suc:
  plus 1 = Suc
  by (simp add: fun-eq-iff)

```

```

lemma Suc-eq-plus1: Suc n = n + 1
  by simp

```

```

lemma Suc-eq-plus1-left: Suc n = 1 + n
  by simp

```

17.3.2 Difference

```

lemma Suc-diff-diff [simp]: (Suc m - n) - Suc k = m - n - k
  by (simp add: diff-diff-add)

```

lemma *diff-Suc-1* [*simp*]: $Suc\ n - 1 = n$
 by *simp*

17.3.3 Multiplication

lemma *mult-is-0* [*simp*]: $m * n = 0 \longleftrightarrow m = 0 \vee n = 0$ **for** $m\ n :: nat$
 by (*induct m*) *auto*

lemma *mult-eq-1-iff* [*simp*]: $m * n = Suc\ 0 \longleftrightarrow m = Suc\ 0 \wedge n = Suc\ 0$

proof (*induct m*)

case 0

then show ?case by *simp*

next

case (*Suc m*)

then show ?case by (*induct n*) *auto*

qed

lemma *one-eq-mult-iff* [*simp*]: $Suc\ 0 = m * n \longleftrightarrow m = Suc\ 0 \wedge n = Suc\ 0$
 by (*simp add: eq-commute flip: mult-eq-1-iff*)

lemma *nat-mult-eq-1-iff* [*simp*]: $m * n = 1 \longleftrightarrow m = 1 \wedge n = 1$

and *nat-1-eq-mult-iff* [*simp*]: $1 = m * n \longleftrightarrow m = 1 \wedge n = 1$ **for** $m\ n :: nat$
 by *auto*

lemma *mult-cancel1* [*simp*]: $k * m = k * n \longleftrightarrow m = n \vee k = 0$
 for $k\ m\ n :: nat$

proof –

have $k \neq 0 \implies k * m = k * n \implies m = n$

proof (*induct n arbitrary: m*)

case 0

then show $m = 0$ by *simp*

next

case (*Suc n*)

then show $m = Suc\ n$

by (*cases m*) (*simp-all add: eq-commute [of 0]*)

qed

then show ?thesis by *auto*

qed

lemma *mult-cancel2* [*simp*]: $m * k = n * k \longleftrightarrow m = n \vee k = 0$

for $k\ m\ n :: nat$

by (*simp add: mult.commute*)

lemma *Suc-mult-cancel1*: $Suc\ k * m = Suc\ k * n \longleftrightarrow m = n$

by (*subst mult-cancel1*) *simp*

17.4 Orders on *nat*

17.4.1 Operation definition

instantiation *nat* :: *linorder*

begin

primrec *less-eq-nat*

where

$(0::nat) \leq n \longleftrightarrow True$

| $Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*]

lemma *le0* [*iff*]: $0 \leq n$ **for**

n :: *nat*

by (*simp add: less-eq-nat.simps*)

lemma [*code*]: $0 \leq n \longleftrightarrow True$

for *n* :: *nat*

by *simp*

definition *less-nat*

where *less-eq-Suc-le*: $n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$

by (*simp add: less-eq-nat.simps(2)*)

lemma *Suc-le-eq* [*code*]: $Suc\ m \leq n \longleftrightarrow m < n$

unfolding *less-eq-Suc-le* ..

lemma *le-0-eq* [*iff*]: $n \leq 0 \longleftrightarrow n = 0$

for *n* :: *nat*

by (*induct n*) (*simp-all add: less-eq-nat.simps(2)*)

lemma *not-less0* [*iff*]: $\neg n < 0$

for *n* :: *nat*

by (*simp add: less-eq-Suc-le*)

lemma *less-nat-zero-code* [*code*]: $n < 0 \longleftrightarrow False$

for *n* :: *nat*

by *simp*

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$

by (*simp add: less-eq-Suc-le*)

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$

by (*simp add: less-eq-Suc-le*)

lemma *Suc-less-eq2*: $Suc\ n < m \longleftrightarrow (\exists m'. m = Suc\ m' \wedge n < m')$

by (cases m) auto

lemma *le-SucI*: $m \leq n \implies m \leq \text{Suc } n$
 by (induct m arbitrary: n) (simp-all add: less-eq-nat.simps(2) split: nat.splits)

lemma *Suc-leD*: $\text{Suc } m \leq n \implies m \leq n$
 by (cases n) (auto intro: le-SucI)

lemma *less-SucI*: $m < n \implies m < \text{Suc } n$
 by (simp add: less-eq-Suc-le) (erule Suc-leD)

lemma *Suc-lessD*: $\text{Suc } m < n \implies m < n$
 by (simp add: less-eq-Suc-le) (erule Suc-leD)

instance

proof

fix n m q :: nat

show $n < m \longleftrightarrow n \leq m \wedge \neg m \leq n$

proof (induct n arbitrary: m)

case 0

then show ?case

by (cases m) (simp-all add: less-eq-Suc-le)

next

case (Suc n)

then show ?case

by (cases m) (simp-all add: less-eq-Suc-le)

qed

show $n \leq n$

by (induct n) simp-all

then show $n = m$ if $n \leq m$ and $m \leq n$

using that by (induct n arbitrary: m)

(simp-all add: less-eq-nat.simps(2) split: nat.splits)

show $n \leq q$ if $n \leq m$ and $m \leq q$

using that

proof (induct n arbitrary: m q)

case 0

show ?case by simp

next

case (Suc n)

then show ?case

by (simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,

simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,

simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits)

qed

show $n \leq m \vee m \leq n$

by (induct n arbitrary: m)

(simp-all add: less-eq-nat.simps(2) split: nat.splits)

qed

end

instantiation *nat* :: *order-bot*
begin

definition *bot-nat* :: *nat*
 where *bot-nat* = 0

instance
 by *standard* (*simp add: bot-nat-def*)

end

instance *nat* :: *no-top*
 by *standard* (*auto intro: less-Suc-eq-le [THEN iffD2]*)

17.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < \text{Suc } n$
 by (*simp add: less-Suc-eq-le*)

lemma *zero-less-Suc* [*iff*]: $0 < \text{Suc } n$
 by (*simp add: less-Suc-eq-le*)

17.4.3 Elimination properties

lemma *less-not-refl*: $\neg n < n$
 for $n :: \text{nat}$
 by (*rule order-less-irrefl*)

lemma *less-not-refl2*: $n < m \implies m \neq n$
 for $m n :: \text{nat}$
 by (*rule not-sym*) (*rule less-imp-neq*)

lemma *less-not-refl3*: $s < t \implies s \neq t$
 for $s t :: \text{nat}$
 by (*rule less-imp-neq*)

lemma *less-irrefl-nat*: $n < n \implies R$
 for $n :: \text{nat}$
 by (*rule notE*, *rule less-not-refl*)

lemma *less-zeroE*: $n < 0 \implies R$
 for $n :: \text{nat}$
 by (*rule notE*) (*rule not-less0*)

lemma *less-Suc-eq*: $m < \text{Suc } n \iff m < n \vee m = n$
 unfolding *less-Suc-eq-le le-less ..*

lemma *less-Suc0* [*iff*]: $(n < \text{Suc } 0) = (n = 0)$

by (*simp add: less-Suc-eq*)

lemma *less-one [iff]*: $n < 1 \longleftrightarrow n = 0$
 for $n :: \text{nat}$
 unfolding *One-nat-def* by (*rule less-Suc0*)

lemma *Suc-mono*: $m < n \implies \text{Suc } m < \text{Suc } n$
 by *simp*

"Less than" is antisymmetric, sort of.

lemma *less-antisym*: $\neg n < m \implies n < \text{Suc } m \implies m = n$
 unfolding *not-less less-Suc-eq-le* by (*rule antisym*)

lemma *nat-neq-iff*: $m \neq n \longleftrightarrow m < n \vee n < m$
 for $m n :: \text{nat}$
 by (*rule linorder-neq-iff*)

17.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \implies \text{Suc } m \neq n \implies \text{Suc } m < n$
 unfolding *less-eq-Suc-le [of m] le-less* by *simp*

lemma *lessE*:
 assumes *major*: $i < k$
 and 1: $k = \text{Suc } i \implies P$
 and 2: $\bigwedge j. i < j \implies k = \text{Suc } j \implies P$
 shows P
proof –
 from *major* have $\exists j. i \leq j \wedge k = \text{Suc } j$
 unfolding *less-eq-Suc-le* by (*induct k*) *simp-all*
 then have $(\exists j. i < j \wedge k = \text{Suc } j) \vee k = \text{Suc } i$
 by (*auto simp add: less-le*)
 with 1 2 show P by *auto*
qed

lemma *less-SucE*:
 assumes *major*: $m < \text{Suc } n$
 and *less*: $m < n \implies P$
 and *eq*: $m = n \implies P$
 shows P
proof (*rule major [THEN lessE]*)
 show $\text{Suc } n = \text{Suc } m \implies P$
 using *eq* by *blast*
 show $\bigwedge j. [m < j; \text{Suc } n = \text{Suc } j] \implies P$
 by (*blast intro: less*)
qed

lemma *Suc-lessE*:
 assumes *major*: $\text{Suc } i < k$

and minor: $\bigwedge j. i < j \implies k = \text{Suc } j \implies P$
shows P
proof (*rule major* [THEN *lessE*])
show $k = \text{Suc } (\text{Suc } i) \implies P$
using *lessI minor by iprover*
show $\bigwedge j. [\text{Suc } i < j; k = \text{Suc } j] \implies P$
using *Suc-lessD minor by iprover*
qed

lemma *Suc-less-SucD*: $\text{Suc } m < \text{Suc } n \implies m < n$
by *simp*

lemma *less-trans-Suc*:
assumes *le*: $i < j$
shows $j < k \implies \text{Suc } i < k$
proof (*induct k*)
case 0
then show *?case by simp*
next
case (*Suc k*)
with *le show ?case*
by *simp (auto simp add: less-Suc-eq dest: Suc-lessD)*
qed

Can be used with *less-Suc-eq* to get $n = m \vee n < m$.

lemma *not-less-eq*: $\neg m < n \longleftrightarrow n < \text{Suc } m$
by (*simp only: not-less less-Suc-eq-le*)

lemma *not-less-eq-eq*: $\neg m \leq n \longleftrightarrow \text{Suc } n \leq m$
by (*simp only: not-le Suc-le-eq*)

Properties of "less than or equal".

lemma *le-imp-less-Suc*: $m \leq n \implies m < \text{Suc } n$
by (*simp only: less-Suc-eq-le*)

lemma *Suc-n-not-le-n*: $\neg \text{Suc } n \leq n$
by (*simp add: not-le less-Suc-eq-le*)

lemma *le-Suc-eq*: $m \leq \text{Suc } n \longleftrightarrow m \leq n \vee m = \text{Suc } n$
by (*simp add: less-Suc-eq-le [symmetric] less-Suc-eq*)

lemma *le-SucE*: $m \leq \text{Suc } n \implies (m \leq n \implies R) \implies (m = \text{Suc } n \implies R) \implies R$
by (*drule le-Suc-eq [THEN iffD1], iprover+*)

lemma *Suc-leI*: $m < n \implies \text{Suc } m \leq n$
by (*simp only: Suc-le-eq*)

Stronger version of *Suc-leD*.

lemma *Suc-le-lessD*: $\text{Suc } m \leq n \implies m < n$

by (*simp only: Suc-le-eq*)

lemma *less-imp-le-nat*: $m < n \implies m \leq n$ **for** $m\ n :: \text{nat}$
unfolding *less-eq-Suc-le* **by** (*rule Suc-leD*)

For instance, $(\text{Suc } m < \text{Suc } n) = (\text{Suc } m \leq n) = (m < n)$

lemmas *le-simps = less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \vee m = n \implies m \leq n$
for $m\ n :: \text{nat}$
unfolding *le-less* .

lemma *le-eq-less-or-eq*: $m \leq n \iff m < n \vee m = n$
for $m\ n :: \text{nat}$
by (*rule le-less*)

Useful with *blast*.

lemma *eq-imp-le*: $m = n \implies m \leq n$
for $m\ n :: \text{nat}$
by *auto*

lemma *le-refl*: $n \leq n$
for $n :: \text{nat}$
by *simp*

lemma *le-trans*: $i \leq j \implies j \leq k \implies i \leq k$
for $i\ j\ k :: \text{nat}$
by (*rule order-trans*)

lemma *le-antisym*: $m \leq n \implies n \leq m \implies m = n$
for $m\ n :: \text{nat}$
by (*rule antisym*)

lemma *nat-less-le*: $m < n \iff m \leq n \wedge m \neq n$
for $m\ n :: \text{nat}$
by (*rule less-le*)

lemma *le-neq-implies-less*: $m \leq n \implies m \neq n \implies m < n$
for $m\ n :: \text{nat}$
unfolding *less-le ..*

lemma *nat-le-linear*: $m \leq n \vee n \leq m$
for $m\ n :: \text{nat}$
by (*rule linear*)

lemmas *linorder-neqE-nat = linorder-neqE* [**where** $'a = \text{nat}$]

lemma *le-less-Suc-eq*: $m \leq n \implies n < \text{Suc } m \iff n = m$

unfolding *less-Suc-eq-le* **by** *auto*

lemma *not-less-less-Suc-eq*: $\neg n < m \implies n < \text{Suc } m \longleftrightarrow n = m$
unfolding *not-less* **by** (*rule le-less-Suc-eq*)

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

lemma *not0-implies-Suc*: $n \neq 0 \implies \exists m. n = \text{Suc } m$
by (*cases n*) *simp-all*

lemma *gr0-implies-Suc*: $n > 0 \implies \exists m. n = \text{Suc } m$
by (*cases n*) *simp-all*

lemma *gr-implies-not0*: $m < n \implies n \neq 0$
for $m\ n :: \text{nat}$
by (*cases n*) *simp-all*

lemma *neq0-conv[iff]*: $n \neq 0 \longleftrightarrow 0 < n$
for $n :: \text{nat}$
by (*cases n*) *simp-all*

This theorem is useful with *blast*

lemma *gr0I*: $(n = 0 \implies \text{False}) \implies 0 < n$
for $n :: \text{nat}$
by (*rule neq0-conv[THEN iffD1]*) *iprover*

lemma *gr0-conv-Suc*: $0 < n \longleftrightarrow (\exists m. n = \text{Suc } m)$
by (*fast intro: not0-implies-Suc*)

lemma *not-gr0 [iff]*: $\neg 0 < n \longleftrightarrow n = 0$
for $n :: \text{nat}$
using *neq0-conv* **by** *blast*

lemma *Suc-le-D*: $\text{Suc } n \leq m' \implies \exists m. m' = \text{Suc } m$
by (*induct m'*) *simp-all*

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $m < \text{Suc } n \longleftrightarrow m = 0 \vee (\exists j. m = \text{Suc } j \wedge j < n)$
by (*cases m*) *simp-all*

lemma *All-less-Suc*: $(\forall i < \text{Suc } n. P\ i) = (P\ n \wedge (\forall i < n. P\ i))$
by (*auto simp: less-Suc-eq*)

lemma *All-less-Suc2*: $(\forall i < \text{Suc } n. P\ i) = (P\ 0 \wedge (\forall i < n. P(\text{Suc } i)))$
by (*auto simp: less-Suc-eq-0-disj*)

lemma *Ex-less-Suc*: $(\exists i < \text{Suc } n. P\ i) = (P\ n \vee (\exists i < n. P\ i))$
by (*auto simp: less-Suc-eq*)

lemma *Ex-less-Suc2*: $(\exists i < \text{Suc } n. P i) = (P 0 \vee (\exists i < n. P(\text{Suc } i)))$
by (*auto simp: less-Suc-eq-0-disj*)

mono (non-strict) doesn't imply increasing, as the function could be constant

lemma *strict-mono-imp-increasing*:
fixes $n::\text{nat}$
assumes *strict-mono* f **shows** $f n \geq n$
proof (*induction n*)
case 0
then show *?case*
by *auto*
next
case $(\text{Suc } n)$
then show *?case*
unfolding *not-less-eq-eq* [*symmetric*]
using *Suc-n-not-le-n* *assms* *order-trans* *strict-mono-less-eq* **by** *blast*
qed

17.4.5 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: $n > 0 \implies \text{Suc } (n - \text{Suc } 0) = n$
by (*simp add: diff-Suc split: nat.split*)

lemma *Suc-diff-1* [*simp*]: $0 < n \implies \text{Suc } (n - 1) = n$
unfolding *One-nat-def* **by** (*rule Suc-pred*)

lemma *nat-add-left-cancel-le* [*simp*]: $k + m \leq k + n \longleftrightarrow m \leq n$
for $k m n :: \text{nat}$
by (*induct k*) *simp-all*

lemma *nat-add-left-cancel-less* [*simp*]: $k + m < k + n \longleftrightarrow m < n$
for $k m n :: \text{nat}$
by (*induct k*) *simp-all*

lemma *add-gr-0* [*iff*]: $m + n > 0 \longleftrightarrow m > 0 \vee n > 0$
for $m n :: \text{nat}$
by (*auto dest: gr0-implies-Suc*)

strict, in 1st argument

lemma *add-less-mono1*: $i < j \implies i + k < j + k$
for $i j k :: \text{nat}$
by (*induct k*) *simp-all*

strict, in both arguments

lemma *add-less-mono*:
fixes $i j k l :: \text{nat}$
assumes $i < j$ $k < l$ **shows** $i + k < j + l$
proof –
have $i + k < j + k$

```

  by (simp add: add-less-mono1 assms)
  also have ... < j + l
  using <i < j> by (induction j) (auto simp: assms)
  finally show ?thesis .
qed

```

```

lemma less-imp-Suc-add:  $m < n \implies \exists k. n = \text{Suc } (m + k)$ 
proof (induct n)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case
    by (simp add: order-le-less)
    (blast elim!: less-SucE intro!: Nat.add-0-right [symmetric] add-Suc-right [symmetric])
qed

```

```

lemma le-Suc-ex:  $k \leq l \implies (\exists n. l = k + n)$ 
for  $k\ l :: \text{nat}$ 
by (auto simp: less-Suc-eq-le[symmetric] dest: less-imp-Suc-add)

```

```

lemma less-natE:
  assumes  $\langle m < n \rangle$ 
  obtains  $q$  where  $\langle n = \text{Suc } (m + q) \rangle$ 
  using assms by (auto dest: less-imp-Suc-add intro: that)

```

strict, in 1st argument; proof is by induction on $k > 0$

```

lemma mult-less-mono2:
  fixes  $i\ j :: \text{nat}$ 
  assumes  $i < j$  and  $0 < k$ 
  shows  $k * i < k * j$ 
  using  $\langle 0 < k \rangle$ 
proof (induct k)
  case 0
  then show ?case by simp
next
  case (Suc k)
  with  $\langle i < j \rangle$  show ?case
    by (cases k) (simp-all add: add-less-mono)
qed

```

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

```

lemma add-diff-inverse-nat:  $\neg m < n \implies n + (m - n) = m$ 
for  $m\ n :: \text{nat}$ 
by (induct m n rule: diff-induct) simp-all

```

```

lemma nat-le-iff-add:  $m \leq n \iff (\exists k. n = m + k)$ 
for  $m\ n :: \text{nat}$ 
using nat-add-left-cancel-le[of m 0] by (auto dest: le-Suc-ex)

```

The naturals form an ordered *semidom* and a *dioid*.

instance *nat* :: *linordered-semidom*

proof

fix *m n q* :: *nat*

show $0 < (1::nat)$

by *simp*

show $m \leq n \implies q + m \leq q + n$

by *simp*

show $m < n \implies 0 < q \implies q * m < q * n$

by (*simp add: mult-less-mono2*)

show $m \neq 0 \implies n \neq 0 \implies m * n \neq 0$

by *simp*

show $n \leq m \implies (m - n) + n = m$

by (*simp add: add-diff-inverse-nat add commute linorder-not-less*)

qed

instance *nat* :: *dioid*

by *standard* (*rule nat-le-iff-add*)

declare *le0*[*simp del*] — This is now $(0::?'a) \leq ?x$

declare *le-0-eq*[*simp del*] — This is now $(?n \leq (0::?'a)) = (?n = (0::?'a))$

declare *not-less0*[*simp del*] — This is now $\neg ?n < (0::?'a)$

declare *not-gr0*[*simp del*] — This is now $(\neg (0::?'a) < ?n) = (?n = (0::?'a))$

instance *nat* :: *ordered-cancel-comm-monoid-add* ..

instance *nat* :: *ordered-cancel-comm-monoid-diff* ..

17.4.6 *min* and *max*

global-interpretation *bot-nat-0*: *ordering-top* $\langle (\geq) \rangle \langle (>) \rangle \langle 0::nat \rangle$

by *standard simp*

global-interpretation *max-nat*: *semilattice-neutr-order* *max* $\langle 0::nat \rangle \langle (\geq) \rangle \langle (>) \rangle$

by *standard* (*simp add: max-def*)

lemma *mono-Suc*: *mono Suc*

by (*rule monoI*) *simp*

lemma *min-0L* [*simp*]: $\min 0 n = 0$

for *n* :: *nat*

by (*rule min-absorb1*) *simp*

lemma *min-0R* [*simp*]: $\min n 0 = 0$

for *n* :: *nat*

by (*rule min-absorb2*) *simp*

lemma *min-Suc-Suc* [*simp*]: $\min (\text{Suc } m) (\text{Suc } n) = \text{Suc } (\min m n)$

by (*simp add: mono-Suc min-of-mono*)

lemma *min-Suc1*: $\min (\text{Suc } n) m = (\text{case } m \text{ of } 0 \Rightarrow 0 \mid \text{Suc } m' \Rightarrow \text{Suc}(\min n m'))$

by (*simp split: nat.split*)

lemma *min-Suc2*: $\min m (\text{Suc } n) = (\text{case } m \text{ of } 0 \Rightarrow 0 \mid \text{Suc } m' \Rightarrow \text{Suc}(\min m' n))$

by (*simp split: nat.split*)

lemma *max-0L* [*simp*]: $\max 0 n = n$

for $n :: \text{nat}$

by (*fact max-nat.left-neutral*)

lemma *max-0R* [*simp*]: $\max n 0 = n$

for $n :: \text{nat}$

by (*fact max-nat.right-neutral*)

lemma *max-Suc-Suc* [*simp*]: $\max (\text{Suc } m) (\text{Suc } n) = \text{Suc} (\max m n)$

by (*simp add: mono-Suc max-of-mono*)

lemma *max-Suc1*: $\max (\text{Suc } n) m = (\text{case } m \text{ of } 0 \Rightarrow \text{Suc } n \mid \text{Suc } m' \Rightarrow \text{Suc} (\max n m'))$

by (*simp split: nat.split*)

lemma *max-Suc2*: $\max m (\text{Suc } n) = (\text{case } m \text{ of } 0 \Rightarrow \text{Suc } n \mid \text{Suc } m' \Rightarrow \text{Suc} (\max m' n))$

by (*simp split: nat.split*)

lemma *nat-mult-min-left*: $\min m n * q = \min (m * q) (n * q)$

for $m n q :: \text{nat}$

by (*simp add: min-def not-le*)

(*auto dest: mult-right-le-imp-le mult-right-less-imp-less le-less-trans*)

lemma *nat-mult-min-right*: $m * \min n q = \min (m * n) (m * q)$

for $m n q :: \text{nat}$

by (*simp add: min-def not-le*)

(*auto dest: mult-left-le-imp-le mult-left-less-imp-less le-less-trans*)

lemma *nat-add-max-left*: $\max m n + q = \max (m + q) (n + q)$

for $m n q :: \text{nat}$

by (*simp add: max-def*)

lemma *nat-add-max-right*: $m + \max n q = \max (m + n) (m + q)$

for $m n q :: \text{nat}$

by (*simp add: max-def*)

lemma *nat-mult-max-left*: $\max m n * q = \max (m * q) (n * q)$

for $m n q :: \text{nat}$

by (*simp add: max-def not-le*)

(*auto dest: mult-right-le-imp-le mult-right-less-imp-less le-less-trans*)

```

lemma nat-mult-max-right:  $m * \max n q = \max (m * n) (m * q)$ 
  for  $m n q :: \text{nat}$ 
  by (simp add: max-def not-le)
      (auto dest: mult-left-le-imp-le mult-left-less-imp-less le-less-trans)

```

17.4.7 Additional theorems about (\leq)

Complete induction, aka course-of-values induction

```

instance nat :: wellorder

```

```

proof

```

```

  fix  $P$  and  $n :: \text{nat}$ 

```

```

  assume step:  $(\bigwedge m. m < n \implies P m) \implies P n$  for  $n :: \text{nat}$ 

```

```

  have  $\bigwedge q. q \leq n \implies P q$ 

```

```

  proof (induct n)

```

```

    case  $(0 n)$ 

```

```

      have  $P 0$  by (rule step) auto

```

```

      with  $0$  show ?case by auto

```

```

  next

```

```

    case  $(\text{Suc } m n)$ 

```

```

      then have  $n \leq m \vee n = \text{Suc } m$ 

```

```

        by (simp add: le-Suc-eq)

```

```

      then show ?case

```

```

        proof

```

```

          assume  $n \leq m$ 

```

```

          then show  $P n$  by (rule Suc(1))

```

```

        next

```

```

          assume  $n: n = \text{Suc } m$ 

```

```

          show  $P n$  by (rule step) (rule Suc(1), simp add: n le-simps)

```

```

        qed

```

```

      qed

```

```

      then show  $P n$  by auto

```

```

qed

```

```

lemma Least-eq-0[simp]:  $P 0 \implies \text{Least } P = 0$ 

```

```

  for  $P :: \text{nat} \Rightarrow \text{bool}$ 

```

```

  by (rule Least-equality[OF - le0])

```

```

lemma Least-Suc:

```

```

  assumes  $P n \neg P 0$ 

```

```

  shows  $(\text{LEAST } n. P n) = \text{Suc } (\text{LEAST } m. P (\text{Suc } m))$ 

```

```

proof (cases n)

```

```

  case  $(\text{Suc } m)$ 

```

```

  show ?thesis

```

```

  proof (rule antisym)

```

```

    show  $(\text{LEAST } x. P x) \leq \text{Suc } (\text{LEAST } x. P (\text{Suc } x))$ 

```

```

      using assms Suc by (force intro: LeastI Least-le)

```

```

    have  $\S: P (\text{LEAST } x. P x)$ 

```

```

    by (blast intro: LeastI assms)
  show Suc (LEAST m. P (Suc m)) ≤ (LEAST n. P n)
  proof (cases (LEAST n. P n))
    case 0
    then show ?thesis
      using § by (simp add: assms)
  next
    case Suc
    with § show ?thesis
      by (auto simp: Least-le)
  qed
qed
qed (use assms in auto)

```

lemma *Least-Suc2*: $P\ n \implies Q\ m \implies \neg P\ 0 \implies \forall k. P\ (Suc\ k) = Q\ k \implies Least\ P = Suc\ (Least\ Q)$
 by (erule (1) *Least-Suc [THEN ssubst]*) *simp*

lemma *ex-least-nat-le*:
 fixes $P :: nat \Rightarrow bool$
 assumes $P\ n \neg P\ 0$
 shows $\exists k \leq n. (\forall i < k. \neg P\ i) \wedge P\ k$
proof (*cases n*)
 case (*Suc m*)
 with *assms* **show** ?thesis
 by (*blast intro: Least-le LeastI-ex dest: not-less-Least*)
qed (*use assms in auto*)

lemma *ex-least-nat-less*:
 fixes $P :: nat \Rightarrow bool$
 assumes $P\ n \neg P\ 0$
 shows $\exists k < n. (\forall i \leq k. \neg P\ i) \wedge P\ (Suc\ k)$
proof (*cases n*)
 case (*Suc m*)
 then **obtain** k **where** $k \leq n \wedge \forall i < k. \neg P\ i \wedge P\ k$
 using *ex-least-nat-le [OF assms]* **by** *blast*
show ?thesis
 by (*cases k*) (*use assms k less-eq-Suc-le in auto*)
qed (*use assms in auto*)

lemma *nat-less-induct*:
 fixes $P :: nat \Rightarrow bool$
 assumes $\bigwedge n. \forall m. m < n \longrightarrow P\ m \implies P\ n$
 shows $P\ n$
 using *assms less-induct* **by** *blast*

lemma *measure-induct-rule [case-names less]*:
 fixes $f :: 'a \Rightarrow 'b::wellorder$

assumes *step*: $\bigwedge x. (\bigwedge y. f\ y < f\ x \implies P\ y) \implies P\ x$
shows $P\ a$
by (*induct* $m \equiv f\ a$ *arbitrary*: *a rule: less-induct*) (*auto intro*: *step*)

old style induction rules:

lemma *measure-induct*:
fixes $f :: 'a \Rightarrow 'b::wellorder$
shows $(\bigwedge x. \forall y. f\ y < f\ x \longrightarrow P\ y \implies P\ x) \implies P\ a$
by (*rule measure-induct-rule* [*of f P a*]) *iprover*

lemma *full-nat-induct*:
assumes *step*: $\bigwedge n. (\forall m. Suc\ m \leq n \longrightarrow P\ m) \implies P\ n$
shows $P\ n$
by (*rule less-induct*) (*auto intro*: *step simp:le-simps*)

An induction rule for establishing binary relations

lemma *less-Suc-induct* [*consumes 1*]:
assumes *less*: $i < j$
and *step*: $\bigwedge i. P\ i\ (Suc\ i)$
and *trans*: $\bigwedge i\ j\ k. i < j \implies j < k \implies P\ i\ j \implies P\ j\ k \implies P\ i\ k$
shows $P\ i\ j$

proof –

from *less* **obtain** k **where** $j = Suc\ (i + k)$
by (*auto dest: less-imp-Suc-add*)
have $P\ i\ (Suc\ (i + k))$
proof (*induct k*)
case 0
show *?case* **by** (*simp add: step*)
next
case $(Suc\ k)$
have $0 + i < Suc\ k + i$ **by** (*rule add-less-mono1*) *simp*
then have $i < Suc\ (i + k)$ **by** (*simp add: add.commute*)
from *trans*[*OF this lessI Suc step*]
show *?case* **by** *simp*
qed
then show $P\ i\ j$ **by** (*simp add: j*)

qed

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. $P\ n$ is true for all natural numbers if

- case “0”: given $n = 0$ prove $P\ n$
- case “smaller”: given $n > 0$ and $\neg P\ n$ prove there exists a smaller natural number m such that $\neg P\ m$.

lemma *infinite-descent*: $(\bigwedge n. \neg P\ n \implies \exists m < n. \neg P\ m) \implies P\ n$ **for** $P :: nat \Rightarrow bool$

— compact version without explicit base case
by (*induct n rule: less-induct*) *auto*

lemma *infinite-descent0* [*case-names 0 smaller*]:
fixes $P :: nat \Rightarrow bool$
assumes $P\ 0$
and $\bigwedge n. n > 0 \Longrightarrow \neg P\ n \Longrightarrow \exists m. m < n \wedge \neg P\ m$
shows $P\ n$
proof (*rule infinite-descent*)
fix n
show $\neg P\ n \Longrightarrow \exists m < n. \neg P\ m$
using *assms* **by** (*cases n > 0*) *auto*
qed

Infinite descent using a mapping to *nat*: $P\ x$ is true for all $x \in D$ if there exists a $V \in D \Rightarrow nat$ and

- case “0”: given $V\ x = 0$ prove $P\ x$
- “smaller”: given $V\ x > 0$ and $\neg P\ x$ prove there exists a $y \in D$ such that $V\ y < V\ x$ and $\neg P\ y$.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
fixes $V :: 'a \Rightarrow nat$
assumes 1: $\bigwedge x. V\ x = 0 \Longrightarrow P\ x$
and 2: $\bigwedge x. V\ x > 0 \Longrightarrow \neg P\ x \Longrightarrow \exists y. V\ y < V\ x \wedge \neg P\ y$
shows $P\ x$
proof –
obtain n **where** $n = V\ x$ **by** *auto*
moreover **have** $\bigwedge x. V\ x = n \Longrightarrow P\ x$
proof (*induct n rule: infinite-descent0*)
case 0
with 1 **show** $P\ x$ **by** *auto*
next
case (*smaller n*)
then **obtain** x **where** *: $V\ x = n$ **and** $V\ x > 0 \wedge \neg P\ x$ **by** *auto*
with 2 **obtain** y **where** $V\ y < V\ x \wedge \neg P\ y$ **by** *auto*
with * **obtain** m **where** $m = V\ y \wedge m < n \wedge \neg P\ y$ **by** *auto*
then **show** *?case* **by** *auto*
qed
ultimately **show** $P\ x$ **by** *auto*
qed

Again, without explicit base case:

lemma *infinite-descent-measure*:
fixes $V :: 'a \Rightarrow nat$
assumes $\bigwedge x. \neg P\ x \Longrightarrow \exists y. V\ y < V\ x \wedge \neg P\ y$
shows $P\ x$
proof –

from *assms* **obtain** n **where** $n = V x$ **by** *auto*
moreover have $\bigwedge x. V x = n \implies P x$
proof –
have $\exists m < V x. \exists y. V y = m \wedge \neg P y$ **if** $\neg P x$ **for** x
using *assms* **and that by** *auto*
then show $\bigwedge x. V x = n \implies P x$
by (*induct n rule: infinite-descent, auto*)
qed
ultimately show $P x$ **by** *auto*
qed

A (clumsy) way of lifting $<$ monotonicity to \leq monotonicity

lemma *less-mono-imp-le-mono*:
fixes $f :: nat \Rightarrow nat$
and $i j :: nat$
assumes $\bigwedge i j :: nat. i < j \implies f i < f j$
and $i \leq j$
shows $f i \leq f j$
using *assms* **by** (*auto simp add: order-le-less*)

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j \implies i + k \leq j + k$
for $i j k :: nat$
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: $i \leq j \implies k \leq l \implies i + k \leq j + l$
for $i j k l :: nat$
by (*rule add-mono*)

lemma *le-add2*: $n \leq m + n$
for $m n :: nat$
by *simp*

lemma *le-add1*: $n \leq n + m$
for $m n :: nat$
by *simp*

lemma *less-add-Suc1*: $i < Suc (i + m)$
by (*rule le-less-trans, rule le-add1, rule lessI*)

lemma *less-add-Suc2*: $i < Suc (m + i)$
by (*rule le-less-trans, rule le-add2, rule lessI*)

lemma *less-iff-Suc-add*: $m < n \longleftrightarrow (\exists k. n = Suc (m + k))$
by (*iprover intro!: less-add-Suc1 less-imp-Suc-add*)

lemma *trans-le-add1*: $i \leq j \implies i \leq j + m$
for $i j m :: nat$

by (rule le-trans, assumption, rule le-add1)

lemma *trans-le-add2*: $i \leq j \implies i \leq m + j$
 for $i j m :: nat$
 by (rule le-trans, assumption, rule le-add2)

lemma *trans-less-add1*: $i < j \implies i < j + m$
 for $i j m :: nat$
 by (rule less-le-trans, assumption, rule le-add1)

lemma *trans-less-add2*: $i < j \implies i < m + j$
 for $i j m :: nat$
 by (rule less-le-trans, assumption, rule le-add2)

lemma *add-lessD1*: $i + j < k \implies i < k$
 for $i j k :: nat$
 by (rule le-less-trans [of - i+j]) (simp-all add: le-add1)

lemma *not-add-less1* [iff]: $\neg i + j < i$
 for $i j :: nat$
 by simp

lemma *not-add-less2* [iff]: $\neg j + i < i$
 for $i j :: nat$
 by simp

lemma *add-leD1*: $m + k \leq n \implies m \leq n$
 for $k m n :: nat$
 by (rule order-trans [of - m + k]) (simp-all add: le-add1)

lemma *add-leD2*: $m + k \leq n \implies k \leq n$
 for $k m n :: nat$
 by (force simp add: add.commute dest: add-leD1)

lemma *add-leE*: $m + k \leq n \implies (m \leq n \implies k \leq n \implies R) \implies R$
 for $k m n :: nat$
 by (blast dest: add-leD1 add-leD2)

needs $\bigwedge k$ for *ac-simps* to work

lemma *less-add-eq-less*: $\bigwedge k. k < l \implies m + l = k + n \implies m < n$
 for $l m n :: nat$
 by (force simp del: add-Suc-right simp add: less-iff-Suc-add add-Suc-right [symmetric] ac-simps)

17.4.8 More results about difference

lemma *Suc-diff-le*: $n \leq m \implies \text{Suc } m - n = \text{Suc } (m - n)$
 by (induct m n rule: diff-induct) simp-all

lemma *diff-less-Suc*: $m - n < \text{Suc } m$
by (*induct m n rule: diff-induct*) (*auto simp: less-Suc-eq*)

lemma *diff-le-self* [*simp*]: $m - n \leq m$
for $m n :: \text{nat}$
by (*induct m n rule: diff-induct*) (*simp-all add: le-SucI*)

lemma *less-imp-diff-less*: $j < k \implies j - n < k$
for $j k n :: \text{nat}$
by (*rule le-less-trans, rule diff-le-self*)

lemma *diff-Suc-less* [*simp*]: $0 < n \implies n - \text{Suc } i < n$
by (*cases n*) (*auto simp add: le-simps*)

lemma *diff-add-assoc*: $k \leq j \implies (i + j) - k = i + (j - k)$
for $i j k :: \text{nat}$
by (*fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc*)

lemma *add-diff-assoc* [*simp*]: $k \leq j \implies i + (j - k) = i + j - k$
for $i j k :: \text{nat}$
by (*fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc*)

lemma *diff-add-assoc2*: $k \leq j \implies (j + i) - k = (j - k) + i$
for $i j k :: \text{nat}$
by (*fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc2*)

lemma *add-diff-assoc2* [*simp*]: $k \leq j \implies j - k + i = j + i - k$
for $i j k :: \text{nat}$
by (*fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc2*)

lemma *le-imp-diff-is-add*: $i \leq j \implies (j - i = k) = (j = k + i)$
for $i j k :: \text{nat}$
by *auto*

lemma *diff-is-0-eq* [*simp*]: $m - n = 0 \iff m \leq n$
for $m n :: \text{nat}$
by (*induct m n rule: diff-induct*) *simp-all*

lemma *diff-is-0-eq'* [*simp*]: $m \leq n \implies m - n = 0$
for $m n :: \text{nat}$
by (*rule iffD2, rule diff-is-0-eq*)

lemma *zero-less-diff* [*simp*]: $0 < n - m \iff m < n$
for $m n :: \text{nat}$
by (*induct m n rule: diff-induct*) *simp-all*

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k :: \text{nat}. 0 < k \wedge i + k = j$

proof

from *assms* **show** $0 < j - i \wedge i + (j - i) = j$

by (*simp add: order-less-imp-le*)

qed

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*: $n - m + m = \max n m$

for $m n :: \text{nat}$

by (*simp add: max-def not-le order-less-imp-le*)

lemma *nat-diff-split*: $P (a - b) \longleftrightarrow (a < b \longrightarrow P 0) \wedge (\forall d. a = b + d \longrightarrow P d)$

for $a b :: \text{nat}$

— elimination of $-$ on *nat*

by (*cases a < b*) (*auto simp add: not-less le-less dest!: add-eq-self-zero [OF sym]*)

lemma *nat-diff-split-asm*: $P (a - b) \longleftrightarrow \neg (a < b \wedge \neg P 0 \vee (\exists d. a = b + d \wedge \neg P d))$

for $a b :: \text{nat}$

— elimination of $-$ on *nat* in assumptions

by (*auto split: nat-diff-split*)

lemma *Suc-pred'*: $0 < n \Longrightarrow n = \text{Suc}(n - 1)$

by *simp*

lemma *add-eq-if*: $m + n = (\text{if } m = 0 \text{ then } n \text{ else } \text{Suc}((m - 1) + n))$

unfolding *One-nat-def* **by** (*cases m*) *simp-all*

lemma *mult-eq-if*: $m * n = (\text{if } m = 0 \text{ then } 0 \text{ else } n + ((m - 1) * n))$

for $m n :: \text{nat}$

by (*cases m*) *simp-all*

lemma *Suc-diff-eq-diff-pred*: $0 < n \Longrightarrow \text{Suc } m - n = m - (n - 1)$

by (*cases n*) *simp-all*

lemma *diff-Suc-eq-diff-pred*: $m - \text{Suc } n = (m - 1) - n$

by (*cases m*) *simp-all*

lemma *Let-Suc* [*simp*]: $\text{Let } (\text{Suc } n) f \equiv f (\text{Suc } n)$

by (*fact Let-def*)

17.4.9 Monotonicity of multiplication

lemma *mult-le-mono1*: $i \leq j \Longrightarrow i * k \leq j * k$

for $i j k :: \text{nat}$

by (*simp add: mult-right-mono*)

lemma *mult-le-mono2*: $i \leq j \Longrightarrow k * i \leq k * j$

for $i j k :: \text{nat}$

by (*simp add: mult-left-mono*)

\leq monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq j \implies k \leq l \implies i * k \leq j * l$
for $i j k l :: nat$
by (*simp add: mult-mono*)

lemma *mult-less-mono1*: $i < j \implies 0 < k \implies i * k < j * k$
for $i j k :: nat$
by (*simp add: mult-strict-right-mono*)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $0 < m * n \longleftrightarrow 0 < m \wedge 0 < n$
for $m n :: nat$
proof (*induct m*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc m*)
then show ?*case* **by** (*cases n*) *simp-all*
qed

lemma *one-le-mult-iff* [*simp*]: $Suc\ 0 \leq m * n \longleftrightarrow Suc\ 0 \leq m \wedge Suc\ 0 \leq n$
proof (*induct m*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc m*)
then show ?*case* **by** (*cases n*) *simp-all*
qed

lemma *mult-less-cancel2* [*simp*]: $m * k < n * k \longleftrightarrow 0 < k \wedge m < n$
for $k m n :: nat$
proof (*intro iffI conjI*)
assume $m: m * k < n * k$
then show $0 < k$
by (*cases k*) *auto*
show $m < n$
proof (*cases k*)
case 0
then show ?*thesis*
using m **by** *auto*
next
case (*Suc k'*)
then show ?*thesis*
using m
by (*simp flip: linorder-not-le*) (*blast intro: add-mono mult-le-mono1*)
qed
next
assume $0 < k \wedge m < n$

```

then show  $m * k < n * k$ 
  by (blast intro: mult-less-mono1)
qed

```

```

lemma mult-less-cancel1 [simp]:  $k * m < k * n \longleftrightarrow 0 < k \wedge m < n$ 
  for  $k\ m\ n :: \text{nat}$ 
  by (simp add: mult.commute [of k])

```

```

lemma mult-le-cancel1 [simp]:  $k * m \leq k * n \longleftrightarrow (0 < k \longrightarrow m \leq n)$ 
  for  $k\ m\ n :: \text{nat}$ 
  by (simp add: linorder-not-less [symmetric], auto)

```

```

lemma mult-le-cancel2 [simp]:  $m * k \leq n * k \longleftrightarrow (0 < k \longrightarrow m \leq n)$ 
  for  $k\ m\ n :: \text{nat}$ 
  by (simp add: linorder-not-less [symmetric], auto)

```

```

lemma Suc-mult-less-cancel1:  $\text{Suc } k * m < \text{Suc } k * n \longleftrightarrow m < n$ 
  by (subst mult-less-cancel1) simp

```

```

lemma Suc-mult-le-cancel1:  $\text{Suc } k * m \leq \text{Suc } k * n \longleftrightarrow m \leq n$ 
  by (subst mult-le-cancel1) simp

```

```

lemma le-square:  $m \leq m * m$ 
  for  $m :: \text{nat}$ 
  by (cases m) (auto intro: le-add1)

```

```

lemma le-cube:  $m \leq m * (m * m)$ 
  for  $m :: \text{nat}$ 
  by (cases m) (auto intro: le-add1)

```

Lemma for *gcd*

```

lemma mult-eq-self-implies-10:
  fixes  $m\ n :: \text{nat}$ 
  assumes  $m = m * n$  shows  $n = 1 \vee m = 0$ 
proof (rule disjCI)
  assume  $m \neq 0$ 
  show  $n = 1$ 
  proof (cases n 1::nat rule: linorder-cases)
    case greater
    show ?thesis
    using assms mult-less-mono2 [OF greater, of m] <m ≠ 0> by auto
  qed (use assms <m ≠ 0> in auto)
qed

```

```

lemma mono-times-nat:
  fixes  $n :: \text{nat}$ 
  assumes  $n > 0$ 
  shows mono (times n)
proof

```



```

fix  $m\ q :: \text{nat}$ 
assume  $m \leq q$ 
with assms show  $n * m \leq n * q$  by simp
qed

```

The lattice order on *nat*.

```

instantiation nat :: distrib-lattice
begin

```

```

definition (inf ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ ) = min

```

```

definition (sup ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ ) = max

```

```

instance

```

```

  by intro-classes
  (auto simp add: inf-nat-def sup-nat-def max-def not-le min-def
   intro: order-less-imp-le antisym elim!: order-trans order-less-trans)

```

```

end

```

17.5 Natural operation of natural numbers on functions

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

```

consts compow ::  $\text{nat} \Rightarrow 'a \Rightarrow 'a$ 

```

```

abbreviation compower ::  $'a \Rightarrow \text{nat} \Rightarrow 'a$  (infixr  $\overset{\sim}{\sim}$  80)
  where  $f \overset{\sim}{\sim} n \equiv \text{compow } n\ f$ 

```

```

notation (latex output)
  compower ((-) [1000] 1000)

```

$f \overset{\sim}{\sim} n = f \circ \dots \circ f$, the n -fold composition of f

```

overloading

```

```

  funpow  $\equiv \text{compow} :: \text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$ 

```

```

begin

```

```

primrec funpow ::  $\text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$ 

```

```

  where

```

```

    funpow 0  $f = \text{id}$ 

```

```

  | funpow (Suc  $n$ )  $f = f \circ \text{funpow } n\ f$ 

```

```

end

```

```

lemma funpow-0 [simp]:  $(f \overset{\sim}{\sim} 0)\ x = x$ 
  by simp

```

```

lemma funpow-Suc-right:  $f \overset{\sim}{\sim} \text{Suc } n = f \overset{\sim}{\sim} n \circ f$ 

```

```

proof (induct n)
  case 0
  then show ?case by simp
next
  fix n
  assume  $f \sim Suc\ n = f \sim n \circ f$ 
  then show  $f \sim Suc\ (Suc\ n) = f \sim Suc\ n \circ f$ 
    by (simp add: o-assoc)
qed

lemmas funpow-simps-right = funpow.simps(1) funpow-Suc-right

For code generation.

context
begin

qualified definition funpow :: nat  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  'a
  where funpow-code-def [code-abbrev]: funpow = compow

lemma [code]:
  funpow (Suc n) f = f  $\circ$  funpow n f
  funpow 0 f = id
  by (simp-all add: funpow-code-def)

end

lemma funpow-add:  $f \sim (m + n) = f \sim m \circ f \sim n$ 
  by (induct m) simp-all

lemma funpow-mult:  $(f \sim m) \sim n = f \sim (m * n)$ 
  for f :: 'a  $\Rightarrow$  'a
  by (induct n) (simp-all add: funpow-add)

lemma funpow-swap1:  $f ((f \sim n) x) = (f \sim n) (f x)$ 
proof -
  have  $f ((f \sim n) x) = (f \sim (n + 1)) x$  by simp
  also have  $\dots = (f \sim n \circ f \sim 1) x$  by (simp only: funpow-add)
  also have  $\dots = (f \sim n) (f x)$  by simp
  finally show ?thesis .
qed

lemma comp-funpow:  $comp\ f \sim n = comp\ (f \sim n)$ 
  for f :: 'a  $\Rightarrow$  'a
  by (induct n) simp-all

lemma Suc-funpow[simp]:  $Suc \sim n = ((+) n)$ 
  by (induct n) simp-all

lemma id-funpow[simp]:  $id \sim n = id$ 

```

by (induct n) simp-all

lemma funpow-mono: $\text{mono } f \implies A \leq B \implies (f \text{ } \sim n) A \leq (f \text{ } \sim n) B$
for $f :: 'a \Rightarrow ('a::\text{order})$
by (induct n arbitrary: A B)
 (auto simp del: funpow.simps(2) simp add: funpow-Suc-right mono-def)

lemma funpow-mono2:
assumes mono f
and $i \leq j$
and $x \leq y$
and $x \leq f x$
shows $(f \text{ } \sim i) x \leq (f \text{ } \sim j) y$
using assms(2,3)
proof (induct j arbitrary: y)
case 0
then show ?case **by** simp
next
case (Suc j)
show ?case
proof(cases i = Suc j)
case True
with assms(1) Suc **show** ?thesis
by (simp del: funpow.simps add: funpow-simps-right monoD funpow-mono)
next
case False
with assms(1,4) Suc **show** ?thesis
by (simp del: funpow.simps add: funpow-simps-right le-eq-less-or-eq less-Suc-eq-le)
 (simp add: Suc.hyps monoD order-subst1)
qed
qed

lemma inj-fn[simp]:
fixes $f :: 'a \Rightarrow 'a$
assumes inj f
shows inj $(f \text{ } \sim n)$
proof (induction n)
case Suc **thus** ?case **using** inj-compose[OF assms Suc.IH] **by** (simp del: comp-apply)
qed simp

lemma surj-fn[simp]:
fixes $f :: 'a \Rightarrow 'a$
assumes surj f
shows surj $(f \text{ } \sim n)$
proof (induction n)
case Suc **thus** ?case **by** (simp add: comp-surj[OF Suc.IH assms] del: comp-apply)
qed simp

lemma bij-fn[simp]:

```

fixes f::'a  $\Rightarrow$  'a
assumes bij f
shows bij (f~n)
by (rule bijI[OF inj-fn[OF bij-is-inj[OF assms]] surj-fn[OF bij-is-surj[OF assms]]])

```

```

lemma bij-betw-funpow:
  assumes bij-betw f S S shows bij-betw (f~n) S S
proof (induct n)
  case 0 then show ?case by (auto simp: id-def[symmetric])
next
  case (Suc n)
  then show ?case unfolding funpow.simps using assms by (rule bij-betw-trans)
qed

```

17.6 Kleene iteration

```

lemma Kleene-iter-lfp:
  fixes f :: 'a::order-bot  $\Rightarrow$  'a
  assumes mono f
  and f p  $\leq$  p
  shows (f~k) bot  $\leq$  p
proof (induct k)
  case 0
  show ?case by simp
next
  case Suc
  show ?case
  using monoD[OF assms(1) Suc] assms(2) by simp
qed

```

```

lemma lfp-Kleene-iter:
  assumes mono f
  and (f~Suc k) bot = (f~k) bot
  shows lfp f = (f~k) bot
proof (rule antisym)
  show lfp f  $\leq$  (f~k) bot
  proof (rule lfp-lowerbound)
    show f ((f~k) bot)  $\leq$  (f~k) bot
    using assms(2) by simp
  qed
  show (f~k) bot  $\leq$  lfp f
  using Kleene-iter-lfp[OF assms(1)] lfp-unfold[OF assms(1)] by simp
qed

```

```

lemma mono-pow: mono f  $\implies$  mono (f~n)
  for f :: 'a  $\Rightarrow$  'a::complete-lattice
  by (induct n) (auto simp: mono-def)

```

```

lemma lfp-funpow:

```

```

assumes  $f$ : mono  $f$ 
shows  $\text{lfp } (f \text{ ^^ } \text{Suc } n) = \text{lfp } f$ 
proof (rule antisym)
  show  $\text{lfp } f \leq \text{lfp } (f \text{ ^^ } \text{Suc } n)$ 
  proof (rule lfp-lowerbound)
    have  $f (\text{lfp } (f \text{ ^^ } \text{Suc } n)) = \text{lfp } (\lambda x. f ((f \text{ ^^ } n) x))$ 
    unfolding funpow-Suc-right by (simp add: lfp-rolling f mono-pow comp-def)
    then show  $f (\text{lfp } (f \text{ ^^ } \text{Suc } n)) \leq \text{lfp } (f \text{ ^^ } \text{Suc } n)$ 
    by (simp add: comp-def)
  qed
  have  $(f \text{ ^^ } n) (\text{lfp } f) = \text{lfp } f$  for  $n$ 
  by (induct n) (auto intro: f lfp-fixpoint)
  then show  $\text{lfp } (f \text{ ^^ } \text{Suc } n) \leq \text{lfp } f$ 
  by (intro lfp-lowerbound) (simp del: funpow.simps)
qed

```

```

lemma gfp-funpow:
  assumes  $f$ : mono  $f$ 
  shows  $\text{gfp } (f \text{ ^^ } \text{Suc } n) = \text{gfp } f$ 
proof (rule antisym)
  show  $\text{gfp } f \geq \text{gfp } (f \text{ ^^ } \text{Suc } n)$ 
  proof (rule gfp-upperbound)
    have  $f (\text{gfp } (f \text{ ^^ } \text{Suc } n)) = \text{gfp } (\lambda x. f ((f \text{ ^^ } n) x))$ 
    unfolding funpow-Suc-right by (simp add: gfp-rolling f mono-pow comp-def)
    then show  $f (\text{gfp } (f \text{ ^^ } \text{Suc } n)) \geq \text{gfp } (f \text{ ^^ } \text{Suc } n)$ 
    by (simp add: comp-def)
  qed
  have  $(f \text{ ^^ } n) (\text{gfp } f) = \text{gfp } f$  for  $n$ 
  by (induct n) (auto intro: f gfp-fixpoint)
  then show  $\text{gfp } (f \text{ ^^ } \text{Suc } n) \geq \text{gfp } f$ 
  by (intro gfp-upperbound) (simp del: funpow.simps)
qed

```

```

lemma Kleene-iter-gfp:
  fixes  $f :: 'a::\text{order-top} \Rightarrow 'a$ 
  assumes mono  $f$ 
  and  $p \leq f p$ 
  shows  $p \leq (f \text{ ^^ } k) \text{ top}$ 
proof (induct k)
  case 0
  show ?case by simp
next
  case Suc
  show ?case
  using monoD[OF assms(1) Suc] assms(2) by simp
qed

```

```

lemma gfp-Kleene-iter:
  assumes mono  $f$ 

```

```

    and  $(f \sim \text{Suc } k) \text{ top} = (f \sim k) \text{ top}$ 
  shows  $\text{gfp } f = (f \sim k) \text{ top}$ 
    (is  $?lhs = ?rhs$ )
proof (rule antisym)
  have  $?rhs \leq f ?rhs$ 
    using  $\text{assms}(2)$  by simp
  then show  $?rhs \leq ?lhs$ 
    by (rule  $\text{gfp-upperbound}$ )
  show  $?lhs \leq ?rhs$ 
    using  $\text{Kleene-iter-gfp}[OF \text{assms}(1)] \text{gfp-unfold}[OF \text{assms}(1)]$  by simp
qed

```

17.7 Embedding of the naturals into any *semiring-1*: *of-nat*

context *semiring-1*

begin

definition *of-nat* :: $\text{nat} \Rightarrow 'a$
 where $\text{of-nat } n = (\text{plus } 1 \sim n) 0$

lemma *of-nat-simps* [*simp*]:
 shows *of-nat-0*: $\text{of-nat } 0 = 0$
 and *of-nat-Suc*: $\text{of-nat } (\text{Suc } m) = 1 + \text{of-nat } m$
 by (*simp-all add: of-nat-def*)

lemma *of-nat-1* [*simp*]: $\text{of-nat } 1 = 1$
 by (*simp add: of-nat-def*)

lemma *of-nat-add* [*simp*]: $\text{of-nat } (m + n) = \text{of-nat } m + \text{of-nat } n$
 by (*induct m*) (*simp-all add: ac-simps*)

lemma *of-nat-mult* [*simp*]: $\text{of-nat } (m * n) = \text{of-nat } m * \text{of-nat } n$
 by (*induct m*) (*simp-all add: ac-simps distrib-right*)

lemma *mult-of-nat-commute*: $\text{of-nat } x * y = y * \text{of-nat } x$
 by (*induct x*) (*simp-all add: algebra-simps*)

primrec *of-nat-aux* :: $('a \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a$
 where
 $\text{of-nat-aux } \text{inc } 0 \ i = i$
 | $\text{of-nat-aux } \text{inc } (\text{Suc } n) \ i = \text{of-nat-aux } \text{inc } n \ (\text{inc } i)$ — tail recursive

lemma *of-nat-code*: $\text{of-nat } n = \text{of-nat-aux } (\lambda i. i + 1) \ n \ 0$

proof (*induct n*)

 case 0

 then show ?case by simp

next

 case (*Suc n*)

 have $\bigwedge i. \text{of-nat-aux } (\lambda i. i + 1) \ n \ (i + 1) = \text{of-nat-aux } (\lambda i. i + 1) \ n \ i + 1$

```

    by (induct n) simp-all
    from this [of 0] have of-nat-aux ( $\lambda i. i + 1$ ) n 1 = of-nat-aux ( $\lambda i. i + 1$ ) n 0
+ 1
    by simp
    with Suc show ?case
    by (simp add: add.commute)
qed

```

```

lemma of-nat-of-bool [simp]:
  of-nat (of-bool P) = of-bool P
  by auto

```

end

```

declare of-nat-code [code]

```

```

context semiring-1-cancel
begin

```

```

lemma of-nat-diff:
   $\langle \text{of-nat } (m - n) = \text{of-nat } m - \text{of-nat } n \rangle$  if  $\langle n \leq m \rangle$ 
proof -
  from that obtain q where  $\langle m = n + q \rangle$ 
  by (blast dest: le-Suc-ex)
  then show ?thesis
  by simp
qed

```

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

```

class semiring-char-0 = semiring-1 +
  assumes inj-of-nat: inj of-nat
begin

```

```

lemma of-nat-eq-iff [simp]: of-nat m = of-nat n  $\longleftrightarrow$  m = n
  by (auto intro: inj-of-nat injD)

```

Special cases where either operand is zero

```

lemma of-nat-0-eq-iff [simp]: 0 = of-nat n  $\longleftrightarrow$  0 = n
  by (fact of-nat-eq-iff [of 0 n, unfolded of-nat-0])

```

```

lemma of-nat-eq-0-iff [simp]: of-nat m = 0  $\longleftrightarrow$  m = 0
  by (fact of-nat-eq-iff [of m 0, unfolded of-nat-0])

```

```

lemma of-nat-1-eq-iff [simp]: 1 = of-nat n  $\longleftrightarrow$  n=1
  using of-nat-eq-iff by fastforce

```

```

lemma of-nat-eq-1-iff [simp]: of-nat n = 1  $\longleftrightarrow$  n=1
  using of-nat-eq-iff by fastforce

lemma of-nat-neq-0 [simp]: of-nat (Suc n)  $\neq$  0
  unfolding of-nat-eq-0-iff by simp

lemma of-nat-0-neq [simp]: 0  $\neq$  of-nat (Suc n)
  unfolding of-nat-0-eq-iff by simp

end

class ring-char-0 = ring-1 + semiring-char-0

context linordered-nonzero-semiring
begin

lemma of-nat-0-le-iff [simp]: 0  $\leq$  of-nat n
  by (induct n) simp-all

lemma of-nat-less-0-iff [simp]:  $\neg$  of-nat m < 0
  by (simp add: not-less)

lemma of-nat-mono[simp]: i  $\leq$  j  $\implies$  of-nat i  $\leq$  of-nat j
  by (auto simp: le-iff-add intro!: add-increasing2)

lemma of-nat-less-iff [simp]: of-nat m < of-nat n  $\longleftrightarrow$  m < n
proof(induct m n rule: diff-induct)
  case (1 m) then show ?case
    by auto
next
  case (2 n) then show ?case
    by (simp add: add-pos-nonneg)
next
  case (3 m n)
  then show ?case
    by (auto simp: add-commute [of 1] add-mono1 not-less add-right-mono leD)
qed

lemma of-nat-le-iff [simp]: of-nat m  $\leq$  of-nat n  $\longleftrightarrow$  m  $\leq$  n
  by (simp add: not-less [symmetric] linorder-not-less [symmetric])

lemma less-imp-of-nat-less: m < n  $\implies$  of-nat m < of-nat n
  by simp

lemma of-nat-less-imp-less: of-nat m < of-nat n  $\implies$  m < n
  by simp

Every linordered-nonzero-semiring has characteristic zero.
subclass semiring-char-0

```


by *standard* (*auto intro!*: *injI simp add: order.eq-iff*)

Special cases where either operand is zero

lemma *of-nat-le-0-iff* [*simp*]: $of\text{-}nat\ m \leq 0 \longleftrightarrow m = 0$
 by (*rule of-nat-le-iff* [*of - 0, simplified*])

lemma *of-nat-0-less-iff* [*simp*]: $0 < of\text{-}nat\ n \longleftrightarrow 0 < n$
 by (*rule of-nat-less-iff* [*of 0, simplified*])

end

context *linordered-nonzero-semiring*
begin

lemma *of-nat-max*: $of\text{-}nat\ (max\ x\ y) = max\ (of\text{-}nat\ x)\ (of\text{-}nat\ y)$
 by (*auto simp: max-def ord-class.max-def*)

lemma *of-nat-min*: $of\text{-}nat\ (min\ x\ y) = min\ (of\text{-}nat\ x)\ (of\text{-}nat\ y)$
 by (*auto simp: min-def ord-class.min-def*)

end

context *linordered-semidom*
begin

subclass *linordered-nonzero-semiring ..*

subclass *semiring-char-0 ..*

end

context *linordered-idom*
begin

lemma *abs-of-nat* [*simp*]:
 $|of\text{-}nat\ n| = of\text{-}nat\ n$
 by (*simp add: abs-if*)

lemma *sgn-of-nat* [*simp*]:
 $sgn\ (of\text{-}nat\ n) = of\text{-}bool\ (n > 0)$
 by *simp*

end

lemma *of-nat-id* [*simp*]: $of\text{-}nat\ n = n$
 by (*induct n*) *simp-all*

lemma *of-nat-eq-id* [*simp*]: $of\text{-}nat = id$
 by (*auto simp add: fun-eq-iff*)

17.8 The set of natural numbers

context *semiring-1*

begin

definition *Nats* :: 'a set (ℕ)

where ℕ = *range of-nat*

lemma *of-nat-in-Nats* [*simp*]: *of-nat* $n \in \mathbb{N}$

by (*simp add: Nats-def*)

lemma *Nats-0* [*simp*]: $0 \in \mathbb{N}$

using *of-nat-0* [*symmetric*] **unfolding** *Nats-def*

by (*rule range-eqI*)

lemma *Nats-1* [*simp*]: $1 \in \mathbb{N}$

using *of-nat-1* [*symmetric*] **unfolding** *Nats-def*

by (*rule range-eqI*)

lemma *Nats-add* [*simp*]: $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a + b \in \mathbb{N}$

unfolding *Nats-def* **using** *of-nat-add* [*symmetric*]

by (*blast intro: range-eqI*)

lemma *Nats-mult* [*simp*]: $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a * b \in \mathbb{N}$

unfolding *Nats-def* **using** *of-nat-mult* [*symmetric*]

by (*blast intro: range-eqI*)

lemma *Nats-cases* [*cases set: Nats*]:

assumes $x \in \mathbb{N}$

obtains (*of-nat*) n **where** $x = \text{of-nat } n$

unfolding *Nats-def*

proof –

from $\langle x \in \mathbb{N} \rangle$ **have** $x \in \text{range of-nat}$ **unfolding** *Nats-def* .

then obtain n **where** $x = \text{of-nat } n$..

then show *thesis* ..

qed

lemma *Nats-induct* [*case-names of-nat, induct set: Nats*]: $x \in \mathbb{N} \implies (\bigwedge n. P$

(*of-nat* n)) $\implies P x$

by (*rule Nats-cases*) *auto*

end

lemma *Nats-diff* [*simp*]:

fixes a :: 'a::*linordered-idom*

assumes $a \in \mathbb{N}$ $b \in \mathbb{N}$ $b \leq a$ **shows** $a - b \in \mathbb{N}$

proof –

obtain i **where** $i = \text{of-nat } i$

using *Nats-cases* *assms* **by** *blast*

obtain j **where** $j = \text{of-nat } j$

```

    using Nats-cases assms by blast
  have  $j \leq i$ 
    using  $\langle b \leq a \rangle$   $i j$  of-nat-le-iff by blast
  then have  $*$ : of-nat  $i - \text{of-nat } j = (\text{of-nat } (i-j)) :: 'a$ 
    by (simp add: of-nat-diff)
  then show ?thesis
    by (simp add:  $*$   $i j$ )
qed

```

17.9 Further arithmetic facts concerning the natural numbers

```

lemma subst-equals:
  assumes  $t = s$  and  $u = t$ 
  shows  $u = s$ 
  using assms(2,1) by (rule trans)

```

```

locale nat-arith
begin

```

```

lemma add1:  $(A::'a::\text{comm-monoid-add}) \equiv k + a \implies A + b \equiv k + (a + b)$ 
  by (simp only: ac-simps)

```

```

lemma add2:  $(B::'a::\text{comm-monoid-add}) \equiv k + b \implies a + B \equiv k + (a + b)$ 
  by (simp only: ac-simps)

```

```

lemma suc1:  $A == k + a \implies \text{Suc } A \equiv k + \text{Suc } a$ 
  by (simp only: add-Suc-right)

```

```

lemma rule0:  $(a::'a::\text{comm-monoid-add}) \equiv a + 0$ 
  by (simp only: add-0-right)

```

```

end

```

```

ML-file  $\langle \text{Tools/nat-arith.ML} \rangle$ 

```

```

simproc-setup nateq-cancel-sums
   $((l::\text{nat}) + m = n \mid (l::\text{nat}) = m + n \mid \text{Suc } m = n \mid m = \text{Suc } n) =$ 
   $\langle K (\text{try } o \text{ Nat-Arith.cancel-eq-conv}) \rangle$ 

```

```

simproc-setup natless-cancel-sums
   $((l::\text{nat}) + m < n \mid (l::\text{nat}) < m + n \mid \text{Suc } m < n \mid m < \text{Suc } n) =$ 
   $\langle K (\text{try } o \text{ Nat-Arith.cancel-less-conv}) \rangle$ 

```

```

simproc-setup natle-cancel-sums
   $((l::\text{nat}) + m \leq n \mid (l::\text{nat}) \leq m + n \mid \text{Suc } m \leq n \mid m \leq \text{Suc } n) =$ 
   $\langle K (\text{try } o \text{ Nat-Arith.cancel-le-conv}) \rangle$ 

```

```

simproc-setup natdiff-cancel-sums

```

$$\langle (l::nat) + m - n \mid (l::nat) - (m + n) \mid Suc\ m - n \mid m - Suc\ n \rangle =$$

$$\langle K\ (try\ o\ Nat\text{-}Arith.\cancel\text{-}diff\text{-}conv) \rangle$$

context *order*
begin

lemma *lift-Suc-mono-le*:
assumes *mono*: $\bigwedge n. f\ n \leq f\ (Suc\ n)$
and $n \leq n'$
shows $f\ n \leq f\ n'$
proof (*cases* $n < n'$)
case *True*
then show *?thesis*
by (*induct* $n\ n'$ *rule*: *less-Suc-induct*) (*auto intro*: *mono*)
next
case *False*
with $\langle n \leq n' \rangle$ **show** *?thesis* **by** *auto*
qed

lemma *lift-Suc-antimono-le*:
assumes *mono*: $\bigwedge n. f\ n \geq f\ (Suc\ n)$
and $n \leq n'$
shows $f\ n \geq f\ n'$
proof (*cases* $n < n'$)
case *True*
then show *?thesis*
by (*induct* $n\ n'$ *rule*: *less-Suc-induct*) (*auto intro*: *mono*)
next
case *False*
with $\langle n \leq n' \rangle$ **show** *?thesis* **by** *auto*
qed

lemma *lift-Suc-mono-less*:
assumes *mono*: $\bigwedge n. f\ n < f\ (Suc\ n)$
and $n < n'$
shows $f\ n < f\ n'$
using $\langle n < n' \rangle$ **by** (*induct* $n\ n'$ *rule*: *less-Suc-induct*) (*auto intro*: *mono*)

lemma *lift-Suc-mono-less-iff*: $(\bigwedge n. f\ n < f\ (Suc\ n)) \implies f\ n < f\ m \iff n < m$
by (*blast intro*: *less-asym'* *lift-Suc-mono-less* [*of f*])
dest: *linorder-not-less*[*THEN iffD1*] *le-eq-less-or-eq* [*THEN iffD1*])

end

lemma *mono-iff-le-Suc*: $mono\ f \iff (\forall n. f\ n \leq f\ (Suc\ n))$
unfolding *mono-def* **by** (*auto intro*: *lift-Suc-mono-le* [*of f*])

lemma *antimono-iff-le-Suc*: $antimono\ f \iff (\forall n. f\ (Suc\ n) \leq f\ n)$
unfolding *antimono-def* **by** (*auto intro*: *lift-Suc-antimono-le* [*of f*])

lemma *strict-mono-Suc-iff*: $strict\ mono\ f \longleftrightarrow (\forall n. f\ n < f\ (Suc\ n))$

proof (*intro iffI strict-monoI*)

assume *: $\forall n. f\ n < f\ (Suc\ n)$

fix $m\ n :: nat$ **assume** $m < n$

thus $f\ m < f\ n$

by (*induction rule: less-Suc-induct*) (*use * in auto*)

qed (*auto simp: strict-mono-def*)

lemma *strict-mono-add*: $strict\ mono\ (\lambda n. 'a::linordered-semidom. n + k)$

by (*auto simp: strict-mono-def*)

lemma *mono-nat-linear-lb*:

fixes $f :: nat \Rightarrow nat$

assumes $\bigwedge m\ n. m < n \implies f\ m < f\ n$

shows $f\ m + k \leq f\ (m + k)$

proof (*induct k*)

case 0

then show *?case* **by** *simp*

next

case (*Suc k*)

then have $Suc\ (f\ m + k) \leq Suc\ (f\ (m + k))$ **by** *simp*

also from *assms* [*of m + k Suc (m + k)*] **have** $Suc\ (f\ (m + k)) \leq f\ (Suc\ (m + k))$

by (*simp add: Suc-le-eq*)

finally show *?case* **by** *simp*

qed

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*:

fixes $a\ b\ c :: nat$

assumes $a < b$ **and** $c \leq a$

shows $a - c < b - c$

proof –

from *assms* **obtain** $d\ e$ **where** $b = c + (d + e)$ **and** $a = c + e$ **and** $d > 0$

by (*auto dest!: le-Suc-ex less-imp-Suc-add simp add: ac-simps*)

then show *?thesis* **by** *simp*

qed

lemma *less-diff-conv*: $i < j - k \longleftrightarrow i + k < j$

for $i\ j\ k :: nat$

by (*cases k ≤ j*) (*auto simp add: not-le dest: less-imp-Suc-add le-Suc-ex*)

lemma *less-diff-conv2*: $k \leq j \implies j - k < i \longleftrightarrow j < i + k$

for $j\ k\ i :: nat$

by (*auto dest: le-Suc-ex*)

lemma *le-diff-conv*: $j - k \leq i \longleftrightarrow j \leq i + k$

for $j\ k\ i :: nat$

by (cases $k \leq j$) (auto simp add: not-le dest!: less-imp-Suc-add le-Suc-ex)

lemma *diff-diff-cancel* [simp]: $i \leq n \implies n - (n - i) = i$
 for $i n :: \text{nat}$
 by (auto dest: le-Suc-ex)

lemma *diff-less* [simp]: $0 < n \implies 0 < m \implies m - n < m$
 for $i n :: \text{nat}$
 by (auto dest: less-imp-Suc-add)

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $k \leq m \implies k \leq n \implies m - k - (n - k) = m - n$
 for $m n k :: \text{nat}$
 by (auto dest!: le-Suc-ex)

hide-fact (open) *diff-diff-eq*

lemma *eq-diff-iff*: $k \leq m \implies k \leq n \implies m - k = n - k \longleftrightarrow m = n$
 for $m n k :: \text{nat}$
 by (auto dest: le-Suc-ex)

lemma *less-diff-iff*: $k \leq m \implies k \leq n \implies m - k < n - k \longleftrightarrow m < n$
 for $m n k :: \text{nat}$
 by (auto dest!: le-Suc-ex)

lemma *le-diff-iff*: $k \leq m \implies k \leq n \implies m - k \leq n - k \longleftrightarrow m \leq n$
 for $m n k :: \text{nat}$
 by (auto dest!: le-Suc-ex)

lemma *le-diff-iff'*: $a \leq c \implies b \leq c \implies c - a \leq c - b \longleftrightarrow b \leq a$
 for $a b c :: \text{nat}$
 by (force dest: le-Suc-ex)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq n \implies m - l \leq n - l$
 for $m n l :: \text{nat}$
 by (auto dest: less-imp-le less-imp-Suc-add split: nat-diff-split)

lemma *diff-le-mono2*: $m \leq n \implies l - n \leq l - m$
 for $m n l :: \text{nat}$
 by (auto dest: less-imp-le le-Suc-ex less-imp-Suc-add less-le-trans split: nat-diff-split)

lemma *diff-less-mono2*: $m < n \implies m < l \implies l - n < l - m$
 for $m n l :: \text{nat}$
 by (auto dest: less-imp-Suc-add split: nat-diff-split)

lemma *diffs0-imp-equal*: $m - n = 0 \implies n - m = 0 \implies m = n$
 for $m n :: \text{nat}$
 by (simp split: nat-diff-split)

lemma *min-diff*: $\min (m - i) (n - i) = \min m n - i$
for $m n i :: \text{nat}$
by (*cases m n rule: le-cases*)
(auto simp add: not-le min.absorb1 min.absorb2 min.absorb-iff1 [symmetric]
diff-le-mono)

lemma *inj-on-diff-nat*:
fixes $k :: \text{nat}$
assumes $\bigwedge n. n \in N \implies k \leq n$
shows *inj-on* $(\lambda n. n - k) N$
proof (*rule inj-onI*)
fix $x y$
assume $a: x \in N y \in N x - k = y - k$
with *assms* **have** $x - k + k = y - k + k$ **by** *auto*
with *a assms* **show** $x = y$ **by** (*auto simp add: eq-diff-iff*)
qed

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j \implies i - (j - k) = i + k - j$
for $i j k :: \text{nat}$
by (*fact diff-diff-right*)

lemma *diff-Suc-diff-eq1* [*simp*]:
assumes $k \leq j$
shows $i - \text{Suc } (j - k) = i + k - \text{Suc } j$
proof –
from *assms* **have** $*$: $\text{Suc } (j - k) = \text{Suc } j - k$
by (*simp add: Suc-diff-le*)
from *assms* **have** $k \leq \text{Suc } j$
by (*rule order-trans*) *simp*
with *diff-diff-right* [*of k Suc j i*] **show** *?thesis*
by *simp*
qed

lemma *diff-Suc-diff-eq2* [*simp*]:
assumes $k \leq j$
shows $\text{Suc } (j - k) - i = \text{Suc } j - (k + i)$
proof –
from *assms* **obtain** n **where** $j = k + n$
by (*auto dest: le-Suc-ex*)
moreover **have** $\text{Suc } n - i = (k + \text{Suc } n) - (k + i)$
using *add-diff-cancel-left* [*of k Suc n i*] **by** *simp*
ultimately **show** *?thesis* **by** *simp*
qed

lemma *Suc-diff-Suc*:
assumes $n < m$
shows $\text{Suc } (m - \text{Suc } n) = m - n$

proof –

from *assms* **obtain** q **where** $m = n + \text{Suc } q$
by (*auto dest: less-imp-Suc-add*)
moreover define r **where** $r = \text{Suc } q$
ultimately have $\text{Suc } (m - \text{Suc } n) = r$ **and** $m = n + r$
by *simp-all*
then show *?thesis* **by** *simp*
qed

lemma *one-less-mult*: $\text{Suc } 0 < n \implies \text{Suc } 0 < m \implies \text{Suc } 0 < m * n$
using *less-1-mult [of n m]* **by** (*simp add: ac-simps*)

lemma *n-less-m-mult-n*: $0 < n \implies \text{Suc } 0 < m \implies n < m * n$
using *mult-strict-right-mono [of 1 m n]* **by** *simp*

lemma *n-less-n-mult-m*: $0 < n \implies \text{Suc } 0 < m \implies n < n * m$
using *mult-strict-left-mono [of 1 m n]* **by** *simp*

Induction starting beyond zero

lemma *nat-induct-at-least* [*consumes 1, case-names base Suc*]:

$P n$ **if** $n \geq m$ $P m \wedge n. n \geq m \implies P n \implies P (\text{Suc } n)$

proof –

define q **where** $q = n - m$
with $\langle n \geq m \rangle$ **have** $n = m + q$
by *simp*
moreover have $P (m + q)$
by (*induction q*) (*use that in simp-all*)
ultimately show $P n$
by *simp*
qed

lemma *nat-induct-non-zero* [*consumes 1, case-names 1 Suc*]:

$P n$ **if** $n > 0$ $P 1 \wedge n. n > 0 \implies P n \implies P (\text{Suc } n)$

proof –

from $\langle n > 0 \rangle$ **have** $n \geq 1$
by (*cases n*) *simp-all*
moreover note $\langle P 1 \rangle$
moreover have $\wedge n. n \geq 1 \implies P n \implies P (\text{Suc } n)$
using $\langle \wedge n. n > 0 \implies P n \implies P (\text{Suc } n) \rangle$
by (*simp add: Suc-le-eq*)
ultimately show $P n$
by (*rule nat-induct-at-least*)
qed

Specialized induction principles that work "backwards":

lemma *inc-induct* [*consumes 1, case-names base step*]:

assumes *less*: $i \leq j$

and *base*: $P j$

and *step*: $\wedge n. i \leq n \implies n < j \implies P (\text{Suc } n) \implies P n$


```

shows  $P\ i$ 
using less step
proof (induct  $j - i$  arbitrary:  $i$ )
  case (0  $i$ )
  then have  $i = j$  by simp
  with base show ?case by simp
next
case (Suc  $d\ n$ )
from Suc.hyps have  $n \neq j$  by auto
with Suc have  $n < j$  by (simp add: less-le)
from  $\langle \text{Suc } d = j - n \rangle$  have  $d + 1 = j - n$  by simp
then have  $d + 1 - 1 = j - n - 1$  by simp
then have  $d = j - n - 1$  by simp
then have  $d = j - (n + 1)$  by (simp add: diff-diff-eq)
then have  $d = j - \text{Suc } n$  by simp
moreover from  $\langle n < j \rangle$  have  $\text{Suc } n \leq j$  by (simp add: Suc-le-eq)
ultimately have  $P (\text{Suc } n)$ 
proof (rule Suc.hyps)
  fix  $q$ 
  assume  $\text{Suc } n \leq q$ 
  then have  $n \leq q$  by (simp add: Suc-le-eq less-imp-le)
  moreover assume  $q < j$ 
  moreover assume  $P (\text{Suc } q)$ 
  ultimately show  $P\ q$  by (rule Suc.prems)
qed
with order-refl  $\langle n < j \rangle$  show  $P\ n$  by (rule Suc.prems)
qed

```

lemma *strict-inc-induct* [*consumes 1, case-names base step*]:

```

assumes less:  $i < j$ 
  and base:  $\bigwedge i. j = \text{Suc } i \implies P\ i$ 
  and step:  $\bigwedge i. i < j \implies P (\text{Suc } i) \implies P\ i$ 
shows  $P\ i$ 
using less proof (induct  $j - i - 1$  arbitrary:  $i$ )
  case (0  $i$ )
  from  $\langle i < j \rangle$  obtain  $n$  where  $j = i + n$  and  $n > 0$ 
  by (auto dest!: less-imp-Suc-add)
  with 0 have  $j = \text{Suc } i$ 
  by (auto intro: order-antisym simp add: Suc-le-eq)
  with base show ?case by simp
next
case (Suc  $d\ i$ )
from  $\langle \text{Suc } d = j - i - 1 \rangle$  have *:  $\text{Suc } d = j - \text{Suc } i$ 
  by (simp add: diff-diff-add)
then have  $\text{Suc } d - 1 = j - \text{Suc } i - 1$  by simp
then have  $d = j - \text{Suc } i - 1$  by simp
moreover from * have  $j - \text{Suc } i \neq 0$  by auto
then have  $\text{Suc } i < j$  by (simp add: not-le)
ultimately have  $P (\text{Suc } i)$  by (rule Suc.hyps)

```

with $\langle i < j \rangle$ **show** $P\ i$ **by** (rule step)
qed

lemma *zero-induct-lemma*: $P\ k \implies (\bigwedge n. P\ (Suc\ n) \implies P\ n) \implies P\ (k - i)$
using *inc-induct*[of $k - i$ $k\ P$, *simplified*] **by** *blast*

lemma *zero-induct*: $P\ k \implies (\bigwedge n. P\ (Suc\ n) \implies P\ n) \implies P\ 0$
using *inc-induct*[of $0\ k\ P$] **by** *blast*

Further induction rule similar to $\llbracket ?i \leq ?j; ?P\ ?j; \bigwedge n. \llbracket ?i \leq n; n < ?j; ?P\ (Suc\ n) \rrbracket \implies ?P\ n \rrbracket \implies ?P\ ?i$.

lemma *dec-induct* [*consumes 1, case-names base step*]:

$i \leq j \implies P\ i \implies (\bigwedge n. i \leq n \implies n < j \implies P\ n \implies P\ (Suc\ n)) \implies P\ j$

proof (*induct j arbitrary: i*)

case 0

then show $?case$ **by** *simp*

next

case $(Suc\ j)$

from *Suc.prem*s **consider** $i \leq j \mid i = Suc\ j$

by (*auto simp add: le-Suc-eq*)

then show $?case$

proof *cases*

case 1

moreover have $j < Suc\ j$ **by** *simp*

moreover have $P\ j$ **using** $\langle i \leq j \rangle \langle P\ i \rangle$

proof (*rule Suc.hyphs*)

fix q

assume $i \leq q$

moreover assume $q < j$ **then have** $q < Suc\ j$

by (*simp add: less-Suc-eq*)

moreover assume $P\ q$

ultimately show $P\ (Suc\ q)$ **by** (*rule Suc.prem*s)

qed

ultimately show $P\ (Suc\ j)$ **by** (*rule Suc.prem*s)

next

case 2

with $\langle P\ i \rangle$ **show** $P\ (Suc\ j)$ **by** *simp*

qed

qed

lemma *transitive-stepwise-le*:

assumes $m \leq n \bigwedge x. R\ x\ x \bigwedge x\ y\ z. R\ x\ y \implies R\ y\ z \implies R\ x\ z$ **and** $\bigwedge n. R\ n\ (Suc\ n)$

shows $R\ m\ n$

using $\langle m \leq n \rangle$

by (*induction rule: dec-induct*) (*use assms in blast*)+

17.9.1 Greatest operator

lemma *ex-has-greatest-nat*:

$P (k::nat) \implies \forall y. P y \longrightarrow y \leq b \implies \exists x. P x \wedge (\forall y. P y \longrightarrow y \leq x)$

proof (*induction b-k arbitrary: b k rule: less-induct*)

case *less*

show *?case*

proof *cases*

assume $\exists n > k. P n$

then obtain n **where** $n > k$ $P n$ **by** *blast*

have $n \leq b$ **using** $\langle P n \rangle$ *less.premis(2)* **by** *auto*

hence $b - n < b - k$

by(*rule diff-less-mono2[OF <k<n> less-le-trans[OF <k<n>]]*)

from *less.hyps[OF this <P n> less.premis(2)]*

show *?thesis* .

next

assume $\neg (\exists n > k. P n)$

hence $\forall y. P y \longrightarrow y \leq k$ **by** (*auto simp: not-less*)

thus *?thesis* **using** *less.premis(1)* **by** *auto*

qed

qed

lemma

fixes $k::nat$

assumes $P k$ **and** *minor*: $\bigwedge y. P y \implies y \leq b$

shows *GreatestI-nat*: $P (Greatest P)$

and *Greatest-le-nat*: $k \leq Greatest P$

proof –

obtain x **where** $P x \bigwedge y. P y \implies y \leq x$

using *assms ex-has-greatest-nat* **by** *blast*

with $\langle P k \rangle$ **show** $P (Greatest P) k \leq Greatest P$

using *GreatestI2-order* **by** *blast+*

qed

lemma *GreatestI-ex-nat*:

$\llbracket \exists k::nat. P k; \bigwedge y. P y \implies y \leq b \rrbracket \implies P (Greatest P)$

by (*blast intro: GreatestI-nat*)

17.10 Monotonicity of funpow

lemma *funpow-increasing*: $m \leq n \implies mono f \implies (f \rightsquigarrow n) \top \leq (f \rightsquigarrow m) \top$

for $f :: 'a::\{lattice,order-top\} \Rightarrow 'a$

by (*induct rule: inc-induct*)

(*auto simp del: funpow.simps(2) simp add: funpow-Suc-right*

intro: order-trans[OF - funpow-mono])

lemma *funpow-decreasing*: $m \leq n \implies mono f \implies (f \rightsquigarrow m) \perp \leq (f \rightsquigarrow n) \perp$

for $f :: 'a::\{lattice,order-bot\} \Rightarrow 'a$

by (*induct rule: dec-induct*)

(*auto simp del: funpow.simps(2) simp add: funpow-Suc-right*

intro: order-trans[OF - funpow-mono])

lemma *mono-funpow*: $\text{mono } Q \implies \text{mono } (\lambda i. (Q \rightsquigarrow i) \perp)$
for $Q :: 'a::\{\text{lattice,order-bot}\} \Rightarrow 'a$
by (*auto intro!: funpow-decreasing simp: mono-def*)

lemma *antimono-funpow*: $\text{mono } Q \implies \text{antimono } (\lambda i. (Q \rightsquigarrow i) \top)$
for $Q :: 'a::\{\text{lattice,order-top}\} \Rightarrow 'a$
by (*auto intro!: funpow-increasing simp: antimono-def*)

17.11 The divides relation on *nat*

lemma *dvd-1-left [iff]*: $\text{Suc } 0 \text{ dvd } k$
by (*simp add: dvd-def*)

lemma *dvd-1-iff-1 [simp]*: $m \text{ dvd } \text{Suc } 0 \iff m = \text{Suc } 0$
by (*simp add: dvd-def*)

lemma *nat-dvd-1-iff-1 [simp]*: $m \text{ dvd } 1 \iff m = 1$
for $m :: \text{nat}$
by (*simp add: dvd-def*)

lemma *dvd-antisym*: $m \text{ dvd } n \implies n \text{ dvd } m \implies m = n$
for $m n :: \text{nat}$
unfolding *dvd-def* **by** (*force dest: mult-eq-self-implies-10 simp add: mult.assoc*)

lemma *dvd-diff-nat [simp]*: $k \text{ dvd } m \implies k \text{ dvd } n \implies k \text{ dvd } (m - n)$
for $k m n :: \text{nat}$
unfolding *dvd-def* **by** (*blast intro: right-diff-distrib' [symmetric]*)

lemma *dvd-diffD*:
fixes $k m n :: \text{nat}$
assumes $k \text{ dvd } m - n$ $k \text{ dvd } n$ $n \leq m$
shows $k \text{ dvd } m$

proof –
have $k \text{ dvd } n + (m - n)$
using *assms* **by** (*blast intro: dvd-add*)
with *assms* **show** *?thesis*
by *simp*

qed

lemma *dvd-diffD1*: $k \text{ dvd } m - n \implies k \text{ dvd } m \implies n \leq m \implies k \text{ dvd } n$
for $k m n :: \text{nat}$
by (*drule-tac m = m in dvd-diff-nat*) *auto*

lemma *dvd-mult-cancel*:
fixes $m n k :: \text{nat}$
assumes $k * m \text{ dvd } k * n$ **and** $0 < k$
shows $m \text{ dvd } n$

proof –

from *assms*(1) **obtain** *q* **where** $k * n = (k * m) * q$..
then have $k * n = k * (m * q)$ **by** (*simp add: ac-simps*)
with $\langle 0 < k \rangle$ **have** $n = m * q$ **by** (*auto simp add: mult-left-cancel*)
then show ?thesis ..

qed

lemma *dvd-mult-cancel1*:

fixes $m\ n :: \text{nat}$
assumes $0 < m$
shows $m * n \text{ dvd } m \longleftrightarrow n = 1$

proof

assume $m * n \text{ dvd } m$
then have $m * n \text{ dvd } m * 1$
by *simp*
then have $n \text{ dvd } 1$
by (*iprover intro: assms dvd-mult-cancel*)
then show $n = 1$
by *auto*

qed *auto*

lemma *dvd-mult-cancel2*: $0 < m \implies n * m \text{ dvd } m \longleftrightarrow n = 1$

for $m\ n :: \text{nat}$
using *dvd-mult-cancel1* [of $m\ n$] **by** (*simp add: ac-simps*)

lemma *dvd-imp-le*: $k \text{ dvd } n \implies 0 < n \implies k \leq n$

for $k\ n :: \text{nat}$
by (*auto elim!: dvdE*) (*auto simp add: gr0-conv-Suc*)

lemma *nat-dvd-not-less*: $0 < m \implies m < n \implies \neg n \text{ dvd } m$

for $m\ n :: \text{nat}$
by (*auto elim!: dvdE*) (*auto simp add: gr0-conv-Suc*)

lemma *less-eq-dvd-minus*:

fixes $m\ n :: \text{nat}$
assumes $m \leq n$
shows $m \text{ dvd } n \longleftrightarrow m \text{ dvd } n - m$

proof –

from *assms* **have** $n = m + (n - m)$ **by** *simp*
then obtain *q* **where** $n = m + q$..
then show ?thesis **by** (*simp add: add.commute* [of m])

qed

lemma *dvd-minus-self*: $m \text{ dvd } n - m \longleftrightarrow n < m \vee m \text{ dvd } n$

for $m\ n :: \text{nat}$
by (*cases* $n < m$) (*auto elim!: dvdE simp add: not-less le-imp-diff-is-add dest: less-imp-le*)

lemma *dvd-minus-add*:

```

fixes  $m\ n\ q\ r :: \text{nat}$ 
assumes  $q \leq n\ q \leq r * m$ 
shows  $m\ \text{dvd}\ n - q \iff m\ \text{dvd}\ n + (r * m - q)$ 
proof -
  have  $m\ \text{dvd}\ n - q \iff m\ \text{dvd}\ r * m + (n - q)$ 
    using dvd-add-times-triv-left-iff [of  $m\ r$ ] by simp
  also from assms have  $\dots \iff m\ \text{dvd}\ r * m + n - q$  by simp
  also from assms have  $\dots \iff m\ \text{dvd}\ (r * m - q) + n$  by simp
  also have  $\dots \iff m\ \text{dvd}\ n + (r * m - q)$  by (simp add: add.commute)
  finally show ?thesis .
qed

```

17.12 Aliasses

lemma *nat-mult-1*: $1 * n = n$

```

for  $n :: \text{nat}$ 
by (fact mult-1-left)

```

lemma *nat-mult-1-right*: $n * 1 = n$

```

for  $n :: \text{nat}$ 
by (fact mult-1-right)

```

lemma *diff-mult-distrib*: $(m - n) * k = (m * k) - (n * k)$

```

for  $k\ m\ n :: \text{nat}$ 
by (fact left-diff-distrib')

```

lemma *diff-mult-distrib2*: $k * (m - n) = (k * m) - (k * n)$

```

for  $k\ m\ n :: \text{nat}$ 
by (fact right-diff-distrib')

```

lemma *le-diff-conv2*: $k \leq j \implies (i \leq j - k) = (i + k \leq j)$

```

for  $i\ j\ k :: \text{nat}$ 
by (fact le-diff-conv2)

```

lemma *diff-self-eq-0* [*simp*]: $m - m = 0$

```

for  $m :: \text{nat}$ 
by (fact diff-cancel)

```

lemma *diff-diff-left* [*simp*]: $i - j - k = i - (j + k)$

```

for  $i\ j\ k :: \text{nat}$ 
by (fact diff-diff-add)

```

lemma *diff-commute*: $i - j - k = i - k - j$

```

for  $i\ j\ k :: \text{nat}$ 
by (fact diff-right-commute)

```

lemma *diff-add-inverse*: $(n + m) - n = m$

```

for  $m\ n :: \text{nat}$ 

```

by (fact add-diff-cancel-left')

lemma *diff-add-inverse2*: $(m + n) - n = m$
 for $m\ n :: \text{nat}$
 by (fact add-diff-cancel-right')

lemma *diff-cancel*: $(k + m) - (k + n) = m - n$
 for $k\ m\ n :: \text{nat}$
 by (fact add-diff-cancel-left)

lemma *diff-cancel2*: $(m + k) - (n + k) = m - n$
 for $k\ m\ n :: \text{nat}$
 by (fact add-diff-cancel-right)

lemma *diff-add-0*: $n - (n + m) = 0$
 for $m\ n :: \text{nat}$
 by (fact diff-add-zero)

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + (k * n)$
 for $k\ m\ n :: \text{nat}$
 by (fact distrib-left)

lemmas *nat-distrib* =
add-mult-distrib distrib-left diff-mult-distrib diff-mult-distrib2

17.13 Size of a datatype value

class *size* =
 fixes *size* :: 'a \Rightarrow nat — see further theory *Wellfounded*

instantiation *nat* :: *size*
begin

definition *size-nat* **where** [*simp*, *code*]: *size* ($n :: \text{nat}$) = n

instance ..

end

lemmas *size-nat* = *size-nat-def*

lemma *size-neq-size-imp-neq*: $\text{size } x \neq \text{size } y \implies x \neq y$
 by (*erule contrapos-nn*) (*rule arg-cong*)

17.14 Code module namespace

code-identifier

code-module *Nat* \rightarrow (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

hide-const (**open**) *of-nat-aux*

end

18 Fields

```
theory Fields
imports Nat
begin
```

18.1 Division rings

A division ring is like a field, but without the commutativity requirement.

```
class inverse = divide +
  fixes inverse :: 'a ⇒ 'a
begin

abbreviation inverse-divide :: 'a ⇒ 'a ⇒ 'a (infixl '/' 70)
where
  inverse-divide ≡ divide

end
```

Setup for linear arithmetic prover

```
ML-file <~~/src/Provers/Arith/fast-lin-arith.ML>
ML-file <Tools/lin-arith.ML>
setup <Lin-Arith.global-setup>
declaration <K (
  Lin-Arith.init-arith-data
  #> Lin-Arith.add-discrete-type type-name <nat>
  #> Lin-Arith.add-lessD @ {thm Suc-leI}
  #> Lin-Arith.add-simps @ {thms simp-thms ring-distrib if-True if-False
    minus-diff-eq
    add-0-left add-0-right order-less-irrefl
    zero-neq-one zero-less-one zero-le-one
    zero-neq-one [THEN not-sym] not-one-le-zero not-one-less-zero
    add-Suc add-Suc-right nat.inject
    Suc-le-mono Suc-less-eq Zero-not-Suc
    Suc-not-Zero le-0-eq One-nat-def}
  #> Lin-Arith.add-simprocs [simproc <group-cancel-add>, simproc <group-cancel-diff>,
    simproc <group-cancel-eq>, simproc <group-cancel-le>,
    simproc <group-cancel-less>,
    simproc <nateq-cancel-sums>, simproc <natless-cancel-sums>,
    simproc <natle-cancel-sums>]]>
```

simproc-setup *fast-arith-nat* (($m::nat < n$ | $m::nat \leq n$ | $m::nat = n$) =
 <K *Lin-Arith.simproc*> — Because of this simproc, the arithmetic solver is really
 only useful to detect inconsistencies among the premises for subgoals which are *not*
 themselves (in)equalities, because the latter activate *fast-nat-arith-simproc* anyway.

However, it seems cheaper to activate the solver all the time rather than add the additional check.

lemmas [*linarith-split*] = *nat-diff-split split-min split-max abs-split*

Lemmas *divide-simps* move division to the outside and eliminates them on (in)equalities.

named-theorems *divide-simps* rewrite rules to eliminate divisions

```
class division-ring = ring-1 + inverse +
  assumes left-inverse [simp]:  $a \neq 0 \implies \text{inverse } a * a = 1$ 
  assumes right-inverse [simp]:  $a \neq 0 \implies a * \text{inverse } a = 1$ 
  assumes divide-inverse:  $a / b = a * \text{inverse } b$ 
  assumes inverse-zero [simp]:  $\text{inverse } 0 = 0$ 
begin
```

```
subclass ring-1-no-zero-divisors
```

```
proof
```

```
  fix a b :: 'a
```

```
  assume a:  $a \neq 0$  and b:  $b \neq 0$ 
```

```
  show  $a * b \neq 0$ 
```

```
  proof
```

```
    assume ab:  $a * b = 0$ 
```

```
    hence  $0 = \text{inverse } a * (a * b) * \text{inverse } b$  by simp
```

```
    also have  $\dots = (\text{inverse } a * a) * (b * \text{inverse } b)$ 
```

```
      by (simp only: mult.assoc)
```

```
    also have  $\dots = 1$  using a b by simp
```

```
    finally show False by simp
```

```
  qed
```

```
qed
```

```
lemma nonzero-imp-inverse-nonzero:
```

```
   $a \neq 0 \implies \text{inverse } a \neq 0$ 
```

```
proof
```

```
  assume ianz:  $\text{inverse } a = 0$ 
```

```
  assume  $a \neq 0$ 
```

```
  hence  $1 = a * \text{inverse } a$  by simp
```

```
  also have  $\dots = 0$  by (simp add: ianz)
```

```
  finally have  $1 = 0$  .
```

```
  thus False by (simp add: eq-commute)
```

```
qed
```

```
lemma inverse-zero-imp-zero:
```

```
  assumes  $\text{inverse } a = 0$  shows  $a = 0$ 
```

```
proof (rule ccontr)
```

```
  assume  $a \neq 0$ 
```

```
  then have  $\text{inverse } a \neq 0$ 
```

```
    by (simp add: nonzero-imp-inverse-nonzero)
```

```
  with assms show False
```

by *auto*
qed

lemma *inverse-unique*:

assumes *ab*: $a * b = 1$

shows $\text{inverse } a = b$

proof –

have $a \neq 0$ **using** *ab* **by** (*cases* $a = 0$) *simp-all*

moreover have $\text{inverse } a * (a * b) = \text{inverse } a$ **by** (*simp add*: *ab*)

ultimately show *?thesis* **by** (*simp add*: *mult.assoc* [*symmetric*])

qed

lemma *nonzero-inverse-minus-eq*:

$a \neq 0 \implies \text{inverse } (- a) = - \text{inverse } a$

by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-inverse-eq*:

$a \neq 0 \implies \text{inverse } (\text{inverse } a) = a$

by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-eq-imp-eq*:

assumes $\text{inverse } a = \text{inverse } b$ **and** $a \neq 0$ **and** $b \neq 0$

shows $a = b$

proof –

from $\langle \text{inverse } a = \text{inverse } b \rangle$

have $\text{inverse } (\text{inverse } a) = \text{inverse } (\text{inverse } b)$ **by** (*rule arg-cong*)

with $\langle a \neq 0 \rangle$ **and** $\langle b \neq 0 \rangle$ **show** $a = b$

by (*simp add*: *nonzero-inverse-inverse-eq*)

qed

lemma *inverse-1* [*simp*]: $\text{inverse } 1 = 1$

by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-mult-distrib*:

assumes $a \neq 0$ **and** $b \neq 0$

shows $\text{inverse } (a * b) = \text{inverse } b * \text{inverse } a$

proof –

have $a * (b * \text{inverse } b) * \text{inverse } a = 1$ **using** *assms* **by** *simp*

hence $a * b * (\text{inverse } b * \text{inverse } a) = 1$ **by** (*simp only*: *mult.assoc*)

thus *?thesis* **by** (*rule inverse-unique*)

qed

lemma *division-ring-inverse-add*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a + \text{inverse } b = \text{inverse } a * (a + b) * \text{inverse } b$

by (*simp add*: *algebra-simps*)

lemma *division-ring-inverse-diff*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a - \text{inverse } b = \text{inverse } a * (b - a) * \text{inverse } b$

by (*simp add*: *algebra-simps*)

lemma *right-inverse-eq*: $b \neq 0 \implies a / b = 1 \iff a = b$

proof

assume *neq*: $b \neq 0$

 {

hence $a = (a / b) * b$ **by** (*simp add: divide-inverse mult.assoc*)

also assume $a / b = 1$

finally show $a = b$ **by** *simp*

next

assume $a = b$

with *neq* **show** $a / b = 1$ **by** (*simp add: divide-inverse*)

 }

qed

lemma *nonzero-inverse-eq-divide*: $a \neq 0 \implies \text{inverse } a = 1 / a$

by (*simp add: divide-inverse*)

lemma *divide-self* [*simp*]: $a \neq 0 \implies a / a = 1$

by (*simp add: divide-inverse*)

lemma *inverse-eq-divide* [*field-simps, field-split-simps, divide-simps*]: $\text{inverse } a = 1 / a$

by (*simp add: divide-inverse*)

lemma *add-divide-distrib*: $(a+b) / c = a/c + b/c$

by (*simp add: divide-inverse algebra-simps*)

lemma *times-divide-eq-right* [*simp*]: $a * (b / c) = (a * b) / c$

by (*simp add: divide-inverse mult.assoc*)

lemma *minus-divide-left*: $-(a / b) = (-a) / b$

by (*simp add: divide-inverse*)

lemma *nonzero-minus-divide-right*: $b \neq 0 \implies -(a / b) = a / (-b)$

by (*simp add: divide-inverse nonzero-inverse-minus-eq*)

lemma *nonzero-minus-divide-divide*: $b \neq 0 \implies (-a) / (-b) = a / b$

by (*simp add: divide-inverse nonzero-inverse-minus-eq*)

lemma *divide-minus-left* [*simp*]: $(-a) / b = -(a / b)$

by (*simp add: divide-inverse*)

lemma *diff-divide-distrib*: $(a - b) / c = a / c - b / c$

using *add-divide-distrib* [*of a - b c*] **by** *simp*

lemma *nonzero-eq-divide-eq* [*field-simps*]: $c \neq 0 \implies a = b / c \iff a * c = b$

proof –

assume [*simp*]: $c \neq 0$

have $a = b / c \iff a * c = (b / c) * c$ **by** *simp*

also have ... $\longleftrightarrow a * c = b$ by (simp add: divide-inverse mult.assoc)
 finally show ?thesis .
 qed

lemma nonzero-divide-eq-eq [field-simps]: $c \neq 0 \implies b / c = a \longleftrightarrow b = a * c$

proof -

assume [simp]: $c \neq 0$

have $b / c = a \longleftrightarrow (b / c) * c = a * c$ by simp

also have ... $\longleftrightarrow b = a * c$ by (simp add: divide-inverse mult.assoc)

finally show ?thesis .

qed

lemma nonzero-neg-divide-eq-eq [field-simps]: $b \neq 0 \implies -(a / b) = c \longleftrightarrow -a = c * b$

using nonzero-divide-eq-eq[of b -a c] by simp

lemma nonzero-neg-divide-eq-eq2 [field-simps]: $b \neq 0 \implies c = -(a / b) \longleftrightarrow c * b = -a$

using nonzero-neg-divide-eq-eq[of b a c] by auto

lemma divide-eq-imp: $c \neq 0 \implies b = a * c \implies b / c = a$

by (simp add: divide-inverse mult.assoc)

lemma eq-divide-imp: $c \neq 0 \implies a * c = b \implies a = b / c$

by (drule sym) (simp add: divide-inverse mult.assoc)

lemma add-divide-eq-iff [field-simps]:

$z \neq 0 \implies x + y / z = (x * z + y) / z$

by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma divide-add-eq-iff [field-simps]:

$z \neq 0 \implies x / z + y = (x + y * z) / z$

by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma diff-divide-eq-iff [field-simps]:

$z \neq 0 \implies x - y / z = (x * z - y) / z$

by (simp add: diff-divide-distrib nonzero-eq-divide-eq eq-diff-eq)

lemma minus-divide-add-eq-iff [field-simps]:

$z \neq 0 \implies -(x / z) + y = (-x + y * z) / z$

by (simp add: add-divide-distrib diff-divide-eq-iff)

lemma divide-diff-eq-iff [field-simps]:

$z \neq 0 \implies x / z - y = (x - y * z) / z$

by (simp add: field-simps)

lemma minus-divide-diff-eq-iff [field-simps]:

$z \neq 0 \implies -(x / z) - y = (-x - y * z) / z$

by (simp add: divide-diff-eq-iff[symmetric])

lemma *division-ring-divide-zero* [simp]:

$$a / 0 = 0$$

by (*simp add: divide-inverse*)

lemma *divide-self-if* [simp]:

$$a / a = (\text{if } a = 0 \text{ then } 0 \text{ else } 1)$$

by *simp*

lemma *inverse-nonzero-iff-nonzero* [simp]:

$$\text{inverse } a = 0 \longleftrightarrow a = 0$$

by (*rule iffI*) (*fact inverse-zero-imp-zero, simp*)

lemma *inverse-minus-eq* [simp]:

$$\text{inverse } (- a) = - \text{inverse } a$$

proof *cases*

assume $a=0$ **thus** *?thesis* **by** *simp*

next

assume $a \neq 0$

thus *?thesis* **by** (*simp add: nonzero-inverse-minus-eq*)

qed

lemma *inverse-inverse-eq* [simp]:

$$\text{inverse } (\text{inverse } a) = a$$

proof *cases*

assume $a=0$ **thus** *?thesis* **by** *simp*

next

assume $a \neq 0$

thus *?thesis* **by** (*simp add: nonzero-inverse-inverse-eq*)

qed

lemma *inverse-eq-imp-eq*:

$$\text{inverse } a = \text{inverse } b \implies a = b$$

by (*drule arg-cong [where f=inverse], simp*)

lemma *inverse-eq-iff-eq* [simp]:

$$\text{inverse } a = \text{inverse } b \longleftrightarrow a = b$$

by (*force dest!: inverse-eq-imp-eq*)

lemma *mult-commute-imp-mult-inverse-commute*:

assumes $y * x = x * y$

shows $\text{inverse } y * x = x * \text{inverse } y$

proof (*cases y=0*)

case *False*

hence $x * \text{inverse } y = \text{inverse } y * y * x * \text{inverse } y$

by *simp*

also have $\dots = \text{inverse } y * (x * y * \text{inverse } y)$

by (*simp add: mult.assoc assms*)

finally show *?thesis* **by** (*simp add: mult.assoc False*)

qed *simp*

lemmas *mult-inverse-of-nat-commute* =
mult-commute-imp-mult-inverse-commute[*OF mult-of-nat-commute*]

lemma *divide-divide-eq-left'*:

$(a / b) / c = a / (c * b)$

by (*cases* $b = 0 \vee c = 0$)

(*auto simp: divide-inverse mult.assoc nonzero-inverse-mult-distrib*)

lemma *add-divide-eq-if-simps* [*field-split-simps, divide-simps*]:

$a + b / z = (\text{if } z = 0 \text{ then } a \text{ else } (a * z + b) / z)$

$a / z + b = (\text{if } z = 0 \text{ then } b \text{ else } (a + b * z) / z)$

$-(a / z) + b = (\text{if } z = 0 \text{ then } b \text{ else } (-a + b * z) / z)$

$a - b / z = (\text{if } z = 0 \text{ then } a \text{ else } (a * z - b) / z)$

$a / z - b = (\text{if } z = 0 \text{ then } -b \text{ else } (a - b * z) / z)$

$-(a / z) - b = (\text{if } z = 0 \text{ then } -b \text{ else } (-a - b * z) / z)$

by (*simp-all add: add-divide-eq-iff divide-add-eq-iff diff-divide-eq-iff divide-diff-eq-iff minus-divide-diff-eq-iff*)

lemma [*field-split-simps, divide-simps*]:

shows *divide-eq-eq*: $b / c = a \longleftrightarrow (\text{if } c \neq 0 \text{ then } b = a * c \text{ else } a = 0)$

and *eq-divide-eq*: $a = b / c \longleftrightarrow (\text{if } c \neq 0 \text{ then } a * c = b \text{ else } a = 0)$

and *minus-divide-eq-eq*: $-(b / c) = a \longleftrightarrow (\text{if } c \neq 0 \text{ then } -b = a * c \text{ else } a = 0)$

and *eq-minus-divide-eq*: $a = -(b / c) \longleftrightarrow (\text{if } c \neq 0 \text{ then } a * c = -b \text{ else } a = 0)$

by (*auto simp add: field-simps*)

end

18.2 Fields

class *field* = *comm-ring-1* + *inverse* +

assumes *field-inverse*: $a \neq 0 \implies \text{inverse } a * a = 1$

assumes *field-divide-inverse*: $a / b = a * \text{inverse } b$

assumes *field-inverse-zero*: $\text{inverse } 0 = 0$

begin

subclass *division-ring*

proof

fix $a :: 'a$

assume $a \neq 0$

thus $\text{inverse } a * a = 1$ **by** (*rule field-inverse*)

thus $a * \text{inverse } a = 1$ **by** (*simp only: mult.commute*)

next

fix $a b :: 'a$

show $a / b = a * \text{inverse } b$ **by** (*rule field-divide-inverse*)

next

```

show  $inverse\ 0 = 0$ 
  by (fact field-inverse-zero)
qed

subclass idom-divide
proof
  fix  $b\ a$ 
  assume  $b \neq 0$ 
  then show  $a * b / b = a$ 
    by (simp add: divide-inverse ac-simps)
next
  fix  $a$ 
  show  $a / 0 = 0$ 
    by (simp add: divide-inverse)
qed

```

There is no slick version using division by zero.

```

lemma inverse-add:
   $a \neq 0 \implies b \neq 0 \implies inverse\ a + inverse\ b = (a + b) * inverse\ a * inverse\ b$ 
  by (simp add: division-ring-inverse-add ac-simps)

```

```

lemma nonzero-mult-divide-mult-cancel-left [simp]:
  assumes [simp]:  $c \neq 0$ 
  shows  $(c * a) / (c * b) = a / b$ 
proof (cases b = 0)
  case True then show ?thesis by simp
next
  case False
  then have  $(c*a)/(c*b) = c * a * (inverse\ b * inverse\ c)$ 
    by (simp add: divide-inverse nonzero-inverse-mult-distrib)
  also have  $\dots = a * inverse\ b * (inverse\ c * c)$ 
    by (simp only: ac-simps)
  also have  $\dots = a * inverse\ b$  by simp
  finally show ?thesis by (simp add: divide-inverse)
qed

```

```

lemma nonzero-mult-divide-mult-cancel-right [simp]:
   $c \neq 0 \implies (a * c) / (b * c) = a / b$ 
  using nonzero-mult-divide-mult-cancel-left [of c a b] by (simp add: ac-simps)

```

```

lemma times-divide-eq-left [simp]:  $(b / c) * a = (b * a) / c$ 
  by (simp add: divide-inverse ac-simps)

```

```

lemma divide-inverse-commute:  $a / b = inverse\ b * a$ 
  by (simp add: divide-inverse mult.commute)

```

```

lemma add-frac-eq:
  assumes  $y \neq 0$  and  $z \neq 0$ 
  shows  $x / y + w / z = (x * z + w * y) / (y * z)$ 

```

proof –
have $x / y + w / z = (x * z) / (y * z) + (y * w) / (y * z)$
using *assms* **by** *simp*
also have $\dots = (x * z + y * w) / (y * z)$
by (*simp only: add-divide-distrib*)
finally show *?thesis*
by (*simp only: mult.commute*)
qed

Special Cancellation Simprules for Division

lemma *nonzero-divide-mult-cancel-right* [*simp*]:
 $b \neq 0 \implies b / (a * b) = 1 / a$
using *nonzero-mult-divide-mult-cancel-right* [*of b 1 a*] **by** *simp*

lemma *nonzero-divide-mult-cancel-left* [*simp*]:
 $a \neq 0 \implies a / (a * b) = 1 / b$
using *nonzero-mult-divide-mult-cancel-left* [*of a 1 b*] **by** *simp*

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*]:
 $c \neq 0 \implies (c * a) / (b * c) = a / b$
using *nonzero-mult-divide-mult-cancel-left* [*of c a b*] **by** (*simp add: ac-simps*)

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*]:
 $c \neq 0 \implies (a * c) / (c * b) = a / b$
using *nonzero-mult-divide-mult-cancel-right* [*of b c a*] **by** (*simp add: ac-simps*)

lemma *diff-frac-eq*:
 $y \neq 0 \implies z \neq 0 \implies x / y - w / z = (x * z - w * y) / (y * z)$
by (*simp add: field-simps*)

lemma *frac-eq-eq*:
 $y \neq 0 \implies z \neq 0 \implies (x / y = w / z) = (x * z = w * y)$
by (*simp add: field-simps*)

lemma *divide-minus1* [*simp*]: $x / - 1 = - x$
using *nonzero-minus-divide-right* [*of 1 x*] **by** *simp*

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
 $inverse (a * b) = inverse a * inverse b$

proof *cases*
assume $a \neq 0 \wedge b \neq 0$
thus *?thesis* **by** (*simp add: nonzero-inverse-mult-distrib ac-simps*)
next
assume $\neg (a \neq 0 \wedge b \neq 0)$
thus *?thesis* **by** *force*
qed

lemma *inverse-divide* [*simp*]:
 $inverse (a / b) = b / a$
by (*simp add: divide-inverse mult.commute*)

Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemmas *mult-divide-mult-cancel-left* = *nonzero-mult-divide-mult-cancel-left*

lemmas *mult-divide-mult-cancel-right* = *nonzero-mult-divide-mult-cancel-right*

lemma *divide-divide-eq-right* [*simp*]:
 $a / (b / c) = (a * c) / b$
by (*simp add: divide-inverse ac-simps*)

lemma *divide-divide-eq-left* [*simp*]:
 $(a / b) / c = a / (b * c)$
by (*simp add: divide-inverse mult.assoc*)

lemma *divide-divide-times-eq*:
 $(x / y) / (z / w) = (x * w) / (y * z)$
by *simp*

Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if* [*simp*]:
shows $(c * a) / (c * b) = (if\ c = 0\ then\ 0\ else\ a / b)$
by *simp*

Division and Unary Minus

lemma *minus-divide-right*:
 $-(a / b) = a / -b$
by (*simp add: divide-inverse*)

lemma *divide-minus-right* [*simp*]:
 $a / -b = -(a / b)$
by (*simp add: divide-inverse*)

lemma *minus-divide-divide*:
 $(-a) / (-b) = a / b$
by (*cases b=0*) (*simp-all add: nonzero-minus-divide-divide*)

lemma *inverse-eq-1-iff* [*simp*]:
 $inverse\ x = 1 \longleftrightarrow x = 1$
using *inverse-eq-iff-eq* [*of x 1*] **by** *simp*

lemma *divide-eq-0-iff* [*simp*]:
 $a / b = 0 \longleftrightarrow a = 0 \vee b = 0$
by (*simp add: divide-inverse*)

lemma *divide-cancel-right* [*simp*]:
 $a / c = b / c \longleftrightarrow c = 0 \vee a = b$
by (*cases c=0*) (*simp-all add: divide-inverse*)

lemma *divide-cancel-left* [*simp*]:
 $c / a = c / b \longleftrightarrow c = 0 \vee a = b$
by (*cases c=0*) (*simp-all add: divide-inverse*)

lemma *divide-eq-1-iff* [*simp*]:
 $a / b = 1 \longleftrightarrow b \neq 0 \wedge a = b$
by (*cases b=0*) (*simp-all add: right-inverse-eq*)

lemma *one-eq-divide-iff* [*simp*]:
 $1 = a / b \longleftrightarrow b \neq 0 \wedge a = b$
by (*simp add: eq-commute [of 1]*)

lemma *divide-eq-minus-1-iff*:
 $(a / b = - 1) \longleftrightarrow b \neq 0 \wedge a = - b$
using *divide-eq-1-iff* **by** *fastforce*

lemma *times-divide-times-eq*:
 $(x / y) * (z / w) = (x * z) / (y * w)$
by *simp*

lemma *add-frac-num*:
 $y \neq 0 \implies x / y + z = (x + z * y) / y$
by (*simp add: add-divide-distrib*)

lemma *add-num-frac*:
 $y \neq 0 \implies z + x / y = (x + z * y) / y$
by (*simp add: add-divide-distrib add.commute*)

lemma *dvd-field-iff*:
 $a \text{ dvd } b \longleftrightarrow (a = 0 \longrightarrow b = 0)$
proof (*cases a = 0*)
case *False*
then have $b = a * (b / a)$
by (*simp add: field-simps*)
then have $a \text{ dvd } b$..
with *False* **show** *?thesis*
by *simp*
qed *simp*

lemma *inj-divide-right* [*simp*]:
 $\text{inj } (\lambda b. b / a) \longleftrightarrow a \neq 0$
proof –
have $(\lambda b. b / a) = (*)$ (*inverse a*)
by (*simp add: field-simps fun-eq-iff*)

```

then have inj ( $\lambda y. y / a$ )  $\longleftrightarrow$  inj ((*) (inverse a))
  by simp
also have ...  $\longleftrightarrow$  inverse a  $\neq$  0
  by simp
also have ...  $\longleftrightarrow$  a  $\neq$  0
  by simp
finally show ?thesis
  by simp
qed

```

```

end

```

```

class field-char-0 = field + ring-char-0

```

18.3 Ordered fields

```

class field-abs-sgn = field + idom-abs-sgn
begin

```

```

lemma sgn-inverse [simp]:
  sgn (inverse a) = inverse (sgn a)
proof (cases a = 0)
  case True then show ?thesis by simp
next
  case False
  then have a * inverse a = 1
    by simp
  then have sgn (a * inverse a) = sgn 1
    by simp
  then have sgn a * sgn (inverse a) = 1
    by (simp add: sgn-mult)
  then have inverse (sgn a) * (sgn a * sgn (inverse a)) = inverse (sgn a) * 1
    by simp
  then have (inverse (sgn a) * sgn a) * sgn (inverse a) = inverse (sgn a)
    by (simp add: ac-simps)
  with False show ?thesis
    by (simp add: sgn-eq-0-iff)
qed

```

```

lemma abs-inverse [simp]:
  |inverse a| = inverse |a|
proof –
  from sgn-mult-abs [of inverse a] sgn-mult-abs [of a]
  have inverse (sgn a) * |inverse a| = inverse (sgn a * |a|)
    by simp
  then show ?thesis by (auto simp add: sgn-eq-0-iff)
qed

```

```

lemma sgn-divide [simp]:

```

$\text{sgn } (a / b) = \text{sgn } a / \text{sgn } b$
unfolding *divide-inverse sgn-mult* **by** *simp*

lemma *abs-divide* [*simp*]:
 $|a / b| = |a| / |b|$
unfolding *divide-inverse abs-mult* **by** *simp*

end

class *linordered-field* = *field* + *linordered-idom*
begin

lemma *positive-imp-inverse-positive*:
assumes *a-gt-0*: $0 < a$
shows $0 < \text{inverse } a$
proof –
have $0 < a * \text{inverse } a$
by (*simp add: a-gt-0 [THEN less-imp-not-eq2]*)
thus $0 < \text{inverse } a$
by (*simp add: a-gt-0 [THEN less-not-sym] zero-less-mult-iff*)
qed

lemma *negative-imp-inverse-negative*:
 $a < 0 \implies \text{inverse } a < 0$
using *positive-imp-inverse-positive* [*of -a*]
by (*simp add: nonzero-inverse-minus-eq less-imp-not-eq*)

lemma *inverse-le-imp-le*:
assumes *invle*: $\text{inverse } a \leq \text{inverse } b$ **and** *apos*: $0 < a$
shows $b \leq a$
proof (*rule classical*)
assume $\neg b \leq a$
hence $a < b$ **by** (*simp add: linorder-not-le*)
hence *bpos*: $0 < b$ **by** (*blast intro: apos less-trans*)
hence $a * \text{inverse } a \leq a * \text{inverse } b$
by (*simp add: apos invle less-imp-le mult-left-mono*)
hence $(a * \text{inverse } a) * b \leq (a * \text{inverse } b) * b$
by (*simp add: bpos less-imp-le mult-right-mono*)
thus $b \leq a$ **by** (*simp add: mult.assoc apos bpos less-imp-not-eq2*)
qed

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 < \text{inverse } a$ **and** *nz*: $a \neq 0$
shows $0 < a$
proof –
have $0 < \text{inverse } (\text{inverse } a)$
using *inv-gt-0* **by** (*rule positive-imp-inverse-positive*)
thus $0 < a$
using *nz* **by** (*simp add: nonzero-inverse-inverse-eq*)

qed

lemma *inverse-negative-imp-negative*:

assumes *inv-less-0*: $\text{inverse } a < 0$ **and** *nz*: $a \neq 0$

shows $a < 0$

proof –

have $\text{inverse } (\text{inverse } a) < 0$

using *inv-less-0* **by** (rule *negative-imp-inverse-negative*)

thus $a < 0$ **using** *nz* **by** (simp add: *nonzero-inverse-inverse-eq*)

qed

lemma *linordered-field-no-lb*:

$\forall x. \exists y. y < x$

proof

fix $x::'a$

have $m1: -(1::'a) < 0$ **by** *simp*

from *add-strict-right-mono*[*OF* *m1*, **where** $c=x$]

have $(-1) + x < x$ **by** *simp*

thus $\exists y. y < x$ **by** *blast*

qed

lemma *linordered-field-no-ub*:

$\forall x. \exists y. y > x$

proof

fix $x::'a$

have $m1: (1::'a) > 0$ **by** *simp*

from *add-strict-right-mono*[*OF* *m1*, **where** $c=x$]

have $1 + x > x$ **by** *simp*

thus $\exists y. y > x$ **by** *blast*

qed

lemma *less-imp-inverse-less*:

assumes *less*: $a < b$ **and** *apos*: $0 < a$

shows $\text{inverse } b < \text{inverse } a$

proof (rule *ccontr*)

assume $\neg \text{inverse } b < \text{inverse } a$

hence $\text{inverse } a \leq \text{inverse } b$ **by** *simp*

hence $\neg (a < b)$

by (simp add: *not-less inverse-le-imp-le* [*OF* - *apos*])

thus *False* **by** (rule *notE* [*OF* - *less*])

qed

lemma *inverse-less-imp-less*:

assumes $\text{inverse } a < \text{inverse } b$ $0 < a$

shows $b < a$

proof –

have $a \neq b$

using *assms* **by** (simp add: *less-le*)

moreover **have** $b \leq a$

using *assms* **by** (*force simp: less-le dest: inverse-le-imp-le*)
ultimately show *?thesis*
by (*simp add: less-le*)
qed

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less [simp]*:
 $0 < a \implies 0 < b \implies \text{inverse } a < \text{inverse } b \iff b < a$
by (*blast intro: less-imp-inverse-less dest: inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
 $a \leq b \implies 0 < a \implies \text{inverse } b \leq \text{inverse } a$
by (*force simp add: le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le [simp]*:
 $0 < a \implies 0 < b \implies \text{inverse } a \leq \text{inverse } b \iff b \leq a$
by (*blast intro: le-imp-inverse-le dest: inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
assumes $\text{inverse } a \leq \text{inverse } b \ b < 0$
shows $b \leq a$
proof (*rule classical*)
assume $\neg b \leq a$
with $\langle b < 0 \rangle$ **have** $a < 0$
by *force*
with *assms* **show** $b \leq a$
using *inverse-le-imp-le [of -b -a]* **by** (*simp add: nonzero-inverse-minus-eq*)
qed

lemma *less-imp-inverse-less-neg*:
assumes $a < b \ b < 0$
shows $\text{inverse } b < \text{inverse } a$
proof –
have $a < 0$
using *assms* **by** (*blast intro: less-trans*)
with *less-imp-inverse-less [of -b -a]* **show** *?thesis*
by (*simp add: nonzero-inverse-minus-eq assms*)
qed

lemma *inverse-less-imp-less-neg*:
assumes $\text{inverse } a < \text{inverse } b \ b < 0$
shows $b < a$
proof (*rule classical*)
assume $\neg b < a$
with $\langle b < 0 \rangle$ **have** $a < 0$
by *force*
with *inverse-less-imp-less [of -b -a]* **show** *?thesis*

by (*simp add: nonzero-inverse-minus-eq assms*)
qed

lemma *inverse-less-iff-less-neg* [*simp*]:
 $a < 0 \implies b < 0 \implies \text{inverse } a < \text{inverse } b \longleftrightarrow b < a$
 using *inverse-less-iff-less* [of $-b -a$]
 by (*simp del: inverse-less-iff-less add: nonzero-inverse-minus-eq*)

lemma *le-imp-inverse-le-neg*:
 $a \leq b \implies b < 0 \implies \text{inverse } b \leq \text{inverse } a$
 by (*force simp add: le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*]:
 $a < 0 \implies b < 0 \implies \text{inverse } a \leq \text{inverse } b \longleftrightarrow b \leq a$
 by (*blast intro: le-imp-inverse-le-neg dest: inverse-le-imp-le-neg*)

lemma *one-less-inverse*:
 $0 < a \implies a < 1 \implies 1 < \text{inverse } a$
 using *less-imp-inverse-less* [of $a 1$, *unfolded inverse-1*].

lemma *one-le-inverse*:
 $0 < a \implies a \leq 1 \implies 1 \leq \text{inverse } a$
 using *le-imp-inverse-le* [of $a 1$, *unfolded inverse-1*].

lemma *pos-le-divide-eq* [*field-simps*]:
 assumes $0 < c$
 shows $a \leq b / c \longleftrightarrow a * c \leq b$
proof –
 from *assms* have $a \leq b / c \longleftrightarrow a * c \leq (b / c) * c$
 using *mult-le-cancel-right* [of $a c b * \text{inverse } c$] by (*auto simp add: field-simps*)
 also have $\dots \longleftrightarrow a * c \leq b$
 by (*simp add: less-imp-not-eq2* [OF *assms*] *divide-inverse mult.assoc*)
 finally show *?thesis*.
 qed

lemma *pos-less-divide-eq* [*field-simps*]:
 assumes $0 < c$
 shows $a < b / c \longleftrightarrow a * c < b$
proof –
 from *assms* have $a < b / c \longleftrightarrow a * c < (b / c) * c$
 using *mult-less-cancel-right* [of $a c b / c$] by *auto*
 also have $\dots = (a * c < b)$
 by (*simp add: less-imp-not-eq2* [OF *assms*] *divide-inverse mult.assoc*)
 finally show *?thesis*.
 qed

lemma *neg-less-divide-eq* [*field-simps*]:
 assumes $c < 0$
 shows $a < b / c \longleftrightarrow b < a * c$

proof –

from *assms* **have** $a < b / c \longleftrightarrow (b / c) * c < a * c$

using *mult-less-cancel-right* [of $b / c c a$] **by** *auto*

also have $\dots \longleftrightarrow b < a * c$

by (*simp add: less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)

finally show *?thesis* .

qed

lemma *neg-le-divide-eq* [*field-simps*]:

assumes $c < 0$

shows $a \leq b / c \longleftrightarrow b \leq a * c$

proof –

from *assms* **have** $a \leq b / c \longleftrightarrow (b / c) * c \leq a * c$

using *mult-le-cancel-right* [of $b * inverse c c a$] **by** (*auto simp add: field-simps*)

also have $\dots \longleftrightarrow b \leq a * c$

by (*simp add: less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)

finally show *?thesis* .

qed

lemma *pos-divide-le-eq* [*field-simps*]:

assumes $0 < c$

shows $b / c \leq a \longleftrightarrow b \leq a * c$

proof –

from *assms* **have** $b / c \leq a \longleftrightarrow (b / c) * c \leq a * c$

using *mult-le-cancel-right* [of $b / c c a$] **by** *auto*

also have $\dots \longleftrightarrow b \leq a * c$

by (*simp add: less-imp-not-eq2* [*OF assms*] *divide-inverse mult.assoc*)

finally show *?thesis* .

qed

lemma *pos-divide-less-eq* [*field-simps*]:

assumes $0 < c$

shows $b / c < a \longleftrightarrow b < a * c$

proof –

from *assms* **have** $b / c < a \longleftrightarrow (b / c) * c < a * c$

using *mult-less-cancel-right* [of $b / c c a$] **by** *auto*

also have $\dots \longleftrightarrow b < a * c$

by (*simp add: less-imp-not-eq2* [*OF assms*] *divide-inverse mult.assoc*)

finally show *?thesis* .

qed

lemma *neg-divide-le-eq* [*field-simps*]:

assumes $c < 0$

shows $b / c \leq a \longleftrightarrow a * c \leq b$

proof –

from *assms* **have** $b / c \leq a \longleftrightarrow a * c \leq (b / c) * c$

using *mult-le-cancel-right* [of $a c b / c$] **by** *auto*

also have $\dots \longleftrightarrow a * c \leq b$

by (*simp add: less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)

finally show *?thesis* .
qed

lemma *neg-divide-less-eq* [*field-simps*]:

assumes $c < 0$

shows $b / c < a \iff a * c < b$

proof –

from *assms* **have** $b / c < a \iff a * c < b / c * c$

using *mult-less-cancel-right* [*of a c b / c*] **by** *auto*

also have $\dots \iff a * c < b$

by (*simp add: less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)

finally show *?thesis* .

qed

The following *field-simps* rules are necessary, as minus is always moved atop of division but we want to get rid of division.

lemma *pos-le-minus-divide-eq* [*field-simps*]: $0 < c \implies a \leq - (b / c) \iff a * c \leq - b$

unfolding *minus-divide-left* **by** (*rule pos-le-divide-eq*)

lemma *neg-le-minus-divide-eq* [*field-simps*]: $c < 0 \implies a \leq - (b / c) \iff - b \leq a * c$

unfolding *minus-divide-left* **by** (*rule neg-le-divide-eq*)

lemma *pos-less-minus-divide-eq* [*field-simps*]: $0 < c \implies a < - (b / c) \iff a * c < - b$

unfolding *minus-divide-left* **by** (*rule pos-less-divide-eq*)

lemma *neg-less-minus-divide-eq* [*field-simps*]: $c < 0 \implies a < - (b / c) \iff - b < a * c$

unfolding *minus-divide-left* **by** (*rule neg-less-divide-eq*)

lemma *pos-minus-divide-less-eq* [*field-simps*]: $0 < c \implies - (b / c) < a \iff - b < a * c$

unfolding *minus-divide-left* **by** (*rule pos-divide-less-eq*)

lemma *neg-minus-divide-less-eq* [*field-simps*]: $c < 0 \implies - (b / c) < a \iff a * c < - b$

unfolding *minus-divide-left* **by** (*rule neg-divide-less-eq*)

lemma *pos-minus-divide-le-eq* [*field-simps*]: $0 < c \implies - (b / c) \leq a \iff - b \leq a * c$

unfolding *minus-divide-left* **by** (*rule pos-divide-le-eq*)

lemma *neg-minus-divide-le-eq* [*field-simps*]: $c < 0 \implies - (b / c) \leq a \iff a * c \leq - b$

unfolding *minus-divide-left* **by** (*rule neg-divide-le-eq*)

lemma *frac-less-eq*:

$y \neq 0 \implies z \neq 0 \implies x / y < w / z \iff (x * z - w * y) / (y * z) < 0$
by (subst less-iff-diff-less-0) (simp add: diff-frac-eq)

lemma *frac-le-eq*:

$y \neq 0 \implies z \neq 0 \implies x / y \leq w / z \iff (x * z - w * y) / (y * z) \leq 0$
by (subst le-iff-diff-le-0) (simp add: diff-frac-eq)

lemma *divide-pos-pos[simp]*:

$0 < x \implies 0 < y \implies 0 < x / y$
by(simp add:field-simps)

lemma *divide-nonneg-pos*:

$0 \leq x \implies 0 < y \implies 0 \leq x / y$
by(simp add:field-simps)

lemma *divide-neg-pos*:

$x < 0 \implies 0 < y \implies x / y < 0$
by(simp add:field-simps)

lemma *divide-nonpos-pos*:

$x \leq 0 \implies 0 < y \implies x / y \leq 0$
by(simp add:field-simps)

lemma *divide-pos-neg*:

$0 < x \implies y < 0 \implies x / y < 0$
by(simp add:field-simps)

lemma *divide-nonneg-neg*:

$0 \leq x \implies y < 0 \implies x / y \leq 0$
by(simp add:field-simps)

lemma *divide-neg-neg*:

$x < 0 \implies y < 0 \implies 0 < x / y$
by(simp add:field-simps)

lemma *divide-nonpos-neg*:

$x \leq 0 \implies y < 0 \implies 0 \leq x / y$
by(simp add:field-simps)

lemma *divide-strict-right-mono*:

$\llbracket a < b; 0 < c \rrbracket \implies a / c < b / c$
by (simp add: less-imp-not-eq2 divide-inverse mult-strict-right-mono
 positive-imp-inverse-positive)

lemma *divide-strict-right-mono-neg*:

assumes $b < a$ $c < 0$ **shows** $a / c < b / c$

proof –

have $b / -c < a / -c$

```

  by (rule divide-strict-right-mono) (use assms in auto)
  then show ?thesis
  by (simp add: less-imp-not-eq)
qed

```

The last premise ensures that a and b have the same sign

```

lemma divide-strict-left-mono:
   $\llbracket b < a; 0 < c; 0 < a*b \rrbracket \implies c / a < c / b$ 
  by (auto simp: field-simps zero-less-mult-iff mult-strict-right-mono)

```

```

lemma divide-left-mono:
   $\llbracket b \leq a; 0 \leq c; 0 < a*b \rrbracket \implies c / a \leq c / b$ 
  by (auto simp: field-simps zero-less-mult-iff mult-right-mono)

```

```

lemma divide-strict-left-mono-neg:
   $\llbracket a < b; c < 0; 0 < a*b \rrbracket \implies c / a < c / b$ 
  by (auto simp: field-simps zero-less-mult-iff mult-strict-right-mono-neg)

```

```

lemma mult-imp-div-pos-le:  $0 < y \implies x \leq z * y \implies x / y \leq z$ 
  by (subst pos-divide-le-eq, assumption+)

```

```

lemma mult-imp-le-div-pos:  $0 < y \implies z * y \leq x \implies z \leq x / y$ 
  by (simp add: field-simps)

```

```

lemma mult-imp-div-pos-less:  $0 < y \implies x < z * y \implies x / y < z$ 
  by (simp add: field-simps)

```

```

lemma mult-imp-less-div-pos:  $0 < y \implies z * y < x \implies z < x / y$ 
  by (simp add: field-simps)

```

```

lemma frac-le:
  assumes  $0 \leq y \ x \leq y \ 0 < w \ w \leq z$ 
  shows  $x / z \leq y / w$ 
proof (rule mult-imp-div-pos-le)
  show  $z > 0$ 
  using assms by simp
  have  $x \leq y * z / w$ 
  proof (rule mult-imp-le-div-pos [OF ‹ $0 < w$ ›])
    show  $x * w \leq y * z$ 
    using assms by (auto intro: mult-mono)
  qed
  also have  $\dots = y / w * z$ 
  by simp
  finally show  $x \leq y / w * z$  .
qed

```

```

lemma frac-less:
  assumes  $0 \leq x \ x < y \ 0 < w \ w \leq z$ 
  shows  $x / z < y / w$ 

```

```

proof (rule mult-imp-div-pos-less)
  show  $z > 0$ 
    using assms by simp
  have  $x < y * z / w$ 
  proof (rule mult-imp-less-div-pos [OF ‹ $0 < w$ ›])
    show  $x * w < y * z$ 
    using assms by (auto intro: mult-less-le-imp-less)
  qed
  also have  $\dots = y / w * z$ 
    by simp
  finally show  $x < y / w * z$  .
qed

```

```

lemma frac-less2:
  assumes  $0 < x \ x \leq y \ 0 < w \ w < z$ 
  shows  $x / z < y / w$ 
proof (rule mult-imp-div-pos-less)
  show  $z > 0$ 
    using assms by simp
  show  $x < y / w * z$ 
    using assms by (force intro: mult-imp-less-div-pos mult-le-less-imp-less)
qed

```

```

lemma less-half-sum:  $a < b \implies a < (a+b) / (1+1)$ 
  by (simp add: field-simps zero-less-two)

```

```

lemma gt-half-sum:  $a < b \implies (a+b)/(1+1) < b$ 
  by (simp add: field-simps zero-less-two)

```

```

subclass unbounded-dense-linorder
proof
  fix  $x \ y :: 'a$ 
  from less-add-one show  $\exists y. x < y$  ..
  from less-add-one have  $x + (-1) < (x + 1) + (-1)$  by (rule add-strict-right-mono)
  then have  $x - 1 < x + 1 - 1$  by simp
  then have  $x - 1 < x$  by (simp add: algebra-simps)
  then show  $\exists y. y < x$  ..
  show  $x < y \implies \exists z > x. z < y$  by (blast intro!: less-half-sum gt-half-sum)
qed

```

```

subclass field-abs-sgn ..

```

```

lemma inverse-sgn [simp]:
   $\text{inverse} (\text{sgn } a) = \text{sgn } a$ 
  by (cases  $a \ 0$  rule: linorder-cases) simp-all

```

```

lemma divide-sgn [simp]:
   $a / \text{sgn } b = a * \text{sgn } b$ 
  by (cases  $b \ 0$  rule: linorder-cases) simp-all

```

lemma nonzero-abs-inverse:

$a \neq 0 \implies |\text{inverse } a| = \text{inverse } |a|$
by (rule abs-inverse)

lemma nonzero-abs-divide:

$b \neq 0 \implies |a / b| = |a| / |b|$
by (rule abs-divide)

lemma field-le-epsilon:

assumes $e: \bigwedge e. 0 < e \implies x \leq y + e$
shows $x \leq y$

proof (rule dense-le)

fix t **assume** $t < x$

hence $0 < x - t$ **by** (simp add: less-diff-eq)

from e [OF this] **have** $x + 0 \leq x + (y - t)$ **by** (simp add: algebra-simps)

then have $0 \leq y - t$ **by** (simp only: add-le-cancel-left)

then show $t \leq y$ **by** (simp add: algebra-simps)

qed

lemma inverse-positive-iff-positive [simp]: $(0 < \text{inverse } a) = (0 < a)$

proof (cases $a = 0$)

case *False*

then show ?thesis

by (blast intro: inverse-positive-imp-positive positive-imp-inverse-positive)

qed auto

lemma inverse-negative-iff-negative [simp]: $(\text{inverse } a < 0) = (a < 0)$

proof (cases $a = 0$)

case *False*

then show ?thesis

by (blast intro: inverse-negative-imp-negative negative-imp-inverse-negative)

qed auto

lemma inverse-nonnegative-iff-nonnegative [simp]: $0 \leq \text{inverse } a \iff 0 \leq a$

by (simp add: not-less [symmetric])

lemma inverse-nonpositive-iff-nonpositive [simp]: $\text{inverse } a \leq 0 \iff a \leq 0$

by (simp add: not-less [symmetric])

lemma one-less-inverse-iff: $1 < \text{inverse } x \iff 0 < x \wedge x < 1$

using less-trans[of 1 x 0 for x]

by (cases x 0 rule: linorder-cases) (auto simp add: field-simps)

lemma one-le-inverse-iff: $1 \leq \text{inverse } x \iff 0 < x \wedge x \leq 1$

proof (cases $x = 1$)

case *True* **then show** ?thesis **by** simp

next

case *False* **then have** $\text{inverse } x \neq 1$ **by** simp

then have $1 \neq \text{inverse } x$ **by** *blast*
 then have $1 \leq \text{inverse } x \iff 1 < \text{inverse } x$ **by** (*simp add: le-less*)
 with *False* **show** *?thesis* **by** (*auto simp add: one-less-inverse-iff*)
qed

lemma *inverse-less-1-iff*: $\text{inverse } x < 1 \iff x \leq 0 \vee 1 < x$
by (*simp add: not-le [symmetric] one-le-inverse-iff*)

lemma *inverse-le-1-iff*: $\text{inverse } x \leq 1 \iff x \leq 0 \vee 1 \leq x$
by (*simp add: not-less [symmetric] one-less-inverse-iff*)

lemma [*field-split-simps, divide-simps*]:
shows *le-divide-eq*: $a \leq b / c \iff (\text{if } 0 < c \text{ then } a * c \leq b \text{ else if } c < 0 \text{ then } b \leq a * c \text{ else } a \leq 0)$
and *divide-le-eq*: $b / c \leq a \iff (\text{if } 0 < c \text{ then } b \leq a * c \text{ else if } c < 0 \text{ then } a * c \leq b \text{ else } 0 \leq a)$
and *less-divide-eq*: $a < b / c \iff (\text{if } 0 < c \text{ then } a * c < b \text{ else if } c < 0 \text{ then } b < a * c \text{ else } a < 0)$
and *divide-less-eq*: $b / c < a \iff (\text{if } 0 < c \text{ then } b < a * c \text{ else if } c < 0 \text{ then } a * c < b \text{ else } 0 < a)$
and *le-minus-divide-eq*: $a \leq -(b / c) \iff (\text{if } 0 < c \text{ then } a * c \leq -b \text{ else if } c < 0 \text{ then } -b \leq a * c \text{ else } a \leq 0)$
and *minus-divide-le-eq*: $-(b / c) \leq a \iff (\text{if } 0 < c \text{ then } -b \leq a * c \text{ else if } c < 0 \text{ then } a * c \leq -b \text{ else } 0 \leq a)$
and *less-minus-divide-eq*: $a < -(b / c) \iff (\text{if } 0 < c \text{ then } a * c < -b \text{ else if } c < 0 \text{ then } -b < a * c \text{ else } a < 0)$
and *minus-divide-less-eq*: $-(b / c) < a \iff (\text{if } 0 < c \text{ then } -b < a * c \text{ else if } c < 0 \text{ then } a * c < -b \text{ else } 0 < a)$
by (*auto simp: field-simps not-less dest: order.antisym*)

Division and Signs

lemma
shows *zero-less-divide-iff*: $0 < a / b \iff 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
and *divide-less-0-iff*: $a / b < 0 \iff 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
and *zero-le-divide-iff*: $0 \leq a / b \iff 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$
and *divide-le-0-iff*: $a / b \leq 0 \iff 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
by (*auto simp add: field-split-simps*)

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp*]: $0 = 1 / a \iff a = 0$
by (*cases a = 0*) (*auto simp: field-simps*)

lemma *one-divide-eq-0-iff* [*simp*]: $1 / a = 0 \iff a = 0$
using *zero-eq-1-divide-iff* [*of a*] **by** *simp*

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma *zero-le-divide-1-iff* [*simp*]:

$0 \leq 1 / a \iff 0 \leq a$
by (*simp add: zero-le-divide-iff*)

lemma *zero-less-divide-1-iff* [*simp*]:
 $0 < 1 / a \iff 0 < a$
by (*simp add: zero-less-divide-iff*)

lemma *divide-le-0-1-iff* [*simp*]:
 $1 / a \leq 0 \iff a \leq 0$
by (*simp add: divide-le-0-iff*)

lemma *divide-less-0-1-iff* [*simp*]:
 $1 / a < 0 \iff a < 0$
by (*simp add: divide-less-0-iff*)

lemma *divide-right-mono*:
 $\llbracket a \leq b; 0 \leq c \rrbracket \implies a/c \leq b/c$
by (*force simp add: divide-strict-right-mono le-less*)

lemma *divide-right-mono-neg*: $a \leq b \implies c \leq 0 \implies b / c \leq a / c$
by (*auto dest: divide-right-mono [of - - - c]*)

lemma *divide-left-mono-neg*: $a \leq b \implies c \leq 0 \implies 0 < a * b \implies c / a \leq c / b$
by (*auto simp add: mult.commute dest: divide-left-mono [of - - - c]*)

lemma *inverse-le-iff*: $\text{inverse } a \leq \text{inverse } b \iff (0 < a * b \implies b \leq a) \wedge (a * b \leq 0 \implies a \leq b)$
by (*cases a 0 b 0 rule: linorder-cases[case-product linorder-cases]*)
(auto simp add: field-simps zero-less-mult-iff mult-le-0-iff)

lemma *inverse-less-iff*: $\text{inverse } a < \text{inverse } b \iff (0 < a * b \implies b < a) \wedge (a * b \leq 0 \implies a < b)$
by (*subst less-le*) (*auto simp: inverse-le-iff*)

lemma *divide-le-cancel*: $a / c \leq b / c \iff (0 < c \implies a \leq b) \wedge (c < 0 \implies b \leq a)$
by (*simp add: divide-inverse mult-le-cancel-right*)

lemma *divide-less-cancel*: $a / c < b / c \iff (0 < c \implies a < b) \wedge (c < 0 \implies b < a) \wedge c \neq 0$
by (*auto simp add: divide-inverse mult-less-cancel-right*)

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1*:
 $(1 \leq b / a) = ((0 < a \wedge a \leq b) \vee (a < 0 \wedge b \leq a))$
by (*auto simp add: le-divide-eq*)

lemma *divide-le-eq-1*:
 $(b / a \leq 1) = ((0 < a \wedge b \leq a) \vee (a < 0 \wedge a \leq b) \vee a=0)$

by (*auto simp add: divide-le-eq*)

lemma *less-divide-eq-1*:

$$(1 < b / a) = ((0 < a \wedge a < b) \vee (a < 0 \wedge b < a))$$

by (*auto simp add: less-divide-eq*)

lemma *divide-less-eq-1*:

$$(b / a < 1) = ((0 < a \wedge b < a) \vee (a < 0 \wedge a < b) \vee a=0)$$

by (*auto simp add: divide-less-eq*)

lemma *divide-nonneg-nonneg* [*simp*]:

$$0 \leq x \implies 0 \leq y \implies 0 \leq x / y$$

by (*auto simp add: field-split-simps*)

lemma *divide-nonpos-nonpos*:

$$x \leq 0 \implies y \leq 0 \implies 0 \leq x / y$$

by (*auto simp add: field-split-simps*)

lemma *divide-nonneg-nonpos*:

$$0 \leq x \implies y \leq 0 \implies x / y \leq 0$$

by (*auto simp add: field-split-simps*)

lemma *divide-nonpos-nonneg*:

$$x \leq 0 \implies 0 \leq y \implies x / y \leq 0$$

by (*auto simp add: field-split-simps*)

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*]:

$$0 < a \implies (1 \leq b/a) = (a \leq b)$$

by (*auto simp add: le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*]:

$$a < 0 \implies (1 \leq b/a) = (b \leq a)$$

by (*auto simp add: le-divide-eq*)

lemma *divide-le-eq-1-pos* [*simp*]:

$$0 < a \implies (b/a \leq 1) = (b \leq a)$$

by (*auto simp add: divide-le-eq*)

lemma *divide-le-eq-1-neg* [*simp*]:

$$a < 0 \implies (b/a \leq 1) = (a \leq b)$$

by (*auto simp add: divide-le-eq*)

lemma *less-divide-eq-1-pos* [*simp*]:

$$0 < a \implies (1 < b/a) = (a < b)$$

by (*auto simp add: less-divide-eq*)

lemma *less-divide-eq-1-neg* [*simp*]:

$$a < 0 \implies (1 < b/a) = (b < a)$$

by (auto simp add: less-divide-eq)

lemma *divide-less-eq-1-pos* [simp]:
 $0 < a \implies (b/a < 1) = (b < a)$
 by (auto simp add: divide-less-eq)

lemma *divide-less-eq-1-neg* [simp]:
 $a < 0 \implies b/a < 1 \longleftrightarrow a < b$
 by (auto simp add: divide-less-eq)

lemma *eq-divide-eq-1* [simp]:
 $(1 = b/a) = ((a \neq 0 \wedge a = b))$
 by (auto simp add: eq-divide-eq)

lemma *divide-eq-eq-1* [simp]:
 $(b/a = 1) = ((a \neq 0 \wedge a = b))$
 by (auto simp add: divide-eq-eq)

lemma *abs-div-pos*: $0 < y \implies |x| / y = |x / y|$
 by (simp add: order-less-imp-le)

lemma *zero-le-divide-abs-iff* [simp]: $(0 \leq a / |b|) = (0 \leq a \vee b = 0)$
 by (auto simp: zero-le-divide-iff)

lemma *divide-le-0-abs-iff* [simp]: $(a / |b| \leq 0) = (a \leq 0 \vee b = 0)$
 by (auto simp: divide-le-0-iff)

lemma *field-le-mult-one-interval*:
 assumes *: $\bigwedge z. [0 < z ; z < 1] \implies z * x \leq y$
 shows $x \leq y$

proof (cases $0 < x$)

assume $0 < x$

thus ?thesis

using *dense-le-bounded*[of 0 1 y/x] *

unfolding *le-divide-eq if-P*[OF $\langle 0 < x \rangle$] by simp

next

assume $\neg 0 < x$ hence $x \leq 0$ by simp

obtain $s::'a$ where $s: 0 < s \wedge s < 1$ using *dense*[of 0 1:: $'a$] by auto

hence $x \leq s * x$ using *mult-le-cancel-right*[of 1 x s] $\langle x \leq 0 \rangle$ by auto

also note $*[OF s]$

finally show ?thesis .

qed

For creating values between u and v .

lemma *scaling-mono*:
 assumes $u \leq v \wedge 0 \leq r \wedge r \leq s$
 shows $u + r * (v - u) / s \leq v$
proof –
 have $r/s \leq 1$ using *assms*

```

    using divide-le-eq-1 by fastforce
    moreover have  $0 \leq v - u$ 
    using assms by simp
    ultimately have  $(r/s) * (v - u) \leq 1 * (v - u)$ 
    by (rule mult-right-mono)
    then show ?thesis
    by (simp add: field-simps)
qed

```

end

Min/max Simplification Rules

```

lemma min-mult-distrib-left:
  fixes  $x::'a::\text{linordered-idom}$ 
  shows  $p * \min x y = (\text{if } 0 \leq p \text{ then } \min (p*x) (p*y) \text{ else } \max (p*x) (p*y))$ 
by (auto simp add: min-def max-def mult-le-cancel-left)

```

```

lemma min-mult-distrib-right:
  fixes  $x::'a::\text{linordered-idom}$ 
  shows  $\min x y * p = (\text{if } 0 \leq p \text{ then } \min (x*p) (y*p) \text{ else } \max (x*p) (y*p))$ 
by (auto simp add: min-def max-def mult-le-cancel-right)

```

```

lemma min-divide-distrib-right:
  fixes  $x::'a::\text{linordered-field}$ 
  shows  $\min x y / p = (\text{if } 0 \leq p \text{ then } \min (x/p) (y/p) \text{ else } \max (x/p) (y/p))$ 
by (simp add: min-mult-distrib-right divide-inverse)

```

```

lemma max-mult-distrib-left:
  fixes  $x::'a::\text{linordered-idom}$ 
  shows  $p * \max x y = (\text{if } 0 \leq p \text{ then } \max (p*x) (p*y) \text{ else } \min (p*x) (p*y))$ 
by (auto simp add: min-def max-def mult-le-cancel-left)

```

```

lemma max-mult-distrib-right:
  fixes  $x::'a::\text{linordered-idom}$ 
  shows  $\max x y * p = (\text{if } 0 \leq p \text{ then } \max (x*p) (y*p) \text{ else } \min (x*p) (y*p))$ 
by (auto simp add: min-def max-def mult-le-cancel-right)

```

```

lemma max-divide-distrib-right:
  fixes  $x::'a::\text{linordered-field}$ 
  shows  $\max x y / p = (\text{if } 0 \leq p \text{ then } \max (x/p) (y/p) \text{ else } \min (x/p) (y/p))$ 
by (simp add: max-mult-distrib-right divide-inverse)

```

hide-fact (open) *field-inverse field-divide-inverse field-inverse-zero*

code-identifier

code-module *Fields* \rightarrow (*SML*) *Arith* and (*OCaml*) *Arith* and (*Haskell*) *Arith*

end

19 Relations – as sets of pairs, and binary predicates

```
theory Relation
  imports Product-Type Sum-Type Fields
begin
```

A preliminary: classical rules for reasoning on predicates

```
declare predicate1I [Pure.intro!, intro!]
declare predicate1D [Pure.dest, dest]
declare predicate2I [Pure.intro!, intro!]
declare predicate2D [Pure.dest, dest]
declare bot1E [elim!]
declare bot2E [elim!]
declare top1I [intro!]
declare top2I [intro!]
declare inf1I [intro!]
declare inf2I [intro!]
declare inf1E [elim!]
declare inf2E [elim!]
declare sup1I1 [intro?]
declare sup2I1 [intro?]
declare sup1I2 [intro?]
declare sup2I2 [intro?]
declare sup1E [elim!]
declare sup2E [elim!]
declare sup1CI [intro!]
declare sup2CI [intro!]
declare Inf1-I [intro!]
declare INF1-I [intro!]
declare Inf2-I [intro!]
declare INF2-I [intro!]
declare Inf1-D [elim]
declare INF1-D [elim]
declare Inf2-D [elim]
declare INF2-D [elim]
declare Inf1-E [elim]
declare INF1-E [elim]
declare Inf2-E [elim]
declare INF2-E [elim]
declare Sup1-I [intro]
declare SUP1-I [intro]
declare Sup2-I [intro]
declare SUP2-I [intro]
declare Sup1-E [elim!]
declare SUP1-E [elim!]
declare Sup2-E [elim!]
declare SUP2-E [elim!]
```

19.1 Fundamental

19.1.1 Relations as sets of pairs

type-synonym $'a\ rel = ('a \times 'a)\ set$

lemma *subrelI*: $(\bigwedge x\ y. (x, y) \in r \implies (x, y) \in s) \implies r \subseteq s$
 — Version of *subsetI* for binary relations
by *auto*

lemma *lfp-induct2*:
 $(a, b) \in \text{lfp } f \implies \text{mono } f \implies$
 $(\bigwedge a\ b. (a, b) \in f (\text{lfp } f \cap \{(x, y). P\ x\ y\})) \implies P\ a\ b \implies P\ a\ b$
 — Version of *lfp-induct* for binary relations
using *lfp-induct-set* [of $(a, b)\ f\ \text{case-prod } P$] **by** *auto*

19.1.2 Conversions between set and predicate relations

lemma *pred-equals-eq* [*pred-set-conv*]: $(\lambda x. x \in R) = (\lambda x. x \in S) \longleftrightarrow R = S$
by (*simp add: set-eq-iff fun-eq-iff*)

lemma *pred-equals-eq2* [*pred-set-conv*]: $(\lambda x\ y. (x, y) \in R) = (\lambda x\ y. (x, y) \in S)$
 $\longleftrightarrow R = S$
by (*simp add: set-eq-iff fun-eq-iff*)

lemma *pred-subset-eq* [*pred-set-conv*]: $(\lambda x. x \in R) \leq (\lambda x. x \in S) \longleftrightarrow R \subseteq S$
by (*simp add: subset-iff le-fun-def*)

lemma *pred-subset-eq2* [*pred-set-conv*]: $(\lambda x\ y. (x, y) \in R) \leq (\lambda x\ y. (x, y) \in S)$
 $\longleftrightarrow R \subseteq S$
by (*simp add: subset-iff le-fun-def*)

lemma *bot-empty-eq* [*pred-set-conv*]: $\perp = (\lambda x. x \in \{\})$
by (*auto simp add: fun-eq-iff*)

lemma *bot-empty-eq2* [*pred-set-conv*]: $\perp = (\lambda x\ y. (x, y) \in \{\})$
by (*auto simp add: fun-eq-iff*)

lemma *top-empty-eq*: $\top = (\lambda x. x \in UNIV)$
by (*auto simp add: fun-eq-iff*)

lemma *top-empty-eq2*: $\top = (\lambda x\ y. (x, y) \in UNIV)$
by (*auto simp add: fun-eq-iff*)

lemma *inf-Int-eq* [*pred-set-conv*]: $(\lambda x. x \in R) \sqcap (\lambda x. x \in S) = (\lambda x. x \in R \cap S)$
by (*simp add: inf-fun-def*)

lemma *inf-Int-eq2* [*pred-set-conv*]: $(\lambda x\ y. (x, y) \in R) \sqcap (\lambda x\ y. (x, y) \in S) = (\lambda x\ y. (x, y) \in R \cap S)$
by (*simp add: inf-fun-def*)

lemma *sup-Un-eq* [*pred-set-conv*]: $(\lambda x. x \in R) \sqcup (\lambda x. x \in S) = (\lambda x. x \in R \cup S)$
by (*simp add: sup-fun-def*)

lemma *sup-Un-eq2* [*pred-set-conv*]: $(\lambda x y. (x, y) \in R) \sqcup (\lambda x y. (x, y) \in S) = (\lambda x y. (x, y) \in R \cup S)$
by (*simp add: sup-fun-def*)

lemma *INF-INT-eq* [*pred-set-conv*]: $(\prod_{i \in S}. (\lambda x. x \in r i)) = (\lambda x. x \in (\bigcap_{i \in S}. r i))$
by (*simp add: fun-eq-iff*)

lemma *INF-INT-eq2* [*pred-set-conv*]: $(\prod_{i \in S}. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in (\bigcap_{i \in S}. r i))$
by (*simp add: fun-eq-iff*)

lemma *SUP-UN-eq* [*pred-set-conv*]: $(\bigsqcup_{i \in S}. (\lambda x. x \in r i)) = (\lambda x. x \in (\bigcup_{i \in S}. r i))$
by (*simp add: fun-eq-iff*)

lemma *SUP-UN-eq2* [*pred-set-conv*]: $(\bigsqcup_{i \in S}. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in (\bigcup_{i \in S}. r i))$
by (*simp add: fun-eq-iff*)

lemma *Inf-INT-eq* [*pred-set-conv*]: $\prod S = (\lambda x. x \in (\bigcap (\text{Collect } 'S)))$
by (*simp add: fun-eq-iff*)

lemma *INF-Int-eq* [*pred-set-conv*]: $(\prod_{i \in S}. (\lambda x. x \in i)) = (\lambda x. x \in \bigcap S)$
by (*simp add: fun-eq-iff*)

lemma *Inf-INT-eq2* [*pred-set-conv*]: $\prod S = (\lambda x y. (x, y) \in (\bigcap (\text{Collect } 'case-prod 'S)))$
by (*simp add: fun-eq-iff*)

lemma *INF-Int-eq2* [*pred-set-conv*]: $(\prod_{i \in S}. (\lambda x y. (x, y) \in i)) = (\lambda x y. (x, y) \in \bigcap S)$
by (*simp add: fun-eq-iff*)

lemma *Sup-SUP-eq* [*pred-set-conv*]: $\bigsqcup S = (\lambda x. x \in \bigcup (\text{Collect } 'S))$
by (*simp add: fun-eq-iff*)

lemma *SUP-Sup-eq* [*pred-set-conv*]: $(\bigsqcup_{i \in S}. (\lambda x. x \in i)) = (\lambda x. x \in \bigcup S)$
by (*simp add: fun-eq-iff*)

lemma *Sup-SUP-eq2* [*pred-set-conv*]: $\bigsqcup S = (\lambda x y. (x, y) \in (\bigcup (\text{Collect } 'case-prod 'S)))$
by (*simp add: fun-eq-iff*)

lemma *SUP-Sup-eq2* [*pred-set-conv*]: $(\bigsqcup_{i \in S}. (\lambda x y. (x, y) \in i)) = (\lambda x y. (x, y) \in \bigcup S)$

$\in \bigcup S$
by (*simp add: fun-eq-iff*)

19.2 Properties of relations

19.2.1 Reflexivity

definition *refl-on* :: 'a set \Rightarrow 'a rel \Rightarrow bool
where *refl-on* A r \longleftrightarrow $r \subseteq A \times A \wedge (\forall x \in A. (x, x) \in r)$

abbreviation *refl* :: 'a rel \Rightarrow bool — reflexivity over a type
where *refl* \equiv *refl-on UNIV*

definition *reflp-on* :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool
where *reflp-on* A R \longleftrightarrow $(\forall x \in A. R x x)$

abbreviation *reflp* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool
where *reflp* \equiv *reflp-on UNIV*

lemma *reflp-def[no-atp]*: *reflp* R \longleftrightarrow $(\forall x. R x x)$
by (*simp add: reflp-on-def*)

reflp-def is for backward compatibility.

lemma *reflp-refl-eq [pred-set-conv]*: *reflp* $(\lambda x y. (x, y) \in r)$ \longleftrightarrow *refl* r
by (*simp add: refl-on-def reflp-def*)

lemma *refl-onI [intro?]*: $r \subseteq A \times A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow (x, x) \in r) \Longrightarrow$ *refl-on* A r
unfolding *refl-on-def* **by** (*iprover intro!: ballI*)

lemma *reflI*: $(\bigwedge x. (x, x) \in r) \Longrightarrow$ *refl* r
by (*auto intro: refl-onI*)

lemma *reflp-onI*:
 $(\bigwedge x. x \in A \Longrightarrow R x x) \Longrightarrow$ *reflp-on* A R
by (*simp add: reflp-on-def*)

lemma *reflpI[intro?]*: $(\bigwedge x. R x x) \Longrightarrow$ *reflp* R
by (*rule reflp-onI*)

lemma *refl-onD*: *refl-on* A r $\Longrightarrow a \in A \Longrightarrow (a, a) \in r$
unfolding *refl-on-def* **by** *blast*

lemma *refl-onD1*: *refl-on* A r $\Longrightarrow (x, y) \in r \Longrightarrow x \in A$
unfolding *refl-on-def* **by** *blast*

lemma *refl-onD2*: *refl-on* A r $\Longrightarrow (x, y) \in r \Longrightarrow y \in A$
unfolding *refl-on-def* **by** *blast*

lemma *reflD*: *refl* r $\Longrightarrow (a, a) \in r$

unfolding *refl-on-def* **by** *blast*

lemma *reflp-onD*:

reflp-on A $R \implies x \in A \implies R$ x x
by (*simp add: reflp-on-def*)

lemma *reflpD[dest?]*: *reflp* $R \implies R$ x x

by (*simp add: reflp-onD*)

lemma *reflpE*:

assumes *reflp* r

obtains r x x

using *assms* **by** (*auto dest: refl-onD simp add: reflp-def*)

lemma *reflp-on-subset*: *reflp-on* A $R \implies B \subseteq A \implies$ *reflp-on* B R

by (*auto intro: reflp-onI dest: reflp-onD*)

lemma *refl-on-Int*: *refl-on* A $r \implies$ *refl-on* B $s \implies$ *refl-on* $(A \cap B)$ $(r \cap s)$

unfolding *refl-on-def* **by** *blast*

lemma *reflp-on-inf*: *reflp-on* A $R \implies$ *reflp-on* B $S \implies$ *reflp-on* $(A \cap B)$ $(R \sqcap S)$

by (*auto intro: reflp-onI dest: reflp-onD*)

lemma *reflp-inf*: *reflp* $r \implies$ *reflp* $s \implies$ *reflp* $(r \sqcap s)$

by (*rule reflp-on-inf[of UNIV - UNIV, unfolded Int-absorb]*)

lemma *refl-on-Un*: *refl-on* A $r \implies$ *refl-on* B $s \implies$ *refl-on* $(A \cup B)$ $(r \cup s)$

unfolding *refl-on-def* **by** *blast*

lemma *reflp-on-sup*: *reflp-on* A $R \implies$ *reflp-on* B $S \implies$ *reflp-on* $(A \cup B)$ $(R \sqcup S)$

by (*auto intro: reflp-onI dest: reflp-onD*)

lemma *reflp-sup*: *reflp* $r \implies$ *reflp* $s \implies$ *reflp* $(r \sqcup s)$

by (*rule reflp-on-sup[of UNIV - UNIV, unfolded Un-absorb]*)

lemma *refl-on-INTER*: $\forall x \in S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\bigcap (A \ ' S)) \ (\bigcap (r \ ' S))$

unfolding *refl-on-def* **by** *fast*

lemma *reflp-on-Inf*: $\forall x \in S. \text{reflp-on } (A \ x) \ (R \ x) \implies \text{reflp-on } (\bigcap (A \ ' S)) \ (\bigcap (R \ ' S))$

by (*auto intro: reflp-onI dest: reflp-onD*)

lemma *refl-on-UNION*: $\forall x \in S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\bigcup (A \ ' S)) \ (\bigcup (r \ ' S))$

unfolding *refl-on-def* **by** *blast*

lemma *reflp-on-Sup*: $\forall x \in S. \text{reflp-on } (A \ x) \ (R \ x) \implies \text{reflp-on } (\bigcup (A \ ' S)) \ (\bigcup (R \ ' S))$

by (auto intro: reflp-onI dest: reflp-onD)

lemma *refl-on-empty* [simp]: *refl-on* {} {}
by (simp add: refl-on-def)

lemma *reflp-on-empty* [simp]: *reflp-on* {} *R*
by (auto intro: reflp-onI)

lemma *refl-on-singleton* [simp]: *refl-on* {*x*} {(*x*, *x*)}
by (blast intro: refl-onI)

lemma *refl-on-def'* [nitpick-unfold, code]:
 $\text{refl-on } A \ r \longleftrightarrow (\forall (x, y) \in r. x \in A \wedge y \in A) \wedge (\forall x \in A. (x, x) \in r)$
by (auto intro: refl-onI dest: refl-onD refl-onD1 refl-onD2)

lemma *reflp-on-equality* [simp]: *reflp-on* *A* (=)
by (simp add: reflp-on-def)

lemma *reflp-on-mono*:
 $\text{reflp-on } A \ R \Longrightarrow (\bigwedge x \ y. x \in A \Longrightarrow y \in A \Longrightarrow R \ x \ y \Longrightarrow Q \ x \ y) \Longrightarrow \text{reflp-on } A \ Q$
by (auto intro: reflp-onI dest: reflp-onD)

lemma *reflp-mono*: *reflp* *R* $\Longrightarrow (\bigwedge x \ y. R \ x \ y \Longrightarrow Q \ x \ y) \Longrightarrow \text{reflp } Q$
by (rule reflp-on-mono[of UNIV *R* *Q*]) simp-all

lemma (in preorder) *reflp-on-le*[simp]: *reflp-on* *A* (\leq)
by (simp add: reflp-onI)

lemma (in preorder) *reflp-on-ge*[simp]: *reflp-on* *A* (\geq)
by (simp add: reflp-onI)

19.2.2 Irreflexivity

definition *irrefl-on* :: 'a set \Rightarrow 'a rel \Rightarrow bool **where**
 $\text{irrefl-on } A \ r \longleftrightarrow (\forall a \in A. (a, a) \notin r)$

abbreviation *irrefl* :: 'a rel \Rightarrow bool **where**
 $\text{irrefl} \equiv \text{irrefl-on UNIV}$

definition *irreflp-on* :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool **where**
 $\text{irreflp-on } A \ R \longleftrightarrow (\forall a \in A. \neg R \ a \ a)$

abbreviation *irreflp* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool **where**
 $\text{irreflp} \equiv \text{irreflp-on UNIV}$

lemma *irrefl-def*[no-atp]: *irrefl* *r* $\longleftrightarrow (\forall a. (a, a) \notin r)$
by (simp add: irrefl-on-def)

lemma *irreflp-def*[*no-atp*]: $irreflp\ R \longleftrightarrow (\forall a. \neg R\ a\ a)$
by (*simp add: irreflp-on-def*)

irrefl-def and *irreflp-def* are for backward compatibility.

lemma *irreflp-on-irrefl-on-eq* [*pred-set-conv*]: $irreflp-on\ A\ (\lambda a\ b. (a, b) \in r) \longleftrightarrow irrefl-on\ A\ r$
by (*simp add: irrefl-on-def irreflp-on-def*)

lemmas *irreflp-irrefl-eq* = *irreflp-on-irrefl-on-eq*[*of UNIV*]

lemma *irrefl-onI*: $(\bigwedge a. a \in A \implies (a, a) \notin r) \implies irrefl-on\ A\ r$
by (*simp add: irrefl-on-def*)

lemma *irreflI*[*intro?*]: $(\bigwedge a. (a, a) \notin r) \implies irrefl\ r$
by (*rule irrefl-onI*[*of UNIV, simplified*])

lemma *irreflp-onI*: $(\bigwedge a. a \in A \implies \neg R\ a\ a) \implies irreflp-on\ A\ R$
by (*rule irrefl-onI*[*to-pred*])

lemma *irreflpI*[*intro?*]: $(\bigwedge a. \neg R\ a\ a) \implies irreflp\ R$
by (*rule irreflI*[*to-pred*])

lemma *irrefl-onD*: $irrefl-on\ A\ r \implies a \in A \implies (a, a) \notin r$
by (*simp add: irrefl-on-def*)

lemma *irreflD*: $irrefl\ r \implies (x, x) \notin r$
by (*rule irrefl-onD*[*of UNIV, simplified*])

lemma *irreflp-onD*: $irreflp-on\ A\ R \implies a \in A \implies \neg R\ a\ a$
by (*rule irrefl-onD*[*to-pred*])

lemma *irreflpD*: $irreflp\ R \implies \neg R\ x\ x$
by (*rule irreflD*[*to-pred*])

lemma *irrefl-on-distinct* [*code*]: $irrefl-on\ A\ r \longleftrightarrow (\forall (a, b) \in r. a \in A \longrightarrow b \in A \longrightarrow a \neq b)$
by (*auto simp add: irrefl-on-def*)

lemmas *irrefl-distinct* = *irrefl-on-distinct* — For backward compatibility

lemma *irrefl-on-subset*: $irrefl-on\ A\ r \implies B \subseteq A \implies irrefl-on\ B\ r$
by (*auto simp: irrefl-on-def*)

lemma *irreflp-on-subset*: $irreflp-on\ A\ R \implies B \subseteq A \implies irreflp-on\ B\ R$
by (*auto simp: irreflp-on-def*)

lemma (*in preorder*) *irreflp-on-less*[*simp*]: $irreflp-on\ A\ (<)$
by (*simp add: irreflp-onI*)

lemma (in *preorder*) *irreflp-on-greater*[*simp*]: *irreflp-on* A ($>$)
 by (*simp* add: *irreflp-onI*)

19.2.3 Asymmetry

definition *asym-on* :: 'a set \Rightarrow 'a rel \Rightarrow bool **where**
asym-on A $r \iff (\forall x \in A. \forall y \in A. (x, y) \in r \longrightarrow (y, x) \notin r)$

abbreviation *asym* :: 'a rel \Rightarrow bool **where**
asym \equiv *asym-on* *UNIV*

definition *asym-p-on* :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool **where**
asym-p-on A $R \iff (\forall x \in A. \forall y \in A. R\ x\ y \longrightarrow \neg R\ y\ x)$

abbreviation *asym-p* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool **where**
asym-p \equiv *asym-p-on* *UNIV*

lemma *asym-p-on-asym-on-eq*[*pred-set-conv*]: *asym-p-on* A $(\lambda x\ y. (x, y) \in r) \iff$
asym-on A r
 by (*simp* add: *asym-p-on-def* *asym-on-def*)

lemmas *asym-p-asym-eq* = *asym-p-on-asym-on-eq*[*of UNIV*] — For backward compatibility

lemma *asym-onI*[*intro*]:
 $(\bigwedge x\ y. x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \notin r) \implies$ *asym-on* A r
 by (*simp* add: *asym-on-def*)

lemma *asymI*[*intro*]: $(\bigwedge x\ y. (x, y) \in r \implies (y, x) \notin r) \implies$ *asym* r
 by (*simp* add: *asym-onI*)

lemma *asym-p-onI*[*intro*]:
 $(\bigwedge x\ y. x \in A \implies y \in A \implies R\ x\ y \implies \neg R\ y\ x) \implies$ *asym-p-on* A R
 by (*rule* *asym-onI*[*to-pred*])

lemma *asym-pI*[*intro*]: $(\bigwedge x\ y. R\ x\ y \implies \neg R\ y\ x) \implies$ *asym-p* R
 by (*rule* *asymI*[*to-pred*])

lemma *asym-onD*: *asym-on* A $r \implies x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \notin r$
 by (*simp* add: *asym-on-def*)

lemma *asymD*: *asym* $r \implies (x, y) \in r \implies (y, x) \notin r$
 by (*simp* add: *asym-onD*)

lemma *asym-p-onD*: *asym-p-on* A $R \implies x \in A \implies y \in A \implies R\ x\ y \implies \neg R\ y\ x$
 by (*rule* *asym-onD*[*to-pred*])

lemma *asym-pD*: *asym-p* $R \implies R\ x\ y \implies \neg R\ y\ x$

by (rule *asymD[to-pred]*)

lemma *asym-iff*: $asym\ r \longleftrightarrow (\forall x\ y. (x,y) \in r \longrightarrow (y,x) \notin r)$
by (blast dest: *asymD*)

lemma *asym-on-subset*: $asym\ on\ A\ r \Longrightarrow B \subseteq A \Longrightarrow asym\ on\ B\ r$
by (auto simp: *asym-on-def*)

lemma *asym-on-subset*: $asym\ on\ A\ R \Longrightarrow B \subseteq A \Longrightarrow asym\ on\ B\ R$
by (auto simp: *asym-on-def*)

lemma *irrefl-on-if-asym-on[simp]*: $asym\ on\ A\ r \Longrightarrow irrefl\ on\ A\ r$
by (auto intro: *irrefl-onI* dest: *asym-onD*)

lemma *irreflp-on-if-asym-on[simp]*: $asym\ on\ A\ r \Longrightarrow irreflp\ on\ A\ r$
by (rule *irrefl-on-if-asym-on[to-pred]*)

lemma (in *preorder*) *asym-on-less[simp]*: $asym\ on\ A\ (<)$
by (auto intro: *dual-order.asym*)

lemma (in *preorder*) *asym-on-greater[simp]*: $asym\ on\ A\ (>)$
by (auto intro: *dual-order.asym*)

19.2.4 Symmetry

definition *sym-on* :: 'a set \Rightarrow 'a rel \Rightarrow bool **where**
 $sym\ on\ A\ r \longleftrightarrow (\forall x \in A. \forall y \in A. (x, y) \in r \longrightarrow (y, x) \in r)$

abbreviation *sym* :: 'a rel \Rightarrow bool **where**
 $sym \equiv sym\ on\ UNIV$

definition *symp-on* :: 'a set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool **where**
 $symp\ on\ A\ R \longleftrightarrow (\forall x \in A. \forall y \in A. R\ x\ y \longrightarrow R\ y\ x)$

abbreviation *symp* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool **where**
 $symp \equiv symp\ on\ UNIV$

lemma *sym-def[no-atp]*: $sym\ r \longleftrightarrow (\forall x\ y. (x, y) \in r \longrightarrow (y, x) \in r)$
by (simp add: *sym-on-def*)

lemma *symp-def[no-atp]*: $symp\ R \longleftrightarrow (\forall x\ y. R\ x\ y \longrightarrow R\ y\ x)$
by (simp add: *symp-on-def*)

sym-def and *symp-def* are for backward compatibility.

lemma *symp-on-sym-on-eq[pred-set-conv]*: $symp\ on\ A\ (\lambda x\ y. (x, y) \in r) \longleftrightarrow sym\ on\ A\ r$
by (simp add: *sym-on-def symp-on-def*)

lemmas *symp-sym-eq = symp-on-sym-on-eq[of UNIV]* — For backward compati-

bility

lemma *sym-on-subset*: $\text{sym-on } A \ r \implies B \subseteq A \implies \text{sym-on } B \ r$
by (*auto simp: sym-on-def*)

lemma *symp-on-subset*: $\text{symp-on } A \ R \implies B \subseteq A \implies \text{symp-on } B \ R$
by (*auto simp: symp-on-def*)

lemma *sym-onI*: $(\bigwedge x \ y. x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r) \implies \text{sym-on } A \ r$
by (*simp add: sym-on-def*)

lemma *symI* [*intro?*]: $(\bigwedge x \ y. (x, y) \in r \implies (y, x) \in r) \implies \text{sym } r$
by (*simp add: sym-onI*)

lemma *symp-onI*: $(\bigwedge x \ y. x \in A \implies y \in A \implies R \ x \ y \implies R \ y \ x) \implies \text{symp-on } A \ R$
by (*rule sym-onI[to-pred]*)

lemma *sympI* [*intro?*]: $(\bigwedge x \ y. R \ x \ y \implies R \ y \ x) \implies \text{symp } R$
by (*rule symI[to-pred]*)

lemma *symE*:
assumes *sym r* **and** $(b, a) \in r$
obtains $(a, b) \in r$
using *assms* **by** (*simp add: sym-def*)

lemma *sympE*:
assumes *symp r* **and** $r \ b \ a$
obtains $r \ a \ b$
using *assms* **by** (*rule symE [to-pred]*)

lemma *sym-onD*: $\text{sym-on } A \ r \implies x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r$
by (*simp add: sym-on-def*)

lemma *symD* [*dest?*]: $\text{sym } r \implies (x, y) \in r \implies (y, x) \in r$
by (*simp add: sym-onD*)

lemma *symp-onD*: $\text{symp-on } A \ R \implies x \in A \implies y \in A \implies R \ x \ y \implies R \ y \ x$
by (*rule sym-onD[to-pred]*)

lemma *sympD* [*dest?*]: $\text{symp } R \implies R \ x \ y \implies R \ y \ x$
by (*rule symD[to-pred]*)

lemma *sym-Int*: $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cap s)$
by (*fast intro: symI elim: symE*)

lemma *symp-inf*: $\text{symp } r \implies \text{symp } s \implies \text{symp } (r \sqcap s)$
by (*fact sym-Int [to-pred]*)

lemma *sym-Un*: $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cup s)$
by (*fast intro: symI elim: symE*)

lemma *symp-sup*: $\text{symp } r \implies \text{symp } s \implies \text{symp } (r \sqcup s)$
by (*fact sym-Un [to-pred]*)

lemma *sym-INTER*: $\forall x \in S. \text{sym } (r x) \implies \text{sym } (\bigcap (r \text{ ' } S))$
by (*fast intro: symI elim: symE*)

lemma *symp-INF*: $\forall x \in S. \text{symp } (r x) \implies \text{symp } (\bigcap (r \text{ ' } S))$
by (*fact sym-INTER [to-pred]*)

lemma *sym-UNION*: $\forall x \in S. \text{sym } (r x) \implies \text{sym } (\bigcup (r \text{ ' } S))$
by (*fast intro: symI elim: symE*)

lemma *symp-SUP*: $\forall x \in S. \text{symp } (r x) \implies \text{symp } (\bigcup (r \text{ ' } S))$
by (*fact sym-UNION [to-pred]*)

19.2.5 Antisymmetry

definition *antisym-on* :: $'a \text{ set} \Rightarrow 'a \text{ rel} \Rightarrow \text{bool}$ **where**
antisym-on $A r \longleftrightarrow (\forall x \in A. \forall y \in A. (x, y) \in r \longrightarrow (y, x) \in r \longrightarrow x = y)$

abbreviation *antisym* :: $'a \text{ rel} \Rightarrow \text{bool}$ **where**
antisym $\equiv \text{antisym-on UNIV}$

definition *antisymp-on* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
antisymp-on $A R \longleftrightarrow (\forall x \in A. \forall y \in A. R x y \longrightarrow R y x \longrightarrow x = y)$

abbreviation *antisymp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
antisymp $\equiv \text{antisymp-on UNIV}$

lemma *antisym-def[no-atp]*: $\text{antisym } r \longleftrightarrow (\forall x y. (x, y) \in r \longrightarrow (y, x) \in r \longrightarrow x = y)$
by (*simp add: antisym-on-def*)

lemma *antisymp-def[no-atp]*: $\text{antisymp } R \longleftrightarrow (\forall x y. R x y \longrightarrow R y x \longrightarrow x = y)$
by (*simp add: antisymp-on-def*)

antisym-def and *antisymp-def* are for backward compatibility.

lemma *antisymp-on-antisym-on-eq[pred-set-conv]*:
antisymp-on $A (\lambda x y. (x, y) \in r) \longleftrightarrow \text{antisym-on } A r$
by (*simp add: antisym-on-def antisymp-on-def*)

lemmas *antisymp-antisym-eq* = *antisymp-on-antisym-on-eq[of UNIV]* — For backward compatibility

lemma antisym-on-subset: $\text{antisym-on } A \ r \implies B \subseteq A \implies \text{antisym-on } B \ r$
by (*auto simp: antisym-on-def*)

lemma antisym-on-subset: $\text{antisym-on } A \ R \implies B \subseteq A \implies \text{antisym-on } B \ R$
by (*auto simp: antisym-on-def*)

lemma antisym-onI:
 $(\bigwedge x \ y. x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r \implies x = y) \implies \text{antisym-on } A \ r$
unfolding *antisym-on-def* **by** *simp*

lemma antisymI [*intro?*]:
 $(\bigwedge x \ y. (x, y) \in r \implies (y, x) \in r \implies x = y) \implies \text{antisym } r$
by (*simp add: antisym-onI*)

lemma antisym-onI:
 $(\bigwedge x \ y. x \in A \implies y \in A \implies R \ x \ y \implies R \ y \ x \implies x = y) \implies \text{antisym-on } A \ R$
by (*rule antisym-onI[to-pred]*)

lemma antisymI [*intro?*]:
 $(\bigwedge x \ y. R \ x \ y \implies R \ y \ x \implies x = y) \implies \text{antisym } R$
by (*rule antisymI[to-pred]*)

lemma antisym-onD:
 $\text{antisym-on } A \ r \implies x \in A \implies y \in A \implies (x, y) \in r \implies (y, x) \in r \implies x = y$
by (*simp add: antisym-on-def*)

lemma antisymD [*dest?*]:
 $\text{antisym } r \implies (x, y) \in r \implies (y, x) \in r \implies x = y$
by (*simp add: antisym-onD*)

lemma antisym-onD:
 $\text{antisym-on } A \ R \implies x \in A \implies y \in A \implies R \ x \ y \implies R \ y \ x \implies x = y$
by (*rule antisym-onD[to-pred]*)

lemma antisymD [*dest?*]:
 $\text{antisym } R \implies R \ x \ y \implies R \ y \ x \implies x = y$
by (*rule antisymD[to-pred]*)

lemma antisym-subset:
 $r \subseteq s \implies \text{antisym } s \implies \text{antisym } r$
unfolding *antisym-def* **by** *blast*

lemma antisym-less-eq:
 $r \leq s \implies \text{antisym } s \implies \text{antisym } r$
by (*fact antisym-subset [to-pred]*)

lemma antisym-empty [*simp*]:
 $\text{antisym } \{\}$

unfolding *antisym-def* **by** *blast*

lemma *antisym-bot* [*simp*]:
antisym \perp
by (*fact antisym-empty* [*to-pred*])

lemma *antisym-equality* [*simp*]:
antisym *HOL.eq*
by (*auto intro: antisymI*)

lemma *antisym-singleton* [*simp*]:
antisym $\{x\}$
by (*blast intro: antisymI*)

lemma *antisym-on-if-asym-on*: *asym-on* A $r \implies$ *antisym-on* A r
by (*auto intro: antisym-onI dest: asym-onD*)

lemma *antisym-on-if-asymp-on*: *asymp-on* A $R \implies$ *antisym-on* A R
by (*rule antisym-on-if-asymp-on*[*to-pred*])

lemma (**in** *preorder*) *antisym-on-less*[*simp*]: *antisym-on* A $(<)$
by (*rule antisym-on-if-asymp-on*[*OF asymp-on-less*])

lemma (**in** *preorder*) *antisym-on-greater*[*simp*]: *antisym-on* A $(>)$
by (*rule antisym-on-if-asymp-on*[*OF asymp-on-greater*])

lemma (**in** *order*) *antisym-on-le*[*simp*]: *antisym-on* A (\leq)
by (*simp add: antisym-onI*)

lemma (**in** *order*) *antisym-on-ge*[*simp*]: *antisym-on* A (\geq)
by (*simp add: antisym-onI*)

19.2.6 Transitivity

definition *trans-on* :: $'a$ *set* \implies $'a$ *rel* \implies *bool* **where**
trans-on A $r \iff (\forall x \in A. \forall y \in A. \forall z \in A. (x, y) \in r \longrightarrow (y, z) \in r \longrightarrow (x, z) \in r)$

abbreviation *trans* :: $'a$ *rel* \implies *bool* **where**
trans \equiv *trans-on* *UNIV*

definition *transp-on* :: $'a$ *set* \implies $('a \implies 'a \implies \text{bool}) \implies$ *bool* **where**
transp-on A $R \iff (\forall x \in A. \forall y \in A. \forall z \in A. R\ x\ y \longrightarrow R\ y\ z \longrightarrow R\ x\ z)$

abbreviation *transp* :: $('a \implies 'a \implies \text{bool}) \implies$ *bool* **where**
transp \equiv *transp-on* *UNIV*

lemma *trans-def*[*no-atp*]: *trans* $r \iff (\forall x\ y\ z. (x, y) \in r \longrightarrow (y, z) \in r \longrightarrow (x, z) \in r)$

by (simp add: trans-on-def)

lemma *transp-def*: $\text{transp } R \longleftrightarrow (\forall x y z. R x y \longrightarrow R y z \longrightarrow R x z)$
 by (simp add: transp-on-def)

trans-def and *transp-def* are for backward compatibility.

lemma *transp-on-trans-on-eq[pred-set-conv]*: $\text{transp-on } A (\lambda x y. (x, y) \in r) \longleftrightarrow \text{trans-on } A r$
 by (simp add: trans-on-def transp-on-def)

lemmas *transp-trans-eq = transp-on-trans-on-eq[of UNIV]* — For backward compatibility

lemma *trans-onI*:

$(\bigwedge x y z. x \in A \Longrightarrow y \in A \Longrightarrow z \in A \Longrightarrow (x, y) \in r \Longrightarrow (y, z) \in r \Longrightarrow (x, z) \in r) \Longrightarrow$
 $\text{trans-on } A r$
unfolding *trans-on-def*
 by (intro ballI) iprover

lemma *transI [intro?]*: $(\bigwedge x y z. (x, y) \in r \Longrightarrow (y, z) \in r \Longrightarrow (x, z) \in r) \Longrightarrow \text{trans } r$
 by (rule trans-onI)

lemma *transp-onI*:

$(\bigwedge x y z. x \in A \Longrightarrow y \in A \Longrightarrow z \in A \Longrightarrow R x y \Longrightarrow R y z \Longrightarrow R x z) \Longrightarrow \text{transp-on } A R$
 by (rule trans-onI[to-pred])

lemma *transpI [intro?]*: $(\bigwedge x y z. R x y \Longrightarrow R y z \Longrightarrow R x z) \Longrightarrow \text{transp } R$
 by (rule transI[to-pred])

lemma *transE*:

assumes *trans r and* $(x, y) \in r$ **and** $(y, z) \in r$
obtains $(x, z) \in r$
using *assms by* (unfold trans-def) iprover

lemma *transpE*:

assumes *transp r and* $r x y$ **and** $r y z$
obtains $r x z$
using *assms by* (rule transE [to-pred])

lemma *trans-onD*:

$\text{trans-on } A r \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow z \in A \Longrightarrow (x, y) \in r \Longrightarrow (y, z) \in r \Longrightarrow (x, z) \in r$
unfolding *trans-on-def*
 by (elim ballE) iprover+

lemma *transD[dest?]*: $\text{trans } r \Longrightarrow (x, y) \in r \Longrightarrow (y, z) \in r \Longrightarrow (x, z) \in r$

by (*simp add: trans-onD*[of UNIV $r\ x\ y\ z$])

lemma *transp-onD*: $\text{transp-on } A\ R \implies x \in A \implies y \in A \implies z \in A \implies R\ x\ y \implies R\ y\ z \implies R\ x\ z$
by (*rule trans-onD*[to-pred])

lemma *transpD*[*dest?*]: $\text{transp } R \implies R\ x\ y \implies R\ y\ z \implies R\ x\ z$
by (*rule transD*[to-pred])

lemma *trans-on-subset*: $\text{trans-on } A\ r \implies B \subseteq A \implies \text{trans-on } B\ r$
by (*auto simp: trans-on-def*)

lemma *transp-on-subset*: $\text{transp-on } A\ R \implies B \subseteq A \implies \text{transp-on } B\ R$
by (*auto simp: transp-on-def*)

lemma *trans-Int*: $\text{trans } r \implies \text{trans } s \implies \text{trans } (r \cap s)$
by (*fast intro: transI elim: transE*)

lemma *transp-inf*: $\text{transp } r \implies \text{transp } s \implies \text{transp } (r \sqcap s)$
by (*fact trans-Int* [to-pred])

lemma *trans-INTER*: $\forall x \in S. \text{trans } (r\ x) \implies \text{trans } (\bigcap (r\ 'S))$
by (*fast intro: transI elim: transD*)

lemma *transp-INF*: $\forall x \in S. \text{transp } (r\ x) \implies \text{transp } (\bigcap (r\ 'S))$
by (*fact trans-INTER* [to-pred])

lemma *trans-on-join* [*code*]:
 $\text{trans-on } A\ r \iff (\forall (x, y1) \in r. x \in A \longrightarrow y1 \in A \longrightarrow$
 $(\forall (y2, z) \in r. y1 = y2 \longrightarrow z \in A \longrightarrow (x, z) \in r))$
by (*auto simp: trans-on-def*)

lemma *trans-join*: $\text{trans } r \iff (\forall (x, y1) \in r. \forall (y2, z) \in r. y1 = y2 \longrightarrow (x, z) \in r)$
by (*auto simp add: trans-def*)

lemma *transp-trans*: $\text{transp } r \iff \text{trans } \{(x, y). r\ x\ y\}$
by (*simp add: trans-def transp-def*)

lemma *transp-equality* [*simp*]: $\text{transp } (=)$
by (*auto intro: transpI*)

lemma *trans-empty* [*simp*]: $\text{trans } \{\}$
by (*blast intro: transI*)

lemma *transp-empty* [*simp*]: $\text{transp } (\lambda x\ y. \text{False})$
using *trans-empty*[to-pred] **by** (*simp add: bot-fun-def*)

lemma *trans-singleton* [*simp*]: $\text{trans } \{(a, a)\}$

by (*blast intro: transI*)

lemma *transp-singleton* [*simp*]: $\text{transp } (\lambda x y. x = a \wedge y = a)$
by (*simp add: transp-def*)

lemma *asym-on-iff-irrefl-on-if-trans-on*: $\text{trans-on } A r \implies \text{asym-on } A r \longleftrightarrow \text{irrefl-on } A r$
by (*auto intro: irrefl-on-if-asym-on dest: trans-onD irrefl-onD*)

lemma *asympt-on-iff-irreflp-on-if-transp-on*: $\text{transp-on } A R \implies \text{asympt-on } A R \longleftrightarrow \text{irreflp-on } A R$
by (*rule asym-on-iff-irreflp-on-if-trans-on[to-pred]*)

lemma (*in preorder*) *transp-on-le* [*simp*]: $\text{transp-on } A (\leq)$
by (*auto intro: transp-onI order-trans*)

lemma (*in preorder*) *transp-on-less* [*simp*]: $\text{transp-on } A (<)$
by (*auto intro: transp-onI less-trans*)

lemma (*in preorder*) *transp-on-ge* [*simp*]: $\text{transp-on } A (\geq)$
by (*auto intro: transp-onI order-trans*)

lemma (*in preorder*) *transp-on-greater* [*simp*]: $\text{transp-on } A (>)$
by (*auto intro: transp-onI less-trans*)

19.2.7 Totality

definition *total-on* :: 'a set \implies 'a rel \implies bool **where**
 $\text{total-on } A r \longleftrightarrow (\forall x \in A. \forall y \in A. x \neq y \longrightarrow (x, y) \in r \vee (y, x) \in r)$

abbreviation *total* :: 'a rel \implies bool **where**
 $\text{total} \equiv \text{total-on UNIV}$

definition *totalp-on* :: 'a set \implies ('a \implies 'a \implies bool) \implies bool **where**
 $\text{totalp-on } A R \longleftrightarrow (\forall x \in A. \forall y \in A. x \neq y \longrightarrow R x y \vee R y x)$

abbreviation *totalp* :: ('a \implies 'a \implies bool) \implies bool **where**
 $\text{totalp} \equiv \text{totalp-on UNIV}$

lemma *totalp-on-total-on-eq* [*pred-set-conv*]: $\text{totalp-on } A (\lambda x y. (x, y) \in r) \longleftrightarrow \text{total-on } A r$
by (*simp add: totalp-on-def total-on-def*)

lemma *total-onI* [*intro?*]:
 $(\bigwedge x y. x \in A \implies y \in A \implies x \neq y \implies (x, y) \in r \vee (y, x) \in r) \implies \text{total-on } A r$
unfolding *total-on-def* **by** *blast*

lemma *totalI*: $(\bigwedge x y. x \neq y \implies (x, y) \in r \vee (y, x) \in r) \implies \text{total } r$
by (*rule total-onI*)

lemma *totalp-onI*: $(\bigwedge x y. x \in A \implies y \in A \implies x \neq y \implies R x y \vee R y x) \implies$
totalp-on A R

by (*rule total-onI[to-pred]*)

lemma *totalpI*: $(\bigwedge x y. x \neq y \implies R x y \vee R y x) \implies$ *totalp R*

by (*rule totalI[to-pred]*)

lemma *totalp-onD*:

totalp-on A R $\implies x \in A \implies y \in A \implies x \neq y \implies R x y \vee R y x$

by (*simp add: totalp-on-def*)

lemma *totalpD*: *totalp R* $\implies x \neq y \implies R x y \vee R y x$

by (*simp add: totalp-onD*)

lemma *total-on-subset*: *total-on A r* $\implies B \subseteq A \implies$ *total-on B r*

by (*auto simp: total-on-def*)

lemma *totalp-on-subset*: *totalp-on A R* $\implies B \subseteq A \implies$ *totalp-on B R*

by (*auto intro: totalp-onI dest: totalp-onD*)

lemma *total-on-empty [simp]*: *total-on {} r*

by (*simp add: total-on-def*)

lemma *totalp-on-empty [simp]*: *totalp-on {} R*

by (*simp add: totalp-on-def*)

lemma *total-on-singleton [simp]*: *total-on {x} r*

by (*simp add: total-on-def*)

lemma *totalp-on-singleton [simp]*: *totalp-on {x} R*

by (*simp add: totalp-on-def*)

lemma (**in** *linorder*) *totalp-on-less[simp]*: *totalp-on A (<)*

by (*auto intro: totalp-onI*)

lemma (**in** *linorder*) *totalp-on-greater[simp]*: *totalp-on A (>)*

by (*auto intro: totalp-onI*)

lemma (**in** *linorder*) *totalp-on-le[simp]*: *totalp-on A (\leq)*

by (*rule totalp-onI, rule linear*)

lemma (**in** *linorder*) *totalp-on-ge[simp]*: *totalp-on A (\geq)*

by (*rule totalp-onI, rule linear*)

19.2.8 Single valued relations

definition *single-valued* :: $('a \times 'b)$ *set* \implies *bool*

where *single-valued* $r \iff (\forall x y. (x, y) \in r \longrightarrow (\forall z. (x, z) \in r \longrightarrow y = z))$

definition *single-valuedp* :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where *single-valuedp* r ⇔ (∀ x y. r x y ⟶ (∀ z. r x z ⟶ y = z))

lemma *single-valuedp-single-valued-eq* [*pred-set-conv*]:
single-valuedp (λx y. (x, y) ∈ r) ⇔ *single-valued* r
by (*simp add: single-valued-def single-valuedp-def*)

lemma *single-valuedp-iff-Uniq*:
single-valuedp r ⇔ (∀ x. ∃ ≤₁ y. r x y)
unfolding *Uniq-def single-valuedp-def* **by** *auto*

lemma *single-valuedI*:
(∧ x y. (x, y) ∈ r ⟶ (∧ z. (x, z) ∈ r ⟶ y = z)) ⟶ *single-valued* r
unfolding *single-valued-def* **by** *blast*

lemma *single-valuedpI*:
(∧ x y. r x y ⟶ (∧ z. r x z ⟶ y = z)) ⟶ *single-valuedp* r
by (*fact single-valuedI [to-pred]*)

lemma *single-valuedD*:
single-valued r ⟶ (x, y) ∈ r ⟶ (x, z) ∈ r ⟶ y = z
by (*simp add: single-valued-def*)

lemma *single-valuedpD*:
single-valuedp r ⟶ r x y ⟶ r x z ⟶ y = z
by (*fact single-valuedD [to-pred]*)

lemma *single-valued-empty* [*simp*]:
single-valued {}
by (*simp add: single-valued-def*)

lemma *single-valuedp-bot* [*simp*]:
single-valuedp ⊥
by (*fact single-valued-empty [to-pred]*)

lemma *single-valued-subset*:
r ⊆ s ⟶ *single-valued* s ⟶ *single-valued* r
unfolding *single-valued-def* **by** *blast*

lemma *single-valuedp-less-eq*:
r ≤ s ⟶ *single-valuedp* s ⟶ *single-valuedp* r
by (*fact single-valued-subset [to-pred]*)

19.3 Relation operations

19.3.1 The identity relation

definition *Id* :: 'a rel
where *Id* = {p. ∃ x. p = (x, x)}

lemma *IdI* [*intro*]: $(a, a) \in Id$
by (*simp add: Id-def*)

lemma *IdE* [*elim!*]: $p \in Id \implies (\bigwedge x. p = (x, x) \implies P) \implies P$
unfolding *Id-def* **by** (*iprover elim: CollectE*)

lemma *pair-in-Id-conv* [*iff*]: $(a, b) \in Id \iff a = b$
unfolding *Id-def* **by** *blast*

lemma *refl-Id*: *refl Id*
by (*simp add: refl-on-def*)

lemma *antisym-Id*: *antisym Id*
 — A strange result, since *Id* is also symmetric.
by (*simp add: antisym-def*)

lemma *sym-Id*: *sym Id*
by (*simp add: sym-def*)

lemma *trans-Id*: *trans Id*
by (*simp add: trans-def*)

lemma *single-valued-Id* [*simp*]: *single-valued Id*
by (*unfold single-valued-def*) *blast*

lemma *irrefl-diff-Id* [*simp*]: *irrefl (r - Id)*
by (*simp add: irrefl-def*)

lemma *trans-diff-Id*: $trans\ r \implies antisym\ r \implies trans\ (r - Id)$
unfolding *antisym-def trans-def* **by** *blast*

lemma *total-on-diff-Id* [*simp*]: $total-on\ A\ (r - Id) = total-on\ A\ r$
by (*simp add: total-on-def*)

lemma *Id-fstsnd-eq*: $Id = \{x. fst\ x = snd\ x\}$
by *force*

19.3.2 Diagonal: identity over a set

definition *Id-on* :: $'a\ set \Rightarrow 'a\ rel$
where $Id-on\ A = (\bigcup x \in A. \{(x, x)\})$

lemma *Id-on-empty* [*simp*]: $Id-on\ \{\} = \{\}$
by (*simp add: Id-on-def*)

lemma *Id-on-eqI*: $a = b \implies a \in A \implies (a, b) \in Id-on\ A$
by (*simp add: Id-on-def*)

lemma *Id-onI* [*intro!*]: $a \in A \implies (a, a) \in \text{Id-on } A$
by (*rule Id-on-eqI*) (*rule refl*)

lemma *Id-onE* [*elim!*]: $c \in \text{Id-on } A \implies (\bigwedge x. x \in A \implies c = (x, x) \implies P) \implies P$
 — The general elimination rule.
unfolding *Id-on-def* **by** (*iprover elim!*: *UN-E singletonE*)

lemma *Id-on-iff*: $(x, y) \in \text{Id-on } A \iff x = y \wedge x \in A$
by *blast*

lemma *Id-on-def'* [*nitpick-unfold*]: $\text{Id-on } \{x. A \ x\} = \text{Collect } (\lambda(x, y). x = y \wedge A \ x)$
by *auto*

lemma *Id-on-subset-Times*: $\text{Id-on } A \subseteq A \times A$
by *blast*

lemma *refl-on-Id-on*: *refl-on* A (*Id-on* A)
by (*rule refl-onI* [*OF Id-on-subset-Times Id-onI*])

lemma *antisym-Id-on* [*simp*]: *antisym* (*Id-on* A)
unfolding *antisym-def* **by** *blast*

lemma *sym-Id-on* [*simp*]: *sym* (*Id-on* A)
by (*rule symI*) *clarify*

lemma *trans-Id-on* [*simp*]: *trans* (*Id-on* A)
by (*fast intro: transI elim: transD*)

lemma *single-valued-Id-on* [*simp*]: *single-valued* (*Id-on* A)
unfolding *single-valued-def* **by** *blast*

19.3.3 Composition

inductive-set *relcomp* :: $('a \times 'b) \text{ set} \Rightarrow ('b \times 'c) \text{ set} \Rightarrow ('a \times 'c) \text{ set}$ (**infixr** *O* 75)

for $r :: ('a \times 'b) \text{ set}$ **and** $s :: ('b \times 'c) \text{ set}$
where *relcompI* [*intro!*]: $(a, b) \in r \implies (b, c) \in s \implies (a, c) \in r \ O \ s$

notation *relcompp* (**infixr** *OO* 75)

lemmas *relcomppI* = *relcompp.intros*

For historic reasons, the elimination rules are not wholly corresponding. Feel free to consolidate this.

inductive-cases *relcompEpair*: $(a, c) \in r \ O \ s$
inductive-cases *relcomppE* [*elim!*]: $(r \ OO \ s) \ a \ c$

lemma *relcompE* [*elim!*]: $xz \in r \ O \ s \implies$

```

( $\wedge x y z. xz = (x, z) \implies (x, y) \in r \implies (y, z) \in s \implies P$ )  $\implies P$ 
apply (cases  $xz$ )
apply simp
apply (erule relcompEpair)
apply iprover
done

```

lemma *R-O-Id* [simp]: $R \ O \ Id = R$
by fast

lemma *Id-O-R* [simp]: $Id \ O \ R = R$
by fast

lemma *relcomp-empty1* [simp]: $\{\} \ O \ R = \{\}$
by blast

lemma *relcompp-bot1* [simp]: $\perp \ OO \ R = \perp$
by (fact *relcomp-empty1* [to-pred])

lemma *relcomp-empty2* [simp]: $R \ O \ \{\} = \{\}$
by blast

lemma *relcompp-bot2* [simp]: $R \ OO \ \perp = \perp$
by (fact *relcomp-empty2* [to-pred])

lemma *O-assoc*: $(R \ O \ S) \ O \ T = R \ O \ (S \ O \ T)$
by blast

lemma *relcompp-assoc*: $(r \ OO \ s) \ OO \ t = r \ OO \ (s \ OO \ t)$
by (fact *O-assoc* [to-pred])

lemma *trans-O-subset*: $\text{trans } r \implies r \ O \ r \subseteq r$
by (unfold *trans-def*) blast

lemma *transp-relcompp-less-eq*: $\text{transp } r \implies r \ OO \ r \leq r$
by (fact *trans-O-subset* [to-pred])

lemma *relcomp-mono*: $r' \subseteq r \implies s' \subseteq s \implies r' \ O \ s' \subseteq r \ O \ s$
by blast

lemma *relcompp-mono*: $r' \leq r \implies s' \leq s \implies r' \ OO \ s' \leq r \ OO \ s$
by (fact *relcomp-mono* [to-pred])

lemma *relcomp-subset-Sigma*: $r \subseteq A \times B \implies s \subseteq B \times C \implies r \ O \ s \subseteq A \times C$
by blast

lemma *relcomp-distrib* [simp]: $R \ O \ (S \cup T) = (R \ O \ S) \cup (R \ O \ T)$
by auto

lemma *relcompp-distrib* [*simp*]: $R \circ O (S \sqcup T) = R \circ O S \sqcup R \circ O T$
by (*fact relcompp-distrib* [*to-pred*])

lemma *relcompp-distrib2* [*simp*]: $(S \cup T) \circ R = (S \circ R) \cup (T \circ R)$
by *auto*

lemma *relcompp-distrib2* [*simp*]: $(S \sqcup T) \circ O R = S \circ O R \sqcup T \circ O R$
by (*fact relcompp-distrib2* [*to-pred*])

lemma *relcompp-UNION-distrib*: $s \circ \bigcup (r \text{ ' } I) = \bigcup_{i \in I}. s \circ r \ i$
by *auto*

lemma *relcompp-SUP-distrib*: $s \circ O \bigsqcup (r \text{ ' } I) = \bigsqcup_{i \in I}. s \circ O r \ i$
by (*fact relcompp-UNION-distrib* [*to-pred*])

lemma *relcompp-UNION-distrib2*: $\bigcup (r \text{ ' } I) \circ s = \bigcup_{i \in I}. r \ i \circ s$
by *auto*

lemma *relcompp-SUP-distrib2*: $\bigsqcup (r \text{ ' } I) \circ O s = \bigsqcup_{i \in I}. r \ i \circ O s$
by (*fact relcompp-UNION-distrib2* [*to-pred*])

lemma *single-valued-relcompp*: *single-valued* $r \implies$ *single-valued* $s \implies$ *single-valued* $(r \circ s)$
unfolding *single-valued-def* **by** *blast*

lemma *relcompp-unfold*: $r \circ s = \{(x, z). \exists y. (x, y) \in r \wedge (y, z) \in s\}$
by (*auto simp add: set-eq-iff*)

lemma *relcompp-apply*: $(R \circ O S) \ a \ c \longleftrightarrow (\exists b. R \ a \ b \wedge S \ b \ c)$
unfolding *relcompp-unfold* [*to-pred*] **..**

lemma *eq-OO*: $(=) \circ O R = R$
by *blast*

lemma *OO-eq*: $R \circ O (=) = R$
by *blast*

19.3.4 Converse

inductive-set *converse* :: $('a \times 'b) \text{ set} \Rightarrow ('b \times 'a) \text{ set}$ ($(-^{-1})$ [1000] 999)
for $r :: ('a \times 'b) \text{ set}$
where $(a, b) \in r \implies (b, a) \in r^{-1}$

notation *conversep* ($(-^{-1-1})$ [1000] 1000)

notation (*ASCII*)
converse ($(\widehat{-}^{-1})$ [1000] 999) **and**
conversep ($(\widehat{-}^{-1-1})$ [1000] 1000)

lemma *converseI* [*sym*]: $(a, b) \in r \implies (b, a) \in r^{-1}$
by (*fact converse.intros*)

lemma *converseI* : $r\ a\ b \implies r^{-1-1}\ b\ a$
by (*fact converse.intros*)

lemma *converseD* [*sym*]: $(a, b) \in r^{-1} \implies (b, a) \in r$
by (*erule converse.cases*) *iprover*

lemma *converseD* : $r^{-1-1}\ b\ a \implies r\ a\ b$
by (*fact converseD [to-pred]*)

lemma *converseE* [*elim!*]: $yx \in r^{-1} \implies (\bigwedge x\ y. yx = (y, x) \implies (x, y) \in r \implies P) \implies P$
— More general than *converseD*, as it “splits” the member of the relation.
apply (*cases yx*)
apply *simp*
apply (*erule converse.cases*)
apply *iprover*
done

lemmas *converseE* [*elim!*] = *converse.cases*

lemma *converse-iff* [*iff*]: $(a, b) \in r^{-1} \longleftrightarrow (b, a) \in r$
by (*auto intro: converseI*)

lemma *converse-iff* [*iff*]: $r^{-1-1}\ a\ b = r\ b\ a$
by (*fact converse-iff [to-pred]*)

lemma *converse-converse* [*simp*]: $(r^{-1})^{-1} = r$
by (*simp add: set-eq-iff*)

lemma *converse-converse* [*simp*]: $(r^{-1-1})^{-1-1} = r$
by (*fact converse-converse [to-pred]*)

lemma *converse-empty*[*simp*]: $\{\}^{-1} = \{\}$
by *auto*

lemma *converse-UNIV*[*simp*]: $UNIV^{-1} = UNIV$
by *auto*

lemma *converse-relcomp*: $(r\ O\ s)^{-1} = s^{-1}\ O\ r^{-1}$
by *blast*

lemma *converse-relcomp*: $(r\ OO\ s)^{-1-1} = s^{-1-1}\ OO\ r^{-1-1}$
by (*iprover intro: order-antisym converseI relcomppI elim: relcomppE dest: converseD*)

lemma *converse-Int*: $(r \cap s)^{-1} = r^{-1} \cap s^{-1}$

by *blast*

lemma *converse-meet*: $(r \sqcap s)^{-1-1} = r^{-1-1} \sqcap s^{-1-1}$
 by (*simp add: inf-fun-def*) (*iprover intro: converseSepI ext dest: converseSepD*)

lemma *converse-Un*: $(r \cup s)^{-1} = r^{-1} \cup s^{-1}$
 by *blast*

lemma *converse-join*: $(r \sqcup s)^{-1-1} = r^{-1-1} \sqcup s^{-1-1}$
 by (*simp add: sup-fun-def*) (*iprover intro: converseSepI ext dest: converseSepD*)

lemma *converse-INTER*: $(\bigcap (r \text{ ‘ } S))^{-1} = (\bigcap x \in S. (r \ x)^{-1})$
 by *fast*

lemma *converse-UNION*: $(\bigcup (r \text{ ‘ } S))^{-1} = (\bigcup x \in S. (r \ x)^{-1})$
 by *blast*

lemma *converse-mono[simp]*: $r^{-1} \subseteq s^{-1} \longleftrightarrow r \subseteq s$
 by *auto*

lemma *converseSep-mono[simp]*: $r^{-1-1} \leq s^{-1-1} \longleftrightarrow r \leq s$
 by (*fact converse-mono[to-pred]*)

lemma *converse-inject[simp]*: $r^{-1} = s^{-1} \longleftrightarrow r = s$
 by *auto*

lemma *converseSep-inject[simp]*: $r^{-1-1} = s^{-1-1} \longleftrightarrow r = s$
 by (*fact converse-inject[to-pred]*)

lemma *converse-subset-swap*: $r \subseteq s^{-1} \longleftrightarrow r^{-1} \subseteq s$
 by *auto*

lemma *converseSep-le-swap*: $r \leq s^{-1-1} \longleftrightarrow r^{-1-1} \leq s$
 by (*fact converse-subset-swap[to-pred]*)

lemma *converse-Id [simp]*: $Id^{-1} = Id$
 by *blast*

lemma *converse-Id-on [simp]*: $(Id \text{-on } A)^{-1} = Id \text{-on } A$
 by *blast*

lemma *refl-on-converse [simp]*: $refl \text{-on } A \ (r^{-1}) = refl \text{-on } A \ r$
 by (*auto simp: refl-on-def*)

lemma *reflp-on-conversp [simp]*: $reflp \text{-on } A \ R^{-1-1} \longleftrightarrow reflp \text{-on } A \ R$
 by (*auto simp: reflp-on-def*)

lemma *irrefl-on-converse [simp]*: $irrefl \text{-on } A \ (r^{-1}) = irrefl \text{-on } A \ r$
 by (*simp add: irrefl-on-def*)

lemma *irreflp-on-converse* [*simp*]: $\text{irreflp-on } A (r^{-1-1}) = \text{irreflp-on } A r$
by (*rule irreflp-on-converse[to-pred]*)

lemma *sym-on-converse* [*simp*]: $\text{sym-on } A (r^{-1}) = \text{sym-on } A r$
by (*auto intro: sym-onI dest: sym-onD*)

lemma *symp-on-conversep* [*simp*]: $\text{symp-on } A R^{-1-1} = \text{symp-on } A R$
by (*rule sym-on-converse[to-pred]*)

lemma *asym-on-converse* [*simp*]: $\text{asym-on } A (r^{-1}) = \text{asym-on } A r$
by (*auto dest: asym-onD*)

lemma *asymp-on-conversep* [*simp*]: $\text{asymp-on } A R^{-1-1} = \text{asymp-on } A R$
by (*rule asym-on-converse[to-pred]*)

lemma *antisym-on-converse* [*simp*]: $\text{antisym-on } A (r^{-1}) = \text{antisym-on } A r$
by (*auto intro: antisym-onI dest: antisym-onD*)

lemma *antisymp-on-conversep* [*simp*]: $\text{antisymp-on } A R^{-1-1} = \text{antisymp-on } A R$
by (*rule antisym-on-converse[to-pred]*)

lemma *trans-on-converse* [*simp*]: $\text{trans-on } A (r^{-1}) = \text{trans-on } A r$
by (*auto intro: trans-onI dest: trans-onD*)

lemma *transp-on-conversep* [*simp*]: $\text{transp-on } A R^{-1-1} = \text{transp-on } A R$
by (*rule trans-on-converse[to-pred]*)

lemma *sym-conv-converse-eq*: $\text{sym } r \longleftrightarrow r^{-1} = r$
unfolding *sym-def* **by** *fast*

lemma *sym-Un-converse*: $\text{sym } (r \cup r^{-1})$
unfolding *sym-def* **by** *blast*

lemma *sym-Int-converse*: $\text{sym } (r \cap r^{-1})$
unfolding *sym-def* **by** *blast*

lemma *total-on-converse* [*simp*]: $\text{total-on } A (r^{-1}) = \text{total-on } A r$
by (*auto simp: total-on-def*)

lemma *totalp-on-converse* [*simp*]: $\text{totalp-on } A R^{-1-1} = \text{totalp-on } A R$
by (*rule total-on-converse[to-pred]*)

lemma *conversep-noteq* [*simp*]: $(\neq)^{-1-1} = (\neq)$
by (*auto simp add: fun-eq-iff*)

lemma *conversep-eq* [*simp*]: $(=)^{-1-1} = (=)$
by (*auto simp add: fun-eq-iff*)

lemma *converse-unfold* [code]: $r^{-1} = \{(y, x). (x, y) \in r\}$
 by (*simp add: set-eq-iff*)

19.3.5 Domain, range and field

inductive-set *Domain* :: ('a × 'b) set ⇒ 'a set **for** r :: ('a × 'b) set
 where *DomainI* [intro]: $(a, b) \in r \implies a \in \text{Domain } r$

lemmas *DomainPI* = *Domainp.DomainI*

inductive-cases *DomainE* [elim!]: $a \in \text{Domain } r$
inductive-cases *DomainpE* [elim!]: $\text{Domainp } r a$

inductive-set *Range* :: ('a × 'b) set ⇒ 'b set **for** r :: ('a × 'b) set
 where *RangeI* [intro]: $(a, b) \in r \implies b \in \text{Range } r$

lemmas *RangePI* = *Rangep.RangeI*

inductive-cases *RangeE* [elim!]: $b \in \text{Range } r$
inductive-cases *RangepE* [elim!]: $\text{Rangep } r b$

definition *Field* :: 'a rel ⇒ 'a set
 where *Field* r = $\text{Domain } r \cup \text{Range } r$

lemma *Field-iff*: $x \in \text{Field } r \iff (\exists y. (x, y) \in r \vee (y, x) \in r)$
 by (*auto simp: Field-def*)

lemma *FieldI1*: $(i, j) \in R \implies i \in \text{Field } R$
 unfolding *Field-def* by *blast*

lemma *FieldI2*: $(i, j) \in R \implies j \in \text{Field } R$
 unfolding *Field-def* by *auto*

lemma *Domain-fst* [code]: $\text{Domain } r = \text{fst } ' r$
 by *force*

lemma *Range-snd* [code]: $\text{Range } r = \text{snd } ' r$
 by *force*

lemma *fst-eq-Domain*: $\text{fst } ' R = \text{Domain } R$
 by *force*

lemma *snd-eq-Range*: $\text{snd } ' R = \text{Range } R$
 by *force*

lemma *range-fst* [simp]: $\text{range } \text{fst} = \text{UNIV}$
 by (*auto simp: fst-eq-Domain*)

lemma *range-snd* [simp]: $\text{range } \text{snd} = \text{UNIV}$

by (auto simp: snd-eq-Range)

lemma *Domain-empty* [simp]: $\text{Domain } \{\} = \{\}$
by auto

lemma *Range-empty* [simp]: $\text{Range } \{\} = \{\}$
by auto

lemma *Field-empty* [simp]: $\text{Field } \{\} = \{\}$
by (simp add: Field-def)

lemma *Domain-empty-iff*: $\text{Domain } r = \{\} \longleftrightarrow r = \{\}$
by auto

lemma *Range-empty-iff*: $\text{Range } r = \{\} \longleftrightarrow r = \{\}$
by auto

lemma *Domain-insert* [simp]: $\text{Domain } (\text{insert } (a, b) r) = \text{insert } a (\text{Domain } r)$
by blast

lemma *Range-insert* [simp]: $\text{Range } (\text{insert } (a, b) r) = \text{insert } b (\text{Range } r)$
by blast

lemma *Field-insert* [simp]: $\text{Field } (\text{insert } (a, b) r) = \{a, b\} \cup \text{Field } r$
by (auto simp add: Field-def)

lemma *Domain-iff*: $a \in \text{Domain } r \longleftrightarrow (\exists y. (a, y) \in r)$
by blast

lemma *Range-iff*: $a \in \text{Range } r \longleftrightarrow (\exists y. (y, a) \in r)$
by blast

lemma *Domain-Id* [simp]: $\text{Domain } \text{Id} = \text{UNIV}$
by blast

lemma *Range-Id* [simp]: $\text{Range } \text{Id} = \text{UNIV}$
by blast

lemma *Domain-Id-on* [simp]: $\text{Domain } (\text{Id-on } A) = A$
by blast

lemma *Range-Id-on* [simp]: $\text{Range } (\text{Id-on } A) = A$
by blast

lemma *Domain-Un-eq*: $\text{Domain } (A \cup B) = \text{Domain } A \cup \text{Domain } B$
by blast

lemma *Range-Un-eq*: $\text{Range } (A \cup B) = \text{Range } A \cup \text{Range } B$
by blast

lemma *Field-Un* [simp]: $\text{Field } (r \cup s) = \text{Field } r \cup \text{Field } s$
 by (auto simp: Field-def)

lemma *Domain-Int-subset*: $\text{Domain } (A \cap B) \subseteq \text{Domain } A \cap \text{Domain } B$
 by blast

lemma *Range-Int-subset*: $\text{Range } (A \cap B) \subseteq \text{Range } A \cap \text{Range } B$
 by blast

lemma *Domain-Diff-subset*: $\text{Domain } A - \text{Domain } B \subseteq \text{Domain } (A - B)$
 by blast

lemma *Range-Diff-subset*: $\text{Range } A - \text{Range } B \subseteq \text{Range } (A - B)$
 by blast

lemma *Domain-Union*: $\text{Domain } (\bigcup S) = (\bigcup A \in S. \text{Domain } A)$
 by blast

lemma *Range-Union*: $\text{Range } (\bigcup S) = (\bigcup A \in S. \text{Range } A)$
 by blast

lemma *Field-Union* [simp]: $\text{Field } (\bigcup R) = \bigcup (\text{Field } ` R)$
 by (auto simp: Field-def)

lemma *Domain-converse* [simp]: $\text{Domain } (r^{-1}) = \text{Range } r$
 by auto

lemma *Range-converse* [simp]: $\text{Range } (r^{-1}) = \text{Domain } r$
 by blast

lemma *Field-converse* [simp]: $\text{Field } (r^{-1}) = \text{Field } r$
 by (auto simp: Field-def)

lemma *Domain-Collect-case-prod* [simp]: $\text{Domain } \{(x, y). P x y\} = \{x. \exists y. P x y\}$
 by auto

lemma *Range-Collect-case-prod* [simp]: $\text{Range } \{(x, y). P x y\} = \{y. \exists x. P x y\}$
 by auto

lemma *Domain-mono*: $r \subseteq s \implies \text{Domain } r \subseteq \text{Domain } s$
 by blast

lemma *Range-mono*: $r \subseteq s \implies \text{Range } r \subseteq \text{Range } s$
 by blast

lemma *mono-Field*: $r \subseteq s \implies \text{Field } r \subseteq \text{Field } s$
 by (auto simp: Field-def Domain-def Range-def)

lemma *Domain-unfold*: $\text{Domain } r = \{x. \exists y. (x, y) \in r\}$
by *blast*

lemma *Field-square* [*simp*]: $\text{Field } (x \times x) = x$
unfolding *Field-def* **by** *blast*

19.3.6 Image of a set under a relation

definition *Image* :: $('a \times 'b) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set}$ (**infixr** “90”)
where $r \text{ “ } s = \{y. \exists x \in s. (x, y) \in r\}$

lemma *Image-iff*: $b \in r \text{ “ } A \longleftrightarrow (\exists x \in A. (x, b) \in r)$
by (*simp add: Image-def*)

lemma *Image-singleton*: $r \text{ “ } \{a\} = \{b. (a, b) \in r\}$
by (*simp add: Image-def*)

lemma *Image-singleton-iff* [*iff*]: $b \in r \text{ “ } \{a\} \longleftrightarrow (a, b) \in r$
by (*rule Image-iff [THEN trans]*) *simp*

lemma *ImageI* [*intro*]: $(a, b) \in r \Longrightarrow a \in A \Longrightarrow b \in r \text{ “ } A$
unfolding *Image-def* **by** *blast*

lemma *ImageE* [*elim!*]: $b \in r \text{ “ } A \Longrightarrow (\bigwedge x. (x, b) \in r \Longrightarrow x \in A \Longrightarrow P) \Longrightarrow P$
unfolding *Image-def* **by** (*iprover elim!: CollectE bexE*)

lemma *rev-ImageI*: $a \in A \Longrightarrow (a, b) \in r \Longrightarrow b \in r \text{ “ } A$
— This version’s more effective when we already have the required a
by *blast*

lemma *Image-empty1* [*simp*]: $\{\} \text{ “ } X = \{\}$
by *auto*

lemma *Image-empty2* [*simp*]: $R \text{ “ } \{\} = \{\}$
by *blast*

lemma *Image-Id* [*simp*]: $\text{Id} \text{ “ } A = A$
by *blast*

lemma *Image-Id-on* [*simp*]: $\text{Id-on } A \text{ “ } B = A \cap B$
by *blast*

lemma *Image-Int-subset*: $R \text{ “ } (A \cap B) \subseteq R \text{ “ } A \cap R \text{ “ } B$
by *blast*

lemma *Image-Int-eq*: *single-valued* (*converse* R) $\Longrightarrow R \text{ “ } (A \cap B) = R \text{ “ } A \cap R \text{ “ } B$
by (*auto simp: single-valued-def*)

lemma *Image-Un*: $R \text{ “ } (A \cup B) = R \text{ “ } A \cup R \text{ “ } B$
by *blast*

lemma *Un-Image*: $(R \cup S) \text{ “ } A = R \text{ “ } A \cup S \text{ “ } A$
by *blast*

lemma *Image-subset*: $r \subseteq A \times B \implies r \text{ “ } C \subseteq B$
by (*iprover intro!*: *subsetI elim!*: *ImageE dest!*: *subsetD SigmaD2*)

lemma *Image-eq-UN*: $r \text{ “ } B = (\bigcup y \in B. r \text{ “ } \{y\})$
 — NOT suitable for rewriting
by *blast*

lemma *Image-mono*: $r' \subseteq r \implies A' \subseteq A \implies (r' \text{ “ } A') \subseteq (r \text{ “ } A)$
by *blast*

lemma *Image-UN*: $r \text{ “ } (\bigcup (B \text{ ‘ } A)) = (\bigcup x \in A. r \text{ “ } (B \ x))$
by *blast*

lemma *UN-Image*: $(\bigcup i \in I. X \ i) \text{ “ } S = (\bigcup i \in I. X \ i \text{ “ } S)$
by *auto*

lemma *Image-INT-subset*: $(r \text{ “ } (\bigcap (B \text{ ‘ } A))) \subseteq (\bigcap x \in A. r \text{ “ } (B \ x))$
by *blast*

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
assumes *single-valued* (r^{-1})
and $A \neq \{\}$
shows $r \text{ “ } (\bigcap (B \text{ ‘ } A)) = (\bigcap x \in A. r \text{ “ } B \ x)$
proof(*rule equalityI*, *rule Image-INT-subset*)
show $(\bigcap x \in A. r \text{ “ } B \ x) \subseteq r \text{ “ } \bigcap (B \text{ ‘ } A)$
proof
fix x
assume $x \in (\bigcap x \in A. r \text{ “ } B \ x)$
then show $x \in r \text{ “ } \bigcap (B \text{ ‘ } A)$
using *assms unfolding single-valued-def* **by** *simp blast*
qed
qed

lemma *Image-subset-eq*: $r \text{ “ } A \subseteq B \longleftrightarrow A \subseteq - ((r^{-1}) \text{ “ } (- B))$
by *blast*

lemma *Image-Collect-case-prod* [*simp*]: $\{(x, y). P \ x \ y\} \text{ “ } A = \{y. \exists x \in A. P \ x \ y\}$
by *auto*

lemma *Sigma-Image*: $(\text{SIGMA } x:A. B \ x) \text{ “ } X = (\bigcup x \in X \cap A. B \ x)$
by *auto*

lemma *relcomp-Image*: $(X \circ Y) \text{ “ } Z = Y \text{ “ } (X \text{ “ } Z)$
by *auto*

19.3.7 Inverse image

definition *inv-image* :: $'b \text{ rel} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ rel}$
where *inv-image* $r \ f = \{(x, y). (f \ x, f \ y) \in r\}$

definition *inv-imagep* :: $('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
where *inv-imagep* $r \ f = (\lambda x \ y. r \ (f \ x) \ (f \ y))$

lemma [*pred-set-conv*]: *inv-imagep* $(\lambda x \ y. (x, y) \in r) \ f = (\lambda x \ y. (x, y) \in \text{inv-image} \ r \ f)$
by (*simp add: inv-image-def inv-imagep-def*)

lemma *sym-inv-image*: $\text{sym } r \Longrightarrow \text{sym } (\text{inv-image } r \ f)$
unfolding *sym-def inv-image-def* **by** *blast*

lemma *trans-inv-image*: $\text{trans } r \Longrightarrow \text{trans } (\text{inv-image } r \ f)$
unfolding *trans-def inv-image-def*
by (*simp (no-asm)*) *blast*

lemma *total-inv-image*: $\llbracket \text{inj } f; \text{total } r \rrbracket \Longrightarrow \text{total } (\text{inv-image } r \ f)$
unfolding *inv-image-def total-on-def* **by** (*auto simp: inj-eq*)

lemma *asym-inv-image*: $\text{asym } R \Longrightarrow \text{asym } (\text{inv-image } R \ f)$
by (*simp add: inv-image-def asym-iff*)

lemma *in-inv-image[simp]*: $(x, y) \in \text{inv-image } r \ f \longleftrightarrow (f \ x, f \ y) \in r$
by (*auto simp: inv-image-def*)

lemma *converse-inv-image[simp]*: $(\text{inv-image } R \ f)^{-1} = \text{inv-image } (R^{-1}) \ f$
unfolding *inv-image-def converse-unfold* **by** *auto*

lemma *in-inv-imagep [simp]*: *inv-imagep* $r \ f \ x \ y = r \ (f \ x) \ (f \ y)$
by (*simp add: inv-imagep-def*)

19.3.8 Powerset

definition *Powp* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$
where *Powp* $A = (\lambda B. \forall x \in B. A \ x)$

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp* $(\lambda x. x \in A) = (\lambda x. x \in \text{Pow } A)$
by (*auto simp add: Powp-def fun-eq-iff*)

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred*]

end

20 Finite sets

```
theory Finite-Set
  imports Product-Type Sum-Type Fields Relation
begin
```

20.1 Predicate for finite sets

```
context notes [[inductive-internals]]
begin
```

```
inductive finite :: 'a set  $\Rightarrow$  bool
  where
    emptyI [simp, intro!]: finite {}
  | insertI [simp, intro!]: finite A  $\Longrightarrow$  finite (insert a A)
```

```
end
```

```
simproc-setup finite-Collect (finite (Collect P)) = ⟨K Set-Comprehension-Pointfree.simproc⟩
```

```
declare [[simproc del: finite-Collect]]
```

```
lemma finite-induct [case-names empty insert, induct set: finite]:
```

— Discharging $x \notin F$ entails extra work.

```
  assumes finite F
```

```
  assumes P {}
```

```
  and insert:  $\bigwedge x F. \text{finite } F \Longrightarrow x \notin F \Longrightarrow P F \Longrightarrow P (\text{insert } x F)$ 
```

```
  shows P F
```

```
  using ⟨finite F⟩
```

```
proof induct
```

```
  show P {} by fact
```

```
next
```

```
  fix x F
```

```
  assume F: finite F and P: P F
```

```
  show P (insert x F)
```

```
  proof cases
```

```
    assume x  $\in$  F
```

```
    then have insert x F = F by (rule insert-absorb)
```

```
    with P show ?thesis by (simp only:)
```

```
  next
```

```
    assume x  $\notin$  F
```

```
    from F this P show ?thesis by (rule insert)
```

```
  qed
```

```
qed
```

```
lemma infinite-finite-induct [case-names infinite empty insert]:
```

```
  assumes infinite:  $\bigwedge A. \neg \text{finite } A \Longrightarrow P A$ 
```

```
  and empty: P {}
```

```
  and insert:  $\bigwedge x F. \text{finite } F \Longrightarrow x \notin F \Longrightarrow P F \Longrightarrow P (\text{insert } x F)$ 
```

```
  shows P A
```

```

proof (cases finite A)
  case False
  with infinite show ?thesis .
next
  case True
  then show ?thesis by (induct A) (fact empty insert)+
qed

```

20.1.1 Choice principles

lemma *ex-new-if-finite*: — does not depend on def of finite at all

assumes \neg finite (UNIV :: 'a set) **and** finite A

shows $\exists a::'a. a \notin A$

proof —

from *assms* **have** $A \neq \text{UNIV}$ **by** *blast*

then show ?thesis **by** *blast*

qed

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*: finite A $\implies \forall x \in A. \exists y. P x y \implies \exists f. \forall x \in A. P x (f x)$

proof (*induct rule: finite-induct*)

case *empty*

then show ?case **by** *simp*

next

case (*insert a A*)

then obtain *f b* **where** $f: \forall x \in A. P x (f x)$ **and** $ab: P a b$

by *auto*

show ?case (**is** $\exists f. ?P f$)

proof

show ?P ($\lambda x. \text{if } x = a \text{ then } b \text{ else } f x$)

using *f ab* **by** *auto*

qed

qed

20.1.2 Finite sets are the images of initial segments of natural numbers

lemma *finite-imp-nat-seg-image-inj-on*:

assumes finite A

shows $\exists (n::\text{nat}) f. A = f \text{ ` } \{i. i < n\} \wedge \text{inj-on } f \text{ } \{i. i < n\}$

using *assms*

proof *induct*

case *empty*

show ?case

proof

show $\exists f. \{\} = f \text{ ` } \{i::\text{nat}. i < 0\} \wedge \text{inj-on } f \text{ } \{i. i < 0\}$

by *simp*

qed

next

```

case (insert a A)
have notinA:  $a \notin A$  by fact
from insert.hyps obtain  $n f$  where  $A = f \text{ ‘ } \{i::\text{nat. } i < n\}$  inj-on  $f \{i. i < n\}$ 
  by blast
then have insert a A =  $f(n:=a) \text{ ‘ } \{i. i < \text{Suc } n\}$  and inj-on ( $f(n:=a)$ )  $\{i. i < \text{Suc } n\}$ 
  using notinA by (auto simp add: image-def Ball-def inj-on-def less-Suc-eq)
then show ?case by blast
qed

```

lemma *nat-seg-image-imp-finite*: $A = f \text{ ‘ } \{i::\text{nat. } i < n\} \implies \text{finite } A$

```

proof (induct n arbitrary: A)
  case 0
  then show ?case by simp
next
  case (Suc n)
  let ?B =  $f \text{ ‘ } \{i. i < n\}$ 
  have finB: finite ?B by (rule Suc.hyps[OF refl])
  show ?case
  proof (cases  $\exists k < n. f n = f k$ )
    case True
    then have  $A = ?B$ 
    using Suc.prems by (auto simp:less-Suc-eq)
    then show ?thesis
    using finB by simp
  next
  case False
  then have  $A = \text{insert } (f n) ?B$ 
  using Suc.prems by (auto simp:less-Suc-eq)
  then show ?thesis using finB by simp
  qed
qed

```

lemma *finite-conv-nat-seg-image*: $\text{finite } A \iff (\exists n f. A = f \text{ ‘ } \{i::\text{nat. } i < n\})$
by (*blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on*)

lemma *finite-imp-inj-to-nat-seg*:

```

assumes finite A
shows  $\exists f n. f \text{ ‘ } A = \{i::\text{nat. } i < n\} \wedge \text{inj-on } f A$ 
proof –
  from finite-imp-nat-seg-image-inj-on [OF  $\langle \text{finite } A \rangle$ ]
  obtain  $f$  and  $n :: \text{nat}$  where bij: bij-betw  $f \{i. i < n\} A$ 
  by (auto simp: bij-betw-def)
  let ?f = the-inv-into  $\{i. i < n\} f$ 
  have inj-on ?f  $A \wedge ?f \text{ ‘ } A = \{i. i < n\}$ 
  by (fold bij-betw-def) (rule bij-betw-the-inv-into[OF bij])
  then show ?thesis by blast
qed

```

lemma *finite-Collect-less-nat* [iff]: *finite* {*n*::*nat*. *n* < *k*}

by (*fastforce simp: finite-conv-nat-seg-image*)

lemma *finite-Collect-le-nat* [iff]: *finite* {*n*::*nat*. *n* ≤ *k*}

by (*simp add: le-eq-less-or-eq Collect-disj-eq*)

20.2 Finiteness and common set operations

lemma *rev-finite-subset*: *finite* *B* ⇒ *A* ⊆ *B* ⇒ *finite* *A*

proof (*induct arbitrary: A rule: finite-induct*)

case *empty*

then show ?*case* **by** *simp*

next

case (*insert* *x* *F* *A*)

have *A*: *A* ⊆ *insert* *x* *F* **and** *r*: *A* − {*x*} ⊆ *F* ⇒ *finite* (*A* − {*x*})

by *fact+*

show *finite* *A*

proof *cases*

assume *x*: *x* ∈ *A*

with *A* **have** *A* − {*x*} ⊆ *F* **by** (*simp add: subset-insert-iff*)

with *r* **have** *finite* (*A* − {*x*}) .

then have *finite* (*insert* *x* (*A* − {*x*})) ..

also have *insert* *x* (*A* − {*x*}) = *A*

using *x* **by** (*rule insert-Diff*)

finally show ?*thesis* .

next

show ?*thesis* **when** *A* ⊆ *F*

using *that* **by** *fact*

assume *x* ∉ *A*

with *A* **show** *A* ⊆ *F*

by (*simp add: subset-insert-iff*)

qed

qed

lemma *finite-subset*: *A* ⊆ *B* ⇒ *finite* *B* ⇒ *finite* *A*

by (*rule rev-finite-subset*)

simproc-setup *finite* (*finite* *A*) = <

let

val *finite-subset* = @{*thm finite-subset*}

val *Eq-TrueI* = @{*thm Eq-TrueI*}

fun *is-subset* *A* *th* = *case* *Thm.prop-of* *th* *of*

(- \$ (Const (**const-name** <*less-eq*>, Type (**type-name** <*fun*>, [Type (**type-name** <*set*>, -), -])) \$ *A* \$ *B*))

=> *if* *A* *aconv* *A'* *then* *SOME*(*B*,*th*) *else* *NONE*

| - => *NONE*;

fun *is-finite* *th* = *case* *Thm.prop-of* *th* *of*

```

      (- $ (Const (const-name ⟨finite⟩, -) $ A)) => SOME(A,th)
    | - => NONE;

  fun comb (A,sub-th) (A',fin-th) ths = if A aconv A' then (sub-th,fin-th) :: ths else
  ths

  fun proc ctxt ct =
    (let
      val - $ A = Thm.term-of ct
      val prems = Simplifier.prems-of ctxt
      val fins = map-filter is-finite prems
      val subsets = map-filter (is-subset A) prems
      in case fold-product comb subsets fins [] of
          (sub-th,fin-th) :: - => SOME((fin-th RS (sub-th RS finite-subset)) RS
Eq-TrueI)
        | - => NONE
      end)
  in K proc end
  >

```

```

declare [[simproc del: finite]]

```

```

lemma finite-UnI:
  assumes finite F and finite G
  shows finite (F ∪ G)
  using assms by induct simp-all

```

```

lemma finite-Un [iff]: finite (F ∪ G) ⟷ finite F ∧ finite G
  by (blast intro: finite-UnI finite-subset [of - F ∪ G])

```

```

lemma finite-insert [simp]: finite (insert a A) ⟷ finite A
proof –
  have finite {a} ∧ finite A ⟷ finite A by simp
  then have finite ({a} ∪ A) ⟷ finite A by (simp only: finite-Un)
  then show ?thesis by simp
qed

```

```

lemma finite-Int [simp, intro]: finite F ∨ finite G ⟹ finite (F ∩ G)
  by (blast intro: finite-subset)

```

```

lemma finite-Collect-conjI [simp, intro]:
  finite {x. P x} ∨ finite {x. Q x} ⟹ finite {x. P x ∧ Q x}
  by (simp add: Collect-conj-eq)

```

```

lemma finite-Collect-disjI [simp]:
  finite {x. P x ∨ Q x} ⟷ finite {x. P x} ∧ finite {x. Q x}
  by (simp add: Collect-disj-eq)

```

lemma *finite-Diff* [*simp, intro*]: $finite\ A \implies finite\ (A - B)$
by (*rule finite-subset, rule Diff-subset*)

lemma *finite-Diff2* [*simp*]:
assumes *finite B*
shows $finite\ (A - B) \longleftrightarrow finite\ A$

proof –
have $finite\ A \longleftrightarrow finite\ ((A - B) \cup (A \cap B))$
by (*simp add: Un-Diff-Int*)
also have $\dots \longleftrightarrow finite\ (A - B)$
using $\langle finite\ B \rangle$ **by** *simp*
finally show *?thesis ..*

qed

lemma *finite-Diff-insert* [*iff*]: $finite\ (A - insert\ a\ B) \longleftrightarrow finite\ (A - B)$

proof –
have $finite\ (A - B) \longleftrightarrow finite\ (A - B - \{a\})$ **by** *simp*
moreover have $A - insert\ a\ B = A - B - \{a\}$ **by** *auto*
ultimately show *?thesis by simp*

qed

lemma *finite-compl* [*simp*]:
 $finite\ (A :: 'a\ set) \implies finite\ (-\ A) \longleftrightarrow finite\ (UNIV :: 'a\ set)$
by (*simp add: Compl-eq-Diff-UNIV*)

lemma *finite-Collect-not* [*simp*]:
 $finite\ \{x :: 'a.\ P\ x\}\ \implies\ finite\ \{x.\ \neg\ P\ x\} \longleftrightarrow finite\ (UNIV :: 'a\ set)$
by (*simp add: Collect-neg-eq*)

lemma *finite-Union* [*simp, intro*]:
 $finite\ A \implies (\bigwedge M.\ M \in A \implies finite\ M) \implies finite\ (\bigcup A)$
by (*induct rule: finite-induct*) *simp-all*

lemma *finite-UN-I* [*intro*]:
 $finite\ A \implies (\bigwedge a.\ a \in A \implies finite\ (B\ a)) \implies finite\ (\bigcup_{a \in A} B\ a)$
by (*induct rule: finite-induct*) *simp-all*

lemma *finite-UN* [*simp*]: $finite\ A \implies finite\ (\bigcup (B\ ` A)) \longleftrightarrow (\forall x \in A.\ finite\ (B\ x))$
by (*blast intro: finite-subset*)

lemma *finite-Inter* [*intro*]: $\exists A \in M.\ finite\ A \implies finite\ (\bigcap M)$
by (*blast intro: Inter-lower finite-subset*)

lemma *finite-INT* [*intro*]: $\exists x \in I.\ finite\ (A\ x) \implies finite\ (\bigcap_{x \in I} A\ x)$
by (*blast intro: INT-lower finite-subset*)

lemma *finite-imageI* [*simp, intro*]: $finite\ F \implies finite\ (h\ ` F)$
by (*induct rule: finite-induct*) *simp-all*

lemma *finite-image-set* [*simp*]: $\text{finite } \{x. P x\} \implies \text{finite } \{f x \mid x. P x\}$
by (*simp add: image-Collect* [*symmetric*])

lemma *finite-image-set2*:

$\text{finite } \{x. P x\} \implies \text{finite } \{y. Q y\} \implies \text{finite } \{f x y \mid x y. P x \wedge Q y\}$
by (*rule finite-subset* [**where** $B = \bigcup x \in \{x. P x\}. \bigcup y \in \{y. Q y\}. \{f x y\}$]) *auto*)

lemma *finite-imageD*:

assumes *finite* ($f^{-1} A$) **and** *inj-on* $f A$

shows *finite* A

using *assms*

proof (*induct* $f^{-1} A$ *arbitrary: A*)

case *empty*

then show *?case* **by** *simp*

next

case (*insert* $x B$)

then have $B-A: \text{insert } x B = f^{-1} A$

by *simp*

then obtain y **where** $x = f y$ **and** $y \in A$

by *blast*

from $B-A \langle x \notin B \rangle$ **have** $B = f^{-1} A - \{x\}$

by *blast*

with $B-A \langle x \notin B \rangle \langle x = f y \rangle \langle \text{inj-on } f A \rangle \langle y \in A \rangle$ **have** $B = f^{-1} (A - \{y\})$

by (*simp add: inj-on-image-set-diff*)

moreover from $\langle \text{inj-on } f A \rangle$ **have** *inj-on* $f (A - \{y\})$

by (*rule inj-on-diff*)

ultimately have *finite* $(A - \{y\})$

by (*rule insert.hyps*)

then show *finite* A

by *simp*

qed

lemma *finite-image-iff*: $\text{inj-on } f A \implies \text{finite } (f^{-1} A) \longleftrightarrow \text{finite } A$

using *finite-imageD* **by** *blast*

lemma *finite-surj*: $\text{finite } A \implies B \subseteq f^{-1} A \implies \text{finite } B$

by (*erule finite-subset*) (*rule finite-imageI*)

lemma *finite-range-imageI*: $\text{finite } (\text{range } g) \implies \text{finite } (\text{range } (\lambda x. f (g x)))$

by (*erule finite-imageI*) (*simp add: range-composition*)

lemma *finite-subset-image*:

assumes *finite* B

shows $B \subseteq f^{-1} A \implies \exists C \subseteq A. \text{finite } C \wedge B = f^{-1} C$

using *assms*

proof *induct*

case *empty*

then show *?case* **by** *simp*

next

case *insert*
then show *?case*
 by (*clarsimp simp del: image-insert simp add: image-insert [symmetric]*) *blast*
qed

lemma *all-subset-image*: $(\forall B. B \subseteq f^{-1} A \longrightarrow P B) \longleftrightarrow (\forall B. B \subseteq A \longrightarrow P(f^{-1} B))$

by (*safe elim!: subset-imageE*) (*use image-mono in <blast+>*)

lemma *all-finite-subset-image*:

$(\forall B. \text{finite } B \wedge B \subseteq f^{-1} A \longrightarrow P B) \longleftrightarrow (\forall B. \text{finite } B \wedge B \subseteq A \longrightarrow P(f^{-1} B))$

proof *safe*

fix *B :: 'a set*

assume *B: finite B B ⊆ f⁻¹ A and P: ∀ B. finite B ∧ B ⊆ A ⟶ P (f⁻¹ B)*

show *P B*

using *finite-subset-image [OF B] P* by *blast*

qed *blast*

lemma *ex-finite-subset-image*:

$(\exists B. \text{finite } B \wedge B \subseteq f^{-1} A \wedge P B) \longleftrightarrow (\exists B. \text{finite } B \wedge B \subseteq A \wedge P(f^{-1} B))$

proof *safe*

fix *B :: 'a set*

assume *B: finite B B ⊆ f⁻¹ A and P B*

show $\exists B. \text{finite } B \wedge B \subseteq A \wedge P(f^{-1} B)$

using *finite-subset-image [OF B] <P B>* by *blast*

qed *blast*

lemma *finite-vimage-IntI*: $\text{finite } F \Longrightarrow \text{inj-on } h A \Longrightarrow \text{finite } (h^{-1} F \cap A)$

proof (*induct rule: finite-induct*)

case (*insert x F*)

then show *?case*

by (*simp add: vimage-insert [of h x F] finite-subset [OF inj-on-vimage-singleton] Int-Un-distrib2*)

qed *simp*

lemma *finite-finite-vimage-IntI*:

assumes *finite F*

and $\bigwedge y. y \in F \Longrightarrow \text{finite } ((h^{-1} \{y\}) \cap A)$

shows $\text{finite } (h^{-1} F \cap A)$

proof –

have $*$: $h^{-1} F \cap A = \bigcup_{y \in F}. (h^{-1} \{y\}) \cap A$

by *blast*

show *?thesis*

by (*simp only: * assms finite-UN-I*)

qed

lemma *finite-vimageI*: $\text{finite } F \Longrightarrow \text{inj } h \Longrightarrow \text{finite } (h^{-1} F)$

using *finite-vimage-IntI[of F h UNIV]* by *auto*

lemma *finite-vimageD'*: $finite (f - ' A) \implies A \subseteq range f \implies finite A$
by (*auto simp add: subset-image-iff intro: finite-subset[rotated]*)

lemma *finite-vimageD*: $finite (h - ' F) \implies surj h \implies finite F$
by (*auto dest: finite-vimageD'*)

lemma *finite-vimage-iff*: $bij h \implies finite (h - ' F) \longleftrightarrow finite F$
unfolding *bij-def* **by** (*auto elim: finite-vimageD finite-vimageI*)

lemma *finite-inverse-image-gen*:
assumes *finite A inj-on f D*
shows $finite \{j \in D. f j \in A\}$
using *finite-vimage-IntI [OF assms]*
by (*simp add: Collect-conj-eq inf-commute vimage-def*)

lemma *finite-inverse-image*:
assumes *finite A inj f*
shows $finite \{j. f j \in A\}$
using *finite-inverse-image-gen [OF assms]* **by** *simp*

lemma *finite-Collect-bex [simp]*:
assumes *finite A*
shows $finite \{x. \exists y \in A. Q x y\} \longleftrightarrow (\forall y \in A. finite \{x. Q x y\})$
proof –
have $\{x. \exists y \in A. Q x y\} = (\bigcup y \in A. \{x. Q x y\})$ **by** *auto*
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *finite-Collect-bounded-ex [simp]*:
assumes *finite {y. P y}*
shows $finite \{x. \exists y. P y \wedge Q x y\} \longleftrightarrow (\forall y. P y \longrightarrow finite \{x. Q x y\})$
proof –
have $\{x. \exists y. P y \wedge Q x y\} = (\bigcup y \in \{y. P y\}. \{x. Q x y\})$
by *auto*
with *assms* **show** *?thesis*
by *simp*
qed

lemma *finite-Plus*: $finite A \implies finite B \implies finite (A <+> B)$
by (*simp add: Plus-def*)

lemma *finite-PlusD*:
fixes $A :: 'a set$ **and** $B :: 'b set$
assumes *fin: finite (A <+> B)*
shows $finite A$ $finite B$
proof –
have $Inl - ' A \subseteq A <+> B$
by *auto*
then **have** $finite (Inl - ' A :: ('a + 'b) set)$

```

    using fin by (rule finite-subset)
  then show finite A
    by (rule finite-imageD) (auto intro: inj-onI)
next
  have Inr ‘ B  $\subseteq$  A <+> B
    by auto
  then have finite (Inr ‘ B :: ('a + 'b) set)
    using fin by (rule finite-subset)
  then show finite B
    by (rule finite-imageD) (auto intro: inj-onI)
qed

```

lemma *finite-Plus-iff* [simp]: $finite (A <+> B) \longleftrightarrow finite A \wedge finite B$
 by (auto intro: finite-PlusD finite-Plus)

lemma *finite-Plus-UNIV-iff* [simp]:
 $finite (UNIV :: ('a + 'b) set) \longleftrightarrow finite (UNIV :: 'a set) \wedge finite (UNIV :: 'b set)$
 by (subst UNIV-Plus-UNIV [symmetric]) (rule finite-Plus-iff)

lemma *finite-SigmaI* [simp, intro]:
 $finite A \implies (\bigwedge a. a \in A \implies finite (B a)) \implies finite (SIGMA a:A. B a)$
 unfolding Sigma-def by blast

lemma *finite-SigmaI2*:
 assumes finite { $x \in A. B x \neq \{\}$ }
 and $\bigwedge a. a \in A \implies finite (B a)$
 shows finite (Sigma A B)
proof –
 from assms have finite (Sigma { $x \in A. B x \neq \{\}$ } B)
 by auto
 also have Sigma { $x:A. B x \neq \{\}$ } B = Sigma A B
 by auto
 finally show ?thesis .
qed

lemma *finite-cartesian-product*: $finite A \implies finite B \implies finite (A \times B)$
 by (rule finite-SigmaI)

lemma *finite-Prod-UNIV*:
 $finite (UNIV :: 'a set) \implies finite (UNIV :: 'b set) \implies finite (UNIV :: ('a \times 'b) set)$
 by (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product)

lemma *finite-cartesian-productD1*:
 assumes finite (A \times B) and $B \neq \{\}$
 shows finite A
proof –
 from assms obtain n f where $A \times B = f ‘ \{i::nat. i < n\}$

```

  by (auto simp add: finite-conv-nat-seg-image)
  then have fst ' (A × B) = fst ' f ' {i::nat. i < n}
    by simp
  with ⟨B ≠ {}⟩ have A = (fst ∘ f) ' {i::nat. i < n}
    by (simp add: image-comp)
  then have ∃ n f. A = f ' {i::nat. i < n}
    by blast
  then show ?thesis
    by (auto simp add: finite-conv-nat-seg-image)
qed

```

lemma *finite-cartesian-productD2*:

assumes *finite* (A × B) and A ≠ {}
 shows *finite* B

proof –

```

  from assms obtain n f where A × B = f ' {i::nat. i < n}
    by (auto simp add: finite-conv-nat-seg-image)
  then have snd ' (A × B) = snd ' f ' {i::nat. i < n}
    by simp
  with ⟨A ≠ {}⟩ have B = (snd ∘ f) ' {i::nat. i < n}
    by (simp add: image-comp)
  then have ∃ n f. B = f ' {i::nat. i < n}
    by blast
  then show ?thesis
    by (auto simp add: finite-conv-nat-seg-image)

```

qed

lemma *finite-cartesian-product-iff*:

finite (A × B) ↔ (A = {} ∨ B = {} ∨ (*finite* A ∧ *finite* B))

by (auto dest: *finite-cartesian-productD1 finite-cartesian-productD2 finite-cartesian-product*)

lemma *finite-prod*:

finite (UNIV :: ('a × 'b) set) ↔ *finite* (UNIV :: 'a set) ∧ *finite* (UNIV :: 'b set)

using *finite-cartesian-product-iff*[of UNIV UNIV] by *simp*

lemma *finite-Pow-iff* [*iff*]: *finite* (Pow A) ↔ *finite* A

proof

assume *finite* (Pow A)

then have *finite* ((λx. {x}) ' A)

by (*blast intro: finite-subset*)

then show *finite* A

by (*rule finite-imageD [unfolded inj-on-def]*) *simp*

next

assume *finite* A

then show *finite* (Pow A)

by (*induct (simp-all add: Pow-insert)*)

qed

corollary *finite-Collect-subsets* [*simp*, *intro*]: $\text{finite } A \implies \text{finite } \{B. B \subseteq A\}$
by (*simp add: Pow-def [symmetric]*)

lemma *finite-set*: $\text{finite } (\text{UNIV} :: 'a \text{ set set}) \longleftrightarrow \text{finite } (\text{UNIV} :: 'a \text{ set})$
by (*simp only: finite-Pow-iff Pow-UNIV[symmetric]*)

lemma *finite-UnionD*: $\text{finite } (\bigcup A) \implies \text{finite } A$
by (*blast intro: finite-subset [OF subset-Pow-Union]*)

lemma *finite-bind*:
assumes *finite S*
assumes $\forall x \in S. \text{finite } (f x)$
shows $\text{finite } (\text{Set.bind } S f)$
using *assms* **by** (*simp add: bind-UNION*)

lemma *finite-filter* [*simp*]: $\text{finite } S \implies \text{finite } (\text{Set.filter } P S)$
unfolding *Set.filter-def* **by** *simp*

lemma *finite-set-of-finite-funs*:
assumes *finite A finite B*
shows $\text{finite } \{f. \forall x. (x \in A \longrightarrow f x \in B) \wedge (x \notin A \longrightarrow f x = d)\}$ (**is** *finite ?S*)
proof –
let $?F = \lambda f. \{(a,b). a \in A \wedge b = f a\}$
have $?F ' ?S \subseteq \text{Pow}(A \times B)$
by *auto*
from *finite-subset[OF this]* **assms** **have** 1: $\text{finite } (?F ' ?S)$
by *simp*
have 2: *inj-on ?F ?S*
by (*fastforce simp add: inj-on-def set-eq-iff fun-eq-iff*)
show *?thesis*
by (*rule finite-imageD [OF 1 2]*)
qed

lemma *not-finite-existsD*:
assumes $\neg \text{finite } \{a. P a\}$
shows $\exists a. P a$
proof (*rule classical*)
assume $\neg ?thesis$
with *assms* **show** *?thesis* **by** *auto*
qed

lemma *finite-converse* [*iff*]: $\text{finite } (r^{-1}) \longleftrightarrow \text{finite } r$
unfolding *converse-def conversep-iff*
using [*simproc add: finite-Collect*]
by (*auto elim: finite-imageD simp: inj-on-def*)

lemma *finite-Domain*: $\text{finite } r \implies \text{finite } (\text{Domain } r)$
by (*induct set: finite*) *auto*

lemma *finite-Range*: $\text{finite } r \implies \text{finite } (\text{Range } r)$
by (*induct set: finite*) *auto*

lemma *finite-Field*: $\text{finite } r \implies \text{finite } (\text{Field } r)$
by (*simp add: Field-def finite-Domain finite-Range*)

lemma *finite-Image[simp]*: $\text{finite } R \implies \text{finite } (R \text{ “ } A)$
by(*rule finite-subset[OF - finite-Range]*) *auto*

20.3 Further induction rules on finite sets

lemma *finite-ne-induct* [*case-names singleton insert, consumes 2*]:
assumes *finite F and* $F \neq \{\}$
assumes $\bigwedge x. P \{x\}$
and $\bigwedge x F. \text{finite } F \implies F \neq \{\} \implies x \notin F \implies P F \implies P (\text{insert } x F)$
shows $P F$
using *assms*
proof *induct*
case *empty*
then show *?case by simp*
next
case (*insert x F*)
then show *?case by cases auto*
qed

lemma *finite-subset-induct* [*consumes 2, case-names empty insert*]:
assumes *finite F and* $F \subseteq A$
and *empty: P {}*
and *insert: $\bigwedge a F. \text{finite } F \implies a \in A \implies a \notin F \implies P F \implies P (\text{insert } a F)$*
shows $P F$
using $\langle \text{finite } F \rangle \langle F \subseteq A \rangle$
proof *induct*
show $P \{\}$ **by** *fact*
next
fix $x F$
assume *finite F and* $x \notin F$ **and** $P: F \subseteq A \implies P F$ **and** $i: \text{insert } x F \subseteq A$
show $P (\text{insert } x F)$
proof (*rule insert*)
from i **show** $x \in A$ **by** *blast*
from i **have** $F \subseteq A$ **by** *blast*
with P **show** $P F$.
show *finite F by fact*
show $x \notin F$ **by** *fact*
qed
qed

lemma *finite-empty-induct*:
assumes *finite A*
and $P A$

```

    and remove:  $\bigwedge a A. \text{finite } A \implies a \in A \implies P A \implies P (A - \{a\})$ 
  shows  $P \{\}$ 
  proof -
    have  $P (A - B)$  if  $B \subseteq A$  for  $B :: 'a \text{ set}$ 
  proof -
    from  $\langle \text{finite } A \rangle$  that have finite  $B$ 
    by (rule rev-finite-subset)
    from this  $\langle B \subseteq A \rangle$  show  $P (A - B)$ 
  proof induct
    case empty
    from  $\langle P A \rangle$  show ?case by simp
  next
    case (insert b B)
    have  $P (A - B - \{b\})$ 
  proof (rule remove)
    from  $\langle \text{finite } A \rangle$  show finite  $(A - B)$ 
    by induct auto
    from insert show  $b \in A - B$ 
    by simp
    from insert show  $P (A - B)$ 
    by simp
  qed
  also have  $A - B - \{b\} = A - \text{insert } b B$ 
  by (rule Diff-insert [symmetric])
  finally show ?case .
  qed
  qed
  then have  $P (A - A)$  by blast
  then show ?thesis by simp
  qed

lemma finite-update-induct [consumes 1, case-names const update]:
  assumes finite: finite  $\{a. f a \neq c\}$ 
    and const:  $P (\lambda a. c)$ 
    and update:  $\bigwedge a b f. \text{finite } \{a. f a \neq c\} \implies f a = c \implies b \neq c \implies P f \implies P$ 
  ( $f(a := b)$ )
  shows  $P f$ 
  using finite
  proof (induct  $\{a. f a \neq c\}$  arbitrary: f)
    case empty
    with const show ?case by simp
  next
    case (insert a A)
    then have  $A = \{a'. (f(a := c)) a' \neq c\}$  and  $f a \neq c$ 
    by auto
    with  $\langle \text{finite } A \rangle$  have finite  $\{a'. (f(a := c)) a' \neq c\}$ 
    by simp
    have  $(f(a := c)) a = c$ 
    by simp
  
```

```

from insert  $\langle A = \{a'. (f(a := c)) a' \neq c\} \rangle$  have  $P (f(a := c))$ 
  by simp
with  $\langle \text{finite } \{a'. (f(a := c)) a' \neq c\} \rangle \langle (f(a := c)) a = c \rangle \langle f a \neq c \rangle$ 
have  $P ((f(a := c))(a := f a))$ 
  by (rule update)
then show ?case by simp
qed

```

```

lemma finite-subset-induct' [consumes 2, case-names empty insert]:
  assumes finite  $F$  and  $F \subseteq A$ 
    and empty:  $P \{\}$ 
    and insert:  $\bigwedge a F. [\text{finite } F; a \in A; F \subseteq A; a \notin F; P F] \implies P (\text{insert } a F)$ 
  shows  $P F$ 
  using assms(1,2)
proof induct
  show  $P \{\}$  by fact
next
  fix  $x F$ 
  assume finite  $F$  and  $x \notin F$  and
     $P: F \subseteq A \implies P F$  and  $i: \text{insert } x F \subseteq A$ 
  show  $P (\text{insert } x F)$ 
  proof (rule insert)
    from  $i$  show  $x \in A$  by blast
    from  $i$  have  $F \subseteq A$  by blast
    with  $P$  show  $P F$  .
    show finite  $F$  by fact
    show  $x \notin F$  by fact
    show  $F \subseteq A$  by fact
  qed
qed

```

20.4 Class *finite*

```

class finite =
  assumes finite-UNIV: finite (UNIV :: 'a set)
begin

```

```

lemma finite [simp]: finite (A :: 'a set)
  by (rule subset-UNIV finite-UNIV finite-subset)+

```

```

lemma finite-code [code]: finite (A :: 'a set)  $\longleftrightarrow$  True
  by simp

```

```

end

```

```

instance prod :: (finite, finite) finite
  by standard (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product
    finite)

```



```

lemma inj-graph: inj ( $\lambda f. \{(x, y). y = f x\}$ )
  by (rule inj-onI) (auto simp add: set-eq-iff fun-eq-iff)

instance fun :: (finite, finite) finite
proof
  show finite (UNIV :: ('a  $\Rightarrow$  'b) set)
  proof (rule finite-imageD)
    let ?graph =  $\lambda f::'a \Rightarrow 'b. \{(x, y). y = f x\}$ 
    have range ?graph  $\subseteq$  Pow UNIV
      by simp
    moreover have finite (Pow (UNIV :: ('a * 'b) set))
      by (simp only: finite-Pow-iff finite)
    ultimately show finite (range ?graph)
      by (rule finite-subset)
    show inj ?graph
      by (rule inj-graph)
  qed
qed

instance bool :: finite
  by standard (simp add: UNIV-bool)

instance set :: (finite) finite
  by standard (simp only: Pow-UNIV [symmetric] finite-Pow-iff finite)

instance unit :: finite
  by standard (simp add: UNIV-unit)

instance sum :: (finite, finite) finite
  by standard (simp only: UNIV-Plus-UNIV [symmetric] finite-Plus finite)

```

20.5 A basic fold functional for finite sets

The intended behaviour is $fold\ f\ z\ \{x_1, \dots, x_n\} = f\ x_1\ (\dots\ (f\ x_n\ z)\dots)$ if f is “left-commutative”. The commutativity requirement is relativised to the carrier set S :

```

locale comp-fun-commute-on =
  fixes S :: 'a set
  fixes f :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b
  assumes comp-fun-commute-on:  $x \in S \Longrightarrow y \in S \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y$ 
begin

lemma fun-left-comm:  $x \in S \Longrightarrow y \in S \Longrightarrow f\ y\ (f\ x\ z) = f\ x\ (f\ y\ z)$ 
  using comp-fun-commute-on by (simp add: fun-eq-iff)

lemma commute-left-comp:  $x \in S \Longrightarrow y \in S \Longrightarrow f\ y \circ (f\ x \circ g) = f\ x \circ (f\ y \circ g)$ 
  by (simp add: o-assoc comp-fun-commute-on)

end

```

inductive *fold-graph* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b ⇒ bool
for *f* :: 'a ⇒ 'b ⇒ 'b **and** *z* :: 'b
where
 emptyI [*intro*]: *fold-graph* *f* *z* {} *z*
 | *insertI* [*intro*]: $x \notin A \implies \text{fold-graph } f \ z \ A \ y \implies \text{fold-graph } f \ z \ (\text{insert } x \ A) \ (f \ x \ y)$

inductive-cases *empty-fold-graphE* [*elim!*]: *fold-graph* *f* *z* {} *x*

lemma *fold-graph-closed-lemma*:

fold-graph *f* *z* *A* *x* ∧ *x* ∈ *B*
if *fold-graph* *g* *z* *A* *x*
 ∧ *a* *b*. *a* ∈ *A* ⇒ *b* ∈ *B* ⇒ *f* *a* *b* = *g* *a* *b*
 ∧ *a* *b*. *a* ∈ *A* ⇒ *b* ∈ *B* ⇒ *g* *a* *b* ∈ *B*
 z ∈ *B*
using *that*(1-3)

proof (*induction rule*: *fold-graph.induct*)

case (*insertI* *x* *A* *y*)
have *fold-graph* *f* *z* *A* *y* *y* ∈ *B*
 unfolding *atomize-conj*
 by (*rule* *insertI.IH*) (*auto* *intro*: *insertI.prem*s)
then have *g* *x* *y* ∈ *B* **and** *f*-*eq*: *f* *x* *y* = *g* *x* *y*
 by (*auto simp*: *insertI.prem*s)
moreover have *fold-graph* *f* *z* (*insert* *x* *A*) (*f* *x* *y*)
 by (*rule* *fold-graph.insertI*; *fact*)
ultimately
show ?*case*
 by (*simp add*: *f*-*eq*)

qed (*auto intro!*: *that*)

lemma *fold-graph-closed-eq*:

fold-graph *f* *z* *A* = *fold-graph* *g* *z* *A*
if ∧ *a* *b*. *a* ∈ *A* ⇒ *b* ∈ *B* ⇒ *f* *a* *b* = *g* *a* *b*
 ∧ *a* *b*. *a* ∈ *A* ⇒ *b* ∈ *B* ⇒ *g* *a* *b* ∈ *B*
 z ∈ *B*
using *fold-graph-closed-lemma*[*of* *f* *z* *A* - *B* *g*] *fold-graph-closed-lemma*[*of* *g* *z* *A* - *B* *f*] *that*
by *auto*

definition *fold* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b

where *fold* *f* *z* *A* = (*if* *finite* *A* *then* (*THE* *y*. *fold-graph* *f* *z* *A* *y*) *else* *z*)

lemma *fold-closed-eq*: *fold* *f* *z* *A* = *fold* *g* *z* *A*

if ∧ *a* *b*. *a* ∈ *A* ⇒ *b* ∈ *B* ⇒ *f* *a* *b* = *g* *a* *b*
 ∧ *a* *b*. *a* ∈ *A* ⇒ *b* ∈ *B* ⇒ *g* *a* *b* ∈ *B*
 z ∈ *B*

unfolding *Finite-Set.fold-def*

by (*subst* *fold-graph-closed-eq*[**where** *B*=*B* **and** *g*=*g*]) (*auto simp*: *that*)

A tempting alternative for the definition is *if finite A then THE y. fold-graph f z A y else e*. It allows the removal of finiteness assumptions from the theorems *fold-comm*, *fold-reindex* and *fold-distrib*. The proofs become ugly. It is not worth the effort. (???)

lemma *finite-imp-fold-graph*: *finite A* $\implies \exists x.$ *fold-graph f z A x*
by (*induct rule: finite-induct*) *auto*

20.5.1 From *fold-graph* to *fold*

context *comp-fun-commute-on*
begin

lemma *fold-graph-finite*:
assumes *fold-graph f z A y*
shows *finite A*
using *assms* **by** *induct simp-all*

lemma *fold-graph-insertE-aux*:
assumes $A \subseteq S$
assumes *fold-graph f z A y a* $a \in A$
shows $\exists y'. y = f a y' \wedge \text{fold-graph } f z (A - \{a\}) y'$
using *assms(2-,1)*

proof (*induct set: fold-graph*)
case *emptyI*
then show *?case* **by** *simp*
next
case (*insertI x A y*)
show *?case*
proof (*cases x = a*)
case *True*
with *insertI* **show** *?thesis* **by** *auto*
next
case *False*
then obtain *y'* **where** *y: y = f a y'* **and** *y': fold-graph f z (A - {a}) y'*
using *insertI* **by** *auto*
from *insertI* **have** $x \in S$ $a \in S$ **by** *auto*
then have $f x y = f a (f x y')$
unfolding *y* **by** (*intro fun-left-comm; simp*)
moreover have *fold-graph f z (insert x A - {a}) (f x y')*
using *y'* **and** $\langle x \neq a \rangle$ **and** $\langle x \notin A \rangle$
by (*simp add: insert-Diff-if fold-graph.insertI*)
ultimately show *?thesis*
by *fast*
qed
qed

lemma *fold-graph-insertE*:
assumes *insert x A* $\subseteq S$
assumes *fold-graph f z (insert x A) v* **and** $x \notin A$

obtains y **where** $v = f x y$ **and** $\text{fold-graph } f z A y$
using assms **by** $(\text{auto dest: fold-graph-insertE-aux}[OF \langle \text{insert } x A \subseteq S \rangle - \text{insertI}])$

lemma fold-graph-determ :

assumes $A \subseteq S$

assumes $\text{fold-graph } f z A x \text{ fold-graph } f z A y$

shows $y = x$

using $\text{assms}(2-1)$

proof $(\text{induct arbitrary: } y \text{ set: fold-graph})$

case emptyI

then show $?case$ **by** fast

next

case $(\text{insertI } x A y v)$

from $\langle \text{insert } x A \subseteq S \rangle$ **and** $\langle \text{fold-graph } f z (\text{insert } x A) v \rangle$ **and** $\langle x \notin A \rangle$

obtain y' **where** $v = f x y'$ **and** $\text{fold-graph } f z A y'$

by $(\text{rule fold-graph-insertE})$

from $\langle \text{fold-graph } f z A y' \rangle \text{ insertI}$ **have** $y' = y$

by simp

with $\langle v = f x y' \rangle$ **show** $v = f x y$

by simp

qed

lemma fold-equality : $A \subseteq S \implies \text{fold-graph } f z A y \implies \text{fold } f z A = y$

by $(\text{cases finite } A) (\text{auto simp add: fold-def intro: fold-graph-determ dest: fold-graph-finite})$

lemma fold-graph-fold :

assumes $A \subseteq S$

assumes $\text{finite } A$

shows $\text{fold-graph } f z A (\text{fold } f z A)$

proof $-$

from $\langle \text{finite } A \rangle$ **have** $\exists x. \text{fold-graph } f z A x$

by $(\text{rule finite-imp-fold-graph})$

moreover note $\text{fold-graph-determ}[OF \langle A \subseteq S \rangle]$

ultimately have $\exists!x. \text{fold-graph } f z A x$

by (rule ex-ex1I)

then have $\text{fold-graph } f z A (\text{The } (\text{fold-graph } f z A))$

by $(\text{rule theI}')$

with assms **show** $?thesis$

by $(\text{simp add: fold-def})$

qed

The base case for fold :

lemma $(\text{in } -) \text{fold-infinite } [\text{simp}]: \neg \text{finite } A \implies \text{fold } f z A = z$

by $(\text{auto simp: fold-def})$

lemma $(\text{in } -) \text{fold-empty } [\text{simp}]: \text{fold } f z \{\} = z$

by $(\text{auto simp: fold-def})$

The various recursion equations for fold :

lemma *fold-insert* [*simp*]:
assumes $insert\ x\ A \subseteq S$
assumes *finite* A **and** $x \notin A$
shows $fold\ f\ z\ (insert\ x\ A) = f\ x\ (fold\ f\ z\ A)$
proof (*rule fold-equality*[*OF* $\langle insert\ x\ A \subseteq S \rangle$])
fix z
from $\langle insert\ x\ A \subseteq S \rangle$ $\langle finite\ A \rangle$ **have** *fold-graph* $f\ z\ A\ (fold\ f\ z\ A)$
by (*blast intro: fold-graph-fold*)
with $\langle x \notin A \rangle$ **have** *fold-graph* $f\ z\ (insert\ x\ A)\ (f\ x\ (fold\ f\ z\ A))$
by (*rule fold-graph.insertI*)
then show *fold-graph* $f\ z\ (insert\ x\ A)\ (f\ x\ (fold\ f\ z\ A))$
by *simp*
qed

declare (**in** $-$) *empty-fold-graphE* [*rule del*] *fold-graph.intros* [*rule del*]
 $-$ No more proofs involve these.

lemma *fold-fun-left-comm*:
assumes $insert\ x\ A \subseteq S$ *finite* A
shows $f\ x\ (fold\ f\ z\ A) = fold\ f\ (f\ x\ z)\ A$
using *assms*(2,1)
proof (*induct rule: finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert* $y\ F$)
then have *fold* $f\ (f\ x\ z)\ (insert\ y\ F) = f\ y\ (fold\ f\ (f\ x\ z)\ F)$
by *simp*
also have $\dots = f\ x\ (f\ y\ (fold\ f\ z\ F))$
using *insert* **by** (*simp add: fun-left-comm*[**where** $?y=x$])
also have $\dots = f\ x\ (fold\ f\ z\ (insert\ y\ F))$
proof $-$
from *insert* **have** $insert\ y\ F \subseteq S$ **by** *simp*
from *fold-insert*[*OF this*] *insert* **show** *?thesis* **by** *simp*
qed
finally show *?case* $..$
qed

lemma *fold-insert2*:
 $insert\ x\ A \subseteq S \implies finite\ A \implies x \notin A \implies fold\ f\ z\ (insert\ x\ A) = fold\ f\ (f\ x\ z)\ A$
by (*simp add: fold-fun-left-comm*)

lemma *fold-rec*:
assumes $A \subseteq S$
assumes *finite* A **and** $x \in A$
shows $fold\ f\ z\ A = f\ x\ (fold\ f\ z\ (A - \{x\}))$
proof $-$
have $A: A = insert\ x\ (A - \{x\})$

using $\langle x \in A \rangle$ **by** *blast*
then have $\text{fold } f \ z \ A = \text{fold } f \ z \ (\text{insert } x \ (A - \{x\}))$
by *simp*
also have $\dots = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$
by (*rule fold-insert*) (*use assms in* $\langle \text{auto} \rangle$)
finally show *?thesis* .
qed

lemma *fold-insert-remove*:
assumes $\text{insert } x \ A \subseteq S$
assumes *finite A*
shows $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$
proof –
from $\langle \text{finite } A \rangle$ **have** *finite (insert x A)*
by *auto*
moreover have $x \in \text{insert } x \ A$
by *auto*
ultimately have $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (\text{insert } x \ A - \{x\}))$
using $\langle \text{insert } x \ A \subseteq S \rangle$ **by** (*blast intro: fold-rec*)
then show *?thesis*
by *simp*
qed

lemma *fold-set-union-disj*:
assumes $A \subseteq S \ B \subseteq S$
assumes *finite A finite B A ∩ B = {}*
shows $\text{Finite-Set.fold } f \ z \ (A \cup B) = \text{Finite-Set.fold } f \ (\text{Finite-Set.fold } f \ z \ A) \ B$
using $\langle \text{finite } B \rangle$ *assms(1,2,3,5)*
proof *induct*
case (*insert x F*)
have $\text{fold } f \ z \ (A \cup \text{insert } x \ F) = f \ x \ (\text{fold } f \ (\text{fold } f \ z \ A) \ F)$
using *insert by auto*
also have $\dots = \text{fold } f \ (\text{fold } f \ z \ A) \ (\text{insert } x \ F)$
using *insert by (blast intro: fold-insert[symmetric])*
finally show *?case* .
qed *simp*

end

Other properties of *fold*:

lemma *fold-graph-image*:
assumes *inj-on g A*
shows $\text{fold-graph } f \ z \ (g \ ` \ A) = \text{fold-graph } (f \ o \ g) \ z \ A$
proof
fix w
show $\text{fold-graph } f \ z \ (g \ ` \ A) \ w = \text{fold-graph } (f \ o \ g) \ z \ A \ w$
proof
assume $\text{fold-graph } f \ z \ (g \ ` \ A) \ w$

```

then show fold-graph (f ∘ g) z A w
  using assms
proof (induct g ' A w arbitrary: A)
  case emptyI
  then show ?case by (auto intro: fold-graph.emptyI)
next
  case (insertI x A r B)
  from ⟨inj-on g B⟩ ⟨x ∉ A⟩ ⟨insert x A = image g B⟩ obtain x' A'
    where x' ∉ A' and [simp]: B = insert x' A' x = g x' A = g ' A'
    by (rule inj-img-insertE)
  from insertI.prem1 have fold-graph (f ∘ g) z A' r
    by (auto intro: insertI.hyps)
  with ⟨x' ∉ A'⟩ have fold-graph (f ∘ g) z (insert x' A') ((f ∘ g) x' r)
    by (rule fold-graph.insertI)
  then show ?case
    by simp
qed
next
assume fold-graph (f ∘ g) z A w
then show fold-graph f z (g ' A) w
  using assms
proof induct
  case emptyI
  then show ?case
    by (auto intro: fold-graph.emptyI)
next
  case (insertI x A r)
  from ⟨x ∉ A⟩ insertI.prem1 have g x ∉ g ' A
    by auto
  moreover from insertI have fold-graph f z (g ' A) r
    by simp
  ultimately have fold-graph f z (insert (g x) (g ' A)) (f (g x) r)
    by (rule fold-graph.insertI)
  then show ?case
    by simp
qed
qed
qed

lemma fold-image:
  assumes inj-on g A
  shows fold f z (g ' A) = fold (f ∘ g) z A
proof (cases finite A)
  case False
  with assms show ?thesis
    by (auto dest: finite-imageD simp add: fold-def)
next
  case True
  then show ?thesis

```

by (auto simp add: fold-def fold-graph-image[OF assms])
qed

lemma fold-cong:

assumes comp-fun-commute-on S f comp-fun-commute-on S g
and $A \subseteq S$ finite A
and cong: $\bigwedge x. x \in A \implies f x = g x$
and $s = t$ and $A = B$
shows fold f s A = fold g t B

proof –

have fold f s A = fold g s A

using ⟨finite A⟩ ⟨ $A \subseteq S$ ⟩ cong

proof (induct A)

case empty

then show ?case by simp

next

case insert

interpret f: comp-fun-commute-on S f by (fact ⟨comp-fun-commute-on S f⟩)

interpret g: comp-fun-commute-on S g by (fact ⟨comp-fun-commute-on S g⟩)

from insert show ?case by simp

qed

with assms show ?thesis by simp

qed

A simplified version for idempotent functions:

locale comp-fun-idem-on = comp-fun-commute-on +
assumes comp-fun-idem-on: $x \in S \implies f x \circ f x = f x$
begin

lemma fun-left-idem: $x \in S \implies f x (f x z) = f x z$
using comp-fun-idem-on by (simp add: fun-eq-iff)

lemma fold-insert-idem:

assumes insert x A \subseteq S

assumes fin: finite A

shows fold f z (insert x A) = f x (fold f z A)

proof cases

assume $x \in A$

then obtain B where $A = \text{insert } x B$ and $x \notin B$

by (rule set-insert)

then show ?thesis

using assms by (simp add: comp-fun-idem-on fun-left-idem)

next

assume $x \notin A$

then show ?thesis

using assms by auto

qed

declare fold-insert [simp del] fold-insert-idem [simp]

lemma *fold-insert-idem2*: $\text{insert } x \ A \subseteq S \implies \text{finite } A \implies \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f \ x \ z) \ A$
by (*simp add: fold-fun-left-comm*)

end

20.5.2 Liftings to *comp-fun-commute-on* etc.

lemma (**in** *comp-fun-commute-on*) *comp-comp-fun-commute-on*:
 $\text{range } g \subseteq S \implies \text{comp-fun-commute-on } R \ (f \circ g)$
by *standard* (*force intro: comp-fun-commute-on*)

lemma (**in** *comp-fun-idem-on*) *comp-comp-fun-idem-on*:
assumes $\text{range } g \subseteq S$
shows *comp-fun-idem-on* $R \ (f \circ g)$

proof

interpret *f-g*: *comp-fun-commute-on* $R \ f \circ g$

by (*fact comp-comp-fun-commute-on*[*OF* $\langle \text{range } g \subseteq S \rangle$])

show $x \in R \implies y \in R \implies (f \circ g) \ y \circ (f \circ g) \ x = (f \circ g) \ x \circ (f \circ g) \ y$ **for** $x \ y$

by (*fact f-g.comp-fun-commute-on*)

qed (*use* $\langle \text{range } g \subseteq S \rangle$ **in** $\langle \text{force intro: comp-fun-idem-on} \rangle$)

lemma (**in** *comp-fun-commute-on*) *comp-fun-commute-on-funpow*:
 $\text{comp-fun-commute-on } S \ (\lambda x. f \ x \ \hat{\hat{}} \ g \ x)$

proof

fix $x \ y$ **assume** $x \in S \ y \in S$

show $f \ y \ \hat{\hat{}} \ g \ y \circ f \ x \ \hat{\hat{}} \ g \ x = f \ x \ \hat{\hat{}} \ g \ x \circ f \ y \ \hat{\hat{}} \ g \ y$

proof (*cases* $x = y$)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

show *?thesis*

proof (*induct* $g \ x$ *arbitrary: g*)

case *0*

then show *?case* **by** *simp*

next

case (*Suc* $n \ g$)

have *hyp1*: $f \ y \ \hat{\hat{}} \ g \ y \circ f \ x = f \ x \circ f \ y \ \hat{\hat{}} \ g \ y$

proof (*induct* $g \ y$ *arbitrary: g*)

case *0*

then show *?case* **by** *simp*

next

case (*Suc* $n \ g$)

define *h* **where** $h \ z = g \ z - 1$ **for** z

with *Suc* **have** $n = h \ y$

by *simp*

with *Suc* **have** *hyp*: $f \ y \ \hat{\hat{}} \ h \ y \circ f \ x = f \ x \circ f \ y \ \hat{\hat{}} \ h \ y$

```

    by auto
  from Suc h-def have g y = Suc (h y)
  by simp
  with ⟨x ∈ S⟩ ⟨y ∈ S⟩ show ?case
  by (simp add: comp-assoc hyp) (simp add: o-assoc comp-fun-commute-on)
qed
define h where h z = (if z = x then g x - 1 else g z) for z
with Suc have n = h x
  by simp
with Suc have f y ~ h y ∘ f x ~ h x = f x ~ h x ∘ f y ~ h y
  by auto
with False h-def have hyp2: f y ~ g y ∘ f x ~ h x = f x ~ h x ∘ f y ~ g y
  by simp
from Suc h-def have g x = Suc (h x)
  by simp
then show ?case
  by (simp del: funpow.simps add: funpow-Suc-right o-assoc hyp2) (simp add:
comp-assoc hyp1)
  qed
  qed
  qed

```

20.5.3 UNIV as carrier set

```

locale comp-fun-commute =
  fixes f :: 'a ⇒ 'b ⇒ 'b
  assumes comp-fun-commute: f y ∘ f x = f x ∘ f y
begin

```

```

lemma (in -) comp-fun-commute-def': comp-fun-commute f = comp-fun-commute-on
UNIV f
  unfolding comp-fun-commute-def comp-fun-commute-on-def by blast

```

We abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

```

sublocale comp-fun-commute-on UNIV f
  rewrites ∧X. (X ⊆ UNIV) ≡ True
  and ∧x. x ∈ UNIV ≡ True
  and ∧P. (True ⇒ P) ≡ Trueprop P
  and ∧P Q. (True ⇒ PROP P ⇒ PROP Q) ≡ (PROP P ⇒ True ⇒
PROP Q)
proof -
  show comp-fun-commute-on UNIV f
  by standard (simp add: comp-fun-commute)
qed simp-all

end

```

```

lemma (in comp-fun-commute) comp-comp-fun-commute: comp-fun-commute (f o

```

g)
unfolding *comp-fun-commute-def'* **by** (*fact comp-comp-fun-commute-on*)

lemma (**in** *comp-fun-commute*) *comp-fun-commute-funpow*: *comp-fun-commute* ($\lambda x. f x \widehat{\sim} g x$)
unfolding *comp-fun-commute-def'* **by** (*fact comp-fun-commute-on-funpow*)

locale *comp-fun-idem* = *comp-fun-commute* +
assumes *comp-fun-idem*: $f x \circ f x = f x$
begin

lemma (**in** $-$) *comp-fun-idem-def'*: *comp-fun-idem* $f = \text{comp-fun-idem-on } UNIV f$
unfolding *comp-fun-idem-on-def* *comp-fun-idem-def* *comp-fun-commute-def'*
unfolding *comp-fun-idem-axioms-def* *comp-fun-idem-on-axioms-def*
by *blast*

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *comp-fun-idem-on UNIV f*
rewrites $\bigwedge X. (X \subseteq UNIV) \equiv True$
and $\bigwedge x. x \in UNIV \equiv True$
and $\bigwedge P. (True \implies P) \equiv Trueprop P$
and $\bigwedge P Q. (True \implies PROP P \implies PROP Q) \equiv (PROP P \implies True \implies PROP Q)$
proof $-$
show *comp-fun-idem-on UNIV f*
by *standard* (*simp-all add: comp-fun-idem comp-fun-commute*)
qed *simp-all*

end

lemma (**in** *comp-fun-idem*) *comp-comp-fun-idem*: *comp-fun-idem* ($f \circ g$)
unfolding *comp-fun-idem-def'* **by** (*fact comp-comp-fun-idem-on*)

20.5.4 Expressing set operations via *fold*

lemma *comp-fun-commute-const*: *comp-fun-commute* ($\lambda-. f$)
by *standard* (*rule refl*)

lemma *comp-fun-idem-insert*: *comp-fun-idem insert*
by *standard auto*

lemma *comp-fun-idem-remove*: *comp-fun-idem Set.remove*
by *standard auto*

lemma (**in** *semilattice-inf*) *comp-fun-idem-inf*: *comp-fun-idem inf*
by *standard* (*auto simp add: inf-left-commute*)

lemma (in *semilattice-sup*) *comp-fun-idem-sup: comp-fun-idem sup*
 by *standard (auto simp add: sup-left-commute)*

lemma *union-fold-insert:*
 assumes *finite A*
 shows $A \cup B = \text{fold insert } B A$
proof –
 interpret *comp-fun-idem insert*
 by (*fact comp-fun-idem-insert*)
 from $\langle \text{finite } A \rangle$ **show** *?thesis*
 by (*induct A arbitrary: B*) *simp-all*
qed

lemma *minus-fold-remove:*
 assumes *finite A*
 shows $B - A = \text{fold Set.remove } B A$
proof –
 interpret *comp-fun-idem Set.remove*
 by (*fact comp-fun-idem-remove*)
 from $\langle \text{finite } A \rangle$ **have** $\text{fold Set.remove } B A = B - A$
 by (*induct A arbitrary: B*) *auto*
 then **show** *?thesis ..*
qed

lemma *comp-fun-commute-filter-fold:*
comp-fun-commute ($\lambda x A'. \text{if } P x \text{ then Set.insert } x A' \text{ else } A')$
proof –
 interpret *comp-fun-idem Set.insert* by (*fact comp-fun-idem-insert*)
 show *?thesis* by *standard (auto simp: fun-eq-iff)*
qed

lemma *Set-filter-fold:*
 assumes *finite A*
 shows $\text{Set.filter } P A = \text{fold } (\lambda x A'. \text{if } P x \text{ then Set.insert } x A' \text{ else } A') \{\} A$
 using *assms*
proof –
 interpret *commute-insert: comp-fun-commute ($\lambda x A'. \text{if } P x \text{ then Set.insert } x A' \text{ else } A')$*
 by (*fact comp-fun-commute-filter-fold*)
 from $\langle \text{finite } A \rangle$ **show** *?thesis*
 by *induct (auto simp add: Set.filter-def)*
qed

lemma *inter-Set-filter:*
 assumes *finite B*
 shows $A \cap B = \text{Set.filter } (\lambda x. x \in A) B$
 using *assms*
 by *induct (auto simp: Set.filter-def)*

lemma *image-fold-insert*:

assumes *finite A*

shows $\text{image } f \ A = \text{fold } (\lambda k \ A. \ \text{Set.insert } (f \ k) \ A) \ \{\} \ A$

proof –

interpret *comp-fun-commute* $\lambda k \ A. \ \text{Set.insert } (f \ k) \ A$

by *standard auto*

show *?thesis*

using *assms* **by** (*induct A*) *auto*

qed

lemma *Ball-fold*:

assumes *finite A*

shows $\text{Ball } A \ P = \text{fold } (\lambda k \ s. \ s \ \wedge \ P \ k) \ \text{True } A$

proof –

interpret *comp-fun-commute* $\lambda k \ s. \ s \ \wedge \ P \ k$

by *standard auto*

show *?thesis*

using *assms* **by** (*induct A*) *auto*

qed

lemma *Bex-fold*:

assumes *finite A*

shows $\text{Bex } A \ P = \text{fold } (\lambda k \ s. \ s \ \vee \ P \ k) \ \text{False } A$

proof –

interpret *comp-fun-commute* $\lambda k \ s. \ s \ \vee \ P \ k$

by *standard auto*

show *?thesis*

using *assms* **by** (*induct A*) *auto*

qed

lemma *comp-fun-commute-Pow-fold*: *comp-fun-commute* $(\lambda x \ A. \ A \cup \text{Set.insert } x \ 'A)$

by (*clarsimp simp: fun-eq-iff comp-fun-commute-def*) *blast*

lemma *Pow-fold*:

assumes *finite A*

shows $\text{Pow } A = \text{fold } (\lambda x \ A. \ A \cup \text{Set.insert } x \ 'A) \ \{\{\}\} \ A$

proof –

interpret *comp-fun-commute* $\lambda x \ A. \ A \cup \text{Set.insert } x \ 'A$

by (*rule comp-fun-commute-Pow-fold*)

show *?thesis*

using *assms* **by** (*induct A*) (*auto simp: Pow-insert*)

qed

lemma *fold-union-pair*:

assumes *finite B*

shows $(\bigcup y \in B. \ \{(x, y)\}) \cup A = \text{fold } (\lambda y. \ \text{Set.insert } (x, y)) \ A \ B$

proof –

interpret *comp-fun-commute* $\lambda y. \ \text{Set.insert } (x, y)$

```

  by standard auto
  show ?thesis
  using assms by (induct arbitrary: A) simp-all
qed

```

```

lemma comp-fun-commute-product-fold:
  finite B  $\implies$  comp-fun-commute ( $\lambda x z. \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B$ )
  by standard (auto simp: fold-union-pair [symmetric])

```

```

lemma product-fold:
  assumes finite A finite B
  shows  $A \times B = \text{fold } (\lambda x z. \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B) \{\} A$ 
proof –
  interpret commute-product: comp-fun-commute ( $\lambda x z. \text{fold } (\lambda y. \text{Set.insert } (x, y)) z B$ )
  by (fact comp-fun-commute-product-fold[OF <finite B>])
  from assms show ?thesis unfolding Sigma-def
  by (induct A) (simp-all add: fold-union-pair)
qed

```

```

context complete-lattice
begin

```

```

lemma inf-Inf-fold-inf:
  assumes finite A
  shows  $\text{inf } (\text{Inf } A) B = \text{fold inf } B A$ 
proof –
  interpret comp-fun-idem inf
  by (fact comp-fun-idem-inf)
  from <finite A> fold-fun-left-comm show ?thesis
  by (induct A arbitrary: B) (simp-all add: inf-commute fun-eq-iff)
qed

```

```

lemma sup-Sup-fold-sup:
  assumes finite A
  shows  $\text{sup } (\text{Sup } A) B = \text{fold sup } B A$ 
proof –
  interpret comp-fun-idem sup
  by (fact comp-fun-idem-sup)
  from <finite A> fold-fun-left-comm show ?thesis
  by (induct A arbitrary: B) (simp-all add: sup-commute fun-eq-iff)
qed

```

```

lemma Inf-fold-inf: finite A  $\implies$   $\text{Inf } A = \text{fold inf top } A$ 
  using inf-Inf-fold-inf [of A top] by (simp add: inf-absorb2)

```

```

lemma Sup-fold-sup: finite A  $\implies$   $\text{Sup } A = \text{fold sup bot } A$ 
  using sup-Sup-fold-sup [of A bot] by (simp add: sup-absorb2)

```

lemma *inf-INF-fold-inf*:

assumes *finite A*

shows $\text{inf } B (\sqcap (f \text{ ` } A)) = \text{fold } (\text{inf } \circ f) B A$ (is *?inf = ?fold*)

proof –

interpret *comp-fun-idem inf* by (fact *comp-fun-idem-inf*)

interpret *comp-fun-idem inf ∘ f* by (fact *comp-comp-fun-idem*)

from $\langle \text{finite } A \rangle$ have *?fold = ?inf*

by (induct *A arbitrary: B*) (*simp-all add: inf-left-commute*)

then show *?thesis ..*

qed

lemma *sup-SUP-fold-sup*:

assumes *finite A*

shows $\text{sup } B (\sqcup (f \text{ ` } A)) = \text{fold } (\text{sup } \circ f) B A$ (is *?sup = ?fold*)

proof –

interpret *comp-fun-idem sup* by (fact *comp-fun-idem-sup*)

interpret *comp-fun-idem sup ∘ f* by (fact *comp-comp-fun-idem*)

from $\langle \text{finite } A \rangle$ have *?fold = ?sup*

by (induct *A arbitrary: B*) (*simp-all add: sup-left-commute*)

then show *?thesis ..*

qed

lemma *INF-fold-inf: finite A $\implies \sqcap (f \text{ ` } A) = \text{fold } (\text{inf } \circ f) \text{ top } A$*

using *inf-INF-fold-inf [of A top]* by *simp*

lemma *SUP-fold-sup: finite A $\implies \sqcup (f \text{ ` } A) = \text{fold } (\text{sup } \circ f) \text{ bot } A$*

using *sup-SUP-fold-sup [of A bot]* by *simp*

lemma *finite-Inf-in*:

assumes *finite A A ≠ {}* and *inf: $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \implies \text{inf } x y \in A$*

shows *Inf A ∈ A*

proof –

have *Inf B ∈ A if B ≤ A B ≠ {} for B*

using *finite-subset [OF <B ⊆ A> <finite A>]* that

by (*induction B*) (*use inf in <force+>*)

then show *?thesis*

by (*simp add: assms*)

qed

lemma *finite-Sup-in*:

assumes *finite A A ≠ {}* and *sup: $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \implies \text{sup } x y \in A$*

shows *Sup A ∈ A*

proof –

have *Sup B ∈ A if B ≤ A B ≠ {} for B*

using *finite-subset [OF <B ⊆ A> <finite A>]* that

by (*induction B*) (*use sup in <force+>*)

then show *?thesis*

by (*simp add: assms*)

qed

end

20.5.5 Expressing relation operations via *fold*

lemma *Id-on-fold*:

assumes *finite A*

shows $Id\text{-on } A = Finite\text{-Set.fold } (\lambda x. Set.insert (Pair x x)) \{\} A$

proof –

interpret *comp-fun-commute* $\lambda x. Set.insert (Pair x x)$

by *standard auto*

from *assms* **show** *?thesis*

unfolding *Id-on-def* **by** (*induct A*) *simp-all*

qed

lemma *comp-fun-commute-Image-fold*:

comp-fun-commute $(\lambda(x,y) A. \text{if } x \in S \text{ then } Set.insert y A \text{ else } A)$

proof –

interpret *comp-fun-idem* *Set.insert*

by (*fact comp-fun-idem-insert*)

show *?thesis*

by *standard (auto simp: fun-eq-iff comp-fun-commute split: prod.split)*

qed

lemma *Image-fold*:

assumes *finite R*

shows $R \text{ “ } S = Finite\text{-Set.fold } (\lambda(x,y) A. \text{if } x \in S \text{ then } Set.insert y A \text{ else } A) \{\} R$

proof –

interpret *comp-fun-commute* $(\lambda(x,y) A. \text{if } x \in S \text{ then } Set.insert y A \text{ else } A)$

by (*rule comp-fun-commute-Image-fold*)

have $*$: $\bigwedge x F. Set.insert x F \text{ “ } S = (\text{if } fst x \in S \text{ then } Set.insert (snd x) (F \text{ “ } S) \text{ else } (F \text{ “ } S))$

by (*force intro: rev-ImageI*)

show *?thesis*

using *assms* **by** (*induct R*) (*auto simp: **)

qed

lemma *insert-relcomp-union-fold*:

assumes *finite S*

shows $\{x\} O S \cup X = Finite\text{-Set.fold } (\lambda(w,z) A'. \text{if } snd x = w \text{ then } Set.insert (fst x,z) A' \text{ else } A') X S$

proof –

interpret *comp-fun-commute* $\lambda(w,z) A'. \text{if } snd x = w \text{ then } Set.insert (fst x,z) A' \text{ else } A'$

proof –

interpret *comp-fun-idem* *Set.insert*

by (*fact comp-fun-idem-insert*)

show *comp-fun-commute* $(\lambda(w,z) A'. \text{if } snd x = w \text{ then } Set.insert (fst x,z) A'$

else A'
 by *standard* (*auto simp add: fun-eq-iff split: prod.split*)
qed
have *: $\{x\} O S = \{(x', z). x' = \text{fst } x \wedge (\text{snd } x, z) \in S\}$
 by (*auto simp: relcomp-unfold intro!: exI*)
show ?thesis
unfolding * **using** $\langle \text{finite } S \rangle$ **by** (*induct S*) (*auto split: prod.split*)
qed

lemma *insert-relcomp-fold*:
assumes *finite S*
shows $\text{Set.insert } x R O S =$
 $\text{Finite-Set.fold } (\lambda(w,z) A'. \text{if } \text{snd } x = w \text{ then } \text{Set.insert } (\text{fst } x, z) A' \text{ else } A') (R$
 $O S) S$
proof –
have $\text{Set.insert } x R O S = (\{x\} O S) \cup (R O S)$
 by *auto*
then show ?thesis
 by (*auto simp: insert-relcomp-union-fold [OF assms]*)
qed

lemma *comp-fun-commute-relcomp-fold*:
assumes *finite S*
shows *comp-fun-commute* $(\lambda(x,y) A.$
 $\text{Finite-Set.fold } (\lambda(w,z) A'. \text{if } y = w \text{ then } \text{Set.insert } (x, z) A' \text{ else } A') A S)$
proof –
have *: $\bigwedge a b A.$
 $\text{Finite-Set.fold } (\lambda(w, z) A'. \text{if } b = w \text{ then } \text{Set.insert } (a, z) A' \text{ else } A') A S =$
 $\{(a,b)\} O S \cup A$
 by (*auto simp: insert-relcomp-union-fold [OF assms] cong: if-cong*)
show ?thesis
 by *standard* (*auto simp: **)
qed

lemma *relcomp-fold*:
assumes *finite R finite S*
shows $R O S = \text{Finite-Set.fold}$
 $(\lambda(x,y) A. \text{Finite-Set.fold } (\lambda(w,z) A'. \text{if } y = w \text{ then } \text{Set.insert } (x, z) A' \text{ else } A')$
 $A S) \{\}$ R
proof –
interpret *commute-relcomp-fold: comp-fun-commute*
 $(\lambda(x, y) A. \text{Finite-Set.fold } (\lambda(w, z) A'. \text{if } y = w \text{ then } \text{insert } (x, z) A' \text{ else } A')$
 $A S)$
by (*fact comp-fun-commute-relcomp-fold [OF $\langle \text{finite } S \rangle$]*)
from *assms* **show** ?thesis
by (*induct R*) (*auto simp: comp-fun-commute-relcomp-fold insert-relcomp-fold*
cong: if-cong)
qed

20.6 Locales as mini-packages for fold operations

20.6.1 The natural case

locale *fold*-on =
 fixes $S :: 'a \text{ set}$
 fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$ and $z :: 'b$
 assumes *comp-fun-commute-on*: $x \in S \Longrightarrow y \in S \Longrightarrow f y \circ f x = f x \circ f y$
begin

interpretation *fold?*: *comp-fun-commute-on* $S f$
 by *standard* (*simp add: comp-fun-commute-on*)

definition $F :: 'a \text{ set} \Rightarrow 'b$
 where *eq-fold*: $F A = \text{Finite-Set.fold } f z A$

lemma *empty* [*simp*]: $F \{\} = z$
 by (*simp add: eq-fold*)

lemma *infinite* [*simp*]: $\neg \text{finite } A \Longrightarrow F A = z$
 by (*simp add: eq-fold*)

lemma *insert* [*simp*]:
 assumes *insert* $x A \subseteq S$ and *finite* A and $x \notin A$
 shows $F (\text{insert } x A) = f x (F A)$
proof –
 from *fold-insert assms*
 have $\text{Finite-Set.fold } f z (\text{insert } x A)$
 $= f x (\text{Finite-Set.fold } f z A)$
 by *simp*
 with $\langle \text{finite } A \rangle$ **show** *?thesis* by (*simp add: eq-fold fun-eq-iff*)
qed

lemma *remove*:
 assumes $A \subseteq S$ and *finite* A and $x \in A$
 shows $F A = f x (F (A - \{x\}))$
proof –
 from $\langle x \in A \rangle$ **obtain** B **where** $A: A = \text{insert } x B$ and $x \notin B$
 by (*auto dest: mk-disjoint-insert*)
 moreover from $\langle \text{finite } A \rangle A$ **have** *finite* B by *simp*
 ultimately **show** *?thesis*
 using $\langle A \subseteq S \rangle$ by *auto*
qed

lemma *insert-remove*:
 assumes *insert* $x A \subseteq S$ and *finite* A
 shows $F (\text{insert } x A) = f x (F (A - \{x\}))$
 using *assms* by (*cases* $x \in A$) (*simp-all add: remove insert-absorb*)

end

20.6.2 With idempotency

```

locale folding-idem-on = folding-on +
  assumes comp-fun-idem-on:  $x \in S \implies y \in S \implies f\ x \circ f\ x = f\ x$ 
begin

declare insert [simp del]

interpretation fold?: comp-fun-idem-on S f
  by standard (simp-all add: comp-fun-commute-on comp-fun-idem-on)

lemma insert-idem [simp]:
  assumes insert  $x\ A \subseteq S$  and finite A
  shows  $F\ (\text{insert}\ x\ A) = f\ x\ (F\ A)$ 
proof –
  from fold-insert-idem assms
  have  $\text{fold}\ f\ z\ (\text{insert}\ x\ A) = f\ x\ (\text{fold}\ f\ z\ A)$  by simp
  with  $\langle \text{finite}\ A \rangle$  show ?thesis by (simp add: eq-fold fun-eq-iff)
qed

end

```

20.6.3 UNIV as the carrier set

```

locale folding =
  fixes  $f :: 'a \Rightarrow 'b \Rightarrow 'b$  and  $z :: 'b$ 
  assumes comp-fun-commute:  $f\ y \circ f\ x = f\ x \circ f\ y$ 
begin

lemma (in  $-$ ) folding-def': folding  $f = \text{folding-on}\ UNIV\ f$ 
  unfolding folding-def folding-on-def by blast

Again, we abuse the rewrites functionality of locales to remove trivial as-
sumptions that result from instantiating the carrier set to UNIV.

sublocale folding-on UNIV f
  rewrites  $\bigwedge X. (X \subseteq UNIV) \equiv True$ 
  and  $\bigwedge x. x \in UNIV \equiv True$ 
  and  $\bigwedge P. (True \implies P) \equiv \text{Trueprop}\ P$ 
  and  $\bigwedge P\ Q. (True \implies PROP\ P \implies PROP\ Q) \equiv (PROP\ P \implies True \implies$ 
PROP Q)
proof –
  show folding-on UNIV f
  by standard (simp add: comp-fun-commute)
qed simp-all

end

locale folding-idem = folding +
  assumes comp-fun-idem:  $f\ x \circ f\ x = f\ x$ 
begin

```

lemma (in $-$) *folding-idem-def'*: *folding-idem* $f = \text{folding-idem-on } UNIV\ f$
unfolding *folding-idem-def* *folding-def'* *folding-idem-on-def*
unfolding *folding-idem-axioms-def* *folding-idem-on-axioms-def*
by *blast*

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *folding-idem-on UNIV f*
rewrites $\bigwedge X. (X \subseteq UNIV) \equiv True$
and $\bigwedge x. x \in UNIV \equiv True$
and $\bigwedge P. (True \implies P) \equiv Trueprop\ P$
and $\bigwedge P\ Q. (True \implies PROP\ P \implies PROP\ Q) \equiv (PROP\ P \implies True \implies PROP\ Q)$
proof $-$
show *folding-idem-on UNIV f*
by *standard (simp add: comp-fun-idem)*
qed *simp-all*
end

20.7 Finite cardinality

The traditional definition $card\ A \equiv LEAST\ n. \exists f. A = \{f\ i \mid i < n\}$ is ugly to work with. But now that we have *fold* things are easy:

global-interpretation *card: folding $\lambda-. Suc\ 0$*
defines $card = \text{folding-on}.F\ (\lambda-. Suc)\ 0$
by *standard (rule refl)*

lemma *card-insert-disjoint: finite A $\implies x \notin A \implies card\ (\text{insert } x\ A) = Suc\ (card\ A)$*
by *(fact card.insert)*

lemma *card-insert-if: finite A $\implies card\ (\text{insert } x\ A) = (\text{if } x \in A\ \text{then } card\ A\ \text{else } Suc\ (card\ A))$*
by *auto (simp add: card.insert-remove card.remove)*

lemma *card-ge-0-finite: card A > 0 \implies finite A*
by *(rule ccontr) simp*

lemma *card-0-eq [simp]: finite A $\implies card\ A = 0 \iff A = \{\}$*
by *(auto dest: mk-disjoint-insert)*

lemma *finite-UNIV-card-ge-0: finite (UNIV :: 'a set) $\implies card\ (UNIV :: 'a set) > 0$*
by *(rule ccontr) simp*

lemma *card-eq-0-iff: card A = 0 $\iff A = \{\} \vee \neg\ \text{finite } A$*

by *auto*

lemma *card-range-greater-zero*: $\text{finite } (\text{range } f) \implies \text{card } (\text{range } f) > 0$
 by (*rule ccontr*) (*simp add: card-eq-0-iff*)

lemma *card-gt-0-iff*: $0 < \text{card } A \iff A \neq \{\} \wedge \text{finite } A$
 by (*simp add: neg0-conv [symmetric] card-eq-0-iff*)

lemma *card-Suc-Diff1*:
 assumes *finite A x ∈ A* shows $\text{Suc } (\text{card } (A - \{x\})) = \text{card } A$
proof –
 have $\text{Suc } (\text{card } (A - \{x\})) = \text{card } (\text{insert } x (A - \{x\}))$
 using *assms* by (*simp add: card.insert-remove*)
 also have $\dots = \text{card } A$
 using *assms* by (*simp add: card-insert-if*)
 finally show *?thesis* .
qed

lemma *card-insert-le-m1*:
 assumes $n > 0$ *card y ≤ n - 1* shows $\text{card } (\text{insert } x y) \leq n$
 using *assms*
 by (*cases finite y*) (*auto simp: card-insert-if*)

lemma *card-Diff-singleton*:
 assumes $x \in A$ shows $\text{card } (A - \{x\}) = \text{card } A - 1$
proof (*cases finite A*)
 case *True*
 with *assms* show *?thesis*
 by (*simp add: card-Suc-Diff1 [symmetric]*)
qed *auto*

lemma *card-Diff-singleton-if*:
 $\text{card } (A - \{x\}) = (\text{if } x \in A \text{ then } \text{card } A - 1 \text{ else } \text{card } A)$
 by (*simp add: card-Diff-singleton*)

lemma *card-Diff-insert[simp]*:
 assumes $a \in A$ and $a \notin B$
 shows $\text{card } (A - \text{insert } a B) = \text{card } (A - B) - 1$
proof –
 have $A - \text{insert } a B = (A - B) - \{a\}$
 using *assms* by *blast*
 then show *?thesis*
 using *assms* by (*simp add: card-Diff-singleton*)
qed

lemma *card-insert-le*: $\text{card } A \leq \text{card } (\text{insert } x A)$
proof (*cases finite A*)
 case *True*
 then show *?thesis* by (*simp add: card-insert-if*)

qed *auto*

lemma *card-Collect-less-nat[simp]*: $\text{card } \{i::\text{nat. } i < n\} = n$
by (*induct n*) (*simp-all add:less-Suc-eq Collect-disj-eq*)

lemma *card-Collect-le-nat[simp]*: $\text{card } \{i::\text{nat. } i \leq n\} = \text{Suc } n$
using *card-Collect-less-nat[of Suc n]* **by** (*simp add: less-Suc-eq-le*)

lemma *card-mono*:

assumes *finite B* **and** $A \subseteq B$

shows $\text{card } A \leq \text{card } B$

proof –

from *assms* **have** *finite A*

by (*auto intro: finite-subset*)

then show *?thesis*

using *assms*

proof (*induct A arbitrary: B*)

case *empty*

then show *?case* **by** *simp*

next

case (*insert x A*)

then have $x \in B$

by *simp*

from *insert* **have** $A \subseteq B - \{x\}$ **and** *finite (B - {x})*

by *auto*

with *insert.hyps* **have** $\text{card } A \leq \text{card } (B - \{x\})$

by *auto*

with $\langle \text{finite } A \rangle \langle x \notin A \rangle \langle \text{finite } B \rangle \langle x \in B \rangle$ **show** *?case*

by *simp (simp only: card.remove)*

qed

qed

lemma *card-seteq*:

assumes *finite B* **and** $A: A \subseteq B$ $\text{card } B \leq \text{card } A$

shows $A = B$

using *assms*

proof (*induction arbitrary: A rule: finite-induct*)

case (*insert b B*)

then have $A: \text{finite } A$ $A - \{b\} \subseteq B$

by *force+*

then have $\text{card } B \leq \text{card } (A - \{b\})$

using *insert* **by** (*auto simp add: card-Diff-singleton-if*)

then have $A - \{b\} = B$

using A *insert.IH* **by** *auto*

then show *?case*

using *insert.hyps insert.prem*s **by** *auto*

qed *auto*

lemma *psubset-card-mono*: $\text{finite } B \implies A < B \implies \text{card } A < \text{card } B$

using *card-seteq* [of $B A$] **by** (*auto simp add: psubset-eq*)

lemma *card-Un-Int*:

assumes *finite A finite B*

shows $\text{card } A + \text{card } B = \text{card } (A \cup B) + \text{card } (A \cap B)$

using *assms*

proof (*induct A*)

case *empty*

then show *?case* **by** *simp*

next

case *insert*

then show *?case*

by (*auto simp add: insert-absorb Int-insert-left*)

qed

lemma *card-Un-disjoint*: $\text{finite } A \implies \text{finite } B \implies A \cap B = \{\} \implies \text{card } (A \cup B) = \text{card } A + \text{card } B$

using *card-Un-Int* [of $A B$] **by** *simp*

lemma *card-Un-disjnt*: $\llbracket \text{finite } A; \text{finite } B; \text{disjnt } A B \rrbracket \implies \text{card } (A \cup B) = \text{card } A + \text{card } B$

by (*simp add: card-Un-disjoint disjnt-def*)

lemma *card-Un-le*: $\text{card } (A \cup B) \leq \text{card } A + \text{card } B$

proof (*cases finite A \wedge finite B*)

case *True*

then show *?thesis*

using *le-iff-add card-Un-Int* [of $A B$] **by** *auto*

qed *auto*

lemma *card-Diff-subset*:

assumes *finite B*

and $B \subseteq A$

shows $\text{card } (A - B) = \text{card } A - \text{card } B$

using *assms*

proof (*cases finite A*)

case *False*

with *assms* **show** *?thesis*

by *simp*

next

case *True*

with *assms* **show** *?thesis*

by (*induct B arbitrary: A*) *simp-all*

qed

lemma *card-Diff-subset-Int*:

assumes *finite (A \cap B)*

shows $\text{card } (A - B) = \text{card } A - \text{card } (A \cap B)$

proof –

have $A - B = A - A \cap B$ **by** *auto*
with *assms* **show** *?thesis*
by (*simp add: card-Diff-subset*)
qed

lemma *diff-card-le-card-Diff*:
assumes *finite B*
shows $\text{card } A - \text{card } B \leq \text{card } (A - B)$
proof –
have $\text{card } A - \text{card } B \leq \text{card } A - \text{card } (A \cap B)$
using *card-mono[OF assms Int-lower2, of A]* **by** *arith*
also have $\dots = \text{card } (A - B)$
using *assms* **by** (*simp add: card-Diff-subset-Int*)
finally show *?thesis* .
qed

lemma *card-le-sym-Diff*:
assumes *finite A finite B card A ≤ card B*
shows $\text{card}(A - B) \leq \text{card}(B - A)$
proof –
have $\text{card}(A - B) = \text{card } A - \text{card } (A \cap B)$ **using** *assms(1,2)* **by**(*simp add: card-Diff-subset-Int*)
also have $\dots \leq \text{card } B - \text{card } (A \cap B)$ **using** *assms(3)* **by** *linarith*
also have $\dots = \text{card}(B - A)$ **using** *assms(1,2)* **by**(*simp add: card-Diff-subset-Int Int-commute*)
finally show *?thesis* .
qed

lemma *card-less-sym-Diff*:
assumes *finite A finite B card A < card B*
shows $\text{card}(A - B) < \text{card}(B - A)$
proof –
have $\text{card}(A - B) = \text{card } A - \text{card } (A \cap B)$ **using** *assms(1,2)* **by**(*simp add: card-Diff-subset-Int*)
also have $\dots < \text{card } B - \text{card } (A \cap B)$ **using** *assms(1,3)* **by** (*simp add: card-mono diff-less-mono*)
also have $\dots = \text{card}(B - A)$ **using** *assms(1,2)* **by**(*simp add: card-Diff-subset-Int Int-commute*)
finally show *?thesis* .
qed

lemma *card-Diff1-less-iff*: $\text{card } (A - \{x\}) < \text{card } A \iff \text{finite } A \wedge x \in A$
proof (*cases finite A ∧ x ∈ A*)
case *True*
then show *?thesis*
by (*auto simp: card-gt-0-iff intro: diff-less*)
qed *auto*

lemma *card-Diff1-less*: $\text{finite } A \implies x \in A \implies \text{card } (A - \{x\}) < \text{card } A$

unfolding *card-Diff1-less-iff* **by** *auto*

lemma *card-Diff2-less*:

assumes *finite A x ∈ A y ∈ A* **shows** $\text{card } (A - \{x\} - \{y\}) < \text{card } A$

proof (*cases x = y*)

case *True*

with *assms* **show** *?thesis*

by (*simp add: card-Diff1-less del: card-Diff-insert*)

next

case *False*

then have $\text{card } (A - \{x\} - \{y\}) < \text{card } (A - \{x\})$ $\text{card } (A - \{x\}) < \text{card } A$

using *assms* **by** (*intro card-Diff1-less; simp*)⁺

then show *?thesis*

by (*blast intro: less-trans*)

qed

lemma *card-Diff1-le*: $\text{card } (A - \{x\}) \leq \text{card } A$

proof (*cases finite A*)

case *True*

then show *?thesis*

by (*cases x ∈ A*) (*simp-all add: card-Diff1-less less-imp-le*)

qed *auto*

lemma *card-psubset*: $\text{finite } B \implies A \subseteq B \implies \text{card } A < \text{card } B \implies A < B$

by (*erule psubsetI*) *blast*

lemma *card-le-inj*:

assumes *fA: finite A*

and *fB: finite B*

and *c: card A ≤ card B*

shows $\exists f. f ' A \subseteq B \wedge \text{inj-on } f A$

using *fA fB c*

proof (*induct arbitrary: B rule: finite-induct*)

case *empty*

then show *?case* **by** *simp*

next

case (*insert x s t*)

then show *?case*

proof (*induct rule: finite-induct [OF insert.prem(1)]*)

case *1*

then show *?case* **by** *simp*

next

case (*2 y t*)

from *2.prem(1,2,5) 2.hyps(1,2)* **have** *cst: card s ≤ card t*

by *simp*

from *2.prem(3) [OF 2.hyps(1) cst]*

obtain *f* **where** $*: f ' s \subseteq t$ *inj-on f s*

by *blast*

let *?g = (λa. if a = x then y else f a)*

```

  have ?g ‘ insert x s  $\subseteq$  insert y t  $\wedge$  inj-on ?g (insert x s)
    using * 2.premis(2) 2.hyps(2) unfolding inj-on-def by auto
  then show ?case by (rule exI[where ?x=?g])
qed

```

lemma *card-subset-eq*:

```

  assumes fB: finite B
    and AB: A  $\subseteq$  B
    and c: card A = card B
  shows A = B
proof –
  from fB AB have fA: finite A
    by (auto intro: finite-subset)
  from fA fB have fBA: finite (B - A)
    by auto
  have e: A  $\cap$  (B - A) = {}
    by blast
  have eq: A  $\cup$  (B - A) = B
    using AB by blast
  from card-Un-disjoint[OF fA fBA e, unfolded eq c] have card (B - A) = 0
    by arith
  then have B - A = {}
    unfolding card-eq-0-iff using fA fB by simp
  with AB show A = B
    by blast
qed

```

lemma *insert-partition*:

```

  x  $\notin$  F  $\implies$   $\forall$  c1  $\in$  insert x F.  $\forall$  c2  $\in$  insert x F. c1  $\neq$  c2  $\longrightarrow$  c1  $\cap$  c2 = {}  $\implies$ 
  x  $\cap$   $\bigcup$  F = {}
  by auto

```

lemma *finite-psubset-induct* [consumes 1, case-names psubset]:

```

  assumes finite: finite A
    and major:  $\bigwedge$ A. finite A  $\implies$  ( $\bigwedge$ B. B  $\subset$  A  $\implies$  P B)  $\implies$  P A
  shows P A
  using finite
proof (induct A taking: card rule: measure-induct-rule)
  case (less A)
  have fin: finite A by fact
  have ih: card B < card A  $\implies$  finite B  $\implies$  P B for B by fact
  have P B if B  $\subset$  A for B
  proof –
    from that have card B < card A
      using psubset-card-mono fin by blast
    moreover
    from that have B  $\subseteq$  A
      by auto
  end
end

```

```

    then have finite B
      using fin finite-subset by blast
    ultimately show ?thesis using ih by simp
  qed
  with fin show P A using major by blast
qed

```

lemma *finite-induct-select* [consumes 1, case-names empty select]:

```

  assumes finite S
    and P {}
    and select:  $\bigwedge T. T \subseteq S \implies P T \implies \exists s \in S - T. P (insert s T)$ 
  shows P S
  proof -
    have  $0 \leq \text{card } S$  by simp
    then have  $\exists T \subseteq S. \text{card } T = \text{card } S \wedge P T$ 
    proof (induct rule: dec-induct)
      case base with ⟨P {}⟩
        show ?case
          by (intro exI[of - {}]) auto
    next
      case (step n)
        then obtain T where  $T: T \subseteq S \text{ card } T = n \wedge P T$ 
          by auto
        with ⟨ $n < \text{card } S$ ⟩ have  $T \subset S \wedge P T$ 
          by auto
        with select[of T] obtain s where  $s \in S \wedge s \notin T \wedge P (insert s T)$ 
          by auto
        with step(2) T ⟨finite S⟩ show ?case
          by (intro exI[of - insert s T]) (auto dest: finite-subset)
    qed
  with ⟨finite S⟩ show P S
    by (auto dest: card-subset-eq)
qed

```

lemma *remove-induct* [case-names empty infinite remove]:

```

  assumes empty:  $P (\{\} :: 'a \text{ set})$ 
    and infinite:  $\neg \text{finite } B \implies P B$ 
    and remove:  $\bigwedge A. \text{finite } A \implies A \neq \{\} \implies A \subseteq B \implies (\bigwedge x. x \in A \implies P (A - \{x\})) \implies P A$ 
  shows P B
  proof (cases finite B)
    case False
      then show ?thesis by (rule infinite)
    next
      case True
        define A where  $A = B$ 
        with True have finite A  $A \subseteq B$ 
          by simp-all
        then show P A

```

```

proof (induct card A arbitrary: A)
  case 0
  then have  $A = \{\}$  by auto
  with empty show ?case by simp
next
  case (Suc n A)
  from  $\langle A \subseteq B \rangle$  and  $\langle \text{finite } B \rangle$  have finite A
    by (rule finite-subset)
  moreover from Suc.hyps have  $A \neq \{\}$  by auto
  moreover note  $\langle A \subseteq B \rangle$ 
  moreover have  $P (A - \{x\})$  if  $x: x \in A$  for x
    using x Suc.prem  $\langle \text{Suc } n = \text{card } A \rangle$  by (intro Suc) auto
  ultimately show ?case by (rule remove)
qed
qed

```

```

lemma finite-remove-induct [consumes 1, case-names empty remove]:
  fixes  $P :: 'a \text{ set} \Rightarrow \text{bool}$ 
  assumes finite B
  and  $P \{\}$ 
  and  $\bigwedge A. \text{finite } A \Rightarrow A \neq \{\} \Rightarrow A \subseteq B \Rightarrow (\bigwedge x. x \in A \Rightarrow P (A - \{x\}))$ 
 $\Rightarrow P A$ 
  defines  $B' \equiv B$ 
  shows  $P B'$ 
  by (induct B' rule: remove-induct) (simp-all add: assms)

```

Main cardinality theorem.

```

lemma card-partition [rule-format]:
  finite C  $\Rightarrow$  finite  $(\bigcup C) \Rightarrow (\forall c \in C. \text{card } c = k) \Rightarrow$ 
   $(\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \rightarrow c1 \cap c2 = \{\}) \Rightarrow$ 
   $k * \text{card } C = \text{card } (\bigcup C)$ 
proof (induct rule: finite-induct)
  case empty
  then show ?case by simp
next
  case (insert x F)
  then show ?case
    by (simp add: card-Un-disjoint insert-partition finite-subset [of -  $\bigcup$  (insert -)])
qed

```

```

lemma card-eq-UNIV-imp-eq-UNIV:
  assumes fin: finite (UNIV :: 'a set)
  and card: card A = card (UNIV :: 'a set)
  shows  $A = (\text{UNIV} :: 'a \text{ set})$ 
proof
  show  $A \subseteq \text{UNIV}$  by simp
  show  $\text{UNIV} \subseteq A$ 
  proof
    show  $x \in A$  for x
  qed

```

```

proof (rule ccontr)
  assume  $x \notin A$ 
  then have  $A \subset UNIV$  by auto
  with fin have  $\text{card } A < \text{card } (UNIV :: 'a \text{ set})$ 
    by (fact psubset-card-mono)
  with card show False by simp
qed
qed
qed

```

The form of a finite set of given cardinality

```

lemma card-eq-SucD:
  assumes  $\text{card } A = \text{Suc } k$ 
  shows  $\exists b B. A = \text{insert } b B \wedge b \notin B \wedge \text{card } B = k \wedge (k = 0 \longrightarrow B = \{\})$ 
proof -
  have fin: finite A
    using assms by (auto intro: ccontr)
  moreover have  $\text{card } A \neq 0$ 
    using assms by auto
  ultimately obtain b where  $b \in A$ 
    by auto
  show ?thesis
proof (intro exI conjI)
  show  $A = \text{insert } b (A - \{b\})$ 
    using b by blast
  show  $b \notin A - \{b\}$ 
    by blast
  show  $\text{card } (A - \{b\}) = k$  and  $k = 0 \longrightarrow A - \{b\} = \{\}$ 
    using assms b fin by (fastforce dest: mk-disjoint-insert)+
qed
qed

```

```

lemma card-Suc-eq:
   $\text{card } A = \text{Suc } k \longleftrightarrow$ 
  ( $\exists b B. A = \text{insert } b B \wedge b \notin B \wedge \text{card } B = k \wedge (k = 0 \longrightarrow B = \{\})$ )
by (auto simp: card-insert-if card-gt-0-iff elim!: card-eq-SucD)

```

```

lemma card-Suc-eq-finite:
   $\text{card } A = \text{Suc } k \longleftrightarrow (\exists b B. A = \text{insert } b B \wedge b \notin B \wedge \text{card } B = k \wedge \text{finite } B)$ 
unfolding card-Suc-eq using card-gt-0-iff by fastforce

```

```

lemma card-1-singletonE:
  assumes  $\text{card } A = 1$ 
  obtains x where  $A = \{x\}$ 
  using assms by (auto simp: card-Suc-eq)

```

```

lemma is-singleton-altdef:  $\text{is-singleton } A \longleftrightarrow \text{card } A = 1$ 
unfolding is-singleton-def
by (auto elim!: card-1-singletonE is-singletonE simp del: One-nat-def)

```

lemma *card-1-singleton-iff*: $\text{card } A = \text{Suc } 0 \longleftrightarrow (\exists x. A = \{x\})$
by (*simp add: card-Suc-eq*)

lemma *card-le-Suc0-iff-eq*:
assumes *finite A*
shows $\text{card } A \leq \text{Suc } 0 \longleftrightarrow (\forall a1 \in A. \forall a2 \in A. a1 = a2)$ (**is** $?C = ?A$)
proof
assume $?C$ **thus** $?A$ **using** *assms* **by** (*auto simp: le-Suc-eq dest: card-eq-SucD*)
next
assume $?A$
show $?C$
proof *cases*
assume $A = \{\}$ **thus** $?C$ **using** $\langle ?A \rangle$ **by** *simp*
next
assume $A \neq \{\}$
then obtain a **where** $A = \{a\}$ **using** $\langle ?A \rangle$ **by** *blast*
thus $?C$ **by** *simp*
qed
qed

lemma *card-le-Suc-iff*:
 $\text{Suc } n \leq \text{card } A = (\exists a B. A = \text{insert } a B \wedge a \notin B \wedge n \leq \text{card } B \wedge \text{finite } B)$
proof (*cases finite A*)
case *True*
then show $?thesis$
by (*fastforce simp: card-Suc-eq less-eq-nat.simps split: nat.splits*)
qed *auto*

lemma *finite-fun-UNIVD2*:
assumes *fin: finite (UNIV :: ('a \Rightarrow 'b) set)*
shows *finite (UNIV :: 'b set)*
proof –
from *fin* **have** *finite (range ($\lambda f :: 'a \Rightarrow 'b. f$ arbitrary))* **for** *arbitrary*
by (*rule finite-imageI*)
moreover have $\text{UNIV} = \text{range } (\lambda f :: 'a \Rightarrow 'b. f \text{ arbitrary})$ **for** *arbitrary*
by (*rule UNIV-eq-I*) *auto*
ultimately show *finite (UNIV :: 'b set)*
by *simp*
qed

lemma *card-UNIV-unit [simp]*: $\text{card } (\text{UNIV} :: \text{unit set}) = 1$
unfolding *UNIV-unit* **by** *simp*

lemma *infinite-arbitrarily-large*:
assumes $\neg \text{finite } A$
shows $\exists B. \text{finite } B \wedge \text{card } B = n \wedge B \subseteq A$
proof (*induction n*)
case 0

```

show ?case by (intro exI[of - {}]) auto
next
case (Suc n)
then obtain B where B: finite B  $\wedge$  card B = n  $\wedge$  B  $\subseteq$  A ..
with  $\langle \neg$  finite A  $\rangle$  have A  $\neq$  B by auto
with B have B  $\subseteq$  A by auto
then have  $\exists x. x \in A - B$ 
  by (elim psubset-imp-ex-mem)
then obtain x where x: x  $\in$  A - B ..
with B have finite (insert x B)  $\wedge$  card (insert x B) = Suc n  $\wedge$  insert x B  $\subseteq$  A
  by auto
then show  $\exists B. \text{finite } B \wedge \text{card } B = \text{Suc } n \wedge B \subseteq A$  ..
qed

```

Sometimes, to prove that a set is finite, it is convenient to work with finite subsets and to show that their cardinalities are uniformly bounded. This possibility is formalized in the next criterion.

lemma finite-if-finite-subsets-card-bdd:

assumes $\bigwedge G. G \subseteq F \implies \text{finite } G \implies \text{card } G \leq C$

shows finite F \wedge card F \leq C

proof (cases finite F)

case False

obtain n::nat **where** n: n > max C 0 **by** auto

obtain G **where** G: G \subseteq F card G = n **using** infinite-arbitrarily-large[OF False]

by auto

hence finite G **using** $\langle n > \text{max } C \ 0 \rangle$ **using** card.infinite gr-implies-not0 **by** blast

hence False **using** assms G n not-less **by** auto

thus ?thesis ..

next

case True **thus** ?thesis **using** assms[of F] **by** auto

qed

lemma obtain-subset-with-card-n:

assumes n \leq card S

obtains T **where** T \subseteq S card T = n finite T

proof –

obtain n' **where** card S = n + n'

using le-Suc-ex[OF assms] **by** blast

with that **show** thesis

proof (induct n' arbitrary: S)

case 0

thus ?case **by** (cases finite S) auto

next

case Suc

thus ?case **by** (auto simp add: card-Suc-eq)

qed

qed

lemma *exists-subset-between*:

assumes

$\text{card } A \leq n$

$n \leq \text{card } C$

$A \subseteq C$

finite C

shows $\exists B. A \subseteq B \wedge B \subseteq C \wedge \text{card } B = n$

using *assms*

proof (*induct* n *arbitrary*: A C)

case 0

thus *?case* **using** *finite-subset*[*of* A C] **by** (*intro* *exI*[*of* - $\{\}$], *auto*)

next

case (*Suc* n A C)

show *?case*

proof (*cases* $A = \{\}$)

case *True*

from *obtain-subset-with-card-n*[*OF* *Suc*(3)]

obtain B **where** $B \subseteq C$ $\text{card } B = \text{Suc } n$ **by** *blast*

thus *?thesis* **unfolding** *True* **by** *blast*

next

case *False*

then obtain a **where** $a: a \in A$ **by** *auto*

let $?A = A - \{a\}$

let $?C = C - \{a\}$

have $1: \text{card } ?A \leq n$ **using** *Suc*($2-$) a

using *finite-subset* **by** *fastforce*

have $2: \text{card } ?C \geq n$ **using** *Suc*($2-$) a **by** *auto*

from *Suc*(1)[*OF* 1 2 - *finite-subset*[*OF* - *Suc*(5)]] *Suc*($2-$)

obtain B **where** $?A \subseteq B$ $B \subseteq ?C$ $\text{card } B = n$ **by** *blast*

thus *?thesis* **using** a *Suc*($2-$)

by (*intro* *exI*[*of* - *insert* a B], *auto* *intro!*: *card-insert-disjoint* *finite-subset*[*of*

B C])

qed

qed

20.7.1 Cardinality of image

lemma *card-image-le*: *finite* $A \implies \text{card } (f \text{ ' } A) \leq \text{card } A$

by (*induct* *rule*: *finite-induct*) (*simp-all* *add*: *le-SucI* *card-insert-if*)

lemma *card-image*: *inj-on* f $A \implies \text{card } (f \text{ ' } A) = \text{card } A$

proof (*induct* A *rule*: *infinite-finite-induct*)

case (*infinite* A)

then have \neg *finite* $(f \text{ ' } A)$ **by** (*auto* *dest*: *finite-imageD*)

with *infinite* **show** *?case* **by** *simp*

qed *simp-all*

lemma *bij-betw-same-card*: *bij-betw* f A $B \implies \text{card } A = \text{card } B$

by (*auto* *simp*: *card-image* *bij-betw-def*)

lemma *endo-inj-surj*: $finite\ A \implies f\ ' \ A \subseteq A \implies inj\ on\ f\ A \implies f\ ' \ A = A$
by (*simp add: card-seteq card-image*)

lemma *eq-card-imp-inj-on*:
assumes *finite A* $card(f\ ' \ A) = card\ A$
shows *inj-on f A*
using *assms*
proof (*induct rule:finite-induct*)
case *empty*
show *?case* **by** *simp*
next
case (*insert x A*)
then show *?case*
using *card-image-le [of A f]* **by** (*simp add: card-insert-if split: if-splits*)
qed

lemma *inj-on-iff-eq-card*: $finite\ A \implies inj\ on\ f\ A \longleftrightarrow card\ (f\ ' \ A) = card\ A$
by (*blast intro: card-image eq-card-imp-inj-on*)

lemma *card-inj-on-le*:
assumes *inj-on f A* $f\ ' \ A \subseteq B$ *finite B*
shows $card\ A \leq card\ B$
proof –
have *finite A*
using *assms* **by** (*blast intro: finite-imageD dest: finite-subset*)
then show *?thesis*
using *assms* **by** (*force intro: card-mono simp: card-image [symmetric]*)
qed

lemma *inj-on-iff-card-le*:
 $\llbracket finite\ A; finite\ B \rrbracket \implies (\exists f. inj\ on\ f\ A \wedge f\ ' \ A \subseteq B) = (card\ A \leq card\ B)$
using *card-inj-on-le [of - A B]* *card-le-inj [of A B]* **by** *blast*

lemma *surj-card-le*: $finite\ A \implies B \subseteq f\ ' \ A \implies card\ B \leq card\ A$
by (*blast intro: card-image-le card-mono le-trans*)

lemma *card-bij-eq*:
 $inj\ on\ f\ A \implies f\ ' \ A \subseteq B \implies inj\ on\ g\ B \implies g\ ' \ B \subseteq A \implies finite\ A \implies finite\ B$
 $\implies card\ A = card\ B$
by (*auto intro: le-antisym card-inj-on-le*)

lemma *bij-betw-finite*: $bij\ betw\ f\ A\ B \implies finite\ A \longleftrightarrow finite\ B$
unfolding *bij-betw-def* **using** *finite-imageD [of f A]* **by** *auto*

lemma *inj-on-finite*: $inj\ on\ f\ A \implies f\ ' \ A \subseteq B \implies finite\ B \implies finite\ A$
using *finite-imageD finite-subset* **by** *blast*

lemma *card-vimage-inj-on-le*:

assumes *inj-on f D finite A*
shows $\text{card } (f^{-1}A \cap D) \leq \text{card } A$
proof (*rule card-inj-on-le*)
show *inj-on f (f⁻¹ A ∩ D)*
by (*blast intro: assms inj-on-subset*)
qed (*use assms in auto*)

lemma *card-vimage-inj*: $\text{inj } f \implies A \subseteq \text{range } f \implies \text{card } (f^{-1}A) = \text{card } A$
by (*auto 4 3 simp: subset-image-iff inj-vimage-image-eq*
intro: card-image[symmetric, OF subset-inj-on])

lemma *card-inverse[simp]*: $\text{card } (R^{-1}) = \text{card } R$
proof –
have $*$: $\bigwedge R. \text{prod.swap }^{-1} R = R^{-1}$ **by** *auto*
{
assume $\neg \text{finite } R$
hence *?thesis*
by *auto*
} **moreover** **{**
assume *finite R*
with *card-image-le[of R prod.swap] card-image-le[of R⁻¹ prod.swap]*
have *?thesis* **by** (*auto simp: **)
} **ultimately show** *?thesis* **by** *blast*
qed

20.7.2 Pigeonhole Principles

lemma *pigeonhole*: $\text{card } A > \text{card } (f^{-1}A) \implies \neg \text{inj-on } f A$
by (*auto dest: card-image less-irrefl-nat*)

lemma *pigeonhole-infinite*:
assumes $\neg \text{finite } A$ **and** *finite (f⁻¹A)*
shows $\exists a0 \in A. \neg \text{finite } \{a \in A. f a = f a0\}$
using *assms(2,1)*
proof (*induct f⁻¹A arbitrary: A rule: finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert b F*)
show *?case*
proof (*cases finite {a ∈ A. f a = b}*)
case *True*
with $\langle \neg \text{finite } A \rangle$ **have** $\neg \text{finite } (A - \{a \in A. f a = b\})$
by *simp*
also have $A - \{a \in A. f a = b\} = \{a \in A. f a \neq b\}$
by *blast*
finally have $\neg \text{finite } \{a \in A. f a \neq b\}$.
from *insert(3)[OF - this] insert(2,4)* **show** *?thesis*
by *simp (blast intro: rev-finite-subset)*

```

next
  case False
  then have  $\{a \in A. f a = b\} \neq \{\}$  by force
  with False show ?thesis by blast
qed
qed

```

lemma *pigeonhole-infinite-rel*:

```

assumes  $\neg$  finite A
  and finite B
  and  $\forall a \in A. \exists b \in B. R a b$ 
shows  $\exists b \in B. \neg$  finite  $\{a:A. R a b\}$ 
proof -
  let ?F =  $\lambda a. \{b \in B. R a b\}$ 
  from finite-Pow-iff[THEN iffD2, OF  $\langle$ finite B $\rangle$ ] have finite (?F ‘ A)
  by (blast intro: rev-finite-subset)
  from pigeonhole-infinite [where f = ?F, OF assms(1) this]
  obtain a0 where a0  $\in$  A and infinite:  $\neg$  finite  $\{a \in A. ?F a = ?F a0\}$  ..
  obtain b0 where b0  $\in$  B and R a0 b0
  using  $\langle a0 \in A \rangle$  assms(3) by blast
  have finite  $\{a \in A. ?F a = ?F a0\}$  if finite  $\{a \in A. R a b0\}$ 
  using  $\langle b0 \in B \rangle$   $\langle R a0 b0 \rangle$  that by (blast intro: rev-finite-subset)
  with infinite  $\langle b0 \in B \rangle$  show ?thesis
  by blast
qed

```

20.7.3 Cardinality of sums

lemma *card-Plus*:

```

assumes finite A finite B
shows card (A  $\langle + \rangle$  B) = card A + card B
proof -
  have Inl‘A  $\cap$  Inr‘B =  $\{\}$  by fast
  with assms show ?thesis
  by (simp add: Plus-def card-Un-disjoint card-image)
qed

```

lemma *card-Plus-conv-if*:

```

card (A  $\langle + \rangle$  B) = (if finite A  $\wedge$  finite B then card A + card B else 0)
by (auto simp add: card-Plus)

```

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma *dvd-partition*:

```

assumes f: finite  $(\bigcup C)$ 
  and  $\forall c \in C. k$  dvd card c  $\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}$ 
shows k dvd card  $(\bigcup C)$ 
proof -
  have finite C
  by (rule finite-UnionD [OF f])

```

```

then show ?thesis
  using assms
proof (induct rule: finite-induct)
  case empty
  show ?case by simp
next
  case (insert c C)
  then have  $c \cap \bigcup C = \{\}$ 
  by auto
  with insert show ?case
  by (simp add: card-Un-disjoint)
qed
qed

```

20.8 Minimal and maximal elements of finite sets

context begin

qualified lemma

```

assumes finite A and asymp-on A R and transp-on A R and  $\exists x \in A. P x$ 
shows
  bex-min-element-with-property:  $\exists x \in A. P x \wedge (\forall y \in A. R y x \longrightarrow \neg P y)$  and
  bex-max-element-with-property:  $\exists x \in A. P x \wedge (\forall y \in A. R x y \longrightarrow \neg P y)$ 
unfolding atomize-conj
using assms
proof (induction A rule: finite-induct)
  case empty
  hence False
  by simp-all
  thus ?case ..
next
  case (insert x F)

from insert.prem1 have asymp-on F R
  using asymp-on-subset by blast

from insert.prem2 have transp-on F R
  using transp-on-subset by blast

show ?case
proof (cases P x)
  case True
  show ?thesis
proof (cases  $\exists a \in F. P a$ )
  case True
  with insert.IH obtain min max where
    min  $\in F$  and P min and  $\forall z \in F. R z min \longrightarrow \neg P z$ 
    max  $\in F$  and P max and  $\forall z \in F. R max z \longrightarrow \neg P z$ 
  using  $\langle$ asymp-on F R $\rangle$   $\langle$ transp-on F R $\rangle$  by auto

```

```

show ?thesis
proof (rule conjI)
  show  $\exists y \in \text{insert } x F. P y \wedge (\forall z \in \text{insert } x F. R y z \longrightarrow \neg P z)$ 
  proof (cases R max x)
    case True
      show ?thesis
      proof (intro bexI conjI ballI impI)
        show  $x \in \text{insert } x F$ 
          by simp
        next
          show  $P x$ 
            using  $\langle P x \rangle$  by simp
        next
          fix  $z$  assume  $z \in \text{insert } x F$  and  $R x z$ 
          hence  $z = x \vee z \in F$ 
            by simp
          thus  $\neg P z$ 
          proof (rule disjE)
            assume  $z = x$ 
            hence  $R x x$ 
              using  $\langle R x x \rangle$  by simp
            moreover have  $\neg R x x$ 
              using  $\langle \text{asyp-on } (\text{insert } x F) R \rangle$  [THEN irreflp-on-if-asyp-on, THEN
irreflp-onD]
              by simp
            ultimately have False
              by simp
            thus ?thesis ..
          next
            assume  $z \in F$ 
            moreover have  $R \text{ max } z$ 
              using  $\langle R \text{ max } x \rangle$   $\langle R x z \rangle$ 
              using  $\langle \text{transp-on } (\text{insert } x F) R \rangle$  [THEN transp-onD, of max x z]
              using  $\langle \text{max} \in F \rangle$   $\langle z \in F \rangle$  by simp
            ultimately show ?thesis
              using  $\langle \forall z \in F. R \text{ max } z \longrightarrow \neg P z \rangle$  by simp
          qed
        qed
      next
        case False
          show ?thesis
          proof (intro bexI conjI ballI impI)
            show  $\text{max} \in \text{insert } x F$ 
              using  $\langle \text{max} \in F \rangle$  by simp
            next
              show  $P \text{ max}$ 
                using  $\langle P \text{ max} \rangle$  by simp
            next

```

```

fix z assume z ∈ insert x F and R max z
hence z = x ∨ z ∈ F
  by simp
thus ¬ P z
proof (rule disjE)
  assume z = x
  hence False
  using ⟨¬ R max x⟩ ⟨R max z⟩ by simp
  thus ?thesis ..
next
  assume z ∈ F
  thus ?thesis
  using ⟨R max z⟩ ⟨∀ z ∈ F. R max z ⟶ ¬ P z⟩ by simp
qed
qed
qed
next
show ∃ y ∈ insert x F. P y ∧ (∀ z ∈ insert x F. R z y ⟶ ¬ P z)
proof (cases R x min)
  case True
  show ?thesis
  proof (intro bexI conjI ballI impI)
    show x ∈ insert x F
    by simp
  next
    show P x
    using ⟨P x⟩ by simp
  next
    fix z assume z ∈ insert x F and R z x
    hence z = x ∨ z ∈ F
    by simp
    thus ¬ P z
    proof (rule disjE)
      assume z = x
      hence R x x
      using ⟨R z x⟩ by simp
      moreover have ¬ R x x
      using ⟨asympt-on (insert x F) R⟩ [THEN irreflp-on-if-asympt-on, THEN
irreflp-onD]
      by simp
      ultimately have False
      by simp
    thus ?thesis ..
  next
    assume z ∈ F
    moreover have R z min
    using ⟨R z x⟩ ⟨R x min⟩
    using ⟨transp-on (insert x F) R⟩ [THEN transp-onD, of z x min]
    using ⟨min ∈ F⟩ ⟨z ∈ F⟩ by simp

```

```

      ultimately show ?thesis
      using ⟨ $\forall z \in F. R z \text{ min} \longrightarrow \neg P z$ ⟩ by simp
    qed
  qed
next
  case False
  show ?thesis
  proof (intro bexI conjI ballI impI)
    show  $\text{min} \in \text{insert } x F$ 
    using ⟨ $\text{min} \in F$ ⟩ by simp
  next
    show  $P \text{ min}$ 
    using ⟨ $P \text{ min}$ ⟩ by simp
  next
    fix z assume  $z \in \text{insert } x F$  and  $R z \text{ min}$ 
    hence  $z = x \vee z \in F$ 
    by simp
    thus  $\neg P z$ 
    proof (rule disjE)
      assume  $z = x$ 
      hence False
      using ⟨ $\neg R x \text{ min}$ ⟩ ⟨ $R z \text{ min}$ ⟩ by simp
    thus ?thesis ..
  next
    assume  $z \in F$ 
    thus ?thesis
    using ⟨ $R z \text{ min}$ ⟩ ⟨ $\forall z \in F. R z \text{ min} \longrightarrow \neg P z$ ⟩ by simp
  qed
  qed
  qed
next
  case False
  then show ?thesis
  using ⟨ $\exists a \in \text{insert } x F. P a$ ⟩
  using ⟨asympt-on ( $\text{insert } x F$ )  $R$ ⟩ [THEN asympt-onD, of  $x$ ] insert-iff [of  $- x$ 
F]
  by blast
  qed
next
  case False
  with insert.prems have  $\exists x \in F. P x$ 
  by simp
  with insert.IH have
   $\exists y \in F. P y \wedge (\forall z \in F. R z y \longrightarrow \neg P z)$ 
   $\exists y \in F. P y \wedge (\forall z \in F. R y z \longrightarrow \neg P z)$ 
  using ⟨asympt-on  $F R$ ⟩ ⟨transp-on  $F R$ ⟩ by auto
  thus ?thesis
  using False by auto

```

qed
qed

qualified lemma

assumes *finite A and asymp-on A R and transp-on A R and $A \neq \{\}$*

shows

be_x-min-element: $\exists m \in A. \forall x \in A. x \neq m \longrightarrow \neg R\ x\ m$ **and**

be_x-max-element: $\exists m \in A. \forall x \in A. x \neq m \longrightarrow \neg R\ m\ x$

using $\langle A \neq \{\} \rangle$

be_x-min-element-with-property[*OF assms(1,2,3), of $\lambda\cdot$. True, simplified*]

be_x-max-element-with-property[*OF assms(1,2,3), of $\lambda\cdot$. True, simplified*]

by *blast+*

end

The following alternative form might sometimes be easier to work with.

lemma *is-min-element-in-set-iff*:

asymp-on A R $\implies (\forall y \in A. y \neq x \longrightarrow \neg R\ y\ x) \longleftrightarrow (\forall y. R\ y\ x \longrightarrow y \notin A)$

by (*auto dest: asymp-onD*)

lemma *is-max-element-in-set-iff*:

asymp-on A R $\implies (\forall y \in A. y \neq x \longrightarrow \neg R\ x\ y) \longleftrightarrow (\forall y. R\ x\ y \longrightarrow y \notin A)$

by (*auto dest: asymp-onD*)

context begin

qualified lemma

assumes *finite A and $A \neq \{\}$ and transp-on A R and totalp-on A R*

shows

be_x-least-element: $\exists l \in A. \forall x \in A. x \neq l \longrightarrow R\ l\ x$ **and**

be_x-greatest-element: $\exists g \in A. \forall x \in A. x \neq g \longrightarrow R\ x\ g$

unfolding *atomize-conj*

using *assms*

proof (*induction A rule: finite-induct*)

case *empty*

hence *False* **by** *simp*

thus *?case ..*

next

case (*insert a A'*)

from *insert.prem_s(2)* **have** *transp-on-A'*: *transp-on A' R*

by (*auto intro: transp-onI dest: transp-onD*)

from *insert.prem_s(3)* **have**

totalp-on-a-A'-raw: $\forall y \in A'. a \neq y \longrightarrow R\ a\ y \vee R\ y\ a$ **and**

totalp-on-A': *totalp-on A' R*

by (*simp-all add: totalp-on-def*)

show *?case*


```

proof (cases A' = {})
  case True
  thus ?thesis by simp
next
  case False
  then obtain least greatest where
    least ∈ A' and least-of-A': ∀ x ∈ A'. x ≠ least ⟶ R least x and
    greatest ∈ A' and greatest-of-A': ∀ x ∈ A'. x ≠ greatest ⟶ R x greatest
  using insert.IH[OF - transp-on-A' totalp-on-A'] by auto

show ?thesis
proof (rule conjI)
  show ∃ l ∈ insert a A'. ∀ x ∈ insert a A'. x ≠ l ⟶ R l x
  proof (cases R a least)
    case True
    show ?thesis
    proof (intro bexI ballI impI)
      show a ∈ insert a A'
      by simp
    next
    fix x
    show ∧x. x ∈ insert a A' ⟹ x ≠ a ⟹ R a x
      using True ⟨least ∈ A'⟩ least-of-A'
      using insert.prem(2)[THEN transp-onD, of a least]
      by auto
    qed
  next
  case False
  show ?thesis
  proof (intro bexI ballI impI)
    show least ∈ insert a A'
    using ⟨least ∈ A'⟩ by simp
  next
  fix x
  show x ∈ insert a A' ⟹ x ≠ least ⟹ R least x
    using False ⟨least ∈ A'⟩ least-of-A' totalp-on-a-A'-raw
    by (cases x = a) auto
  qed
qed
next
show ∃ g ∈ insert a A'. ∀ x ∈ insert a A'. x ≠ g ⟶ R x g
proof (cases R greatest a)
  case True
  show ?thesis
  proof (intro bexI ballI impI)
    show a ∈ insert a A'
    by simp
  next
  fix x

```

```

show  $\bigwedge x. x \in \text{insert } a \ A' \implies x \neq a \implies R \ x \ a$ 
  using True  $\langle \text{greatest} \in A' \rangle$  greatest-of-A'
  using insert.premis(2)[THEN transp-onD, of - greatest a]
  by auto
qed
next
case False
show ?thesis
proof (intro bexI ballI impI)
  show  $\text{greatest} \in \text{insert } a \ A'$ 
  using  $\langle \text{greatest} \in A' \rangle$  by simp
next
fix x
show  $x \in \text{insert } a \ A' \implies x \neq \text{greatest} \implies R \ x \ \text{greatest}$ 
  using False  $\langle \text{greatest} \in A' \rangle$  greatest-of-A' totalp-on-a-A'-raw
  by (cases x = a) auto
qed
qed
qed
qed
qed
end

```

20.8.1 Finite orders

```

context order
begin

```

lemma *finite-has-maximal*:

```

assumes finite A and A ≠ {}
shows  $\exists m \in A. \forall b \in A. m \leq b \implies m = b$ 

```

proof –

```

obtain m where m ∈ A and m-is-max:  $\forall x \in A. x \neq m \implies \neg m < x$ 
  using Finite-Set.bex-max-element[OF  $\langle \text{finite } A \rangle$  - -  $\langle A \neq \{\} \rangle$ , of (<)] by auto
moreover have  $\forall b \in A. m \leq b \implies m = b$ 
  using m-is-max by (auto simp: le-less)
ultimately show ?thesis
  by auto

```

qed

lemma *finite-has-maximal2*:

```

[[ finite A; a ∈ A ]]  $\implies \exists m \in A. a \leq m \wedge (\forall b \in A. m \leq b \implies m = b)$ 
using finite-has-maximal[of  $\{b \in A. a \leq b\}$ ] by fastforce

```

lemma *finite-has-minimal*:

```

assumes finite A and A ≠ {}
shows  $\exists m \in A. \forall b \in A. b \leq m \implies m = b$ 

```

proof –

obtain m **where** $m \in A$ **and** m -is-min: $\forall x \in A. x \neq m \longrightarrow \neg x < m$
using *Finite-Set.bex-min-element*[*OF* $\langle \text{finite } A \rangle$ - - $\langle A \neq \{\} \rangle$, of $(<)$] **by** *auto*
moreover have $\forall b \in A. b \leq m \longrightarrow m = b$
using m -is-min **by** (*auto simp: le-less*)
ultimately show *?thesis*
by *auto*
qed

lemma *finite-has-minimal2*:

$\llbracket \text{finite } A; a \in A \rrbracket \implies \exists m \in A. m \leq a \wedge (\forall b \in A. b \leq m \longrightarrow m = b)$
using *finite-has-minimal*[*of* $\{b \in A. b \leq a\}$] **by** *fastforce*

end

20.8.2 Relating injectivity and surjectivity

lemma *finite-surj-inj*:

assumes *finite* A $A \subseteq f' A$
shows *inj-on* $f A$

proof –

have $f' A = A$

by (*rule card-seteq* [*THEN sym*]) (*auto simp add: assms card-image-le*)

then show *?thesis* **using** *assms*

by (*simp add: eq-card-imp-inj-on*)

qed

lemma *finite-UNIV-surj-inj*: *finite*(*UNIV*:: 'a set) \implies *surj* $f \implies$ *inj* f

for $f :: 'a \Rightarrow 'a$

by (*blast intro: finite-surj-inj subset-UNIV*)

lemma *finite-UNIV-inj-surj*: *finite*(*UNIV*:: 'a set) \implies *inj* $f \implies$ *surj* f

for $f :: 'a \Rightarrow 'a$

by (*fastforce simp:surj-def dest!: endo-inj-surj*)

lemma *surjective-iff-injective-gen*:

assumes fS : *finite* S

and fT : *finite* T

and c : *card* $S = \text{card } T$

and ST : $f' S \subseteq T$

shows $(\forall y \in T. \exists x \in S. f x = y) \longleftrightarrow$ *inj-on* $f S$

(*is ?lhs* \longleftrightarrow *?rhs*)

proof

assume h : *?lhs*

{

fix $x y$

assume x : $x \in S$

assume y : $y \in S$

assume f : $f x = f y$

from $x fS$ **have** $S0$: *card* $S \neq 0$

```

    by auto
  have  $x = y$ 
  proof (rule ccontr)
    assume  $xy: \neg ?thesis$ 
    have  $th: \text{card } S \leq \text{card } (f '(S - \{y\}))$ 
      unfolding  $c$ 
    proof (rule card-mono)
      show  $\text{finite } (f '(S - \{y\}))$ 
        by (simp add:  $fS$ )
      have  $\llbracket x \neq y; x \in S; z \in S; f x = f y \rrbracket$ 
         $\implies \exists x \in S. x \neq y \wedge f z = f x$  for  $z$ 
        by (cases  $z = y \longrightarrow z = x$ ) auto
      then show  $T \subseteq f '(S - \{y\})$ 
        using  $h xy x y f$  by fastforce
    qed
    also have  $\dots \leq \text{card } (S - \{y\})$ 
      by (simp add: card-image-le  $fS$ )
    also have  $\dots \leq \text{card } S - 1$  using  $y fS$  by simp
    finally show  $False$  using  $S0$  by arith
  qed
}
then show  $?rhs$ 
  unfolding  $inj\text{-on}\text{-def}$  by blast
next
  assume  $h: ?rhs$ 
  have  $f ' S = T$ 
    by (simp add:  $ST c$  card-image card-subset-eq  $fT h$ )
  then show  $?lhs$  by blast
qed

hide-const (open)  $Finite\text{-Set.fold}$ 

```

20.9 Infinite Sets

Some elementary facts about infinite sets, mostly by Stephan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation $infinite :: 'a \text{ set} \Rightarrow \text{bool}$
 where $infinite S \equiv \neg finite S$

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma $infinite\text{-UNIV-nat}$ [*iff*]: $infinite (UNIV :: \text{nat set})$
proof
 assume $finite (UNIV :: \text{nat set})$
 with $finite\text{-UNIV-inj-surj}$ [*of Suc*] show $False$
 by $simp$ ($blast dest: Suc\text{-neq}\text{-Zero surjD}$)
qed

lemma *infinite-UNIV-char-0*: *infinite* (*UNIV* :: 'a::semiring-char-0 set)

proof

assume *finite* (*UNIV* :: 'a set)

with *subset-UNIV* **have** *finite* (*range of-nat* :: 'a set)

by (*rule finite-subset*)

moreover **have** *inj* (*of-nat* :: *nat* \Rightarrow 'a)

by (*simp add: inj-on-def*)

ultimately **have** *finite* (*UNIV* :: *nat* set)

by (*rule finite-imageD*)

then **show** *False*

by *simp*

qed

lemma *infinite-imp-nonempty*: *infinite* *S* \implies *S* \neq {}

by *auto*

lemma *infinite-remove*: *infinite* *S* \implies *infinite* (*S* - {*a*})

by *simp*

lemma *Diff-infinite-finite*:

assumes *finite* *T* *infinite* *S*

shows *infinite* (*S* - *T*)

using \langle *finite* *T* \rangle

proof *induct*

from \langle *infinite* *S* \rangle **show** *infinite* (*S* - {})

by *auto*

next

fix *T* *x*

assume *ih*: *infinite* (*S* - *T*)

have *S* - (*insert* *x* *T*) = (*S* - *T*) - {*x*}

by (*rule Diff-insert*)

with *ih* **show** *infinite* (*S* - (*insert* *x* *T*))

by (*simp add: infinite-remove*)

qed

lemma *Un-infinite*: *infinite* *S* \implies *infinite* (*S* \cup *T*)

by *simp*

lemma *infinite-Un*: *infinite* (*S* \cup *T*) \iff *infinite* *S* \vee *infinite* *T*

by *simp*

lemma *infinite-super*:

assumes *S* \subseteq *T*

and *infinite* *S*

shows *infinite* *T*

proof

assume *finite* *T*

with \langle *S* \subseteq *T* \rangle **have** *finite* *S* **by** (*simp add: finite-subset*)

with $\langle \text{infinite } S \rangle$ **show** *False* **by** *simp*
qed

proposition *infinite-coinduct* [*consumes 1, case-names infinite*]:

assumes $X A$
and step: $\bigwedge A. X A \implies \exists x \in A. X (A - \{x\}) \vee \text{infinite } (A - \{x\})$
shows *infinite A*

proof

assume *finite A*
then show *False*
using $\langle X A \rangle$
proof (*induction rule: finite-psubset-induct*)
case (*psubset A*)
then obtain x **where** $x \in A$ $X (A - \{x\}) \vee \text{infinite } (A - \{x\})$
using *local.step psubset.prem*s **by** *blast*
then have $X (A - \{x\})$
using *psubset.hyps* **by** *blast*
show *False*
proof (*rule psubset.IH* [**where** $B = A - \{x\}$])
show $A - \{x\} \subset A$
using $\langle x \in A \rangle$ **by** *blast*
qed *fact*
qed
qed

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

lemma *inf-img-fin-dom'*:

assumes *img: finite* ($f' A$)
and *dom: infinite* A
shows $\exists y \in f' A. \text{infinite } (f - \{y\} \cap A)$
proof (*rule ccontr*)
have $A \subseteq (\bigcup y \in f' A. f - \{y\} \cap A)$ **by** *auto*
moreover assume $\neg ?thesis$
with *img* **have** *finite* $(\bigcup y \in f' A. f - \{y\} \cap A)$ **by** *blast*
ultimately have *finite A* **by** (*rule finite-subset*)
with *dom* **show** *False* **by** *contradiction*
qed

lemma *inf-img-fin-domE'*:

assumes *finite* ($f' A$) **and** *infinite* A
obtains y **where** $y \in f' A$ **and** *infinite* $(f - \{y\} \cap A)$
using *assms* **by** (*blast dest: inf-img-fin-dom'*)

lemma *inf-img-fin-dom*:

assumes *img: finite* ($f' A$) **and** *dom: infinite* A
shows $\exists y \in f' A. \text{infinite } (f - \{y\})$

using *inf-img-fin-dom* [OF *assms*] by *auto*

lemma *inf-img-fin-domE*:
 assumes *finite* ($f'A$) and *infinite* A
 obtains y where $y \in f'A$ and *infinite* ($f - \{y\}$)
 using *assms* by (*blast dest: inf-img-fin-dom*)

proposition *finite-image-absD*: *finite* ($abs \ 'S$) \implies *finite* S
 for $S :: 'a::linordered-ring\ set$
 by (*rule ccontr*) (*auto simp: abs-eq-iff vimage-def dest: inf-img-fin-dom*)

20.10 The finite powerset operator

definition *Fpow* :: $'a\ set \Rightarrow 'a\ set\ set$
 where $Fpow\ A \equiv \{X. X \subseteq A \wedge finite\ X\}$

lemma *Fpow-mono*: $A \subseteq B \implies Fpow\ A \subseteq Fpow\ B$
 unfolding *Fpow-def* by *auto*

lemma *empty-in-Fpow*: $\{\} \in Fpow\ A$
 unfolding *Fpow-def* by *auto*

lemma *Fpow-not-empty*: $Fpow\ A \neq \{\}$
 using *empty-in-Fpow* by *blast*

lemma *Fpow-subset-Pow*: $Fpow\ A \subseteq Pow\ A$
 unfolding *Fpow-def* by *auto*

lemma *Fpow-Pow-finite*: $Fpow\ A = Pow\ A\ Int\ \{A. finite\ A\}$
 unfolding *Fpow-def Pow-def* by *blast*

lemma *inj-on-image-Fpow*:
 assumes *inj-on* $f\ A$
 shows *inj-on* (*image* f) ($Fpow\ A$)
 using *assms Fpow-subset-Pow* [of A] *subset-inj-on* [of *image* $f\ Pow\ A$]
inj-on-image-Pow by *blast*

lemma *image-Fpow-mono*:
 assumes $f \ 'A \subseteq B$
 shows (*image* f) ' $(Fpow\ A) \subseteq Fpow\ B$
 using *assms* by (*unfold Fpow-def, auto*)

end

21 Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
 imports *Finite-Set*
 abbrevs $\hat{*} = * **$

```

and  $\widehat{+} = + ++$ 
and  $\widehat{=} = = ==$ 
begin

```

ML-file $\langle \sim\sim / \text{src} / \text{Provers} / \text{trancl.ML} \rangle$

rtrancl is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

```

context notes  $[[\text{inductive-internals}]]$ 
begin

```

inductive-set *rtrancl* :: ('a × 'a) set ⇒ ('a × 'a) set ((-*) [1000] 999)

for *r* :: ('a × 'a) set

where

rtrancl-refl [*intro!*, *Pure.intro!*, *simp*]: (a, a) ∈ *r**

| *rtrancl-into-rtrancl* [*Pure.intro*]: (a, b) ∈ *r** ⇒ (b, c) ∈ *r* ⇒ (a, c) ∈ *r**

inductive-set *trancl* :: ('a × 'a) set ⇒ ('a × 'a) set ((-+) [1000] 999)

for *r* :: ('a × 'a) set

where

r-into-trancl [*intro*, *Pure.intro*]: (a, b) ∈ *r* ⇒ (a, b) ∈ *r*⁺

| *trancl-into-trancl* [*Pure.intro*]: (a, b) ∈ *r*⁺ ⇒ (b, c) ∈ *r* ⇒ (a, c) ∈ *r*⁺

notation

rtranclp ((-**) [1000] 1000) **and**

tranclp ((-++) [1000] 1000)

declare

rtrancl-def [*nitpick-unfold del*]

rtranclp-def [*nitpick-unfold del*]

trancl-def [*nitpick-unfold del*]

tranclp-def [*nitpick-unfold del*]

end

abbreviation *reflcl* :: ('a × 'a) set ⇒ ('a × 'a) set ((-⁼) [1000] 999)

where *r⁼* ≡ *r* ∪ *Id*

abbreviation *reflclp* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool ((-⁼⁼) [1000] 1000)

where *r⁼⁼* ≡ *sup r* (=)

notation (*ASCII*)

rtrancl ((-[∗]) [1000] 999) **and**

trancl ((-⁺) [1000] 999) **and**

reflcl ((-⁼) [1000] 999) **and**

rtranclp ((-^{∗∗}) [1000] 1000) **and**

tranclp ((-⁺⁺) [1000] 1000) **and**

reflclp $((-\hat{=}))$ [1000] 1000)

21.1 Reflexive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: $(\sup (\lambda x y. (x, y) \in r) (=)) = (\lambda x y. (x, y) \in r \cup Id)$

by (*auto simp: fun-eq-iff*)

lemma *refl-reflcl*[*simp*]: *refl* $(r^=)$

by (*simp add: refl-on-def*)

lemma *reflp-on-reflclp*[*simp*]: *reflp-on* $A R^{==}$

by (*simp add: reflp-on-def*)

lemma *antisym-on-reflcl*[*simp*]: *antisym-on* $A (r^=) \longleftrightarrow$ *antisym-on* $A R$

by (*simp add: antisym-on-def*)

lemma *antisym-on-reflcp*[*simp*]: *antisym-on* $A R^{==} \longleftrightarrow$ *antisym-on* $A R$

by (*rule antisym-on-reflcl[to-pred]*)

lemma *trans-on-reflcl*[*simp*]: *trans-on* $A r \Longrightarrow$ *trans-on* $A (r^=)$

by (*auto intro: trans-onI dest: trans-onD*)

lemma *transp-on-reflclp*[*simp*]: *transp-on* $A R \Longrightarrow$ *transp-on* $A R^{==}$

by (*rule trans-on-reflcl[to-pred]*)

lemma *reflclp-idemp* [*simp*]: $(P^{==})^{==} = P^{==}$

by *blast*

lemma *reflclp-ident-if-reflp*[*simp*]: *reflp* $R \Longrightarrow R^{==} = R$

by (*auto dest: reflpD*)

21.2 Reflexive-transitive closure

lemma *r-into-rtrancl* [*intro*]: $\bigwedge p. p \in r \Longrightarrow p \in r^*$

— *rtrancl* of r contains r

by (*simp add: split-tupled-all rtrancl-refl [THEN rtrancl-into-rtrancl]*)

lemma *r-into-rtranclp* [*intro*]: $r x y \Longrightarrow r^{**} x y$

— *rtrancl* of r contains r

by (*erule rtranclp.rtrancl-refl [THEN rtranclp.rtrancl-into-rtrancl]*)

lemma *rtranclp-mono*: $r \leq s \Longrightarrow r^{**} \leq s^{**}$

— monotonicity of *rtrancl*

proof (*rule predicate2I*)

show $s^{**} x y$ **if** $r \leq s$ $r^{**} x y$ **for** $x y$

using $\langle r^{**} x y \rangle \langle r \leq s \rangle$

by (*induction rule: rtranclp.induct*) (*blast intro: rtranclp.rtrancl-into-rtrancl*)+

qed

lemma *mono-rtranclp*[*mono*]: $(\bigwedge a b. x a b \longrightarrow y a b) \Longrightarrow x^{**} a b \longrightarrow y^{**} a b$
using *rtranclp-mono*[*of x y*] **by** *auto*

lemmas *rtrancl-mono* = *rtranclp-mono* [*to-set*]

theorem *rtranclp-induct* [*consumes 1, case-names base step, induct set: rtranclp*]:
assumes *a*: $r^{**} a b$
and cases: $P a \bigwedge y z. r^{**} a y \Longrightarrow r y z \Longrightarrow P y \Longrightarrow P z$
shows $P b$
using *a* **by** (*induct x≡a b*) (*rule cases*)+

lemmas *rtrancl-induct* [*induct set: rtrancl*] = *rtranclp-induct* [*to-set*]

lemmas *rtranclp-induct2* =
rtranclp-induct[*of - (ax,ay) (bx,by), split-rule, consumes 1, case-names refl step*]

lemmas *rtrancl-induct2* =
rtrancl-induct[*of (ax,ay) (bx,by), split-format (complete), consumes 1, case-names refl step*]

lemma *refl-rtrancl*: *refl* (r^*)
unfolding *refl-on-def* **by** *fast*

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* (r^*)

proof (*rule transI*)

fix *x y z*

assume $(x, y) \in r^*$

assume $(y, z) \in r^*$

then show $(x, z) \in r^*$

proof *induct*

case *base*

show $(x, y) \in r^*$ **by** *fact*

next

case (*step u v*)

from $\langle (x, u) \in r^* \rangle$ **and** $\langle (u, v) \in r \rangle$

show $(x, v) \in r^*$ **..**

qed

qed

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD*]

lemma *rtranclp-trans*:

assumes $r^{**} x y$

and $r^{**} y z$

shows $r^{**} x z$

using *assms(2,1)* **by** *induct iprover*+

lemma *rtranclE* [*cases set: rtrancl*]:

fixes $a b :: 'a$
assumes $major: (a, b) \in r^*$
obtains
 (base) $a = b$
 | (step) y **where** $(a, y) \in r^*$ **and** $(y, b) \in r$
 — elimination of $rtrancl$ – by induction on a special formula
proof –
have $a = b \vee (\exists y. (a, y) \in r^* \wedge (y, b) \in r)$
by (rule major [THEN rtrancl-induct]) blast+
then show ?thesis
by (auto intro: base step)
qed

lemma $rtrancl\text{-}Int\text{-}subset: Id \subseteq s \implies (r^* \cap s) \circ r \subseteq s \implies r^* \subseteq s$
by (fastforce elim: rtrancl-induct)

lemma $converse\text{-}rtranclp\text{-}into\text{-}rtranclp: r a b \implies r^{**} b c \implies r^{**} a c$
by (rule rtranclp-trans) iprover+

lemmas $converse\text{-}rtrancl\text{-}into\text{-}rtrancl = converse\text{-}rtranclp\text{-}into\text{-}rtranclp$ [to-set]

More r^* equations and inclusions.

lemma $rtranclp\text{-}idemp$ [simp]: $(r^{**})^{**} = r^{**}$

proof –
have $r^{****} x y \implies r^{**} x y$ **for** $x y$
by (induction rule: rtranclp-induct) (blast intro: rtranclp-trans)+
then show ?thesis
by (auto intro!: order-antisym)
qed

lemmas $rtrancl\text{-}idemp$ [simp] = $rtranclp\text{-}idemp$ [to-set]

lemma $rtrancl\text{-}idemp\text{-}self\text{-}comp$ [simp]: $R^* \circ R^* = R^*$
by (force intro: rtrancl-trans)

lemma $rtrancl\text{-}subset\text{-}rtrancl: r \subseteq s^* \implies r^* \subseteq s^*$
by (drule rtrancl-mono, simp)

lemma $rtranclp\text{-}subset: R \leq S \implies S \leq R^{**} \implies S^{**} = R^{**}$
by (fastforce dest: rtranclp-mono)

lemmas $rtrancl\text{-}subset = rtranclp\text{-}subset$ [to-set]

lemma $rtranclp\text{-}sup\text{-}rtranclp: (sup (R^{**}) (S^{**}))^{**} = (sup R S)^{**}$
by (blast intro!: rtranclp-subset intro: rtranclp-mono [THEN predicate2D])

lemmas $rtrancl\text{-}Un\text{-}rtrancl = rtranclp\text{-}sup\text{-}rtranclp$ [to-set]

lemma $rtranclp\text{-}reflclp$ [simp]: $(R^{==})^{**} = R^{**}$

by (*blast intro!*: *rtranclp-subset*)

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - \text{Id})^* = r^*$
 by (*rule rtrancl-subset* [*symmetric*]) *auto*

lemma *rtranclp-r-diff-Id*: $(\text{inf } r (\neq))^{**} = r^{**}$
 by (*rule rtranclp-subset* [*symmetric*]) *auto*

theorem *rtranclp-converseD*:
 assumes $(r^{-1-1})^{**} x y$
 shows $r^{**} y x$
 using *assms* by *induct* (*iprover intro: rtranclp-trans dest!: conversepD*)+

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

theorem *rtranclp-converseI*:
 assumes $r^{**} y x$
 shows $(r^{-1-1})^{**} x y$
 using *assms* by *induct* (*iprover intro: rtranclp-trans conversepI*)+

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

lemma *rtrancl-converse*: $(r^{-1})^* = (r^*)^{-1}$
 by (*fast dest!: rtrancl-converseD intro!: rtrancl-converseI*)

lemma *sym-rtrancl*: $\text{sym } r \implies \text{sym } (r^*)$
 by (*simp only: sym-conv-converse-eq rtrancl-converse* [*symmetric*])

theorem *converse-rtranclp-induct* [*consumes 1, case-names base step*]:
 assumes *major*: $r^{**} a b$
 and *cases*: $P b \wedge y z. r y z \implies r^{**} z b \implies P z \implies P y$
 shows $P a$
 using *rtranclp-converseI* [*OF major*]
 by *induct* (*iprover intro: cases dest!: conversepD rtranclp-converseD*)+

lemmas *converse-rtrancl-induct* = *converse-rtranclp-induct* [*to-set*]

lemmas *converse-rtranclp-induct2* =
converse-rtranclp-induct [*of - (ax, ay) (bx, by), split-rule, consumes 1, case-names refl step*]

lemmas *converse-rtrancl-induct2* =
converse-rtrancl-induct [*of (ax, ay) (bx, by), split-format (complete), consumes 1, case-names refl step*]

lemma *converse-rtranclpE* [*consumes 1, case-names base step*]:
 assumes *major*: $r^{**} x z$

and cases: $x = z \implies P \wedge y. r x y \implies r^{**} y z \implies P$
shows P

proof –

have $x = z \vee (\exists y. r x y \wedge r^{**} y z)$

by (rule major [THEN converse-rtranclp-induct]) iprover+

then show ?thesis

by (auto intro: cases)

qed

lemmas converse-rtranclE = converse-rtranclpE [to-set]

lemmas converse-rtranclpE2 = converse-rtranclpE [of - (xa,xb) (za,zb), split-rule]

lemmas converse-rtranclE2 = converse-rtranclE [of (xa,xb) (za,zb), split-rule]

lemma r-comp-rtrancl-eq: $r \circ r^* = r^* \circ r$

by (blast elim: rtranclE converse-rtranclE

intro: rtrancl-into-rtrancl converse-rtrancl-into-rtrancl)

lemma rtrancl-unfold: $r^* = Id \cup r^* \circ r$

by (auto intro: rtrancl-into-rtrancl elim: rtranclE)

lemma rtrancl-Un-separatorE:

$(a, b) \in (P \cup Q)^* \implies \forall x y. (a, x) \in P^* \longrightarrow (x, y) \in Q \longrightarrow x = y \implies (a, b) \in P^*$

proof (induct rule: rtrancl.induct)

case rtrancl-refl

then show ?case **by** blast

next

case rtrancl-into-rtrancl

then show ?case **by** (blast intro: rtrancl-trans)

qed

lemma rtrancl-Un-separator-converseE:

$(a, b) \in (P \cup Q)^* \implies \forall x y. (x, b) \in P^* \longrightarrow (y, x) \in Q \longrightarrow y = x \implies (a, b) \in P^*$

proof (induct rule: converse-rtrancl-induct)

case base

then show ?case **by** blast

next

case step

then show ?case **by** (blast intro: rtrancl-trans)

qed

lemma Image-closed-trancl:

assumes $r \text{ “ } X \subseteq X$

shows $r^* \text{ “ } X = X$

proof –

from *assms* **have** **: $\{y. \exists x \in X. (x, y) \in r\} \subseteq X$

```

  by auto
  have  $x \in X$  if 1:  $(y, x) \in r^*$  and 2:  $y \in X$  for  $x y$ 
  proof -
    from 1 show  $x \in X$ 
    proof induct
      case base
        show ?case by (fact 2)
      next
        case step
          with ** show ?case by auto
    qed
  qed
  then show ?thesis by auto
  qed

```

21.3 Transitive closure

lemma *totalp-on-tranclp*: $\text{totalp-on } A \ R \Longrightarrow \text{totalp-on } A \ (\text{tranclp } R)$
 by (*auto intro: totalp-onI dest: totalp-onD*)

lemma *total-on-trancl*: $\text{total-on } A \ r \Longrightarrow \text{total-on } A \ (\text{trancl } r)$
 by (*rule totalp-on-tranclp[to-set]*)

lemma *trancl-mono*:
 assumes $p \in r^+ \ r \subseteq s$
 shows $p \in s^+$
 proof -
 have $\llbracket (a, b) \in r^+; r \subseteq s \rrbracket \Longrightarrow (a, b) \in s^+$ for $a b$
 by (*induction rule: trancl.induct*) (*iprover dest: subsetD*)+
 with *assms* show ?thesis
 by (*cases p*) *force*
 qed

lemma *r-into-trancl'*: $\bigwedge p. p \in r \Longrightarrow p \in r^+$
 by (*simp only: split-tupled-all*) (*erule r-into-trancl*)

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: $r^{++} \ a \ b \Longrightarrow r^{**} \ a \ b$
 by (*erule tranclp.induct*) *iprover+*

lemmas *trancl-into-rtrancl* = *tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*:
 assumes $r^{**} \ a \ b$
 shows $r \ b \ c \Longrightarrow r^{++} \ a \ c$
 using *assms* by (*induct arbitrary: c*) *iprover+*

lemmas *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*:
assumes $r\ a\ b\ r^{**}\ b\ c$ **shows** $r^{++}\ a\ c$
— intro rule from r and *rtrancl*
using $\langle r^{**}\ b\ c \rangle$
proof (*cases rule: rtranclp.cases*)
case *rtrancl-refl*
with *assms show ?thesis*
by *iprover*
next
case *rtrancl-into-rtrancl*
with *assms show ?thesis*
by (*auto intro: rtranclp-trans [THEN rtranclp-into-tranclp1]*)
qed

lemmas *rtrancl-into-trancl2 = rtranclp-into-tranclp2 [to-set]*

Nice induction rule for *trancl*

lemma *tranclp-induct* [*consumes 1, case-names base step, induct pred: tranclp*]:
assumes $a: r^{++}\ a\ b$
and *cases*: $\bigwedge y. r\ a\ y \implies P\ y \bigwedge y\ z. r^{++}\ a\ y \implies r\ y\ z \implies P\ y \implies P\ z$
shows $P\ b$
using a **by** (*induct $x \equiv a\ b$ (iprover intro: cases)+*)

lemmas *trancl-induct [induct set: trancl] = tranclp-induct [to-set]*

lemmas *tranclp-induct2 =*
tranclp-induct [of - (ax, ay) (bx, by), split-rule, consumes 1, case-names base step]

lemmas *trancl-induct2 =*
trancl-induct [of (ax, ay) (bx, by), split-format (complete), consumes 1, case-names base step]

lemma *tranclp-trans-induct*:
assumes *major*: $r^{++}\ x\ y$
and *cases*: $\bigwedge x\ y. r\ x\ y \implies P\ x\ y \bigwedge x\ y\ z. r^{++}\ x\ y \implies P\ x\ y \implies r^{++}\ y\ z \implies P\ y\ z \implies P\ x\ z$
shows $P\ x\ y$
— Another induction rule for *trancl*, incorporating transitivity
by (*iprover intro: major [THEN tranclp-induct] cases*)

lemmas *trancl-trans-induct = tranclp-trans-induct [to-set]*

lemma *tranclE* [*cases set: trancl*]:
assumes $(a, b) \in r^+$
obtains
(base) $(a, b) \in r$
| *(step)* c **where** $(a, c) \in r^+$ **and** $(c, b) \in r$
using *assms by cases simp-all*

lemma *trancl-Int-subset*: $r \subseteq s \implies (r^+ \cap s) \subseteq r \subseteq s \implies r^+ \subseteq s$
by (*fastforce simp add: elim: trancl-induct*)

lemma *trancl-unfold*: $r^+ = r \cup r^+ \subseteq r$
by (*auto intro: trancl-into-trancl elim: tranclE*)

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* (r^+)

proof (*rule transI*)

fix $x\ y\ z$

assume $(x, y) \in r^+$

assume $(y, z) \in r^+$

then show $(x, z) \in r^+$

proof *induct*

case (*base* u)

from $\langle (x, y) \in r^+ \rangle$ **and** $\langle (y, u) \in r \rangle$

show $(x, u) \in r^+ \dots$

next

case (*step* $u\ v$)

from $\langle (x, u) \in r^+ \rangle$ **and** $\langle (u, v) \in r \rangle$

show $(x, v) \in r^+ \dots$

qed

qed

lemmas *trancl-trans = trans-trancl* [*THEN transD*]

lemma *tranclp-trans*:

assumes $r^{++}\ x\ y$

and $r^{++}\ y\ z$

shows $r^{++}\ x\ z$

using *assms(2,1)* **by** *induct iprover+*

lemma *trancl-id* [*simp*]: *trans* $r \implies r^+ = r$

unfolding *trans-def* **by** (*fastforce simp add: elim: trancl-induct*)

lemma *rtranclp-tranclp-tranclp*:

assumes $r^{**}\ x\ y$

shows $\bigwedge z. r^{++}\ y\ z \implies r^{++}\ x\ z$

using *assms* **by** *induct (iprover intro: tranclp-trans)+*

lemmas *rtrancl-trancl-trancl = rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: $r\ a\ b \implies r^{++}\ b\ c \implies r^{++}\ a\ c$

by (*erule tranclp-trans* [*OF tranclp.r-into-trancl*])

lemmas *trancl-into-trancl2 = tranclp-into-tranclp2* [*to-set*]

lemma *tranclp-converseI*:


```

assumes  $(r^{++})^{-1-1} x y$  shows  $(r^{-1-1})^{++} x y$ 
using conversepD [OF assms]
proof (induction rule: tranclp-induct)
  case (base y)
  then show ?case
    by (iprover intro: conversepI)
next
  case (step y z)
  then show ?case
    by (iprover intro: conversepI tranclp-trans)
qed

```

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

```

lemma tranclp-converseD:
  assumes  $(r^{-1-1})^{++} x y$  shows  $(r^{++})^{-1-1} x y$ 
proof –
  have  $r^{++} y x$ 
    using assms
    by (induction rule: tranclp-induct) (iprover dest: conversepD intro: tranclp-trans)+
  then show ?thesis
    by (rule conversepI)
qed

```

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

```

lemma tranclp-converse:  $(r^{-1-1})^{++} = (r^{++})^{-1-1}$ 
by (fastforce simp add: fun-eq-iff intro!: tranclp-converseI dest!: tranclp-converseD)

```

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

```

lemma sym-trancl:  $\text{sym } r \implies \text{sym } (r^+)$ 
by (simp only: sym-conv-converse-eq trancl-converse [symmetric])

```

```

lemma converse-tranclp-induct [consumes 1, case-names base step]:
  assumes major:  $r^{++} a b$ 
  and cases:  $\bigwedge y. r y b \implies P y \bigwedge y z. r y z \implies r^{++} z b \implies P z \implies P y$ 
  shows  $P a$ 
proof –
  have  $r^{-1-1++} b a$ 
    by (intro tranclp-converseI conversepI major)
  then show ?thesis
    by (induction rule: tranclp-induct) (blast intro: cases dest: tranclp-converseD)+
qed

```

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

```

lemma tranclpD:  $R^{++} x y \implies \exists z. R x z \wedge R^{**} z y$ 
proof (induction rule: converse-tranclp-induct)

```

```

  case (step u v)
  then show ?case
    by (blast intro: rtranclp-trans)
qed auto

```

lemmas tranclD = tranclpD [to-set]

```

lemma converse-tranclpE:
  assumes major: tranclp r x z
  and base: r x z  $\implies$  P
  and step:  $\bigwedge y. r x y \implies \text{tranclp } r y z \implies P$ 
  shows P
proof -
  from tranclpD [OF major] obtain y where r x y and rtranclp r y z
  by iprover
  from this(2) show P
  proof (cases rule: rtranclp.cases)
  case rtrancl-refl
  with <r x y> base show P
  by iprover
  next
  case rtrancl-into-rtrancl
  then have tranclp r y z
  by (iprover intro: rtranclp-into-tranclp1)
  with <r x y> step show P
  by iprover
  qed
qed

```

lemmas converse-tranclE = converse-tranclpE [to-set]

```

lemma tranclD2:  $(x, y) \in R^+ \implies \exists z. (x, z) \in R^* \wedge (z, y) \in R$ 
  by (blast elim: tranclE intro: trancl-into-rtrancl)

```

```

lemma irrefl-tranclI:  $r^{-1} \cap r^* = \{\}$   $\implies (x, x) \notin r^+$ 
  by (blast elim: tranclE dest: trancl-into-rtrancl)

```

```

lemma irrefl-trancl-rD:  $\forall x. (x, x) \notin r^+ \implies (x, y) \in r \implies x \neq y$ 
  by (blast dest: r-into-trancl)

```

```

lemma trancl-subset-Sigma-aux:  $(a, b) \in r^* \implies r \subseteq A \times A \implies a = b \vee a \in A$ 
  by (induct rule: rtrancl-induct) auto

```

```

lemma trancl-subset-Sigma:
  assumes  $r \subseteq A \times A$  shows  $r^+ \subseteq A \times A$ 
proof (rule trancl-Int-subset [OF assms])
  show  $(r^+ \cap A \times A) \subseteq A \times A$ 
  using assms by auto
qed

```

lemma *reflclp-tranclp* [*simp*]: $(r^{++})^{==} = r^{**}$
 by (*fast elim: rtranclp.cases tranclp-into-rtranclp dest: rtranclp-into-tranclp1*)

lemmas *reflcl-trancl* [*simp*] = *reflclp-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r^{-})^{+} = r^{*}$
proof –
 have $(a, b) \in (r^{-})^{+} \implies (a, b) \in r^{*}$ **for** $a\ b$
 by (*force dest: trancl-into-rtrancl*)
 moreover have $(a, b) \in (r^{-})^{+}$ **if** $(a, b) \in r^{*}$ **for** $a\ b$
 using *that*
proof (*cases a b rule: rtranclE*)
 case *step*
 show *?thesis*
 by (*rule rtrancl-into-trancl1*) (*use step in auto*)
qed *auto*
 ultimately show *?thesis*
 by *auto*
qed

lemma *rtrancl-trancl-reflcl* [*code*]: $r^{*} = (r^{+})^{=}$
 by *simp*

lemma *trancl-empty* [*simp*]: $\{\}^{+} = \{\}$
 by (*auto elim: trancl-induct*)

lemma *rtrancl-empty* [*simp*]: $\{\}^{*} = Id$
 by (*rule subst [OF reflcl-trancl]*) *simp*

lemma *rtranclpD*: $R^{**}\ a\ b \implies a = b \vee a \neq b \wedge R^{++}\ a\ b$
 by (*force simp: reflclp-tranclp [symmetric] simp del: reflclp-tranclp*)

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*: $(x,y) \in R^{*} \iff x = y \vee x \neq y \wedge (x, y) \in R^{+}$
 by (*fast elim: trancl-into-rtrancl dest: rtranclD*)

lemma *trancl-unfold-right*: $r^{+} = r^{*}\ O\ r$
 by (*auto dest: tranclD2 intro: rtrancl-into-trancl1*)

lemma *trancl-unfold-left*: $r^{+} = r\ O\ r^{*}$
 by (*auto dest: tranclD intro: rtrancl-into-trancl2*)

lemma *trancl-insert*: $(insert\ (y, x)\ r)^{+} = r^{+} \cup \{(a, b). (a, y) \in r^{*} \wedge (x, b) \in r^{*}\}$
 — primitive recursion for *trancl* over finite relations
proof –
 have $\bigwedge a\ b. (a, b) \in (insert\ (y, x)\ r)^{+} \implies$
 $(a, b) \in r^{+} \cup \{(a, b). (a, y) \in r^{*} \wedge (x, b) \in r^{*}\}$

by (*erule trancl-induct*) (*blast intro: rtrancl-into-trancl1 trancl-into-rtrancl trancl-trans*)⁺
moreover have $r^+ \cup \{(a, b). (a, y) \in r^* \wedge (x, b) \in r^*\} \subseteq (\text{insert } (y, x) r)^+$
 by (*blast intro: trancl-mono rtrancl-mono [THEN [2] rev-subsetD]*)
 rtrancl-trancl-trancl rtrancl-into-trancl2)
ultimately show *?thesis*
 by *auto*
qed

lemma *trancl-insert2*:

$(\text{insert } (a, b) r)^+ = r^+ \cup \{(x, y). ((x, a) \in r^+ \vee x = a) \wedge ((b, y) \in r^+ \vee y = b)\}$
 by (*auto simp: trancl-insert rtrancl-eq-or-trancl*)

lemma *rtrancl-insert*: $(\text{insert } (a, b) r)^* = r^* \cup \{(x, y). (x, a) \in r^* \wedge (b, y) \in r^*\}$
 using *trancl-insert[of a b r]*
 by (*simp add: rtrancl-trancl-reflcl del: reflcl-trancl*) *blast*

Simplifying nested closures

lemma *rtrancl-trancl-absorb[simp]*: $(R^*)^+ = R^*$
 by (*simp add: trans-rtrancl*)

lemma *trancl-rtrancl-absorb[simp]*: $(R^+)^* = R^*$
 by (*subst reflcl-trancl[symmetric]*) *simp*

lemma *rtrancl-reflcl-absorb[simp]*: $(R^*)^= = R^*$
 by *auto*

Domain and Range

lemma *Domain-rtrancl [simp]*: $\text{Domain } (R^*) = \text{UNIV}$
 by *blast*

lemma *Range-rtrancl [simp]*: $\text{Range } (R^*) = \text{UNIV}$
 by *blast*

lemma *rtrancl-Un-subset*: $(R^* \cup S^*) \subseteq (R \cup S)^*$
 by (*rule rtrancl-Un-rtrancl [THEN subst]*) *fast*

lemma *in-rtrancl-UnI*: $x \in R^* \vee x \in S^* \implies x \in (R \cup S)^*$
 by (*blast intro: subsetD [OF rtrancl-Un-subset]*)

lemma *trancl-domain [simp]*: $\text{Domain } (r^+) = \text{Domain } r$
 by (*unfold Domain-unfold*) (*blast dest: tranclD*)

lemma *trancl-range [simp]*: $\text{Range } (r^+) = \text{Range } r$
 unfolding *Domain-converse [symmetric]* by (*simp add: trancl-converse [symmetric]*)

lemma *Not-Domain-rtrancl*:

assumes $x \notin \text{Domain } R$ **shows** $(x, y) \in R^* \longleftrightarrow x = y$

```

proof –
have  $(x, y) \in R^* \implies x = y$ 
  by (erule rtrancl-induct) (use assms in auto)
  then show ?thesis
    by auto
qed

```

```

lemma trancl-subset-Field2:  $r^+ \subseteq \text{Field } r \times \text{Field } r$ 
  by (rule trancl-Int-subset) (auto simp: Field-def)

```

```

lemma finite-trancl[simp]:  $\text{finite } (r^+) = \text{finite } r$ 
proof
  show  $\text{finite } (r^+) \implies \text{finite } r$ 
    by (blast intro: r-into-trancl' finite-subset)
  show  $\text{finite } r \implies \text{finite } (r^+)$ 
    by (auto simp: finite-Field trancl-subset-Field2 [THEN finite-subset])
qed

```

```

lemma finite-rtrancl-Image[simp]: assumes finite R finite A shows  $\text{finite } (R^* \text{ `` } A)$ 
proof (rule ccontr)
  assume infinite (R^* `` A)
  with assms show False
    by(simp add: rtrancl-trancl-reflcl Un-Image del: reflcl-trancl)
qed

```

More about converse *rtrancl* and *trancl*, should be merged with main body.

```

lemma single-valued-confluent:
  assumes single-valued r and xy: (x, y) ∈ r* and xz: (x, z) ∈ r*
  shows  $(y, z) \in r^* \vee (z, y) \in r^*$ 
  using xy
proof (induction rule: rtrancl-induct)
  case base
  show ?case
    by (simp add: assms)
next
  case (step y z)
  with xz  $\langle \text{single-valued } r \rangle$  show ?case
    by (auto elim: converse-rtranclE dest: single-valuedD intro: rtrancl-trans)
qed

```

```

lemma r-r-into-trancl:  $(a, b) \in R \implies (b, c) \in R \implies (a, c) \in R^+$ 
  by (fast intro: trancl-trans)

```

```

lemma trancl-into-trancl:  $(a, b) \in r^+ \implies (b, c) \in r \implies (a, c) \in r^+$ 
  by (induct rule: trancl-induct) (fast intro: r-r-into-trancl trancl-trans)+

```

```

lemma tranclp-rtranclp-tranclp:
  assumes  $r^{++} a b r^{**} b c$  shows  $r^{++} a c$ 

```

proof –

obtain z **where** $r\ a\ z\ r^{**}\ z\ c$
using *assms* **by** (*iprover dest: tranclpD rtranclp-trans*)
then show *?thesis*
by (*blast dest: rtranclp-into-tranclp2*)
qed

lemma *rtranclp-conversep*: $r^{-1-1^{**}} = r^{**-1-1}$
by(*auto simp add: fun-eq-iff intro: rtranclp-converseI rtranclp-converseD*)

lemmas *symp-rtranclp* = *sym-rtrancl*[*to-pred*]

lemmas *symp-conv-conversep-eq* = *sym-conv-converse-eq*[*to-pred*]

lemmas *rtranclp-tranclp-absorb* [*simp*] = *rtrancl-trancl-absorb*[*to-pred*]

lemmas *tranclp-rtranclp-absorb* [*simp*] = *trancl-rtrancl-absorb*[*to-pred*]

lemmas *rtranclp-reflclp-absorb* [*simp*] = *rtrancl-reflcl-absorb*[*to-pred*]

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

21.4 Symmetric closure

definition *symclp* :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
where *symclp* $r\ x\ y \longleftrightarrow r\ x\ y \vee r\ y\ x$

lemma *symclpI* [*simp, intro?*]:
shows *symclpI1*: $r\ x\ y \Longrightarrow \text{symclp}\ r\ x\ y$
and *symclpI2*: $r\ y\ x \Longrightarrow \text{symclp}\ r\ x\ y$
by(*simp-all add: symclp-def*)

lemma *symclpE* [*consumes 1, cases pred*]:
assumes *symclp* $r\ x\ y$
obtains (*base*) $r\ x\ y \mid$ (*sym*) $r\ y\ x$
using *assms* **by**(*auto simp add: symclp-def*)

lemma *symclp-pointfree*: $\text{symclp } r = \text{sup } r \ r^{-1-1}$
by(*auto simp add: symclp-def fun-eq-iff*)

lemma *symclp-greater*: $r \leq \text{symclp } r$
by(*simp add: symclp-pointfree*)

lemma *symclp-conversep* [*simp*]: $\text{symclp } r^{-1-1} = \text{symclp } r$
by(*simp add: symclp-pointfree sup commute*)

lemma *symp-on-symclp* [*simp*]: *symp-on* A (*symclp* R)
by(*auto simp add: symp-on-def elim: symclpE intro: symclpI*)

lemma *symp-symclp-eq*: $\text{symp } r \implies \text{symclp } r = r$
by(*simp add: symclp-pointfree symp-conv-conversep-eq*)

lemma *symp-rtranclp-symclp* [*simp*]: $\text{symp } (\text{symclp } r)^{**}$
by(*simp add: symp-rtranclp*)

lemma *rtranclp-symclp-sym* [*sym*]: $(\text{symclp } r)^{**} \ x \ y \implies (\text{symclp } r)^{**} \ y \ x$
by(*rule sympD[OF symp-rtranclp-symclp]*)

lemma *symclp-idem* [*simp*]: $\text{symclp } (\text{symclp } r) = \text{symclp } r$
by(*simp add: symclp-pointfree sup-commute converse-join*)

lemma *reflp-on-rtranclp* [*simp*]: *reflp-on* $A \ R^{**}$
by (*simp add: reflp-on-def*)

21.5 The power operation on relations

$R \overset{\sim}{\sim} n = R \ O \ \dots \ O \ R$, the n -fold composition of R

overloading

relpow \equiv *compow* :: $\text{nat} \Rightarrow ('a \times 'a) \ \text{set} \Rightarrow ('a \times 'a) \ \text{set}$

relpowp \equiv *compow* :: $\text{nat} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$

begin

primrec *relpow* :: $\text{nat} \Rightarrow ('a \times 'a) \ \text{set} \Rightarrow ('a \times 'a) \ \text{set}$

where

relpow $0 \ R = \text{Id}$

| *relpow* (*Suc* n) $R = (R \overset{\sim}{\sim} n) \ O \ R$

primrec *relpowp* :: $\text{nat} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$

where

relpowp $0 \ R = \text{HOL.eq}$

| *relpowp* (*Suc* n) $R = (R \overset{\sim}{\sim} n) \ OO \ R$

end

lemma *relpowp-relpow-eq* [*pred-set-conv*]:

$(\lambda x \ y. (x, y) \in R) \overset{\sim}{\sim} n = (\lambda x \ y. (x, y) \in R \overset{\sim}{\sim} n)$ **for** $R :: 'a \ \text{rel}$

by (induct n) (simp-all add: relcompp-relcomp-eq)

For code generation:

definition *relpow* :: $\text{nat} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set}$
 where *relpow-code-def* [*code-abbrev*]: *relpow* = *compow*

definition *relpowp* :: $\text{nat} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$
 where *relpowp-code-def* [*code-abbrev*]: *relpowp* = *compow*

lemma [*code*]:
relpow (Suc n) R = (*relpow* n R) O R
relpow 0 R = Id
 by (simp-all add: *relpow-code-def*)

lemma [*code*]:
relpowp (Suc n) R = (R $\overset{\sim}{\sim}$ n) OO R
relpowp 0 R = HOL.eq
 by (simp-all add: *relpowp-code-def*)

hide-const (open) *relpow*
hide-const (open) *relpowp*

lemma *relpow-1* [*simp*]: $R \overset{\sim}{\sim} 1 = R$
 for R :: $('a \times 'a) \text{ set}$
 by *simp*

lemma *relpowp-1* [*simp*]: $P \overset{\sim}{\sim} 1 = P$
 for P :: $'a \Rightarrow 'a \Rightarrow \text{bool}$
 by (fact *relpow-1* [to-pred])

lemma *relpow-0-I*: $(x, x) \in R \overset{\sim}{\sim} 0$
 by *simp*

lemma *relpowp-0-I*: $(P \overset{\sim}{\sim} 0) x x$
 by (fact *relpow-0-I* [to-pred])

lemma *relpow-Suc-I*: $(x, y) \in R \overset{\sim}{\sim} n \Longrightarrow (y, z) \in R \Longrightarrow (x, z) \in R \overset{\sim}{\sim} \text{Suc } n$
 by *auto*

lemma *relpowp-Suc-I*: $(P \overset{\sim}{\sim} n) x y \Longrightarrow P y z \Longrightarrow (P \overset{\sim}{\sim} \text{Suc } n) x z$
 by (fact *relpow-Suc-I* [to-pred])

lemma *relpow-Suc-I2*: $(x, y) \in R \Longrightarrow (y, z) \in R \overset{\sim}{\sim} n \Longrightarrow (x, z) \in R \overset{\sim}{\sim} \text{Suc } n$
 by (induct n arbitrary: z) (simp, fastforce)

lemma *relpowp-Suc-I2*: $P x y \Longrightarrow (P \overset{\sim}{\sim} n) y z \Longrightarrow (P \overset{\sim}{\sim} \text{Suc } n) x z$
 by (fact *relpow-Suc-I2* [to-pred])

lemma *relpow-0-E*: $(x, y) \in R \overset{\sim}{\sim} 0 \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$

by *simp*

lemma *relpow-0-E*: $(P \overset{\sim}{\sim} 0) x y \implies (x = y \implies Q) \implies Q$
 by (*fact relpow-0-E [to-pred]*)

lemma *relpow-Suc-E*: $(x, z) \in R \overset{\sim}{\sim} \text{Suc } n \implies (\bigwedge y. (x, y) \in R \overset{\sim}{\sim} n \implies (y, z) \in R \implies P) \implies P$
 by *auto*

lemma *relpow-Suc-E*: $(P \overset{\sim}{\sim} \text{Suc } n) x z \implies (\bigwedge y. (P \overset{\sim}{\sim} n) x y \implies P y z \implies Q) \implies Q$
 by (*fact relpow-Suc-E [to-pred]*)

lemma *relpow-E*:
 $(x, z) \in R \overset{\sim}{\sim} n \implies$
 $(n = 0 \implies x = z \implies P) \implies$
 $(\bigwedge y m. n = \text{Suc } m \implies (x, y) \in R \overset{\sim}{\sim} m \implies (y, z) \in R \implies P) \implies P$
 by (*cases n*) *auto*

lemma *relpowp-E*:
 $(P \overset{\sim}{\sim} n) x z \implies$
 $(n = 0 \implies x = z \implies Q) \implies$
 $(\bigwedge y m. n = \text{Suc } m \implies (P \overset{\sim}{\sim} m) x y \implies P y z \implies Q) \implies Q$
 by (*fact relpow-E [to-pred]*)

lemma *relpow-Suc-D2*: $(x, z) \in R \overset{\sim}{\sim} \text{Suc } n \implies (\exists y. (x, y) \in R \wedge (y, z) \in R \overset{\sim}{\sim} n)$
 by (*induct n arbitrary: x z*)
 (*blast intro: relpow-0-I relpow-Suc-I elim: relpow-0-E relpow-Suc-E*)+

lemma *relpowp-Suc-D2*: $(P \overset{\sim}{\sim} \text{Suc } n) x z \implies \exists y. P x y \wedge (P \overset{\sim}{\sim} n) y z$
 by (*fact relpow-Suc-D2 [to-pred]*)

lemma *relpow-Suc-E2*: $(x, z) \in R \overset{\sim}{\sim} \text{Suc } n \implies (\bigwedge y. (x, y) \in R \implies (y, z) \in R \overset{\sim}{\sim} n \implies P) \implies P$
 by (*blast dest: relpow-Suc-D2*)

lemma *relpowp-Suc-E2*: $(P \overset{\sim}{\sim} \text{Suc } n) x z \implies (\bigwedge y. P x y \implies (P \overset{\sim}{\sim} n) y z \implies Q) \implies Q$
 by (*fact relpow-Suc-E2 [to-pred]*)

lemma *relpow-Suc-D2'*: $\forall x y z. (x, y) \in R \overset{\sim}{\sim} n \wedge (y, z) \in R \longrightarrow (\exists w. (x, w) \in R \wedge (w, z) \in R \overset{\sim}{\sim} n)$
 by (*induct n*) (*simp-all, blast*)

lemma *relpowp-Suc-D2'*: $\forall x y z. (P \overset{\sim}{\sim} n) x y \wedge P y z \longrightarrow (\exists w. P x w \wedge (P \overset{\sim}{\sim} n) w z)$
 by (*fact relpow-Suc-D2' [to-pred]*)

lemma *relpow-E2*:

assumes $(x, z) \in R \overset{\sim}{\sim} n \ n = 0 \implies x = z \implies P$

$\bigwedge y \ m. \ n = \text{Suc } m \implies (x, y) \in R \implies (y, z) \in R \overset{\sim}{\sim} m \implies P$

shows P

proof (*cases n*)

case 0

with *assms* **show** *?thesis*

by *simp*

next

case $(\text{Suc } m)$

with *assms* *relpow-Suc-D2'* [*of m R*] **show** *?thesis*

by *force*

qed

lemma *relpowp-E2*:

$(P \overset{\sim}{\sim} n) \ x \ z \implies$

$(n = 0 \implies x = z \implies Q) \implies$

$(\bigwedge y \ m. \ n = \text{Suc } m \implies P \ x \ y \implies (P \overset{\sim}{\sim} m) \ y \ z \implies Q) \implies Q$

by (*fact relpow-E2* [*to-pred*])

lemma *relpow-add*: $R \overset{\sim}{\sim} (m + n) = R \overset{\sim}{\sim} m \ O \ R \overset{\sim}{\sim} n$

by (*induct n*) *auto*

lemma *relpowp-add*: $P \overset{\sim}{\sim} (m + n) = P \overset{\sim}{\sim} m \ O O \ P \overset{\sim}{\sim} n$

by (*fact relpow-add* [*to-pred*])

lemma *relpow-commute*: $R \ O \ R \overset{\sim}{\sim} n = R \overset{\sim}{\sim} n \ O \ R$

by (*induct n*) (*simp-all add: O-assoc* [*symmetric*])

lemma *relpowp-commute*: $P \ O O \ P \overset{\sim}{\sim} n = P \overset{\sim}{\sim} n \ O O \ P$

by (*fact relpow-commute* [*to-pred*])

lemma *relpow-empty*: $0 < n \implies (\{\} :: ('a \times 'a) \text{ set}) \overset{\sim}{\sim} n = \{\}$

by (*cases n*) *auto*

lemma *relpowp-bot*: $0 < n \implies (\perp :: 'a \Rightarrow 'a \Rightarrow \text{bool}) \overset{\sim}{\sim} n = \perp$

by (*fact relpow-empty* [*to-pred*])

lemma *rtrancl-imp-UN-relpow*:

assumes $p \in R^*$

shows $p \in (\bigcup n. R \overset{\sim}{\sim} n)$

proof (*cases p*)

case $(\text{Pair } x \ y)$

with *assms* **have** $(x, y) \in R^*$ **by** *simp*

then **have** $(x, y) \in (\bigcup n. R \overset{\sim}{\sim} n)$

proof *induct*

case *base*

show *?case* **by** (*blast intro: relpow-0-I*)

next

case *step*
then show *?case* **by** (*blast intro: relpow-Suc-I*)
qed
with *Pair* **show** *?thesis* **by** *simp*
qed

lemma *rtranclp-imp-Sup-relpow*:
assumes $(P^{**}) x y$
shows $(\bigsqcup n. P \tilde{n}) x y$
using *assms* **and** *rtrancl-imp-UN-relpow [of (x, y), to-pred]* **by** *simp*

lemma *relpow-imp-rtrancl*:
assumes $p \in R \tilde{n}$
shows $p \in R^*$
proof (*cases p*)
case (*Pair x y*)
with *assms* **have** $(x, y) \in R \tilde{n}$ **by** *simp*
then have $(x, y) \in R^*$
proof (*induct n arbitrary: x y*)
case 0
then show *?case* **by** *simp*
next
case *Suc*
then show *?case*
by (*blast elim: relpow-Suc-E intro: rtrancl-into-rtrancl*)
qed
with *Pair* **show** *?thesis* **by** *simp*
qed

lemma *relpowp-imp-rtranclp*: $(P \tilde{n}) x y \implies (P^{**}) x y$
using *relpow-imp-rtrancl [of (x, y), to-pred]* **by** *simp*

lemma *rtrancl-is-UN-relpow*: $R^* = (\bigcup n. R \tilde{n})$
by (*blast intro: rtrancl-imp-UN-relpow relpow-imp-rtrancl*)

lemma *rtranclp-is-Sup-relpowp*: $P^{**} = (\bigsqcup n. P \tilde{n})$
using *rtrancl-is-UN-relpow [to-pred, of P]* **by** *auto*

lemma *rtrancl-power*: $p \in R^* \iff (\exists n. p \in R \tilde{n})$
by (*simp add: rtrancl-is-UN-relpow*)

lemma *rtranclp-power*: $(P^{**}) x y \iff (\exists n. (P \tilde{n}) x y)$
by (*simp add: rtranclp-is-Sup-relpowp*)

lemma *trancl-power*: $p \in R^+ \iff (\exists n > 0. p \in R \tilde{n})$
proof –

have $(a, b) \in R^+ \iff (\exists n > 0. (a, b) \in R \tilde{n})$ **for** *a b*

proof *safe*

show $(a, b) \in R^+ \implies \exists n > 0. (a, b) \in R \tilde{n}$

```

    by (fastforce simp: rtrancl-is-UN-relpow relcomp-unfold dest: tranclD2)
  show  $(a, b) \in R^+$  if  $n > 0$   $(a, b) \in R \overset{\sim}{\sim} n$  for  $n$ 
  proof (cases  $n$ )
    case (Suc  $m$ )
    with that show ?thesis
    by (auto simp: dest: relpow-imp-rtrancl rtrancl-into-trancl1)
  qed (use that in auto)
  qed
  then show ?thesis
  by (cases  $p$ ) auto
  qed

```

lemma *tranclp-power*: $(P^{++}) x y \longleftrightarrow (\exists n > 0. (P \overset{\sim}{\sim} n) x y)$
 using *trancl-power* [to-pred, of $P(x, y)$] by *simp*

lemma *rtrancl-imp-relpow*: $p \in R^* \implies \exists n. p \in R \overset{\sim}{\sim} n$
 by (auto dest: rtrancl-imp-UN-relpow)

lemma *rtranclp-imp-relpowp*: $(P^{**}) x y \implies \exists n. (P \overset{\sim}{\sim} n) x y$
 by (auto dest: rtranclp-imp-Sup-relpowp)

By Sternagel/Thiemann:

lemma *relpow-fun-conv*: $(a, b) \in R \overset{\sim}{\sim} n \longleftrightarrow (\exists f. f 0 = a \wedge f n = b \wedge (\forall i < n. (f i, f (Suc i)) \in R))$

proof (*induct n arbitrary: b*)

case 0

show ?case by *auto*

next

case (Suc n)

show ?case

proof –

have $(\exists y. (\exists f. f 0 = a \wedge f n = y \wedge (\forall i < n. (f i, f (Suc i)) \in R)) \wedge (y, b) \in R)$
 \longleftrightarrow

$(\exists f. f 0 = a \wedge f (Suc n) = b \wedge (\forall i < Suc n. (f i, f (Suc i)) \in R))$

(is ?l \longleftrightarrow ?r)

proof

assume ?l

then obtain $c f$

where $1: f 0 = a \ f n = c \ \wedge i. i < n \implies (f i, f (Suc i)) \in R \ (c, b) \in R$

by *auto*

let ?g = $\lambda m. \text{if } m = \text{Suc } n \text{ then } b \text{ else } f m$

show ?r by (rule *exI*[of - ?g]) (*simp add: 1*)

next

assume ?r

then obtain f where $1: f 0 = a \ b = f (Suc n) \ \wedge i. i < Suc n \implies (f i, f (Suc i)) \in R$

by *auto*

show ?l by (rule *exI*[of - $f n$], rule *conjI*, rule *exI*[of - f], *auto simp add: 1*)

qed

then show *?thesis* **by** (*simp add: relcomp-unfold Suc*)
qed
qed

lemma *relpowp-fun-conv*: $(P \text{ } \sim n) x y \longleftrightarrow (\exists f. f 0 = x \wedge f n = y \wedge (\forall i < n. P (f i) (f (Suc i))))$
by (*fact relpow-fun-conv [to-pred]*)

lemma *relpow-finite-bounded1*:

fixes $R :: ('a \times 'a)$ *set*

assumes *finite R* **and** $k > 0$

shows $R \text{ } \sim k \subseteq (\bigcup n \in \{n. 0 < n \wedge n \leq \text{card } R\}. R \text{ } \sim n)$
(is - \subseteq ?r)

proof –

have $(a, b) \in R \text{ } \sim (Suc k) \implies \exists n. 0 < n \wedge n \leq \text{card } R \wedge (a, b) \in R \text{ } \sim n$ **for** a
 $b k$

proof (*induct k arbitrary: b*)

case 0

then have $R \neq \{\}$ **by** *auto*

with *card-0-eq[OF <finite R>]* **have** $\text{card } R \geq \text{Suc } 0$ **by** *auto*

then show *?case* **using** 0 **by** *force*

next

case $(Suc k)$

then obtain a' **where** $(a, a') \in R \text{ } \sim (Suc k)$ **and** $(a', b) \in R$

by *auto*

from *Suc(1)[OF <(a, a') $\in R \text{ } \sim (Suc k)$ >]* **obtain** n **where** $n \leq \text{card } R$ **and** $(a,$
 $a') \in R \text{ } \sim n$

by *auto*

have $(a, b) \in R \text{ } \sim (Suc n)$

using $\langle (a, a') \in R \text{ } \sim n \rangle$ **and** $\langle (a', b) \in R \rangle$ **by** *auto*

from $\langle n \leq \text{card } R \rangle$ **consider** $n < \text{card } R \mid n = \text{card } R$ **by** *force*

then show *?case*

proof *cases*

case 1

then show *?thesis*

using $\langle (a, b) \in R \text{ } \sim (Suc n) \rangle$ *Suc-leI[OF <n < card R>]* **by** *blast*

next

case 2

from $\langle (a, b) \in R \text{ } \sim (Suc n) \rangle$ [*unfolded relpow-fun-conv*]

obtain f **where** $f 0 = a$ **and** $f (Suc n) = b$

and *steps*: $\bigwedge i. i \leq n \implies (f i, f (Suc i)) \in R$ **by** *auto*

let $?p = \lambda i. (f i, f (Suc i))$

let $?N = \{i. i \leq n\}$

have $?p \text{ } \text{' } ?N \subseteq R$

using *steps* **by** *auto*

from *card-mono[OF assms(1) this]* **have** $\text{card } (?p \text{ } \text{' } ?N) \leq \text{card } R$.

also have $\dots < \text{card } ?N$

using $\langle n = \text{card } R \rangle$ **by** *simp*

finally have $\neg \text{inj-on } ?p \text{ } ?N$

```

    by (rule pigeonhole)
  then obtain  $i\ j$  where  $i: i \leq n$  and  $j: j \leq n$  and  $ij: i \neq j$  and  $p_{ij}: ?p\ i =$ 
 $?p\ j$ 
    by (auto simp: inj-on-def)
  let  $?i = \min\ i\ j$ 
  let  $?j = \max\ i\ j$ 
  have  $i: ?i \leq n$  and  $j: ?j \leq n$  and  $p_{ij}: ?p\ ?i = ?p\ ?j$  and  $ij: ?i < ?j$ 
    using  $i\ j\ ij\ p_{ij}$  unfolding min-def max-def by auto
  from  $i\ j\ p_{ij}\ ij$  obtain  $i\ j$  where  $i: i \leq n$  and  $j: j \leq n$  and  $ij: i < j$ 
    and  $p_{ij}: ?p\ i = ?p\ j$ 
    by blast
  let  $?g = \lambda l. \text{if } l \leq i \text{ then } f\ l \text{ else } f\ (l + (j - i))$ 
  let  $?n = \text{Suc } (n - (j - i))$ 
  have  $abl: (a, b) \in R \overset{\sim}{\sim} ?n$ 
    unfolding relpow-fun-conv
  proof (rule exI[of - ?g], intro conjI impI allI)
    show  $?g\ ?n = b$ 
      using  $\langle f(\text{Suc } n) = b \rangle\ j\ ij$  by auto
  next
  fix  $k$ 
  assume  $k < ?n$ 
  show  $(?g\ k, ?g\ (\text{Suc } k)) \in R$ 
  proof (cases  $k < i$ )
    case True
    with  $i$  have  $k \leq n$ 
      by auto
    from steps[OF this] show ?thesis
      using True by simp
  next
  case False
  then have  $i \leq k$  by auto
  show ?thesis
  proof (cases  $k = i$ )
    case True
    then show ?thesis
      using  $ij\ p_{ij}\ steps[OF\ i]$  by simp
  next
  case False
  with  $\langle i \leq k \rangle$  have  $i < k$  by auto
  then have small:  $k + (j - i) \leq n$ 
    using  $\langle k < ?n \rangle$  by arith
  show ?thesis
    using steps[OF small]  $\langle i < k \rangle$  by auto
  qed
  qed
  qed (simp add:  $\langle f\ 0 = a \rangle$ )
  moreover have  $?n \leq n$ 
    using  $i\ j\ ij$  by arith
  ultimately show ?thesis

```

```

    using ⟨n = card R⟩ by blast
  qed
qed
then show ?thesis
  using gr0-implies-Suc[OF ⟨k > 0⟩] by auto
qed

```

```

lemma relpow-finite-bounded:
  fixes R :: ('a × 'a) set
  assumes finite R
  shows  $R^{\sim k} \subseteq (\bigcup_{n \in \{n. n \leq \text{card } R\}}. R^{\sim n})$ 
proof (cases k)
  case (Suc k')
  then show ?thesis
    using relpow-finite-bounded1[OF assms, of k] by auto
qed force

```

```

lemma rtrancl-finite-eq-relpow: finite R  $\implies R^* = (\bigcup_{n \in \{n. n \leq \text{card } R\}}. R^{\sim n})$ 
  by (fastforce simp: rtrancl-power dest: relpow-finite-bounded)

```

```

lemma trancl-finite-eq-relpow:
  assumes finite R shows  $R^+ = (\bigcup_{n \in \{n. 0 < n \wedge n \leq \text{card } R\}}. R^{\sim n})$ 
proof -
  have  $\bigwedge a b n. \llbracket 0 < n; (a, b) \in R^{\sim n} \rrbracket \implies \exists x > 0. x \leq \text{card } R \wedge (a, b) \in R^{\sim x}$ 
  using assms by (auto dest: relpow-finite-bounded1)
  then show ?thesis
    by (auto simp: trancl-power)
qed

```

```

lemma finite-relcomp[simp,intro]:
  assumes finite R and finite S
  shows finite (R O S)
proof -
  have  $R O S = (\bigcup_{(x, y) \in R. \bigcup_{(u, v) \in S. \text{if } u = y \text{ then } \{(x, v)\} \text{ else } \{\}})$ 
  by (force simp: split-def image-constant-conv split: if-splits)
  then show ?thesis
    using assms by clarsimp
qed

```

```

lemma finite-relpow [simp, intro]:
  fixes R :: ('a × 'a) set
  assumes finite R
  shows  $n > 0 \implies \text{finite } (R^{\sim n})$ 
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)

```

then show *?case* **by** (*cases* *n*) (*use assms in simp-all*)
qed

lemma *single-valued-relpow*:
fixes *R* :: ('a × 'a) set
shows *single-valued R* \implies *single-valued (R \sim n)*
proof (*induct n arbitrary: R*)
case 0
then show *?case* **by** *simp*
next
case (*Suc n*)
show *?case*
by (*rule single-valuedI*)
(use Suc in <fast dest: single-valuedD elim: relpow-Suc-E>)
qed

21.6 Bounded transitive closure

definition *ntrancl* :: nat \Rightarrow ('a × 'a) set \Rightarrow ('a × 'a) set
where *ntrancl n R* = ($\bigcup_{i \in \{i. 0 < i \wedge i \leq \text{Suc } n\}} R \sim i$)

lemma *ntrancl-Zero* [*simp, code*]: *ntrancl 0 R* = *R*

proof
show *R* \subseteq *ntrancl 0 R*
unfolding *ntrancl-def* **by** *fastforce*
have $0 < i \wedge i \leq \text{Suc } 0 \longleftrightarrow i = 1$ **for** *i*
by *auto*
then show *ntrancl 0 R* \leq *R*
unfolding *ntrancl-def* **by** *auto*
qed

lemma *ntrancl-Suc* [*simp*]: *ntrancl (Suc n) R* = *ntrancl n R O (Id \cup R)*

proof
have $(a, b) \in \text{ntrancl } n \text{ } R \text{ } O \text{ } (Id \cup R)$ **if** $(a, b) \in \text{ntrancl } (Suc \ n) \ R$ **for** *a b*
proof –
from that obtain *i* **where** $0 < i \wedge i \leq \text{Suc } (Suc \ n)$ $(a, b) \in R \sim i$
unfolding *ntrancl-def* **by** *auto*
show *?thesis*
proof (*cases i = 1*)
case *True*
with $\langle (a, b) \in R \sim i \rangle$ **show** *?thesis*
by (*auto simp: ntrancl-def*)
next
case *False*
with $\langle 0 < i \rangle$ **obtain** *j* **where** $i = \text{Suc } j \ 0 < j$
by (*cases i*) *auto*
with $\langle (a, b) \in R \sim i \rangle$ **obtain** *c* **where** *c1*: $(a, c) \in R \sim j$ **and** *c2*: $(c, b) \in R$
by *auto*


```

from  $c1\ j\ \langle i \leq \text{Suc } (\text{Suc } n) \rangle$  have  $(a, c) \in \text{ntrancl } n\ R$ 
  by (fastforce simp: ntrancl-def)
with  $c2$  show ?thesis by fastforce
qed
qed
then show  $\text{ntrancl } (\text{Suc } n)\ R \subseteq \text{ntrancl } n\ R\ O\ (Id \cup R)$ 
  by auto
show  $\text{ntrancl } n\ R\ O\ (Id \cup R) \subseteq \text{ntrancl } (\text{Suc } n)\ R$ 
  by (fastforce simp: ntrancl-def)
qed

```

```

lemma [code]:  $\text{ntrancl } (\text{Suc } n)\ r = (\text{let } r' = \text{ntrancl } n\ r \text{ in } r' \cup r' O r)$ 
  by (auto simp: Let-def)

```

```

lemma finite-trancl-ntrancl:  $\text{finite } R \implies \text{trancl } R = \text{ntrancl } (\text{card } R - 1)\ R$ 
  by (cases card R) (auto simp: trancl-finite-eq-relpow relpow-empty ntrancl-def)

```

21.7 Acyclic relations

```

definition acyclic ::  $('a \times 'a)\ \text{set} \Rightarrow \text{bool}$ 
  where  $\text{acyclic } r \iff (\forall x. (x, x) \notin r^+)$ 

```

```

abbreviation acyclicP ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ 
  where  $\text{acyclicP } r \equiv \text{acyclic } \{(x, y). r\ x\ y\}$ 

```

```

lemma acyclic-irrefl [code]:  $\text{acyclic } r \iff \text{irrefl } (r^+)$ 
  by (simp add: acyclic-def irrefl-def)

```

```

lemma acyclicI:  $\forall x. (x, x) \notin r^+ \implies \text{acyclic } r$ 
  by (simp add: acyclic-def)

```

```

lemma (in preorder) acyclicI-order:
  assumes *:  $\bigwedge a\ b. (a, b) \in r \implies f\ b < f\ a$ 
  shows  $\text{acyclic } r$ 

```

```

proof –
  have  $f\ b < f\ a$  if  $(a, b) \in r^+$  for  $a\ b$ 
    using that by induct (auto intro: * less-trans)
  then show ?thesis
    by (auto intro!: acyclicI)

```

qed

```

lemma acyclic-insert [iff]:  $\text{acyclic } (\text{insert } (y, x)\ r) \iff \text{acyclic } r \wedge (x, y) \notin r^*$ 
  by (simp add: acyclic-def trancl-insert) (blast intro: rtrancl-trans)

```

```

lemma acyclic-converse [iff]:  $\text{acyclic } (r^{-1}) \iff \text{acyclic } r$ 
  by (simp add: acyclic-def trancl-converse)

```

```

lemmas acyclicP-converse [iff] = acyclic-converse [to-pred]

```

lemma *acyclic-impl-antisym-rtrancl*: *acyclic r* \implies *antisym (r*)*
by (*simp add: acyclic-def antisym-def*)
(blast elim: rtranclE intro: rtrancl-into-trancl1 rtrancl-trancl-trancl)

lemma *acyclic-subset*: *acyclic s* \implies $r \subseteq s \implies$ *acyclic r*
unfolding *acyclic-def* **by** (*blast intro: trancl-mono*)

21.8 Setup of transitivity reasoner

ML \langle
structure Trancl-Tac = Trancl-Tac
 $($
val r-into-trancl = @{thm trancl.r-into-trancl};
val trancl-trans = @{thm trancl-trans};
val rtrancl-refl = @{thm rtrancl.rtrancl-refl};
val r-into-rtrancl = @{thm r-into-rtrancl};
val trancl-into-rtrancl = @{thm trancl-into-rtrancl};
val rtrancl-trancl-trancl = @{thm rtrancl-trancl-trancl};
val trancl-rtrancl-trancl = @{thm trancl-rtrancl-trancl};
val rtrancl-trans = @{thm rtrancl-trans};

fun decomp Const- \langle Trueprop for t \rangle =
let
fun dec Const- \langle Set.member - for Const- \langle Pair - - for a b \rangle rel \rangle =
let
fun decr Const- \langle rtrancl - for r \rangle = (r,r)*
| decr Const- \langle trancl - for r \rangle = (r,r+)
| decr r = (r,r);
val (rel,r) = decr (Envir.beta-eta-contract rel);
in SOME (a,b,rel,r) end
| dec - = NONE
in dec t end
| decomp - = NONE;
 \rangle);

structure Tranclp-Tac = Trancl-Tac
 $($
val r-into-trancl = @{thm tranclp.r-into-trancl};
val trancl-trans = @{thm tranclp-trans};
val rtrancl-refl = @{thm rtranclp.rtrancl-refl};
val r-into-rtrancl = @{thm r-into-rtranclp};
val trancl-into-rtrancl = @{thm tranclp-into-rtranclp};
val rtrancl-trancl-trancl = @{thm rtranclp-tranclp-tranclp};
val trancl-rtrancl-trancl = @{thm tranclp-rtranclp-tranclp};
val rtrancl-trans = @{thm rtranclp-trans};

fun decomp Const- \langle Trueprop for t \rangle =

```

let
  fun dec (rel $ a $ b) =
    let
      fun decr Const-⟨rtranclp - for r⟩ = (r,r*)
        | decr Const-⟨tranclp - for r⟩ = (r,r+)
        | decr r = (r,r);
      val (rel,r) = decr rel;
      in SOME (a, b, rel, r) end
    | dec - = NONE
  in dec t end
| decomp - = NONE;
);
⟩

```

```

setup ⟨
  map-theory-simpset (fn ctxt => ctxt
    addSolver (mk-solver Trancl Trancl-Tac.trancl-tac)
    addSolver (mk-solver Rtrancl Trancl-Tac.rtrancl-tac)
    addSolver (mk-solver Tranclp Tranclp-Tac.trancl-tac)
    addSolver (mk-solver Rtranclp Tranclp-Tac.rtrancl-tac))
  ⟩

```

```

lemma transp-rtranclp [simp]: transp R**
  by(auto simp add: transp-def)

```

Optional methods.

```

method-setup trancl =
  ⟨Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.trancl-tac)⟩
  ⟨simple transitivity reasoner⟩
method-setup rtrancl =
  ⟨Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.rtrancl-tac)⟩
  ⟨simple transitivity reasoner⟩
method-setup tranclp =
  ⟨Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.trancl-tac)⟩
  ⟨simple transitivity reasoner (predicate version)⟩
method-setup rtranclp =
  ⟨Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.rtrancl-tac)⟩
  ⟨simple transitivity reasoner (predicate version)⟩

```

end

22 Well-founded Recursion

```

theory Wellfounded
  imports Transitive-Closure
begin

```

22.1 Basic Definitions

definition $wf :: ('a \times 'a) \text{ set} \Rightarrow \text{bool}$

where $wf\ r \longleftrightarrow (\forall P. (\forall x. (\forall y. (y, x) \in r \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall x. P\ x))$

definition $wfP :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where $wfP\ r \longleftrightarrow wf\ \{(x, y). r\ x\ y\}$

lemma $wfP\text{-}wf\text{-}eq$ [*pred-set-conv*]: $wfP\ (\lambda x\ y. (x, y) \in r) = wf\ r$

by (*simp add: wfP-def*)

lemma $wfUNIVI$: $(\bigwedge P\ x. (\forall x. (\forall y. (y, x) \in r \longrightarrow P\ y) \longrightarrow P\ x) \Longrightarrow P\ x) \Longrightarrow wf\ r$

unfolding $wf\text{-}def$ **by** $blast$

lemmas $wfPUNIVI = wfUNIVI$ [*to-pred*]

Restriction to domain A and range B . If r is well-founded over their intersection, then $wf\ r$.

lemma wfI :

assumes $r \subseteq A \times B$

and $\bigwedge x\ P. [\forall y. (y, x) \in r \longrightarrow P\ y] \longrightarrow P\ x; x \in A; x \in B] \Longrightarrow P\ x$

shows $wf\ r$

using $assms$ **unfolding** $wf\text{-}def$ **by** $blast$

lemma $wf\text{-}induct$:

assumes $wf\ r$

and $\bigwedge x. \forall y. (y, x) \in r \longrightarrow P\ y \Longrightarrow P\ x$

shows $P\ a$

using $assms$ **unfolding** $wf\text{-}def$ **by** $blast$

lemmas $wfP\text{-}induct = wf\text{-}induct$ [*to-pred*]

lemmas $wf\text{-}induct\text{-}rule = wf\text{-}induct$ [*rule-format, consumes 1, case-names less, induct set: wf*]

lemmas $wfP\text{-}induct\text{-}rule = wf\text{-}induct\text{-}rule$ [*to-pred, induct set: wfP*]

lemma $wf\text{-}not\text{-}sym$: $wf\ r \Longrightarrow (a, x) \in r \Longrightarrow (x, a) \notin r$

by (*induct a arbitrary: x set: wf*) $blast$

lemma $wf\text{-}asym$:

assumes $wf\ r\ (a, x) \in r$

obtains $(x, a) \notin r$

by (*drule wf-not-sym[OF assms]*)

lemma $wf\text{-}imp\text{-}asym$: $wf\ r \Longrightarrow asym\ r$

by (*auto intro: asymI elim: wf-asym*)

lemma $wfP\text{-}imp\text{-}asymp$: $wfP\ r \Longrightarrow asymp\ r$

by (rule wf-imp-asymp[to-pred])

lemma *wf-not-refl* [simp]: $wf\ r \implies (a, a) \notin r$
by (blast elim: wf-asymp)

lemma *wf-irrefl*:
assumes $wf\ r$
obtains $(a, a) \notin r$
by (drule wf-not-refl[OF assms])

lemma *wf-imp-irrefl*:
assumes $wf\ r$ shows *irrefl* r
using *wf-irrefl* [OF assms] by (auto simp add: irrefl-def)

lemma *wfP-imp-irreflp*: $wfP\ r \implies irreflp\ r$
by (rule wf-imp-irrefl[to-pred])

lemma *wf-wellorderI*:
assumes $wf: wf\ \{(x::'a::ord, y). x < y\}$
and $lin: OFCLASS('a::ord, linorder-class)$
shows $OFCLASS('a::ord, wellorder-class)$
apply (rule wellorder-class.intro [OF lin])
apply (simp add: wellorder-class.intro class.wellorder-axioms.intro wf-induct-rule [OF wf])
done

lemma (in *wellorder*) $wf: wf\ \{(x, y). x < y\}$
unfolding *wf-def* by (blast intro: less-induct)

lemma (in *wellorder*) *wfP-less*[simp]: $wfP\ (<)$
by (simp add: wf wfP-def)

22.2 Basic Results

Point-free characterization of well-foundedness

lemma *wfE-pf*:
assumes $wf: wf\ R$
and $a: A \subseteq R \text{ “ } A$
shows $A = \{\}$
proof –
from wf have $x \notin A$ for x
proof *induct*
fix x assume $\bigwedge y. (y, x) \in R \implies y \notin A$
then have $x \notin R \text{ “ } A$ by *blast*
with a show $x \notin A$ by *blast*
qed
then show *?thesis* by *auto*
qed

lemma *wfI-pf*:
 assumes $a: \bigwedge A. A \subseteq R \text{ “ } A \implies A = \{\}$
 shows *wf R*
proof (*rule wfUNIVI*)
 fix $P :: 'a \Rightarrow \text{bool}$ and x
 let $?A = \{x. \neg P x\}$
 assume $\forall x. (\forall y. (y, x) \in R \longrightarrow P y) \longrightarrow P x$
 then have $?A \subseteq R \text{ “ } ?A$ by *blast*
 with a show $P x$ by *blast*
qed

22.2.1 Minimal-element characterization of well-foundedness

lemma *wfE-min*:
 assumes *wf R* and $Q: x \in Q$
 obtains z where $z \in Q \bigwedge y. (y, z) \in R \implies y \notin Q$
 using Q *wfE-pf[OF wf, of Q]* by *blast*

lemma *wfE-min'*:
 $wf R \implies Q \neq \{\} \implies (\bigwedge z. z \in Q \implies (\bigwedge y. (y, z) \in R \implies y \notin Q)) \implies \text{thesis}$
 $\implies \text{thesis}$
 using *wfE-min[of R - Q]* by *blast*

lemma *wfI-min*:
 assumes $a: \bigwedge x Q. x \in Q \implies \exists z \in Q. \forall y. (y, z) \in R \longrightarrow y \notin Q$
 shows *wf R*
proof (*rule wfI-pf*)
 fix A
 assume $b: A \subseteq R \text{ “ } A$
 have *False* if $x \in A$ for x
 using a [*OF that*] b by *blast*
 then show $A = \{\}$ by *blast*
qed

lemma *wf-eq-minimal*: $wf r \iff (\forall Q x. x \in Q \longrightarrow (\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q))$
 apply (*rule iffI*)
 apply (*blast intro: elim!: wfE-min*)
 by (*rule wfI-min*) *auto*

lemmas *wfP-eq-minimal = wf-eq-minimal* [*to-pred*]

22.2.2 Well-foundedness of transitive closure

lemma *wf-trancl*:
 assumes *wf r*
 shows *wf (r⁺)*
proof –
 have $P x$ if *induct-step*: $\bigwedge x. (\bigwedge y. (y, x) \in r^+ \implies P y) \implies P x$ for $P x$
proof (*rule induct-step*)

```

show  $P\ y$  if  $(y, x) \in r^+$  for  $y$ 
  using  $\langle wf\ r \rangle$  and that
proof (induct  $x$  arbitrary:  $y$ )
  case (less  $x$ )
  note  $hyp = \langle \bigwedge x' y'. (x', x) \in r \implies (y', x') \in r^+ \implies P\ y' \rangle$ 
  from  $\langle (y, x) \in r^+ \rangle$  show  $P\ y$ 
  proof cases
    case base
    show  $P\ y$ 
    proof (rule induct-step)
      fix  $y'$ 
      assume  $(y', y) \in r^+$ 
      with  $\langle (y, x) \in r \rangle$  show  $P\ y'$ 
      by (rule  $hyp$  [of  $y\ y'$ ])
    qed
  next
  case step
  then obtain  $x'$  where  $(x', x) \in r$  and  $(y, x') \in r^+$ 
  by simp
  then show  $P\ y$  by (rule  $hyp$  [of  $x'\ y$ ])
  qed
qed
qed
then show ?thesis unfolding wf-def by blast
qed

```

lemmas $wfP\text{-trancl} = wf\text{-trancl}$ [*to-pred*]

lemma $wf\text{-converse-trancl}$: $wf\ (r^{-1}) \implies wf\ ((r^+)^{-1})$
apply (*subst* *trancl-converse* [*symmetric*])
apply (*erule* $wf\text{-trancl}$)
done

Well-foundedness of subsets

lemma $wf\text{-subset}$: $wf\ r \implies p \subseteq r \implies wf\ p$
by (*simp* *add*: $wf\text{-eq-minimal}$) *fast*

lemmas $wfP\text{-subset} = wf\text{-subset}$ [*to-pred*]

Well-foundedness of the empty relation

lemma $wf\text{-empty}$ [*iff*]: $wf\ \{\}$
by (*simp* *add*: $wf\text{-def}$)

lemma $wfP\text{-empty}$ [*iff*]: $wfP\ (\lambda x\ y. False)$

proof –
have $wfP\ bot$
by (*fact* $wf\text{-empty}$ [*to-pred* $bot\text{-empty-eq2}$])
then **show** *?thesis*
by (*simp* *add*: $bot\text{-fun-def}$)

qed

lemma *wf-Int1*: $wf\ r \implies wf\ (r \cap r')$
by (*erule wf-subset*) (*rule Int-lower1*)

lemma *wf-Int2*: $wf\ r \implies wf\ (r' \cap r)$
by (*erule wf-subset*) (*rule Int-lower2*)

Exponentiation.

lemma *wf-exp*:
assumes $wf\ (R \rightsquigarrow n)$
shows $wf\ R$
proof (*rule wfI-pf*)
fix A **assume** $A \subseteq R$ “ A ”
then have $A \subseteq (R \rightsquigarrow n)$ “ A ”
by (*induct n*) *force+*
with $\langle wf\ (R \rightsquigarrow n) \rangle$ **show** $A = \{\}$
by (*rule wfE-pf*)

qed

Well-foundedness of *insert*.

lemma *wf-insert [iff]*: $wf\ (insert\ (y,x)\ r) \longleftrightarrow wf\ r \wedge (x,y) \notin r^*$ (**is** *?lhs = ?rhs*)

proof

assume *?lhs* **then show** *?rhs*
by (*blast elim: wf-trancl [THEN wf-irrefl]*)
intro: rtrancl-into-trancl1 wf-subset rtrancl-mono [THEN subsetD])

next

assume R : *?rhs*
then have R' : $Q \neq \{\} \implies (\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q)$ **for** Q
by (*auto simp: wf-eq-minimal*)

show *?lhs*

unfolding *wf-eq-minimal*

proof *clarify*

fix Q :: ‘*a set and q*’

assume $q \in Q$

then obtain a **where** $a \in Q$ **and** a : $\bigwedge y. (y, a) \in r \implies y \notin Q$

using R **by** (*auto simp: wf-eq-minimal*)

show $\exists z \in Q. \forall y'. (y', z) \in insert\ (y, x)\ r \longrightarrow y' \notin Q$

proof (*cases a=x*)

case *True*

show *?thesis*

proof (*cases y \in Q*)

case *True*

then obtain z **where** $z \in Q$ $(z, y) \in r^*$

$\bigwedge z'. (z', z) \in r \longrightarrow z' \in Q \longrightarrow (z', y) \notin r^*$

using R' [*of* $\{z \in Q. (z, y) \in r^*\}]$ **by** *auto*

then have $\forall y'. (y', z) \in insert\ (y, x)\ r \longrightarrow y' \notin Q$

using R **by** (*blast intro: rtrancl-trans*) $+$

then show *?thesis*


```

      by (rule bexI) fact
    next
      case False
      then show ?thesis
        using  $a \langle a \in Q \rangle$  by blast
    qed
  next
    case False
    with  $a \langle a \in Q \rangle$  show ?thesis
      by blast
    qed
  qed
qed

```

22.2.3 Well-foundedness of image

lemma *wf-map-prod-image-Dom-Ran*:

```

fixes  $r :: 'a \times 'a$  set
and  $f :: 'a \Rightarrow 'b$ 
assumes  $wf\text{-}r$ :  $wf\ r$ 
and  $inj$ :  $\bigwedge a\ a'.\ a \in Domain\ r \implies a' \in Range\ r \implies f\ a = f\ a' \implies a = a'$ 
shows  $wf\ (map\text{-}prod\ f\ f\ 'r)$ 
proof (unfold wf-eq-minimal, clarify)
fix  $B :: 'b$  set and  $b :: 'b$ 
assume  $b \in B$ 
define  $A$  where  $A = f\ \text{-}\ 'B \cap Domain\ r$ 
show  $\exists z \in B.\ \forall y.\ (y, z) \in map\text{-}prod\ f\ f\ 'r \longrightarrow y \notin B$ 
proof (cases  $A = \{\}$ )
  case False
  then obtain  $a0$  where  $a0 \in A$  and  $\forall a.\ (a, a0) \in r \longrightarrow a \notin A$ 
    using wfE-min[OF wf-r] by auto
  thus ?thesis
    using inj unfolding A-def
    by (intro bexI[of -  $f\ a0$ ]) auto
qed (use  $\langle b \in B \rangle$  in  $\langle unfold\ A\text{-}def,\ auto \rangle$ )
qed

```

lemma *wf-map-prod-image*: $wf\ r \implies inj\ f \implies wf\ (map\text{-}prod\ f\ f\ 'r)$
by(rule *wf-map-prod-image-Dom-Ran*) (auto dest: *inj-onD*)

22.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:

```

assumes  $wf\ R\ wf\ S$ 
assumes  $R\ O\ S \subseteq R$ 
shows  $wf\ (R \cup S)$ 
proof (rule wfI-min)
fix  $x :: 'a$  and  $Q$ 
let  $?Q' = \{x \in Q.\ \forall y.\ (y, x) \in R \longrightarrow y \notin Q\}$ 
assume  $x \in Q$ 

```

obtain a **where** $a \in ?Q'$
by (rule *wfE-min* [*OF* $\langle wf R \rangle \langle x \in Q \rangle$]) *blast*
with $\langle wf S \rangle$ **obtain** z **where** $z \in ?Q'$ **and** $zmin: \bigwedge y. (y, z) \in S \implies y \notin ?Q'$
by (erule *wfE-min*)
have $y \notin Q$ **if** $(y, z) \in S$ **for** y
proof
from *that have* $y \notin ?Q'$ **by** (rule *zmin*)
assume $y \in Q$
with $\langle y \notin ?Q' \rangle$ **obtain** w **where** $(w, y) \in R$ **and** $w \in Q$ **by** *auto*
from $\langle (w, y) \in R \rangle \langle (y, z) \in S \rangle$ **have** $(w, z) \in R \cap S$ **by** (rule *relcompI*)
with $\langle R \cap S \subseteq R \rangle$ **have** $(w, z) \in R$ **..**
with $\langle z \in ?Q' \rangle$ **have** $w \notin Q$ **by** *blast*
with $\langle w \in Q \rangle$ **show** *False* **by** *contradiction*
qed
with $\langle z \in ?Q' \rangle$ **show** $\exists z \in Q. \forall y. (y, z) \in R \cup S \longrightarrow y \notin Q$ **by** *blast*
qed

Well-foundedness of indexed union with disjoint domains and ranges.

lemma *wf-UN*:

assumes $r: \bigwedge i. i \in I \implies wf (r i)$
and $disj: \bigwedge i j. \llbracket i \in I; j \in I; r i \neq r j \rrbracket \implies Domain (r i) \cap Range (r j) = \{\}$
shows $wf (\bigcup_{i \in I} r i)$
unfolding *wf-eq-minimal*
proof *clarify*
fix A **and** $a :: 'b$
assume $a \in A$
show $\exists z \in A. \forall y. (y, z) \in \bigcup (r ` I) \longrightarrow y \notin A$
proof (cases $\exists i \in I. \exists a \in A. \exists b \in A. (b, a) \in r i$)
case *True*
then obtain $i b c$ **where** $ibc: i \in I b \in A c \in A (c, b) \in r i$
by *blast*
have $ri: \bigwedge Q. Q \neq \{\} \implies \exists z \in Q. \forall y. (y, z) \in r i \longrightarrow y \notin Q$
using r [*OF* $\langle i \in I \rangle$] **unfolding** *wf-eq-minimal* **by** *auto*
show *?thesis*
using ri [*of* $\{a. a \in A \wedge (\exists b \in A. (b, a) \in r i)\}$] ibc *disj*
by *blast*
next
case *False*
with $\langle a \in A \rangle$ **show** *?thesis*
by *blast*
qed
qed

lemma *wfP-SUP*:

$\forall i. wfP (r i) \implies \forall i j. r i \neq r j \longrightarrow \inf (Domainp (r i)) (Rangep (r j)) = bot$
 \implies
 $wfP (\bigsqcup (range r))$
by (rule *wf-UN[to-pred]*) *simp-all*

lemma *wf-Union*:
assumes $\forall r \in R. \text{wf } r$
and $\forall r \in R. \forall s \in R. r \neq s \longrightarrow \text{Domain } r \cap \text{Range } s = \{\}$
shows $\text{wf } (\bigcup R)$
using *assms wf-UN[of R $\lambda i. i$] by simp*

Intuition: We find an $R \cup S$ -min element of a nonempty subset A by case distinction.

1. There is a step $a -R \rightarrow b$ with $a, b \in A$. Pick an R -min element z of the (nonempty) set $\{a \in A \mid \exists b \in A. a -R \rightarrow b\}$. By definition, there is $z' \in A$ s.t. $z -R \rightarrow z'$. Because z is R -min in the subset, z' must be R -min in A . Because z' has an R -predecessor, it cannot have an S -successor and is thus S -min in A as well.
2. There is no such step. Pick an S -min element of A . In this case it must be an R -min element of A as well.

lemma *wf-Un*: $\text{wf } r \Longrightarrow \text{wf } s \Longrightarrow \text{Domain } r \cap \text{Range } s = \{\} \Longrightarrow \text{wf } (r \cup s)$
using *wf-union-compatible[of s r]*
by (*auto simp: Un-ac*)

lemma *wf-union-merge*: $\text{wf } (R \cup S) = \text{wf } (R \circ R \cup S \circ R \cup S)$
(is $\text{wf } ?A = \text{wf } ?B$)

proof

assume $\text{wf } ?A$

with *wf-trancl* **have** $\text{wf } T: \text{wf } (?A^+)$.

moreover have $?B \subseteq ?A^+$

by (*subst trancl-unfold, subst trancl-unfold*) *blast*

ultimately show $\text{wf } ?B$ **by** (*rule wf-subset*)

next

assume $\text{wf } ?B$

show $\text{wf } ?A$

proof (*rule wfI-min*)

fix $Q :: 'a \text{ set}$ **and** x

assume $x \in Q$

with $\langle \text{wf } ?B \rangle$ **obtain** z **where** $z \in Q$ **and** $\bigwedge y. (y, z) \in ?B \Longrightarrow y \notin Q$

by (*erule wfE-min*)

then have 1: $\bigwedge y. (y, z) \in R \circ R \Longrightarrow y \notin Q$

and 2: $\bigwedge y. (y, z) \in S \circ R \Longrightarrow y \notin Q$

and 3: $\bigwedge y. (y, z) \in S \Longrightarrow y \notin Q$

by *auto*

show $\exists z \in Q. \forall y. (y, z) \in ?A \longrightarrow y \notin Q$

proof (*cases* $\forall y. (y, z) \in R \longrightarrow y \notin Q$)

case *True*

with $\langle z \in Q \rangle$ 3 **show** *?thesis* **by** *blast*

next

case *False*

```

then obtain  $z'$  where  $z' \in Q$   $\langle z', z \rangle \in R$  by blast
have  $\forall y. (y, z') \in ?A \longrightarrow y \notin Q$ 
proof (intro allI impI)
  fix  $y$  assume  $(y, z') \in ?A$ 
  then show  $y \notin Q$ 
  proof
    assume  $(y, z') \in R$ 
    then have  $(y, z) \in R \ O \ R$  using  $\langle z', z \rangle \in R$  ..
    with 1 show  $y \notin Q$  .
  next
    assume  $(y, z') \in S$ 
    then have  $(y, z) \in S \ O \ R$  using  $\langle z', z \rangle \in R$  ..
    with 2 show  $y \notin Q$  .
  qed
qed
with  $\langle z' \in Q \rangle$  show ?thesis ..
qed
qed
qed

```

lemma *wf-comp-self*: $wf \ R \longleftrightarrow wf \ (R \ O \ R)$ — special case
by (*rule wf-union-merge* [**where** $S = \{\}$, *simplified*])

22.4 Well-Foundedness of Composition

Bachmair and Dershowitz 1986, Lemma 2. [Provided by Tjark Weber]

lemma *qc-wf-relto-iff*:

assumes $R \ O \ S \subseteq (R \cup S)^* \ O \ R$ — R quasi-commutes over S

shows $wf \ (S^* \ O \ R \ O \ S^*) \longleftrightarrow wf \ R$

(**is** $wf \ ?S \longleftrightarrow -$)

proof

show $wf \ R$ **if** $wf \ ?S$

proof —

have $R \subseteq ?S$ **by** *auto*

with *wf-subset* [*of* $?S$] **that show** $wf \ R$

by *auto*

qed

next

show $wf \ ?S$ **if** $wf \ R$

proof (*rule wfI-pf*)

fix A

assume $A: A \subseteq ?S$ “ A

let $?X = (R \cup S)^*$ “ A

have $*$: $R \ O \ (R \cup S)^* \subseteq (R \cup S)^* \ O \ R$

proof —

have $(x, z) \in (R \cup S)^* \ O \ R$ **if** $(y, z) \in (R \cup S)^*$ **and** $(x, y) \in R$ **for** $x \ y \ z$

using *that*

proof (*induct* $y \ z$)

case *rtrancl-refl*

```

    then show ?case by auto
  next
  case (rtrancl-into-rtrancl a b c)
  then have  $(x, c) \in ((R \cup S)^* O (R \cup S)^*) O R$ 
    using assms by blast
  then show ?case by simp
qed
then show ?thesis by auto
qed
then have  $R O S^* \subseteq (R \cup S)^* O R$ 
  using rtrancl-Un-subset by blast
then have  $?S \subseteq (R \cup S)^* O (R \cup S)^* O R$ 
  by (simp add: relcomp-mono rtrancl-mono)
also have  $\dots = (R \cup S)^* O R$ 
  by (simp add: O-assoc[symmetric])
finally have  $?S O (R \cup S)^* \subseteq (R \cup S)^* O R O (R \cup S)^*$ 
  by (simp add: O-assoc[symmetric] relcomp-mono)
also have  $\dots \subseteq (R \cup S)^* O (R \cup S)^* O R$ 
  using * by (simp add: relcomp-mono)
finally have  $?S O (R \cup S)^* \subseteq (R \cup S)^* O R$ 
  by (simp add: O-assoc[symmetric])
then have  $(?S O (R \cup S)^*) \text{ `` } A \subseteq ((R \cup S)^* O R) \text{ `` } A$ 
  by (simp add: Image-mono)
moreover have  $?X \subseteq (?S O (R \cup S)^*) \text{ `` } A$ 
  using A by (auto simp: relcomp-Image)
ultimately have  $?X \subseteq R \text{ `` } ?X$ 
  by (auto simp: relcomp-Image)
then have  $?X = \{\}$ 
  using  $\langle wf R \rangle$  by (simp add: wfE-pf)
moreover have  $A \subseteq ?X$  by auto
ultimately show  $A = \{\}$  by simp
qed
qed

```

corollary *wf-relcomp-compatible*:

assumes *wf R* and $R O S \subseteq S O R$

shows *wf (S O R)*

proof –

have $R O S \subseteq (R \cup S)^* O R$

using *assms* by *blast*

then have *wf (S* O R O S*)*

by (*simp add: assms qc-wf-relto-iff*)

then show ?thesis

by (*rule Wellfounded.wf-subset*) *blast*

qed

22.5 Acyclic relations

lemma *wf-acyclic*: $wf r \implies acyclic r$

by (simp add: acyclic-def) (blast elim: wf-trancl [THEN wf-irrefl])

lemmas wfP-acyclicP = wf-acyclic [to-pred]

22.5.1 Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf*:
 assumes *finite r acyclic r* shows *wf r*
 using *assms*
proof (*induction r rule: finite-induct*)
 case (*insert x r*)
 then show ?*case*
 by (*cases x*) *simp*
qed *simp*

lemma *finite-acyclic-wf-converse*: *finite r \implies acyclic r \implies wf (r⁻¹)*
 apply (*erule finite-converse [THEN iffD2, THEN finite-acyclic-wf]*)
 apply (*erule acyclic-converse [THEN iffD2]*)
 done

Observe that the converse of an irreflexive, transitive, and finite relation is again well-founded. Thus, we may employ it for well-founded induction.

lemma *wf-converse*:
 assumes *irrefl r and trans r and finite r*
 shows *wf (r⁻¹)*
proof –
 have *acyclic r*
 using \langle *irrefl r* \rangle and \langle *trans r* \rangle
 by (*simp add: irrefl-def acyclic-irrefl*)
 with \langle *finite r* \rangle show ?*thesis*
 by (*rule finite-acyclic-wf-converse*)
qed

lemma *wf-iff-acyclic-if-finite*: *finite r \implies wf r = acyclic r*
 by (*blast intro: finite-acyclic-wf wf-acyclic*)

22.6 nat is well-founded

lemma *less-nat-rel*: $(<) = (\lambda m n. n = \text{Suc } m)^{++}$
proof (*rule ext, rule ext, rule iffI*)
 fix *n m :: nat*
 show $(\lambda m n. n = \text{Suc } m)^{++} m n$ if $m < n$
 using *that*
proof (*induct n*)
 case 0
 then show ?*case* by *auto*
next
 case (*Suc n*)
 then show ?*case*

by (auto simp add: less-Suc-eq-le le-less intro: tranclp.trancl-into-trancl)
 qed
 show $m < n$ if $(\lambda m n. n = \text{Suc } m)^{++} m n$
 using that by (induct n) (simp-all add: less-Suc-eq-le reflexive le-less)
 qed

definition *pred-nat* :: $(\text{nat} \times \text{nat})$ set
 where $\text{pred-nat} = \{(m, n). n = \text{Suc } m\}$

definition *less-than* :: $(\text{nat} \times \text{nat})$ set
 where $\text{less-than} = \text{pred-nat}^+$

lemma *less-eq*: $(m, n) \in \text{pred-nat}^+ \longleftrightarrow m < n$
 unfolding *less-nat-rel pred-nat-def trancl-def* by *simp*

lemma *pred-nat-trancl-eq-le*: $(m, n) \in \text{pred-nat}^* \longleftrightarrow m \leq n$
 unfolding *less-eq rtrancl-eq-or-trancl* by *auto*

lemma *wf-pred-nat*: *wf pred-nat*
 unfolding *wf-def*

proof *clarify*

fix $P x$

assume $\forall x'. (\forall y. (y, x') \in \text{pred-nat} \longrightarrow P y) \longrightarrow P x'$

then show $P x$

unfolding *pred-nat-def* by (*induction x*) *blast+*

qed

lemma *wf-less-than* [*iff*]: *wf less-than*
 by (*simp add: less-than-def wf-pred-nat [THEN wf-trancl]*)

lemma *trans-less-than* [*iff*]: *trans less-than*
 by (*simp add: less-than-def*)

lemma *less-than-iff* [*iff*]: $((x, y) \in \text{less-than}) = (x < y)$
 by (*simp add: less-than-def less-eq*)

lemma *irrefl-less-than*: *irrefl less-than*
 using *irrefl-def* by *blast*

lemma *asym-less-than*: *asym less-than*
 by (*rule asymI*) *simp*

lemma *total-less-than*: *total less-than* and *total-on-less-than* [*simp*]: *total-on A less-than*
 using *total-on-def* by *force+*

lemma *wf-less*: *wf* $\{(x, y::\text{nat}). x < y\}$
 by (*rule Wellfounded.wellorder-class.wf*)

22.7 Accessible Part

Inductive definition of the accessible part $acc\ r$ of a relation; see also [6].

inductive-set $acc :: ('a \times 'a)\ set \Rightarrow 'a\ set$ **for** $r :: ('a \times 'a)\ set$
where $accI: (\bigwedge y. (y, x) \in r \Longrightarrow y \in acc\ r) \Longrightarrow x \in acc\ r$

abbreviation $termip :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$
where $termip\ r \equiv accp\ (r^{-1-1})$

abbreviation $termi :: ('a \times 'a)\ set \Rightarrow 'a\ set$
where $termi\ r \equiv acc\ (r^{-1})$

lemmas $accpI = accp.accI$

lemma $accp\text{-}eq\text{-}acc$ [code]: $accp\ r = (\lambda x. x \in Wellfounded.acc\ \{(x, y). r\ x\ y\})$
by (*simp add: acc-def*)

Induction rules

theorem $accp\text{-}induct$:

assumes *major*: $accp\ r\ a$
assumes *hyp*: $\bigwedge x. accp\ r\ x \Longrightarrow \forall y. r\ y\ x \longrightarrow P\ y \Longrightarrow P\ x$
shows $P\ a$
apply (*rule major* [THEN *accp.induct*])
apply (*rule hyp*)
apply (*rule accp.accI*)
apply *auto*
done

lemmas $accp\text{-}induct\text{-}rule = accp\text{-}induct$ [rule-format, induct set: *accp*]

theorem $accp\text{-}downward$: $accp\ r\ b \Longrightarrow r\ a\ b \Longrightarrow accp\ r\ a$
by (*cases rule: accp.cases*)

lemma $not\text{-}accp\text{-}down$:

assumes *na*: $\neg accp\ R\ x$
obtains z **where** $R\ z\ x$ **and** $\neg accp\ R\ z$

proof –

assume $a: \bigwedge z. R\ z\ x \Longrightarrow \neg accp\ R\ z \Longrightarrow thesis$

show *thesis*

proof (*cases* $\forall z. R\ z\ x \longrightarrow accp\ R\ z$)

case *True*

then have $\bigwedge z. R\ z\ x \Longrightarrow accp\ R\ z$ **by** *auto*

then have $accp\ R\ x$ **by** (*rule accp.accI*)

with *na* **show** *thesis* ..

next

case *False* **then obtain** z **where** $R\ z\ x$ **and** $\neg accp\ R\ z$

by *auto*

with a **show** *thesis* .

qed

qed

lemma *accp-downwards-aux*: $r^{**} b a \implies accp\ r\ a \longrightarrow accp\ r\ b$
by (*erule rtranclp-induct*) (*blast dest: accp-downward*)⁺

theorem *accp-downwards*: $accp\ r\ a \implies r^{**} b a \implies accp\ r\ b$
by (*blast dest: accp-downwards-aux*)

theorem *accp-wfPI*: $\forall x. accp\ r\ x \implies wfP\ r$

proof (*rule wfPUNIVI*)

fix $P\ x$

assume $\forall x. accp\ r\ x \ \forall x. (\forall y. r\ y\ x \longrightarrow P\ y) \longrightarrow P\ x$

then show $P\ x$

using *accp-induct*[**where** $P = P$] **by** *blast*

qed

theorem *accp-wfPD*: $wfP\ r \implies accp\ r\ x$

apply (*erule wfP-induct-rule*)

apply (*rule accp.accI*)

apply *blast*

done

theorem *wfP-accp-iff*: $wfP\ r = (\forall x. accp\ r\ x)$

by (*blast intro: accp-wfPI dest: accp-wfPD*)

Smaller relations have bigger accessible parts:

lemma *accp-subset*:

assumes $R1 \leq R2$

shows $accp\ R2 \leq accp\ R1$

proof (*rule predicateI1*)

fix x

assume $accp\ R2\ x$

then show $accp\ R1\ x$

proof (*induct x*)

fix x

assume $\bigwedge y. R2\ y\ x \implies accp\ R1\ y$

with *assms* **show** $accp\ R1\ x$

by (*blast intro: accp.accI*)

qed

qed

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:

assumes *subset*: $D \leq accp\ R$

and *dcl*: $\bigwedge x\ z. D\ x \implies R\ z\ x \implies D\ z$

and $D\ x$

and *istep*: $\bigwedge x. D\ x \implies (\bigwedge z. R\ z\ x \implies P\ z) \implies P\ x$

shows $P\ x$

```

proof –
  from subset and  $\langle D \ x \rangle$ 
  have accp  $R \ x \ ..$ 
  then show  $P \ x$  using  $\langle D \ x \rangle$ 
  proof (induct  $x$ )
    fix  $x$ 
    assume  $D \ x$  and  $\bigwedge y. R \ y \ x \implies D \ y \implies P \ y$ 
    with dcl and istep show  $P \ x$  by blast
  qed
qed

```

Set versions of the above theorems

```

lemmas acc-induct = accp-induct [to-set]
lemmas acc-induct-rule = acc-induct [rule-format, induct set: acc]
lemmas acc-downward = accp-downward [to-set]
lemmas not-acc-down = not-accp-down [to-set]
lemmas acc-downwards-aux = accp-downwards-aux [to-set]
lemmas acc-downwards = accp-downwards [to-set]
lemmas acc-wfI = accp-wfPI [to-set]
lemmas acc-wfD = accp-wfPD [to-set]
lemmas wf-acc-iff = wfP-accp-iff [to-set]
lemmas acc-subset = accp-subset [to-set]
lemmas acc-subset-induct = accp-subset-induct [to-set]

```

22.8 Tools for building wellfounded relations

Inverse Image

```

lemma wf-inv-image [simp,intro!]:
  fixes  $f :: 'a \Rightarrow 'b$ 
  assumes wf  $r$ 
  shows wf (inv-image  $r \ f$ )
proof –
  have  $\bigwedge x \ P. x \in P \implies \exists z \in P. \forall y. (f \ y, f \ z) \in r \longrightarrow y \notin P$ 
  proof –
    fix  $P$  and  $x :: 'a$ 
    assume  $x \in P$ 
    then obtain  $w$  where  $w \in \{w. \exists x :: 'a. x \in P \wedge f \ x = w\}$ 
    by auto
    have  $*$ :  $\bigwedge Q \ u. u \in Q \implies \exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q$ 
    using assms by (auto simp add: wf-eq-minimal)
    show  $\exists z \in P. \forall y. (f \ y, f \ z) \in r \longrightarrow y \notin P$ 
    using  $*$  [OF  $w$ ] by auto
  qed
then show ?thesis
  by (clarsimp simp: inv-image-def wf-eq-minimal)
qed

```

22.8.1 Conversion to a known well-founded relation

lemma *wf-if-convertible-to-wf*:

fixes $r :: 'a \text{ rel}$ **and** $s :: 'b \text{ rel}$ **and** $f :: 'a \Rightarrow 'b$

assumes *wf s* **and** *convertible*: $\bigwedge x y. (x, y) \in r \Longrightarrow (f x, f y) \in s$

shows *wf r*

proof (*rule wfI-min[of r]*)

fix $x :: 'a$ **and** $Q :: 'a \text{ set}$

assume $x \in Q$

then obtain y **where** $y \in Q$ **and** $\bigwedge z. (f z, f y) \in s \Longrightarrow z \notin Q$

by (*auto elim: wfE-min[OF wf-inv-image[of s f, OF ‹wf s›], unfolded in-inv-image]*)

thus $\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q$

by (*auto intro: convertible*)

qed

lemma *wfP-if-convertible-to-wfP*: $wfP S \Longrightarrow (\bigwedge x y. R x y \Longrightarrow S (f x) (f y)) \Longrightarrow wfP R$

using *wf-if-convertible-to-wf[to-pred, of S R f]* **by** *simp*

Converting to *nat* is a very common special case that might be found more easily by Sledgehammer.

lemma *wfP-if-convertible-to-nat*:

fixes $f :: - \Rightarrow nat$

shows $(\bigwedge x y. R x y \Longrightarrow f x < f y) \Longrightarrow wfP R$

by (*rule wfP-if-convertible-to-wfP[of (<) :: nat \Rightarrow nat \Rightarrow bool, simplified]*)

22.8.2 Measure functions into *nat*

definition *measure* :: $('a \Rightarrow nat) \Rightarrow ('a \times 'a) \text{ set}$

where *measure* = *inv-image less-than*

lemma *in-measure[simp, code-unfold]*: $(x, y) \in \text{measure } f \iff f x < f y$

by (*simp add:measure-def*)

lemma *wf-measure [iff]*: *wf (measure f)*

unfolding *measure-def* **by** (*rule wf-less-than [THEN wf-inv-image]*)

lemma *wf-if-measure*: $(\bigwedge x. P x \Longrightarrow f(g x) < f x) \Longrightarrow wf \{(y, x). P x \wedge y = g x\}$

for $f :: 'a \Rightarrow nat$

using *wf-measure[of f]* **unfolding** *measure-def inv-image-def less-than-def less-eq*

by (*rule wf-subset auto*)

22.8.3 Lexicographic combinations

definition *lex-prod* :: $('a \times 'a) \text{ set} \Rightarrow ('b \times 'b) \text{ set} \Rightarrow (('a \times 'b) \times ('a \times 'b)) \text{ set}$

(**infixr** *<*lex*>* 80)

where $ra \text{ <*lex*> } rb = \{((a, b), (a', b')). (a, a') \in ra \vee a = a' \wedge (b, b') \in rb\}$

lemma *in-lex-prod[simp]*: $((a, b), (a', b')) \in r \text{ <*lex*> } s \iff (a, a') \in r \vee a = a' \wedge (b, b') \in s$

by (auto simp:lex-prod-def)

lemma wf-lex-prod [intro!]:
 assumes wf ra wf rb
 shows wf (ra <*lex*> rb)
proof (rule wfI)
 fix z :: 'a × 'b and P
 assume * [rule-format]: $\forall u. (\forall v. (v, u) \in ra \text{ <*lex*> } rb \longrightarrow P v) \longrightarrow P u$
 obtain x y where zeq: $z = (x, y)$
 by fastforce
 have P(x,y) using ⟨wf ra⟩
proof (induction x arbitrary: y rule: wf-induct-rule)
 case (less x)
 note lessx = less
 show ?case using ⟨wf rb⟩ less
proof (induction y rule: wf-induct-rule)
 case (less y)
 show ?case
 by (force intro: * less.IH lessx)
 qed
 qed
 then show P z
 by (simp add: zeq)
 qed auto

lemma refl-lex-prod[simp]: $\text{refl } r_B \Longrightarrow \text{refl } (r_A \text{ <*lex*> } r_B)$
 by (auto intro!: reflI dest: refl-onD)

lemma irrefl-on-lex-prod[simp]:
 $\text{irrefl-on } A \ r_A \Longrightarrow \text{irrefl-on } B \ r_B \Longrightarrow \text{irrefl-on } (A \times B) \ (r_A \text{ <*lex*> } r_B)$
 by (auto intro!: irrefl-onI dest: irrefl-onD)

lemma irrefl-lex-prod[simp]: $\text{irrefl } r_A \Longrightarrow \text{irrefl } r_B \Longrightarrow \text{irrefl } (r_A \text{ <*lex*> } r_B)$
 by (rule irrefl-on-lex-prod[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lemma sym-on-lex-prod[simp]:
 $\text{sym-on } A \ r_A \Longrightarrow \text{sym-on } B \ r_B \Longrightarrow \text{sym-on } (A \times B) \ (r_A \text{ <*lex*> } r_B)$
 by (auto intro!: sym-onI dest: sym-onD)

lemma sym-lex-prod[simp]:
 $\text{sym } r_A \Longrightarrow \text{sym } r_B \Longrightarrow \text{sym } (r_A \text{ <*lex*> } r_B)$
 by (rule sym-on-lex-prod[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lemma asym-on-lex-prod[simp]:
 $\text{asym-on } A \ r_A \Longrightarrow \text{asym-on } B \ r_B \Longrightarrow \text{asym-on } (A \times B) \ (r_A \text{ <*lex*> } r_B)$
 by (auto intro!: asym-onI dest: asym-onD)

lemma asym-lex-prod[simp]:
 $\text{asym } r_A \Longrightarrow \text{asym } r_B \Longrightarrow \text{asym } (r_A \text{ <*lex*> } r_B)$

by (rule *asym-on-lex-prod*[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lemma *trans-on-lex-prod*[simp]:

assumes *trans-on* A r_A and *trans-on* B r_B

shows *trans-on* (A × B) ($r_A <*\text{lex}*> r_B$)

proof (rule *trans-onI*)

fix $x\ y\ z$

show $x \in A \times B \implies y \in A \times B \implies z \in A \times B \implies$

$(x, y) \in r_A <*\text{lex}*> r_B \implies (y, z) \in r_A <*\text{lex}*> r_B \implies (x, z) \in r_A <*\text{lex}*> r_B$

using *trans-onD*[OF $\langle \text{trans-on } A\ r_A \rangle$, of *fst* x *fst* y *fst* z]

using *trans-onD*[OF $\langle \text{trans-on } B\ r_B \rangle$, of *snd* x *snd* y *snd* z]

by *auto*

qed

lemma *trans-lex-prod* [simp,intro!]: *trans* $r_A \implies \text{trans } r_B \implies \text{trans } (r_A <*\text{lex}*> r_B)$

by (rule *trans-on-lex-prod*[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lemma *total-on-lex-prod*[simp]:

total-on A $r_A \implies \text{total-on } B\ r_B \implies \text{total-on } (A \times B) (r_A <*\text{lex}*> r_B)$

by (auto simp: *total-on-def*)

lemma *total-lex-prod*[simp]: *total* $r_A \implies \text{total } r_B \implies \text{total } (r_A <*\text{lex}*> r_B)$

by (rule *total-on-lex-prod*[of UNIV - UNIV, unfolded UNIV-Times-UNIV])

lexicographic combinations with measure functions

definition *mlex-prod* :: ($'a \Rightarrow \text{nat}$) $\Rightarrow ('a \times 'a)$ set $\Rightarrow ('a \times 'a)$ set (**infix** $<*\text{mlex}*> 80$)

where $f <*\text{mlex}*> R = \text{inv-image } (\text{less-than } <*\text{lex}*> R) (\lambda x. (f\ x, x))$

lemma

wf-mlex: $wf\ R \implies wf\ (f <*\text{mlex}*> R)$ and

mlex-less: $f\ x < f\ y \implies (x, y) \in f <*\text{mlex}*> R$ and

mlex-leq: $f\ x \leq f\ y \implies (x, y) \in R \implies (x, y) \in f <*\text{mlex}*> R$ and

mlex-iff: $(x, y) \in f <*\text{mlex}*> R \iff f\ x < f\ y \vee f\ x = f\ y \wedge (x, y) \in R$

by (auto simp: *mlex-prod-def*)

Proper subset relation on finite sets.

definition *finite-psubset* :: ($'a$ set $\times 'a$ set) set

where *finite-psubset* = $\{(A, B). A \subset B \wedge \text{finite } B\}$

lemma *wf-finite-psubset*[simp]: $wf\ \text{finite-psubset}$

apply (unfold *finite-psubset-def*)

apply (rule *wf-measure* [THEN *wf-subset*])

apply (simp add: *measure-def inv-image-def less-than-def less-eq*)

apply (fast elim!: *psubset-card-mono*)

done

lemma *trans-finite-psubset*: *trans finite-psubset*
by (*auto simp: finite-psubset-def less-le trans-def*)

lemma *in-finite-psubset[simp]*: $(A, B) \in \text{finite-psubset} \iff A \subset B \wedge \text{finite } B$
unfolding *finite-psubset-def* **by** *auto*

max- and min-extension of order to finite sets

inductive-set *max-ext* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ set} \times 'a \text{ set}) \text{ set}$
for $R :: ('a \times 'a) \text{ set}$
where *max-extI*[*intro*]:
 $\text{finite } X \implies \text{finite } Y \implies Y \neq \{\} \implies (\bigwedge x. x \in X \implies \exists y \in Y. (x, y) \in R) \implies$
 $(X, Y) \in \text{max-ext } R$

lemma *max-ext-wf*:

assumes *wf*: *wf r*
shows *wf* (*max-ext r*)

proof (*rule acc-wfI, intro allI*)

show $M \in \text{acc } (\text{max-ext } r)$ (**is** $- \in ?W$) **for** M

proof (*induct M rule: infinite-finite-induct*)

case *empty*

show *?case*

by (*rule accI*) (*auto elim: max-ext.cases*)

next

case (*insert a M*)

from *wf* $\langle M \in ?W \rangle \langle \text{finite } M \rangle$ **show** $\text{insert } a \ M \in ?W$

proof (*induct arbitrary: M*)

fix $M \ a$

assume $M \in ?W$

assume [*intro*]: *finite M*

assume *hyp*: $\bigwedge b \ M. (b, a) \in r \implies M \in ?W \implies \text{finite } M \implies \text{insert } b \ M \in ?W$

have *add-less*: $M \in ?W \implies (\bigwedge y. y \in N \implies (y, a) \in r) \implies N \cup M \in ?W$

if *finite N finite M* **for** $N \ M :: 'a \text{ set}$

using *that* **by** (*induct N arbitrary: M*) (*auto simp: hyp*)

show $\text{insert } a \ M \in ?W$

proof (*rule accI*)

fix N

assume *Nless*: $(N, \text{insert } a \ M) \in \text{max-ext } r$

then **have** $*$: $\bigwedge x. x \in N \implies (x, a) \in r \vee (\exists y \in M. (x, y) \in r)$

by (*auto elim!: max-ext.cases*)

let $?N1 = \{n \in N. (n, a) \in r\}$

let $?N2 = \{n \in N. (n, a) \notin r\}$

have N : $?N1 \cup ?N2 = N$ **by** (*rule set-eqI*) *auto*

from *Nless* **have** *finite N* **by** (*auto elim: max-ext.cases*)

then **have** *finites*: *finite ?N1 finite ?N2* **by** *auto*

have $?N2 \in ?W$

proof (*cases M = {}*)

```

    case [simp]: True
    have Mw: {} ∈ ?W by (rule accI) (auto elim: max-ext.cases)
    from * have ?N2 = {} by auto
    with Mw show ?N2 ∈ ?W by (simp only:)
next
  case False
  from * finites have N2: (?N2, M) ∈ max-ext r
    using max-extI[OF - - ⟨M ≠ {}⟩, where ?X = ?N2] by auto
  with ⟨M ∈ ?W⟩ show ?N2 ∈ ?W by (rule acc-downward)
qed
with finites have ?N1 ∪ ?N2 ∈ ?W
  by (rule add-less) simp
then show N ∈ ?W by (simp only: N)
qed
qed
next
  case infinite
  show ?case
    by (rule accI) (auto elim: max-ext.cases simp: infinite)
qed
qed

lemma max-ext-additive: (A, B) ∈ max-ext R ⇒ (C, D) ∈ max-ext R ⇒ (A ∪
C, B ∪ D) ∈ max-ext R
  by (force elim!: max-ext.cases)

definition min-ext :: ('a × 'a) set ⇒ ('a set × 'a set) set
  where min-ext r = {(X, Y) | X Y. X ≠ {} ∧ (∀ y ∈ Y. (∃ x ∈ X. (x, y) ∈ r))}

lemma min-ext-wf:
  assumes wf r
  shows wf (min-ext r)
proof (rule wfI-min)
  show ∃ m ∈ Q. (∀ n. (n, m) ∈ min-ext r → n ∉ Q) if nonempty: x ∈ Q
    for Q :: 'a set set and x
  proof (cases Q = {})
    case True
    then show ?thesis by (simp add: min-ext-def)
  next
    case False
    with nonempty obtain e x where x ∈ Q e ∈ x by force
    then have eU: e ∈ ∪ Q by auto
    with ⟨wf r⟩
    obtain z where z: z ∈ ∪ Q ∧ y. (y, z) ∈ r ⇒ y ∉ ∪ Q
      by (erule wfE-min)
    from z obtain m where m ∈ Q z ∈ m by auto
    from ⟨m ∈ Q⟩ show ?thesis
  proof (intro rev-bexI allI impI)
    fix n

```

```

    assume smaller: (n, m) ∈ min-ext r
    with ⟨z ∈ m⟩ obtain y where y ∈ n (y, z) ∈ r
      by (auto simp: min-ext-def)
    with z(2) show n ∉ Q by auto
  qed
  qed
  qed

```

22.8.4 Bounded increase must terminate

lemma *wf-bounded-measure*:

```

  fixes ub :: 'a ⇒ nat
    and f :: 'a ⇒ nat
  assumes ∧a b. (b, a) ∈ r ⇒ ub b ≤ ub a ∧ ub a ≥ f b ∧ f b > f a
  shows wf r
  by (rule wf-subset[OF wf-measure[of λa. ub a - f a]]) (auto dest: assms)

```

lemma *wf-bounded-set*:

```

  fixes ub :: 'a ⇒ 'b set
    and f :: 'a ⇒ 'b set
  assumes ∧a b. (b, a) ∈ r ⇒ finite (ub a) ∧ ub b ⊆ ub a ∧ ub a ⊇ f b ∧ f b ⊃ f
  a
  shows wf r
  apply (rule wf-bounded-measure[of r λa. card (ub a) λa. card (f a)])
  apply (drule assms)
  apply (blast intro: card-mono finite-subset psubset-card-mono dest: psubset-eq[THEN
  iffD2])
  done

```

lemma *finite-subset-wf*:

```

  assumes finite A
  shows wf {(X, Y). X ⊂ Y ∧ Y ⊆ A}
  by (rule wf-subset[OF wf-finite-psubset[unfolded finite-psubset-def]])
    (auto intro: finite-subset[OF - assms])

```

hide-const (open) *acc accp*

end

23 Well-Founded Recursion Combinator

theory *Wfrec*

imports *Wellfounded*

begin

inductive *wfrec-rel* :: ('a × 'a) set ⇒ (('a ⇒ 'b) ⇒ ('a ⇒ 'b)) ⇒ 'a ⇒ 'b ⇒ bool
for *R F*

where *wfrecI*: (∧z. (z, x) ∈ R ⇒ wfrec-rel R F z (g z)) ⇒ wfrec-rel R F x (F g x)

definition $cut :: ('a \Rightarrow 'b) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b$
where $cut\ f\ R\ x = (\lambda y. \text{if } (y, x) \in R \text{ then } f\ y \text{ else undefined})$

definition $adm\text{-}wf :: ('a \times 'a) \text{ set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow bool$
where $adm\text{-}wf\ R\ F \longleftrightarrow (\forall f\ g\ x. (\forall z. (z, x) \in R \longrightarrow f\ z = g\ z) \longrightarrow F\ f\ x = F\ g\ x)$

definition $wfrec :: ('a \times 'a) \text{ set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow ('a \Rightarrow 'b)$
where $wfrec\ R\ F = (\lambda x. \text{THE } y. wfrec\text{-}rel\ R\ (\lambda f\ x. F\ (cut\ f\ R\ x)\ x)\ x\ y)$

lemma $cuts\text{-}eq: (cut\ f\ R\ x = cut\ g\ R\ x) \longleftrightarrow (\forall y. (y, x) \in R \longrightarrow f\ y = g\ y)$
by (*simp add: fun-eq-iff cut-def*)

lemma $cut\text{-}apply: (x, a) \in R \Longrightarrow cut\ f\ R\ a\ x = f\ x$
by (*simp add: cut-def*)

Inductive characterization of $wfrec$ combinator; for details see: John Harrison, "Inductive definitions: automation and application".

lemma $theI\text{-}unique: \exists! x. P\ x \Longrightarrow P\ x \longleftrightarrow x = \text{The } P$
by (*auto intro: the-equality[symmetric] theI*)

lemma $wfrec\text{-}unique:$

assumes $adm\text{-}wf\ R\ F\ wf\ R$
shows $\exists! y. wfrec\text{-}rel\ R\ F\ x\ y$
using $\langle wf\ R \rangle$

proof *induct*

define f **where** $f\ y = (\text{THE } z. wfrec\text{-}rel\ R\ F\ y\ z)$ **for** y
case (*less x*)
then have $\bigwedge y\ z. (y, x) \in R \Longrightarrow wfrec\text{-}rel\ R\ F\ y\ z \longleftrightarrow z = f\ y$
unfolding $f\text{-}def$ **by** (*rule theI-unique*)
with $\langle adm\text{-}wf\ R\ F \rangle$ **show** $?case$
by (*subst wfrec-rel.simps*) (*auto simp: adm-wf-def*)

qed

lemma $adm\text{-}lemma: adm\text{-}wf\ R\ (\lambda f\ x. F\ (cut\ f\ R\ x)\ x)$
by (*auto simp: adm-wf-def intro!: arg-cong[where f= $\lambda x. F\ x\ y$ for y] cuts-eq[THEN iffD2]*)

lemma $wfrec: wf\ R \Longrightarrow wfrec\ R\ F\ a = F\ (cut\ (wfrec\ R\ F)\ R\ a)\ a$
apply (*simp add: wfrec-def*)
apply (*rule adm-lemma [THEN wfrec-unique, THEN the1-equality]*)
apply *assumption*
apply (*rule wfrec-rel.wfrecI*)
apply (*erule adm-lemma [THEN wfrec-unique, THEN theI']*)
done

This form avoids giant explosions in proofs. NOTE USE OF \equiv .

lemma $def\text{-}wfrec: f \equiv wfrec\ R\ F \Longrightarrow wf\ R \Longrightarrow f\ a = F\ (cut\ f\ R\ a)\ a$

by (auto intro: wfrec)

23.0.1 Well-founded recursion via genuine fixpoints

lemma *wfrec-fixpoint*:

assumes *wf*: *wf R*

and *adm*: *adm-wf R F*

shows $wfrec\ R\ F = F\ (wfrec\ R\ F)$

proof (rule ext)

fix *x*

have $wfrec\ R\ F\ x = F\ (cut\ (wfrec\ R\ F)\ R\ x)\ x$

using *wfrec[of R F] wf* by *simp*

also

have $\bigwedge y. (y, x) \in R \implies cut\ (wfrec\ R\ F)\ R\ x\ y = wfrec\ R\ F\ y$

by (auto simp add: cut-apply)

then have $F\ (cut\ (wfrec\ R\ F)\ R\ x)\ x = F\ (wfrec\ R\ F)\ x$

using *adm adm-wf-def[of R F]* by *auto*

finally show $wfrec\ R\ F\ x = F\ (wfrec\ R\ F)\ x$.

qed

lemma *wfrec-def-adm*: $f \equiv wfrec\ R\ F \implies wf\ R \implies adm-wf\ R\ F \implies f = F\ f$

using *wfrec-fixpoint* by *simp*

23.1 Wellfoundedness of same-fst

definition *same-fst* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow ('b \times 'b)\ set) \Rightarrow (('a \times 'b) \times ('a \times 'b))\ set$

where *same-fst* *P R* = $\{((x', y'), (x, y)) \cdot x' = x \wedge P\ x \wedge (y', y) \in R\ x\}$

— For *wfrec* declarations where the first *n* parameters stay unchanged in the recursive call.

lemma *same-fstI [intro!]*: $P\ x \implies (y', y) \in R\ x \implies ((x, y'), (x, y)) \in same-fst\ P\ R$

by (simp add: *same-fst-def*)

lemma *wf-same-fst*:

assumes $\bigwedge x. P\ x \implies wf\ (R\ x)$

shows $wf\ (same-fst\ P\ R)$

proof –

have $\bigwedge a\ b\ Q. \forall a\ b. (\forall x. P\ a \wedge (x, b) \in R\ a \longrightarrow Q\ (a, x)) \longrightarrow Q\ (a, b) \implies Q\ (a, b)$

proof –

fix *Q a b*

assume *: $\forall a\ b. (\forall x. P\ a \wedge (x, b) \in R\ a \longrightarrow Q\ (a, x)) \longrightarrow Q\ (a, b)$

show $Q(a, b)$

proof (cases *wf (R a)*)

case *True*

then show *?thesis*

by (induction *b* rule: *wf-induct-rule*) (use * in *blast*)

qed (use * *assms* in *blast*)

```

qed
then show ?thesis
  by (clarsimp simp add: wf-def same-fst-def)
qed
end

```

24 Orders as Relations

```

theory Order-Relation
imports Wfrec
begin

```

24.1 Orders on a set

definition *preorder-on* A $r \equiv \text{refl-on } A \ r \ \wedge \ \text{trans } r$

definition *partial-order-on* A $r \equiv \text{preorder-on } A \ r \ \wedge \ \text{antisym } r$

definition *linear-order-on* A $r \equiv \text{partial-order-on } A \ r \ \wedge \ \text{total-on } A \ r$

definition *strict-linear-order-on* A $r \equiv \text{trans } r \ \wedge \ \text{irrefl } r \ \wedge \ \text{total-on } A \ r$

definition *well-order-on* A $r \equiv \text{linear-order-on } A \ r \ \wedge \ \text{wf}(r - \text{Id})$

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def
strict-linear-order-on-def well-order-on-def

lemma *partial-order-onD*:
assumes *partial-order-on* A r **shows** *refl-on* A r **and** *trans* r **and** *antisym* r
using *assms* **unfolding** *partial-order-on-def preorder-on-def* **by** *auto*

lemma *preorder-on-empty[simp]*: *preorder-on* $\{\}$ $\{\}$
by (*simp add: preorder-on-def trans-def*)

lemma *partial-order-on-empty[simp]*: *partial-order-on* $\{\}$ $\{\}$
by (*simp add: partial-order-on-def*)

lemma *linear-order-on-empty[simp]*: *linear-order-on* $\{\}$ $\{\}$
by (*simp add: linear-order-on-def*)

lemma *well-order-on-empty[simp]*: *well-order-on* $\{\}$ $\{\}$
by (*simp add: well-order-on-def*)

lemma *preorder-on-converse[simp]*: *preorder-on* A $(r^{-1}) = \text{preorder-on } A \ r$
by (*simp add: preorder-on-def*)

lemma *partial-order-on-converse*[simp]: *partial-order-on* A $(r^{-1}) = \text{partial-order-on } A \ r$

by (*simp add: partial-order-on-def*)

lemma *linear-order-on-converse*[simp]: *linear-order-on* A $(r^{-1}) = \text{linear-order-on } A \ r$

by (*simp add: linear-order-on-def*)

lemma *partial-order-on-acyclic*:

partial-order-on $A \ r \implies \text{acyclic } (r - \text{Id})$

by (*simp add: acyclic-irrefl partial-order-on-def preorder-on-def trans-diff-Id*)

lemma *partial-order-on-well-order-on*:

finite $r \implies \text{partial-order-on } A \ r \implies \text{wf } (r - \text{Id})$

by (*simp add: finite-acyclic-wf partial-order-on-acyclic*)

lemma *strict-linear-order-on-diff-Id*: *linear-order-on* $A \ r \implies \text{strict-linear-order-on } A \ (r - \text{Id})$

by (*simp add: order-on-defs trans-diff-Id*)

lemma *linear-order-on-singleton* [simp]: *linear-order-on* $\{x\} \{(x, x)\}$

by (*simp add: order-on-defs*)

lemma *linear-order-on-acyclic*:

assumes *linear-order-on* $A \ r$

shows *acyclic* $(r - \text{Id})$

using *strict-linear-order-on-diff-Id*[OF *assms*]

by (*auto simp add: acyclic-irrefl strict-linear-order-on-def*)

lemma *linear-order-on-well-order-on*:

assumes *finite* r

shows *linear-order-on* $A \ r \longleftrightarrow \text{well-order-on } A \ r$

unfolding *well-order-on-def*

using *assms finite-acyclic-wf*[OF - *linear-order-on-acyclic, of r*] **by** *blast*

24.2 Orders on the field

abbreviation *Refl* $r \equiv \text{refl-on } (\text{Field } r) \ r$

abbreviation *Preorder* $r \equiv \text{preorder-on } (\text{Field } r) \ r$

abbreviation *Partial-order* $r \equiv \text{partial-order-on } (\text{Field } r) \ r$

abbreviation *Total* $r \equiv \text{total-on } (\text{Field } r) \ r$

abbreviation *Linear-order* $r \equiv \text{linear-order-on } (\text{Field } r) \ r$

abbreviation *Well-order* $r \equiv \text{well-order-on } (\text{Field } r) \ r$

lemma *subset-Image-Image-iff*:

Preorder $r \implies A \subseteq \text{Field } r \implies B \subseteq \text{Field } r \implies$

$r \text{ “ } A \subseteq r \text{ “ } B \iff (\forall a \in A. \exists b \in B. (b, a) \in r)$

apply (*simp add: preorder-on-def refl-on-def Image-def subset-eq*)

apply (*simp only: trans-def*)

apply *fast*

done

lemma *subset-Image1-Image1-iff*:

Preorder $r \implies a \in \text{Field } r \implies b \in \text{Field } r \implies r \text{ “ } \{a\} \subseteq r \text{ “ } \{b\} \iff (b, a) \in r$

by (*simp add: subset-Image-Image-iff*)

lemma *Refl-antisym-eq-Image1-Image1-iff*:

assumes *Refl* r

and *as: antisym* r

and *abf: a ∈ Field r b ∈ Field r*

shows $r \text{ “ } \{a\} = r \text{ “ } \{b\} \iff a = b$

(*is ?lhs* \iff *?rhs*)

proof

assume *?lhs*

then have $*$: $\bigwedge x. (a, x) \in r \iff (b, x) \in r$

by (*simp add: set-eq-iff*)

have $(a, a) \in r$ $(b, b) \in r$ **using** $\langle \text{Refl } r \rangle$ *abf* **by** (*simp-all add: refl-on-def*)

then have $(a, b) \in r$ $(b, a) \in r$ **using** $*$ [*of a*] $*$ [*of b*] **by** *simp-all*

then show *?rhs*

using $\langle \text{antisym } r \rangle$ [*unfolded antisym-def*] **by** *blast*

next

assume *?rhs*

then show *?lhs* **by** *fast*

qed

lemma *Partial-order-eq-Image1-Image1-iff*:

Partial-order $r \implies a \in \text{Field } r \implies b \in \text{Field } r \implies r \text{ “ } \{a\} = r \text{ “ } \{b\} \iff a = b$

by (*auto simp: order-on-defs Refl-antisym-eq-Image1-Image1-iff*)

lemma *Total-Id-Field*:

assumes *Total* r

and *not-Id: $\neg r \subseteq \text{Id}$*

shows $\text{Field } r = \text{Field } (r - \text{Id})$

proof –

have $\text{Field } r \subseteq \text{Field } (r - \text{Id})$

proof (*rule subsetI*)

fix a **assume** $*$: $a \in \text{Field } r$

from *not-Id* **have** $r \neq \{\}$ **by** *fast*

with *not-Id* **obtain** b **and** c **where** $b \neq c \wedge (b, c) \in r$ **by** *auto*

then have $b \neq c \wedge \{b, c\} \subseteq \text{Field } r$ **by** (*auto simp: Field-def*)
with * **obtain** d **where** $d \in \text{Field } r$ $d \neq a$ **by** *auto*
with * $\langle \text{Total } r \rangle$ **have** $(a, d) \in r \vee (d, a) \in r$ **by** (*simp add: total-on-def*)
with $\langle d \neq a \rangle$ **show** $a \in \text{Field } (r - \text{Id})$ **unfolding** *Field-def* **by** *blast*
qed
then show *?thesis*
using *mono-Field*[*of* $r - \text{Id}$ r] *Diff-subset*[*of* r Id] **by** *auto*
qed

24.3 Relations given by a predicate and the field

definition *relation-of* $:: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow ('a \times 'a) \text{ set}$
where *relation-of* $P A \equiv \{ (a, b) \in A \times A. P a b \}$

lemma *Field-relation-of*:
assumes *refl-on* A (*relation-of* $P A$) **shows** $\text{Field } (\text{relation-of } P A) = A$
using *assms* **unfolding** *refl-on-def* *Field-def* **by** *auto*

lemma *partial-order-on-relation-ofI*:
assumes *refl*: $\bigwedge a. a \in A \Longrightarrow P a a$
and *trans*: $\bigwedge a b c. \llbracket a \in A; b \in A; c \in A \rrbracket \Longrightarrow P a b \Longrightarrow P b c \Longrightarrow P a c$
and *antisym*: $\bigwedge a b. \llbracket a \in A; b \in A \rrbracket \Longrightarrow P a b \Longrightarrow P b a \Longrightarrow a = b$
shows *partial-order-on* A (*relation-of* $P A$)

proof –

from *refl* **have** *refl-on* A (*relation-of* $P A$)
unfolding *refl-on-def* *relation-of-def* **by** *auto*
moreover **have** *trans* (*relation-of* $P A$) **and** *antisym* (*relation-of* $P A$)
unfolding *relation-of-def*
by (*auto intro: transI dest: trans, auto intro: antisymI dest: antisym*)
ultimately show *?thesis*
unfolding *partial-order-on-def* *preorder-on-def* **by** *simp*
qed

lemma *Partial-order-relation-ofI*:
assumes *partial-order-on* A (*relation-of* $P A$) **shows** *Partial-order* (*relation-of* $P A$)
using *Field-relation-of* *assms* *partial-order-on-def* *preorder-on-def* **by** *fastforce*

24.4 Orders on a type

abbreviation *strict-linear-order* \equiv *strict-linear-order-on UNIV*

abbreviation *linear-order* \equiv *linear-order-on UNIV*

abbreviation *well-order* \equiv *well-order-on UNIV*

24.5 Order-like relations

In this subsection, we develop basic concepts and results pertaining to order-like relations, i.e., to reflexive and/or transitive and/or symmetric and/or total relations. We also further define upper and lower bounds operators.

24.5.1 Auxiliaries

lemma *refl-on-domain*: $\text{refl-on } A \ r \implies (a, b) \in r \implies a \in A \wedge b \in A$
by (*auto simp add: refl-on-def*)

corollary *well-order-on-domain*: $\text{well-order-on } A \ r \implies (a, b) \in r \implies a \in A \wedge b \in A$
by (*auto simp add: refl-on-domain order-on-defs*)

lemma *well-order-on-Field*: $\text{well-order-on } A \ r \implies A = \text{Field } r$
by (*auto simp add: refl-on-def Field-def order-on-defs*)

lemma *well-order-on-Well-order*: $\text{well-order-on } A \ r \implies A = \text{Field } r \wedge \text{Well-order } r$
using *well-order-on-Field [of A]* **by** *auto*

lemma *Total-subset-Id*:

assumes *Total* r

and $r \subseteq \text{Id}$

shows $r = \{\} \vee (\exists a. r = \{(a, a)\})$

proof –

have $\exists a. r = \{(a, a)\}$ **if** $r \neq \{\}$

proof –

from *that* **obtain** $a \ b$ **where** $ab: (a, b) \in r$ **by** *fast*

with $\langle r \subseteq \text{Id} \rangle$ **have** $a = b$ **by** *blast*

with ab **have** $aa: (a, a) \in r$ **by** *simp*

have $a = c \wedge a = d$ **if** $(c, d) \in r$ **for** $c \ d$

proof –

from *that* **have** $\{a, c, d\} \subseteq \text{Field } r$

using ab **unfolding** *Field-def* **by** *blast*

then **have** $((a, c) \in r \vee (c, a) \in r \vee a = c) \wedge ((a, d) \in r \vee (d, a) \in r \vee a = d)$

using $\langle \text{Total } r \rangle$ **unfolding** *total-on-def* **by** *blast*

with $\langle r \subseteq \text{Id} \rangle$ **show** *?thesis* **by** *blast*

qed

then **have** $r \subseteq \{(a, a)\}$ **by** *auto*

with aa **show** *?thesis* **by** *blast*

qed

then **show** *?thesis* **by** *blast*

qed

lemma *Linear-order-in-diff-Id*:

assumes *Linear-order* r

and $a \in \text{Field } r$
and $b \in \text{Field } r$
shows $(a, b) \in r \iff (b, a) \notin r - \text{Id}$
using *assms* **unfolding** *order-on-defs total-on-def antisym-def Id-def refl-on-def*
by *force*

24.5.2 The upper and lower bounds operators

Here we define upper (“above”) and lower (“below”) bounds operators. We think of r as a *non-strict* relation. The suffix S at the names of some operators indicates that the bounds are strict – e.g., $\text{underS } a$ is the set of all strict lower bounds of a (w.r.t. r). Capitalization of the first letter in the name reminds that the operator acts on sets, rather than on individual elements.

definition $\text{under} :: 'a \text{ rel} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where $\text{under } r \ a \equiv \{b. (b, a) \in r\}$

definition $\text{underS} :: 'a \text{ rel} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where $\text{underS } r \ a \equiv \{b. b \neq a \wedge (b, a) \in r\}$

definition $\text{Under} :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $\text{Under } r \ A \equiv \{b \in \text{Field } r. \forall a \in A. (b, a) \in r\}$

definition $\text{UnderS} :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $\text{UnderS } r \ A \equiv \{b \in \text{Field } r. \forall a \in A. b \neq a \wedge (b, a) \in r\}$

definition $\text{above} :: 'a \text{ rel} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where $\text{above } r \ a \equiv \{b. (a, b) \in r\}$

definition $\text{aboveS} :: 'a \text{ rel} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where $\text{aboveS } r \ a \equiv \{b. b \neq a \wedge (a, b) \in r\}$

definition $\text{Above} :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $\text{Above } r \ A \equiv \{b \in \text{Field } r. \forall a \in A. (a, b) \in r\}$

definition $\text{AboveS} :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
where $\text{AboveS } r \ A \equiv \{b \in \text{Field } r. \forall a \in A. b \neq a \wedge (a, b) \in r\}$

definition $\text{ofilter} :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$
where $\text{ofilter } r \ A \equiv A \subseteq \text{Field } r \wedge (\forall a \in A. \text{under } r \ a \subseteq A)$

Note: In the definitions of $\text{Above}[S]$ and $\text{Under}[S]$, we bounded comprehension by $\text{Field } r$ in order to properly cover the case of A being empty.

lemma $\text{underS-subset-under}: \text{underS } r \ a \subseteq \text{under } r \ a$
by (*auto simp add: underS-def under-def*)

lemma $\text{underS-notIn}: a \notin \text{underS } r \ a$
by (*simp add: underS-def*)

lemma *Refl-under-in*: $\text{Refl } r \implies a \in \text{Field } r \implies a \in \text{under } r a$
by (*simp add: refl-on-def under-def*)

lemma *AboveS-disjoint*: $A \cap (\text{AboveS } r A) = \{\}$
by (*auto simp add: AboveS-def*)

lemma *in-AboveS-underS*: $a \in \text{Field } r \implies a \in \text{AboveS } r (\text{underS } r a)$
by (*auto simp add: AboveS-def underS-def*)

lemma *Refl-under-underS*: $\text{Refl } r \implies a \in \text{Field } r \implies \text{under } r a = \text{underS } r a \cup \{a\}$
unfolding *under-def underS-def*
using *refl-on-def[of - r]* **by** *fastforce*

lemma *underS-empty*: $a \notin \text{Field } r \implies \text{underS } r a = \{\}$
by (*auto simp: Field-def underS-def*)

lemma *under-Field*: $\text{under } r a \subseteq \text{Field } r$
by (*auto simp: under-def Field-def*)

lemma *underS-Field*: $\text{underS } r a \subseteq \text{Field } r$
by (*auto simp: underS-def Field-def*)

lemma *underS-Field2*: $a \in \text{Field } r \implies \text{underS } r a \subset \text{Field } r$
using *underS-notIn underS-Field* **by** *fast*

lemma *underS-Field3*: $\text{Field } r \neq \{\} \implies \text{underS } r a \subset \text{Field } r$
by (*cases a \in Field r*) (*auto simp: underS-Field2 underS-empty*)

lemma *AboveS-Field*: $\text{AboveS } r A \subseteq \text{Field } r$
by (*auto simp: AboveS-def Field-def*)

lemma *under-incr*:
assumes *trans r*
and $(a, b) \in r$
shows $\text{under } r a \subseteq \text{under } r b$
unfolding *under-def*
proof *safe*
fix x **assume** $(x, a) \in r$
with *assms trans-def[of r]* **show** $(x, b) \in r$ **by** *blast*
qed

lemma *underS-incr*:
assumes *trans r*
and *antisym r*
and $ab: (a, b) \in r$
shows $\text{underS } r a \subseteq \text{underS } r b$
unfolding *underS-def*

```

proof safe
  assume *:  $b \neq a$  and **:  $(b, a) \in r$ 
  with  $\langle antisym\ r \rangle$  antisym-def[of r]  $ab$  show False
    by blast
next
  fix  $x$  assume  $x \neq a$   $(x, a) \in r$ 
  with  $ab$   $\langle trans\ r \rangle$  trans-def[of r] show  $(x, b) \in r$ 
    by blast
qed

lemma underS-incl-iff:
  assumes LO: Linear-order r
    and INa:  $a \in Field\ r$ 
    and INb:  $b \in Field\ r$ 
  shows  $underS\ r\ a \subseteq underS\ r\ b \longleftrightarrow (a, b) \in r$ 
    (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs
  with  $\langle Linear-order\ r \rangle$  show ?lhs
    by (simp add: order-on-defs underS-incr)
next
  assume *: ?lhs
  have  $(a, b) \in r$  if  $a = b$ 
    using assms that by (simp add: order-on-defs refl-on-def)
  moreover have False if  $a \neq b$   $(b, a) \in r$ 
  proof –
    from that have  $b \in underS\ r\ a$  unfolding underS-def by blast
    with * have  $b \in underS\ r\ b$  by blast
    then show ?thesis by (simp add: underS-notIn)
  qed
  ultimately show  $(a, b) \in r$ 
    using assms order-on-defs[of Field r r] total-on-def[of Field r r] by blast
qed

lemma finite-Partial-order-induct[consumes 3, case-names step]:
  assumes Partial-order r
    and  $x \in Field\ r$ 
    and finite r
    and step:  $\bigwedge x. x \in Field\ r \implies (\bigwedge y. y \in aboveS\ r\ x \implies P\ y) \implies P\ x$ 
  shows  $P\ x$ 
  using assms(2)
proof (induct rule: wf-induct[of  $r^{-1} - Id$ ])
  case 1
  from assms(1,3) show  $wf\ (r^{-1} - Id)$ 
    using partial-order-on-well-order-on partial-order-on-converse by blast
next
  case prems: (2 x)
  show ?case
    by (rule step) (use prems in  $\langle auto\ simp: aboveS-def\ intro: FieldI2 \rangle$ )

```

qed

lemma *finite-Linear-order-induct*[consumes 3, case-names step]:

assumes *Linear-order* r
and $x \in \text{Field } r$
and *finite* r
and *step*: $\bigwedge x. x \in \text{Field } r \implies (\bigwedge y. y \in \text{aboveS } r \ x \implies P \ y) \implies P \ x$
shows $P \ x$
using *assms*(2)
proof (*induct rule*: *wf-induct*[of $r^{-1} - \text{Id}$])
case 1
from *assms*(1,3) **show** *wf* ($r^{-1} - \text{Id}$)
using *linear-order-on-well-order-on* *linear-order-on-converse*
unfolding *well-order-on-def* **by** *blast*
next
case *prems*: (2 x)
show ?*case*
by (*rule step*) (*use prems in* $\langle \text{auto simp: aboveS-def intro: FieldI2} \rangle$)
 qed

24.6 Variations on Well-Founded Relations

This subsection contains some variations of the results from *HOL.Wellfounded*:

- means for slightly more direct definitions by well-founded recursion;
- variations of well-founded induction;
- means for proving a linear order to be a well-order.

24.6.1 Characterizations of well-foundedness

A transitive relation is well-founded iff it is “locally” well-founded, i.e., iff its restriction to the lower bounds of any element is well-founded.

lemma *trans-wf-iff*:

assumes *trans* r
shows $\text{wf } r \iff (\forall a. \text{wf } (r \cap (r^{-1} \{a\} \times r^{-1} \{a\})))$
proof –
define R **where** $R \ a = r \cap (r^{-1} \{a\} \times r^{-1} \{a\})$ **for** a
have $\text{wf } (R \ a)$ **if** $\text{wf } r$ **for** a
using *that R-def wf-subset*[of $r \ R \ a$] **by** *auto*
moreover
have $\text{wf } r$ **if** $\ast: \forall a. \text{wf } (R \ a)$
unfolding *wf-def*
proof *clarify*
fix $\text{phi } a$
assume $\ast\ast: \forall a. (\forall b. (b, a) \in r \longrightarrow \text{phi } b) \longrightarrow \text{phi } a$

```

define chi where  $chi\ b \longleftrightarrow (b, a) \in r \longrightarrow phi\ b$  for b
with * have  $wf\ (R\ a)$  by auto
then have  $(\forall b. (\forall c. (c, b) \in R\ a \longrightarrow chi\ c) \longrightarrow chi\ b) \longrightarrow (\forall b. chi\ b)$ 
  unfolding wf-def by blast
also have  $\forall b. (\forall c. (c, b) \in R\ a \longrightarrow chi\ c) \longrightarrow chi\ b$ 
proof safe
  fix b
  assume  $\forall c. (c, b) \in R\ a \longrightarrow chi\ c$ 
  moreover have  $(b, a) \in r \implies \forall c. (c, b) \in r \wedge (c, a) \in r \longrightarrow phi\ c \implies phi\ b$ 
proof –
  assume  $(b, a) \in r$  and  $\forall c. (c, b) \in r \wedge (c, a) \in r \longrightarrow phi\ c$ 
  then have  $\forall c. (c, b) \in r \longrightarrow phi\ c$ 
    using assms trans-def[of r] by blast
  with ** show  $phi\ b$  by blast
qed
ultimately show  $chi\ b$ 
  by (auto simp add: chi-def R-def)
qed
finally have  $\forall b. chi\ b$  .
with ** chi-def show  $phi\ a$  by blast
qed
ultimately show ?thesis unfolding R-def by blast
qed

```

A transitive relation is well-founded if all initial segments are finite.

corollary *wf-finite-segments*:

```

assumes irrefl r and trans r and  $\bigwedge x. finite\ \{y. (y, x) \in r\}$ 
shows  $wf\ r$ 
proof –
  have  $\bigwedge a. acyclic\ (r \cap \{x. (x, a) \in r\} \times \{x. (x, a) \in r\})$ 
  proof –
    fix a
    have  $trans\ (r \cap (\{x. (x, a) \in r\} \times \{x. (x, a) \in r\}))$ 
      using assms unfolding trans-def Field-def by blast
    then show  $acyclic\ (r \cap \{x. (x, a) \in r\} \times \{x. (x, a) \in r\})$ 
      using assms acyclic-def assms irrefl-def by fastforce
  qed
  then show ?thesis
  by (clarsimp simp: trans-wf-iff wf-iff-acyclic-if-finite converse-def assms)
qed

```

The next lemma is a variation of *wf-eg-minimal* from Wellfounded, allowing one to assume the set included in the field.

lemma *wf-eg-minimal2*: $wf\ r \longleftrightarrow (\forall A. A \subseteq Field\ r \wedge A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a', a) \notin r))$

```

proof –
  let  $?phi = \lambda A. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a', a) \notin r)$ 
  have  $wf\ r \longleftrightarrow (\forall A. ?phi\ A)$ 

```

```

proof
  assume wf r
  show  $\forall A. ?phi A$ 
  proof clarify
    fix A :: 'a set
    assume  $A \neq \{\}$ 
    then obtain x where  $x \in A$ 
    by auto
    show  $\exists a \in A. \forall a' \in A. (a', a) \notin r$ 
    apply (rule wfE-min[of r x A])
    apply fact+
    by blast
  qed
next
  assume *:  $\forall A. ?phi A$ 
  then show wf r
    apply (clarsimp simp: ex-in-conv [THEN sym])
    apply (rule wfI-min)
    by fast
  qed
also have  $(\forall A. ?phi A) \longleftrightarrow (\forall B \subseteq Field\ r. ?phi B)$ 
proof
  assume  $\forall A. ?phi A$ 
  then show  $\forall B \subseteq Field\ r. ?phi B$  by simp
next
  assume *:  $\forall B \subseteq Field\ r. ?phi B$ 
  show  $\forall A. ?phi A$ 
  proof clarify
    fix A :: 'a set
    assume **:  $A \neq \{\}$ 
    define B where  $B = A \cap Field\ r$ 
    show  $\exists a \in A. \forall a' \in A. (a', a) \notin r$ 
    proof (cases B = \{\})
      case True
        with ** obtain a where  $a \in A\ a \notin Field\ r$ 
        unfolding B-def by blast
        with a have  $\forall a' \in A. (a', a) \notin r$ 
        unfolding Field-def by blast
        with a show ?thesis by blast
      next
        case False
        have  $B \subseteq Field\ r$  unfolding B-def by blast
        with False * obtain a where  $a \in B\ \forall a' \in B. (a', a) \notin r$ 
        by blast
        have  $(a', a) \notin r$  if  $a' \in A$  for a'
        proof
          assume a'a:  $(a', a) \in r$ 
          with that have  $a' \in B$  unfolding B-def Field-def by blast
          with a a'a show False by blast
        qed
    qed
  qed

```

```

    qed
    with a show ?thesis unfolding B-def by blast
  qed
  qed
  qed
  finally show ?thesis by blast
qed

```

24.6.2 Characterizations of well-foundedness

The next lemma and its corollary enable one to prove that a linear order is a well-order in a way which is more standard than via well-foundedness of the strict version of the relation.

lemma *Linear-order-wf-diff-Id*:

assumes *Linear-order r*

shows $wf (r - Id) \longleftrightarrow (\forall A \subseteq Field\ r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r))$

proof (*cases* $r \subseteq Id$)

case *True*

then have $*$: $r - Id = \{\}$ **by** *blast*

have $wf (r - Id)$ **by** (*simp add: **)

moreover have $\exists a \in A. \forall a' \in A. (a, a') \in r$

if $*$: $A \subseteq Field\ r$ **and** $**$: $A \neq \{\}$ **for** A

proof –

from $\langle Linear-order\ r \rangle$ *True*

obtain a **where** $a: r = \{\} \vee r = \{(a, a)\}$

unfolding *order-on-defs* **using** *Total-subset-Id [of r]* **by** *blast*

with $*$ $**$ **have** $A = \{a\} \wedge r = \{(a, a)\}$

unfolding *Field-def* **by** *blast*

with a **show** ?thesis **by** *blast*

qed

ultimately show ?thesis **by** *blast*

next

case *False*

with $\langle Linear-order\ r \rangle$ **have** *Field*: $Field\ r = Field\ (r - Id)$

unfolding *order-on-defs* **using** *Total-Id-Field [of r]* **by** *blast*

show ?thesis

proof

assume $*$: $wf (r - Id)$

show $\forall A \subseteq Field\ r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r)$

proof *clarify*

fix A

assume $**$: $A \subseteq Field\ r$ **and** $***$: $A \neq \{\}$

then have $\exists a \in A. \forall a' \in A. (a', a) \notin r - Id$

using *Field ** **unfolding** *wf-eq-minimal2* **by** *simp*

moreover have $\forall a \in A. \forall a' \in A. (a, a') \in r \longleftrightarrow (a', a) \notin r - Id$

using *Linear-order-in-diff-Id [OF $\langle Linear-order\ r \rangle$]* $**$ **by** *blast*

ultimately show $\exists a \in A. \forall a' \in A. (a, a') \in r$ **by** *blast*

qed

```

next
  assume *:  $\forall A \subseteq \text{Field } r. A \neq \{\}$   $\longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r)$ 
  show wf (r - Id)
    unfolding wf-eq-minimal2
  proof clarify
    fix A
    assume **:  $A \subseteq \text{Field}(r - \text{Id})$  and ***:  $A \neq \{\}$ 
    then have  $\exists a \in A. \forall a' \in A. (a, a') \in r$ 
      using Field * by simp
    moreover have  $\forall a \in A. \forall a' \in A. (a, a') \in r \longleftrightarrow (a', a) \notin r - \text{Id}$ 
      using Linear-order-in-diff-Id [OF ‹Linear-order r›] ** mono-Field[of r -
Id r] by blast
    ultimately show  $\exists a \in A. \forall a' \in A. (a', a) \notin r - \text{Id}$ 
      by blast
    qed
  qed
qed

corollary Linear-order-Well-order-iff:
  Linear-order r  $\implies$ 
  Well-order r  $\longleftrightarrow (\forall A \subseteq \text{Field } r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r))$ 
  unfolding well-order-on-def using Linear-order-wf-diff-Id[of r] by blast

end

```

25 Hilbert’s Epsilon-Operator and the Axiom of Choice

```

theory Hilbert-Choice
  imports Wellfounded
  keywords specification :: thy-goal-defn
begin

```

25.1 Hilbert’s epsilon

```

axiomatization Eps :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a
  where someI:  $P x \implies P (Eps P)$ 

syntax (epsilon)
  -Eps :: ptrn  $\Rightarrow$  bool  $\Rightarrow$  'a (( $\exists \epsilon$  -./ -) [0, 10] 10)
syntax (input)
  -Eps :: ptrn  $\Rightarrow$  bool  $\Rightarrow$  'a (( $\exists @$  -./ -) [0, 10] 10)
syntax
  -Eps :: ptrn  $\Rightarrow$  bool  $\Rightarrow$  'a (( $\exists \text{SOME}$  -./ -) [0, 10] 10)
translations
  SOME x. P  $\equiv$  CONST Eps ( $\lambda x. P$ )

print-translation ‹

```

[[**const-syntax** $\langle Eps \rangle$, $fn - => fn [Abs abs] =>$
 $let\ val\ (x, t) = Syntax-Trans.atomic-abs-tr'\ abs$
 $in\ Syntax.const\ \mathbf{syntax-const}\ \langle -Eps \rangle\ \$\ x\ \$\ t\ end$]]
 \rangle — to avoid eta-contraction of body

definition $inv\text{-}into :: 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$ **where**
 $inv\text{-}into\ A\ f = (\lambda x. SOME\ y. y \in A \wedge f\ y = x)$

lemma $inv\text{-}into\text{-}def2$: $inv\text{-}into\ A\ f\ x = (SOME\ y. y \in A \wedge f\ y = x)$
by ($simp\ add$: $inv\text{-}into\text{-}def$)

abbreviation $inv :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$ **where**
 $inv \equiv inv\text{-}into\ UNIV$

25.2 Hilbert’s Epsilon-operator

lemma $Eps\text{-}cong$:
assumes $\bigwedge x. P\ x = Q\ x$
shows $Eps\ P = Eps\ Q$
using ext [of $P\ Q$, $OF\ assms$] **by** $simp$

Easier to use than $someI$ if the witness comes from an existential formula.

lemma $someI\text{-}ex$ [$elim?$]: $\exists x. P\ x \Longrightarrow P\ (SOME\ x. P\ x)$
by ($elim\ exE\ someI$)

lemma $some\text{-}eq\text{-}imp$:
assumes $Eps\ P = a\ P\ b$ **shows** $P\ a$
using $assms\ someI\text{-}ex$ **by** $force$

Easier to use than $someI$ because the conclusion has only one occurrence of P .

lemma $someI2$: $P\ a \Longrightarrow (\bigwedge x. P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x. P\ x)$
by ($blast\ intro$: $someI$)

Easier to use than $someI2$ if the witness comes from an existential formula.

lemma $someI2\text{-}ex$: $\exists a. P\ a \Longrightarrow (\bigwedge x. P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x. P\ x)$
by ($blast\ intro$: $someI2$)

lemma $someI2\text{-}bex$: $\exists a \in A. P\ a \Longrightarrow (\bigwedge x. x \in A \wedge P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x. x \in A \wedge P\ x)$
by ($blast\ intro$: $someI2$)

lemma $some\text{-}equality$ [$intro$]: $P\ a \Longrightarrow (\bigwedge x. P\ x \Longrightarrow x = a) \Longrightarrow (SOME\ x. P\ x) = a$
by ($blast\ intro$: $someI2$)

lemma $some1\text{-}equality$: $\exists! x. P\ x \Longrightarrow P\ a \Longrightarrow (SOME\ x. P\ x) = a$
by $blast$

lemma *some-eq-ex*: $P (SOME\ x.\ P\ x) \longleftrightarrow (\exists\ x.\ P\ x)$
by (*blast intro: someI*)

lemma *some-in-eq*: $(SOME\ x.\ x \in A) \in A \longleftrightarrow A \neq \{\}$
unfolding *ex-in-conv[symmetric]* **by** (*rule some-eq-ex*)

lemma *some-eq-trivial* [*simp*]: $(SOME\ y.\ y = x) = x$
by (*rule some-equality*) (*rule refl*)

lemma *some-sym-eq-trivial* [*simp*]: $(SOME\ y.\ x = y) = x$
by (*iprover intro: some-equality*)

25.3 Axiom of Choice, Proved Using the Description Operator

lemma *choice*: $\forall x.\ \exists y.\ Q\ x\ y \implies \exists f.\ \forall x.\ Q\ x\ (f\ x)$
by (*fast elim: someI*)

lemma *bchoice*: $\forall x \in S.\ \exists y.\ Q\ x\ y \implies \exists f.\ \forall x \in S.\ Q\ x\ (f\ x)$
by (*fast elim: someI*)

lemma *choice-iff*: $(\forall x.\ \exists y.\ Q\ x\ y) \longleftrightarrow (\exists f.\ \forall x.\ Q\ x\ (f\ x))$
by (*fast elim: someI*)

lemma *choice-iff'*: $(\forall x.\ P\ x \longrightarrow (\exists y.\ Q\ x\ y)) \longleftrightarrow (\exists f.\ \forall x.\ P\ x \longrightarrow Q\ x\ (f\ x))$
by (*fast elim: someI*)

lemma *bchoice-iff*: $(\forall x \in S.\ \exists y.\ Q\ x\ y) \longleftrightarrow (\exists f.\ \forall x \in S.\ Q\ x\ (f\ x))$
by (*fast elim: someI*)

lemma *bchoice-iff'*: $(\forall x \in S.\ P\ x \longrightarrow (\exists y.\ Q\ x\ y)) \longleftrightarrow (\exists f.\ \forall x \in S.\ P\ x \longrightarrow Q\ x\ (f\ x))$
by (*fast elim: someI*)

lemma *dependent-nat-choice*:

assumes 1: $\exists x.\ P\ 0\ x$

and 2: $\bigwedge n.\ P\ n\ x \implies \exists y.\ P\ (Suc\ n)\ y \wedge Q\ n\ x\ y$

shows $\exists f.\ \forall n.\ P\ n\ (f\ n) \wedge Q\ n\ (f\ n)\ (f\ (Suc\ n))$

proof (*intro exI allI conjI*)

fix n

define f **where** $f = \text{rec-nat } (SOME\ x.\ P\ 0\ x) (\lambda n\ x.\ SOME\ y.\ P\ (Suc\ n)\ y \wedge Q\ n\ x\ y)$

then have $P\ 0\ (f\ 0) \wedge n.\ P\ n\ (f\ n) \implies P\ (Suc\ n)\ (f\ (Suc\ n)) \wedge Q\ n\ (f\ n)\ (f\ (Suc\ n))$

using *someI-ex[OF 1]* *someI-ex[OF 2]* **by** *simp-all*

then show $P\ n\ (f\ n) \wedge Q\ n\ (f\ n)\ (f\ (Suc\ n))$

by (*induct n*) *auto*

qed

lemma *finite-subset-Union*:
assumes *finite A* $A \subseteq \bigcup \mathcal{B}$
obtains \mathcal{F} **where** *finite F* $\mathcal{F} \subseteq \mathcal{B}$ $A \subseteq \bigcup \mathcal{F}$
proof –
have $\forall x \in A. \exists B \in \mathcal{B}. x \in B$
using *assms* **by** *blast*
then obtain f **where** $f: \bigwedge x. x \in A \implies f x \in \mathcal{B} \wedge x \in f x$
by (*auto simp add: bchoice-iff Bex-def*)
show *thesis*
proof
show *finite (f ‘ A)*
using *assms* **by** *auto*
qed (*use f in auto*)
qed

25.4 Function Inverse

lemma *inv-def*: $\text{inv } f = (\lambda y. \text{SOME } x. f x = y)$
by (*simp add: inv-into-def*)

lemma *inv-into-into*: $x \in f \text{ ‘ } A \implies \text{inv-into } A f x \in A$
by (*simp add: inv-into-def*) (*fast intro: someI2*)

lemma *inv-identity* [*simp*]: $\text{inv } (\lambda a. a) = (\lambda a. a)$
by (*simp add: inv-def*)

lemma *inv-id* [*simp*]: $\text{inv } \text{id} = \text{id}$
by (*simp add: id-def*)

lemma *inv-into-f-f* [*simp*]: $\text{inj-on } f A \implies x \in A \implies \text{inv-into } A f (f x) = x$
by (*simp add: inv-into-def inj-on-def*) (*blast intro: someI2*)

lemma *inv-f-f*: $\text{inj } f \implies \text{inv } f (f x) = x$
by *simp*

lemma *f-inv-into-f*: $y \in f \text{ ‘ } A \implies f (\text{inv-into } A f y) = y$
by (*simp add: inv-into-def*) (*fast intro: someI2*)

lemma *inv-into-f-eq*: $\text{inj-on } f A \implies x \in A \implies f x = y \implies \text{inv-into } A f y = x$
by (*erule subst*) (*fast intro: inv-into-f-f*)

lemma *inv-f-eq*: $\text{inj } f \implies f x = y \implies \text{inv } f y = x$
by (*simp add: inv-into-f-eq*)

lemma *inj-imp-inv-eq*: $\text{inj } f \implies \forall x. f (g x) = x \implies \text{inv } f = g$
by (*blast intro: inv-into-f-eq*)

But is it useful?

lemma *inj-transfer*:

assumes $inj: inj\ f$
and $minor: \bigwedge y. y \in range\ f \implies P\ (inv\ f\ y)$
shows $P\ x$
proof –
have $f\ x \in range\ f$ **by** *auto*
then have $P(inv\ f\ (f\ x))$ **by** (*rule minor*)
then show $P\ x$ **by** (*simp add: inv-into-f-f [OF inj]*)
qed

lemma $inj\text{-}iff: inj\ f \longleftrightarrow inv\ f \circ f = id$
by (*simp add: o-def fun-eq-iff*) (*blast intro: inj-on-inverseI inv-into-f-f*)

lemma $inv\text{-}o\text{-}cancel[simp]: inj\ f \implies inv\ f \circ f = id$
by (*simp add: inj-iff*)

lemma $o\text{-}inv\text{-}o\text{-}cancel[simp]: inj\ f \implies g \circ inv\ f \circ f = g$
by (*simp add: comp-assoc*)

lemma $inv\text{-}into\text{-}image\text{-}cancel[simp]: inj\text{-}on\ f\ A \implies S \subseteq A \implies inv\text{-}into\ A\ f\ 'f\ 'S = S$
by (*fastforce simp: image-def*)

lemma $inj\text{-}imp\text{-}surj\text{-}inv: inj\ f \implies surj\ (inv\ f)$
by (*blast intro!: surjI inv-into-f-f*)

lemma $surj\text{-}f\text{-}inv\text{-}f: surj\ f \implies f\ (inv\ f\ y) = y$
by (*simp add: f-inv-into-f*)

lemma $bij\text{-}inv\text{-}eq\text{-}iff: bij\ p \implies x = inv\ p\ y \longleftrightarrow p\ x = y$
using $surj\text{-}f\text{-}inv\text{-}f[of\ p]$ **by** (*auto simp add: bij-def*)

lemma $inv\text{-}into\text{-}injective:$
assumes $eq: inv\text{-}into\ A\ f\ x = inv\text{-}into\ A\ f\ y$
and $x: x \in f\ 'A$
and $y: y \in f\ 'A$
shows $x = y$
proof –
from eq **have** $f\ (inv\text{-}into\ A\ f\ x) = f\ (inv\text{-}into\ A\ f\ y)$
by *simp*
with $x\ y$ **show** *?thesis*
by (*simp add: f-inv-into-f*)
qed

lemma $inj\text{-}on\text{-}inv\text{-}into: B \subseteq f\ 'A \implies inj\text{-}on\ (inv\text{-}into\ A\ f)\ B$
by (*blast intro: inj-onI dest: inv-into-injective injD*)

lemma $inj\text{-}imp\text{-}bij\text{-}betw\text{-}inv: inj\ f \implies bij\text{-}betw\ (inv\ f)\ (f\ 'M)\ M$
by (*simp add: bij-betw-def image-subsetI inj-on-inv-into*)

lemma *bij-betw-inv-into*: $\text{bij-betw } f \ A \ B \implies \text{bij-betw } (\text{inv-into } A \ f) \ B \ A$
by (*auto simp add: bij-betw-def inj-on-inv-into*)

lemma *surj-imp-inj-inv*: $\text{surj } f \implies \text{inj } (\text{inv } f)$
by (*simp add: inj-on-inv-into*)

lemma *surj-iff*: $\text{surj } f \iff f \circ \text{inv } f = \text{id}$
by (*auto intro!: surjI simp: surj-f-inv-f fun-eq-iff [where 'b='a]*)

lemma *surj-iff-all*: $\text{surj } f \iff (\forall x. f (\text{inv } f \ x) = x)$
by (*simp add: o-def surj-iff fun-eq-iff*)

lemma *surj-imp-inv-eq*:
assumes *surj f and gf*: $\bigwedge x. g (f \ x) = x$
shows $\text{inv } f = g$
proof (*rule ext*)
fix x
have $g (f (\text{inv } f \ x)) = \text{inv } f \ x$
by (*rule gf*)
then show $\text{inv } f \ x = g \ x$
by (*simp add: surj-f-inv-f ‹surj f›*)

qed

lemma *bij-imp-bij-inv*: $\text{bij } f \implies \text{bij } (\text{inv } f)$
by (*simp add: bij-def inj-imp-surj-inv surj-imp-inj-inv*)

lemma *inv-equality*: $(\bigwedge x. g (f \ x) = x) \implies (\bigwedge y. f (g \ y) = y) \implies \text{inv } f = g$
by (*rule ext*) (*auto simp add: inv-into-def*)

lemma *inv-inv-eq*: $\text{bij } f \implies \text{inv } (\text{inv } f) = f$
by (*rule inv-equality*) (*auto simp add: bij-def surj-f-inv-f*)

bij (inv f) implies little about *f*. Consider $f :: \text{bool} \Rightarrow \text{bool}$ such that $f \ \text{True} = f \ \text{False} = \text{True}$. Then it is consistent with axiom *someI* that *inv f* could be any function at all, including the identity function. If $\text{inv } f = \text{id}$ then *inv f* is a bijection, but *inj f*, *surj f* and $\text{inv } (\text{inv } f) = f$ all fail.

lemma *inv-into-comp*:
 $\text{inj-on } f \ (g \ ' \ A) \implies \text{inj-on } g \ A \implies x \in f \ ' \ g \ ' \ A \implies$
 $\text{inv-into } A \ (f \circ g) \ x = (\text{inv-into } A \ g \circ \text{inv-into } (g \ ' \ A) \ f) \ x$
by (*auto simp: f-inv-into-f inv-into-into intro: inv-into-f-eq comp-inj-on*)

lemma *o-inv-distrib*: $\text{bij } f \implies \text{bij } g \implies \text{inv } (f \circ g) = \text{inv } g \circ \text{inv } f$
by (*rule inv-equality*) (*auto simp add: bij-def surj-f-inv-f*)

lemma *image-f-inv-f*: $\text{surj } f \implies f \ ' \ (\text{inv } f \ ' \ A) = A$
by (*simp add: surj-f-inv-f image-comp comp-def*)

lemma *image-inv-f-f*: $\text{inj } f \implies \text{inv } f \ ' \ (f \ ' \ A) = A$
by *simp*

lemma *bij-image-Collect-eq*:
 assumes *bij f*
 shows $f \text{ ' Collect } P = \{y. P (inv f y)\}$
proof
 show $f \text{ ' Collect } P \subseteq \{y. P (inv f y)\}$
 using *assms* **by** (*force simp add: bij-is-inj*)
 show $\{y. P (inv f y)\} \subseteq f \text{ ' Collect } P$
 using *assms* **by** (*blast intro: bij-is-surj [THEN surj-f-inv-f, symmetric]*)
qed

lemma *bij-vimage-eq-inv-image*:
 assumes *bij f*
 shows $f \text{ - ' } A = inv f \text{ ' } A$
proof
 show $f \text{ - ' } A \subseteq inv f \text{ ' } A$
 using *assms* **by** (*blast intro: bij-is-inj [THEN inv-into-f-f, symmetric]*)
 show $inv f \text{ ' } A \subseteq f \text{ - ' } A$
 using *assms* **by** (*auto simp add: bij-is-surj [THEN surj-f-inv-f]*)
qed

lemma *inv-fn-o-fn-is-id*:
 fixes $f::'a \Rightarrow 'a$
 assumes *bij f*
 shows $((inv f) \text{ ~ } n) o (f \text{ ~ } n) = (\lambda x. x)$
proof –
 have $((inv f) \text{ ~ } n)((f \text{ ~ } n) x) = x$ **for** $x n$
proof (*induction n*)
 case (*Suc n*)
 have $*$: $(inv f) (f y) = y$ **for** y
 by (*simp add: assms bij-is-inj*)
 have $(inv f \text{ ~ } Suc n) ((f \text{ ~ } Suc n) x) = (inv f \text{ ~ } n) (inv f (f ((f \text{ ~ } n) x)))$
 by (*simp add: funpow-swap1*)
 also have $\dots = (inv f \text{ ~ } n) ((f \text{ ~ } n) x)$
 using $*$ **by** *auto*
 also have $\dots = x$ **using** *Suc.IH* **by** *auto*
 finally show *?case* **by** *simp*
qed (*auto*)
then show *?thesis* **unfolding** *o-def* **by** *blast*
qed

lemma *fn-o-inv-fn-is-id*:
 fixes $f::'a \Rightarrow 'a$
 assumes *bij f*
 shows $(f \text{ ~ } n) o ((inv f) \text{ ~ } n) = (\lambda x. x)$
proof –
 have $(f \text{ ~ } n) (((inv f) \text{ ~ } n) x) = x$ **for** $x n$
proof (*induction n*)
 case (*Suc n*)

```

have *: f(inv f y) = y for y
  using bij-inv-eq-iff[OF assms] by auto
have (f ~ Suc n) ((inv f ~ Suc n) x) = (f ~ n) (f (inv f ((inv f ~ n) x)))
  by (simp add: funpow-swap1)
also have ... = (f ~ n) ((inv f ~ n) x)
  using * by auto
also have ... = x using Suc.IH by auto
finally show ?case by simp
qed (auto)
then show ?thesis unfolding o-def by blast
qed

```

```

lemma inv-fn:
  fixes f::'a ⇒ 'a
  assumes bij f
  shows inv (f ~ n) = ((inv f) ~ n)
proof -
  have inv (f ~ n) x = ((inv f) ~ n) x for x
  proof (rule inv-into-f-eq)
    show inj (f ~ n)
      by (simp add: inj-fn[OF bij-is-inj [OF assms]])
    show (f ~ n) ((inv f ~ n) x) = x
      using fn-o-inv-fn-is-id[OF assms, THEN fun-cong] by force
  qed auto
  then show ?thesis by auto
qed

```

```

lemma funpow-inj-finite:
  assumes ⟨inj p⟩ ⟨finite {y. ∃ n. y = (p ~ n) x}⟩
  obtains n where ⟨n > 0⟩ ⟨(p ~ n) x = x⟩
proof -
  have ⟨infinite (UNIV :: nat set)⟩
    by simp
  moreover have ⟨{y. ∃ n. y = (p ~ n) x} = (λn. (p ~ n) x) ‘ UNIV⟩
    by auto
  with assms have ⟨finite ...⟩
    by simp
  ultimately have ∃ n ∈ UNIV. ¬ finite {m ∈ UNIV. (p ~ m) x = (p ~ n) x}
    by (rule pigeonhole-infinite)
  then obtain n where infinite {m. (p ~ m) x = (p ~ n) x} by auto
  then have infinite ({m. (p ~ m) x = (p ~ n) x} - {n}) by auto
  then have ({m. (p ~ m) x = (p ~ n) x} - {n}) ≠ {}
    by (auto simp add: subset-singleton-iff)
  then obtain m where m: (p ~ m) x = (p ~ n) x m ≠ n by auto

{ fix m n assume (p ~ n) x = (p ~ m) x m < n
  have (p ~ (n - m)) x = inv (p ~ m) ((p ~ m) ((p ~ (n - m)) x))
    using ⟨inj p⟩ by (simp add: inv-f-f)
  also have ((p ~ m) ((p ~ (n - m)) x)) = (p ~ n) x

```

```

    using ⟨m < n⟩ funpow-add [of m ⟨n - m⟩ p] by simp
  also have inv (p ~ m) ... = x
    using ⟨inj p⟩ by (simp add: ⟨p ~ n⟩ x = -)
  finally have (p ~ (n - m)) x = x 0 < n - m
    using ⟨m < n⟩ by auto }
note general = this

show thesis
proof (cases m n rule: linorder-cases)
  case less
  then have ⟨n - m > 0⟩ ⟨p ~ (n - m)⟩ x = x
    using general [of n m] m by simp-all
  then show thesis by (blast intro: that)
next
  case equal
  then show thesis using m by simp
next
  case greater
  then have ⟨m - n > 0⟩ ⟨p ~ (m - n)⟩ x = x
    using general [of m n] m by simp-all
  then show thesis by (blast intro: that)
qed
qed

lemma mono-inv:
  fixes f :: 'a::linorder ⇒ 'b::linorder
  assumes mono f bij f
  shows mono (inv f)
proof
  fix x y :: 'b assume x ≤ y
  from ⟨bij f⟩ obtain a b where x: x = f a and y: y = f b by (fastforce simp:
bij-def surj-def)
  show inv f x ≤ inv f y
  proof (rule le-cases)
    assume a ≤ b
    thus ?thesis using ⟨bij f⟩ x y by (simp add: bij-def inv-f-f)
  next
    assume b ≤ a
    hence f b ≤ f a by (rule monoD[OF ⟨mono f⟩])
    hence y ≤ x using x y by simp
    hence x = y using ⟨x ≤ y⟩ by auto
    thus ?thesis by simp
  qed
qed

lemma strict-mono-inv-on-range:
  fixes f :: 'a::linorder ⇒ 'b::order
  assumes strict-mono f

```

```

shows strict-mono-on (range f) (inv f)
proof (clarsimp simp: strict-mono-on-def)
  fix x y
  assume f x < f y
  then show inv f (f x) < inv f (f y)
    using assms strict-mono-imp-inj-on strict-mono-less by fastforce
qed

```

```

lemma mono-bij-Inf:
  fixes f :: 'a::complete-linorder  $\Rightarrow$  'b::complete-linorder
  assumes mono f bij f
  shows f (Inf A) = Inf (f'A)
proof –
  have surj f using <bij f> by (auto simp: bij-betw-def)
  have *: (inv f) (Inf (f'A))  $\leq$  Inf ((inv f) '(f'A))
    using mono-Inf[OF mono-inv[OF assms], of f'A] by simp
  have Inf (f'A)  $\leq$  f (Inf ((inv f) '(f'A)))
    using monoD[OF <mono f> *] by (simp add: surj-f-inv-f[OF <surj f>])
  also have ... = f(Inf A)
    using assms by (simp add: bij-is-inj)
  finally show ?thesis using mono-Inf[OF assms(1), of A] by auto
qed

```

```

lemma finite-fun-UNIVD1:
  assumes fin: finite (UNIV :: 'a  $\Rightarrow$  'b) set
  and card: card (UNIV :: 'b set)  $\neq$  Suc 0
  shows finite (UNIV :: 'a set)
proof –
  let ?UNIV-b = UNIV :: 'b set
  from fin have finite ?UNIV-b
    by (rule finite-fun-UNIVD2)
  with card have card ?UNIV-b  $\geq$  Suc (Suc 0)
    by (cases card ?UNIV-b (auto simp: card-eq-0-iff))
  then have card ?UNIV-b = Suc (Suc (card ?UNIV-b - Suc (Suc 0)))
    by simp
  then obtain b1 b2 :: 'b where b1b2: b1  $\neq$  b2
    by (auto simp: card-Suc-eq)
  from fin have fin': finite (range ( $\lambda$ f :: 'a  $\Rightarrow$  'b. inv f b1))
    by (rule finite-imageI)
  have UNIV = range ( $\lambda$ f :: 'a  $\Rightarrow$  'b. inv f b1)
  proof (rule UNIV-eq-I)
    fix x :: 'a
    from b1b2 have x = inv ( $\lambda$ y. if y = x then b1 else b2) b1
      by (simp add: inv-into-def)
    then show x  $\in$  range ( $\lambda$ f::'a  $\Rightarrow$  'b. inv f b1)
      by blast
  qed
  with fin' show ?thesis
    by simp

```


qed

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S .

The “only if” direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S . The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S .

lemma *infinite-countable-subset*:

assumes *inf*: \neg *finite* S

shows $\exists f::\text{nat} \Rightarrow 'a. \text{inj } f \wedge \text{range } f \subseteq S$

— Courtesy of Stephan Merz

proof —

define *Sseq* **where** $Sseq = \text{rec-nat } S (\lambda n T. T - \{\text{SOME } e. e \in T\})$

define *pick* **where** $\text{pick } n = (\text{SOME } e. e \in Sseq\ n)$ **for** n

have $*$: $Sseq\ n \subseteq S \neg$ *finite* ($Sseq\ n$) **for** n

by (*induct* n) (*auto simp: Sseq-def inf*)

then have $**$: $\bigwedge n. \text{pick } n \in Sseq\ n$

unfolding *pick-def* **by** (*subst (asm) finite.simps*) (*auto simp add: ex-in-conv intro: someI-ex*)

with $*$ **have** $\text{range } \text{pick} \subseteq S$ **by** *auto*

moreover have $\text{pick } n \neq \text{pick } (n + \text{Suc } m)$ **for** $m\ n$

proof —

have $\text{pick } n \notin Sseq\ (n + \text{Suc } m)$

by (*induct* m) (*auto simp add: Sseq-def pick-def*)

with $**$ **show** *?thesis* **by** *auto*

qed

then have *inj pick*

by (*intro linorder-injI*) (*auto simp add: less-iff-Suc-add*)

ultimately show *?thesis* **by** *blast*

qed

lemma *infinite-iff-countable-subset*: \neg *finite* $S \longleftrightarrow (\exists f::\text{nat} \Rightarrow 'a. \text{inj } f \wedge \text{range } f \subseteq S)$

— Courtesy of Stephan Merz

using *finite-imageD finite-subset infinite-UNIV-char-0 infinite-countable-subset*
by *auto*

lemma *image-inv-into-cancel*:

assumes *surj*: $f'A = A'$

and *sub*: $B' \subseteq A'$

shows $f \text{ `} ((\text{inv-into } A\ f) 'B') = B'$

using *assms*

proof (*auto simp: f-inv-into-f*)

let $?f' = \text{inv-into } A\ f$

fix a'

assume $*$: $a' \in B'$

with *sub* **have** $a' \in A'$ **by** *auto*
with *surj* **have** $a' = f$ ($?f'$ a')
by (*auto simp: f-inv-into-f*)
with * **show** $a' \in f$ ' ($?f'$ ' B') **by** *blast*
qed

lemma *inv-into-inv-into-eq*:
assumes *bij-betw* f A A'
and $a: a \in A$
shows *inv-into* A' (*inv-into* A f) $a = f$ a
proof –
let $?f' = \text{inv-into } A$ f
let $?f'' = \text{inv-into } A'$ $?f'$
from *assms* **have** *: *bij-betw* $?f'$ A' A
by (*auto simp: bij-betw-inv-into*)
with a **obtain** a' **where** $a': a' \in A'$ $?f'$ $a' = a$
unfolding *bij-betw-def* **by** *force*
with a * **have** $?f''$ $a = a'$
by (*auto simp: f-inv-into-f bij-betw-def*)
moreover from *assms* a' **have** f $a = a'$
by (*auto simp: bij-betw-def*)
ultimately show $?f''$ $a = f$ a **by** *simp*
qed

lemma *inj-on-iff-surj*:
assumes $A \neq \{\}$
shows $(\exists f. \text{inj-on } f$ $A \wedge f$ ' $A \subseteq A')$ \longleftrightarrow $(\exists g. g$ ' $A' = A)$
proof *safe*
fix f
assume *inj*: *inj-on* f A **and** *incl*: f ' $A \subseteq A'$
let $?phi = \lambda a'. a. a \in A \wedge f$ $a = a'$
let $?csi = \lambda a. a \in A$
let $?g = \lambda a'. \text{if } a' \in f$ ' A **then** (*SOME* $a. ?phi$ a' a) **else** (*SOME* $a. ?csi$ a)
have $?g$ ' $A' = A$
proof
show $?g$ ' $A' \subseteq A$
proof *clarify*
fix a'
assume *: $a' \in A'$
show $?g$ $a' \in A$
proof (*cases* $a' \in f$ ' A)
case *True*
then obtain a **where** $?phi$ a' a **by** *blast*
then have $?phi$ a' (*SOME* $a. ?phi$ a' a)
using *someI*[*of* $?phi$ a' a] **by** *blast*
with *True* **show** *thesis* **by** *auto*
next
case *False*
with *assms* **have** $?csi$ (*SOME* $a. ?csi$ a)

```

      using someI-ex[of ?csi] by blast
      with False show ?thesis by auto
    qed
  qed
next
show  $A \subseteq ?g \text{ ' } A'$ 
proof -
  have  $?g (f a) = a \wedge f a \in A'$  if  $a: a \in A$  for  $a$ 
  proof -
    let ?b = SOME aa. ?phi (f a) aa
    from a have ?phi (f a) a by auto
    then have *: ?phi (f a) ?b
      using someI[of ?phi(f a) a] by blast
    then have  $?g (f a) = ?b$  using a by auto
    moreover from inj * a have  $a = ?b$ 
      by (auto simp add: inj-on-def)
    ultimately have  $?g(f a) = a$  by simp
    with incl a show ?thesis by auto
  qed
  then show ?thesis by force
  qed
qed
then show  $\exists g. g \text{ ' } A' = A$  by blast
next
fix g
let ?f = inv-into A' g
have inj-on ?f (g ' A')
  by (auto simp: inj-on-inv-into)
moreover have  $?f (g a') \in A'$  if  $a': a' \in A'$  for  $a'$ 
proof -
  let ?phi =  $\lambda b'. b' \in A' \wedge g b' = g a'$ 
  from a' have ?phi a' by auto
  then have ?phi (SOME b'. ?phi b')
    using someI[of ?phi] by blast
  then show ?thesis by (auto simp: inv-into-def)
qed
ultimately show  $\exists f. \text{inj-on } f (g \text{ ' } A') \wedge f \text{ ' } g \text{ ' } A' \subseteq A'$ 
  by auto
qed

```

lemma *Ex-inj-on-UNION-Sigma:*

$\exists f. (\text{inj-on } f (\bigcup i \in I. A i) \wedge f \text{ ' } (\bigcup i \in I. A i) \subseteq (\text{SIGMA } i : I. A i))$

proof

```

let ?phi =  $\lambda a i. i \in I \wedge a \in A i$ 
let ?sm =  $\lambda a. \text{SOME } i. ?phi a i$ 
let ?f =  $\lambda a. (?sm a, a)$ 
have inj-on ?f ( $\bigcup i \in I. A i$ )
  by (auto simp: inj-on-def)
moreover

```

have $?sm\ a \in I \wedge a \in A(?sm\ a)$ **if** $i \in I$ **and** $a \in A\ i$ **for** $i\ a$
using *that someI[of ?phi a i]* **by** *auto*
then have $?f\ ' (\bigcup i \in I. A\ i) \subseteq (SIGMA\ i : I. A\ i)$
by *auto*
ultimately show $inj\text{-on}\ ?f\ (\bigcup i \in I. A\ i) \wedge ?f\ ' (\bigcup i \in I. A\ i) \subseteq (SIGMA\ i : I. A\ i)$
by *auto*
qed

lemma *inv-unique-comp*:
assumes $fg: f \circ g = id$
and $gf: g \circ f = id$
shows $inv\ f = g$
using $fg\ gf\ inv\text{-equality}[of\ g\ f]$ **by** (*auto simp add: fun-eq-iff*)

lemma *subset-image-inj*:
 $S \subseteq f\ ' T \iff (\exists U. U \subseteq T \wedge inj\text{-on}\ f\ U \wedge S = f\ ' U)$
proof *safe*
show $\exists U \subseteq T. inj\text{-on}\ f\ U \wedge S = f\ ' U$
if $S \subseteq f\ ' T$
proof $-$
from *that [unfolded subset-image-iff subset-iff]*
obtain g **where** $g: \bigwedge x. x \in S \implies g\ x \in T \wedge x = f\ (g\ x)$
by (*auto simp add: image-iff Bex-def choice-iff'*)
show *?thesis*
proof (*intro exI conjI*)
show $g\ ' S \subseteq T$
by (*simp add: g image-subsetI*)
show $inj\text{-on}\ f\ (g\ ' S)$
using g **by** (*auto simp: inj-on-def*)
show $S = f\ ' (g\ ' S)$
using $g\ image\text{-subset-iff}$ **by** *auto*
qed
qed
qed *blast*

25.5 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the *split* Operator

Looping *simprule*!

lemma *split-paired-Eps*: $(SOME\ x. P\ x) = (SOME\ (a, b). P\ (a, b))$
by *simp*

lemma *Eps-case-prod*: $Eps\ (case\text{-prod}\ P) = (SOME\ xy. P\ (fst\ xy)\ (snd\ xy))$
by (*simp add: split-def*)

lemma *Eps-case-prod-eq* [*simp*]: $(SOME\ (x', y'). x = x' \wedge y = y') = (x, y)$
by *blast*

A relation is wellfounded iff it has no infinite descending chain.

lemma *wf-iff-no-infinite-down-chain*: $wf\ r \longleftrightarrow (\nexists f. \forall i. (f\ (Suc\ i), f\ i) \in r)$
(is - $\longleftrightarrow \neg\ ?ex$)

proof

assume $wf\ r$

show $\neg\ ?ex$

proof

assume $?ex$

then obtain f **where** $f: (f\ (Suc\ i), f\ i) \in r$ **for** i

by *blast*

from $\langle wf\ r \rangle$ **have** *minimal*: $x \in Q \implies \exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q$ **for** x

Q

by (*auto simp: wf-eq-minimal*)

let $?Q = \{w. \exists i. w = f\ i\}$

fix n

have $f\ n \in ?Q$ **by** *blast*

from *minimal* [*OF this*] **obtain** j **where** $(y, f\ j) \in r \implies y \notin ?Q$ **for** y **by**

blast

with *this* [*OF* $\langle (f\ (Suc\ j), f\ j) \in r \rangle$] **have** $f\ (Suc\ j) \notin ?Q$ **by** *simp*

then show *False* **by** *blast*

qed

next

assume $\neg\ ?ex$

then show $wf\ r$

proof (*rule contrapos-np*)

assume $\neg\ wf\ r$

then obtain Q x **where** $x: x \in Q$ **and** *rec*: $z \in Q \implies \exists y. (y, z) \in r \wedge y \in$

Q **for** z

by (*auto simp add: wf-eq-minimal*)

obtain *descend* $:: nat \Rightarrow 'a$

where *descend-0*: $descend\ 0 = x$

and *descend-Suc*: $descend\ (Suc\ n) = (SOME\ y. y \in Q \wedge (y, descend\ n) \in$

$r)$ **for** n

by (*rule that [of rec-nat x (λ - rec. (SOME y. y \in Q \wedge (y, rec) \in r))] simp-all*)

have *descend-Q*: $descend\ n \in Q$ **for** n

proof (*induct n*)

case 0

with x **show** *?case* **by** (*simp only: descend-0*)

next

case *Suc*

then show *?case* **by** (*simp only: descend-Suc*) (*rule someI2-ex; use rec in*

blast)

qed

have $(descend\ (Suc\ i), descend\ i) \in r$ **for** i

by (*simp only: descend-Suc*) (*rule someI2-ex; use descend-Q rec in blast*)

then show $\exists f. \forall i. (f\ (Suc\ i), f\ i) \in r$ **by** *blast*

qed

qed

lemma *wf-no-infinite-down-chainE*:
assumes *wf r*
obtains *k* **where** $(f (Suc\ k), f\ k) \notin r$
using *assms wf-iff-no-infinite-down-chain[of r]* **by** *blast*

A dynamically-scoped fact for TFL

lemma *tfl-some*: $\forall P\ x.\ P\ x \longrightarrow P\ (Eps\ P)$
by (*blast intro: someI*)

25.6 An aside: bounded accessible part

Finite monotone eventually stable sequences

lemma *finite-mono-remains-stable-implies-strict-prefix*:
fixes $f :: nat \Rightarrow 'a::order$
assumes $S: finite\ (range\ f)\ mono\ f$
and $eq: \forall n.\ f\ n = f\ (Suc\ n) \longrightarrow f\ (Suc\ n) = f\ (Suc\ (Suc\ n))$
shows $\exists N.\ (\forall n \leq N.\ \forall m \leq N.\ m < n \longrightarrow f\ m < f\ n) \wedge (\forall n \geq N.\ f\ N = f\ n)$
using *assms*

proof –

have $\exists n.\ f\ n = f\ (Suc\ n)$

proof (*rule ccontr*)

assume $\neg ?thesis$

then have $\bigwedge n.\ f\ n \neq f\ (Suc\ n)$ **by** *auto*

with $\langle mono\ f \rangle$ **have** $\bigwedge n.\ f\ n < f\ (Suc\ n)$

by (*auto simp: le-less mono-iff-le-Suc*)

with *lift-Suc-mono-less-iff*[of *f*] **have** $*$: $\bigwedge n\ m.\ n < m \Longrightarrow f\ n < f\ m$

by *auto*

have *inj f*

proof (*intro injI*)

fix $x\ y$

assume $f\ x = f\ y$

then show $x = y$

by (*cases x y rule: linorder-cases*) (*auto dest: **)

qed

with $\langle finite\ (range\ f) \rangle$ **have** *finite* (*UNIV::nat set*)

by (*rule finite-imageD*)

then show *False* **by** *simp*

qed

then obtain n **where** $n: f\ n = f\ (Suc\ n)$ **..**

define N **where** $N = (LEAST\ n.\ f\ n = f\ (Suc\ n))$

have $N: f\ N = f\ (Suc\ N)$

unfolding *N-def* **using** n **by** (*rule LeastI*)

show *?thesis*

proof (*intro exI[of - N] conjI allI impI*)

fix n

assume $N \leq n$

then have $\bigwedge m.\ N \leq m \Longrightarrow m \leq n \Longrightarrow f\ m = f\ N$

proof (*induct rule: dec-induct*)

case *base*

```

    then show ?case by simp
  next
    case (step n)
    then show ?case
      using eq [rule-format, of n - 1] N
      by (cases n) (auto simp add: le-Suc-eq)
    qed
  from this[of n] ⟨N ≤ n⟩ show f N = f n by auto
next
fix n m :: nat
assume m < n n ≤ N
then show f m < f n
proof (induct rule: less-Suc-induct)
  case (1 i)
  then have i < N by simp
  then have f i ≠ f (Suc i)
    unfolding N-def by (rule not-less-Least)
  with ⟨mono f⟩ show ?case by (simp add: mono-iff-le-Suc less-le)
next
case 2
then show ?case by simp
qed
qed
qed

```

lemma *finite-mono-strict-prefix-implies-finite-fixpoint*:

```

fixes f :: nat ⇒ 'a set
assumes S: ∧i. f i ⊆ S finite S
  and ex: ∃ N. (∀ n ≤ N. ∀ m ≤ N. m < n ⟶ f m ⊂ f n) ∧ (∀ n ≥ N. f N = f n)
shows f (card S) = (⋃ n. f n)
proof -
  from ex obtain N where inj: ∧n m. n ≤ N ⟹ m ≤ N ⟹ m < n ⟹ f m
  ⊂ f n
  and eq: ∀ n ≥ N. f N = f n
  by atomize auto
  have i ≤ N ⟹ i ≤ card (f i) for i
  proof (induct i)
    case 0
    then show ?case by simp
  next
    case (Suc i)
    with inj [of Suc i i] have (f i) ⊂ (f (Suc i)) by auto
    moreover have finite (f (Suc i)) using S by (rule finite-subset)
    ultimately have card (f i) < card (f (Suc i)) by (intro psubset-card-mono)
    with Suc inj show ?case by auto
  qed
  then have N ≤ card (f N) by simp
  also have ... ≤ card S using S by (intro card-mono)
  finally have §: f (card S) = f N using eq by auto

```

```

moreover have  $\bigcup (\text{range } f) \subseteq f N$ 
proof clarify
  fix  $x n$ 
  assume  $x \in f n$ 
  with  $eq\ inj [of\ N]$  show  $x \in f N$ 
    by ( $cases\ n < N$ ) ( $auto\ simp: not-less$ )
qed
ultimately show  $?thesis$ 
  by  $auto$ 
qed

```

25.7 More on injections, bijections, and inverses

```

locale  $bijection =$ 
  fixes  $f :: 'a \Rightarrow 'a$ 
  assumes  $bij: bij\ f$ 
begin

```

```

lemma  $bij\text{-}inv: bij\ (inv\ f)$ 
  using  $bij$  by ( $rule\ bij\text{-}imp\text{-}bij\text{-}inv$ )

```

```

lemma  $surj [simp]: surj\ f$ 
  using  $bij$  by ( $rule\ bij\text{-}is\text{-}surj$ )

```

```

lemma  $inj: inj\ f$ 
  using  $bij$  by ( $rule\ bij\text{-}is\text{-}inj$ )

```

```

lemma  $surj\text{-}inv [simp]: surj\ (inv\ f)$ 
  using  $inj$  by ( $rule\ inj\text{-}imp\text{-}surj\text{-}inv$ )

```

```

lemma  $inj\text{-}inv: inj\ (inv\ f)$ 
  using  $surj$  by ( $rule\ surj\text{-}imp\text{-}inj\text{-}inv$ )

```

```

lemma  $eqI: f\ a = f\ b \implies a = b$ 
  using  $inj$  by ( $rule\ injD$ )

```

```

lemma  $eq\text{-}iff [simp]: f\ a = f\ b \iff a = b$ 
  by ( $auto\ intro: eqI$ )

```

```

lemma  $eq\text{-}invI: inv\ f\ a = inv\ f\ b \implies a = b$ 
  using  $inj\text{-}inv$  by ( $rule\ injD$ )

```

```

lemma  $eq\text{-}inv\text{-}iff [simp]: inv\ f\ a = inv\ f\ b \iff a = b$ 
  by ( $auto\ intro: eq\text{-}invI$ )

```

```

lemma  $inv\text{-}left [simp]: inv\ f\ (f\ a) = a$ 
  using  $inj$  by ( $simp\ add: inv\text{-}f\text{-}eq$ )

```

```

lemma  $inv\text{-}comp\text{-}left [simp]: inv\ f \circ f = id$ 

```


by (simp add: fun-eq-iff)

lemma *inv-right* [simp]: $f (inv f a) = a$
 using *surj* by (simp add: surj-f-inv-f)

lemma *inv-comp-right* [simp]: $f \circ inv f = id$
 by (simp add: fun-eq-iff)

lemma *inv-left-eq-iff* [simp]: $inv f a = b \longleftrightarrow f b = a$
 by *auto*

lemma *inv-right-eq-iff* [simp]: $b = inv f a \longleftrightarrow f b = a$
 by *auto*

end

lemma *infinite-imp-bij-betw*:
 assumes *infinite*: \neg *finite* A
 shows $\exists h. \text{bij-betw } h \ A \ (A - \{a\})$
proof (cases $a \in A$)
 case *False*
 then have $A - \{a\} = A$ by *blast*
 then show ?thesis
 using *bij-betw-id*[of A] by *auto*

next

case *True*
with *infinite* have \neg *finite* $(A - \{a\})$ by *auto*
with *infinite-iff-countable-subset*[of $A - \{a\}$]
obtain $f :: nat \Rightarrow 'a$ **where** *inj* f **and** $f ' UNIV \subseteq A - \{a\}$ by *blast*
define g **where** $g \ n = (if \ n = 0 \ then \ a \ else \ f \ (Suc \ n))$ **for** n
define A' **where** $A' = g ' UNIV$
have $*$: $\forall y. f \ y \neq a$ **using** f by *blast*
have \exists : $inj\text{-on } g \ UNIV \wedge g ' UNIV \subseteq A \wedge a \in g ' UNIV$
using $\langle inj \ f \rangle f *$ **unfolding** *inj-on-def* $g\text{-def}$
 by (auto simp add: *True image-subset-iff*)
then have \exists : *bij-betw* $g \ UNIV \ A' \wedge a \in A' \wedge A' \subseteq A$
using *inj-on-imp-bij-betw*[of g] by (auto simp: *A'-def*)
then have \exists : *bij-betw* $(inv \ g) \ A' \ UNIV$
 by (auto simp add: *bij-betw-inv-into*)
from \exists **obtain** n **where** $g \ n = a$ by *auto*
have \exists : *bij-betw* $g \ (UNIV - \{n\}) \ (A' - \{a\})$
 by (rule *bij-betw-subset*) (use $\exists \ 4 \ n$ in $\langle auto \ simp: \ image\text{-set-diff } A'\text{-def} \rangle$)
define v **where** $v \ m = (if \ m < n \ then \ m \ else \ Suc \ m)$ **for** m
have $m < n \vee m = n$ **if** $\bigwedge k. k < n \vee m \neq Suc \ k$ **for** m
using *that* [of $m-1$] by *auto*
then have \exists : *bij-betw* $v \ UNIV \ (UNIV - \{n\})$
unfolding *bij-betw-def inj-on-def v-def* by *auto*
define h' **where** $h' = g \circ v \circ (inv \ g)$
with $\exists \ 6 \ 7$ **have** \exists : *bij-betw* $h' \ A' \ (A' - \{a\})$

by (auto simp add: bij-betw-trans)
define h **where** $h\ b = (if\ b \in A' \text{ then } h'\ b \text{ else } b)$ **for** b
with δ **have** $bij\text{-betw}\ h\ A' (A' - \{a\})$
using $bij\text{-betw}\text{-cong}[of\ A'\ h]$ **by** *auto*
moreover
have $\forall b \in A - A'. h\ b = b$ **by** (auto simp: h-def)
then **have** $bij\text{-betw}\ h\ (A - A') (A - A')$
using $bij\text{-betw}\text{-cong}[of\ A - A'\ h\ id]$ $bij\text{-betw}\text{-id}[of\ A - A']$ **by** *auto*
moreover
from δ **have** $(A' \cap (A - A') = \{\}) \wedge A' \cup (A - A') = A \wedge$
 $((A' - \{a\}) \cap (A - A') = \{\}) \wedge (A' - \{a\}) \cup (A - A') = A - \{a\}$
by *blast*
ultimately **have** $bij\text{-betw}\ h\ A (A - \{a\})$
using $bij\text{-betw}\text{-combine}[of\ h\ A'\ A' - \{a\}\ A - A'\ A - A']$ **by** *simp*
then **show** *?thesis* **by** *blast*
qed

lemma *infinite-imp-bij-betw2*:

assumes $\neg finite\ A$
shows $\exists h. bij\text{-betw}\ h\ A (A \cup \{a\})$
proof (cases $a \in A$)
case *True*
then **have** $A \cup \{a\} = A$ **by** *blast*
then **show** *?thesis* **using** $bij\text{-betw}\text{-id}[of\ A]$ **by** *auto*
next
case *False*
let $?A' = A \cup \{a\}$
from *False* **have** $A = ?A' - \{a\}$ **by** *blast*
moreover **from** *assms* **have** $\neg finite\ ?A'$ **by** *auto*
ultimately **obtain** f **where** $bij\text{-betw}\ f\ ?A'\ A$
using *infinite-imp-bij-betw*[*of* $?A'\ a$] **by** *auto*
then **have** $bij\text{-betw}\ (inv\text{-into}\ ?A'\ f)\ A\ ?A'$ **by** (rule *bij-betw-inv-into*)
then **show** *?thesis* **by** *auto*
qed

lemma *bij-betw-inv-into-left*: $bij\text{-betw}\ f\ A\ A' \implies a \in A \implies inv\text{-into}\ A\ f\ (f\ a) = a$
unfolding *bij-betw-def* **by** *clarify* (rule *inv-into-f-f*)

lemma *bij-betw-inv-into-right*: $bij\text{-betw}\ f\ A\ A' \implies a' \in A' \implies f\ (inv\text{-into}\ A\ f\ a') = a'$
unfolding *bij-betw-def* **using** *f-inv-into-f* **by** *force*

lemma *bij-betw-inv-into-subset*:

$bij\text{-betw}\ f\ A\ A' \implies B \subseteq A \implies f\ ` B = B' \implies bij\text{-betw}\ (inv\text{-into}\ A\ f)\ B'\ B$
by (auto simp: *bij-betw-def* *intro: inj-on-inv-into*)

25.8 Specification package – Hilbertized version

lemma *exE-some*: $Ex\ P \implies c \equiv Eps\ P \implies P\ c$

by (*simp only: someI-ex*)

ML-file \langle Tools/choice-specification.ML \rangle

25.9 Complete Distributive Lattices – Properties depending on Hilbert Choice

context *complete-distrib-lattice*
begin

lemma *Sup-Inf*: $\bigsqcup (Inf \ 'A) = \sqcap (Sup \ ' \{f \ 'A \mid f. \forall B \in A. f B \in B\})$

proof (*rule order.antisym*)

show $\bigsqcup (Inf \ 'A) \leq \sqcap (Sup \ ' \{f \ 'A \mid f. \forall B \in A. f B \in B\})$

using *Inf-lower2 Sup-upper*

by (*fastforce simp add: intro: Sup-least INF-greatest*)

next

show $\sqcap (Sup \ ' \{f \ 'A \mid f. \forall B \in A. f B \in B\}) \leq \bigsqcup (Inf \ 'A)$

proof (*simp add: Inf-Sup, rule SUP-least, simp, safe*)

fix *f*

assume $\forall Y. (\exists f. Y = f \ 'A \wedge (\forall Y \in A. f Y \in Y)) \longrightarrow f Y \in Y$

then have $B: \bigwedge F. (\forall Y \in A. F Y \in Y) \implies \exists Z \in A. f (F \ 'A) = F Z$

by *auto*

show $\sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq \bigsqcup (Inf \ 'A)$

proof (*cases* $\exists Z \in A. \sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq Inf Z$)

case *True*

from *this* obtain *Z* where [*simp*]: $Z \in A$ and $A: \sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq Inf Z$

by *blast*

have $B: \dots \leq \bigsqcup (Inf \ 'A)$

by (*simp add: SUP-upper*)

from *A* and *B* show *?thesis*

by *simp*

next

case *False*

then have $X: \bigwedge Z. Z \in A \implies \exists x. x \in Z \wedge \neg \sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq x$

using *Inf-greatest* by *blast*

define *F* where $F = (\lambda Z. SOME x. x \in Z \wedge \neg \sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq x)$

have $C: \bigwedge Y. Y \in A \implies F Y \in Y$

using *X* by (*simp add: F-def, rule someI2-ex, auto*)

have $E: \bigwedge Y. Y \in A \implies \neg \sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq F Y$

using *X* by (*simp add: F-def, rule someI2-ex, auto*)

from *C* and *B* obtain *Z* where $D: Z \in A$ and $Y: f (F \ 'A) = F Z$

by *blast*

from *E* and *D* have $W: \neg \sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq F Z$

by *simp*

have $\sqcap (f \ ' \{f \ 'A \mid f. \forall Y \in A. f Y \in Y\}) \leq f (F \ 'A)$

using *C* by (*blast intro: INF-lower*)

```

    with W Y show ?thesis
      by simp
    qed
  qed
qed

```

lemma *dual-complete-distrib-lattice*:

```

class.complete-distrib-lattice Sup Inf sup (≥) (>) inf ⊔ ⊥
by (simp add: class.complete-distrib-lattice.intro [OF dual-complete-lattice]
    class.complete-distrib-lattice-axioms-def Sup-Inf)

```

lemma *sup-Inf*: $a \sqcup \sqcap B = \sqcap ((\sqcup) a \text{ ‘ } B)$

proof (*rule order.antisym*)

show $a \sqcup \sqcap B \leq \sqcap ((\sqcup) a \text{ ‘ } B)$

using *Inf-lower sup.mono* **by** (*fastforce intro: INF-greatest*)

next

have $\sqcap ((\sqcup) a \text{ ‘ } B) \leq \sqcap (\text{Sup ‘ } \{\{f \{a\}, f B\} \mid f. f \{a\} = a \wedge f B \in B\})$

by (*rule INF-greatest, auto simp add: INF-lower*)

also have $\dots = \sqcup (\text{Inf ‘ } \{\{a\}, B\})$

by (*unfold Sup-Inf, simp*)

finally show $\sqcap ((\sqcup) a \text{ ‘ } B) \leq a \sqcup \sqcap B$

by *simp*

qed

lemma *inf-Sup*: $a \sqcap \sqcup B = \sqcup ((\sqcap) a \text{ ‘ } B)$

using *dual-complete-distrib-lattice*

by (*rule complete-distrib-lattice.sup-Inf*)

lemma *INF-SUP*: $(\sqcap y. \sqcup x. P x y) = (\sqcup f. \sqcap x. P (f x) x)$

proof (*rule order.antisym*)

show $(\text{SUP } x. \text{INF } y. P (x y) y) \leq (\text{INF } y. \text{SUP } x. P x y)$

by (*rule SUP-least, rule INF-greatest, rule SUP-upper2, simp-all, rule INF-lower2, simp, blast*)

next

have $(\text{INF } y. \text{SUP } x. ((P x y))) \leq \text{Inf } (\text{Sup ‘ } \{\{P x y \mid x. \text{True}\} \mid y. \text{True}\})$

(**is** $?A \leq ?B$)

proof (*rule INF-greatest, clarsimp*)

fix y

have $?A \leq (\text{SUP } x. P x y)$

by (*rule INF-lower, simp*)

also have $\dots \leq \text{Sup } \{uu. \exists x. uu = P x y\}$

by (*simp add: full-SetCompr-eq*)

finally show $?A \leq \text{Sup } \{uu. \exists x. uu = P x y\}$

by *simp*

qed

also have $\dots \leq (\text{SUP } x. \text{INF } y. P (x y) y)$

proof (*subst Inf-Sup, rule SUP-least, clarsimp*)

fix f

assume $A: \forall Y. (\exists y. Y = \{uu. \exists x. uu = P x y\}) \longrightarrow f Y \in Y$

```

have  $\prod (f \text{ ' } \{uu. \exists y. uu = \{uu. \exists x. uu = P x y\}\}) \leq$ 
   $(\prod y. P (\text{SOME } x. f \{P x y \mid x. \text{True}\} = P x y) y)$ 
proof (rule INF-greatest, clarsimp)
  fix  $y$ 
    have  $(\text{INF } x \in \{uu. \exists y. uu = \{uu. \exists x. uu = P x y\}\}. f x) \leq f \{uu. \exists x. uu$ 
     $= P x y\}$ 
      by (rule INF-lower, blast)
    also have  $\dots \leq P (\text{SOME } x. f \{uu. \exists x. uu = P x y\} = P x y) y$ 
      by (rule someI2-ex) (use A in auto)
    finally show  $\prod (f \text{ ' } \{uu. \exists y. uu = \{uu. \exists x. uu = P x y\}\}) \leq$ 
       $P (\text{SOME } x. f \{uu. \exists x. uu = P x y\} = P x y) y$ 
      by simp
    qed
  also have  $\dots \leq (\text{SUP } x. \text{INF } y. P (x y) y)$ 
    by (rule SUP-upper, simp)
  finally show  $\prod (f \text{ ' } \{uu. \exists y. uu = \{uu. \exists x. uu = P x y\}\}) \leq (\bigsqcup x. \prod y. P$ 
   $(x y) y)$ 
    by simp
  qed
  finally show  $(\text{INF } y. \text{SUP } x. P x y) \leq (\text{SUP } x. \text{INF } y. P (x y) y)$ 
    by simp
qed

lemma INF-SUP-set:  $(\prod B \in A. \bigsqcup (g \text{ ' } B)) = (\bigsqcup B \in \{f \text{ ' } A \mid f. \forall C \in A. f C \in C\}.$ 
 $\prod (g \text{ ' } B))$ 
  (is  $- = (\bigsqcup B \in ?F. -)$ )
proof (rule order.antisym)
  have  $\prod ((g \circ f) \text{ ' } A) \leq \bigsqcup (g \text{ ' } B)$  if  $\bigwedge B. B \in A \implies f B \in B$  B  $B \in A$  for  $f B$ 
    using that by (auto intro: SUP-upper2 INF-lower2)
  then show  $(\bigsqcup x \in ?F. \prod a \in x. g a) \leq (\prod x \in A. \bigsqcup a \in x. g a)$ 
    by (auto intro!: SUP-least INF-greatest simp add: image-comp)
next
  show  $(\prod x \in A. \bigsqcup a \in x. g a) \leq (\bigsqcup x \in ?F. \prod a \in x. g a)$ 
  proof (cases  $\{\} \in A$ )
    case True
      then show ?thesis
      by (rule INF-lower2) simp-all
    next
    case False
      {fix  $x$ 
        have  $(\prod x \in A. \bigsqcup x \in x. g x) \leq (\bigsqcup u. \text{if } x \in A \text{ then if } u \in x \text{ then } g u \text{ else } \perp \text{ else}$ 
         $\top)$ 
          proof (cases  $x \in A$ )
            case True
              then show ?thesis
              by (intro INF-lower2 SUP-least SUP-upper2) auto
            qed auto
          }
      }

```

then have $(\prod Y \in A. \sqcup a \in Y. g a) \leq (\prod Y. \sqcup y. \text{if } Y \in A \text{ then if } y \in Y \text{ then } g y \text{ else } \perp \text{ else } \top)$
by (rule *INF-greatest*)
also have $\dots = (\sqcup x. \prod Y. \text{if } Y \in A \text{ then if } x \in Y \text{ then } g (x Y) \text{ else } \perp \text{ else } \top)$
by (simp only: *INF-SUP*)
also have $\dots \leq (\sqcup x \in ?F. \prod a \in x. g a)$
proof (rule *SUP-least*)
show $(\prod B. \text{if } B \in A \text{ then if } x \in B \text{ then } g (x B) \text{ else } \perp \text{ else } \top)$
 $\leq (\sqcup x \in ?F. \prod x \in x. g x)$ **for** x
proof –
define G **where** $G \equiv \lambda Y. \text{if } x \in Y \text{ then } x Y \text{ else } (\text{SOME } x. x \in Y)$
have $\forall Y \in A. G Y \in Y$
using *False some-in-eq G-def by auto*
then have $A: G ' A \in ?F$
by *blast*
show $(\prod Y. \text{if } Y \in A \text{ then if } x \in Y \text{ then } g (x Y) \text{ else } \perp \text{ else } \top) \leq$
 $(\sqcup x \in ?F. \prod x \in x. g x)$
by (fastforce simp: *G-def intro: SUP-upper2 [OF A] INF-greatest INF-lower2*)
qed
qed
finally show *?thesis by simp*
qed
qed

lemma *SUP-INF*: $(\sqcup y. \prod x. P x y) = (\prod x. \sqcup y. P (x y) y)$
using *dual-complete-distrib-lattice*
by (rule *complete-distrib-lattice.INF-SUP*)

lemma *SUP-INF-set*: $(\sqcup x \in A. \prod (g ' x)) = (\prod x \in \{f ' A \mid f. \forall Y \in A. f Y \in Y\}. \sqcup (g ' x))$
using *dual-complete-distrib-lattice*
by (rule *complete-distrib-lattice.INF-SUP-set*)

end

context *complete-distrib-lattice*
begin

lemma *sup-INF*: $a \sqcup (\prod b \in B. f b) = (\prod b \in B. a \sqcup f b)$
by (simp add: *sup-Inf image-comp*)

lemma *inf-SUP*: $a \sqcap (\sqcup b \in B. f b) = (\sqcup b \in B. a \sqcap f b)$
by (simp add: *inf-Sup image-comp*)

lemma *Inf-sup*: $\prod B \sqcup a = (\prod b \in B. b \sqcup a)$
by (simp add: *sup-Inf sup-commute*)

lemma *Sup-inf*: $\bigsqcup B \sqcap a = (\bigsqcup b \in B. b \sqcap a)$
by (*simp add: inf-Sup inf-commute*)

lemma *INF-sup*: $(\prod b \in B. f b) \sqcup a = (\prod b \in B. f b \sqcup a)$
by (*simp add: sup-INF sup-commute*)

lemma *SUP-inf*: $(\bigsqcup b \in B. f b) \sqcap a = (\bigsqcup b \in B. f b \sqcap a)$
by (*simp add: inf-SUP inf-commute*)

lemma *Inf-sup-eq-top-iff*: $(\prod B \sqcup a = \top) \longleftrightarrow (\forall b \in B. b \sqcup a = \top)$
by (*simp only: Inf-sup INF-top-conv*)

lemma *Sup-inf-eq-bot-iff*: $(\bigsqcup B \sqcap a = \perp) \longleftrightarrow (\forall b \in B. b \sqcap a = \perp)$
by (*simp only: Sup-inf SUP-bot-conv*)

lemma *INF-sup-distrib2*: $(\prod a \in A. f a) \sqcup (\prod b \in B. g b) = (\prod a \in A. \prod b \in B. f a \sqcup g b)$
by (*subst INF-commute*) (*simp add: sup-INF INF-sup*)

lemma *SUP-inf-distrib2*: $(\bigsqcup a \in A. f a) \sqcap (\bigsqcup b \in B. g b) = (\bigsqcup a \in A. \bigsqcup b \in B. f a \sqcap g b)$
by (*subst SUP-commute*) (*simp add: inf-SUP SUP-inf*)

end

instantiation *set* :: (type) complete-distrib-lattice
begin
instance proof (*standard, clarsimp*)
fix *A* :: (('a set) set) set
fix *x*::'a
assume *A*: $\forall S \in A. \exists X \in S. x \in X$
define *F* **where** $F \equiv \lambda Y. \text{SOME } X. Y \in A \wedge X \in Y \wedge x \in X$
have $(\forall S \in F \text{ ' } A. x \in S)$
using *A* **unfolding** *F-def* **by** (*fastforce intro: someI2-ex*)
moreover have $\forall Y \in A. F Y \in Y$
using *A* **unfolding** *F-def* **by** (*fastforce intro: someI2-ex*)
then have $\exists f. F \text{ ' } A = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)$
by *blast*
ultimately show $\exists X. (\exists f. X = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)) \wedge (\forall S \in X. x \in S)$
by *auto*
qed
end

instance *set* :: (type) complete-boolean-algebra ..

instantiation *fun* :: (type, complete-distrib-lattice) complete-distrib-lattice
begin
instance by *standard* (*simp add: le-fun-def INF-SUP-set image-comp*)
end

instance *fun* :: (type, complete-boolean-algebra) complete-boolean-algebra ..

context *complete-linorder*

begin

subclass *complete-distrib-lattice*

proof (*standard, rule ccontr*)

fix *A* :: 'a set set

let $?F = \{f \text{ ' } A \mid f. \forall Y \in A. f Y \in Y\}$

assume $\neg \bigcap (Sup \text{ ' } A) \leq \bigsqcup (Inf \text{ ' } ?F)$

then have $C: \bigcap (Sup \text{ ' } A) > \bigsqcup (Inf \text{ ' } ?F)$

by (*simp add: not-le*)

show *False*

proof (*cases* $\exists z. \bigcap (Sup \text{ ' } A) > z \wedge z > \bigsqcup (Inf \text{ ' } ?F)$)

case *True*

then obtain *z* **where** $A: z < \bigcap (Sup \text{ ' } A)$ **and** $X: z > \bigsqcup (Inf \text{ ' } ?F)$

by *blast*

then have $B: \bigwedge Y. Y \in A \implies \exists k \in Y. z < k$

using *local.less-Sup-iff* **by** (*force dest: less-INF-D*)

define *G* **where** $G \equiv \lambda Y. \text{SOME } k. k \in Y \wedge z < k$

have $E: \bigwedge Y. Y \in A \implies G Y \in Y$

using *B* **unfolding** *G-def* **by** (*fastforce intro: someI2-ex*)

have $z \leq Inf (G \text{ ' } A)$

proof (*rule INF-greatest*)

show $\bigwedge Y. Y \in A \implies z \leq G Y$

using *B* **unfolding** *G-def* **by** (*fastforce intro: someI2-ex*)

qed

also have $\dots \leq \bigsqcup (Inf \text{ ' } ?F)$

by (*rule SUP-upper*) (*use E in blast*)

finally have $z \leq \bigsqcup (Inf \text{ ' } ?F)$

by *simp*

with *X* **show** *?thesis*

using *local.not-less* **by** *blast*

next

case *False*

have $B: \bigwedge Y. Y \in A \implies \exists k \in Y. \bigsqcup (Inf \text{ ' } ?F) < k$

using *C* *local.less-Sup-iff* **by** (*force dest: less-INF-D*)

define *G* **where** $G \equiv \lambda Y. \text{SOME } k. k \in Y \wedge \bigsqcup (Inf \text{ ' } ?F) < k$

have $E: \bigwedge Y. Y \in A \implies G Y \in Y$

using *B* **unfolding** *G-def* **by** (*fastforce intro: someI2-ex*)

have $\bigwedge Y. Y \in A \implies \bigcap (Sup \text{ ' } A) \leq G Y$

using *B* *False* *local.leI* **unfolding** *G-def* **by** (*fastforce intro: someI2-ex*)

then have $\bigcap (Sup \text{ ' } A) \leq Inf (G \text{ ' } A)$

by (*simp add: local.INF-greatest*)

also have $Inf (G \text{ ' } A) \leq \bigsqcup (Inf \text{ ' } ?F)$

by (*rule SUP-upper*) (*use E in blast*)


```

    finally have  $\sqcap (Sup \ 'A) \leq \sqcup (Inf \ ?F)$ 
      by simp
    with C show ?thesis
      using not-less by blast
  qed
qed
end

end

```

26 Zorn’s Lemma and the Well-ordering Theorem

```

theory Zorn
  imports Order-Relation Hilbert-Choice
begin

```

26.1 Zorn’s Lemma for the Subset Relation

26.1.1 Results that do not require an order

Let P be a binary predicate on the set A .

```

locale pred-on =
  fixes A :: 'a set
  and P :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infix  $\sqsubseteq$  50)
begin

```

```

abbreviation Peq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infix  $\sqsubseteq$  50)
  where  $x \sqsubseteq y \equiv P^{==} x y$ 

```

A chain is a totally ordered subset of A .

```

definition chain :: 'a set  $\Rightarrow$  bool
  where chain C  $\longleftrightarrow C \subseteq A \wedge (\forall x \in C. \forall y \in C. x \sqsubseteq y \vee y \sqsubseteq x)$ 

```

We call a chain that is a proper superset of some set X , but not necessarily a chain itself, a superchain of X .

```

abbreviation superchain :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool (infix  $<c$  50)
  where  $X <c C \equiv chain C \wedge X \subset C$ 

```

A maximal chain is a chain that does not have a superchain.

```

definition maxchain :: 'a set  $\Rightarrow$  bool
  where maxchain C  $\longleftrightarrow chain C \wedge (\nexists S. C <c S)$ 

```

We define the successor of a set to be an arbitrary superchain, if such exists, or the set itself, otherwise.

```

definition suc :: 'a set  $\Rightarrow$  'a set
  where  $suc C = (if \neg chain C \vee maxchain C then C else (SOME D. C <c D))$ 

```

lemma *chainI* [*Pure.intro?*]: $C \subseteq A \implies (\bigwedge x y. x \in C \implies y \in C \implies x \sqsubseteq y \vee y \sqsubseteq x) \implies \text{chain } C$

unfolding *chain-def* **by** *blast*

lemma *chain-total*: $\text{chain } C \implies x \in C \implies y \in C \implies x \sqsubseteq y \vee y \sqsubseteq x$

by (*simp add: chain-def*)

lemma *not-chain-suc* [*simp*]: $\neg \text{chain } X \implies \text{suc } X = X$

by (*simp add: suc-def*)

lemma *maxchain-suc* [*simp*]: $\text{maxchain } X \implies \text{suc } X = X$

by (*simp add: suc-def*)

lemma *suc-subset*: $X \subseteq \text{suc } X$

by (*auto simp: suc-def maxchain-def intro: someI2*)

lemma *chain-empty* [*simp*]: $\text{chain } \{\}$

by (*auto simp: chain-def*)

lemma *not-maxchain-Some*: $\text{chain } C \implies \neg \text{maxchain } C \implies C <_c (\text{SOME } D. C <_c D)$

by (*rule someI-ex*) (*auto simp: maxchain-def*)

lemma *suc-not-equals*: $\text{chain } C \implies \neg \text{maxchain } C \implies \text{suc } C \neq C$

using *not-maxchain-Some* **by** (*auto simp: suc-def*)

lemma *subset-suc*:

assumes $X \subseteq Y$

shows $X \subseteq \text{suc } Y$

using *assms* **by** (*rule subset-trans*) (*rule suc-subset*)

We build a set \mathcal{C} that is closed under applications of *suc* and contains the union of all its subsets.

inductive-set *suc-Union-closed* (\mathcal{C})

where

$\text{suc: } X \in \mathcal{C} \implies \text{suc } X \in \mathcal{C}$

| *Union* [*unfolded Pow-iff*]: $X \in \text{Pow } \mathcal{C} \implies \bigcup X \in \mathcal{C}$

Since the empty set as well as the set itself is a subset of every set, \mathcal{C} contains at least $\{\} \in \mathcal{C}$ and $\bigcup \mathcal{C} \in \mathcal{C}$.

lemma *suc-Union-closed-empty*: $\{\} \in \mathcal{C}$

and *suc-Union-closed-Union*: $\bigcup \mathcal{C} \in \mathcal{C}$

using *Union* [*of* $\{\}$] **and** *Union* [*of* \mathcal{C}] **by** *simp-all*

Thus closure under *suc* will hit a maximal chain eventually, as is shown below.

lemma *suc-Union-closed-induct* [*consumes 1, case-names suc Union, induct pred: suc-Union-closed*]:

assumes $X \in \mathcal{C}$
and $\bigwedge X. X \in \mathcal{C} \implies Q X \implies Q (\text{suc } X)$
and $\bigwedge X. X \subseteq \mathcal{C} \implies \forall x \in X. Q x \implies Q (\bigcup X)$
shows $Q X$
using *assms* **by** *induct blast+*

lemma *suc-Union-closed-cases* [*consumes 1, case-names suc Union, cases pred: suc-Union-closed*]:

assumes $X \in \mathcal{C}$
and $\bigwedge Y. X = \text{suc } Y \implies Y \in \mathcal{C} \implies Q$
and $\bigwedge Y. X = \bigcup Y \implies Y \subseteq \mathcal{C} \implies Q$
shows Q
using *assms* **by** *cases simp-all*

On chains, *suc* yields a chain.

lemma *chain-suc*:

assumes *chain* X
shows *chain* ($\text{suc } X$)
using *assms*
by (*cases* \neg *chain* $X \vee$ *maxchain* X) (*force simp: suc-def dest: not-maxchain-Some*)+

lemma *chain-sucD*:

assumes *chain* X
shows $\text{suc } X \subseteq A \wedge$ *chain* ($\text{suc } X$)
proof –
from \langle *chain* X \rangle **have** $*$: *chain* ($\text{suc } X$)
by (*rule chain-suc*)
then have $\text{suc } X \subseteq A$
unfolding *chain-def* **by** *blast*
with $*$ **show** *?thesis* **by** *blast*
qed

lemma *suc-Union-closed-total'*:

assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
and $*$: $\bigwedge Z. Z \in \mathcal{C} \implies Z \subseteq Y \implies Z = Y \vee \text{suc } Z \subseteq Y$
shows $X \subseteq Y \vee \text{suc } Y \subseteq X$
using $\langle X \in \mathcal{C} \rangle$
proof *induct*
case ($\text{suc } X$)
with $*$ **show** *?case* **by** (*blast del: subsetI intro: subset-suc*)
next
case *Union*
then show *?case* **by** *blast*
qed

lemma *suc-Union-closed-subsetD*:

assumes $Y \subseteq X$ **and** $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
shows $X = Y \vee \text{suc } Y \subseteq X$
using *assms*(2,3,1)

```

proof (induct arbitrary: Y)
  case (suc X)
    note * =  $\langle \bigwedge Y. Y \in \mathcal{C} \implies Y \subseteq X \implies X = Y \vee \text{suc } Y \subseteq X \rangle$ 
    with suc-Union-closed-total' [OF  $\langle Y \in \mathcal{C} \rangle \langle X \in \mathcal{C} \rangle$ ]
    have  $Y \subseteq X \vee \text{suc } X \subseteq Y$  by blast
    then show ?case
    proof
      assume  $Y \subseteq X$ 
      with * and  $\langle Y \in \mathcal{C} \rangle$  subset-suc show ?thesis
      by fastforce
    next
      assume  $\text{suc } X \subseteq Y$ 
      with  $\langle Y \subseteq \text{suc } X \rangle$  show ?thesis by blast
    qed
  next
    case (Union X)
    show ?case
    proof (rule ccontr)
      assume  $\neg$  ?thesis
      with  $\langle Y \subseteq \bigcup X \rangle$  obtain x y z
      where  $\neg \text{suc } Y \subseteq \bigcup X$ 
      and  $x \in X$  and  $y \in x$  and  $y \notin Y$ 
      and  $z \in \text{suc } Y$  and  $\forall x \in X. z \notin x$  by blast
      with  $\langle X \subseteq \mathcal{C} \rangle$  have  $x \in \mathcal{C}$  by blast
      from Union and  $\langle x \in X \rangle$  have *:  $\bigwedge y. y \in \mathcal{C} \implies y \subseteq x \implies x = y \vee \text{suc } y \subseteq$ 
x
      by blast
      with suc-Union-closed-total' [OF  $\langle Y \in \mathcal{C} \rangle \langle x \in \mathcal{C} \rangle$ ] have  $Y \subseteq x \vee \text{suc } x \subseteq Y$ 
      by blast
      then show False
      proof
        assume  $Y \subseteq x$ 
        with * [OF  $\langle Y \in \mathcal{C} \rangle$ ]  $\langle y \in x \rangle \langle y \notin Y \rangle \langle x \in X \rangle \langle \neg \text{suc } Y \subseteq \bigcup X \rangle$  show False
        by blast
      next
        assume  $\text{suc } x \subseteq Y$ 
        with  $\langle y \notin Y \rangle$  suc-subset  $\langle y \in x \rangle$  show False by blast
      qed
    qed
  qed

```

The elements of \mathcal{C} are totally ordered by the subset relation.

lemma suc-Union-closed-total:

assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$

shows $X \subseteq Y \vee Y \subseteq X$

proof (cases $\forall Z \in \mathcal{C}. Z \subseteq Y \longrightarrow Z = Y \vee \text{suc } Z \subseteq Y$)

case True

with suc-Union-closed-total' [OF *assms*]

have $X \subseteq Y \vee \text{suc } Y \subseteq X$ **by** blast

```

with suc-subset [of  $Y$ ] show ?thesis by blast
next
  case False
  then obtain  $Z$  where  $Z \in \mathcal{C}$  and  $Z \subseteq Y$  and  $Z \neq Y$  and  $\neg \text{suc } Z \subseteq Y$ 
    by blast
  with suc-Union-closed-subsetD and  $\langle Y \in \mathcal{C} \rangle$  show ?thesis
    by blast
qed

```

Once we hit a fixed point w.r.t. *suc*, all other elements of \mathcal{C} are subsets of this fixed point.

```

lemma suc-Union-closed-suc:
  assumes  $X \in \mathcal{C}$  and  $Y \in \mathcal{C}$  and  $\text{suc } Y = Y$ 
  shows  $X \subseteq Y$ 
  using  $\langle X \in \mathcal{C} \rangle$ 
proof induct
  case (suc X)
  with  $\langle Y \in \mathcal{C} \rangle$  and suc-Union-closed-subsetD have  $X = Y \vee \text{suc } X \subseteq Y$ 
    by blast
  then show ?case
    by (auto simp: \langle suc Y = Y \rangle)
next
  case Union
  then show ?case by blast
qed

```

```

lemma eq-suc-Union:
  assumes  $X \in \mathcal{C}$ 
  shows  $\text{suc } X = X \iff X = \bigcup \mathcal{C}$ 
    (is ?lhs \iff ?rhs)
proof
  assume ?lhs
  then have  $\bigcup \mathcal{C} \subseteq X$ 
    by (rule suc-Union-closed-suc [OF suc-Union-closed-Union \langle X \in \mathcal{C} \rangle])
  with  $\langle X \in \mathcal{C} \rangle$  show ?rhs
    by blast
next
  from  $\langle X \in \mathcal{C} \rangle$  have  $\text{suc } X \in \mathcal{C}$  by (rule suc)
  then have  $\text{suc } X \subseteq \bigcup \mathcal{C}$  by blast
  moreover assume ?rhs
  ultimately have  $\text{suc } X \subseteq X$  by simp
  moreover have  $X \subseteq \text{suc } X$  by (rule suc-subset)
  ultimately show ?lhs ..
qed

```

```

lemma suc-in-carrier:
  assumes  $X \subseteq A$ 
  shows  $\text{suc } X \subseteq A$ 
  using assms

```

by (cases \neg chain $X \vee$ maxchain X) (auto dest: chain-sucD)

lemma *suc-Union-closed-in-carrier*:

assumes $X \in \mathcal{C}$

shows $X \subseteq A$

using *assms*

by *induct* (auto dest: *suc-in-carrier*)

All elements of \mathcal{C} are chains.

lemma *suc-Union-closed-chain*:

assumes $X \in \mathcal{C}$

shows chain X

using *assms*

proof *induct*

case (*suc* X)

then show ?*case*

using *not-maxchain-Some* by (*simp* add: *suc-def*)

next

case (*Union* X)

then have $\bigcup X \subseteq A$

by (auto dest: *suc-Union-closed-in-carrier*)

moreover have $\forall x \in \bigcup X. \forall y \in \bigcup X. x \sqsubseteq y \vee y \sqsubseteq x$

proof (*intro ballI*)

fix $x y$

assume $x \in \bigcup X$ and $y \in \bigcup X$

then obtain $u v$ where $x \in u$ and $u \in X$ and $y \in v$ and $v \in X$

by *blast*

with *Union* have $u \in \mathcal{C}$ and $v \in \mathcal{C}$ and chain u and chain v

by *blast+*

with *suc-Union-closed-total* have $u \subseteq v \vee v \subseteq u$

by *blast*

then show $x \sqsubseteq y \vee y \sqsubseteq x$

proof

assume $u \subseteq v$

from \langle chain v \rangle show ?*thesis*

proof (*rule chain-total*)

show $y \in v$ by *fact*

show $x \in v$ using $\langle u \subseteq v \rangle$ and $\langle x \in u \rangle$ by *blast*

qed

next

assume $v \subseteq u$

from \langle chain u \rangle show ?*thesis*

proof (*rule chain-total*)

show $x \in u$ by *fact*

show $y \in u$ using $\langle v \subseteq u \rangle$ and $\langle y \in v \rangle$ by *blast*

qed

qed

qed

ultimately show ?*case* unfolding *chain-def* ..

qed

26.1.2 Hausdorff’s Maximum Principle

There exists a maximal totally ordered subset of A . (Note that we do not require A to be partially ordered.)

theorem *Hausdorff*: $\exists C. \text{maxchain } C$

proof –

let $?M = \bigcup C$

have *maxchain* $?M$

proof (*rule ccontr*)

assume $\neg ?thesis$

then have *suc* $?M \neq ?M$

using *suc-not-equals* and *suc-Union-closed-chain* [*OF suc-Union-closed-Union*]

by *simp*

moreover have *suc* $?M = ?M$

using *eq-suc-Union* [*OF suc-Union-closed-Union*] by *simp*

ultimately show *False* by *contradiction*

qed

then show *?thesis* by *blast*

qed

Make notation \mathcal{C} available again.

no-notation *suc-Union-closed* (\mathcal{C})

lemma *chain-extend*: $\text{chain } C \implies z \in A \implies \forall x \in C. x \sqsubseteq z \implies \text{chain } (\{z\} \cup C)$

unfolding *chain-def* by *blast*

lemma *maxchain-imp-chain*: $\text{maxchain } C \implies \text{chain } C$

by (*simp add: maxchain-def*)

end

Hide constant *pred-on.suc-Union-closed*, which was just needed for the proof of Hausdorff’s maximum principle.

hide-const *pred-on.suc-Union-closed*

lemma *chain-mono*:

assumes $\bigwedge x y. x \in A \implies y \in A \implies P x y \implies Q x y$

and *pred-on.chain* $A P C$

shows *pred-on.chain* $A Q C$

using *assms* unfolding *pred-on.chain-def* by *blast*

26.1.3 Results for the proper subset relation

interpretation *subset: pred-on* $A (\subset)$ for A .

lemma *subset-maxchain-max*:

assumes *subset.maxchain* $A C$
and $X \in A$
and $\bigcup C \subseteq X$
shows $\bigcup C = X$
proof (*rule ccontr*)
let $?C = \{X\} \cup C$
from $\langle \text{subset.maxchain } A C \rangle$ **have** *subset.chain* $A C$
and $*$: $\bigwedge S. \text{subset.chain } A S \implies \neg C \subset S$
by (*auto simp: subset.maxchain-def*)
moreover have $\forall x \in C. x \subseteq X$ **using** $\langle \bigcup C \subseteq X \rangle$ **by** *auto*
ultimately have *subset.chain* $A ?C$
using *subset.chain-extend* [*of* $A C X$] **and** $\langle X \in A \rangle$ **by** *auto*
moreover assume $**$: $\bigcup C \neq X$
moreover from $**$ **have** $C \subset ?C$ **using** $\langle \bigcup C \subseteq X \rangle$ **by** *auto*
ultimately show *False* **using** $*$ **by** *blast*
qed

lemma *subset-chain-def*: $\bigwedge A. \text{subset.chain } A C = (C \subseteq A \wedge (\forall X \in C. \forall Y \in C. X \subseteq Y \vee Y \subseteq X))$
by (*auto simp: subset.chain-def*)

lemma *subset-chain-insert*:
 $\text{subset.chain } A (\text{insert } B B) \longleftrightarrow B \in A \wedge (\forall X \in B. X \subseteq B \vee B \subseteq X) \wedge \text{subset.chain } A B$
by (*fastforce simp add: subset-chain-def*)

26.1.4 Zorn’s lemma

If every chain has an upper bound, then there is a maximal set.

theorem *subset-Zorn*:

assumes $\bigwedge C. \text{subset.chain } A C \implies \exists U \in A. \forall X \in C. X \subseteq U$

shows $\exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$

proof –

from *subset.Hausdorff* [*of* A] **obtain** M **where** *subset.maxchain* $A M$..

then have *subset.chain* $A M$

by (*rule subset.maxchain-imp-chain*)

with *assms* **obtain** Y **where** $Y \in A$ **and** $\forall X \in M. X \subseteq Y$

by *blast*

moreover have $\forall X \in A. Y \subseteq X \longrightarrow Y = X$

proof (*intro ballI impI*)

fix X

assume $X \in A$ **and** $Y \subseteq X$

show $Y = X$

proof (*rule ccontr*)

assume $\neg ?thesis$

with $\langle Y \subseteq X \rangle$ **have** $\neg X \subseteq Y$ **by** *blast*

from *subset.chain-extend* [*OF* $\langle \text{subset.chain } A M \rangle \langle X \in A \rangle$] **and** $\langle \forall X \in M. X \subseteq Y \rangle$

have *subset.chain* $A (\{X\} \cup M)$


```

    using  $\langle Y \subseteq X \rangle$  by auto
  moreover have  $M \subseteq \{X\} \cup M$ 
    using  $\langle \forall X \in M. X \subseteq Y \rangle$  and  $\langle \neg X \subseteq Y \rangle$  by auto
  ultimately show False
    using  $\langle \text{subset.maxchain } A \ M \rangle$  by (auto simp: subset.maxchain-def)
  qed
  qed
  ultimately show ?thesis by blast
  qed

```

Alternative version of Zorn’s lemma for the subset relation.

lemma *subset-Zorn'*:

```

  assumes  $\bigwedge C. \text{subset.chain } A \ C \implies \bigcup C \in A$ 
  shows  $\exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$ 
  proof -
    from subset.Hausdorff [of A] obtain M where subset.maxchain A M ..
    then have subset.chain A M
      by (rule subset.maxchain-imp-chain)
    with assms have  $\bigcup M \in A$  .
    moreover have  $\forall Z \in A. \bigcup M \subseteq Z \longrightarrow \bigcup M = Z$ 
    proof (intro ball impI)
      fix Z
      assume  $Z \in A$  and  $\bigcup M \subseteq Z$ 
      with subset.maxchain-max [OF  $\langle \text{subset.maxchain } A \ M \rangle$ ]
      show  $\bigcup M = Z$  .
    qed
    ultimately show ?thesis by blast
  qed

```

26.2 Zorn’s Lemma for Partial Orders

Relate old to new definitions.

definition *chain-subset* :: 'a set set \Rightarrow bool (*chain_⊆*)
 where *chain_⊆* C $\longleftrightarrow (\forall A \in C. \forall B \in C. A \subseteq B \vee B \subseteq A)$

definition *chains* :: 'a set set \Rightarrow 'a set set set
 where *chains* A = {C. C \subseteq A \wedge *chain_⊆* C}

definition *Chains* :: ('a \times 'a) set \Rightarrow 'a set set
 where *Chains* r = {C. $\forall a \in C. \forall b \in C. (a, b) \in r \vee (b, a) \in r$ }

lemma *chains-extend*: $c \in \text{chains } S \implies z \in S \implies \forall x \in c. x \subseteq z \implies \{z\} \cup c \in \text{chains } S$

```

  for z :: 'a set
  unfolding chains-def chain-subset-def by blast

```

lemma *mono-Chains*: $r \subseteq s \implies \text{Chains } r \subseteq \text{Chains } s$
 unfolding Chains-def by blast

lemma *chain-subset-alt-def*: $\text{chain}_{\subseteq} C = \text{subset.chain UNIV } C$
unfolding *chain-subset-def subset.chain-def* **by** *fast*

lemma *chains-alt-def*: $\text{chains } A = \{C. \text{subset.chain } A \ C\}$
by (*simp add: chains-def chain-subset-alt-def subset.chain-def*)

lemma *Chains-subset*: $\text{Chains } r \subseteq \{C. \text{pred-on.chain UNIV } (\lambda x y. (x, y) \in r) \ C\}$
by (*force simp add: Chains-def pred-on.chain-def*)

lemma *Chains-subset'*:
assumes *refl r*
shows $\{C. \text{pred-on.chain UNIV } (\lambda x y. (x, y) \in r) \ C\} \subseteq \text{Chains } r$
using *assms*
by (*auto simp add: Chains-def pred-on.chain-def refl-on-def*)

lemma *Chains-alt-def*:
assumes *refl r*
shows $\text{Chains } r = \{C. \text{pred-on.chain UNIV } (\lambda x y. (x, y) \in r) \ C\}$
using *assms Chains-subset Chains-subset'* **by** *blast*

lemma *Chains-relation-of*:
assumes $C \in \text{Chains } (\text{relation-of } P \ A)$ **shows** $C \subseteq A$
using *assms* **unfolding** *Chains-def relation-of-def* **by** *auto*

lemma *pairwise-chain-Union*:
assumes $P: \bigwedge S. S \in \mathcal{C} \implies \text{pairwise } R \ S$ **and** $\text{chain}_{\subseteq} \mathcal{C}$
shows $\text{pairwise } R \ (\bigcup \mathcal{C})$
using $\langle \text{chain}_{\subseteq} \mathcal{C} \rangle$ **unfolding** *pairwise-def chain-subset-def*
by (*blast intro: P [unfolded pairwise-def, rule-format]*)

lemma *Zorn-Lemma*: $\forall C \in \text{chains } A. \bigcup C \in A \implies \exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$
using *subset-Zorn' [of A]* **by** (*force simp: chains-alt-def*)

lemma *Zorn-Lemma2*: $\forall C \in \text{chains } A. \exists U \in A. \forall X \in C. X \subseteq U \implies \exists M \in A. \forall X \in A. M \subseteq X \longrightarrow X = M$
using *subset-Zorn [of A]* **by** (*auto simp: chains-alt-def*)

26.3 Other variants of Zorn’s Lemma

lemma *chainsD*: $c \in \text{chains } S \implies x \in c \implies y \in c \implies x \subseteq y \vee y \subseteq x$
unfolding *chains-def chain-subset-def* **by** *blast*

lemma *chainsD2*: $c \in \text{chains } S \implies c \subseteq S$
unfolding *chains-def* **by** *blast*

lemma *Zorns-po-lemma*:
assumes *po: Partial-order r*
and $u: \bigwedge C. C \in \text{Chains } r \implies \exists u \in \text{Field } r. \forall a \in C. (a, u) \in r$

shows $\exists m \in \text{Field } r. \forall a \in \text{Field } r. (m, a) \in r \longrightarrow a = m$
proof –
have *Preorder* r
using *po* **by** (*simp add: partial-order-on-def*)
 Mirror r in the set of subsets below (wrt r) elements of A .
let $?B = \lambda x. r^{-1} \text{ `` } \{x\}$
let $?S = ?B \text{ ‘ } \text{Field } r$
have $\exists u \in \text{Field } r. \forall A \in C. A \subseteq r^{-1} \text{ `` } \{u\}$ (**is** $\exists u \in \text{Field } r. ?P u$)
if 1: $C \subseteq ?S$ **and** 2: $\forall A \in C. \forall B \in C. A \subseteq B \vee B \subseteq A$ **for** C
proof –
let $?A = \{x \in \text{Field } r. \exists M \in C. M = ?B x\}$
from 1 **have** $C = ?B \text{ ‘ } ?A$ **by** (*auto simp: image-def*)
have $?A \in \text{Chains } r$
proof (*simp add: Chains-def, intro allI impI, elim conjE*)
fix $a b$
assume $a \in \text{Field } r$ **and** $?B a \in C$ **and** $b \in \text{Field } r$ **and** $?B b \in C$
with 2 **have** $?B a \subseteq ?B b \vee ?B b \subseteq ?B a$ **by** *auto*
then show $(a, b) \in r \vee (b, a) \in r$
using $\langle \text{Preorder } r \rangle$ **and** $\langle a \in \text{Field } r \rangle$ **and** $\langle b \in \text{Field } r \rangle$
by (*simp add: subset-Image1-Image1-iff*)
qed
then obtain u **where** $uA: u \in \text{Field } r \forall a \in ?A. (a, u) \in r$
by (*auto simp: dest: u*)
have $?P u$
proof *auto*
fix $a B$ **assume** $aB: B \in C a \in B$
with 1 **obtain** x **where** $x \in \text{Field } r$ **and** $B = r^{-1} \text{ `` } \{x\}$ **by** *auto*
then show $(a, u) \in r$
using uA **and** aB **and** $\langle \text{Preorder } r \rangle$
unfolding *preorder-on-def refl-on-def* **by** *simp (fast dest: transD)*
qed
then show *?thesis*
using $\langle u \in \text{Field } r \rangle$ **by** *blast*
qed
then have $\forall C \in \text{chains } ?S. \exists U \in ?S. \forall A \in C. A \subseteq U$
by (*auto simp: chains-def chain-subset-def*)
from *Zorn-Lemma2 [OF this]* **obtain** $m B$
where $m \in \text{Field } r$
and $B = r^{-1} \text{ `` } \{m\}$
and $\forall x \in \text{Field } r. B \subseteq r^{-1} \text{ `` } \{x\} \longrightarrow r^{-1} \text{ `` } \{x\} = B$
by *auto*
then have $\forall a \in \text{Field } r. (m, a) \in r \longrightarrow a = m$
using *po* **and** $\langle \text{Preorder } r \rangle$ **and** $\langle m \in \text{Field } r \rangle$
by (*auto simp: subset-Image1-Image1-iff Partial-order-eq-Image1-Image1-iff*)
then show *?thesis*
using $\langle m \in \text{Field } r \rangle$ **by** *blast*
qed

lemma *predicate-Zorn*:

assumes *po*: *partial-order-on* A (*relation-of* $P A$)

and *ch*: $\bigwedge C. C \in \text{Chains} \text{ (relation-of } P A) \implies \exists u \in A. \forall a \in C. P a u$

shows $\exists m \in A. \forall a \in A. P m a \longrightarrow a = m$

proof –

have $a \in A$ **if** $C \in \text{Chains} \text{ (relation-of } P A)$ **and** $a \in C$ **for** $C a$

using *that unfolding Chains-def relation-of-def by auto*

moreover have $(a, u) \in \text{relation-of } P A$ **if** $a \in A$ **and** $u \in A$ **and** $P a u$ **for** $a u$

unfolding *relation-of-def using that by auto*

ultimately have $\exists m \in A. \forall a \in A. (m, a) \in \text{relation-of } P A \longrightarrow a = m$

using *Zorns-po-lemma[OF Partial-order-relation-ofI[OF po], rule-format] ch*

unfolding *Field-relation-of[OF partial-order-onD(1)[OF po]] by blast*

then show *?thesis*

by (*auto simp: relation-of-def*)

qed

lemma *Union-in-chain*: $\llbracket \text{finite } \mathcal{B}; \mathcal{B} \neq \{\} \rrbracket; \text{subset.chain } \mathcal{A} \mathcal{B} \rrbracket \implies \bigcup \mathcal{B} \in \mathcal{B}$

proof (*induction* \mathcal{B} *rule: finite-induct*)

case (*insert* $B \mathcal{B}$)

show *?case*

proof (*cases* $\mathcal{B} = \{\}$)

case *False*

then show *?thesis*

using *insert sup.absorb2 by (auto simp: subset-chain-insert dest!: bspec [where*
 $x = \bigcup \mathcal{B}])$

qed *auto*

qed *simp*

lemma *Inter-in-chain*: $\llbracket \text{finite } \mathcal{B}; \mathcal{B} \neq \{\} \rrbracket; \text{subset.chain } \mathcal{A} \mathcal{B} \rrbracket \implies \bigcap \mathcal{B} \in \mathcal{B}$

proof (*induction* \mathcal{B} *rule: finite-induct*)

case (*insert* $B \mathcal{B}$)

show *?case*

proof (*cases* $\mathcal{B} = \{\}$)

case *False*

then show *?thesis*

using *insert inf.absorb2 by (auto simp: subset-chain-insert dest!: bspec [where*
 $x = \bigcap \mathcal{B}])$

qed *auto*

qed *simp*

lemma *finite-subset-Union-chain*:

assumes *finite* $A A \subseteq \bigcup \mathcal{B} \mathcal{B} \neq \{\}$ **and** *sub*: *subset.chain* $\mathcal{A} \mathcal{B}$

obtains B **where** $B \in \mathcal{B} A \subseteq B$

proof –

obtain \mathcal{F} **where** \mathcal{F} : *finite* $\mathcal{F} \mathcal{F} \subseteq \mathcal{B} A \subseteq \bigcup \mathcal{F}$

using *assms by (auto intro: finite-subset-Union)*

show *thesis*

proof (*cases* $\mathcal{F} = \{\}$)

case *True*

```

then show ?thesis
  using  $\langle A \subseteq \bigcup \mathcal{F} \rangle \langle \mathcal{B} \neq \{\} \rangle$  that by fastforce
next
  case False
  show ?thesis
  proof
    show  $\bigcup \mathcal{F} \in \mathcal{B}$ 
      using sub  $\langle \mathcal{F} \subseteq \mathcal{B} \rangle \langle \text{finite } \mathcal{F} \rangle$ 
      by (simp add: Union-in-chain False subset.chain-def subset-iff)
    show  $A \subseteq \bigcup \mathcal{F}$ 
      using  $\langle A \subseteq \bigcup \mathcal{F} \rangle$  by blast
  qed
qed
qed

```

```

lemma subset-Zorn-nonempty:
  assumes  $\mathcal{A} \neq \{\}$  and ch:  $\bigwedge \mathcal{C}. [\mathcal{C} \neq \{\}; \text{subset.chain } \mathcal{A} \mathcal{C}] \implies \bigcup \mathcal{C} \in \mathcal{A}$ 
  shows  $\exists M \in \mathcal{A}. \forall X \in \mathcal{A}. M \subseteq X \longrightarrow X = M$ 
proof (rule subset-Zorn)
  show  $\exists U \in \mathcal{A}. \forall X \in \mathcal{C}. X \subseteq U$  if subset.chain  $\mathcal{A} \mathcal{C}$  for  $\mathcal{C}$ 
  proof (cases  $\mathcal{C} = \{\}$ )
    case True
    then show ?thesis
      using  $\langle \mathcal{A} \neq \{\} \rangle$  by blast
  next
    case False
    show ?thesis
      by (blast intro!: ch False that Union-upper)
  qed
qed

```

26.4 The Well Ordering Theorem

definition *init-seg-of* :: $((\text{'a} \times \text{'a}) \text{ set} \times (\text{'a} \times \text{'a}) \text{ set}) \text{ set}$
where *init-seg-of* = $\{(r, s). r \subseteq s \wedge (\forall a b c. (a, b) \in s \wedge (b, c) \in r \longrightarrow (a, b) \in r)\}$

abbreviation *initial-segment-of-syntax* :: $(\text{'a} \times \text{'a}) \text{ set} \Rightarrow (\text{'a} \times \text{'a}) \text{ set} \Rightarrow \text{bool}$
 (infix *initial'-segment'-of* 55)
where *r initial-segment-of s* $\equiv (r, s) \in \text{init-seg-of}$

lemma *refl-on-init-seg-of* [simp]: *r initial-segment-of r*
by (simp add: *init-seg-of-def*)

lemma *trans-init-seg-of*:
r initial-segment-of s \implies *s initial-segment-of t* \implies *r initial-segment-of t*
by (simp (no-asm-use) add: *init-seg-of-def*) blast

lemma *antisym-init-seg-of*: *r initial-segment-of s* \implies *s initial-segment-of r* \implies *r*

= s
unfolding *init-seg-of-def* **by** *safe*

lemma *Chains-init-seg-of-Union*: $R \in \text{Chains } \textit{init-seg-of} \implies r \in R \implies r \textit{ initial-segment-of} \bigcup R$
by (*auto simp: init-seg-of-def Ball-def Chains-def*) *blast*

lemma *chain-subset-trans-Union*:
assumes $\textit{chain}_{\subseteq} R \forall r \in R. \textit{trans } r$
shows $\textit{trans} (\bigcup R)$
proof (*intro transI, elim UnionE*)
fix $S1 S2 :: 'a \textit{ rel}$ **and** $x y z :: 'a$
assume $S1 \in R S2 \in R$
with $\textit{assms}(1)$ **have** $S1 \subseteq S2 \vee S2 \subseteq S1$
unfolding *chain-subset-def* **by** *blast*
moreover assume $(x, y) \in S1 (y, z) \in S2$
ultimately have $((x, y) \in S1 \wedge (y, z) \in S1) \vee ((x, y) \in S2 \wedge (y, z) \in S2)$
by *blast*
with $\langle S1 \in R \rangle \langle S2 \in R \rangle \textit{assms}(2)$ **show** $(x, z) \in \bigcup R$
by (*auto elim: transE*)
qed

lemma *chain-subset-antisym-Union*:
assumes $\textit{chain}_{\subseteq} R \forall r \in R. \textit{antisym } r$
shows $\textit{antisym} (\bigcup R)$
proof (*intro antisymI, elim UnionE*)
fix $S1 S2 :: 'a \textit{ rel}$ **and** $x y :: 'a$
assume $S1 \in R S2 \in R$
with $\textit{assms}(1)$ **have** $S1 \subseteq S2 \vee S2 \subseteq S1$
unfolding *chain-subset-def* **by** *blast*
moreover assume $(x, y) \in S1 (y, x) \in S2$
ultimately have $((x, y) \in S1 \wedge (y, x) \in S1) \vee ((x, y) \in S2 \wedge (y, x) \in S2)$
by *blast*
with $\langle S1 \in R \rangle \langle S2 \in R \rangle \textit{assms}(2)$ **show** $x = y$
unfolding *antisym-def* **by** *auto*
qed

lemma *chain-subset-Total-Union*:
assumes $\textit{chain}_{\subseteq} R$ **and** $\forall r \in R. \textit{Total } r$
shows $\textit{Total} (\bigcup R)$
proof (*simp add: total-on-def Ball-def, auto del: disjCI*)
fix $r s a b$
assume $A: r \in R s \in R a \in \textit{Field } r b \in \textit{Field } s a \neq b$
from $\langle \textit{chain}_{\subseteq} R \rangle$ **and** $\langle r \in R \rangle$ **and** $\langle s \in R \rangle$ **have** $r \subseteq s \vee s \subseteq r$
by (*auto simp add: chain-subset-def*)
then show $(\exists r \in R. (a, b) \in r) \vee (\exists r \in R. (b, a) \in r)$
proof
assume $r \subseteq s$
then have $(a, b) \in s \vee (b, a) \in s$

```

    using assms(2) A mono-Field[of r s]
    by (auto simp add: total-on-def)
  then show ?thesis
    using  $\langle s \in R \rangle$  by blast
next
  assume  $s \subseteq r$ 
  then have  $(a, b) \in r \vee (b, a) \in r$ 
    using assms(2) A mono-Field[of s r]
    by (fastforce simp add: total-on-def)
  then show ?thesis
    using  $\langle r \in R \rangle$  by blast
qed
qed

lemma wf-Union-wf-init-segs:
  assumes  $R \in \text{Chains init-seg-of}$ 
    and  $\forall r \in R. \text{wf } r$ 
  shows  $\text{wf } (\bigcup R)$ 
proof (simp add: wf-iff-no-infinite-down-chain, rule ccontr, auto)
  fix f
  assume 1:  $\forall i. \exists r \in R. (f (\text{Suc } i), f i) \in r$ 
  then obtain r where  $r \in R$  and  $(f (\text{Suc } 0), f 0) \in r$  by auto
  have  $(f (\text{Suc } i), f i) \in r$  for i
  proof (induct i)
    case 0
    show ?case by fact
  next
    case (Suc i)
    then obtain s where  $s: s \in R (f (\text{Suc } (\text{Suc } i)), f (\text{Suc } i)) \in s$ 
      using 1 by auto
    then have s initial-segment-of  $r \vee r$  initial-segment-of s
      using assms(1)  $\langle r \in R \rangle$  by (simp add: Chains-def)
    with Suc s show ?case by (simp add: init-seg-of-def) blast
  qed
  then show False
    using assms(2) and  $\langle r \in R \rangle$ 
    by (simp add: wf-iff-no-infinite-down-chain) blast
qed

```

```

lemma initial-segment-of-Diff:  $p$  initial-segment-of  $q \implies p - s$  initial-segment-of
 $q - s$ 
  unfolding init-seg-of-def by blast

```

```

lemma Chains-inits-DiffI:  $R \in \text{Chains init-seg-of} \implies \{r - s \mid r. r \in R\} \in \text{Chains}$ 
init-seg-of
  unfolding Chains-def by (blast intro: initial-segment-of-Diff)

```

```

theorem well-ordering:  $\exists r::'a \text{ rel. Well-order } r \wedge \text{Field } r = \text{UNIV}$ 
proof -

```

— The initial segment relation on well-orders:
let $?WO = \{r::'a \text{ rel. Well-order } r\}$
define I **where** $I = \text{init-seg-of} \cap ?WO \times ?WO$
then have $I\text{-init}: I \subseteq \text{init-seg-of}$ **by** *simp*
then have $\text{subch}: \bigwedge R. R \in \text{Chains } I \implies \text{chain}_{\subseteq} R$
unfolding *init-seg-of-def chain-subset-def Chains-def* **by** *blast*
have $\text{Chains-wo}: \bigwedge R r. R \in \text{Chains } I \implies r \in R \implies \text{Well-order } r$
by (*simp add: Chains-def I-def*) *blast*
have $FI: \text{Field } I = ?WO$
by (*auto simp add: I-def init-seg-of-def Field-def*)
then have $0: \text{Partial-order } I$
by (*auto simp: partial-order-on-def preorder-on-def antisym-def antisym-init-seg-of refl-on-def trans-def I-def elim!: trans-init-seg-of*)

— I -chains have upper bounds in $?WO$ wrt I : their Union
have $\bigcup R \in ?WO \wedge (\forall r \in R. (r, \bigcup R) \in I)$ **if** $R \in \text{Chains } I$ **for** R
proof —
from *that* **have** $Ris: R \in \text{Chains init-seg-of}$
using *mono-Chains [OF I-init]* **by** *blast*
have $\text{subch}: \text{chain}_{\subseteq} R$
using $\langle R \in \text{Chains } I \rangle I\text{-init}$ **by** (*auto simp: init-seg-of-def chain-subset-def Chains-def*)
have $\forall r \in R. \text{Refl } r$ **and** $\forall r \in R. \text{trans } r$ **and** $\forall r \in R. \text{antisym } r$
and $\forall r \in R. \text{Total } r$ **and** $\forall r \in R. \text{wf } (r - Id)$
using Chains-wo [*OF* $\langle R \in \text{Chains } I \rangle$] **by** (*simp-all add: order-on-defs*)
have $\text{Refl } (\bigcup R)$
using $\langle \forall r \in R. \text{Refl } r \rangle$ **unfolding** *refl-on-def* **by** *fastforce*
moreover have $\text{trans } (\bigcup R)$
by (*rule chain-subset-trans-Union [OF subch* $\langle \forall r \in R. \text{trans } r \rangle$])
moreover have $\text{antisym } (\bigcup R)$
by (*rule chain-subset-antisym-Union [OF subch* $\langle \forall r \in R. \text{antisym } r \rangle$])
moreover have $\text{Total } (\bigcup R)$
by (*rule chain-subset-Total-Union [OF subch* $\langle \forall r \in R. \text{Total } r \rangle$])
moreover have $\text{wf } ((\bigcup R) - Id)$
proof —
have $(\bigcup R) - Id = \bigcup \{r - Id \mid r. r \in R\}$ **by** *blast*
with $\langle \forall r \in R. \text{wf } (r - Id) \rangle$ **and** *wf-Union-wf-init-segs* [*OF Chains-inits-DiffI* [*OF Ris*]]
show *?thesis* **by** *fastforce*
qed
ultimately have $\text{Well-order } (\bigcup R)$
by (*simp add: order-on-defs*)
moreover have $\forall r \in R. r \text{ initial-segment-of } \bigcup R$
using Ris **by** (*simp add: Chains-init-seg-of-Union*)
ultimately show *?thesis*
using *mono-Chains [OF I-init] Chains-wo*[*of* R] **and** $\langle R \in \text{Chains } I \rangle$
unfolding *I-def* **by** *blast*
qed
then have $1: \exists u \in \text{Field } I. \forall r \in R. (r, u) \in I$ **if** $R \in \text{Chains } I$ **for** R

using *that by* (*subst FI*) *blast*
 — Zorn’s Lemma yields a maximal well-order m :
then obtain $m :: 'a \text{ rel}$
where *Well-order* m
and $max: \forall r. \text{Well-order } r \wedge (m, r) \in I \longrightarrow r = m$
using *Zorns-po-lemma*[*OF 0 1*] **unfolding** *FI* **by** *fastforce*
 — Now show by contradiction that m covers the whole type:
have *False* **if** $x \notin \text{Field } m$ **for** $x :: 'a$
proof —
 — Assuming that x is not covered and extend m at the top with x
have $m \neq \{\}$
proof
assume $m = \{\}$
moreover have *Well-order* $\{(x, x)\}$
by (*simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def*
Field-def)
ultimately show *False* **using** *max*
by (*auto simp: I-def init-seg-of-def simp del: Field-insert*)
qed
then have $\text{Field } m \neq \{\}$ **by** (*auto simp: Field-def*)
moreover have *wf* $(m - \text{Id})$
using $\langle \text{Well-order } m \rangle$ **by** (*simp add: well-order-on-def*)
 — The extension of m by x :
let $?s = \{(a, x) \mid a. a \in \text{Field } m\}$
let $?m = \text{insert } (x, x) m \cup ?s$
have $Fm: \text{Field } ?m = \text{insert } x (\text{Field } m)$
by (*auto simp: Field-def*)
have *Refl* m **and** *trans* m **and** *antisym* m **and** *Total* m **and** *wf* $(m - \text{Id})$
using $\langle \text{Well-order } m \rangle$ **by** (*simp-all add: order-on-defs*)
 — We show that the extension is a well-order
have *Refl* $?m$
using $\langle \text{Refl } m \rangle$ Fm **unfolding** *refl-on-def* **by** *blast*
moreover have *trans* $?m$ **using** $\langle \text{trans } m \rangle$ **and** $\langle x \notin \text{Field } m \rangle$
unfolding *trans-def* *Field-def* **by** *blast*
moreover have *antisym* $?m$
using $\langle \text{antisym } m \rangle$ **and** $\langle x \notin \text{Field } m \rangle$ **unfolding** *antisym-def* *Field-def* **by**
blast
moreover have *Total* $?m$
using $\langle \text{Total } m \rangle$ **and** Fm **by** (*auto simp: total-on-def*)
moreover have *wf* $(?m - \text{Id})$
proof —
have *wf* $?s$
using $\langle x \notin \text{Field } m \rangle$ **by** (*auto simp: wf-eq-minimal Field-def Bex-def*)
then show *?thesis*
using $\langle \text{wf } (m - \text{Id}) \rangle$ **and** $\langle x \notin \text{Field } m \rangle$ *wf-subset* [*OF* $\langle \text{wf } ?s \rangle$ *Diff-subset*]
by (*auto simp: Un-Diff Field-def intro: wf-Un*)
qed
ultimately have *Well-order* $?m$
by (*simp add: order-on-defs*)

— We show that the extension is above m
moreover have $(m, ?m) \in I$
using $\langle \text{Well-order } ?m \rangle$ **and** $\langle \text{Well-order } m \rangle$ **and** $\langle x \notin \text{Field } m \rangle$
by (*fastforce simp: I-def init-seg-of-def Field-def*)
ultimately
— This contradicts maximality of m :
show *False*
using *max* **and** $\langle x \notin \text{Field } m \rangle$ **unfolding** *Field-def* **by** *blast*
qed
then have $\text{Field } m = \text{UNIV}$ **by** *auto*
with $\langle \text{Well-order } m \rangle$ **show** *?thesis* **by** *blast*
qed

corollary *well-order-on*: $\exists r :: 'a \text{ rel. well-order-on } A \ r$

proof —

obtain $r :: 'a \text{ rel}$ **where** *wo*: *Well-order* r **and** *univ*: $\text{Field } r = \text{UNIV}$
using *well-ordering* [**where** $'a = 'a$] **by** *blast*
let $?r = \{(x, y). x \in A \wedge y \in A \wedge (x, y) \in r\}$
have $1: \text{Field } ?r = A$
using *wo univ* **by** (*fastforce simp: Field-def order-on-defs refl-on-def*)
from $\langle \text{Well-order } r \rangle$ **have** *Refl* r *trans* r *antisym* r *Total* r *wf* $(r - \text{Id})$
by (*simp-all add: order-on-defs*)
from $\langle \text{Refl } r \rangle$ **have** *Refl* $?r$
by (*auto simp: refl-on-def 1 univ*)
moreover from $\langle \text{trans } r \rangle$ **have** *trans* $?r$
unfolding *trans-def* **by** *blast*
moreover from $\langle \text{antisym } r \rangle$ **have** *antisym* $?r$
unfolding *antisym-def* **by** *blast*
moreover from $\langle \text{Total } r \rangle$ **have** *Total* $?r$
by (*simp add: total-on-def 1 univ*)
moreover have *wf* $(?r - \text{Id})$
by (*rule wf-subset [OF $\langle \text{wf } (r - \text{Id}) \rangle]$)* *blast*
ultimately have *Well-order* $?r$
by (*simp add: order-on-defs*)
with 1 **show** *?thesis* **by** *auto*
qed

lemma *dependent-wf-choice*:

fixes $P :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$
assumes *wf* R
and *adm*: $\bigwedge f g x r. (\bigwedge z. (z, x) \in R \Longrightarrow f z = g z) \Longrightarrow P f x r = P g x r$
and P : $\bigwedge x f. (\bigwedge y. (y, x) \in R \Longrightarrow P f y (f y)) \Longrightarrow \exists r. P f x r$
shows $\exists f. \forall x. P f x (f x)$
proof (*intro exI allI*)
fix x
define f **where** $f \equiv \text{wfrec } R (\lambda f x. \text{SOME } r. P f x r)$
from $\langle \text{wf } R \rangle$ **show** $P f x (f x)$
proof (*induct* x)
case (*less* x)

```

show P f x (f x)
proof (subst (2) wfrec-def-adm[OF f-def ‹wf R›])
  show adm-wf R (λf x. SOME r. P f x r)
    by (auto simp: adm-wf-def intro!: arg-cong[where f=Eps] adm)
  show P f x (Eps (P f x))
    using P by (rule someI-ex) fact
qed
qed
qed

```

```

lemma (in wellorder) dependent-wellorder-choice:
  assumes  $\bigwedge r f g x. (\bigwedge y. y < x \implies f y = g y) \implies P f x r = P g x r$ 
  and P:  $\bigwedge x f. (\bigwedge y. y < x \implies P f y (f y)) \implies \exists r. P f x r$ 
  shows  $\exists f. \forall x. P f x (f x)$ 
  using wf by (rule dependent-wf-choice) (auto intro!: assms)

```

end

27 Well-Order Relations as Needed by Bounded Natural Functors

```

theory BNF-Wellorder-Relation
  imports Order-Relation
begin

```

In this section, we develop basic concepts and results pertaining to well-order relations. Note that we consider well-order relations as *non-strict relations*, i.e., as containing the diagonals of their fields.

```

locale wo-rel =
  fixes r :: 'a rel
  assumes WELL: Well-order r
begin

```

The following context encompasses all this section. In other words, for the whole section, we consider a fixed well-order relation r .

```

abbreviation under where under  $\equiv$  Order-Relation.under r
abbreviation underS where underS  $\equiv$  Order-Relation.underS r
abbreviation Under where Under  $\equiv$  Order-Relation.Under r
abbreviation UnderS where UnderS  $\equiv$  Order-Relation.UnderS r
abbreviation above where above  $\equiv$  Order-Relation.above r
abbreviation aboveS where aboveS  $\equiv$  Order-Relation.aboveS r
abbreviation Above where Above  $\equiv$  Order-Relation.Above r
abbreviation AboveS where AboveS  $\equiv$  Order-Relation.AboveS r
abbreviation ofilter where ofilter  $\equiv$  Order-Relation.ofilter r
lemmas ofilter-def = Order-Relation.ofilter-def[of r]

```

27.1 Auxiliaries

lemma *REFL: Refl r*
using *WELL order-on-defs[of - r]* **by** *auto*

lemma *TRANS: trans r*
using *WELL order-on-defs[of - r]* **by** *auto*

lemma *ANTISYM: antisym r*
using *WELL order-on-defs[of - r]* **by** *auto*

lemma *TOTAL: Total r*
using *WELL order-on-defs[of - r]* **by** *auto*

lemma *TOTALS: $\forall a \in \text{Field } r. \forall b \in \text{Field } r. (a,b) \in r \vee (b,a) \in r$*
using *REFL TOTAL refl-on-def[of - r] total-on-def[of - r]* **by** *force*

lemma *LIN: Linear-order r*
using *WELL well-order-on-def[of - r]* **by** *auto*

lemma *WF: wf (r - Id)*
using *WELL well-order-on-def[of - r]* **by** *auto*

lemma *cases-Total:*
 $\bigwedge \text{phi } a \text{ b. } [\{a,b\} \leq \text{Field } r; ((a,b) \in r \implies \text{phi } a \text{ b}); ((b,a) \in r \implies \text{phi } a \text{ b})]$
 $\implies \text{phi } a \text{ b}$
using *TOTALS* **by** *auto*

lemma *cases-Total3:*
 $\bigwedge \text{phi } a \text{ b. } [\{a,b\} \leq \text{Field } r; ((a,b) \in r - \text{Id} \vee (b,a) \in r - \text{Id} \implies \text{phi } a \text{ b});$
 $(a = b \implies \text{phi } a \text{ b})] \implies \text{phi } a \text{ b}$
using *TOTALS* **by** *auto*

27.2 Well-founded induction and recursion adapted to non-strict well-order relations

Here we provide induction and recursion principles specific to *non-strict* well-order relations. Although minor variations of those for well-founded relations, they will be useful for doing away with the tediousness of having to take out the diagonal each time in order to switch to a well-founded relation.

lemma *well-order-induct:*
assumes *IND: $\bigwedge x. \forall y. y \neq x \wedge (y, x) \in r \longrightarrow P y \implies P x$*
shows *P a*

proof –

have $\bigwedge x. \forall y. (y, x) \in r - \text{Id} \longrightarrow P y \implies P x$

using *IND* **by** *blast*

thus *P a* **using** *WF wf-induct[of r - Id P a]* **by** *blast*

qed

definition

$worec :: (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where
 $worec F \equiv wfrec (r - Id) F$

definition

$adm-wo :: (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow bool$
where
 $adm-wo H \equiv \forall f g x. (\forall y \in underS x. f y = g y) \longrightarrow H f x = H g x$

lemma *worec-fixpoint*:

assumes *ADM*: $adm-wo H$
shows $worec H = H (worec H)$

proof –

let $?rS = r - Id$
have $adm-wf (r - Id) H$
unfolding *adm-wf-def*
using *ADM adm-wo-def*[of *H*] *underS-def*[of *r*] **by** *auto*
hence $wfrec ?rS H = H (wfrec ?rS H)$
using *WF wfrec-fixpoint*[of *?rS H*] **by** *simp*
thus *?thesis* **unfolding** *worec-def* .

qed

27.3 The notions of maximum, minimum, supremum, successor and order filter

We define the successor of a set, and not of an element (the latter is of course a particular case). Also, we define the maximum of two elements, *max2*, and the minimum of a set, *minim* – we chose these variants since we consider them the most useful for well-orders. The minimum is defined in terms of the auxiliary relational operator *isMinim*. Then, supremum and successor are defined in terms of minimum as expected. The minimum is only meaningful for non-empty sets, and the successor is only meaningful for sets for which strict upper bounds exist. Order filters for well-orders are also known as “initial segments”.

definition *max2* :: $'a \Rightarrow 'a \Rightarrow 'a$
where $max2 a b \equiv if (a,b) \in r \text{ then } b \text{ else } a$

definition *isMinim* :: $'a \text{ set} \Rightarrow 'a \Rightarrow bool$
where $isMinim A b \equiv b \in A \wedge (\forall a \in A. (b,a) \in r)$

definition *minim* :: $'a \text{ set} \Rightarrow 'a$
where $minim A \equiv THE b. isMinim A b$

definition *supr* :: $'a \text{ set} \Rightarrow 'a$
where $supr A \equiv minim (Above A)$

definition $suc :: 'a \text{ set} \Rightarrow 'a$
where $suc A \equiv minim (AboveS A)$

27.3.1 Properties of max2

lemma *max2-greater-among*:

assumes $a \in Field\ r$ **and** $b \in Field\ r$

shows $(a, max2\ a\ b) \in r \wedge (b, max2\ a\ b) \in r \wedge max2\ a\ b \in \{a, b\}$

proof –

{**assume** $(a, b) \in r$

hence *?thesis* **using** *max2-def* *assms* *REFL* *refl-on-def*

by (*auto simp add: refl-on-def*)

}

moreover

{**assume** $a = b$

hence $(a, b) \in r$ **using** *REFL* *assms*

by (*auto simp add: refl-on-def*)

}

moreover

{**assume** $*$: $a \neq b \wedge (b, a) \in r$

hence $(a, b) \notin r$ **using** *ANTISYM*

by (*auto simp add: antisym-def*)

hence *?thesis* **using** $*$ *max2-def* *assms* *REFL* *refl-on-def*

by (*auto simp add: refl-on-def*)

}

ultimately show *?thesis* **using** *assms* *TOTAL*

total-on-def[of *Field* r] **by** *blast*

qed

lemma *max2-greater*:

assumes $a \in Field\ r$ **and** $b \in Field\ r$

shows $(a, max2\ a\ b) \in r \wedge (b, max2\ a\ b) \in r$

using *assms* **by** (*auto simp add: max2-greater-among*)

lemma *max2-among*:

assumes $a \in Field\ r$ **and** $b \in Field\ r$

shows $max2\ a\ b \in \{a, b\}$

using *assms* *max2-greater-among*[of $a\ b$] **by** *simp*

lemma *max2-equals1*:

assumes $a \in Field\ r$ **and** $b \in Field\ r$

shows $(max2\ a\ b = a) = ((b, a) \in r)$

using *assms* *ANTISYM* **unfolding** *antisym-def* **using** *TOTALS*

by(*auto simp add: max2-def max2-among*)

lemma *max2-equals2*:

assumes $a \in Field\ r$ **and** $b \in Field\ r$

shows $(max2\ a\ b = b) = ((a, b) \in r)$

using *assms* *ANTISYM* **unfolding** *antisym-def* **using** *TOTALS*

unfolding *max2-def* **by** *auto*

lemma *in-notinI*:

assumes $(j,i) \notin r \vee j = i$ **and** $i \in \text{Field } r$ **and** $j \in \text{Field } r$

shows $(i,j) \in r$ **using** *assms max2-def max2-greater-among* **by** *fastforce*

27.3.2 Existence and uniqueness for *isMinim* and well-definedness of *minim*

lemma *isMinim-unique*:

assumes *isMinim* B a *isMinim* B a'

shows $a = a'$

using *assms ANTISYM antisym-def*[*of* r] **by** (*auto simp: isMinim-def*)

lemma *Well-order-isMinim-exists*:

assumes *SUB*: $B \leq \text{Field } r$ **and** *NE*: $B \neq \{\}$

shows $\exists b. \text{isMinim } B b$

proof –

from *spec*[*OF WF*[*unfolded wf-eq-minimal*[*of* $r - \text{Id}$], *of* B] *NE* **obtain** b **where**

$*$: $b \in B \wedge (\forall b'. b' \neq b \wedge (b',b) \in r \longrightarrow b' \notin B)$ **by** *auto*

have $\forall b'. b' \in B \longrightarrow (b, b') \in r$

proof

fix b'

show $b' \in B \longrightarrow (b, b') \in r$

proof

assume $As: b' \in B$

hence $**$: $b \in \text{Field } r \wedge b' \in \text{Field } r$ **using** As *SUB* $*$ **by** *auto*

from As $*$ **have** $b' = b \vee (b',b) \notin r$ **by** *auto*

moreover **have** $b' = b \implies (b, b') \in r$

using $**$ *REFL* **by** (*auto simp add: refl-on-def*)

moreover **have** $b' \neq b \wedge (b',b) \notin r \implies (b,b') \in r$

using $**$ *TOTAL* **by** (*auto simp add: total-on-def*)

ultimately **show** $(b,b') \in r$ **by** *blast*

qed

qed

then **show** *?thesis*

unfolding *isMinim-def* **using** $*$ **by** *auto*

qed

lemma *minim-isMinim*:

assumes *SUB*: $B \leq \text{Field } r$ **and** *NE*: $B \neq \{\}$

shows *isMinim* B (*minim* B)

proof –

let $?phi = (\lambda b. \text{isMinim } B b)$

from *assms Well-order-isMinim-exists*

obtain b **where** $*$: $?phi b$ **by** *blast*

moreover

have $\bigwedge b'. ?phi b' \implies b' = b$

using *isMinim-unique* $*$ **by** *auto*

ultimately show *?thesis*
 unfolding *minim-def* using *theI[of ?phi b]* by *blast*
 qed

27.3.3 Properties of minim

lemma *minim-in*:
 assumes $B \leq \text{Field } r$ and $B \neq \{\}$
 shows $\text{minim } B \in B$
 using *assms minim-isMinim[of B]* by (*auto simp: isMinim-def*)

lemma *minim-inField*:
 assumes $B \leq \text{Field } r$ and $B \neq \{\}$
 shows $\text{minim } B \in \text{Field } r$
 proof –
 have $\text{minim } B \in B$ using *assms* by (*simp add: minim-in*)
 thus *?thesis* using *assms* by *blast*
 qed

lemma *minim-least*:
 assumes *SUB*: $B \leq \text{Field } r$ and *IN*: $b \in B$
 shows $(\text{minim } B, b) \in r$
 proof –
 from *minim-isMinim[of B]* *assms*
 have *isMinim B (minim B)* by *auto*
 thus *?thesis* by (*auto simp add: isMinim-def IN*)
 qed

lemma *equals-minim*:
 assumes *SUB*: $B \leq \text{Field } r$ and *IN*: $a \in B$ and
LEAST: $\bigwedge b. b \in B \implies (a, b) \in r$
 shows $a = \text{minim } B$
 proof –
 from *minim-isMinim[of B]* *assms*
 have *isMinim B (minim B)* by *auto*
 moreover have *isMinim B a* using *IN LEAST isMinim-def* by *auto*
 ultimately show *?thesis*
 using *isMinim-unique* by *auto*
 qed

27.3.4 Properties of successor

lemma *suc-AboveS*:
 assumes *SUB*: $B \leq \text{Field } r$ and *ABOVES*: $\text{AboveS } B \neq \{\}$
 shows $\text{suc } B \in \text{AboveS } B$
 proof(*unfold suc-def*)
 have $\text{AboveS } B \leq \text{Field } r$
 using *AboveS-Field[of r]* by *auto*
 thus $\text{minim } (\text{AboveS } B) \in \text{AboveS } B$
 using *assms* by (*simp add: minim-in*)

qed

lemma *suc-greater*:

assumes *SUB*: $B \leq \text{Field } r$ **and** *ABOVES*: $\text{AboveS } B \neq \{\}$ **and** *IN*: $b \in B$
shows $\text{suc } B \neq b \wedge (b, \text{suc } B) \in r$
using *IN AboveS-def*[of *r*] *assms suc-AboveS* **by** *auto*

lemma *suc-least-AboveS*:

assumes *ABOVES*: $a \in \text{AboveS } B$
shows $(\text{suc } B, a) \in r$
using *assms minim-least AboveS-Field*[of *r*] **by** (*auto simp: suc-def*)

lemma *suc-inField*:

assumes $B \leq \text{Field } r$ **and** $\text{AboveS } B \neq \{\}$
shows $\text{suc } B \in \text{Field } r$
using *suc-AboveS assms AboveS-Field*[of *r*] **by** *auto*

lemma *equals-suc-AboveS*:

assumes $B \leq \text{Field } r$ **and** $a \in \text{AboveS } B$ **and** $\bigwedge a'. a' \in \text{AboveS } B \implies (a, a') \in r$
shows $a = \text{suc } B$
using *assms equals-minim AboveS-Field*[of *r B*] **by** (*auto simp: suc-def*)

lemma *suc-underS*:

assumes *IN*: $a \in \text{Field } r$
shows $a = \text{suc } (\text{underS } a)$

proof–

have $\text{underS } a \leq \text{Field } r$
using *underS-Field*[of *r*] **by** *auto*

moreover

have $a \in \text{AboveS } (\text{underS } a)$
using *in-AboveS-underS IN* **by** *fast*

moreover

have $\forall a' \in \text{AboveS } (\text{underS } a). (a, a') \in r$

proof(*clarify*)

fix a'

assume *: $a' \in \text{AboveS } (\text{underS } a)$

hence **: $a' \in \text{Field } r$

using *AboveS-Field* **by** *fast*

{**assume** $(a, a') \notin r$

hence $a' = a \vee (a', a) \in r$

using *TOTAL IN *** **by** (*auto simp add: total-on-def*)

moreover

{**assume** $a' = a$

hence $(a, a') \in r$

using *REFL IN *** **by** (*auto simp add: refl-on-def*)

}

moreover

{**assume** $a' \neq a \wedge (a', a) \in r$

```

    hence  $a' \in \text{underS } a$ 
      unfolding underS-def by simp
    hence  $a' \notin \text{AboveS } (\text{underS } a)$ 
      using AboveS-disjoint by fast
    with * have False by simp
  }
  ultimately have  $(a, a') \in r$  by blast
}
thus  $(a, a') \in r$  by blast
qed
ultimately show ?thesis
  using equals-suc-AboveS by auto
qed

```

27.3.5 Properties of order filters

```

lemma under-ofilter: ofilter (under  $a$ )
  using TRANS by (auto simp: ofilter-def under-def Field-iff trans-def)

```

```

lemma underS-ofilter: ofilter (underS  $a$ )
  unfolding ofilter-def underS-def under-def

```

```

proof safe
  fix  $b$  assume  $(a, b) \in r$   $(b, a) \in r$  and DIFF:  $b \neq a$ 
  thus False
    using ANTISYM antisym-def[of r] by blast

```

```

next
  fix  $b$   $x$ 
  assume  $(b, a) \in r$   $b \neq a$   $(x, b) \in r$ 
  thus  $(x, a) \in r$ 
    using TRANS trans-def[of r] by blast

```

```

next
  fix  $x$ 
  assume  $x \neq a$  and  $(x, a) \in r$ 
  then show  $x \in \text{Field } r$ 
    unfolding Field-def
    by auto

```

qed

```

lemma Field-ofilter:
  ofilter (Field  $r$ )
  by(unfold ofilter-def under-def, auto simp add: Field-def)

```

```

lemma ofilter-underS-Field:
  ofilter  $A = ((\exists a \in \text{Field } r. A = \text{underS } a) \vee (A = \text{Field } r))$ 

```

```

proof
  assume  $(\exists a \in \text{Field } r. A = \text{underS } a) \vee A = \text{Field } r$ 
  thus ofilter  $A$ 
    by (auto simp: underS-ofilter Field-ofilter)

```

next

```

assume *: ofilter A
let ?One = ( $\exists a \in \text{Field } r. A = \text{underS } a$ )
let ?Two = ( $A = \text{Field } r$ )
show ?One  $\vee$  ?Two
proof(cases ?Two)
  let ?B = ( $\text{Field } r$ ) - A
  let ?a = minim ?B
  assume  $A \neq \text{Field } r$ 
  moreover have  $A \leq \text{Field } r$  using * ofilter-def by simp
  ultimately have 1: ?B  $\neq \{\}$  by blast
  hence 2: ?a  $\in \text{Field } r$  using minim-inField[of ?B] by blast
  have 3: ?a  $\in ?B$  using minim-in[of ?B] 1 by blast
  hence 4: ?a  $\notin A$  by blast
  have 5:  $A \leq \text{Field } r$  using * ofilter-def by auto

moreover
have  $A = \text{underS } ?a$ 
proof
  show  $A \leq \text{underS } ?a$ 
  proof
    fix x assume **:  $x \in A$ 
    hence 11:  $x \in \text{Field } r$  using 5 by auto
    have 12:  $x \neq ?a$  using 4 ** by auto
    have 13:  $\text{under } x \leq A$  using * ofilter-def ** by auto
    {assume  $(x, ?a) \notin r$ 
      hence  $(?a, x) \in r$ 
        using TOTAL total-on-def[of Field r r]
        2 4 11 12 by auto
      hence ?a  $\in \text{under } x$  using under-def[of r] by auto
      hence ?a  $\in A$  using ** 13 by blast
      with 4 have False by simp
    }
    then have  $(x, ?a) \in r$  by blast
    thus  $x \in \text{underS } ?a$ 
    unfolding underS-def by (auto simp add: 12)
  qed
next
show  $\text{underS } ?a \leq A$ 
proof
  fix x
  assume **:  $x \in \text{underS } ?a$ 
  hence 11:  $x \in \text{Field } r$ 
    using Field-def unfolding underS-def by fastforce
  {assume  $x \notin A$ 
    hence  $x \in ?B$  using 11 by auto
    hence  $(?a, x) \in r$  using 3 minim-least[of ?B x] by blast
    hence False
    using ANTISYM antisym-def[of r] ** unfolding underS-def by auto
  }

```

```

    thus  $x \in A$  by blast
  qed
qed
ultimately have ?One using 2 by blast
thus ?thesis by simp
next
  assume  $A = \text{Field } r$ 
  then show ?thesis
    by simp
  qed
qed

```

lemma *ofilter-UNION*:

```

( $\bigwedge i. i \in I \implies \text{ofilter}(A \ i)$ )  $\implies \text{ofilter}(\bigcup i \in I. A \ i)$ 
unfolding ofilter-def by blast

```

lemma *ofilter-under-UNION*:

```

assumes ofilter A
shows  $A = (\bigcup a \in A. \text{under } a)$ 
proof
  have  $\forall a \in A. \text{under } a \leq A$ 
    using assms ofilter-def by auto
  thus  $(\bigcup a \in A. \text{under } a) \leq A$  by blast
next
  have  $\forall a \in A. a \in \text{under } a$ 
    using REFL Refl-under-in[of r] assms ofilter-def[of A] by blast
  thus  $A \leq (\bigcup a \in A. \text{under } a)$  by blast
qed

```

27.3.6 Other properties

lemma *ofilter-linord*:

```

assumes OF1: ofilter A and OF2: ofilter B
shows  $A \leq B \vee B \leq A$ 
proof(cases  $A = \text{Field } r$ )
  assume Case1:  $A = \text{Field } r$ 
  hence  $B \leq A$  using OF2 ofilter-def by auto
  thus ?thesis by simp
next
  assume Case2:  $A \neq \text{Field } r$ 
  with ofilter-underS-Field OF1 obtain a where
    1:  $a \in \text{Field } r \wedge A = \text{underS } a$  by auto
  show ?thesis
  proof(cases  $B = \text{Field } r$ )
    assume Case21:  $B = \text{Field } r$ 
    hence  $A \leq B$  using OF1 ofilter-def by auto
    thus ?thesis by simp
  next
    assume Case22:  $B \neq \text{Field } r$ 

```

```

with ofilter-underS-Field OF2 obtain b where
  2:  $b \in \text{Field } r \wedge B = \text{underS } b$  by auto
have  $a = b \vee (a,b) \in r \vee (b,a) \in r$ 
  using 1 2 TOTAL total-on-def[of - r] by auto
moreover
  {assume  $a = b$  with 1 2 have ?thesis by auto
  }
moreover
  {assume  $(a,b) \in r$ 
  with underS-incr[of r] TRANS ANTISYM 1 2
  have  $A \leq B$  by auto
  hence ?thesis by auto
  }
moreover
  {assume  $(b,a) \in r$ 
  with underS-incr[of r] TRANS ANTISYM 1 2
  have  $B \leq A$  by auto
  hence ?thesis by auto
  }
  }
ultimately show ?thesis by blast
qed
qed

lemma ofilter-AboveS-Field:
  assumes ofilter A
  shows  $A \cup (\text{AboveS } A) = \text{Field } r$ 
proof
  show  $A \cup (\text{AboveS } A) \leq \text{Field } r$ 
  using assms ofilter-def AboveS-Field[of r] by auto
next
  {fix x assume *:  $x \in \text{Field } r$  and **:  $x \notin A$ 
  {fix y assume **:  $y \in A$ 
  with ** have 1:  $y \neq x$  by auto
  {assume  $(y,x) \notin r$ 
  moreover
  have  $y \in \text{Field } r$  using assms ofilter-def *** by auto
  ultimately have  $(x,y) \in r$ 
  using 1 * TOTAL total-on-def[of - r] by auto
  with ** assms ofilter-def under-def[of r] have  $x \in A$  by auto
  with ** have False by contradiction
  }
  }
  hence  $(y,x) \in r$  by blast
  with 1 have  $y \neq x \wedge (y,x) \in r$  by auto
  }
  }
  with * have  $x \in \text{AboveS } A$  unfolding AboveS-def by auto
  }
  }
thus  $\text{Field } r \leq A \cup (\text{AboveS } A)$  by blast
qed

```

```

lemma suc-ofilter-in:
  assumes OF: ofilter A and ABOVE-NE: AboveS A ≠ {} and
    REL: (b, suc A) ∈ r and DIFF: b ≠ suc A
  shows b ∈ A
proof –
  have *: suc A ∈ Field r ∧ b ∈ Field r
    using WELL REL well-order-on-domain[of Field r] by auto
  {assume **: b ∉ A
    hence b ∈ AboveS A
      using OF * ofilter-AboveS-Field by auto
    hence (suc A, b) ∈ r
      using suc-least-AboveS by auto
    hence False using REL DIFF ANTISYM *
      by (auto simp add: antisym-def)
  }
  thus ?thesis by blast
qed

end

end

```

28 Well-Order Embeddings as Needed by Bounded Natural Functors

```

theory BNF-Wellorder-Embedding
  imports Hilbert-Choice BNF-Wellorder-Relation
begin

```

In this section, we introduce well-order *embeddings* and *isomorphisms* and prove their basic properties. The notion of embedding is considered from the point of view of the theory of ordinals, and therefore requires the source to be injected as an *initial segment* (i.e., *order filter*) of the target. A main result of this section is the existence of embeddings (in one direction or another) between any two well-orders, having as a consequence the fact that, given any two sets on any two types, one is smaller than (i.e., can be injected into) the other.

28.1 Auxiliaries

```

lemma UNION-inj-on-ofilter:
  assumes WELL: Well-order r and
    OF:  $\bigwedge i. i \in I \implies \text{wo-rel.ofilter } r (A \ i)$  and
    INJ:  $\bigwedge i. i \in I \implies \text{inj-on } f (A \ i)$ 
  shows inj-on f ( $\bigcup i \in I. A \ i$ )
proof –
  have wo-rel r using WELL by (simp add: wo-rel-def)

```

hence $\bigwedge i j. \llbracket i \in I; j \in I \rrbracket \implies A i \leq A j \vee A j \leq A i$
 using *wo-rel.ofilter-linord*[of *r*] *OF* by *blast*
 with *WELL INJ* show *?thesis*
 by (*auto simp add: inj-on-UNION-chain*)
 qed

lemma *under-underS-bij-betw*:

assumes *WELL*: *Well-order r* and *WELL'*: *Well-order r'* and
IN: $a \in \text{Field } r$ and *IN'*: $f a \in \text{Field } r'$ and
BIJ: *bij-betw f (underS r a) (underS r' (f a))*
 shows *bij-betw f (under r a) (under r' (f a))*
 proof –
 have $a \notin \text{underS } r a \wedge f a \notin \text{underS } r' (f a)$
 unfolding *underS-def* by *auto*
 moreover
 {have *Refl r* \wedge *Refl r'* using *WELL WELL'*
 by (*auto simp add: order-on-defs*)
 hence $\text{under } r a = \text{underS } r a \cup \{a\} \wedge$
 $\text{under } r' (f a) = \text{underS } r' (f a) \cup \{f a\}$
 using *IN IN'* by(*auto simp add: Refl-under-underS*)
 }
 ultimately show *?thesis*
 using *BIJ notIn-Un-bij-betw*[of a *underS r a f underS r' (f a)*] by *auto*
 qed

28.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions

Standardly, a function is an embedding of a well-order in another if it injectively and order-compatibly maps the former into an order filter of the latter. Here we opt for a more succinct definition (operator *embed*), asking that, for any element in the source, the function should be a bijection between the set of strict lower bounds of that element and the set of strict lower bounds of its image. (Later we prove equivalence with the standard definition – lemma *embed-iff-compat-inj-on-ofilter*.) A *strict embedding* (operator *embedS*) is a non-bijective embedding and an isomorphism (operator *iso*) is a bijective embedding.

definition *embed* :: $'a \text{ rel} \Rightarrow 'a' \text{ rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow \text{bool}$

where

$\text{embed } r r' f \equiv \forall a \in \text{Field } r. \text{bij-betw } f (\text{under } r a) (\text{under } r' (f a))$

lemmas *embed-defs* = *embed-def embed-def*[*abs-def*]

Strict embeddings:

definition *embedS* :: $'a \text{ rel} \Rightarrow 'a' \text{ rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow \text{bool}$

where

$\text{embedS } r r' f \equiv \text{embed } r r' f \wedge \neg \text{bij-betw } f (\text{Field } r) (\text{Field } r')$

lemmas $embedS-defs = embedS-def\ embedS-def[abs-def]$

definition $iso :: 'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where
 $iso\ r\ r'\ f \equiv embed\ r\ r'\ f \wedge bij-betw\ f\ (Field\ r)\ (Field\ r')$

lemmas $iso-defs = iso-def\ iso-def[abs-def]$

definition $compat :: 'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where
 $compat\ r\ r'\ f \equiv \forall a\ b. (a,b) \in r \longrightarrow (f\ a, f\ b) \in r'$

lemma $compat-wf$:
assumes $CMP: compat\ r\ r'\ f$ **and** $WF: wf\ r'$
shows $wf\ r$
proof –
have $r \leq inv-image\ r'\ f$
unfolding $inv-image-def$ **using** CMP
by $(auto\ simp\ add: compat-def)$
with WF **show** $?thesis$
using $wf-inv-image[of\ r'\ f]\ wf-subset[of\ inv-image\ r'\ f]$ **by** $auto$
qed

lemma $id-embed: embed\ r\ r\ id$
by $(auto\ simp\ add: id-def\ embed-def\ bij-betw-def)$

lemma $id-iso: iso\ r\ r\ id$
by $(auto\ simp\ add: id-def\ embed-def\ iso-def\ bij-betw-def)$

lemma $embed-compat$:
assumes $EMB: embed\ r\ r'\ f$
shows $compat\ r\ r'\ f$
unfolding $compat-def$
proof $clarify$
fix $a\ b$
assume $*$: $(a,b) \in r$
hence 1 : $b \in Field\ r$ **using** $Field-def[of\ r]$ **by** $blast$
have $a \in under\ r\ b$
using $*$ $under-def[of\ r]$ **by** $simp$
hence $f\ a \in under\ r'\ (f\ b)$
using $EMB\ embed-def[of\ r\ r'\ f]$
 $bij-betw-def[of\ f\ under\ r\ b\ under\ r'\ (f\ b)]$
 $image-def[of\ f\ under\ r\ b]\ 1$ **by** $auto$
thus $(f\ a, f\ b) \in r'$
by $(auto\ simp\ add: under-def)$
qed

lemma $embed-in-Field$:

assumes EMB : $embed\ r\ r'\ f$ **and** IN : $a \in Field\ r$
shows $f\ a \in Field\ r'$
proof –
have $a \in Domain\ r \vee a \in Range\ r$
using IN **unfolding** $Field-def$ **by** $blast$
then show $?thesis$
using $embed-compat$ [$OF\ EMB$]
unfolding $Field-def\ compat-def$ **by** $force$
qed

lemma $comp-embed$:
assumes EMB : $embed\ r\ r'\ f$ **and** EMB' : $embed\ r'\ r''\ f'$
shows $embed\ r\ r''\ (f' \circ f)$
proof($unfold\ embed-def$, $auto$)
fix a **assume** $*$: $a \in Field\ r$
hence $bij-betw\ f\ (under\ r\ a)\ (under\ r'\ (f\ a))$
using $embed-def[of\ r]$ EMB **by** $auto$
moreover
{ **have** $f\ a \in Field\ r'$
using $EMB\ *$ **by** ($auto\ simp\ add$: $embed-in-Field$)
hence $bij-betw\ f'\ (under\ r'\ (f\ a))\ (under\ r''\ (f'\ (f\ a)))$
using $embed-def[of\ r']$ EMB' **by** $auto$
}
ultimately
show $bij-betw\ (f' \circ f)\ (under\ r\ a)\ (under\ r''\ (f'\ (f\ a)))$
by($auto\ simp\ add$: $bij-betw-trans$)
qed

lemma $comp-iso$:
assumes EMB : $iso\ r\ r'\ f$ **and** EMB' : $iso\ r'\ r''\ f'$
shows $iso\ r\ r''\ (f' \circ f)$
using $assms$ **unfolding** $iso-def$
by ($auto\ simp\ add$: $comp-embed\ bij-betw-trans$)

That $embedS$ is also preserved by function composition shall be proved only later.

lemma $embed-Field$: $embed\ r\ r'\ f \implies f'(Field\ r) \leq Field\ r'$
by ($auto\ simp\ add$: $embed-in-Field$)

lemma $embed-preserves-ofilter$:
assumes $WELL$: $Well-order\ r$ **and** $WELL'$: $Well-order\ r'$ **and**
 EMB : $embed\ r\ r'\ f$ **and** OF : $wo-rel.ofilter\ r\ A$
shows $wo-rel.ofilter\ r'\ (f'A)$
proof–

from $WELL$ **have** $Well$: $wo-rel\ r$ **unfolding** $wo-rel-def$.
from $WELL'$ **have** $Well'$: $wo-rel\ r'$ **unfolding** $wo-rel-def$.
from OF **have** 0 : $A \leq Field\ r$ **by**($auto\ simp\ add$: $Well\ wo-rel.ofilter-def$)

```

show ?thesis using Well' WELL EMB 0 embed-Field[of r r' f]
proof(unfold wo-rel.ofilter-def, auto simp add: image-def)
  fix a b'
  assume *: a ∈ A and **: b' ∈ under r' (f a)
  hence a ∈ Field r using 0 by auto
  hence bij-betw f (under r a) (under r' (f a))
    using * EMB by (auto simp add: embed-def)
  hence f'(under r a) = under r' (f a)
    by (simp add: bij-betw-def)
  with ** image-def[of f under r a] obtain b where
    1: b ∈ under r a ∧ b' = f b by blast
  hence b ∈ A using Well * OF
    by (auto simp add: wo-rel.ofilter-def)
  with 1 show ∃ b ∈ A. b' = f b by blast
qed
qed

```

```

lemma embed-Field-ofilter:
  assumes WELL: Well-order r and WELL': Well-order r' and
    EMB: embed r r' f
  shows wo-rel.ofilter r' (f'(Field r))
proof–
  have wo-rel.ofilter r (Field r)
    using WELL by (auto simp add: wo-rel-def wo-rel.Field-ofilter)
  with WELL WELL' EMB
  show ?thesis by (auto simp add: embed-preserves-ofilter)
qed

```

```

lemma embed-inj-on:
  assumes WELL: Well-order r and EMB: embed r r' f
  shows inj-on f (Field r)
proof(unfold inj-on-def, clarify)

```

```

  from WELL have Well: wo-rel r unfolding wo-rel-def .
  with wo-rel.TOTAL[of r]
  have Total: Total r by simp
  from Well wo-rel.REFL[of r]
  have Refl: Refl r by simp

```

```

  fix a b
  assume *: a ∈ Field r and **: b ∈ Field r and
    ***: f a = f b
  hence 1: a ∈ Field r ∧ b ∈ Field r
    unfolding Field-def by auto
  {assume (a,b) ∈ r
  hence a ∈ under r b ∧ b ∈ under r b
    using Refl by(auto simp add: under-def refl-on-def)
  hence a = b
  using EMB 1 ***

```

```

    by (auto simp add: embed-def bij-betw-def inj-on-def)
  }
  moreover
  {assume (b,a) ∈ r
    hence a ∈ under r a ∧ b ∈ under r a
    using Reft by(auto simp add: under-def refl-on-def)
    hence a = b
    using EMB 1 ***
    by (auto simp add: embed-def bij-betw-def inj-on-def)
  }
  ultimately
  show a = b using Total 1
    by (auto simp add: total-on-def)
qed

```

lemma *embed-underS*:

```

  assumes WELL: Well-order r and
    EMB: embed r r' f and IN: a ∈ Field r
  shows bij-betw f (underS r a) (underS r' (f a))
proof -
  have f a ∈ Field r' using assms embed-Field[of r r' f] by auto
  then have 0: under r a = underS r a ∪ {a}
    by (simp add: IN Reft-under-underS WELL wo-rel.REFL wo-rel.intro)
  moreover have 1: bij-betw f (under r a) (under r' (f a))
    using assms by (auto simp add: embed-def)
  moreover have under r' (f a) = underS r' (f a) ∪ {f a}
  proof
    show under r' (f a) ⊆ underS r' (f a) ∪ {f a}
      using underS-def under-def by fastforce
    show underS r' (f a) ∪ {f a} ⊆ under r' (f a)
      using bij-betwE 0 1 underS-subset-under by fastforce
  qed
  moreover have a ∉ underS r a ∧ f a ∉ underS r' (f a)
    unfolding underS-def by blast
  ultimately show ?thesis
    by (auto simp add: notIn-Un-bij-betw3)
qed

```

lemma *embed-iff-compat-inj-on-ofilter*:

```

  assumes WELL: Well-order r and WELL': Well-order r'
  shows embed r r' f = (compat r r' f ∧ inj-on f (Field r) ∧ wo-rel.ofilter r'
    (f'(Field r)))
  using assms
proof(auto simp add: embed-compat embed-inj-on embed-Field-ofilter,
  unfold embed-def, auto)
  fix a
  assume *: inj-on f (Field r) and
    **: compat r r' f and
    ***: wo-rel.ofilter r' (f'(Field r)) and

```

```

****:  $a \in \text{Field } r$ 

have Well: wo-rel  $r$ 
  using WELL wo-rel-def[of  $r$ ] by simp
hence Reft: Reft  $r$ 
  using wo-rel.REFL[of  $r$ ] by simp
have Total: Total  $r$ 
  using Well wo-rel.TOTAL[of  $r$ ] by simp
have Well': wo-rel  $r'$ 
  using WELL' wo-rel-def[of  $r'$ ] by simp
hence Antisym': antisym  $r'$ 
  using wo-rel.ANTISYM[of  $r'$ ] by simp
have  $(a, a) \in r$ 
  using **** Well wo-rel.REFL[of  $r$ ]
  refl-on-def[of -  $r$ ] by auto
hence  $(f a, f a) \in r'$ 
  using ** by(auto simp add: compat-def)
hence 0:  $f a \in \text{Field } r'$ 
  unfolding Field-def by auto
have  $f a \in f'(\text{Field } r)$ 
  using **** by auto
hence 2: under  $r'$   $(f a) \leq f'(\text{Field } r)$ 
  using Well' *** wo-rel.ofilter-def[of  $r'$   $f'(\text{Field } r)$ ] by fastforce

show bij-betw  $f$  (under  $r$   $a$ ) (under  $r'$   $(f a)$ )
proof(unfold bij-betw-def, auto)
  show inj-on  $f$  (under  $r$   $a$ ) by (rule subset-inj-on[OF * under-Field])
next
  fix  $b$  assume  $b \in \text{under } r$   $a$ 
  thus  $f b \in \text{under } r'$   $(f a)$ 
    unfolding under-def using **
    by (auto simp add: compat-def)
next
  fix  $b'$  assume ****:  $b' \in \text{under } r'$   $(f a)$ 
  hence  $b' \in f'(\text{Field } r)$ 
    using 2 by auto
  with Field-def[of  $r$ ] obtain  $b$  where
    3:  $b \in \text{Field } r$  and 4:  $b' = f b$  by auto
  have  $(b, a) \in r$ 
  proof-
    {assume  $(a, b) \in r$ 
      with ** 4 have  $(f a, b') \in r'$ 
        by (auto simp add: compat-def)
      with **** Antisym' have  $f a = b'$ 
        by(auto simp add: under-def antisym-def)
      with 3 **** 4 * have  $a = b$ 
        by(auto simp add: inj-on-def)
    }
  moreover

```

```

    {assume a = b
      hence (b,a) ∈ r using Refl **** 3
        by (auto simp add: refl-on-def)
    }
    ultimately
    show ?thesis using Total **** 3 by (fastforce simp add: total-on-def)
  qed
  with 4 show b' ∈ f“(under r a)
    unfolding under-def by auto
  qed
qed

```

lemma *inv-into-ofilter-embed*:

```

  assumes WELL: Well-order r and OF: wo-rel.ofilter r A and
    BIJ: ∀ b ∈ A. bij-betw f (under r b) (under r' (f b)) and
    IMAGE: f ‘ A = Field r'
  shows embed r' r (inv-into A f)
proof –

```

```

  have Well: wo-rel r
    using WELL wo-rel-def[of r] by simp
  have Refl: Refl r
    using Well wo-rel.REFL[of r] by simp
  have Total: Total r
    using Well wo-rel.TOTAL[of r] by simp

```

```

  have 1: bij-betw f A (Field r')

```

```

proof(unfold bij-betw-def inj-on-def, auto simp add: IMAGE)

```

```

  fix b1 b2

```

```

  assume *: b1 ∈ A and **: b2 ∈ A and

```

```

  ***: f b1 = f b2

```

```

  have 11: b1 ∈ Field r ∧ b2 ∈ Field r

```

```

  using * ** Well OF by (auto simp add: wo-rel.ofilter-def)

```

```

  moreover

```

```

  {assume (b1,b2) ∈ r

```

```

    hence b1 ∈ under r b2 ∧ b2 ∈ under r b2

```

```

      unfolding under-def using 11 Refl

```

```

      by (auto simp add: refl-on-def)

```

```

    hence b1 = b2 using BIJ * ** ***

```

```

      by (simp add: bij-betw-def inj-on-def)

```

```

  }

```

```

  moreover

```

```

  {assume (b2,b1) ∈ r

```

```

    hence b1 ∈ under r b1 ∧ b2 ∈ under r b1

```

```

      unfolding under-def using 11 Refl

```

```

      by (auto simp add: refl-on-def)

```

```

    hence b1 = b2 using BIJ * ** ***

```

```

      by (simp add: bij-betw-def inj-on-def)

```

```

  }

```

```

ultimately
show  $b1 = b2$ 
  using Total by (auto simp add: total-on-def)
qed

let  $?f' = (inv\text{-into } A f)$ 

have 2:  $\forall b \in A. \text{bij\_betw } ?f' (\text{under } r' (f b)) (\text{under } r b)$ 
proof (clarify)
  fix  $b$  assume *:  $b \in A$ 
  hence  $\text{under } r b \leq A$ 
  using Well OF by (auto simp add: wo-rel.ofilter-def)
  moreover
  have  $f' (\text{under } r b) = \text{under } r' (f b)$ 
  using * BIJ by (auto simp add: bij-betw-def)
  ultimately
  show  $\text{bij\_betw } ?f' (\text{under } r' (f b)) (\text{under } r b)$ 
  using 1 by (auto simp add: bij-betw-inv-into-subset)
qed

have 3:  $\forall b' \in \text{Field } r'. \text{bij\_betw } ?f' (\text{under } r' b') (\text{under } r (?f' b'))$ 
proof (clarify)
  fix  $b'$  assume *:  $b' \in \text{Field } r'$ 
  have  $b' = f' (?f' b')$  using * 1
  by (auto simp add: bij-betw-inv-into-right)
  moreover
  {obtain  $b$  where 31:  $b \in A$  and  $f b = b'$  using IMAGE * by force
  hence  $?f' b' = b$  using 1 by (auto simp add: bij-betw-inv-into-left)
  with 31 have  $?f' b' \in A$  by auto
  }
  ultimately
  show  $\text{bij\_betw } ?f' (\text{under } r' b') (\text{under } r (?f' b'))$ 
  using 2 by auto
qed

thus thesis unfolding embed-def .
qed

```

lemma *inv-into-underS-embed*:

```

assumes WELL: Well-order  $r$  and
  BIJ:  $\forall b \in \text{underS } r a. \text{bij\_betw } f (\text{under } r b) (\text{under } r' (f b))$  and
  IN:  $a \in \text{Field } r$  and
  IMAGE:  $f' (\text{underS } r a) = \text{Field } r'$ 
shows  $\text{embed } r' r (\text{inv-into } (\text{underS } r a) f)$ 
using assms
by (auto simp add: wo-rel-def wo-rel.underS-ofilter inv-into-ofilter-embed)

```

lemma *inv-into-Field-embed*:

```

assumes WELL: Well-order  $r$  and EMB:  $\text{embed } r r' f$  and

```

```

  IMAGE: Field r' ≤ f ' (Field r)
  shows embed r' r (inv-into (Field r) f)
  proof –
  have (∀ b ∈ Field r. bij-betw f (under r b) (under r' (f b)))
    using EMB by (auto simp add: embed-def)
  moreover
  have f ' (Field r) ≤ Field r'
    using EMB WELL by (auto simp add: embed-Field)
  ultimately
  show ?thesis using assms
    by(auto simp add: wo-rel-def wo-rel.Field-ofilter inv-into-ofilter-embed)
  qed

```

lemma *inv-into-Field-embed-bij-betw*:

```

  assumes EMB: embed r r' f and BIJ: bij-betw f (Field r) (Field r')
  shows embed r' r (inv-into (Field r) f)
  proof –
  have Field r' ≤ f ' (Field r)
    using BIJ by (auto simp add: bij-betw-def)
  then have iso: iso r r' f
    by (simp add: BIJ EMB iso-def)
  have *: ∀ a. a ∈ Field r → bij-betw f (under r a) (under r' (f a))
    using EMB embed-def by fastforce
  show ?thesis
  proof (clarsimp simp add: embed-def)
  fix a
  assume a: a ∈ Field r'
  then have ar: a ∈ f ' Field r
    using BIJ bij-betw-imp-surj-on by blast
  have [simp]: f (inv-into (Field r) f a) = a
    by (simp add: ar f-inv-into-f)
  show bij-betw (inv-into (Field r) f) (under r' a) (under r (inv-into (Field r) f
  a))
  proof (rule bij-betw-inv-into-subset [OF BIJ])
  show under r (inv-into (Field r) f a) ⊆ Field r
    by (simp add: under-Field)
  have inv-into (Field r) f a ∈ Field r
    by (simp add: ar inv-into-into)
  then show f ' under r (inv-into (Field r) f a) = under r' a
    using bij-betw-imp-surj-on * by fastforce
  qed
  qed
  qed

```

28.3 Given any two well-orders, one can be embedded in the other

Here is an overview of the proof of of this fact, stated in theorem *wellorders-totally-ordered*:
 Fix the well-orders $r::'a \text{ rel}$ and $r'::'a' \text{ rel}$. Attempt to define an embedding

$f::'a \Rightarrow 'a'$ from r to r' in the natural way by well-order recursion ("hoping" that $\text{Field } r$ turns out to be smaller than $\text{Field } r'$), but also record, at the recursive step, in a function $g::'a \Rightarrow \text{bool}$, the extra information of whether $\text{Field } r'$ gets exhausted or not.

If $\text{Field } r'$ does not get exhausted, then $\text{Field } r$ is indeed smaller and f is the desired embedding from r to r' (lemma *wellorders-totally-ordered-aux*). Otherwise, it means that $\text{Field } r'$ is the smaller one, and the inverse of (the "good" segment of) f is the desired embedding from r' to r (lemma *wellorders-totally-ordered-aux2*).

lemma *wellorders-totally-ordered-aux*:

fixes $r :: 'a \text{ rel}$ **and** $r' :: 'a' \text{ rel}$ **and**

$f :: 'a \Rightarrow 'a'$ **and** $a :: 'a$

assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r' **and** *IN*: $a \in \text{Field } r$ **and**

IH: $\forall b \in \text{underS } r \ a. \text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b))$ **and**

NOT: $f \ '(\text{underS } r \ a) \neq \text{Field } r'$ **and** *SUC*: $f \ a = \text{wo-rel.suc } r' \ (f \ (\text{underS } r \ a))$

shows $\text{bij-betw } f \ (\text{under } r \ a) \ (\text{under } r' \ (f \ a))$

proof–

have *Well*: *wo-rel* r **using** *WELL* **unfolding** *wo-rel-def* .

hence *Refl*: *Refl* r **using** *wo-rel.REFL*[of r] **by** *auto*

have *Trans*: *trans* r **using** *Well wo-rel.TRANS*[of r] **by** *auto*

have *Well'*: *wo-rel* r' **using** *WELL'* **unfolding** *wo-rel-def* .

have *OF*: *wo-rel.ofilter* r (*underS* $r \ a$)

by (*auto simp add: Well wo-rel.underS-ofilter*)

hence *UN*: $\text{underS } r \ a = (\bigcup b \in \text{underS } r \ a. \text{under } r \ b)$

using *Well wo-rel.ofilter-under-UNION*[of r *underS* $r \ a$] **by** *blast*

{fix b **assume** $*$: $b \in \text{underS } r \ a$

hence $t0$: $(b, a) \in r \wedge b \neq a$ **unfolding** *underS-def* **by** *auto*

have $t1$: $b \in \text{Field } r$

using $*$ *underS-Field*[of $r \ a$] **by** *auto*

have $t2$: $f \ (\text{under } r \ b) = \text{under } r' \ (f \ b)$

using *IH* $*$ **by** (*auto simp add: bij-betw-def*)

hence $t3$: *wo-rel.ofilter* r' ($f \ (\text{under } r \ b)$)

using *Well'* **by** (*auto simp add: wo-rel.under-ofilter*)

have $f \ (\text{under } r \ b) \leq \text{Field } r'$

using $t2$ **by** (*auto simp add: under-Field*)

moreover

have $b \in \text{under } r \ b$

using $t1$ **by**(*auto simp add: Refl Refl-under-in*)

ultimately

have $t4$: $f \ b \in \text{Field } r'$ **by** *auto*

have $f \ (\text{under } r \ b) = \text{under } r' \ (f \ b) \wedge$

wo-rel.ofilter $r' \ (f \ (\text{under } r \ b)) \wedge$

$f \ b \in \text{Field } r'$


```

    using t2 t3 t4 by auto
  }
  hence bFact:
     $\forall b \in \text{underS } r \ a. f'(\text{under } r \ b) = \text{under } r' (f \ b) \wedge$ 
       $\text{wo-rel.ofilter } r' (f'(\text{under } r \ b)) \wedge$ 
       $f \ b \in \text{Field } r' \text{ by blast}$ 

  have subField:  $f'(\text{underS } r \ a) \leq \text{Field } r'$ 
    using bFact by blast

  have OF':  $\text{wo-rel.ofilter } r' (f'(\text{underS } r \ a))$ 
  proof-
    have  $f'(\text{underS } r \ a) = f'(\bigcup b \in \text{underS } r \ a. \text{under } r \ b)$ 
      using UN by auto
    also have ... =  $(\bigcup b \in \text{underS } r \ a. f'(\text{under } r \ b))$  by blast
    also have ... =  $(\bigcup b \in \text{underS } r \ a. (\text{under } r' (f \ b)))$ 
      using bFact by auto
    finally
    have  $f'(\text{underS } r \ a) = (\bigcup b \in \text{underS } r \ a. (\text{under } r' (f \ b)))$  .
    thus ?thesis
      using Well' bFact
       $\text{wo-rel.ofilter-UNION}[of } r' \ \text{underS } r \ a \ \lambda \ b. \ \text{under } r' (f \ b)] \text{ by fastforce}$ 
  qed

  have  $f'(\text{underS } r \ a) \cup \text{AboveS } r' (f'(\text{underS } r \ a)) = \text{Field } r'$ 
    using Well' OF' by (auto simp add: wo-rel.ofilter-AboveS-Field)
  hence NE:  $\text{AboveS } r' (f'(\text{underS } r \ a)) \neq \{\}$ 
    using subField NOT by blast

  have INCL1:  $f'(\text{underS } r \ a) \leq \text{underS } r' (f \ a)$ 
  proof(auto)
    fix b assume *:  $b \in \text{underS } r \ a$ 
    have  $f \ b \neq f \ a \wedge (f \ b, f \ a) \in r'$ 
      using subField Well' SUC NE *
       $\text{wo-rel.suc-greater}[of } r' \ f'(\text{underS } r \ a) \ f \ b] \text{ by force}$ 
    thus  $f \ b \in \text{underS } r' (f \ a)$ 
      unfolding underS-def by simp
  qed

  have INCL2:  $\text{underS } r' (f \ a) \leq f'(\text{underS } r \ a)$ 
  proof
    fix b' assume  $b' \in \text{underS } r' (f \ a)$ 
    hence  $b' \neq f \ a \wedge (b', f \ a) \in r'$ 
      unfolding underS-def by simp
    thus  $b' \in f'(\text{underS } r \ a)$ 
      using Well' SUC NE OF'
       $\text{wo-rel.suc-ofilter-in}[of } r' \ f' \ \text{underS } r \ a \ b'] \text{ by auto}$ 
  qed

```

```

have INJ: inj-on f (underS r a)
proof–
  have  $\forall b \in \text{underS } r \ a. \text{ inj-on } f \ (\text{under } r \ b)$ 
    using IH by (auto simp add: bij-betw-def)
  moreover
  have  $\forall b. \text{ wo-rel.ofilter } r \ (\text{under } r \ b)$ 
    using Well by (auto simp add: wo-rel.under-ofilter)
  ultimately show ?thesis
    using WELL bFact UN
      UNION-inj-on-ofilter[of r underS r a  $\lambda b. \text{ under } r \ b \ f$ ]
    by auto
qed

have BIJ: bij-betw f (underS r a) (underS r' (f a))
  unfolding bij-betw-def
  using INJ INCL1 INCL2 by auto

have f a  $\in \text{Field } r'$ 
  using Well' subField NE SUC
  by (auto simp add: wo-rel.suc-inField)
thus ?thesis
  using WELL WELL' IN BIJ under-underS-bij-betw[of r r' a f] by auto
qed

lemma wellorders-totally-ordered-aux2:
  fixes r :: 'a rel and r' :: 'a' rel and
    f :: 'a  $\Rightarrow$  'a' and g :: 'a  $\Rightarrow$  bool and a :: 'a
  assumes WELL: Well-order r and WELL': Well-order r' and
    MAIN1:
       $\bigwedge a. (\text{False} \notin g'(\text{underS } r \ a) \wedge f'(\text{underS } r \ a) \neq \text{Field } r' \rightarrow f \ a = \text{wo-rel.suc } r' (f'(\text{underS } r \ a)) \wedge g \ a = \text{True})$ 
       $\wedge$ 
       $(\neg(\text{False} \notin (g'(\text{underS } r \ a)) \wedge f'(\text{underS } r \ a) \neq \text{Field } r') \rightarrow g \ a = \text{False})$  and
    MAIN2:  $\bigwedge a. a \in \text{Field } r \wedge \text{False} \notin g'(\text{under } r \ a) \rightarrow$ 
      bij-betw f (under r a) (under r' (f a)) and
    Case:  $a \in \text{Field } r \wedge \text{False} \in g'(\text{under } r \ a)$ 
  shows  $\exists f'. \text{ embed } r' \ r \ f'$ 
proof–
  have Well: wo-rel r using WELL unfolding wo-rel-def .
  hence Refl: Refl r using wo-rel.REFL[of r] by auto
  have Trans: trans r using Well wo-rel.TRANS[of r] by auto
  have Antisym: antisym r using Well wo-rel.ANTISYM[of r] by auto
  have Well': wo-rel r' using WELL' unfolding wo-rel-def .

  have 0: under r a = underS r a  $\cup \{a\}$ 
    using Refl Case by(auto simp add: Refl-under-underS)

  have 1: g a = False

```

proof–

{**assume** $g\ a \neq \text{False}$
with $0\ \text{Case}$ **have** $\text{False} \in g'(\text{underS}\ r\ a)$ **by** *blast*
with *MAIN1* **have** $g\ a = \text{False}$ **by** *blast*}
thus *?thesis* **by** *blast*

qed

let $?A = \{a \in \text{Field}\ r.\ g\ a = \text{False}\}$
let $?a = (\text{wo-rel.minim}\ r\ ?A)$

have $2: ?A \neq \{\} \wedge ?A \leq \text{Field}\ r$ **using** *Case 1* **by** *blast*

have $3: \text{False} \notin g'(\text{underS}\ r\ ?a)$

proof

assume $\text{False} \in g'(\text{underS}\ r\ ?a)$
then obtain b **where** $b \in \text{underS}\ r\ ?a$ **and** $31: g\ b = \text{False}$ **by** *auto*
hence $32: (b, ?a) \in r \wedge b \neq ?a$
by *(auto simp add: underS-def)*
hence $b \in \text{Field}\ r$ **unfolding** *Field-def* **by** *auto*
with 31 **have** $b \in ?A$ **by** *auto*
hence $(?a, b) \in r$ **using** *wo-rel.minim-least 2 Well* **by** *fastforce*

with 32 *Antisym* **show** *False*
by *(auto simp add: antisym-def)*

qed

have *temp*: $?a \in ?A$
using *Well 2 wo-rel.minim-in[of r ?A]* **by** *auto*
hence $4: ?a \in \text{Field}\ r$ **by** *auto*

have $5: g\ ?a = \text{False}$ **using** *temp* **by** *blast*

have $6: f'(\text{underS}\ r\ ?a) = \text{Field}\ r'$
using *MAIN1[of ?a] 3 5* **by** *blast*

have $7: \forall b \in \text{underS}\ r\ ?a.\ \text{bij-betw}\ f\ (\text{under}\ r\ b)\ (\text{under}\ r'\ (f\ b))$

proof

fix b **assume** *as*: $b \in \text{underS}\ r\ ?a$
moreover
have *wo-rel.ofilter* $r\ (\text{underS}\ r\ ?a)$
using *Well* **by** *(auto simp add: wo-rel.underS-ofilter)*
ultimately
have $\text{False} \notin g'(\text{under}\ r\ b)$ **using** $3\ \text{Well}$ **by** *(subst (asm) wo-rel.ofilter-def)*

fast+

moreover **have** $b \in \text{Field}\ r$
unfolding *Field-def* **using** *as* **by** *(auto simp add: underS-def)*
ultimately
show *bij-betw* $f\ (\text{under}\ r\ b)\ (\text{under}\ r'\ (f\ b))$
using *MAIN2* **by** *auto*

qed

have *embed* $r' r$ (*inv-into* (*underS* r $?a$) f)
using *WELL WELL'* 7 4 6 *inv-into-underS-embed*[*of* r $?a$ $f r'$] **by** *auto*
thus *?thesis*
unfolding *embed-def* **by** *blast*
qed

theorem *wellorders-totally-ordered*:

fixes $r :: 'a$ *rel* **and** $r' :: 'a'$ *rel*
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $(\exists f. \text{embed } r r' f) \vee (\exists f'. \text{embed } r' r f')$
proof –

have *Well*: *wo-rel* r **using** *WELL* **unfolding** *wo-rel-def* .
hence *Refl*: *Refl* r **using** *wo-rel.REFL*[*of* r] **by** *auto*
have *Trans*: *trans* r **using** *Well wo-rel.TRANS*[*of* r] **by** *auto*
have *Well'*: *wo-rel* r' **using** *WELL'* **unfolding** *wo-rel-def* .

obtain H **where** *H-def*: $H =$

$(\lambda h a. \text{if } \text{False} \notin (\text{snd} \circ h)'(\text{underS } r a) \wedge (\text{fst} \circ h)'(\text{underS } r a) \neq \text{Field } r'$
 $\text{then } (\text{wo-rel.suc } r' ((\text{fst} \circ h)'(\text{underS } r a)), \text{True})$
 $\text{else } (\text{undefined}, \text{False}))$ **by** *blast*

have *Adm*: *wo-rel.adm-wo* r H

using *Well*

proof(*unfold wo-rel.adm-wo-def, clarify*)

fix $h1 :: 'a \Rightarrow 'a' * \text{bool}$ **and** $h2 :: 'a \Rightarrow 'a' * \text{bool}$ **and** x

assume $\forall y \in \text{underS } r x. h1 y = h2 y$

hence $\forall y \in \text{underS } r x. (\text{fst} \circ h1) y = (\text{fst} \circ h2) y \wedge$

$(\text{snd} \circ h1) y = (\text{snd} \circ h2) y$ **by** *auto*

hence $(\text{fst} \circ h1)'(\text{underS } r x) = (\text{fst} \circ h2)'(\text{underS } r x) \wedge$

$(\text{snd} \circ h1)'(\text{underS } r x) = (\text{snd} \circ h2)'(\text{underS } r x)$

by (*auto simp add: image-def*)

thus $H h1 x = H h2 x$ **by** (*simp add: H-def del: not-False-in-image-Ball*)

qed

obtain $h :: 'a \Rightarrow 'a' * \text{bool}$ **and** $f :: 'a \Rightarrow 'a'$ **and** $g :: 'a \Rightarrow \text{bool}$

where *h-def*: $h = \text{wo-rel.worec } r H$ **and**

f-def: $f = \text{fst} \circ h$ **and** *g-def*: $g = \text{snd} \circ h$ **by** *blast*

obtain *test* **where** *test-def*:

$\text{test} = (\lambda a. \text{False} \notin (g'(\text{underS } r a)) \wedge f'(\text{underS } r a) \neq \text{Field } r')$ **by** *blast*

have $*$: $\bigwedge a. h a = H h a$

using *Adm Well wo-rel.worec-fixpoint*[*of* $r H$] **by** (*simp add: h-def*)

have *Main1*:

$\bigwedge a. (\text{test } a \longrightarrow f a = \text{wo-rel.suc } r' (f'(\text{underS } r a)) \wedge g a = \text{True}) \wedge$

$(\neg(\text{test } a) \longrightarrow g a = \text{False})$

proof –

fix a **show** $(\text{test } a \longrightarrow f a = \text{wo-rel.suc } r' (f'(\text{underS } r a)) \wedge g a = \text{True}) \wedge$

$(\neg(\text{test } a) \longrightarrow g a = \text{False})$

using $*[$ *of* $a]$ *test-def f-def g-def H-def* **by** *auto*

qed

let $?phi = \lambda a. a \in \text{Field } r \wedge \text{False} \notin g'(\text{under } r \ a) \longrightarrow$
 $\text{bij-betw } f \ (\text{under } r \ a) \ (\text{under } r' \ (f \ a))$
 have *Main2*: $\bigwedge a. ?phi \ a$
 proof –
 fix *a* show $?phi \ a$
 proof(*rule wo-rel.well-order-induct[of r ?phi]*,
simp only: Well, clarify)
 fix *a*
 assume *IH*: $\forall b. b \neq a \wedge (b, a) \in r \longrightarrow ?phi \ b$ and
 *: $a \in \text{Field } r$ and
 **: $\text{False} \notin g'(\text{under } r \ a)$
 have *1*: $\forall b \in \text{underS } r \ a. \text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b))$
 proof(*clarify*)
 fix *b* assume ***: $b \in \text{underS } r \ a$
 hence *0*: $(b, a) \in r \wedge b \neq a$ **unfolding** *underS-def* **by** *auto*
 moreover have $b \in \text{Field } r$
 using *** *underS-Field[of r a]* **by** *auto*
 moreover have $\text{False} \notin g'(\text{under } r \ b)$
 using *0* ** *Trans under-incr[of r b a]* **by** *auto*
 ultimately show $\text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b))$
 using *IH* **by** *auto*
 qed

have *21*: $\text{False} \notin g'(\text{underS } r \ a)$
 using ** *underS-subset-under[of r a]* **by** *auto*
 have *22*: $g'(\text{under } r \ a) \leq \{\text{True}\}$ **using** ** **by** *auto*
 moreover have *23*: $a \in \text{under } r \ a$
 using *Refl ** **by** (*auto simp add: Refl-under-in*)
 ultimately have *24*: $g \ a = \text{True}$ **by** *blast*
 have *2*: $f'(\text{underS } r \ a) \neq \text{Field } r'$
 proof
 assume $f'(\text{underS } r \ a) = \text{Field } r'$
 hence $g \ a = \text{False}$ **using** *Main1 test-def* **by** *blast*
 with *24* show False **using** ** **by** *blast*
 qed

have *3*: $f \ a = \text{wo-rel.suc } r' \ (f'(\text{underS } r \ a))$
 using *21 2 Main1 test-def* **by** *blast*

show $\text{bij-betw } f \ (\text{under } r \ a) \ (\text{under } r' \ (f \ a))$
 using *WELL WELL' 1 2 3 **
wellorders-totally-ordered-aux[of r r' a f] **by** *auto*

qed

qed

let $?chi = (\lambda a. a \in \text{Field } r \wedge \text{False} \in g'(\text{under } r \ a))$
 show *?thesis*

proof(cases $\exists a. ?chi\ a$)
assume $\neg (\exists a. ?chi\ a)$
hence $\forall a \in Field\ r. \text{bij-betw } f\ (\text{under } r\ a)\ (\text{under } r'\ (f\ a))$
using *Main2* **by** *blast*
thus *?thesis unfolding embed-def* **by** *blast*
next
assume $\exists a. ?chi\ a$
then obtain *a* **where** *?chi a* **by** *blast*
hence $\exists f'. \text{embed } r'\ r\ f'$
using *wellorders-totally-ordered-aux2[of r r' g f a]*
WELL WELL' Main1 Main2 test-def **by** *fast*
thus *?thesis* **by** *blast*
qed
qed

28.4 Uniqueness of embeddings

Here we show a fact complementary to the one from the previous subsection – namely, that between any two well-orders there is *at most* one embedding, and is the one definable by the expected well-order recursive equation. As a consequence, any two embeddings of opposite directions are mutually inverse.

lemma *embed-determined*:

assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
EMB: *embed r r' f* **and** *IN*: $a \in Field\ r$
shows $f\ a = \text{wo-rel.suc } r'\ (f'(\text{underS } r\ a))$

proof –

have $\text{bij-betw } f\ (\text{underS } r\ a)\ (\text{underS } r'\ (f\ a))$
using *assms* **by** (*auto simp add: embed-underS*)
hence $f'(\text{underS } r\ a) = \text{underS } r'\ (f\ a)$
by (*auto simp add: bij-betw-def*)
moreover
{ **have** $f\ a \in Field\ r'$ **using** *IN*
using *EMB WELL embed-Field[of r r' f]* **by** *auto*
hence $f\ a = \text{wo-rel.suc } r'\ (\text{underS } r'\ (f\ a))$
using *WELL'* **by** (*auto simp add: wo-rel-def wo-rel.suc-underS*)
}

ultimately show *?thesis* **by** *simp*
qed

lemma *embed-unique*:

assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
EMBf: *embed r r' f* **and** *EMBg*: *embed r r' g*
shows $a \in Field\ r \longrightarrow f\ a = g\ a$

proof(*rule wo-rel.well-order-induct[of r], auto simp add: WELL wo-rel-def*)
fix *a*

assume *IH*: $\forall b. b \neq a \wedge (b, a) \in r \longrightarrow b \in Field\ r \longrightarrow f\ b = g\ b$ **and**
 $*$: $a \in Field\ r$

hence $\forall b \in \text{underS } r \ a. f \ b = g \ b$
 unfolding *underS-def* by (*auto simp add: Field-def*)
 hence $f'(\text{underS } r \ a) = g'(\text{underS } r \ a)$ by *force*
 thus $f \ a = g \ a$
 using *assms * embed-determined[of r r' f a] embed-determined[of r r' g a]* by
auto
 qed

lemma *embed-bothWays-inverse:*

assumes *WELL: Well-order r* and *WELL': Well-order r'* and
EMB: embed r r' f and *EMB': embed r' r f'*
 shows $(\forall a \in \text{Field } r. f'(f \ a) = a) \wedge (\forall a' \in \text{Field } r'. f(f' \ a') = a')$
proof –
 have *embed r r (f' o f)* using *assms*
 by(*auto simp add: comp-embed*)
 moreover have *embed r r id* using *assms*
 by (*auto simp add: id-embed*)
 ultimately have $\forall a \in \text{Field } r. f'(f \ a) = a$
 using *assms embed-unique[of r r f' o f id] id-def* by *auto*
 moreover
 { have *embed r' r' (f o f')* using *assms*
 by(*auto simp add: comp-embed*)
 moreover have *embed r' r' id* using *assms*
 by (*auto simp add: id-embed*)
 ultimately have $\forall a' \in \text{Field } r'. f(f' \ a') = a'$
 using *assms embed-unique[of r' r' f o f' id] id-def* by *auto*
 }
 ultimately show *?thesis* by *blast*
 qed

lemma *embed-bothWays-bij-betw:*

assumes *WELL: Well-order r* and *WELL': Well-order r'* and
EMB: embed r r' f and *EMB': embed r' r g*
 shows *bij-betw f (Field r) (Field r')*
proof –
 let *?A = Field r* let *?A' = Field r'*
 have *embed r r (g o f) o f* and *embed r' r' (f o g) o g*
 using *assms* by (*auto simp add: comp-embed*)
 hence 1: $(\forall a \in ?A. g(f \ a) = a) \wedge (\forall a' \in ?A'. f(g \ a') = a')$
 using *WELL id-embed[of r] embed-unique[of r r g o f id]*
WELL' id-embed[of r'] embed-unique[of r' r' f o g id]
id-def by *auto*
 have 2: $(\forall a \in ?A. f \ a \in ?A') \wedge (\forall a' \in ?A'. g \ a' \in ?A)$
 using *assms embed-Field[of r r' f] embed-Field[of r' r g]* by *blast*

show *?thesis*
proof(*unfold bij-betw-def inj-on-def, auto simp add: 2*)
 fix *a b* assume *: $a \in ?A \ b \in ?A'$ and **: $f \ a = g \ b$
 have $a = g(f \ a) \wedge b = g(f \ b)$ using * 1 by *auto*

```

  with ** show  $a = b$  by auto
next
  fix  $a'$  assume *:  $a' \in ?A'$ 
  hence  $g\ a' \in ?A \wedge f(g\ a') = a'$  using 1 2 by auto
  thus  $a' \in f\ ' ?A$  by force
qed
qed

```

lemma *embed-bothWays-iso*:

```

  assumes WELL: Well-order  $r$  and WELL': Well-order  $r'$  and
    EMB: embed  $r\ r'\ f$  and EMB': embed  $r'\ r\ g$ 
  shows iso  $r\ r'\ f$ 
  unfolding iso-def using assms by (auto simp add: embed-bothWays-bij-betw)

```

28.5 More properties of embeddings, strict embeddings and isomorphisms

lemma *embed-bothWays-Field-bij-betw*:

```

  assumes WELL: Well-order  $r$  and WELL': Well-order  $r'$  and
    EMB: embed  $r\ r'\ f$  and EMB': embed  $r'\ r\ f'$ 
  shows bij-betw  $f$  (Field  $r$ ) (Field  $r'$ )

```

proof –

```

  have  $(\forall a \in \text{Field } r. f'(f\ a) = a) \wedge (\forall a' \in \text{Field } r'. f(f'\ a') = a')$ 
    using assms by (auto simp add: embed-bothWays-inverse)

```

moreover

```

  have  $f'(\text{Field } r) \leq \text{Field } r' \wedge f'\ ' (\text{Field } r') \leq \text{Field } r$ 
    using assms by (auto simp add: embed-Field)

```

ultimately

```

  show ?thesis using bij-betw-byWitness[of Field  $r\ f'\ f\ \text{Field } r'$ ] by auto

```

qed

lemma *embedS-comp-embed*:

```

  assumes WELL: Well-order  $r$  and WELL': Well-order  $r'$ 
    and EMB: embedS  $r\ r'\ f$  and EMB': embed  $r'\ r''\ f'$ 
  shows embedS  $r\ r''\ (f' \circ f)$ 

```

proof –

```

  let ?g =  $(f' \circ f)$  let ?h = inv-into (Field  $r$ ) ?g

```

```

  have 1: embed  $r\ r'\ f \wedge \neg (\text{bij-betw } f \text{ (Field } r) \text{ (Field } r'))$ 
    using EMB by (auto simp add: embedS-def)

```

```

  hence 2: embed  $r\ r''\ ?g$ 

```

```

    using EMB' comp-embed[of  $r\ r'\ f\ r''\ f'$ ] by auto

```

moreover

```

  {assume bij-betw ?g (Field  $r$ ) (Field  $r''$ )

```

```

    hence embed  $r''\ r\ ?h$  using 2

```

```

      by (auto simp add: inv-into-Field-embed-bij-betw)

```

```

    hence embed  $r'\ r\ (?h \circ f')$  using EMB'

```

```

      by (auto simp add: comp-embed)

```

```

    hence bij-betw  $f$  (Field  $r$ ) (Field  $r'$ ) using WELL WELL' 1

```

```

      by (auto simp add: embed-bothWays-Field-bij-betw)

```


with 1 have *False* by *blast*
 }
 ultimately show *?thesis unfolding embedS-def* by *auto*
 qed

lemma *embed-comp-embedS*:

assumes *WELL*: *Well-order r* and *WELL'*: *Well-order r'*
 and *EMB*: *embed r r' f* and *EMB'*: *embedS r' r'' f'*
 shows *embedS r r'' (f' o f)*

proof –

let $?g = (f' \circ f)$ let $?h = \text{inv-into } (\text{Field } r) ?g$
 have 1: $\text{embed } r' r'' f' \wedge \neg (\text{bij-betw } f' (\text{Field } r') (\text{Field } r''))$
 using *EMB'* by (*auto simp add: embedS-def*)
 hence 2: $\text{embed } r r'' ?g$
 using *WELL EMB comp-embed[of r r' f r'' f']* by *auto*
 moreover have $\S: f' \text{ ' } \text{Field } r' \subseteq \text{Field } r''$
 by (*simp add: 1 embed-Field*)
 {assume $\S: \text{bij-betw } ?g (\text{Field } r) (\text{Field } r'')$
 hence $\text{embed } r'' r ?h$ using 2 *WELL*
 by (*auto simp add: inv-into-Field-embed-bij-betw*)
 hence $\text{embed } r' r (\text{inv-into } (\text{Field } r) ?g \circ f')$
 using 1 *BNF-Wellorder-Embedding.comp-embed WELL'* by *blast*
 then have $\text{bij-betw } f' (\text{Field } r') (\text{Field } r'')$
 using *EMB WELL WELL' § bij-betw-comp-iff* by (*blast intro: embed-bothWays-Field-bij-betw*)
 with 1 have *False* by *blast*

}
 ultimately show *?thesis unfolding embedS-def* by *auto*
 qed

lemma *embed-comp-iso*:

assumes *EMB*: *embed r r' f* and *EMB'*: *iso r' r'' f'*
 shows $\text{embed } r r'' (f' \circ f)$ using *assms unfolding iso-def*
 by (*auto simp add: comp-embed*)

lemma *iso-comp-embed*:

assumes *EMB*: *iso r r' f* and *EMB'*: *embed r' r'' f'*
 shows $\text{embed } r r'' (f' \circ f)$
 using *assms unfolding iso-def* by (*auto simp add: comp-embed*)

lemma *embedS-comp-iso*:

assumes *EMB*: *embedS r r' f* and *EMB'*: *iso r' r'' f'*
 shows $\text{embedS } r r'' (f' \circ f)$

proof –

have $\S: \text{embed } r r' f \wedge \neg \text{bij-betw } f (\text{Field } r) (\text{Field } r')$
 using *EMB embedS-def* by *blast*
 then have $\text{embed } r r'' (f' \circ f)$
 using *embed-comp-iso EMB'* by *blast*
 then have $f \text{ ' } \text{Field } r \subseteq \text{Field } r'$
 by (*simp add: embed-Field §*)

then have $\neg \text{bij-betw } (f' \circ f) \text{ (Field } r) \text{ (Field } r'')$
using § *EMB'* **by** *(auto simp: bij-betw-comp-iff2 iso-def)*
then show *?thesis*
by *(simp add: ‹embed r r'' (f' ∘ f)› embedS-def)*
qed

lemma *iso-comp-embedS*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
and *EMB*: *iso r r' f* **and** *EMB'*: *embedS r' r'' f'*
shows *embedS r r'' (f' ∘ f)*
using *assms unfolding iso-def* **by** *(auto simp add: embed-comp-embedS)*

lemma *embedS-Field*:
assumes *WELL*: *Well-order r* **and** *EMB*: *embedS r r' f*
shows $f' \text{ (Field } r) < \text{Field } r'$
proof –
have $f' \text{ (Field } r) \leq \text{Field } r'$ **using** *assms*
by *(auto simp add: embed-Field embedS-def)*
moreover
{ **have** *inj-on f (Field r)* **using** *assms*
by *(auto simp add: embedS-def embed-inj-on)*
hence $f' \text{ (Field } r) \neq \text{Field } r'$ **using** *EMB*
by *(auto simp add: embedS-def bij-betw-def)*
}
ultimately show *?thesis* **by** *blast*
qed

lemma *embedS-iff*:
assumes *WELL*: *Well-order r* **and** *ISO*: *embed r r' f*
shows $\text{embedS } r \text{ } r' \text{ } f = (f' \text{ (Field } r) < \text{Field } r')$
proof
assume *embedS r r' f*
thus $f' \text{ (Field } r) \subset \text{Field } r'$
using *WELL* **by** *(auto simp add: embedS-Field)*
next
assume $f' \text{ (Field } r) \subset \text{Field } r'$
hence $\neg \text{bij-betw } f \text{ (Field } r) \text{ (Field } r')$
unfolding *bij-betw-def* **by** *blast*
thus *embedS r r' f* **unfolding** *embedS-def*
using *ISO* **by** *auto*
qed

lemma *iso-Field*: $\text{iso } r \text{ } r' \text{ } f \implies f' \text{ (Field } r) = \text{Field } r'$
by *(auto simp add: iso-def bij-betw-def)*

lemma *iso-iff*:
assumes *Well-order r*
shows $\text{iso } r \text{ } r' \text{ } f = (\text{embed } r \text{ } r' \text{ } f \wedge f' \text{ (Field } r) = \text{Field } r')$
proof

assume $iso\ r\ r'\ f$
thus $embed\ r\ r'\ f \wedge f' (Field\ r) = Field\ r'$
by (*auto simp add: iso-Field iso-def*)
next
assume $*$: $embed\ r\ r'\ f \wedge f' (Field\ r) = Field\ r'$
hence $inj-on\ f (Field\ r)$ **using** *assms* **by** (*auto simp add: embed-inj-on*)
with $*$ **have** $bij-betw\ f (Field\ r) (Field\ r')$
unfolding *bij-betw-def* **by** *simp*
with $*$ **show** $iso\ r\ r'\ f$ **unfolding** *iso-def* **by** *auto*
qed

lemma *iso-iff2*: $iso\ r\ r'\ f \longleftrightarrow$
 $bij-betw\ f (Field\ r) (Field\ r') \wedge$
 $(\forall a \in Field\ r. \forall b \in Field\ r. (a, b) \in r \longleftrightarrow (f\ a, f\ b) \in r')$
(is *?lhs* **=** *?rhs***)**

proof
assume *L*: *?lhs*
then **have** $bij-betw\ f (Field\ r) (Field\ r')$ **and** emb : $embed\ r\ r'\ f$
by (*auto simp: bij-betw-def iso-def*)
then **obtain** g **where** $g: \bigwedge x. x \in Field\ r \implies g (f\ x) = x$
by (*auto simp: bij-betw-iff-bijections*)
moreover
have $(a, b) \in r$ **if** $a \in Field\ r\ b \in Field\ r\ (f\ a, f\ b) \in r'$ **for** $a\ b$
using *that emb g g [OF FieldI1]* — yes it's weird
by (*force simp add: embed-def under-def bij-betw-iff-bijections*)
ultimately **show** *?rhs*
using *L* **by** (*auto simp: compat-def iso-def dest: embed-compat*)

next
assume *R*: *?rhs*
then **show** *?lhs*
apply (*clarsimp simp add: iso-def embed-def under-def bij-betw-iff-bijections*)
apply (*rule-tac x=g in exI*)
apply (*fastforce simp add: intro: FieldI1*)
done

qed

lemma *iso-iff3*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
shows $iso\ r\ r'\ f = (bij-betw\ f (Field\ r) (Field\ r') \wedge compat\ r\ r'\ f)$
proof
assume $iso\ r\ r'\ f$
thus $bij-betw\ f (Field\ r) (Field\ r') \wedge compat\ r\ r'\ f$
unfolding *compat-def* **using** *WELL* **by** (*auto simp add: iso-iff2 Field-def*)
next
have *Well*: $wo-rel\ r \wedge wo-rel\ r'$ **using** *WELL WELL'*
by (*auto simp add: wo-rel-def*)
assume $*$: $bij-betw\ f (Field\ r) (Field\ r') \wedge compat\ r\ r'\ f$
thus $iso\ r\ r'\ f$
unfolding *compat-def* **using** *assms*

proof (*auto simp add: iso-iff2*)
fix $a\ b$ **assume** $**$: $a \in \text{Field } r\ b \in \text{Field } r$ **and**
 $***$: $(f\ a, f\ b) \in r'$
{ **assume** $(b, a) \in r \vee b = a$
hence $(b, a) \in r$ **using** *Well ** wo-rel.REFL[of r] refl-on-def[of - r]* **by** *blast*
hence $(f\ b, f\ a) \in r'$ **using** $*$ **unfolding** *compat-def* **by** *auto*
hence $f\ a = f\ b$
using *Well *** wo-rel.ANTISYM[of r'] antisym-def[of r']* **by** *blast*
hence $a = b$ **using** $*$ **unfolding** *bij-betw-def inj-on-def* **by** *auto*
hence $(a, b) \in r$ **using** *Well ** wo-rel.REFL[of r] refl-on-def[of - r]* **by** *blast*
}
thus $(a, b) \in r$
using *Well ** wo-rel.TOTAL[of r] total-on-def[of - r]* **by** *blast*
qed
qed

lemma *iso-imp-inj-on*:
assumes *iso r r' f* **shows** *inj-on f (Field r)*
using *assms unfolding iso-iff2 bij-betw-def* **by** *blast*

lemma *iso-backward-Field*:
assumes $x \in \text{Field } r'$ *iso r r' f*
shows *inv-into (Field r) f x* $\in \text{Field } r$
using *assms iso-Field* **by** (*blast intro!: inv-into-into*)

lemma *iso-backward*:
assumes $(x, y) \in r'$ **and** *iso: iso r r' f*
shows $(\text{inv-into } (\text{Field } r) f\ x, \text{inv-into } (\text{Field } r) f\ y) \in r$
proof –
have \S : $\bigwedge x. (\exists xa \in \text{Field } r. x = f\ xa) = (x \in \text{Field } r')$
using *assms iso-Field [OF iso]* **by** (*force simp add:*)
have $x \in \text{Field } r'\ y \in \text{Field } r'$
using *assms* **by** (*auto simp: Field-def*)
with \S [of x] \S [of y] *assms* **show** *?thesis*
by (*auto simp add: iso-iff2 bij-betw-inv-into-left*)
qed

lemma *iso-forward*:
assumes $(x, y) \in r$ *iso r r' f* **shows** $(f\ x, f\ y) \in r'$
using *assms* **by** (*auto simp: Field-def iso-iff2*)

lemma *iso-trans*:
assumes *trans r* **and** *iso: iso r r' f* **shows** *trans r'*
unfolding *trans-def*
proof *clarify*
fix $x\ y\ z$
assume *xyz: (x, y) ∈ r' (y, z) ∈ r'*
then have $*$: $(\text{inv-into } (\text{Field } r) f\ x, \text{inv-into } (\text{Field } r) f\ y) \in r$
 $(\text{inv-into } (\text{Field } r) f\ y, \text{inv-into } (\text{Field } r) f\ z) \in r$

using *iso-backward* [*OF - iso*] **by** *blast+*
then have *inv-into* (*Field r*) *f x* \in *Field r* *inv-into* (*Field r*) *f y* \in *Field r*
by (*auto simp: Field-def*)
with * **have** (*inv-into* (*Field r*) *f x*, *inv-into* (*Field r*) *f z*) \in *r*
using *assms(1)* **by** (*blast intro: transD*)
then have (*f* (*inv-into* (*Field r*) *f x*), *f* (*inv-into* (*Field r*) *f z*)) \in *r'*
by (*blast intro: iso iso-forward*)
moreover have *x* \in *f* ‘ *Field r* *z* \in *f* ‘ *Field r*
using *xyz iso iso-Field* **by** (*blast intro: FieldI1 FieldI2*)
ultimately show (*x*, *z*) \in *r'*
by (*simp add: f-inv-into-f*)
qed

lemma *iso-Total*:

assumes *Total r* **and** *iso: iso r r' f* **shows** *Total r'*
unfolding *total-on-def*
proof *clarify*
fix *x y*
assume *xy: x* \in *Field r'* *y* \in *Field r'* *x* \neq *y* (*y,x*) \notin *r'*
then have §: *inv-into* (*Field r*) *f x* \in *Field r* *inv-into* (*Field r*) *f y* \in *Field r*
using *iso-backward-Field* [*OF - iso*] **by** *auto*
moreover have *x* \in *f* ‘ *Field r* *y* \in *f* ‘ *Field r*
using *xy iso iso-Field* **by** (*blast intro: FieldI1 FieldI2*)
ultimately have *False* **if** *inv-into* (*Field r*) *f x* = *inv-into* (*Field r*) *f y*
using *inv-into-injective* [*OF that*] $\langle x \neq y \rangle$ **by** *simp*
then have (*inv-into* (*Field r*) *f x*, *inv-into* (*Field r*) *f y*) \in *r* \vee (*inv-into* (*Field r*) *f y*, *inv-into* (*Field r*) *f x*) \in *r*
using *assms* § **by** (*auto simp: total-on-def*)
then show (*x*, *y*) \in *r'*
using *assms xy* **by** (*auto simp: iso-Field f-inv-into-f dest!: iso-forward* [*OF - iso*])
qed

lemma *iso-wf*:

assumes *wf r* **and** *iso: iso r r' f* **shows** *wf r'*
proof –
have *bij-betw f* (*Field r*) (*Field r'*)
and *iff: ($\forall a \in$ Field r. $\forall b \in$ Field r. $(a, b) \in r \iff (f a, f b) \in r'$)*
using *assms* **by** (*auto simp: iso-iff2*)
show *?thesis*
proof (*rule wfI-min*)
fix *x::'b* **and** *Q*
assume *x* \in *Q*
let *?g* = *inv-into* (*Field r*) *f*
obtain *z0* **where** *z0* \in *?g* ‘ *Q*
using $\langle x \in Q \rangle$ **by** *blast*
then obtain *z* **where** *z* : *z* \in *?g* ‘ *Q* **and** $\bigwedge x y. [(y, z) \in r; x \in Q] \implies y \neq ?g x$
by (*rule wfE-min* [*OF* $\langle wf r \rangle$]) *auto*

```

then have  $\forall y. (y, \text{inv-into } Q \text{ ?}g z) \in r' \longrightarrow y \notin Q$ 
  by (clarsimp simp: f-inv-into-f[OF z] z dest!: iso-backward [OF - iso]) blast
moreover have  $\text{inv-into } Q \text{ ?}g z \in Q$ 
  by (simp add: inv-into-into z)
ultimately show  $\exists z \in Q. \forall y. (y, z) \in r' \longrightarrow y \notin Q ..$ 
qed
qed

end

```

29 Constructions on Wellorders as Needed by Bounded Natural Functors

```

theory BNF-Wellorder-Constructions
  imports BNF-Wellorder-Embedding
begin

```

In this section, we study basic constructions on well-orders, such as restriction to a set/order filter, copy via direct images, ordinal-like sum of disjoint well-orders, and bounded square. We also define between well-orders the relations *ordLeq*, of being embedded (abbreviated $\leq o$), *ordLess*, of being strictly embedded (abbreviated $< o$), and *ordIso*, of being isomorphic (abbreviated $= o$). We study the connections between these relations, order filters, and the aforementioned constructions. A main result of this section is that $< o$ is well-founded.

29.1 Restriction to a set

```

abbreviation Restr :: 'a rel  $\Rightarrow$  'a set  $\Rightarrow$  'a rel
  where Restr r A  $\equiv$  r Int (A  $\times$  A)

```

```

lemma Restr-subset:
  A  $\leq$  B  $\Longrightarrow$  Restr (Restr r B) A = Restr r A
by blast

```

```

lemma Restr-Field: Restr r (Field r) = r
unfolding Field-def by auto

```

```

lemma Refl-Restr: Refl r  $\Longrightarrow$  Refl(Restr r A)
unfolding refl-on-def Field-def by auto

```

```

lemma linear-order-on-Restr:
  linear-order-on A r  $\Longrightarrow$  linear-order-on (A  $\cap$  above r x) (Restr r (above r x))
by(simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def)(safe; blast)

```

```

lemma antisym-Restr:

```

antisym $r \implies \text{antisym}(\text{Restr } r A)$
unfolding *antisym-def Field-def* **by** *auto*

lemma *Total-Restr*:
 $\text{Total } r \implies \text{Total}(\text{Restr } r A)$
unfolding *total-on-def Field-def* **by** *auto*

lemma *total-on-imp-Total-Restr*: $\text{total-on } A r \implies \text{Total}(\text{Restr } r A)$
by (*auto simp: Field-def total-on-def*)

lemma *trans-Restr*:
 $\text{trans } r \implies \text{trans}(\text{Restr } r A)$
unfolding *trans-def Field-def* **by** *blast*

lemma *Preorder-Restr*:
 $\text{Preorder } r \implies \text{Preorder}(\text{Restr } r A)$
unfolding *preorder-on-def* **by** (*simp add: Refl-Restr trans-Restr*)

lemma *Partial-order-Restr*:
 $\text{Partial-order } r \implies \text{Partial-order}(\text{Restr } r A)$
unfolding *partial-order-on-def* **by** (*simp add: Preorder-Restr antisym-Restr*)

lemma *Linear-order-Restr*:
 $\text{Linear-order } r \implies \text{Linear-order}(\text{Restr } r A)$
unfolding *linear-order-on-def* **by** (*simp add: Partial-order-Restr Total-Restr*)

lemma *Well-order-Restr*:
assumes *Well-order* r
shows $\text{Well-order}(\text{Restr } r A)$
using *assms*
by (*auto simp: well-order-on-def Linear-order-Restr elim: wf-subset*)

lemma *Field-Restr-subset*: $\text{Field}(\text{Restr } r A) \leq A$
by (*auto simp add: Field-def*)

lemma *Refl-Field-Restr*:
 $\text{Refl } r \implies \text{Field}(\text{Restr } r A) = (\text{Field } r) \text{ Int } A$
unfolding *refl-on-def Field-def* **by** *blast*

lemma *Refl-Field-Restr2*:
 $\llbracket \text{Refl } r; A \leq \text{Field } r \rrbracket \implies \text{Field}(\text{Restr } r A) = A$
by (*auto simp add: Refl-Field-Restr*)

lemma *well-order-on-Restr*:
assumes *WELL*: $\text{Well-order } r$ **and** *SUB*: $A \leq \text{Field } r$
shows $\text{well-order-on } A (\text{Restr } r A)$
using *assms*
using $\text{Well-order-Restr}[of\ r\ A]$ $\text{Refl-Field-Restr2}[of\ r\ A]$
order-on-defs[of Field r r] **by** *auto*

29.2 Order filters versus restrictions and embeddings

lemma *Field-Restr-ofilter*:

$\llbracket \text{Well-order } r; \text{wo-rel.ofilter } r A \rrbracket \implies \text{Field}(\text{Restr } r A) = A$

by (*auto simp add: wo-rel-def wo-rel.ofilter-def wo-rel.REFL Refl-Field-Restr2*)

lemma *ofilter-Restr-under*:

assumes *WELL*: *Well-order* r and *OF*: *wo-rel.ofilter* $r A$ and *IN*: $a \in A$

shows *under* $(\text{Restr } r A) a = \text{under } r a$

unfolding *wo-rel.ofilter-def under-def*

proof

show $\{b. (b, a) \in \text{Restr } r A\} \subseteq \{b. (b, a) \in r\}$

by *auto*

next

have *under* $r a \subseteq A$

proof

fix x

assume *: $x \in \text{under } r a$

then have $a \in \text{Field } r$

unfolding *under-def using Field-def by fastforce*

then show $x \in A$ **using** *IN assms **

by (*auto simp add: wo-rel-def wo-rel.ofilter-def*)

qed

then show $\{b. (b, a) \in r\} \subseteq \{b. (b, a) \in \text{Restr } r A\}$

unfolding *under-def using assms by auto*

qed

lemma *ofilter-embed*:

assumes *Well-order* r

shows *wo-rel.ofilter* $r A = (A \leq \text{Field } r \wedge \text{embed } (\text{Restr } r A) r \text{ id})$

proof

assume *: *wo-rel.ofilter* $r A$

show $A \leq \text{Field } r \wedge \text{embed } (\text{Restr } r A) r \text{ id}$

unfolding *embed-def*

proof *safe*

fix a assume $a \in A$ **thus** $a \in \text{Field } r$ **using** *assms **

by (*auto simp add: wo-rel-def wo-rel.ofilter-def*)

next

fix a assume $a \in \text{Field } (\text{Restr } r A)$

thus *bij-betw id (under (Restr r A) a) (under r (id a)) using assms **

by (*simp add: ofilter-Restr-under Field-Restr-ofilter*)

qed

next

assume *: $A \leq \text{Field } r \wedge \text{embed } (\text{Restr } r A) r \text{ id}$

hence $\text{Field}(\text{Restr } r A) \leq \text{Field } r$

using *assms embed-Field[of Restr r A r id] id-def*

Well-order-Restr[of r] by auto

{**fix** a **assume** $a \in A$

hence $a \in \text{Field}(\text{Restr } r A)$ **using** ** assms*

by (*simp add: order-on-defs Refl-Field-Restr2*)


```

    hence bij-betw id (under (Restr r A) a) (under r a)
      using * unfolding embed-def by auto
    hence under r a ≤ under (Restr r A) a
      unfolding bij-betw-def by auto
    also have ... ≤ Field (Restr r A) by (simp add: under-Field)
    also have ... ≤ A by (simp add: Field-Restr-subset)
    finally have under r a ≤ A .
  }
thus wo-rel.ofilter r A using assms * by (simp add: wo-rel-def wo-rel.ofilter-def)
qed

```

lemma *ofilter-Restr-Int:*

assumes *WELL: Well-order r* **and** *OFA: wo-rel.ofilter r A*
shows *wo-rel.ofilter (Restr r B) (A Int B)*

proof –

```

  let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence Field: Field ?rB = Field r Int B
    using Refl-Field-Restr by blast
  have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
    by (simp add: Well-order-Restr wo-rel-def)

```

show *?thesis* **using** *WellB assms*

unfolding *wo-rel.ofilter-def under-def ofilter-def*

proof *safe*

fix *a* **assume** *a ∈ A* **and** **: a ∈ B*

hence *a ∈ Field r* **using** *OFA Well* **by** (*auto simp add: wo-rel.ofilter-def*)

with *** **show** *a ∈ Field ?rB* **using** *Field* **by** *auto*

next

fix *a b* **assume** *a ∈ A* **and** *(b,a) ∈ r*

thus *b ∈ A* **using** *Well OFA* **by** (*auto simp add: wo-rel.ofilter-def under-def*)

qed

qed

lemma *ofilter-Restr-subset:*

assumes *WELL: Well-order r* **and** *OFA: wo-rel.ofilter r A* **and** *SUB: A ≤ B*
shows *wo-rel.ofilter (Restr r B) A*

proof –

have *A Int B = A* **using** *SUB* **by** *blast*

thus *?thesis* **using** *assms ofilter-Restr-Int[of r A B]* **by** *auto*

qed

lemma *ofilter-subset-embed:*

assumes *WELL: Well-order r* **and**

OFA: wo-rel.ofilter r A **and** *OFB: wo-rel.ofilter r B*

shows *(A ≤ B) = (embed (Restr r A) (Restr r B) id)*

proof –

let *?rA = Restr r A* **let** *?rB = Restr r B*

```

have Well: wo-rel r unfolding wo-rel-def using WELL .
hence Refl: Refl r by (simp add: wo-rel.REFL)
hence FieldA: Field ?rA = Field r Int A
  using Refl-Field-Restr by blast
have FieldB: Field ?rB = Field r Int B
  using Refl Refl-Field-Restr by blast
have WellA: wo-rel ?rA ∧ Well-order ?rA using WELL
  by (simp add: Well-order-Restr wo-rel-def)
have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
  by (simp add: Well-order-Restr wo-rel-def)

show ?thesis
proof
  assume *: A ≤ B
  hence wo-rel.ofilter (Restr r B) A using assms
    by (simp add: ofilter-Restr-subset)
  hence embed (Restr ?rB A) (Restr r B) id
    using WellB ofilter-embed[of ?rB A] by auto
  thus embed (Restr r A) (Restr r B) id
    using * by (simp add: Restr-subset)
next
  assume *: embed (Restr r A) (Restr r B) id
  {fix a assume **: a ∈ A
   hence a ∈ Field r using Well OFA by (auto simp add: wo-rel.ofilter-def)
   with ** FieldA have a ∈ Field ?rA by auto
   hence a ∈ Field ?rB using * WellA embed-Field[of ?rA ?rB id] by auto
   hence a ∈ B using FieldB by auto
  }
  thus A ≤ B by blast
qed
qed

lemma ofilter-subset-embedS-iso:
  assumes WELL: Well-order r and
    OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows ((A < B) = (embedS (Restr r A) (Restr r B) id)) ∧
    ((A = B) = (iso (Restr r A) (Restr r B) id))
proof-
  let ?rA = Restr r A let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence Field ?rA = Field r Int A
    using Refl-Field-Restr by blast
  hence FieldA: Field ?rA = A using OFA Well
    by (auto simp add: wo-rel.ofilter-def)
  have Field ?rB = Field r Int B
    using Refl Refl-Field-Restr by blast
  hence FieldB: Field ?rB = B using OFB Well
    by (auto simp add: wo-rel.ofilter-def)

```

show *?thesis unfolding embedS-def iso-def*
using *assms ofilter-subset-embed[of r A B]*
FieldA FieldB bij-betw-id-iff[of A B] **by** *auto*
qed

lemma *ofilter-subset-embedS:*
assumes *WELL: Well-order r* **and**
OFA: wo-rel.ofilter r A **and** *OFB: wo-rel.ofilter r B*
shows $(A < B) = \text{embedS } (\text{Restr } r A) (\text{Restr } r B) \text{ id}$
using *assms* **by** *(simp add: ofilter-subset-embedS-iso)*

lemma *embed-implies-iso-Restr:*
assumes *WELL: Well-order r* **and** *WELL': Well-order r'* **and**
EMB: embed r' r f
shows $\text{iso } r' (\text{Restr } r (f \text{ ` } (\text{Field } r'))) f$

proof –

let $?A' = \text{Field } r'$
let $?r'' = \text{Restr } r (f \text{ ` } ?A')$
have *0: Well-order ?r''* **using** *WELL Well-order-Restr* **by** *blast*
have *1: wo-rel.ofilter r (f ` ?A')* **using** *assms embed-Field-ofilter* **by** *blast*
hence $\text{Field } ?r'' = f \text{ ` } (\text{Field } r')$ **using** *WELL Field-Restr-ofilter* **by** *blast*
hence *bij-betw f ?A' (Field ?r'')*
using *EMB embed-inj-on WELL' unfolding bij-betw-def* **by** *blast*
moreover
{ **have** $\forall a b. (a, b) \in r' \longrightarrow a \in \text{Field } r' \wedge b \in \text{Field } r'$
unfolding *Field-def* **by** *auto*
hence *compat r' ?r'' f*
using *assms embed-iff-compat-inj-on-ofilter*
unfolding *compat-def* **by** *blast*
}
ultimately show *?thesis* **using** *WELL' 0 iso-iff3* **by** *blast*
qed

29.3 The strict inclusion on proper ofilters is well-founded

definition *ofilterIncl* :: $'a \text{ rel} \Rightarrow 'a \text{ set rel}$

where

$$\text{ofilterIncl } r \equiv \{(A, B). \text{wo-rel.ofilter } r A \wedge A \neq \text{Field } r \wedge \\ \text{wo-rel.ofilter } r B \wedge B \neq \text{Field } r \wedge A < B\}$$

lemma *wf-ofilterIncl:*
assumes *WELL: Well-order r*
shows $\text{wf}(\text{ofilterIncl } r)$

proof –

have *Well: wo-rel r* **using** *WELL* **by** *(simp add: wo-rel-def)*
hence *Lo: Linear-order r* **by** *(simp add: wo-rel.LIN)*
let $?h = (\lambda A. \text{wo-rel.suc } r A)$
let $?rS = r - \text{Id}$

```

have wf ?rS using WELL by (simp add: order-on-defs)
moreover
have compat (ofilterIncl r) ?rS ?h
proof(unfold compat-def ofilterIncl-def,
  intro allI impI, simp, elim conjE)
fix A B
assume *: wo-rel.ofilter r A A ≠ Field r and
  **: wo-rel.ofilter r B B ≠ Field r and ***: A < B
then obtain a and b where 0: a ∈ Field r ∧ b ∈ Field r and
  1: A = underS r a ∧ B = underS r b
using Well by (auto simp add: wo-rel.ofilter-underS-Field)
hence a ≠ b using *** by auto
moreover
have (a,b) ∈ r using 0 1 Lo ***
by (auto simp add: underS-incl-iff)
moreover
have a = wo-rel.suc r A ∧ b = wo-rel.suc r B
using Well 0 1 by (simp add: wo-rel.suc-underS)
ultimately
show (wo-rel.suc r A, wo-rel.suc r B) ∈ r ∧ wo-rel.suc r A ≠ wo-rel.suc r B
by simp
qed
ultimately show wf (ofilterIncl r) by (simp add: compat-wf)
qed

```

29.4 Ordering the well-orders by existence of embeddings

We define three relations between well-orders:

- *ordLeq*, of being embedded (abbreviated $\leq o$);
- *ordLess*, of being strictly embedded (abbreviated $< o$);
- *ordIso*, of being isomorphic (abbreviated $= o$).

The prefix "ord" and the index "o" in these names stand for "ordinal-like". These relations shall be proved to be inter-connected in a similar fashion as the trio $\leq, <, =$ associated to a total order on a set.

definition *ordLeq* :: ('a rel * 'a' rel) set
where
ordLeq = {(r,r'). Well-order r ∧ Well-order r' ∧ (∃f. embed r r' f)}

abbreviation *ordLeq2* :: 'a rel ⇒ 'a' rel ⇒ bool (**infix** <=o 50)
where r <=o r' ≡ (r,r') ∈ *ordLeq*

abbreviation *ordLeq3* :: 'a rel ⇒ 'a' rel ⇒ bool (**infix** ≤o 50)
where r ≤o r' ≡ r <=o r'

definition *ordLess* :: ('a rel * 'a' rel) set

where

$ordLess = \{(r,r'). \text{ Well-order } r \wedge \text{ Well-order } r' \wedge (\exists f. \text{ embedS } r \ r' \ f)\}$

abbreviation $ordLess2 :: 'a \ \text{rel} \Rightarrow 'a' \ \text{rel} \Rightarrow \text{bool}$ (**infix** $<o$ 50)

where $r <o r' \equiv (r,r') \in ordLess$

definition $ordIso :: ('a \ \text{rel} * 'a' \ \text{rel}) \ \text{set}$

where

$ordIso = \{(r,r'). \text{ Well-order } r \wedge \text{ Well-order } r' \wedge (\exists f. \text{ iso } r \ r' \ f)\}$

abbreviation $ordIso2 :: 'a \ \text{rel} \Rightarrow 'a' \ \text{rel} \Rightarrow \text{bool}$ (**infix** $=o$ 50)

where $r =o r' \equiv (r,r') \in ordIso$

lemmas $ordRels-def = ordLeq-def \ ordLess-def \ ordIso-def$

lemma $ordLeq-Well-order-simp$:

assumes $r \leq_o r'$

shows $\text{Well-order } r \wedge \text{Well-order } r'$

using $assms$ **unfolding** $ordLeq-def$ **by** $simp$

Notice that the relations \leq_o , $<o$, $=o$ connect well-orders on potentially *distinct* types. However, some of the lemmas below, including the next one, restrict implicitly the type of these relations to $(('a \ \text{rel}) * ('a' \ \text{rel})) \ \text{set}$, i.e., to $'a \ \text{rel} \ \text{rel}$.

lemma $ordLeq-reflexive$:

$\text{Well-order } r \Longrightarrow r \leq_o r$

unfolding $ordLeq-def$ **using** $id-embed[of \ r]$ **by** $blast$

lemma $ordLeq-transitive[trans]$:

assumes $r \leq_o r'$ **and** $r' \leq_o r''$

shows $r \leq_o r''$

using $assms$ **by** $(\text{auto } simp: ordLeq-def \ \text{intro: comp-embed})$

lemma $ordLeq-total$:

$[\text{Well-order } r; \text{Well-order } r'] \Longrightarrow r \leq_o r' \vee r' \leq_o r$

unfolding $ordLeq-def$ **using** $wellorders-totally-ordered$ **by** $blast$

lemma $ordIso-reflexive$:

$\text{Well-order } r \Longrightarrow r =o r$

unfolding $ordIso-def$ **using** $id-iso[of \ r]$ **by** $blast$

lemma $ordIso-transitive[trans]$:

assumes $*$: $r =o r'$ **and** $**$: $r' =o r''$

shows $r =o r''$

using $assms$ **by** $(\text{auto } simp: ordIso-def \ \text{intro: comp-iso})$

lemma $ordIso-symmetric$:

assumes $*$: $r =o r'$

shows $r' =o r$

proof –

obtain f **where** 1: *Well-order* $r \wedge$ *Well-order* r' **and**
 2: *embed* r $r' f \wedge$ *bij-betw* f (*Field* r) (*Field* r')
using * **by** (*auto simp add: ordIso-def iso-def*)
let $?f' =$ *inv-into* (*Field* r) f
have *embed* $r' r$ $?f' \wedge$ *bij-betw* $?f'$ (*Field* r') (*Field* r)
using 1 2 **by** (*simp add: bij-betw-inv-into inv-into-Field-embed-bij-betw*)
thus $r' =_o r$ **unfolding** *ordIso-def* **using** 1 **by** (*auto simp add: iso-def*)
qed

lemma *ordLeq-ordLess-trans*[*trans*]:

assumes $r \leq_o r'$ **and** $r' <_o r''$
shows $r <_o r''$

proof –

have *Well-order* $r \wedge$ *Well-order* r''
using *assms unfolding ordLeq-def ordLess-def* **by** *auto*
thus $?thesis$ **using** *assms unfolding ordLeq-def ordLess-def*
using *embed-comp-embedS* **by** *blast*
qed

lemma *ordLess-ordLeq-trans*[*trans*]:

assumes $r <_o r'$ **and** $r' \leq_o r''$
shows $r <_o r''$

using *embedS-comp-embed assms* **by** (*force simp: ordLeq-def ordLess-def*)

lemma *ordLeq-ordIso-trans*[*trans*]:

assumes $r \leq_o r'$ **and** $r' =_o r''$
shows $r \leq_o r''$

using *embed-comp-iso assms* **by** (*force simp: ordLeq-def ordIso-def*)

lemma *ordIso-ordLeq-trans*[*trans*]:

assumes $r =_o r'$ **and** $r' \leq_o r''$
shows $r \leq_o r''$

using *iso-comp-embed assms* **by** (*force simp: ordLeq-def ordIso-def*)

lemma *ordLess-ordIso-trans*[*trans*]:

assumes $r <_o r'$ **and** $r' =_o r''$
shows $r <_o r''$

using *embedS-comp-iso assms* **by** (*force simp: ordLess-def ordIso-def*)

lemma *ordIso-ordLess-trans*[*trans*]:

assumes $r =_o r'$ **and** $r' <_o r''$
shows $r <_o r''$

using *iso-comp-embedS assms* **by** (*force simp: ordLess-def ordIso-def*)

lemma *ordLess-not-embed*:

assumes $r <_o r'$

shows $\neg(\exists f'. \text{embed } r' r f')$

proof –

obtain f where 1: *Well-order $r \wedge$ Well-order r'* **and** 2: *embed $r r' f$* **and**
 3: \neg *bij-betw f (Field r) (Field r')*
using *assms unfolding ordLess-def* **by** *(auto simp add: embedS-def)*
{fix f' assume $*$: embed $r' r f'$
hence *bij-betw f (Field r) (Field r')* **using** 1 2
by *(simp add: embed-bothWays-Field-bij-betw)*
with 3 have *False* **by** *contradiction*
}
thus *?thesis* **by** *blast*
qed

lemma *ordLess-Field*:

assumes *OL: $r1 <_o r2$* **and** *EMB: embed $r1 r2 f$*
shows \neg *($f'(Field\ r1) = Field\ r2$)*
proof –
let *?A1 = Field $r1$* **let** *?A2 = Field $r2$*
obtain g where
 0: *Well-order $r1 \wedge$ Well-order $r2$* **and**
 1: *embed $r1 r2 g \wedge \neg$ (bij-betw g ?A1 ?A2)*
using *OL unfolding ordLess-def* **by** *(auto simp add: embedS-def)*
hence $\forall a \in ?A1. f\ a = g\ a$
using 0 *EMB embed-unique[of $r1$]* **by** *auto*
hence \neg (*bij-betw f ?A1 ?A2*)
using 1 *bij-betw-cong[of ?A1]* **by** *blast*
moreover
have *inj-on f ?A1* **using** *EMB 0* **by** *(simp add: embed-inj-on)*
ultimately show *?thesis* **by** *(simp add: bij-betw-def)*
qed

lemma *ordLess-iff*:

$r <_o r' = (Well\ order\ r \wedge Well\ order\ r' \wedge \neg(\exists f'.\ embed\ r' r f'))$
proof
assume $*$: $r <_o r'$
hence $\neg(\exists f'.\ embed\ r' r f')$ **using** *ordLess-not-embed[of $r r'$]* **by** *simp*
with $*$ show *Well-order $r \wedge$ Well-order $r' \wedge \neg(\exists f'.\ embed\ r' r f')$*
unfolding *ordLess-def* **by** *auto*
next
assume $*$: $Well\ order\ r \wedge Well\ order\ r' \wedge \neg(\exists f'.\ embed\ r' r f')$
then obtain f where 1: *embed $r r' f$*
using *wellorders-totally-ordered[of $r r'$]* **by** *blast*
moreover
{assume *bij-betw f (Field r) (Field r')*
with $*$ 1 have *embed $r' r$ (inv-into (Field r) f)*
using *inv-into-Field-embed-bij-betw[of $r r' f$]* **by** *auto*
with $*$ have *False* **by** *blast*
}
ultimately show *(r, r') \in ordLess*
unfolding *ordLess-def* **using** $*$ **by** *(fastforce simp add: embedS-def)*
qed

lemma *ordLess-irreflexive*: $\neg r <_o r$
using *id-embed*[of *r*] **by** (*auto simp: ordLess-iff*)

lemma *ordLeq-iff-ordLess-or-ordIso*:
 $r \leq_o r' = (r <_o r' \vee r =_o r')$
unfolding *ordRels-def embedS-defs iso-defs* **by** *blast*

lemma *ordIso-iff-ordLeq*:
 $(r =_o r') = (r \leq_o r' \wedge r' \leq_o r)$
proof
assume $r =_o r'$
then obtain *f* **where** 1: *Well-order* $r \wedge$ *Well-order* $r' \wedge$
 $\text{embed } r \ r' \ f \wedge \text{bij-betw } f \ (\text{Field } r) \ (\text{Field } r')$
unfolding *ordIso-def iso-defs* **by** *auto*
hence $\text{embed } r \ r' \ f \wedge \text{embed } r' \ r \ (\text{inv-into } (\text{Field } r) \ f)$
by (*simp add: inv-into-Field-embed-bij-betw*)
thus $r \leq_o r' \wedge r' \leq_o r$
unfolding *ordLeq-def* **using** 1 **by** *auto*

next
assume $r \leq_o r' \wedge r' \leq_o r$
then obtain *f* **and** *g* **where** 1: *Well-order* $r \wedge$ *Well-order* $r' \wedge$
 $\text{embed } r \ r' \ f \wedge \text{embed } r' \ r \ g$
unfolding *ordLeq-def* **by** *auto*
hence $\text{iso } r \ r' \ f$ **by** (*auto simp add: embed-bothWays-iso*)
thus $r =_o r'$ **unfolding** *ordIso-def* **using** 1 **by** *auto*

qed

lemma *not-ordLess-ordLeq*:
 $r <_o r' \implies \neg r' \leq_o r$
using *ordLess-ordLeq-trans ordLess-irreflexive* **by** *blast*

lemma *not-ordLeq-ordLess*:
 $r \leq_o r' \implies \neg r' <_o r$
using *not-ordLess-ordLeq* **by** *blast*

lemma *ordLess-or-ordLeq*:
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $r <_o r' \vee r' \leq_o r$
proof –
have $r \leq_o r' \vee r' \leq_o r$
using *assms* **by** (*simp add: ordLeq-total*)
moreover
{ **assume** $\neg r <_o r' \wedge r \leq_o r'$
hence $r =_o r'$ **using** *ordLeq-iff-ordLess-or-ordIso* **by** *blast*
hence $r' \leq_o r$ **using** *ordIso-symmetric ordIso-iff-ordLeq* **by** *blast*
}
ultimately show *?thesis* **by** *blast*

qed

lemma *not-ordLess-ordIso*:

$$r <_o r' \implies \neg r =_o r'$$

using *ordLess-ordIso-trans ordIso-symmetric ordLess-irreflexive* **by** *blast*

lemma *not-ordLeq-iff-ordLess*:

assumes *WELL*: Well-order r **and** *WELL'*: Well-order r'

shows $(\neg r' \leq_o r) = (r <_o r')$

using *assms not-ordLess-ordLeq ordLess-or-ordLeq* **by** *blast*

lemma *not-ordLess-iff-ordLeq*:

assumes *WELL*: Well-order r **and** *WELL'*: Well-order r'

shows $(\neg r' <_o r) = (r \leq_o r')$

using *assms not-ordLess-ordLeq ordLess-or-ordLeq* **by** *blast*

lemma *ordLess-transitive[trans]*:

$$\llbracket r <_o r'; r' <_o r'' \rrbracket \implies r <_o r''$$

using *ordLess-ordLeq-trans ordLeq-iff-ordLess-or-ordIso* **by** *blast*

corollary *ordLess-trans: trans ordLess*

unfolding *trans-def* **using** *ordLess-transitive* **by** *blast*

lemmas *ordIso-equivalence = ordIso-transitive ordIso-reflexive ordIso-symmetric*

lemma *ordIso-imp-ordLeq*:

$$r =_o r' \implies r \leq_o r'$$

using *ordIso-iff-ordLeq* **by** *blast*

lemma *ordLess-imp-ordLeq*:

$$r <_o r' \implies r \leq_o r'$$

using *ordLeq-iff-ordLess-or-ordIso* **by** *blast*

lemma *ofilter-subset-ordLeq*:

assumes *WELL*: Well-order r **and**

OFA: *wo-rel.ofilter* r A **and** *OFB*: *wo-rel.ofilter* r B

shows $(A \leq B) = (\text{Restr } r A \leq_o \text{Restr } r B)$

proof

assume $A \leq B$

thus $\text{Restr } r A \leq_o \text{Restr } r B$

unfolding *ordLeq-def* **using** *assms*

Well-order-Restr Well-order-Restr ofilter-subset-embed **by** *blast*

next

assume $*$: $\text{Restr } r A \leq_o \text{Restr } r B$

then obtain f **where** *embed* $(\text{Restr } r A) (\text{Restr } r B) f$

unfolding *ordLeq-def* **by** *blast*

{**assume** $B < A$

hence $\text{Restr } r B <_o \text{Restr } r A$

unfolding *ordLess-def* **using** *assms*

Well-order-Restr Well-order-Restr ofilter-subset-embedS **by** *blast*

hence *False* using * *not-ordLess-ordLeq* by *blast*
 }
 thus $A \leq B$ using *OFA OFB WELL*
 wo-rel-def[of r] wo-rel.ofilter-linord[of r A B] by *blast*
 qed

lemma *ofilter-subset-ordLess*:
 assumes *WELL: Well-order r* and
 OFA: wo-rel.ofilter r A and *OFB: wo-rel.ofilter r B*
 shows $(A < B) = (\text{Restr } r A <_o \text{Restr } r B)$
proof –
 let $?rA = \text{Restr } r A$ let $?rB = \text{Restr } r B$
 have 1: *Well-order ?rA* \wedge *Well-order ?rB*
 using *WELL Well-order-Restr* by *blast*
 have $(A < B) = (\neg B \leq A)$ using *assms*
 wo-rel-def wo-rel.ofilter-linord[of r A B] by *blast*
 also have $\dots = (\neg \text{Restr } r B <_o \text{Restr } r A)$
 using *assms ofilter-subset-ordLeq* by *blast*
 also have $\dots = (\text{Restr } r A <_o \text{Restr } r B)$
 using 1 *not-ordLeq-iff-ordLess* by *blast*
 finally show *?thesis* .
 qed

lemma *ofilter-ordLess*:
 $[\text{Well-order } r; \text{wo-rel.ofilter } r A] \implies (A < \text{Field } r) = (\text{Restr } r A <_o r)$
 by (*simp add: ofilter-subset-ordLess wo-rel.Field-ofilter*
 wo-rel-def Restr-Field)

corollary *underS-Restr-ordLess*:
 assumes *Well-order r* and *Field r* $\neq \{\}$
 shows $\text{Restr } r (\text{underS } r a) <_o r$
proof –
 have $\text{underS } r a < \text{Field } r$ using *assms*
 by (*simp add: underS-Field3*)
 thus *?thesis* using *assms*
 by (*simp add: ofilter-ordLess wo-rel.underS-ofilter wo-rel-def*)
 qed

lemma *embed-ordLess-ofilterIncl*:
 assumes
 OL12: r1 <_o r2 and *OL23: r2 <_o r3* and
 EMB13: embed r1 r3 f13 and *EMB23: embed r2 r3 f23*
 shows $(f13^{\text{Field } r1}, f23^{\text{Field } r2}) \in (\text{ofilterIncl } r3)$
proof –
 have *OL13: r1 <_o r3*
 using *OL12 OL23* using *ordLess-transitive* by *auto*
 let $?A1 = \text{Field } r1$ let $?A2 = \text{Field } r2$ let $?A3 = \text{Field } r3$
 obtain *f12 g23* where
 0: Well-order r1 \wedge *Well-order r2* \wedge *Well-order r3* and

1: $\text{embed } r1 \ r2 \ f12 \wedge \neg(\text{bij-betw } f12 \ ?A1 \ ?A2)$ **and**
 2: $\text{embed } r2 \ r3 \ g23 \wedge \neg(\text{bij-betw } g23 \ ?A2 \ ?A3)$
using *OL12 OL23* **by** (*auto simp add: ordLess-def embedS-def*)
hence $\forall a \in ?A2. f23 \ a = g23 \ a$
using *EMB23 embed-unique[of r2 r3]* **by** *blast*
hence 3: $\neg(\text{bij-betw } f23 \ ?A2 \ ?A3)$
using 2 *bij-betw-cong[of ?A2 f23 g23]* **by** *blast*

have 4: $\text{wo-rel.ofilter } r2 \ (f12 \ ' \ ?A1) \wedge f12 \ ' \ ?A1 \neq ?A2$
using 0 1 *OL12* **by** (*simp add: embed-Field-ofilter ordLess-Field*)
have 5: $\text{wo-rel.ofilter } r3 \ (f23 \ ' \ ?A2) \wedge f23 \ ' \ ?A2 \neq ?A3$
using 0 *EMB23 OL23* **by** (*simp add: embed-Field-ofilter ordLess-Field*)
have 6: $\text{wo-rel.ofilter } r3 \ (f13 \ ' \ ?A1) \wedge f13 \ ' \ ?A1 \neq ?A3$
using 0 *EMB13 OL13* **by** (*simp add: embed-Field-ofilter ordLess-Field*)

have $f12 \ ' \ ?A1 < ?A2$
using 0 4 **by** (*auto simp add: wo-rel-def wo-rel.ofilter-def*)
moreover **have** *inj-on f23 ?A2*
using *EMB23 0* **by** (*simp add: wo-rel-def embed-inj-on*)
ultimately
have $f23 \ ' \ (f12 \ ' \ ?A1) < f23 \ ' \ ?A2$ **by** (*simp add: image-strict-mono*)
moreover
 {
have *embed r1 r3 (f23 o f12)*
using 1 *EMB23 0* **by** (*auto simp add: comp-embed*)
hence $\forall a \in ?A1. f23(f12 \ a) = f13 \ a$
using *EMB13 0 embed-unique[of r1 r3 f23 o f12 f13]* **by** *auto*
hence $f23 \ ' \ (f12 \ ' \ ?A1) = f13 \ ' \ ?A1$ **by** *force*
 }
ultimately
have $f13 \ ' \ ?A1 < f23 \ ' \ ?A2$ **by** *simp*

with 5 6 **show** *?thesis*
unfolding *ofilterIncl-def* **by** *auto*
qed

lemma *ordLess-iff-ordIso-Restr*:
assumes *WELL: Well-order r and WELL': Well-order r'*
shows $(r' <_o r) = (\exists a \in \text{Field } r. r' =_o \text{Restr } r \ (\text{underS } r \ a))$
proof *safe*
fix *a* **assume** *: $a \in \text{Field } r$ **and** **: $r' =_o \text{Restr } r \ (\text{underS } r \ a)$
hence $\text{Restr } r \ (\text{underS } r \ a) <_o r$ **using** *WELL underS-Restr-ordLess[of r]* **by**
blast
thus $r' <_o r$ **using** ** *ordIso-ordLess-trans* **by** *blast*

next
assume $r' <_o r$
then **obtain** *f* **where** 1: *Well-order r* \wedge *Well-order r'* **and**
 2: $\text{embed } r' \ r \ f \wedge f \ ' \ (\text{Field } r') \neq \text{Field } r$
unfolding *ordLess-def embedS-def[abs-def] bij-betw-def* **using** *embed-inj-on* **by**
blast

hence $wo\text{-rel.}\text{ofilter } r (f' (Field\ r'))$ **using** $embed\text{-Field}\text{-ofilter}$ **by** $blast$
then obtain a **where** $\exists a \in Field\ r$ **and** $\exists a: underS\ r\ a = f' (Field\ r')$
using $1\ 2$ **by** $(auto\ simp\ add: wo\text{-rel.}\text{ofilter}\text{-underS}\text{-Field}\ wo\text{-rel}\text{-def})$
have $iso\ r' (Restr\ r (f' (Field\ r')))$ f
using $embed\text{-implies}\text{-iso}\text{-Restr}\ 2\ assms$ **by** $blast$
moreover have $Well\text{-order} (Restr\ r (f' (Field\ r')))$
using $WELL\ Well\text{-order}\text{-Restr}$ **by** $blast$
ultimately have $r' =_o Restr\ r (f' (Field\ r'))$
using $WELL'$ **unfolding** $ordIso\text{-def}$ **by** $auto$
hence $r' =_o Restr\ r (underS\ r\ a)$ **using** 4 **by** $auto$
thus $\exists a \in Field\ r. r' =_o Restr\ r (underS\ r\ a)$ **using** 3 **by** $auto$
qed

lemma $internalize\text{-ordLess}$:

$$(r' <_o r) = (\exists p. Field\ p < Field\ r \wedge r' =_o p \wedge p <_o r)$$

proof

assume $*$: $r' <_o r$

hence 0 : $Well\text{-order } r \wedge Well\text{-order } r'$ **unfolding** $ordLess\text{-def}$ **by** $auto$

with $*$ **obtain** a **where** 1 : $a \in Field\ r$ **and** 2 : $r' =_o Restr\ r (underS\ r\ a)$

using $ordLess\text{-iff}\text{-ordIso}\text{-Restr}$ **by** $blast$

let $?p = Restr\ r (underS\ r\ a)$

have $wo\text{-rel.}\text{ofilter } r (underS\ r\ a)$ **using** 0

by $(simp\ add: wo\text{-rel}\text{-def}\ wo\text{-rel.}\text{underS}\text{-ofilter})$

hence $Field\ ?p = underS\ r\ a$ **using** 0 $Field\text{-Restr}\text{-ofilter}$ **by** $blast$

hence $Field\ ?p < Field\ r$ **using** $underS\text{-Field}2\ 1$ **by** $fast$

moreover have $?p <_o r$ **using** $underS\text{-Restr}\text{-ordLess}[of\ r\ a]\ 0\ 1$ **by** $blast$

ultimately show $\exists p. Field\ p < Field\ r \wedge r' =_o p \wedge p <_o r$ **using** 2 **by** $blast$

next

assume $\exists p. Field\ p < Field\ r \wedge r' =_o p \wedge p <_o r$

thus $r' <_o r$ **using** $ordIso\text{-ordLess}\text{-trans}$ **by** $blast$

qed

lemma $internalize\text{-ordLeq}$:

$$(r' \leq_o r) = (\exists p. Field\ p \leq Field\ r \wedge r' =_o p \wedge p \leq_o r)$$

proof

assume $*$: $r' \leq_o r$

moreover

have $r' <_o r \implies \exists p. Field\ p \leq Field\ r \wedge r' =_o p \wedge p \leq_o r$

using $ordLeq\text{-iff}\text{-ordLess}\text{-or}\text{-ordIso}\ internalize\text{-ordLess}[of\ r'\ r]$ **by** $blast$

moreover

have $r \leq_o r$ **using** $*$ $ordLeq\text{-def}\ ordLeq\text{-reflexive}$ **by** $blast$

ultimately show $\exists p. Field\ p \leq Field\ r \wedge r' =_o p \wedge p \leq_o r$

using $ordLeq\text{-iff}\text{-ordLess}\text{-or}\text{-ordIso}$ **by** $blast$

next

assume $\exists p. Field\ p \leq Field\ r \wedge r' =_o p \wedge p \leq_o r$

thus $r' \leq_o r$ **using** $ordIso\text{-ordLeq}\text{-trans}$ **by** $blast$

qed

lemma $ordLeq\text{-iff}\text{-ordLess}\text{-Restr}$:

assumes *WELL*: Well-order r **and** *WELL'*: Well-order r'
shows $(r \leq_o r') = (\forall a \in \text{Field } r. \text{Restr } r (\text{underS } r a) <_o r')$
proof *safe*
assume *: $r \leq_o r'$
fix a **assume** $a \in \text{Field } r$
hence $\text{Restr } r (\text{underS } r a) <_o r$
using *WELL* *underS-Restr-ordLess*[of r] **by** *blast*
thus $\text{Restr } r (\text{underS } r a) <_o r'$
using * *ordLess-ordLeq-trans* **by** *blast*
next
assume *: $\forall a \in \text{Field } r. \text{Restr } r (\text{underS } r a) <_o r'$
{**assume** $r' <_o r$
then obtain a **where** $a \in \text{Field } r \wedge r' =_o \text{Restr } r (\text{underS } r a)$
using *assms ordLess-iff-ordIso-Restr* **by** *blast*
hence *False* **using** * *not-ordLess-ordIso ordIso-symmetric* **by** *blast*
}
thus $r \leq_o r'$ **using** *ordLess-or-ordLeq assms* **by** *blast*
qed

lemma *finite-ordLess-infinite*:
assumes *WELL*: Well-order r **and** *WELL'*: Well-order r' **and**
FIN: *finite*(*Field* r) **and** *INF*: \neg *finite*(*Field* r')
shows $r <_o r'$
proof –
{**assume** $r' \leq_o r$
then obtain h **where** $\text{inj-on } h (\text{Field } r') \wedge h ' (\text{Field } r') \leq \text{Field } r$
unfolding *ordLeq-def* **using** *assms embed-inj-on embed-Field* **by** *blast*
hence *False* **using** *finite-imageD finite-subset FIN INF* **by** *blast*
}
thus *?thesis* **using** *WELL WELL' ordLess-or-ordLeq* **by** *blast*
qed

lemma *finite-well-order-on-ordIso*:
assumes *FIN*: *finite* A **and**
WELL: *well-order-on* A r **and** *WELL'*: *well-order-on* A r'
shows $r =_o r'$
proof –
have 0 : $\text{Well-order } r \wedge \text{Well-order } r' \wedge \text{Field } r = A \wedge \text{Field } r' = A$
using *assms well-order-on-Well-order* **by** *blast*
moreover
have $\forall r r'. \text{well-order-on } A r \wedge \text{well-order-on } A r' \wedge r \leq_o r'$
 $\longrightarrow r =_o r'$
proof(*clarify*)
fix $r r'$ **assume** *: *well-order-on* A r **and** **: *well-order-on* A r'
have 2 : $\text{Well-order } r \wedge \text{Well-order } r' \wedge \text{Field } r = A \wedge \text{Field } r' = A$
using * ** *well-order-on-Well-order* **by** *blast*
assume $r \leq_o r'$
then obtain f **where** 1 : *embed* $r r' f$ **and**
 $\text{inj-on } f A \wedge f ' A \leq A$

```

unfolding ordLeq-def using 2 embed-inj-on embed-Field by blast
hence bij-betw f A A unfolding bij-betw-def using FIN endo-inj-surj by blast
thus r = o r' unfolding ordIso-def iso-def[abs-def] using 1 2 by auto
qed
ultimately show ?thesis using assms ordLeq-total ordIso-symmetric by blast
qed

```

29.5 $< o$ is well-founded

Of course, it only makes sense to state that the $< o$ is well-founded on the restricted type $'a \text{ rel } \text{rel}$. We prove this by first showing that, for any set of well-orders all embedded in a fixed well-order, the function mapping each well-order in the set to an order filter of the fixed well-order is compatible w.r.t. to $< o$ versus *strict inclusion*; and we already know that strict inclusion of order filters is well-founded.

definition *ord-to-filter* :: $'a \text{ rel} \Rightarrow 'a \text{ rel} \Rightarrow 'a \text{ set}$
where *ord-to-filter* r0 r $\equiv (\text{SOME } f. \text{ embed } r \text{ r0 } f) \text{ ' (Field } r)$

lemma *ord-to-filter-compat*:

```

compat (ordLess Int (ordLess-1 "{r0}" × ordLess-1 "{r0}"))
  (ofilterIncl r0)
  (ord-to-filter r0)

```

proof(*unfold compat-def ord-to-filter-def, clarify*)

```

fix r1::'a rel and r2::'a rel
let ?A1 = Field r1 let ?A2 = Field r2 let ?A0 = Field r0
let ?phi10 = λ f10. embed r1 r0 f10 let ?f10 = SOME f. ?phi10 f
let ?phi20 = λ f20. embed r2 r0 f20 let ?f20 = SOME f. ?phi20 f
assume *: r1 < o r0 r2 < o r0 and **: r1 < o r2
hence (∃ f. ?phi10 f) ∧ (∃ f. ?phi20 f)
by (auto simp add: ordLess-def embedS-def)
hence ?phi10 ?f10 ∧ ?phi20 ?f20 by (auto simp add: someI-ex)
thus (?f10 ' ?A1, ?f20 ' ?A2) ∈ ofilterIncl r0
using * ** by (simp add: embed-ordLess-ofilterIncl)
qed

```

theorem *wf-ordLess*: wf ordLess

proof–

```

{ fix r0 :: ('a × 'a) set

```

```

  let ?ordLess = ordLess::('d rel * 'd rel) set
  let ?R = ?ordLess Int (?ordLess-1 "{r0}" × ?ordLess-1 "{r0}")
  { assume Case1: Well-order r0
    hence wf ?R
    using wf-ofilterIncl[of r0]
      compat-wf[of ?R ofilterIncl r0 ord-to-filter r0]
      ord-to-filter-compat[of r0] by auto
  }
moreover

```

```

{assume Case2: ¬ Well-order r0
  hence ?R = {} unfolding ordLess-def by auto
  hence wf ?R using wf-empty by simp
}
ultimately have wf ?R by blast
}
thus ?thesis by (simp add: trans-wf-iff ordLess-trans)
qed

```

corollary *exists-minim-Well-order:*

assumes *NE*: $R \neq \{\}$ and *WELL*: $\forall r \in R. \text{Well-order } r$
 shows $\exists r \in R. \forall r' \in R. r \leq_o r'$

proof –

```

obtain r where r ∈ R ∧ (∀ r' ∈ R. ¬ r' <_o r)
  using NE spec[OF spec[OF subst[OF wf-eq-minimal, of %x. x, OF wf-ordLess]],
of - R]
  equalsOI[of R] by blast
with not-ordLeq-iff-ordLess WELL show ?thesis by blast
qed

```

29.6 Copy via direct images

The direct image operator is the dual of the inverse image operator *inv-image* from *Relation.thy*. It is useful for transporting a well-order between different types.

definition *dir-image* :: $'a \text{ rel} \Rightarrow ('a \Rightarrow 'a') \Rightarrow 'a' \text{ rel}$

where

$$\text{dir-image } r \ f = \{(f \ a, \ f \ b) \mid a \ b. (a, b) \in r\}$$

lemma *dir-image-Field:*

$$\text{Field}(\text{dir-image } r \ f) = f \ ' \ (\text{Field } r)$$

unfolding *dir-image-def Field-def Range-def Domain-def* **by** *fast*

lemma *dir-image-minus-Id:*

$$\text{inj-on } f \ (\text{Field } r) \Longrightarrow (\text{dir-image } r \ f) - \text{Id} = \text{dir-image } (r - \text{Id}) \ f$$

unfolding *inj-on-def Field-def dir-image-def* **by** *auto*

lemma *Refl-dir-image:*

assumes *Refl* r

shows *Refl*(*dir-image* $r \ f$)

proof –

{**fix** $a' \ b'$

assume $(a', b') \in \text{dir-image } r \ f$

then obtain $a \ b$ where $1: a' = f \ a \wedge b' = f \ b \wedge (a, b) \in r$

unfolding *dir-image-def* **by** *blast*

hence $a \in \text{Field } r \wedge b \in \text{Field } r$ using *Field-def* **by** *fastforce*

hence $(a, a) \in r \wedge (b, b) \in r$ using *assms* **by** (*simp add: refl-on-def*)

with 1 have $(a', a') \in \text{dir-image } r \ f \wedge (b', b') \in \text{dir-image } r \ f$

unfolding *dir-image-def* **by** *auto*

```

}
thus ?thesis
  by(unfold refl-on-def Field-def Domain-def Range-def, auto)
qed

```

lemma *trans-dir-image*:

```

assumes TRANS: trans r and INJ: inj-on f (Field r)
shows trans(dir-image r f)
unfolding trans-def
proof safe
  fix a' b' c'
  assume  $(a',b') \in \text{dir-image } r \ f$   $(b',c') \in \text{dir-image } r \ f$ 
  then obtain a b1 b2 c where  $1: a' = f \ a \wedge b' = f \ b1 \wedge b' = f \ b2 \wedge c' = f \ c$ 
and
   $2: (a,b1) \in r \wedge (b2,c) \in r$ 
  unfolding dir-image-def by blast
  hence  $b1 \in \text{Field } r \wedge b2 \in \text{Field } r$ 
  unfolding Field-def by auto
  hence  $b1 = b2$  using  $1$  INJ unfolding inj-on-def by auto
  hence  $(a,c) \in r$  using  $2$  TRANS unfolding trans-def by blast
  thus  $(a',c') \in \text{dir-image } r \ f$ 
  unfolding dir-image-def using  $1$  by auto
qed

```

lemma *Preorder-dir-image*:

```

[[Preorder r; inj-on f (Field r)]]  $\implies$  Preorder (dir-image r f)
by (simp add: preorder-on-def Refl-dir-image trans-dir-image)

```

lemma *antisym-dir-image*:

```

assumes AN: antisym r and INJ: inj-on f (Field r)
shows antisym(dir-image r f)
unfolding antisym-def
proof safe
  fix a' b'
  assume  $(a',b') \in \text{dir-image } r \ f$   $(b',a') \in \text{dir-image } r \ f$ 
  then obtain a1 b1 a2 b2 where  $1: a' = f \ a1 \wedge a' = f \ a2 \wedge b' = f \ b1 \wedge b' = f \ b2$ 
and
   $2: (a1,b1) \in r \wedge (b2,a2) \in r$  and
   $3: \{a1,a2,b1,b2\} \leq \text{Field } r$ 
  unfolding dir-image-def Field-def by blast
  hence  $a1 = a2 \wedge b1 = b2$  using INJ unfolding inj-on-def by auto
  hence  $a1 = b2$  using  $2$  AN unfolding antisym-def by auto
  thus  $a' = b'$  using  $1$  by auto
qed

```

lemma *Partial-order-dir-image*:

```

[[Partial-order r; inj-on f (Field r)]]  $\implies$  Partial-order (dir-image r f)
by (simp add: partial-order-on-def Preorder-dir-image antisym-dir-image)

```


lemma *Total-dir-image*:
assumes *TOT*: *Total r* **and** *INJ*: *inj-on f (Field r)*
shows *Total(dir-image r f)*
proof(*unfold total-on-def, intro ballI impI*)
fix $a' b'$
assume $a' \in \text{Field } (\text{dir-image } r \ f)$ $b' \in \text{Field } (\text{dir-image } r \ f)$
then obtain a **and** b **where** $1: a \in \text{Field } r \wedge b \in \text{Field } r \wedge f a = a' \wedge f b = b'$
unfolding *dir-image-Field[of r f]* **by** *blast*
moreover assume $a' \neq b'$
ultimately have $a \neq b$ **using** *INJ* **unfolding** *inj-on-def* **by** *auto*
hence $(a,b) \in r \vee (b,a) \in r$ **using** 1 *TOT* **unfolding** *total-on-def* **by** *auto*
thus $(a',b') \in \text{dir-image } r \ f \vee (b',a') \in \text{dir-image } r \ f$
using 1 **unfolding** *dir-image-def* **by** *auto*
qed

lemma *Linear-order-dir-image*:
 $\llbracket \text{Linear-order } r; \text{inj-on } f \text{ (Field } r) \rrbracket \implies \text{Linear-order } (\text{dir-image } r \ f)$
by (*simp add: linear-order-on-def Partial-order-dir-image Total-dir-image*)

lemma *wf-dir-image*:
assumes *WF*: *wf r* **and** *INJ*: *inj-on f (Field r)*
shows *wf(dir-image r f)*
proof(*unfold wf-eq-minimal2, intro allI impI, elim conjE*)
fix $A'::'b \text{ set}$
assume *SUB*: $A' \leq \text{Field}(\text{dir-image } r \ f)$ **and** *NE*: $A' \neq \{\}$
obtain A **where** *A-def*: $A = \{a \in \text{Field } r. f a \in A'\}$ **by** *blast*
have $A \neq \{\} \wedge A \leq \text{Field } r$ **using** *A-def SUB NE* **by** (*auto simp: dir-image-Field*)
then obtain a **where** $1: a \in A \wedge (\forall b \in A. (b,a) \notin r)$
using *spec[OF WF[unfolded wf-eq-minimal2], of A]* **by** *blast*
have $\forall b' \in A'. (b',f a) \notin \text{dir-image } r \ f$
proof(*clarify*)
fix b' **assume** $*$: $b' \in A'$ **and** $**$: $(b',f a) \in \text{dir-image } r \ f$
obtain $b1 \ a1$ **where** $2: b' = f b1 \wedge f a = f a1$ **and**
 $3: (b1,a1) \in r \wedge \{a1,b1\} \leq \text{Field } r$
using $**$ **unfolding** *dir-image-def Field-def* **by** *blast*
hence $a = a1$ **using** 1 *A-def INJ* **unfolding** *inj-on-def* **by** *auto*
hence $b1 \in A \wedge (b1,a) \in r$ **using** $2 \ 3$ *A-def ** **by** *auto*
with 1 **show** *False* **by** *auto*
qed
thus $\exists a' \in A'. \forall b' \in A'. (b', a') \notin \text{dir-image } r \ f$
using *A-def 1* **by** *blast*
qed

lemma *Well-order-dir-image*:
 $\llbracket \text{Well-order } r; \text{inj-on } f \text{ (Field } r) \rrbracket \implies \text{Well-order } (\text{dir-image } r \ f)$
unfolding *well-order-on-def*
using *Linear-order-dir-image[of r f] wf-dir-image[of r - Id f]*
dir-image-minus-Id[of f r]
subset-inj-on[of f Field r Field(r - Id)]

mono-Field[of $r - Id\ r$] **by auto**

lemma *dir-image-bij-betw*:

$\llbracket inj\text{-on } f \text{ (Field } r) \rrbracket \implies bij\text{-betw } f \text{ (Field } r) \text{ (Field (dir-image } r\ f))$
unfolding *bij-betw-def* **by** (*simp add: dir-image-Field order-on-defs*)

lemma *dir-image-compat*:

compat $r \text{ (dir-image } r\ f) \text{ } f$
unfolding *compat-def dir-image-def* **by auto**

lemma *dir-image-iso*:

$\llbracket Well\text{-order } r; inj\text{-on } f \text{ (Field } r) \rrbracket \implies iso\ r \text{ (dir-image } r\ f) \text{ } f$
using *iso-iff3 dir-image-compat dir-image-bij-betw Well-order-dir-image* **by blast**

lemma *dir-image-ordIso*:

$\llbracket Well\text{-order } r; inj\text{-on } f \text{ (Field } r) \rrbracket \implies r =_o \text{ dir-image } r\ f$
unfolding *ordIso-def* **using** *dir-image-iso Well-order-dir-image* **by blast**

lemma *Well-order-iso-copy*:

assumes *WELL: well-order-on* $A\ r$ **and** *BIJ: bij-betw* $f\ A\ A'$
shows $\exists r'. \text{ well-order-on } A'\ r' \wedge r =_o r'$

proof –

let $?r' = \text{dir-image } r\ f$
have $1: A = \text{Field } r \wedge \text{Well-order } r$
using *WELL well-order-on-Well-order* **by blast**
hence $2: iso\ r\ ?r'\ f$
using *dir-image-iso* **using** *BIJ unfolding bij-betw-def* **by auto**
hence $f' \text{ (Field } r) = \text{Field } ?r'$ **using** 1 *iso-iff[of r ?r']* **by blast**
hence $\text{Field } ?r' = A'$
using 1 *BIJ unfolding bij-betw-def* **by auto**
moreover **have** *Well-order* $?r'$
using 1 *Well-order-dir-image BIJ unfolding bij-betw-def* **by blast**
ultimately show *?thesis* **unfolding** *ordIso-def* **using** $1\ 2$ **by blast**

qed

29.7 Bounded square

This construction essentially defines, for an order relation r , a lexicographic order *bsqr* r on $(\text{Field } r) \times (\text{Field } r)$, applying the following criteria (in this order):

- compare the maximums;
- compare the first components;
- compare the second components.

The only application of this construction that we are aware of is at proving that the square of an infinite set has the same cardinal as that set. The

essential property required there (and which is ensured by this construction) is that any proper order filter of the product order is included in a rectangle, i.e., in a product of proper filters on the original relation (assumed to be a well-order).

definition $bsqr :: 'a\ rel \Rightarrow ('a * 'a)\ rel$

where

$$bsqr\ r = \{((a1,a2),(b1,b2)). \\ \{a1,a2,b1,b2\} \leq Field\ r \wedge \\ (a1 = b1 \wedge a2 = b2 \vee \\ (wo-rel.max2\ r\ a1\ a2, wo-rel.max2\ r\ b1\ b2) \in r - Id \vee \\ wo-rel.max2\ r\ a1\ a2 = wo-rel.max2\ r\ b1\ b2 \wedge (a1,b1) \in r - Id \vee \\ wo-rel.max2\ r\ a1\ a2 = wo-rel.max2\ r\ b1\ b2 \wedge a1 = b1 \wedge (a2,b2) \in r \\ - Id \\)\}$$

lemma *Field-bsqr*:

$Field\ (bsqr\ r) = Field\ r \times Field\ r$

proof

show $Field\ (bsqr\ r) \leq Field\ r \times Field\ r$

proof–

{fix $a1\ a2$ **assume** $(a1,a2) \in Field\ (bsqr\ r)$

moreover

have $\bigwedge b1\ b2. ((a1,a2),(b1,b2)) \in bsqr\ r \vee ((b1,b2),(a1,a2)) \in bsqr\ r \implies \\ a1 \in Field\ r \wedge a2 \in Field\ r$ **unfolding** *bsqr-def* **by** *auto*

ultimately have $a1 \in Field\ r \wedge a2 \in Field\ r$ **unfolding** *Field-def* **by** *auto*

}

thus *?thesis* **unfolding** *Field-def* **by** *force*

qed

next

show $Field\ r \times Field\ r \leq Field\ (bsqr\ r)$

proof *safe*

fix $a1\ a2$ **assume** $a1 \in Field\ r$ **and** $a2 \in Field\ r$

hence $((a1,a2),(a1,a2)) \in bsqr\ r$ **unfolding** *bsqr-def* **by** *blast*

thus $(a1,a2) \in Field\ (bsqr\ r)$ **unfolding** *Field-def* **by** *auto*

qed

qed

lemma *bsqr-Refl*: $Refl\ (bsqr\ r)$

by $(unfold\ refl-on-def\ Field-bsqr, auto\ simp\ add: bsqr-def)$

lemma *bsqr-Trans*:

assumes *Well-order* r

shows $trans\ (bsqr\ r)$

unfolding *trans-def*

proof *safe*

have *Well*: $wo-rel\ r$ **using** *assms* $wo-rel-def$ **by** *auto*

hence *Trans*: $trans\ r$ **using** $wo-rel.TRANS$ **by** *auto*

have *Anti*: $antisym\ r$ **using** $wo-rel.ANTISYM$ *Well* **by** *auto*

hence *TransS*: $\text{trans}(r - \text{Id})$ using *Trans* by (*simp add: trans-diff-Id*)

fix $a1\ a2\ b1\ b2\ c1\ c2$
 assume *: $((a1, a2), (b1, b2)) \in \text{bsqr } r$ and **: $((b1, b2), (c1, c2)) \in \text{bsqr } r$
 hence 0: $\{a1, a2, b1, b2, c1, c2\} \leq \text{Field } r$ unfolding *bsqr-def* by *auto*
 have 1: $a1 = b1 \wedge a2 = b2 \vee (\text{wo-rel.max2 } r\ a1\ a2, \text{wo-rel.max2 } r\ b1\ b2) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ a1\ a2 = \text{wo-rel.max2 } r\ b1\ b2 \wedge (a1, b1) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ a1\ a2 = \text{wo-rel.max2 } r\ b1\ b2 \wedge a1 = b1 \wedge (a2, b2) \in r - \text{Id}$
 using * unfolding *bsqr-def* by *auto*
 have 2: $b1 = c1 \wedge b2 = c2 \vee (\text{wo-rel.max2 } r\ b1\ b2, \text{wo-rel.max2 } r\ c1\ c2) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ b1\ b2 = \text{wo-rel.max2 } r\ c1\ c2 \wedge (b1, c1) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ b1\ b2 = \text{wo-rel.max2 } r\ c1\ c2 \wedge b1 = c1 \wedge (b2, c2) \in r - \text{Id}$
 using ** unfolding *bsqr-def* by *auto*
 show $((a1, a2), (c1, c2)) \in \text{bsqr } r$
 proof –
 {assume *Case1*: $a1 = b1 \wedge a2 = b2$
 hence ?thesis using ** by *simp*
 }
 moreover
 {assume *Case2*: $(\text{wo-rel.max2 } r\ a1\ a2, \text{wo-rel.max2 } r\ b1\ b2) \in r - \text{Id}$
 {assume *Case21*: $b1 = c1 \wedge b2 = c2$
 hence ?thesis using * by *simp*
 }
 moreover
 {assume *Case22*: $(\text{wo-rel.max2 } r\ b1\ b2, \text{wo-rel.max2 } r\ c1\ c2) \in r - \text{Id}$
 hence $(\text{wo-rel.max2 } r\ a1\ a2, \text{wo-rel.max2 } r\ c1\ c2) \in r - \text{Id}$
 using *Case2 TransS trans-def*[of $r - \text{Id}$] by *blast*
 hence ?thesis using 0 unfolding *bsqr-def* by *auto*
 }
 moreover
 {assume *Case23-4*: $\text{wo-rel.max2 } r\ b1\ b2 = \text{wo-rel.max2 } r\ c1\ c2$
 hence ?thesis using *Case2* 0 unfolding *bsqr-def* by *auto*
 }
 ultimately have ?thesis using 0 2 by *auto*
 }
 moreover
 {assume *Case3*: $\text{wo-rel.max2 } r\ a1\ a2 = \text{wo-rel.max2 } r\ b1\ b2 \wedge (a1, b1) \in r - \text{Id}$
 {assume *Case31*: $b1 = c1 \wedge b2 = c2$
 hence ?thesis using * by *simp*
 }
 moreover
 {assume *Case32*: $(\text{wo-rel.max2 } r\ b1\ b2, \text{wo-rel.max2 } r\ c1\ c2) \in r - \text{Id}$
 hence ?thesis using *Case3* 0 unfolding *bsqr-def* by *auto*
 }
 }

```

moreover
  {assume Case33: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2  $\wedge$  (b1,c1)  $\in$  r
- Id
  hence (a1,c1)  $\in$  r - Id
    using Case3 TransS trans-def[of r - Id] by blast
  hence ?thesis using Case3 Case33 0 unfolding bsqr-def by auto
  }
moreover
  {assume Case33: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2  $\wedge$  b1 = c1
  hence ?thesis using Case3 0 unfolding bsqr-def by auto
  }
ultimately have ?thesis using 0 2 by auto
}
moreover
  {assume Case4: wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2  $\wedge$  a1 = b1  $\wedge$ 
(a2,b2)  $\in$  r - Id
  {assume Case41: b1 = c1  $\wedge$  b2 = c2
  hence ?thesis using * by simp
  }
moreover
  {assume Case42: (wo-rel.max2 r b1 b2, wo-rel.max2 r c1 c2)  $\in$  r - Id
  hence ?thesis using Case4 0 unfolding bsqr-def by force
  }
moreover
  {assume Case43: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2  $\wedge$  (b1,c1)  $\in$  r
- Id
  hence ?thesis using Case4 0 unfolding bsqr-def by auto
  }
moreover
  {assume Case44: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2  $\wedge$  b1 = c1  $\wedge$ 
(b2,c2)  $\in$  r - Id
  hence (a2,c2)  $\in$  r - Id
    using Case4 TransS trans-def[of r - Id] by blast
  hence ?thesis using Case4 Case44 0 unfolding bsqr-def by auto
  }
ultimately have ?thesis using 0 2 by auto
}
ultimately show ?thesis using 0 1 by auto
qed
qed

```

lemma *bsqr-antisym*:

assumes *Well-order* *r*

shows *antisym* (*bsqr* *r*)

proof(*unfold antisym-def*, *clarify*)

have *Well*: *wo-rel* *r* **using** *assms* *wo-rel-def* **by** *auto*

hence *Trans*: *trans* *r* **using** *wo-rel.TRANS* **by** *auto*

have *Anti*: *antisym* *r* **using** *wo-rel.ANTISYM* *Well* **by** *auto*

hence *TransS*: $\text{trans}(r - \text{Id})$ **using** *Trans* **by** (*simp add: trans-diff-Id*)
hence *IrrS*: $\forall a b. \neg((a,b) \in r - \text{Id} \wedge (b,a) \in r - \text{Id})$
using *Anti trans-def*[of $r - \text{Id}$] *antisym-def*[of $r - \text{Id}$] **by** *blast*

fix $a1\ a2\ b1\ b2$
assume *: $((a1,a2),(b1,b2)) \in \text{bsqr } r$ **and** **: $((b1,b2),(a1,a2)) \in \text{bsqr } r$
hence $\{a1,a2,b1,b2\} \leq \text{Field } r$ **unfolding** *bsqr-def* **by** *auto*
moreover
have $a1 = b1 \wedge a2 = b2 \vee (\text{wo-rel.max2 } r\ a1\ a2, \text{wo-rel.max2 } r\ b1\ b2) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ a1\ a2 = \text{wo-rel.max2 } r\ b1\ b2 \wedge (a1,b1) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ a1\ a2 = \text{wo-rel.max2 } r\ b1\ b2 \wedge a1 = b1 \wedge (a2,b2) \in r - \text{Id}$
using * **unfolding** *bsqr-def* **by** *auto*
moreover
have $b1 = a1 \wedge b2 = a2 \vee (\text{wo-rel.max2 } r\ b1\ b2, \text{wo-rel.max2 } r\ a1\ a2) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ b1\ b2 = \text{wo-rel.max2 } r\ a1\ a2 \wedge (b1,a1) \in r - \text{Id} \vee$
 $\text{wo-rel.max2 } r\ b1\ b2 = \text{wo-rel.max2 } r\ a1\ a2 \wedge b1 = a1 \wedge (b2,a2) \in r - \text{Id}$
using ** **unfolding** *bsqr-def* **by** *auto*
ultimately show $a1 = b1 \wedge a2 = b2$
using *IrrS* **by** *auto*
qed

lemma *bsqr-Total*:
assumes *Well-order r*
shows *Total(bsqr r)*
proof –

have *Well*: *wo-rel r* **using** *assms wo-rel-def* **by** *auto*
hence *Total*: $\forall a \in \text{Field } r. \forall b \in \text{Field } r. (a,b) \in r \vee (b,a) \in r$
using *wo-rel.TOTALS* **by** *auto*

{fix $a1\ a2\ b1\ b2$ **assume** $\{(a1,a2), (b1,b2)\} \leq \text{Field}(\text{bsqr } r)$
hence *0*: $a1 \in \text{Field } r \wedge a2 \in \text{Field } r \wedge b1 \in \text{Field } r \wedge b2 \in \text{Field } r$
using *Field-bsqr* **by** *blast*
have $((a1,a2) = (b1,b2) \vee ((a1,a2),(b1,b2)) \in \text{bsqr } r \vee ((b1,b2),(a1,a2)) \in \text{bsqr } r)$
proof(*rule wo-rel.cases-Total*[of $r\ a1\ a2$], *clarsimp simp add: Well, simp add: 0*)

assume *Case1*: $(a1,a2) \in r$
hence *1*: $\text{wo-rel.max2 } r\ a1\ a2 = a2$
using *Well 0* **by** (*simp add: wo-rel.max2-equals2*)
show *?thesis*
proof(*rule wo-rel.cases-Total*[of $r\ b1\ b2$], *clarsimp simp add: Well, simp add: 0*)

assume *Case11*: $(b1,b2) \in r$

```

hence 2: wo-rel.max2 r b1 b2 = b2
  using Well 0 by (simp add: wo-rel.max2-equals2)
show ?thesis
proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
  assume Case111: (a2,b2) ∈ r - Id ∨ (b2,a2) ∈ r - Id
  thus ?thesis using 0 1 2 unfolding bsqr-def by auto
next
  assume Case112: a2 = b2
  show ?thesis
proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
  assume Case1121: (a1,b1) ∈ r - Id ∨ (b1,a1) ∈ r - Id
  thus ?thesis using 0 1 2 Case112 unfolding bsqr-def by auto
next
  assume Case1122: a1 = b1
  thus ?thesis using Case112 by auto
qed
qed
next
  assume Case12: (b2,b1) ∈ r
hence 3: wo-rel.max2 r b1 b2 = b1 using Well 0 by (simp add: wo-rel.max2-equals1)
show ?thesis
proof(rule wo-rel.cases-Total3[of r a2 b1], clarsimp simp add: Well, simp
add: 0)
  assume Case121: (a2,b1) ∈ r - Id ∨ (b1,a2) ∈ r - Id
  thus ?thesis using 0 1 3 unfolding bsqr-def by auto
next
  assume Case122: a2 = b1
  show ?thesis
proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
  assume Case1221: (a1,b1) ∈ r - Id ∨ (b1,a1) ∈ r - Id
  thus ?thesis using 0 1 3 Case122 unfolding bsqr-def by auto
next
  assume Case1222: a1 = b1
  show ?thesis
proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
  assume Case12221: (a2,b2) ∈ r - Id ∨ (b2,a2) ∈ r - Id
  thus ?thesis using 0 1 3 Case122 Case1222 unfolding bsqr-def by
auto
next
  assume Case12222: a2 = b2
  thus ?thesis using Case122 Case1222 by auto
qed
qed
qed
qed

```

```

next
  assume Case2: (a2,a1) ∈ r
  hence 1: wo-rel.max2 r a1 a2 = a1 using Well 0 by (simp add: wo-rel.max2-equals1)
  show ?thesis
  proof(rule wo-rel.cases-Total[of r b1 b2], clarsimp simp add: Well, simp add:
0)
    assume Case21: (b1,b2) ∈ r
    hence 2: wo-rel.max2 r b1 b2 = b2 using Well 0 by (simp add: wo-rel.max2-equals2)
    show ?thesis
    proof(rule wo-rel.cases-Total3[of r a1 b2], clarsimp simp add: Well, simp
add: 0)
      assume Case211: (a1,b2) ∈ r - Id ∨ (b2,a1) ∈ r - Id
      thus ?thesis using 0 1 2 unfolding bsqr-def by auto
    next
      assume Case212: a1 = b2
      show ?thesis
      proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
        assume Case2121: (a1,b1) ∈ r - Id ∨ (b1,a1) ∈ r - Id
        thus ?thesis using 0 1 2 Case212 unfolding bsqr-def by auto
      next
        assume Case2122: a1 = b1
        show ?thesis
        proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
          assume Case21221: (a2,b2) ∈ r - Id ∨ (b2,a2) ∈ r - Id
          thus ?thesis using 0 1 2 Case212 Case2122 unfolding bsqr-def by
auto
        next
          assume Case21222: a2 = b2
          thus ?thesis using Case2122 Case212 by auto
        qed
      qed
    qed
  next
  assume Case22: (b2,b1) ∈ r
  hence 3: wo-rel.max2 r b1 b2 = b1 using Well 0 by (simp add:
wo-rel.max2-equals1)
  show ?thesis
  proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
    assume Case221: (a1,b1) ∈ r - Id ∨ (b1,a1) ∈ r - Id
    thus ?thesis using 0 1 3 unfolding bsqr-def by auto
  next
    assume Case222: a1 = b1
    show ?thesis
    proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
      assume Case2221: (a2,b2) ∈ r - Id ∨ (b2,a2) ∈ r - Id

```



```

      thus ?thesis using 0 1 3 Case222 unfolding bsqr-def by auto
    next
      assume Case2222: a2 = b2
      thus ?thesis using Case222 by auto
    qed
  qed
  qed
  qed
}
thus ?thesis unfolding total-on-def by fast
qed

```

lemma *bsqr-Linear-order*:
 assumes *Well-order r*
 shows *Linear-order(bsqr r)*
 unfolding *order-on-defs*
 using *assms bsqr-Refl bsqr-Trans bsqr-antisym bsqr-Total* by *blast*

lemma *bsqr-Well-order*:
 assumes *Well-order r*
 shows *Well-order(bsqr r)*
 using *assms*
proof(*simp add: bsqr-Linear-order Linear-order-Well-order-iff, intro allI impI*)
 have $0: \forall A \leq \text{Field } r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r)$
 using *assms well-order-on-def Linear-order-Well-order-iff* by *blast*
 fix *D* assume *: $D \leq \text{Field } (bsqr\ r)$ and **: $D \neq \{\}$
 hence $1: D \leq \text{Field } r \times \text{Field } r$ unfolding *Field-bsqr* by *simp*

obtain *M* where *M-def*: $M = \{wo\text{-rel.max2 } r\ a1\ a2 \mid a1\ a2. (a1, a2) \in D\}$ by
blast
 have $M \neq \{\}$ using $1\ M\text{-def}\ **$ by *auto*
 moreover
 have $M \leq \text{Field } r$ unfolding *M-def*
 using $1\ assms\ wo\text{-rel-def}[of\ r]\ wo\text{-rel.max2-among}[of\ r]$ by *fastforce*
 ultimately obtain *m* where *m-min*: $m \in M \wedge (\forall a \in M. (m, a) \in r)$
 using 0 by *blast*

obtain *A1* where *A1-def*: $A1 = \{a1. \exists a2. (a1, a2) \in D \wedge wo\text{-rel.max2 } r\ a1\ a2 = m\}$ by *blast*
 have $A1 \leq \text{Field } r$ unfolding *A1-def* using 1 by *auto*
 moreover have $A1 \neq \{\}$ unfolding *A1-def* using *m-min* unfolding *M-def* by
blast
 ultimately obtain *a1* where *a1-min*: $a1 \in A1 \wedge (\forall a \in A1. (a1, a) \in r)$
 using 0 by *blast*

obtain *A2* where *A2-def*: $A2 = \{a2. (a1, a2) \in D \wedge wo\text{-rel.max2 } r\ a1\ a2 = m\}$ by *blast*
 have $A2 \leq \text{Field } r$ unfolding *A2-def* using 1 by *auto*
 moreover have $A2 \neq \{\}$ unfolding *A2-def*

using *m-min a1-min unfolding A1-def M-def by blast*
ultimately obtain *a2* where *a2-min*: $a2 \in A2 \wedge (\forall a \in A2. (a2, a) \in r)$
using *0* by *blast*

have *2*: $wo-rel.max2\ r\ a1\ a2 = m$
using *a1-min a2-min unfolding A1-def A2-def by auto*
have *3*: $(a1, a2) \in D$ using *a2-min unfolding A2-def by auto*

moreover

{fix *b1 b2* assume *****: $(b1, b2) \in D$
hence *4*: $\{a1, a2, b1, b2\} \leq Field\ r$ using *1 3* by *blast*
have *5*: $(wo-rel.max2\ r\ a1\ a2, wo-rel.max2\ r\ b1\ b2) \in r$
using **** a1-min a2-min m-min unfolding A1-def A2-def M-def by auto*
have $((a1, a2), (b1, b2)) \in bsqr\ r$
proof(cases $wo-rel.max2\ r\ a1\ a2 = wo-rel.max2\ r\ b1\ b2$)
assume *Case1*: $wo-rel.max2\ r\ a1\ a2 \neq wo-rel.max2\ r\ b1\ b2$
thus *?thesis* unfolding *bsqr-def* using *4 5* by *auto*
next
assume *Case2*: $wo-rel.max2\ r\ a1\ a2 = wo-rel.max2\ r\ b1\ b2$
hence *b1* $\in A1$ unfolding *A1-def* using *2 **** by *auto*
hence *6*: $(a1, b1) \in r$ using *a1-min* by *auto*
show *?thesis*
proof(cases $a1 = b1$)
assume *Case21*: $a1 \neq b1$
thus *?thesis* unfolding *bsqr-def* using *4 Case2 6* by *auto*
next
assume *Case22*: $a1 = b1$
hence *b2* $\in A2$ unfolding *A2-def* using *2 *** Case2* by *auto*
hence *7*: $(a2, b2) \in r$ using *a2-min* by *auto*
thus *?thesis* unfolding *bsqr-def* using *4 7 Case2 Case22* by *auto*
qed
qed
}

ultimately show $\exists d \in D. \forall d' \in D. (d, d') \in bsqr\ r$ by *fastforce*
qed

lemma *bsqr-max2*:

assumes *WELL*: *Well-order r* and *LEQ*: $((a1, a2), (b1, b2)) \in bsqr\ r$
shows $(wo-rel.max2\ r\ a1\ a2, wo-rel.max2\ r\ b1\ b2) \in r$
proof –
have $\{(a1, a2), (b1, b2)\} \leq Field(bsqr\ r)$
using *LEQ unfolding Field-def by auto*
hence $\{a1, a2, b1, b2\} \leq Field\ r$ unfolding *Field-bsqr* by *auto*
hence $\{wo-rel.max2\ r\ a1\ a2, wo-rel.max2\ r\ b1\ b2\} \leq Field\ r$
using *WELL wo-rel-def[of r] wo-rel.max2-among[of r]* by *fastforce*
moreover have $(wo-rel.max2\ r\ a1\ a2, wo-rel.max2\ r\ b1\ b2) \in r \vee wo-rel.max2\ r\ a1\ a2 = wo-rel.max2\ r\ b1\ b2$
using *LEQ unfolding bsqr-def by auto*

ultimately show *?thesis* using **WELL unfolding order-on-defs refl-on-def** by *auto*
qed

lemma *bsqr-ofilter*:

assumes *WELL*: *Well-order r* **and**

OF: *wo-rel.ofilter (bsqr r) D* **and** *SUB*: $D < \text{Field } r \times \text{Field } r$ **and**

NE: $\neg (\exists a. \text{Field } r = \text{under } r a)$

shows $\exists A. \text{wo-rel.ofilter } r A \wedge A < \text{Field } r \wedge D \leq A \times A$

proof –

let $?r' = \text{bsqr } r$

have *Well*: *wo-rel r* using **WELL wo-rel-def** by *blast*

hence *Trans*: *trans r* using *wo-rel.TRANS* by *blast*

have *Well'*: *Well-order ?r' \wedge wo-rel ?r'*

using **WELL bsqr-Well-order wo-rel-def** by *blast*

have $D < \text{Field } ?r'$ **unfolding** *Field-bsqr* using *SUB* .

with *OF* **obtain** *a1* **and** *a2* **where**

$(a1, a2) \in \text{Field } ?r'$ **and** $1: D = \text{underS } ?r' (a1, a2)$

using *Well' wo-rel.ofilter-underS-Field[of ?r' D]* by *auto*

hence $2: \{a1, a2\} \leq \text{Field } r$ **unfolding** *Field-bsqr* by *auto*

let $?m = \text{wo-rel.max2 } r a1 a2$

have $D \leq (\text{under } r ?m) \times (\text{under } r ?m)$

proof(*unfold 1*)

{**fix** *b1 b2*

let $?n = \text{wo-rel.max2 } r b1 b2$

assume $(b1, b2) \in \text{underS } ?r' (a1, a2)$

hence $3: ((b1, b2), (a1, a2)) \in ?r'$

unfolding *underS-def* by *blast*

hence $(?n, ?m) \in r$ using **WELL** by (*simp add: bsqr-max2*)

moreover

{**have** $(b1, b2) \in \text{Field } ?r'$ using 3 **unfolding** *Field-def* by *auto*

hence $\{b1, b2\} \leq \text{Field } r$ **unfolding** *Field-bsqr* by *auto*

hence $(b1, ?n) \in r \wedge (b2, ?n) \in r$

using *Well* by (*simp add: wo-rel.max2-greater*)

}

ultimately have $(b1, ?m) \in r \wedge (b2, ?m) \in r$

using *Trans trans-def[of r]* by *blast*

hence $(b1, b2) \in (\text{under } r ?m) \times (\text{under } r ?m)$ **unfolding** *under-def* by

simp}

thus $\text{underS } ?r' (a1, a2) \leq (\text{under } r ?m) \times (\text{under } r ?m)$ by *auto*

qed

moreover have *wo-rel.ofilter r (under r ?m)*

using *Well* by (*simp add: wo-rel.under-ofilter*)

moreover have $\text{under } r ?m < \text{Field } r$

using *NE under-Field[of r ?m]* by *blast*

ultimately show *?thesis* by *blast*

qed

definition *Func* where

$Func\ A\ B = \{f . (\forall a \in A. f\ a \in B) \wedge (\forall a. a \notin A \longrightarrow f\ a = \text{undefined})\}$

lemma *Func-empty*:

$Func\ \{\}\ B = \{\lambda x. \text{undefined}\}$

unfolding *Func-def* by *auto*

lemma *Func-elim*:

assumes $g \in Func\ A\ B$ and $a \in A$

shows $\exists b. b \in B \wedge g\ a = b$

using *assms* **unfolding** *Func-def* by (*cases* $g\ a = \text{undefined}$) *auto*

definition *curr* where

$curr\ A\ f \equiv \lambda a. \text{if } a \in A \text{ then } \lambda b. f\ (a,b) \text{ else } \text{undefined}$

lemma *curr-in*:

assumes $f: f \in Func\ (A \times B)\ C$

shows $curr\ A\ f \in Func\ A\ (Func\ B\ C)$

using *assms* **unfolding** *curr-def* *Func-def* by *auto*

lemma *curr-inj*:

assumes $f1 \in Func\ (A \times B)\ C$ and $f2 \in Func\ (A \times B)\ C$

shows $curr\ A\ f1 = curr\ A\ f2 \longleftrightarrow f1 = f2$

proof *safe*

assume $c: curr\ A\ f1 = curr\ A\ f2$

show $f1 = f2$

proof (*rule ext*, *clarify*)

fix $a\ b$ **show** $f1\ (a, b) = f2\ (a, b)$

proof (*cases* $(a,b) \in A \times B$)

case *False*

thus *?thesis* **using** *assms* **unfolding** *Func-def* by *auto*

next

case *True* **hence** $a: a \in A$ and $b: b \in B$ by *auto*

thus *?thesis*

using c **unfolding** *curr-def* *fun-eq-iff* by (*elim alle*[*of - a*]) *simp*

qed

qed

qed

lemma *curr-surj*:

assumes $g \in Func\ A\ (Func\ B\ C)$

shows $\exists f \in Func\ (A \times B)\ C. curr\ A\ f = g$

proof

let $?f = \lambda ab. \text{if } fst\ ab \in A \wedge snd\ ab \in B \text{ then } g\ (fst\ ab)\ (snd\ ab) \text{ else } \text{undefined}$

show $curr\ A\ ?f = g$

proof (*rule ext*)

fix a **show** $curr\ A\ ?f\ a = g\ a$

proof (*cases* $a \in A$)

case *False*

hence $g\ a = \text{undefined}$ **using** *assms* **unfolding** *Func-def* **by** *auto*
thus *?thesis* **unfolding** *curr-def* **using** *False* **by** *simp*
next
case *True*
obtain $g1$ **where** $g1 \in \text{Func } B\ C$ **and** $g\ a = g1$
using *assms* **using** *Func-elim[OF assms True]* **by** *blast*
thus *?thesis* **using** *True* **unfolding** *Func-def curr-def* **by** *auto*
qed
qed
show $?f \in \text{Func } (A \times B)\ C$ **using** *assms* **unfolding** *Func-def mem-Collect-eq*
by *auto*
qed

lemma *bij-betw-curr*:
 $\text{bij-betw } (\text{curr } A)\ (\text{Func } (A \times B)\ C)\ (\text{Func } A\ (\text{Func } B\ C))$
unfolding *bij-betw-def inj-on-def*
using *curr-surj curr-in curr-inj* **by** *blast*

definition *Func-map* **where**
 $\text{Func-map } B2\ f1\ f2\ g\ b2 \equiv \text{if } b2 \in B2 \text{ then } f1\ (g\ (f2\ b2)) \text{ else undefined}$

lemma *Func-map*:
assumes $g: g \in \text{Func } A2\ A1$ **and** $f1: f1\ 'A1 \subseteq B1$ **and** $f2: f2\ 'B2 \subseteq A2$
shows $\text{Func-map } B2\ f1\ f2\ g \in \text{Func } B2\ B1$
using *assms* **unfolding** *Func-def Func-map-def mem-Collect-eq* **by** *auto*

lemma *Func-non-emp*:
assumes $B \neq \{\}$
shows $\text{Func } A\ B \neq \{\}$
proof–
obtain b **where** $b: b \in B$ **using** *assms* **by** *auto*
hence $(\lambda a. \text{if } a \in A \text{ then } b \text{ else undefined}) \in \text{Func } A\ B$ **unfolding** *Func-def* **by**
auto
thus *?thesis* **by** *blast*
qed

lemma *Func-is-emp*:
 $\text{Func } A\ B = \{\} \longleftrightarrow A \neq \{\} \wedge B = \{\}$ (**is** *?L* \longleftrightarrow *?R*)
proof
assume *?L*
then show *?R*
using *Func-empty Func-non-emp[of B A]* **by** *blast*
next
assume *?R*
then show *?L*
using *Func-empty Func-non-emp[of B A]* **by** (*auto simp: Func-def*)
qed

lemma *Func-map-surj*:

```

assumes B1: f1 ‘ A1 = B1 and A2: inj-on f2 B2 f2 ‘ B2 ⊆ A2
and B2A2: B2 = {} ⇒ A2 = {}
shows Func B2 B1 = Func-map B2 f1 f2 ‘ Func A2 A1
proof(cases B2 = {})
  case True
    thus ?thesis using B2A2 by (auto simp: Func-empty Func-map-def)
  next
    case False note B2 = False
    show ?thesis
    proof safe
      fix h assume h: h ∈ Func B2 B1
      define j1 where j1 = inv-into A1 f1
      have ∀ a2 ∈ f2 ‘ B2. ∃ b2. b2 ∈ B2 ∧ f2 b2 = a2 by blast
      then obtain k where k: ∀ a2 ∈ f2 ‘ B2. k a2 ∈ B2 ∧ f2 (k a2) = a2
        by atomize-elim (rule bchoice)
      {fix b2 assume b2: b2 ∈ B2
        hence f2 (k (f2 b2)) = f2 b2 using k A2(2) by auto
        moreover have k (f2 b2) ∈ B2 using b2 A2(2) k by auto
        ultimately have k (f2 b2) = b2 using b2 A2(1) unfolding inj-on-def by
        blast
      } note kk = this
      obtain b22 where b22: b22 ∈ B2 using B2 by auto
      define j2 where [abs-def]: j2 a2 = (if a2 ∈ f2 ‘ B2 then k a2 else b22) for a2
      have j2A2: j2 ‘ A2 ⊆ B2 unfolding j2-def using k b22 by auto
      have j2: ∧ b2. b2 ∈ B2 ⇒ j2 (f2 b2) = b2
        using kk unfolding j2-def by auto
      define g where g = Func-map A2 j1 j2 h
      have Func-map B2 f1 f2 g = h
      proof (rule ext)
        fix b2 show Func-map B2 f1 f2 g b2 = h b2
        proof(cases b2 ∈ B2)
          case True
            show ?thesis
            proof (cases h b2 = undefined)
              case True
                hence b1: h b2 ∈ f1 ‘ A1 using h ⟨b2 ∈ B2⟩ unfolding B1 Func-def by
                auto
                show ?thesis using A2 f-inv-into-f[OF b1]
                  unfolding True g-def Func-map-def j1-def j2[OF ⟨b2 ∈ B2⟩] by auto
                  qed(insert A2 True j2[OF True] h B1, unfold j1-def g-def Func-def
                Func-map-def,
                  auto intro: f-inv-into-f)
                qed(insert h, unfold Func-def Func-map-def, auto)
              qed
            moreover have g ∈ Func A2 A1 unfolding g-def apply(rule Func-map[OF
            h])
              using j2A2 B1 A2 unfolding j1-def by (fast intro: inv-into-into)+
              ultimately show h ∈ Func-map B2 f1 f2 ‘ Func A2 A1
                unfolding Func-map-def[abs-def] by auto

```

```
qed(use B1 Func-map[OF - - A2(2)] in auto)
qed

end
```

30 Cardinal-Order Relations as Needed by Bounded Natural Functors

```
theory BNF-Cardinal-Order-Relation
  imports Zorn BNF-Wellorder-Constructions
begin
```

In this section, we define cardinal-order relations to be *minim* well-orders on their field. Then we define the cardinal of a set to be *some* cardinal-order relation on that set, which will be unique up to order isomorphism. Then we study the connection between cardinals and:

- standard set-theoretic constructions: products, sums, unions, lists, powersets, set-of finite sets operator;
- finiteness and infiniteness (in particular, with the numeric cardinal operator for finite sets, *card*, from the theory *Finite-Sets.thy*).

On the way, we define the canonical ω cardinal and finite cardinals. We also define (again, up to order isomorphism) the successor of a cardinal, and show that any cardinal admits a successor.

Main results of this section are the existence of cardinal relations and the facts that, in the presence of infiniteness, most of the standard set-theoretic constructions (except for the powerset) *do not increase cardinality*. In particular, e.g., the set of words/lists over any infinite set has the same cardinality (hence, is in bijection) with that set.

30.1 Cardinal orders

A cardinal order in our setting shall be a well-order *minim* w.r.t. the order-embedding relation, \leq_o (which is the same as being *minimal* w.r.t. the strict order-embedding relation, $<_o$), among all the well-orders on its field.

definition *card-order-on* :: 'a set \Rightarrow 'a rel \Rightarrow bool

where

card-order-on A r \equiv *well-order-on* A r \wedge ($\forall r'. \text{well-order-on } A r' \longrightarrow r \leq_o r'$)

abbreviation *Card-order* r \equiv *card-order-on* (Field r) r

abbreviation *card-order* r \equiv *card-order-on* UNIV r

lemma *card-order-on-well-order-on*:

assumes *card-order-on A r*
shows *well-order-on A r*
using *assms unfolding card-order-on-def* **by** *simp*

lemma *card-order-on-Card-order*:
card-order-on A r \implies A = Field r \wedge Card-order r
unfolding *card-order-on-def* **using** *well-order-on-Field* **by** *blast*

The existence of a cardinal relation on any given set (which will mean that any set has a cardinal) follows from two facts:

- Zermelo’s theorem (proved in *Zorn.thy* as theorem *well-order-on*), which states that on any given set there exists a well-order;
- The well-founded-ness of $<o$, ensuring that then there exists a minimal such well-order, i.e., a cardinal order.

theorem *card-order-on*: $\exists r. \text{card-order-on } A \ r$
proof –
define *R* **where** $R \equiv \{r. \text{well-order-on } A \ r\}$
have $R \neq \{\}$ $\wedge (\forall r \in R. \text{Well-order } r)$
using *well-order-on[of A] R-def well-order-on-Well-order* **by** *blast*
with *exists-minim-Well-order[of R]* **show** *?thesis*
by (*auto simp: R-def card-order-on-def*)
qed

lemma *card-order-on-ordIso*:
assumes *CO: card-order-on A r* **and** *CO': card-order-on A r'*
shows $r =_o r'$
using *assms unfolding card-order-on-def*
using *ordIso-iff-ordLeq* **by** *blast*

lemma *Card-order-ordIso*:
assumes *CO: Card-order r* **and** *ISO: r' =_o r*
shows *Card-order r'*
using *ISO unfolding ordIso-def*
proof(*unfold card-order-on-def, auto*)
fix *p'* **assume** *well-order-on (Field r') p'*
hence $0: \text{Well-order } p' \wedge \text{Field } p' = \text{Field } r'$
using *well-order-on-Well-order* **by** *blast*
obtain *f* **where** $1: \text{iso } r' \ r \ f$ **and** $2: \text{Well-order } r \wedge \text{Well-order } r'$
using *ISO unfolding ordIso-def* **by** *auto*
hence $3: \text{inj-on } f \ (\text{Field } r') \wedge f \ ' \ (\text{Field } r') = \text{Field } r$
by (*auto simp add: iso-iff embed-inj-on*)
let $?p = \text{dir-image } p' \ f$
have $4: p' =_o ?p \wedge \text{Well-order } ?p$
using $0 \ 2 \ 3$ **by** (*auto simp add: dir-image-ordIso Well-order-dir-image*)
moreover **have** $\text{Field } ?p = \text{Field } r$
using $0 \ 3$ **by** (*auto simp add: dir-image-Field*)

ultimately have *well-order-on* (*Field* r) $?p$ **by** *auto*
 hence $r \leq_o ?p$ **using** *CO unfolding card-order-on-def* **by** *auto*
 thus $r' \leq_o p'$
using *ISO 4 ordLeq-ordIso-trans ordIso-ordLeq-trans ordIso-symmetric* **by** *blast*
qed

lemma *Card-order-ordIso2*:
assumes *CO: Card-order* r **and** *ISO: $r =_o r'$*
shows *Card-order* r'
using *assms Card-order-ordIso ordIso-symmetric* **by** *blast*

30.2 Cardinal of a set

We define the cardinal of set to be *some* cardinal order on that set. We shall prove that this notion is unique up to order isomorphism, meaning that order isomorphism shall be the true identity of cardinals.

definition *card-of* :: '*a set* \Rightarrow '*a rel* ($|-|$)
where *card-of* $A = (\text{SOME } r. \text{card-order-on } A \ r)$

lemma *card-of-card-order-on: card-order-on* $A \ |A|$
unfolding *card-of-def* **by** (*auto simp add: card-order-on someI-ex*)

lemma *card-of-well-order-on: well-order-on* $A \ |A|$
using *card-of-card-order-on card-order-on-def* **by** *blast*

lemma *Field-card-of: Field* $|A| = A$
using *card-of-card-order-on[of A]* **unfolding** *card-order-on-def*
using *well-order-on-Field* **by** *blast*

lemma *card-of-Card-order: Card-order* $|A|$
by (*simp only: card-of-card-order-on Field-card-of*)

corollary *ordIso-card-of-imp-Card-order*:
 $r =_o |A| \implies \text{Card-order } r$
using *card-of-Card-order Card-order-ordIso* **by** *blast*

lemma *card-of-Well-order: Well-order* $|A|$
using *card-of-Card-order* **unfolding** *card-order-on-def* **by** *auto*

lemma *card-of-refl: $|A| =_o |A|$*
using *card-of-Well-order ordIso-reflexive* **by** *blast*

lemma *card-of-least: well-order-on* $A \ r \implies |A| \leq_o r$
using *card-of-card-order-on* **unfolding** *card-order-on-def* **by** *blast*

lemma *card-of-ordIso*:
 $(\exists f. \text{bij-betw } f \ A \ B) = (|A| =_o |B|)$
proof(*auto*)
fix f **assume** $*$: *bij-betw* $f \ A \ B$

then obtain r **where** $\text{well-order-on } B \wedge |A| =_o r$
using $\text{Well-order-iso-copy card-of-well-order-on}$ **by** blast
hence $|B| \leq_o |A|$ **using** card-of-least
 $\text{ordLeq-ordIso-trans ordIso-symmetric}$ **by** blast
moreover
{let $?g = \text{inv-into } A \text{ } f$
have $\text{bij-betw } ?g \text{ } B \text{ } A$ **using** $*$ bij-betw-inv-into **by** blast
then obtain r **where** $\text{well-order-on } A \wedge |B| =_o r$
using $\text{Well-order-iso-copy card-of-well-order-on}$ **by** blast
hence $|A| \leq_o |B|$
using $\text{card-of-least ordLeq-ordIso-trans ordIso-symmetric}$ **by** blast
}
ultimately show $|A| =_o |B|$ **using** ordIso-iff-ordLeq **by** blast
next
assume $|A| =_o |B|$
then obtain f **where** $\text{iso } (|A|) (|B|) f$
unfolding ordIso-def **by** auto
hence $\text{bij-betw } f \text{ } A \text{ } B$ **unfolding** $\text{iso-def Field-card-of}$ **by** simp
thus $\exists f. \text{bij-betw } f \text{ } A \text{ } B$ **by** auto
qed

lemma card-of-ordLeq :

$(\exists f. \text{inj-on } f \text{ } A \wedge f ' A \leq B) = (|A| \leq_o |B|)$

proof(auto)

fix f **assume** $*$: $\text{inj-on } f \text{ } A$ **and** $**$: $f ' A \leq B$

{assume $|B| <_o |A|$

hence $|B| \leq_o |A|$ **using** $\text{ordLeq-iff-ordLess-or-ordIso}$ **by** blast

then obtain g **where** $\text{embed } (|B|) (|A|) g$

unfolding ordLeq-def **by** auto

hence $1: \text{inj-on } g \text{ } B \wedge g ' B \leq A$ **using** $\text{embed-inj-on}[of \text{ } |B| \text{ } |A| \text{ } g]$

$\text{card-of-Well-order}[of \text{ } B] \text{ Field-card-of}[of \text{ } B] \text{ Field-card-of}[of \text{ } A]$

$\text{embed-Field}[of \text{ } |B| \text{ } |A| \text{ } g]$ **by** auto

obtain h **where** $\text{bij-betw } h \text{ } A \text{ } B$

using $*$ $** \text{ } 1 \text{ Schroeder-Bernstein}[of \text{ } f]$ **by** fastforce

hence $|A| \leq_o |B|$ **using** $\text{card-of-ordIso ordIso-iff-ordLeq}$ **by** auto

}

thus $|A| \leq_o |B|$ **using** $\text{ordLess-or-ordLeq}[of \text{ } |B| \text{ } |A|]$

by ($\text{auto simp: card-of-Well-order}$)

next

assume $*$: $|A| \leq_o |B|$

obtain f **where** $\text{embed } |A| \text{ } |B| \text{ } f$

using $*$ **unfolding** ordLeq-def **by** auto

hence $\text{inj-on } f \text{ } A \wedge f ' A \leq B$

using $\text{embed-inj-on}[of \text{ } |A| \text{ } |B|] \text{ card-of-Well-order embed-Field}[of \text{ } |A| \text{ } |B|]$

by ($\text{auto simp: Field-card-of}$)

thus $\exists f. \text{inj-on } f \text{ } A \wedge f ' A \leq B$ **by** auto

qed

lemma card-of-ordLeq2 :

$A \neq \{\} \implies (\exists g. g \text{ ‘ } B = A) = (|A| \leq_o |B|)$
using *card-of-ordLeq*[of $A \ B$] *inj-on-iff-surj*[of $A \ B$] **by** *auto*

lemma *card-of-ordLess*:

$(\neg(\exists f. \text{inj-on } f \ A \wedge f \text{ ‘ } A \leq B)) = (|B| <_o |A|)$

proof –

have $(\neg(\exists f. \text{inj-on } f \ A \wedge f \text{ ‘ } A \leq B)) = (\neg |A| \leq_o |B|)$

using *card-of-ordLeq* **by** *blast*

also have $\dots = (|B| <_o |A|)$

using *not-ordLeq-iff-ordLess* **by** (*auto intro: card-of-Well-order*)

finally show *?thesis* .

qed

lemma *card-of-ordLess2*:

$B \neq \{\} \implies (\neg(\exists f. f \text{ ‘ } A = B)) = (|A| <_o |B|)$

using *card-of-ordLess*[of $B \ A$] *inj-on-iff-surj*[of $B \ A$] **by** *auto*

lemma *card-of-ordIsoI*:

assumes *bij-betw* $f \ A \ B$

shows $|A| =_o |B|$

using *assms unfolding card-of-ordIso[symmetric]* **by** *auto*

lemma *card-of-ordLeqI*:

assumes *inj-on* $f \ A$ **and** $\bigwedge a. a \in A \implies f \ a \in B$

shows $|A| \leq_o |B|$

using *assms unfolding card-of-ordLeq[symmetric]* **by** *auto*

lemma *card-of-unique*:

card-order-on $A \ r \implies r =_o |A|$

by (*simp only: card-order-on-ordIso card-of-card-order-on*)

lemma *card-of-mono1*:

$A \leq B \implies |A| \leq_o |B|$

using *inj-on-id*[of A] *card-of-ordLeq*[of $A \ B$] **by** *fastforce*

lemma *card-of-mono2*:

assumes $r \leq_o r'$

shows $|Field \ r| \leq_o |Field \ r'|$

proof –

obtain f **where**

1: well-order-on $(Field \ r) \ r \wedge$ *well-order-on* $(Field \ r) \ r \wedge$ *embed* $r \ r' \ f$

using *assms unfolding ordLeq-def*

by (*auto simp add: well-order-on-Well-order*)

hence *inj-on* $f \ (Field \ r) \wedge f \text{ ‘ } (Field \ r) \leq Field \ r'$

by (*auto simp add: embed-inj-on embed-Field*)

thus $|Field \ r| \leq_o |Field \ r'|$ **using** *card-of-ordLeq* **by** *blast*

qed

lemma *card-of-cong*: $r =_o r' \implies |Field \ r| =_o |Field \ r'|$

by (*simp add: ordIso-iff-ordLeq card-of-mono2*)

lemma *card-of-Field-ordIso*:

assumes *Card-order r*

shows $|Field\ r| =_o r$

proof –

have *card-order-on (Field r) r*

using *assms card-order-on-Card-order* **by** *blast*

moreover have *card-order-on (Field r) |Field r|*

using *card-of-card-order-on* **by** *blast*

ultimately show *?thesis* **using** *card-order-on-ordIso* **by** *blast*

qed

lemma *Card-order-iff-ordIso-card-of*:

Card-order r = (r =_o |Field r|)

using *ordIso-card-of-imp-Card-order card-of-Field-ordIso ordIso-symmetric* **by** *blast*

lemma *Card-order-iff-ordLeq-card-of*:

Card-order r = (r ≤_o |Field r|)

proof –

have *Card-order r = (r =_o |Field r|)*

unfolding *Card-order-iff-ordIso-card-of* **by** *simp*

also have $\dots = (r \leq_o |Field\ r| \wedge |Field\ r| \leq_o r)$

unfolding *ordIso-iff-ordLeq* **by** *simp*

also have $\dots = (r \leq_o |Field\ r|)$

using *card-of-least*

by (*auto simp: card-of-least ordLeq-Well-order-simp*)

finally show *?thesis* .

qed

lemma *Card-order-iff-Restr-underS*:

assumes *Well-order r*

shows *Card-order r = ($\forall a \in Field\ r. Restr\ r\ (underS\ r\ a) <_o |Field\ r|$)*

using *assms ordLeq-iff-ordLess-Restr card-of-Well-order*

unfolding *Card-order-iff-ordLeq-card-of* **by** *blast*

lemma *card-of-underS*:

assumes *r: Card-order r* **and** *a: a ∈ Field r*

shows $|underS\ r\ a| <_o r$

proof –

let $?A = underS\ r\ a$ **let** $?r' = Restr\ r\ ?A$

have *1: Well-order r*

using *r* **unfolding** *card-order-on-def* **by** *simp*

have *Well-order ?r'* **using** *1 Well-order-Restr* **by** *auto*

with *card-of-card-order-on* **have** $|Field\ ?r'| \leq_o ?r'$

unfolding *card-order-on-def* **by** *auto*

moreover have *Field ?r' = ?A*

using *1 wo-rel.underS-ofilter Field-Restr-ofilter*

unfolding *wo-rel-def* **by** *fastforce*
ultimately have $|?A| \leq_o ?r'$ **by** *simp*
also have $?r' <_o |Field\ r|$
using *1 a r Card-order-iff-Restr-underS* **by** *blast*
also have $|Field\ r| =_o r$
using *r ordIso-symmetric* **unfolding** *Card-order-iff-ordIso-card-of* **by** *auto*
finally show *?thesis* .
qed

lemma *ordLess-Field*:

assumes $r <_o r'$
shows $|Field\ r| <_o r'$
proof –
have *well-order-on (Field r) r* **using** *assms* **unfolding** *ordLess-def*
by (*auto simp add: well-order-on-Well-order*)
hence $|Field\ r| \leq_o r$ **using** *card-of-least* **by** *blast*
thus *?thesis* **using** *assms ordLeq-ordLess-trans* **by** *blast*
qed

lemma *internalize-card-of-ordLeq*:

$(|A| \leq_o r) = (\exists B \leq Field\ r. |A| =_o |B| \wedge |B| \leq_o r)$
proof
assume $|A| \leq_o r$
then obtain p **where** *1: Field p ≤ Field r ∧ |A| =_o p ∧ p ≤_o r*
using *internalize-ordLeq[of |A| r]* **by** *blast*
hence *Card-order p* **using** *card-of-Card-order Card-order-ordIso2* **by** *blast*
hence $|Field\ p| =_o p$ **using** *card-of-Field-ordIso* **by** *blast*
hence $|A| =_o |Field\ p| \wedge |Field\ p| \leq_o r$
using *1 ordIso-equivalence ordIso-ordLeq-trans* **by** *blast*
thus $\exists B \leq Field\ r. |A| =_o |B| \wedge |B| \leq_o r$ **using** *1* **by** *blast*
next
assume $\exists B \leq Field\ r. |A| =_o |B| \wedge |B| \leq_o r$
thus $|A| \leq_o r$ **using** *ordIso-ordLeq-trans* **by** *blast*
qed

lemma *internalize-card-of-ordLeq2*:

$(|A| \leq_o |C|) = (\exists B \leq C. |A| =_o |B| \wedge |B| \leq_o |C|)$
using *internalize-card-of-ordLeq[of A |C|] Field-card-of[of C]* **by** *auto*

30.3 Cardinals versus set operations on arbitrary sets

Here we embark in a long journey of simple results showing that the standard set-theoretic operations are well-behaved w.r.t. the notion of cardinal – essentially, this means that they preserve the “cardinal identity” $=_o$ and are monotonic w.r.t. \leq_o .

lemma *card-of-empty*: $|\{\}| \leq_o |A|$
using *card-of-ordLeq inj-on-id* **by** *blast*

lemma *card-of-empty1*:
 assumes *Well-order r* \vee *Card-order r*
 shows $|\{\}| \leq_o r$
proof –
 have *Well-order r* **using** *assms* **unfolding** *card-order-on-def* **by** *auto*
 hence $|Field\ r| \leq_o r$
using *assms* *card-of-least* **by** *blast*
 moreover have $|\{\}| \leq_o |Field\ r|$ **by** (*simp add: card-of-empty*)
 ultimately show *?thesis* **using** *ordLeq-transitive* **by** *blast*
qed

corollary *Card-order-empty*:
Card-order r $\implies |\{\}| \leq_o r$ **by** (*simp add: card-of-empty1*)

lemma *card-of-empty2*:
 assumes $|A| =_o |\{\}|$
 shows $A = \{\}$
 using *assms* *card-of-ordIso[of A]* *bij-betw-empty2* **by** *blast*

lemma *card-of-empty3*:
 assumes $|A| \leq_o |\{\}|$
 shows $A = \{\}$
 using *assms*
by (*simp add: ordIso-iff-ordLeq card-of-empty1 card-of-empty2*
ordLeq-Well-order-simp)

lemma *card-of-empty-ordIso*:
 $|\{\}::'a\ set| =_o |\{\}::'b\ set|$
 using *card-of-ordIso* **unfolding** *bij-betw-def inj-on-def* **by** *blast*

lemma *card-of-image*:
 $|f\ 'A| \leq_o |A|$
proof(*cases A = \{\}*)
 case *False*
 hence $f\ 'A \neq \{\}$ **by** *auto*
 thus *?thesis*
 using *card-of-ordLeq2[of f\ 'A A]* **by** *auto*
qed (*simp add: card-of-empty*)

lemma *surj-imp-ordLeq*:
 assumes $B \subseteq f\ 'A$
 shows $|B| \leq_o |A|$
proof –
 have $|B| \leq_o |f\ 'A|$ **using** *assms* *card-of-mono1* **by** *auto*
 thus *?thesis* **using** *card-of-image* *ordLeq-transitive* **by** *blast*
qed

lemma *card-of-singl-ordLeq*:
 assumes $A \neq \{\}$

shows $|\{b\}| \leq_o |A|$
proof –
 obtain a where $*$: $a \in A$ using *assms* by *auto*
 let $?h = \lambda b'::'b$. if $b' = b$ then a else *undefined*
 have *inj-on* $?h \{b\} \wedge ?h \{b\} \leq A$
 using $*$ **unfolding** *inj-on-def* by *auto*
 thus *?thesis* **unfolding** *card-of-ordLeq[symmetric]* by (*intro exI*)
qed

corollary *Card-order-singl-ordLeq*:
 $\llbracket \text{Card-order } r; \text{Field } r \neq \{\} \rrbracket \implies |\{b\}| \leq_o r$
 using *card-of-singl-ordLeq[of Field r b]*
card-of-Field-ordIso[of r] *ordLeq-ordIso-trans* by *blast*

lemma *card-of-Pow*: $|A| <_o |\text{Pow } A|$
 using *card-of-ordLess2[of Pow A A]* *Cantors-theorem[of A]*
Pow-not-empty[of A] by *auto*

corollary *Card-order-Pow*:
 $\text{Card-order } r \implies r <_o |\text{Pow}(\text{Field } r)|$
 using *card-of-Pow* *card-of-Field-ordIso* *ordIso-ordLess-trans* *ordIso-symmetric* by
blast

lemma *card-of-Plus1*: $|A| \leq_o |A <+> B|$ and *card-of-Plus2*: $|B| \leq_o |A <+> B|$
 using *card-of-ordLeq* by *force+*

corollary *Card-order-Plus1*:
 $\text{Card-order } r \implies r \leq_o |(\text{Field } r) <+> B|$
 using *card-of-Plus1* *card-of-Field-ordIso* *ordIso-ordLeq-trans* *ordIso-symmetric* by
blast

corollary *Card-order-Plus2*:
 $\text{Card-order } r \implies r \leq_o |A <+> (\text{Field } r)|$
 using *card-of-Plus2* *card-of-Field-ordIso* *ordIso-ordLeq-trans* *ordIso-symmetric* by
blast

lemma *card-of-Plus-empty1*: $|A| =_o |A <+> \{\}|$
proof –
 have *bij-betw* *Inl* $A (A <+> \{\})$ **unfolding** *bij-betw-def* *inj-on-def* by *auto*
 thus *?thesis* using *card-of-ordIso* by *auto*
qed

lemma *card-of-Plus-empty2*: $|A| =_o |\{\} <+> A|$
proof –
 have *bij-betw* *Inr* $A (\{\} <+> A)$ **unfolding** *bij-betw-def* *inj-on-def* by *auto*
 thus *?thesis* using *card-of-ordIso* by *auto*
qed

lemma *card-of-Plus-commute*: $|A <+> B| =_o |B <+> A|$

proof –

let $?f = \lambda c. \text{ case } c \text{ of } \text{Inl } a \Rightarrow \text{Inr } a \mid \text{Inr } b \Rightarrow \text{Inl } b$
have $\text{bij-betw } ?f (A \langle + \rangle B) (B \langle + \rangle A)$
unfolding $\text{bij-betw-def inj-on-def}$ **by** force
thus $?thesis$ **using** card-of-ordIso **by** blast
qed

lemma $\text{card-of-Plus-assoc}$:

fixes $A :: 'a \text{ set}$ **and** $B :: 'b \text{ set}$ **and** $C :: 'c \text{ set}$
shows $|(A \langle + \rangle B) \langle + \rangle C| =_o |A \langle + \rangle B \langle + \rangle C|$

proof –

define $f :: ('a + 'b) + 'c \Rightarrow 'a + 'b + 'c$

where $[\text{abs-def}]: f k =$

$(\text{case } k \text{ of}$
 $\text{Inl } ab \Rightarrow$
 $(\text{case } ab \text{ of}$
 $\text{Inl } a \Rightarrow \text{Inl } a$
 $\mid \text{Inr } b \Rightarrow \text{Inr } (\text{Inl } b))$
 $\mid \text{Inr } c \Rightarrow \text{Inr } (\text{Inr } c))$

for k

have $A \langle + \rangle B \langle + \rangle C \subseteq f' ((A \langle + \rangle B) \langle + \rangle C)$

proof

fix x **assume** $x: x \in A \langle + \rangle B \langle + \rangle C$

show $x \in f' ((A \langle + \rangle B) \langle + \rangle C)$

proof $(\text{cases } x)$

case $(\text{Inl } a)$

hence $a \in A \ x = f (\text{Inl } (\text{Inl } a))$

using x **unfolding** $f\text{-def}$ **by** auto

thus $?thesis$ **by** auto

next

case $(\text{Inr } bc)$ **with** x **show** $?thesis$

by $(\text{cases } bc)$ $(\text{force simp: } f\text{-def})+$

qed

qed

hence $\text{bij-betw } f ((A \langle + \rangle B) \langle + \rangle C) (A \langle + \rangle B \langle + \rangle C)$

unfolding $\text{bij-betw-def inj-on-def } f\text{-def}$ **by** fastforce

thus $?thesis$ **using** card-of-ordIso **by** blast

qed

lemma $\text{card-of-Plus-mono1}$:

assumes $|A| \leq_o |B|$

shows $|A \langle + \rangle C| \leq_o |B \langle + \rangle C|$

proof –

obtain f **where** $f: \text{inj-on } f \ A \wedge f' \ A \leq B$

using $\text{assms card-of-ordLeq[of } A]$ **by** fastforce

define g **where** $g \equiv \lambda d. \text{ case } d \text{ of } \text{Inl } a \Rightarrow \text{Inl } (f a) \mid \text{Inr } (c::'c) \Rightarrow \text{Inr } c$

have $\text{inj-on } g (A \langle + \rangle C) \wedge g' (A \langle + \rangle C) \leq (B \langle + \rangle C)$

using f **unfolding** $\text{inj-on-def } g\text{-def}$ **by** force

thus $?thesis$ **using** card-of-ordLeq **by** blast

qed

corollary *ordLeq-Plus-mono1*:

assumes $r \leq_o r'$

shows $|(Field\ r) \langle + \rangle C| \leq_o |(Field\ r') \langle + \rangle C|$

using *assms card-of-mono2 card-of-Plus-mono1* **by** *blast*

lemma *card-of-Plus-mono2*:

assumes $|A| \leq_o |B|$

shows $|C \langle + \rangle A| \leq_o |C \langle + \rangle B|$

using *card-of-Plus-mono1[OF assms]*

by (*blast intro: card-of-Plus-commute ordIso-ordLeq-trans ordLeq-ordIso-trans*)

corollary *ordLeq-Plus-mono2*:

assumes $r \leq_o r'$

shows $|A \langle + \rangle (Field\ r)| \leq_o |A \langle + \rangle (Field\ r')|$

using *assms card-of-mono2 card-of-Plus-mono2* **by** *blast*

lemma *card-of-Plus-mono*:

assumes $|A| \leq_o |B|$ **and** $|C| \leq_o |D|$

shows $|A \langle + \rangle C| \leq_o |B \langle + \rangle D|$

using *assms card-of-Plus-mono1[of A B C] card-of-Plus-mono2[of C D B]*

ordLeq-transitive **by** *blast*

corollary *ordLeq-Plus-mono*:

assumes $r \leq_o r'$ **and** $p \leq_o p'$

shows $|(Field\ r) \langle + \rangle (Field\ p)| \leq_o |(Field\ r') \langle + \rangle (Field\ p')|$

using *assms card-of-mono2[of r r'] card-of-mono2[of p p'] card-of-Plus-mono* **by** *blast*

lemma *card-of-Plus-cong1*:

assumes $|A| =_o |B|$

shows $|A \langle + \rangle C| =_o |B \langle + \rangle C|$

using *assms* **by** (*simp add: ordIso-iff-ordLeq card-of-Plus-mono1*)

corollary *ordIso-Plus-cong1*:

assumes $r =_o r'$

shows $|(Field\ r) \langle + \rangle C| =_o |(Field\ r') \langle + \rangle C|$

using *assms card-of-cong card-of-Plus-cong1* **by** *blast*

lemma *card-of-Plus-cong2*:

assumes $|A| =_o |B|$

shows $|C \langle + \rangle A| =_o |C \langle + \rangle B|$

using *assms* **by** (*simp add: ordIso-iff-ordLeq card-of-Plus-mono2*)

corollary *ordIso-Plus-cong2*:

assumes $r =_o r'$

shows $|A \langle + \rangle (Field\ r)| =_o |A \langle + \rangle (Field\ r')|$

using *assms card-of-cong card-of-Plus-cong2* **by** *blast*

lemma *card-of-Plus-cong*:

assumes $|A| =_o |B|$ **and** $|C| =_o |D|$

shows $|A <+> C| =_o |B <+> D|$

using *assms* **by** (*simp add: ordIso-iff-ordLeq card-of-Plus-mono*)

corollary *ordIso-Plus-cong*:

assumes $r =_o r'$ **and** $p =_o p'$

shows $|(Field\ r) <+> (Field\ p)| =_o |(Field\ r') <+> (Field\ p')|$

using *assms card-of-cong[of r r'] card-of-cong[of p p'] card-of-Plus-cong* **by** *blast*

lemma *card-of-Un-Plus-ordLeq*:

$|A \cup B| \leq_o |A <+> B|$

proof –

let $?f = \lambda\ c.$ *if* $c \in A$ *then* *Inl* c *else* *Inr* c

have *inj-on* $?f\ (A \cup B) \wedge ?f\ ' (A \cup B) \leq A <+> B$

unfolding *inj-on-def* **by** *auto*

thus *?thesis* **using** *card-of-ordLeq* **by** *blast*

qed

lemma *card-of-Times1*:

assumes $A \neq \{\}$

shows $|B| \leq_o |B \times A|$

proof(*cases* $B = \{\}$)

case *False*

have *fst* $'(B \times A) = B$ **using** *assms* **by** *auto*

thus *?thesis* **using** *inj-on-iff-surj[of B B × A]*

card-of-ordLeq False **by** *blast*

qed (*simp add: card-of-empty*)

lemma *card-of-Times-commute*: $|A \times B| =_o |B \times A|$

proof –

have *bij-betw* $(\lambda(a,b). (b,a))\ (A \times B)\ (B \times A)$

unfolding *bij-betw-def inj-on-def* **by** *auto*

thus *?thesis* **using** *card-of-ordIso* **by** *blast*

qed

lemma *card-of-Times2*:

assumes $A \neq \{\}$ **shows** $|B| \leq_o |A \times B|$

using *assms card-of-Times1[of A B] card-of-Times-commute[of B A]*

ordLeq-ordIso-trans **by** *blast*

corollary *Card-order-Times1*:

$\llbracket Card\text{-order}\ r; B \neq \{\} \rrbracket \implies r \leq_o |(Field\ r) \times B|$

using *card-of-Times1[of B] card-of-Field-ordIso*

ordIso-ordLeq-trans ordIso-symmetric **by** *blast*

corollary *Card-order-Times2*:

$\llbracket Card\text{-order}\ r; A \neq \{\} \rrbracket \implies r \leq_o |A \times (Field\ r)|$

using *card-of-Times2*[of A] *card-of-Field-ordIso*
ordIso-ordLeq-trans ordIso-symmetric **by** *blast*

lemma *card-of-Times3*: $|A| \leq_o |A \times A|$
using *card-of-Times1*[of A]
by(*cases* $A = \{\}$, *simp add: card-of-empty*)

lemma *card-of-Plus-Times-bool*: $|A <+> A| =_o |A \times (UNIV::\text{bool set})|$
proof –

let $?f = \lambda c::'a + 'a. \text{case } c \text{ of } \text{Inl } a \Rightarrow (a, \text{True})$
 $|\text{Inr } a \Rightarrow (a, \text{False})$

have *bij-betw* $?f$ ($A <+> A$) ($A \times (UNIV::\text{bool set})$)

proof –

have $\bigwedge c1\ c2. ?f\ c1 = ?f\ c2 \implies c1 = c2$
by (*force split: sum.split-asm*)

moreover

have $\bigwedge c. c \in A <+> A \implies ?f\ c \in A \times (UNIV::\text{bool set})$
by (*force split: sum.split-asm*)

moreover

{fix $a\ bl$ **assume** $(a, bl) \in A \times (UNIV::\text{bool set})$
hence $(a, bl) \in ?f\ ' (A <+> A)$
by (*cases* bl) (*force split: sum.split-asm*)+

}

ultimately show *?thesis unfolding bij-betw-def inj-on-def* **by** *auto*

qed

thus *?thesis* **using** *card-of-ordIso* **by** *blast*

qed

lemma *card-of-Times-mono1*:

assumes $|A| \leq_o |B|$

shows $|A \times C| \leq_o |B \times C|$

proof –

obtain f **where** $f: \text{inj-on } f\ A \wedge f\ ' A \leq B$

using *assms card-of-ordLeq*[of A] **by** *fastforce*

define g **where** $g \equiv (\lambda(a, c::'c). (f\ a, c))$

have *inj-on* g ($A \times C$) $\wedge g\ ' (A \times C) \leq (B \times C)$

using f **unfolding** *inj-on-def* **using** g -*def* **by** *auto*

thus *?thesis* **using** *card-of-ordLeq* **by** *blast*

qed

corollary *ordLeq-Times-mono1*:

assumes $r \leq_o r'$

shows $|(Field\ r) \times C| \leq_o |(Field\ r') \times C|$

using *assms card-of-mono2 card-of-Times-mono1* **by** *blast*

lemma *card-of-Times-mono2*:

assumes $|A| \leq_o |B|$

shows $|C \times A| \leq_o |C \times B|$

using *assms card-of-Times-mono1*[of $A\ B\ C$]

by (blast intro: card-of-Times-commute ordIso-ordLeq-trans ordLeq-ordIso-trans)

corollary *ordLeq-Times-mono2*:

assumes $r \leq_o r'$

shows $|A \times (\text{Field } r)| \leq_o |A \times (\text{Field } r')|$

using *assms card-of-mono2 card-of-Times-mono2* by blast

lemma *card-of-Sigma-mono1*:

assumes $\forall i \in I. |A \ i| \leq_o |B \ i|$

shows $|\text{SIGMA } i : I. A \ i| \leq_o |\text{SIGMA } i : I. B \ i|$

proof –

have $\forall i. i \in I \longrightarrow (\exists f. \text{inj-on } f \ (A \ i) \wedge f \ ' \ (A \ i) \leq B \ i)$

using *assms* by (auto simp add: card-of-ordLeq)

with choice[of $\lambda \ i \ f. i \in I \longrightarrow \text{inj-on } f \ (A \ i) \wedge f \ ' \ (A \ i) \leq B \ i$]

obtain *F* where $F: \forall i \in I. \text{inj-on } (F \ i) \ (A \ i) \wedge (F \ i) \ ' \ (A \ i) \leq B \ i$

by *atomize-elim* (auto intro: bchoice)

define *g* where $g \equiv (\lambda(i,a::'b). (i, F \ i \ a))$

have $\text{inj-on } g \ (\text{Sigma } I \ A) \wedge g \ ' \ (\text{Sigma } I \ A) \leq (\text{Sigma } I \ B)$

using *F unfolding inj-on-def* using *g-def* by force

thus *?thesis* using *card-of-ordLeq* by blast

qed

lemma *card-of-UNION-Sigma*:

$|\bigcup i \in I. A \ i| \leq_o |\text{SIGMA } i : I. A \ i|$

using *Ex-inj-on-UNION-Sigma* [of *A I*] *card-of-ordLeq* by blast

lemma *card-of-bool*:

assumes $a1 \neq a2$

shows $|\text{UNIV}::\text{bool set}| =_o |\{a1, a2\}|$

proof –

let $?f = \lambda \ bl. \text{if } bl \text{ then } a1 \text{ else } a2$

have *bij-betw* $?f \ \text{UNIV} \ \{a1, a2\}$

proof –

have $\bigwedge bl1 \ bl2. ?f \ bl1 = ?f \ bl2 \implies bl1 = bl2$

using *assms* by (force split: if-split-asm)

moreover

have $\bigwedge bl. ?f \ bl \in \{a1, a2\}$

using *assms* by (force split: if-split-asm)

ultimately show *?thesis* unfolding *bij-betw-def inj-on-def* by force

qed

thus *?thesis* using *card-of-ordIso* by blast

qed

lemma *card-of-Plus-Times-aux*:

assumes $A2: a1 \neq a2 \wedge \{a1, a2\} \leq A$ and

LEQ: $|A| \leq_o |B|$

shows $|A \ <+> \ B| \leq_o |A \times \ B|$

proof –

have *1*: $|\text{UNIV}::\text{bool set}| \leq_o |A|$

```

  using A2 card-of-mono1 [of {a1,a2}] card-of-bool [of a1 a2]
  by (blast intro: ordIso-ordLeq-trans)
have |A <+> B| ≤o |B <+> B|
  using LEQ card-of-Plus-mono1 by blast
moreover have |B <+> B| =o |B × (UNIV::bool set)|
  using card-of-Plus-Times-bool by blast
moreover have |B × (UNIV::bool set)| ≤o |B × A|
  using 1 by (simp add: card-of-Times-mono2)
moreover have |B × A| =o |A × B|
  using card-of-Times-commute by blast
ultimately show |A <+> B| ≤o |A × B|
  by (blast intro: ordLeq-transitive dest: ordLeq-ordIso-trans)
qed

```

lemma *card-of-Plus-Times*:

```

  assumes A2: a1 ≠ a2 ∧ {a1,a2} ≤ A and B2: b1 ≠ b2 ∧ {b1,b2} ≤ B
  shows |A <+> B| ≤o |A × B|
proof -
  {assume |A| ≤o |B|
   hence ?thesis using assms by (auto simp add: card-of-Plus-Times-aux)
  }
  moreover
  {assume |B| ≤o |A|
   hence |B <+> A| ≤o |B × A|
   using assms by (auto simp add: card-of-Plus-Times-aux)
   hence ?thesis
   using card-of-Plus-commute card-of-Times-commute
   ordIso-ordLeq-trans ordLeq-ordIso-trans by blast
  }
  ultimately show ?thesis
  using card-of-Well-order [of A] card-of-Well-order [of B]
  ordLeq-total [of |A|] by blast
qed

```

lemma *card-of-Times-Plus-distrib*:

```

|A × (B <+> C)| =o |A × B <+> A × C| (is |?RHS| =o |?LHS|)
proof -
  let ?f = λ(a, bc). case bc of Inl b ⇒ Inl (a, b) | Inr c ⇒ Inr (a, c)
  have bij-betw ?f ?RHS ?LHS unfolding bij-betw-def inj-on-def by force
  thus ?thesis using card-of-ordIso by blast
qed

```

lemma *card-of-ordLeq-finite*:

```

  assumes |A| ≤o |B| and finite B
  shows finite A
  using assms unfolding ordLeq-def
  using embed-inj-on [of |A| |B|] embed-Field [of |A| |B|]
  Field-card-of [of A] Field-card-of [of B] inj-on-finite [of - A B] by fastforce

```

lemma *card-of-ordLeq-infinite*:
assumes $|A| \leq_o |B|$ **and** \neg *finite* A
shows \neg *finite* B
using *assms card-of-ordLeq-finite* **by** *auto*

lemma *card-of-ordIso-finite*:
assumes $|A| =_o |B|$
shows *finite* $A =$ *finite* B
using *assms unfolding ordIso-def iso-def[abs-def]*
by (*auto simp: bij-betw-finite Field-card-of*)

lemma *card-of-ordIso-finite-Field*:
assumes *Card-order* r **and** $r =_o |A|$
shows *finite*(*Field* r) = *finite* A
using *assms card-of-Field-ordIso card-of-ordIso-finite ordIso-equivalence* **by** *blast*

30.4 Cardinals versus set operations involving infinite sets

Here we show that, for infinite sets, most set-theoretic constructions do not increase the cardinality. The cornerstone for this is theorem *Card-order-Times-same-infinite*, which states that self-product does not increase cardinality – the proof of this fact adapts a standard set-theoretic argument, as presented, e.g., in the proof of theorem 1.5.11 at page 47 in [4]. Then everything else follows fairly easily.

lemma *infinite-iff-card-of-nat*:
 \neg *finite* $A \iff (|UNIV::nat\ set| \leq_o |A|)$
unfolding *infinite-iff-countable-subset card-of-ordLeq ..*

The next two results correspond to the ZF fact that all infinite cardinals are limit ordinals:

lemma *Card-order-infinite-not-under*:
assumes *CARD*: *Card-order* r **and** *INF*: \neg *finite* (*Field* r)
shows \neg ($\exists a.$ *Field* $r =$ *under* r a)
proof(*auto*)
have 0 : *Well-order* $r \wedge$ *wo-rel* $r \wedge$ *Refl* r
using *CARD unfolding wo-rel-def card-order-on-def order-on-defs* **by** *auto*
fix a **assume** $*$: *Field* $r =$ *under* r a
show *False*
proof(*cases* $a \in$ *Field* r)
assume *Case1*: $a \notin$ *Field* r
hence *under* r $a = \{\}$ **unfolding** *Field-def under-def* **by** *auto*
thus *False* **using** *INF ** **by** *auto*
next
let $?r' =$ *Restr* r (*underS* r a)
assume *Case2*: $a \in$ *Field* r
hence 1 : *under* r $a =$ *underS* r $a \cup \{a\} \wedge$ $a \notin$ *underS* r a
using 0 *Refl-under-underS[of r a] underS-notIn[of a r]* **by** *blast*
have 2 : *wo-rel.ofilter* r (*underS* r a) \wedge *underS* r $a <$ *Field* r

using 0 *wo-rel.underS-ofilter* * 1 *Case2* **by** *fast*
hence $?r' <_o r$ **using** 0 **using** *ofilter-ordLess* **by** *blast*
moreover
have *Field* $?r' = \text{underS } r \ a \wedge \text{Well-order } ?r'$
using 2 0 *Field-Restr-ofilter*[of r] *Well-order-Restr*[of r] **by** *blast*
ultimately have $|\text{underS } r \ a| <_o r$ **using** *ordLess-Field*[of $?r'$] **by** *auto*
moreover have $|\text{under } r \ a| =_o r$ **using** * *CARD card-of-Field-ordIso*[of r] **by**
auto
ultimately have $|\text{underS } r \ a| <_o |\text{under } r \ a|$
using *ordIso-symmetric ordLess-ordIso-trans* **by** *blast*
moreover
{ **have** $\exists f. \text{bij-betw } f \ (\text{under } r \ a) \ (\text{underS } r \ a)$
using *infinite-imp-bij-betw*[of *Field* $r \ a$] *INF* * 1 **by** *auto*
hence $|\text{under } r \ a| =_o |\text{underS } r \ a|$ **using** *card-of-ordIso* **by** *blast*
}

ultimately show *False* **using** *not-ordLess-ordIso ordIso-symmetric* **by** *blast*
qed
qed

lemma *infinite-Card-order-limit*:

assumes r : *Card-order* r **and** $\neg \text{finite}$ (*Field* r)
and a : $a \in \text{Field } r$
shows $\exists b \in \text{Field } r. a \neq b \wedge (a,b) \in r$
proof –
have *Field* $r \neq \text{under } r \ a$
using *assms Card-order-infinite-not-under* **by** *blast*
moreover have $\text{under } r \ a \leq \text{Field } r$
using *under-Field* .
ultimately obtain b **where** $b \in \text{Field } r \wedge \neg (b,a) \in r$
unfolding *under-def* **by** *blast*
moreover have $ba: b \neq a$
using $b \ r$ **unfolding** *card-order-on-def well-order-on-def*
linear-order-on-def partial-order-on-def preorder-on-def refl-on-def **by** *auto*
ultimately have $(a,b) \in r$
using $a \ r$ **unfolding** *card-order-on-def well-order-on-def linear-order-on-def*
total-on-def **by** *blast*
thus *?thesis* **using** $b \ ba$ **by** *auto*
qed

theorem *Card-order-Times-same-infinite*:

assumes *CO*: *Card-order* r **and** *INF*: $\neg \text{finite}$ (*Field* r)
shows $|\text{Field } r \times \text{Field } r| \leq_o r$
proof –
define phi **where**
 $\text{phi} \equiv \lambda r::'a \ \text{rel}. \text{Card-order } r \wedge \neg \text{finite}(\text{Field } r) \wedge \neg |\text{Field } r \times \text{Field } r| \leq_o r$
have *temp1*: $\forall r. \text{phi } r \longrightarrow \text{Well-order } r$
unfolding *phi-def card-order-on-def* **by** *auto*
have *Ft*: $\neg(\exists r. \text{phi } r)$
proof

assume $\exists r. \text{phi } r$
hence $\{r. \text{phi } r\} \neq \{\} \wedge \{r. \text{phi } r\} \leq \{r. \text{Well-order } r\}$
using *temp1* **by** *auto*
then obtain r **where** $1: \text{phi } r$ **and** $2: \forall r'. \text{phi } r' \longrightarrow r \leq_o r'$ **and**
 $3: \text{Card-order } r \wedge \text{Well-order } r$
using *exists-minim-Well-order*[of $\{r. \text{phi } r\}$] *temp1* *phi-def* **by** *blast*
let $?A = \text{Field } r$ **let** $?r' = \text{bsqr } r$
have $4: \text{Well-order } ?r' \wedge \text{Field } ?r' = ?A \times ?A \wedge |?A| =_o r$
using 3 *bsqr-Well-order* *Field-bsqr* *card-of-Field-ordIso* **by** *blast*
have $5: \text{Card-order } |?A \times ?A| \wedge \text{Well-order } |?A \times ?A|$
using *card-of-Card-order* *card-of-Well-order* **by** *blast*

have $r <_o |?A \times ?A|$
using 1 3 5 *ordLess-or-ordLeq* **unfolding** *phi-def* **by** *blast*
moreover have $|?A \times ?A| \leq_o ?r'$
using *card-of-least*[of $?A \times ?A$] 4 **by** *auto*
ultimately have $r <_o ?r'$ **using** *ordLess-ordLeq-trans* **by** *auto*
then obtain f **where** $6: \text{embed } r ?r' f$ **and** $7: \neg \text{bij-betw } f ?A (?A \times ?A)$
unfolding *ordLess-def* *embedS-def*[*abs-def*]
by (*auto simp add: Field-bsqr*)
let $?B = f ' ?A$
have $|?A| =_o |?B|$
using 3 6 *embed-inj-on* *inj-on-imp-bij-betw* *card-of-ordIso* **by** *blast*
hence $8: r =_o |?B|$ **using** 4 *ordIso-transitive* *ordIso-symmetric* **by** *blast*

have *wo-rel.ofilter* $?r' ?B$
using 6 *embed-Field-ofilter* 3 4 **by** *blast*
hence *wo-rel.ofilter* $?r' ?B \wedge ?B \neq ?A \times ?A \wedge ?B \neq \text{Field } ?r'$
using 7 **unfolding** *bij-betw-def* **using** 6 3 *embed-inj-on* 4 **by** *auto*
hence *temp2*: *wo-rel.ofilter* $?r' ?B \wedge ?B < ?A \times ?A$
using 4 *wo-rel-def*[of $?r'$] *wo-rel.ofilter-def*[of $?r' ?B$] **by** *blast*
have $\neg (\exists a. \text{Field } r = \text{under } r a)$
using 1 **unfolding** *phi-def* **using** *Card-order-infinite-not-under*[of r] **by** *auto*
then obtain $A1$ **where** *temp3*: *wo-rel.ofilter* $r A1 \wedge A1 < ?A$ **and** $9: ?B \leq A1 \times A1$
using *temp2* 3 *bsqr-ofilter*[of $r ?B$] **by** *blast*
hence $|?B| \leq_o |A1 \times A1|$ **using** *card-of-mono1* **by** *blast*
hence $10: r \leq_o |A1 \times A1|$ **using** 8 *ordIso-ordLeq-trans* **by** *blast*
let $?r1 = \text{Restr } r A1$
have $?r1 <_o r$ **using** *temp3* *ofilter-ordLess* 3 **by** *blast*
moreover
{ **have** *well-order-on* $A1 ?r1$ **using** 3 *temp3* *well-order-on-Restr* **by** *blast*
hence $|A1| \leq_o ?r1$ **using** 3 *Well-order-Restr* *card-of-least* **by** *blast*
}
ultimately have $11: |A1| <_o r$ **using** *ordLeq-ordLess-trans* **by** *blast*

have $\neg \text{finite } (\text{Field } r)$ **using** 1 **unfolding** *phi-def* **by** *simp*
hence $\neg \text{finite } ?B$ **using** 8 3 *card-of-ordIso-finite-Field*[of $r ?B$] **by** *blast*
hence $\neg \text{finite } A1$ **using** 9 *finite-cartesian-product* *finite-subset* **by** *blast*

moreover have $temp_4$: $Field\ |A1| = A1 \wedge Well\text{-}order\ |A1| \wedge Card\text{-}order\ |A1|$
using $card\text{-}of\text{-}Card\text{-}order[of\ A1]$ $card\text{-}of\text{-}Well\text{-}order[of\ A1]$
by ($simp\ add$: $Field\text{-}card\text{-}of$)
moreover have $\neg r \leq_o |A1|$
using $temp_4\ 11\ 3$ **using** $not\text{-}ordLeq\text{-}iff\text{-}ordLess$ **by** $blast$
ultimately have $\neg finite(Field\ |A1|) \wedge Card\text{-}order\ |A1| \wedge \neg r \leq_o |A1|$
by ($simp\ add$: $card\text{-}of\text{-}card\text{-}order\text{-}on$)
hence $|Field\ |A1| \times Field\ |A1| | \leq_o |A1|$
using 2 **unfolding** $phi\text{-}def$ **by** $blast$
hence $|A1 \times A1| \leq_o |A1|$ **using** $temp_4$ **by** $auto$
hence $r \leq_o |A1|$ **using** $10\ ordLeq\text{-}transitive$ **by** $blast$
thus $False$ **using** $11\ not\text{-}ordLess\text{-}ordLeq$ **by** $auto$
qed
thus $?thesis$ **using** $assms$ **unfolding** $phi\text{-}def$ **by** $blast$
qed

corollary $card\text{-}of\text{-}Times\text{-}same\text{-}infinite$:

assumes $\neg finite\ A$
shows $|A \times A| =_o |A|$
proof –
let $?r = |A|$
have $Field\ ?r = A \wedge Card\text{-}order\ ?r$
using $Field\text{-}card\text{-}of\ card\text{-}of\text{-}Card\text{-}order[of\ A]$ **by** $fastforce$
hence $|A \times A| \leq_o |A|$
using $Card\text{-}order\text{-}Times\text{-}same\text{-}infinite[of\ ?r]$ $assms$ **by** $auto$
thus $?thesis$ **using** $card\text{-}of\text{-}Times3\ ordIso\text{-}iff\text{-}ordLeq$ **by** $blast$
qed

lemma $card\text{-}of\text{-}Times\text{-}infinite$:

assumes INF : $\neg finite\ A$ **and** NE : $B \neq \{\}$ **and** LEQ : $|B| \leq_o |A|$
shows $|A \times B| =_o |A| \wedge |B \times A| =_o |A|$
proof –
have $|A| \leq_o |A \times B| \wedge |A| \leq_o |B \times A|$
using $assms$ **by** ($simp\ add$: $card\text{-}of\text{-}Times1\ card\text{-}of\text{-}Times2$)
moreover
{ **have** $|A \times B| \leq_o |A \times A| \wedge |B \times A| \leq_o |A \times A|$
using $LEQ\ card\text{-}of\text{-}Times\text{-}mono1\ card\text{-}of\text{-}Times\text{-}mono2$ **by** $blast$
moreover have $|A \times A| =_o |A|$ **using** $INF\ card\text{-}of\text{-}Times\text{-}same\text{-}infinite$ **by**
 $blast$
ultimately have $|A \times B| \leq_o |A| \wedge |B \times A| \leq_o |A|$
using $ordLeq\text{-}ordIso\text{-}trans[of\ |A \times B|]$ $ordLeq\text{-}ordIso\text{-}trans[of\ |B \times A|]$ **by**
 $auto$
}
ultimately show $?thesis$ **by** ($simp\ add$: $ordIso\text{-}iff\text{-}ordLeq$)
qed

corollary $Card\text{-}order\text{-}Times\text{-}infinite$:

assumes INF : $\neg finite(Field\ r)$ **and** $CARD$: $Card\text{-}order\ r$ **and**
 NE : $Field\ p \neq \{\}$ **and** LEQ : $p \leq_o r$

shows $|(\text{Field } r) \times (\text{Field } p)| =_o r \wedge |(\text{Field } p) \times (\text{Field } r)| =_o r$
proof –
have $|\text{Field } r \times \text{Field } p| =_o |\text{Field } r| \wedge |\text{Field } p \times \text{Field } r| =_o |\text{Field } r|$
using *assms* **by** (*simp add: card-of-Times-infinite card-of-mono2*)
thus *?thesis*
using *assms card-of-Field-ordIso* **by** (*blast intro: ordIso-transitive*)
qed

lemma *card-of-Sigma-ordLeq-infinite*:

assumes *INF*: $\neg \text{finite } B$ **and**
LEQ-I: $|I| \leq_o |B|$ **and** *LEQ*: $\forall i \in I. |A \ i| \leq_o |B|$
shows $|\text{SIGMA } i : I. A \ i| \leq_o |B|$
proof(*cases I = {}*)
case *False*
have $|\text{SIGMA } i : I. A \ i| \leq_o |I \times B|$
using *card-of-Sigma-mono1 [OF LEQ]* **by** *blast*
moreover **have** $|I \times B| =_o |B|$
using *INF False LEQ-I* **by** (*auto simp add: card-of-Times-infinite*)
ultimately show *?thesis* **using** *ordLeq-ordIso-trans* **by** *blast*
qed (*simp add: card-of-empty*)

lemma *card-of-Sigma-ordLeq-infinite-Field*:

assumes *INF*: $\neg \text{finite } (\text{Field } r)$ **and** *r*: *Card-order r* **and**
LEQ-I: $|I| \leq_o r$ **and** *LEQ*: $\forall i \in I. |A \ i| \leq_o r$
shows $|\text{SIGMA } i : I. A \ i| \leq_o r$
proof –
let *?B* = *Field r*
have *1*: $r =_o |?B| \wedge |?B| =_o r$
using *r card-of-Field-ordIso ordIso-symmetric* **by** *blast*
hence $|I| \leq_o |?B| \ \forall i \in I. |A \ i| \leq_o |?B|$
using *LEQ-I LEQ ordLeq-ordIso-trans* **by** *blast+*
hence $|\text{SIGMA } i : I. A \ i| \leq_o |?B|$ **using** *INF LEQ*
card-of-Sigma-ordLeq-infinite **by** *blast*
thus *?thesis* **using** *1 ordLeq-ordIso-trans* **by** *blast*
qed

lemma *card-of-Times-ordLeq-infinite-Field*:

$[\neg \text{finite } (\text{Field } r); |A| \leq_o r; |B| \leq_o r; \text{Card-order } r] \implies |A \times B| \leq_o r$
by (*simp add: card-of-Sigma-ordLeq-infinite-Field*)

lemma *card-of-Times-infinite-simps*:

$[\neg \text{finite } A; B \neq \{\}; |B| \leq_o |A|] \implies |A \times B| =_o |A|$
 $[\neg \text{finite } A; B \neq \{\}; |B| \leq_o |A|] \implies |A| =_o |A \times B|$
 $[\neg \text{finite } A; B \neq \{\}; |B| \leq_o |A|] \implies |B \times A| =_o |A|$
 $[\neg \text{finite } A; B \neq \{\}; |B| \leq_o |A|] \implies |A| =_o |B \times A|$
by (*auto simp add: card-of-Times-infinite ordIso-symmetric*)

lemma *card-of-UNION-ordLeq-infinite*:

assumes *INF*: $\neg \text{finite } B$ **and** *LEQ-I*: $|I| \leq_o |B|$ **and** *LEQ*: $\forall i \in I. |A \ i| \leq_o |B|$

shows $|\bigcup i \in I. A\ i| \leq_o |B|$
proof(cases $I = \{\}$)
 case *False*
 have $|\bigcup i \in I. A\ i| \leq_o |\text{SIGMA } i : I. A\ i|$
 using *card-of-UNION-Sigma* by *blast*
 moreover have $|\text{SIGMA } i : I. A\ i| \leq_o |B|$
 using *assms card-of-Sigma-ordLeq-infinite* by *blast*
 ultimately show *?thesis* using *ordLeq-transitive* by *blast*
qed (*simp add: card-of-empty*)

corollary *card-of-UNION-ordLeq-infinite-Field*:
 assumes *INF*: \neg *finite* (*Field* r) and r : *Card-order* r and
LEQ-I: $|I| \leq_o r$ and *LEQ*: $\forall i \in I. |A\ i| \leq_o r$
 shows $|\bigcup i \in I. A\ i| \leq_o r$
proof –
 let $?B = \text{Field } r$
 have $1: r =_o |?B| \wedge |?B| =_o r$
 using r *card-of-Field-ordIso ordIso-symmetric* by *blast*
 hence $|I| \leq_o |?B| \ \forall i \in I. |A\ i| \leq_o |?B|$
 using *LEQ-I LEQ ordLeq-ordIso-trans* by *blast+*
 hence $|\bigcup i \in I. A\ i| \leq_o |?B|$ using *INF LEQ*
card-of-UNION-ordLeq-infinite by *blast*
 thus *?thesis* using 1 *ordLeq-ordIso-trans* by *blast*
qed

lemma *card-of-Plus-infinite1*:
 assumes *INF*: \neg *finite* A and *LEQ*: $|B| \leq_o |A|$
 shows $|A <+> B| =_o |A|$
proof(cases $B = \{\}$)
 case *True*
 then show *?thesis*
 by (*simp add: card-of-Plus-empty1 card-of-Plus-empty2 ordIso-symmetric*)
next
 case *False*
 let $?Inl = \text{Inl}::'a \Rightarrow 'a + 'b$ let $?Inr = \text{Inr}::'b \Rightarrow 'a + 'b$
 assume $*$: $B \neq \{\}$
 then obtain $b1$ where $1: b1 \in B$ by *blast*
 show *?thesis*
proof(cases $B = \{b1\}$)
 case *True*
 have $2: \text{bij-betw } ?Inl\ A\ ((?Inl\ 'A))$
 unfolding *bij-betw-def inj-on-def* by *auto*
 hence $3: \neg$ *finite* ($?Inl\ 'A$)
 using *INF bij-betw-finite[of ?Inl A]* by *blast*
 let $?A' = ?Inl\ 'A \cup \{?Inr\ b1\}$
 obtain g where *bij-betw* $g\ (?Inl\ 'A)\ ?A'$
 using 3 *infinite-imp-bij-betw2[of ?Inl 'A]* by *auto*
 moreover have $?A' = A <+> B$ using *True* by *blast*
 ultimately have *bij-betw* $g\ (?Inl\ 'A)\ (A <+> B)$ by *simp*

hence *bij-betw* ($g \circ ?Inl$) A ($A <+> B$)
 using 2 by (*auto simp add: bij-betw-trans*)
 thus *?thesis using card-of-ordIso ordIso-symmetric by blast*
 next
 case *False*
 with * 1 obtain $b2$ where 3: $b1 \neq b2 \wedge \{b1, b2\} \leq B$ by *fastforce*
 obtain f where *inj-on* $f B \wedge f ' B \leq A$
 using *LEQ card-of-ordLeq[of B]* by *fastforce*
 with 3 have $f b1 \neq f b2 \wedge \{f b1, f b2\} \leq A$
 unfolding *inj-on-def* by *auto*
 with 3 have $|A <+> B| \leq_o |A \times B|$
 by (*auto simp add: card-of-Plus-Times*)
 moreover have $|A \times B| =_o |A|$
 using *assms * by (simp add: card-of-Times-infinite-simps)*
 ultimately have $|A <+> B| \leq_o |A|$ using *ordLeq-ordIso-trans by blast*
 thus *?thesis using card-of-Plus1 ordIso-iff-ordLeq by blast*
 qed
 qed

lemma *card-of-Plus-infinite2*:
 assumes *INF*: $\neg finite A$ and *LEQ*: $|B| \leq_o |A|$
 shows $|B <+> A| =_o |A|$
 using *assms card-of-Plus-commute card-of-Plus-infinite1*
ordIso-equivalence by blast

lemma *card-of-Plus-infinite*:
 assumes *INF*: $\neg finite A$ and *LEQ*: $|B| \leq_o |A|$
 shows $|A <+> B| =_o |A| \wedge |B <+> A| =_o |A|$
 using *assms by (auto simp: card-of-Plus-infinite1 card-of-Plus-infinite2)*

corollary *Card-order-Plus-infinite*:
 assumes *INF*: $\neg finite(Field r)$ and *CARD*: *Card-order* r and
LEQ: $p \leq_o r$
 shows $|(Field r) <+> (Field p)| =_o r \wedge |(Field p) <+> (Field r)| =_o r$
proof –
 have $|Field r <+> Field p| =_o |Field r| \wedge$
 $|Field p <+> Field r| =_o |Field r|$
 using *assms by (simp add: card-of-Plus-infinite card-of-mono2)*
 thus *?thesis*
 using *assms card-of-Field-ordIso by (blast intro: ordIso-transitive)*

qed

30.5 The cardinal ω and the finite cardinals

The cardinal ω , of natural numbers, shall be the standard non-strict order relation on *nat*, that we abbreviate by *natLeq*. The finite cardinals shall be the restrictions of these relations to the numbers smaller than fixed numbers n , that we abbreviate by *natLeq-on n*.

definition $(\text{natLeq}::(\text{nat} * \text{nat}) \text{ set}) \equiv \{(x,y). x \leq y\}$

definition $(\text{natLess}::(\text{nat} * \text{nat}) \text{ set}) \equiv \{(x,y). x < y\}$

abbreviation $\text{natLeq-on} :: \text{nat} \Rightarrow (\text{nat} * \text{nat}) \text{ set}$

where $\text{natLeq-on } n \equiv \{(x,y). x < n \wedge y < n \wedge x \leq y\}$

lemma *infinite-cartesian-product*:

assumes $\neg \text{finite } A \ \neg \text{finite } B$

shows $\neg \text{finite } (A \times B)$

using *assms finite-cartesian-productD2* **by** *auto*

30.5.1 First as well-orders

lemma *Field-natLeq*: $\text{Field } \text{natLeq} = (\text{UNIV}::\text{nat } \text{set})$

by *(unfold Field-def natLeq-def, auto)*

lemma *natLeq-Refl*: $\text{Refl } \text{natLeq}$

unfolding *refl-on-def Field-def natLeq-def* **by** *auto*

lemma *natLeq-trans*: $\text{trans } \text{natLeq}$

unfolding *trans-def natLeq-def* **by** *auto*

lemma *natLeq-Preorder*: $\text{Preorder } \text{natLeq}$

unfolding *preorder-on-def*

by *(auto simp add: natLeq-Refl natLeq-trans)*

lemma *natLeq-antisym*: $\text{antisym } \text{natLeq}$

unfolding *antisym-def natLeq-def* **by** *auto*

lemma *natLeq-Partial-order*: $\text{Partial-order } \text{natLeq}$

unfolding *partial-order-on-def*

by *(auto simp add: natLeq-Preorder natLeq-antisym)*

lemma *natLeq-Total*: $\text{Total } \text{natLeq}$

unfolding *total-on-def natLeq-def* **by** *auto*

lemma *natLeq-Linear-order*: $\text{Linear-order } \text{natLeq}$

unfolding *linear-order-on-def*

by *(auto simp add: natLeq-Partial-order natLeq-Total)*

lemma *natLeq-natLess-Id*: $\text{natLess} = \text{natLeq} - \text{Id}$

unfolding *natLeq-def natLess-def* **by** *auto*

lemma *natLeq-Well-order*: $\text{Well-order } \text{natLeq}$

unfolding *well-order-on-def*

using *natLeq-Linear-order wf-less natLeq-natLess-Id natLeq-def natLess-def* **by** *auto*

lemma *Field-natLeq-on*: $\text{Field } (\text{natLeq-on } n) = \{x. x < n\}$

unfolding *Field-def* **by** *auto*

lemma *natLeq-underS-less*: $\text{underS natLeq } n = \{x. x < n\}$
unfolding *underS-def natLeq-def* **by** *auto*

lemma *Restr-natLeq*: $\text{Restr natLeq } \{x. x < n\} = \text{natLeq-on } n$
unfolding *natLeq-def* **by** *force*

lemma *Restr-natLeq2*:
 $\text{Restr natLeq } (\text{underS natLeq } n) = \text{natLeq-on } n$
by (*auto simp add: Restr-natLeq natLeq-underS-less*)

lemma *natLeq-on-Well-order*: $\text{Well-order}(\text{natLeq-on } n)$
using *Restr-natLeq[of n] natLeq-Well-order*
 $\text{Well-order-Restr}[of \text{natLeq } \{x. x < n\}]$ **by** *auto*

corollary *natLeq-on-well-order-on*: $\text{well-order-on } \{x. x < n\} (\text{natLeq-on } n)$
using *natLeq-on-Well-order Field-natLeq-on* **by** *auto*

lemma *natLeq-on-wo-rel*: $\text{wo-rel}(\text{natLeq-on } n)$
unfolding *wo-rel-def* **using** *natLeq-on-Well-order* .

30.5.2 Then as cardinals

lemma *natLeq-Card-order*: Card-order natLeq

proof –

have $\text{natLeq-on } n < o \mid \text{UNIV}::\text{nat set} \mid$ **for** n

proof –

have $\text{finite}(\text{Field } (\text{natLeq-on } n))$ **by** (*auto simp: Field-def*)

moreover have $\neg \text{finite}(\text{UNIV}::\text{nat set})$ **by** *auto*

ultimately show *?thesis*

using $\text{finite-ordLess-infinite}[of \text{natLeq-on } n \mid \text{UNIV}::\text{nat set}]$

$\text{card-of-Well-order}[of \text{UNIV}::\text{nat set}] \text{ natLeq-on-Well-order}$

by (*force simp add: Field-card-of*)

qed

then show *?thesis*

apply (*simp add: natLeq-Well-order Card-order-iff-Restr-underS Restr-natLeq2*)

apply (*force simp add: Field-natLeq*)

done

qed

corollary *card-of-Field-natLeq*:

$\mid \text{Field natLeq} \mid = o \text{ natLeq}$

using $\text{Field-natLeq natLeq-Card-order Card-order-iff-ordIso-card-of}[of \text{natLeq}]$

$\text{ordIso-symmetric}[of \text{natLeq}]$ **by** *blast*

corollary *card-of-nat*:

$\mid \text{UNIV}::\text{nat set} \mid = o \text{ natLeq}$

using $\text{Field-natLeq card-of-Field-natLeq}$ **by** *auto*

corollary *infinite-iff-natLeq-ordLeq*:

$\neg \text{finite } A = (\text{natLeq } \leq_o |A|)$

using *infinite-iff-card-of-nat*[of A] *card-of-nat*

ordIso-ordLeq-trans ordLeq-ordIso-trans ordIso-symmetric **by** *blast*

corollary *finite-iff-ordLess-natLeq*:

$\text{finite } A = (|A| <_o \text{natLeq})$

using *infinite-iff-natLeq-ordLeq not-ordLeq-iff-ordLess*

card-of-Well-order natLeq-Well-order **by** *blast*

30.6 The successor of a cardinal

First we define *isCardSuc* r r' , the notion of r' being a successor cardinal of r . Although the definition does not require r to be a cardinal, only this case will be meaningful.

definition *isCardSuc* $:: 'a$ *rel* $\Rightarrow 'a$ *set rel* $\Rightarrow \text{bool}$

where

isCardSuc r $r' \equiv$

Card-order $r' \wedge r <_o r' \wedge$

$(\forall (r''::'a \text{ set rel}). \text{Card-order } r'' \wedge r <_o r'' \longrightarrow r' \leq_o r'')$

Now we introduce the cardinal-successor operator *cardSuc*, by picking *some* cardinal-order relation fulfilling *isCardSuc*. Again, the picked item shall be proved unique up to order-isomorphism.

definition *cardSuc* $:: 'a$ *rel* $\Rightarrow 'a$ *set rel*

where *cardSuc* $r \equiv \text{SOME } r'. \text{isCardSuc } r$ r'

lemma *exists-minim-Card-order*:

$\llbracket R \neq \{\}; \forall r \in R. \text{Card-order } r \rrbracket \implies \exists r \in R. \forall r' \in R. r \leq_o r'$

unfolding *card-order-on-def* **using** *exists-minim-Well-order* **by** *blast*

lemma *exists-isCardSuc*:

assumes *Card-order* r

shows $\exists r'. \text{isCardSuc } r$ r'

proof –

let $?R = \{(r'::'a \text{ set rel}). \text{Card-order } r' \wedge r <_o r'\}$

have $|Pow(\text{Field } r)| \in ?R \wedge (\forall r \in ?R. \text{Card-order } r)$ **using** *assms*

by (*simp add: card-of-Card-order Card-order-Pow*)

then obtain r **where** $r \in ?R \wedge (\forall r' \in ?R. r \leq_o r')$

using *exists-minim-Card-order*[of $?R$] **by** *blast*

thus *thesis* **unfolding** *isCardSuc-def* **by** *auto*

qed

lemma *cardSuc-isCardSuc*:

assumes *Card-order* r

shows $\text{isCardSuc } r$ (*cardSuc* r)

unfolding *cardSuc-def* **using** *assms*

by (*simp add: exists-isCardSuc someI-ex*)

lemma *cardSuc-Card-order*:

Card-order r \implies *Card-order*(*cardSuc r*)

using *cardSuc-isCardSuc unfolding isCardSuc-def* by *blast*

lemma *cardSuc-greater*:

Card-order r \implies *r < o cardSuc r*

using *cardSuc-isCardSuc unfolding isCardSuc-def* by *blast*

lemma *cardSuc-ordLeq*:

Card-order r \implies *r ≤ o cardSuc r*

using *cardSuc-greater ordLeq-iff-ordLess-or-ordIso* by *blast*

The minimality property of *cardSuc* originally present in its definition is local to the type *'a set rel*, i.e., that of *cardSuc r*:

lemma *cardSuc-least-aux*:

$\llbracket \text{Card-order } (r::'a \text{ rel}); \text{Card-order } (r'::'a \text{ set rel}); r < o r' \rrbracket \implies \text{cardSuc } r \leq o r'$

using *cardSuc-isCardSuc unfolding isCardSuc-def* by *blast*

But from this we can infer general minimality:

lemma *cardSuc-least*:

assumes *CARD*: *Card-order r* and *CARD'*: *Card-order r'* and *LESS*: *r < o r'*

shows *cardSuc r ≤ o r'*

proof –

let *?p = cardSuc r*

have *0*: *Well-order ?p* \wedge *Well-order r'*

using *assms cardSuc-Card-order unfolding card-order-on-def* by *blast*

{ assume *r' < o ?p*

then obtain *r''* where *1*: *Field r'' < Field ?p* and *2*: *r' = o r''* \wedge *r'' < o ?p*

using *internalize-ordLess[of r' ?p]* by *blast*

have *Card-order r''* using *CARD'* *Card-order-ordIso2 2* by *blast*

moreover have *r < o r''* using *LESS 2 ordLess-ordIso-trans* by *blast*

ultimately have *?p ≤ o r''* using *cardSuc-least-aux CARD* by *blast*

hence *False* using *2 not-ordLess-ordLeq* by *blast*

}

thus *?thesis* using *0 ordLess-or-ordLeq* by *blast*

qed

lemma *cardSuc-ordLess-ordLeq*:

assumes *CARD*: *Card-order r* and *CARD'*: *Card-order r'*

shows $(r < o r') = (\text{cardSuc } r \leq o r')$

proof

show *cardSuc r ≤ o r'* \implies *r < o r'*

using *assms cardSuc-greater ordLess-ordLeq-trans* by *blast*

qed (*auto simp add: assms cardSuc-least*)

lemma *cardSuc-ordLeq-ordLess*:

assumes *CARD*: Card-order r **and** *CARD'*: Card-order r'
shows $(r' <_o \text{cardSuc } r) = (r' \leq_o r)$
proof –
have $\text{Well-order } r \wedge \text{Well-order } r'$
using *assms unfolding card-order-on-def* **by** *auto*
moreover have $\text{Well-order}(\text{cardSuc } r)$
using *assms cardSuc-Card-order card-order-on-def* **by** *blast*
ultimately show *?thesis*
using *assms cardSuc-ordLess-ordLeq* **by** (*blast dest: not-ordLeq-iff-ordLess*)
qed

lemma *cardSuc-mono-ordLeq*:
assumes *CARD*: Card-order r **and** *CARD'*: Card-order r'
shows $(\text{cardSuc } r \leq_o \text{cardSuc } r') = (r \leq_o r')$
using *assms cardSuc-ordLeq-ordLess cardSuc-ordLess-ordLeq cardSuc-Card-order*
by *blast*

lemma *cardSuc-invar-ordIso*:
assumes *CARD*: Card-order r **and** *CARD'*: Card-order r'
shows $(\text{cardSuc } r =_o \text{cardSuc } r') = (r =_o r')$
proof –
have 0 : $\text{Well-order } r \wedge \text{Well-order } r' \wedge \text{Well-order}(\text{cardSuc } r) \wedge \text{Well-order}(\text{cardSuc } r')$
using *assms* **by** (*simp add: card-order-on-well-order-on cardSuc-Card-order*)
thus *?thesis*
using *ordIso-iff-ordLeq[of r r'] ordIso-iff-ordLeq*
using *cardSuc-mono-ordLeq[of r r'] cardSuc-mono-ordLeq[of r' r] assms* **by**
blast
qed

lemma *card-of-cardSuc-finite*:
 $\text{finite}(\text{Field}(\text{cardSuc } |A|)) = \text{finite } A$
proof
assume $*$: $\text{finite}(\text{Field}(\text{cardSuc } |A|))$
have 0 : $|\text{Field}(\text{cardSuc } |A|)| =_o \text{cardSuc } |A|$
using *card-of-Card-order cardSuc-Card-order card-of-Field-ordIso* **by** *blast*
hence $|A| \leq_o |\text{Field}(\text{cardSuc } |A|)|$
using *card-of-Card-order[of A] cardSuc-ordLeq[of |A|] ordIso-symmetric*
ordLeq-ordIso-trans **by** *blast*
thus $\text{finite } A$ **using** $*$ *card-of-ordLeq-finite* **by** *blast*
next
assume $\text{finite } A$
then have $\text{finite}(\text{Field } |\text{Pow } A|)$ **unfolding** *Field-card-of* **by** *simp*
moreover
have $\text{cardSuc } |A| \leq_o |\text{Pow } A|$
by (*rule iffD1[OF cardSuc-ordLess-ordLeq card-of-Pow]*) (*simp-all add: card-of-Card-order*)
ultimately show $\text{finite}(\text{Field}(\text{cardSuc } |A|))$
by (*blast intro: card-of-ordLeq-finite card-of-mono2*)
qed

lemma *cardSuc-finite*:
assumes *Card-order* *r*
shows $\text{finite} (\text{Field} (\text{cardSuc } r)) = \text{finite} (\text{Field } r)$
proof –
let $?A = \text{Field } r$
have $|?A| =_o r$ **using** *assms* **by** (*simp add: card-of-Field-ordIso*)
hence $\text{cardSuc } |?A| =_o \text{cardSuc } r$ **using** *assms*
by (*simp add: card-of-Card-order cardSuc-invar-ordIso*)
moreover **have** $|\text{Field} (\text{cardSuc } |?A|)| =_o \text{cardSuc } |?A|$
by (*simp add: card-of-card-order-on Field-card-of card-of-Field-ordIso card-Suc-Card-order*)
moreover
{ **have** $|\text{Field} (\text{cardSuc } r)| =_o \text{cardSuc } r$
using *assms* **by** (*simp add: card-of-Field-ordIso cardSuc-Card-order*)
hence $\text{cardSuc } r =_o |\text{Field} (\text{cardSuc } r)|$
using *ordIso-symmetric* **by** *blast*
}
ultimately **have** $|\text{Field} (\text{cardSuc } |?A|)| =_o |\text{Field} (\text{cardSuc } r)|$
using *ordIso-transitive* **by** *blast*
hence $\text{finite} (\text{Field} (\text{cardSuc } |?A|)) = \text{finite} (\text{Field} (\text{cardSuc } r))$
using *card-of-ordIso-finite* **by** *blast*
thus *?thesis* **by** (*simp only: card-of-cardSuc-finite*)
qed

lemma *Field-cardSuc-not-empty*:
assumes *Card-order* *r*
shows $\text{Field} (\text{cardSuc } r) \neq \{\}$
proof
assume $\text{Field} (\text{cardSuc } r) = \{\}$
then **have** $|\text{Field} (\text{cardSuc } r)| \leq_o r$ **using** *assms* *Card-order-empty[of r]* **by** *auto*
then **have** $\text{cardSuc } r \leq_o r$ **using** *assms* *card-of-Field-ordIso*
cardSuc-Card-order ordIso-symmetric ordIso-ordLeq-trans **by** *blast*
then **show** *False* **using** *cardSuc-greater not-ordLess-ordLeq* *assms* **by** *blast*
qed

typedef $'a \text{ suc} = \text{Field} (\text{cardSuc } |UNIV :: 'a \text{ set}|)$
using *Field-cardSuc-not-empty card-of-Card-order* **by** *blast*

definition *card-suc* :: $'a \text{ rel} \Rightarrow 'a \text{ suc rel}$ **where**
 $\text{card-suc} \equiv \lambda-. \text{map-prod Abs-suc Abs-suc } ' \text{cardSuc } |UNIV :: 'a \text{ set}|$

lemma *Field-card-suc*: $\text{Field} (\text{card-suc } r) = UNIV$

proof –
let $?r = \text{cardSuc } |UNIV|$
let $?ar = \lambda x. \text{Abs-suc} (\text{Rep-suc } x)$
have 1: $\bigwedge P. (\forall x \in \text{Field } ?r. P x) = (\forall x. P (\text{Rep-suc } x))$ **using** *Rep-suc-induct*
Rep-suc **by** *blast*
have 2: $\bigwedge P. (\exists x \in \text{Field } ?r. P x) = (\exists x. P (\text{Rep-suc } x))$ **using** *Rep-suc-cases*

Rep-suc **by** *blast*

have $\exists: \bigwedge A a b. (a, b) \in A \implies (Abs\text{-}suc\ a, Abs\text{-}suc\ b) \in map\text{-}prod\ Abs\text{-}suc\ Abs\text{-}suc\ 'A$ **unfolding** *map-prod-def* **by** *auto*

have $\forall x \in Field\ ?r. (\exists b \in Field\ ?r. (x, b) \in ?r) \vee (\exists a \in Field\ ?r. (a, x) \in ?r)$

unfolding *Field-def Range.simps Domain.simps Un-iff* **by** *blast*

then have $\forall x. (\exists b. (Rep\text{-}suc\ x, Rep\text{-}suc\ b) \in ?r) \vee (\exists a. (Rep\text{-}suc\ a, Rep\text{-}suc\ x) \in ?r)$ **unfolding** *1 2* .

then have $\forall x. (\exists b. (?ar\ x, ?ar\ b) \in map\text{-}prod\ Abs\text{-}suc\ Abs\text{-}suc\ ' ?r) \vee (\exists a. (?ar\ a, ?ar\ x) \in map\text{-}prod\ Abs\text{-}suc\ Abs\text{-}suc\ ' ?r)$ **using** \exists **by** *fast*

then have $\forall x. (\exists b. (x, b) \in card\text{-}suc\ r) \vee (\exists a. (a, x) \in card\text{-}suc\ r)$ **unfolding** *card-suc-def Rep-suc-inverse* .

then show *?thesis* **unfolding** *Field-def Domain.simps Range.simps set-eq-iff Un-iff eqTrueI[OF UNIV-I] ex-simps simp-thms* .

qed

lemma *card-suc-alt*: $card\text{-}suc\ r = dir\text{-}image\ (cardSuc\ |UNIV :: 'a\ set|)$ *Abs-suc*

unfolding *card-suc-def dir-image-def* **by** *auto*

lemma *cardSuc-Well-order*: $Card\text{-}order\ r \implies Well\text{-}order(cardSuc\ r)$

using *cardSuc-Card-order* **unfolding** *card-order-on-def* **by** *blast*

lemma *cardSuc-ordIso-card-suc*:

assumes *card-order* *r*

shows $cardSuc\ r =_o\ card\text{-}suc\ (r :: 'a\ rel)$

proof –

have $cardSuc\ (r :: 'a\ rel) =_o\ cardSuc\ |UNIV :: 'a\ set|$

using *cardSuc-invar-ordIso[THEN iffD2, OF - card-of-Card-order card-of-unique[OF assms]]* *assms*

by (*simp add: card-order-on-Card-order*)

also have $cardSuc\ |UNIV :: 'a\ set| =_o\ card\text{-}suc\ (r :: 'a\ rel)$

unfolding *card-suc-alt*

by (*rule dir-image-ordIso*) (*simp-all add: inj-on-def Abs-suc-inject cardSuc-Well-order card-of-Card-order*)

finally show *?thesis* .

qed

lemma *Card-order-card-suc*: $card\text{-}order\ r \implies Card\text{-}order\ (card\text{-}suc\ r)$

using *cardSuc-Card-order[THEN Card-order-ordIso2[OF - cardSuc-ordIso-card-suc]]* *card-order-on-Card-order* **by** *blast*

lemma *card-order-card-suc*: $card\text{-}order\ r \implies card\text{-}order\ (card\text{-}suc\ r)$

using *Card-order-card-suc arg-cong2[OF Field-card-suc refl, of card-order-on]* **by** *blast*

lemma *card-suc-greater*: $card\text{-}order\ r \implies r <_o\ card\text{-}suc\ r$

using *ordLess-ordIso-trans[OF cardSuc-greater cardSuc-ordIso-card-suc]* *card-order-on-Card-order* **by** *blast*

lemma *card-of-Plus-ordLess-infinite*:

assumes *INF*: \neg *finite* *C* **and** *LESS1*: $|A| <_o |C|$ **and** *LESS2*: $|B| <_o |C|$
shows $|A <+> B| <_o |C|$
proof (*cases* $A = \{\} \vee B = \{\}$)
 case *True*
 hence $|A| =_o |A <+> B| \vee |B| =_o |A <+> B|$
 using *card-of-Plus-empty1* *card-of-Plus-empty2* **by** *blast*
 hence $|A <+> B| =_o |A| \vee |A <+> B| =_o |B|$
 using *ordIso-symmetric*[*of* $|A|$] *ordIso-symmetric*[*of* $|B|$] **by** *blast*
 thus *?thesis* **using** *LESS1* *LESS2*
 ordIso-ordLess-trans[*of* $|A <+> B|$ $|A|$]
 ordIso-ordLess-trans[*of* $|A <+> B|$ $|B|$] **by** *blast*
 next
 case *False*
 have *False* **if** $|C| \leq_o |A <+> B|$
 proof –
 have \S : \neg *finite* $A \vee \neg$ *finite* B
 using *that* *INF* *card-of-ordLeq-finite* *finite-Plus* **by** *blast*
 consider $|A| \leq_o |B| \mid |B| \leq_o |A|$
 using *ordLeq-total* [*OF* *card-of-Well-order* *card-of-Well-order*] **by** *blast*
 then show *False*
 proof *cases*
 case *1*
 hence \neg *finite* B **using** \S *card-of-ordLeq-finite* **by** *blast*
 hence $|A <+> B| =_o |B|$ **using** *False 1*
 by (*auto simp add: card-of-Plus-infinite*)
 thus *False* **using** *LESS2* *not-ordLess-ordLeq* *that* *ordLeq-ordIso-trans* **by** *blast*
 next
 case *2*
 hence \neg *finite* A **using** \S *card-of-ordLeq-finite* **by** *blast*
 hence $|A <+> B| =_o |A|$ **using** *False 2*
 by (*auto simp add: card-of-Plus-infinite*)
 thus *False* **using** *LESS1* *not-ordLess-ordLeq* *that* *ordLeq-ordIso-trans* **by** *blast*
 qed
 qed
 thus *?thesis*
 using *ordLess-or-ordLeq*[*of* $|A <+> B|$ $|C|$]
 card-of-Well-order[*of* $A <+> B$] *card-of-Well-order*[*of* C] **by** *auto*
 qed

lemma *card-of-Plus-ordLess-infinite-Field*:
assumes *INF*: \neg *finite* (*Field* r) **and** r : *Card-order* r **and**
 LESS1: $|A| <_o r$ **and** *LESS2*: $|B| <_o r$
shows $|A <+> B| <_o r$
proof –
 let $?C =$ *Field* r
 have 1 : $r =_o |?C| \wedge |?C| =_o r$
 using r *card-of-Field-ordIso* *ordIso-symmetric* **by** *blast*
 hence $|A| <_o |?C| \mid |B| <_o |?C|$
 using *LESS1* *LESS2* *ordLess-ordIso-trans* **by** *blast+*

hence $|A <+> B| <o |?C|$ **using** *INF*
card-of-Plus-ordLess-infinite **by** *blast*
thus *?thesis* **using** *1 ordLess-ordIso-trans* **by** *blast*
qed

lemma *card-of-Plus-ordLeq-infinite-Field*:
assumes $r: \neg\text{finite}(\text{Field } r)$ **and** $A: |A| \leq_o r$ **and** $B: |B| \leq_o r$
and $c: \text{Card-order } r$
shows $|A <+> B| \leq_o r$
proof –
let $?r' = \text{cardSuc } r$
have *Card-order* $?r' \wedge \neg\text{finite}(\text{Field } ?r')$ **using** *assms*
by (*simp add: cardSuc-Card-order cardSuc-finite*)
moreover **have** $|A| <o ?r'$ **and** $|B| <o ?r'$ **using** $A B c$
by (*auto simp: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess*)
ultimately **have** $|A <+> B| <o ?r'$
using *card-of-Plus-ordLess-infinite-Field* **by** *blast*
thus *?thesis* **using** $c r$
by (*simp add: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess*)
qed

lemma *card-of-Un-ordLeq-infinite-Field*:
assumes $C: \neg\text{finite}(\text{Field } r)$ **and** $A: |A| \leq_o r$ **and** $B: |B| \leq_o r$
and *Card-order* r
shows $|A \text{ Un } B| \leq_o r$
using *assms card-of-Plus-ordLeq-infinite-Field card-of-Un-Plus-ordLeq*
ordLeq-transitive **by** *fast*

lemma *card-of-Un-ordLess-infinite*:
assumes *INF*: $\neg\text{finite } C$ **and**
LESS1: $|A| <o |C|$ **and** *LESS2*: $|B| <o |C|$
shows $|A \cup B| <o |C|$
using *assms card-of-Plus-ordLess-infinite card-of-Un-Plus-ordLeq*
ordLeq-ordLess-trans **by** *blast*

lemma *card-of-Un-ordLess-infinite-Field*:
assumes *INF*: $\neg\text{finite}(\text{Field } r)$ **and** $r: \text{Card-order } r$ **and**
LESS1: $|A| <o r$ **and** *LESS2*: $|B| <o r$
shows $|A \text{ Un } B| <o r$
proof –
let $?C = \text{Field } r$
have $1: r =_o |?C| \wedge |?C| =_o r$ **using** r *card-of-Field-ordIso*
ordIso-symmetric **by** *blast*
hence $|A| <o |?C|$ $|B| <o |?C|$
using *LESS1 LESS2 ordLess-ordIso-trans* **by** *blast+*
hence $|A \text{ Un } B| <o |?C|$ **using** *INF*
card-of-Un-ordLess-infinite **by** *blast*
thus *?thesis* **using** *1 ordLess-ordIso-trans* **by** *blast*
qed

30.7 Regular cardinals

definition *cofinal* where

$$\text{cofinal } A \equiv \forall a \in \text{Field } r. \exists b \in A. a \neq b \wedge (a,b) \in r$$

definition *regularCard* where

$$\text{regularCard } r \equiv \forall K. K \leq \text{Field } r \wedge \text{cofinal } K \ r \longrightarrow |K| = o \ r$$

definition *relChain* where

$$\text{relChain } r \ As \equiv \forall i \ j. (i,j) \in r \longrightarrow As \ i \leq As \ j$$

lemma *regularCard-UNION*:

assumes r : *Card-order* r *regularCard* r

and As : *relChain* r As

and $Bsub$: $B \leq (\bigcup i \in \text{Field } r. As \ i)$

and $cardB$: $|B| < o \ r$

shows $\exists i \in \text{Field } r. B \leq As \ i$

proof –

let $?phi = \lambda b \ j. j \in \text{Field } r \wedge b \in As \ j$

have $\forall b \in B. \exists j. ?phi \ b \ j$ **using** $Bsub$ **by** *blast*

then obtain f **where** $f: \bigwedge b. b \in B \implies ?phi \ b \ (f \ b)$

using $bchoice[of \ B \ ?phi]$ **by** *blast*

let $?K = f \ 'B$

{assume $1: \bigwedge i. i \in \text{Field } r \implies \neg B \leq As \ i$

have $2: \text{cofinal } ?K \ r$

unfolding *cofinal-def*

proof (*intro strip*)

fix i **assume** $i: i \in \text{Field } r$

with 1 **obtain** b **where** $b: b \in B \wedge b \notin As \ i$ **by** *blast*

hence $i \neq f \ b \wedge \neg (f \ b, i) \in r$

using $As \ f$ **unfolding** *relChain-def* **by** *auto*

hence $i \neq f \ b \wedge (i, f \ b) \in r$ **using** r

unfolding *card-order-on-def well-order-on-def linear-order-on-def*

total-on-def **using** $i \ f \ b$ **by** *auto*

with b **show** $\exists b \in f \ 'B. i \neq b \wedge (i,b) \in r$ **by** *blast*

qed

moreover have $?K \leq \text{Field } r$ **using** f **by** *blast*

ultimately have $|?K| = o \ r$ **using** $2 \ r$ **unfolding** *regularCard-def* **by** *blast*

moreover

have $|?K| < o \ r$ **using** $cardB \ ordLeq-ordLess-trans \ card-of-image$ **by** *blast*

ultimately have *False* **using** *not-ordLess-ordIso* **by** *blast*

}

thus $?thesis$ **by** *blast*

qed

lemma *infinite-cardSuc-regularCard*:

assumes $r-inf: \neg finite \ (\text{Field } r)$ **and** $r-card: \text{Card-order } r$

shows *regularCard* $(cardSuc \ r)$

proof –

let $?r' = cardSuc \ r$

```

have r': Card-order ?r'  $\wedge$  p. Card-order p  $\longrightarrow$  (p  $\leq_o$  r) = (p <_o ?r')
  using r-card by (auto simp: cardSuc-Card-order cardSuc-ordLeq-ordLess)
show ?thesis
  unfolding regularCard-def proof auto
  fix K assume 1: K  $\leq$  Field ?r' and 2: cofinal K ?r'
  hence |K|  $\leq_o$  |Field ?r'| by (simp only: card-of-mono1)
  also have 22: |Field ?r'| =_o ?r'
    using r' by (simp add: card-of-Field-ordIso[of ?r'])
  finally have |K|  $\leq_o$  ?r' .
  moreover
  { let ?L =  $\bigcup$  j  $\in$  K. underS ?r' j
    let ?J = Field r
    have rJ: r =_o |?J|
      using r-card card-of-Field-ordIso ordIso-symmetric by blast
    assume |K| <_o ?r'
    hence |K|  $\leq_o$  r using r' card-of-Card-order[of K] by blast
    hence |K|  $\leq_o$  |?J| using rJ ordLeq-ordIso-trans by blast
    moreover
    { have  $\forall j \in K. |underS ?r' j| <_o ?r'$ 
      using r' 1 by (auto simp: card-of-underS)
      hence  $\forall j \in K. |underS ?r' j| \leq_o r$ 
        using r' card-of-Card-order by blast
      hence  $\forall j \in K. |underS ?r' j| \leq_o |?J|$ 
        using rJ ordLeq-ordIso-trans by blast
    }
    }
  ultimately have |?L|  $\leq_o$  |?J|
    using r-inf card-of-UNION-ordLeq-infinite by blast
  hence |?L|  $\leq_o$  r using rJ ordIso-symmetric ordLeq-ordIso-trans by blast
  hence |?L| <_o ?r' using r' card-of-Card-order by blast
  moreover
  {
  have Field ?r'  $\leq$  ?L
    using 2 unfolding underS-def cofinal-def by auto
  hence |Field ?r'|  $\leq_o$  |?L| by (simp add: card-of-mono1)
  hence ?r'  $\leq_o$  |?L|
    using 22 ordIso-ordLeq-trans ordIso-symmetric by blast
  }
  ultimately have |?L| <_o |?L| using ordLess-ordLeq-trans by blast
  hence False using ordLess-irreflexive by blast
  }
  ultimately show |K| =_o ?r'
    unfolding ordLeq-iff-ordLess-or-ordIso by blast
qed
qed

```

lemma cardSuc-UNION:

```

assumes r: Card-order r and  $\neg$ finite (Field r)
  and As: relChain (cardSuc r) As
  and Bsub: B  $\leq$  ( $\bigcup$  i  $\in$  Field (cardSuc r). As i)

```

```

  and cardB: |B| ≤o r
  shows ∃ i ∈ Field (cardSuc r). B ≤As i
proof -
  let ?r' = cardSuc r
  have Card-order ?r' ∧ |B| <o ?r'
    using r cardB cardSuc-ordLeq-ordLess cardSuc-Card-order
    card-of-Card-order by blast
  moreover have regularCard ?r'
    using assms by (simp add: infinite-cardSuc-regularCard)
  ultimately show ?thesis
    using As Bsub cardB regularCard-UNION by blast
qed

```

30.8 Others

```

lemma card-of-Func-Times:
  |Func (A × B) C| =o |Func A (Func B C)|
  unfolding card-of-ordIso[symmetric]
  using bij-betw-curr by blast

```

```

lemma card-of-Pow-Func:
  |Pow A| =o |Func A (UNIV::bool set)|
proof -
  define F where [abs-def]: F A' a ≡
    (if a ∈ A then (if a ∈ A' then True else False) else undefined) for A' a
  have bij-betw F (Pow A) (Func A (UNIV::bool set))
    unfolding bij-betw-def inj-on-def proof (intro ballI impI conjI)
    fix A1 A2 assume A1 ∈ Pow A A2 ∈ Pow A F A1 = F A2
    thus A1 = A2 unfolding F-def Pow-def fun-eq-iff by (auto split: if-split-asm)
  next
    show F ' Pow A = Func A UNIV
    proof safe
      fix f assume f: f ∈ Func A (UNIV::bool set)
      show f ∈ F ' Pow A
        unfolding image-iff
      proof
        show f = F {a ∈ A. f a = True}
          using f unfolding Func-def F-def by force
        qed auto
      qed (unfold Func-def F-def, auto)
    qed
  thus ?thesis unfolding card-of-ordIso[symmetric] by blast
qed

```

```

lemma card-of-Func-UNIV:
  |Func (UNIV::'a set) (B::'b set)| =o |{f::'a ⇒ 'b. range f ⊆ B}|
proof -
  let ?F = λ f (a::'a). ((f a)::'b)
  have bij-betw ?F {f. range f ⊆ B} (Func UNIV B)

```


unfolding *bij-betw-def inj-on-def*
proof safe
fix $h :: 'a \Rightarrow 'b$ **assume** $h: h \in \text{Func UNIV } B$
then obtain f **where** $f: \forall a. h a = f a$ **by** *blast*
hence $\text{range } f \subseteq B$ **using** h **unfolding** *Func-def* **by** *auto*
thus $h \in (\lambda f a. f a) \cdot \{f. \text{range } f \subseteq B\}$ **using** f **by** *auto*
qed(*unfold Func-def fun-eq-iff, auto*)
then show *?thesis*
by (*blast intro: ordIso-symmetric card-of-ordIsoI*)
qed

lemma *Func-Times-Range*:
 $|\text{Func } A (B \times C)| = o |\text{Func } A B \times \text{Func } A C|$ (**is** $|\text{?LHS}| = o |\text{?RHS}|$)
proof –
let $?F = \lambda fg. (\lambda x. \text{if } x \in A \text{ then } \text{fst } (fg \ x) \text{ else } \text{undefined},$
 $\lambda x. \text{if } x \in A \text{ then } \text{snd } (fg \ x) \text{ else } \text{undefined})$
let $?G = \lambda (f, g) x. \text{if } x \in A \text{ then } (f \ x, g \ x) \text{ else } \text{undefined}$
have *bij-betw ?F ?LHS ?RHS* **unfolding** *bij-betw-def inj-on-def*
proof (*intro conjI impI ballI equalityI subsetI*)
fix $f \ g$ **assume** $*$: $f \in \text{Func } A (B \times C) \ g \in \text{Func } A (B \times C) \ ?F \ f = ?F \ g$
show $f = g$
proof
fix x **from** $*$ **have** $\text{fst } (f \ x) = \text{fst } (g \ x) \wedge \text{snd } (f \ x) = \text{snd } (g \ x)$
by (*cases x \in A*) (*auto simp: Func-def fun-eq-iff split: if-splits*)
then show $f \ x = g \ x$ **by** (*subst (1 2) surjective-pairing*) *simp*
qed
next
fix fg **assume** $fg \in \text{Func } A B \times \text{Func } A C$
thus $fg \in ?F \cdot \text{Func } A (B \times C)$
by (*intro image-eqI[of - - ?G fg]*) (*auto simp: Func-def*)
qed (*auto simp: Func-def fun-eq-iff*)
thus *?thesis* **using** *card-of-ordIso* **by** *blast*
qed

30.9 Regular vs. stable cardinals

definition *stable* $:: 'a \text{ rel} \Rightarrow \text{bool}$
where
 $\text{stable } r \equiv \forall (A::'a \text{ set}) (F :: 'a \Rightarrow 'a \text{ set}).$
 $|\text{A}| < o \ r \wedge (\forall a \in A. |F \ a| < o \ r)$
 $\longrightarrow |\text{SIGMA } a : A. F \ a| < o \ r$

lemma *regularCard-stable*:
assumes cr : *Card-order* r **and** ir : $\neg \text{finite } (Field \ r)$ **and** reg : *regularCard* r
shows *stable* r
unfolding *stable-def* **proof safe**
fix $A :: 'a \text{ set}$ **and** $F :: 'a \Rightarrow 'a \text{ set}$ **assume** $A: |A| < o \ r$ **and** $F: \forall a \in A. |F \ a| < o \ r$
{assume $r \leq o |\text{Sigma } A \ F|$

hence $|Field\ r| \leq_o |Sigma\ A\ F|$ **using** *card-of-Field-ordIso*[OF cr] *ordIso-ordLeq-trans*
by *blast*
moreover have $Fi: Field\ r \neq \{\}$ **using** *ir* **by** *auto*
ultimately have $\exists f. f \text{ ' } Sigma\ A\ F = Field\ r$ **using** *card-of-ordLeq2*[OF Fi]
by *blast*
then obtain f **where** $f: f \text{ ' } Sigma\ A\ F = Field\ r$ **by** *blast*
have $r: wo-rel\ r$ **using** *cr* **unfolding** *card-order-on-def wo-rel-def* **by** *auto*
{fix a **assume** $a: a \in A$
define L **where** $L = \{(a,u) \mid u. u \in F\ a\}$
have $fL: f \text{ ' } L \subseteq Field\ r$ **using** $f\ a$ **unfolding** *L-def* **by** *auto*
have *bij-betw snd* $\{(a, u) \mid u. u \in F\ a\}$ $(F\ a)$
unfolding *bij-betw-def inj-on-def* **by** $(auto\ simp: image-def)$
then have $|L| =_o |F\ a|$ **unfolding** *L-def card-of-ordIso*[*symmetric*] **by** *blast*
hence $|L| <_o r$ **using** $F\ a$ *ordIso-ordLess-trans*[of $|L|$ $|F\ a|$] **unfolding** *L-def*
by *auto*
hence $|f \text{ ' } L| <_o r$ **using** *ordLeq-ordLess-trans*[OF *card-of-image, of L*] **un-**
folding *L-def* **by** *auto*
hence $\neg cofinal\ (f \text{ ' } L)\ r$ **using** *reg fL* **unfolding** *regularCard-def*
by $(force\ simp\ add: dest: not-ordLess-ordIso)$
then obtain k **where** $k: k \in Field\ r$ **and** $\forall l \in L. \neg (f\ l \neq k \wedge (k, f\ l) \in r)$
unfolding *cofinal-def image-def* **by** *auto*
hence $\exists k \in Field\ r. \forall l \in L. (f\ l, k) \in r$
using *wo-rel.in-notinI*[OF $r - \langle k \in Field\ r \rangle$] fL **unfolding** *image-subset-iff*
by *fast*
hence $\exists k \in Field\ r. \forall u \in F\ a. (f\ (a,u), k) \in r$ **unfolding** *L-def* **by** *auto*
}
then have $x: \bigwedge a. a \in A \implies \exists k. k \in Field\ r \wedge (\forall u \in F\ a. (f\ (a, u), k) \in r)$ **by**
blast
obtain gg **where** $\bigwedge a. a \in A \implies gg\ a = (SOME\ k. k \in Field\ r \wedge (\forall u \in F\ a. (f\ (a, u), k) \in r))$ **by** *simp*
then have $gg: \forall a \in A. \forall u \in F\ a. (f\ (a, u), gg\ a) \in r$ **using** *someI-ex*[OF x] **by**
auto
obtain $j0$ **where** $j0: j0 \in Field\ r$ **using** Fi **by** *auto*
define g **where** [*abs-def*]: $g\ a = (if\ F\ a \neq \{\} then\ gg\ a\ else\ j0)$ **for** a
have $g: \forall a \in A. \forall u \in F\ a. (f\ (a,u), g\ a) \in r$ **using** gg **unfolding** *g-def* **by**
auto
hence $1: Field\ r \subseteq (\bigcup a \in A. under\ r\ (g\ a))$
using f [*symmetric*] **unfolding** *under-def image-def* **by** *auto*
have $gA: g \text{ ' } A \subseteq Field\ r$ **using** $gg\ j0$ **unfolding** *Field-def g-def* **by** *auto*
moreover have *cofinal* $(g \text{ ' } A)\ r$ **unfolding** *cofinal-def*
proof *safe*
fix i **assume** $i \in Field\ r$
then obtain j **where** $ij: (i,j) \in r\ i \neq j$ **using** *cr ir infinite-Card-order-limit*
by *fast*
hence $j \in Field\ r$ **using** *card-order-on-def cr well-order-on-domain* **by** *fast*
then obtain a **where** $a: a \in A$ **and** $j: (j, g\ a) \in r$
using 1 **unfolding** *under-def* **by** *auto*
hence $(i, g\ a) \in r$ **using** $ij\ wo-rel.TRANS$ [OF r] **unfolding** *trans-def* **by**
blast

moreover have $i \neq g a$
using $ij j r$ **unfolding** $wo\text{-}rel\text{-}def$ **unfolding** $well\text{-}order\text{-}on\text{-}def$ $linear\text{-}order\text{-}on\text{-}def$
 $partial\text{-}order\text{-}on\text{-}def$ $antisym\text{-}def$ **by** $auto$
ultimately show $\exists j \in g \text{ ' } A. i \neq j \wedge (i, j) \in r$ **using** a **by** $auto$
qed
ultimately have $|g \text{ ' } A| =_o r$ **using** reg **unfolding** $regularCard\text{-}def$ **by** $auto$
moreover have $|g \text{ ' } A| \leq_o |A|$ **using** $card\text{-}of\text{-}image$ **by** $blast$
ultimately have $False$ **using** A **using** $not\text{-}ordLess\text{-}ordIso$ $ordLeq\text{-}ordLess\text{-}trans$
by $blast$
}
thus $|\Sigma A F| <_o r$
using cr $not\text{-}ordLess\text{-}iff\text{-}ordLeq$ **using** $card\text{-}of\text{-}Well\text{-}order$ $card\text{-}order\text{-}on\text{-}well\text{-}order\text{-}on$
by $blast$
qed

lemma $stable\text{-}regularCard$:

assumes cr : $Card\text{-}order r$ **and** ir : $\neg finite (Field r)$ **and** st : $stable r$
shows $regularCard r$
unfolding $regularCard\text{-}def$ **proof** $safe$
fix K **assume** K : $K \subseteq Field r$ **and** cof : $cofinal K r$
have $|K| \leq_o r$ **using** K $card\text{-}of\text{-}Field\text{-}ordIso$ $card\text{-}of\text{-}mono1$ cr $ordLeq\text{-}ordIso\text{-}trans$
by $blast$
moreover
{ **assume** Kr : $|K| <_o r$
have x : $\bigwedge a. a \in Field r \implies \exists b. b \in K \wedge a \neq b \wedge (a, b) \in r$ **using** cof
unfolding $cofinal\text{-}def$ **by** $blast$
then obtain f **where** $\bigwedge a. a \in Field r \implies f a = (SOME b. b \in K \wedge a \neq b \wedge (a, b) \in r)$ **by** $simp$
then have $\forall a \in Field r. f a \in K \wedge a \neq f a \wedge (a, f a) \in r$ **using** $someI\text{-}ex[OF x]$ **by** $simp$
hence $Field r \subseteq (\bigcup a \in K. underS r a)$ **unfolding** $underS\text{-}def$ **by** $auto$
hence $r \leq_o |\bigcup a \in K. underS r a|$
using cr $Card\text{-}order\text{-}iff\text{-}ordLeq\text{-}card\text{-}of$ $card\text{-}of\text{-}mono1$ $ordLeq\text{-}transitive$ **by**
 $blast$
also have $|\bigcup a \in K. underS r a| \leq_o |\Sigma a: K. underS r a|$ **by** (rule
 $card\text{-}of\text{-}UNION\text{-}Sigma$)
also
{ **have** $\forall a \in K. |underS r a| <_o r$ **using** K $card\text{-}of\text{-}underS[OF cr]$ $subsetD$ **by**
 $auto$
hence $|\Sigma a: K. underS r a| <_o r$ **using** st Kr **unfolding** $stable\text{-}def$ **by**
 $auto$
}
finally have $r <_o r$.
hence $False$ **using** $ordLess\text{-}irreflexive$ **by** $blast$
}
ultimately show $|K| =_o r$ **using** $ordLeq\text{-}iff\text{-}ordLess\text{-}or\text{-}ordIso$ **by** $blast$
qed

lemma $internalize\text{-}card\text{-}of\text{-}ordLess$:

$$(|A| <_o r) = (\exists B < \text{Field } r. |A| =_o |B| \wedge |B| <_o r)$$
proof**assume** $|A| <_o r$ **then obtain** p **where** $1: \text{Field } p < \text{Field } r \wedge |A| =_o p \wedge p <_o r$ **using** *internalize-ordLess*[of $|A|$ r] **by** *blast***hence** *Card-order* p **using** *card-of-Card-order* *Card-order-ordIso2* **by** *blast***hence** $|\text{Field } p| =_o p$ **using** *card-of-Field-ordIso* **by** *blast***hence** $|A| =_o |\text{Field } p| \wedge |\text{Field } p| <_o r$ **using** *1 ordIso-equivalence* *ordIso-ordLess-trans* **by** *blast***thus** $\exists B < \text{Field } r. |A| =_o |B| \wedge |B| <_o r$ **using** *1* **by** *blast***next****assume** $\exists B < \text{Field } r. |A| =_o |B| \wedge |B| <_o r$ **thus** $|A| <_o r$ **using** *ordIso-ordLess-trans* **by** *blast***qed****lemma** *card-of-Sigma-cong1*:**assumes** $\forall i \in I. |A \ i| =_o |B \ i|$ **shows** $|\text{SIGMA } i : I. A \ i| =_o |\text{SIGMA } i : I. B \ i|$ **using** *assms* **by** (*auto simp add: card-of-Sigma-mono1 ordIso-iff-ordLeq*)**lemma** *card-of-Sigma-cong2*:**assumes** *bij-betw* f ($I::'i \text{ set}$) ($J::'j \text{ set}$)**shows** $|\text{SIGMA } i : I. (A::'j \Rightarrow 'a \text{ set}) (f \ i)| =_o |\text{SIGMA } j : J. A \ j|$ **proof** –**let** $?LEFT = \text{SIGMA } i : I. A \ (f \ i)$ **let** $?RIGHT = \text{SIGMA } j : J. A \ j$ **define** u **where** $u \equiv \lambda(i::'i, a::'a). (f \ i, a)$ **have** *bij-betw* u $?LEFT$ $?RIGHT$ **using** *assms* **unfolding** u -*def* *bij-betw-def* *inj-on-def* **by** *auto***thus** $?thesis$ **using** *card-of-ordIso* **by** *blast***qed****lemma** *card-of-Sigma-cong*:**assumes** *BIJ*: *bij-betw* f I J **and** *ISO*: $\forall j \in J. |A \ j| =_o |B \ j|$ **shows** $|\text{SIGMA } i : I. A \ (f \ i)| =_o |\text{SIGMA } j : J. B \ j|$ **proof** –**have** $\forall i \in I. |A \ (f \ i)| =_o |B \ (f \ i)|$ **using** *ISO BIJ* **unfolding** *bij-betw-def* **by** *blast***hence** $|\text{SIGMA } i : I. A \ (f \ i)| =_o |\text{SIGMA } i : I. B \ (f \ i)|$ **by** (*rule card-of-Sigma-cong1*)**moreover** **have** $|\text{SIGMA } i : I. B \ (f \ i)| =_o |\text{SIGMA } j : J. B \ j|$ **using** *BIJ card-of-Sigma-cong2* **by** *blast***ultimately show** $?thesis$ **using** *ordIso-transitive* **by** *blast***qed****lemma** *stable-elim*:**assumes** *ST*: *stable* r **and** *A-LESS*: $|A| <_o r$ **and***F-LESS*: $\bigwedge a. a \in A \Longrightarrow |F \ a| <_o r$ **shows** $|\text{SIGMA } a : A. F \ a| <_o r$

proof –

obtain A' **where** $1: A' < \text{Field } r \wedge |A'| < o r$ **and** $2: |A| = o |A'|$
using *internalize-card-of-ordLess*[of A r] *A-LESS* **by** *blast*
then obtain G **where** $3: \text{bij-betw } G \ A' \ A$
using *card-of-ordIso* *ordIso-symmetric* **by** *blast*

{**fix** a **assume** $a \in A$
hence $\exists B'. B' \leq \text{Field } r \wedge |F a| = o |B'| \wedge |B'| < o r$
using *internalize-card-of-ordLess*[of $F a$ r] *F-LESS* **by** *blast*

}

then obtain F' **where**

temp: $\forall a \in A. F' a \leq \text{Field } r \wedge |F a| = o |F' a| \wedge |F' a| < o r$

using *bchoice*[of $A \ \lambda a \ B'. B' \leq \text{Field } r \wedge |F a| = o |B'| \wedge |B'| < o r$] **by** *blast*

hence $4: \forall a \in A. F' a \leq \text{Field } r \wedge |F' a| < o r$ **by** *auto*

have $5: \forall a \in A. |F' a| = o |F a|$ **using** *temp* *ordIso-symmetric* **by** *auto*

have $\forall a' \in A'. F'(G a') \leq \text{Field } r \wedge |F'(G a')| < o r$

using $3 \ 4$ *bij-betw-ball*[of $G \ A' \ A$] **by** *auto*

hence $|\text{SIGMA } a' : A'. F'(G a')| < o r$

using *ST 1 unfolding* *stable-def* **by** *auto*

moreover have $|\text{SIGMA } a' : A'. F'(G a')| = o |\text{SIGMA } a : A. F a|$

using *card-of-Sigma-cong*[of $G \ A' \ A \ F' \ F$] $5 \ 3$ **by** *blast*

ultimately show *?thesis* **using** *ordIso-symmetric* *ordIso-ordLess-trans* **by** *blast*

qed

lemma *stable-natLeq*: *stable natLeq*

proof(*unfold stable-def*, *safe*)

fix $A :: 'a \ \text{set}$ **and** $F :: 'a \Rightarrow 'a \ \text{set}$

assume $|A| < o \ \text{natLeq}$ **and** $\forall a \in A. |F a| < o \ \text{natLeq}$

hence $\text{finite } A \wedge (\forall a \in A. \text{finite}(F a))$

by (*auto simp add: finite-iff-ordLess-natLeq*)

hence $\text{finite}(\text{Sigma } A \ F)$ **by** (*simp only: finite-SigmaI*[of $A \ F$])

thus $|\text{Sigma } A \ F| < o \ \text{natLeq}$

by (*auto simp add: finite-iff-ordLess-natLeq*)

qed

corollary *regularCard-natLeq*: *regularCard natLeq*

using *stable-regularCard*[*OF natLeq-Card-order - stable-natLeq*] *Field-natLeq* **by** *simp*

lemma *stable-ordIso1*:

assumes *ST*: *stable r* **and** *ISO*: $r' = o r$

shows *stable r'*

proof(*unfold stable-def*, *auto*)

fix $A :: 'b \ \text{set}$ **and** $F :: 'b \Rightarrow 'b \ \text{set}$

assume $|A| < o \ r'$ **and** $\forall a \in A. |F a| < o \ r'$

hence $(|A| < o \ r) \wedge (\forall a \in A. |F a| < o \ r)$

using *ISO ordLess-ordIso-trans* **by** *blast*

hence $|\text{SIGMA } a : A. F a| < o \ r$ **using** *assms stable-elim* **by** *blast*

thus $|SIGMA a : A. F a| <_o r'$
 using *ISO ordIso-symmetric ordLess-ordIso-trans* by *blast*
 qed

lemma *stable-UNION*:

assumes *stable r* and $|A| <_o r$ and $\bigwedge a. a \in A \implies |F a| <_o r$
 shows $|\bigcup a \in A. F a| <_o r$
 using *assms card-of-UNION-Sigma stable-elim ordLeq-ordLess-trans* by *blast*

corollary *card-of-UNION-ordLess-infinite*:

assumes *stable |B|* and $|I| <_o |B|$ and $\forall i \in I. |A i| <_o |B|$
 shows $|\bigcup i \in I. A i| <_o |B|$
 using *assms stable-UNION* by *blast*

corollary *card-of-UNION-ordLess-infinite-Field*:

assumes *ST: stable r* and *r: Card-order r* and
LEQ-I: |I| <_o r and *LEQ: $\forall i \in I. |A i| <_o r$*
 shows $|\bigcup i \in I. A i| <_o r$

proof –

let $?B = Field r$
 have $1: r =_o |?B| \wedge |?B| =_o r$ using *r card-of-Field-ordIso*
ordIso-symmetric by *blast*
 hence $|I| <_o |?B| \ \forall i \in I. |A i| <_o |?B|$
 using *LEQ-I LEQ ordLess-ordIso-trans* by *blast+*
 moreover have *stable |?B|* using *stable-ordIso1 ST 1* by *blast*
 ultimately have $|\bigcup i \in I. A i| <_o |?B|$ using *LEQ*
card-of-UNION-ordLess-infinite by *blast*
 thus *?thesis* using *1 ordLess-ordIso-trans* by *blast*

qed

end

31 Cardinal Arithmetic as Needed by Bounded Natural Functors

theory *BNF-Cardinal-Arithmetic*

imports *BNF-Cardinal-Order-Relation*

begin

lemma *dir-image*: $\llbracket \bigwedge x y. (f x = f y) = (x = y); Card-order r \rrbracket \implies r =_o dir-image$
 $r f$

by (*rule dir-image-ordIso*) (*auto simp add: inj-on-def card-order-on-def*)

lemma *card-order-dir-image*:

assumes *bij: bij f* and *co: card-order r*

shows *card-order (dir-image r f)*

proof –

have *Field (dir-image r f) = UNIV*

using *assms card-order-on-Card-order*[of UNIV *r*]
unfolding *bij-def dir-image-Field* **by** *auto*
moreover from *bij* **have** $\bigwedge x y. (f x = f y) = (x = y)$
unfolding *bij-def inj-on-def* **by** *auto*
with *co* **have** *Card-order* (*dir-image r f*)
using *card-order-on-Card-order Card-order-ordIso2*[OF - *dir-image*] **by** *blast*
ultimately show *?thesis* **by** *auto*
qed

lemma *ordIso-refl*: *Card-order r* $\implies r =_o r$
by (*rule card-order-on-ordIso*)

lemma *ordLeq-refl*: *Card-order r* $\implies r \leq_o r$
by (*rule ordIso-imp-ordLeq, rule card-order-on-ordIso*)

lemma *card-of-ordIso-subst*: $A = B \implies |A| =_o |B|$
by (*simp only: ordIso-refl card-of-Card-order*)

lemma *Field-card-order*: *card-order r* \implies *Field r = UNIV*
using *card-order-on-Card-order*[of UNIV *r*] **by** *simp*

31.1 Zero

definition *czero where*
czero = card-of {}

lemma *czero-ordIso*: *czero =_o czero*
using *card-of-empty-ordIso* **by** (*simp add: czero-def*)

lemma *card-of-ordIso-czero-iff-empty*:
 $|A| =_o (czero :: 'b \text{ rel}) \iff A = (\{\} :: 'a \text{ set})$
unfolding *czero-def* **by** (*rule iffI*[OF *card-of-empty2*]) (*auto simp: card-of-refl card-of-empty-ordIso*)

abbreviation *Cnotzero where*
*Cnotzero (r :: 'a rel) $\equiv \neg(r =_o (czero :: 'a rel)) \wedge$ Card-order *r**

lemma *Cnotzero-imp-not-empty*: *Cnotzero r* \implies *Field r \neq {}*
unfolding *Card-order-iff-ordIso-card-of czero-def* **by** *force*

lemma *czeroI*:
 $\llbracket \text{Card-order } r; \text{Field } r = \{\} \rrbracket \implies r =_o czero$
using *Cnotzero-imp-not-empty ordIso-transitive*[OF - *czero-ordIso*] **by** *blast*

lemma *czeroE*:
 $r =_o czero \implies \text{Field } r = \{\}$
unfolding *czero-def*

by (drule card-of-cong) (simp only: Field-card-of card-of-empty2)

lemma Cnotzero-mono:

[[Cnotzero r; Card-order q; r ≤_o q]] ⇒ Cnotzero q
by (force intro: czeroI dest: card-of-mono2 card-of-empty3 czeroE)

31.2 (In)finite cardinals

definition cfinite where

cfinite r ≡ (¬ finite (Field r))

abbreviation Cfinite where

Cfinite r ≡ cfinite r ∧ Card-order r

definition finite where

finite r = finite (Field r)

abbreviation Cfinite where

Cfinite r ≡ cfinite r ∧ Card-order r

lemma Cfinite-ordLess-Cfinite: [[Cfinite r; Cfinite s]] ⇒ r <_o s

unfolding cfinite-def cfinite-def

by (blast intro: finite-ordLess-infinite card-order-on-well-order-on)

lemmas natLeq-card-order = natLeq-Card-order[unfolded Field-natLeq]

lemma natLeq-cfinite: cfinite natLeq

unfolding cfinite-def Field-natLeq by (rule infinite-UNIV-nat)

lemma natLeq-Cfinite: Cfinite natLeq

using natLeq-cfinite natLeq-Card-order by simp

lemma natLeq-ordLeq-cfinite:

assumes inf: Cfinite r

shows natLeq ≤_o r

proof –

from inf **have** natLeq ≤_o |Field r| **unfolding** cfinite-def

using infinite-iff-natLeq-ordLeq **by** blast

also from inf **have** |Field r| =_o r **by** (simp add: card-of-unique ordIso-symmetric)

finally show ?thesis .

qed

lemma cfinite-not-czero: cfinite r ⇒ ¬ (r =_o (czero :: 'a rel))

unfolding cfinite-def **by** (cases Field r = {}) (auto dest: czeroE)

lemma Cfinite-Cnotzero: Cfinite r ⇒ Cnotzero r

using cfinite-not-czero **by** auto

lemma Cfinite-cong: [[r1 =_o r2; Cfinite r1]] ⇒ Cfinite r2

using *Card-order-ordIso2*[of *r1 r2*] **unfolding** *cinfinite-def ordIso-iff-ordLeq*
by (*auto dest: card-of-ordLeq-infinite[OF card-of-mono2]*)

lemma *cinfinite-mono*: $\llbracket r1 \leq_o r2; \text{cinfinite } r1 \rrbracket \implies \text{cinfinite } r2$
unfolding *cinfinite-def* **by** (*auto dest: card-of-ordLeq-infinite[OF card-of-mono2]*)

lemma *regularCard-ordIso*:

assumes $k =_o k'$ **and** *Cinfinite k* **and** *regularCard k*
shows *regularCard k'*

proof –

have *stable k* **using** *assms infinite-def regularCard-stable* **by** *blast*

hence *stable k'* **using** *assms stable-ordIso1 ordIso-symmetric* **by** *blast*

thus *?thesis* **using** *assms infinite-def stable-regularCard Cinfinite-cong* **by** *blast*

qed

corollary *card-of-UNION-ordLess-infinite-Field-regularCard*:

assumes *regularCard r* **and** *Cinfinite r* **and** $|I| <_o r$ **and** $\forall i \in I. |A\ i| <_o r$

shows $|\bigcup_{i \in I}. A\ i| <_o r$

using *card-of-UNION-ordLess-infinite-Field regularCard-stable assms infinite-def*

by *blast*

31.3 Binary sum

definition *csum* (*infixr +c 65*)

where $r1 +_c r2 \equiv |Field\ r1\ <+>\ Field\ r2|$

lemma *Field-csum*: $Field\ (r +_c s) = Inl\ 'Field\ r \cup Inr\ 'Field\ s$

unfolding *csum-def Field-card-of* **by** *auto*

lemma *Card-order-csum*: $Card-order\ (r1 +_c r2)$

unfolding *csum-def* **by** (*simp add: card-of-Card-order*)

lemma *csum-Cnotzero1*: $Cnotzero\ r1 \implies Cnotzero\ (r1 +_c r2)$

using *Cnotzero-imp-not-empty*

by (*auto intro: card-of-Card-order simp: csum-def card-of-ordIso-czero-iff-empty*)

lemma *card-order-csum*:

assumes *card-order r1 card-order r2*

shows *card-order (r1 +c r2)*

proof –

have $Field\ r1 = UNIV\ Field\ r2 = UNIV$ **using** *assms card-order-on-Card-order*

by *auto*

thus *?thesis* **unfolding** *csum-def* **by** (*auto simp: card-of-card-order-on*)

qed

lemma *cinfinite-csum*:

$\text{cinfinite } r1 \vee \text{cinfinite } r2 \implies \text{cinfinite } (r1 +_c r2)$

unfolding *cinfinite-def csum-def* **by** (*auto simp: Field-card-of*)

lemma *Cinfinite-csum*:

$Cinfinite\ r1 \vee Cinfinite\ r2 \implies Cinfinite\ (r1 + c\ r2)$

using *card-of-Card-order*

by (*auto simp: cinfinite-def csum-def Field-card-of*)

lemma *Cinfinite-csum1*: $Cinfinite\ r1 \implies Cinfinite\ (r1 + c\ r2)$

by (*blast intro!: Cinfinite-csum elim:*)

lemma *Cinfinite-csum-weak*:

$\llbracket Cinfinite\ r1; Cinfinite\ r2 \rrbracket \implies Cinfinite\ (r1 + c\ r2)$

by (*erule Cinfinite-csum1*)

lemma *csum-cong*: $\llbracket p1 =o\ r1; p2 =o\ r2 \rrbracket \implies p1 + c\ p2 =o\ r1 + c\ r2$

by (*simp only: csum-def ordIso-Plus-cong*)

lemma *csum-cong1*: $p1 =o\ r1 \implies p1 + c\ q =o\ r1 + c\ q$

by (*simp only: csum-def ordIso-Plus-cong1*)

lemma *csum-cong2*: $p2 =o\ r2 \implies q + c\ p2 =o\ q + c\ r2$

by (*simp only: csum-def ordIso-Plus-cong2*)

lemma *csum-mono*: $\llbracket p1 \leqo\ r1; p2 \leqo\ r2 \rrbracket \implies p1 + c\ p2 \leqo\ r1 + c\ r2$

by (*simp only: csum-def ordLeq-Plus-mono*)

lemma *csum-mono1*: $p1 \leqo\ r1 \implies p1 + c\ q \leqo\ r1 + c\ q$

by (*simp only: csum-def ordLeq-Plus-mono1*)

lemma *csum-mono2*: $p2 \leqo\ r2 \implies q + c\ p2 \leqo\ q + c\ r2$

by (*simp only: csum-def ordLeq-Plus-mono2*)

lemma *ordLeq-csum1*: $Card\text{-}order\ p1 \implies p1 \leqo\ p1 + c\ p2$

by (*simp only: csum-def Card-order-Plus1*)

lemma *ordLeq-csum2*: $Card\text{-}order\ p2 \implies p2 \leqo\ p1 + c\ p2$

by (*simp only: csum-def Card-order-Plus2*)

lemma *csum-com*: $p1 + c\ p2 =o\ p2 + c\ p1$

by (*simp only: csum-def card-of-Plus-commute*)

lemma *csum-assoc*: $(p1 + c\ p2) + c\ p3 =o\ p1 + c\ p2 + c\ p3$

by (*simp only: csum-def Field-card-of card-of-Plus-assoc*)

lemma *Cfinite-csum*: $\llbracket Cfinite\ r; Cfinite\ s \rrbracket \implies Cfinite\ (r + c\ s)$

unfolding *cfinite-def csum-def Field-card-of* **using** *card-of-card-order-on* **by** *simp*

lemma *csum-csum*: $(r1 + c\ r2) + c\ (r3 + c\ r4) =o\ (r1 + c\ r3) + c\ (r2 + c\ r4)$

proof –

have $(r1 + c\ r2) + c\ (r3 + c\ r4) =o\ r1 + c\ r2 + c\ (r3 + c\ r4)$

by (rule csum-assoc)
 also have $r1 + c r2 + c (r3 + c r4) = o r1 + c (r2 + c r3) + c r4$
 by (intro csum-assoc csum-cong2 ordIso-symmetric)
 also have $r1 + c (r2 + c r3) + c r4 = o r1 + c (r3 + c r2) + c r4$
 by (intro csum-com csum-cong1 csum-cong2)
 also have $r1 + c (r3 + c r2) + c r4 = o r1 + c r3 + c r2 + c r4$
 by (intro csum-assoc csum-cong2 ordIso-symmetric)
 also have $r1 + c r3 + c r2 + c r4 = o (r1 + c r3) + c (r2 + c r4)$
 by (intro csum-assoc ordIso-symmetric)
 finally show ?thesis .
 qed

lemma *Plus-csum*: $|A <+> B| = o |A| + c |B|$
 by (simp only: csum-def Field-card-of card-of-refl)

lemma *Un-csum*: $|A \cup B| \leq o |A| + c |B|$
 using ordLeq-ordIso-trans[OF card-of-Un-Plus-ordLeq Plus-csum] by blast

31.4 One

definition *cone* where
 $cone = card-of \{()\}$

lemma *Card-order-cone*: *Card-order cone*
 unfolding cone-def by (rule card-of-Card-order)

lemma *Cfinite-cone*: *Cfinite cone*
 unfolding cfinite-def by (simp add: Card-order-cone)

lemma *cone-not-czero*: $\neg (cone = o czero)$
 unfolding czero-def cone-def ordIso-iff-ordLeq
 using card-of-empty3 empty-not-insert by blast

lemma *cone-ordLeq-Cnotzero*: $Cnotzero r \implies cone \leq o r$
 unfolding cone-def by (rule Card-order-singl-ordLeq) (auto intro: czeroI)

31.5 Two

definition *ctwo* where
 $ctwo = |UNIV :: bool set|$

lemma *Card-order-ctwo*: *Card-order ctwo*
 unfolding ctwo-def by (rule card-of-Card-order)

lemma *ctwo-not-czero*: $\neg (ctwo = o czero)$
 using card-of-empty3[of UNIV :: bool set] ordIso-iff-ordLeq
 unfolding czero-def ctwo-def using UNIV-not-empty by auto

lemma *ctwo-Cnotzero*: *Cnotzero ctwo*
 by (simp add: ctwo-not-czero Card-order-ctwo)

31.6 Family sum

definition *Csum* where

$$Csum\ r\ rs \equiv |\text{SIGMA } i : \text{Field } r. \text{Field } (rs\ i)|$$

syntax *-Csum* ::

$$pttrn \Rightarrow ('a * 'a)\ set \Rightarrow 'b * 'b\ set \Rightarrow (('a * 'b) * ('a * 'b))\ set \\ ((\exists CSUM\ :-.\ -)\ [0, 51, 10]\ 10)$$

translations

$$CSUM\ i:r.\ rs == CONST\ Csum\ r\ (\%i.\ rs)$$

lemma *SIGMA-CSUM*: $|\text{SIGMA } i : I. As\ i| = (CSUM\ i : |I|. |As\ i|)$

by (*auto simp: Csum-def Field-card-of*)

31.7 Product

definition *cprod* (*infixr *c 80*) where

$$r1 *c r2 = |\text{Field } r1 \times \text{Field } r2|$$

lemma *card-order-cprod*:

assumes *card-order* *r1* *card-order* *r2*

shows *card-order* (*r1 *c r2*)

proof –

have *Field* *r1* = *UNIV* *Field* *r2* = *UNIV*

using *assms card-order-on-Card-order* by *auto*

thus *?thesis* by (*auto simp: cprod-def card-of-card-order-on*)

qed

lemma *Card-order-cprod*: *Card-order* (*r1 *c r2*)

by (*simp only: cprod-def Field-card-of card-of-card-order-on*)

lemma *cprod-mono1*: $p1 \leq_o r1 \implies p1 *c q \leq_o r1 *c q$

by (*simp only: cprod-def ordLeq-Times-mono1*)

lemma *cprod-mono2*: $p2 \leq_o r2 \implies q *c p2 \leq_o q *c r2$

by (*simp only: cprod-def ordLeq-Times-mono2*)

lemma *cprod-mono*: $\llbracket p1 \leq_o r1; p2 \leq_o r2 \rrbracket \implies p1 *c p2 \leq_o r1 *c r2$

by (*rule ordLeq-transitive[OF cprod-mono1 cprod-mono2]*)

lemma *ordLeq-cprod2*: $\llbracket Cnotzero\ p1; \text{Card-order } p2 \rrbracket \implies p2 \leq_o p1 *c p2$

unfolding *cprod-def* by (*rule Card-order-Times2*) (*auto intro: czeroI*)

lemma *cinfinite-cprod*: $\llbracket \text{cinfinite } r1; \text{cinfinite } r2 \rrbracket \implies \text{cinfinite } (r1 *c r2)$

by (*simp add: cinfinite-def cprod-def Field-card-of infinite-cartesian-product*)

lemma *cinfinite-cprod2*: $\llbracket Cnotzero\ r1; \text{Cinfinite } r2 \rrbracket \implies \text{cinfinite } (r1 *c r2)$

by (*rule cinfinite-mono*) (*auto intro: ordLeq-cprod2*)

lemma *Cinfinite-cprod2*: $\llbracket \text{Cnotzero } r1; \text{Cinfinite } r2 \rrbracket \implies \text{Cinfinite } (r1 *c r2)$
by (*blast intro: cinfinite-cprod2 Card-order-cprod*)

lemma *cprod-cong*: $\llbracket p1 =o r1; p2 =o r2 \rrbracket \implies p1 *c p2 =o r1 *c r2$
unfolding *ordIso-iff-ordLeq* **by** (*blast intro: cprod-mono*)

lemma *cprod-cong1*: $\llbracket p1 =o r1 \rrbracket \implies p1 *c p2 =o r1 *c p2$
unfolding *ordIso-iff-ordLeq* **by** (*blast intro: cprod-mono1*)

lemma *cprod-cong2*: $p2 =o r2 \implies q *c p2 =o q *c r2$
unfolding *ordIso-iff-ordLeq* **by** (*blast intro: cprod-mono2*)

lemma *cprod-com*: $p1 *c p2 =o p2 *c p1$
by (*simp only: cprod-def card-of-Times-commute*)

lemma *card-of-Csum-Times*:
 $\forall i \in I. |A \ i| \leq_o |B| \implies (\text{CSUM } i : |I|. |A \ i|) \leq_o |I| *c |B|$
by (*simp only: Csum-def cprod-def Field-card-of card-of-Sigma-mono1*)

lemma *card-of-Csum-Times'*:
assumes *Card-order* $r \forall i \in I. |A \ i| \leq_o r$
shows $(\text{CSUM } i : |I|. |A \ i|) \leq_o |I| *c r$
proof –
from *assms(1)* **have** $*$: $r =o |Field \ r|$ **by** (*simp add: card-of-unique*)
with *assms(2)* **have** $\forall i \in I. |A \ i| \leq_o |Field \ r|$ **by** (*blast intro: ordLeq-ordIso-trans*)
hence $(\text{CSUM } i : |I|. |A \ i|) \leq_o |I| *c |Field \ r|$ **by** (*simp only: card-of-Csum-Times*)
also from $*$ **have** $|I| *c |Field \ r| \leq_o |I| *c r$
by (*simp only: Field-card-of card-of-refl cprod-def ordIso-imp-ordLeq*)
finally show *?thesis* .
qed

lemma *cprod-csum-distrib1*: $r1 *c r2 +c r1 *c r3 =o r1 *c (r2 +c r3)$
unfolding *csum-def cprod-def* **by** (*simp add: Field-card-of card-of-Times-Plus-distrib ordIso-symmetric*)

lemma *csum-absorb2'*: $\llbracket \text{Card-order } r2; r1 \leq_o r2; \text{cinfinite } r1 \vee \text{cinfinite } r2 \rrbracket \implies$
 $r1 +c r2 =o r2$
unfolding *csum-def*
using *Card-order-Plus-infinite*
by (*fastforce simp: cinfinite-def dest: cinfinite-mono*)

lemma *csum-absorb1'*:
assumes *card: Card-order* $r2$
and $r12: r1 \leq_o r2$ **and** $cr12: \text{cinfinite } r1 \vee \text{cinfinite } r2$
shows $r2 +c r1 =o r2$
proof –
have $r1 +c r2 =o r2$
by (*simp add: csum-absorb2' assms*)

```

then show ?thesis
  by (blast intro: ordIso-transitive csum-com)
qed

lemma csum-absorb1:  $\llbracket \text{Cinfinite } r2; r1 \leq_o r2 \rrbracket \implies r2 +_c r1 =_o r2$ 
  by (rule csum-absorb1') auto

lemma csum-absorb2:  $\llbracket \text{Cinfinite } r2 ; r1 \leq_o r2 \rrbracket \implies r1 +_c r2 =_o r2$ 
  using ordIso-transitive csum-com csum-absorb1 by blast

lemma regularCard-csum:
  assumes Cinfinite r Cinfinite s regularCard r regularCard s
  shows regularCard (r +_c s)
proof (cases r  $\leq_o$  s)
  case True
  then show ?thesis using regularCard-ordIso[of s] csum-absorb2'[THEN ordIso-symmetric]
  assms by auto
next
  case False
  have Well-order s Well-order r using assms card-order-on-well-order-on by auto
  then have s  $\leq_o$  r using not-ordLeq-iff-ordLess False ordLess-imp-ordLeq by
  auto
  then show ?thesis using regularCard-ordIso[of r] csum-absorb1'[THEN ord-
  dIso-symmetric] assms by auto
qed

lemma csum-mono-strict:
  assumes Card-order: Card-order r Card-order q
  and Cinfinite: Cinfinite r' Cinfinite q'
  and less: r  $<_o$  r' q  $<_o$  q'
  shows r +_c q  $<_o$  r' +_c q'
proof –
  have Well-order: Well-order r Well-order q Well-order r' Well-order q'
  using card-order-on-well-order-on Card-order Cinfinite by auto
  show ?thesis
proof (cases Cinfinite r)
  case outer: True
  then show ?thesis
proof (cases Cinfinite q)
  case inner: True
  then show ?thesis
proof (cases r  $\leq_o$  q)
  case True
  then have r +_c q =_o q using csum-absorb2 inner by blast
  then show ?thesis
  using ordIso-ordLess-trans ordLess-ordLeq-trans less Cinfinite ordLeq-csum2
by blast
next
  case False

```

```

    then have  $q \leq o r$  using not-ordLeq-iff-ordLess Well-order ordLess-imp-ordLeq
  by blast
    then have  $r + c q = o r$  using csum-absorb1 outer by blast
    then show ?thesis
    using ordIso-ordLess-trans ordLess-ordLeq-trans less Cfinite ordLeq-csum1
  by blast
  qed
next
  case False
  then have Cfinite q using Card-order cfinite-def cfinite-def by blast
  then have  $q \leq o r$  using finite-ordLess-infinite cfinite-def cfinite-def outer
    Well-order ordLess-imp-ordLeq by blast
  then have  $r + c q = o r$  by (rule csum-absorb1[OF outer])
  then show ?thesis using ordIso-ordLess-trans ordLess-ordLeq-trans less ord-
dLeq-csum1 Cfinite by blast
  qed
next
  case False
  then have outer: Cfinite r using Card-order cfinite-def cfinite-def by blast
  then show ?thesis
  proof (cases Cfinite q)
    case True
    then have  $r \leq o q$  using finite-ordLess-infinite cfinite-def cfinite-def outer
      Well-order
      ordLess-imp-ordLeq by blast
    then have  $r + c q = o q$  by (rule csum-absorb2[OF True])
    then show ?thesis using ordIso-ordLess-trans ordLess-ordLeq-trans less ord-
      dLeq-csum2 Cfinite by blast
  next
    case False
    then have Cfinite q using Card-order cfinite-def cfinite-def by blast
    then have Cfinite (r + c q) using Cfinite-csum outer by blast
    moreover have Cfinite (r' + c q') using Cfinite-csum1 Cfinite by blast
    ultimately show ?thesis using Cfinite-ordLess-Cfinite by blast
  qed
  qed
qed

```

31.8 Exponentiation

definition *cexp* (infixr \hat{c} 90) where
 $r1 \hat{c} r2 \equiv |\text{Func } (\text{Field } r2) (\text{Field } r1)|$

lemma *Card-order-cexp*: *Card-order (r1 \hat{c} r2)*
 unfolding *cexp-def* by (rule *card-of-Card-order*)

lemma *cexp-mono'*:
 assumes 1: $p1 \leq o r1$ and 2: $p2 \leq o r2$
 and n: $\text{Field } p2 = \{\}$ $\implies \text{Field } r2 = \{\}$

shows $p1 \hat{c} p2 \leq_o r1 \hat{c} r2$
proof(cases $Field\ p1 = \{\}$)
 case *True*
 hence $Field\ p2 \neq \{\} \implies Func\ (Field\ p2)\ \{\} = \{\}$ **unfolding** *Func-is-emp* **by**
simp
 with *True* **have** $|Field\ |Func\ (Field\ p2)\ (Field\ p1)|| \leq_o\ cone$
 unfolding *cone-def* *Field-card-of*
 by (cases $Field\ p2 = \{\}$, *auto* *intro: surj-imp-ordLeq* *simp: Func-empty*)
 hence $|Func\ (Field\ p2)\ (Field\ p1)| \leq_o\ cone$ **by** (*simp* *add: Field-card-of cexp-def*)
 hence $p1 \hat{c} p2 \leq_o\ cone$ **unfolding** *cexp-def* .
 thus *?thesis*
 proof (cases $Field\ p2 = \{\}$)
 case *True*
 with n **have** $Field\ r2 = \{\}$.
 hence $cone \leq_o\ r1 \hat{c} r2$ **unfolding** *cone-def* *cexp-def* *Func-def*
 by (*auto* *intro: card-of-ordLeqI*[**where** $f=\lambda\ -.\ undefined$])
 thus *?thesis* **using** $\langle p1 \hat{c} p2 \leq_o\ cone \rangle$ *ordLeq-transitive* **by** *auto*
 next
 case *False* **with** *True* **have** $|Field\ (p1 \hat{c} p2)| =_o\ czero$
 unfolding *card-of-ordIso-czero-iff-empty* *cexp-def* *Field-card-of* *Func-def* **by**
auto
 thus *?thesis* **unfolding** *cexp-def* *card-of-ordIso-czero-iff-empty* *Field-card-of*
 by (*simp* *add: card-of-empty*)
 qed
 next
 case *False*
 have $1: |Field\ p1| \leq_o\ |Field\ r1|$ **and** $2: |Field\ p2| \leq_o\ |Field\ r2|$
 using $1\ 2$ **by** (*auto* *simp: card-of-mono2*)
 obtain $f1$ **where** $f1: f1 \text{ ‘ } Field\ r1 = Field\ p1$
 using 1 **unfolding** *card-of-ordLeq2*[*OF* *False, symmetric*] **by** *auto*
 obtain $f2$ **where** $f2: inj\ on\ f2\ (Field\ p2)\ f2 \text{ ‘ } Field\ p2 \subseteq Field\ r2$
 using 2 **unfolding** *card-of-ordLeq*[*symmetric*] **by** *blast*
 have $0: Func\ map\ (Field\ p2)\ f1\ f2 \text{ ‘ } (Field\ (r1 \hat{c} r2)) = Field\ (p1 \hat{c} p2)$
 unfolding *cexp-def* *Field-card-of* **using** *Func-map-surj*[*OF* $f1\ f2\ n,$ *symmetric*]
 .
 have $00: Field\ (p1 \hat{c} p2) \neq \{\}$ **unfolding** *cexp-def* *Field-card-of* *Func-is-emp*
 using *False* **by** *simp*
 show *?thesis*
 using 0 *card-of-ordLeq2*[*OF* 00] **unfolding** *cexp-def* *Field-card-of* **by** *blast*
 qed

lemma *cexp-mono*:
 assumes $1: p1 \leq_o\ r1$ **and** $2: p2 \leq_o\ r2$
 and $n: p2 =_o\ czero \implies r2 =_o\ czero$ **and** *card: Card-order* $p2$
 shows $p1 \hat{c} p2 \leq_o\ r1 \hat{c} r2$
 by (*rule* *cexp-mono'*[*OF* $1\ 2\ czeroE$ [*OF* n [*OF* *czeroI*[*OF* *card*]]]])

lemma *cexp-mono1*:
 assumes $1: p1 \leq_o\ r1$ **and** $q: Card-order\ q$

shows $p1 \hat{c} q \leq_o r1 \hat{c} q$
 using *ordLeq-refl*[OF q] by (rule *cexp-mono*[OF 1]) (auto simp: q)

lemma *cexp-mono2'*:

assumes $2: p2 \leq_o r2$ and $q: \text{Card-order } q$
 and $n: \text{Field } p2 = \{\} \implies \text{Field } r2 = \{\}$
 shows $q \hat{c} p2 \leq_o q \hat{c} r2$
 using *ordLeq-refl*[OF q] by (rule *cexp-mono'*[OF - 2 n]) auto

lemma *cexp-mono2*:

assumes $2: p2 \leq_o r2$ and $q: \text{Card-order } q$
 and $n: p2 =_o \text{czero} \implies r2 =_o \text{czero}$ and $\text{card}: \text{Card-order } p2$
 shows $q \hat{c} p2 \leq_o q \hat{c} r2$
 using *ordLeq-refl*[OF q] by (rule *cexp-mono*[OF - 2 n card]) auto

lemma *cexp-mono2-Cnotzero*:

assumes $p2 \leq_o r2$ *Card-order* q *Cnotzero* $p2$
 shows $q \hat{c} p2 \leq_o q \hat{c} r2$
 using *assms*(3) *czeroI* by (blast intro: *cexp-mono2'*[OF *assms*(1,2)])

lemma *cexp-cong*:

assumes $1: p1 =_o r1$ and $2: p2 =_o r2$
 and $Cr: \text{Card-order } r2$
 and $Cp: \text{Card-order } p2$
 shows $p1 \hat{c} p2 =_o r1 \hat{c} r2$

proof –

obtain f where *bij-betw* f (*Field* $p2$) (*Field* $r2$)
 using 2 *card-of-ordIso*[of *Field* $p2$ *Field* $r2$] *card-of-cong* by auto
 hence $0: \text{Field } p2 = \{\} \longleftrightarrow \text{Field } r2 = \{\}$ **unfolding** *bij-betw-def* by auto
 have $r: p2 =_o \text{czero} \implies r2 =_o \text{czero}$
 and $p: r2 =_o \text{czero} \implies p2 =_o \text{czero}$
 using 0 Cr Cp *czeroE* *czeroI* by auto
 show *?thesis* using 0 1 2 **unfolding** *ordIso-iff-ordLeq*
 using r p *cexp-mono*[OF - - - Cp] *cexp-mono*[OF - - - Cr] by blast

qed

lemma *cexp-cong1*:

assumes $1: p1 =_o r1$ and $q: \text{Card-order } q$
 shows $p1 \hat{c} q =_o r1 \hat{c} q$
 by (rule *cexp-cong*[OF 1 - q]) (rule *ordIso-refl*[OF q])

lemma *cexp-cong2*:

assumes $2: p2 =_o r2$ and $q: \text{Card-order } q$ and $p: \text{Card-order } p2$
 shows $q \hat{c} p2 =_o q \hat{c} r2$
 by (rule *cexp-cong*[OF - 2]) (auto simp only: *ordIso-refl* *Card-order-ordIso2*[OF p 2] q p)

lemma *cexp-cone*:

assumes *Card-order* r

shows $r \hat{c} \text{ cone} =_o r$
proof –
 have $r \hat{c} \text{ cone} =_o |Field\ r|$
 unfolding *cexp-def cone-def Field-card-of Func-empty*
card-of-ordIso[symmetric] bij-betw-def Func-def inj-on-def image-def
 by (rule *exI[of - $\lambda f. f$ ()]*) *auto*
 also have $|Field\ r| =_o r$ by (rule *card-of-Field-ordIso[OF assms]*)
 finally show *?thesis* .
qed

lemma *cexp-cprod*:
 assumes $r1$: *Card-order* $r1$
 shows $(r1 \hat{c} r2) \hat{c} r3 =_o r1 \hat{c} (r2 *c r3)$ (is $?L =_o ?R$)
proof –
 have $?L =_o r1 \hat{c} (r3 *c r2)$
 unfolding *cprod-def cexp-def Field-card-of*
 using *card-of-Func-Times* by (rule *ordIso-symmetric*)
 also have $r1 \hat{c} (r3 *c r2) =_o ?R$
 using *cprod-com* $r1$ by (intro *cexp-cong2*, *auto simp: Card-order-cprod*)
 finally show *?thesis* .
qed

lemma *cprod-infinite1'*: $\llbracket Cinfinite\ r; Cnotzero\ p; p \leq_o r \rrbracket \implies r *c p =_o r$
 unfolding *cinfinite-def cprod-def*
 by (rule *Card-order-Times-infinite[THEN conjunct1]*) (blast intro: *czeroI*)+

lemma *cprod-infinite*: $Cinfinite\ r \implies r *c r =_o r$
 using *cprod-infinite1' Cinfinite-Cnotzero ordLeq-refl* by *blast*

lemma *cexp-cprod-ordLeq*:
 assumes $r1$: *Card-order* $r1$ and $r2$: *Cinfinite* $r2$
 and $r3$: *Cnotzero* $r3$ $r3 \leq_o r2$
 shows $(r1 \hat{c} r2) \hat{c} r3 =_o r1 \hat{c} r2$ (is $?L =_o ?R$)
proof –
 have $?L =_o r1 \hat{c} (r2 *c r3)$ using *cexp-cprod[OF r1]* .
 also have $r1 \hat{c} (r2 *c r3) =_o ?R$
 using *assms* by (fastforce *simp: Card-order-cprod intro: cprod-infinite1' cexp-cong2*)
 finally show *?thesis* .
qed

lemma *Cnotzero-UNIV*: $Cnotzero\ |UNIV|$
 by (*auto simp: card-of-Card-order card-of-ordIso-czero-iff-empty*)

lemma *ordLess-ctwo-cexp*:
 assumes *Card-order* r
 shows $r <_o \text{ctwo} \hat{c} r$
proof –
 have $r <_o |Pow\ (Field\ r)|$ using *assms* by (rule *Card-order-Pow*)
 also have $|Pow\ (Field\ r)| =_o \text{ctwo} \hat{c} r$

unfolding *ctwo-def cexp-def Field-card-of* **by** (rule *card-of-Pow-Func*)
finally show *?thesis* .
qed

lemma *ordLeq-cexp1*:
assumes *Cnotzero r Card-order q*
shows $q \leq_o q \hat{c} r$
proof (*cases q = o (czero :: 'a rel)*)
case *True* **thus** *?thesis* **by** (*simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans*)
next
case *False*
have $q =_o q \hat{c} cone$
by (*blast intro: assms ordIso-symmetric cexp-cone*)
also have $q \hat{c} cone \leq_o q \hat{c} r$
using *assms*
by (*intro cexp-mono2*) (*simp-all add: cone-ordLeq-Cnotzero cone-not-czero Card-order-cone*)
finally show *?thesis* .
qed

lemma *ordLeq-cexp2*:
assumes $ctwo \leq_o q$ *Card-order r*
shows $r \leq_o q \hat{c} r$
proof (*cases r = o (czero :: 'a rel)*)
case *True* **thus** *?thesis* **by** (*simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans*)
next
case *False*
have $r <_o ctwo \hat{c} r$
by (*blast intro: assms ordLess-ctwo-cexp*)
also have $ctwo \hat{c} r \leq_o q \hat{c} r$
by (*blast intro: assms cexp-mono1*)
finally show *?thesis* **by** (rule *ordLess-imp-ordLeq*)
qed

lemma *cinfinite-cexp*: $\llbracket ctwo \leq_o q; Cinfinite r \rrbracket \implies cinfinite (q \hat{c} r)$
by (rule *cinfinite-mono[OF ordLeq-cexp2]*) *simp-all*

lemma *Cinfinite-cexp*:
 $\llbracket ctwo \leq_o q; Cinfinite r \rrbracket \implies Cinfinite (q \hat{c} r)$
by (*simp add: cinfinite-cexp Card-order-cexp*)

lemma *card-order-cexp*:
assumes *card-order r1 card-order r2*
shows *card-order (r1 \hat{c} r2)*
proof –
have *Field r1 = UNIV Field r2 = UNIV* **using** *assms card-order-on-Card-order*
by *auto*

thus *?thesis* **unfolding** *cexp-def Func-def* **using** *card-of-card-order-on* **by** *simp*
qed

lemma *ctwo-ordLess-natLeq*: *ctwo < o natLeq*
unfolding *ctwo-def* **using** *finite-UNIV natLeq-cinfinite natLeq-Card-order*
by (*intro Cfinite-ordLess-Cinfinite*) (*auto simp: cfinite-def card-of-Card-order*)

lemma *ctwo-ordLess-Cinfinite*: *Cinfinite r \implies ctwo < o r*
by (*rule ordLess-ordLeq-trans[OF ctwo-ordLess-natLeq natLeq-ordLeq-cinfinite]*)

lemma *ctwo-ordLeq-Cinfinite*:
assumes *Cinfinite r*
shows *ctwo \leq o r*
by (*rule ordLess-imp-ordLeq[OF ctwo-ordLess-Cinfinite[OF assms]]*)

lemma *Un-Cinfinite-bound*: $\llbracket |A| \leq o r; |B| \leq o r; Cinfinite r \rrbracket \implies |A \cup B| \leq o r$
by (*auto simp add: cinfinite-def card-of-Un-ordLeq-infinite-Field*)

lemma *Un-Cinfinite-bound-strict*: $\llbracket |A| < o r; |B| < o r; Cinfinite r \rrbracket \implies |A \cup B| < o r$
by (*auto simp add: cinfinite-def card-of-Un-ordLess-infinite-Field*)

lemma *UNION-Cinfinite-bound*: $\llbracket |I| \leq o r; \forall i \in I. |A i| \leq o r; Cinfinite r \rrbracket \implies |\bigcup_{i \in I} A i| \leq o r$
by (*auto simp add: card-of-UNION-ordLeq-infinite-Field cinfinite-def*)

lemma *csum-cinfinite-bound*:
assumes *p \leq o r q \leq o r Card-order p Card-order q Cinfinite r*
shows *p + c q \leq o r*
proof –
have *|Field p| \leq o r |Field q| \leq o r*
using *assms card-of-least ordLeq-transitive* **unfolding** *card-order-on-def* **by**
blast+
with *assms* **show** *?thesis* **unfolding** *cinfinite-def csum-def*
by (*blast intro: card-of-Plus-ordLeq-infinite-Field*)
qed

lemma *cprod-cinfinite-bound*:
assumes *p \leq o r q \leq o r Card-order p Card-order q Cinfinite r*
shows *p * c q \leq o r*
proof –
from *assms(1-4)* **have** *|Field p| \leq o r |Field q| \leq o r*
unfolding *card-order-on-def* **using** *card-of-least ordLeq-transitive* **by** *blast+*
with *assms* **show** *?thesis* **unfolding** *cinfinite-def cprod-def*
by (*blast intro: card-of-Times-ordLeq-infinite-Field*)
qed

lemma *cprod-infinite2'*: $\llbracket Cnotzero r1; Cinfinite r2; r1 \leq o r2 \rrbracket \implies r1 * c r2 = o r2$

unfolding *ordIso-iff-ordLeq*
by (*intro conjI cprod-cinfinite-bound ordLeq-cprod2 ordLeq-refl*)
(auto dest!: ordIso-imp-ordLeq not-ordLeq-ordLess simp: czero-def Card-order-empty)

lemma *regularCard-cprod*:
assumes *Cinfinite r Cinfinite s regularCard r regularCard s*
shows *regularCard (r *c s)*
proof (*cases r ≤o s*)
case *True*
with *assms Cinfinite-Cnotzero* **show** *?thesis*
by (*force intro: regularCard-ordIso ordIso-symmetric[OF cprod-infinite2]*)
next
case *False*
have *Well-order r Well-order s* **using** *assms card-order-on-well-order-on* **by** *auto*
then have *s ≤o r* **using** *not-ordLeq-iff-ordLess ordLess-imp-ordLeq False* **by**
blast
with *assms Cinfinite-Cnotzero* **show** *?thesis*
by (*force intro: regularCard-ordIso ordIso-symmetric[OF cprod-infinite1]*)
qed

lemma *cprod-csum-cexp*:
 $r1 *c r2 \leq_o (r1 +c r2) \hat{c} ctwo$
unfolding *cprod-def csum-def cexp-def ctwo-def Field-card-of*
proof –
let *?f = λ(a, b). %x. if x then Inl a else Inr b*
have *inj-on ?f (Field r1 × Field r2) (is inj-on - ?LHS)*
by (*auto simp: inj-on-def fun-eq-iff split: bool.split*)
moreover
have *?f ‘ ?LHS ⊆ Func (UNIV :: bool set) (Field r1 <+> Field r2) (is - ⊆ ?RHS)*
by (*auto simp: Func-def*)
ultimately show $|?LHS| \leq_o |?RHS|$ **using** *card-of-ordLeq* **by** *blast*
qed

lemma *Cfinite-cprod-Cinfinite*: $\llbracket Cfinite\ r; Cinfinite\ s \rrbracket \implies r *c s \leq_o s$
by (*intro cprod-cinfinite-bound*)
(auto intro: ordLeq-refl ordLess-imp-ordLeq[OF Cfinite-ordLess-Cinfinite])

lemma *cprod-cexp*: $(r *c s) \hat{c} t =_o r \hat{c} t *c s \hat{c} t$
unfolding *cprod-def cexp-def Field-card-of* **by** (*rule Func-Times-Range*)

lemma *cprod-cexp-csum-cexp-Cinfinite*:
assumes *t: Cinfinite t*
shows $(r *c s) \hat{c} t \leq_o (r +c s) \hat{c} t$
proof –
have $(r *c s) \hat{c} t \leq_o ((r +c s) \hat{c} ctwo) \hat{c} t$
by (*rule cexp-mono1[OF cprod-csum-cexp conjunct2[OF t]]*)
also have $((r +c s) \hat{c} ctwo) \hat{c} t =_o (r +c s) \hat{c} (ctwo *c t)$
by (*rule cexp-cprod[OF Card-order-csum]*)

also have $(r + c s) \hat{c} (ctwo * c t) =_o (r + c s) \hat{c} (t * c ctwo)$
by (rule *cexp-cong2*[OF *cprod-com* *Card-order-csum* *Card-order-cprod*])
also have $(r + c s) \hat{c} (t * c ctwo) =_o ((r + c s) \hat{c} t) \hat{c} ctwo$
by (rule *ordIso-symmetric*[OF *cexp-cprod*[OF *Card-order-csum*]])
also have $((r + c s) \hat{c} t) \hat{c} ctwo =_o (r + c s) \hat{c} t$
by (rule *cexp-cprod-ordLeq*[OF *Card-order-csum* *t ctwo-Cnotzero* *ctwo-ordLeq-Cinfinite*[OF
t]])
finally show *?thesis* .
qed

lemma *Cfinite-cexp-Cinfinite*:

assumes *s*: *Cfinite s* **and** *t*: *Cinfinite t*
shows $s \hat{c} t \leq_o ctwo \hat{c} t$
proof (cases *s* $\leq_o ctwo$)
case *True* **thus** *?thesis* **using** *t* **by** (*blast intro: cexp-mono1*)
next
case *False*
hence $ctwo \leq_o s$ **using** *ordLeq-total*[of *s ctwo*] *Card-order-ctwo s*
by (*auto intro: card-order-on-well-order-on*)
hence *Cnotzero s* **using** *Cnotzero-mono*[OF *ctwo-Cnotzero*] *s* **by** *blast*
hence *st*: *Cnotzero (s * c t)* **by** (*intro Cinfinite-Cnotzero*[OF *Cinfinite-cprod2*])
(*auto simp: t*)
have $s \hat{c} t \leq_o (ctwo \hat{c} s) \hat{c} t$
using *assms* **by** (*blast intro: cexp-mono1 ordLess-imp-ordLeq*[OF *ordLess-ctwo-cexp*])
also have $(ctwo \hat{c} s) \hat{c} t =_o ctwo \hat{c} (s * c t)$
by (*blast intro: Card-order-ctwo cexp-cprod*)
also have $ctwo \hat{c} (s * c t) \leq_o ctwo \hat{c} t$
using *assms st* **by** (*intro cexp-mono2-Cnotzero Cfinite-cprod-Cinfinite* *Card-order-ctwo*)
finally show *?thesis* .
qed

lemma *csum-Cfinite-cexp-Cinfinite*:

assumes *r*: *Card-order r* **and** *s*: *Cfinite s* **and** *t*: *Cinfinite t*
shows $(r + c s) \hat{c} t \leq_o (r + c ctwo) \hat{c} t$
proof (cases *Cinfinite r*)
case *True*
hence $r + c s =_o r$ **by** (*intro csum-absorb1 ordLess-imp-ordLeq*[OF *Cfinite-ordLess-Cinfinite*]
s)
hence $(r + c s) \hat{c} t =_o r \hat{c} t$ **using** *t* **by** (*blast intro: cexp-cong1*)
also have $r \hat{c} t \leq_o (r + c ctwo) \hat{c} t$ **using** *t* **by** (*blast intro: cexp-mono1*
ordLeq-csum1 r)
finally show *?thesis* .
next
case *False*
with *r* **have** *Cfinite r* **unfolding** *cinfinite-def* *cfinite-def* **by** *auto*
hence *Cfinite (r + c s)* **by** (*intro Cfinite-csum s*)
hence $(r + c s) \hat{c} t \leq_o ctwo \hat{c} t$ **by** (*intro Cfinite-cexp-Cinfinite t*)
also have $ctwo \hat{c} t \leq_o (r + c ctwo) \hat{c} t$ **using** *t*
by (*blast intro: cexp-mono1 ordLeq-csum2* *Card-order-ctwo*)

finally show *?thesis* .
qed

lemma *Cinfinite-cardSuc*: $Cinfinite\ r \implies Cinfinite\ (cardSuc\ r)$
by (*simp add: cinfinite-def cardSuc-Card-order cardSuc-finite*)

lemma *cardSuc-UNION-Cinfinite*:
assumes $Cinfinite\ r\ relChain\ (cardSuc\ r)\ As\ B \leq (\bigcup i \in Field\ (cardSuc\ r)).\ As$
i) $|B| \leq_o\ r$
shows $\exists i \in Field\ (cardSuc\ r).\ B \leq As\ i$
using *cardSuc-UNION assms unfolding cinfinite-def by blast*

lemma *Cinfinite-card-suc*: $\llbracket Cinfinite\ r ; card-order\ r \rrbracket \implies Cinfinite\ (card-suc\ r)$
using *Cinfinite-cong[OF cardSuc-ordIso-card-suc Cinfinite-cardSuc]* .

lemma *card-suc-least*: $\llbracket card-order\ r ; Card-order\ s ; r <_o\ s \rrbracket \implies card-suc\ r \leq_o\ s$
by (*rule ordIso-ordLeq-trans[OF ordIso-symmetric[OF cardSuc-ordIso-card-suc]]*)
(auto intro!: cardSuc-least simp: card-order-on-Card-order)

lemma *regularCard-cardSuc*: $Cinfinite\ k \implies regularCard\ (cardSuc\ k)$
by (*rule infinite-cardSuc-regularCard*) *(auto simp: cinfinite-def)*

lemma *regularCard-card-suc*: $card-order\ r \implies Cinfinite\ r \implies regularCard\ (card-suc\ r)$
using *cardSuc-ordIso-card-suc Cinfinite-cardSuc regularCard-cardSuc regularCard-ordIso*
by *blast*

end

32 Function Definition Base

theory *Fun-Def-Base*
imports *Ctr-Sugar Set Wellfounded*
begin

ML-file $\langle Tools/Function/function-lib.ML \rangle$

named-theorems *termination-simp simplification rules for termination proofs*

ML-file $\langle Tools/Function/function-common.ML \rangle$

ML-file $\langle Tools/Function/function-context-tree.ML \rangle$

attribute-setup *fundef-cong =*

$\langle Attrib.add-del\ Function-Context-Tree.cong-add\ Function-Context-Tree.cong-del \rangle$
declaration of congruence rule for function definitions

ML-file $\langle Tools/Function/sum-tree.ML \rangle$

end

33 Definition of Bounded Natural Functors

theory *BNF-Def*

imports *BNF-Cardinal-Arithmetic Fun-Def-Base*

keywords

print-bnfs :: *diag* **and**

bnf :: *thy-goal-defn*

begin

lemma *Collect-case-prodD*: $x \in \text{Collect } (\text{case-prod } A) \implies A (\text{fst } x) (\text{snd } x)$

by *auto*

inductive

rel-sum :: $('a \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow 'a + 'b \Rightarrow 'c + 'd \Rightarrow \text{bool}$

for *R1 R2*

where

$R1 \ a \ c \implies \text{rel-sum } R1 \ R2 \ (\text{Inl } a) \ (\text{Inl } c)$

$| \ R2 \ b \ d \implies \text{rel-sum } R1 \ R2 \ (\text{Inr } b) \ (\text{Inr } d)$

definition

rel-fun :: $('a \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow \text{bool}$

where

$\text{rel-fun } A \ B = (\lambda f \ g. \forall x \ y. A \ x \ y \longrightarrow B \ (f \ x) \ (g \ y))$

lemma *rel-funI* [*intro*]:

assumes $\bigwedge x \ y. A \ x \ y \implies B \ (f \ x) \ (g \ y)$

shows $\text{rel-fun } A \ B \ f \ g$

using *assms* **by** (*simp add: rel-fun-def*)

lemma *rel-funD*:

assumes $\text{rel-fun } A \ B \ f \ g$ **and** $A \ x \ y$

shows $B \ (f \ x) \ (g \ y)$

using *assms* **by** (*simp add: rel-fun-def*)

lemma *rel-fun-mono*:

$\llbracket \text{rel-fun } X \ A \ f \ g; \bigwedge x \ y. Y \ x \ y \longrightarrow X \ x \ y; \bigwedge x \ y. A \ x \ y \implies B \ x \ y \rrbracket \implies \text{rel-fun } Y \ B \ f \ g$

by(*simp add: rel-fun-def*)

lemma *rel-fun-mono'* [*mono*]:

$\llbracket \bigwedge x \ y. Y \ x \ y \longrightarrow X \ x \ y; \bigwedge x \ y. A \ x \ y \longrightarrow B \ x \ y \rrbracket \implies \text{rel-fun } X \ A \ f \ g \longrightarrow \text{rel-fun } Y \ B \ f \ g$

by(*simp add: rel-fun-def*)

definition *rel-set* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \ \text{set} \Rightarrow 'b \ \text{set} \Rightarrow \text{bool}$

where $\text{rel-set } R = (\lambda A \ B. (\forall x \in A. \exists y \in B. R \ x \ y) \wedge (\forall y \in B. \exists x \in A. R \ x \ y))$

lemma *rel-setI*:

assumes $\bigwedge x. x \in A \implies \exists y \in B. R \ x \ y$

assumes $\bigwedge y. y \in B \implies \exists x \in A. R x y$
shows *rel-set* $R A B$
using *assms* **unfolding** *rel-set-def* **by** *simp*

lemma *predicate2-transferD*:

$\llbracket \text{rel-fun } R1 \text{ (rel-fun } R2 \text{ (=)) } P Q; a \in A; b \in B; A \subseteq \{(x, y). R1 x y\}; B \subseteq \{(x, y). R2 x y\} \rrbracket \implies$
 $P (fst a) (fst b) \longleftrightarrow Q (snd a) (snd b)$
unfolding *rel-fun-def* **by** (*blast dest!*: *Collect-case-prodD*)

definition *collect where*

collect $F x = (\bigcup f \in F. f x)$

lemma *fstI*: $x = (y, z) \implies fst x = y$
by *simp*

lemma *sndI*: $x = (y, z) \implies snd x = z$
by *simp*

lemma *bijI'*: $\llbracket \bigwedge x y. (f x = f y) = (x = y); \bigwedge y. \exists x. y = f x \rrbracket \implies \text{bij } f$
unfolding *bij-def inj-on-def* **by** *auto blast*

definition *Gr A f* = $\{(a, f a) \mid a. a \in A\}$

definition *Grp A f* = $(\lambda a b. b = f a \wedge a \in A)$

definition *vimage2p where*

vimage2p $f g R = (\lambda x y. R (f x) (g y))$

lemma *collect-comp*: *collect* $F \circ g = \text{collect } ((\lambda f. f \circ g) ' F)$
by (*rule ext*) (*simp add*: *collect-def*)

definition *convol ((-, / -)) where*

$\langle f, g \rangle \equiv \lambda a. (f a, g a)$

lemma *fst-convol*: $fst \circ \langle f, g \rangle = f$
apply(*rule ext*)
unfolding *convol-def* **by** *simp*

lemma *snd-convol*: $snd \circ \langle f, g \rangle = g$
apply(*rule ext*)
unfolding *convol-def* **by** *simp*

lemma *convol-mem-GrpI*:

$x \in A \implies \langle id, g \rangle x \in (\text{Collect } (\text{case-prod } (Grp A g)))$
unfolding *convol-def Grp-def* **by** *auto*

definition *csquare where*

csquare $A f1 f2 p1 p2 \longleftrightarrow (\forall a \in A. f1 (p1 a) = f2 (p2 a))$

lemma *eq-alt*: $(=) = \text{Grp UNIV id}$
unfolding *Grp-def* **by** *auto*

lemma *leq-conversepI*: $R = (=) \implies R \leq R^{-1-1}$
by *auto*

lemma *leq-OOI*: $R = (=) \implies R \leq R \text{ OO } R$
by *auto*

lemma *OO-Grp-alt*: $(\text{Grp } A f)^{-1-1} \text{ OO } \text{Grp } A g = (\lambda x y. \exists z. z \in A \wedge f z = x \wedge g z = y)$
unfolding *Grp-def* **by** *auto*

lemma *Grp-UNIV-id*: $f = \text{id} \implies (\text{Grp UNIV } f)^{-1-1} \text{ OO } \text{Grp UNIV } f = \text{Grp UNIV } f$
unfolding *Grp-def* **by** *auto*

lemma *Grp-UNIV-idI*: $x = y \implies \text{Grp UNIV id } x y$
unfolding *Grp-def* **by** *auto*

lemma *Grp-mono*: $A \leq B \implies \text{Grp } A f \leq \text{Grp } B f$
unfolding *Grp-def* **by** *auto*

lemma *GrpI*: $\llbracket f x = y; x \in A \rrbracket \implies \text{Grp } A f x y$
unfolding *Grp-def* **by** *auto*

lemma *GrpE*: $\text{Grp } A f x y \implies (\llbracket f x = y; x \in A \rrbracket \implies R) \implies R$
unfolding *Grp-def* **by** *auto*

lemma *Collect-case-prod-Grp-eqD*: $z \in \text{Collect } (\text{case-prod } (\text{Grp } A f)) \implies (f \circ \text{fst}) z = \text{snd } z$
unfolding *Grp-def comp-def* **by** *auto*

lemma *Collect-case-prod-Grp-in*: $z \in \text{Collect } (\text{case-prod } (\text{Grp } A f)) \implies \text{fst } z \in A$
unfolding *Grp-def comp-def* **by** *auto*

definition *pick-middlep* $P Q a c = (\text{SOME } b. P a b \wedge Q b c)$

lemma *pick-middlep*:
 $(P \text{ OO } Q) a c \implies P a (\text{pick-middlep } P Q a c) \wedge Q (\text{pick-middlep } P Q a c) c$
unfolding *pick-middlep-def* **by** *(rule someI-ex) auto*

definition *fstOp* **where**
 $\text{fstOp } P Q ac = (\text{fst } ac, \text{pick-middlep } P Q (\text{fst } ac) (\text{snd } ac))$

definition *sndOp* **where**
 $\text{sndOp } P Q ac = (\text{pick-middlep } P Q (\text{fst } ac) (\text{snd } ac), (\text{snd } ac))$

lemma *fstOp-in*: $ac \in Collect (case\text{-}prod (P \text{ OO } Q)) \implies fstOp P Q ac \in Collect (case\text{-}prod P)$

unfolding *fstOp-def mem-Collect-eq*
by (*subst (asm) surjective-pairing, unfold prod.case*) (*erule pick-middlep[THEN conjunct1]*)

lemma *fst-fstOp*: $fst bc = (fst \circ fstOp P Q) bc$

unfolding *comp-def fstOp-def* **by** *simp*

lemma *snd-sndOp*: $snd bc = (snd \circ sndOp P Q) bc$

unfolding *comp-def sndOp-def* **by** *simp*

lemma *sndOp-in*: $ac \in Collect (case\text{-}prod (P \text{ OO } Q)) \implies sndOp P Q ac \in Collect (case\text{-}prod Q)$

unfolding *sndOp-def mem-Collect-eq*
by (*subst (asm) surjective-pairing, unfold prod.case*) (*erule pick-middlep[THEN conjunct2]*)

lemma *csquare-fstOp-sndOp*:

csquare (Collect (f (P OO Q))) snd fst (fstOp P Q) (sndOp P Q)

unfolding *csquare-def fstOp-def sndOp-def* **using** *pick-middlep* **by** *simp*

lemma *snd-fst-flip*: $snd xy = (fst \circ (\% (x, y). (y, x))) xy$

by (*simp split: prod.split*)

lemma *fst-snd-flip*: $fst xy = (snd \circ (\% (x, y). (y, x))) xy$

by (*simp split: prod.split*)

lemma *flip-pred*: $A \subseteq Collect (case\text{-}prod (R^{-1-1})) \implies (\% (x, y). (y, x)) ' A \subseteq Collect (case\text{-}prod R)$

by *auto*

lemma *predicate2-eqD*: $A = B \implies A a b \longleftrightarrow B a b$

by *simp*

lemma *case-sum-o-inj*: $case\text{-}sum f g \circ Inl = f case\text{-}sum f g \circ Inr = g$

by *auto*

lemma *map-sum-o-inj*: $map\text{-}sum f g \circ Inl = Inl \circ f map\text{-}sum f g \circ Inr = Inr \circ g$

by *auto*

lemma *card-order-csum-cone-cexp-def*:

card-order r \implies (|A1| + c cone) ^c r = |Func UNIV (Inl ' A1 \cup \{Inr ()\})|

unfolding *cexp-def cone-def Field-csum Field-card-of* **by** (*auto dest: Field-card-order*)

lemma *If-the-inv-into-in-Func*:

$\llbracket inj\text{-}on g C; C \subseteq B \cup \{x\} \rrbracket \implies$

$(\lambda i. \text{if } i \in g ' C \text{ then the-inv-into } C g i \text{ else } x) \in Func UNIV (B \cup \{x\})$

unfolding *Func-def* **by** (*auto dest: the-inv-into-into*)

lemma *If-the-inv-into-f-f*:

$\llbracket i \in C; \text{inj-on } g \ C \rrbracket \implies ((\lambda i. \text{if } i \in g \ ' \ C \text{ then the-inv-into } C \ g \ i \text{ else } x) \circ g) \ i = \text{id } i$

unfolding *Func-def* **by** (*auto elim: the-inv-into-f-f*)

lemma *the-inv-f-o-f-id*: $\text{inj } f \implies (\text{the-inv } f \circ f) \ z = \text{id } z$

by (*simp add: the-inv-f-f*)

lemma *vimage2pI*: $R \ (f \ x) \ (g \ y) \implies \text{vimage2p } f \ g \ R \ x \ y$

unfolding *vimage2p-def* .

lemma *rel-fun-iff-leq-vimage2p*: $(\text{rel-fun } R \ S) \ f \ g = (R \leq \text{vimage2p } f \ g \ S)$

unfolding *rel-fun-def vimage2p-def* **by** *auto*

lemma *convol-image-vimage2p*: $\langle f \circ \text{fst}, g \circ \text{snd} \rangle \ ' \ \text{Collect } (\text{case-prod } (\text{vimage2p } f \ g \ R)) \subseteq \text{Collect } (\text{case-prod } R)$

unfolding *vimage2p-def convol-def* **by** *auto*

lemma *vimage2p-Grp*: $\text{vimage2p } f \ g \ P = \text{Grp } \text{UNIV } f \ \text{OO } P \ \text{OO } (\text{Grp } \text{UNIV } g)^{-1-1}$

unfolding *vimage2p-def Grp-def* **by** *auto*

lemma *subst-Pair*: $P \ x \ y \implies a = (x, y) \implies P \ (\text{fst } a) \ (\text{snd } a)$

by *simp*

lemma *comp-apply-eq*: $f \ (g \ x) = h \ (k \ x) \implies (f \circ g) \ x = (h \circ k) \ x$

unfolding *comp-apply* **by** *assumption*

lemma *refl-ge-eq*: $(\bigwedge x. R \ x \ x) \implies (=) \leq R$

by *auto*

lemma *ge-eq-refl*: $(=) \leq R \implies R \ x \ x$

by *auto*

lemma *reflp-eq*: $\text{reflp } R = ((=) \leq R)$

by (*auto simp: reflp-def fun-eq-iff*)

lemma *transp-relcompp*: $\text{transp } r \longleftrightarrow r \ \text{OO } r \leq r$

by (*auto simp: transp-def*)

lemma *symp-conversep*: $\text{symp } R = (R^{-1-1} \leq R)$

by (*auto simp: symp-def fun-eq-iff*)

lemma *diag-imp-eq-le*: $(\bigwedge x. x \in A \implies R \ x \ x) \implies \forall x \ y. x \in A \longrightarrow y \in A \longrightarrow x = y \longrightarrow R \ x \ y$

by *blast*

definition *eq-onp* :: ('a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool
where *eq-onp* R = (λx y. R x ∧ x = y)

lemma *eq-onp-Grp*: *eq-onp* P = *BNF-Def.Grp* (Collect P) *id*
unfolding *eq-onp-def Grp-def* **by** *auto*

lemma *eq-onp-to-eq*: *eq-onp* P x y ⇒ x = y
by (*simp add: eq-onp-def*)

lemma *eq-onp-top-eq-eq*: *eq-onp* top = (=)
by (*simp add: eq-onp-def*)

lemma *eq-onp-same-args*: *eq-onp* P x x = P x
by (*auto simp add: eq-onp-def*)

lemma *eq-onp-eqD*: *eq-onp* P = Q ⇒ P x = Q x x
unfolding *eq-onp-def* **by** *blast*

lemma *Ball-Collect*: *Ball* A P = (A ⊆ (Collect P))
by *auto*

lemma *eq-onp-mono0*: ∀ x ∈ A. P x → Q x ⇒ ∀ x ∈ A. ∀ y ∈ A. *eq-onp* P x y →
eq-onp Q x y
unfolding *eq-onp-def* **by** *auto*

lemma *eq-onp-True*: *eq-onp* (λ-. True) = (=)
unfolding *eq-onp-def* **by** *simp*

lemma *Ball-image-comp*: *Ball* (f ' A) g = *Ball* A (g ∘ f)
by *auto*

lemma *rel-fun-Collect-case-prodD*:
rel-fun A B f g ⇒ X ⊆ Collect (case-prod A) ⇒ x ∈ X ⇒ B ((f ∘ fst) x) ((g
 ∘ snd) x)
unfolding *rel-fun-def* **by** *auto*

lemma *eq-onp-mono-iff*: *eq-onp* P ≤ *eq-onp* Q ⇔ P ≤ Q
unfolding *eq-onp-def* **by** *auto*

ML-file <Tools/BNF/bnf-util.ML>

ML-file <Tools/BNF/bnf-tactics.ML>

ML-file <Tools/BNF/bnf-def-tactics.ML>

ML-file <Tools/BNF/bnf-def.ML>

end

34 Composition of Bounded Natural Functors

theory *BNF-Composition*

imports *BNF-Def*
begin

lemma *ssubst-mem*: $\llbracket t = s; s \in X \rrbracket \implies t \in X$
by *simp*

lemma *empty-natural*: $(\lambda-. \{\}) \circ f = \text{image } g \circ (\lambda-. \{\})$
by (*rule ext*) *simp*

lemma *Cinfinite-gt-empty*: $\text{Cinfinite } r \implies |\{\}| <_o r$
by (*simp add: cinfinite-def finite-ordLess-infinite card-of-ordIso-finite Field-card-of card-of-well-order-on emptyI card-order-on-well-order-on*)

lemma *Union-natural*: $\text{Union} \circ \text{image } (f) = \text{image } f \circ \text{Union}$
by (*rule ext*) (*auto simp only: comp-apply*)

lemma *in-Union-o-assoc*: $x \in (\text{Union} \circ \text{gset} \circ \text{gmap}) A \implies x \in (\text{Union} \circ (\text{gset} \circ \text{gmap})) A$
by (*unfold comp-assoc*)

lemma *regularCard-UNION-bound*:
assumes *Cinfinite* *r* *regularCard* *r* **and** $|I| <_o r \wedge i. i \in I \implies |A \ i| <_o r$
shows $|\bigcup_{i \in I}. A \ i| <_o r$
using *assms cinfinite-def regularCard-stable stable-UNION* **by** *blast*

lemma *comp-single-set-bd-strict*:
assumes *fbd*: *Cinfinite* *fbd* *regularCard* *fbd* **and**
gbd: *Cinfinite* *gbd* *regularCard* *gbd* **and**
fset-bd: $\bigwedge x. |fset \ x| <_o \text{fbd}$ **and**
gset-bd: $\bigwedge x. |gset \ x| <_o \text{gbd}$
shows $|\bigcup (fset \ ' \ gset \ x)| <_o \text{gbd} * c \ \text{fbd}$
proof (*cases fbd <_o gbd*)
case *True*
then have $|fset \ x| <_o \text{gbd}$ **for** *x* **using** *fset-bd ordLess-transitive* **by** *blast*
then have $|\bigcup (fset \ ' \ gset \ x)| <_o \text{gbd}$ **using** *regularCard-UNION-bound[OF gbd gset-bd]* **by** *blast*
then have $|\bigcup (fset \ ' \ gset \ x)| <_o \text{fbd} * c \ \text{gbd}$
using *ordLess-ordLeq-trans ordLeq-cprod2 gbd(1) fbd(1) cinfinite-not-czero* **by**
blast
then show *?thesis* **using** *ordLess-ordIso-trans cprod-com* **by** *blast*
next
case *False*
have *Well-order* *fbd* *Well-order* *gbd* **using** *fbd(1) gbd(1) card-order-on-well-order-on*
by *auto*
then have $\text{gbd} \leq_o \text{fbd}$ **using** *not-ordLess-iff-ordLeq* *False* **by** *blast*
then have $|gset \ x| <_o \text{fbd}$ **for** *x* **using** *gset-bd ordLess-ordLeq-trans* **by** *blast*
then have $|\bigcup (fset \ ' \ gset \ x)| <_o \text{fbd}$ **using** *regularCard-UNION-bound[OF fbd fset-bd]* **by** *blast*
then show *?thesis* **using** *ordLess-ordLeq-trans ordLeq-cprod2 gbd(1) fbd(1) cin-*

finite-not-czero **by** *blast*
qed

lemma *comp-single-set-bd*:
assumes *fbd-Card-order*: *Card-order fbd* **and**
fset-bd: $\bigwedge x. |fset\ x| \leq o\ fbd$ **and**
gset-bd: $\bigwedge x. |gset\ x| \leq o\ gbd$
shows $|\bigcup (fset\ 'gset\ x)| \leq o\ gbd *c\ fbd$
apply *simp*
apply (*rule ordLeq-transitive*)
apply (*rule card-of-UNION-Sigma*)
apply (*subst SIGMA-CSUM*)
apply (*rule ordLeq-transitive*)
apply (*rule card-of-Csum-Times^*)
apply (*rule fbd-Card-order*)
apply (*rule ballI*)
apply (*rule fset-bd*)
apply (*rule ordLeq-transitive*)
apply (*rule cprod-mono1*)
apply (*rule gset-bd*)
apply (*rule ordIso-imp-ordLeq*)
apply (*rule ordIso-refl*)
apply (*rule Card-order-cprod*)
done

lemma *csum-dup*: $cinfinite\ r \implies Card-order\ r \implies p +c\ p' =o\ r +c\ r \implies p +c\ p' =o\ r$
apply (*erule ordIso-transitive*)
apply (*erule csum-absorb2'*)
apply (*erule ordLeq-refl*)
by *simp*

lemma *cprod-dup*: $cinfinite\ r \implies Card-order\ r \implies p *c\ p' =o\ r *c\ r \implies p *c\ p' =o\ r$
apply (*erule ordIso-transitive*)
apply (*rule cprod-infinite*)
by *simp*

lemma *Union-image-insert*: $\bigcup (f\ 'insert\ a\ B) = f\ a \cup \bigcup (f\ 'B)$
by *simp*

lemma *Union-image-empty*: $A \cup \bigcup (f\ '\{\}) = A$
by *simp*

lemma *image-o-collect*: $collect\ ((\lambda f. image\ g \circ f)\ 'F) = image\ g \circ collect\ F$
by (*rule ext*) (*auto simp add: collect-def*)

lemma *conj-subset-def*: $A \subseteq \{x. P\ x \wedge Q\ x\} = (A \subseteq \{x. P\ x\} \wedge A \subseteq \{x. Q\ x\})$
by *blast*

lemma *UN-image-subset*: $\bigcup (f \text{ ‘ } g \ x) \subseteq X = (g \ x \subseteq \{x. f \ x \subseteq X\})$
by *blast*

lemma *comp-set-bd-Union-o-collect*: $|\bigcup (\bigcup ((\lambda f. f \ x) \text{ ‘ } X))| \leq_o \ hbd \implies |(Union \circ collect \ X) \ x| \leq_o \ hbd$
by (*unfold comp-apply collect-def*) *simp*

lemma *comp-set-bd-Union-o-collect-strict*: $|\bigcup (\bigcup ((\lambda f. f \ x) \text{ ‘ } X))| <_o \ hbd \implies |(Union \circ collect \ X) \ x| <_o \ hbd$
by (*unfold comp-apply collect-def*) *simp*

lemma *Collect-inj*: $Collect \ P = Collect \ Q \implies P = Q$
by *blast*

lemma *Grp-fst-snd*: $(Grp \ (Collect \ (case\text{-}prod \ R)) \ fst)^{-1-1} \ OO \ Grp \ (Collect \ (case\text{-}prod \ R)) \ snd = R$
unfolding *Grp-def fun-eq-iff relcompp.simps* **by** *auto*

lemma *OO-Grp-cong*: $A = B \implies (Grp \ A \ f)^{-1-1} \ OO \ Grp \ A \ g = (Grp \ B \ f)^{-1-1} \ OO \ Grp \ B \ g$
by (*rule arg-cong*)

lemma *vimage2p-relcompp-mono*: $R \ OO \ S \leq T \implies vimage2p \ f \ g \ R \ OO \ vimage2p \ g \ h \ S \leq vimage2p \ f \ h \ T$
unfolding *vimage2p-def* **by** *auto*

lemma *type-copy-map-cong0*: $M \ (g \ x) = N \ (h \ x) \implies (f \circ M \circ g) \ x = (f \circ N \circ h) \ x$
by *auto*

lemma *type-copy-set-bd*: $(\bigwedge y. |S \ y| <_o \ bd) \implies |(S \circ Rep) \ x| <_o \ bd$
by *auto*

lemma *vimage2p-cong*: $R = S \implies vimage2p \ f \ g \ R = vimage2p \ f \ g \ S$
by *simp*

lemma *Ball-comp-iff*: $(\lambda x. Ball \ (A \ x) \ f) \circ g = (\lambda x. Ball \ ((A \circ g) \ x) \ f)$
unfolding *o-def* **by** *auto*

lemma *conj-comp-iff*: $(\lambda x. P \ x \wedge Q \ x) \circ g = (\lambda x. (P \circ g) \ x \wedge (Q \circ g) \ x)$
unfolding *o-def* **by** *auto*

context

fixes *Rep Abs*

assumes *type-copy: type-definition Rep Abs UNIV*

begin

lemma *type-copy-map-id0*: $M = id \implies Abs \circ M \circ Rep = id$


```

using type-definition.Rep-inverse[OF type-copy] by auto

lemma type-copy-map-comp0:  $M = M1 \circ M2 \implies f \circ M \circ g = (f \circ M1 \circ Rep) \circ (Abs \circ M2 \circ g)$ 
using type-definition.Abs-inverse[OF type-copy UNIV-I] by auto

lemma type-copy-set-map0:  $S \circ M = image\ f \circ S' \implies (S \circ Rep) \circ (Abs \circ M \circ g) = image\ f \circ (S' \circ g)$ 
using type-definition.Abs-inverse[OF type-copy UNIV-I] by (auto simp: o-def fun-eq-iff)

lemma type-copy-wit:  $x \in (S \circ Rep)\ (Abs\ y) \implies x \in S\ y$ 
using type-definition.Abs-inverse[OF type-copy UNIV-I] by auto

lemma type-copy-vimage2p-Grp-Rep:  $vimage2p\ f\ Rep\ (Grp\ (Collect\ P)\ h) = Grp\ (Collect\ (\lambda x.\ P\ (f\ x)))\ (Abs \circ h \circ f)$ 
unfolding vimage2p-def Grp-def fun-eq-iff
by (auto simp: type-definition.Abs-inverse[OF type-copy UNIV-I] type-definition.Rep-inverse[OF type-copy] dest: sym)

lemma type-copy-vimage2p-Grp-Abs:
 $\bigwedge h.\ vimage2p\ g\ Abs\ (Grp\ (Collect\ P)\ h) = Grp\ (Collect\ (\lambda x.\ P\ (g\ x)))\ (Rep \circ h \circ g)$ 
unfolding vimage2p-def Grp-def fun-eq-iff
by (auto simp: type-definition.Abs-inverse[OF type-copy UNIV-I] type-definition.Rep-inverse[OF type-copy] dest: sym)

lemma type-copy-ex-RepI:  $(\exists b.\ F\ b) = (\exists b.\ F\ (Rep\ b))$ 
proof safe
  fix b assume F b
  show  $\exists b'. F\ (Rep\ b')$ 
  proof (rule exI)
    from  $\langle F\ b \rangle$  show  $F\ (Rep\ (Abs\ b))$  using type-definition.Abs-inverse[OF type-copy]
  by auto
  qed
qed blast

lemma vimage2p-relcompp-converse:
 $vimage2p\ f\ g\ (R^{-1-1}\ OO\ S) = (vimage2p\ Rep\ f\ R)^{-1-1}\ OO\ vimage2p\ Rep\ g\ S$ 
unfolding vimage2p-def relcompp.simps conversep.simps fun-eq-iff image-def
by (auto simp: type-copy-ex-RepI)

end

bnf DEADID: 'a
  map: id :: 'a  $\Rightarrow$  'a
  bd: natLeq
  rel: (=) :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  by (auto simp add: natLeq-card-order natLeq-cinfinite regularCard-natLeq)

```

definition *id-bnf* :: 'a ⇒ 'a **where**

id-bnf ≡ (λx. x)

lemma *id-bnf-apply*: *id-bnf* x = x

unfolding *id-bnf-def* **by** *simp*

bnf *ID*: 'a

map: *id-bnf* :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b

sets: λx. {x}

bd: *natLeq*

rel: *id-bnf* :: ('a ⇒ 'b ⇒ bool) ⇒ 'a ⇒ 'b ⇒ bool

pred: *id-bnf* :: ('a ⇒ bool) ⇒ 'a ⇒ bool

unfolding *id-bnf-def*

apply (*auto simp*: *Grp-def fun-eq-iff relcompp.simps natLeq-card-order natLeq-cinfinite regularCard-natLeq*)

apply (*rule finite-ordLess-infinite[OF - natLeq-Well-order]*)

apply (*auto simp add: Field-card-of Field-natLeq card-of-well-order-on*)[3]

done

lemma *type-definition-id-bnf-UNIV*: *type-definition id-bnf id-bnf UNIV*

unfolding *id-bnf-def* **by** *unfold-locales auto*

ML-file <Tools/BNF/bnf-comp-tactics.ML>

ML-file <Tools/BNF/bnf-comp.ML>

hide-fact

DEADID.inj-map DEADID.inj-map-strong DEADID.map-comp DEADID.map-cong

DEADID.map-cong0

DEADID.map-cong-simp DEADID.map-id DEADID.map-id0 DEADID.map-ident

DEADID.map-transfer

DEADID.rel-Grp DEADID.rel-compp DEADID.rel-compp-Grp DEADID.rel-conversep

DEADID.rel-eq

DEADID.rel-flip DEADID.rel-map DEADID.rel-mono DEADID.rel-transfer

ID.inj-map ID.inj-map-strong ID.map-comp ID.map-cong ID.map-cong0 ID.map-cong-simp

ID.map-id

ID.map-id0 ID.map-ident ID.map-transfer ID.rel-Grp ID.rel-compp ID.rel-compp-Grp

ID.rel-conversep

ID.rel-eq ID.rel-flip ID.rel-map ID.rel-mono ID.rel-transfer ID.set-map ID.set-transfer

end

35 Registration of Basic Types as Bounded Natural Functors

theory *Basic-BNFs*

imports *BNF-Def*

begin

inductive-set *setl* :: 'a + 'b \Rightarrow 'a set for *s* :: 'a + 'b where
s = *Inl* *x* \Longrightarrow *x* \in *setl* *s*
inductive-set *setr* :: 'a + 'b \Rightarrow 'b set for *s* :: 'a + 'b where
s = *Inr* *x* \Longrightarrow *x* \in *setr* *s*

lemma *sum-set-defs*[code]:
setl = ($\lambda x.$ case *x* of *Inl* *z* \Rightarrow {*z*} | - \Rightarrow {})
setr = ($\lambda x.$ case *x* of *Inr* *z* \Rightarrow {*z*} | - \Rightarrow {})
by (*auto simp: fun-eq-iff intro: setl.intros setr.intros elim: setl.cases setr.cases split: sum.splits*)

lemma *rel-sum-simps*[code, simp]:
rel-sum *R1* *R2* (*Inl* *a1*) (*Inl* *b1*) = *R1* *a1* *b1*
rel-sum *R1* *R2* (*Inl* *a1*) (*Inr* *b2*) = *False*
rel-sum *R1* *R2* (*Inr* *a2*) (*Inl* *b1*) = *False*
rel-sum *R1* *R2* (*Inr* *a2*) (*Inr* *b2*) = *R2* *a2* *b2*
by (*auto intro: rel-sum.intros elim: rel-sum.cases*)

inductive
pred-sum :: ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a + 'b \Rightarrow bool for *P1* *P2*
where
P1 *a* \Longrightarrow *pred-sum* *P1* *P2* (*Inl* *a*)
| *P2* *b* \Longrightarrow *pred-sum* *P1* *P2* (*Inr* *b*)

lemma *pred-sum-inject*[code, simp]:
pred-sum *P1* *P2* (*Inl* *a*) \longleftrightarrow *P1* *a*
pred-sum *P1* *P2* (*Inr* *b*) \longleftrightarrow *P2* *b*
by (*simp add: pred-sum.simps*)+

bnf 'a + 'b
map: *map-sum*
sets: *setl* *setr*
bd: *natLeq*
wits: *Inl* *Inr*
rel: *rel-sum*
pred: *pred-sum*

proof –
show *map-sum* *id* *id* = *id* **by** (*rule map-sum.id*)
next
fix *f1* :: 'o \Rightarrow 's and *f2* :: 'p \Rightarrow 't and *g1* :: 's \Rightarrow 'q and *g2* :: 't \Rightarrow 'r
show *map-sum* (*g1* \circ *f1*) (*g2* \circ *f2*) = *map-sum* *g1* *g2* \circ *map-sum* *f1* *f2*
by (*rule map-sum.comp[symmetric]*)
next
fix *x* and *f1* :: 'o \Rightarrow 'q and *f2* :: 'p \Rightarrow 'r and *g1* *g2*
assume *a1*: $\bigwedge z. z \in \text{setl } x \Longrightarrow f1\ z = g1\ z$ and
a2: $\bigwedge z. z \in \text{setr } x \Longrightarrow f2\ z = g2\ z$
thus *map-sum* *f1* *f2* *x* = *map-sum* *g1* *g2* *x*
proof (*cases* *x*)

```

    case Inl thus ?thesis using a1 by (clarsimp simp: sum-set-defs(1))
  next
    case Inr thus ?thesis using a2 by (clarsimp simp: sum-set-defs(2))
  qed
next
fix f1 :: 'o ⇒ 'q and f2 :: 'p ⇒ 'r
show setl ∘ map-sum f1 f2 = image f1 ∘ setl
  by (rule ext, unfold o-apply) (simp add: sum-set-defs(1) split: sum.split)
next
fix f1 :: 'o ⇒ 'q and f2 :: 'p ⇒ 'r
show setr ∘ map-sum f1 f2 = image f2 ∘ setr
  by (rule ext, unfold o-apply) (simp add: sum-set-defs(2) split: sum.split)
next
show card-order natLeq by (rule natLeq-card-order)
next
show cinfinitive natLeq by (rule natLeq-cinfinitive)
next
show regularCard natLeq by (rule regularCard-natLeq)
next
fix x :: 'o + 'p
show |setl x| <o natLeq
  apply (rule finite-iff-ordLess-natLeq[THEN iffD1])
  by (simp add: sum-set-defs(1) split: sum.split)
next
fix x :: 'o + 'p
show |setr x| <o natLeq
  apply (rule finite-iff-ordLess-natLeq[THEN iffD1])
  by (simp add: sum-set-defs(2) split: sum.split)
next
fix R1 R2 S1 S2
show rel-sum R1 R2 OO rel-sum S1 S2 ≤ rel-sum (R1 OO S1) (R2 OO S2)
  by (force elim: rel-sum.cases)
next
fix R S
show rel-sum R S = (λx y.
  ∃ z. (setl z ⊆ {(x, y). R x y} ∧ setr z ⊆ {(x, y). S x y}) ∧
  map-sum fst fst z = x ∧ map-sum snd snd z = y)
  unfolding sum-set-defs relcompp.simps conversep.simps fun-eq-iff
  by (fastforce elim: rel-sum.cases split: sum.splits)
qed (auto simp: sum-set-defs fun-eq-iff pred-sum.simps split: sum.splits)

inductive-set fsts :: 'a × 'b ⇒ 'a set for p :: 'a × 'b where
  fst p ∈ fsts p
inductive-set snds :: 'a × 'b ⇒ 'b set for p :: 'a × 'b where
  snd p ∈ snds p

lemma prod-set-defs[code]: fsts = (λp. {fst p}) snds = (λp. {snd p})
  by (auto intros: fsts.intros snds.intros elim: fsts.cases snds.cases)

```

inductive

$rel\text{-}prod :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('c \Rightarrow 'd \Rightarrow bool) \Rightarrow 'a \times 'c \Rightarrow 'b \times 'd \Rightarrow bool$
for $R1\ R2$

where

$\llbracket R1\ a\ b; R2\ c\ d \rrbracket \Longrightarrow rel\text{-}prod\ R1\ R2\ (a, c)\ (b, d)$

inductive

$pred\text{-}prod :: ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \times 'b \Rightarrow bool$ **for** $P1\ P2$

where

$\llbracket P1\ a; P2\ b \rrbracket \Longrightarrow pred\text{-}prod\ P1\ P2\ (a, b)$

lemma $rel\text{-}prod\text{-}inject$ [*code, simp*]:

$rel\text{-}prod\ R1\ R2\ (a, b)\ (c, d) \longleftrightarrow R1\ a\ c \wedge R2\ b\ d$
by (*auto intro: rel-prod.intros elim: rel-prod.cases*)

lemma $pred\text{-}prod\text{-}inject$ [*code, simp*]:

$pred\text{-}prod\ P1\ P2\ (a, b) \longleftrightarrow P1\ a \wedge P2\ b$
by (*auto intro: pred-prod.intros elim: pred-prod.cases*)

lemma $rel\text{-}prod\text{-}conv$:

$rel\text{-}prod\ R1\ R2 = (\lambda(a, b)\ (c, d). R1\ a\ c \wedge R2\ b\ d)$
by *force*

definition

$pred\text{-}fun :: ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where

$pred\text{-}fun\ A\ B = (\lambda f. \forall x. A\ x \longrightarrow B\ (f\ x))$

lemma $pred\text{-}funI$: $(\bigwedge x. A\ x \Longrightarrow B\ (f\ x)) \Longrightarrow pred\text{-}fun\ A\ B\ f$

unfolding $pred\text{-}fun\text{-}def$ **by** *simp*

bnf $'a \times 'b$

map: map-prod

sets: fst snd

bd: natLeq

rel: rel-prod

pred: pred-prod

proof (*unfold prod-set-defs*)

show $map\text{-}prod\ id\ id = id$ **by** (*rule map-prod.id*)

next

fix $f1\ f2\ g1\ g2$

show $map\text{-}prod\ (g1 \circ f1)\ (g2 \circ f2) = map\text{-}prod\ g1\ g2 \circ map\text{-}prod\ f1\ f2$

by (*rule map-prod.comp[symmetric]*)

next

fix $x\ f1\ f2\ g1\ g2$

assume $\bigwedge z. z \in \{fst\ x\} \Longrightarrow f1\ z = g1\ z \wedge z. z \in \{snd\ x\} \Longrightarrow f2\ z = g2\ z$

thus $map\text{-}prod\ f1\ f2\ x = map\text{-}prod\ g1\ g2\ x$ **by** (*cases x simp*)

next

fix $f1\ f2$

```

  show  $(\lambda x. \{fst\ x\}) \circ map\text{-}prod\ f1\ f2 = image\ f1 \circ (\lambda x. \{fst\ x\})$ 
    by (rule ext, unfold o-apply) simp
next
  fix f1 f2
  show  $(\lambda x. \{snd\ x\}) \circ map\text{-}prod\ f1\ f2 = image\ f2 \circ (\lambda x. \{snd\ x\})$ 
    by (rule ext, unfold o-apply) simp
next
  show card-order natLeq by (rule natLeq-card-order)
next
  show cinfinitesimal natLeq by (rule natLeq-cinfinitesimal)
next
  show regularCard natLeq by (rule regularCard-natLeq)
next
  fix x
  show  $|\{fst\ x\}| <_o\ natLeq$ 
    by (simp add: finite-iff-ordLess-natLeq[symmetric])
next
  fix x
  show  $|\{snd\ x\}| <_o\ natLeq$ 
    by (simp add: finite-iff-ordLess-natLeq[symmetric])
next
  fix R1 R2 S1 S2
  show  $rel\text{-}prod\ R1\ R2\ OO\ rel\text{-}prod\ S1\ S2 \leq rel\text{-}prod\ (R1\ OO\ S1)\ (R2\ OO\ S2)$ 
  by auto
next
  fix R S
  show  $rel\text{-}prod\ R\ S = (\lambda x\ y. \exists z. (\{fst\ z\} \subseteq \{(x, y). R\ x\ y\} \wedge \{snd\ z\} \subseteq \{(x, y). S\ x\ y\}) \wedge map\text{-}prod\ fst\ fst\ z = x \wedge map\text{-}prod\ snd\ snd\ z = y)$ 
  unfolding prod-set-defs rel-prod-inject relcompp.simps converseSep.simps fun-eq-iff
  by auto
qed auto

lemma card-order-bd-fun: card-order (natLeq +c card-suc (|UNIV|))
  by (auto simp: card-order-csum natLeq-card-order card-order-card-suc card-of-card-order-on)

lemma Cinfinitesimal-bd-fun: Cinfinitesimal (natLeq +c card-suc (|UNIV|))
  by (auto simp: Cinfinitesimal-csum natLeq-Cinfinitesimal)

lemma regularCard-bd-fun: regularCard (natLeq +c card-suc (|UNIV|))
  (is regularCard (- +c card-suc ?U))
proof (cases Cinfinitesimal ?U)
  case True
  then show ?thesis
    by (intro regularCard-csum natLeq-Cinfinitesimal Cinfinitesimal-card-suc
      card-of-card-order-on regularCard-natLeq regularCard-card-suc)
next
  case False
  then have  $card\text{-}suc\ ?U \leq_o\ natLeq$ 

```

```

unfolding cinfinite-def Field-card-of
by (intro card-suc-least;
      simp add: natLeq-Card-order card-of-card-order-on flip: finite-iff-ordLess-natLeq)
then have natLeq =o natLeq +c card-suc ?U
using natLeq-Cinfinite csum-absorb1 ordIso-symmetric by blast
then show ?thesis
by (intro regularCard-ordIso[OF - natLeq-Cinfinite regularCard-natLeq])
qed

lemma ordLess-bd-fun: |UNIV::'a set| <o natLeq +c card-suc ( |UNIV::'a set| )
(is - <o (- +c card-suc (?U :: 'a rel)))
proof (cases Cinfinite ?U)
case True
have ?U <o card-suc ?U using card-of-card-order-on natLeq-card-order card-suc-greater
by blast
also have card-suc ?U =o natLeq +c card-suc ?U by (rule csum-absorb2[THEN
ordIso-symmetric])
(auto simp: True card-of-card-order-on intro!: Cinfinite-card-suc natLeq-ordLeq-cinfinite)
finally show ?thesis .
next
case False
then have ?U <o natLeq
by (auto simp: cinfinite-def Field-card-of card-of-card-order-on finite-iff-ordLess-natLeq[symmetric])
then show ?thesis
by (rule ordLess-ordLeq-trans[OF - ordLeq-csum1[OF natLeq-Card-order]])
qed

bnf 'a ⇒ 'b
map: (o)
sets: range
bd: natLeq +c card-suc ( |UNIV::'a set| )
rel: rel-fun (=)
pred: pred-fun (λ-. True)
proof
fix f show id o f = id f by simp
next
fix f g show (o) (g o f) = (o) g o (o) f
unfolding comp-def[abs-def] ..
next
fix x f g
assume  $\bigwedge z. z \in \text{range } x \implies f z = g z$ 
thus f o x = g o x by auto
next
fix f show range o (o) f = (o) f o range
by (auto simp add: fun-eq-iff)
next
show card-order (natLeq +c card-suc ( |UNIV| ))
by (rule card-order-bd-fun)
next

```

```

show cinfinite (natLeq +c card-suc ( |UNIV| ))
  by (rule Cinfinite-bd-fun[THEN conjunct1])
next
show regularCard (natLeq +c card-suc ( |UNIV| ))
  by (rule regularCard-bd-fun)
next
fix f :: 'd  $\Rightarrow$  'a
show |range f| <o natLeq +c card-suc |UNIV :: 'd set|
  by (rule ordLeq-ordLess-trans[OF card-of-image ordLess-bd-fun])
next
fix R S
show rel-fun (=) R OO rel-fun (=) S  $\leq$  rel-fun (=) (R OO S) by (auto simp:
rel-fun-def)
next
fix R
show rel-fun (=) R = ( $\lambda x y.$ 
   $\exists z. \text{range } z \subseteq \{(x, y). R \ x \ y\} \wedge \text{fst } \circ z = x \wedge \text{snd } \circ z = y$ )
  unfolding rel-fun-def subset-iff by (force simp: fun-eq-iff[symmetric])
qed (auto simp: pred-fun-def)

end

```

36 Shared Fixpoint Operations on Bounded Natural Functors

```

theory BNF-Fixpoint-Base
imports BNF-Composition Basic-BNFs
begin

```

```

lemma conj-imp-eq-imp-imp: ( $P \wedge Q \Longrightarrow PROP \ R$ )  $\equiv$  ( $P \Longrightarrow Q \Longrightarrow PROP \ R$ )
  by standard simp-all

```

```

lemma predicate2D-conj:  $P \leq Q \wedge R \Longrightarrow R \wedge (P \ x \ y \longrightarrow Q \ x \ y)$ 
  by blast

```

```

lemma eq-sym-Unity-conv: ( $x = ((\ ) = (\ ))$ ) = x
  by blast

```

```

lemma case-unit-Unity: (case u of ( $\ ) \Rightarrow f$ ) = f
  by (cases u) (hypsubst, rule unit.case)

```

```

lemma case-prod-Pair-iden: (case p of ( $(x, y) \Rightarrow (x, y)$ )) = p
  by simp

```

```

lemma unit-all-impI: ( $P (\ ) \Longrightarrow Q (\ )$ )  $\Longrightarrow \forall x. P \ x \longrightarrow Q \ x$ 
  by simp

```

```

lemma pointfree-idE:  $f \circ g = id \Longrightarrow f (g \ x) = x$ 

```


unfolding *comp-def fun-eq-iff* **by** *simp*

lemma *o-bij*:

assumes *gf*: $g \circ f = id$ **and** *fg*: $f \circ g = id$

shows *bij* *f*

unfolding *bij-def inj-on-def surj-def* **proof** *safe*

fix *a1 a2* **assume** $f a1 = f a2$

hence $g (f a1) = g (f a2)$ **by** *simp*

thus $a1 = a2$ **using** *gf* **unfolding** *fun-eq-iff* **by** *simp*

next

fix *b*

have $b = f (g b)$

using *fg* **unfolding** *fun-eq-iff* **by** *simp*

thus $\exists a. b = f a$ **by** *blast*

qed

lemma *case-sum-step*:

case-sum (*case-sum* *f' g'*) *g* (*Inl* *p*) = *case-sum* *f' g' p*

case-sum *f* (*case-sum* *f' g'*) (*Inr* *p*) = *case-sum* *f' g' p*

by *auto*

lemma *obj-one-pointE*: $\forall x. s = x \longrightarrow P \Longrightarrow P$

by *blast*

lemma *type-copy-obj-one-point-absE*:

assumes *type-definition* *Rep Abs UNIV* $\forall x. s = Abs x \longrightarrow P$ **shows** *P*

using *type-definition.Rep-inverse*[*OF* *assms*(1)]

by (*intro* *mp*[*OF* *spec*[*OF* *assms*(2), *of* *Rep* *s*]]) *simp*

lemma *obj-sumE-f*:

assumes $\forall x. s = f (Inl x) \longrightarrow P \quad \forall x. s = f (Inr x) \longrightarrow P$

shows $\forall x. s = f x \longrightarrow P$

proof

fix *x* **from** *assms* **show** $s = f x \longrightarrow P$ **by** (*cases* *x*) *auto*

qed

lemma *case-sum-if*:

case-sum *f g* (*if* *p* *then* *Inl* *x* *else* *Inr* *y*) = (*if* *p* *then* *f* *x* *else* *g* *y*)

by *simp*

lemma *prod-set-simps*[*simp*]:

fsts (*x*, *y*) = {*x*}

snds (*x*, *y*) = {*y*}

unfolding *prod-set-defs* **by** *simp+*

lemma *sum-set-simps*[*simp*]:

setl (*Inl* *x*) = {*x*}

setl (*Inr* *x*) = {}

setr (*Inl* *x*) = {}

$setr (Inr x) = \{x\}$
unfolding *sum-set-defs* **by** *simp+*

lemma *Inl-Inr-False*: $(Inl x = Inr y) = False$
by *simp*

lemma *Inr-Inl-False*: $(Inr x = Inl y) = False$
by *simp*

lemma *spec2*: $\forall x y. P x y \implies P x y$
by *blast*

lemma *rewriteR-comp-comp*: $\llbracket g \circ h = r \rrbracket \implies f \circ g \circ h = f \circ r$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *rewriteR-comp-comp2*: $\llbracket g \circ h = r1 \circ r2; f \circ r1 = l \rrbracket \implies f \circ g \circ h = l \circ r2$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *rewriteL-comp-comp*: $\llbracket f \circ g = l \rrbracket \implies f \circ (g \circ h) = l \circ h$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *rewriteL-comp-comp2*: $\llbracket f \circ g = l1 \circ l2; l2 \circ h = r \rrbracket \implies f \circ (g \circ h) = l1 \circ r$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *convol-o*: $\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$
unfolding *convol-def* **by** *auto*

lemma *map-prod-o-convol*: $map\text{-}prod\ h1\ h2 \circ \langle f, g \rangle = \langle h1 \circ f, h2 \circ g \rangle$
unfolding *convol-def* **by** *auto*

lemma *map-prod-o-convol-id*: $(map\text{-}prod\ f\ id \circ \langle id, g \rangle) x = \langle id \circ f, g \rangle x$
unfolding *map-prod-o-convol id-comp comp-id ..*

lemma *o-case-sum*: $h \circ case\text{-}sum\ f\ g = case\text{-}sum\ (h \circ f)\ (h \circ g)$
unfolding *comp-def* **by** *(auto split: sum.splits)*

lemma *case-sum-o-map-sum*: $case\text{-}sum\ f\ g \circ map\text{-}sum\ h1\ h2 = case\text{-}sum\ (f \circ h1)\ (g \circ h2)$
unfolding *comp-def* **by** *(auto split: sum.splits)*

lemma *case-sum-o-map-sum-id*: $(case\text{-}sum\ id\ g \circ map\text{-}sum\ f\ id) x = case\text{-}sum\ (f \circ id)\ g\ x$
unfolding *case-sum-o-map-sum id-comp comp-id ..*

lemma *rel-fun-def-butlast*:
 $rel\text{-}fun\ R\ (rel\text{-}fun\ S\ T)\ f\ g = (\forall x y. R\ x\ y \longrightarrow (rel\text{-}fun\ S\ T)\ (f\ x)\ (g\ y))$
unfolding *rel-fun-def ..*

lemma *subst-eq-imp*: $(\forall a b. a = b \longrightarrow P a b) \equiv (\forall a. P a a)$
by *auto*

lemma *eq-subset*: $(=) \leq (\lambda a b. P a b \vee a = b)$
by *auto*

lemma *eq-le-Grp-id-iff*: $((=) \leq \text{Grp } (\text{Collect } R) \text{ id}) = (\text{All } R)$
unfolding *Grp-def id-apply* **by** *blast*

lemma *Grp-id-mono-subst*: $(\bigwedge x y. \text{Grp } P \text{ id } x y \Longrightarrow \text{Grp } Q \text{ id } (f x) (f y)) \equiv$
 $(\bigwedge x. x \in P \Longrightarrow f x \in Q)$
unfolding *Grp-def* **by** *rule auto*

lemma *vimage2p-mono*: $\text{vimage2p } f g R x y \Longrightarrow R \leq S \Longrightarrow \text{vimage2p } f g S x y$
unfolding *vimage2p-def* **by** *blast*

lemma *vimage2p-refl*: $(\bigwedge x. R x x) \Longrightarrow \text{vimage2p } f f R x x$
unfolding *vimage2p-def* **by** *auto*

lemma

assumes *type-definition Rep Abs UNIV*

shows *type-copy-Rep-o-Abs*: $\text{Rep} \circ \text{Abs} = \text{id}$ **and** *type-copy-Abs-o-Rep*: $\text{Abs} \circ \text{Rep} = \text{id}$

unfolding *fun-eq-iff comp-apply id-apply*

type-definition.Abs-inverse[OF assms UNIV-I] *type-definition.Rep-inverse[OF assms]* **by** *simp-all*

lemma *type-copy-map-comp0-undo*:

assumes *type-definition Rep Abs UNIV*

type-definition Rep' Abs' UNIV

type-definition Rep'' Abs'' UNIV

shows $\text{Abs}' \circ M \circ \text{Rep}'' = (\text{Abs}' \circ M1 \circ \text{Rep}) \circ (\text{Abs} \circ M2 \circ \text{Rep}'') \Longrightarrow M1 \circ M2 = M$

by (*rule sym*) (*auto simp: fun-eq-iff type-definition.Abs-inject[OF assms(2) UNIV-I UNIV-I]*)

type-definition.Abs-inverse[OF assms(1) UNIV-I]

type-definition.Abs-inverse[OF assms(3) UNIV-I] *dest: spec[of - Abs'' x for x]*)

lemma *vimage2p-id*: $\text{vimage2p } \text{id } \text{id } R = R$

unfolding *vimage2p-def* **by** *auto*

lemma *vimage2p-comp*: $\text{vimage2p } (f1 \circ f2) (g1 \circ g2) = \text{vimage2p } f2 g2 \circ \text{vimage2p } f1 g1$

unfolding *fun-eq-iff vimage2p-def o-apply* **by** *simp*

lemma *vimage2p-rel-fun*: $\text{rel-fun } (\text{vimage2p } f g R) R f g$

unfolding *rel-fun-def vimage2p-def* **by** *auto*

lemma *fun-cong-unused-0*: $f = (\lambda x. g) \implies f (\lambda x. 0) = g$
by (*erule arg-cong*)

lemma *inj-on-convol-ident*: $\text{inj-on } (\lambda x. (x, f x)) X$
unfolding *inj-on-def* **by** *simp*

lemma *map-sum-if-distrib-then*:

$\bigwedge f g e x y. \text{map-sum } f g (\text{if } e \text{ then } \text{Inl } x \text{ else } y) = (\text{if } e \text{ then } \text{Inl } (f x) \text{ else } \text{map-sum } f g y)$

$\bigwedge f g e x y. \text{map-sum } f g (\text{if } e \text{ then } \text{Inr } x \text{ else } y) = (\text{if } e \text{ then } \text{Inr } (g x) \text{ else } \text{map-sum } f g y)$

by *simp-all*

lemma *map-sum-if-distrib-else*:

$\bigwedge f g e x y. \text{map-sum } f g (\text{if } e \text{ then } x \text{ else } \text{Inl } y) = (\text{if } e \text{ then } \text{map-sum } f g x \text{ else } \text{Inl } (f y))$

$\bigwedge f g e x y. \text{map-sum } f g (\text{if } e \text{ then } x \text{ else } \text{Inr } y) = (\text{if } e \text{ then } \text{map-sum } f g x \text{ else } \text{Inr } (g y))$

by *simp-all*

lemma *case-prod-app*: $\text{case-prod } f x y = \text{case-prod } (\lambda l r. f l r y) x$
by (*cases x*) *simp*

lemma *case-sum-map-sum*: $\text{case-sum } l r (\text{map-sum } f g x) = \text{case-sum } (l \circ f) (r \circ g) x$
by (*cases x*) *simp-all*

lemma *case-sum-transfer*:

$\text{rel-fun } (\text{rel-fun } R T) (\text{rel-fun } (\text{rel-fun } S T) (\text{rel-fun } (\text{rel-sum } R S) T)) \text{ case-sum case-sum}$

unfolding *rel-fun-def* **by** (*auto split: sum.splits*)

lemma *case-prod-map-prod*: $\text{case-prod } h (\text{map-prod } f g x) = \text{case-prod } (\lambda l r. h (f l) (g r)) x$

by (*cases x*) *simp-all*

lemma *case-prod-o-map-prod*: $\text{case-prod } f \circ \text{map-prod } g1 g2 = \text{case-prod } (\lambda l r. f (g1 l) (g2 r))$

unfolding *comp-def* **by** *auto*

lemma *case-prod-transfer*:

$(\text{rel-fun } (\text{rel-fun } A (\text{rel-fun } B C)) (\text{rel-fun } (\text{rel-prod } A B) C)) \text{ case-prod case-prod}$

unfolding *rel-fun-def* **by** *simp*

lemma *eq-iff*: $(P \longrightarrow t = u1) \implies (\neg P \longrightarrow t = u2) \implies t = (\text{if } P \text{ then } u1 \text{ else } u2)$

by *simp*

lemma *comp-transfer*:

rel-fun (*rel-fun* *B C*) (*rel-fun* (*rel-fun* *A B*) (*rel-fun* *A C*)) (\circ) (\circ)
unfolding *rel-fun-def* **by** *simp*

lemma *If-transfer*: *rel-fun* (=) (*rel-fun* *A* (*rel-fun* *A A*)) *If If*
unfolding *rel-fun-def* **by** *simp*

lemma *Abs-transfer*:

assumes *type-copy1*: *type-definition* *Rep1 Abs1 UNIV*
assumes *type-copy2*: *type-definition* *Rep2 Abs2 UNIV*
shows *rel-fun* *R* (*vimage2p* *Rep1 Rep2 R*) *Abs1 Abs2*
unfolding *vimage2p-def rel-fun-def*
type-definition.Abs-inverse[*OF type-copy1 UNIV-I*]
type-definition.Abs-inverse[*OF type-copy2 UNIV-I*] **by** *simp*

lemma *Inl-transfer*:

rel-fun *S* (*rel-sum* *S T*) *Inl Inl*
by *auto*

lemma *Inr-transfer*:

rel-fun *T* (*rel-sum* *S T*) *Inr Inr*
by *auto*

lemma *Pair-transfer*: *rel-fun* *A* (*rel-fun* *B* (*rel-prod* *A B*)) *Pair Pair*
unfolding *rel-fun-def* **by** *simp*

lemma *eq-onp-live-step*: $x = y \implies \text{eq-onp } P \ a \ a \wedge x \longleftrightarrow P \ a \wedge y$
by (*simp only: eq-onp-same-args*)

lemma *top-conj*: $\text{top } x \wedge P \longleftrightarrow P \ P \wedge \text{top } x \longleftrightarrow P$
by *blast+*

lemma *fst-convol'*: $\text{fst } (\langle f, g \rangle x) = f \ x$
using *fst-convol* **unfolding** *convol-def* **by** *simp*

lemma *snd-convol'*: $\text{snd } (\langle f, g \rangle x) = g \ x$
using *snd-convol* **unfolding** *convol-def* **by** *simp*

lemma *convol-expand-snd*: $\text{fst } \circ f = g \implies \langle g, \text{snd } \circ f \rangle = f$
unfolding *convol-def* **by** *auto*

lemma *convol-expand-snd'*:

assumes (*fst* \circ *f* = *g*)
shows $h = \text{snd } \circ f \longleftrightarrow \langle g, h \rangle = f$

proof –

from *assms* **have** \ast : $\langle g, \text{snd } \circ f \rangle = f$ **by** (*rule convol-expand-snd*)

then **have** $h = \text{snd } \circ f \longleftrightarrow h = \text{snd } \circ \langle g, \text{snd } \circ f \rangle$ **by** *simp*

moreover **have** $\dots \longleftrightarrow h = \text{snd } \circ f$ **by** (*simp add: snd-convol*)

moreover **have** $\dots \longleftrightarrow \langle g, h \rangle = f$ **by** (*subst* (2) \ast [*symmetric*]) (*auto simp: convol-def fun-eq-iff*))

ultimately show *?thesis* **by** *simp*
qed

lemma *case-sum-expand-Inr-pointfree*: $f \circ \text{Inl} = g \implies \text{case-sum } g (f \circ \text{Inr}) = f$
by (*auto split: sum.splits*)

lemma *case-sum-expand-Inr'*: $f \circ \text{Inl} = g \implies h = f \circ \text{Inr} \longleftrightarrow \text{case-sum } g h = f$
by (*rule iffI*) (*auto simp add: fun-eq-iff split: sum.splits*)

lemma *case-sum-expand-Inr*: $f \circ \text{Inl} = g \implies f x = \text{case-sum } g (f \circ \text{Inr}) x$
by (*auto split: sum.splits*)

lemma *id-transfer*: *rel-fun* $A A$ *id id*
unfolding *rel-fun-def* **by** *simp*

lemma *fst-transfer*: *rel-fun* (*rel-prod* $A B$) A *fst fst*
unfolding *rel-fun-def* **by** *simp*

lemma *snd-transfer*: *rel-fun* (*rel-prod* $A B$) B *snd snd*
unfolding *rel-fun-def* **by** *simp*

lemma *convol-transfer*:
rel-fun (*rel-fun* $R S$) (*rel-fun* (*rel-fun* $R T$) (*rel-fun* R (*rel-prod* $S T$))) *BNF-Def.convol*
BNF-Def.convol
unfolding *rel-fun-def convol-def* **by** *auto*

lemma *Let-const*: *Let* $x (\lambda-. c) = c$
unfolding *Let-def* **..**

ML-file $\langle \text{Tools/BNF/bnf-fp-util-tactics.ML} \rangle$
ML-file $\langle \text{Tools/BNF/bnf-fp-util.ML} \rangle$
ML-file $\langle \text{Tools/BNF/bnf-fp-def-sugar-tactics.ML} \rangle$
ML-file $\langle \text{Tools/BNF/bnf-fp-def-sugar.ML} \rangle$
ML-file $\langle \text{Tools/BNF/bnf-fp-n2m-tactics.ML} \rangle$
ML-file $\langle \text{Tools/BNF/bnf-fp-n2m.ML} \rangle$
ML-file $\langle \text{Tools/BNF/bnf-fp-n2m-sugar.ML} \rangle$

end

37 Least Fixpoint (Datatype) Operation on Bounded Natural Functors

theory *BNF-Least-Fixpoint*
imports *BNF-Fixpoint-Base*
keywords
datatype **::** *thy-defn* **and**
datatype-compat **::** *thy-defn*
begin

lemma *subset-emptyI*: $(\bigwedge x. x \in A \implies \text{False}) \implies A \subseteq \{\}$
by *blast*

lemma *image-Collect-subsetI*: $(\bigwedge x. P x \implies f x \in B) \implies f ` \{x. P x\} \subseteq B$
by *blast*

lemma *Collect-restrict*: $\{x. x \in X \wedge P x\} \subseteq X$
by *auto*

lemma *prop-restrict*: $\llbracket x \in Z; Z \subseteq \{x. x \in X \wedge P x\} \rrbracket \implies P x$
by *auto*

lemma *underS-I*: $\llbracket i \neq j; (i, j) \in R \rrbracket \implies i \in \text{underS } R j$
unfolding *underS-def* **by** *simp*

lemma *underS-E*: $i \in \text{underS } R j \implies i \neq j \wedge (i, j) \in R$
unfolding *underS-def* **by** *simp*

lemma *underS-Field*: $i \in \text{underS } R j \implies i \in \text{Field } R$
unfolding *underS-def* *Field-def* **by** *auto*

lemma *ex-bij-betw*: $|A| \leq o (r :: 'b \text{ rel}) \implies \exists f B :: 'b \text{ set. } \text{bij-betw } f B A$
by (*subst (asm) internalize-card-of-ordLeq*) (*auto dest!: iffD2[OF card-of-ordIso ordIso-symmetric]*)

lemma *bij-betwI'*:
 $\llbracket \bigwedge x y. \llbracket x \in X; y \in X \rrbracket \implies (f x = f y) = (x = y);$
 $\bigwedge x. x \in X \implies f x \in Y;$
 $\bigwedge y. y \in Y \implies \exists x \in X. y = f x \rrbracket \implies \text{bij-betw } f X Y$
unfolding *bij-betw-def inj-on-def* **by** *blast*

lemma *surj-fun-eq*:
assumes *surj-on*: $f ` X = \text{UNIV}$ **and** *eq-on*: $\forall x \in X. (g1 \circ f) x = (g2 \circ f) x$
shows $g1 = g2$
proof (*rule ext*)
fix y
from *surj-on* **obtain** x **where** $x \in X$ **and** $y = f x$ **by** *blast*
thus $g1 y = g2 y$ **using** *eq-on* **by** *simp*
qed

lemma *Card-order-wo-rel*: $\text{Card-order } r \implies \text{wo-rel } r$
unfolding *wo-rel-def card-order-on-def* **by** *blast*

lemma *Cinfinite-limit*: $\llbracket x \in \text{Field } r; \text{Cinfinite } r \rrbracket \implies \exists y \in \text{Field } r. x \neq y \wedge (x, y) \in r$
unfolding *cinfinite-def* **by** (*auto simp add: infinite-Card-order-limit*)

lemma *Card-order-trans*:

[[*Card-order* r ; $x \neq y$; $(x, y) \in r$; $y \neq z$; $(y, z) \in r$] $\implies x \neq z \wedge (x, z) \in r$
unfolding *card-order-on-def well-order-on-def linear-order-on-def*
partial-order-on-def preorder-on-def trans-def antisym-def **by** *blast*

lemma *Cinfinite-limit2*:

assumes $x1: x1 \in \text{Field } r$ **and** $x2: x2 \in \text{Field } r$ **and** $r: \text{Cinfinite } r$
shows $\exists y \in \text{Field } r. (x1 \neq y \wedge (x1, y) \in r) \wedge (x2 \neq y \wedge (x2, y) \in r)$

proof –

from r **have** *trans: trans* r **and** *total: Total* r **and** *antisym: antisym* r
unfolding *card-order-on-def well-order-on-def linear-order-on-def*
partial-order-on-def preorder-on-def **by** *auto*

obtain $y1$ **where** $y1: y1 \in \text{Field } r$ $x1 \neq y1$ $(x1, y1) \in r$
using *Cinfinite-limit[OF x1 r]* **by** *blast*

obtain $y2$ **where** $y2: y2 \in \text{Field } r$ $x2 \neq y2$ $(x2, y2) \in r$
using *Cinfinite-limit[OF x2 r]* **by** *blast*

show *?thesis*

proof (*cases* $y1 = y2$)

case *True* **with** $y1$ $y2$ **show** *?thesis* **by** *blast*

next

case *False*

with $y1(1)$ $y2(1)$ *total* **have** $(y1, y2) \in r \vee (y2, y1) \in r$

unfolding *total-on-def* **by** *auto*

thus *?thesis*

proof

assume $*$: $(y1, y2) \in r$

with *trans* $y1(3)$ **have** $(x1, y2) \in r$ **unfolding** *trans-def* **by** *blast*

with *False* $y1$ $y2$ $*$ *antisym* **show** *?thesis* **by** (*cases* $x1 = y2$) (*auto simp*:

antisym-def)

next

assume $*$: $(y2, y1) \in r$

with *trans* $y2(3)$ **have** $(x2, y1) \in r$ **unfolding** *trans-def* **by** *blast*

with *False* $y1$ $y2$ $*$ *antisym* **show** *?thesis* **by** (*cases* $x2 = y1$) (*auto simp*:

antisym-def)

qed

qed

qed

lemma *Cinfinite-limit-finite*:

[[*finite* X ; $X \subseteq \text{Field } r$; *Cinfinite* r] $\implies \exists y \in \text{Field } r. \forall x \in X. (x \neq y \wedge (x, y) \in r)$

proof (*induct* X *rule: finite-induct*)

case *empty* **thus** *?case* **unfolding** *cinfinite-def* **using** *ex-in-conv[of Field r]*
finite.emptyI **by** *auto*

next

case (*insert* x X)

then obtain y **where** $y: y \in \text{Field } r$ $\forall x \in X. (x \neq y \wedge (x, y) \in r)$ **by** *blast*

then obtain z **where** $z: z \in \text{Field } r$ $x \neq z \wedge (x, z) \in r$ $y \neq z \wedge (y, z) \in r$

using *Cinfinite-limit2[OF - y(1) insert(5), of x] insert(4)* **by** *blast*

show *?case*


```

apply (intro beXI ballI)
apply (erule insertE)
apply hypsubst
apply (rule z(2))
using Card-order-trans[OF insert(5)[THEN conjunct2]] y(2) z(3)
apply blast
apply (rule z(1))
done
qed

```

```

lemma insert-subsetI:  $\llbracket x \in A; X \subseteq A \rrbracket \implies \text{insert } x X \subseteq A$ 
by auto

```

```

lemmas well-order-induct-imp = wo-rel.well-order-induct[of r  $\lambda x. x \in \text{Field } r \longrightarrow P x$  for r P]

```

```

lemma meta-spec2:
assumes ( $\bigwedge x y. \text{PROP } P x y$ )
shows  $\text{PROP } P x y$ 
by (rule assms)

```

```

lemma nchotomy-relcomppE:
assumes  $\bigwedge y. \exists x. y = f x (r \text{ OO } s) a c \bigwedge b. r a (f b) \implies s (f b) c \implies P$ 
shows P
proof (rule relcompp.cases[OF assms(2)], hypsubst)
fix b assume r a b s b c
moreover from assms(1) obtain b' where b = f b' by blast
ultimately show P by (blast intro: assms(3))
qed

```

```

lemma predicate2D-vimage2p:  $\llbracket R \leq \text{vimage2p } f g S; R x y \rrbracket \implies S (f x) (g y)$ 
unfolding vimage2p-def by auto

```

```

lemma ssubst-Pair-rhs:  $\llbracket (r, s) \in R; s' = s \rrbracket \implies (r, s') \in R$ 
by (rule ssubst)

```

```

lemma all-mem-range1:
( $\bigwedge y. y \in \text{range } f \implies P y$ )  $\equiv$  ( $\bigwedge x. P (f x)$ )
by (rule equal-intr-rule) fast+

```

```

lemma all-mem-range2:
( $\bigwedge fa y. fa \in \text{range } f \implies y \in \text{range } fa \implies P y$ )  $\equiv$  ( $\bigwedge x xa. P (f x xa)$ )
by (rule equal-intr-rule) fast+

```

```

lemma all-mem-range3:
( $\bigwedge fa fb y. fa \in \text{range } f \implies fb \in \text{range } fa \implies y \in \text{range } fb \implies P y$ )  $\equiv$  ( $\bigwedge x xa$ 
xb. P (f x xa xb))
by (rule equal-intr-rule) fast+

```

lemma *all-mem-range4*:

$$\begin{aligned} & (\bigwedge fa\ fb\ fc\ y. fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies y \in \text{range } fc \\ & \implies P\ y) \equiv \\ & (\bigwedge x\ xa\ xb\ xc. P\ (f\ x\ xa\ xb\ xc)) \\ & \text{by (rule equal-intr-rule) fast+} \end{aligned}$$

lemma *all-mem-range5*:

$$\begin{aligned} & (\bigwedge fa\ fb\ fc\ fd\ y. fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies fd \in \text{range } \\ & fc \implies \\ & y \in \text{range } fd \implies P\ y) \equiv \\ & (\bigwedge x\ xa\ xb\ xc\ xd. P\ (f\ x\ xa\ xb\ xc\ xd)) \\ & \text{by (rule equal-intr-rule) fast+} \end{aligned}$$

lemma *all-mem-range6*:

$$\begin{aligned} & (\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ y. fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies fd \in \\ & \text{range } fc \implies \\ & fe \in \text{range } fd \implies ff \in \text{range } fe \implies y \in \text{range } ff \implies P\ y) \equiv \\ & (\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf. P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf)) \\ & \text{by (rule equal-intr-rule) (fastforce, fast)} \end{aligned}$$

lemma *all-mem-range7*:

$$\begin{aligned} & (\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ y. fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies fd \\ & \in \text{range } fc \implies \\ & fe \in \text{range } fd \implies ff \in \text{range } fe \implies fg \in \text{range } ff \implies y \in \text{range } fg \implies P\ y) \equiv \\ & (\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg. P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg)) \\ & \text{by (rule equal-intr-rule) (fastforce, fast)} \end{aligned}$$

lemma *all-mem-range8*:

$$\begin{aligned} & (\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ fh\ y. fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies \\ & fd \in \text{range } fc \implies \\ & fe \in \text{range } fd \implies ff \in \text{range } fe \implies fg \in \text{range } ff \implies fh \in \text{range } fg \implies y \in \\ & \text{range } fh \implies P\ y) \equiv \\ & (\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh. P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh)) \\ & \text{by (rule equal-intr-rule) (fastforce, fast)} \end{aligned}$$

lemmas *all-mem-range = all-mem-range1 all-mem-range2 all-mem-range3 all-mem-range4*
all-mem-range5
all-mem-range6 all-mem-range7 all-mem-range8

lemma *pred-fun-True-id*: *NO-MATCH* $id\ p \implies \text{pred-fun } (\lambda x. \text{True})\ p\ f = \text{pred-fun}$
 $(\lambda x. \text{True})\ id\ (p \circ f)$

unfolding *fun.pred-map unfolding comp-def id-def ..*

ML-file $\langle \text{Tools/BNF/bnf-lfp-util.ML} \rangle$

ML-file $\langle \text{Tools/BNF/bnf-lfp-tactics.ML} \rangle$

ML-file $\langle \text{Tools/BNF/bnf-lfp.ML} \rangle$

ML-file $\langle \text{Tools/BNF/bnf-lfp-compat.ML} \rangle$

ML-file $\langle \text{Tools/BNF/bnf-lfp-rec-sugar-more.ML} \rangle$

ML-file $\langle \text{Tools/BNF/bnf-lfp-size.ML} \rangle$

ML-file $\langle Tools/datatype-simprocs.ML \rangle$

```

simproc-setup datatype-no-proper-subterm
  ((x :: 'a :: size) = y) =  $\langle K \text{ Datatype-Simprocs.no-proper-subterm-simproc} \rangle$ 

end

```

38 Equivalence Relations in Higher-Order Set Theory

```

theory Equiv-Relations
  imports BNF-Least-Fixpoint
begin

```

38.1 Equivalence relations – set version

```

definition equiv :: 'a set  $\Rightarrow$  ('a  $\times$  'a) set  $\Rightarrow$  bool
  where equiv A r  $\longleftrightarrow$  refl-on A r  $\wedge$  sym r  $\wedge$  trans r

```

```

lemma equivI: refl-on A r  $\Longrightarrow$  sym r  $\Longrightarrow$  trans r  $\Longrightarrow$  equiv A r
  by (simp add: equiv-def)

```

```

lemma equivE:
  assumes equiv A r
  obtains refl-on A r and sym r and trans r
  using assms by (simp add: equiv-def)

```

Suppes, Theorem 70: r is an equiv relation iff $r^{-1} \circ r = r$.

First half: $\text{equiv } A \ r \Longrightarrow r^{-1} \circ r = r$.

```

lemma sym-trans-comp-subset: sym r  $\Longrightarrow$  trans r  $\Longrightarrow$   $r^{-1} \circ r \subseteq r$ 
  unfolding trans-def sym-def converse-unfold by blast

```

```

lemma refl-on-comp-subset: refl-on A r  $\Longrightarrow$   $r \subseteq r^{-1} \circ r$ 
  unfolding refl-on-def by blast

```

```

lemma equiv-comp-eq: equiv A r  $\Longrightarrow$   $r^{-1} \circ r = r$ 
  unfolding equiv-def
  by (iprover intro: sym-trans-comp-subset refl-on-comp-subset equalityI)

```

Second half.

```

lemma comp-equivI:
  assumes  $r^{-1} \circ r = r$  Domain r = A
  shows equiv A r
proof –
  have *:  $\bigwedge x \ y. (x, y) \in r \Longrightarrow (y, x) \in r$ 
    using assms by blast
  show ?thesis

```

unfolding *equiv-def refl-on-def sym-def trans-def*
using *assms* **by** (*auto intro: **)
qed

38.2 Equivalence classes

lemma *equiv-class-subset*: $\text{equiv } A \ r \implies (a, b) \in r \implies r^{\{\!| a |\!| \} \subseteq r^{\{\!| b |\!| \}}$
— lemma for the next result
unfolding *equiv-def trans-def sym-def* **by** *blast*

theorem *equiv-class-eq*: $\text{equiv } A \ r \implies (a, b) \in r \implies r^{\{\!| a |\!| \} = r^{\{\!| b |\!| \}}$
by (*intro equalityI equiv-class-subset; force simp add: equiv-def sym-def*)

lemma *equiv-class-self*: $\text{equiv } A \ r \implies a \in A \implies a \in r^{\{\!| a |\!| \}}$
unfolding *equiv-def refl-on-def* **by** *blast*

lemma *subset-equiv-class*: $\text{equiv } A \ r \implies r^{\{\!| b |\!| \} \subseteq r^{\{\!| a |\!| \}} \implies b \in A \implies (a, b) \in r$
— lemma for the next result
unfolding *equiv-def refl-on-def* **by** *blast*

lemma *eq-equiv-class*: $r^{\{\!| a |\!| \} = r^{\{\!| b |\!| \}} \implies \text{equiv } A \ r \implies b \in A \implies (a, b) \in r$
by (*iprover intro: equalityD2 subset-equiv-class*)

lemma *equiv-class-nondisjoint*: $\text{equiv } A \ r \implies x \in (r^{\{\!| a |\!| \} \cap r^{\{\!| b |\!| \}} \implies (a, b) \in r$
unfolding *equiv-def trans-def sym-def* **by** *blast*

lemma *equiv-type*: $\text{equiv } A \ r \implies r \subseteq A \times A$
unfolding *equiv-def refl-on-def* **by** *blast*

lemma *equiv-class-eq-iff*: $\text{equiv } A \ r \implies (x, y) \in r \iff r^{\{\!| x |\!| \} = r^{\{\!| y |\!| \}} \wedge x \in A \wedge y \in A$
by (*blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type*)

lemma *eq-equiv-class-iff*: $\text{equiv } A \ r \implies x \in A \implies y \in A \implies r^{\{\!| x |\!| \} = r^{\{\!| y |\!| \}} \iff (x, y) \in r$
by (*blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type*)

lemma *disjnt-equiv-class*: $\text{equiv } A \ r \implies \text{disjnt } (r^{\{\!| a |\!| \}) \ (r^{\{\!| b |\!| \}) \iff (a, b) \notin r$
by (*auto dest: equiv-class-self simp: equiv-class-eq-iff disjnt-def*)

38.3 Quotients

definition *quotient* :: $'a \ \text{set} \Rightarrow ('a \times 'a) \ \text{set} \Rightarrow 'a \ \text{set} \ \text{set}$ (**infixl** $'/'/$ 90)
where $A//r = (\bigcup x \in A. \{r^{\{\!| x |\!| \}})$ — set of equiv classes

lemma *quotientI*: $x \in A \implies r^{\{\!| x |\!| \} \in A//r$
unfolding *quotient-def* **by** *blast*

lemma *quotientE*: $X \in A//r \implies (\bigwedge x. X = r^{\{\!| x |\!| \}} \implies x \in A \implies P) \implies P$
unfolding *quotient-def* **by** *blast*

lemma *Union-quotient*: $\text{equiv } A \ r \implies \bigcup (A//r) = A$
unfolding *equiv-def refl-on-def quotient-def by blast*

lemma *quotient-disj*: $\text{equiv } A \ r \implies X \in A//r \implies Y \in A//r \implies X = Y \vee X \cap Y = \{\}$
unfolding *quotient-def equiv-def trans-def sym-def by blast*

lemma *quotient-eqI*:

assumes *equiv* $A \ r \ X \in A//r \ Y \in A//r$ **and** $xy: x \in X \ y \in Y \ (x, y) \in r$
shows $X = Y$

proof –

obtain $a \ b$ **where** $a \in A$ **and** $a: X = r \ \{a\}$ **and** $b \in A$ **and** $b: Y = r \ \{b\}$
using *assms by (auto elim!: quotientE)*

then have $(a, b) \in r$

using $xy \ \langle \text{equiv } A \ r \rangle$ **unfolding** *equiv-def sym-def trans-def by blast*

then show *?thesis*

unfolding $a \ b$ **by** (*rule equiv-class-eq [OF $\langle \text{equiv } A \ r \rangle$]*)

qed

lemma *quotient-eq-iff*:

assumes *equiv* $A \ r \ X \in A//r \ Y \in A//r$ **and** $xy: x \in X \ y \in Y$
shows $X = Y \longleftrightarrow (x, y) \in r$

proof

assume $L: X = Y$

with *assms* **show** $(x, y) \in r$

unfolding *equiv-def sym-def trans-def by (blast elim!: quotientE)*

next

assume $\S: (x, y) \in r$ **show** $X = Y$

by (*rule quotient-eqI (use \S assms in $\langle \text{blast+} \rangle$)*)

qed

lemma *eq-equiv-class-iff2*: $\text{equiv } A \ r \implies x \in A \implies y \in A \implies \{x\}//r = \{y\}//r$
 $\longleftrightarrow (x, y) \in r$

by (*simp add: quotient-def eq-equiv-class-iff*)

lemma *quotient-empty [simp]*: $\{\}//r = \{\}$

by (*simp add: quotient-def*)

lemma *quotient-is-empty [iff]*: $A//r = \{\} \longleftrightarrow A = \{\}$

by (*simp add: quotient-def*)

lemma *quotient-is-empty2 [iff]*: $\{\} = A//r \longleftrightarrow A = \{\}$

by (*simp add: quotient-def*)

lemma *singleton-quotient*: $\{x\}//r = \{r \ \{x\}\}$

by (*simp add: quotient-def*)

lemma *quotient-diff1*: *inj-on* $(\lambda a. \{a\}//r) \ A \implies a \in A \implies (A - \{a\})//r =$

$A//r - \{a\}//r$
unfolding *quotient-def inj-on-def* **by** *blast*

38.4 Refinement of one equivalence relation WRT another

lemma *refines-equiv-class-eq*: $R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies R''(S''\{a\}) = S''\{a\}$
by (*auto simp: equiv-class-eq-iff*)

lemma *refines-equiv-class-eq2*: $R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies S''(R''\{a\}) = S''\{a\}$
by (*auto simp: equiv-class-eq-iff*)

lemma *refines-equiv-image-eq*: $R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies (\lambda X. S''X) '(A//R) = A//S$
by (*auto simp: quotient-def image-UN refines-equiv-class-eq2*)

lemma *finite-refines-finite*:
 $\text{finite } (A//R) \implies R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies \text{finite } (A//S)$
by (*erule finite-surj [where f = $\lambda X. S''X$] (simp add: refines-equiv-image-eq)*)

lemma *finite-refines-card-le*:
 $\text{finite } (A//R) \implies R \subseteq S \implies \text{equiv } A \ R \implies \text{equiv } A \ S \implies \text{card } (A//S) \leq \text{card } (A//R)$
by (*subst refines-equiv-image-eq [of R S A, symmetric]*)
(auto simp: card-image-le [where f = $\lambda X. S''X$])

38.5 Defining unary operations upon equivalence classes

A congruence-preserving function.

definition *congruent* :: $('a \times 'a) \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$
where $\text{congruent } r \ f \longleftrightarrow (\forall (y, z) \in r. f \ y = f \ z)$

lemma *congruentI*: $(\bigwedge y \ z. (y, z) \in r \implies f \ y = f \ z) \implies \text{congruent } r \ f$
by (*auto simp add: congruent-def*)

lemma *congruentD*: $\text{congruent } r \ f \implies (y, z) \in r \implies f \ y = f \ z$
by (*auto simp add: congruent-def*)

abbreviation *RESPECTS* :: $('a \Rightarrow 'b) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow \text{bool}$ (**infixr** *respects* 80)
where $f \ \text{respects } r \equiv \text{congruent } r \ f$

lemma *UN-constant-eq*: $a \in A \implies \forall y \in A. f \ y = c \implies (\bigcup y \in A. f \ y) = c$
— lemma required to prove *UN-equiv-class*
by *auto*

lemma *UN-equiv-class*:

assumes *equiv A r f respects r a* $a \in A$
shows $(\bigcup x \in r \text{ `` } \{a\}. f x) = f a$
 — Conversion rule
proof –
have $\S: \forall x \in r \text{ `` } \{a\}. f x = f a$
using *assms unfolding equiv-def congruent-def sym-def* **by** *blast*
show *?thesis*
by (*iprover intro: assms UN-constant-eq [OF equiv-class-self §]*)
qed

lemma *UN-equiv-class-type:*

assumes *r: equiv A r f respects r* **and** *X: X* $X \in A//r$ **and** *AB: $\bigwedge x. x \in A \implies f x \in B$*
shows $(\bigcup x \in X. f x) \in B$
using *assms unfolding quotient-def*
by (*auto simp: UN-equiv-class [OF r]*)

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; *bcong* could be $\bigwedge y. y \in A \implies f y \in B$.

lemma *UN-equiv-class-inject:*

assumes *equiv A r f respects r*
and *eq: $(\bigcup x \in X. f x) = (\bigcup y \in Y. f y)$*
and *X: X* $X \in A//r$ **and** *Y: Y* $Y \in A//r$
and *fr: $\bigwedge x y. x \in A \implies y \in A \implies f x = f y \implies (x, y) \in r$*
shows $X = Y$

proof –

obtain *a b where a* $a \in A$ **and** *a: X = r* “ $\{a\}$ ” **and** *b* $b \in A$ **and** *b: Y = r* “ $\{b\}$ ”
using *assms by (auto elim!: quotientE)*
then have $\bigcup (f \text{ ` } r \text{ `` } \{a\}) = f a \bigcup (f \text{ ` } r \text{ `` } \{b\}) = f b$
by (*iprover intro: UN-equiv-class [OF <equiv A r>] assms*)
then have $f a = f b$
using *eq unfolding a b by (iprover intro: trans sym)*
then have $(a, b) \in r$
using *fr <a* $a \in A$ *>* *<b* $b \in A$ *>* **by** *blast*
then show *?thesis*
unfolding *a b by (rule equiv-class-eq [OF <equiv A r>])*
qed

38.6 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments.

definition *congruent2* $:: ('a \times 'a) \text{ set} \implies ('b \times 'b) \text{ set} \implies ('a \implies 'b \implies 'c) \implies \text{bool}$
where *congruent2 r1 r2 f* $\longleftrightarrow (\forall (y1, z1) \in r1. \forall (y2, z2) \in r2. f y1 y2 = f z1 z2)$

lemma *congruent2I'*:

assumes $\bigwedge y1 z1 y2 z2. (y1, z1) \in r1 \implies (y2, z2) \in r2 \implies f y1 y2 = f z1 z2$
shows *congruent2 r1 r2 f*

using *assms* **by** (*auto simp add: congruent2-def*)

lemma *congruent2D*: $\text{congruent2 } r1 \ r2 \ f \implies (y1, z1) \in r1 \implies (y2, z2) \in r2 \implies f \ y1 \ y2 = f \ z1 \ z2$
by (*auto simp add: congruent2-def*)

Abbreviation for the common case where the relations are identical.

abbreviation *RESPECTS2*:: $(\ 'a \Rightarrow \ 'a \Rightarrow \ 'b) \Rightarrow (\ 'a \times \ 'a) \ \text{set} \Rightarrow \ \text{bool}$ (**infixr** *respects2* 80)
where $f \ \text{respects2} \ r \equiv \ \text{congruent2} \ r \ r \ f$

lemma *congruent2-implies-congruent*:

equiv A r1 \implies *congruent2 r1 r2 f* \implies $a \in A \implies$ *congruent r2 (f a)*
unfolding *congruent-def congruent2-def equiv-def refl-on-def* **by** *blast*

lemma *congruent2-implies-congruent-UN*:

assumes *equiv A1 r1 equiv A2 r2 congruent2 r1 r2 f a* $\in A2$
shows *congruent r1* $(\lambda x1. \bigcup x2 \in r2^{\{a\}}. f \ x1 \ x2)$
unfolding *congruent-def*

proof *clarify*

fix *c d*

assume *cd*: $(c, d) \in r1$

then have $c \in A1 \ d \in A1$

using $\langle \text{equiv } A1 \ r1 \rangle$ **by** (*auto elim!: equiv-type [THEN subsetD, THEN SigmaE2]*)

moreover have $f \ c \ a = f \ d \ a$

using *assms cd* **unfolding** *congruent2-def equiv-def refl-on-def* **by** *blast*

ultimately show $\bigcup (f \ c \ 'r2 \ \{a\}) = \bigcup (f \ d \ 'r2 \ \{a\})$

using *assms* **by** (*simp add: UN-equiv-class congruent2-implies-congruent*)

qed

lemma *UN-equiv-class2*:

equiv A1 r1 \implies *equiv A2 r2* \implies *congruent2 r1 r2 f* \implies $a1 \in A1 \implies a2 \in A2$
 \implies

$(\bigcup x1 \in r1^{\{a1\}}. \bigcup x2 \in r2^{\{a2\}}. f \ x1 \ x2) = f \ a1 \ a2$

by (*simp add: UN-equiv-class congruent2-implies-congruent congruent2-implies-congruent-UN*)

lemma *UN-equiv-class-type2*:

equiv A1 r1 \implies *equiv A2 r2* \implies *congruent2 r1 r2 f*

$\implies X1 \in A1 // r1 \implies X2 \in A2 // r2$

$\implies (\bigwedge x1 \ x2. x1 \in A1 \implies x2 \in A2 \implies f \ x1 \ x2 \in B)$

$\implies (\bigcup x1 \in X1. \bigcup x2 \in X2. f \ x1 \ x2) \in B$

unfolding *quotient-def*

by (*blast intro: UN-equiv-class-type congruent2-implies-congruent-UN congruent2-implies-congruent quotientI*)

lemma *UN-UN-split-split-eq*:

$(\bigcup (x1, x2) \in X. \bigcup (y1, y2) \in Y. A\ x1\ x2\ y1\ y2) =$
 $(\bigcup x \in X. \bigcup y \in Y. (\lambda(x1, x2). (\lambda(y1, y2). A\ x1\ x2\ y1\ y2)\ y)\ x)$
 — Allows a natural expression of binary operators,
 — without explicit calls to *split*
by *auto*

lemma *congruent2I*:

equiv A1 r1 \implies *equiv A2 r2*
 $\implies (\bigwedge y\ z\ w. w \in A2 \implies (y, z) \in r1 \implies f\ y\ w = f\ z\ w)$
 $\implies (\bigwedge y\ z\ w. w \in A1 \implies (y, z) \in r2 \implies f\ w\ y = f\ w\ z)$
 \implies *congruent2 r1 r2 f*
 — Suggested by John Harrison – the two subproofs may be
 — *much* simpler than the direct proof.
unfolding *congruent2-def equiv-def refl-on-def*
by (*blast intro: trans*)

lemma *congruent2-commuteI*:

assumes *equivA: equiv A r*
and *commute: $\bigwedge y\ z. y \in A \implies z \in A \implies f\ y\ z = f\ z\ y$*
and *cong: $\bigwedge y\ z\ w. w \in A \implies (y, z) \in r \implies f\ w\ y = f\ w\ z$*
shows *f respects2 r*
proof (*rule congruent2I [OF equivA equivA]*)
note *eqv = equivA [THEN equiv-type, THEN subsetD, THEN SigmaE2]*
show $\bigwedge y\ z\ w. \llbracket w \in A; (y, z) \in r \rrbracket \implies f\ y\ w = f\ z\ w$
by (*iprover intro: commute [THEN trans] sym cong elim: eqv*)
show $\bigwedge y\ z\ w. \llbracket w \in A; (y, z) \in r \rrbracket \implies f\ w\ y = f\ w\ z$
by (*iprover intro: cong elim: eqv*)
qed

38.7 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*:

assumes *finite A r $\subseteq A \times A$*
shows *finite (A//r)*
 — recall *equiv ?A ?r $\implies ?r \subseteq ?A \times ?A$*
proof –
have *A//r $\subseteq Pow\ A$*
using *assms unfolding quotient-def by blast*
moreover have *finite (Pow A)*
using *assms by simp*
ultimately show *?thesis*
by (*iprover intro: finite-subset*)
qed

lemma *finite-equiv-class: finite A $\implies r \subseteq A \times A \implies X \in A//r \implies finite\ X$*

unfolding *quotient-def*
by (*erule rev-finite-subset*) *blast*

lemma *equiv-imp-dvd-card*:
assumes *finite A equiv A r* $\wedge X. X \in A//r \implies k \text{ dvd card } X$
shows $k \text{ dvd card } A$
proof (*rule Union-quotient [THEN subst]*)
show $k \text{ dvd card } (\bigcup (A // r))$
apply (*rule dvd-partition*)
using *assms*
by (*auto simp: Union-quotient dest: quotient-disj*)
qed (*use assms in blast*)

38.8 Projection

definition *proj* :: $('b \times 'a) \text{ set} \Rightarrow 'b \Rightarrow 'a \text{ set}$
where $\text{proj } r \ x = r \ \{x\}$

lemma *proj-preserves*: $x \in A \implies \text{proj } r \ x \in A//r$
unfolding *proj-def* **by** (*rule quotientI*)

lemma *proj-in-iff*:
assumes *equiv A r*
shows $\text{proj } r \ x \in A//r \longleftrightarrow x \in A$
(is ?lhs \longleftrightarrow ?rhs)

proof
assume *?rhs*
then show *?lhs* **by** (*simp add: proj-preserves*)
next
assume *?lhs*
then show *?rhs*
unfolding *proj-def quotient-def*
proof *safe*
fix *y*
assume $y \in A$ **and** $r \ \{x\} = r \ \{y\}$
moreover have $y \in r \ \{y\}$
using *assms y* **unfolding** *equiv-def refl-on-def* **by** *blast*
ultimately have $(x, y) \in r$ **by** *blast*
then show $x \in A$
using *assms* **unfolding** *equiv-def refl-on-def* **by** *blast*
qed
qed

lemma *proj-iff*: $\text{equiv } A \ r \implies \{x, y\} \subseteq A \implies \text{proj } r \ x = \text{proj } r \ y \longleftrightarrow (x, y) \in r$
by (*simp add: proj-def eq-equiv-class-iff*)

lemma *proj-image*: $\text{proj } r \ ` A = A//r$
unfolding *proj-def[abs-def] quotient-def* **by** *blast*

lemma *in-quotient-imp-non-empty*: $\text{equiv } A \ r \implies X \in A//r \implies X \neq \{\}$

unfolding *quotient-def* **using** *equiv-class-self* **by** *fast*

lemma *in-quotient-imp-in-rel*: $\text{equiv } A \ r \implies X \in A//r \implies \{x, y\} \subseteq X \implies (x, y) \in r$
using *quotient-eq-iff*[*THEN iffD1*] **by** *fastforce*

lemma *in-quotient-imp-closed*: $\text{equiv } A \ r \implies X \in A//r \implies x \in X \implies (x, y) \in r \implies y \in X$
unfolding *quotient-def equiv-def trans-def* **by** *blast*

lemma *in-quotient-imp-subset*: $\text{equiv } A \ r \implies X \in A//r \implies X \subseteq A$
using *in-quotient-imp-in-rel equiv-type* **by** *fastforce*

38.9 Equivalence relations – predicate version

Partial equivalences.

definition *part-equivp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *part-equivp* $R \longleftrightarrow (\exists x. R \ x \ x) \wedge (\forall x \ y. R \ x \ y \longleftrightarrow R \ x \ x \wedge R \ y \ y \wedge R \ x = R \ y)$
 — John-Harrison-style characterization

lemma *part-equivpI*: $\exists x. R \ x \ x \implies \text{symp } R \implies \text{transp } R \implies \text{part-equivp } R$
by (*auto simp add: part-equivp-def*) (*auto elim: sympE transpE*)

lemma *part-equivpE*:
assumes *part-equivp* R
obtains x **where** $R \ x \ x$ **and** *symp* R **and** *transp* R

proof –

from *assms* **have** 1 : $\exists x. R \ x \ x$
and 2 : $\bigwedge x \ y. R \ x \ y \longleftrightarrow R \ x \ x \wedge R \ y \ y \wedge R \ x = R \ y$
unfolding *part-equivp-def* **by** *blast+*

from 1 **obtain** x **where** $R \ x \ x$..

moreover **have** *symp* R

proof (*rule sympI*)

fix $x \ y$

assume $R \ x \ y$

with 2 [*of* $x \ y$] **show** $R \ y \ x$ **by** *auto*

qed

moreover **have** *transp* R

proof (*rule transpI*)

fix $x \ y \ z$

assume $R \ x \ y$ **and** $R \ y \ z$

with 2 [*of* $x \ y$] 2 [*of* $y \ z$] **show** $R \ x \ z$ **by** *auto*

qed

ultimately **show** *thesis* **by** (*rule that*)

qed

lemma *part-equivp-refl-symp-transp*: $\text{part-equivp } R \longleftrightarrow (\exists x. R \ x \ x) \wedge \text{symp } R \wedge \text{transp } R$

by (*auto intro: part-equivpI elim: part-equivpE*)

lemma *part-equivp-symp*: $\text{part-equivp } R \implies R \ x \ y \implies R \ y \ x$
by (*erule part-equivpE, erule sympE*)

lemma *part-equivp-transp*: $\text{part-equivp } R \implies R \ x \ y \implies R \ y \ z \implies R \ x \ z$
by (*erule part-equivpE, erule transpE*)

lemma *part-equivp-typedef*: $\text{part-equivp } R \implies \exists d. d \in \{c. \exists x. R \ x \ x \wedge c = \text{Collect } (R \ x)\}$
by (*auto elim: part-equivpE*)

Total equivalences.

definition *equivp* :: $(\ 'a \Rightarrow \ 'a \Rightarrow \ \text{bool}) \Rightarrow \ \text{bool}$
where *equivp* $R \longleftrightarrow (\forall x \ y. R \ x \ y = (R \ x = R \ y))$ — John-Harrison-style characterization

lemma *equivpI*: $\text{reflp } R \implies \text{symp } R \implies \text{transp } R \implies \text{equivp } R$
by (*auto elim: reflpE sympE transpE simp add: equivp-def*)

lemma *equivpE*:
assumes *equivp* R
obtains *reflp* R **and** *symp* R **and** *transp* R
using *assms* **by** (*auto intro!: that reflpI sympI transpI simp add: equivp-def*)

lemma *equivp-implies-part-equivp*: $\text{equivp } R \implies \text{part-equivp } R$
by (*auto intro: part-equivpI elim: equivpE reflpE*)

lemma *equivp-equiv*: $\text{equiv } \text{UNIV } A \longleftrightarrow \text{equivp } (\lambda x \ y. (x, y) \in A)$
by (*auto intro!: equivI equivpI [to-set] elim!: equivE equivpE [to-set]*)

lemma *equivp-reflp-symp-transp*: $\text{equivp } R \longleftrightarrow \text{reflp } R \wedge \text{symp } R \wedge \text{transp } R$
by (*auto intro: equivpI elim: equivpE*)

lemma *identity-equivp*: $\text{equivp } (=)$
by (*auto intro: equivpI reflpI sympI transpI*)

lemma *equivp-reflp*: $\text{equivp } R \implies R \ x \ x$
by (*erule equivpE, erule reflpE*)

lemma *equivp-symp*: $\text{equivp } R \implies R \ x \ y \implies R \ y \ x$
by (*erule equivpE, erule sympE*)

lemma *equivp-transp*: $\text{equivp } R \implies R \ x \ y \implies R \ y \ z \implies R \ x \ z$
by (*erule equivpE, erule transpE*)

lemma *equivp-rtranclp*: $\text{symp } r \implies \text{equivp } r^{**}$
by(*intro equivpI reflpI sympI transpI*)(*auto dest: sympD[OF symp-rtranclp]*)

lemmas *equivp-rtranclp-symclp* [*simp*] = *equivp-rtranclp*[*OF symp-on-symclp*]

lemma *equivp-vimage2p*: *equivp R* \implies *equivp (vimage2p f f R)*
by(*auto simp add: equivp-def vimage2p-def dest: fun-cong*)

lemma *equivp-imp-transp*: *equivp R* \implies *transp R*
by(*simp add: equivp-reflp-symp-transp*)

38.10 Equivalence closure

definition *equivclp* :: (*'a* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow *'a* \Rightarrow *'a* \Rightarrow *bool* **where**
equivclp r = (*symclp r*)^{**}

lemma *transp-equivclp* [*simp*]: *transp (equivclp r)*
by(*simp add: equivclp-def*)

lemma *reflp-equivclp* [*simp*]: *reflp (equivclp r)*
by(*simp add: equivclp-def*)

lemma *symp-equivclp* [*simp*]: *symp (equivclp r)*
by(*simp add: equivclp-def*)

lemma *equivp-evequivclp* [*simp*]: *equivp (equivclp r)*
by(*simp add: equivpI*)

lemma *tranclp-equivclp* [*simp*]: (*equivclp r*)⁺⁺ = *equivclp r*
by(*simp add: equivclp-def*)

lemma *rtranclp-equivclp* [*simp*]: (*equivclp r*)^{**} = *equivclp r*
by(*simp add: equivclp-def*)

lemma *symclp-equivclp* [*simp*]: *symclp (equivclp r)* = *equivclp r*
by(*simp add: equivclp-def symp-symclp-eq*)

lemma *equivclp-symclp* [*simp*]: *equivclp (symclp r)* = *equivclp r*
by(*simp add: equivclp-def*)

lemma *equivclp-conversep* [*simp*]: *equivclp (conversep r)* = *equivclp r*
by(*simp add: equivclp-def*)

lemma *equivclp-sym* [*sym*]: *equivclp r x y* \implies *equivclp r y x*
by(*rule sympD*[*OF symp-equivclp*])

lemma *equivclp-OO-equivclp-le-equivclp*: *equivclp r OO equivclp r* \leq *equivclp r*
by(*rule transp-relcompp-less-eq transp-equivclp*)⁺

lemma *rtranclp-le-equivclp*: *r*^{**} \leq *equivclp r*
unfolding *equivclp-def* **by**(*rule rtranclp-mono*)(*simp add: symclp-pointfree*)

lemma *rtranclp-conversep-le-equivclp*: $r^{-1-1^{**}} \leq \text{equivclp } r$
unfolding *equivclp-def* **by**(rule *rtranclp-mono*)(simp add: *symclp-pointfree*)

lemma *symclp-rtranclp-le-equivclp*: $\text{symclp } r^{**} \leq \text{equivclp } r$
unfolding *symclp-pointfree*
by(rule *le-supI*)(simp-all add: *rtranclp-conversep[symmetric]* *rtranclp-le-equivclp* *rtranclp-conversep-le-equivclp*)

lemma *r-OO-conversep-into-equivclp*:
 $r^{**} \text{ OO } r^{-1-1^{**}} \leq \text{equivclp } r$
by(blast intro: *order-trans[OF - equivclp-OO-equivclp-le-equivclp]* *relcompp-mono* *rtranclp-le-equivclp* *rtranclp-conversep-le-equivclp* del: *predicate2I*)

lemma *equivclp-induct* [*consumes 1, case-names base step, induct pred: equivclp*]:
assumes *a: equivclp r a b*
and cases: $P a \wedge y z. \text{equivclp } r a y \implies r y z \vee r z y \implies P y \implies P z$
shows $P b$
using *a* **unfolding** *equivclp-def*
by(*induction rule: rtranclp-induct; fold equivclp-def; blast intro: cases elim: symclpE*)

lemma *converse-equivclp-induct* [*consumes 1, case-names base step*]:
assumes *major: equivclp r a b*
and cases: $P b \wedge y z. r y z \vee r z y \implies \text{equivclp } r z b \implies P z \implies P y$
shows $P a$
using *major* **unfolding** *equivclp-def*
by(*induction rule: converse-rtranclp-induct; fold equivclp-def; blast intro: cases elim: symclpE*)

lemma *equivclp-refl* [*simp*]: $\text{equivclp } r x x$
by(rule *reflpD[OF refl-equivclp]*)

lemma *r-into-equivclp* [*intro*]: $r x y \implies \text{equivclp } r x y$
unfolding *equivclp-def* **by**(blast intro: *symclpI*)

lemma *converse-r-into-equivclp* [*intro*]: $r y x \implies \text{equivclp } r x y$
unfolding *equivclp-def* **by**(blast intro: *symclpI*)

lemma *rtranclp-into-equivclp*: $r^{**} x y \implies \text{equivclp } r x y$
using *rtranclp-le-equivclp[of r]* **by** *blast*

lemma *converse-rtranclp-into-equivclp*: $r^{**} y x \implies \text{equivclp } r x y$
by(blast intro: *equivclp-sym rtranclp-into-equivclp*)

lemma *equivclp-into-equivclp*: $\llbracket \text{equivclp } r a b; r b c \vee r c b \rrbracket \implies \text{equivclp } r a c$
unfolding *equivclp-def* **by**(erule *rtranclp.rtrancl-into-rtrancl*)(auto intro: *symclpI*)

lemma *equivclp-trans* [*trans*]: $\llbracket \text{equivclp } r a b; \text{equivclp } r b c \rrbracket \implies \text{equivclp } r a c$

```

using equivclp-OO-equivclp-le-equivclp[of r] by blast

hide-const (open) proj

end

theory Basic-BNF-LFPs
imports BNF-Least-Fixpoint
begin

definition xlor :: 'a  $\Rightarrow$  'a where
  xlor x = x

lemma xlor-map:  $f$  (xlor x) = xlor ( $f$  x)
  unfolding xlor-def by (rule refl)

lemma xlor-map-unique:  $u \circ \text{xlor} = \text{xlor} \circ f \implies u = f$ 
  unfolding o-def xlor-def .

lemma xlor-set:  $f$  (xlor x) =  $f$  x
  unfolding xlor-def by (rule refl)

lemma xlor-rel:  $R$  (xlor x) (xlor y) =  $R$  x y
  unfolding xlor-def by (rule refl)

lemma xlor-induct:  $(\bigwedge x. P$  (xlor x))  $\implies P$  z
  unfolding xlor-def by assumption

lemma xlor-xlor: xlor (xlor x) = x
  unfolding xlor-def by (rule refl)

lemmas xlor-inject = xlor-rel[of (=)]

lemma xlor-rel-induct:  $(\bigwedge x y. \text{vimage2p } \text{id-bnf } \text{id-bnf } R \ x \ y \implies IR$  (xlor x) (xlor
y))  $\implies R \leq IR$ 
  unfolding xlor-def vimage2p-def id-bnf-def ..

lemma xlor-iff-xlor:  $u = \text{xlor } w \iff \text{xlor } u = w$ 
  unfolding xlor-def ..

lemma Inl-def-alt:  $Inl \equiv (\lambda a. \text{xlor } (\text{id-bnf } (Inl \ a)))$ 
  unfolding xlor-def id-bnf-def by (rule reflexive)

lemma Inr-def-alt:  $Inr \equiv (\lambda a. \text{xlor } (\text{id-bnf } (Inr \ a)))$ 
  unfolding xlor-def id-bnf-def by (rule reflexive)

lemma Pair-def-alt:  $Pair \equiv (\lambda a \ b. \text{xlor } (\text{id-bnf } (a, \ b)))$ 
  unfolding xlor-def id-bnf-def by (rule reflexive)

```

definition $ctor-rec :: 'a \Rightarrow 'a$ **where**

$ctor-rec\ x = x$

lemma $ctor-rec: g = id \Longrightarrow ctor-rec\ f\ (xtor\ x) = f\ ((id-bnf \circ g \circ id-bnf)\ x)$

unfolding $ctor-rec-def\ id-bnf-def\ xtor-def\ comp-def\ id-def$ **by** $hypsubst$ (rule refl)

lemma $ctor-rec-unique: g = id \Longrightarrow f \circ xtor = s \circ (id-bnf \circ g \circ id-bnf) \Longrightarrow f = ctor-rec\ s$

unfolding $ctor-rec-def\ id-bnf-def\ xtor-def\ comp-def\ id-def$ **by** $hypsubst$ (rule refl)

lemma $ctor-rec-def-alt: f = ctor-rec\ (f \circ id-bnf)$

unfolding $ctor-rec-def\ id-bnf-def\ comp-def$ **by** (rule refl)

lemma $ctor-rec-o-map: ctor-rec\ f \circ g = ctor-rec\ (f \circ (id-bnf \circ g \circ id-bnf))$

unfolding $ctor-rec-def\ id-bnf-def\ comp-def$ **by** (rule refl)

lemma $ctor-rec-transfer: rel-fun\ (rel-fun\ (vimage2p\ id-bnf\ id-bnf\ R)\ S)\ (rel-fun\ R\ S)\ ctor-rec\ ctor-rec$

unfolding $rel-fun-def\ vimage2p-def\ id-bnf-def\ ctor-rec-def$ **by** $simp$

lemma $eqfst-iff: a = fst\ p \longleftrightarrow (\exists b. p = (a, b))$

by (cases p) auto

lemma $eqsnd-iff: b = snd\ p \longleftrightarrow (\exists a. p = (a, b))$

by (cases p) auto

lemma $ex-neg-all-pos: ((\exists x. P\ x) \Longrightarrow Q) \equiv (\bigwedge x. P\ x \Longrightarrow Q)$

by standard blast+

lemma $hypsubst-in-prems: (\bigwedge x. y = x \Longrightarrow z = f\ x \Longrightarrow P) \equiv (z = f\ y \Longrightarrow P)$

by standard blast+

lemma $isl-map-sum:$

$isl\ (map-sum\ f\ g\ s) = isl\ s$

by (cases s) simp-all

lemma $map-sum-sel:$

$isl\ s \Longrightarrow projl\ (map-sum\ f\ g\ s) = f\ (projl\ s)$

$\neg isl\ s \Longrightarrow projr\ (map-sum\ f\ g\ s) = g\ (projr\ s)$

by (cases s; simp)+

lemma $set-sum-sel:$

$isl\ s \Longrightarrow projl\ s \in setl\ s$

$\neg isl\ s \Longrightarrow projr\ s \in setr\ s$

by (cases s; auto intro: setl.intros setr.intros)+

lemma $rel-sum-sel: rel-sum\ R1\ R2\ a\ b = (isl\ a = isl\ b \wedge$

$(isl\ a \longrightarrow isl\ b \longrightarrow R1\ (projl\ a)\ (projl\ b)) \wedge$

(\neg isl $a \longrightarrow \neg$ isl $b \longrightarrow R2$ (*projr* a) (*projr* b)))
by (*cases* a b *rule: sum.exhaust[case-product sum.exhaust]*) *simp-all*

lemma *isl-transfer*: *rel-fun* (*rel-sum* A B) (=) *isl isl*
unfolding *rel-fun-def rel-sum-sel* **by** *simp*

lemma *rel-prod-sel*: *rel-prod* $R1$ $R2$ p q = ($R1$ (*fst* p) (*fst* q) \wedge $R2$ (*snd* p) (*snd* q))
by (*force simp: rel-prod.simps elim: rel-prod.cases*)

ML-file \langle Tools/BNF/bnf-lfp-basic-sugar.ML \rangle

declare *prod.size* [*no-atp*]

hide-const (**open**) *xtor ctor-rec*

hide-fact (**open**)

xtor-def xtor-map xtor-set xtor-rel xtor-induct xtor-xtor xtor-inject ctor-rec-def
ctor-rec
ctor-rec-def-alt ctor-rec-o-map xtor-rel-induct Inl-def-alt Inr-def-alt Pair-def-alt

end

39 MESON Proof Method

theory *Meson*

imports *Nat*

begin

39.1 Negation Normal Form

de Morgan laws

lemma *not-conjD*: $\neg(P \wedge Q) \implies \neg P \vee \neg Q$
and *not-disjD*: $\neg(P \vee Q) \implies \neg P \wedge \neg Q$
and *not-notD*: $\neg\neg P \implies P$
and *not-allD*: $\bigwedge P. \neg(\forall x. P(x)) \implies \exists x. \neg P(x)$
and *not-exD*: $\bigwedge P. \neg(\exists x. P(x)) \implies \forall x. \neg P(x)$
by *fast+*

Removal of \longrightarrow and \longleftrightarrow (positive and negative occurrences)

lemma *imp-to-disjD*: $P \longrightarrow Q \implies \neg P \vee Q$
and *not-impD*: $\neg(P \longrightarrow Q) \implies P \wedge \neg Q$
and *iff-to-disjD*: $P = Q \implies (\neg P \vee Q) \wedge (\neg Q \vee P)$
and *not-iffD*: $\neg(P = Q) \implies (P \vee Q) \wedge (\neg P \vee \neg Q)$
— Much more efficient than $P \wedge \neg Q \vee Q \wedge \neg P$ for computing CNF
and *not-refl-disj-D*: $x \neq x \vee P \implies P$
by *fast+*

39.2 Pulling out the existential quantifiers

Conjunction

lemma *conj-exD1*: $\bigwedge P Q. (\exists x. P(x)) \wedge Q \Longrightarrow \exists x. P(x) \wedge Q$
and *conj-exD2*: $\bigwedge P Q. P \wedge (\exists x. Q(x)) \Longrightarrow \exists x. P \wedge Q(x)$
by *fast+*

Disjunction

lemma *disj-exD*: $\bigwedge P Q. (\exists x. P(x)) \vee (\exists x. Q(x)) \Longrightarrow \exists x. P(x) \vee Q(x)$
 — DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
 — With ex-Skolemization, makes fewer Skolem constants
and *disj-exD1*: $\bigwedge P Q. (\exists x. P(x)) \vee Q \Longrightarrow \exists x. P(x) \vee Q$
and *disj-exD2*: $\bigwedge P Q. P \vee (\exists x. Q(x)) \Longrightarrow \exists x. P \vee Q(x)$
by *fast+*

lemma *disj-assoc*: $(P \vee Q) \vee R \Longrightarrow P \vee (Q \vee R)$
and *disj-comm*: $P \vee Q \Longrightarrow Q \vee P$
and *disj-FalseD1*: $\text{False} \vee P \Longrightarrow P$
and *disj-FalseD2*: $P \vee \text{False} \Longrightarrow P$
by *fast+*

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

lemma *make-neg-rule*: $\neg P \vee Q \Longrightarrow ((\neg P \Longrightarrow P) \Longrightarrow Q)$
by *blast*

Version for Plaisted’s “Positive refinement” of the Meson procedure

lemma *make-refined-neg-rule*: $\neg P \vee Q \Longrightarrow (P \Longrightarrow Q)$
by *blast*

P should be a literal

lemma *make-pos-rule*: $P \vee Q \Longrightarrow ((P \Longrightarrow \neg P) \Longrightarrow Q)$
by *blast*

Versions of *make-neg-rule* and *make-pos-rule* that don’t insert new assumptions, for ordinary resolution.

lemmas *make-neg-rule'* = *make-refined-neg-rule*

lemma *make-pos-rule'*: $\llbracket P \vee Q; \neg P \rrbracket \Longrightarrow Q$
by *blast*

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: $\neg P \Longrightarrow ((\neg P \Longrightarrow P) \Longrightarrow \text{False})$
by *blast*

lemma *make-pos-goal*: $P \implies ((P \implies \neg P) \implies \text{False})$
by *blast*

39.3 Lemmas for Forward Proof

There is a similarity to congruence rules. They are also useful in ordinary proofs.

lemma *conj-forward*: $\llbracket P' \wedge Q'; P' \implies P; Q' \implies Q \rrbracket \implies P \wedge Q$
by *blast*

lemma *disj-forward*: $\llbracket P' \vee Q'; P' \implies P; Q' \implies Q \rrbracket \implies P \vee Q$
by *blast*

lemma *imp-forward*: $\llbracket P' \longrightarrow Q'; P \implies P'; Q' \implies Q \rrbracket \implies P \longrightarrow Q$
by *blast*

lemma *imp-forward2*: $\llbracket P' \longrightarrow Q'; P \implies P'; P' \implies Q' \implies Q \rrbracket \implies P \longrightarrow Q$
by *blast*

lemma *disj-forward2*: $\llbracket P' \vee Q'; P' \implies P; \llbracket Q'; P \implies \text{False} \rrbracket \implies Q \rrbracket \implies P \vee Q$
apply *blast*
done

lemma *all-forward*: $\llbracket \forall x. P'(x); !!x. P'(x) \implies P(x) \rrbracket \implies \forall x. P(x)$
by *blast*

lemma *ex-forward*: $\llbracket \exists x. P'(x); !!x. P'(x) \implies P(x) \rrbracket \implies \exists x. P(x)$
by *blast*

39.4 Clausification helper

lemma *TruepropI*: $P \equiv Q \implies \text{Trueprop } P \equiv \text{Trueprop } Q$
by *simp*

lemma *ext-cong-neq*: $F g \neq F h \implies F g \neq F h \wedge (\exists x. g x \neq h x)$
apply (*erule contrapos-np*)
apply *clarsimp*
apply (*rule cong[where f = F]*)
by *auto*

Combinator translation helpers

definition *COMBI* :: $'a \Rightarrow 'a$ **where**
COMBI $P = P$

definition *COMBK* :: $'a \Rightarrow 'b \Rightarrow 'a$ **where**
COMBK $P Q = P$

definition *COMBB* :: ('b => 'c) => ('a => 'b) => 'a => 'c **where**
COMBB P Q R = P (Q R)

definition *COMBC* :: ('a => 'b => 'c) => 'b => 'a => 'c **where**
COMBC P Q R = P R Q

definition *COMBS* :: ('a => 'b => 'c) => ('a => 'b) => 'a => 'c **where**
COMBS P Q R = P R (Q R)

lemma *abs-S*: $\lambda x. (f\ x)\ (g\ x) \equiv \text{COMBS}\ f\ g$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add: COMBS-def*)
done

lemma *abs-I*: $\lambda x. x \equiv \text{COMBI}$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add: COMBI-def*)
done

lemma *abs-K*: $\lambda x. y \equiv \text{COMBK}\ y$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add: COMBK-def*)
done

lemma *abs-B*: $\lambda x. a\ (g\ x) \equiv \text{COMBB}\ a\ g$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add: COMBB-def*)
done

lemma *abs-C*: $\lambda x. (f\ x)\ b \equiv \text{COMBC}\ f\ b$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add: COMBC-def*)
done

39.5 Skolemization helpers

definition *skolem* :: 'a => 'a **where**
skolem = ($\lambda x. x$)

lemma *skolem-COMBK-iff*: $P \longleftrightarrow \text{skolem}\ (\text{COMBK}\ P\ (i::\text{nat}))$
unfolding *skolem-def* *COMBK-def* **by** (*rule refl*)

lemmas *skolem-COMBK-I* = *iffD1* [*OF* *skolem-COMBK-iff*]

39.6 Meson package

ML-file \langle Tools/Meson/meson.ML \rangle

ML-file \langle Tools/Meson/meson-clausify.ML \rangle

ML-file \langle Tools/Meson/meson-tactic.ML \rangle

hide-const (**open**) COMBI COMBK COMBB COMBC COMBS skolem

hide-fact (**open**) not-conjD not-disjD not-notD not-allD not-exD imp-to-disjD

not-impD iff-to-disjD not-iffD not-refl-disj-D conj-exD1 conj-exD2 disj-exD

disj-exD1 disj-exD2 disj-assoc disj-comm disj-FalseD1 disj-FalseD2 TruepropI

ext-cong-neq COMBI-def COMBK-def COMBB-def COMBC-def COMBS-def

abs-I abs-K

abs-B abs-C abs-S skolem-def skolem-COMBK-iff skolem-COMBK-I

end

40 Automatic Theorem Provers (ATPs)

theory ATP

imports Meson Hilbert-Choice

begin

40.1 ATP problems and proofs

ML-file \langle Tools/ATP/atp-util.ML \rangle

ML-file \langle Tools/ATP/atp-problem.ML \rangle

ML-file \langle Tools/ATP/atp-proof.ML \rangle

ML-file \langle Tools/ATP/atp-proof-redirect.ML \rangle

40.2 Higher-order reasoning helpers

definition *fFalse* :: bool where

fFalse \longleftrightarrow False

definition *fTrue* :: bool where

fTrue \longleftrightarrow True

definition *fNot* :: bool \Rightarrow bool where

fNot *P* \longleftrightarrow \neg *P*

definition *fComp* :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool where

fComp *P* = ($\lambda x. \neg$ *P* *x*)

definition *fconj* :: bool \Rightarrow bool \Rightarrow bool where

fconj *P* *Q* \longleftrightarrow *P* \wedge *Q*

definition *fdisj* :: bool \Rightarrow bool \Rightarrow bool where

fdisj *P* *Q* \longleftrightarrow *P* \vee *Q*

definition *fimplies* :: bool \Rightarrow bool \Rightarrow bool where

fimplies *P* *Q* \longleftrightarrow (*P* \longrightarrow *Q*)

definition $fAll :: ('a \Rightarrow bool) \Rightarrow bool$ **where**
 $fAll P \longleftrightarrow All P$

definition $fEx :: ('a \Rightarrow bool) \Rightarrow bool$ **where**
 $fEx P \longleftrightarrow Ex P$

definition $fequal :: 'a \Rightarrow 'a \Rightarrow bool$ **where**
 $fequal x y \longleftrightarrow (x = y)$

definition $fChoice :: ('a \Rightarrow bool) \Rightarrow 'a$ **where**
 $fChoice \equiv Hilbert-Choice.Eps$

lemma $fTrue-ne-fFalse: fFalse \neq fTrue$
unfolding $fFalse-def fTrue-def$ **by** *simp*

lemma $fNot-table:$
 $fNot fFalse = fTrue$
 $fNot fTrue = fFalse$
unfolding $fFalse-def fTrue-def fNot-def$ **by** *auto*

lemma $fconj-table:$
 $fconj fFalse P = fFalse$
 $fconj P fFalse = fFalse$
 $fconj fTrue fTrue = fTrue$
unfolding $fFalse-def fTrue-def fconj-def$ **by** *auto*

lemma $fdisj-table:$
 $fdisj fTrue P = fTrue$
 $fdisj P fTrue = fTrue$
 $fdisj fFalse fFalse = fFalse$
unfolding $fFalse-def fTrue-def fdisj-def$ **by** *auto*

lemma $fimplies-table:$
 $fimplies P fTrue = fTrue$
 $fimplies fFalse P = fTrue$
 $fimplies fTrue fFalse = fFalse$
unfolding $fFalse-def fTrue-def fimplies-def$ **by** *auto*

lemma $fAll-table:$
 $Ex (fComp P) \vee fAll P = fTrue$
 $All P \vee fAll P = fFalse$
unfolding $fFalse-def fTrue-def fComp-def fAll-def$ **by** *auto*

lemma $fEx-table:$
 $All (fComp P) \vee fEx P = fTrue$
 $Ex P \vee fEx P = fFalse$
unfolding $fFalse-def fTrue-def fComp-def fEx-def$ **by** *auto*

lemma *fequal-table*:

fequal x $x = fTrue$

$x = y \vee fequal$ x $y = fFalse$

unfolding *fFalse-def fTrue-def fequal-def* **by** *auto*

lemma *fNot-law*:

fNot $P \neq P$

unfolding *fNot-def* **by** *auto*

lemma *fComp-law*:

fComp P $x \longleftrightarrow \neg P$ x

unfolding *fComp-def* **..**

lemma *fconj-laws*:

fconj P $P \longleftrightarrow P$

fconj P $Q \longleftrightarrow fconj$ Q P

fNot (*fconj* P Q) $\longleftrightarrow fdisj$ (*fNot* P) (*fNot* Q)

unfolding *fNot-def fconj-def fdisj-def* **by** *auto*

lemma *fdisj-laws*:

fdisj P $P \longleftrightarrow P$

fdisj P $Q \longleftrightarrow fdisj$ Q P

fNot (*fdisj* P Q) $\longleftrightarrow fconj$ (*fNot* P) (*fNot* Q)

unfolding *fNot-def fconj-def fdisj-def* **by** *auto*

lemma *fimplies-laws*:

fimplies P $Q \longleftrightarrow fdisj$ ($\neg P$) Q

fNot (*fimplies* P Q) $\longleftrightarrow fconj$ P (*fNot* Q)

unfolding *fNot-def fconj-def fdisj-def fimplies-def* **by** *auto*

lemma *fAll-law*:

fNot (*fAll* R) $\longleftrightarrow fEx$ (*fComp* R)

unfolding *fNot-def fComp-def fAll-def fEx-def* **by** *auto*

lemma *fEx-law*:

fNot (*fEx* R) $\longleftrightarrow fAll$ (*fComp* R)

unfolding *fNot-def fComp-def fAll-def fEx-def* **by** *auto*

lemma *fequal-laws*:

fequal x $y = fequal$ y x

fequal x $y = fFalse \vee fequal$ y $z = fFalse \vee fequal$ x $z = fTrue$

fequal x $y = fFalse \vee fequal$ (f x) (f y) = *fTrue*

unfolding *fFalse-def fTrue-def fequal-def* **by** *auto*

lemma *fChoice-iff-Ex*: P (*fChoice* P) $\longleftrightarrow HOL.Ex$ P

unfolding *fChoice-def*

by (*fact some-eq-ex*)

We use the *Ex* constant on the right-hand side of *fChoice-iff-Ex* because

we want to use the TPTP-native version if fChoice is introduced in a logic that supports FOOL. In logics that don't support it, it gets replaced by fEx during processing. Notice that we cannot use $\exists x. P x$, as existentials are not skolemized by the metis proof method but $Ex P$ is eta-expanded if FOOL is supported.

40.3 Basic connection between ATPs and HOL

ML-file $\langle Tools/lambda-lifting.ML \rangle$
ML-file $\langle Tools/monomorph.ML \rangle$
ML-file $\langle Tools/ATP/atp-problem-generate.ML \rangle$
ML-file $\langle Tools/ATP/atp-proof-reconstruct.ML \rangle$

end

41 Metis Proof Method

theory *Metis*
imports *ATP*
begin

ML-file $\langle \sim\sim/src/Tools/Metis/metis.ML \rangle$

41.1 Literal selection and lambda-lifting helpers

definition *select* :: 'a \Rightarrow 'a **where**
select = ($\lambda x. x$)

lemma *not-atomize*: ($\neg A \Longrightarrow False$) \equiv *Trueprop* *A*
by (*cut-tac atomize-not [of $\neg A$]*) *simp*

lemma *atomize-not-select*: ($A \Longrightarrow select\ False$) \equiv *Trueprop* ($\neg A$)
unfolding *select-def* **by** (*rule atomize-not*)

lemma *not-atomize-select*: ($\neg A \Longrightarrow select\ False$) \equiv *Trueprop* *A*
unfolding *select-def* **by** (*rule not-atomize*)

lemma *select-FalseI*: $False \Longrightarrow select\ False$
by *simp*

definition *lambda* :: 'a \Rightarrow 'a **where**
lambda = ($\lambda x. x$)

lemma *eq-lambdaI*: $x \equiv y \Longrightarrow x \equiv lambda\ y$
unfolding *lambda-def* **by** *assumption*

41.2 Metis package

ML-file \langle Tools/Metis/metis-generate.ML \rangle

ML-file \langle Tools/Metis/metis-reconstruct.ML \rangle

ML-file \langle Tools/Metis/metis-tactic.ML \rangle

hide-const (**open**) *select fFalse fTrue fNot fComp fconj fdisj fimplies fAll fEx fequal lambda*

hide-fact (**open**) *select-def not-atomize atomize-not-select not-atomize-select select-FalseI*

fFalse-def fTrue-def fNot-def fconj-def fdisj-def fimplies-def fAll-def fEx-def fequal-def

fTrue-ne-fFalse fNot-table fconj-table fdisj-table fimplies-table fAll-table fEx-table fequal-table fAll-table fEx-table fNot-law fComp-law fconj-laws fdisj-laws fimplies-laws

fequal-laws fAll-law fEx-law lambda-def eq-lambdaI

end

42 Generic theorem transfer using relations

theory *Transfer*

imports *Basic-BNF-LFPs Hilbert-Choice Metis*

begin

42.1 Relator for function space

bundle *lifting-syntax*

begin

notation *rel-fun* (**infixr** $====>$ 55)

notation *map-fun* (**infixr** $--->$ 55)

end

context includes *lifting-syntax*

begin

lemma *rel-funD2*:

assumes *rel-fun A B f g and A x x*

shows *B (f x) (g x)*

using *assms by (rule rel-funD)*

lemma *rel-funE*:

assumes *rel-fun A B f g and A x y*

obtains *B (f x) (g y)*

using *assms by (simp add: rel-fun-def)*

lemmas *rel-fun-eq = fun.rel-eq*

lemma *rel-fun-eq-rel*:

shows *rel-fun* $(=) R = (\lambda f g. \forall x. R (f x) (g x))$
by (*simp add: rel-fun-def*)

42.2 Transfer method

Explicit tag for relation membership allows for backward proof methods.

definition *Rel* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$
where *Rel* $r \equiv r$

Handling of equality relations

definition *is-equality* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *is-equality* $R \longleftrightarrow R = (=)$

lemma *is-equality-eq*: *is-equality* $(=)$
unfolding *is-equality-def* **by** *simp*

Reverse implication for monotonicity rules

definition *rev-implies* **where**
rev-implies $x y \longleftrightarrow (y \longrightarrow x)$

Handling of meta-logic connectives

definition *transfer-forall* **where**
transfer-forall $\equiv \text{All}$

definition *transfer-implies* **where**
transfer-implies $\equiv (\longrightarrow)$

definition *transfer-bforall* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *transfer-bforall* $\equiv (\lambda P Q. \forall x. P x \longrightarrow Q x)$

lemma *transfer-forall-eq*: $(\bigwedge x. P x) \equiv \text{Trueprop} (\text{transfer-forall} (\lambda x. P x))$
unfolding *atomize-all* *transfer-forall-def* **..**

lemma *transfer-implies-eq*: $(A \Longrightarrow B) \equiv \text{Trueprop} (\text{transfer-implies} A B)$
unfolding *atomize-imp* *transfer-implies-def* **..**

lemma *transfer-bforall-unfold*:
 $\text{Trueprop} (\text{transfer-bforall} P (\lambda x. Q x)) \equiv (\bigwedge x. P x \Longrightarrow Q x)$
unfolding *transfer-bforall-def* *atomize-imp* *atomize-all* **..**

lemma *transfer-start*: $\llbracket P; \text{Rel} (=) P Q \rrbracket \Longrightarrow Q$
unfolding *Rel-def* **by** *simp*

lemma *transfer-start'*: $\llbracket P; \text{Rel} (\longrightarrow) P Q \rrbracket \Longrightarrow Q$
unfolding *Rel-def* **by** *simp*

lemma *transfer-prover-start*: $\llbracket x = x'; \text{Rel} R x' y \rrbracket \Longrightarrow \text{Rel} R x y$
by *simp*

lemma *untransfer-start*: $\llbracket Q; \text{Rel } (=) P Q \rrbracket \Longrightarrow P$
unfolding *Rel-def* **by** *simp*

lemma *Rel-eq-reft*: $\text{Rel } (=) x x$
unfolding *Rel-def* **..**

lemma *Rel-app*:
assumes $\text{Rel } (A \Longrightarrow B) f g$ **and** $\text{Rel } A x y$
shows $\text{Rel } B (f x) (g y)$
using *assms* **unfolding** *Rel-def rel-fun-def* **by** *fast*

lemma *Rel-abs*:
assumes $\bigwedge x y. \text{Rel } A x y \Longrightarrow \text{Rel } B (f x) (g y)$
shows $\text{Rel } (A \Longrightarrow B) (\lambda x. f x) (\lambda y. g y)$
using *assms* **unfolding** *Rel-def rel-fun-def* **by** *fast*

42.3 Predicates on relations, i.e. “class constraints”

definition *left-total* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *left-total* $R \longleftrightarrow (\forall x. \exists y. R x y)$

definition *left-unique* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *left-unique* $R \longleftrightarrow (\forall x y z. R x z \longrightarrow R y z \longrightarrow x = y)$

definition *right-total* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *right-total* $R \longleftrightarrow (\forall y. \exists x. R x y)$

definition *right-unique* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *right-unique* $R \longleftrightarrow (\forall x y z. R x y \longrightarrow R x z \longrightarrow y = z)$

definition *bi-total* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *bi-total* $R \longleftrightarrow (\forall x. \exists y. R x y) \wedge (\forall y. \exists x. R x y)$

definition *bi-unique* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *bi-unique* $R \longleftrightarrow$
 $(\forall x y z. R x y \longrightarrow R x z \longrightarrow y = z) \wedge$
 $(\forall x y z. R x z \longrightarrow R y z \longrightarrow x = y)$

lemma *left-unique-iff*: $\text{left-unique } R \longleftrightarrow (\forall z. \exists_{\leq 1} x. R x z)$
unfolding *Uniq-def left-unique-def* **by** *force*

lemma *left-uniqueI*: $(\bigwedge x y z. \llbracket A x z; A y z \rrbracket \Longrightarrow x = y) \Longrightarrow \text{left-unique } A$
unfolding *left-unique-def* **by** *blast*

lemma *left-uniqueD*: $\llbracket \text{left-unique } A; A x z; A y z \rrbracket \Longrightarrow x = y$
unfolding *left-unique-def* **by** *blast*

lemma *left-totalI*:

$(\bigwedge x. \exists y. R x y) \implies \text{left-total } R$
unfolding *left-total-def* **by** *blast*

lemma *left-totalE*:
assumes *left-total R*
obtains $(\bigwedge x. \exists y. R x y)$
using *assms* **unfolding** *left-total-def* **by** *blast*

lemma *bi-uniqueDr*: $\llbracket \text{bi-unique } A; A x y; A x z \rrbracket \implies y = z$
by(*simp add: bi-unique-def*)

lemma *bi-uniqueDl*: $\llbracket \text{bi-unique } A; A x y; A z y \rrbracket \implies x = z$
by(*simp add: bi-unique-def*)

lemma *bi-unique-iff*: $\text{bi-unique } R \longleftrightarrow (\forall z. \exists_{\leq 1} x. R x z) \wedge (\forall z. \exists_{\leq 1} x. R z x)$
unfolding *Uniq-def bi-unique-def* **by** *force*

lemma *right-unique-iff*: $\text{right-unique } R \longleftrightarrow (\forall z. \exists_{\leq 1} x. R z x)$
unfolding *Uniq-def right-unique-def* **by** *force*

lemma *right-uniqueI*: $(\bigwedge x y z. \llbracket A x y; A x z \rrbracket \implies y = z) \implies \text{right-unique } A$
unfolding *right-unique-def* **by** *fast*

lemma *right-uniqueD*: $\llbracket \text{right-unique } A; A x y; A x z \rrbracket \implies y = z$
unfolding *right-unique-def* **by** *fast*

lemma *right-totalI*: $(\bigwedge y. \exists x. A x y) \implies \text{right-total } A$
by(*simp add: right-total-def*)

lemma *right-totalE*:
assumes *right-total A*
obtains *x where A x y*
using *assms* **by**(*auto simp add: right-total-def*)

lemma *right-total-alt-def2*:
 $\text{right-total } R \longleftrightarrow ((R \implies\!\!\implies (\longrightarrow)) \implies\!\!\implies (\longrightarrow)) \text{ All All (is ?lhs = ?rhs)}$

proof

assume *?lhs then show ?rhs*
unfolding *right-total-def rel-fun-def* **by** *blast*

next

assume $\S: ?rhs$

show *?lhs*

using \S [*unfolded rel-fun-def, rule-format, of $\lambda x. \text{True } \lambda y. \exists x. R x y$*]

unfolding *right-total-def* **by** *blast*

qed

lemma *right-unique-alt-def2*:
 $\text{right-unique } R \longleftrightarrow (R \implies\!\!\implies R \implies\!\!\implies (\longrightarrow)) (=) (=)$
unfolding *right-unique-def rel-fun-def* **by** *auto*

lemma *bi-total-alt-def2*:

bi-total $R \longleftrightarrow ((R \implies (=)) \implies (=)) \text{ All All (is ?lhs = ?rhs)}$

proof

assume *?lhs then show ?rhs*

unfolding *bi-total-def rel-fun-def* **by** *blast*

next

assume §: *?rhs*

show *?lhs*

using § [*unfolded rel-fun-def, rule-format, of $\lambda x. \exists y. R x y \lambda y. True$*]

using § [*unfolded rel-fun-def, rule-format, of $\lambda x. True \lambda y. \exists x. R x y$*]

by (*auto simp: bi-total-def*)

qed

lemma *bi-unique-alt-def2*:

bi-unique $R \longleftrightarrow (R \implies R \implies (=)) (=) (=)$

unfolding *bi-unique-def rel-fun-def* **by** *auto*

lemma [*simp*]:

shows *left-unique-conversep: left-unique $A^{-1-1} \longleftrightarrow right-unique A$*

and *right-unique-conversep: right-unique $A^{-1-1} \longleftrightarrow left-unique A$*

by(*auto simp add: left-unique-def right-unique-def*)

lemma [*simp*]:

shows *left-total-conversep: left-total $A^{-1-1} \longleftrightarrow right-total A$*

and *right-total-conversep: right-total $A^{-1-1} \longleftrightarrow left-total A$*

by(*simp-all add: left-total-def right-total-def*)

lemma *bi-unique-conversep* [*simp*]: *bi-unique $R^{-1-1} = bi-unique R$*

by(*auto simp add: bi-unique-def*)

lemma *bi-total-conversep* [*simp*]: *bi-total $R^{-1-1} = bi-total R$*

by(*auto simp add: bi-total-def*)

lemma *right-unique-alt-def: right-unique $R = (conversep R \circ\circ R \leq (=))$ unfolding *right-unique-def* **by** *blast**

lemma *left-unique-alt-def: left-unique $R = (R \circ\circ (conversep R) \leq (=))$ unfolding *left-unique-def* **by** *blast**

lemma *right-total-alt-def: right-total $R = (conversep R \circ\circ R \geq (=))$ unfolding *right-total-def* **by** *blast**

lemma *left-total-alt-def: left-total $R = (R \circ\circ conversep R \geq (=))$ unfolding *left-total-def* **by** *blast**

lemma *bi-total-alt-def: bi-total $A = (left-total A \wedge right-total A)$*

unfolding *left-total-def right-total-def bi-total-def* **by** *blast*

lemma *bi-unique-alt-def: bi-unique $A = (left-unique A \wedge right-unique A)$*

unfolding *left-unique-def right-unique-def bi-unique-def* **by** *blast*

lemma *bi-totalI*: $\text{left-total } R \implies \text{right-total } R \implies \text{bi-total } R$
unfolding *bi-total-alt-def* ..

lemma *bi-uniqueI*: $\text{left-unique } R \implies \text{right-unique } R \implies \text{bi-unique } R$
unfolding *bi-unique-alt-def* ..

end

lemma *is-equality-lemma*: $(\bigwedge R. \text{is-equality } R \implies \text{PROP } (P R)) \equiv \text{PROP } (P (=))$
unfolding *is-equality-def*
proof (*rule equal-intr-rule*)
 show $(\bigwedge R. R = (=) \implies \text{PROP } P R) \implies \text{PROP } P (=)$
 apply (*drule meta-spec*)
 apply (*erule meta-mp [OF - refl]*)
 done
qed *simp*

lemma *Domainp-lemma*: $(\bigwedge R. \text{Domainp } T = R \implies \text{PROP } (P R)) \equiv \text{PROP } (P (\text{Domainp } T))$
proof (*rule equal-intr-rule*)
 show $(\bigwedge R. \text{Domainp } T = R \implies \text{PROP } P R) \implies \text{PROP } P (\text{Domainp } T)$
 apply (*drule meta-spec*)
 apply (*erule meta-mp [OF - refl]*)
 done
qed *simp*

ML-file $\langle \text{Tools/Transfer/transfer.ML} \rangle$
declare *refl* [*transfer-rule*]

hide-const (**open**) *Rel*

context **includes** *lifting-syntax*
begin

Handling of domains

lemma *Domainp-iff*: $\text{Domainp } T x \longleftrightarrow (\exists y. T x y)$
 by *auto*

lemma *Domainp-refl*[*transfer-domain-rule*]:
 $\text{Domainp } T = \text{Domainp } T$..

lemma *Domain-eq-top*[*transfer-domain-rule*]: $\text{Domainp } (=) = \text{top}$ **by** *auto*

lemma *Domainp-pred-fun-eq*[*relator-domain*]:
 assumes *left-unique T*
 shows $\text{Domainp } (T \implies S) = \text{pred-fun } (\text{Domainp } T) (\text{Domainp } S)$ (**is** *?lhs*
 = *?rhs*)

```

proof (intro ext iffI)
  fix x
  assume ?lhs x
  then show ?rhs x
    using assms unfolding rel-fun-def pred-fun-def by blast
next
  fix x
  assume ?rhs x
  then have  $\exists g. \forall y xa. T xa y \longrightarrow S (x xa) (g y)$ 
    using assms unfolding Domainp-iff left-unique-def pred-fun-def
    by (intro choice) blast
  then show ?lhs x
    by blast
qed

```

Properties are preserved by relation composition.

lemma *OO-def*: $R OO S = (\lambda x z. \exists y. R x y \wedge S y z)$
by *auto*

lemma *bi-total-OO*: $\llbracket bi\text{-total } A; bi\text{-total } B \rrbracket \Longrightarrow bi\text{-total } (A OO B)$
unfolding *bi-total-def OO-def* **by** *fast*

lemma *bi-unique-OO*: $\llbracket bi\text{-unique } A; bi\text{-unique } B \rrbracket \Longrightarrow bi\text{-unique } (A OO B)$
unfolding *bi-unique-def OO-def* **by** *blast*

lemma *right-total-OO*:
 $\llbracket right\text{-total } A; right\text{-total } B \rrbracket \Longrightarrow right\text{-total } (A OO B)$
unfolding *right-total-def OO-def* **by** *fast*

lemma *right-unique-OO*:
 $\llbracket right\text{-unique } A; right\text{-unique } B \rrbracket \Longrightarrow right\text{-unique } (A OO B)$
unfolding *right-unique-def OO-def* **by** *fast*

lemma *left-total-OO*: $left\text{-total } R \Longrightarrow left\text{-total } S \Longrightarrow left\text{-total } (R OO S)$
unfolding *left-total-def OO-def* **by** *fast*

lemma *left-unique-OO*: $left\text{-unique } R \Longrightarrow left\text{-unique } S \Longrightarrow left\text{-unique } (R OO S)$
unfolding *left-unique-def OO-def* **by** *blast*

42.4 Properties of relators

lemma *left-total-eq[transfer-rule]*: $left\text{-total } (=)$
unfolding *left-total-def* **by** *blast*

lemma *left-unique-eq[transfer-rule]*: $left\text{-unique } (=)$
unfolding *left-unique-def* **by** *blast*

lemma *right-total-eq [transfer-rule]*: $right\text{-total } (=)$
unfolding *right-total-def* **by** *simp*

```

lemma right-unique-eq [transfer-rule]: right-unique (=)
  unfolding right-unique-def by simp

lemma bi-total-eq[transfer-rule]: bi-total (=)
  unfolding bi-total-def by simp

lemma bi-unique-eq[transfer-rule]: bi-unique (=)
  unfolding bi-unique-def by simp

lemma left-total-fun[transfer-rule]:
  assumes left-unique A left-total B
  shows left-total (A ===> B)
  unfolding left-total-def
proof
  fix f
  show Ex ((A ===> B) f)
    unfolding rel-fun-def
    proof (intro exI strip)
      fix x y
      assume A: A x y
      have (THE x. A x y) = x
        using A assms by (simp add: left-unique-def the-equality)
      then show B (f x) (SOME z. B (f (THE x. A x y)) z)
        using assms by (force simp: left-total-def intro: someI-ex)
    qed
  qed

lemma left-unique-fun[transfer-rule]:
  [left-total A; left-unique B]  $\implies$  left-unique (A ===> B)
  unfolding left-total-def left-unique-def rel-fun-def
  by (clarify, rule ext, fast)

lemma right-total-fun [transfer-rule]:
  assumes right-unique A right-total B
  shows right-total (A ===> B)
  unfolding right-total-def
proof
  fix g
  show  $\exists x. (A ===> B) x g$ 
    unfolding rel-fun-def
    proof (intro exI strip)
      fix x y
      assume A: A x y
      have (THE y. A x y) = y
        using A assms by (simp add: right-unique-def the-equality)
      then show B (SOME z. B z (g (THE y. A x y))) (g y)
        using assms by (force simp: right-total-def intro: someI-ex)
    qed
  qed

```


qed

lemma *right-unique-fun* [*transfer-rule*]:

[[*right-total A*; *right-unique B*]] \implies *right-unique* ($A \implies B$)

unfolding *right-total-def right-unique-def rel-fun-def*

by (*clarify, rule ext, fast*)

lemma *bi-total-fun*[*transfer-rule*]:

[[*bi-unique A*; *bi-total B*]] \implies *bi-total* ($A \implies B$)

unfolding *bi-unique-alt-def bi-total-alt-def*

by (*blast intro: right-total-fun left-total-fun*)

lemma *bi-unique-fun*[*transfer-rule*]:

[[*bi-total A*; *bi-unique B*]] \implies *bi-unique* ($A \implies B$)

unfolding *bi-unique-alt-def bi-total-alt-def*

by (*blast intro: right-unique-fun left-unique-fun*)

end

lemma *if-conn*:

(*if P* \wedge *Q* *then t else e*) = (*if P* *then if Q* *then t else e else e*)

(*if P* \vee *Q* *then t else e*) = (*if P* *then t else if Q* *then t else e*)

(*if P* \longrightarrow *Q* *then t else e*) = (*if P* *then if Q* *then t else e else t*)

(*if* \neg *P* *then t else e*) = (*if P* *then e else t*)

by *auto*

ML-file \langle *Tools/Transfer/transfer-bnf.ML* \rangle

ML-file \langle *Tools/BNF/bnf-fp-rec-sugar-transfer.ML* \rangle

declare *pred-fun-def* [*simp*]

declare *rel-fun-eq* [*relator-eq*]

declare *fun.Domainp-rel*[*relator-domain del*]

42.5 Transfer rules

context includes *lifting-syntax*

begin

lemma *Domainp-forall-transfer* [*transfer-rule*]:

assumes *right-total A*

shows (($A \implies (=)$) \implies ($=$))

(*transfer-bforall* (*Domainp A*)) *transfer-forall*

using *assms* **unfolding** *right-total-def*

unfolding *transfer-forall-def transfer-bforall-def rel-fun-def Domainp-iff*

by *fast*

Transfer rules using implication instead of equality on booleans.

lemma *transfer-forall-transfer* [*transfer-rule*]:

bi-total $A \implies ((A \implies (=)) \implies (=))$ *transfer-forall transfer-forall*
right-total $A \implies ((A \implies (=)) \implies \text{implies})$ *transfer-forall transfer-forall*
right-total $A \implies ((A \implies \text{implies}) \implies \text{implies})$ *transfer-forall transfer-forall*
bi-total $A \implies ((A \implies (=)) \implies \text{rev-implies})$ *transfer-forall transfer-forall*
bi-total $A \implies ((A \implies \text{rev-implies}) \implies \text{rev-implies})$ *transfer-forall transfer-forall*
unfolding *transfer-forall-def rev-implies-def rel-fun-def right-total-def bi-total-def*
by *fast+*

lemma *transfer-implies-transfer* [*transfer-rule*]:

$((=) \implies (=) \implies (=))$ *transfer-implies transfer-implies*
 $(\text{rev-implies} \implies \text{implies} \implies \text{implies})$ *transfer-implies transfer-implies*
 $(\text{rev-implies} \implies (=) \implies \text{implies})$ *transfer-implies transfer-implies*
 $((=) \implies \text{implies} \implies \text{implies})$ *transfer-implies transfer-implies*
 $((=) \implies (=) \implies \text{implies})$ *transfer-implies transfer-implies*
 $(\text{implies} \implies \text{rev-implies} \implies \text{rev-implies})$ *transfer-implies transfer-implies*
 $(\text{implies} \implies (=) \implies \text{rev-implies})$ *transfer-implies transfer-implies*
 $((=) \implies \text{rev-implies} \implies \text{rev-implies})$ *transfer-implies transfer-implies*
 $((=) \implies (=) \implies \text{rev-implies})$ *transfer-implies transfer-implies*
unfolding *transfer-implies-def rev-implies-def rel-fun-def* **by** *auto*

lemma *eq-imp-transfer* [*transfer-rule*]:

right-unique $A \implies (A \implies A \implies (\longrightarrow)) (=) (=)$
unfolding *right-unique-alt-def2* .

Transfer rules using equality.

lemma *left-unique-transfer* [*transfer-rule*]:

assumes *right-total A*
assumes *right-total B*
assumes *bi-unique A*
shows $((A \implies B \implies (=)) \implies \text{implies})$ *left-unique left-unique*
using *assms* **unfolding** *left-unique-def right-total-def bi-unique-def rel-fun-def*
by *metis*

lemma *eq-transfer* [*transfer-rule*]:

assumes *bi-unique A*
shows $(A \implies A \implies (=)) (=) (=)$
using *assms* **unfolding** *bi-unique-def rel-fun-def* **by** *auto*

lemma *right-total-Ex-transfer*[*transfer-rule*]:

assumes *right-total A*
shows $((A \implies (=)) \implies (=)) (Bex (\text{Collect} (\text{Domainp} A))) Ex$
using *assms* **unfolding** *right-total-def Bex-def rel-fun-def Domainp-iff*
by *fast*

lemma *right-total-All-transfer*[*transfer-rule*]:

assumes *right-total A*

shows $((A \text{====>} (=)) \text{====>} (=)) (Ball (Collect (Domainp A))) All$
using *assms* **unfolding** *right-total-def Ball-def rel-fun-def Domainp-iff*
by *fast*

context

includes *lifting-syntax*

begin

lemma *right-total-fun-eq-transfer*:

assumes [*transfer-rule*]: *right-total A bi-unique B*

shows $((A \text{====>} B) \text{====>} (A \text{====>} B) \text{====>} (=)) (\lambda f g. \forall x \in Collect(Domainp A). f x = g x) (=)$

unfolding *fun-eq-iff*

by *transfer-prover*

end

lemma *All-transfer* [*transfer-rule*]:

assumes *bi-total A*

shows $((A \text{====>} (=)) \text{====>} (=)) All All$

using *assms* **unfolding** *bi-total-def rel-fun-def* **by** *fast*

lemma *Ex-transfer* [*transfer-rule*]:

assumes *bi-total A*

shows $((A \text{====>} (=)) \text{====>} (=)) Ex Ex$

using *assms* **unfolding** *bi-total-def rel-fun-def* **by** *fast*

lemma *Ex1-parametric* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique A bi-total A*

shows $((A \text{====>} (=)) \text{====>} (=)) Ex1 Ex1$

unfolding *Ex1-def* **by** *transfer-prover*

declare *If-transfer* [*transfer-rule*]

lemma *Let-transfer* [*transfer-rule*]: $(A \text{====>} (A \text{====>} B) \text{====>} B) Let Let$

unfolding *rel-fun-def* **by** *simp*

declare *id-transfer* [*transfer-rule*]

declare *comp-transfer* [*transfer-rule*]

lemma *curry-transfer* [*transfer-rule*]:

$((rel\text{-}prod A B \text{====>} C) \text{====>} A \text{====>} B \text{====>} C) curry curry$

unfolding *curry-def* **by** *transfer-prover*

lemma *fun-upd-transfer* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique A*

shows $((A \text{====>} B) \text{====>} A \text{====>} B \text{====>} A \text{====>} B) fun\text{-}upd fun\text{-}upd$

unfolding *fun-upd-def* **by** *transfer-prover*

lemma *case-nat-transfer* [*transfer-rule*]:
 $(A \text{ ==== } \Rightarrow ((=) \text{ ==== } \Rightarrow A) \text{ ==== } \Rightarrow (=) \text{ ==== } \Rightarrow A)$ *case-nat case-nat*
unfolding *rel-fun-def* **by** (*simp split: nat.split*)

lemma *rec-nat-transfer* [*transfer-rule*]:
 $(A \text{ ==== } \Rightarrow ((=) \text{ ==== } \Rightarrow A \text{ ==== } \Rightarrow A) \text{ ==== } \Rightarrow (=) \text{ ==== } \Rightarrow A)$ *rec-nat rec-nat*
unfolding *rel-fun-def*
apply *safe*
subgoal for - - - - *n*
by (*induction n simp-all*)
done

lemma *funpow-transfer* [*transfer-rule*]:
 $((=) \text{ ==== } \Rightarrow (A \text{ ==== } \Rightarrow A) \text{ ==== } \Rightarrow (A \text{ ==== } \Rightarrow A))$ *compow compow*
unfolding *funpow-def* **by** *transfer-prover*

lemma *mono-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
assumes [*transfer-rule*]: $(A \text{ ==== } \Rightarrow A \text{ ==== } \Rightarrow (=)) (\leq) (\leq)$
assumes [*transfer-rule*]: $(B \text{ ==== } \Rightarrow B \text{ ==== } \Rightarrow (=)) (\leq) (\leq)$
shows $((A \text{ ==== } \Rightarrow B) \text{ ==== } \Rightarrow (=))$ *mono mono*
unfolding *mono-def* **by** *transfer-prover*

lemma *right-total-relcompp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *right-total B*
shows $((A \text{ ==== } \Rightarrow B \text{ ==== } \Rightarrow (=)) \text{ ==== } \Rightarrow (B \text{ ==== } \Rightarrow C \text{ ==== } \Rightarrow (=)) \text{ ==== } \Rightarrow A \text{ ==== } \Rightarrow C \text{ ==== } \Rightarrow (=))$
 $(\lambda R S x z. \exists y \in \text{Collect } (\text{Domainp } B). R x y \wedge S y z)$ (*OO*)
unfolding *OO-def* **by** *transfer-prover*

lemma *relcompp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total B*
shows $((A \text{ ==== } \Rightarrow B \text{ ==== } \Rightarrow (=)) \text{ ==== } \Rightarrow (B \text{ ==== } \Rightarrow C \text{ ==== } \Rightarrow (=)) \text{ ==== } \Rightarrow A \text{ ==== } \Rightarrow C \text{ ==== } \Rightarrow (=))$ (*OO*) (*OO*)
unfolding *OO-def* **by** *transfer-prover*

lemma *right-total-Domainp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *right-total B*
shows $((A \text{ ==== } \Rightarrow B \text{ ==== } \Rightarrow (=)) \text{ ==== } \Rightarrow A \text{ ==== } \Rightarrow (=)) (\lambda T x. \exists y \in \text{Collect}(\text{Domainp } B). T x y)$ *Domainp*
apply(*subst(2) Domainp-iff[abs-def]*) **by** *transfer-prover*

lemma *Domainp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total B*
shows $((A \text{ ==== } \Rightarrow B \text{ ==== } \Rightarrow (=)) \text{ ==== } \Rightarrow A \text{ ==== } \Rightarrow (=))$ *Domainp Domainp*
unfolding *Domainp-iff* **by** *transfer-prover*

lemma *reflp-transfer* [*transfer-rule*]:

bi-total $A \implies ((A \implies A \implies (=)) \implies (=)) \text{ reflp reflp}$

right-total $A \implies ((A \implies A \implies \text{implies}) \implies \text{implies}) \text{ reflp reflp}$

right-total $A \implies ((A \implies A \implies (=)) \implies \text{implies}) \text{ reflp reflp}$

bi-total $A \implies ((A \implies A \implies \text{rev-implies}) \implies \text{rev-implies}) \text{ reflp reflp}$

bi-total $A \implies ((A \implies A \implies (=)) \implies \text{rev-implies}) \text{ reflp reflp}$

unfolding *reflp-def rev-implies-def bi-total-def right-total-def rel-fun-def*

by *fast+*

lemma *right-unique-transfer* [*transfer-rule*]:

$\llbracket \text{right-total } A; \text{right-total } B; \text{bi-unique } B \rrbracket$

$\implies ((A \implies B \implies (=)) \implies \text{implies}) \text{ right-unique right-unique}$

unfolding *right-unique-def right-total-def bi-unique-def rel-fun-def*

by *metis*

lemma *left-total-parametric* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-total* A *bi-total* B

shows $((A \implies B \implies (=)) \implies (=)) \text{ left-total left-total}$

unfolding *left-total-def* **by** *transfer-prover*

lemma *right-total-parametric* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-total* A *bi-total* B

shows $((A \implies B \implies (=)) \implies (=)) \text{ right-total right-total}$

unfolding *right-total-def* **by** *transfer-prover*

lemma *left-unique-parametric* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique* A *bi-total* A *bi-total* B

shows $((A \implies B \implies (=)) \implies (=)) \text{ left-unique left-unique}$

unfolding *left-unique-def* **by** *transfer-prover*

lemma *prod-pred-parametric* [*transfer-rule*]:

$((A \implies (=)) \implies (B \implies (=)) \implies \text{rel-prod } A \ B \implies (=))$

pred-prod pred-prod

unfolding *prod.pred-set Basic-BNFs.fsts-def Basic-BNFs.snds-def fstsp.simps sndsp.simps*

by *simp transfer-prover*

lemma *apfst-parametric* [*transfer-rule*]:

$((A \implies B) \implies \text{rel-prod } A \ C \implies \text{rel-prod } B \ C) \text{ apfst apfst}$

unfolding *apfst-def* **by** *transfer-prover*

lemma *rel-fun-eq-eq-onp*: $((=) \implies \text{eq-onp } P) = \text{eq-onp } (\lambda f. \forall x. P(f x))$

unfolding *eq-onp-def rel-fun-def* **by** *auto*

lemma *rel-fun-eq-onp-rel*:

shows $((\text{eq-onp } R) \implies S) = (\lambda f g. \forall x. R x \longrightarrow S (f x) (g x))$

by (*auto simp add: eq-onp-def rel-fun-def*)

lemma *eq-onp-transfer* [*transfer-rule*]:

assumes [transfer-rule]: *bi-unique A*
 shows $((A \text{====>} (=)) \text{====>} A \text{====>} A \text{====>} (=))$ *eq-onp eq-onp*
 unfolding *eq-onp-def* by *transfer-prover*

lemma *rtranclp-parametric* [transfer-rule]:

assumes *bi-unique A bi-total A*
 shows $((A \text{====>} A \text{====>} (=)) \text{====>} A \text{====>} A \text{====>} (=))$ *rtranclp*
rtranclp

proof(*rule rel-funI iffI*)+

fix *R* :: 'a \Rightarrow 'a \Rightarrow bool and *R'* *x y x' y'*

assume *R*: $(A \text{====>} A \text{====>} (=))$ *R R'* and *A x x'*

{

 assume *R** x y A y y'*

 thus *R** x' y'*

 proof(*induction arbitrary: y'*)

 case *base*

 with \langle *bi-unique A* \rangle \langle *A x x'* \rangle have *x' = y'* by(*rule bi-uniqueDr*)

 thus ?*case* by *simp*

 next

 case $($ *step y z z'* $)$

 from \langle *bi-total A* \rangle obtain *y'* where *A y y'* unfolding *bi-total-def* by *blast*

 hence *R** x' y'* by(*rule step.IH*)

 moreover from *R* \langle *A y y'* \rangle \langle *A z z'* \rangle \langle *R y z* \rangle

 have *R' y' z'* by(*auto dest: rel-funD*)

 ultimately show ?*case* ..

 qed

next

 assume *R** x' y' A y y'*

 thus *R** x y*

 proof(*induction arbitrary: y*)

 case *base*

 with \langle *bi-unique A* \rangle \langle *A x x'* \rangle have *x = y* by(*rule bi-uniqueDl*)

 thus ?*case* by *simp*

 next

 case $($ *step y' z' z* $)$

 from \langle *bi-total A* \rangle obtain *y* where *A y y'* unfolding *bi-total-def* by *blast*

 hence *R** x y* by(*rule step.IH*)

 moreover from *R* \langle *A y y'* \rangle \langle *A z z'* \rangle \langle *R' y' z'* \rangle

 have *R y z* by(*auto dest: rel-funD*)

 ultimately show ?*case* ..

 qed

}

qed

lemma *right-unique-parametric* [transfer-rule]:

assumes [transfer-rule]: *bi-total A bi-unique B bi-total B*

shows $((A \text{====>} B \text{====>} (=)) \text{====>} (=))$ *right-unique right-unique*

unfolding *right-unique-def* by *transfer-prover*

```

lemma map-fun-parametric [transfer-rule]:
  ((A ==> B) ==> (C ==> D) ==> (B ==> C) ==> A ==>
  D) map-fun map-fun
  unfolding map-fun-def by transfer-prover

end

```

42.6 *of-bool* and *of-nat*

```

context
  includes lifting-syntax
begin

```

```

lemma transfer-rule-of-bool:
  ⟨((⟨←→⟩ ==> (≅)) of-bool of-bool)
  if [transfer-rule]: ⟨0 ≅ 0⟩ ⟨1 ≅ 1⟩
  for R :: ⟨'a::zero-neq-one ⇒ 'b::zero-neq-one ⇒ bool⟩ (infix ⟨≅⟩ 50)
  unfolding of-bool-def by transfer-prover

```

```

lemma transfer-rule-of-nat:
  ((=) ==> (≅)) of-nat of-nat
  if [transfer-rule]: ⟨0 ≅ 0⟩ ⟨1 ≅ 1⟩
  ⟨((≅) ==> (≅)) ==> (≅)⟩ (+) (+)
  for R :: ⟨'a::semiring-1 ⇒ 'b::semiring-1 ⇒ bool⟩ (infix ⟨≅⟩ 50)
  unfolding of-nat-def by transfer-prover

```

end

end

43 Lifting package

```

theory Lifting
imports Equiv-Relations Transfer
keywords
  parametric and
  print-quot-maps print-quotients :: diag and
  lift-definition :: thy-goal-defn and
  setup-lifting lifting-forget lifting-update :: thy-decl
begin

```

43.1 Function map

```

context includes lifting-syntax
begin

```

```

lemma map-fun-id:
  (id ---> id) = id
  by (simp add: fun-eq-iff)

```

43.2 Quotient Predicate

definition

Quotient R *Abs* *Rep* $T \longleftrightarrow$
 $(\forall a. \text{Abs } (\text{Rep } a) = a) \wedge$
 $(\forall a. R (\text{Rep } a) (\text{Rep } a)) \wedge$
 $(\forall r s. R r s \longleftrightarrow R r r \wedge R s s \wedge \text{Abs } r = \text{Abs } s) \wedge$
 $T = (\lambda x y. R x x \wedge \text{Abs } x = y)$

lemma *QuotientI*:

assumes $\bigwedge a. \text{Abs } (\text{Rep } a) = a$
and $\bigwedge a. R (\text{Rep } a) (\text{Rep } a)$
and $\bigwedge r s. R r s \longleftrightarrow R r r \wedge R s s \wedge \text{Abs } r = \text{Abs } s$
and $T = (\lambda x y. R x x \wedge \text{Abs } x = y)$
shows *Quotient* R *Abs* *Rep* T
using *assms* **unfolding** *Quotient-def* **by** *blast*

context

fixes R *Abs* *Rep* T
assumes a : *Quotient* R *Abs* *Rep* T

begin

lemma *Quotient-abs-rep*: $\text{Abs } (\text{Rep } a) = a$
using a **unfolding** *Quotient-def*
by *simp*

lemma *Quotient-rep-reflp*: $R (\text{Rep } a) (\text{Rep } a)$
using a **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rel*:

$R r r \wedge R s s \wedge \text{Abs } r = \text{Abs } s \longleftrightarrow R r s$ — orientation does not loop on rewriting
using a **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-cr-rel*: $T = (\lambda x y. R x x \wedge \text{Abs } x = y)$
using a **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-refl1*: $R r s \Longrightarrow R r r$
using a **unfolding** *Quotient-def*
by *fast*

lemma *Quotient-refl2*: $R r s \Longrightarrow R s s$
using a **unfolding** *Quotient-def*
by *fast*

lemma *Quotient-rel-rep*: $R (\text{Rep } a) (\text{Rep } b) \longleftrightarrow a = b$
using a **unfolding** *Quotient-def*
by *metis*

lemma *Quotient-rep-abs*: $R\ r\ r \implies R\ (Rep\ (Abs\ r))\ r$
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rep-abs-eq*: $R\ t\ t \implies R\ \leq\ (=) \implies Rep\ (Abs\ t) = t$
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rep-abs-fold-unmap*:
assumes $x' \equiv Abs\ x$ **and** $R\ x\ x$ **and** $Rep\ x' \equiv Rep'\ x'$
shows $R\ (Rep'\ x')\ x$
proof –
have $R\ (Rep\ x')\ x$ **using** *assms(1–2)* *Quotient-rep-abs* **by** *auto*
then show *?thesis* **using** *assms(3)* **by** *simp*
qed

lemma *Quotient-Rep-eq*:
assumes $x' \equiv Abs\ x$
shows $Rep\ x' \equiv Rep\ x'$
by *simp*

lemma *Quotient-rel-abs*: $R\ r\ s \implies Abs\ r = Abs\ s$
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rel-abs2*:
assumes $R\ (Rep\ x)\ y$
shows $x = Abs\ y$
proof –
from *assms* **have** $Abs\ (Rep\ x) = Abs\ y$ **by** (*auto intro: Quotient-rel-abs*)
then show *?thesis* **using** *assms(1)* **by** (*simp add: Quotient-abs-rep*)
qed

lemma *Quotient-symp*: *symp* R
using *a* **unfolding** *Quotient-def* **using** *sympI* **by** (*metis (full-types)*)

lemma *Quotient-transp*: *transp* R
using *a* **unfolding** *Quotient-def* **using** *transpI* **by** (*metis (full-types)*)

lemma *Quotient-part-equivp*: *part-equivp* R
by (*metis Quotient-rep-reflp Quotient-symp Quotient-transp part-equivpI*)

end

lemma *identity-quotient*: *Quotient* $(=)$ *id id* $(=)$
unfolding *Quotient-def* **by** *simp*

TODO: Use one of these alternatives as the real definition.

lemma *Quotient-alt-def:*

$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
 $(\forall a\ b. T\ a\ b \longrightarrow Abs\ a = b) \wedge$
 $(\forall b. T\ (Rep\ b)\ b) \wedge$
 $(\forall x\ y. R\ x\ y \longleftrightarrow T\ x\ (Abs\ x) \wedge T\ y\ (Abs\ y) \wedge Abs\ x = Abs\ y)$

apply *safe*

apply (*simp* (*no-asm-use*) *only: Quotient-def, fast*)

apply (*simp* (*no-asm-use*) *only: Quotient-def, fast*)

apply (*simp* (*no-asm-use*) *only: Quotient-def, fast*)

apply (*simp* (*no-asm-use*) *only: Quotient-def, fast*)

apply (*simp* (*no-asm-use*) *only: Quotient-def, fast*)

apply (*simp* (*no-asm-use*) *only: Quotient-def, fast*)

apply (*rule QuotientI*)

apply *simp*

apply *metis*

apply *simp*

apply (*rule ext, rule ext, metis*)

done

lemma *Quotient-alt-def2:*

$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
 $(\forall a\ b. T\ a\ b \longrightarrow Abs\ a = b) \wedge$
 $(\forall b. T\ (Rep\ b)\ b) \wedge$
 $(\forall x\ y. R\ x\ y \longleftrightarrow T\ x\ (Abs\ y) \wedge T\ y\ (Abs\ x))$

unfolding *Quotient-alt-def* **by** (*safe, metis+*)

lemma *Quotient-alt-def3:*

$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
 $(\forall a\ b. T\ a\ b \longrightarrow Abs\ a = b) \wedge (\forall b. T\ (Rep\ b)\ b) \wedge$
 $(\forall x\ y. R\ x\ y \longleftrightarrow (\exists z. T\ x\ z \wedge T\ y\ z))$

unfolding *Quotient-alt-def2* **by** (*safe, metis+*)

lemma *Quotient-alt-def4:*

$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
 $(\forall a\ b. T\ a\ b \longrightarrow Abs\ a = b) \wedge (\forall b. T\ (Rep\ b)\ b) \wedge R = T\ OO\ conversesep\ T$

unfolding *Quotient-alt-def3* *fun-eq-iff* **by** *auto*

lemma *Quotient-alt-def5:*

$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
 $T \leq BNF-Def.Grp\ UNIV\ Abs \wedge BNF-Def.Grp\ UNIV\ Rep \leq T^{-1-1} \wedge R = T$
 $OO\ T^{-1-1}$

unfolding *Quotient-alt-def4* *Grp-def* **by** *blast*

lemma *fun-quotient:*

assumes *1: Quotient R1 abs1 rep1 T1*

assumes *2: Quotient R2 abs2 rep2 T2*

shows $(R1 \implies R2) (rep1 \dashrightarrow abs2) (abs1 \dashrightarrow rep2) (T1 \implies T2)$

using *assms* **unfolding** *Quotient-alt-def2*

unfolding *rel-fun-def fun-eq-iff map-fun-apply*
by (*safe, metis+*)

lemma *apply-rsp*:
fixes $f g :: 'a \Rightarrow 'c$
assumes $q: \text{Quotient } R1 \text{ Abs1 Rep1 } T1$
and $a: (R1 \implies R2) f g R1 x y$
shows $R2 (f x) (g y)$
using a **by** (*auto elim: rel-funE*)

lemma *apply-rsp'*:
assumes $a: (R1 \implies R2) f g R1 x y$
shows $R2 (f x) (g y)$
using a **by** (*auto elim: rel-funE*)

lemma *apply-rsp''*:
assumes $\text{Quotient } R \text{ Abs Rep } T$
and $(R \implies S) f f$
shows $S (f (\text{Rep } x)) (f (\text{Rep } x))$

proof –

from *assms(1)* **have** $R (\text{Rep } x) (\text{Rep } x)$ **by** (*rule Quotient-rep-reflp*)
then show *?thesis* **using** *assms(2)* **by** (*auto intro: apply-rsp'*)

qed

43.3 Quotient composition

lemma *Quotient-compose*:
assumes $1: \text{Quotient } R1 \text{ Abs1 Rep1 } T1$
assumes $2: \text{Quotient } R2 \text{ Abs2 Rep2 } T2$
shows $\text{Quotient } (T1 \text{ OO } R2 \text{ OO } \text{conversep } T1) (\text{Abs2} \circ \text{Abs1}) (\text{Rep1} \circ \text{Rep2})$
 $(T1 \text{ OO } T2)$
using *assms* **unfolding** *Quotient-alt-def4* **by** *fastforce*

lemma *equivp-reflp2*:
 $\text{equivp } R \implies \text{reflp } R$
by (*erule equivpE*)

43.4 Respects predicate

definition *Respects* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set}$
where $\text{Respects } R = \{x. R x x\}$

lemma *in-respects*: $x \in \text{Respects } R \longleftrightarrow R x x$
unfolding *Respects-def* **by** *simp*

lemma *UNIV-typedef-to-Quotient*:
assumes *type-definition Rep Abs UNIV*
and *T-def*: $T \equiv (\lambda x y. x = \text{Rep } y)$
shows $\text{Quotient } (=) \text{ Abs Rep } T$
proof –

```

interpret type-definition Rep Abs UNIV by fact
from Abs-inject Rep-inverse Abs-inverse T-def show ?thesis
  by (fastforce intro!: QuotientI fun-eq-iff)
qed

```

```

lemma UNIV-typedef-to-equivp:
  fixes Abs :: 'a  $\Rightarrow$  'b
  and Rep :: 'b  $\Rightarrow$  'a
  assumes type-definition Rep Abs (UNIV::'a set)
  shows equivp ((=) :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool)
by (rule identity-equivp)

```

```

lemma typedef-to-Quotient:
  assumes type-definition Rep Abs S
  and T-def: T  $\equiv$  ( $\lambda x y. x = Rep y$ )
  shows Quotient (eq-onp ( $\lambda x. x \in S$ )) Abs Rep T
proof –
  interpret type-definition Rep Abs S by fact
  from Rep Abs-inject Rep-inverse Abs-inverse T-def show ?thesis
  by (auto intro!: QuotientI simp: eq-onp-def fun-eq-iff)
qed

```

```

lemma typedef-to-part-equivp:
  assumes type-definition Rep Abs S
  shows part-equivp (eq-onp ( $\lambda x. x \in S$ ))
proof (intro part-equivpI)
  interpret type-definition Rep Abs S by fact
  show  $\exists x. eq-onp (\lambda x. x \in S) x x$  using Rep by (auto simp: eq-onp-def)
next
  show symp (eq-onp ( $\lambda x. x \in S$ )) by (auto intro: sympI simp: eq-onp-def)
next
  show transp (eq-onp ( $\lambda x. x \in S$ )) by (auto intro: transpI simp: eq-onp-def)
qed

```

```

lemma open-typedef-to-Quotient:
  assumes type-definition Rep Abs {x. P x}
  and T-def: T  $\equiv$  ( $\lambda x y. x = Rep y$ )
  shows Quotient (eq-onp P) Abs Rep T
  using typedef-to-Quotient [OF assms] by simp

```

```

lemma open-typedef-to-part-equivp:
  assumes type-definition Rep Abs {x. P x}
  shows part-equivp (eq-onp P)
  using typedef-to-part-equivp [OF assms] by simp

```

```

lemma type-definition-Quotient-not-empty: Quotient (eq-onp P) Abs Rep T  $\implies$ 
 $\exists x. P x$ 
unfolding eq-onp-def by (drule Quotient-rep-reflp) blast

```

lemma *type-definition-Quotient-not-empty-witness*: $Quotient (eq-onp P) Abs Rep$
 $T \implies P (Rep\ undefined)$

unfolding *eq-onp-def* **by** (*drule Quotient-rep-reflp*) *blast*

Generating transfer rules for quotients.

context

fixes $R Abs Rep T$

assumes 1: $Quotient R Abs Rep T$

begin

lemma *Quotient-right-unique: right-unique* T

using 1 **unfolding** *Quotient-alt-def right-unique-def* **by** *metis*

lemma *Quotient-right-total: right-total* T

using 1 **unfolding** *Quotient-alt-def right-total-def* **by** *metis*

lemma *Quotient-rel-eq-transfer: $(T \implies T \implies (=)) R (=)$*

using 1 **unfolding** *Quotient-alt-def rel-fun-def* **by** *simp*

lemma *Quotient-abs-induct*:

assumes $\bigwedge y. R y y \implies P (Abs y)$ **shows** $P x$

using 1 *assms* **unfolding** *Quotient-def* **by** *metis*

end

Generating transfer rules for total quotients.

context

fixes $R Abs Rep T$

assumes 1: $Quotient R Abs Rep T$ **and** 2: $reflp R$

begin

lemma *Quotient-left-total: left-total* T

using 1 2 **unfolding** *Quotient-alt-def left-total-def reflp-def* **by** *auto*

lemma *Quotient-bi-total: bi-total* T

using 1 2 **unfolding** *Quotient-alt-def bi-total-def reflp-def* **by** *auto*

lemma *Quotient-id-abs-transfer: $((=) \implies T) (\lambda x. x) Abs$*

using 1 2 **unfolding** *Quotient-alt-def reflp-def rel-fun-def* **by** *simp*

lemma *Quotient-total-abs-induct: $(\bigwedge y. P (Abs y)) \implies P x$*

using 1 2 **unfolding** *Quotient-alt-def reflp-def* **by** *metis*

lemma *Quotient-total-abs-eq-iff: $Abs x = Abs y \longleftrightarrow R x y$*

using *Quotient-rel [OF 1]* 2 **unfolding** *reflp-def* **by** *simp*

end

Generating transfer rules for a type defined with *typedef*.

```

context
  fixes Rep Abs A T
  assumes type: type-definition Rep Abs A
  assumes T-def:  $T \equiv (\lambda(x::'a) (y::'b). x = \text{Rep } y)$ 
begin

lemma typedef-left-unique: left-unique T
  unfolding left-unique-def T-def
  by (simp add: type-definition.Rep-inject [OF type])

lemma typedef-bi-unique: bi-unique T
  unfolding bi-unique-def T-def
  by (simp add: type-definition.Rep-inject [OF type])

lemma typedef-right-unique: right-unique T
  using T-def type Quotient-right-unique typedef-to-Quotient
  by blast

lemma typedef-right-total: right-total T
  using T-def type Quotient-right-total typedef-to-Quotient
  by blast

lemma typedef-rep-transfer:  $(T \implies (=)) (\lambda x. x) \text{Rep}$ 
  unfolding rel-fun-def T-def by simp

end

```

Generating the correspondence rule for a constant defined with *lift-definition*.

```

lemma Quotient-to-transfer:
  assumes Quotient R Abs Rep T and  $R \ c \ c$  and  $c' \equiv \text{Abs } c$ 
  shows  $T \ c \ c'$ 
  using assms by (auto dest: Quotient-cr-rel)

```

Proving reflexivity

```

lemma Quotient-to-left-total:
  assumes q: Quotient R Abs Rep T
  and r-R: reflp R
  shows left-total T
using r-R Quotient-cr-rel[OF q] unfolding left-total-def by (auto elim: reflpE)

```

```

lemma Quotient-composition-ge-eq:
  assumes left-total T
  assumes  $R \geq (=)$ 
  shows  $(T \circ\circ R \circ\circ T^{-1-1}) \geq (=)$ 
using assms unfolding left-total-def by fast

```

```

lemma Quotient-composition-le-eq:

```

assumes *left-unique T*
assumes $R \leq (=)$
shows $(T \text{ OO } R \text{ OO } T^{-1-1}) \leq (=)$
using *assms unfolding left-unique-def by blast*

lemma *eq-onp-le-eq:*
 $eq\text{-onp } P \leq (=)$ **unfolding** *eq-onp-def by blast*

lemma *reflp-ge-eq:*
 $reflp R \implies R \geq (=)$ **unfolding** *reflp-def by blast*

Proving a parametrized correspondence relation

definition $POS :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
 $POS A B \equiv A \leq B$

definition $NEG :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
 $NEG A B \equiv B \leq A$

lemma *pos-OO-eq:*
shows $POS (A \text{ OO } (=)) A$
unfolding *POS-def OO-def by blast*

lemma *pos-eq-OO:*
shows $POS ((=) \text{ OO } A) A$
unfolding *POS-def OO-def by blast*

lemma *neg-OO-eq:*
shows $NEG (A \text{ OO } (=)) A$
unfolding *NEG-def OO-def by auto*

lemma *neg-eq-OO:*
shows $NEG ((=) \text{ OO } A) A$
unfolding *NEG-def OO-def by blast*

lemma *POS-trans:*
assumes $POS A B$
assumes $POS B C$
shows $POS A C$
using *assms unfolding POS-def by auto*

lemma *NEG-trans:*
assumes $NEG A B$
assumes $NEG B C$
shows $NEG A C$
using *assms unfolding NEG-def by auto*

lemma *POS-NEG:*
 $POS A B \equiv NEG B A$
unfolding *POS-def NEG-def by auto*

lemma *NEG-POS*:

NEG $A B \equiv POS B A$

unfolding *POS-def NEG-def* **by** *auto*

lemma *POS-pcr-rule*:

assumes *POS* $(A OO B) C$

shows *POS* $(A OO B OO X) (C OO X)$

using *assms unfolding POS-def OO-def* **by** *blast*

lemma *NEG-pcr-rule*:

assumes *NEG* $(A OO B) C$

shows *NEG* $(A OO B OO X) (C OO X)$

using *assms unfolding NEG-def OO-def* **by** *blast*

lemma *POS-apply*:

assumes *POS* $R R'$

assumes $R f g$

shows $R' f g$

using *assms unfolding POS-def* **by** *auto*

Proving a parametrized correspondence relation

lemma *fun-mono*:

assumes $A \geq C$

assumes $B \leq D$

shows $(A \implies B) \leq (C \implies D)$

using *assms unfolding rel-fun-def* **by** *blast*

lemma *pos-fun-distr*: $((R \implies S) OO (R' \implies S')) \leq ((R OO R') \implies (S OO S'))$

unfolding *OO-def rel-fun-def* **by** *blast*

lemma *functional-relation*: $right\text{-}unique R \implies left\text{-}total R \implies \forall x. \exists!y. R x y$

unfolding *right-unique-def left-total-def* **by** *blast*

lemma *functional-converse-relation*: $left\text{-}unique R \implies right\text{-}total R \implies \forall y. \exists!x. R x y$

unfolding *left-unique-def right-total-def* **by** *blast*

lemma *neg-fun-distr1*:

assumes *1*: $left\text{-}unique R right\text{-}total R$

assumes *2*: $right\text{-}unique R' left\text{-}total R'$

shows $(R OO R' \implies S OO S') \leq ((R \implies S) OO (R' \implies S'))$

using *functional-relation[OF 2] functional-converse-relation[OF 1]*

unfolding *rel-fun-def OO-def*

apply *clarify*

apply *(subst all-comm)*

apply *(subst all-conj-distrib[symmetric])*

apply *(intro choice)*

by *metis*

lemma *neg-fun-distr2*:

assumes 1: *right-unique R' left-total R'*

assumes 2: *left-unique S' right-total S'*

shows $(R \text{ OO } R' \implies S \text{ OO } S') \leq ((R \implies S) \text{ OO } (R' \implies S'))$

using *functional-converse-relation[OF 2]* *functional-relation[OF 1]*

unfolding *rel-fun-def OO-def*

apply *clarify*

apply (*subst all-comm*)

apply (*subst all-conj-distrib[symmetric]*)

apply (*intro choice*)

by *metis*

43.5 Domains

lemma *composed-equiv-rel-eq-onp*:

assumes *left-unique R*

assumes $(R \implies (=)) P P'$

assumes *Domainp R = P''*

shows $(R \text{ OO } eq\text{-onp } P' \text{ OO } R^{-1-1}) = eq\text{-onp } (inf P'' P)$

using *assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def*

rel-fun-def eq-onp-def

fun-eq-iff by *blast*

lemma *composed-equiv-rel-eq-eq-onp*:

assumes *left-unique R*

assumes *Domainp R = P*

shows $(R \text{ OO } (=) \text{ OO } R^{-1-1}) = eq\text{-onp } P$

using *assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def*

eq-onp-def

fun-eq-iff is-equality-def by *metis*

lemma *pcr-Domainp-par-left-total*:

assumes *Domainp B = P*

assumes *left-total A*

assumes $(A \implies (=)) P' P$

shows *Domainp (A OO B) = P'*

using *assms*

unfolding *Domainp-iff[abs-def] OO-def bi-unique-def left-total-def rel-fun-def*

by (*fast intro: fun-eq-iff*)

lemma *pcr-Domainp-par*:

assumes *Domainp B = P2*

assumes *Domainp A = P1*

assumes $(A \implies (=)) P2' P2$

shows *Domainp (A OO B) = (inf P1 P2')*

using *assms unfolding rel-fun-def Domainp-iff[abs-def] OO-def*

by (*fast intro: fun-eq-iff*)

definition *rel-pred-comp* :: ('a => 'b => bool) => ('b => bool) => 'a => bool
where *rel-pred-comp* R P ≡ λx. ∃y. R x y ∧ P y

lemma *pcr-Domainp*:
assumes *Domainp* B = P
shows *Domainp* (A OO B) = (λx. ∃y. A x y ∧ P y)
using *assms* **by** *blast*

lemma *pcr-Domainp-total*:
assumes *left-total* B
assumes *Domainp* A = P
shows *Domainp* (A OO B) = P
using *assms* **unfolding** *left-total-def*
by *fast*

lemma *Quotient-to-Domainp*:
assumes *Quotient* R Abs Rep T
shows *Domainp* T = (λx. R x x)
by (*simp* *add*: *Domainp-iff*[*abs-def*] *Quotient-cr-rel*[*OF* *assms*])

lemma *eq-onp-to-Domainp*:
assumes *Quotient* (*eq-onp* P) Abs Rep T
shows *Domainp* T = P
by (*simp* *add*: *eq-onp-def* *Domainp-iff*[*abs-def*] *Quotient-cr-rel*[*OF* *assms*])

end

lemma *right-total-UNIV-transfer*:
assumes *right-total* A
shows (*rel-set* A) (Collect (*Domainp* A)) UNIV
using *assms* **unfolding** *right-total-def* *rel-set-def* *Domainp-iff* **by** *blast*

43.6 ML setup

ML-file <Tools/Lifting/lifting-util.ML>

named-theorems *relator-eq-onp*

theorems that a relator of an eq-onp is an eq-onp of the corresponding predicate

ML-file <Tools/Lifting/lifting-info.ML>

declare *fun-quotient*[*quot-map*]

declare *fun-mono*[*relator-mono*]

lemmas [*relator-distr*] = *pos-fun-distr* *neg-fun-distr1* *neg-fun-distr2*

ML-file <Tools/Lifting/lifting-bnf.ML>

ML-file <Tools/Lifting/lifting-term.ML>

ML-file \langle Tools/Lifting/lifting-def.ML \rangle
ML-file \langle Tools/Lifting/lifting-setup.ML \rangle
ML-file \langle Tools/Lifting/lifting-def-code-dt.ML \rangle

lemma *pred-prod-beta*: $\text{pred-prod } P \ Q \ xy \longleftrightarrow P \ (\text{fst } xy) \wedge Q \ (\text{snd } xy)$
by(*cases xy simp*)

lemma *pred-prod-split*: $P \ (\text{pred-prod } Q \ R \ xy) \longleftrightarrow (\forall x \ y. \ xy = (x, y) \longrightarrow P \ (Q \ x \wedge R \ y))$
by(*cases xy simp*)

hide-const (**open**) *POS NEG*

end

44 Definition of Quotient Types

theory *Quotient*

imports *Lifting*

keywords

print-quotmapsQ3 print-quotientsQ3 print-quotconsts :: diag and
quotient-type :: thy-goal-defn and / and
quotient-definition :: thy-goal-defn and
copy-bnf :: thy-defn and
lift-bnf :: thy-goal-defn

begin

Basic definition for equivalence relations that are represented by predicates.

Composition of Relations

abbreviation

rel-conj :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$ (**infixr** *OOO* 75)

where

$r1 \ OOO \ r2 \equiv r1 \ OO \ r2 \ OO \ r1$

lemma *eq-comp-r*:

shows $((=) \ OOO \ R) = R$

by (*auto simp add: fun-eq-iff*)

context **includes** *lifting-syntax*

begin

44.1 Quotient Predicate

definition

Quotient3 $R \ Abs \ Rep \longleftrightarrow$
 $(\forall a. \ Abs \ (Rep \ a) = a) \wedge (\forall a. \ R \ (Rep \ a) \ (Rep \ a)) \wedge$
 $(\forall r \ s. \ R \ r \ s \longleftrightarrow R \ r \ r \wedge R \ s \ s \wedge Abs \ r = Abs \ s)$

lemma *Quotient3I*:

assumes $\bigwedge a. \text{Abs } (\text{Rep } a) = a$
and $\bigwedge a. R (\text{Rep } a) (\text{Rep } a)$
and $\bigwedge r s. R r s \longleftrightarrow R r r \wedge R s s \wedge \text{Abs } r = \text{Abs } s$
shows *Quotient3* *R Abs Rep*
using *assms* **unfolding** *Quotient3-def* **by** *blast*

context

fixes *R Abs Rep*
assumes *a*: *Quotient3* *R Abs Rep*
begin

lemma *Quotient3-abs-rep*:

$\text{Abs } (\text{Rep } a) = a$
using *a*
unfolding *Quotient3-def*
by *simp*

lemma *Quotient3-rep-reflp*:

$R (\text{Rep } a) (\text{Rep } a)$
using *a*
unfolding *Quotient3-def*
by *blast*

lemma *Quotient3-rel*:

$R r r \wedge R s s \wedge \text{Abs } r = \text{Abs } s \longleftrightarrow R r s$ — orientation does not loop on rewriting
using *a*
unfolding *Quotient3-def*
by *blast*

lemma *Quotient3-refl1*:

$R r s \Longrightarrow R r r$
using *a* **unfolding** *Quotient3-def*
by *fast*

lemma *Quotient3-refl2*:

$R r s \Longrightarrow R s s$
using *a* **unfolding** *Quotient3-def*
by *fast*

lemma *Quotient3-rel-rep*:

$R (\text{Rep } a) (\text{Rep } b) \longleftrightarrow a = b$
using *a*
unfolding *Quotient3-def*
by *metis*

lemma *Quotient3-rep-abs*:

$R r r \Longrightarrow R (\text{Rep } (\text{Abs } r)) r$

using a unfolding *Quotient3-def*
by *blast*

lemma *Quotient3-rel-abs*:
 $R\ r\ s \implies Abs\ r = Abs\ s$
using a unfolding *Quotient3-def*
by *blast*

lemma *Quotient3-symp*:
 $symp\ R$
using a unfolding *Quotient3-def* **using** *sympI* **by** *metis*

lemma *Quotient3-transp*:
 $transp\ R$
using a unfolding *Quotient3-def* **using** *transpI* **by** (*metis (full-types)*)

lemma *Quotient3-part-equivp*:
 $part-equivp\ R$
by (*metis Quotient3-rep-reflp Quotient3-symp Quotient3-transp part-equivpI*)

lemma *abs-o-rep*:
 $Abs \circ Rep = id$
unfolding *fun-eq-iff*
by (*simp add: Quotient3-abs-rep*)

lemma *equals-rsp*:
assumes $b: R\ xa\ xb\ R\ ya\ yb$
shows $R\ xa\ ya = R\ xb\ yb$
using b *Quotient3-symp Quotient3-transp*
by (*blast elim: sympE transpE*)

lemma *rep-abs-rsp*:
assumes $b: R\ x1\ x2$
shows $R\ x1\ (Rep\ (Abs\ x2))$
using b *Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp*
by *metis*

lemma *rep-abs-rsp-left*:
assumes $b: R\ x1\ x2$
shows $R\ (Rep\ (Abs\ x1))\ x2$
using b *Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp*
by *metis*

end

lemma *identity-quotient3*:
 $Quotient3\ (=)\ id\ id$
unfolding *Quotient3-def id-def*
by *blast*

lemma fun-quotient3:
assumes $q1: \text{Quotient3 } R1 \text{ abs1 rep1}$
and $q2: \text{Quotient3 } R2 \text{ abs2 rep2}$
shows $\text{Quotient3 } (R1 \implies R2) (\text{rep1} \dashrightarrow \text{abs2}) (\text{abs1} \dashrightarrow \text{rep2})$
proof –
have $(\text{rep1} \dashrightarrow \text{abs2}) ((\text{abs1} \dashrightarrow \text{rep2}) a) = a$ **for** a
using $q1 \ q2$ **by** (*simp add: Quotient3-def fun-eq-iff*)
moreover
have $(R1 \implies R2) ((\text{abs1} \dashrightarrow \text{rep2}) a) ((\text{abs1} \dashrightarrow \text{rep2}) a)$ **for** a
by (*rule rel-funI*)
(use $q1 \ q2$ Quotient3-rel-abs [of $R1 \text{ abs1 rep1}$] Quotient3-rel-rep [of $R2 \text{ abs2 rep2}$])
in *(simp (no-asm) add: Quotient3-def, simp)*
moreover
have $(R1 \implies R2) r s = ((R1 \implies R2) r r \wedge (R1 \implies R2) s s \wedge$
 $(\text{rep1} \dashrightarrow \text{abs2}) r = (\text{rep1} \dashrightarrow \text{abs2}) s)$ **for** $r \ s$
proof –
have $(R1 \implies R2) r s \implies (R1 \implies R2) r r$ **unfolding** *rel-fun-def*
using *Quotient3-part-equivp[OF $q1$] Quotient3-part-equivp[OF $q2$]*
by (*metis (full-types) part-equivp-def*)
moreover have $(R1 \implies R2) r s \implies (R1 \implies R2) s s$ **unfolding**
rel-fun-def
using *Quotient3-part-equivp[OF $q1$] Quotient3-part-equivp[OF $q2$]*
by (*metis (full-types) part-equivp-def*)
moreover have $(R1 \implies R2) r s \implies (\text{rep1} \dashrightarrow \text{abs2}) r = (\text{rep1} \dashrightarrow$
 $\text{abs2}) s$
by (*auto simp add: rel-fun-def fun-eq-iff*)
(use $q1 \ q2$ in (unfold Quotient3-def, metis))
moreover have $((R1 \implies R2) r r \wedge (R1 \implies R2) s s \wedge$
 $(\text{rep1} \dashrightarrow \text{abs2}) r = (\text{rep1} \dashrightarrow \text{abs2}) s) \implies (R1 \implies R2) r s$
by (*auto simp add: rel-fun-def fun-eq-iff*)
(use $q1 \ q2$ in (unfold Quotient3-def, metis map-fun-apply))
ultimately show *?thesis* **by** *blast*
qed
ultimately show *?thesis* **by** (*intro Quotient3I (assumption+)*)
qed

lemma lambda-prs:
assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$
and $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$
shows $(\text{Rep1} \dashrightarrow \text{Abs2}) (\lambda x. \text{Rep2 } (f (\text{Abs1 } x))) = (\lambda x. f x)$
unfolding *fun-eq-iff*
using *Quotient3-abs-rep[OF $q1$] Quotient3-abs-rep[OF $q2$]*
by *simp*

lemma lambda-prs1:
assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$
and $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $(Rep1 \dashrightarrow Abs2) (\lambda x. (Abs1 \dashrightarrow Rep2) f x) = (\lambda x. f x)$
unfolding *fun-eq-iff*
using *Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]*
by *simp*

In the following theorem R1 can be instantiated with anything, but we know some of the types of the Rep and Abs functions; so by solving Quotient assumptions we can get a unique R1 that will be provable; which is why we need to use *apply-rsp* and not the primed version

lemma *apply-rspQ3*:
fixes $f g :: 'a \Rightarrow 'c$
assumes $q: Quotient3\ R1\ Abs1\ Rep1$
and $a: (R1 \implies R2) f g\ R1\ x\ y$
shows $R2 (f x) (g y)$
using a **by** *(auto elim: rel-funE)*

lemma *apply-rspQ3''*:
assumes $Quotient3\ R\ Abs\ Rep$
and $(R \implies S) f f$
shows $S (f (Rep x)) (f (Rep x))$

proof –

from *assms(1)* **have** $R (Rep x) (Rep x)$ **by** *(rule Quotient3-rep-reflp)*
then show *?thesis* **using** *assms(2)* **by** *(auto intro: apply-rsp')*

qed

44.2 lemmas for regularisation of ball and bex

lemma *ball-reg-eqv*:
fixes $P :: 'a \Rightarrow bool$
assumes $a: equivp\ R$
shows $Ball (Respects\ R) P = (All\ P)$
using a
unfolding *equivp-def*
by *(auto simp add: in-respects)*

lemma *bex-reg-eqv*:
fixes $P :: 'a \Rightarrow bool$
assumes $a: equivp\ R$
shows $Bex (Respects\ R) P = (Ex\ P)$
using a
unfolding *equivp-def*
by *(auto simp add: in-respects)*

lemma *ball-reg-right*:
assumes $a: \bigwedge x. x \in R \implies P x \longrightarrow Q x$
shows $All\ P \longrightarrow Ball\ R\ Q$
using a **by** *fast*

lemma *bex-reg-left*:

assumes a : $\bigwedge x. x \in R \implies Q x \longrightarrow P x$
shows $Bex R Q \longrightarrow Ex P$
using a **by** *fast*

lemma *ball-reg-left*:

assumes a : *equivp* R
shows $(\bigwedge x. (Q x \longrightarrow P x)) \implies Ball (Respects R) Q \longrightarrow All P$
using a **by** (*metis equivp-reflp in-respects*)

lemma *bex-reg-right*:

assumes a : *equivp* R
shows $(\bigwedge x. (Q x \longrightarrow P x)) \implies Ex Q \longrightarrow Bex (Respects R) P$
using a **by** (*metis equivp-reflp in-respects*)

lemma *ball-reg-equiv-range*:

fixes $P::'a \Rightarrow bool$
and $x::'a$
assumes a : *equivp* $R2$
shows $(Ball (Respects (R1 ==> R2)) (\lambda f. P (f x))) = All (\lambda f. P (f x))$
proof (*intro allI iffI*)
fix f
assume $\forall f \in Respects (R1 ==> R2). P (f x)$
moreover have $(\lambda y. f x) \in Respects (R1 ==> R2)$
using a *equivp-reflp-symp-transp*[of $R2$]
by (*auto simp add: in-respects rel-fun-def elim: equivpE refltE*)
ultimately show $P (f x)$
by *auto*
qed *auto*

lemma *bex-reg-equiv-range*:

assumes a : *equivp* $R2$
shows $(Bex (Respects (R1 ==> R2)) (\lambda f. P (f x))) = Ex (\lambda f. P (f x))$
proof –
{ **fix** f
assume $P (f x)$
have $(\lambda y. f x) \in Respects (R1 ==> R2)$
using a *equivp-reflp-symp-transp*[of $R2$]
by (*auto simp add: Respects-def in-respects rel-fun-def elim: equivpE refltE*)
}
then show *?thesis*
by *auto*
qed

lemma *all-reg*:

assumes a : $\forall x :: 'a. (P x \longrightarrow Q x)$
and b : *All* P
shows *All* Q
using a b **by** *fast*

lemma *ex-reg*:

assumes $a: \forall x :: 'a. (P\ x \longrightarrow Q\ x)$
and $b: Ex\ P$
shows $Ex\ Q$
using $a\ b$ **by** *fast*

lemma *ball-reg*:

assumes $a: \forall x :: 'a. (x \in R \longrightarrow P\ x \longrightarrow Q\ x)$
and $b: Ball\ R\ P$
shows $Ball\ R\ Q$
using $a\ b$ **by** *fast*

lemma *bex-reg*:

assumes $a: \forall x :: 'a. (x \in R \longrightarrow P\ x \longrightarrow Q\ x)$
and $b: Bex\ R\ P$
shows $Bex\ R\ Q$
using $a\ b$ **by** *fast*

lemma *ball-all-comm*:

assumes $\bigwedge y. (\forall x \in P. A\ x\ y) \longrightarrow (\forall x. B\ x\ y)$
shows $(\forall x \in P. \forall y. A\ x\ y) \longrightarrow (\forall x. \forall y. B\ x\ y)$
using *assms* **by** *auto*

lemma *bex-ex-comm*:

assumes $(\exists y. \exists x. A\ x\ y) \longrightarrow (\exists y. \exists x \in P. B\ x\ y)$
shows $(\exists x. \exists y. A\ x\ y) \longrightarrow (\exists x \in P. \exists y. B\ x\ y)$
using *assms* **by** *auto*

44.3 Bounded abstraction

definition

$Babs :: 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

where

$x \in p \Longrightarrow Babs\ p\ m\ x = m\ x$

lemma *babs-rsp*:

assumes $q: Quotient3\ R1\ Abs1\ Rep1$
and $a: (R1 \Longrightarrow R2)\ f\ g$
shows $(R1 \Longrightarrow R2)\ (Babs\ (Respects\ R1)\ f)\ (Babs\ (Respects\ R1)\ g)$

proof

fix $x\ y$

assume $R1\ x\ y$

then have $x \in Respects\ R1 \wedge y \in Respects\ R1$

unfolding *in-respects rel-fun-def* **using** *Quotient3-rel[OF q]* **by** *metis*

then show $R2\ (Babs\ (Respects\ R1)\ f\ x)\ (Babs\ (Respects\ R1)\ g\ y)$

using $\langle R1\ x\ y \rangle a$ **by** (*simp add: Babs-def rel-fun-def*)

qed

lemma *babs-prs*:
assumes $q1$: *Quotient3* $R1$ $Abs1$ $Rep1$
and $q2$: *Quotient3* $R2$ $Abs2$ $Rep2$
shows $((Rep1 \text{ ----> } Abs2) (Babs (Respects R1) ((Abs1 \text{ ----> } Rep2) f))) = f$
proof –
{ **fix** x
have $Rep1\ x \in Respects\ R1$
by (*simp add: in-respects Quotient3-rel-rep[OF q1]*)
then have $Abs2 (Babs (Respects R1) ((Abs1 \text{ ----> } Rep2) f) (Rep1\ x)) = f$
 x
by (*simp add: Babs-def Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]*)
}
then show *?thesis*
by force
qed

lemma *babs-simp*:
assumes q : *Quotient3* $R1$ Abs Rep
shows $((R1 \text{ ===> } R2) (Babs (Respects R1) f) (Babs (Respects R1) g)) = ((R1 \text{ ===> } R2) f g)$
(is ?lhs = ?rhs)
proof
assume *?lhs*
then show *?rhs*
unfolding *rel-fun-def* **by** (*metis Babs-def in-respects Quotient3-rel[OF q]*)
qed (*simp add: babs-rsp[OF q]*)

If a user proves that a particular functional relation is an equivalence, this may be useful in regularising

lemma *babs-reg-eqv*:
shows $equivp\ R \implies Babs (Respects\ R)\ P = P$
by (*simp add: fun-eq-iff Babs-def in-respects equivp-reflp*)

lemma *ball-rsp*:
assumes a : $(R \text{ ===> } (=))\ f\ g$
shows $Ball (Respects\ R)\ f = Ball (Respects\ R)\ g$
using a **by** (*auto simp add: Ball-def in-respects elim: rel-funE*)

lemma *bex-rsp*:
assumes a : $(R \text{ ===> } (=))\ f\ g$
shows $Bex (Respects\ R)\ f = Bex (Respects\ R)\ g$
using a **by** (*auto simp add: Bex-def in-respects elim: rel-funE*)

lemma *bex1-rsp*:
assumes a : $(R \text{ ===> } (=))\ f\ g$
shows $Ex1 (\lambda x. x \in Respects\ R \wedge f\ x) = Ex1 (\lambda x. x \in Respects\ R \wedge g\ x)$

using *a* by (*auto elim: rel-funE simp add: Ex1-def in-respects*)

Two lemmas needed for cleaning of quantifiers

lemma *all-prs*:

assumes *a*: *Quotient3 R absf repf*

shows *Ball (Respects R) ((absf ----> id) f) = All f*

using *a* **unfolding** *Quotient3-def Ball-def in-respects id-apply comp-def map-fun-def*
by *metis*

lemma *ex-prs*:

assumes *a*: *Quotient3 R absf repf*

shows *Bex (Respects R) ((absf ----> id) f) = Ex f*

using *a* **unfolding** *Quotient3-def Bex-def in-respects id-apply comp-def map-fun-def*
by *metis*

44.4 *Bex1-rel* quantifier

definition

Bex1-rel :: (*'a* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow (*'a* \Rightarrow *bool*) \Rightarrow *bool*

where

Bex1-rel R P \longleftrightarrow ($\exists x \in \text{Respects } R. P x$) \wedge ($\forall x \in \text{Respects } R. \forall y \in \text{Respects } R. ((P x \wedge P y) \longrightarrow (R x y))$)

lemma *bex1-rel-aux*:

$\llbracket \forall xa ya. R xa ya \longrightarrow x xa = y ya; Bex1-rel R x \rrbracket \Longrightarrow Bex1-rel R y$

unfolding *Bex1-rel-def*

by (*metis in-respects*)

lemma *bex1-rel-aux2*:

$\llbracket \forall xa ya. R xa ya \longrightarrow x xa = y ya; Bex1-rel R y \rrbracket \Longrightarrow Bex1-rel R x$

unfolding *Bex1-rel-def*

by (*metis in-respects*)

lemma *bex1-rel-rsp*:

assumes *a*: *Quotient3 R absf repf*

shows ($(R == => (=)) == => (=)$) (*Bex1-rel R*) (*Bex1-rel R*)

unfolding *rel-fun-def* **by** (*metis bex1-rel-aux bex1-rel-aux2*)

lemma *ex1-prs*:

assumes *Quotient3 R absf repf*

shows ($((absf ----> id) ----> id)$ (*Bex1-rel R*) *f* = *Ex1 f*)
(*is ?lhs = ?rhs*)

using *assms*

apply (*auto simp add: Bex1-rel-def Respects-def*)

by (*metis (full-types) Quotient3-def*)⁺

lemma *bex1-bexeq-reg*:

shows ($\exists !x \in \text{Respects } R. P x$) \longrightarrow (*Bex1-rel R* ($\lambda x. P x$))

by (*auto simp add: Ex1-def Bex1-rel-def Bex-def Ball-def in-respects*)

lemma *bex1-bexeq-reg-eqv*:
assumes *a*: *equivp R*
shows $(\exists!x. P x) \longrightarrow \text{Bex1-rel } R P$
using *equivp-reflp[OF a]*
by (*metis (full-types) Bex1-rel-def in-respects*)

44.5 Various respects and preserve lemmas

lemma *quot-rel-rsp*:
assumes *a*: *Quotient3 R Abs Rep*
shows $(R \text{====>} R \text{====>} (=)) R R$
apply(*rule rel-funI*)
by (*meson assms equals-rsp*)

lemma *o-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
and *q3*: *Quotient3 R3 Abs3 Rep3*
shows $((\text{Abs2} \text{---->} \text{Rep3}) \text{---->} (\text{Abs1} \text{---->} \text{Rep2}) \text{---->} (\text{Rep1} \text{---->} \text{Abs3})) (\circ) = (\circ)$
and $(\text{id} \text{---->} (\text{Abs1} \text{---->} \text{id}) \text{---->} \text{Rep1} \text{---->} \text{id}) (\circ) = (\circ)$
using *Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF q3]*
by (*simp-all add: fun-eq-iff*)

lemma *o-rsp*:
 $((R2 \text{====>} R3) \text{====>} (R1 \text{====>} R2) \text{====>} (R1 \text{====>} R3)) (\circ) (\circ)$
 $((=) \text{====>} (R1 \text{====>} (=)) \text{====>} R1 \text{====>} (=)) (\circ) (\circ)$
by (*force elim: rel-funE*)

lemma *cond-prs*:
assumes *a*: *Quotient3 R absf repf*
shows *absf (if a then repf b else repf c) = (if a then b else c)*
using *a unfolding Quotient3-def* **by** *auto*

lemma *if-prs*:
assumes *q*: *Quotient3 R Abs Rep*
shows $(\text{id} \text{---->} \text{Rep} \text{---->} \text{Rep} \text{---->} \text{Abs}) \text{If} = \text{If}$
using *Quotient3-abs-rep[OF q]*
by (*auto simp add: fun-eq-iff*)

lemma *if-rsp*:
assumes *q*: *Quotient3 R Abs Rep*
shows $((=) \text{====>} R \text{====>} R \text{====>} R) \text{If If}$
by *force*

lemma *let-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*

and $q2$: *Quotient3 R2 Abs2 Rep2*
shows $(Rep2 \dashrightarrow (Abs2 \dashrightarrow Rep1) \dashrightarrow Abs1) Let = Let$
using *Quotient3-abs-rep[OF q1]* *Quotient3-abs-rep[OF q2]*
by (*auto simp add: fun-eq-iff*)

lemma *let-rsp*:
shows $(R1 \implies (R1 \implies R2) \implies R2) Let Let$
by (*force elim: rel-funE*)

lemma *id-rsp*:
shows $(R \implies R) id id$
by *auto*

lemma *id-prs*:
assumes a : *Quotient3 R Abs Rep*
shows $(Rep \dashrightarrow Abs) id = id$
by (*simp add: fun-eq-iff Quotient3-abs-rep [OF a]*)

end

locale *quot-type* =
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$
and $Abs :: 'a set \Rightarrow 'b$
and $Rep :: 'b \Rightarrow 'a set$
assumes *equivp: part-equivp R*
and *rep-prop*: $\bigwedge y. \exists x. R x x \wedge Rep y = Collect (R x)$
and *rep-inverse*: $\bigwedge x. Abs (Rep x) = x$
and *abs-inverse*: $\bigwedge c. (\exists x. ((R x x) \wedge (c = Collect (R x)))) \implies (Rep (Abs c)) = c$
and *rep-inject*: $\bigwedge x y. (Rep x = Rep y) = (x = y)$
begin

definition
 $abs :: 'a \Rightarrow 'b$
where
 $abs x = Abs (Collect (R x))$

definition
 $rep :: 'b \Rightarrow 'a$
where
 $rep a = (SOME x. x \in Rep a)$

lemma *some-collect*:
assumes $R r r$
shows $R (SOME x. x \in Collect (R r)) = R r$
apply *simp*
by (*metis assms exE-some equivp[simplified part-equivp-def]*)

lemma *Quotient*:

```

shows Quotient3 R abs rep
unfolding Quotient3-def abs-def rep-def
proof (intro conjI allI)
  fix a r s
  show x: R (SOME x. x ∈ Rep a) (SOME x. x ∈ Rep a) proof –
    obtain x where r: R x x and rep: Rep a = Collect (R x) using rep-prop[of
a] by auto
    have R (SOME x. x ∈ Rep a) x using r rep some-collect by metis
    then have R x (SOME x. x ∈ Rep a) using part-equivp-symp[OF equivp] by
fast
    then show R (SOME x. x ∈ Rep a) (SOME x. x ∈ Rep a)
      using part-equivp-transp[OF equivp] by (metis ⟨R (SOME x. x ∈ Rep a) x⟩)
    qed
    have Collect (R (SOME x. x ∈ Rep a)) = (Rep a) by (metis some-collect
rep-prop)
    then show Abs (Collect (R (SOME x. x ∈ Rep a))) = a using rep-inverse by
auto
    have R r r ⇒ R s s ⇒ Abs (Collect (R r)) = Abs (Collect (R s)) ⟷ R r
= R s
    proof –
      assume R r r and R s s
      then have Abs (Collect (R r)) = Abs (Collect (R s)) ⟷ Collect (R r) =
Collect (R s)
        by (metis abs-inverse)
      also have Collect (R r) = Collect (R s) ⟷ (λA x. x ∈ A) (Collect (R r))
= (λA x. x ∈ A) (Collect (R s))
        by (rule iffI) simp-all
      finally show Abs (Collect (R r)) = Abs (Collect (R s)) ⟷ R r = R s by
simp
    qed
    then show R r s ⟷ R r r ∧ R s s ∧ (Abs (Collect (R r)) = Abs (Collect (R
s)))
      using equivp[simplified part-equivp-def] by metis
    qed
end

```

44.6 Quotient composition

lemma *OOO-quotient3*:

```

fixes R1 :: 'a ⇒ 'a ⇒ bool
fixes Abs1 :: 'a ⇒ 'b and Rep1 :: 'b ⇒ 'a
fixes Abs2 :: 'b ⇒ 'c and Rep2 :: 'c ⇒ 'b
fixes R2' :: 'a ⇒ 'a ⇒ bool
fixes R2 :: 'b ⇒ 'b ⇒ bool
assumes R1: Quotient3 R1 Abs1 Rep1
assumes R2: Quotient3 R2 Abs2 Rep2
assumes Abs1: ∧x y. R2' x y ⇒ R1 x x ⇒ R1 y y ⇒ R2 (Abs1 x) (Abs1 y)
assumes Rep1: ∧x y. R2 x y ⇒ R2' (Rep1 x) (Rep1 y)

```

shows $Quotient3 (R1 \text{ OO } R2' \text{ OO } R1) (Abs2 \circ Abs1) (Rep1 \circ Rep2)$
proof –
have *: $(R1 \text{ OOO } R2') r r \wedge (R1 \text{ OOO } R2') s s \wedge (Abs2 \circ Abs1) r = (Abs2 \circ Abs1) s$
 $\longleftrightarrow (R1 \text{ OOO } R2') r s$ **for** $r s$
proof (*intro iffI conjI; clarify*)
show $(R1 \text{ OOO } R2') r s$
if $r: R1 r a R2' a b R1 b r$ **and** $eq: (Abs2 \circ Abs1) r = (Abs2 \circ Abs1) s$
and $s: R1 s c R2' c d R1 d s$ **for** $a b c d$
proof –
have $R1 r (Rep1 (Abs1 r))$
using r $Quotient3-refl1 R1 rep-abs-rsp$ **by** *fastforce*
moreover **have** $R2' (Rep1 (Abs1 r)) (Rep1 (Abs1 s))$
using *that*
apply *simp*
apply (*metis (full-types) Rep1 Abs1 Quotient3-rel R2 Quotient3-refl1 [OF R1] Quotient3-refl2 [OF R1] Quotient3-rel-abs [OF R1]*)
done
moreover **have** $R1 (Rep1 (Abs1 s)) s$
by (*metis s Quotient3-rel R1 rep-abs-rsp-left*)
ultimately show *?thesis*
by (*metis relcomppI*)
qed
next
fix $x y$
assume $xy: R1 r x R2' x y R1 y s$
then **have** $R2 (Abs1 x) (Abs1 y)$
by (*iprover dest: Abs1 elim: Quotient3-refl1 [OF R1] Quotient3-refl2 [OF R1]*)
then **have** $R2' (Rep1 (Abs1 x)) (Rep1 (Abs1 x)) R2' (Rep1 (Abs1 y)) (Rep1 (Abs1 y))$
by (*simp-all add: Quotient3-refl1 [OF R2] Quotient3-refl2 [OF R2] Rep1*)
with $\langle R1 r x \rangle \langle R1 y s \rangle$ **show** $(R1 \text{ OOO } R2') r r (R1 \text{ OOO } R2') s s$
by (*metis (full-types) Quotient3-def R1 relcompp.relcomppI*)
show $(Abs2 \circ Abs1) r = (Abs2 \circ Abs1) s$
using xy **by** *simp (metis (full-types) Abs1 Quotient3-rel R1 R2)*
qed
show *?thesis*
apply (*rule Quotient3I*)
using * **apply** (*simp-all add: o-def Quotient3-abs-rep [OF R2] Quotient3-abs-rep [OF R1]*)
apply (*metis Quotient3-rep-reflp R1 R2 Rep1 relcompp.relcomppI*)
done
qed

lemma *OOO-eq-quotient3*:
fixes $R1 :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
fixes $Abs1 :: 'a \Rightarrow 'b$ **and** $Rep1 :: 'b \Rightarrow 'a$
fixes $Abs2 :: 'b \Rightarrow 'c$ **and** $Rep2 :: 'c \Rightarrow 'b$

```

assumes R1: Quotient3 R1 Abs1 Rep1
assumes R2: Quotient3 (=) Abs2 Rep2
shows Quotient3 (R1 OOO (=)) (Abs2  $\circ$  Abs1) (Rep1  $\circ$  Rep2)
using assms
by (rule OOO-quotient3) auto

```

44.7 Quotient3 to Quotient

```

lemma Quotient3-to-Quotient:
assumes Quotient3 R Abs Rep
and T  $\equiv$   $\lambda x y. R\ x\ x \wedge Abs\ x = y$ 
shows Quotient R Abs Rep T
using assms unfolding Quotient3-def by (intro QuotientI) blast+

```

```

lemma Quotient3-to-Quotient-equivp:
assumes q: Quotient3 R Abs Rep
and T-def: T  $\equiv$   $\lambda x y. Abs\ x = y$ 
and eR: equivp R
shows Quotient R Abs Rep T
proof (intro QuotientI)
  fix a
    show Abs (Rep a) = a using q by(rule Quotient3-abs-rep)
  next
    fix a
      show R (Rep a) (Rep a) using q by(rule Quotient3-rep-reflp)
  next
    fix r s
      show R r s = (R r r  $\wedge$  R s s  $\wedge$  Abs r = Abs s) using q by(rule Quotient3-rel[symmetric])
  next
    show T = ( $\lambda x y. R\ x\ x \wedge Abs\ x = y$ ) using T-def equivp-reflp[OF eR] by simp
qed

```

44.8 ML setup

Auxiliary data for the quotient package

```

named-theorems quot-equiv equivalence relation theorems
and quot-respect respectfulness theorems
and quot-preserve preservation theorems
and id-simps identity simp rules for maps
and quot-thm quotient theorems
ML-file  $\langle$  Tools/Quotient/quotient-info.ML  $\rangle$ 

```

```

declare [[mapQ3 fun = (rel-fun, fun-quotient3)]]

```

```

lemmas [quot-thm] = fun-quotient3
lemmas [quot-respect] = quot-rel-rsp if-rsp o-rsp let-rsp id-rsp
lemmas [quot-preserve] = if-prs o-prs let-prs id-prs
lemmas [quot-equiv] = identity-equivp

```


Lemmas about simplifying id’s.

```
lemmas [id-simps] =
  id-def[symmetric]
  map-fun-id
  id-apply
  id-o
  o-id
  eq-comp-r
  vimage-id
```

Translation functions for the lifting process.

ML-file \langle Tools/Quotient/quotient-term.ML \rangle

Definitions of the quotient types.

ML-file \langle Tools/Quotient/quotient-type.ML \rangle

Definitions for quotient constants.

ML-file \langle Tools/Quotient/quotient-def.ML \rangle

An auxiliary constant for recording some information about the lifted theorem in a tactic.

definition

Quot-True :: 'a \Rightarrow bool

where

Quot-True x \longleftrightarrow True

lemma

shows *QT-all*: *Quot-True* (All P) \Longrightarrow *Quot-True* P

and *QT-ex*: *Quot-True* (Ex P) \Longrightarrow *Quot-True* P

and *QT-ex1*: *Quot-True* (Ex1 P) \Longrightarrow *Quot-True* P

and *QT-lam*: *Quot-True* (λ x. P x) \Longrightarrow (\bigwedge x. *Quot-True* (P x))

and *QT-ext*: (\bigwedge x. *Quot-True* (a x) \Longrightarrow f x = g x) \Longrightarrow (*Quot-True* a \Longrightarrow f = g)

by (*simp-all add: Quot-True-def ext*)

lemma *QT-imp*: *Quot-True* a \equiv *Quot-True* b

by (*simp add: Quot-True-def*)

context includes *lifting-syntax*

begin

Tactics for proving the lifted theorems

ML-file \langle Tools/Quotient/quotient-tacs.ML \rangle

end

44.9 Methods / Interface

method-setup *lifting* =

⟨*Attrib.thms* >> (fn *thms* => fn *ctxt* =>
 SIMPLE-METHOD' (*Quotient-Tacs.lift-tac ctxt [] thms*))
 ⟨*lift theorems to quotient types*⟩

method-setup *lifting-setup* =

⟨*Attrib.thm* >> (fn *thm* => fn *ctxt* =>
 SIMPLE-METHOD' (*Quotient-Tacs.lift-procedure-tac ctxt [] thm*))
 ⟨*set up the three goals for the quotient lifting procedure*⟩

method-setup *descending* =

⟨*Scan.succeed* (fn *ctxt* => SIMPLE-METHOD' (*Quotient-Tacs.descend-tac ctxt []*))
 ⟩
 ⟨*descend theorems to the raw level*⟩

method-setup *descending-setup* =

⟨*Scan.succeed* (fn *ctxt* => SIMPLE-METHOD' (*Quotient-Tacs.descend-procedure-tac ctxt []*))
 ⟩
 ⟨*set up the three goals for the descending theorems*⟩

method-setup *partiality-descending* =

⟨*Scan.succeed* (fn *ctxt* => SIMPLE-METHOD' (*Quotient-Tacs.partiality-descend-tac ctxt []*))
 ⟩
 ⟨*descend theorems to the raw level*⟩

method-setup *partiality-descending-setup* =

⟨*Scan.succeed* (fn *ctxt* =>
 SIMPLE-METHOD' (*Quotient-Tacs.partiality-descend-procedure-tac ctxt []*))
 ⟩
 ⟨*set up the three goals for the descending theorems*⟩

method-setup *regularize* =

⟨*Scan.succeed* (fn *ctxt* => SIMPLE-METHOD' (*Quotient-Tacs.regularize-tac ctxt*))
 ⟩
 ⟨*prove the regularization goals from the quotient lifting procedure*⟩

method-setup *injection* =

⟨*Scan.succeed* (fn *ctxt* => SIMPLE-METHOD' (*Quotient-Tacs.all-injection-tac ctxt*))
 ⟩
 ⟨*prove the rep/abs injection goals from the quotient lifting procedure*⟩

method-setup *cleaning* =

⟨*Scan.succeed* (fn *ctxt* => SIMPLE-METHOD' (*Quotient-Tacs.clean-tac ctxt*))
 ⟩
 ⟨*prove the cleaning goals from the quotient lifting procedure*⟩

attribute-setup *quot-lifted* =

⟨*Scan.succeed* *Quotient-Tacs.lifted-attrib*
 ⟨*lift theorems to quotient types*⟩

no-notation*rel-conj* (**infixr** *OOO* 75)**45 Lifting of BNFs**

lemma *sum-insert-Inl-unit*: $x \in A \implies (\bigwedge y. x = \text{Inr } y \implies \text{Inr } y \in B) \implies x \in \text{insert } (\text{Inl } ()) B$

by (*cases x*) (*simp-all*)

lemma *lift-sum-unit-vimage-commute*:

 $\text{insert } (\text{Inl } ()) (\text{Inr } ' f - ' A) = \text{map-sum id } f - ' \text{insert } (\text{Inl } ()) (\text{Inr } ' A)$ **by** (*auto simp: map-sum-def split: sum.splits*)

lemma *insert-Inl-int-map-sum-unit*: $\text{insert } (\text{Inl } ()) A \cap \text{range } (\text{map-sum id } f) \neq \{\}$

by (*auto simp: map-sum-def split: sum.splits*)

lemma *image-map-sum-unit-subset*:

 $A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } ' B) \implies \text{map-sum id } f - ' A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } ' f - ' B)$ **by** *auto*

lemma *subset-lift-sum-unitD*: $A \subseteq \text{insert } (\text{Inl } ()) (\text{Inr } ' B) \implies \text{Inr } x \in A \implies x \in B$

unfolding *insert-def* **by** *auto*

lemma *UNIV-sum-unit-conv*: $\text{insert } (\text{Inl } ()) (\text{range } \text{Inr}) = \text{UNIV}$

unfolding *UNIV-sum UNIV-unit image-insert image-empty Un-insert-left sup-bot.left-neutral..*

lemma *subset-vimage-image-subset*: $A \subseteq f - ' B \implies f - ' A \subseteq B$

by *auto*

lemma *relcompp-mem-Grp-neq-bot*:

 $A \cap \text{range } f \neq \{\} \implies (\lambda x y. x \in A \wedge y \in A) \text{ OO } (\text{Grp UNIV } f)^{-1-1} \neq \text{bot}$ **unfolding** *Grp-def relcompp-apply fun-eq-iff* **by** *blast*

lemma *comp-projr-Inr*: $\text{projr} \circ \text{Inr} = \text{id}$

by *auto*

lemma *in-rel-sum-in-image-projr*:

 $B \subseteq \{(x,y). \text{rel-sum } ((=) :: \text{unit} \Rightarrow \text{unit} \Rightarrow \text{bool}) A x y\} \implies$ $\text{Inr } ' C = \text{fst } ' B \implies \text{snd } ' B = \text{Inr } ' D \implies \text{map-prod projr projr } ' B \subseteq \{(x,y). A x y\}$ **by** (*force simp: projr-def image-iff dest!: spec[of - Inl ()] split: sum.splits*)

lemma *subset-rel-sumI*: $B \subseteq \{(x,y). A x y\} \implies \text{rel-sum } ((=) :: \text{unit} \Rightarrow \text{unit} \Rightarrow \text{bool}) A$

*(if x ∈ B then Inr (fst x) else Inl ())**(if x ∈ B then Inr (snd x) else Inl ())***by** *auto*

lemma *relcompp-eq-Grp-req-bot*: $(=) \text{ OO } (\text{Grp UNIV } f)^{-1-1} \neq \text{bot}$
unfolding *Grp-def relcompp-apply fun-eq-iff* **by** *blast*

lemma *rel-fun-rel-OO1*: $(\text{rel-fun } Q (\text{rel-fun } R (=))) A B \implies \text{conversep } Q \text{ OO } A \text{ OO } R \leq B$
by *(auto simp: rel-fun-def)*

lemma *rel-fun-rel-OO2*: $(\text{rel-fun } Q (\text{rel-fun } R (=))) A B \implies Q \text{ OO } B \text{ OO } \text{conversep } R \leq A$
by *(auto simp: rel-fun-def)*

lemma *rel-sum-eq2-nonempty*: $\text{rel-sum } (=) A \text{ OO } \text{rel-sum } (=) B \neq \text{bot}$
by *(auto simp: fun-eq-iff relcompp-apply intro!: exI[of - Inl -])*

lemma *rel-sum-eq3-nonempty*: $\text{rel-sum } (=) A \text{ OO } (\text{rel-sum } (=) B \text{ OO } \text{rel-sum } (=) C) \neq \text{bot}$
by *(auto simp: fun-eq-iff relcompp-apply intro!: exI[of - Inl -])*

lemma *hypsubst*: $A = B \implies x \in B \implies (x \in A \implies P) \implies P$ **by** *simp*

lemma *Quotient-crel-quotient*: $\text{Quotient } R \text{ Abs Rep } T \implies \text{equivp } R \implies T \equiv (\lambda x y. \text{Abs } x = y)$
by *(drule Quotient-cr-rel) (auto simp: fun-eq-iff equivp-reflp intro!: eq-reflection)*

lemma *Quotient-crel-typedef*: $\text{Quotient } (\text{eq-onp } P) \text{ Abs Rep } T \implies T \equiv (\lambda x y. x = \text{Rep } y)$
unfolding *Quotient-def*
by *(auto 0 4 simp: fun-eq-iff eq-onp-def intro: sym intro!: eq-reflection)*

lemma *Quotient-crel-typecopy*: $\text{Quotient } (=) \text{ Abs Rep } T \implies T \equiv (\lambda x y. x = \text{Rep } y)$
by *(subst (asm) eq-onp-True[symmetric]) (rule Quotient-crel-typedef)*

lemma *equivp-add-relconj*:
assumes *equiv*: $\text{equivp } R \text{ equivp } R'$ **and** *le*: $S \text{ OO } T \text{ OO } U \leq R \text{ OO } STU \text{ OO } R'$
shows $R \text{ OO } S \text{ OO } T \text{ OO } U \text{ OO } R' \leq R \text{ OO } STU \text{ OO } R'$

proof –

have *trans*: $R \text{ OO } R \leq R \text{ OO } R' \text{ OO } R' \leq R'$

using *equiv* **unfolding** *equivp-reflp-symp-transp transp-relcompp* **by** *blast+*

have $R \text{ OO } S \text{ OO } T \text{ OO } U \text{ OO } R' = R \text{ OO } (S \text{ OO } T \text{ OO } U) \text{ OO } R'$

unfolding *relcompp-assoc* ..

also have $\dots \leq R \text{ OO } (R \text{ OO } STU \text{ OO } R') \text{ OO } R'$

by *(intro le relcompp-mono order-refl)*

also have $\dots \leq (R \text{ OO } R) \text{ OO } STU \text{ OO } (R' \text{ OO } R')$

unfolding *relcompp-assoc* ..

also have $\dots \leq R \text{ OO } STU \text{ OO } R'$

by *(intro trans relcompp-mono order-refl)*

finally show *?thesis* .

qed

lemma *Grp-conversep-eq-onp*: $((\text{BNF-Def.Grp UNIV } f)^{-1-1} \text{ OO } \text{BNF-Def.Grp UNIV } f) = \text{eq-onp } (\lambda x. x \in \text{range } f)$
by (*auto simp: fun-eq-iff Grp-def eq-onp-def image-iff*)

lemma *Grp-conversep-nonempty*: $(\text{BNF-Def.Grp UNIV } f)^{-1-1} \text{ OO } \text{BNF-Def.Grp UNIV } f \neq \text{bot}$
by (*auto simp: fun-eq-iff Grp-def*)

lemma *relcomppI2*: $r \ a \ b \implies s \ b \ c \implies t \ c \ d \implies (r \text{ OO } s \text{ OO } t) \ a \ d$
by (*auto*)

lemma *rel-conj-eq-onp*: $\text{equivp } R \implies \text{rel-conj } R \ (\text{eq-onp } P) \leq R$
by (*auto simp: eq-onp-def transp-def equivp-def*)

lemma *Quotient-Quotient3*: $\text{Quotient } R \ \text{Abs } \text{Rep } T \implies \text{Quotient3 } R \ \text{Abs } \text{Rep}$
unfolding *Quotient-def Quotient3-def* **by** *blast*

lemma *Quotient-reflp-imp-equivp*: $\text{Quotient } R \ \text{Abs } \text{Rep } T \implies \text{reflp } R \implies \text{equivp } R$
using *Quotient-symp Quotient-transp equivpI* **by** *blast*

lemma *Quotient-eq-onp-typedef*:
 $\text{Quotient } (\text{eq-onp } P) \ \text{Abs } \text{Rep } cr \implies \text{type-definition } \text{Rep } \text{Abs } \{x. P \ x\}$
unfolding *Quotient-def eq-onp-def*
by *unfold-locales auto*

lemma *Quotient-eq-onp-type-copy*:
 $\text{Quotient } (=) \ \text{Abs } \text{Rep } cr \implies \text{type-definition } \text{Rep } \text{Abs } \text{UNIV}$
unfolding *Quotient-def eq-onp-def*
by *unfold-locales auto*

ML-file $\langle \text{Tools/BNF/bnf-lift.ML} \rangle$

hide-fact

*sum-insert-Inl-unit lift-sum-unit-vimage-commute insert-Inl-int-map-sum-unit
image-map-sum-unit-subset subset-lift-sum-unitD UNIV-sum-unit-conv subset-vimage-image-subset
relcompp-mem-Grp-neq-bot comp-projr-Inr in-rel-sum-in-image-projr subset-rel-sumI
relcompp-eq-Grp-neq-bot rel-fun-rel-OO1 rel-fun-rel-OO2 rel-sum-eq2-nonempty
rel-sum-eq3-nonempty
hypsubst equivp-add-relconj Grp-conversep-eq-onp Grp-conversep-nonempty rel-
comppI2 rel-conj-eq-onp
Quotient-reflp-imp-equivp Quotient-Quotient3*

end

46 Binary Numerals

```
theory Num
  imports BNF-Least-Fixpoint Transfer
begin
```

46.1 The *num* type

```
datatype num = One | Bit0 num | Bit1 num
```

Increment function for type *num*

```
primrec inc :: num ⇒ num
  where
    inc One = Bit0 One
  | inc (Bit0 x) = Bit1 x
  | inc (Bit1 x) = Bit0 (inc x)
```

Converting between type *num* and type *nat*

```
primrec nat-of-num :: num ⇒ nat
  where
    nat-of-num One = Suc 0
  | nat-of-num (Bit0 x) = nat-of-num x + nat-of-num x
  | nat-of-num (Bit1 x) = Suc (nat-of-num x + nat-of-num x)
```

```
primrec num-of-nat :: nat ⇒ num
  where
    num-of-nat 0 = One
  | num-of-nat (Suc n) = (if 0 < n then inc (num-of-nat n) else One)
```

```
lemma nat-of-num-pos: 0 < nat-of-num x
  by (induct x) simp-all
```

```
lemma nat-of-num-neq-0: nat-of-num x ≠ 0
  by (induct x) simp-all
```

```
lemma nat-of-num-inc: nat-of-num (inc x) = Suc (nat-of-num x)
  by (induct x) simp-all
```

```
lemma num-of-nat-double: 0 < n ⇒ num-of-nat (n + n) = Bit0 (num-of-nat n)
  by (induct n) simp-all
```

Type *num* is isomorphic to the strictly positive natural numbers.

```
lemma nat-of-num-inverse: num-of-nat (nat-of-num x) = x
  by (induct x) (simp-all add: num-of-nat-double nat-of-num-pos)
```

```
lemma num-of-nat-inverse: 0 < n ⇒ nat-of-num (num-of-nat n) = n
  by (induct n) (simp-all add: nat-of-num-inc)
```

```
lemma num-eq-iff: x = y ⟷ nat-of-num x = nat-of-num y
```

```

apply safe
apply (drule arg-cong [where f=num-of-nat])
apply (simp add: nat-of-num-inverse)
done

lemma num-induct [case-names One inc]:
  fixes P :: num  $\Rightarrow$  bool
  assumes One: P One
    and inc:  $\bigwedge x. P\ x \Longrightarrow P\ (inc\ x)$ 
  shows P x
proof –
  obtain n where n: Suc n = nat-of-num x
    by (cases nat-of-num x) (simp-all add: nat-of-num-neq-0)
  have P (num-of-nat (Suc n))
  proof (induct n)
    case 0
    from One show ?case by simp
  next
    case (Suc n)
    then have P (inc (num-of-nat (Suc n))) by (rule inc)
    then show P (num-of-nat (Suc (Suc n))) by simp
  qed
  with n show P x
    by (simp add: nat-of-num-inverse)
qed

```

From now on, there are two possible models for *num*: as positive naturals (rule *num-induct*) and as digit representation (rules *num.induct*, *num.cases*).

46.2 Numeral operations

```

instantiation num :: {plus, times, linorder}
begin

```

```

definition [code del]:  $m + n = num-of-nat\ (nat-of-num\ m + nat-of-num\ n)$ 

```

```

definition [code del]:  $m * n = num-of-nat\ (nat-of-num\ m * nat-of-num\ n)$ 

```

```

definition [code del]:  $m \leq n \longleftrightarrow nat-of-num\ m \leq nat-of-num\ n$ 

```

```

definition [code del]:  $m < n \longleftrightarrow nat-of-num\ m < nat-of-num\ n$ 

```

```

instance

```

```

  by standard (auto simp add: less-num-def less-eq-num-def num-eq-iff)

```

```

end

```

```

lemma nat-of-num-add:  $nat-of-num\ (x + y) = nat-of-num\ x + nat-of-num\ y$ 
  unfolding plus-num-def

```

by (intro num-of-nat-inverse add-pos-pos nat-of-num-pos)

lemma *nat-of-num-mult*: $\text{nat-of-num } (x * y) = \text{nat-of-num } x * \text{nat-of-num } y$
unfolding *times-num-def*
 by (intro num-of-nat-inverse mult-pos-pos nat-of-num-pos)

lemma *add-num-simps* [*simp*, *code*]:

$\text{One} + \text{One} = \text{Bit0 One}$
 $\text{One} + \text{Bit0 } n = \text{Bit1 } n$
 $\text{One} + \text{Bit1 } n = \text{Bit0 } (n + \text{One})$
 $\text{Bit0 } m + \text{One} = \text{Bit1 } m$
 $\text{Bit0 } m + \text{Bit0 } n = \text{Bit0 } (m + n)$
 $\text{Bit0 } m + \text{Bit1 } n = \text{Bit1 } (m + n)$
 $\text{Bit1 } m + \text{One} = \text{Bit0 } (m + \text{One})$
 $\text{Bit1 } m + \text{Bit0 } n = \text{Bit1 } (m + n)$
 $\text{Bit1 } m + \text{Bit1 } n = \text{Bit0 } (m + n + \text{One})$
 by (simp-all add: num-eq-iff nat-of-num-add)

lemma *mult-num-simps* [*simp*, *code*]:

$m * \text{One} = m$
 $\text{One} * n = n$
 $\text{Bit0 } m * \text{Bit0 } n = \text{Bit0 } (\text{Bit0 } (m * n))$
 $\text{Bit0 } m * \text{Bit1 } n = \text{Bit0 } (m * \text{Bit1 } n)$
 $\text{Bit1 } m * \text{Bit0 } n = \text{Bit0 } (\text{Bit1 } m * n)$
 $\text{Bit1 } m * \text{Bit1 } n = \text{Bit1 } (m + n + \text{Bit0 } (m * n))$
 by (simp-all add: num-eq-iff nat-of-num-add nat-of-num-mult distrib-right distrib-left)

lemma *eq-num-simps*:

$\text{One} = \text{One} \longleftrightarrow \text{True}$
 $\text{One} = \text{Bit0 } n \longleftrightarrow \text{False}$
 $\text{One} = \text{Bit1 } n \longleftrightarrow \text{False}$
 $\text{Bit0 } m = \text{One} \longleftrightarrow \text{False}$
 $\text{Bit1 } m = \text{One} \longleftrightarrow \text{False}$
 $\text{Bit0 } m = \text{Bit0 } n \longleftrightarrow m = n$
 $\text{Bit0 } m = \text{Bit1 } n \longleftrightarrow \text{False}$
 $\text{Bit1 } m = \text{Bit0 } n \longleftrightarrow \text{False}$
 $\text{Bit1 } m = \text{Bit1 } n \longleftrightarrow m = n$
 by simp-all

lemma *le-num-simps* [*simp*, *code*]:

$\text{One} \leq n \longleftrightarrow \text{True}$
 $\text{Bit0 } m \leq \text{One} \longleftrightarrow \text{False}$
 $\text{Bit1 } m \leq \text{One} \longleftrightarrow \text{False}$
 $\text{Bit0 } m \leq \text{Bit0 } n \longleftrightarrow m \leq n$
 $\text{Bit0 } m \leq \text{Bit1 } n \longleftrightarrow m \leq n$
 $\text{Bit1 } m \leq \text{Bit1 } n \longleftrightarrow m \leq n$
 $\text{Bit1 } m \leq \text{Bit0 } n \longleftrightarrow m < n$
 using nat-of-num-pos [of *n*] nat-of-num-pos [of *m*]

by (auto simp add: less-eq-num-def less-num-def)

lemma less-num-simps [simp, code]:

$m < \text{One} \longleftrightarrow \text{False}$

$\text{One} < \text{Bit0 } n \longleftrightarrow \text{True}$

$\text{One} < \text{Bit1 } n \longleftrightarrow \text{True}$

$\text{Bit0 } m < \text{Bit0 } n \longleftrightarrow m < n$

$\text{Bit0 } m < \text{Bit1 } n \longleftrightarrow m \leq n$

$\text{Bit1 } m < \text{Bit1 } n \longleftrightarrow m < n$

$\text{Bit1 } m < \text{Bit0 } n \longleftrightarrow m < n$

using nat-of-num-pos [of n] nat-of-num-pos [of m]

by (auto simp add: less-eq-num-def less-num-def)

lemma le-num-One-iff: $x \leq \text{num.One} \longleftrightarrow x = \text{num.One}$

by (simp add: antisym-conv)

Rules using *One* and *inc* as constructors.

lemma add-One: $x + \text{One} = \text{inc } x$

by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma add-One-commute: $\text{One} + n = n + \text{One}$

by (induct n) simp-all

lemma add-inc: $x + \text{inc } y = \text{inc } (x + y)$

by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma mult-inc: $x * \text{inc } y = x * y + x$

by (simp add: num-eq-iff nat-of-num-mult nat-of-num-add nat-of-num-inc)

The *num-of-nat* conversion.

lemma num-of-nat-One: $n \leq 1 \implies \text{num-of-nat } n = \text{One}$

by (cases n) simp-all

lemma num-of-nat-plus-distrib:

$0 < m \implies 0 < n \implies \text{num-of-nat } (m + n) = \text{num-of-nat } m + \text{num-of-nat } n$

by (induct n) (auto simp add: add-One add-One-commute add-inc)

A double-and-decrement function.

primrec *BitM* :: *num* \Rightarrow *num*

where

$\text{BitM } \text{One} = \text{One}$

| $\text{BitM } (\text{Bit0 } n) = \text{Bit1 } (\text{BitM } n)$

| $\text{BitM } (\text{Bit1 } n) = \text{Bit1 } (\text{Bit0 } n)$

lemma *BitM-plus-one*: $\text{BitM } n + \text{One} = \text{Bit0 } n$

by (induct n) simp-all

lemma *one-plus-BitM*: $\text{One} + \text{BitM } n = \text{Bit0 } n$

unfolding add-One-commute *BitM-plus-one* ..

lemma *BitM-inc-eq*:
 $\langle \text{Num.BitM } (\text{Num.inc } n) = \text{Num.Bit1 } n \rangle$
by (*induction n simp-all*)

lemma *inc-BitM-eq*:
 $\langle \text{Num.inc } (\text{Num.BitM } n) = \text{num.Bit0 } n \rangle$
by (*simp add: BitM-plus-one[symmetric] add-One*)

Squaring and exponentiation.

primrec *sqr* :: *num* \Rightarrow *num*
where
sqr One = *One*
| *sqr (Bit0 n)* = *Bit0 (sqr n)*
| *sqr (Bit1 n)* = *Bit1 (sqr n + n)*

primrec *pow* :: *num* \Rightarrow *num* \Rightarrow *num*
where
pow x One = *x*
| *pow x (Bit0 y)* = *sqr (pow x y)*
| *pow x (Bit1 y)* = *sqr (pow x y) * x*

lemma *nat-of-num-sqr*: *nat-of-num (sqr x)* = *nat-of-num x * nat-of-num x*
by (*induct x (simp-all add: algebra-simps nat-of-num-add)*)

lemma *sqr-conv-mult*: *sqr x* = *x * x*
by (*simp add: num-eq-iff nat-of-num-sqr nat-of-num-mult*)

lemma *num-double [simp]*:
*num.Bit0 num.One * n* = *num.Bit0 n*
by (*simp add: num-eq-iff nat-of-num-mult*)

46.3 Binary numerals

We embed binary representations into a generic algebraic structure using *numeral*.

class *numeral* = *one + semigroup-add*
begin

primrec *numeral* :: *num* \Rightarrow 'a
where
numeral-One: *numeral One* = 1
| *numeral-Bit0*: *numeral (Bit0 n)* = *numeral n + numeral n*
| *numeral-Bit1*: *numeral (Bit1 n)* = *numeral n + numeral n + 1*

lemma *numeral-code [code]*:
numeral One = 1
numeral (Bit0 n) = (*let m = numeral n in m + m*)

```
numeral (Bit1 n) = (let m = numeral n in m + m + 1)
by (simp-all add: Let-def)
```

```
lemma one-plus-numeral-commute: 1 + numeral x = numeral x + 1
proof (induct x)
  case One
  then show ?case by simp
next
  case Bit0
  then show ?case by (simp add: add.assoc [symmetric]) (simp add: add.assoc)
next
  case Bit1
  then show ?case by (simp add: add.assoc [symmetric]) (simp add: add.assoc)
qed
```

```
lemma numeral-inc: numeral (inc x) = numeral x + 1
proof (induct x)
  case One
  then show ?case by simp
next
  case Bit0
  then show ?case by simp
next
  case (Bit1 x)
  have numeral x + (1 + numeral x) + 1 = numeral x + (numeral x + 1) + 1
  by (simp only: one-plus-numeral-commute)
  with Bit1 show ?case
  by (simp add: add.assoc)
qed
```

```
declare numeral.simps [simp del]
```

```
abbreviation Numeral1  $\equiv$  numeral One
```

```
declare numeral-One [code-post]
```

```
end
```

Numeral syntax.

```
syntax
-Numeral :: num-const  $\Rightarrow$  'a (-)
```

```
ML-file <Tools/numeral.ML>
```

```
parse-translation <
  let
    fun numeral-tr [(c as Const (syntax-const <-constrain>, -)) $ t $ u] =
      c $ numeral-tr [t] $ u
    | numeral-tr [Const (num, -)] =
```

```

    (Numeral.mk-number-syntax o #value o Lexicon.read-num) num
  | numeral-tr ts = raise TERM (numeral-tr, ts);
in [(syntax-const <-Numeral>, K numeral-tr)] end
>

```

```

typed-print-translation <
  let
    fun num-tr' ctxt T [n] =
      let
        val k = Numeral.dest-num-syntax n;
        val t' =
          Syntax.const syntax-const <-Numeral> $
            Syntax.free (string-of-int k);
      in
        (case T of
          Type (type-name <fun>, [-, T']) =>
            if Printer.type-emphasis ctxt T' then
              Syntax.const syntax-const <-constrain> $ t' $
                Syntax-Phases.term-of-typ ctxt T'
            else t'
          | - => if T = dummyT then t' else raise Match)
        end;
      in
        [(const-syntax <numeral>, num-tr')]
      end
  >

```

46.4 Class-specific numeral rules

numeral is a morphism.

46.4.1 Structures with addition: class *numeral*

context *numeral*
begin

lemma *numeral-add*: *numeral* ($m + n$) = *numeral* m + *numeral* n
by (*induct* n rule: *num-induct*)
 (*simp-all* only: *numeral-One* *add-One* *add-inc* *numeral-inc* *add.assoc*)

lemma *numeral-plus-numeral*: *numeral* m + *numeral* n = *numeral* ($m + n$)
by (rule *numeral-add* [*symmetric*])

lemma *numeral-plus-one*: *numeral* n + 1 = *numeral* ($n + One$)
using *numeral-add* [*of* n *One*] **by** (*simp* *add*: *numeral-One*)

lemma *one-plus-numeral*: 1 + *numeral* n = *numeral* ($One + n$)
using *numeral-add* [*of* *One* n] **by** (*simp* *add*: *numeral-One*)

```
lemma one-add-one: 1 + 1 = 2
  using numeral-add [of One One] by (simp add: numeral-One)
```

```
lemmas add-numeral-special =
  numeral-plus-one one-plus-numeral one-add-one
```

```
end
```

46.4.2 Structures with negation: class *neg-numeral*

```
class neg-numeral = numeral + group-add
begin
```

```
lemma uminus-numeral-One: - Numeral1 = - 1
  by (simp add: numeral-One)
```

Numerals form an abelian subgroup.

```
inductive is-num :: 'a ⇒ bool
  where
    is-num 1
  | is-num x ⇒ is-num (- x)
  | is-num x ⇒ is-num y ⇒ is-num (x + y)
```

```
lemma is-num-numeral: is-num (numeral k)
  by (induct k) (simp-all add: numeral.simps is-num.intros)
```

```
lemma is-num-add-commute: is-num x ⇒ is-num y ⇒ x + y = y + x
proof (induction x rule: is-num.induct)
```

```
  case 1
  then show ?case
  proof (induction y rule: is-num.induct)
    case 1
    then show ?case by simp
  next
    case (2 y)
    then have y + (1 + - y) + y = y + (- y + 1) + y
      by (simp add: add.assoc)
    then have y + (1 + - y) = y + (- y + 1)
      by simp
    then show ?case
      by (rule add-left-imp-eq[of y])
  next
    case (3 x y)
    then have 1 + (x + y) = x + 1 + y
      by (simp add: add.assoc [symmetric])
    then show ?case using 3
      by (simp add: add.assoc)
  qed
next
```

```

case (2 x)
then have  $x + (-x + y) + x = x + (y + -x) + x$ 
  by (simp add: add.assoc)
then have  $x + (-x + y) = x + (y + -x)$ 
  by simp
then show ?case
  by (rule add-left-imp-eq[of x])
next
case (3 x z)
moreover have  $x + (y + z) = (x + y) + z$ 
  by (simp add: add.assoc[symmetric])
ultimately show ?case
  by (simp add: add.assoc)
qed

```

lemma *is-num-add-left-commute*: $is\text{-}num\ x \implies is\text{-}num\ y \implies x + (y + z) = y + (x + z)$
by (simp only: add.assoc [symmetric] is-num-add-commute)

lemmas *is-num-normalize =*
add.assoc is-num-add-commute is-num-add-left-commute
is-num.intros is-num-numeral
minus-add

definition *dbl* :: 'a \Rightarrow 'a
where $dbl\ x = x + x$

definition *dbl-inc* :: 'a \Rightarrow 'a
where $dbl\text{-}inc\ x = x + x + 1$

definition *dbl-dec* :: 'a \Rightarrow 'a
where $dbl\text{-}dec\ x = x + x - 1$

definition *sub* :: num \Rightarrow num \Rightarrow 'a
where $sub\ k\ l = numeral\ k - numeral\ l$

lemma *numeral-BitM*: $numeral\ (BitM\ n) = numeral\ (Bit0\ n) - 1$
by (simp only: BitM-plus-one [symmetric] numeral-add numeral-One eq-diff-eq)

lemma *sub-inc-One-eq*:
 $\langle Num.sub\ (Num.inc\ n)\ num.One = numeral\ n \rangle$
by (simp-all add: sub-def diff-eq-eq numeral-inc numeral.numeral-One)

lemma *dbl-simps* [simp]:
 $dbl\ (-\ numeral\ k) = -\ dbl\ (numeral\ k)$
 $dbl\ 0 = 0$
 $dbl\ 1 = 2$
 $dbl\ (-\ 1) = -\ 2$
 $dbl\ (numeral\ k) = numeral\ (Bit0\ k)$

by (*simp-all add: dbl-def numeral.simps minus-add*)

lemma *dbl-inc-simps* [*simp*]:

dbl-inc ($- \text{numeral } k$) = $- \text{dbl-dec} (\text{numeral } k)$

dbl-inc 0 = 1

dbl-inc 1 = 3

dbl-inc ($- 1$) = $- 1$

dbl-inc (*numeral* k) = *numeral* (*Bit1* k)

by (*simp-all add: dbl-inc-def dbl-dec-def numeral.simps numeral-BitM is-num-normalize algebra-simps*

del: add-uminus-conv-diff)

lemma *dbl-dec-simps* [*simp*]:

dbl-dec ($- \text{numeral } k$) = $- \text{dbl-inc} (\text{numeral } k)$

dbl-dec 0 = $- 1$

dbl-dec 1 = 1

dbl-dec ($- 1$) = $- 3$

dbl-dec (*numeral* k) = *numeral* (*BitM* k)

by (*simp-all add: dbl-dec-def dbl-inc-def numeral.simps numeral-BitM is-num-normalize*)

lemma *sub-num-simps* [*simp*]:

sub One One = 0

sub One (*Bit0* l) = $- \text{numeral} (\text{BitM } l)$

sub One (*Bit1* l) = $- \text{numeral} (\text{Bit0 } l)$

sub (*Bit0* k) *One* = *numeral* (*BitM* k)

sub (*Bit1* k) *One* = *numeral* (*Bit0* k)

sub (*Bit0* k) (*Bit0* l) = *dbl* (*sub* $k l$)

sub (*Bit0* k) (*Bit1* l) = *dbl-dec* (*sub* $k l$)

sub (*Bit1* k) (*Bit0* l) = *dbl-inc* (*sub* $k l$)

sub (*Bit1* k) (*Bit1* l) = *dbl* (*sub* $k l$)

by (*simp-all add: dbl-def dbl-dec-def dbl-inc-def sub-def numeral.simps*

numeral-BitM is-num-normalize del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma *add-neg-numeral-simps*:

numeral $m + - \text{numeral } n$ = *sub* $m n$

$- \text{numeral } m + \text{numeral } n$ = *sub* $n m$

$- \text{numeral } m + - \text{numeral } n$ = $- (\text{numeral } m + \text{numeral } n)$

by (*simp-all add: sub-def numeral-add numeral.simps is-num-normalize*

del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma *add-neg-numeral-special*:

1 + $- \text{numeral } m$ = *sub One* m

$- \text{numeral } m + 1$ = *sub One* m

numeral $m + - 1$ = *sub* $m One$

$- 1 + \text{numeral } n$ = *sub* $n One$

$- 1 + - \text{numeral } n$ = $- \text{numeral} (\text{inc } n)$

$- \text{numeral } m + - 1$ = $- \text{numeral} (\text{inc } m)$

1 + $- 1$ = 0

$- 1 + 1$ = 0

$$- 1 + - 1 = - 2$$

by (*simp-all add: sub-def numeral-add numeral.simps is-num-normalize right-minus numeral-inc*
del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma *diff-numeral-simps:*

$$\text{numeral } m - \text{numeral } n = \text{sub } m \ n$$

$$\text{numeral } m - - \text{numeral } n = \text{numeral } (m + n)$$

$$- \text{numeral } m - \text{numeral } n = - \text{numeral } (m + n)$$

$$- \text{numeral } m - - \text{numeral } n = \text{sub } n \ m$$

by (*simp-all add: sub-def numeral-add numeral.simps is-num-normalize*
del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma *diff-numeral-special:*

$$1 - \text{numeral } n = \text{sub } \text{One } n$$

$$\text{numeral } m - 1 = \text{sub } m \ \text{One}$$

$$1 - - \text{numeral } n = \text{numeral } (\text{One} + n)$$

$$- \text{numeral } m - 1 = - \text{numeral } (m + \text{One})$$

$$- 1 - \text{numeral } n = - \text{numeral } (\text{inc } n)$$

$$\text{numeral } m - - 1 = \text{numeral } (\text{inc } m)$$

$$- 1 - - \text{numeral } n = \text{sub } n \ \text{One}$$

$$- \text{numeral } m - - 1 = \text{sub } \text{One } m$$

$$1 - 1 = 0$$

$$- 1 - 1 = - 2$$

$$1 - - 1 = 2$$

$$- 1 - - 1 = 0$$

by (*simp-all add: sub-def numeral-add numeral.simps is-num-normalize numeral-inc*
del: add-uminus-conv-diff add: diff-conv-add-uminus)

end

46.4.3 Structures with multiplication: class *semiring-numeral*

class *semiring-numeral* = *semiring* + *monoid-mult*

begin

subclass *numeral* ..

lemma *numeral-mult: numeral (m * n) = numeral m * numeral n*

by (*induct n rule: num-induct*)

(*simp-all add: numeral-One mult-inc numeral-inc numeral-add distrib-left*)

lemma *numeral-times-numeral: numeral m * numeral n = numeral (m * n)*

by (*rule numeral-mult [symmetric]*)

lemma *mult-2: 2 * z = z + z*

by (*simp add: one-add-one [symmetric] distrib-right*)

lemma *mult-2-right: z * 2 = z + z*

by (simp add: one-add-one [symmetric] distrib-left)

lemma left-add-twice:

$a + (a + b) = 2 * a + b$

by (simp add: mult-2 ac-simps)

end

46.4.4 Structures with a zero: class *semiring-1*

context *semiring-1*

begin

subclass *semiring-numeral* ..

lemma of-nat-numeral [simp]: $of\text{-}nat\ (numeral\ n) = numeral\ n$

by (induct n) (simp-all only: numeral.simps numeral-class.numeral.simps of-nat-add of-nat-1)

end

lemma nat-of-num-numeral [code-abbrev]: $nat\text{-}of\text{-}num = numeral$

proof

fix n

have $numeral\ n = nat\text{-}of\text{-}num\ n$

by (induct n) (simp-all add: numeral.simps)

then show $nat\text{-}of\text{-}num\ n = numeral\ n$

by simp

qed

lemma nat-of-num-code [code]:

$nat\text{-}of\text{-}num\ One = 1$

$nat\text{-}of\text{-}num\ (Bit0\ n) = (let\ m = nat\text{-}of\text{-}num\ n\ in\ m + m)$

$nat\text{-}of\text{-}num\ (Bit1\ n) = (let\ m = nat\text{-}of\text{-}num\ n\ in\ Suc\ (m + m))$

by (simp-all add: Let-def)

46.4.5 Equality: class *semiring-char-0*

context *semiring-char-0*

begin

lemma numeral-eq-iff: $numeral\ m = numeral\ n \longleftrightarrow m = n$

by (simp only: of-nat-numeral [symmetric] nat-of-num-numeral [symmetric] of-nat-eq-iff num-eq-iff)

lemma numeral-eq-one-iff: $numeral\ n = 1 \longleftrightarrow n = One$

by (rule numeral-eq-iff [of n One, unfolded numeral-One])

lemma one-eq-numeral-iff: $1 = numeral\ n \longleftrightarrow One = n$

by (rule numeral-eq-iff [of One n, unfolded numeral-One])

lemma *numeral-neq-zero*: numeral $n \neq 0$
by (*simp add*: *of-nat-numeral* [*symmetric*] *nat-of-num-numeral* [*symmetric*] *nat-of-num-pos*)

lemma *zero-neq-numeral*: $0 \neq$ numeral n
unfolding *eq-commute* [*of 0*] **by** (*rule numeral-neq-zero*)

lemmas *eq-numeral-simps* [*simp*] =
numeral-eq-iff
numeral-eq-one-iff
one-eq-numeral-iff
numeral-neq-zero
zero-neq-numeral

end

46.4.6 Comparisons: class *linordered-nonzero-semiring*

context *linordered-nonzero-semiring*
begin

lemma *numeral-le-iff*: numeral $m \leq$ numeral $n \iff m \leq n$

proof –

have *of-nat* (numeral m) \leq *of-nat* (numeral n) $\iff m \leq n$

by (*simp only*: *less-eq-num-def nat-of-num-numeral of-nat-le-iff*)

then show *?thesis* **by** *simp*

qed

lemma *one-le-numeral*: $1 \leq$ numeral n

using *numeral-le-iff* [*of num.One n*] **by** (*simp add*: *numeral-One*)

lemma *numeral-le-one-iff*: numeral $n \leq 1 \iff n \leq$ *num.One*

using *numeral-le-iff* [*of n num.One*] **by** (*simp add*: *numeral-One*)

lemma *numeral-less-iff*: numeral $m <$ numeral $n \iff m < n$

proof –

have *of-nat* (numeral m) $<$ *of-nat* (numeral n) $\iff m < n$

unfolding *less-num-def nat-of-num-numeral of-nat-less-iff* ..

then show *?thesis* **by** *simp*

qed

lemma *not-numeral-less-one*: \neg numeral $n < 1$

using *numeral-less-iff* [*of n num.One*] **by** (*simp add*: *numeral-One*)

lemma *one-less-numeral-iff*: $1 <$ numeral $n \iff$ *num.One* $<$ n

using *numeral-less-iff* [*of num.One n*] **by** (*simp add*: *numeral-One*)

lemma *zero-le-numeral*: $0 \leq$ numeral n

using *dual-order.trans one-le-numeral zero-le-one* **by** *blast*

lemma *zero-less-numeral*: $0 < \text{numeral } n$
using *less-linear not-numeral-less-one order.strict-trans zero-less-one* **by** *blast*

lemma *not-numeral-le-zero*: $\neg \text{numeral } n \leq 0$
by (*simp add: not-le zero-less-numeral*)

lemma *not-numeral-less-zero*: $\neg \text{numeral } n < 0$
by (*simp add: not-less zero-le-numeral*)

lemmas *le-numeral-extra* =
zero-le-one not-one-le-zero
order-refl [of 0] order-refl [of 1]

lemmas *less-numeral-extra* =
zero-less-one not-one-less-zero
less-irrefl [of 0] less-irrefl [of 1]

lemmas *le-numeral-simps [simp]* =
numeral-le-iff
one-le-numeral
numeral-le-one-iff
zero-le-numeral
not-numeral-le-zero

lemmas *less-numeral-simps [simp]* =
numeral-less-iff
one-less-numeral-iff
not-numeral-less-one
zero-less-numeral
not-numeral-less-zero

lemma *min-0-1 [simp]*:
fixes *min'* :: $'a \Rightarrow 'a \Rightarrow 'a$
defines *min'* $\equiv \text{min}$
shows
 $\text{min}' 0 1 = 0$
 $\text{min}' 1 0 = 0$
 $\text{min}' 0 (\text{numeral } x) = 0$
 $\text{min}' (\text{numeral } x) 0 = 0$
 $\text{min}' 1 (\text{numeral } x) = 1$
 $\text{min}' (\text{numeral } x) 1 = 1$
by (*simp-all add: min'-def min-def le-num-One-iff*)

lemma *max-0-1 [simp]*:
fixes *max'* :: $'a \Rightarrow 'a \Rightarrow 'a$
defines *max'* $\equiv \text{max}$
shows
 $\text{max}' 0 1 = 1$

```

max' 1 0 = 1
max' 0 (numeral x) = numeral x
max' (numeral x) 0 = numeral x
max' 1 (numeral x) = numeral x
max' (numeral x) 1 = numeral x
by (simp-all add: max'-def max-def le-num-One-iff)

```

end

Unfold *min* and *max* on numerals.

```

lemmas max-number-of [simp] =
  max-def [of numeral u numeral v]
  max-def [of numeral u - numeral v]
  max-def [of - numeral u numeral v]
  max-def [of - numeral u - numeral v] for u v

```

```

lemmas min-number-of [simp] =
  min-def [of numeral u numeral v]
  min-def [of numeral u - numeral v]
  min-def [of - numeral u numeral v]
  min-def [of - numeral u - numeral v] for u v

```

46.4.7 Multiplication and negation: class *ring-1*

```

context ring-1
begin

```

```

subclass neg-numeral ..

```

```

lemma mult-neg-numeral-simps:
  - numeral m * - numeral n = numeral (m * n)
  - numeral m * numeral n = - numeral (m * n)
  numeral m * - numeral n = - numeral (m * n)
by (simp-all only: mult-minus-left mult-minus-right minus-minus numeral-mult)

```

```

lemma mult-minus1 [simp]: - 1 * z = - z
by (simp add: numeral.simps)

```

```

lemma mult-minus1-right [simp]: z * - 1 = - z
by (simp add: numeral.simps)

```

```

lemma minus-sub-one-diff-one [simp]:
  <- sub m One - 1 = - numeral m>
proof -
  have <sub m One + 1 = numeral m>
    by (simp flip: eq-diff-eq add: diff-numeral-special)
  then have <- (sub m One + 1) = - numeral m>
    by simp
  then show ?thesis

```

by *simp*
qed

end

46.4.8 Equality using *iszero* for rings with non-zero characteristic

context *ring-1*

begin

definition *iszero* :: 'a \Rightarrow bool
where *iszero* z \longleftrightarrow z = 0

lemma *iszero-0* [*simp*]: *iszero* 0
by (*simp* add: *iszero-def*)

lemma *not-iszero-1* [*simp*]: \neg *iszero* 1
by (*simp* add: *iszero-def*)

lemma *not-iszero-Numeral1*: \neg *iszero* *Numeral1*
by (*simp* add: *numeral-One*)

lemma *not-iszero-neg-1* [*simp*]: \neg *iszero* (- 1)
by (*simp* add: *iszero-def*)

lemma *not-iszero-neg-Numeral1*: \neg *iszero* (- *Numeral1*)
by (*simp* add: *numeral-One*)

lemma *iszero-neg-numeral* [*simp*]: *iszero* (- numeral w) \longleftrightarrow *iszero* (numeral w)
unfolding *iszero-def* by (*rule* *neg-equal-0-iff-equal*)

lemma *eq-iff-iszero-diff*: x = y \longleftrightarrow *iszero* (x - y)
unfolding *iszero-def* by (*rule* *eq-iff-diff-eq-0*)

The *eq-numeral-iff-iszero* lemmas are not declared [*simp*] by default, because for rings of characteristic zero, better *simp* rules are possible. For a type like integers mod *n*, type-instantiated versions of these rules should be added to the simplifier, along with a type-specific rule for deciding propositions of the form *iszero* (numeral w).

bh: Maybe it would not be so bad to just declare these as *simp* rules anyway? I should test whether these rules take precedence over the *ring-char-0* rules in the simplifier.

lemma *eq-numeral-iff-iszero*:
numeral x = numeral y \longleftrightarrow *iszero* (sub x y)
numeral x = - numeral y \longleftrightarrow *iszero* (numeral (x + y))
- numeral x = numeral y \longleftrightarrow *iszero* (numeral (x + y))
- numeral x = - numeral y \longleftrightarrow *iszero* (sub y x)
numeral x = 1 \longleftrightarrow *iszero* (sub x One)

```

1 = numeral y  $\longleftrightarrow$  iszero (sub One y)
- numeral x = 1  $\longleftrightarrow$  iszero (numeral (x + One))
1 = - numeral y  $\longleftrightarrow$  iszero (numeral (One + y))
numeral x = 0  $\longleftrightarrow$  iszero (numeral x)
0 = numeral y  $\longleftrightarrow$  iszero (numeral y)
- numeral x = 0  $\longleftrightarrow$  iszero (numeral x)
0 = - numeral y  $\longleftrightarrow$  iszero (numeral y)
unfolding eq-iff-iszero-diff diff-numeral-simps diff-numeral-special
by simp-all

```

end

46.4.9 Equality and negation: class *ring-char-0*

context *ring-char-0*

begin

lemma *not-iszero-numeral* [*simp*]: \neg iszero (numeral *w*)
by (*simp add: iszero-def*)

lemma *neg-numeral-eq-iff*: $-$ numeral *m* = $-$ numeral *n* \longleftrightarrow *m* = *n*
by *simp*

lemma *numeral-neq-neg-numeral*: numeral *m* \neq $-$ numeral *n*
by (*simp add: eq-neg-iff-add-eq-0 numeral-plus-numeral*)

lemma *neg-numeral-neq-numeral*: $-$ numeral *m* \neq numeral *n*
by (*rule numeral-neq-neg-numeral [symmetric]*)

lemma *zero-neq-neg-numeral*: 0 \neq $-$ numeral *n*
by *simp*

lemma *neg-numeral-neq-zero*: $-$ numeral *n* \neq 0
by *simp*

lemma *one-neq-neg-numeral*: 1 \neq $-$ numeral *n*
using *numeral-neq-neg-numeral [of One n]* **by** (*simp add: numeral-One*)

lemma *neg-numeral-neq-one*: $-$ numeral *n* \neq 1
using *neg-numeral-neq-numeral [of n One]* **by** (*simp add: numeral-One*)

lemma *neg-one-neq-numeral*: $-$ 1 \neq numeral *n*
using *neg-numeral-neq-numeral [of One n]* **by** (*simp add: numeral-One*)

lemma *numeral-neq-neg-one*: numeral *n* \neq $-$ 1
using *numeral-neq-neg-numeral [of n One]* **by** (*simp add: numeral-One*)

lemma *neg-one-eq-numeral-iff*: $-$ 1 = $-$ numeral *n* \longleftrightarrow *n* = One
using *neg-numeral-eq-iff [of One n]* **by** (*auto simp add: numeral-One*)

```

lemma numeral-eq-neg-one-iff:  $- \text{numeral } n = - 1 \longleftrightarrow n = \text{One}$ 
  using neg-numeral-eq-iff [of n One] by (auto simp add: numeral-One)

lemma neg-one-neq-zero:  $- 1 \neq 0$ 
  by simp

lemma zero-neq-neg-one:  $0 \neq - 1$ 
  by simp

lemma neg-one-neq-one:  $- 1 \neq 1$ 
  using neg-numeral-neq-numeral [of One One] by (simp only: numeral-One not-False-eq-True)

lemma one-neq-neg-one:  $1 \neq - 1$ 
  using numeral-neq-neg-numeral [of One One] by (simp only: numeral-One not-False-eq-True)

lemmas eq-neg-numeral-simps [simp] =
  neg-numeral-eq-iff
  numeral-neq-neg-numeral neg-numeral-neq-numeral
  one-neq-neg-numeral neg-numeral-neq-one
  zero-neq-neg-numeral neg-numeral-neq-zero
  neg-one-neq-numeral numeral-neq-neg-one
  neg-one-eq-numeral-iff numeral-eq-neg-one-iff
  neg-one-neq-zero zero-neq-neg-one
  neg-one-neq-one one-neq-neg-one

end

```

46.4.10 Structures with negation and order: class *linordered-idom*

```

context linordered-idom

```

```

begin

```

```

subclass ring-char-0 ..

```

```

lemma neg-numeral-le-iff:  $- \text{numeral } m \leq - \text{numeral } n \longleftrightarrow n \leq m$ 
  by (simp only: neg-le-iff-le numeral-le-iff)

```

```

lemma neg-numeral-less-iff:  $- \text{numeral } m < - \text{numeral } n \longleftrightarrow n < m$ 
  by (simp only: neg-less-iff-less numeral-less-iff)

```

```

lemma neg-numeral-less-zero:  $- \text{numeral } n < 0$ 
  by (simp only: neg-less-0-iff-less zero-less-numeral)

```

```

lemma neg-numeral-le-zero:  $- \text{numeral } n \leq 0$ 
  by (simp only: neg-le-0-iff-le zero-le-numeral)

```

```

lemma not-zero-less-neg-numeral:  $\neg 0 < - \text{numeral } n$ 
  by (simp only: not-less neg-numeral-le-zero)

```

lemma *not-zero-le-neg-numeral*: $\neg 0 \leq - \text{numeral } n$
by (*simp only: not-le neg-numeral-less-zero*)

lemma *neg-numeral-less-numeral*: $- \text{numeral } m < \text{numeral } n$
using *neg-numeral-less-zero zero-less-numeral* **by** (*rule less-trans*)

lemma *neg-numeral-le-numeral*: $- \text{numeral } m \leq \text{numeral } n$
by (*simp only: less-imp-le neg-numeral-less-numeral*)

lemma *not-numeral-less-neg-numeral*: $\neg \text{numeral } m < - \text{numeral } n$
by (*simp only: not-less neg-numeral-le-numeral*)

lemma *not-numeral-le-neg-numeral*: $\neg \text{numeral } m \leq - \text{numeral } n$
by (*simp only: not-le neg-numeral-less-numeral*)

lemma *neg-numeral-less-one*: $- \text{numeral } m < 1$
by (*rule neg-numeral-less-numeral [of m One, unfolded numeral-One]*)

lemma *neg-numeral-le-one*: $- \text{numeral } m \leq 1$
by (*rule neg-numeral-le-numeral [of m One, unfolded numeral-One]*)

lemma *not-one-less-neg-numeral*: $\neg 1 < - \text{numeral } m$
by (*simp only: not-less neg-numeral-le-one*)

lemma *not-one-le-neg-numeral*: $\neg 1 \leq - \text{numeral } m$
by (*simp only: not-le neg-numeral-less-one*)

lemma *not-numeral-less-neg-one*: $\neg \text{numeral } m < - 1$
using *not-numeral-less-neg-numeral [of m One]* **by** (*simp add: numeral-One*)

lemma *not-numeral-le-neg-one*: $\neg \text{numeral } m \leq - 1$
using *not-numeral-le-neg-numeral [of m One]* **by** (*simp add: numeral-One*)

lemma *neg-one-less-numeral*: $- 1 < \text{numeral } m$
using *neg-numeral-less-numeral [of One m]* **by** (*simp add: numeral-One*)

lemma *neg-one-le-numeral*: $- 1 \leq \text{numeral } m$
using *neg-numeral-le-numeral [of One m]* **by** (*simp add: numeral-One*)

lemma *neg-numeral-less-neg-one-iff*: $- \text{numeral } m < - 1 \iff m \neq \text{One}$
by (*cases m*) *simp-all*

lemma *neg-numeral-le-neg-one*: $- \text{numeral } m \leq - 1$
by *simp*

lemma *not-neg-one-less-neg-numeral*: $\neg - 1 < - \text{numeral } m$
by *simp*

lemma *not-neg-one-le-neg-numeral-iff*: $\neg - 1 \leq - \text{numeral } m \longleftrightarrow m \neq \text{One}$
by (*cases m*) *simp-all*

lemma *sub-non-negative*: $\text{sub } n \ m \geq 0 \longleftrightarrow n \geq m$
by (*simp only: sub-def le-diff-eq*) *simp*

lemma *sub-positive*: $\text{sub } n \ m > 0 \longleftrightarrow n > m$
by (*simp only: sub-def less-diff-eq*) *simp*

lemma *sub-non-positive*: $\text{sub } n \ m \leq 0 \longleftrightarrow n \leq m$
by (*simp only: sub-def diff-le-eq*) *simp*

lemma *sub-negative*: $\text{sub } n \ m < 0 \longleftrightarrow n < m$
by (*simp only: sub-def diff-less-eq*) *simp*

lemmas *le-neg-numeral-simps* [*simp*] =
neg-numeral-le-iff
neg-numeral-le-numeral not-numeral-le-neg-numeral
neg-numeral-le-zero not-zero-le-neg-numeral
neg-numeral-le-one not-one-le-neg-numeral
neg-one-le-numeral not-numeral-le-neg-one
neg-numeral-le-neg-one not-neg-one-le-neg-numeral-iff

lemma *le-minus-one-simps* [*simp*]:
 $- 1 \leq 0$
 $- 1 \leq 1$
 $\neg 0 \leq - 1$
 $\neg 1 \leq - 1$
by *simp-all*

lemmas *less-neg-numeral-simps* [*simp*] =
neg-numeral-less-iff
neg-numeral-less-numeral not-numeral-less-neg-numeral
neg-numeral-less-zero not-zero-less-neg-numeral
neg-numeral-less-one not-one-less-neg-numeral
neg-one-less-numeral not-numeral-less-neg-one
neg-numeral-less-neg-one-iff not-neg-one-less-neg-numeral

lemma *less-minus-one-simps* [*simp*]:
 $- 1 < 0$
 $- 1 < 1$
 $\neg 0 < - 1$
 $\neg 1 < - 1$
by (*simp-all add: less-le*)

lemma *abs-numeral* [*simp*]: $|\text{numeral } n| = \text{numeral } n$
by *simp*

lemma *abs-neg-numeral* [*simp*]: $|- \text{numeral } n| = \text{numeral } n$

by (*simp only: abs-minus-cancel abs-numeral*)

lemma *abs-neg-one* [*simp*]: $|- 1| = 1$
by *simp*

end

46.4.11 Natural numbers

lemma *numeral-num-of-nat*:
numeral (num-of-nat n) = n if n > 0
using *that nat-of-num-numeral num-of-nat-inverse* **by** *simp*

lemma *Suc-1* [*simp*]: $Suc\ 1 = 2$
unfolding *Suc-eq-plus1* **by** (*rule one-add-one*)

lemma *Suc-numeral* [*simp*]: $Suc\ (numeral\ n) = numeral\ (n + One)$
unfolding *Suc-eq-plus1* **by** (*rule numeral-plus-one*)

definition *pred-numeral* :: $num \Rightarrow nat$
where *pred-numeral k = numeral k - 1*

declare [[*code drop: pred-numeral*]]

lemma *numeral-eq-Suc*: $numeral\ k = Suc\ (pred-numeral\ k)$
by (*simp add: pred-numeral-def*)

lemma *eval-nat-numeral*:
numeral One = Suc 0
numeral (Bit0 n) = Suc (numeral (BitM n))
numeral (Bit1 n) = Suc (numeral (Bit0 n))
by (*simp-all add: numeral.simps BitM-plus-one*)

lemma *pred-numeral-simps* [*simp*]:
pred-numeral One = 0
pred-numeral (Bit0 k) = numeral (BitM k)
pred-numeral (Bit1 k) = numeral (Bit0 k)
by (*simp-all only: pred-numeral-def eval-nat-numeral diff-Suc-Suc diff-0*)

lemma *pred-numeral-inc* [*simp*]:
pred-numeral (Num.inc k) = numeral k
by (*simp only: pred-numeral-def numeral-inc diff-add-inverse2*)

lemma *numeral-2-eq-2*: $2 = Suc\ (Suc\ 0)$
by (*simp add: eval-nat-numeral*)

lemma *numeral-3-eq-3*: $3 = Suc\ (Suc\ (Suc\ 0))$
by (*simp add: eval-nat-numeral*)

lemma *numeral-1-eq-Suc-0*: $\text{Numeral1} = \text{Suc } 0$
by (*simp only: numeral-One One-nat-def*)

lemma *Suc-nat-number-of-add*: $\text{Suc} (\text{numeral } v + n) = \text{numeral} (v + \text{One}) + n$
by *simp*

lemma *numerals*: $\text{Numeral1} = (1::\text{nat}) 2 = \text{Suc} (\text{Suc } 0)$
by (*rule numeral-One (rule numeral-2-eq-2)*)

lemmas *numeral-nat* = *eval-nat-numeral BitM.simps One-nat-def*

Comparisons involving *Suc*.

lemma *eq-numeral-Suc* [*simp*]: $\text{numeral } k = \text{Suc } n \longleftrightarrow \text{pred-numeral } k = n$
by (*simp add: numeral-eq-Suc*)

lemma *Suc-eq-numeral* [*simp*]: $\text{Suc } n = \text{numeral } k \longleftrightarrow n = \text{pred-numeral } k$
by (*simp add: numeral-eq-Suc*)

lemma *less-numeral-Suc* [*simp*]: $\text{numeral } k < \text{Suc } n \longleftrightarrow \text{pred-numeral } k < n$
by (*simp add: numeral-eq-Suc*)

lemma *less-Suc-numeral* [*simp*]: $\text{Suc } n < \text{numeral } k \longleftrightarrow n < \text{pred-numeral } k$
by (*simp add: numeral-eq-Suc*)

lemma *le-numeral-Suc* [*simp*]: $\text{numeral } k \leq \text{Suc } n \longleftrightarrow \text{pred-numeral } k \leq n$
by (*simp add: numeral-eq-Suc*)

lemma *le-Suc-numeral* [*simp*]: $\text{Suc } n \leq \text{numeral } k \longleftrightarrow n \leq \text{pred-numeral } k$
by (*simp add: numeral-eq-Suc*)

lemma *diff-Suc-numeral* [*simp*]: $\text{Suc } n - \text{numeral } k = n - \text{pred-numeral } k$
by (*simp add: numeral-eq-Suc*)

lemma *diff-numeral-Suc* [*simp*]: $\text{numeral } k - \text{Suc } n = \text{pred-numeral } k - n$
by (*simp add: numeral-eq-Suc*)

lemma *max-Suc-numeral* [*simp*]: $\max (\text{Suc } n) (\text{numeral } k) = \text{Suc} (\max n (\text{pred-numeral } k))$
by (*simp add: numeral-eq-Suc*)

lemma *max-numeral-Suc* [*simp*]: $\max (\text{numeral } k) (\text{Suc } n) = \text{Suc} (\max (\text{pred-numeral } k) n)$
by (*simp add: numeral-eq-Suc*)

lemma *min-Suc-numeral* [*simp*]: $\min (\text{Suc } n) (\text{numeral } k) = \text{Suc} (\min n (\text{pred-numeral } k))$
by (*simp add: numeral-eq-Suc*)

lemma *min-numeral-Suc* [*simp*]: $\min (\text{numeral } k) (\text{Suc } n) = \text{Suc} (\min (\text{pred-numeral } k) n)$

k) n)
by (*simp add: numeral-eq-Suc*)

For *case-nat* and *rec-nat*.

lemma *case-nat-numeral* [*simp*]: $\text{case-nat } a \ f \ (\text{numeral } v) = (\text{let } pv = \text{pred-numeral } v \ \text{in } f \ pv)$
by (*simp add: numeral-eq-Suc*)

lemma *case-nat-add-eq-if* [*simp*]:
 $\text{case-nat } a \ f \ ((\text{numeral } v) + n) = (\text{let } pv = \text{pred-numeral } v \ \text{in } f \ (pv + n))$
by (*simp add: numeral-eq-Suc*)

lemma *rec-nat-numeral* [*simp*]:
 $\text{rec-nat } a \ f \ (\text{numeral } v) = (\text{let } pv = \text{pred-numeral } v \ \text{in } f \ pv \ (\text{rec-nat } a \ f \ pv))$
by (*simp add: numeral-eq-Suc Let-def*)

lemma *rec-nat-add-eq-if* [*simp*]:
 $\text{rec-nat } a \ f \ (\text{numeral } v + n) = (\text{let } pv = \text{pred-numeral } v \ \text{in } f \ (pv + n) \ (\text{rec-nat } a \ f \ (pv + n)))$
by (*simp add: numeral-eq-Suc Let-def*)

Case analysis on $n < (2::'a)$.

lemma *less-2-cases*: $n < 2 \implies n = 0 \vee n = \text{Suc } 0$
by (*auto simp add: numeral-2-eq-2*)

lemma *less-2-cases-iff*: $n < 2 \iff n = 0 \vee n = \text{Suc } 0$
by (*auto simp add: numeral-2-eq-2*)

Removal of Small Numerals: 0, 1 and (in additive positions) 2.

bh: Are these rules really a good idea? LCP: well, it already happens for 0 and 1!

lemma *add-2-eq-Suc* [*simp*]: $2 + n = \text{Suc } (\text{Suc } n)$
by *simp*

lemma *add-2-eq-Suc'* [*simp*]: $n + 2 = \text{Suc } (\text{Suc } n)$
by *simp*

Can be used to eliminate long strings of Sucs, but not by default.

lemma *Suc3-eq-add-3*: $\text{Suc } (\text{Suc } (\text{Suc } n)) = 3 + n$
by *simp*

lemmas *nat-1-add-1 = one-add-one* [**where** $'a = \text{nat}$]

context *semiring-numeral*
begin

lemma *numeral-add-unfold-funpow*:

```

  ⟨numeral k + a = ((+) 1 ~ numeral k) a⟩
proof (rule sym, induction k arbitrary: a)
  case One
  then show ?case
    by (simp add: numeral-One Num.numeral-One)
next
  case (Bit0 k)
  then show ?case
    by (simp add: numeral-Bit0 Num.numeral-Bit0 ac-simps funpow-add)
next
  case (Bit1 k)
  then show ?case
    by (simp add: numeral-Bit1 Num.numeral-Bit1 ac-simps funpow-add)
qed

end

context semiring-1
begin

lemma numeral-unfold-funpow:
  ⟨numeral k = ((+) 1 ~ numeral k) 0⟩
  using numeral-add-unfold-funpow [of k 0] by simp

end

context
  includes lifting-syntax
begin

lemma transfer-rule-numeral:
  ⟨((=) == => R) numeral numeral⟩
  if [transfer-rule]: ⟨R 0 0⟩ ⟨R 1 1⟩
  ⟨(R == => R == => R) (+) (+)⟩
  for R :: ⟨'a::{semiring-numeral,monoid-add} => 'b::{semiring-numeral,monoid-add}
=> bool⟩
proof –
  have ((=) == => R) (λk. ((+) 1 ~ numeral k) 0) (λk. ((+) 1 ~ numeral k)
0)
  by transfer-prover
  moreover have ⟨numeral = (λk. ((+) (1::'a) ~ numeral k) 0)⟩
  using numeral-add-unfold-funpow [where ?'a = 'a, of - 0]
  by (simp add: fun-eq-iff)
  moreover have ⟨numeral = (λk. ((+) (1::'b) ~ numeral k) 0)⟩
  using numeral-add-unfold-funpow [where ?'a = 'b, of - 0]
  by (simp add: fun-eq-iff)
  ultimately show ?thesis
  by simp
qed

```

end

46.5 Particular lemmas concerning $\mathbb{Z}::'a$

context *linordered-field*
begin

subclass *field-char-0* ..

lemma *half-gt-zero-iff*: $0 < a / 2 \iff 0 < a$
by (*auto simp add: field-simps*)

lemma *half-gt-zero* [*simp*]: $0 < a \implies 0 < a / 2$
by (*simp add: half-gt-zero-iff*)

end

46.6 Numeral equations as default simplification rules

declare (in *numeral*) *numeral-One* [*simp*]
declare (in *numeral*) *numeral-plus-numeral* [*simp*]
declare (in *numeral*) *add-numeral-special* [*simp*]
declare (in *neg-numeral*) *add-neg-numeral-simps* [*simp*]
declare (in *neg-numeral*) *add-neg-numeral-special* [*simp*]
declare (in *neg-numeral*) *diff-numeral-simps* [*simp*]
declare (in *neg-numeral*) *diff-numeral-special* [*simp*]
declare (in *semiring-numeral*) *numeral-times-numeral* [*simp*]
declare (in *ring-1*) *mult-neg-numeral-simps* [*simp*]

46.6.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

lemmas *distrib-right-numeral* [*simp*] = *distrib-right* [of - - numeral *v*] **for** *v*
lemmas *distrib-left-numeral* [*simp*] = *distrib-left* [of numeral *v*] **for** *v*
lemmas *left-diff-distrib-numeral* [*simp*] = *left-diff-distrib* [of - - numeral *v*] **for** *v*
lemmas *right-diff-distrib-numeral* [*simp*] = *right-diff-distrib* [of numeral *v*] **for** *v*

These are actually for fields, like real

lemmas *zero-less-divide-iff-numeral* [*simp, no-atp*] = *zero-less-divide-iff* [of numeral *w*] **for** *w*
lemmas *divide-less-0-iff-numeral* [*simp, no-atp*] = *divide-less-0-iff* [of numeral *w*] **for** *w*
lemmas *zero-le-divide-iff-numeral* [*simp, no-atp*] = *zero-le-divide-iff* [of numeral *w*] **for** *w*
lemmas *divide-le-0-iff-numeral* [*simp, no-atp*] = *divide-le-0-iff* [of numeral *w*] **for** *w*

Replaces *inverse #nn* by $1/\#nn$. It looks strange, but then other simprocs simplify the quotient.

lemmas *inverse-eq-divide-numeral* [*simp*] =
inverse-eq-divide [of numeral *w*] **for** *w*

lemmas *inverse-eq-divide-neg-numeral* [*simp*] =
inverse-eq-divide [of - numeral *w*] **for** *w*

These laws simplify inequalities, moving unary minus from a term into the literal.

lemmas *equation-minus-iff-numeral* [*no-atp*] =
equation-minus-iff [of numeral *v*] **for** *v*

lemmas *minus-equation-iff-numeral* [*no-atp*] =
minus-equation-iff [of - numeral *v*] **for** *v*

lemmas *le-minus-iff-numeral* [*no-atp*] =
le-minus-iff [of numeral *v*] **for** *v*

lemmas *minus-le-iff-numeral* [*no-atp*] =
minus-le-iff [of - numeral *v*] **for** *v*

lemmas *less-minus-iff-numeral* [*no-atp*] =
less-minus-iff [of numeral *v*] **for** *v*

lemmas *minus-less-iff-numeral* [*no-atp*] =
minus-less-iff [of - numeral *v*] **for** *v*

Cancellation of constant factors in comparisons ($<$ and \leq)

lemmas *mult-less-cancel-left-numeral* [*simp, no-atp*] = *mult-less-cancel-left* [of numeral *v*] **for** *v*

lemmas *mult-less-cancel-right-numeral* [*simp, no-atp*] = *mult-less-cancel-right* [of - numeral *v*] **for** *v*

lemmas *mult-le-cancel-left-numeral* [*simp, no-atp*] = *mult-le-cancel-left* [of numeral *v*] **for** *v*

lemmas *mult-le-cancel-right-numeral* [*simp, no-atp*] = *mult-le-cancel-right* [of - numeral *v*] **for** *v*

Multiplying out constant divisors in comparisons ($<$, \leq and $=$)

named-theorems *divide-const-simps* simplification rules to simplify comparisons involving constant divisors

lemmas *le-divide-eq-numeral1* [*simp, divide-const-simps*] =
pos-le-divide-eq [of numeral *w*, OF zero-less-numeral]
neg-le-divide-eq [of - numeral *w*, OF neg-numeral-less-zero] **for** *w*

lemmas *divide-le-eq-numeral1* [*simp, divide-const-simps*] =
pos-divide-le-eq [of numeral *w*, OF zero-less-numeral]

neg-divide-le-eq [of - numeral w , OF *neg-numeral-less-zero*] **for** w

lemmas *less-divide-eq-numeral1* [*simp, divide-const-simps*] =
pos-less-divide-eq [of numeral w , OF *zero-less-numeral*]
neg-less-divide-eq [of - numeral w , OF *neg-numeral-less-zero*] **for** w

lemmas *divide-less-eq-numeral1* [*simp, divide-const-simps*] =
pos-divide-less-eq [of numeral w , OF *zero-less-numeral*]
neg-divide-less-eq [of - numeral w , OF *neg-numeral-less-zero*] **for** w

lemmas *eq-divide-eq-numeral1* [*simp, divide-const-simps*] =
eq-divide-eq [of - - numeral w]
eq-divide-eq [of - - - numeral w] **for** w

lemmas *divide-eq-eq-numeral1* [*simp, divide-const-simps*] =
divide-eq-eq [of - numeral w]
divide-eq-eq [of - - numeral w] **for** w

46.6.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas *le-divide-eq-numeral* [*divide-const-simps*] =
le-divide-eq [of numeral w]
le-divide-eq [of - numeral w] **for** w

lemmas *divide-le-eq-numeral* [*divide-const-simps*] =
divide-le-eq [of - - numeral w]
divide-le-eq [of - - - numeral w] **for** w

lemmas *less-divide-eq-numeral* [*divide-const-simps*] =
less-divide-eq [of numeral w]
less-divide-eq [of - numeral w] **for** w

lemmas *divide-less-eq-numeral* [*divide-const-simps*] =
divide-less-eq [of - - numeral w]
divide-less-eq [of - - - numeral w] **for** w

lemmas *eq-divide-eq-numeral* [*divide-const-simps*] =
eq-divide-eq [of numeral w]
eq-divide-eq [of - numeral w] **for** w

lemmas *divide-eq-eq-numeral* [*divide-const-simps*] =
divide-eq-eq [of - - numeral w]
divide-eq-eq [of - - - numeral w] **for** w

Not good as automatic simprules because they cause case splits.

lemmas [*divide-const-simps*] =
le-divide-eq-1 *divide-le-eq-1* *less-divide-eq-1* *divide-less-eq-1*

46.7 Setting up simprocs

```
lemma mult-numeral-1: Numeral1 * a = a
  for a :: 'a::semiring-numeral
  by simp
```

```
lemma mult-numeral-1-right: a * Numeral1 = a
  for a :: 'a::semiring-numeral
  by simp
```

```
lemma divide-numeral-1: a / Numeral1 = a
  for a :: 'a::field
  by simp
```

```
lemma inverse-numeral-1: inverse Numeral1 = (Numeral1::'a::division-ring)
  by simp
```

Theorem lists for the cancellation simprocs. The use of a binary numeral for 1 reduces the number of special cases.

```
lemma mult-1s-semiring-numeral:
  Numeral1 * a = a
  a * Numeral1 = a
  for a :: 'a::semiring-numeral
  by simp-all
```

```
lemma mult-1s-ring-1:
  - Numeral1 * b = - b
  b * - Numeral1 = - b
  for b :: 'a::ring-1
  by simp-all
```

```
lemmas mult-1s = mult-1s-semiring-numeral mult-1s-ring-1
```

```
setup <
  Reorient-Proc.add
  (fn Const (const-name <numeral>, -) $ - => true
   | Const (const-name <uminus>, -) $ (Const (const-name <numeral>, -) $ -)
=> true
   | - => false)
>
```

```
simproc-setup reorient-numeral (numeral w = x | - numeral w = y) =
  <K Reorient-Proc.proc>
```

46.7.1 Simplification of arithmetic operations on integer constants

```
lemmas arith-special =
  add-numeral-special add-neg-numeral-special
  diff-numeral-special
```

lemmas *arith-extra-simps* =
numeral-plus-numeral add-neg-numeral-simps add-0-left add-0-right
minus-zero
diff-numeral-simps diff-0 diff-0-right
numeral-times-numeral mult-neg-numeral-simps
mult-zero-left mult-zero-right
abs-numeral abs-neg-numeral

For making a minimal simpset, one must include these default simprules.
 Also include *simp-thms*.

lemmas *arith-simps* =
add-num-simps mult-num-simps sub-num-simps
BitM.simps dbl-simps dbl-inc-simps dbl-dec-simps
abs-zero abs-one arith-extra-simps

lemmas *more-arith-simps* =
neg-le-iff-le
minus-zero left-minus right-minus
mult-1-left mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult.assoc

lemmas *of-nat-simps* =
of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult

Simplification of relational operations.

lemmas *eq-numeral-extra* =
zero-neq-one one-neq-zero

lemmas *rel-simps* =
le-num-simps less-num-simps eq-num-simps
le-numeral-simps le-neg-numeral-simps le-minus-one-simps le-numeral-extra
less-numeral-simps less-neg-numeral-simps less-minus-one-simps less-numeral-extra
eq-numeral-simps eq-neg-numeral-simps eq-numeral-extra

lemma *Let-numeral [simp]: Let (numeral v) f = f (numeral v)*
 — Unfold all *lets* involving constants
unfolding *Let-def ..*

lemma *Let-neg-numeral [simp]: Let (- numeral v) f = f (- numeral v)*
 — Unfold all *lets* involving constants
unfolding *Let-def ..*

declaration <

let

fun number-of ctxt T n =
*if not (Sign.of-sort (Proof-Context.theory-of ctxt) (T, **sort** <numeral>))*
then raise CTERM (number-of, [])

```

    else Numeral.mk-cnumber (Thm.ctyp-of ctxt T) n;
in
  K (
    Lin-Arith.set-number-of number-of
    #> Lin-Arith.add-simps
    @{thms arith-simps more-arith-simps rel-simps pred-numeral-simps
      arith-special numeral-One of-nat-simps uminus-numeral-One
      Suc-numeral Let-numeral Let-neg-numeral Let-0 Let-1
      le-Suc-numeral le-numeral-Suc less-Suc-numeral less-numeral-Suc
      Suc-eq-numeral eq-numeral-Suc mult-Suc mult-Suc-right of-nat-numeral}
  end
  >

```

46.7.2 Simplification of arithmetic when nested to the right

lemma *add-numeral-left* [simp]: numeral $v + (\text{numeral } w + z) = (\text{numeral}(v + w) + z)$
by (*simp-all add: add.assoc [symmetric]*)

lemma *add-neg-numeral-left* [simp]:
 numeral $v + (- \text{numeral } w + y) = (\text{sub } v \ w + y)$
 $- \text{numeral } v + (\text{numeral } w + y) = (\text{sub } w \ v + y)$
 $- \text{numeral } v + (- \text{numeral } w + y) = (- \text{numeral}(v + w) + y)$
by (*simp-all add: add.assoc [symmetric]*)

lemma *mult-numeral-left-semiring-numeral*:
 numeral $v * (\text{numeral } w * z) = (\text{numeral}(v * w) * z :: 'a::\text{semiring-numeral})$
by (*simp add: mult.assoc [symmetric]*)

lemma *mult-numeral-left-ring-1*:
 $- \text{numeral } v * (\text{numeral } w * y) = (- \text{numeral}(v * w) * y :: 'a::\text{ring-1})$
 $\text{numeral } v * (- \text{numeral } w * y) = (- \text{numeral}(v * w) * y :: 'a::\text{ring-1})$
 $- \text{numeral } v * (- \text{numeral } w * y) = (\text{numeral}(v * w) * y :: 'a::\text{ring-1})$
by (*simp-all add: mult.assoc [symmetric]*)

lemmas *mult-numeral-left* [simp] =
mult-numeral-left-semiring-numeral
mult-numeral-left-ring-1

hide-const (**open**) *One Bit0 Bit1 BitM inc pow sqr sub dbl dbl-inc dbl-dec*

46.8 Code module namespace

code-identifier

code-module *Num* \rightarrow (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

46.9 Printing of evaluated natural numbers as numerals

lemma [*code-post*]:
Suc $0 = 1$

```

Suc 1 = 2
Suc (numeral n) = numeral (Num.inc n)
by (simp-all add: numeral-inc)

```

```

lemmas [code-post] = Num.inc.simps

```

46.10 More on auxiliary conversion

```

context semiring-1
begin

```

```

lemma numeral-num-of-nat-unfold:
  ⟨numeral (num-of-nat n) = (if n = 0 then 1 else of-nat n)⟩
  by (induction n) (simp-all add: numeral-inc ac-simps)

```

```

lemma num-of-nat-numeral-eq [simp]:

```

```

  ⟨num-of-nat (numeral q) = q⟩

```

```

proof (induction q)

```

```

  case One

```

```

    then show ?case

```

```

      by simp

```

```

  next

```

```

    case (Bit0 q)

```

```

      then have num-of-nat (numeral (num.Bit0 q)) = num-of-nat (numeral q + numeral q)

```

```

        by (simp only: Num.numeral-Bit0 Num.numeral-add)

```

```

      also have ... = num.Bit0 (num-of-nat (numeral q))

```

```

        by (rule num-of-nat-double) simp

```

```

      finally show ?case

```

```

        using Bit0.IH by simp

```

```

  next

```

```

    case (Bit1 q)

```

```

      then have num-of-nat (numeral (num.Bit1 q)) = num-of-nat (numeral q + numeral q + 1)

```

```

        by (simp only: Num.numeral-Bit1 Num.numeral-add)

```

```

      also have ... = num-of-nat (numeral q + numeral q) + num-of-nat 1

```

```

        by (rule num-of-nat-plus-distrib) auto

```

```

      also have ... = num.Bit0 (num-of-nat (numeral q)) + num-of-nat 1

```

```

        by (subst num-of-nat-double) auto

```

```

      finally show ?case

```

```

        using Bit1.IH by simp

```

```

qed

```

```

end

```

```

end

```

47 Exponentiation

```
theory Power
  imports Num
begin
```

47.1 Powers for Arbitrary Monoids

```
class power = one + times
begin
```

```
primrec power :: 'a ⇒ nat ⇒ 'a (infixr ^ 80)
  where
    power-0: a ^ 0 = 1
  | power-Suc: a ^ Suc n = a * a ^ n
```

```
notation (latex output)
  power ((-) [1000] 1000)
```

Special syntax for squares.

```
abbreviation power2 :: 'a ⇒ 'a ((-2) [1000] 999)
  where x2 ≡ x ^ 2
```

```
end
```

```
context
  includes lifting-syntax
begin
```

```
lemma power-transfer [transfer-rule]:
  ⟨(R ==> (=) ==> R) (∧) (∧)⟩
  if [transfer-rule]: ⟨R 1 1⟩
  ⟨(R ==> R ==> R) (*) (*)⟩
  for R :: ⟨'a::power ⇒ 'b::power ⇒ bool⟩
  by (simp only: power-def [abs-def]) transfer-prover
```

```
end
```

```
context monoid-mult
begin
```

```
subclass power .
```

```
lemma power-one [simp]: 1 ^ n = 1
  by (induct n) simp-all
```

```
lemma power-one-right [simp]: a ^ 1 = a
  by simp
```

```
lemma power-Suc0-right [simp]: a ^ Suc 0 = a
```

by *simp*

lemma *power-commutes*: $a \wedge n * a = a * a \wedge n$
 by (*induct n*) (*simp-all add: mult.assoc*)

lemma *power-Suc2*: $a \wedge \text{Suc } n = a \wedge n * a$
 by (*simp add: power-commutes*)

lemma *power-add*: $a \wedge (m + n) = a \wedge m * a \wedge n$
 by (*induct m*) (*simp-all add: algebra-simps*)

lemma *power-mult*: $a \wedge (m * n) = (a \wedge m) \wedge n$
 by (*induct n*) (*simp-all add: power-add*)

lemma *power-even-eq*: $a \wedge (2 * n) = (a \wedge n)^2$
 by (*subst mult.commute*) (*simp add: power-mult*)

lemma *power-odd-eq*: $a \wedge \text{Suc } (2 * n) = a * (a \wedge n)^2$
 by (*simp add: power-even-eq*)

lemma *power-numeral-even*: $z \wedge \text{numeral } (\text{Num.Bit0 } w) = (\text{let } w = z \wedge (\text{numeral } w) \text{ in } w * w)$
 by (*simp only: numeral-Bit0 power-add Let-def*)

lemma *power-numeral-odd*: $z \wedge \text{numeral } (\text{Num.Bit1 } w) = (\text{let } w = z \wedge (\text{numeral } w) \text{ in } z * w * w)$
 by (*simp only: numeral-Bit1 One-nat-def add-Suc-right add-0-right power-Suc power-add Let-def mult.assoc*)

lemma *power2-eq-square*: $a^2 = a * a$
 by (*simp add: numeral-2-eq-2*)

lemma *power3-eq-cube*: $a \wedge 3 = a * a * a$
 by (*simp add: numeral-3-eq-3 mult.assoc*)

lemma *power4-eq-xxxx*: $x \wedge 4 = x * x * x * x$
 by (*simp add: mult.assoc power-numeral-even*)

lemma *power-numeral-reduce*: $x \wedge \text{numeral } n = x * x \wedge \text{pred-numeral } n$
 by (*simp add: numeral-eq-Suc*)

lemma *funpow-times-power*: $(\text{times } x \wedge\wedge f x) = \text{times } (x \wedge f x)$

proof (*induct f x arbitrary: f*)

case 0

then show ?case by (*simp add: fun-eq-iff*)

next

case (*Suc n*)

define *g* where $g x = f x - 1$ for *x*

with *Suc* have $n = g x$ by *simp*

with *Suc* **have** $x \hat{=} g x = \text{times } (x \hat{=} g x)$ **by** *simp*
moreover from *Suc g-def* **have** $f x = g x + 1$ **by** *simp*
ultimately show *?case*
by (*simp add: power-add funpow-add fun-eq-iff mult.assoc*)
qed

lemma *power-commuting-commutes*:

assumes $x * y = y * x$
shows $x \hat{=} n * y = y * x \hat{=} n$
proof (*induct n*)
case 0
then show *?case* **by** *simp*
next
case (*Suc n*)
have $x \hat{=} \text{Suc } n * y = x \hat{=} n * y * x$
by (*subst power-Suc2*) (*simp add: assms ac-simps*)
also have $\dots = y * x \hat{=} \text{Suc } n$
by (*simp only: Suc power-Suc2*) (*simp add: ac-simps*)
finally show *?case* .
qed

lemma *power-minus-mult*: $0 < n \implies a \hat{=} (n - 1) * a = a \hat{=} n$
by (*simp add: power-commutes split: nat-diff-split*)

lemma *left-right-inverse-power*:

assumes $x * y = 1$
shows $x \hat{=} n * y \hat{=} n = 1$
proof (*induct n*)
case (*Suc n*)
moreover have $x \hat{=} \text{Suc } n * y \hat{=} \text{Suc } n = x \hat{=} n * (x * y) * y \hat{=} n$
by (*simp add: power-Suc2[symmetric] mult.assoc[symmetric]*)
ultimately show *?case* **by** (*simp add: assms*)
qed *simp*

end

context *comm-monoid-mult*

begin

lemma *power-mult-distrib* [*algebra-simps, algebra-split-simps, field-simps, field-split-simps, divide-simps*]:

$(a * b) \hat{=} n = (a \hat{=} n) * (b \hat{=} n)$
by (*induction n*) (*simp-all add: ac-simps*)

end

Extract constant factors from powers.

declare *power-mult-distrib* [**where** $a = \text{numeral } w$ **for** w , *simp*]

declare *power-mult-distrib* [**where** $b = \text{numeral } w$ **for** w , *simp*]

lemma *power-add-numeral* [*simp*]: $a^{\text{numeral } m} * a^{\text{numeral } n} = a^{\text{numeral } (m + n)}$

for $a :: 'a::\text{monoid-mult}$
by (*simp add: power-add [symmetric]*)

lemma *power-add-numeral2* [*simp*]: $a^{\text{numeral } m} * (a^{\text{numeral } n} * b) = a^{\text{numeral } (m + n)} * b$

for $a :: 'a::\text{monoid-mult}$
by (*simp add: mult.assoc [symmetric]*)

lemma *power-mult-numeral* [*simp*]: $(a^{\text{numeral } m})^{\text{numeral } n} = a^{\text{numeral } (m * n)}$

for $a :: 'a::\text{monoid-mult}$
by (*simp only: numeral-mult power-mult*)

context *semiring-numeral*

begin

lemma *numeral-sqr*: $\text{numeral } (\text{Num.sqr } k) = \text{numeral } k * \text{numeral } k$

by (*simp only: sqr-conv-mult numeral-mult*)

lemma *numeral-pow*: $\text{numeral } (\text{Num.pow } k \ l) = \text{numeral } k^{\text{numeral } l}$

by (*induct l*)
(simp-all only: numeral-class.numeral.simps pow.simps numeral-sqr numeral-mult power-add power-one-right)

lemma *power-numeral* [*simp*]: $\text{numeral } k^{\text{numeral } l} = \text{numeral } (\text{Num.pow } k \ l)$

by (*rule numeral-pow [symmetric]*)

end

context *semiring-1*

begin

lemma *of-nat-power* [*simp*]: $\text{of-nat } (m^{\text{nat } n}) = \text{of-nat } m^{\text{nat } n}$

by (*induct n simp-all*)

lemma *zero-power*: $0 < n \implies 0^{\text{nat } n} = 0$

by (*cases n simp-all*)

lemma *power-zero-numeral* [*simp*]: $0^{\text{numeral } k} = 0$

by (*simp add: numeral-eq-Suc*)

lemma *zero-power2*: $0^2 = 0$

by (*rule power-zero-numeral*)

lemma *one-power2*: $1^2 = 1$

by (*rule power-one*)

lemma *power-0-Suc* [*simp*]: $0 \wedge \text{Suc } n = 0$
by *simp*

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: $0 \wedge n = (\text{if } n = 0 \text{ then } 1 \text{ else } 0)$
by (*cases n*) *simp-all*

end

context *semiring-char-0* **begin**

lemma *numeral-power-eq-of-nat-cancel-iff* [*simp*]:
 $\text{numeral } x \wedge n = \text{of-nat } y \iff \text{numeral } x \wedge n = y$
using *of-nat-eq-iff* **by** *fastforce*

lemma *real-of-nat-eq-numeral-power-cancel-iff* [*simp*]:
 $\text{of-nat } y = \text{numeral } x \wedge n \iff y = \text{numeral } x \wedge n$
using *numeral-power-eq-of-nat-cancel-iff* [*of x n y*] **by** (*metis (mono-tags)*)

lemma *of-nat-eq-of-nat-power-cancel-iff* [*simp*]: $(\text{of-nat } b) \wedge w = \text{of-nat } x \iff b \wedge w = x$
by (*metis of-nat-power of-nat-eq-iff*)

lemma *of-nat-power-eq-of-nat-cancel-iff* [*simp*]: $\text{of-nat } x = (\text{of-nat } b) \wedge w \iff x = b \wedge w$
by (*metis of-nat-eq-of-nat-power-cancel-iff*)

end

context *comm-semiring-1*
begin

The divides relation.

lemma *le-imp-power-dvd*:
assumes $m \leq n$
shows $a \wedge m \text{ dvd } a \wedge n$

proof

from *assms* **have** $a \wedge n = a \wedge (m + (n - m))$ **by** *simp*
also have $\dots = a \wedge m * a \wedge (n - m)$ **by** (*rule power-add*)
finally show $a \wedge n = a \wedge m * a \wedge (n - m)$.

qed

lemma *power-le-dvd*: $a \wedge n \text{ dvd } b \implies m \leq n \implies a \wedge m \text{ dvd } b$
by (*rule dvd-trans [OF le-imp-power-dvd]*)

lemma *dvd-power-same*: $x \text{ dvd } y \implies x \wedge n \text{ dvd } y \wedge n$
by (*induct n*) (*auto simp add: mult-dvd-mono*)

lemma *dvd-power-le*: $x \text{ dvd } y \implies m \geq n \implies x \wedge n \text{ dvd } y \wedge m$
 by (*rule power-le-dvd [OF dvd-power-same]*)

lemma *dvd-power [simp]*:
 fixes $n :: \text{nat}$
 assumes $n > 0 \vee x = 1$
 shows $x \text{ dvd } (x \wedge n)$
 using *assms*
proof
 assume $0 < n$
 then have $x \wedge n = x \wedge \text{Suc } (n - 1)$ by *simp*
 then show $x \text{ dvd } (x \wedge n)$ by *simp*
next
 assume $x = 1$
 then show $x \text{ dvd } (x \wedge n)$ by *simp*
qed
end

context *semiring-1-no-zero-divisors*
begin

subclass *power* .

lemma *power-eq-0-iff [simp]*: $a \wedge n = 0 \iff a = 0 \wedge n > 0$
 by (*induct n*) *auto*

lemma *power-not-zero*: $a \neq 0 \implies a \wedge n \neq 0$
 by (*induct n*) *auto*

lemma *zero-eq-power2 [simp]*: $a^2 = 0 \iff a = 0$
 unfolding *power2-eq-square* by *simp*

end

context *ring-1*
begin

lemma *power-minus*: $(- a) \wedge n = (- 1) \wedge n * a \wedge n$
proof (*induct n*)
 case 0
 show ?case by *simp*
next
 case (*Suc n*)
 then show ?case
 by (*simp del: power-Suc add: power-Suc2 mult.assoc*)
qed

lemma *power-minus'*: *NO-MATCH* $1 x \implies (-x) \wedge n = (-1) \wedge n * x \wedge n$

by (rule power-minus)

lemma *power-minus-Bit0*: $(- x) \wedge \text{numeral } (\text{Num.Bit0 } k) = x \wedge \text{numeral } (\text{Num.Bit0 } k)$

by (induct k, simp-all only: numeral-class.numeral.simps power-add power-one-right mult-minus-left mult-minus-right minus-minus)

lemma *power-minus-Bit1*: $(- x) \wedge \text{numeral } (\text{Num.Bit1 } k) = - (x \wedge \text{numeral } (\text{Num.Bit1 } k))$

by (simp only: eval-nat-numeral(3) power-Suc power-minus-Bit0 mult-minus-left)

lemma *power2-minus* [simp]: $(- a)^2 = a^2$

by (fact power-minus-Bit0)

lemma *power-minus1-even* [simp]: $(- 1) \wedge (2*n) = 1$

proof (induct n)

case 0

show ?case by simp

next

case (Suc n)

then show ?case by (simp add: power-add power2-eq-square)

qed

lemma *power-minus1-odd*: $(- 1) \wedge \text{Suc } (2*n) = -1$

by simp

lemma *power-minus-even* [simp]: $(-a) \wedge (2*n) = a \wedge (2*n)$

by (simp add: power-minus [of a])

end

context *ring-1-no-zero-divisors*

begin

lemma *power2-eq-1-iff*: $a^2 = 1 \iff a = 1 \vee a = -1$

using *square-eq-1-iff* [of a] by (simp add: power2-eq-square)

end

context *idom*

begin

lemma *power2-eq-iff*: $x^2 = y^2 \iff x = y \vee x = -y$

unfolding *power2-eq-square* by (rule *square-eq-iff*)

end

context *semidom-divide*

begin

lemma *power-diff*:

$a^{\wedge}(m - n) = (a^{\wedge}m) \text{ div } (a^{\wedge}n)$ if $a \neq 0$ and $n \leq m$

proof –

define q where $q = m - n$

with $\langle n \leq m \rangle$ **have** $m = q + n$ **by** *simp*

with $\langle a \neq 0 \rangle$ q -*def* **show** *?thesis*

by (*simp add: power-add*)

qed

end

context *algebraic-semidom*

begin

lemma *div-power*: $b \text{ dvd } a \implies (a \text{ div } b)^{\wedge}n = a^{\wedge}n \text{ div } b^{\wedge}n$

by (*induct n*) (*simp-all add: div-mult-div-if-dvd dvd-power-same*)

lemma *is-unit-power-iff*: $\text{is-unit } (a^{\wedge}n) \iff \text{is-unit } a \vee n = 0$

by (*induct n*) (*auto simp add: is-unit-mult-iff*)

lemma *dvd-power-iff*:

assumes $x \neq 0$

shows $x^{\wedge}m \text{ dvd } x^{\wedge}n \iff \text{is-unit } x \vee m \leq n$

proof

assume $*$: $x^{\wedge}m \text{ dvd } x^{\wedge}n$

{

assume $m > n$

note $*$

also have $x^{\wedge}n = x^{\wedge}n * 1$ **by** *simp*

also from $\langle m > n \rangle$ **have** $m = n + (m - n)$ **by** *simp*

also have $x^{\wedge}\dots = x^{\wedge}n * x^{\wedge}(m - n)$ **by** (*rule power-add*)

finally have $x^{\wedge}(m - n) \text{ dvd } 1$

using *assms* **by** (*subst (asm) dvd-times-left-cancel-iff*) *simp-all*

with $\langle m > n \rangle$ **have** $\text{is-unit } x$ **by** (*simp add: is-unit-power-iff*)

}

thus $\text{is-unit } x \vee m \leq n$ **by** *force*

qed (*auto intro: unit-imp-dvd simp: is-unit-power-iff le-imp-power-dvd*)

end

context *normalization-semidom-multiplicative*

begin

lemma *normalize-power*: $\text{normalize } (a^{\wedge}n) = \text{normalize } a^{\wedge}n$

by (*induct n*) (*simp-all add: normalize-mult*)

lemma *unit-factor-power*: $\text{unit-factor } (a^{\wedge}n) = \text{unit-factor } a^{\wedge}n$

by (induct n) (simp-all add: unit-factor-mult)

end

context division-ring

begin

Perhaps these should be simprules.

lemma power-inverse [field-simps, field-split-simps, divide-simps]: inverse $a^n =$
inverse (a^n)

proof (cases a = 0)

case True

then show ?thesis by (simp add: power-0-left)

next

case False

then have inverse (a^n) = inverse a^n

by (induct n) (simp-all add: nonzero-inverse-mult-distrib power-commutes)

then show ?thesis by simp

qed

lemma power-one-over [field-simps, field-split-simps, divide-simps]: $(1 / a)^n =$
 $1 / a^n$

using power-inverse [of a] by (simp add: divide-inverse)

end

context field

begin

lemma power-divide [field-simps, field-split-simps, divide-simps]: $(a / b)^n = a^n / b^n$

by (induct n) simp-all

end

47.2 Exponentiation on ordered types

context linordered-semidom

begin

lemma zero-less-power [simp]: $0 < a \implies 0 < a^n$

by (induct n) simp-all

lemma zero-le-power [simp]: $0 \leq a \implies 0 \leq a^n$

by (induct n) simp-all

lemma power-mono: $a \leq b \implies 0 \leq a \implies a^n \leq b^n$

by (induct n) (auto intro: mult-mono order-trans [of 0 a b])

lemma *one-le-power* [*simp*]: $1 \leq a \implies 1 \leq a \wedge n$
 using *power-mono* [*of 1 a n*] **by** *simp*

lemma *power-le-one*: $0 \leq a \implies a \leq 1 \implies a \wedge n \leq 1$
 using *power-mono* [*of a 1 n*] **by** *simp*

lemma *power-gt1-lemma*:

assumes *gt1*: $1 < a$

shows $1 < a * a \wedge n$

proof –

from *gt1* have $0 \leq a$

by (*fact order-trans* [*OF zero-le-one less-imp-le*])

from *gt1* have $1 * 1 < a * 1$ **by** *simp*

also from *gt1* have $\dots \leq a * a \wedge n$

by (*simp only: mult-mono* $\langle 0 \leq a \rangle$ *one-le-power order-less-imp-le zero-le-one order-refl*)

finally show *?thesis* **by** *simp*

qed

lemma *power-gt1*: $1 < a \implies 1 < a \wedge \text{Suc } n$
 by (*simp add: power-gt1-lemma*)

lemma *one-less-power* [*simp*]: $1 < a \implies 0 < n \implies 1 < a \wedge n$
 by (*cases n*) (*simp-all add: power-gt1-lemma*)

lemma *power-le-imp-le-exp*:

assumes *gt1*: $1 < a$

shows $a \wedge m \leq a \wedge n \implies m \leq n$

proof (*induct m arbitrary: n*)

case 0

show *?case* **by** *simp*

next

case (*Suc m*)

show *?case*

proof (*cases n*)

case 0

with *Suc* have $a * a \wedge m \leq 1$ **by** *simp*

with *gt1* show *?thesis*

by (*force simp only: power-gt1-lemma not-less* [*symmetric*])

next

case (*Suc n*)

with *Suc.prem*s *Suc.hyps* show *?thesis*

by (*force dest: mult-left-le-imp-le simp add: less-trans* [*OF zero-less-one gt1*])

qed

qed

lemma *of-nat-zero-less-power-iff* [*simp*]: $\text{of-nat } x \wedge n > 0 \iff x > 0 \vee n = 0$
 by (*induct n*) *auto*

Surely we can strengthen this? It holds for $0 < a < 1$ too.

lemma *power-inject-exp* [simp]:

$\langle a^m = a^n \longleftrightarrow m = n \rangle$ **if** $\langle 1 < a \rangle$

using *that* **by** (force simp add: order-class.order.antisym power-le-imp-le-exp)

Can relax the first premise to $(0::'a) < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*: $1 < a \implies a^m < a^n \implies m < n$

by (simp add: order-less-le [of m n] less-le [of a^m a^n] power-le-imp-le-exp)

lemma *power-strict-mono*: $a < b \implies 0 \leq a \implies 0 < n \implies a^n < b^n$

proof (induct n)

case 0

then show ?case **by** simp

next

case (Suc n)

then show ?case

by (cases n = 0) (auto simp: mult-strict-mono le-less-trans [of 0 a b])

qed

lemma *power-mono-iff* [simp]:

shows $\llbracket a \geq 0; b \geq 0; n > 0 \rrbracket \implies a^n \leq b^n \longleftrightarrow a \leq b$

using *power-mono* [of a b] *power-strict-mono* [of b a] *not-le* **by** auto

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*: $0 < a \implies a < 1 \implies a * a^n < a^n$

by (induct n) (auto simp: mult-strict-left-mono)

lemma *power-strict-decreasing*: $n < N \implies 0 < a \implies a < 1 \implies a^N < a^n$

proof (induction N)

case 0

then show ?case **by** simp

next

case (Suc N)

then show ?case

using *mult-strict-mono*[of a 1 a^N a^n]

by (auto simp add: power-Suc-less less-Suc-eq)

qed

Proof resembles that of *power-strict-decreasing*.

lemma *power-decreasing*: $n \leq N \implies 0 \leq a \implies a \leq 1 \implies a^N \leq a^n$

proof (induction N)

case 0

then show ?case **by** simp

next

case (Suc N)

then show ?case

using *mult-mono*[of a 1 a^N a^n]

by (auto simp add: le-Suc-eq)

qed

lemma *power-decreasing-iff* [simp]: $\llbracket 0 < b; b < 1 \rrbracket \implies b \wedge m \leq b \wedge n \longleftrightarrow n \leq m$

using *power-strict-decreasing* [of $m\ n\ b$]
by (*auto intro: power-decreasing ccontr*)

lemma *power-strict-decreasing-iff* [simp]: $\llbracket 0 < b; b < 1 \rrbracket \implies b \wedge m < b \wedge n \longleftrightarrow n < m$

using *power-decreasing-iff* [of $b\ m\ n$] **unfolding** *le-less*
by (*auto dest: power-strict-decreasing le-neq-implies-less*)

lemma *power-Suc-less-one*: $0 < a \implies a < 1 \implies a \wedge \text{Suc } n < 1$
using *power-strict-decreasing* [of $0\ \text{Suc } n\ a$] **by** *simp*

Proof again resembles that of *power-strict-decreasing*.

lemma *power-increasing*: $n \leq N \implies 1 \leq a \implies a \wedge n \leq a \wedge N$

proof (*induct N*)

case 0

then show *?case* **by** *simp*

next

case ($\text{Suc } N$)

then show *?case*

using *mult-mono*[of $1\ a\ a \wedge n\ a \wedge N$]

by (*auto simp add: le-Suc-eq order-trans [OF zero-le-one]*)

qed

Lemma for *power-strict-increasing*.

lemma *power-less-power-Suc*: $1 < a \implies a \wedge n < a * a \wedge n$

by (*induct n*) (*auto simp: mult-strict-left-mono less-trans [OF zero-less-one]*)

lemma *power-strict-increasing*: $n < N \implies 1 < a \implies a \wedge n < a \wedge N$

proof (*induct N*)

case 0

then show *?case* **by** *simp*

next

case ($\text{Suc } N$)

then show *?case*

using *mult-strict-mono*[of $1\ a\ a \wedge n\ a \wedge N$]

by (*auto simp add: power-less-power-Suc less-Suc-eq less-trans [OF zero-less-one] less-imp-le*)

qed

lemma *power-increasing-iff* [simp]: $1 < b \implies b \wedge x \leq b \wedge y \longleftrightarrow x \leq y$

by (*blast intro: power-le-imp-le-exp power-increasing less-imp-le*)

lemma *power-strict-increasing-iff* [simp]: $1 < b \implies b \wedge x < b \wedge y \longleftrightarrow x < y$

by (*blast intro: power-less-imp-less-exp power-strict-increasing*)

lemma *power-le-imp-le-base*:

assumes *le*: $a \wedge \text{Suc } n \leq b \wedge \text{Suc } n$

and $0 \leq b$
shows $a \leq b$
proof (rule *ccontr*)
assume $\neg ?thesis$
then have $b < a$ **by** (simp only: *linorder-not-le*)
then have $b \wedge \text{Suc } n < a \wedge \text{Suc } n$
by (simp only: *assms(2) power-strict-mono*)
with *le* **show** *False*
by (simp add: *linorder-not-less [symmetric]*)
qed

lemma *power-less-imp-less-base*:
assumes *less*: $a \wedge n < b \wedge n$
assumes *nonneg*: $0 \leq b$
shows $a < b$
proof (rule *contrapos-pp [OF less]*)
assume $\neg ?thesis$
then have $b \leq a$ **by** (simp only: *linorder-not-less*)
from *this nonneg* **have** $b \wedge n \leq a \wedge n$ **by** (rule *power-mono*)
then show $\neg a \wedge n < b \wedge n$ **by** (simp only: *linorder-not-less*)
qed

lemma *power-inject-base*: $a \wedge \text{Suc } n = b \wedge \text{Suc } n \implies 0 \leq a \implies 0 \leq b \implies a = b$
by (*blast intro: power-le-imp-le-base order.antisym eq-refl sym*)

lemma *power-eq-imp-eq-base*: $a \wedge n = b \wedge n \implies 0 \leq a \implies 0 \leq b \implies 0 < n \implies a = b$
by (*cases n*) (simp-all del: *power-Suc*, rule *power-inject-base*)

lemma *power-eq-iff-eq-base*: $0 < n \implies 0 \leq a \implies 0 \leq b \implies a \wedge n = b \wedge n \longleftrightarrow a = b$
using *power-eq-imp-eq-base [of a n b]* **by** *auto*

lemma *power2-le-imp-le*: $x^2 \leq y^2 \implies 0 \leq y \implies x \leq y$
unfolding *numeral-2-eq-2* **by** (rule *power-le-imp-le-base*)

lemma *power2-less-imp-less*: $x^2 < y^2 \implies 0 \leq y \implies x < y$
by (rule *power-less-imp-less-base*)

lemma *power2-eq-imp-eq*: $x^2 = y^2 \implies 0 \leq x \implies 0 \leq y \implies x = y$
unfolding *numeral-2-eq-2* **by** (erule (2) *power-eq-imp-eq-base*) *simp*

lemma *power-Suc-le-self*: $0 \leq a \implies a \leq 1 \implies a \wedge \text{Suc } n \leq a$
using *power-decreasing [of 1 Suc n a]* **by** *simp*

lemma *power2-eq-iff-nonneg [simp]*:
assumes $0 \leq x \ 0 \leq y$
shows $(x \wedge 2 = y \wedge 2) \longleftrightarrow x = y$
using *assms power2-eq-imp-eq* **by** *blast*

lemma *of-nat-less-numeral-power-cancel-iff*[simp]:
 $of\text{-}nat\ x < numeral\ i\ ^n \longleftrightarrow x < numeral\ i\ ^n$
using *of-nat-less-iff*[of $x\ numeral\ i\ ^n$, *unfolded of-nat-numeral of-nat-power*].

lemma *of-nat-le-numeral-power-cancel-iff*[simp]:
 $of\text{-}nat\ x \leq numeral\ i\ ^n \longleftrightarrow x \leq numeral\ i\ ^n$
using *of-nat-le-iff*[of $x\ numeral\ i\ ^n$, *unfolded of-nat-numeral of-nat-power*].

lemma *numeral-power-less-of-nat-cancel-iff*[simp]:
 $numeral\ i\ ^n < of\text{-}nat\ x \longleftrightarrow numeral\ i\ ^n < x$
using *of-nat-less-iff*[of $numeral\ i\ ^n\ x$, *unfolded of-nat-numeral of-nat-power*].

lemma *numeral-power-le-of-nat-cancel-iff*[simp]:
 $numeral\ i\ ^n \leq of\text{-}nat\ x \longleftrightarrow numeral\ i\ ^n \leq x$
using *of-nat-le-iff*[of $numeral\ i\ ^n\ x$, *unfolded of-nat-numeral of-nat-power*].

lemma *of-nat-le-of-nat-power-cancel-iff*[simp]: $(of\text{-}nat\ b)\ ^w \leq of\text{-}nat\ x \longleftrightarrow b\ ^w \leq x$
by (*metis of-nat-le-iff of-nat-power*)

lemma *of-nat-power-le-of-nat-cancel-iff*[simp]: $of\text{-}nat\ x \leq (of\text{-}nat\ b)\ ^w \longleftrightarrow x \leq b\ ^w$
by (*metis of-nat-le-iff of-nat-power*)

lemma *of-nat-less-of-nat-power-cancel-iff*[simp]: $(of\text{-}nat\ b)\ ^w < of\text{-}nat\ x \longleftrightarrow b\ ^w < x$
by (*metis of-nat-less-iff of-nat-power*)

lemma *of-nat-power-less-of-nat-cancel-iff*[simp]: $of\text{-}nat\ x < (of\text{-}nat\ b)\ ^w \longleftrightarrow x < b\ ^w$
by (*metis of-nat-less-iff of-nat-power*)

lemma *power2-nonneg-ge-1-iff*:
assumes $x \geq 0$
shows $x\ ^2 \geq 1 \longleftrightarrow x \geq 1$
using *assms* **by** (*auto intro: power2-le-imp-le*)

lemma *power2-nonneg-gt-1-iff*:
assumes $x \geq 0$
shows $x\ ^2 > 1 \longleftrightarrow x > 1$
using *assms* **by** (*auto intro: power-less-imp-less-base*)

end

Some *nat*-specific lemmas:

lemma *mono-ge2-power-minus-self*:
assumes $k \geq 2$ **shows** *mono* $(\lambda m. k\ ^m - m)$
unfolding *mono-iff-le-Suc*

```

proof
  fix  $n$ 
  have  $k^n < k^{Suc\ n}$  using power-strict-increasing-iff[of k n Suc n] assms by
linarith
  thus  $k^n - n \leq k^{Suc\ n} - Suc\ n$  by linarith
qed

lemma self-le-ge2-pow[simp]:
  assumes  $k \geq 2$  shows  $m \leq k^m$ 
proof (induction m)
  case 0 show ?case by simp
next
  case (Suc m)
  hence  $Suc\ m \leq Suc\ (k^m)$  by simp
  also have  $\dots \leq k^m + k^m$  using one-le-power[of k m] assms by linarith
  also have  $\dots \leq k * k^m$  by (metis mult-2 mult-le-mono1[OF assms])
  finally show ?case by simp
qed

lemma diff-le-diff-pow[simp]:
  assumes  $k \geq 2$  shows  $m - n \leq k^m - k^n$ 
proof (cases n ≤ m)
  case True
  thus ?thesis
  using monoD[OF mono-ge2-power-minus-self[OF assms] True] self-le-ge2-pow[OF
assms, of m]
  by (simp add: le-diff-conv le-diff-conv2)
qed auto

context linordered-ring-strict
begin

lemma sum-squares-eq-zero-iff:  $x * x + y * y = 0 \iff x = 0 \wedge y = 0$ 
  by (simp add: add-nonneg-eq-0-iff)

lemma sum-squares-le-zero-iff:  $x * x + y * y \leq 0 \iff x = 0 \wedge y = 0$ 
  by (simp add: le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff)

lemma sum-squares-gt-zero-iff:  $0 < x * x + y * y \iff x \neq 0 \vee y \neq 0$ 
  by (simp add: not-le [symmetric] sum-squares-le-zero-iff)

end

context linordered-idom
begin

lemma zero-le-power2 [simp]:  $0 \leq a^2$ 
  by (simp add: power2-eq-square)

```

lemma *zero-less-power2* [*simp*]: $0 < a^2 \longleftrightarrow a \neq 0$
by (*force simp add: power2-eq-square zero-less-mult-iff linorder-neq-iff*)

lemma *power2-less-0* [*simp*]: $\neg a^2 < 0$
by (*force simp add: power2-eq-square mult-less-0-iff*)

lemma *power-abs*: $|a \wedge n| = |a| \wedge n$ — FIXME *simp*?
by (*induct n*) (*simp-all add: abs-mult*)

lemma *power-sgn* [*simp*]: $\text{sgn } (a \wedge n) = \text{sgn } a \wedge n$
by (*induct n*) (*simp-all add: sgn-mult*)

lemma *abs-power-minus* [*simp*]: $|(- a) \wedge n| = |a \wedge n|$
by (*simp add: power-abs*)

lemma *zero-less-power-abs-iff* [*simp*]: $0 < |a| \wedge n \longleftrightarrow a \neq 0 \vee n = 0$
proof (*induct n*)
case 0
show ?*case* **by** *simp*
next
case *Suc*
then show ?*case* **by** (*auto simp: zero-less-mult-iff*)
qed

lemma *zero-le-power-abs* [*simp*]: $0 \leq |a| \wedge n$
by (*rule zero-le-power [OF abs-ge-zero]*)

lemma *power2-less-eq-zero-iff* [*simp*]: $a^2 \leq 0 \longleftrightarrow a = 0$
by (*simp add: le-less*)

lemma *abs-power2* [*simp*]: $|a^2| = a^2$
by (*simp add: power2-eq-square*)

lemma *power2-abs* [*simp*]: $|a|^2 = a^2$
by (*simp add: power2-eq-square*)

lemma *odd-power-less-zero*: $a < 0 \implies a \wedge \text{Suc } (2 * n) < 0$
proof (*induct n*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc n*)
have $a \wedge \text{Suc } (2 * \text{Suc } n) = (a * a) * a \wedge \text{Suc } (2 * n)$
by (*simp add: ac-simps power-add power2-eq-square*)
then show ?*case*
by (*simp del: power-Suc add: Suc mult-less-0-iff mult-neg-neg*)
qed

lemma *odd-0-le-power-imp-0-le*: $0 \leq a \wedge \text{Suc } (2 * n) \implies 0 \leq a$
using *odd-power-less-zero* [of a n]
by (*force simp add: linorder-not-less* [symmetric])

lemma *zero-le-even-power'*[simp]: $0 \leq a \wedge (2 * n)$
proof (*induct n*)
case 0
show ?case **by** *simp*
next
case (*Suc n*)
have $a \wedge (2 * \text{Suc } n) = (a * a) * a \wedge (2 * n)$
by (*simp add: ac-simps power-add power2-eq-square*)
then show ?case
by (*simp add: Suc zero-le-mult-iff*)
qed

lemma *sum-power2-ge-zero*: $0 \leq x^2 + y^2$
by (*intro add-nonneg-nonneg zero-le-power2*)

lemma *not-sum-power2-lt-zero*: $\neg x^2 + y^2 < 0$
unfolding *not-less* **by** (*rule sum-power2-ge-zero*)

lemma *sum-power2-eq-zero-iff*: $x^2 + y^2 = 0 \longleftrightarrow x = 0 \wedge y = 0$
unfolding *power2-eq-square* **by** (*simp add: add-nonneg-eq-0-iff*)

lemma *sum-power2-le-zero-iff*: $x^2 + y^2 \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*simp add: le-less sum-power2-eq-zero-iff not-sum-power2-lt-zero*)

lemma *sum-power2-gt-zero-iff*: $0 < x^2 + y^2 \longleftrightarrow x \neq 0 \vee y \neq 0$
unfolding *not-le* [symmetric] **by** (*simp add: sum-power2-le-zero-iff*)

lemma *abs-le-square-iff*: $|x| \leq |y| \longleftrightarrow x^2 \leq y^2$
(is ?lhs \longleftrightarrow ?rhs)

proof
assume ?lhs
then have $|x|^2 \leq |y|^2$ **by** (*rule power-mono*) *simp*
then show ?rhs **by** *simp*
next
assume ?rhs
then show ?lhs
by (*auto intro!: power2-le-imp-le* [OF - *abs-ge-zero*])
qed

lemma *power2-le-iff-abs-le*:
 $y \geq 0 \implies x^2 \leq y^2 \longleftrightarrow |x| \leq y$
by (*metis abs-le-square-iff abs-of-nonneg*)

lemma *abs-square-le-1*: $x^2 \leq 1 \longleftrightarrow |x| \leq 1$
using *abs-le-square-iff* [of x 1] **by** *simp*

lemma *abs-square-eq-1*: $x^2 = 1 \iff |x| = 1$

by (*auto simp add: abs-if power2-eq-1-iff*)

lemma *abs-square-less-1*: $x^2 < 1 \iff |x| < 1$

using *abs-square-eq-1 [of x] abs-square-le-1 [of x]* **by** (*auto simp add: le-less*)

lemma *square-le-1*:

assumes $-1 \leq x \leq 1$

shows $x^2 \leq 1$

using *assms*

by (*metis add.inverse-inverse linear mult-le-one neg-equal-0-iff-equal neg-le-iff-le power2-eq-square power-minus-Bit0*)

end

47.3 Miscellaneous rules

lemma (*in linordered-semidom*) *self-le-power*: $1 \leq a \implies 0 < n \implies a \leq a^{\wedge n}$

using *power-increasing [of 1 n a] power-one-right [of a]* **by** *auto*

lemma *power2-ge-1-iff*: $x^{\wedge 2} \geq 1 \iff x \geq 1 \vee x \leq (-1 :: 'a :: linordered-idom)$

using *abs-le-square-iff [of 1 x]* **by** (*auto simp: abs-if split: if-splits*)

lemma (*in power*) *power-eq-if*: $p^{\wedge m} = (\text{if } m=0 \text{ then } 1 \text{ else } p * (p^{\wedge (m-1)}))$

unfolding *One-nat-def* **by** (*cases m*) *simp-all*

lemma (*in comm-semiring-1*) *power2-sum*: $(x + y)^2 = x^2 + y^2 + 2 * x * y$

by (*simp add: algebra-simps power2-eq-square mult-2-right*)

context *comm-ring-1*

begin

lemma *power2-diff*: $(x - y)^2 = x^2 + y^2 - 2 * x * y$

by (*simp add: algebra-simps power2-eq-square mult-2-right*)

lemma *power2-commute*: $(x - y)^2 = (y - x)^2$

by (*simp add: algebra-simps power2-eq-square*)

lemma *minus-power-mult-self*: $(-a)^{\wedge n} * (-a)^{\wedge n} = a^{\wedge (2 * n)}$

by (*simp add: power-mult-distrib [symmetric]*)

(*simp add: power2-eq-square [symmetric] power-mult [symmetric]*)

lemma *minus-one-mult-self [simp]*: $(-1)^{\wedge n} * (-1)^{\wedge n} = 1$

using *minus-power-mult-self [of 1 n]* **by** *simp*

lemma *left-minus-one-mult-self [simp]*: $(-1)^{\wedge n} * ((-1)^{\wedge n} * a) = a$

by (*simp add: mult.assoc [symmetric]*)

end

Simprules for comparisons where common factors can be cancelled.

lemmas *zero-compare-simps* =
add-strict-increasing add-strict-increasing2 add-increasing
zero-le-mult-iff zero-le-divide-iff
zero-less-mult-iff zero-less-divide-iff
mult-le-0-iff divide-le-0-iff
mult-less-0-iff divide-less-0-iff
zero-le-power2 power2-less-0

47.4 Exponentiation for the Natural Numbers

lemma *nat-one-le-power* [*simp*]: $Suc\ 0 \leq i \implies Suc\ 0 \leq i \wedge n$
 by (*rule one-le-power [of i n, unfolded One-nat-def]*)

lemma *nat-zero-less-power-iff* [*simp*]: $x \wedge n > 0 \iff x > 0 \vee n = 0$
 for $x :: nat$
 by (*induct n*) *auto*

lemma *nat-power-eq-Suc-0-iff* [*simp*]: $x \wedge m = Suc\ 0 \iff m = 0 \vee x = Suc\ 0$
 by (*induct m*) *auto*

lemma *power-Suc-0* [*simp*]: $Suc\ 0 \wedge n = Suc\ 0$
 by *simp*

Valid for the naturals, but what if $0 < i < 1$? Premises cannot be weakened:
 consider the case where $i = 0$, $m = 1$ and $n = 0$.

lemma *nat-power-less-imp-less*:
 fixes $i :: nat$
 assumes *nonneg*: $0 < i$
 assumes *less*: $i \wedge m < i \wedge n$
 shows $m < n$
proof (*cases i = 1*)
 case *True*
 with *less power-one* [**where** $'a = nat$] **show** *?thesis* by *simp*
next
 case *False*
 with *nonneg* **have** $1 < i$ by *auto*
 from *power-strict-increasing-iff [OF this] less* **show** *?thesis* ..
qed

lemma *power-gt-expt*: $n > Suc\ 0 \implies n \wedge k > k$
 by (*induction k*) (*auto simp: less-trans-Suc n-less-m-mult-n*)

lemma *less-exp* [*simp*]:
 $\langle n < 2 \wedge n \rangle$
 by (*simp add: power-gt-expt*)

```

lemma power-dvd-imp-le:
  fixes i :: nat
  assumes  $i \wedge m \text{ dvd } i \wedge n \ 1 < i$ 
  shows  $m \leq n$ 
  using assms by (auto intro: power-le-imp-le-exp [OF <1 < i> dvd-imp-le])

lemma dvd-power-iff-le:
  fixes k::nat
  shows  $2 \leq k \implies ((k \wedge m) \text{ dvd } (k \wedge n) \longleftrightarrow m \leq n)$ 
  using le-imp-power-dvd power-dvd-imp-le by force

lemma power2-nat-le-eq-le:  $m^2 \leq n^2 \longleftrightarrow m \leq n$ 
  for m n :: nat
  by (auto intro: power2-le-imp-le power-mono)

lemma power2-nat-le-imp-le:
  fixes m n :: nat
  assumes  $m^2 \leq n$ 
  shows  $m \leq n$ 
proof (cases m)
  case 0
  then show ?thesis by simp
next
  case (Suc k)
  show ?thesis
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then have  $n < m$  by simp
    with assms Suc show False
    by (simp add: power2-eq-square)
  qed
qed

lemma ex-power-ivl1: fixes b k :: nat assumes  $b \geq 2$ 
shows  $k \geq 1 \implies \exists n. b \wedge n \leq k \wedge k < b \wedge (n+1)$  (is -  $\implies \exists n. ?P \ k \ n$ )
proof(induction k)
  case 0 thus ?case by simp
next
  case (Suc k)
  show ?case
  proof cases
    assume  $k=0$ 
    hence ?P (Suc k) 0 using assms by simp
    thus ?case ..
  next
  assume  $k \neq 0$ 
  with Suc obtain n where IH: ?P k n by auto
  show ?case
  proof (cases k = b \wedge (n+1) - 1)

```



```

    case True
    hence ?P (Suc k) (n+1) using assms
      by (simp add: power-less-power-Suc)
    thus ?thesis ..
  next
  case False
  hence ?P (Suc k) n using IH by auto
  thus ?thesis ..
qed
qed
qed

```

```

lemma ex-power-ivl2: fixes b k :: nat assumes b ≥ 2 k ≥ 2
  shows ∃ n. bn < k ∧ k ≤ b(n+1)
proof -
  have 1 ≤ k - 1 using assms(2) by arith
  from ex-power-ivl1[OF assms(1) this]
  obtain n where bn ≤ k - 1 ∧ k - 1 < b(n + 1) ..
  hence bn < k ∧ k ≤ b(n+1) using assms by auto
  thus ?thesis ..
qed

```

47.4.1 Cardinality of the Powerset

```

lemma card-UNIV-bool [simp]: card (UNIV :: bool set) = 2
  unfolding UNIV-bool by simp

```

```

lemma card-Pow: finite A ⇒ card (Pow A) = 2card A

```

```

proof (induct rule: finite-induct)
  case empty
  show ?case by simp
next
  case (insert x A)
  from ⟨x ∉ A⟩ have disjoint: Pow A ∩ insert x ‘ Pow A = {} by blast
  from ⟨x ∉ A⟩ have inj-on: inj-on (insert x) (Pow A)
    unfolding inj-on-def by auto

  have card (Pow (insert x A)) = card (Pow A ∪ insert x ‘ Pow A)
    by (simp only: Pow-insert)
  also have ... = card (Pow A) + card (insert x ‘ Pow A)
    by (rule card-Un-disjoint) (use ⟨finite A⟩ disjoint in simp-all)
  also from inj-on have card (insert x ‘ Pow A) = card (Pow A)
    by (rule card-image)
  also have ... + ... = 2 * ... by (simp add: mult-2)
  also from insert(3) have ... = 2Suc (card A) by simp
  also from insert(1,2) have Suc (card A) = card (insert x A)
    by (rule card-insert-disjoint [symmetric])
  finally show ?case .
qed

```

47.5 Code generator tweak

```
code-identifier
  code-module Power  $\rightarrow$  (SML) Arith and (OCaml) Arith and (Haskell) Arith
end
```

48 Big sum and product over finite (non-empty) sets

```
theory Groups-Big
  imports Power Equiv-Relations
begin
```

48.1 Generic monoid operation over a set

```
locale comm-monoid-set = comm-monoid
begin
```

48.1.1 Standard sum or product indexed by a finite set

```
interpretation comp-fun-commute f
  by standard (simp add: fun-eq-iff left-commute)
```

```
interpretation comp?: comp-fun-commute f  $\circ$  g
  by (fact comp-comp-fun-commute)
```

```
definition F :: ('b  $\Rightarrow$  'a)  $\Rightarrow$  'b set  $\Rightarrow$  'a
  where eq-fold: F g A = Finite-Set.fold (f  $\circ$  g) 1 A
```

```
lemma infinite [simp]:  $\neg$  finite A  $\Longrightarrow$  F g A = 1
  by (simp add: eq-fold)
```

```
lemma empty [simp]: F g {} = 1
  by (simp add: eq-fold)
```

```
lemma insert [simp]: finite A  $\Longrightarrow$  x  $\notin$  A  $\Longrightarrow$  F g (insert x A) = g x * F g A
  by (simp add: eq-fold)
```

```
lemma remove:
```

```
  assumes finite A and x  $\in$  A
  shows F g A = g x * F g (A - {x})
```

```
proof -
```

```
  from  $\langle x \in A \rangle$  obtain B where B: A = insert x B and x  $\notin$  B
  by (auto dest: mk-disjoint-insert)
```

```
  moreover from  $\langle$ finite A $\rangle$  B have finite B by simp
```

```
  ultimately show ?thesis by simp
```

```
qed
```

lemma *insert-remove*: $finite\ A \implies F\ g\ (insert\ x\ A) = g\ x * F\ g\ (A - \{x\})$
by (*cases* $x \in A$) (*simp-all add: remove insert-absorb*)

lemma *insert-if*: $finite\ A \implies F\ g\ (insert\ x\ A) = (if\ x \in A\ then\ F\ g\ A\ else\ g\ x * F\ g\ A)$
by (*cases* $x \in A$) (*simp-all add: insert-absorb*)

lemma *neutral*: $\forall x \in A. g\ x = \mathbf{1} \implies F\ g\ A = \mathbf{1}$
by (*induct* A *rule: infinite-finite-induct*) *simp-all*

lemma *neutral-const* [*simp*]: $F\ (\lambda-. \mathbf{1})\ A = \mathbf{1}$
by (*simp add: neutral*)

lemma *union-inter*:

assumes *finite* A **and** *finite* B

shows $F\ g\ (A \cup B) * F\ g\ (A \cap B) = F\ g\ A * F\ g\ B$

— The reversed orientation looks more natural, but LOOPS as a simprule!

using *assms*

proof (*induct* A)

case *empty*

then show *?case* **by** *simp*

next

case (*insert* $x\ A$)

then show *?case*

by (*auto simp: insert-absorb Int-insert-left commute [of - $g\ x$] assoc left-commute*)

qed

corollary *union-inter-neutral*:

assumes *finite* A **and** *finite* B

and $\forall x \in A \cap B. g\ x = \mathbf{1}$

shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$

using *assms* **by** (*simp add: union-inter [symmetric] neutral*)

corollary *union-disjoint*:

assumes *finite* A **and** *finite* B

assumes $A \cap B = \{\}$

shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$

using *assms* **by** (*simp add: union-inter-neutral*)

lemma *union-diff2*:

assumes *finite* A **and** *finite* B

shows $F\ g\ (A \cup B) = F\ g\ (A - B) * F\ g\ (B - A) * F\ g\ (A \cap B)$

proof —

have $A \cup B = A - B \cup (B - A) \cup A \cap B$

by *auto*

with *assms* **show** *?thesis*

by *simp (subst union-disjoint, auto)+*

qed

lemma *subset-diff*:
 assumes $B \subseteq A$ and *finite* A
 shows $F\ g\ A = F\ g\ (A - B) * F\ g\ B$
proof –
 from *assms* have *finite* $(A - B)$ by *auto*
 moreover from *assms* have *finite* B by (*rule finite-subset*)
 moreover from *assms* have $(A - B) \cap B = \{\}$ by *auto*
 ultimately have $F\ g\ (A - B \cup B) = F\ g\ (A - B) * F\ g\ B$ by (*rule union-disjoint*)
 moreover from *assms* have $A \cup B = A$ by *auto*
 ultimately show *?thesis* by *simp*
qed

lemma *Int-Diff*:
 assumes *finite* A
 shows $F\ g\ A = F\ g\ (A \cap B) * F\ g\ (A - B)$
 by (*subst subset-diff* [**where** $B = A - B$]) (*auto simp: Diff-Diff-Int assms*)

lemma *setdiff-irrelevant*:
 assumes *finite* A
 shows $F\ g\ (A - \{x. g\ x = z\}) = F\ g\ A$
 using *assms* by (*induct* A) (*simp-all add: insert-Diff-if*)

lemma *not-neutral-contains-not-neutral*:
 assumes $F\ g\ A \neq 1$
 obtains a where $a \in A$ and $g\ a \neq 1$
proof –
 from *assms* have $\exists a \in A. g\ a \neq 1$
proof (*induct* A *rule: infinite-finite-induct*)
 case *infinite*
 then show *?case* by *simp*
 next
 case *empty*
 then show *?case* by *simp*
 next
 case (*insert* $a\ A$)
 then show *?case* by *fastforce*
qed
 with *that* show *thesis* by *blast*
qed

lemma *reindex*:
 assumes *inj-on* $h\ A$
 shows $F\ g\ (h\ ` A) = F\ (g \circ h)\ A$
proof (*cases* *finite* A)
 case *True*
 with *assms* show *?thesis*
 by (*simp add: eq-fold fold-image comp-assoc*)
 next
 case *False*

with *assms* **have** $\neg \text{finite } (h \text{ ‘ } A)$ **by** (*blast dest: finite-imageD*)
with *False* **show** *?thesis* **by** *simp*
qed

lemma *cong [fundef-cong]*:
assumes $A = B$
assumes $g\text{-}h: \bigwedge x. x \in B \implies g\ x = h\ x$
shows $F\ g\ A = F\ h\ B$
using $g\text{-}h$ **unfolding** $\langle A = B \rangle$
by (*induct B rule: infinite-finite-induct*) *auto*

lemma *cong-simp [cong]*:
 $\llbracket A = B; \bigwedge x. x \in B = \text{simp} \implies g\ x = h\ x \rrbracket \implies F\ (\lambda x. g\ x)\ A = F\ (\lambda x. h\ x)\ B$
by (*rule cong*) (*simp-all add: simp-implies-def*)

lemma *reindex-cong*:
assumes *inj-on l B*
assumes $A = l \text{ ‘ } B$
assumes $\bigwedge x. x \in B \implies g\ (l\ x) = h\ x$
shows $F\ g\ A = F\ h\ B$
using *assms* **by** (*simp add: reindex*)

lemma *image-eq*:
assumes *inj-on g A*
shows $F\ (\lambda x. x)\ (g \text{ ‘ } A) = F\ g\ A$
using *assms* *reindex-cong* **by** *fastforce*

lemma *UNION-disjoint*:
assumes *finite I* **and** $\forall i \in I. \text{finite } (A\ i)$
and $\forall i \in I. \forall j \in I. i \neq j \longrightarrow A\ i \cap A\ j = \{\}$
shows $F\ g\ (\bigcup (A \text{ ‘ } I)) = F\ (\lambda x. F\ g\ (A\ x))\ I$
using *assms*
proof (*induction rule: finite-induct*)
case (*insert i I*)
then have $\forall j \in I. j \neq i$
by *blast*
with *insert.prem*s **have** $A\ i \cap \bigcup (A \text{ ‘ } I) = \{\}$
by *blast*
with *insert* **show** *?case*
by (*simp add: union-disjoint*)
qed *auto*

lemma *Union-disjoint*:
assumes $\forall A \in C. \text{finite } A \ \forall A \in C. \forall B \in C. A \neq B \longrightarrow A \cap B = \{\}$
shows $F\ g\ (\bigcup C) = (F \circ F)\ g\ C$
proof (*cases finite C*)
case *True*
from *UNION-disjoint* [*OF this assms*] **show** *?thesis* **by** *simp*
next

case *False*
then show *?thesis* **by** (*auto dest: finite-UnionD intro: infinite*)
qed

lemma *distrib*: $F (\lambda x. g x * h x) A = F g A * F h A$
by (*induct A rule: infinite-finite-induct*) (*simp-all add: assoc commute left-commute*)

lemma *Sigma*:

assumes *finite A* $\forall x \in A. \text{finite } (B x)$
shows $F (\lambda x. F (g x) (B x)) A = F (\text{case-prod } g) (\text{SIGMA } x:A. B x)$
unfolding *Sigma-def*
proof (*subst UNION-disjoint*)
show $F (\lambda x. F (g x) (B x)) A = F (\lambda x. F (\lambda(x, y). g x y) (\bigcup y \in B x. \{(x, y)\}))$
 A
proof (*rule cong [OF refl]*)
show $F (g x) (B x) = F (\lambda(x, y). g x y) (\bigcup y \in B x. \{(x, y)\})$
if $x \in A$ **for** x
using *that assms* **by** (*simp add: UNION-disjoint*)
qed
qed (*use assms in auto*)

lemma *related*:

assumes *Re: R 1 1*
and *Rop*: $\forall x1 y1 x2 y2. R x1 x2 \wedge R y1 y2 \longrightarrow R (x1 * y1) (x2 * y2)$
and *fin*: *finite S*
and *R-h-g*: $\forall x \in S. R (h x) (g x)$
shows $R (F h S) (F g S)$
using *fin* **by** (*rule finite-subset-induct*) (*use assms in auto*)

lemma *mono-neutral-cong-left*:

assumes *finite T*
and $S \subseteq T$
and $\forall i \in T - S. h i = \mathbf{1}$
and $\bigwedge x. x \in S \Longrightarrow g x = h x$
shows $F g S = F h T$
proof –
have *eq*: $T = S \cup (T - S)$ **using** $\langle S \subseteq T \rangle$ **by** *blast*
have *d*: $S \cap (T - S) = \{\}$ **using** $\langle S \subseteq T \rangle$ **by** *blast*
from $\langle \text{finite } T \rangle \langle S \subseteq T \rangle$ **have** *f*: *finite S finite (T - S)*
by (*auto intro: finite-subset*)
show *?thesis* **using** *assms(4)*
by (*simp add: union-disjoint [OF f d, unfolded eq [symmetric]] neutral [OF assms(3)]*)
qed

lemma *mono-neutral-cong-right*:

$\text{finite } T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S. g i = \mathbf{1} \Longrightarrow (\bigwedge x. x \in S \Longrightarrow g x = h x)$
 \Longrightarrow
 $F g T = F h S$

by (auto intro!: mono-neutral-cong-left [symmetric])

lemma *mono-neutral-left: finite T $\implies S \subseteq T \implies \forall i \in T - S. g i = \mathbf{1} \implies F g S = F g T$*

by (blast intro: mono-neutral-cong-left)

lemma *mono-neutral-right: finite T $\implies S \subseteq T \implies \forall i \in T - S. g i = \mathbf{1} \implies F g T = F g S$*

by (blast intro!: mono-neutral-left [symmetric])

lemma *mono-neutral-cong:*

assumes [simp]: *finite T finite S*

and *: $\bigwedge i. i \in T - S \implies h i = \mathbf{1} \wedge i. i \in S - T \implies g i = \mathbf{1}$

and gh: $\bigwedge x. x \in S \cap T \implies g x = h x$

shows $F g S = F h T$

proof –

have $F g S = F g (S \cap T)$

by (rule mono-neutral-right) (auto intro: *)

also have $\dots = F h (S \cap T)$ using refl gh by (rule cong)

also have $\dots = F h T$

by (rule mono-neutral-left) (auto intro: *)

finally show ?thesis .

qed

lemma *reindex-bij-betw: bij-betw h S T $\implies F (\lambda x. g (h x)) S = F g T$*

by (auto simp: bij-betw-def reindex)

lemma *reindex-bij-witness:*

assumes *witness:*

$\bigwedge a. a \in S \implies i (j a) = a$

$\bigwedge a. a \in S \implies j a \in T$

$\bigwedge b. b \in T \implies j (i b) = b$

$\bigwedge b. b \in T \implies i b \in S$

assumes *eq:*

$\bigwedge a. a \in S \implies h (j a) = g a$

shows $F g S = F h T$

proof –

have *bij-betw j S T*

using *bij-betw-byWitness* [where $A=S$ and $f=j$ and $f'=i$ and $A'=T$] *witness*

by *auto*

moreover have $F g S = F (\lambda x. h (j x)) S$

by (*intro cong*) (*auto simp: eq*)

ultimately show ?thesis

by (*simp add: reindex-bij-betw*)

qed

lemma *reindex-bij-betw-not-neutral:*

assumes *fin: finite S' finite T'*

assumes *bij: bij-betw h (S - S') (T - T')*

assumes *nn*:
 $\bigwedge a. a \in S' \implies g (h a) = z$
 $\bigwedge b. b \in T' \implies g b = z$
shows $F (\lambda x. g (h x)) S = F g T$
proof –
have [*simp*]: $finite\ S \longleftrightarrow finite\ T$
using *bij-betw-finite*[*OF* *bij*] *fin* **by** *auto*
show *?thesis*
proof (*cases* *finite* *S*)
case *True*
with *nn* **have** $F (\lambda x. g (h x)) S = F (\lambda x. g (h x)) (S - S')$
by (*intro* *mono-neutral-cong-right*) *auto*
also **have** $\dots = F g (T - T')$
using *bij* **by** (*rule* *reindex-bij-betw*)
also **have** $\dots = F g T$
using *nn* $\langle finite\ S \rangle$ **by** (*intro* *mono-neutral-cong-left*) *auto*
finally **show** *?thesis* .
next
case *False*
then **show** *?thesis* **by** *simp*
qed
qed

lemma *reindex-nontrivial*:
assumes *finite* *A*
and *nz*: $\bigwedge x\ y. x \in A \implies y \in A \implies x \neq y \implies h\ x = h\ y \implies g (h\ x) = \mathbf{1}$
shows $F g (h\ ' A) = F (g \circ h)\ A$
proof (*subst* *reindex-bij-betw-not-neutral* [*symmetric*])
show *bij-betw* $h (A - \{x \in A. (g \circ h)\ x = \mathbf{1}\}) (h\ ' A - h\ ' \{x \in A. (g \circ h)\ x = \mathbf{1}\})$
using *nz* **by** (*auto* *intro!*: *inj-onI* *simp*: *bij-betw-def*)
qed (*use* $\langle finite\ A \rangle$ **in** *auto*)

lemma *reindex-bij-witness-not-neutral*:
assumes *fin*: *finite* *S'* *finite* *T'*
assumes *witness*:
 $\bigwedge a. a \in S - S' \implies i (j\ a) = a$
 $\bigwedge a. a \in S - S' \implies j\ a \in T - T'$
 $\bigwedge b. b \in T - T' \implies j (i\ b) = b$
 $\bigwedge b. b \in T - T' \implies i\ b \in S - S'$
assumes *nn*:
 $\bigwedge a. a \in S' \implies g\ a = z$
 $\bigwedge b. b \in T' \implies h\ b = z$
assumes *eq*:
 $\bigwedge a. a \in S \implies h (j\ a) = g\ a$
shows $F g S = F h T$
proof –
have *bij*: *bij-betw* $j (S - (S' \cap S)) (T - (T' \cap T))$
using *witness* **by** (*intro* *bij-betw-byWitness*[**where** $f'=i$]) *auto*


```

have F-eq: F g S = F (λx. h (j x)) S
  by (intro cong) (auto simp: eq)
show ?thesis
  unfolding F-eq using fin nn eq
  by (intro reindex-bij-betw-not-neutral[OF - - bij]) auto
qed

```

lemma *delta-remove*:

```

assumes fS: finite S
shows F (λk. if k = a then b k else c k) S = (if a ∈ S then b a * F c (S - {a})
else F c (S - {a}))
proof -
  let ?f = (λk. if k = a then b k else c k)
  show ?thesis
  proof (cases a ∈ S)
    case False
    then have ∀ k ∈ S. ?f k = c k by simp
    with False show ?thesis by simp
  next
    case True
    let ?A = S - {a}
    let ?B = {a}
    from True have eq: S = ?A ∪ ?B by blast
    have dj: ?A ∩ ?B = {} by simp
    from fS have fAB: finite ?A finite ?B by auto
    have F ?f S = F ?f ?A * F ?f ?B
      using union-disjoint [OF fAB dj, of ?f, unfolded eq [symmetric]] by simp
    with True show ?thesis
      using comm-monoid-set.remove comm-monoid-set-axioms fS by fastforce
  qed
qed

```

lemma *delta [simp]*:

```

assumes fS: finite S
shows F (λk. if k = a then b k else 1) S = (if a ∈ S then b a else 1)
by (simp add: delta-remove [OF assms])

```

lemma *delta' [simp]*:

```

assumes fin: finite S
shows F (λk. if a = k then b k else 1) S = (if a ∈ S then b a else 1)
using delta [OF fin, of a b, symmetric] by (auto intro: cong)

```

lemma *If-cases*:

```

fixes P :: 'b ⇒ bool and g h :: 'b ⇒ 'a
assumes fin: finite A
shows F (λx. if P x then h x else g x) A = F h (A ∩ {x. P x}) * F g (A ∩ -
{x. P x})
proof -
  have a: A = A ∩ {x. P x} ∪ A ∩ -{x. P x} (A ∩ {x. P x}) ∩ (A ∩ -{x. P x})

```

```

= {}
  by blast+
  from fin have f: finite (A ∩ {x. P x}) finite (A ∩ ¬{x. P x}) by auto
  let ?g = λx. if P x then h x else g x
  from union-disjoint [OF f a(2), of ?g] a(1) show ?thesis
  by (subst (1 2) cong) simp-all
qed

```

lemma cartesian-product: $F (\lambda x. F (g x) B) A = F (\text{case-prod } g) (A \times B)$

proof (cases $A = \{\} \vee B = \{\}$)

case True

then show ?thesis

by auto

next

case False

then have $A \neq \{\} \wedge B \neq \{\}$ by auto

show ?thesis

proof (cases $\text{finite } A \wedge \text{finite } B$)

case True

then show ?thesis

by (simp add: Sigma)

next

case False

then consider $\text{infinite } A \mid \text{infinite } B$ by auto

then have $\text{infinite } (A \times B)$

by cases (use $\langle A \neq \{\} \rangle \langle B \neq \{\} \rangle$ in $\langle \text{auto dest: finite-cartesian-productD1 finite-cartesian-productD2} \rangle$)

then show ?thesis

using False by auto

qed

qed

lemma inter-restrict:

assumes $\text{finite } A$

shows $F g (A \cap B) = F (\lambda x. \text{if } x \in B \text{ then } g x \text{ else } \mathbf{1}) A$

proof –

let $?g = \lambda x. \text{if } x \in A \cap B \text{ then } g x \text{ else } \mathbf{1}$

have $\forall i \in A - A \cap B. (\text{if } i \in A \cap B \text{ then } g i \text{ else } \mathbf{1}) = \mathbf{1}$ by simp

moreover have $A \cap B \subseteq A$ by blast

ultimately have $F ?g (A \cap B) = F ?g A$

using $\langle \text{finite } A \rangle$ by (intro mono-neutral-left) auto

then show ?thesis by simp

qed

lemma inter-filter:

$\text{finite } A \implies F g \{x \in A. P x\} = F (\lambda x. \text{if } P x \text{ then } g x \text{ else } \mathbf{1}) A$

by (simp add: inter-restrict [symmetric, of $A \{x. P x\}$], simplified mem-Collect-eq Int-def)

lemma *Union-comp*:

assumes $\forall A \in B. \text{finite } A$
and $\bigwedge A1 A2 x. A1 \in B \implies A2 \in B \implies A1 \neq A2 \implies x \in A1 \implies x \in A2$
 $\implies g x = \mathbf{1}$
shows $F g (\bigcup B) = (F \circ F) g B$
using *assms*
proof (*induct B rule: infinite-finite-induct*)
case (*infinite A*)
then have $\neg \text{finite } (\bigcup A)$ **by** (*blast dest: finite-UnionD*)
with *infinite show ?case by simp*
next
case *empty*
then show *?case by simp*
next
case (*insert A B*)
then have *finite A finite B finite ($\bigcup B$) $A \notin B$*
and $\forall x \in A \cap \bigcup B. g x = \mathbf{1}$
and $H: F g (\bigcup B) = (F \circ F) g B$ **by** *auto*
then have $F g (A \cup \bigcup B) = F g A * F g (\bigcup B)$
by (*simp add: union-inter-neutral*)
with $\langle \text{finite } B \rangle \langle A \notin B \rangle$ **show** *?case*
by (*simp add: H*)
qed

lemma *swap*: $F (\lambda i. F (g i) B) A = F (\lambda j. F (\lambda i. g i j) A) B$

unfolding *cartesian-product*
by (*rule reindex-bij-witness [where $i = \lambda(i, j). (j, i)$ and $j = \lambda(i, j). (j, i)$]*)
auto

lemma *swap-restrict*:

finite A \implies finite B \implies
 $F (\lambda x. F (g x) \{y. y \in B \wedge R x y\}) A = F (\lambda y. F (\lambda x. g x y) \{x. x \in A \wedge R x y\}) B$
by (*simp add: inter-filter (rule swap)*)

lemma *image-gen*:

assumes *fin: finite S*
shows $F h S = F (\lambda y. F h \{x. x \in S \wedge g x = y\}) (g \text{' } S)$
proof –
have $\{y. y \in g \text{' } S \wedge g x = y\} = \{g x\}$ **if** $x \in S$ **for** x
using *that by auto*
then have $F h S = F (\lambda x. F (\lambda y. h x) \{y. y \in g \text{' } S \wedge g x = y\}) S$
by *simp*
also have $\dots = F (\lambda y. F h \{x. x \in S \wedge g x = y\}) (g \text{' } S)$
by (*rule swap-restrict [OF fin finite-imageI [OF fin]]*)
finally show *?thesis .*

qed

lemma *group*:

assumes fS : finite S and fT : finite T and fST : $g \text{ ' } S \subseteq T$
 shows $F (\lambda y. F h \{x. x \in S \wedge g x = y\}) T = F h S$
 unfolding *image-gen*[*OF fS*, of $h g$]
 by (*auto intro*: *neutral mono-neutral-right*[*OF fT fST*])

lemma *Plus*:

fixes $A :: 'b \text{ set}$ and $B :: 'c \text{ set}$
 assumes *fin*: finite A finite B
 shows $F g (A <+> B) = F (g \circ \text{Inl}) A * F (g \circ \text{Inr}) B$
 proof –
 have $A <+> B = \text{Inl ' } A \cup \text{Inr ' } B$ by *auto*
 moreover from *fin* have finite ($\text{Inl ' } A$) finite ($\text{Inr ' } B$) by *auto*
 moreover have $\text{Inl ' } A \cap \text{Inr ' } B = \{\}$ by *auto*
 moreover have *inj-on* $\text{Inl } A$ *inj-on* $\text{Inr } B$ by (*auto intro*: *inj-onI*)
 ultimately show ?*thesis*
 using *fin* by (*simp add*: *union-disjoint reindex*)
 qed

lemma *same-carrier*:

assumes finite C
 assumes *subset*: $A \subseteq C$ $B \subseteq C$
 assumes *trivial*: $\bigwedge a. a \in C - A \implies g a = \mathbf{1} \wedge b. b \in C - B \implies h b = \mathbf{1}$
 shows $F g A = F h B \iff F g C = F h C$
 proof –
 have finite A and finite B and finite $(C - A)$ and finite $(C - B)$
 using $\langle \text{finite } C \rangle$ *subset* by (*auto elim*: *finite-subset*)
 from *subset* have [*simp*]: $A - (C - A) = A$ by *auto*
 from *subset* have [*simp*]: $B - (C - B) = B$ by *auto*
 from *subset* have $C = A \cup (C - A)$ by *auto*
 then have $F g C = F g (A \cup (C - A))$ by *simp*
 also have $\dots = F g (A - (C - A)) * F g (C - A - A) * F g (A \cap (C - A))$
 using $\langle \text{finite } A \rangle \langle \text{finite } (C - A) \rangle$ by (*simp only*: *union-diff2*)
 finally have $*$: $F g C = F g A$ using *trivial* by *simp*
 from *subset* have $C = B \cup (C - B)$ by *auto*
 then have $F h C = F h (B \cup (C - B))$ by *simp*
 also have $\dots = F h (B - (C - B)) * F h (C - B - B) * F h (B \cap (C - B))$
 using $\langle \text{finite } B \rangle \langle \text{finite } (C - B) \rangle$ by (*simp only*: *union-diff2*)
 finally have $F h C = F h B$
 using *trivial* by *simp*
 with $*$ show ?*thesis* by *simp*
 qed

lemma *same-carrierI*:

assumes finite C
 assumes *subset*: $A \subseteq C$ $B \subseteq C$
 assumes *trivial*: $\bigwedge a. a \in C - A \implies g a = \mathbf{1} \wedge b. b \in C - B \implies h b = \mathbf{1}$
 assumes $F g C = F h C$
 shows $F g A = F h B$
 using *assms same-carrier* [of $C A B$] by *simp*

lemma *eq-general*:

assumes $B: \bigwedge y. y \in B \implies \exists!x. x \in A \wedge h x = y$ **and** $A: \bigwedge x. x \in A \implies h x \in B \wedge \gamma(h x) = \varphi x$
shows $F \varphi A = F \gamma B$

proof –

have $eq: B = h \text{ ` } A$
by (*auto dest: assms*)
have $h: inj\text{-on } h A$
using *assms* **by** (*blast intro: inj-onI*)
have $F \varphi A = F (\gamma \circ h) A$
using A **by** *auto*
also have $\dots = F \gamma B$
by (*simp add: eq reindex h*)
finally show *?thesis* .

qed

lemma *eq-general-inverses*:

assumes $B: \bigwedge y. y \in B \implies k y \in A \wedge h(k y) = y$ **and** $A: \bigwedge x. x \in A \implies h x \in B \wedge k(h x) = x \wedge \gamma(h x) = \varphi x$
shows $F \varphi A = F \gamma B$
by (*rule eq-general [where h=h]*) (*force intro: dest: A B*)+

48.1.2 HOL Light variant: sum/product indexed by the non-neutral subset

NB only a subset of the properties above are proved

definition $G :: ['b \Rightarrow 'a, 'b \text{ set}] \Rightarrow 'a$

where $G p I \equiv$ *if finite* $\{x \in I. p x \neq \mathbf{1}\}$ *then* $F p \{x \in I. p x \neq \mathbf{1}\}$ *else* $\mathbf{1}$

lemma *finite-Collect-op*:

shows $\llbracket \text{finite } \{i \in I. x i \neq \mathbf{1}\}; \text{finite } \{i \in I. y i \neq \mathbf{1}\} \rrbracket \implies \text{finite } \{i \in I. x i * y i \neq \mathbf{1}\}$

apply (*rule finite-subset [where B =* $\{i \in I. x i \neq \mathbf{1}\} \cup \{i \in I. y i \neq \mathbf{1}\}$ *]*)
using *left-neutral* **by** *force+*

lemma *empty'* [*simp*]: $G p \{\} = \mathbf{1}$

by (*auto simp: G-def*)

lemma *eq-sum* [*simp*]: $\text{finite } I \implies G p I = F p I$

by (*auto simp: G-def intro: mono-neutral-cong-left*)

lemma *insert'* [*simp*]:

assumes *finite* $\{x \in I. p x \neq \mathbf{1}\}$

shows $G p (\text{insert } i I) = (\text{if } i \in I \text{ then } G p I \text{ else } p i * G p I)$

proof –

have $\{x. x = i \wedge p x \neq \mathbf{1} \vee x \in I \wedge p x \neq \mathbf{1}\} = (\text{if } p i = \mathbf{1} \text{ then } \{x \in I. p x \neq \mathbf{1}\} \text{ else } \text{insert } i \{x \in I. p x \neq \mathbf{1}\})$

by *auto*

then show *?thesis*
using *assms* **by** (*simp add: G-def conj-disj-distribR insert-absorb*)
qed

lemma *distrib-triv'*:
assumes *finite I*
shows $G (\lambda i. g i * h i) I = G g I * G h I$
by (*simp add: assms local.distrib*)

lemma *non-neutral'*: $G g \{x \in I. g x \neq \mathbf{1}\} = G g I$
by (*simp add: G-def*)

lemma *distrib'*:
assumes *finite {x \in I. g x \neq \mathbf{1}}* *finite {x \in I. h x \neq \mathbf{1}}*
shows $G (\lambda i. g i * h i) I = G g I * G h I$
proof –
have $a * a \neq a \implies a \neq \mathbf{1}$ **for** a
by *auto*
then have $G (\lambda i. g i * h i) I = G (\lambda i. g i * h i) (\{i \in I. g i \neq \mathbf{1}\} \cup \{i \in I. h i \neq \mathbf{1}\})$
using *assms* **by** (*force simp: G-def finite-Collect-op intro!: mono-neutral-cong*)
also have $\dots = G g I * G h I$
proof –
have $F g (\{i \in I. g i \neq \mathbf{1}\} \cup \{i \in I. h i \neq \mathbf{1}\}) = G g I$
 $F h (\{i \in I. g i \neq \mathbf{1}\} \cup \{i \in I. h i \neq \mathbf{1}\}) = G h I$
by (*auto simp: G-def assms intro: mono-neutral-right*)
then show *?thesis*
using *assms* **by** (*simp add: distrib*)
qed
finally show *?thesis* .
qed

lemma *cong'*:
assumes $A = B$
assumes *g-h: $\bigwedge x. x \in B \implies g x = h x$*
shows $G g A = G h B$
using *assms* **by** (*auto simp: G-def cong: conj-cong intro: cong*)

lemma *mono-neutral-cong-left'*:
assumes $S \subseteq T$
and $\bigwedge i. i \in T - S \implies h i = \mathbf{1}$
and $\bigwedge x. x \in S \implies g x = h x$
shows $G g S = G h T$
proof –
have $*$: $\{x \in S. g x \neq \mathbf{1}\} = \{x \in T. h x \neq \mathbf{1}\}$
using *assms* **by** (*metis DiffI subset-eq*)
then have *finite* $\{x \in S. g x \neq \mathbf{1}\} = \text{finite } \{x \in T. h x \neq \mathbf{1}\}$
by *simp*

then show *?thesis*
using *assms* **by** (*auto simp add: G-def * intro: cong*)
qed

lemma *mono-neutral-cong-right'*:
 $S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies (\bigwedge x. x \in S \implies g\ x = h\ x) \implies$
 $G\ g\ T = G\ h\ S$
by (*auto intro!: mono-neutral-cong-left' [symmetric]*)

lemma *mono-neutral-left'*: $S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies G\ g\ S = G\ g\ T$
by (*blast intro: mono-neutral-cong-left'*)

lemma *mono-neutral-right'*: $S \subseteq T \implies \forall i \in T - S. g\ i = \mathbf{1} \implies G\ g\ T = G\ g\ S$
by (*blast intro!: mono-neutral-left' [symmetric]*)

end

48.2 Generalized summation over a set

context *comm-monoid-add*
begin

sublocale *sum: comm-monoid-set plus 0*
defines *sum = sum.F and sum' = sum.G ..*

abbreviation *Sum* (\sum)
where $\sum \equiv \text{sum } (\lambda x. x)$

end

Now: lots of fancy syntax. First, *sum* ($\lambda x. e$) *A* is written $\sum_{x \in A}. e$.

syntax (*ASCII*)
 $\text{-sum} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-add } ((\exists \text{SUM } (-/:-)/ -) [0, 51, 10] 10)$

syntax
 $\text{-sum} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-add } ((\exists \sum (-/\in-)/ -) [0, 51, 10] 10)$

translations — Beware of argument permutation!

$\sum_{i \in A}. b \equiv \text{CONST sum } (\lambda i. b) A$

Instead of $\sum_{x \in \{x. P\}}. e$ we introduce the shorter $\sum x|P. e$.

syntax (*ASCII*)
 $\text{-qsum} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a ((\exists \text{SUM } - | / - / -) [0, 0, 10] 10)$

syntax
 $\text{-qsum} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a ((\exists \sum - | (-) / -) [0, 0, 10] 10)$

translations
 $\sum x|P. t \Rightarrow \text{CONST sum } (\lambda x. t) \{x. P\}$

print-translation <

```

let
  fun sum-tr' [Abs (x, Tx, t), Const (const-syntax ⟨Collect⟩, -) $ Abs (y, Ty, P)]
  =
    if x <> y then raise Match
    else
      let
        val x' = Syntax-Trans.mark-bound-body (x, Tx);
        val t' = subst-bound (x', t);
        val P' = subst-bound (x', P);
      in
        Syntax.const syntax-const ⟨-qsum⟩ $ Syntax-Trans.mark-bound-abs (x,
Tx) $ P' $ t'
      end
    | sum-tr' - = raise Match;
in [(const-syntax ⟨sum⟩, K sum-tr')] end
>

```

48.2.1 Properties in more restricted classes of structures

lemma *sum-Un*:

```

finite A ⇒ finite B ⇒ sum f (A ∪ B) = sum f A + sum f B - sum f (A ∩ B)
for f :: 'b ⇒ 'a::ab-group-add
by (subst sum.union-inter [symmetric]) (auto simp add: algebra-simps)

```

lemma *sum-Un2*:

```

assumes finite (A ∪ B)
shows sum f (A ∪ B) = sum f (A - B) + sum f (B - A) + sum f (A ∩ B)
proof -
  have A ∪ B = A - B ∪ (B - A) ∪ A ∩ B
  by auto
  with assms show ?thesis
  by simp (subst sum.union-disjoint, auto)+
qed

```

lemma *sum-diff*:

```

fixes f :: 'b ⇒ 'a::ab-group-add
assumes finite A B ⊆ A
shows sum f (A - B) = sum f A - sum f B
using sum.subset-diff [of B A f] assms by simp

```

lemma *sum-diff1*:

```

fixes f :: 'b ⇒ 'a::ab-group-add
assumes finite A
shows sum f (A - {a}) = (if a ∈ A then sum f A - f a else sum f A)
using assms by (simp add: sum-diff)

```

lemma *sum-diff1'-aux*:

```

fixes f :: 'a ⇒ 'b::ab-group-add

```



```

assumes finite  $F \{i \in I. f i \neq 0\} \subseteq F$ 
shows  $\text{sum}' f (I - \{i\}) = (\text{if } i \in I \text{ then } \text{sum}' f I - f i \text{ else } \text{sum}' f I)$ 
using assms
proof induct
  case (insert  $x F$ )
  have 1:  $\text{finite } \{x \in I. f x \neq 0\} \implies \text{finite } \{x \in I. x \neq i \wedge f x \neq 0\}$ 
    by (erule rev-finite-subset) auto
  have 2:  $\text{finite } \{x \in I. x \neq i \wedge f x \neq 0\} \implies \text{finite } \{x \in I. f x \neq 0\}$ 
    apply (drule finite-insert [THEN iffD2])
    by (erule rev-finite-subset) auto
  have 3:  $\text{finite } \{i \in I. f i \neq 0\}$ 
    using finite-subset insert by blast
  show ?case
    using insert sum-diff1 [of {i \in I. f i \neq 0} f i]
    by (auto simp: sum.G-def 1 2 3 set-diff-eq conj-ac)
qed (simp add: sum.G-def)

```

```

lemma sum-diff1':
  fixes  $f :: 'a \Rightarrow 'b::\text{ab-group-add}$ 
  assumes  $\text{finite } \{i \in I. f i \neq 0\}$ 
  shows  $\text{sum}' f (I - \{i\}) = (\text{if } i \in I \text{ then } \text{sum}' f I - f i \text{ else } \text{sum}' f I)$ 
  by (rule sum-diff1'-aux [OF assms order-refl])

```

```

lemma (in ordered-comm-monoid-add) sum-mono:
   $(\bigwedge i. i \in K \implies f i \leq g i) \implies (\sum i \in K. f i) \leq (\sum i \in K. g i)$ 
  by (induct K rule: infinite-finite-induct) (use add-mono in auto)

```

```

lemma (in strict-ordered-comm-monoid-add) sum-strict-mono:
  assumes  $\text{finite } A \ A \neq \{\}$ 
  and  $\bigwedge x. x \in A \implies f x < g x$ 
  shows  $\text{sum } f A < \text{sum } g A$ 
  using assms
proof (induct rule: finite-ne-induct)
  case singleton
  then show ?case by simp
next
  case insert
  then show ?case by (auto simp: add-strict-mono)
qed

```

```

lemma sum-strict-mono-ex1:
  fixes  $f g :: 'i \Rightarrow 'a::\text{ordered-cancel-comm-monoid-add}$ 
  assumes  $\text{finite } A$ 
  and  $\forall x \in A. f x \leq g x$ 
  and  $\exists a \in A. f a < g a$ 
  shows  $\text{sum } f A < \text{sum } g A$ 
proof –
  from assms(3) obtain  $a$  where  $a \in A \ f a < g a$  by blast
  have  $\text{sum } f A = \text{sum } f ((A - \{a\}) \cup \{a\})$ 

```

```

  by (simp add: insert-absorb[OF ‹a ∈ A›])
  also have ... = sum f (A - {a}) + sum f {a}
    using ‹finite A› by (subst sum.union-disjoint) auto
  also have sum f (A - {a}) ≤ sum g (A - {a})
    by (rule sum-mono) (simp add: assms(2))
  also from a have sum f {a} < sum g {a} by simp
  also have sum g (A - {a}) + sum g {a} = sum g ((A - {a}) ∪ {a})
    using ‹finite A› by (subst sum.union-disjoint[symmetric]) auto
  also have ... = sum g A by (simp add: insert-absorb[OF ‹a ∈ A›])
  finally show ?thesis
    by (auto simp add: add-right-mono add-strict-left-mono)
qed

```

lemma *sum-mono-inv*:

```

  fixes f g :: 'i ⇒ 'a :: ordered-cancel-comm-monoid-add
  assumes eq: sum f I = sum g I
  assumes le: ∧i. i ∈ I ⇒ f i ≤ g i
  assumes i: i ∈ I
  assumes I: finite I
  shows f i = g i
proof (rule ccontr)
  assume ¬ ?thesis
  with le[OF i] have f i < g i by simp
  with i have ∃ i ∈ I. f i < g i ..
  from sum-strict-mono-ex1[OF I - this] le have sum f I < sum g I
    by blast
  with eq show False by simp
qed

```

lemma *member-le-sum*:

```

  fixes f :: - ⇒ 'b::{semiring-1, ordered-comm-monoid-add}
  assumes i ∈ A
  and le: ∧x. x ∈ A - {i} ⇒ 0 ≤ f x
  and finite A
  shows f i ≤ sum f A
proof -
  have f i ≤ sum f (A ∩ {i})
    by (simp add: assms)
  also have ... = (∑ x ∈ A. if x ∈ {i} then f x else 0)
    using assms sum.inter-restrict by blast
  also have ... ≤ sum f A
    apply (rule sum-mono)
    apply (auto simp: le)
    done
  finally show ?thesis .
qed

```

lemma *sum-negf*: $(\sum x \in A. - f x) = - (\sum x \in A. f x)$
 for $f :: 'b \Rightarrow 'a :: ab\text{-group-add}$

by (induct A rule: infinite-finite-induct) auto

lemma *sum-subtractf*: $(\sum x \in A. f x - g x) = (\sum x \in A. f x) - (\sum x \in A. g x)$
 for $f g :: 'b \Rightarrow 'a :: ab\text{-group-add}$
 using *sum.distrib* [of $f - g$ A] by (simp add: *sum-negf*)

lemma *sum-subtractf-nat*:
 $(\bigwedge x. x \in A \implies g x \leq f x) \implies (\sum x \in A. f x - g x) = (\sum x \in A. f x) - (\sum x \in A. g x)$
 for $f g :: 'a \Rightarrow nat$
 by (induct A rule: infinite-finite-induct) (auto simp: *sum-mono*)

context *ordered-comm-monoid-add*
begin

lemma *sum-nonneg*: $(\bigwedge x. x \in A \implies 0 \leq f x) \implies 0 \leq \text{sum } f A$
proof (induct A rule: infinite-finite-induct)
 case *infinite*
 then show ?case by simp
next
 case *empty*
 then show ?case by simp
next
 case (insert x F)
 then have $0 + 0 \leq f x + \text{sum } f F$ by (blast intro: *add-mono*)
 with insert show ?case by simp
qed

lemma *sum-nonpos*: $(\bigwedge x. x \in A \implies f x \leq 0) \implies \text{sum } f A \leq 0$
proof (induct A rule: infinite-finite-induct)
 case *infinite*
 then show ?case by simp
next
 case *empty*
 then show ?case by simp
next
 case (insert x F)
 then have $f x + \text{sum } f F \leq 0 + 0$ by (blast intro: *add-mono*)
 with insert show ?case by simp
qed

lemma *sum-nonneg-eq-0-iff*:
 $\text{finite } A \implies (\bigwedge x. x \in A \implies 0 \leq f x) \implies \text{sum } f A = 0 \iff (\forall x \in A. f x = 0)$
 by (induct set: *finite*) (simp-all add: *add-nonneg-eq-0-iff sum-nonneg*)

lemma *sum-nonneg-0*:
 $\text{finite } s \implies (\bigwedge i. i \in s \implies f i \geq 0) \implies (\sum i \in s. f i) = 0 \implies i \in s \implies f i = 0$
 by (simp add: *sum-nonneg-eq-0-iff*)

lemma *sum-nonneg-leq-bound*:

assumes *finite s* $\wedge i. i \in s \implies f i \geq 0$ $(\sum i \in s. f i) = B$ $i \in s$
shows $f i \leq B$

proof –

from *assms* **have** $f i \leq f i + (\sum i \in s - \{i\}. f i)$

by (*intro add-increasing2 sum-nonneg*) *auto*

also have $\dots = B$

using *sum.remove[of s i f]* *assms* **by** *simp*

finally show *?thesis* **by** *auto*

qed

lemma *sum-mono2*:

assumes *fin: finite B*

and *sub: A* \subseteq *B*

and *nn: $\bigwedge b. b \in B - A \implies 0 \leq f b$*

shows $\text{sum } f A \leq \text{sum } f B$

proof –

have $\text{sum } f A \leq \text{sum } f A + \text{sum } f (B - A)$

by (*auto intro: add-increasing2 [OF sum-nonneg] nn*)

also from *fin finite-subset[OF sub fin]* **have** $\dots = \text{sum } f (A \cup (B - A))$

by (*simp add: sum.union-disjoint del: Un-Diff-cancel*)

also from *sub* **have** $A \cup (B - A) = B$ **by** *blast*

finally show *?thesis* .

qed

lemma *sum-le-included*:

assumes *finite s finite t*

and $\forall y \in t. 0 \leq g y$ $(\forall x \in s. \exists y \in t. i y = x \wedge f x \leq g y)$

shows $\text{sum } f s \leq \text{sum } g t$

proof –

have $\text{sum } f s \leq \text{sum } (\lambda y. \text{sum } g \{x. x \in t \wedge i x = y\}) s$

proof (*rule sum-mono*)

fix *y*

assume $y \in s$

with *assms* **obtain** *z* **where** $z \in t$ $y = i z$ $f y \leq g z$ **by** *auto*

with *assms* **show** $f y \leq \text{sum } g \{x \in t. i x = y\}$ (**is** $?A y \leq ?B y$)

using *order-trans[of ?A (i z) sum g {z} ?B (i z), intro]*

by (*auto intro!: sum-mono2*)

qed

also have $\dots \leq \text{sum } (\lambda y. \text{sum } g \{x. x \in t \wedge i x = y\}) (i \text{ ` } t)$

using *assms(2-4)* **by** (*auto intro!: sum-mono2 sum-nonneg*)

also have $\dots \leq \text{sum } g t$

using *assms* **by** (*auto simp: sum.image-gen[symmetric]*)

finally show *?thesis* .

qed

end

lemma (*in canonically-ordered-monoid-add*) *sum-eq-0-iff* [*simp*]:

$finite\ F \implies (sum\ f\ F = 0) = (\forall a \in F. f\ a = 0)$
by (*intro ballI sum-nonneg-eq-0-iff zero-le*)

context *semiring-0*
begin

lemma *sum-distrib-left*: $r * sum\ f\ A = (\sum n \in A. r * f\ n)$
by (*induct A rule: infinite-finite-induct*) (*simp-all add: algebra-simps*)

lemma *sum-distrib-right*: $sum\ f\ A * r = (\sum n \in A. f\ n * r)$
by (*induct A rule: infinite-finite-induct*) (*simp-all add: algebra-simps*)

end

lemma *sum-divide-distrib*: $sum\ f\ A / r = (\sum n \in A. f\ n / r)$

for $r :: 'a::field$

proof (*induct A rule: infinite-finite-induct*)

case *infinite*

then show *?case by simp*

next

case *empty*

then show *?case by simp*

next

case *insert*

then show *?case by (simp add: add-divide-distrib)*

qed

lemma *sum-abs[iff]*: $|sum\ f\ A| \leq sum\ (\lambda i. |f\ i|)\ A$

for $f :: 'a \Rightarrow 'b::ordered-ab-group-add-abs$

proof (*induct A rule: infinite-finite-induct*)

case *infinite*

then show *?case by simp*

next

case *empty*

then show *?case by simp*

next

case *insert*

then show *?case by (auto intro: abs-triangle-ineq order-trans)*

qed

lemma *sum-abs-ge-zero[iff]*: $0 \leq sum\ (\lambda i. |f\ i|)\ A$

for $f :: 'a \Rightarrow 'b::ordered-ab-group-add-abs$

by (*simp add: sum-nonneg*)

lemma *abs-sum-abs[simp]*: $|\sum a \in A. |f\ a|| = (\sum a \in A. |f\ a|)$

for $f :: 'a \Rightarrow 'b::ordered-ab-group-add-abs$

proof (*induct A rule: infinite-finite-induct*)

case *infinite*

then show *?case by simp*

```

next
  case empty
  then show ?case by simp
next
  case (insert a A)
  then have  $|\sum a \in \text{insert } a \ A. |f \ a|| = ||f \ a| + (\sum a \in A. |f \ a|)|$  by simp
  also from insert have  $\dots = ||f \ a| + |\sum a \in A. |f \ a||$  by simp
  also have  $\dots = |f \ a| + |\sum a \in A. |f \ a|$  by (simp del: abs-of-nonneg)
  also from insert have  $\dots = (\sum a \in \text{insert } a \ A. |f \ a|)$  by simp
  finally show ?case .
qed

```

lemma *sum-product*:

```

fixes f :: 'a  $\Rightarrow$  'b::semiring-0
shows sum f A * sum g B = ( $\sum i \in A. \sum j \in B. f \ i * g \ j$ )
by (simp add: sum-distrib-left sum-distrib-right) (rule sum.swap)

```

lemma *sum-mult-sum-if-inj*:

```

fixes f :: 'a  $\Rightarrow$  'b::semiring-0
shows inj-on ( $\lambda(a, b). f \ a * g \ b$ ) (A  $\times$  B)  $\implies$ 
  sum f A * sum g B = sum id {f a * g b | a b. a  $\in$  A  $\wedge$  b  $\in$  B}
by(auto simp: sum-product sum.cartesian-product intro!: sum.reindex-cong[symmetric])

```

lemma *sum-SucD*: $\text{sum } f \ A = \text{Suc } n \implies \exists a \in A. 0 < f \ a$
 by (induct A rule: infinite-finite-induct) auto

lemma *sum-eq-Suc0-iff*:

```

finite A  $\implies$  sum f A = Suc 0  $\iff$  ( $\exists a \in A. f \ a = \text{Suc } 0 \wedge (\forall b \in A. a \neq b \longrightarrow f \ b = 0)$ )
by (induct A rule: finite-induct) (auto simp add: add-is-1)

```

lemmas *sum-eq-1-iff* = *sum-eq-Suc0-iff*[*simplified One-nat-def*[*symmetric*]]

lemma *sum-Un-nat*:

```

finite A  $\implies$  finite B  $\implies$  sum f (A  $\cup$  B) = sum f A + sum f B - sum f (A  $\cap$  B)
for f :: 'a  $\Rightarrow$  nat
— For the natural numbers, we have subtraction.
by (subst sum.union-inter [symmetric]) (auto simp: algebra-simps)

```

lemma *sum-diff1-nat*: $\text{sum } f \ (A - \{a\}) = (\text{if } a \in A \text{ then } \text{sum } f \ A - f \ a \text{ else } \text{sum } f \ A)$
 for f :: 'a \Rightarrow nat
proof (induct A rule: infinite-finite-induct)
 case infinite
 then show ?case by simp
next
 case empty
 then show ?case by simp
next

```

case (insert  $x$   $F$ )
then show ?case
proof (cases  $a \in F$ )
  case True
  then have  $\exists B. F = \text{insert } a B \wedge a \notin B$ 
  by (auto simp: mk-disjoint-insert)
  then show ?thesis using insert
  by (auto simp: insert-Diff-if)
qed (auto)
qed

```

lemma *sum-diff-nat*:

```

fixes  $f :: 'a \Rightarrow \text{nat}$ 
assumes finite  $B$  and  $B \subseteq A$ 
shows  $\text{sum } f (A - B) = \text{sum } f A - \text{sum } f B$ 
using assms
proof induct
  case empty
  then show ?case by simp
next
  case (insert  $x$   $F$ )
  note  $IH = \langle F \subseteq A \implies \text{sum } f (A - F) = \text{sum } f A - \text{sum } f F \rangle$ 
  from  $\langle x \notin F \rangle \langle \text{insert } x F \subseteq A \rangle$  have  $x \in A - F$  by simp
  then have  $A: \text{sum } f ((A - F) - \{x\}) = \text{sum } f (A - F) - f x$ 
  by (simp add: sum-diff1-nat)
  from  $\langle \text{insert } x F \subseteq A \rangle$  have  $F \subseteq A$  by simp
  with  $IH$  have  $\text{sum } f (A - F) = \text{sum } f A - \text{sum } f F$  by simp
  with  $A$  have  $B: \text{sum } f ((A - F) - \{x\}) = \text{sum } f A - \text{sum } f F - f x$ 
  by simp
  from  $\langle x \notin F \rangle$  have  $A - \text{insert } x F = (A - F) - \{x\}$  by auto
  with  $B$  have  $C: \text{sum } f (A - \text{insert } x F) = \text{sum } f A - \text{sum } f F - f x$ 
  by simp
  from  $\langle \text{finite } F \rangle \langle x \notin F \rangle$  have  $\text{sum } f (\text{insert } x F) = \text{sum } f F + f x$ 
  by simp
  with  $C$  have  $\text{sum } f (A - \text{insert } x F) = \text{sum } f A - \text{sum } f (\text{insert } x F)$ 
  by simp
  then show ?case by simp
qed

```

lemma *sum-comp-morphism*:

```

 $h 0 = 0 \implies (\bigwedge x y. h (x + y) = h x + h y) \implies \text{sum } (h \circ g) A = h (\text{sum } g A)$ 
by (induct  $A$  rule: infinite-finite-induct) simp-all

```

lemma (*in comm-semiring-1*) *dvd-sum*: $(\bigwedge a. a \in A \implies d \text{ dvd } f a) \implies d \text{ dvd } \text{sum } f A$

```

by (induct  $A$  rule: infinite-finite-induct) simp-all

```

lemma (*in ordered-comm-monoid-add*) *sum-pos*:

```

 $\text{finite } I \implies I \neq \{\} \implies (\bigwedge i. i \in I \implies 0 < f i) \implies 0 < \text{sum } f I$ 

```

by (induct I rule: finite-ne-induct) (auto intro: add-pos-pos)

lemma (in ordered-comm-monoid-add) sum-pos2:

assumes I: finite I $i \in I \Rightarrow 0 < f i \wedge i. i \in I \Rightarrow 0 \leq f i$

shows $0 < \text{sum } f I$

proof –

have $0 < f i + \text{sum } f (I - \{i\})$

using assms by (intro add-pos-nonneg sum-nonneg) auto

also have $\dots = \text{sum } f I$

using assms by (simp add: sum.remove)

finally show ?thesis .

qed

lemma sum-strict-mono2:

fixes $f :: 'a \Rightarrow 'b::\text{ordered-cancel-comm-monoid-add}$

assumes finite B $A \subseteq B$ $b \in B - A$ $f b > 0$ and $\bigwedge x. x \in B \Rightarrow f x \geq 0$

shows $\text{sum } f A < \text{sum } f B$

proof –

have $B - A \neq \{\}$

using assms(3) by blast

have $\text{sum } f (B - A) > 0$

by (rule sum-pos2) (use assms in auto)

moreover have $\text{sum } f B = \text{sum } f (B - A) + \text{sum } f A$

by (rule sum.subset-diff) (use assms in auto)

ultimately show ?thesis

using add-strict-increasing by auto

qed

lemma sum-cong-Suc:

assumes $0 \notin A$ $\bigwedge x. \text{Suc } x \in A \Rightarrow f (\text{Suc } x) = g (\text{Suc } x)$

shows $\text{sum } f A = \text{sum } g A$

proof (rule sum.cong)

fix x

assume $x \in A$

with assms(1) show $f x = g x$

by (cases x) (auto intro!: assms(2))

qed simp-all

48.2.2 Cardinality as special case of sum

lemma card-eq-sum: $\text{card } A = \text{sum } (\lambda x. 1) A$

proof –

have $\text{plus} \circ (\lambda -. \text{Suc } 0) = (\lambda -. \text{Suc})$

by (simp add: fun-eq-iff)

then have $\text{Finite-Set.fold } (\text{plus} \circ (\lambda -. \text{Suc } 0)) = \text{Finite-Set.fold } (\lambda -. \text{Suc})$

by (rule arg-cong)

then have $\text{Finite-Set.fold } (\text{plus} \circ (\lambda -. \text{Suc } 0)) 0 A = \text{Finite-Set.fold } (\lambda -. \text{Suc}) 0$

A

by (blast intro: fun-cong)


```

then show ?thesis
  by (simp add: card.eq-fold sum.eq-fold)
qed

```

```

context semiring-1
begin

```

```

lemma sum-constant [simp]:
   $(\sum x \in A. y) = \text{of-nat} (\text{card } A) * y$ 
  by (induct A rule: infinite-finite-induct) (simp-all add: algebra-simps)

```

```

context
  fixes A
  assumes ⟨finite A⟩
begin

```

```

lemma sum-of-bool-eq [simp]:
   $\langle \sum x \in A. \text{of-bool} (P x) = \text{of-nat} (\text{card} (A \cap \{x. P x\})) \rangle$  if ⟨finite A⟩
  using ⟨finite A⟩ by induction simp-all

```

```

lemma sum-mult-of-bool-eq [simp]:
   $\langle \sum x \in A. f x * \text{of-bool} (P x) = (\sum x \in (A \cap \{x. P x\}). f x) \rangle$ 
  by (rule sum.mono-neutral-cong) (use ⟨finite A⟩ in auto)

```

```

lemma sum-of-bool-mult-eq [simp]:
   $\langle \sum x \in A. \text{of-bool} (P x) * f x = (\sum x \in (A \cap \{x. P x\}). f x) \rangle$ 
  by (rule sum.mono-neutral-cong) (use ⟨finite A⟩ in auto)

```

```

end

```

```

end

```

```

lemma sum-Suc:  $\text{sum} (\lambda x. \text{Suc}(f x)) A = \text{sum } f A + \text{card } A$ 
  using sum.distrib[of f λ-. 1 A] by simp

```

```

lemma sum-bounded-above:
  fixes K :: 'a::{semiring-1, ordered-comm-monoid-add}
  assumes le:  $\bigwedge i. i \in A \implies f i \leq K$ 
  shows  $\text{sum } f A \leq \text{of-nat} (\text{card } A) * K$ 
proof (cases finite A)
  case True
  then show ?thesis
    using le sum-mono[where K=A and g = λx. K] by simp
  next
  case False
  then show ?thesis by simp
qed

```

```

lemma sum-bounded-above-divide:

```

fixes $K :: 'a::linordered-field$
 assumes $le: \bigwedge i. i \in A \implies f i \leq K / of\text{-}nat (card A)$ and $fin: finite A A \neq \{\}$
 shows $sum f A \leq K$
 using $sum\text{-}bounded\text{-}above [of A f K / of\text{-}nat (card A), OF le] fin$ by $simp$

lemma $sum\text{-}bounded\text{-}above\text{-}strict$:
 fixes $K :: 'a::\{ordered\text{-}cancel\text{-}comm\text{-}monoid\text{-}add, semiring\text{-}1\}$
 assumes $\bigwedge i. i \in A \implies f i < K$ and $card A > 0$
 shows $sum f A < of\text{-}nat (card A) * K$
 using $assms sum\text{-}strict\text{-}mono$ [where $A=A$ and $g = \lambda x. K$]
 by ($simp add: card\text{-}gt\text{-}0\text{-}iff$)

lemma $sum\text{-}bounded\text{-}below$:
 fixes $K :: 'a::\{semiring\text{-}1, ordered\text{-}comm\text{-}monoid\text{-}add\}$
 assumes $le: \bigwedge i. i \in A \implies K \leq f i$
 shows $of\text{-}nat (card A) * K \leq sum f A$
 proof ($cases finite A$)
 case $True$
 then show $?thesis$
 using $le sum\text{-}mono$ [where $K=A$ and $f = \lambda x. K$] by $simp$
 next
 case $False$
 then show $?thesis$ by $simp$
 qed

lemma $convex\text{-}sum\text{-}bound\text{-}le$:
 fixes $x :: 'a \Rightarrow 'b::linordered\text{-}idom$
 assumes $0: \bigwedge i. i \in I \implies 0 \leq x i$ and $1: sum x I = 1$
 and $\delta: \bigwedge i. i \in I \implies |a i - b| \leq \delta$
 shows $|\sum i \in I. a i * x i - b| \leq \delta$
 proof -
 have $[simp]: (\sum i \in I. c * x i) = c$ for c
 by ($simp flip: sum\text{-}distrib\text{-}left 1$)
 then have $|\sum i \in I. a i * x i - b| = |\sum i \in I. (a i - b) * x i|$
 by ($simp add: sum\text{-}subtractf left\text{-}diff\text{-}distrib$)
 also have $\dots \leq (\sum i \in I. |(a i - b) * x i|)$
 using $abs\text{-}abs abs\text{-}of\text{-}nonneg$ by $blast$
 also have $\dots \leq (\sum i \in I. |(a i - b)| * x i)$
 by ($simp add: abs\text{-}mult 0$)
 also have $\dots \leq (\sum i \in I. \delta * x i)$
 by ($rule sum\text{-}mono$) (use $\delta 0 mult\text{-}right\text{-}mono$ in $blast$)
 also have $\dots = \delta$
 by $simp$
 finally show $?thesis$.
 qed

lemma $card\text{-}UN\text{-}disjoint$:
 assumes $finite I$ and $\forall i \in I. finite (A i)$
 and $\forall i \in I. \forall j \in I. i \neq j \longrightarrow A i \cap A j = \{\}$

shows $\text{card} (\bigcup (A \text{ ' } I)) = (\sum i \in I. \text{card}(A \ i))$
proof –
have $(\sum i \in I. \text{card} (A \ i)) = (\sum i \in I. \sum x \in A \ i. 1)$
by *simp*
with *assms* **show** *?thesis*
by (*simp add: card-eq-sum sum.UNION-disjoint del: sum-constant*)
qed

lemma *card-Union-disjoint*:
assumes *pairwise disjnt C and fin: $\bigwedge A. A \in C \implies \text{finite } A$*
shows $\text{card} (\bigcup C) = \text{sum card } C$
proof (*cases finite C*)
case *True*
then show *?thesis*
using *card-UN-disjoint [OF True, of $\lambda x. x$] assms*
by (*simp add: disjnt-def fin pairwise-def*)
next
case *False*
then show *?thesis*
using *assms card-eq-0-iff finite-UnionD by fastforce*
qed

lemma *card-Union-le-sum-card-weak*:
fixes $U :: 'a \text{ set set}$
assumes $\forall u \in U. \text{finite } u$
shows $\text{card} (\bigcup U) \leq \text{sum card } U$
proof (*cases finite U*)
case *False*
then show $\text{card} (\bigcup U) \leq \text{sum card } U$
using *card-eq-0-iff finite-UnionD by auto*
next
case *True*
then show $\text{card} (\bigcup U) \leq \text{sum card } U$
proof (*induct U rule: finite-induct*)
case *empty*
then show *?case by auto*
next
case (*insert x F*)
then have $\text{card}(\bigcup(\text{insert } x \ F)) \leq \text{card}(x) + \text{card} (\bigcup F)$ **using** *card-Un-le by auto*
also have $\dots \leq \text{card}(x) + \text{sum card } F$ **using** *insert.hyps by auto*
also have $\dots = \text{sum card} (\text{insert } x \ F)$ **using** *sum.insert-if and insert.hyps by auto*
finally show *?case .*
qed
qed

lemma *card-Union-le-sum-card*:
fixes $U :: 'a \text{ set set}$

shows $\text{card} (\bigcup U) \leq \text{sum card } U$
 by (*metis Union-upper card.infinite card-Union-le-sum-card-weak finite-subset zero-le*)

lemma *card-UN-le*:

assumes *finite I*

shows $\text{card}(\bigcup_{i \in I}. A\ i) \leq (\sum_{i \in I}. \text{card}(A\ i))$

using *assms*

proof *induction*

case (*insert i I*)

then show *?case*

using *card-Un-le nat-add-left-cancel-le* by (*force intro: order-trans*)

qed *auto*

lemma *card-quotient-disjoint*:

assumes *finite A inj-on* $(\lambda x. \{x\} // r)$ *A*

shows $\text{card} (A//r) = \text{card } A$

proof –

have $\forall i \in A. \forall j \in A. i \neq j \longrightarrow r \text{ “ } \{j\} \neq r \text{ “ } \{i\}$

using *assms* by (*fastforce simp add: quotient-def inj-on-def*)

with *assms* **show** *?thesis*

by (*simp add: quotient-def card-UN-disjoint*)

qed

lemma *sum-multicount-gen*:

assumes *finite s finite t* $\forall j \in t. (\text{card } \{i \in s. R\ i\ j\} = k\ j)$

shows $\text{sum } (\lambda i. (\text{card } \{j \in t. R\ i\ j\}))\ s = \text{sum } k\ t$

(*is ?l = ?r*)

proof –

have $?l = \text{sum } (\lambda i. \text{sum } (\lambda x.1)\ \{j \in t. R\ i\ j\})\ s$

by *auto*

also have $\dots = ?r$

unfolding *sum.swap-restrict* [*OF assms(1-2)*]

using *assms(3)* **by** *auto*

finally **show** *?thesis* .

qed

lemma *sum-multicount*:

assumes *finite S finite T* $\forall j \in T. (\text{card } \{i \in S. R\ i\ j\} = k)$

shows $\text{sum } (\lambda i. \text{card } \{j \in T. R\ i\ j\})\ S = k * \text{card } T$ (*is ?l = ?r*)

proof –

have $?l = \text{sum } (\lambda i. k)\ T$

by (*rule sum-multicount-gen*) (*auto simp: assms*)

also have $\dots = ?r$ **by** (*simp add: mult.commute*)

finally **show** *?thesis* **by** *auto*

qed

lemma *sum-card-image*:

assumes *finite A*

```

assumes pairwise ( $\lambda s t. \text{disjnt } (f s) (f t)$ )  $A$ 
shows  $\text{sum card } (f \text{ ' } A) = \text{sum } (\lambda a. \text{card } (f a)) A$ 
using assms
proof (induct A)
  case (insert a A)
  show ?case
  proof cases
    assume  $f a = \{\}$ 
    with insert show ?case
    by (subst sum.mono-neutral-right[where  $S=f \text{ ' } A$ ]) (auto simp: pairwise-insert)
  next
    assume  $f a \neq \{\}$ 
    then have  $\text{sum card } (\text{insert } (f a) (f \text{ ' } A)) = \text{card } (f a) + \text{sum card } (f \text{ ' } A)$ 
    using insert
    by (subst sum.insert) (auto simp: pairwise-insert)
    with insert show ?case by (simp add: pairwise-insert)
  qed
qed simp

```

By Jakub Kdzioka:

lemma *sum-fun-comp*:

```

assumes finite S finite R g \text{ ' } S \subseteq R
shows  $(\sum x \in S. f (g x)) = (\sum y \in R. \text{of-nat } (\text{card } \{x \in S. g x = y\}) * f y)$ 
proof –
  let ?r = relation-of ( $\lambda p q. g p = g q$ )  $S$ 
  have eqv: equiv S ?r
  unfolding relation-of-def by (auto intro: comp-equivI)
  have finite: C \in S // ?r \implies finite C for  $C$ 
  by (fact finite-equiv-class[OF  $\langle \text{finite } S \rangle$  equiv-type[OF  $\langle \text{equiv } S \text{ ?r} \rangle$ ]])
  have disjoint: A \in S // ?r \implies B \in S // ?r \implies A \neq B \implies A \cap B = \{\} for  $A B$ 
  using eqv quotient-disj by blast

  let ?cls = \lambda y. \{x \in S. y = g x\}
  have quot-as-img: S // ?r = ?cls \text{ ' } g \text{ ' } S
  by (auto simp add: relation-of-def quotient-def)
  have cls-inj: inj-on ?cls (g \text{ ' } S)
  by (auto intro: inj-onI)

  have rest-0: (\sum y \in R - g \text{ ' } S. \text{of-nat } (\text{card } (?cls y)) * f y) = 0
  proof –
    have of-nat (card (?cls y)) * f y = 0 if asm: y \in R - g \text{ ' } S for  $y$ 
    proof –
      from asm have  $*: ?cls y = \{\}$  by auto
      show ?thesis unfolding  $*$  by simp
    qed
    thus ?thesis by simp
  qed

  have  $(\sum x \in S. f (g x)) = (\sum C \in S // ?r. \sum x \in C. f (g x))$ 

```

```

using eqv finite disjoint
by (simp flip: sum.Union-disjoint[simplified] add: Union-quotient)
also have ... =  $(\sum y \in g \text{ ' } S. \sum x \in ?cls \ y. f \ (g \ x))$ 
unfolding quot-as-img by (simp add: sum.reindex[OF cls-inj])
also have ... =  $(\sum y \in g \text{ ' } S. \sum x \in ?cls \ y. f \ y)$ 
by auto
also have ... =  $(\sum y \in g \text{ ' } S. of\text{-nat} \ (card \ (?cls \ y)) * f \ y)$ 
by (simp flip: sum-constant)
also have ... =  $(\sum y \in R. of\text{-nat} \ (card \ (?cls \ y)) * f \ y)$ 
using rest-0 by (simp add: sum.subset-diff[OF ‹g ′ S ⊆ R› ‹finite R›])
finally show ?thesis
by (simp add: eq-commute)
qed

```

48.2.3 Cardinality of products

lemma *card-SigmaI [simp]*:

```

finite A  $\implies \forall a \in A. \text{finite} \ (B \ a) \implies card \ (SIGMA \ x: A. B \ x) = (\sum a \in A. card \ (B \ a))$ 
by (simp add: card-eq-sum sum.Sigma del: sum-constant)

```

lemma *card-cartesian-product*: $card \ (A \times B) = card \ A * card \ B$

```

by (cases finite A ∧ finite B)
(auto simp add: card-eq-0-iff dest: finite-cartesian-productD1 finite-cartesian-productD2)

```

lemma *card-cartesian-product-singleton*: $card \ (\{x\} \times A) = card \ A$

```

by (simp add: card-cartesian-product)

```

48.3 Generalized product over a set

context *comm-monoid-mult*

begin

sublocale *prod: comm-monoid-set times 1*

```

defines prod = prod.F and prod' = prod.G ..

```

abbreviation *Prod* ($\prod - [1000] 999$)

```

where  $\prod A \equiv prod \ (\lambda x. x) \ A$ 

```

end

syntax (*ASCII*)

```

-prod :: ptrn => 'a set => 'b => 'b::comm-monoid-mult ((4PROD (-/:-)/ -)
[0, 51, 10] 10)

```

syntax

```

-prod :: ptrn => 'a set => 'b => 'b::comm-monoid-mult ((2∏ (-/∈-)/ -) [0,
51, 10] 10)

```

translations — Beware of argument permutation!

$\prod_{i \in A}. b == \text{CONST prod } (\lambda i. b) A$

Instead of $\prod_{x \in \{x. P\}}. e$ we introduce the shorter $\prod x | P. e$.

syntax (ASCII)

$-qprod :: pttm \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\text{4PROD} - | / - / -) [0, 0, 10] 10)$

syntax

$-qprod :: pttm \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\text{2}\prod - | (-) / -) [0, 0, 10] 10)$

translations

$\prod x | P. t \Rightarrow \text{CONST prod } (\lambda x. t) \{x. P\}$

context *comm-monoid-mult*

begin

lemma *prod-dvd-prod*: $(\bigwedge a. a \in A \Longrightarrow f a \text{ dvd } g a) \Longrightarrow \text{prod } f A \text{ dvd } \text{prod } g A$

proof (*induct A rule: infinite-finite-induct*)

case *infinite*

then show *?case* **by** (*auto intro: dvdI*)

next

case *empty*

then show *?case* **by** (*auto intro: dvdI*)

next

case (*insert a A*)

then have $f a \text{ dvd } g a$ **and** $\text{prod } f A \text{ dvd } \text{prod } g A$

by *simp-all*

then obtain $r s$ **where** $g a = f a * r$ **and** $\text{prod } g A = \text{prod } f A * s$

by (*auto elim!: dvdE*)

then have $g a * \text{prod } g A = f a * \text{prod } f A * (r * s)$

by (*simp add: ac-simps*)

with *insert.hyps* **show** *?case*

by (*auto intro: dvdI*)

qed

lemma *prod-dvd-prod-subset*: $\text{finite } B \Longrightarrow A \subseteq B \Longrightarrow \text{prod } f A \text{ dvd } \text{prod } f B$

by (*auto simp add: prod.subset-diff ac-simps intro: dvdI*)

end

48.3.1 Properties in more restricted classes of structures

context *linordered-nonzero-semiring*

begin

lemma *prod-ge-1*: $(\bigwedge x. x \in A \Longrightarrow 1 \leq f x) \Longrightarrow 1 \leq \text{prod } f A$

proof (*induct A rule: infinite-finite-induct*)

case *infinite*

then show *?case* **by** *simp*

next

case *empty*

then show *?case* **by** *simp*

```

next
  case (insert x F)
  have  $1 * 1 \leq f x * \text{prod } f F$ 
    by (rule mult-mono') (use insert in auto)
  with insert show ?case by simp
qed

lemma prod-le-1:
  fixes f :: 'b  $\Rightarrow$  'a
  assumes  $\bigwedge x. x \in A \implies 0 \leq f x \wedge f x \leq 1$ 
  shows  $\text{prod } f A \leq 1$ 
    using assms
proof (induct A rule: infinite-finite-induct)
  case infinte
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case (insert x F)
  then show ?case by (force simp: mult.commute intro: dest: mult-le-one)
qed

end

context comm-semiring-1
begin

lemma dvd-prod-eqI [intro]:
  assumes finite A and  $a \in A$  and  $b = f a$ 
  shows  $b \text{ dvd } \text{prod } f A$ 
proof -
  from  $\langle \text{finite } A \rangle$  have  $\text{prod } f (\text{insert } a (A - \{a\})) = f a * \text{prod } f (A - \{a\})$ 
    by (intro prod.insert) auto
  also from  $\langle a \in A \rangle$  have  $\text{insert } a (A - \{a\}) = A$ 
    by blast
  finally have  $\text{prod } f A = f a * \text{prod } f (A - \{a\})$  .
  with  $\langle b = f a \rangle$  show ?thesis
    by simp
qed

lemma dvd-prodI [intro]:  $\text{finite } A \implies a \in A \implies f a \text{ dvd } \text{prod } f A$ 
  by auto

lemma prod-zero:
  assumes finite A and  $\exists a \in A. f a = 0$ 
  shows  $\text{prod } f A = 0$ 
  using assms
proof (induct A)

```



```

  case empty
  then show ?case by simp
next
  case (insert a A)
  then have  $f a = 0 \vee (\exists a \in A. f a = 0)$  by simp
  then have  $f a * \text{prod } f A = 0$  by (rule disjE) (simp-all add: insert)
  with insert show ?case by simp
qed

```

```

lemma prod-dvd-prod-subset2:
  assumes finite B and  $A \subseteq B$  and  $\bigwedge a. a \in A \implies f a \text{ dvd } g a$ 
  shows  $\text{prod } f A \text{ dvd } \text{prod } g B$ 
proof -
  from assms have  $\text{prod } f A \text{ dvd } \text{prod } g A$ 
  by (auto intro: prod-dvd-prod)
  moreover from assms have  $\text{prod } g A \text{ dvd } \text{prod } g B$ 
  by (auto intro: prod-dvd-prod-subset)
  ultimately show ?thesis by (rule dvd-trans)
qed

```

end

```

lemma (in semidom) prod-zero-iff [simp]:
  fixes  $f :: 'b \Rightarrow 'a$ 
  assumes finite A
  shows  $\text{prod } f A = 0 \iff (\exists a \in A. f a = 0)$ 
  using assms by (induct A) (auto simp: no-zero-divisors)

```

```

lemma (in semidom-divide) prod-diff1:
  assumes finite A and  $f a \neq 0$ 
  shows  $\text{prod } f (A - \{a\}) = (\text{if } a \in A \text{ then } \text{prod } f A \text{ div } f a \text{ else } \text{prod } f A)$ 
proof (cases  $a \notin A$ )
  case True
  then show ?thesis by simp
next
  case False
  with assms show ?thesis
  proof induct
    case empty
    then show ?case by simp
  next
    case (insert b B)
    then show ?case
    proof (cases  $a = b$ )
      case True
      with insert show ?thesis by simp
    next
      case False
      with insert have  $a \in B$  by simp
    end
  end

```

```

define  $C$  where  $C = B - \{a\}$ 
with  $\langle \text{finite } B \rangle \langle a \in B \rangle$  have  $B = \text{insert } a \ C$   $\text{finite } C \ a \notin C$ 
by auto
with insert show ?thesis
by (auto simp add: insert-commute ac-simps)
qed
qed
qed

```

```

lemma sum-zero-power [simp]:  $(\sum i \in A. c \ i * 0^{\widehat{i}}) = (\text{if } \text{finite } A \wedge 0 \in A \text{ then } c \ 0 \text{ else } 0)$ 
for  $c :: \text{nat} \Rightarrow 'a :: \text{division-ring}$ 
by (induct A rule: infinite-finite-induct) auto

```

```

lemma sum-zero-power' [simp]:
 $(\sum i \in A. c \ i * 0^{\widehat{i}} / d \ i) = (\text{if } \text{finite } A \wedge 0 \in A \text{ then } c \ 0 / d \ 0 \text{ else } 0)$ 
for  $c :: \text{nat} \Rightarrow 'a :: \text{field}$ 
using sum-zero-power [of  $\lambda i. c \ i / d \ i \ A$ ] by auto

```

```

lemma (in field) prod-inversef:  $\text{prod } (\text{inverse} \circ f) \ A = \text{inverse } (\text{prod } f \ A)$ 
proof (cases finite A)
case True
then show ?thesis
by (induct A rule: finite-induct) simp-all
next
case False
then show ?thesis
by auto
qed

```

```

lemma (in field) prod-dividef:  $(\prod x \in A. f \ x / g \ x) = \text{prod } f \ A / \text{prod } g \ A$ 
using prod-inversef [of  $g \ A$ ] by (simp add: divide-inverse prod.distrib)

```

```

lemma prod-Un:
fixes  $f :: 'b \Rightarrow 'a :: \text{field}$ 
assumes finite A and finite B
and  $\forall x \in A \cap B. f \ x \neq 0$ 
shows  $\text{prod } f \ (A \cup B) = \text{prod } f \ A * \text{prod } f \ B / \text{prod } f \ (A \cap B)$ 
proof –
from assms have  $\text{prod } f \ A * \text{prod } f \ B = \text{prod } f \ (A \cup B) * \text{prod } f \ (A \cap B)$ 
by (simp add: prod.union-inter [symmetric, of A B])
with assms show ?thesis
by simp
qed

```

```

context linordered-semidom
begin

```

```

lemma prod-nonneg:  $(\forall a \in A. 0 \leq f \ a) \implies 0 \leq \text{prod } f \ A$ 

```

by (induct A rule: infinite-finite-induct) simp-all

lemma prod-pos: $(\forall a \in A. 0 < f a) \implies 0 < \text{prod } f A$
 by (induct A rule: infinite-finite-induct) simp-all

lemma prod-mono:

$(\bigwedge i. i \in A \implies 0 \leq f i \wedge f i \leq g i) \implies \text{prod } f A \leq \text{prod } g A$
 by (induct A rule: infinite-finite-induct) (force intro!: prod-nonneg mult-mono)+

lemma prod-mono-strict:

assumes finite A $\bigwedge i. i \in A \implies 0 \leq f i \wedge f i < g i$ A $\neq \{\}$
 shows $\text{prod } f A < \text{prod } g A$
 using assms

proof (induct A rule: finite-induct)

case empty

then show ?case by simp

next

case insert

then show ?case by (force intro: mult-strict-mono' prod-nonneg)

qed

lemma prod-le-power:

assumes A: $\bigwedge i. i \in A \implies 0 \leq f i \wedge f i \leq n$ card A $\leq k$ and $n \geq 1$
 shows $\text{prod } f A \leq n \wedge k$
 using A

proof (induction A arbitrary: k rule: infinite-finite-induct)

case (insert i A)

then obtain k' where $k': \text{card } A \leq k'$ $k = \text{Suc } k'$

using Suc-le-D by force

have $f i * \text{prod } f A \leq n * n \wedge k'$

using insert $\langle n \geq 1 \rangle$ k' by (intro prod-nonneg mult-mono; force)

then show ?case

by (auto simp: $\langle k = \text{Suc } k' \rangle$ insert.hyps)

qed (use $\langle n \geq 1 \rangle$ in auto)

end

lemma prod-mono2:

fixes $f :: 'a \Rightarrow 'b :: \text{linordered-idom}$

assumes fin: finite B

and sub: $A \subseteq B$

and nn: $\bigwedge b. b \in B - A \implies 1 \leq f b$

and A: $\bigwedge a. a \in A \implies 0 \leq f a$

shows $\text{prod } f A \leq \text{prod } f B$

proof –

have $\text{prod } f A \leq \text{prod } f A * \text{prod } f (B - A)$

by (metis prod-ge-1 A mult-le-cancel-left1 nn not-less prod-nonneg)

also from fin finite-subset[OF sub fin] have $\dots = \text{prod } f (A \cup (B - A))$

by (simp add: prod.union-disjoint del: Un-Diff-cancel)

also from *sub* have $A \cup (B - A) = B$ by *blast*
 finally show *?thesis* .
 qed

lemma *less-1-prod*:
 fixes $f :: 'a \Rightarrow 'b::\text{linordered-idom}$
 shows $\text{finite } I \Longrightarrow I \neq \{\} \Longrightarrow (\bigwedge i. i \in I \Longrightarrow 1 < f i) \Longrightarrow 1 < \text{prod } f I$
 by (*induct* I rule: *finite-ne-induct*) (*auto* intro: *less-1-mult*)

lemma *less-1-prod2*:
 fixes $f :: 'a \Rightarrow 'b::\text{linordered-idom}$
 assumes $I: \text{finite } I \ i \in I \ 1 < f i \ \bigwedge i. i \in I \Longrightarrow 1 \leq f i$
 shows $1 < \text{prod } f I$

proof –
 have $1 < f i * \text{prod } f (I - \{i\})$
 using *assms*
 by (*meson* *DiffD1* *leI* *less-1-mult* *less-le-trans* *mult-le-cancel-left1* *prod-ge-1*)
 also have $\dots = \text{prod } f I$
 using *assms* by (*simp* add: *prod.remove*)
 finally show *?thesis* .
 qed

lemma (*in linordered-field*) *abs-prod*: $|\text{prod } f A| = (\prod x \in A. |f x|)$
 by (*induct* A rule: *infinite-finite-induct*) (*simp-all* add: *abs-mult*)

lemma *prod-eq-1-iff* [*simp*]: $\text{finite } A \Longrightarrow \text{prod } f A = 1 \longleftrightarrow (\forall a \in A. f a = 1)$
 for $f :: 'a \Rightarrow \text{nat}$
 by (*induct* A rule: *finite-induct*) *simp-all*

lemma *prod-pos-nat-iff* [*simp*]: $\text{finite } A \Longrightarrow \text{prod } f A > 0 \longleftrightarrow (\forall a \in A. f a > 0)$
 for $f :: 'a \Rightarrow \text{nat}$
 using *prod-zero-iff* by (*simp* del: *neq0-conv* add: *zero-less-iff-neq-zero*)

lemma *prod-constant* [*simp*]: $(\prod x \in A. y) = y \wedge \text{card } A$
 for $y :: 'a::\text{comm-monoid-mult}$
 by (*induct* A rule: *infinite-finite-induct*) *simp-all*

lemma *prod-power-distrib*: $\text{prod } f A \wedge n = \text{prod } (\lambda x. (f x) \wedge n) A$
 for $f :: 'a \Rightarrow 'b::\text{comm-semiring-1}$
 by (*induct* A rule: *infinite-finite-induct*) (*auto* *simp* add: *power-mult-distrib*)

lemma *power-sum*: $c \wedge (\sum a \in A. f a) = (\prod a \in A. c \wedge f a)$
 by (*induct* A rule: *infinite-finite-induct*) (*simp-all* add: *power-add*)

lemma *prod-gen-delta*:
 fixes $b :: 'b \Rightarrow 'a::\text{comm-monoid-mult}$
 assumes *fin*: *finite* S
 shows $\text{prod } (\lambda k. \text{if } k = a \text{ then } b k \text{ else } c) S =$
 (*if* $a \in S$ then $b a * c \wedge (\text{card } S - 1)$ else $c \wedge \text{card } S$)

```

proof –
  let ?f = ( $\lambda k. \text{if } k=a \text{ then } b \ k \text{ else } c$ )
  show ?thesis
  proof (cases a ∈ S)
    case False
      then have  $\forall k \in S. ?f \ k = c$  by simp
      with False show ?thesis by (simp add: prod-constant)
    next
      case True
        let ?A = S – {a}
        let ?B = {a}
        from True have eq: S = ?A ∪ ?B by blast
        have disjoint: ?A ∩ ?B = {} by simp
        from fin have fin': finite ?A finite ?B by auto
        have f-A0: prod ?f ?A = prod ( $\lambda i. c$ ) ?A
          by (rule prod.cong) auto
        from fin True have card-A: card ?A = card S – 1 by auto
        have f-A1: prod ?f ?A = c ^ card ?A
          unfolding f-A0 by (rule prod-constant)
        have prod ?f ?A * prod ?f ?B = prod ?f S
          using prod.union-disjoint[OF fin' disjoint, of ?f, unfolded eq[symmetric]]
          by simp
        with True card-A show ?thesis
          by (simp add: f-A1 field-simps cong add: prod.cong cong del: if-weak-cong)
      qed
    qed

lemma sum-image-le:
  fixes g :: 'a ⇒ 'b::ordered-comm-monoid-add
  assumes finite I  $\wedge i. i \in I \implies 0 \leq g(f \ i)$ 
  shows sum g (f ' I) ≤ sum (g ∘ f) I
  using assms
proof induction
  case empty
  then show ?case by auto
next
  case (insert x F)
  from insertI1 have 0 ≤ g (f x) by (rule insert)
  hence 1: sum g (f ' F) ≤ g (f x) + sum g (f ' F) using add-increasing by blast
  have 2: sum g (f ' F) ≤ sum (g ∘ f) F using insert by blast
  have sum g (f ' insert x F) = sum g (insert (f x) (f ' F)) by simp
  also have ... ≤ g (f x) + sum g (f ' F) by (simp add: 1 insert sum.insert-if)
  also from 2 have ... ≤ g (f x) + sum (g ∘ f) F by (rule add-left-mono)
  also from insert(1, 2) have ... = sum (g ∘ f) (insert x F) by (simp add:
  sum.insert-if)
  finally show ?case .
qed

end

```

49 Chain-complete partial orders and their fix-points

```
theory Complete-Partial-Order
  imports Product-Type
begin
```

49.1 Chains

A chain is a totally-ordered set. Chains are parameterized over the order for maximal flexibility, since type classes are not enough.

```
definition chain :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool
  where chain ord S ↔ (∀ x∈S. ∀ y∈S. ord x y ∨ ord y x)
```

```
lemma chainI:
  assumes ∧x y. x ∈ S ⇒ y ∈ S ⇒ ord x y ∨ ord y x
  shows chain ord S
  using assms unfolding chain-def by fast
```

```
lemma chainD:
  assumes chain ord S and x ∈ S and y ∈ S
  shows ord x y ∨ ord y x
  using assms unfolding chain-def by fast
```

```
lemma chainE:
  assumes chain ord S and x ∈ S and y ∈ S
  obtains ord x y | ord y x
  using assms unfolding chain-def by fast
```

```
lemma chain-empty: chain ord {}
  by (simp add: chain-def)
```

```
lemma chain-equality: chain (=) A ↔ (∀ x∈A. ∀ y∈A. x = y)
  by (auto simp add: chain-def)
```

```
lemma chain-subset: chain ord A ⇒ B ⊆ A ⇒ chain ord B
  by (rule chainI) (blast dest: chainD)
```

```
lemma chain-imageI:
  assumes chain: chain le-a Y
  and mono: ∧x y. x ∈ Y ⇒ y ∈ Y ⇒ le-a x y ⇒ le-b (f x) (f y)
  shows chain le-b (f ` Y)
  by (blast intro: chainI dest: chainD[OF chain] mono)
```

49.2 Chain-complete partial orders

A *ccpo* has a least upper bound for any chain. In particular, the empty set is a chain, so every *ccpo* must have a bottom element.

```

class ccpo = order + Sup +
  assumes ccpo-Sup-upper: chain ( $\leq$ ) A  $\implies x \in A \implies x \leq \text{Sup } A$ 
  assumes ccpo-Sup-least: chain ( $\leq$ ) A  $\implies (\bigwedge x. x \in A \implies x \leq z) \implies \text{Sup } A \leq z$ 
begin

lemma chain-singleton: Complete-Partial-Order.chain ( $\leq$ ) {x}
  by (rule chainI) simp

lemma ccpo-Sup-singleton [simp]:  $\bigsqcup \{x\} = x$ 
  by (rule order.antisym) (auto intro: ccpo-Sup-least ccpo-Sup-upper simp add: chain-singleton)

```

49.3 Transfinite iteration of a function

```

context notes [[inductive-internals]]
begin

```

```

inductive-set iterates :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a set
  for f :: 'a  $\Rightarrow$  'a
  where
    step:  $x \in \text{iterates } f \implies f x \in \text{iterates } f$ 
  | Sup: chain ( $\leq$ ) M  $\implies \forall x \in M. x \in \text{iterates } f \implies \text{Sup } M \in \text{iterates } f$ 

```

```
end
```

```

lemma iterates-le-f:  $x \in \text{iterates } f \implies \text{monotone } (\leq) (\leq) f \implies x \leq f x$ 
  by (induct x rule: iterates.induct)
  (force dest: monotoneD intro!: ccpo-Sup-upper ccpo-Sup-least)+

```

```

lemma chain-iterates:
  assumes f: monotone ( $\leq$ ) ( $\leq$ ) f
  shows chain ( $\leq$ ) (iterates f) (is chain - ?C)
proof (rule chainI)
  fix x y
  assume  $x \in ?C \ y \in ?C$ 
  then show  $x \leq y \vee y \leq x$ 
proof (induct x arbitrary: y rule: iterates.induct)
  fix x y
  assume y:  $y \in ?C$ 
  and IH:  $\bigwedge z. z \in ?C \implies x \leq z \vee z \leq x$ 
  from y show  $f x \leq y \vee y \leq f x$ 
proof (induct y rule: iterates.induct)
  case (step y)
  with IH f show ?case by (auto dest: monotoneD)
next
  case (Sup M)
  then have chM: chain ( $\leq$ ) M
  and IH':  $\bigwedge z. z \in M \implies f x \leq z \vee z \leq f x$  by auto

```

```

show  $f x \leq \text{Sup } M \vee \text{Sup } M \leq f x$ 
proof (cases  $\exists z \in M. f x \leq z$ )
  case True
  then have  $f x \leq \text{Sup } M$ 
    by (blast intro: ccpo-Sup-upper[OF chM] order-trans)
  then show ?thesis ..
next
  case False
  with IH' show ?thesis
    by (auto intro: ccpo-Sup-least[OF chM])
  qed
qed
next
  case ( $\text{Sup } M y$ )
  show ?case
  proof (cases  $\exists x \in M. y \leq x$ )
    case True
    then have  $y \leq \text{Sup } M$ 
      by (blast intro: ccpo-Sup-upper[OF Sup(1)] order-trans)
    then show ?thesis ..
  next
    case False with Sup
    show ?thesis by (auto intro: ccpo-Sup-least)
  qed
qed
qed

```

lemma *bot-in-iterates*: $\text{Sup } \{\} \in \text{iterates } f$
by (auto intro: iterates.Sup simp add: chain-empty)

49.4 Fixpoint combinator

definition *fixp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where $\text{fixp } f = \text{Sup } (\text{iterates } f)$

lemma *iterates-fixp*:
assumes f : *monotone* $(\leq) (\leq) f$
shows $\text{fixp } f \in \text{iterates } f$
unfolding *fixp-def*
by (*simp add: iterates.Sup chain-iterates f*)

lemma *fixp-unfold*:
assumes f : *monotone* $(\leq) (\leq) f$
shows $\text{fixp } f = f (\text{fixp } f)$
proof (*rule order.antisym*)
show $\text{fixp } f \leq f (\text{fixp } f)$
by (*intro iterates-le-f iterates-fixp f*)
have $f (\text{fixp } f) \leq \text{Sup } (\text{iterates } f)$
by (*intro ccpo-Sup-upper chain-iterates f iterates.step iterates-fixp*)


```

then show  $f \text{ (fixp } f) \leq \text{fixp } f$ 
  by (simp only: fixp-def)
qed

lemma fixp-lowerbound:
  assumes  $f$ : monotone  $(\leq) (\leq) f$ 
    and  $z$ :  $f z \leq z$ 
  shows  $\text{fixp } f \leq z$ 
  unfolding fixp-def
proof (rule ccpo-Sup-least[OF chain-iterates[OF f]])
  fix  $x$ 
  assume  $x \in \text{iterates } f$ 
  then show  $x \leq z$ 
  proof (induct x rule: iterates.induct)
    case (step x)
    from  $f \langle x \leq z \rangle$  have  $f x \leq f z$  by (rule monotoneD)
    also note  $z$ 
    finally show  $f x \leq z$  .
  next
    case (Sup M)
    then show ?case
      by (auto intro: ccpo-Sup-least)
  qed
qed

end

```

49.5 Fixpoint induction

setup $\langle \text{Sign.map-naming (Name-Space.mandatory-path ccpo)} \rangle$

definition *admissible* :: $('a \text{ set} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *admissible lub ord P* $\longleftrightarrow (\forall A. \text{chain ord } A \longrightarrow A \neq \{\} \longrightarrow (\forall x \in A. P x) \longrightarrow P (\text{lub } A))$

lemma *admissibleI*:
assumes $\bigwedge A. \text{chain ord } A \Longrightarrow A \neq \{\} \Longrightarrow \forall x \in A. P x \Longrightarrow P (\text{lub } A)$
shows *ccpo.admissible lub ord P*
using *assms* **unfolding** *ccpo.admissible-def* **by** *fast*

lemma *admissibleD*:
assumes *ccpo.admissible lub ord P*
assumes *chain ord A*
assumes $A \neq \{\}$
assumes $\bigwedge x. x \in A \Longrightarrow P x$
shows $P (\text{lub } A)$
using *assms* **by** (*auto simp: ccpo.admissible-def*)

setup $\langle \text{Sign.map-naming Name-Space.parent-path} \rangle$

```

lemma (in ccpo) fixp-induct:
  assumes adm: ccpo.admissible Sup ( $\leq$ ) P
  assumes mono: monotone ( $\leq$ ) ( $\leq$ ) f
  assumes bot: P (Sup {})
  assumes step:  $\bigwedge x. P\ x \implies P\ (f\ x)$ 
  shows P (fixp f)
  unfolding fixp-def
  using adm chain-iterates[OF mono]
proof (rule ccpo.admissibleD)
  show iterates f  $\neq$  {}
    using bot-in-iterates by auto
next
  fix x
  assume  $x \in \textit{iterates}\ f$ 
  then show P x
  proof (induct rule: iterates.induct)
    case prems: (step x)
    from this(2) show ?case by (rule step)
  next
  case (Sup M)
  then show ?case by (cases  $M = \{\}$ ) (auto intro: step bot ccpo.admissibleD
adm)
  qed
qed

```

```

lemma admissible-True: ccpo.admissible lub ord ( $\lambda x. \textit{True}$ )
  unfolding ccpo.admissible-def by simp

```

```

lemma admissible-const: ccpo.admissible lub ord ( $\lambda x. t$ )
  by (auto intro: ccpo.admissibleI)

```

```

lemma admissible-conj:
  assumes ccpo.admissible lub ord ( $\lambda x. P\ x$ )
  assumes ccpo.admissible lub ord ( $\lambda x. Q\ x$ )
  shows ccpo.admissible lub ord ( $\lambda x. P\ x \wedge Q\ x$ )
  using assms unfolding ccpo.admissible-def by simp

```

```

lemma admissible-all:
  assumes  $\bigwedge y. \textit{ccpo.admissible}\ \textit{lub}\ \textit{ord}\ (\lambda x. P\ x\ y)$ 
  shows ccpo.admissible lub ord ( $\lambda x. \forall y. P\ x\ y$ )
  using assms unfolding ccpo.admissible-def by fast

```

```

lemma admissible-ball:
  assumes  $\bigwedge y. y \in A \implies \textit{ccpo.admissible}\ \textit{lub}\ \textit{ord}\ (\lambda x. P\ x\ y)$ 
  shows ccpo.admissible lub ord ( $\lambda x. \forall y \in A. P\ x\ y$ )
  using assms unfolding ccpo.admissible-def by fast

```

lemma *chain-compr*: $chain\ ord\ A \implies chain\ ord\ \{x \in A.\ P\ x\}$
unfolding *chain-def* **by** *fast*

context *ccpo*
begin

lemma *admissible-disj*:

fixes $P\ Q :: 'a \Rightarrow bool$

assumes P : *ccpo.admissible Sup* (\leq) $(\lambda x.\ P\ x)$

assumes Q : *ccpo.admissible Sup* (\leq) $(\lambda x.\ Q\ x)$

shows *ccpo.admissible Sup* (\leq) $(\lambda x.\ P\ x \vee Q\ x)$

proof (*rule ccpo.admissibleI*)

fix $A :: 'a\ set$

assume *chain*: *chain* $(\leq)\ A$

assume A : $A \neq \{\}$ **and** P - Q : $\forall x \in A.\ P\ x \vee Q\ x$

have $(\exists x \in A.\ P\ x) \wedge (\forall x \in A.\ \exists y \in A.\ x \leq y \wedge P\ y) \vee (\exists x \in A.\ Q\ x) \wedge (\forall x \in A.\ \exists y \in A.\ x \leq y \wedge Q\ y)$

(**is** $?P \vee ?Q$ **is** $?P1 \wedge ?P2 \vee -$)

proof (*rule disjCI*)

assume $\neg ?Q$

then consider $\forall x \in A.\ \neg Q\ x \mid a$ **where** $a \in A \forall y \in A.\ a \leq y \longrightarrow \neg Q\ y$

by *blast*

then show $?P$

proof *cases*

case 1

with P - Q **have** $\forall x \in A.\ P\ x$ **by** *blast*

with A **show** $?P$ **by** *blast*

next

case 2

note $a = \langle a \in A \rangle$

show $?P$

proof

from P - Q 2 **have** $*$: $\forall y \in A.\ a \leq y \longrightarrow P\ y$ **by** *blast*

with a **have** $P\ a$ **by** *blast*

with a **show** $?P1$ **by** *blast*

show $?P2$

proof

fix x

assume x : $x \in A$

with *chain* a **show** $\exists y \in A.\ x \leq y \wedge P\ y$

proof (*rule chainE*)

assume le : $a \leq x$

with $*$ $a\ x$ **have** $P\ x$ **by** *blast*

with $x\ le$ **show** $?thesis$ **by** *blast*

next

assume $a \geq x$

with $a \langle P\ a \rangle$ **show** $?thesis$ **by** *blast*

qed

qed

```

    qed
  qed
  moreover
  have  $Sup A = Sup \{x \in A. P x\}$  if  $\bigwedge x. x \in A \implies \exists y \in A. x \leq y \wedge P y$  for  $P$ 
  proof (rule order.antisym)
    have chain- $P$ : chain ( $\leq$ )  $\{x \in A. P x\}$ 
    by (rule chain-compr [OF chain])
    show  $Sup A \leq Sup \{x \in A. P x\}$ 
    proof (rule ccpo-Sup-least [OF chain])
      show  $\bigwedge x. x \in A \implies x \leq \bigsqcup \{x \in A. P x\}$ 
      by (blast intro: ccpo-Sup-upper [OF chain- $P$ ] order-trans dest: that)
    qed
    show  $Sup \{x \in A. P x\} \leq Sup A$ 
    apply (rule ccpo-Sup-least [OF chain- $P$ ])
    apply (simp add: ccpo-Sup-upper [OF chain])
    done
  qed
  ultimately
  consider  $\exists x. x \in A \wedge P x$   $Sup A = Sup \{x \in A. P x\}$ 
  |  $\exists x. x \in A \wedge Q x$   $Sup A = Sup \{x \in A. Q x\}$ 
  by blast
  then show  $P (Sup A) \vee Q (Sup A)$ 
  proof cases
    case 1
    then show ?thesis
    using ccpo.admissibleD [OF P chain-compr [OF chain]] by force
  next
    case 2
    then show ?thesis
    using ccpo.admissibleD [OF Q chain-compr [OF chain]] by force
  qed
  qed
end

instance complete-lattice  $\subseteq$  ccpo
  by standard (fast intro: Sup-upper Sup-least)+

lemma lfp-eq-fixp:
  assumes mono: mono  $f$ 
  shows lfp  $f =$  fixp  $f$ 
  proof (rule order.antisym)
    from mono have  $f'$ : monotone ( $\leq$ ) ( $\leq$ )  $f$ 
    unfolding mono-def monotone-def .
    show lfp  $f \leq$  fixp  $f$ 
    by (rule lfp-lowerbound, subst fixp-unfold [OF  $f'$ ], rule order-refl)
    show fixp  $f \leq$  lfp  $f$ 
    by (rule fixp-lowerbound [OF  $f'$ ] (simp add: lfp-fixpoint [OF mono]))
  qed

```

qed

hide-const (open) iterates fixp

end

50 Datatype option

theory Option
 imports Lifting
 begin

datatype 'a option =
 None
 | Some (the: 'a)

datatype-compact option

lemma [case-names None Some, cases type: option]:
 — for backward compatibility – names of variables differ
 $(y = \text{None} \implies P) \implies (\bigwedge a. y = \text{Some } a \implies P) \implies P$
 by (rule option.exhaust)

lemma [case-names None Some, induct type: option]:
 — for backward compatibility – names of variables differ
 $P \text{ None} \implies (\bigwedge \text{option}. P (\text{Some } \text{option})) \implies P \text{ option}$
 by (rule option.induct)

Compatibility:

setup <Sign.mandatory-path option>
 lemmas inducts = option.induct
 lemmas cases = option.case
 setup <Sign.parent-path>

lemma not-None-eq [iff]: $x \neq \text{None} \longleftrightarrow (\exists y. x = \text{Some } y)$
 by (induct x) auto

lemma not-Some-eq [iff]: $(\forall y. x \neq \text{Some } y) \longleftrightarrow x = \text{None}$
 by (induct x) auto

lemma comp-the-Some[simp]: $\text{the } o \text{ Some} = \text{id}$
 by auto

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

lemma inj-Some [simp]: $\text{inj-on } \text{Some } A$
 by (rule inj-onI) simp

lemma *case-optionE*:

assumes c : (*case* x of *None* $\Rightarrow P$ | *Some* $y \Rightarrow Q y$)
obtains
 (*None*) $x = \text{None}$ **and** P
 | (*Some*) y **where** $x = \text{Some } y$ **and** $Q y$
using c **by** (*cases* x) *simp-all*

lemma *split-option-all*: $(\forall x. P x) \longleftrightarrow P \text{ None} \wedge (\forall x. P (\text{Some } x))$
by (*auto intro: option.induct*)

lemma *split-option-ex*: $(\exists x. P x) \longleftrightarrow P \text{ None} \vee (\exists x. P (\text{Some } x))$
using *split-option-all*[of $\lambda x. \neg P x$] **by** *blast*

lemma *UNIV-option-conv*: $\text{UNIV} = \text{insert None } (\text{range } \text{Some})$
by (*auto intro: classical*)

lemma *rel-option-None1* [*simp*]: $\text{rel-option } P \text{ None } x \longleftrightarrow x = \text{None}$
by (*cases* x) *simp-all*

lemma *rel-option-None2* [*simp*]: $\text{rel-option } P x \text{ None} \longleftrightarrow x = \text{None}$
by (*cases* x) *simp-all*

lemma *option-rel-Some1*: $\text{rel-option } A (\text{Some } x) y \longleftrightarrow (\exists y'. y = \text{Some } y' \wedge A x y')$
by(*cases* y) *simp-all*

lemma *option-rel-Some2*: $\text{rel-option } A x (\text{Some } y) \longleftrightarrow (\exists x'. x = \text{Some } x' \wedge A x' y)$
by(*cases* x) *simp-all*

lemma *rel-option-inf*: $\text{inf } (\text{rel-option } A) (\text{rel-option } B) = \text{rel-option } (\text{inf } A B)$
 (*is ?lhs = ?rhs*)

proof (*rule antisym*)

show $?lhs \leq ?rhs$ **by** (*auto elim: option.rel-cases*)

show $?rhs \leq ?lhs$ **by** (*auto elim: option.rel-mono-strong*)

qed

lemma *rel-option-refl*:

$(\bigwedge x. x \in \text{set-option } y \Longrightarrow P x x) \Longrightarrow \text{rel-option } P y y$
by (*cases* y) *auto*

50.0.1 Operations

lemma *ospec* [*dest*]: $(\forall x \in \text{set-option } A. P x) \Longrightarrow A = \text{Some } x \Longrightarrow P x$
by *simp*

setup $\langle \text{map-theory-claset } (\text{fn } \text{ctxt} \Rightarrow \text{ctxt addSD2 } (\text{ospec}, @\{\text{thm ospec}\})) \rangle$

lemma *elem-set* [iff]: $(x \in \text{set-option } xo) = (xo = \text{Some } x)$
by (cases xo) auto

lemma *set-empty-eq* [simp]: $(\text{set-option } xo = \{\}) = (xo = \text{None})$
by (cases xo) auto

lemma *map-option-case*: $\text{map-option } f y = (\text{case } y \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow \text{Some } (f x))$
by (auto split: option.split)

lemma *map-option-is-None* [iff]: $(\text{map-option } f \text{ opt} = \text{None}) = (\text{opt} = \text{None})$
by (simp add: map-option-case split: option.split)

lemma *None-eq-map-option-iff* [iff]: $\text{None} = \text{map-option } f x \longleftrightarrow x = \text{None}$
by(cases x) simp-all

lemma *map-option-eq-Some* [iff]: $(\text{map-option } f xo = \text{Some } y) = (\exists z. xo = \text{Some } z \wedge f z = y)$
by (simp add: map-option-case split: option.split)

lemma *map-option-o-case-sum* [simp]:
 $\text{map-option } f \circ \text{case-sum } g h = \text{case-sum } (\text{map-option } f \circ g) (\text{map-option } f \circ h)$
by (rule o-case-sum)

lemma *map-option-cong*: $x = y \implies (\bigwedge a. y = \text{Some } a \implies f a = g a) \implies \text{map-option } f x = \text{map-option } g y$
by (cases x) auto

lemma *map-option-idI*: $(\bigwedge y. y \in \text{set-option } x \implies f y = y) \implies \text{map-option } f x = x$
by(cases x)(simp-all)

functor *map-option*: *map-option*
by (simp-all add: option.map-comp fun-eq-iff option.map-id)

lemma *case-map-option* [simp]: $\text{case-option } g h (\text{map-option } f x) = \text{case-option } g (h \circ f) x$
by (cases x) simp-all

lemma *None-notin-image-Some* [simp]: $\text{None} \notin \text{Some } ` A$
by auto

lemma *notin-range-Some*: $x \notin \text{range } \text{Some} \longleftrightarrow x = \text{None}$
by(cases x) auto

lemma *rel-option-iff*:
 $\text{rel-option } R x y = (\text{case } (x, y) \text{ of } (\text{None}, \text{None}) \Rightarrow \text{True} \mid (\text{Some } x, \text{Some } y) \Rightarrow R x y)$

| - \Rightarrow *False*)
by (*auto split: prod.split option.split*)

definition *combine-options* :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a option \Rightarrow 'a option \Rightarrow 'a option
where *combine-options* f x y =
 (case x of None \Rightarrow y | Some x \Rightarrow (case y of None \Rightarrow Some x | Some y \Rightarrow Some (f x y)))

lemma *combine-options-simps* [*simp*]:
combine-options f None y = y
combine-options f x None = x
combine-options f (Some a) (Some b) = Some (f a b)
by (*simp-all add: combine-options-def split: option.splits*)

lemma *combine-options-cases* [*case-names None1 None2 Some*]:
 (x = None \Longrightarrow P x y) \Longrightarrow (y = None \Longrightarrow P x y) \Longrightarrow
 (\wedge a b. x = Some a \Longrightarrow y = Some b \Longrightarrow P x y) \Longrightarrow P x y
by (*cases x; cases y simp-all*)

lemma *combine-options-commute*:
 (\wedge x y. f x y = f y x) \Longrightarrow *combine-options* f x y = *combine-options* f y x
using *combine-options-cases*[*of x*]
by (*induction x y rule: combine-options-cases*) *simp-all*

lemma *combine-options-assoc*:
 (\wedge x y z. f (f x y) z = f x (f y z)) \Longrightarrow
combine-options f (combine-options f x y) z =
combine-options f x (combine-options f y z)
by (*auto simp: combine-options-def split: option.splits*)

lemma *combine-options-left-commute*:
 (\wedge x y. f x y = f y x) \Longrightarrow (\wedge x y z. f (f x y) z = f x (f y z)) \Longrightarrow
combine-options f y (combine-options f x z) =
combine-options f x (combine-options f y z)
by (*auto simp: combine-options-def split: option.splits*)

lemmas *combine-options-ac* =
combine-options-commute combine-options-assoc combine-options-left-commute

context
begin

qualified definition *is-none* :: 'a option \Rightarrow bool
where [*code-post*]: *is-none* x \longleftrightarrow x = None

lemma *is-none-simps* [*simp*]:
is-none None

\neg *is-none* (*Some* *x*)
by (*simp-all add: is-none-def*)

lemma *is-none-code* [*code*]:
is-none *None* = *True*
is-none (*Some* *x*) = *False*
by *simp-all*

lemma *rel-option-unfold*:
rel-option *R* *x* *y* \longleftrightarrow
(*is-none* *x* \longleftrightarrow *is-none* *y*) \wedge (\neg *is-none* *x* \longrightarrow \neg *is-none* *y* \longrightarrow *R* (*the* *x*) (*the* *y*))
by (*simp add: rel-option-iff split: option.split*)

lemma *rel-optionI*:
 \llbracket *is-none* *x* \longleftrightarrow *is-none* *y*; \llbracket \neg *is-none* *x*; \neg *is-none* *y* $\rrbracket \Longrightarrow$ *P* (*the* *x*) (*the* *y*) \rrbracket
 \Longrightarrow *rel-option* *P* *x* *y*
by (*simp add: rel-option-unfold*)

lemma *is-none-map-option* [*simp*]: *is-none* (*map-option* *f* *x*) \longleftrightarrow *is-none* *x*
by (*simp add: is-none-def*)

lemma *the-map-option*: \neg *is-none* *x* \Longrightarrow *the* (*map-option* *f* *x*) = *f* (*the* *x*)
by (*auto simp add: is-none-def*)

qualified primrec *bind* :: '*a* option \Rightarrow ('*a* \Rightarrow '*b* option) \Rightarrow '*b* option
where

bind-lzero: *bind* *None* *f* = *None*
| *bind-lunit*: *bind* (*Some* *x*) *f* = *f* *x*

lemma *is-none-bind*: *is-none* (*bind* *f* *g*) \longleftrightarrow *is-none* *f* \vee *is-none* (*g* (*the* *f*))
by (*cases f*) *simp-all*

lemma *bind-runit*[*simp*]: *bind* *x* *Some* = *x*
by (*cases x*) *auto*

lemma *bind-assoc*[*simp*]: *bind* (*bind* *x* *f*) *g* = *bind* *x* ($\lambda y.$ *bind* (*f* *y*) *g*)
by (*cases x*) *auto*

lemma *bind-rzero*[*simp*]: *bind* *x* ($\lambda x.$ *None*) = *None*
by (*cases x*) *auto*

qualified lemma *bind-cong*: *x* = *y* \Longrightarrow ($\bigwedge a.$ *y* = *Some* *a* \Longrightarrow *f* *a* = *g* *a*) \Longrightarrow *bind* *x* *f* = *bind* *y* *g*
by (*cases x*) *auto*

lemma *bind-split*: *P* (*bind* *m* *f*) \longleftrightarrow (*m* = *None* \longrightarrow *P* *None*) \wedge ($\forall v.$ *m* = *Some* *v* \longrightarrow *P* (*f* *v*))

by (*cases m*) *auto*

lemma *bind-split-asm*: $P (\text{bind } m \ f) \longleftrightarrow \neg (m = \text{None} \wedge \neg P \ \text{None} \vee (\exists x. m = \text{Some } x \wedge \neg P (f \ x)))$

by (*cases m*) *auto*

lemmas *bind-splits* = *bind-split bind-split-asm*

lemma *bind-eq-Some-conv*: $\text{bind } f \ g = \text{Some } x \longleftrightarrow (\exists y. f = \text{Some } y \wedge g \ y = \text{Some } x)$

by (*cases f*) *simp-all*

lemma *bind-eq-None-conv*: $\text{Option.bind } a \ f = \text{None} \longleftrightarrow a = \text{None} \vee f (\text{the } a) = \text{None}$

by(*cases a*) *simp-all*

lemma *map-option-bind*: $\text{map-option } f (\text{bind } x \ g) = \text{bind } x (\text{map-option } f \circ g)$

by (*cases x*) *simp-all*

lemma *bind-option-cong*:

$\llbracket x = y; \bigwedge z. z \in \text{set-option } y \implies f \ z = g \ z \rrbracket \implies \text{bind } x \ f = \text{bind } y \ g$

by (*cases y*) *simp-all*

lemma *bind-option-cong-simp*:

$\llbracket x = y; \bigwedge z. z \in \text{set-option } y = \text{simp} \implies f \ z = g \ z \rrbracket \implies \text{bind } x \ f = \text{bind } y \ g$

unfolding *simp-implies-def* **by** (*rule bind-option-cong*)

lemma *bind-option-cong-code*: $x = y \implies \text{bind } x \ f = \text{bind } y \ f$

by *simp*

lemma *bind-map-option*: $\text{bind } (\text{map-option } f \ x) \ g = \text{bind } x (g \circ f)$

by(*cases x*) *simp-all*

lemma *set-bind-option* [*simp*]: $\text{set-option } (\text{bind } x \ f) = (\bigcup ((\text{set-option} \circ f) \ ` \ \text{set-option } x))$

by(*cases x*) *auto*

lemma *map-conv-bind-option*: $\text{map-option } f \ x = \text{Option.bind } x (\text{Some} \circ f)$

by(*cases x*) *simp-all*

end

setup $\langle \text{Code-Simp.map-ss } (\text{Simplifier.add-cong } @\{\text{thm bind-option-cong-code}\}) \rangle$

context

begin

qualified definition *these* :: $'a \ \text{option set} \Rightarrow 'a \ \text{set}$

```

where these A = the ‘ {x ∈ A. x ≠ None}

lemma these-empty [simp]: these {} = {}
  by (simp add: these-def)

lemma these-insert-None [simp]: these (insert None A) = these A
  by (auto simp add: these-def)

lemma these-insert-Some [simp]: these (insert (Some x) A) = insert x (these A)
proof –
  have {y ∈ insert (Some x) A. y ≠ None} = insert (Some x) {y ∈ A. y ≠ None}
    by auto
  then show ?thesis by (simp add: these-def)
qed

lemma in-these-eq: x ∈ these A ↔ Some x ∈ A
proof
  assume Some x ∈ A
  then obtain B where A = insert (Some x) B by auto
  then show x ∈ these A by (auto simp add: these-def intro!: image-eqI)
next
  assume x ∈ these A
  then show Some x ∈ A by (auto simp add: these-def)
qed

lemma these-image-Some-eq [simp]: these (Some ‘ A) = A
  by (auto simp add: these-def intro!: image-eqI)

lemma Some-image-these-eq: Some ‘ these A = {x ∈ A. x ≠ None}
  by (auto simp add: these-def image-image intro!: image-eqI)

lemma these-empty-eq: these B = {} ↔ B = {} ∨ B = {None}
  by (auto simp add: these-def)

lemma these-not-empty-eq: these B ≠ {} ↔ B ≠ {} ∧ B ≠ {None}
  by (auto simp add: these-empty-eq)

end

lemma finite-range-Some: finite (range (Some :: ‘a ⇒ ‘a option)) = finite (UNIV
  :: ‘a set)
  by (auto dest: finite-imageD intro: inj-Some)

```

50.1 Transfer rules for the Transfer package

```

context includes lifting-syntax
begin

```

```

lemma option-bind-transfer [transfer-rule]:

```

```
(rel-option A ===> (A ===> rel-option B) ===> rel-option B)
  Option.bind Option.bind
unfolding rel-fun-def split-option-all by simp
```

```
lemma pred-option-parametric [transfer-rule]:
  ((A ===> (=)) ===> rel-option A ===> (=)) pred-option pred-option
  by (rule rel-funI)+ (auto simp add: rel-option-unfold Option.is-none-def dest:
rel-funD)
```

end

50.1.1 Interaction with finite sets

```
lemma finite-option-UNIV [simp]:
  finite (UNIV :: 'a option set) = finite (UNIV :: 'a set)
  by (auto simp add: UNIV-option-conv elim: finite-imageD intro: inj-Some)
```

```
instance option :: (finite) finite
  by standard (simp add: UNIV-option-conv)
```

50.1.2 Code generator setup

```
lemma equal-None-code-unfold [code-unfold]:
  HOL.equal x None  $\longleftrightarrow$  Option.is-none x
  HOL.equal None = Option.is-none
  by (auto simp add: equal Option.is-none-def)
```

code-printing

```
type-constructor option  $\rightarrow$ 
  (SML) - option
  and (OCaml) - option
  and (Haskell) Maybe -
  and (Scala) !Option[(-)]
| constant None  $\rightarrow$ 
  (SML) NONE
  and (OCaml) None
  and (Haskell) Nothing
  and (Scala) !None
| constant Some  $\rightarrow$ 
  (SML) SOME
  and (OCaml) Some -
  and (Haskell) Just
  and (Scala) Some
| class-instance option :: equal  $\rightarrow$ 
  (Haskell) -
| constant HOL.equal :: 'a option  $\Rightarrow$  'a option  $\Rightarrow$  bool  $\rightarrow$ 
  (Haskell) infix 4 ==
```

```
code-reserved SML
  option NONE SOME
```

code-reserved *OCaml*
option None Some

code-reserved *Scala*
Option None Some

end

51 Partial Function Definitions

theory *Partial-Function*

imports *Complete-Partial-Order Option*

keywords *partial-function :: thy-defn*

begin

named-theorems *partial-function-mono monotonicity rules for partial function definitions*

ML-file *⟨Tools/Function/partial-function.ML⟩*

lemma (in *ccpo*) *in-chain-finite*:

assumes *Complete-Partial-Order.chain* (\leq) *A finite A A* \neq $\{\}$

shows $\bigsqcup A \in A$

using *assms(2,1,3)*

proof *induction*

case empty thus *?case by simp*

next

case (*insert x A*)

note *chain* = *⟨Complete-Partial-Order.chain* (\leq) (*insert x A*) \rangle

show *?case*

proof(*cases A = {}*)

case True thus *?thesis by simp*

next

case *False*

from *chain* **have** *chain'*: *Complete-Partial-Order.chain* (\leq) *A*

by(*rule chain-subset*) *blast*

hence $\bigsqcup A \in A$ **using** *False* **by**(*rule insert.IH*)

show *?thesis*

proof(*cases x* \leq $\bigsqcup A$)

case *True*

have $\bigsqcup(\textit{insert } x A) \leq \bigsqcup A$ **using** *chain*

by(*rule ccpo-Sup-least*)(*auto simp add: True intro: ccpo-Sup-upper*[*OF*

chain'])

hence $\bigsqcup(\textit{insert } x A) = \bigsqcup A$

by(*rule order.antisym*)(*blast intro: ccpo-Sup-upper*[*OF chain'*] *ccpo-Sup-least*[*OF*

chain'])

with $\langle \bigsqcup A \in A \rangle$ **show** *?thesis by simp*

next

case *False*

```

with chainD[OF chain, of x  $\sqcup$  A]  $\langle \sqcup A \in A \rangle$ 
have  $\sqcup$  (insert x A) = x
      by (auto intro: order.antisym ccpo-Sup-least[OF chain] order-trans[OF
ccpo-Sup-upper[OF chain']] ccpo-Sup-upper[OF chain])
      thus ?thesis by simp
qed
qed
qed

```

```

lemma (in ccpo) admissible-chfin:
  ( $\forall S$ . Complete-Partial-Order.chain  $(\leq)$  S  $\longrightarrow$  finite S)
   $\implies$  ccpo.admissible Sup  $(\leq)$  P
using in-chain-finite by (blast intro: ccpo.admissibleI)

```

51.1 Axiomatic setup

This technical locale contains the requirements for function definitions with ccpo fixed points.

definition fun-ord ord f g \longleftrightarrow ($\forall x$. ord (f x) (g x))

definition fun-lub L A = (λx . L {y. $\exists f \in A$. y = f x})

definition img-ord f ord = ($\lambda x y$. ord (f x) (f y))

definition img-lub f g Lub = (λA . g (Lub (f ‘ A)))

lemma chain-fun:

assumes A: chain (fun-ord ord) A

shows chain ord {y. $\exists f \in A$. y = f a} (is chain ord ?C)

proof (rule chainI)

fix x y **assume** x \in ?C y \in ?C

then obtain f g **where** fg: f \in A g \in A

and [simp]: x = f a y = g a **by** blast

from chainD[OF A fg]

show ord x y \vee ord y x **unfolding** fun-ord-def **by** auto

qed

lemma call-mono[partial-function-mono]: monotone (fun-ord ord) ord (λf . f t)
by (rule monotoneI) (auto simp: fun-ord-def)

lemma let-mono[partial-function-mono]:

($\wedge x$. monotone orda ordb (λf . b f x))

\implies monotone orda ordb (λf . Let t (b f))

by (simp add: Let-def)

lemma if-mono[partial-function-mono]: monotone orda ordb F

\implies monotone orda ordb G

\implies monotone orda ordb (λf . if c then F f else G f)

unfolding monotone-def **by** simp

definition mk-less R = ($\lambda x y$. R x y \wedge \neg R y x)

```

locale partial-function-definitions =
  fixes leq :: 'a ⇒ 'a ⇒ bool
  fixes lub :: 'a set ⇒ 'a
  assumes leq-refl: leq x x
  assumes leq-trans: leq x y ⇒ leq y z ⇒ leq x z
  assumes leq-antisym: leq x y ⇒ leq y x ⇒ x = y
  assumes lub-upper: chain leq A ⇒ x ∈ A ⇒ leq x (lub A)
  assumes lub-least: chain leq A ⇒ (∧x. x ∈ A ⇒ leq x z) ⇒ leq (lub A) z

lemma partial-function-lift:
  assumes partial-function-definitions ord lb
  shows partial-function-definitions (fun-ord ord) (fun-lub lb) (is partial-function-definitions
    ?ordf ?lubf)
proof –
  interpret partial-function-definitions ord lb by fact

  show ?thesis
proof
  fix x show ?ordf x x
    unfolding fun-ord-def by (auto simp: leq-refl)
next
  fix x y z assume ?ordf x y ?ordf y z
  thus ?ordf x z unfolding fun-ord-def
    by (force dest: leq-trans)
next
  fix x y assume ?ordf x y ?ordf y x
  thus x = y unfolding fun-ord-def
    by (force intro!: dest: leq-antisym)
next
  fix A f assume f: f ∈ A and A: chain ?ordf A
  thus ?ordf f (?lubf A)
    unfolding fun-lub-def fun-ord-def
    by (blast intro: lub-upper chain-fun[OF A] f)
next
  fix A :: ('b ⇒ 'a) set and g :: 'b ⇒ 'a
  assume A: chain ?ordf A and g: ∧f. f ∈ A ⇒ ?ordf f g
  show ?ordf (?lubf A) g unfolding fun-lub-def fun-ord-def
    by (blast intro: lub-least chain-fun[OF A] dest: g[unfolded fun-ord-def])
  qed
qed

lemma ccpo: assumes partial-function-definitions ord lb
  shows class.ccpo lb ord (mk-less ord)
using assms unfolding partial-function-definitions-def mk-less-def
by unfold-locales blast+

lemma partial-function-image:
  assumes partial-function-definitions ord Lub
  assumes inj: ∧x y. f x = f y ⇒ x = y

```

```

assumes inv:  $\bigwedge x. f (g x) = x$ 
shows partial-function-definitions (img-ord f ord) (img-lub f g Lub)
proof –
  let ?iord = img-ord f ord
  let ?ilub = img-lub f g Lub

  interpret partial-function-definitions ord Lub by fact
  show ?thesis
  proof
    fix A x assume chain ?iord A x ∈ A
    then have chain ord (f ‘ A) f x ∈ f ‘ A
      by (auto simp: img-ord-def intro: chainI dest: chainD)
    thus ?iord x (?ilub A)
      unfolding inv img-lub-def img-ord-def by (rule lub-upper)
  next
    fix A x assume chain ?iord A
    and 1:  $\bigwedge z. z \in A \implies ?iord z x$ 
    then have chain ord (f ‘ A)
      by (auto simp: img-ord-def intro: chainI dest: chainD)
    thus ?iord (?ilub A) x
      unfolding inv img-lub-def img-ord-def
      by (rule lub-least) (auto dest: 1[unfolded img-ord-def])
    qed (auto simp: img-ord-def intro: leq-refl dest: leq-trans leq-antisym inj)
qed

context partial-function-definitions
begin

abbreviation le-fun  $\equiv$  fun-ord leq
abbreviation lub-fun  $\equiv$  fun-lub lub
abbreviation fixp-fun  $\equiv$  ccpo.fixp lub-fun le-fun
abbreviation mono-body  $\equiv$  monotone le-fun leq
abbreviation admissible  $\equiv$  ccpo.admissible lub-fun le-fun

```

Interpret manually, to avoid flooding everything with facts about orders

```

lemma ccpo: class.ccpo lub-fun le-fun (mk-less le-fun)
apply (rule ccpo)
apply (rule partial-function-lift)
apply (rule partial-function-definitions-axioms)
done

```

The crucial fixed-point theorem

```

lemma mono-body-fixp:
  ( $\bigwedge x. \text{mono-body } (\lambda f. F f x) \implies \text{fixp-fun } F = F (\text{fixp-fun } F)$ )
by (rule ccpo.fixp-unfold[OF ccpo]) (auto simp: monotone-def fun-ord-def)

```

Version with curry/uncurry combinators, to be used by package

```

lemma fixp-rule-uc:
  fixes F :: 'c  $\Rightarrow$  'c and

```



```

    U :: 'c ⇒ 'b ⇒ 'a and
    C :: ('b ⇒ 'a) ⇒ 'c
  assumes mono:  $\bigwedge x. \text{mono-body } (\lambda f. U (F (C f))) x$ 
  assumes eq:  $f \equiv C (\text{fixp-fun } (\lambda f. U (F (C f))))$ 
  assumes inverse:  $\bigwedge f. C (U f) = f$ 
  shows  $f = F f$ 
proof –
  have  $f = C (\text{fixp-fun } (\lambda f. U (F (C f))))$  by (simp add: eq)
  also have ... =  $C (U (F (C (\text{fixp-fun } (\lambda f. U (F (C f)))))))$ 
    by (subst mono-body-fixp[of %f. U (F (C f)), OF mono] (rule refl))
  also have ... =  $F (C (\text{fixp-fun } (\lambda f. U (F (C f)))))$  by (rule inverse)
  also have ... =  $F f$  by (simp add: eq)
  finally show  $f = F f$  .
qed

```

Fixpoint induction rule

```

lemma fixp-induct-uc:
  fixes F :: 'c ⇒ 'c
    and U :: 'c ⇒ 'b ⇒ 'a
    and C :: ('b ⇒ 'a) ⇒ 'c
    and P :: ('b ⇒ 'a) ⇒ bool
  assumes mono:  $\bigwedge x. \text{mono-body } (\lambda f. U (F (C f))) x$ 
    and eq:  $f \equiv C (\text{fixp-fun } (\lambda f. U (F (C f))))$ 
    and inverse:  $\bigwedge f. U (C f) = f$ 
    and adm: ccpo.admissible lub-fun le-fun P
    and bot:  $P (\lambda-. \text{lub } \{\})$ 
    and step:  $\bigwedge f. P (U f) \implies P (U (F f))$ 
  shows  $P (U f)$ 
unfolding eq inverse
proof (rule ccpo.fixp-induct[OF ccpo adm])
  show monotone le-fun le-fun ( $\lambda f. U (F (C f))$ )
    using mono by (auto simp: monotone-def fun-ord-def)
next
  show  $P (\text{lub-fun } \{\})$ 
    by (auto simp: bot fun-lub-def)
next
  fix x
  assume  $P x$ 
  then show  $P (U (F (C x)))$ 
    using step[of C x] by (simp add: inverse)
qed

```

Rules for *monotone le-fun leq*:

```

lemma const-mono[partial-function-mono]: monotone ord leq ( $\lambda f. c$ )
by (rule monotoneI) (rule leq-refl)

end

```

51.2 Flat interpretation: tailrec and option

definition

$flat\text{-}ord\ b\ x\ y \longleftrightarrow x = b \vee x = y$

definition

$flat\text{-}lub\ b\ A = (if\ A \subseteq \{b\}\ \text{then}\ b\ \text{else}\ (THE\ x.\ x \in A - \{b\}))$

lemma *flat-interpretation:*

partial-function-definitions (*flat-ord* *b*) (*flat-lub* *b*)

proof

fix *A x* **assume** *1: chain (flat-ord b) A x ∈ A*

show *flat-ord b x (flat-lub b A)*

proof *cases*

assume $x = b$

thus *?thesis* **by** (*simp add: flat-ord-def*)

next

assume $x \neq b$

with *1* **have** $A - \{b\} = \{x\}$

by (*auto elim: chainE simp: flat-ord-def*)

then **have** $flat\text{-}lub\ b\ A = x$

by (*auto simp: flat-lub-def*)

thus *?thesis* **by** (*auto simp: flat-ord-def*)

qed

next

fix *A z* **assume** *A: chain (flat-ord b) A*

and *z: $\bigwedge x. x \in A \implies flat\text{-}ord\ b\ x\ z$*

show $flat\text{-}ord\ b\ (flat\text{-}lub\ b\ A)\ z$

proof *cases*

assume $A \subseteq \{b\}$

thus *?thesis*

by (*auto simp: flat-lub-def flat-ord-def*)

next

assume *nb: $\neg A \subseteq \{b\}$*

then **obtain** *y* **where** $y \in A\ y \neq b$ **by** *auto*

with *A* **have** $A - \{b\} = \{y\}$

by (*auto elim: chainE simp: flat-ord-def*)

with *nb* **have** $flat\text{-}lub\ b\ A = y$

by (*auto simp: flat-lub-def*)

with *z y* **show** *?thesis* **by** *auto*

qed

qed (*auto simp: flat-ord-def*)

lemma *flat-ordI: $(x \neq a \implies x = y) \implies flat\text{-}ord\ a\ x\ y$*

by(*auto simp add: flat-ord-def*)

lemma *flat-ord-antisym: $\llbracket flat\text{-}ord\ a\ x\ y; flat\text{-}ord\ a\ y\ x \rrbracket \implies x = y$*

by(*auto simp add: flat-ord-def*)

lemma *antisymp-flat-ord: antisymp (flat-ord a)*

by(rule *antisymI*)(auto dest: *flat-ord-antisym*)

interpretation *tailrec*:

partial-function-definitions flat-ord undefined flat-lub undefined

rewrites *flat-lub undefined {} ≡ undefined*

by (rule *flat-interpretation*)(simp add: *flat-lub-def*)

interpretation *option*:

partial-function-definitions flat-ord None flat-lub None

rewrites *flat-lub None {} ≡ None*

by (rule *flat-interpretation*)(simp add: *flat-lub-def*)

abbreviation *tailrec-ord ≡ flat-ord undefined*

abbreviation *mono-tailrec ≡ monotone (fun-ord tailrec-ord) tailrec-ord*

lemma *tailrec-admissible*:

ccpo.admissible (fun-lub (flat-lub c)) (fun-ord (flat-ord c))

(λa. ∀x. a x ≠ c → P x (a x))

proof(intro *ccpo.admissibleI strip*)

fix *A x*

assume *chain: Complete-Partial-Order.chain (fun-ord (flat-ord c)) A*

and *P [rule-format]: ∀f∈A. ∀x. f x ≠ c → P x (f x)*

and *defined: fun-lub (flat-lub c) A x ≠ c*

from *defined* **obtain** *f* **where** *f: f ∈ A f x ≠ c*

by(auto simp add: *fun-lub-def flat-lub-def split: if-split-asm*)

hence *P x (f x)* **by**(rule *P*)

moreover from *chain f* **have** *∀f' ∈ A. f' x = c ∨ f' x = f x*

by(auto 4 4 simp add: *Complete-Partial-Order.chain-def flat-ord-def fun-ord-def*)

hence *fun-lub (flat-lub c) A x = f x*

using *f* **by**(auto simp add: *fun-lub-def flat-lub-def*)

ultimately show *P x (fun-lub (flat-lub c) A x)* **by** *simp*

qed

lemma *fixp-induct-tailrec*:

fixes *F :: 'c ⇒ 'c* **and**

U :: 'c ⇒ 'b ⇒ 'a **and**

C :: ('b ⇒ 'a) ⇒ 'c **and**

P :: 'b ⇒ 'a ⇒ bool **and**

x :: 'b

assumes *mono: ∧x. monotone (fun-ord (flat-ord c)) (flat-ord c) (λf. U (F (C f)) x)*

assumes *eq: f ≡ C (ccpo.fixp (fun-lub (flat-lub c)) (fun-ord (flat-ord c)) (λf. U (F (C f))))*

assumes *inverse2: ∧f. U (C f) = f*

assumes *step: ∧f x y. (∧x y. U f x = y ⇒ y ≠ c ⇒ P x y) ⇒ U (F f) x = y ⇒ y ≠ c ⇒ P x y*

assumes *result: U f x = y*

assumes *defined: y ≠ c*

shows $P x y$
proof –
 have $\forall x y. U f x = y \longrightarrow y \neq c \longrightarrow P x y$
 by (rule partial-function-definitions.fixp-induct-uc[OF flat-interpretation, of - U
 F C, OF mono eq inverse2])
 (auto intro: step tailrec-admissible simp add: fun-lub-def flat-lub-def)
 thus ?thesis using result defined by blast
qed

lemma *admissible-image*:
 assumes pfun: partial-function-definitions le lub
 assumes adm: ccpo.admissible lub le (P o g)
 assumes inj: $\bigwedge x y. f x = f y \implies x = y$
 assumes inv: $\bigwedge x. f (g x) = x$
 shows ccpo.admissible (img-lub f g lub) (img-ord f le) P
proof (rule ccpo.admissibleI)
 fix A assume chain (img-ord f le) A
 then have ch': chain le (f ' A)
 by (auto simp: img-ord-def intro: chainI dest: chainD)
 assume $A \neq \{\}$
 assume P-A: $\forall x \in A. P x$
 have (P o g) (lub (f ' A)) using adm ch'
proof (rule ccpo.admissibleD)
 fix x assume $x \in f ' A$
 with P-A show (P o g) x by (auto simp: inj[OF inv])
qed(simp add: $\langle A \neq \{\} \rangle$)
 thus P (img-lub f g lub A) unfolding img-lub-def by simp
qed

lemma *admissible-fun*:
 assumes pfun: partial-function-definitions le lub
 assumes adm: $\bigwedge x. \text{ccpo.admissible lub le } (Q x)$
 shows ccpo.admissible (fun-lub lub) (fun-ord le) ($\lambda f. \forall x. Q x (f x)$)
proof (rule ccpo.admissibleI)
 fix A :: ('b \Rightarrow 'a) set
 assume Q: $\forall f \in A. \forall x. Q x (f x)$
 assume ch: chain (fun-ord le) A
 assume $A \neq \{\}$
 hence non-empty: $\bigwedge a. \{y. \exists f \in A. y = f a\} \neq \{\}$ by auto
 show $\forall x. Q x (\text{fun-lub lub } A x)$
 unfolding fun-lub-def
 by (rule allI, rule ccpo.admissibleD[OF adm chain-fun[OF ch] non-empty])
 (auto simp: Q)
qed

abbreviation option-ord \equiv flat-ord None

abbreviation mono-option \equiv monotone (fun-ord option-ord) option-ord

lemma *bind-mono*[*partial-function-mono*]:
assumes *mf*: *mono-option B* **and** *mg*: $\bigwedge y. \text{mono-option } (\lambda f. C\ y\ f)$
shows *mono-option* $(\lambda f. \text{Option.bind } (B\ f)\ (\lambda y. C\ y\ f))$
proof (*rule monotoneI*)
 fix *f g* :: *'a* \Rightarrow *'b option* **assume** *fg*: *fun-ord option-ord f g*
 with *mf*
 have *option-ord* $(B\ f)\ (B\ g)$ **by** (*rule monotoneD*[*of - - - f g*])
 then have *option-ord* $(\text{Option.bind } (B\ f)\ (\lambda y. C\ y\ f))\ (\text{Option.bind } (B\ g)\ (\lambda y. C\ y\ f))$
 unfolding *flat-ord-def* **by** *auto*
 also from *mg*
 have $\bigwedge y'. \text{option-ord } (C\ y'\ f)\ (C\ y'\ g)$
 by (*rule monotoneD*) (*rule fg*)
 then have *option-ord* $(\text{Option.bind } (B\ g)\ (\lambda y'. C\ y'\ f))\ (\text{Option.bind } (B\ g)\ (\lambda y'. C\ y'\ g))$
 unfolding *flat-ord-def* **by** (*cases B g*) *auto*
 finally (*option.leq-trans*)
 show *option-ord* $(\text{Option.bind } (B\ f)\ (\lambda y. C\ y\ f))\ (\text{Option.bind } (B\ g)\ (\lambda y'. C\ y'\ g))$.
qed

lemma *flat-lub-in-chain*:
assumes *ch*: *chain (flat-ord b) A*
assumes *lub*: *flat-lub b A = a*
shows $a = b \vee a \in A$
proof (*cases A* \subseteq $\{b\}$)
 case *True*
 then have *flat-lub b A = b* **unfolding** *flat-lub-def* **by** *simp*
 with *lub* **show** *?thesis* **by** *simp*
next
 case *False*
 then obtain *c* **where** $c \in A$ **and** $c \neq b$ **by** *auto*
 { fix *z* **assume** $z \in A$
 from *chainD*[*OF ch* $\langle c \in A \rangle$ *this*] **have** $z = c \vee z = b$
 unfolding *flat-ord-def* **using** $\langle c \neq b \rangle$ **by** *auto* }
 with *False* **have** $A - \{b\} = \{c\}$ **by** *auto*
 with *False* **have** *flat-lub b A = c* **by** (*auto simp: flat-lub-def*)
 with $\langle c \in A \rangle$ *lub* **show** *?thesis* **by** *simp*
qed

lemma *option-admissible*: *option.admissible* $(\% (f :: 'a \Rightarrow 'b\ option))$.
 $(\forall x\ y. f\ x = \text{Some } y \longrightarrow P\ x\ y)$
proof (*rule ccpo.admissibleI*)
fix *A* :: $'a \Rightarrow 'b\ option$ *set*
assume *ch*: *chain option.le-fun A*
 and *IH*: $\forall f \in A. \forall x\ y. f\ x = \text{Some } y \longrightarrow P\ x\ y$
from *ch* **have** *ch'*: $\bigwedge x. \text{chain option-ord } \{y. \exists f \in A. y = f\ x\}$ **by** (*rule chain-fun*)
show $\forall x\ y. \text{option.lub-fun } A\ x = \text{Some } y \longrightarrow P\ x\ y$
proof (*intro allI impI*)

```

fix  $x\ y$  assume option.lub-fun  $A\ x = \text{Some } y$ 
from flat-lub-in-chain[OF ch' this[unfolded fun-lub-def]]
have  $\text{Some } y \in \{y. \exists f \in A. y = f\ x\}$  by simp
then have  $\exists f \in A. f\ x = \text{Some } y$  by auto
with IH show  $P\ x\ y$  by auto
qed
qed

```

lemma *fixp-induct-option*:

```

fixes  $F :: 'c \Rightarrow 'c$  and
   $U :: 'c \Rightarrow 'b \Rightarrow 'a$  option and
   $C :: ('b \Rightarrow 'a$  option)  $\Rightarrow 'c$  and
   $P :: 'b \Rightarrow 'a \Rightarrow \text{bool}$ 
assumes mono:  $\bigwedge x. \text{mono-option } (\lambda f. U\ (F\ (C\ f))\ x)$ 
assumes eq:  $f \equiv C\ (\text{ccpo.fixp } (\text{fun-lub } (\text{flat-lub } \text{None}))\ (\text{fun-ord } \text{option-ord})\ (\lambda f. U\ (F\ (C\ f))))$ 
assumes inverse2:  $\bigwedge f. U\ (C\ f) = f$ 
assumes step:  $\bigwedge f\ x\ y. (\bigwedge x\ y. U\ f\ x = \text{Some } y \Longrightarrow P\ x\ y) \Longrightarrow U\ (F\ f)\ x = \text{Some } y \Longrightarrow P\ x\ y$ 
assumes defined:  $U\ f\ x = \text{Some } y$ 
shows  $P\ x\ y$ 
using step defined option.fixp-induct-uc[of U F C, OF mono eq inverse2 option-admissible]
unfolding fun-lub-def flat-lub-def by(auto 9 2)

```

```

declaration  $\langle \text{Partial-Function.init tailrec term } \langle \text{tailrec.fixp-fun} \rangle$ 
   $\text{term } \langle \text{tailrec.mono-body} \rangle @\{\text{thm tailrec.fixp-rule-uc}\} @\{\text{thm tailrec.fixp-induct-uc}\}$ 
   $(\text{SOME } @\{\text{thm fixp-induct-tailrec}[\text{where } c = \text{undefined}]\}) \rangle$ 

```

```

declaration  $\langle \text{Partial-Function.init option term } \langle \text{option.fixp-fun} \rangle$ 
   $\text{term } \langle \text{option.mono-body} \rangle @\{\text{thm option.fixp-rule-uc}\} @\{\text{thm option.fixp-induct-uc}\}$ 
   $(\text{SOME } @\{\text{thm fixp-induct-option}\}) \rangle$ 

```

hide-const (**open**) *chain*

end

```

theory Argo
imports HOL
begin

```

```

ML-file  $\langle \sim\sim / \text{src} / \text{Tools} / \text{Argo} / \text{argo-expr.ML} \rangle$ 
ML-file  $\langle \sim\sim / \text{src} / \text{Tools} / \text{Argo} / \text{argo-term.ML} \rangle$ 
ML-file  $\langle \sim\sim / \text{src} / \text{Tools} / \text{Argo} / \text{argo-lit.ML} \rangle$ 
ML-file  $\langle \sim\sim / \text{src} / \text{Tools} / \text{Argo} / \text{argo-proof.ML} \rangle$ 
ML-file  $\langle \sim\sim / \text{src} / \text{Tools} / \text{Argo} / \text{argo-rewr.ML} \rangle$ 
ML-file  $\langle \sim\sim / \text{src} / \text{Tools} / \text{Argo} / \text{argo-cls.ML} \rangle$ 
ML-file  $\langle \sim\sim / \text{src} / \text{Tools} / \text{Argo} / \text{argo-common.ML} \rangle$ 

```

```

ML-file <~/src/Tools/Argo/argo-cc.ML>
ML-file <~/src/Tools/Argo/argo-simplex.ML>
ML-file <~/src/Tools/Argo/argo-thy.ML>
ML-file <~/src/Tools/Argo/argo-heap.ML>
ML-file <~/src/Tools/Argo/argo-cdcl.ML>
ML-file <~/src/Tools/Argo/argo-core.ML>
ML-file <~/src/Tools/Argo/argo-clausify.ML>
ML-file <~/src/Tools/Argo/argo-solver.ML>

```

```
ML-file <Tools/Argo/argo-tactic.ML>
```

```
end
```

52 Reconstructing external resolution proofs for propositional logic

```

theory SAT
imports Argo
begin

```

```

ML-file <Tools/prop-logic.ML>
ML-file <Tools/sat-solver.ML>
ML-file <Tools/sat.ML>
ML-file <Tools/Argo/argo-sat-solver.ML>

```

```

method-setup sat = <Scan.succeed (SIMPLE-METHOD' o SAT.sat-tac)>
  SAT solver

```

```

method-setup satx = <Scan.succeed (SIMPLE-METHOD' o SAT.satx-tac)>
  SAT solver (with definitional CNF)

```

```
end
```

53 Function Definitions and Termination Proofs

```

theory Fun-Def
  imports Basic-BNF-LFPs Partial-Function SAT
  keywords
    function termination :: thy-goal-defn and
    fun fun-cases :: thy-defn
begin

```

53.1 Definitions with default value

```

definition THE-default :: 'a ⇒ ('a ⇒ bool) ⇒ 'a
  where THE-default d P = (if (∃!x. P x) then (THE x. P x) else d)

```

```

lemma THE-defaultI': ∃!x. P x ⇒ P (THE-default d P)

```

by (simp add: theI' THE-default-def)

lemma THE-default1-equality: $\exists!x. P x \implies P a \implies \text{THE-default } d P = a$
 by (simp add: the1-equality THE-default-def)

lemma THE-default-none: $\neg (\exists!x. P x) \implies \text{THE-default } d P = d$
 by (simp add: THE-default-def)

lemma fundef-ex1-existence:
 assumes f-def: $f \equiv (\lambda x::'a. \text{THE-default } (d x) (\lambda y. G x y))$
 assumes ex1: $\exists!y. G x y$
 shows $G x (f x)$
 apply (simp only: f-def)
 apply (rule THE-defaultI')
 apply (rule ex1)
 done

lemma fundef-ex1-uniqueness:
 assumes f-def: $f \equiv (\lambda x::'a. \text{THE-default } (d x) (\lambda y. G x y))$
 assumes ex1: $\exists!y. G x y$
 assumes elm: $G x (h x)$
 shows $h x = f x$
 by (auto simp add: f-def ex1 elm THE-default1-equality[symmetric])

lemma fundef-ex1-iff:
 assumes f-def: $f \equiv (\lambda x::'a. \text{THE-default } (d x) (\lambda y. G x y))$
 assumes ex1: $\exists!y. G x y$
 shows $(G x y) = (f x = y)$
 by (auto simp add: ex1 f-def THE-default1-equality THE-defaultI')

lemma fundef-default-value:
 assumes f-def: $f \equiv (\lambda x::'a. \text{THE-default } (d x) (\lambda y. G x y))$
 assumes graph: $\bigwedge x y. G x y \implies D x$
 assumes $\neg D x$
 shows $f x = d x$

proof –
 have $\neg(\exists y. G x y)$
proof
 assume $\exists y. G x y$
 then have $D x$ using graph ..
 with $\langle \neg D x \rangle$ show False ..
qed
 then have $\neg(\exists!y. G x y)$ by blast
 then show ?thesis
 unfolding f-def by (rule THE-default-none)
qed

definition *in-rel-def*[simp]: $in\text{-}rel\ R\ x\ y \equiv (x, y) \in R$

lemma *wf-in-rel*: $wf\ R \implies wfP\ (in\text{-}rel\ R)$
by (*simp add: wfP-def*)

ML-file $\langle Tools/Function/function\text{-}core.ML \rangle$

ML-file $\langle Tools/Function/mutual.ML \rangle$

ML-file $\langle Tools/Function/pattern\text{-}split.ML \rangle$

ML-file $\langle Tools/Function/relation.ML \rangle$

ML-file $\langle Tools/Function/function\text{-}elims.ML \rangle$

method-setup *relation* = \langle
Args.term $\gg (fn\ t \implies fn\ ctxt \implies SIMPLE\text{-}METHOD' (Function\text{-}Relation.relation\text{-}infer\text{-}tac\ ctxt\ t))$
 \rangle *prove termination using a user-specified wellfounded relation*

ML-file $\langle Tools/Function/function.ML \rangle$

ML-file $\langle Tools/Function/pat\text{-}completeness.ML \rangle$

method-setup *pat-completeness* = \langle
Scan.succeed ($SIMPLE\text{-}METHOD' \circ Pat\text{-}Completeness.pat\text{-}completeness\text{-}tac$)
 \rangle *prove completeness of (co)datatype patterns*

ML-file $\langle Tools/Function/fun.ML \rangle$

ML-file $\langle Tools/Function/induction\text{-}schema.ML \rangle$

method-setup *induction-schema* = \langle
Scan.succeed ($CONTEXT\text{-}TACTIC \circ Induction\text{-}Schema.induction\text{-}schema\text{-}tac$)
 \rangle *prove an induction principle*

53.2 Measure functions

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$
where *is-measure-trivial*: $is\text{-}measure\ f$

named-theorems *measure-function* rules that guide the heuristic generation of measure functions

ML-file $\langle Tools/Function/measure\text{-}functions.ML \rangle$

lemma *measure-size*[*measure-function*]: $is\text{-}measure\ size$
by (*rule is-measure-trivial*)

lemma *measure-fst*[*measure-function*]: $is\text{-}measure\ f \implies is\text{-}measure\ (\lambda p. f\ (fst\ p))$
by (*rule is-measure-trivial*)

lemma *measure-snd*[*measure-function*]: $is\text{-}measure\ f \implies is\text{-}measure\ (\lambda p. f\ (snd\ p))$
by (*rule is-measure-trivial*)

ML-file $\langle \text{Tools/Function/lexicographic-order.ML} \rangle$

method-setup *lexicographic-order* = \langle
Method.sections clasimp-modifiers $\rangle \rangle$
 (*K (SIMPLE-METHOD o Lexicographic-Order.lexicographic-order-tac false)*)
 \rangle *termination prover for lexicographic orderings*

53.3 Congruence rules

lemma *let-cong* [*fundef-cong*]: $M = N \implies (\bigwedge x. x = N \implies f x = g x) \implies \text{Let } M$
 $f = \text{Let } N g$
unfolding *Let-def* **by** *blast*

lemmas [*fundef-cong*] =
if-cong image-cong
bex-cong ball-cong imp-cong map-option-cong Option.bind-cong

lemma *split-cong* [*fundef-cong*]:
 $(\bigwedge x y. (x, y) = q \implies f x y = g x y) \implies p = q \implies \text{case-prod } f p = \text{case-prod } g q$
by (*auto simp: split-def*)

lemma *comp-cong* [*fundef-cong*]: $f (g x) = f' (g' x') \implies (f \circ g) x = (f' \circ g') x'$
by (*simp only: o-apply*)

53.4 Simp rules for termination proofs

declare

trans-less-add1 [*termination-simp*]
trans-less-add2 [*termination-simp*]
trans-le-add1 [*termination-simp*]
trans-le-add2 [*termination-simp*]
less-imp-le-nat [*termination-simp*]
le-imp-less-Suc [*termination-simp*]

lemma *size-prod-simp* [*termination-simp*]: $\text{size-prod } f g p = f (\text{fst } p) + g (\text{snd } p)$
 $+ \text{Suc } 0$
by (*induct p*) *auto*

53.5 Decomposition

lemma *less-by-empty*: $A = \{\} \implies A \subseteq B$
and *union-comp-emptyL*: $A \circ C = \{\} \implies B \circ C = \{\} \implies (A \cup B) \circ C = \{\}$
and *union-comp-emptyR*: $A \circ B = \{\} \implies A \circ C = \{\} \implies A \circ (B \cup C) = \{\}$
and *wf-no-loop*: $R \circ R = \{\} \implies \text{wf } R$
by (*auto simp add: wf-comp-self [of R]*)

53.6 Reduction pairs

definition *reduction-pair* $P \longleftrightarrow \text{wf } (\text{fst } P) \wedge \text{fst } P \circ \text{snd } P \subseteq \text{fst } P$

lemma *reduction-pairI*[intro]: $wf\ R \implies R\ O\ S \subseteq R \implies reduction\text{-}pair\ (R, S)$
by (*auto simp: reduction-pair-def*)

lemma *reduction-pair-lemma*:

assumes *rp*: *reduction-pair* *P*

assumes $R \subseteq fst\ P$

assumes $S \subseteq snd\ P$

assumes $wf\ S$

shows $wf\ (R \cup S)$

proof –

from *rp* $\langle S \subseteq snd\ P \rangle$ **have** $wf\ (fst\ P)\ fst\ P\ O\ S \subseteq fst\ P$

unfolding *reduction-pair-def* **by** *auto*

with $\langle wf\ S \rangle$ **have** $wf\ (fst\ P \cup S)$

by (*auto intro: wf-union-compatible*)

moreover from $\langle R \subseteq fst\ P \rangle$ **have** $R \cup S \subseteq fst\ P \cup S$ **by** *auto*

ultimately show *?thesis* **by** (*rule wf-subset*)

qed

definition *rp-inv-image* = $(\lambda(R,S)\ f.\ (inv\text{-}image\ R\ f,\ inv\text{-}image\ S\ f))$

lemma *rp-inv-image-rp*: *reduction-pair* *P* \implies *reduction-pair* (*rp-inv-image* *P* *f*)

unfolding *reduction-pair-def* *rp-inv-image-def* *split-def* **by** *force*

53.7 Concrete orders for SCNP termination proofs

definition *pair-less* = *less-than* $\langle *lex* \rangle$ *less-than*

definition *pair-leq* = *pair-less*⁼

definition *max-strict* = *max-ext* *pair-less*

definition *max-weak* = *max-ext* *pair-leq* $\cup \{(\{\}, \{\})\}$

definition *min-strict* = *min-ext* *pair-less*

definition *min-weak* = *min-ext* *pair-leq* $\cup \{(\{\}, \{\})\}$

lemma *wf-pair-less*[*simp*]: $wf\ pair\text{-}less$

by (*auto simp: pair-less-def*)

lemma *total-pair-less* [*iff*]: *total-on* *A* *pair-less* **and** *trans-pair-less* [*iff*]: *trans* *pair-less*

by (*auto simp: total-on-def pair-less-def*)

Introduction rules for *pair-less*/*pair-leq*

lemma *pair-leqI1*: $a < b \implies ((a, s), (b, t)) \in pair\text{-}leq$

and *pair-leqI2*: $a \leq b \implies s \leq t \implies ((a, s), (b, t)) \in pair\text{-}leq$

and *pair-lessI1*: $a < b \implies ((a, s), (b, t)) \in pair\text{-}less$

and *pair-lessI2*: $a \leq b \implies s < t \implies ((a, s), (b, t)) \in pair\text{-}less$

by (*auto simp: pair-leq-def pair-less-def*)

lemma *pair-less-iff1* [*simp*]: $((x, y), (x, z)) \in pair\text{-}less \iff y < z$

by (*simp add: pair-less-def*)

Introduction rules for *max*

lemma *smax-emptyI*: $\text{finite } Y \implies Y \neq \{\} \implies (\{\}, Y) \in \text{max-strict}$
and *smax-insertI*:
 $y \in Y \implies (x, y) \in \text{pair-less} \implies (X, Y) \in \text{max-strict} \implies (\text{insert } x \ X, Y) \in \text{max-strict}$
and *wmax-emptyI*: $\text{finite } X \implies (\{\}, X) \in \text{max-weak}$
and *wmax-insertI*:
 $y \in YS \implies (x, y) \in \text{pair-leq} \implies (XS, YS) \in \text{max-weak} \implies (\text{insert } x \ XS, YS) \in \text{max-weak}$
by (*auto simp: max-strict-def max-weak-def elim!: max-ext.cases*)

Introduction rules for min

lemma *smin-emptyI*: $X \neq \{\} \implies (X, \{\}) \in \text{min-strict}$
and *smin-insertI*:
 $x \in XS \implies (x, y) \in \text{pair-less} \implies (XS, YS) \in \text{min-strict} \implies (XS, \text{insert } y \ YS) \in \text{min-strict}$
and *wmin-emptyI*: $(X, \{\}) \in \text{min-weak}$
and *wmin-insertI*:
 $x \in XS \implies (x, y) \in \text{pair-leq} \implies (XS, YS) \in \text{min-weak} \implies (XS, \text{insert } y \ YS) \in \text{min-weak}$
by (*auto simp: min-strict-def min-weak-def min-ext-def*)

Reduction Pairs.

lemma *max-ext-compat*:

assumes $R \ O \ S \subseteq R$

shows $\text{max-ext } R \ O \ (\text{max-ext } S \cup \{(\{\}, \{\})\}) \subseteq \text{max-ext } R$

proof –

have $\bigwedge X \ Y \ Z. (X, Y) \in \text{max-ext } R \implies (Y, Z) \in \text{max-ext } S \implies (X, Z) \in \text{max-ext } R$

proof –

fix $X \ Y \ Z$

assume $(X, Y) \in \text{max-ext } R$

$(Y, Z) \in \text{max-ext } S$

then have *: $\text{finite } X \ \text{finite } Y \ \text{finite } Z \ Y \neq \{\} \ Z \neq \{\}$

$(\bigwedge x. x \in X \implies \exists y \in Y. (x, y) \in R)$

$(\bigwedge y. y \in Y \implies \exists z \in Z. (y, z) \in S)$

by (*auto elim: max-ext.cases*)

moreover have $\bigwedge x. x \in X \implies \exists z \in Z. (x, z) \in R$

proof –

fix x

assume $x \in X$

then obtain y **where** $1: y \in Y \ (x, y) \in R$

using * **by** *auto*

then obtain z **where** $z \in Z \ (y, z) \in S$

using * **by** *auto*

then show $\exists z \in Z. (x, z) \in R$

using *assms 1* **by** (*auto elim: max-ext.cases*)

qed

ultimately show $(X, Z) \in \text{max-ext } R$

by *auto*

```

qed
then show max-ext R O (max-ext S ∪ {{{}, {}}}) ⊆ max-ext R
  by auto
qed

```

```

lemma max-rpair-set: reduction-pair (max-strict, max-weak)
  unfolding max-strict-def max-weak-def
  apply (intro reduction-pairI max-ext-wf)
  apply simp
  apply (rule max-ext-compat)
  apply (auto simp: pair-less-def pair-leq-def)
  done

```

```

lemma min-ext-compat:
  assumes R O S ⊆ R
  shows min-ext R O (min-ext S ∪ {{{}, {}}}) ⊆ min-ext R
proof -
  have  $\bigwedge X Y Z z. \forall y \in Y. \exists x \in X. (x, y) \in R \implies \forall z \in Z. \exists y \in Y. (y, z) \in S$ 
 $\implies z \in Z \implies \exists x \in X. (x, z) \in R$ 
  proof -
    fix X Y Z z
    assume *:  $\forall y \in Y. \exists x \in X. (x, y) \in R$ 
       $\forall z \in Z. \exists y \in Y. (y, z) \in S$ 
       $z \in Z$ 
    then obtain y' where 1:  $y' \in Y (y', z) \in S$ 
      by auto
    then obtain x' where 2:  $x' \in X (x', y') \in R$ 
      using * by auto
    show  $\exists x \in X. (x, z) \in R$ 
      using 1 2 assms by auto
  qed
  then show ?thesis
    using assms by (auto simp: min-ext-def)
qed

```

```

lemma min-rpair-set: reduction-pair (min-strict, min-weak)
  unfolding min-strict-def min-weak-def
  apply (intro reduction-pairI min-ext-wf)
  apply simp
  apply (rule min-ext-compat)
  apply (auto simp: pair-less-def pair-leq-def)
  done

```

53.8 Yet another induction principle on the natural numbers

```

lemma nat-descend-induct [case-names base descend]:
  fixes P :: nat ⇒ bool
  assumes H1:  $\bigwedge k. k > n \implies P k$ 
  assumes H2:  $\bigwedge k. k \leq n \implies (\bigwedge i. i > k \implies P i) \implies P k$ 

```

shows $P\ m$
 using *assms* by *induction-schema* (*force intro!*: *wf-measure* [of $\lambda k. \text{Suc } n - k$])+

53.9 Tool setup

ML-file $\langle \text{Tools/Function/termination.ML} \rangle$

ML-file $\langle \text{Tools/Function/scnp-solve.ML} \rangle$

ML-file $\langle \text{Tools/Function/scnp-reconstruct.ML} \rangle$

ML-file $\langle \text{Tools/Function/fun-cases.ML} \rangle$

ML-val — setup inactive

```
<
  Context.theory-map (Function-Common.set-termination-prover
    (K (ScnpReconstruct.decomp-scnp-tac [ScnpSolve.MAX, ScnpSolve.MIN, Sc-
      npSolve.MS])))
  >
```

end

54 The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *Int*

imports *Quotient Groups-Big Fun-Def*

begin

54.1 Definition of integers as a quotient type

definition *intrel* :: $(\text{nat} \times \text{nat}) \Rightarrow (\text{nat} \times \text{nat}) \Rightarrow \text{bool}$

where *intrel* = $(\lambda(x, y) (u, v). x + v = u + y)$

lemma *intrel-iff* [*simp*]: *intrel* $(x, y) (u, v) \longleftrightarrow x + v = u + y$

by (*simp add: intrel-def*)

quotient-type *int* = $\text{nat} \times \text{nat} / \text{intrel}$

morphisms *Rep-Integ Abs-Integ*

proof (*rule equivpI*)

show *reflp intrel* by (*auto simp: reflp-def*)

show *symp intrel* by (*auto simp: symp-def*)

show *transp intrel* by (*auto simp: transp-def*)

qed

lemma *eq-Abs-Integ* [*case-names Abs-Integ, cases type: int*]:

$(\bigwedge x\ y. z = \text{Abs-Integ } (x, y) \implies P) \implies P$

by (*induct z*) *auto*

54.2 Integers form a commutative ring

instantiation *int* :: *comm-ring-1*

begin

lift-definition *zero-int* :: *int* **is** $(0, 0)$.

lift-definition *one-int* :: *int* **is** $(1, 0)$.

lift-definition *plus-int* :: *int* \Rightarrow *int* \Rightarrow *int*
is $\lambda(x, y) (u, v). (x + u, y + v)$
by *clarsimp*

lift-definition *uminus-int* :: *int* \Rightarrow *int*
is $\lambda(x, y). (y, x)$
by *clarsimp*

lift-definition *minus-int* :: *int* \Rightarrow *int* \Rightarrow *int*
is $\lambda(x, y) (u, v). (x + v, y + u)$
by *clarsimp*

lift-definition *times-int* :: *int* \Rightarrow *int* \Rightarrow *int*
is $\lambda(x, y) (u, v). (x*u + y*v, x*v + y*u)$

proof (*unfold intrel-def, clarify*)

fix *s t u v w x y z* :: *nat*

assume $s + v = u + t$ **and** $w + z = y + x$

then have $(s + v) * w + (u + t) * x + u * (w + z) + v * (y + x) =$
 $(u + t) * w + (s + v) * x + u * (y + x) + v * (w + z)$

by *simp*

then show $(s * w + t * x) + (u * z + v * y) = (u * y + v * z) + (s * x + t * w)$

by (*simp add: algebra-simps*)

qed

instance

by *standard (transfer; clarsimp simp: algebra-simps)+*

end

abbreviation *int* :: *nat* \Rightarrow *int*

where $int \equiv of\text{-}nat$

lemma *int-def*: $int\ n = Abs\text{-}Integ\ (n, 0)$

by (*induct n (simp add: zero-int.abs-eq, simp add: one-int.abs-eq plus-int.abs-eq)*)

lemma *int-transfer* [*transfer-rule*]:

includes *lifting-syntax*

shows *rel-fun* $(=)$ *pcr-int* $(\lambda n. (n, 0))$ *int*

by (*simp add: rel-fun-def int.pcr-cr-eq cr-int-def int-def*)

lemma *int-diff-cases*: **obtains** (*diff*) *m n* **where** $z = int\ m - int\ n$

by *transfer clarsimp*

54.3 Integers are totally ordered

instantiation *int* :: *linorder*
begin

lift-definition *less-eq-int* :: *int* \Rightarrow *int* \Rightarrow *bool*
is $\lambda(x, y) (u, v). x + v \leq u + y$
by *auto*

lift-definition *less-int* :: *int* \Rightarrow *int* \Rightarrow *bool*
is $\lambda(x, y) (u, v). x + v < u + y$
by *auto*

instance
by *standard* (*transfer*, *force*)+

end

instantiation *int* :: *distrib-lattice*
begin

definition (*inf* :: *int* \Rightarrow *int* \Rightarrow *int*) = *min*

definition (*sup* :: *int* \Rightarrow *int* \Rightarrow *int*) = *max*

instance
by *standard* (*auto simp add: inf-int-def sup-int-def max-min-distrib2*)

end

54.4 Ordering properties of arithmetic operations

instance *int* :: *ordered-cancel-ab-semigroup-add*
proof
fix *i j k* :: *int*
show $i \leq j \Longrightarrow k + i \leq k + j$
by *transfer clarsimp*
qed

Strict Monotonicity of Multiplication.

Strict, in 1st argument; proof is by induction on $k > 0$.

lemma *zmult-zless-mono2-lemma*: $i < j \Longrightarrow 0 < k \Longrightarrow \text{int } k * i < \text{int } k * j$
for *i j* :: *int*
proof (*induct k*)
case 0
then show *?case* **by** *simp*
next
case (*Suc k*)
then show *?case*

by (cases $k = 0$) (simp-all add: distrib-right add-strict-mono)
qed

lemma zero-le-imp-eq-int:
assumes $k \geq (0::int)$ shows $\exists n. k = int\ n$
proof –
have $b \leq a \implies \exists n::nat. a = n + b$ for $a\ b$
using exI[of - $a - b$] by simp
with assms show ?thesis
by transfer auto
qed

lemma zero-less-imp-eq-int:
assumes $k > (0::int)$ shows $\exists n>0. k = int\ n$
proof –
have $b < a \implies \exists n::nat. n>0 \wedge a = n + b$ for $a\ b$
using exI[of - $a - b$] by simp
with assms show ?thesis
by transfer auto
qed

lemma zmult-zless-mono2: $i < j \implies 0 < k \implies k * i < k * j$
for $i\ j\ k :: int$
by (drule zero-less-imp-eq-int) (auto simp add: zmult-zless-mono2-lemma)

The integers form an ordered integral domain.

instantiation $int :: linordered-idom$
begin

definition zabs-def: $|i::int| = (if\ i < 0\ then\ -\ i\ else\ i)$

definition zsgn-def: $sgn\ (i::int) = (if\ i = 0\ then\ 0\ else\ if\ 0 < i\ then\ 1\ else\ -\ 1)$

instance
proof
fix $i\ j\ k :: int$
show $i < j \implies 0 < k \implies k * i < k * j$
by (rule zmult-zless-mono2)
show $|i| = (if\ i < 0\ then\ -i\ else\ i)$
by (simp only: zabs-def)
show $sgn\ (i::int) = (if\ i=0\ then\ 0\ else\ if\ 0 < i\ then\ 1\ else\ -\ 1)$
by (simp only: zsgn-def)
qed

end

lemma zless-imp-add1-zle: $w < z \implies w + 1 \leq z$
for $w\ z :: int$
by transfer clarsimp

```

lemma zless-iff-Suc-zadd:  $w < z \iff (\exists n. z = w + \text{int } (\text{Suc } n))$ 
  for  $w z :: \text{int}$ 
proof -
  have  $\bigwedge a b c d. a + d < c + b \implies \exists n. c + b = \text{Suc } (a + n + d)$ 
  proof -
    fix  $a b c d :: \text{nat}$ 
    assume  $a + d < c + b$ 
    then have  $c + b = \text{Suc } (a + (c + b - \text{Suc } (a + d)) + d)$ 
    by arith
    then show  $\exists n. c + b = \text{Suc } (a + n + d)$ 
    by (rule exI)
  qed
then show ?thesis
  by transfer auto
qed

```

```

lemma zabs-less-one-iff [simp]:  $|z| < 1 \iff z = 0$  (is ?lhs  $\iff$  ?rhs)
  for  $z :: \text{int}$ 
proof
  assume ?rhs
  then show ?lhs by simp
next
  assume ?lhs
  with zless-imp-add1-zle [of  $|z|$  1] have  $|z| + 1 \leq 1$  by simp
  then have  $|z| \leq 0$  by simp
  then show ?rhs by simp
qed

```

54.5 Embedding of the Integers into any *ring-1*: *of-int*

```

context ring-1
begin

```

```

lift-definition of-int ::  $\text{int} \Rightarrow 'a$ 
is  $\lambda(i, j). \text{of-nat } i - \text{of-nat } j$ 
by (clarsimp simp add: diff-eq-eq eq-diff-eq diff-add-eq
  of-nat-add [symmetric] simp del: of-nat-add)

```

```

lemma of-int-0 [simp]:  $\text{of-int } 0 = 0$ 
  by transfer simp

```

```

lemma of-int-1 [simp]:  $\text{of-int } 1 = 1$ 
  by transfer simp

```

```

lemma of-int-add [simp]:  $\text{of-int } (w + z) = \text{of-int } w + \text{of-int } z$ 
  by transfer (clarsimp simp add: algebra-simps)

```

```

lemma of-int-minus [simp]:  $\text{of-int } (- z) = - (\text{of-int } z)$ 

```

by (*transfer fixing: uminus*) *clarsimp*

lemma *of-int-diff* [*simp*]: $of-int (w - z) = of-int w - of-int z$
using *of-int-add* [*of w - z*] **by** *simp*

lemma *of-int-mult* [*simp*]: $of-int (w * z) = of-int w * of-int z$
by (*transfer fixing: times*) (*clarsimp simp add: algebra-simps*)

lemma *mult-of-int-commute*: $of-int x * y = y * of-int x$
by (*transfer fixing: times*) (*auto simp: algebra-simps mult-of-nat-commute*)

Collapse nested embeddings.

lemma *of-int-of-nat-eq* [*simp*]: $of-int (int n) = of-nat n$
by (*induct n*) *auto*

lemma *of-int-numeral* [*simp, code-post*]: $of-int (numeral k) = numeral k$
by (*simp add: of-nat-numeral [symmetric] of-int-of-nat-eq [symmetric]*)

lemma *of-int-neg-numeral* [*code-post*]: $of-int (- numeral k) = - numeral k$
by *simp*

lemma *of-int-power* [*simp*]: $of-int (z ^ n) = of-int z ^ n$
by (*induct n*) *simp-all*

lemma *of-int-of-bool* [*simp*]:
 $of-int (of-bool P) = of-bool P$
by *auto*

end

context *ring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]: $of-int w = of-int z \longleftrightarrow w = z$
by *transfer (clarsimp simp add: algebra-simps of-nat-add [symmetric] simp del: of-nat-add)*

Special cases where either operand is zero.

lemma *of-int-eq-0-iff* [*simp*]: $of-int z = 0 \longleftrightarrow z = 0$
using *of-int-eq-iff* [*of z 0*] **by** *simp*

lemma *of-int-0-eq-iff* [*simp*]: $0 = of-int z \longleftrightarrow z = 0$
using *of-int-eq-iff* [*of 0 z*] **by** *simp*

lemma *of-int-eq-1-iff* [*iff*]: $of-int z = 1 \longleftrightarrow z = 1$
using *of-int-eq-iff* [*of z 1*] **by** *simp*

lemma *numeral-power-eq-of-int-cancel-iff* [*simp*]:
 $numeral x ^ n = of-int y \longleftrightarrow numeral x ^ n = y$

using *of-int-eq-iff*[*of numeral x ^ n y, unfolded of-int-numeral of-int-power*].

lemma *of-int-eq-numeral-power-cancel-iff* [*simp*]:
 $of-int\ y = numeral\ x \wedge n \iff y = numeral\ x \wedge n$
using *numeral-power-eq-of-int-cancel-iff* [*of x n y*] **by** (*metis (mono-tags)*)

lemma *neg-numeral-power-eq-of-int-cancel-iff* [*simp*]:
 $(- numeral\ x) \wedge n = of-int\ y \iff (- numeral\ x) \wedge n = y$
using *of-int-eq-iff*[*of (- numeral x) ^ n y*]
by *simp*

lemma *of-int-eq-neg-numeral-power-cancel-iff* [*simp*]:
 $of-int\ y = (- numeral\ x) \wedge n \iff y = (- numeral\ x) \wedge n$
using *neg-numeral-power-eq-of-int-cancel-iff*[*of x n y*] **by** (*metis (mono-tags)*)

lemma *of-int-eq-of-int-power-cancel-iff*[*simp*]: $(of-int\ b) \wedge w = of-int\ x \iff b \wedge w = x$
by (*metis of-int-power of-int-eq-iff*)

lemma *of-int-power-eq-of-int-cancel-iff*[*simp*]: $of-int\ x = (of-int\ b) \wedge w \iff x = b \wedge w$
by (*metis of-int-eq-of-int-power-cancel-iff*)

end

context *linordered-idom*
begin

Every *linordered-idom* has characteristic zero.

subclass *ring-char-0 ..*

lemma *of-int-le-iff* [*simp*]: $of-int\ w \leq of-int\ z \iff w \leq z$
by (*transfer fixing: less-eq*)
(*clarsimp simp add: algebra-simps of-nat-add [symmetric] simp del: of-nat-add*)

lemma *of-int-less-iff* [*simp*]: $of-int\ w < of-int\ z \iff w < z$
by (*simp add: less-le order-less-le*)

lemma *of-int-0-le-iff* [*simp*]: $0 \leq of-int\ z \iff 0 \leq z$
using *of-int-le-iff* [*of 0 z*] **by** *simp*

lemma *of-int-le-0-iff* [*simp*]: $of-int\ z \leq 0 \iff z \leq 0$
using *of-int-le-iff* [*of z 0*] **by** *simp*

lemma *of-int-0-less-iff* [*simp*]: $0 < of-int\ z \iff 0 < z$
using *of-int-less-iff* [*of 0 z*] **by** *simp*

lemma *of-int-less-0-iff* [*simp*]: $of-int\ z < 0 \iff z < 0$
using *of-int-less-iff* [*of z 0*] **by** *simp*

lemma *of-int-1-le-iff* [*simp*]: $1 \leq \text{of-int } z \longleftrightarrow 1 \leq z$
using *of-int-le-iff* [*of 1 z*] **by** *simp*

lemma *of-int-le-1-iff* [*simp*]: $\text{of-int } z \leq 1 \longleftrightarrow z \leq 1$
using *of-int-le-iff* [*of z 1*] **by** *simp*

lemma *of-int-1-less-iff* [*simp*]: $1 < \text{of-int } z \longleftrightarrow 1 < z$
using *of-int-less-iff* [*of 1 z*] **by** *simp*

lemma *of-int-less-1-iff* [*simp*]: $\text{of-int } z < 1 \longleftrightarrow z < 1$
using *of-int-less-iff* [*of z 1*] **by** *simp*

lemma *of-int-pos*: $z > 0 \implies \text{of-int } z > 0$
by *simp*

lemma *of-int-nonneg*: $z \geq 0 \implies \text{of-int } z \geq 0$
by *simp*

lemma *of-int-abs* [*simp*]: $\text{of-int } |x| = |\text{of-int } x|$
by (*auto simp add: abs-if*)

lemma *of-int-lessD*:
assumes $|\text{of-int } n| < x$
shows $n = 0 \vee x > 1$
proof (*cases n = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $|n| \neq 0$ **by** *simp*
then have $|n| > 0$ **by** *simp*
then have $|n| \geq 1$
using *zless-imp-add1-zle* [*of 0 |n|*] **by** *simp*
then have $|\text{of-int } n| \geq 1$
unfolding *of-int-1-le-iff* [*of |n|, symmetric*] **by** *simp*
then have $1 < x$ **using** *assms* **by** (*rule le-less-trans*)
then show *?thesis* **..**
qed

lemma *of-int-leD*:
assumes $|\text{of-int } n| \leq x$
shows $n = 0 \vee 1 \leq x$
proof (*cases n = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $|n| \neq 0$ **by** *simp*

then have $|n| > 0$ **by** *simp*
then have $|n| \geq 1$
using *zless-imp-add1-zle* [of 0 |n|] **by** *simp*
then have $|of-int\ n| \geq 1$
unfolding *of-int-1-le-iff* [of |n|, symmetric] **by** *simp*
then have $1 \leq x$ **using** *assms* **by** (rule *order-trans*)
then show *?thesis* ..

qed

lemma *numeral-power-le-of-int-cancel-iff* [*simp*]:
 $numeral\ x \wedge n \leq of-int\ a \longleftrightarrow numeral\ x \wedge n \leq a$
by (*metis* (*mono-tags*) *local.of-int-eq-numeral-power-cancel-iff of-int-le-iff*)

lemma *of-int-le-numeral-power-cancel-iff* [*simp*]:
 $of-int\ a \leq numeral\ x \wedge n \longleftrightarrow a \leq numeral\ x \wedge n$
by (*metis* (*mono-tags*) *local.numeral-power-eq-of-int-cancel-iff of-int-le-iff*)

lemma *numeral-power-less-of-int-cancel-iff* [*simp*]:
 $numeral\ x \wedge n < of-int\ a \longleftrightarrow numeral\ x \wedge n < a$
by (*metis* (*mono-tags*) *local.of-int-eq-numeral-power-cancel-iff of-int-less-iff*)

lemma *of-int-less-numeral-power-cancel-iff* [*simp*]:
 $of-int\ a < numeral\ x \wedge n \longleftrightarrow a < numeral\ x \wedge n$
by (*metis* (*mono-tags*) *local.of-int-eq-numeral-power-cancel-iff of-int-less-iff*)

lemma *neg-numeral-power-le-of-int-cancel-iff* [*simp*]:
 $(- numeral\ x) \wedge n \leq of-int\ a \longleftrightarrow (- numeral\ x) \wedge n \leq a$
by (*metis* (*mono-tags*) *of-int-le-iff of-int-neg-numeral of-int-power*)

lemma *of-int-le-neg-numeral-power-cancel-iff* [*simp*]:
 $of-int\ a \leq (- numeral\ x) \wedge n \longleftrightarrow a \leq (- numeral\ x) \wedge n$
by (*metis* (*mono-tags*) *of-int-le-iff of-int-neg-numeral of-int-power*)

lemma *neg-numeral-power-less-of-int-cancel-iff* [*simp*]:
 $(- numeral\ x) \wedge n < of-int\ a \longleftrightarrow (- numeral\ x) \wedge n < a$
using *of-int-less-iff*[of $(- numeral\ x) \wedge n\ a$]
by *simp*

lemma *of-int-less-neg-numeral-power-cancel-iff* [*simp*]:
 $of-int\ a < (- numeral\ x) \wedge n \longleftrightarrow a < (- numeral\ x::int) \wedge n$
using *of-int-less-iff*[of $a\ (- numeral\ x) \wedge n$]
by *simp*

lemma *of-int-le-of-int-power-cancel-iff*[*simp*]: $(of-int\ b) \wedge w \leq of-int\ x \longleftrightarrow b \wedge w \leq x$
by (*metis* (*mono-tags*) *of-int-le-iff of-int-power*)

lemma *of-int-power-le-of-int-cancel-iff*[*simp*]: $of-int\ x \leq (of-int\ b) \wedge w \longleftrightarrow x \leq b \wedge w$

by (*metis* (*mono-tags*) *of-int-le-iff* *of-int-power*)

lemma *of-int-less-of-int-power-cancel-iff* [*simp*]: $(\text{of-int } b) \wedge w < \text{of-int } x \longleftrightarrow b \wedge w < x$

by (*metis* (*mono-tags*) *of-int-less-iff* *of-int-power*)

lemma *of-int-power-less-of-int-cancel-iff* [*simp*]: $\text{of-int } x < (\text{of-int } b) \wedge w \longleftrightarrow x < b \wedge w$

by (*metis* (*mono-tags*) *of-int-less-iff* *of-int-power*)

lemma *of-int-max*: $\text{of-int } (\max x y) = \max (\text{of-int } x) (\text{of-int } y)$

by (*auto simp*: *max-def*)

lemma *of-int-min*: $\text{of-int } (\min x y) = \min (\text{of-int } x) (\text{of-int } y)$

by (*auto simp*: *min-def*)

end

context *division-ring*

begin

lemmas *mult-inverse-of-int-commute* =

mult-commute-imp-mult-inverse-commute [*OF mult-of-int-commute*]

end

Comparisons involving *of-int*.

lemma *of-int-eq-numeral-iff* [*iff*]: $\text{of-int } z = (\text{numeral } n :: 'a::\text{ring-char-0}) \longleftrightarrow z = \text{numeral } n$

using *of-int-eq-iff* **by** *fastforce*

lemma *of-int-le-numeral-iff* [*simp*]:

$\text{of-int } z \leq (\text{numeral } n :: 'a::\text{linordered-idom}) \longleftrightarrow z \leq \text{numeral } n$

using *of-int-le-iff* [*of z numeral n*] **by** *simp*

lemma *of-int-numeral-le-iff* [*simp*]:

$(\text{numeral } n :: 'a::\text{linordered-idom}) \leq \text{of-int } z \longleftrightarrow \text{numeral } n \leq z$

using *of-int-le-iff* [*of numeral n*] **by** *simp*

lemma *of-int-less-numeral-iff* [*simp*]:

$\text{of-int } z < (\text{numeral } n :: 'a::\text{linordered-idom}) \longleftrightarrow z < \text{numeral } n$

using *of-int-less-iff* [*of z numeral n*] **by** *simp*

lemma *of-int-numeral-less-iff* [*simp*]:

$(\text{numeral } n :: 'a::\text{linordered-idom}) < \text{of-int } z \longleftrightarrow \text{numeral } n < z$

using *of-int-less-iff* [*of numeral n z*] **by** *simp*

lemma *of-nat-less-of-int-iff*: $(\text{of-nat } n :: 'a::\text{linordered-idom}) < \text{of-int } x \longleftrightarrow \text{int } n < x$

by (*metis of-int-of-nat-eq of-int-less-iff*)

lemma *of-int-eq-id* [*simp*]: *of-int = id*

proof

show *of-int z = id z* for *z*

by (*cases z rule: int-diff-cases*) *simp*

qed

instance *int :: no-top*

proof

fix *x::int*

have $x < x + 1$

by *simp*

then show $\exists y. x < y$

by (*rule exI*)

qed

instance *int :: no-bot*

proof

fix *x::int*

have $x - 1 < x$

by *simp*

then show $\exists y. y < x$

by (*rule exI*)

qed

54.6 Magnitude of an Integer, as a Natural Number: *nat*

lift-definition *nat :: int \Rightarrow nat* is $\lambda(x, y). x - y$

by *auto*

lemma *nat-int* [*simp*]: *nat (int n) = n*

by *transfer simp*

lemma *int-nat-eq* [*simp*]: *int (nat z) = (if $0 \leq z$ then z else 0)*

by *transfer clarsimp*

lemma *nat-0-le*: $0 \leq z \implies \text{int } (\text{nat } z) = z$

by *simp*

lemma *nat-le-0* [*simp*]: $z \leq 0 \implies \text{nat } z = 0$

by *transfer clarsimp*

lemma *nat-le-eq-zle*: $0 < w \vee 0 \leq z \implies \text{nat } w \leq \text{nat } z \longleftrightarrow w \leq z$

by *transfer (clarsimp, arith)*

An alternative condition is $(0::'a) \leq w$.

lemma *nat-mono-iff*: $0 < z \implies \text{nat } w < \text{nat } z \longleftrightarrow w < z$

by (*simp add: nat-le-eq-zle linorder-not-le [symmetric]*)

lemma *nat-less-eq-zless*: $0 \leq w \implies \text{nat } w < \text{nat } z \iff w < z$
by (*simp add: nat-le-eq-zle linorder-not-le [symmetric]*)

lemma *zless-nat-conj* [*simp*]: $\text{nat } w < \text{nat } z \iff 0 < z \wedge w < z$
by *transfer (clarsimp, arith)*

lemma *nonneg-int-cases*:
assumes $0 \leq k$
obtains n **where** $k = \text{int } n$
proof –
from *assms* **have** $k = \text{int } (\text{nat } k)$
by *simp*
then show *thesis*
by (*rule that*)
qed

lemma *pos-int-cases*:
assumes $0 < k$
obtains n **where** $k = \text{int } n$ **and** $n > 0$
proof –
from *assms* **have** $0 \leq k$
by *simp*
then obtain n **where** $k = \text{int } n$
by (*rule nonneg-int-cases*)
moreover have $n > 0$
using $\langle k = \text{int } n \rangle$ *assms* **by** *simp*
ultimately show *thesis*
by (*rule that*)
qed

lemma *nonpos-int-cases*:
assumes $k \leq 0$
obtains n **where** $k = - \text{int } n$
proof –
from *assms* **have** $-k \geq 0$
by *simp*
then obtain n **where** $-k = \text{int } n$
by (*rule nonneg-int-cases*)
then have $k = - \text{int } n$
by *simp*
then show *thesis*
by (*rule that*)
qed

lemma *neg-int-cases*:
assumes $k < 0$
obtains n **where** $k = - \text{int } n$ **and** $n > 0$
proof –

from *assms* **have** $-k > 0$
by *simp*
then obtain n **where** $-k = \text{int } n$ **and** $-k > 0$
by (*blast elim: pos-int-cases*)
then have $k = -\text{int } n$ **and** $n > 0$
by *simp-all*
then show *thesis*
by (*rule that*)
qed

lemma *nat-eq-iff*: $\text{nat } w = m \longleftrightarrow (\text{if } 0 \leq w \text{ then } w = \text{int } m \text{ else } m = 0)$
by *transfer (clarsimp simp add: le-imp-diff-is-add)*

lemma *nat-eq-iff2*: $m = \text{nat } w \longleftrightarrow (\text{if } 0 \leq w \text{ then } w = \text{int } m \text{ else } m = 0)$
using *nat-eq-iff [of w m]* **by** *auto*

lemma *nat-0 [simp]*: $\text{nat } 0 = 0$
by (*simp add: nat-eq-iff*)

lemma *nat-1 [simp]*: $\text{nat } 1 = \text{Suc } 0$
by (*simp add: nat-eq-iff*)

lemma *nat-numeral [simp]*: $\text{nat } (\text{numeral } k) = \text{numeral } k$
by (*simp add: nat-eq-iff*)

lemma *nat-neg-numeral [simp]*: $\text{nat } (-\text{numeral } k) = 0$
by *simp*

lemma *nat-2*: $\text{nat } 2 = \text{Suc } (\text{Suc } 0)$
by *simp*

lemma *nat-less-iff*: $0 \leq w \implies \text{nat } w < m \longleftrightarrow w < \text{of-nat } m$
by *transfer (clarsimp, arith)*

lemma *nat-le-iff*: $\text{nat } x \leq n \longleftrightarrow x \leq \text{int } n$
by *transfer (clarsimp simp add: le-diff-conv)*

lemma *nat-mono*: $x \leq y \implies \text{nat } x \leq \text{nat } y$
by *transfer auto*

lemma *nat-0-iff [simp]*: $\text{nat } i = 0 \longleftrightarrow i \leq 0$
for $i :: \text{int}$
by *transfer clarsimp*

lemma *int-eq-iff*: $\text{of-nat } m = z \longleftrightarrow m = \text{nat } z \wedge 0 \leq z$
by (*auto simp add: nat-eq-iff2*)

lemma *zero-less-nat-eq [simp]*: $0 < \text{nat } z \longleftrightarrow 0 < z$
using *zless-nat-conj [of 0]* **by** *auto*

lemma *nat-add-distrib*: $0 \leq z \implies 0 \leq z' \implies \text{nat } (z + z') = \text{nat } z + \text{nat } z'$
by *transfer clarsimp*

lemma *nat-diff-distrib'*: $0 \leq x \implies 0 \leq y \implies \text{nat } (x - y) = \text{nat } x - \text{nat } y$
by *transfer clarsimp*

lemma *nat-diff-distrib*: $0 \leq z' \implies z' \leq z \implies \text{nat } (z - z') = \text{nat } z - \text{nat } z'$
by (*rule nat-diff-distrib'*) *auto*

lemma *nat-zminus-int* [*simp*]: $\text{nat } (- \text{int } n) = 0$
by *transfer simp*

lemma *le-nat-iff*: $k \geq 0 \implies n \leq \text{nat } k \longleftrightarrow \text{int } n \leq k$
by *transfer auto*

lemma *zless-nat-eq-int-zless*: $m < \text{nat } z \longleftrightarrow \text{int } m < z$
by *transfer (clarsimp simp add: less-diff-conv)*

lemma (*in ring-1*) *of-nat-nat* [*simp*]: $0 \leq z \implies \text{of-nat } (\text{nat } z) = \text{of-int } z$
by *transfer (clarsimp simp add: of-nat-diff)*

lemma *diff-nat-numeral* [*simp*]: $(\text{numeral } v :: \text{nat}) - \text{numeral } v' = \text{nat } (\text{numeral } v - \text{numeral } v')$
by (*simp only: nat-diff-distrib' zero-le-numeral nat-numeral*)

lemma *nat-abs-triangle-ineq*:
 $\text{nat } |k + l| \leq \text{nat } |k| + \text{nat } |l|$
by (*simp add: nat-add-distrib [symmetric] nat-le-eq-zle abs-triangle-ineq*)

lemma *nat-of-bool* [*simp*]:
 $\text{nat } (\text{of-bool } P) = \text{of-bool } P$
by *auto*

lemma *split-nat* [*linarith-split*]: $P (\text{nat } i) \longleftrightarrow ((\forall n. i = \text{int } n \longrightarrow P n) \wedge (i < 0 \longrightarrow P 0))$

(**is** $?P = (?L \wedge ?R)$)

for $i :: \text{int}$

proof (*cases* $i < 0$)

case *True*

then show $?thesis$

by *auto*

next

case *False*

have $?P = ?L$

proof

assume $?P$

then show $?L$ *using False by auto*

next

```

assume ?L
moreover from False have int (nat i) = i
  by (simp add: not-less)
ultimately show ?P
  by simp
qed
with False show ?thesis by simp
qed

```

```

lemma all-nat:  $(\forall x. P x) \longleftrightarrow (\forall x \geq 0. P (\text{nat } x))$ 
  by (auto split: split-nat)

```

```

lemma ex-nat:  $(\exists x. P x) \longleftrightarrow (\exists x. 0 \leq x \wedge P (\text{nat } x))$ 
proof

```

```

  assume  $\exists x. P x$ 
  then obtain x where  $P x$  ..
  then have  $\text{int } x \geq 0 \wedge P (\text{nat } (\text{int } x))$  by simp
  then show  $\exists x \geq 0. P (\text{nat } x)$  ..
next
  assume  $\exists x \geq 0. P (\text{nat } x)$ 
  then show  $\exists x. P x$  by auto
qed

```

For termination proofs:

```

lemma measure-function-int[measure-function]: is-measure (nat o abs) ..

```

54.7 Lemmas about the Function *of-nat* and Orderings

```

lemma negative-zless-0:  $-(\text{int } (\text{Suc } n)) < (0 :: \text{int})$ 
  by (simp add: order-less-le del: of-nat-Suc)

```

```

lemma negative-zless [iff]:  $-(\text{int } (\text{Suc } n)) < \text{int } m$ 
  by (rule negative-zless-0 [THEN order-less-le-trans], simp)

```

```

lemma negative-zle-0:  $-\text{int } n \leq 0$ 
  by (simp add: minus-le-iff)

```

```

lemma negative-zle [iff]:  $-\text{int } n \leq \text{int } m$ 
  by (rule order-trans [OF negative-zle-0 of-nat-0-le-iff])

```

```

lemma not-zle-0-negative [simp]:  $\neg 0 \leq -\text{int } (\text{Suc } n)$ 
  by (subst le-minus-iff) (simp del: of-nat-Suc)

```

```

lemma int-zle-neg:  $\text{int } n \leq -\text{int } m \longleftrightarrow n = 0 \wedge m = 0$ 
  by transfer simp

```

```

lemma not-int-zless-negative [simp]:  $\neg \text{int } n < -\text{int } m$ 
  by (simp add: linorder-not-less)

```

lemma *negative-eq-positive* [*simp*]: $- \text{int } n = \text{of-nat } m \longleftrightarrow n = 0 \wedge m = 0$
by (*force simp add: order-eq-iff [of - of-nat n] int-zle-neg*)

lemma *zle-iff-zadd*: $w \leq z \longleftrightarrow (\exists n. z = w + \text{int } n)$
(is ?lhs \longleftrightarrow ?rhs)

proof

assume *?rhs*
then show *?lhs* **by** *auto*

next

assume *?lhs*
then have $0 \leq z - w$ **by** *simp*
then obtain *n* **where** $z - w = \text{int } n$
using *zero-le-imp-eq-int [of z - w]* **by** *blast*
then have $z = w + \text{int } n$ **by** *simp*
then show *?rhs* **..**

qed

lemma *zadd-int-left*: $\text{int } m + (\text{int } n + z) = \text{int } (m + n) + z$
by *simp*

lemma *negD*:

assumes $x < 0$ **shows** $\exists n. x = - (\text{int } (\text{Suc } n))$

proof $-$

have $\bigwedge a b. a < b \implies \exists n. \text{Suc } (a + n) = b$

proof $-$

fix $a b :: \text{nat}$

assume $a < b$

then have $\text{Suc } (a + (b - \text{Suc } a)) = b$

by *arith*

then show $\exists n. \text{Suc } (a + n) = b$

by (*rule exI*)

qed

with *assms* **show** *?thesis*

by *transfer auto*

qed

54.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

This version is symmetric in the two subgoals.

lemma *int-cases2* [*case-names nonneg nonpos, cases type: int*]:

$(\bigwedge n. z = \text{int } n \implies P) \implies (\bigwedge n. z = - (\text{int } n) \implies P) \implies P$

by (*cases z < 0*) (*auto simp add: linorder-not-less dest!: negD nat-0-le [THEN sym]*)

This is the default, with a negative case.

lemma *int-cases* [*case-names nonneg neg, cases type: int*]:

assumes $pos: \bigwedge n. z = \text{int } n \implies P$ **and** $neg: \bigwedge n. z = -(\text{int } (\text{Suc } n)) \implies P$
shows P
proof (*cases* $z < 0$)
case *True*
with *neg* **show** *?thesis*
by (*blast dest!*: *negD*)
next
case *False*
with *pos* **show** *?thesis*
by (*force simp add: linorder-not-less dest: nat-0-le [THEN sym]*)
qed

lemma *int-cases3* [*case-names zero pos neg*]:
fixes $k :: \text{int}$
assumes $k = 0 \implies P$ **and** $\bigwedge n. k = \text{int } n \implies n > 0 \implies P$
and $\bigwedge n. k = -\text{int } n \implies n > 0 \implies P$
shows P
proof (*cases* $k 0::\text{int}$ *rule: linorder-cases*)
case *equal*
with *assms(1)* **show** P **by** *simp*
next
case *greater*
then have $*$: $\text{nat } k > 0$ **by** *simp*
moreover from $*$ **have** $k = \text{int } (\text{nat } k)$ **by** *auto*
ultimately show P **using** *assms(2)* **by** *blast*
next
case *less*
then have $*$: $\text{nat } (-k) > 0$ **by** *simp*
moreover from $*$ **have** $k = -\text{int } (\text{nat } (-k))$ **by** *auto*
ultimately show P **using** *assms(3)* **by** *blast*
qed

lemma *int-of-nat-induct* [*case-names nonneg neg, induct type: int*]:
 $(\bigwedge n. P (\text{int } n)) \implies (\bigwedge n. P (-\text{int } (\text{Suc } n))) \implies P z$
by (*cases* z) *auto*

lemma *sgn-mult-dvd-iff* [*simp*]:
 $\text{sgn } r * l \text{ dvd } k \iff l \text{ dvd } k \wedge (r = 0 \longrightarrow k = 0)$ **for** $k \ l \ r :: \text{int}$
by (*cases* r *rule: int-cases3*) *auto*

lemma *mult-sgn-dvd-iff* [*simp*]:
 $l * \text{sgn } r \text{ dvd } k \iff l \text{ dvd } k \wedge (r = 0 \longrightarrow k = 0)$ **for** $k \ l \ r :: \text{int}$
using *sgn-mult-dvd-iff* [*of* $r \ l \ k$] **by** (*simp add: ac-simps*)

lemma *dvd-sgn-mult-iff* [*simp*]:
 $l \text{ dvd } \text{sgn } r * k \iff l \text{ dvd } k \vee r = 0$ **for** $k \ l \ r :: \text{int}$
by (*cases* r *rule: int-cases3*) *simp-all*

lemma *dvd-mult-sgn-iff* [*simp*]:

$l \text{ dvd } k * \text{sgn } r \iff l \text{ dvd } k \vee r = 0$ **for** $k \ l \ r :: \text{int}$
using *dvd-sgn-mult-iff* [of $l \ r \ k$] **by** (*simp add: ac-simps*)

lemma *int-sgnE*:
fixes $k :: \text{int}$
obtains n **and** l **where** $k = \text{sgn } l * \text{int } n$
proof –
have $k = \text{sgn } k * \text{int } (\text{nat } |k|)$
by (*simp add: sgn-mult-abs*)
then show *?thesis* ..
qed

54.8.1 Binary comparisons

Preliminaries

lemma *le-imp-0-less*:
fixes $z :: \text{int}$
assumes $le: 0 \leq z$
shows $0 < 1 + z$
proof –
have $0 \leq z$ **by** *fact*
also have $\dots < z + 1$ **by** (*rule less-add-one*)
also have $\dots = 1 + z$ **by** (*simp add: ac-simps*)
finally show $0 < 1 + z$.
qed

lemma *odd-less-0-iff*: $1 + z + z < 0 \iff z < 0$
for $z :: \text{int}$
proof (*cases z*)
case (*nonneg n*)
then show *?thesis*
by (*simp add: linorder-not-less add.assoc add-increasing le-imp-0-less [THEN order-less-imp-le]*)
next
case (*neg n*)
then show *?thesis*
by (*simp del: of-nat-Suc of-nat-add of-nat-1*
add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric])
qed

54.8.2 Comparisons, for Ordered Rings

lemma *odd-nonzero*: $1 + z + z \neq 0$
for $z :: \text{int}$
proof (*cases z*)
case (*nonneg n*)
have $le: 0 \leq z + z$
by (*simp add: nonneg add-increasing*)
then show *?thesis*

```

    using le-imp-0-less [OF le] by (auto simp: ac-simps)
next
case (neg n)
show ?thesis
proof
  assume eq:  $1 + z + z = 0$ 
  have  $0 < 1 + (\text{int } n + \text{int } n)$ 
    by (simp add: le-imp-0-less add-increasing)
  also have  $\dots = - (1 + z + z)$ 
    by (simp add: neg add.assoc [symmetric])
  also have  $\dots = 0$  by (simp add: eq)
  finally have  $0 < 0$  ..
  then show False by blast
qed
qed

```

54.9 The Set of Integers

```

context ring-1
begin

```

```

definition Ints :: 'a set ( $\mathbb{Z}$ )
  where  $\mathbb{Z} = \text{range of-int}$ 

```

```

lemma Ints-of-int [simp]: of-int  $z \in \mathbb{Z}$ 
  by (simp add: Ints-def)

```

```

lemma Ints-of-nat [simp]: of-nat  $n \in \mathbb{Z}$ 
  using Ints-of-int [of of-nat n] by simp

```

```

lemma Ints-0 [simp]:  $0 \in \mathbb{Z}$ 
  using Ints-of-int [of 0] by simp

```

```

lemma Ints-1 [simp]:  $1 \in \mathbb{Z}$ 
  using Ints-of-int [of 1] by simp

```

```

lemma Ints-numeral [simp]: numeral  $n \in \mathbb{Z}$ 
  by (subst of-nat-numeral [symmetric], rule Ints-of-nat)

```

```

lemma Ints-add [simp]:  $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a + b \in \mathbb{Z}$ 
  by (force simp add: Ints-def simp flip: of-int-add intro: range-eqI)

```

```

lemma Ints-minus [simp]:  $a \in \mathbb{Z} \implies -a \in \mathbb{Z}$ 
  by (force simp add: Ints-def simp flip: of-int-minus intro: range-eqI)

```

```

lemma minus-in-Ints-iff:  $-x \in \mathbb{Z} \iff x \in \mathbb{Z}$ 
  using Ints-minus[of x] Ints-minus[of -x] by auto

```

```

lemma Ints-diff [simp]:  $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a - b \in \mathbb{Z}$ 

```


by (force simp add: Ints-def simp flip: of-int-diff intro: range-eqI)

lemma Ints-mult [simp]: $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a * b \in \mathbb{Z}$

by (force simp add: Ints-def simp flip: of-int-mult intro: range-eqI)

lemma Ints-power [simp]: $a \in \mathbb{Z} \implies a \wedge n \in \mathbb{Z}$

by (induct n) simp-all

lemma Ints-cases [cases set: Ints]:

assumes $q \in \mathbb{Z}$

obtains (of-int) z where $q = \text{of-int } z$

unfolding Ints-def

proof –

from $\langle q \in \mathbb{Z} \rangle$ have $q \in \text{range of-int}$ unfolding Ints-def .

then obtain z where $q = \text{of-int } z$..

then show thesis ..

qed

lemma Ints-induct [case-names of-int, induct set: Ints]:

$q \in \mathbb{Z} \implies (\bigwedge z. P (\text{of-int } z)) \implies P q$

by (rule Ints-cases) auto

lemma Nats-subset-Ints: $\mathbb{N} \subseteq \mathbb{Z}$

unfolding Nats-def Ints-def

by (rule subsetI, elim imageE, hypsubst, subst of-int-of-nat-eq[symmetric], rule imageI) simp-all

lemma Nats-altdef1: $\mathbb{N} = \{\text{of-int } n \mid n. n \geq 0\}$

proof (intro subsetI equalityI)

fix $x :: 'a$

assume $x \in \{\text{of-int } n \mid n. n \geq 0\}$

then obtain n where $x = \text{of-int } n$ $n \geq 0$

by (auto elim!: Ints-cases)

then have $x = \text{of-nat } (\text{nat } n)$

by (subst of-nat-nat) simp-all

then show $x \in \mathbb{N}$

by simp

next

fix $x :: 'a$

assume $x \in \mathbb{N}$

then obtain n where $x = \text{of-nat } n$

by (auto elim!: Nats-cases)

then have $x = \text{of-int } (\text{int } n)$ by simp

also have $\text{int } n \geq 0$ by simp

then have $\text{of-int } (\text{int } n) \in \{\text{of-int } n \mid n. n \geq 0\}$ by blast

finally show $x \in \{\text{of-int } n \mid n. n \geq 0\}$.

qed

end

lemma *Ints-sum* [*intro*]: $(\bigwedge x. x \in A \implies f x \in \mathbb{Z}) \implies \text{sum } f A \in \mathbb{Z}$
by (*induction A rule: infinite-finite-induct*) *auto*

lemma *Ints-prod* [*intro*]: $(\bigwedge x. x \in A \implies f x \in \mathbb{Z}) \implies \text{prod } f A \in \mathbb{Z}$
by (*induction A rule: infinite-finite-induct*) *auto*

lemma (*in linordered-idom*) *Ints-abs* [*simp*]:
shows $a \in \mathbb{Z} \implies \text{abs } a \in \mathbb{Z}$
by (*auto simp: abs-if*)

lemma (*in linordered-idom*) *Nats-altdef2*: $\mathbb{N} = \{n \in \mathbb{Z}. n \geq 0\}$

proof (*intro subsetI equalityI*)

fix $x :: 'a$

assume $x \in \{n \in \mathbb{Z}. n \geq 0\}$

then obtain n **where** $x = \text{of-int } n$ $n \geq 0$

by (*auto elim!: Ints-cases*)

then have $x = \text{of-nat } (\text{nat } n)$

by (*subst of-nat-nat*) *simp-all*

then show $x \in \mathbb{N}$

by *simp*

qed (*auto elim!: Nats-cases*)

lemma (*in idom-divide*) *of-int-divide-in-Ints*:

of-int a div of-int b $\in \mathbb{Z}$ **if** $b \text{ dvd } a$

proof –

from that obtain c **where** $a = b * c$..

then show *?thesis*

by (*cases of-int b = 0*) *simp-all*

qed

The premise involving \mathbb{Z} prevents $a = (1::'a) / (2::'a)$.

lemma *Ints-double-eq-0-iff*:

fixes $a :: 'a::\text{ring-char-0}$

assumes *in-Ints*: $a \in \mathbb{Z}$

shows $a + a = 0 \iff a = 0$

(*is ?lhs* \iff *?rhs*)

proof –

from in-Ints have $a \in \text{range of-int}$

unfolding *Ints-def* [*symmetric*] .

then obtain z **where** $a = \text{of-int } z$..

show *?thesis*

proof

assume *?rhs*

then show *?lhs* **by** *simp*

next

assume *?lhs*

with a have $\text{of-int } (z + z) = (\text{of-int } 0 :: 'a)$ **by** *simp*

then have $z + z = 0$ **by** (*simp only: of-int-eq-iff*)

then have $z = 0$ by (simp only: double-zero)
 with a show ?rhs by simp
 qed
 qed

lemma *Ints-odd-nonzero*:

fixes $a :: 'a::ring-char-0$

assumes *in-Ints*: $a \in \mathbb{Z}$

shows $1 + a + a \neq 0$

proof –

from *in-Ints* have $a \in \text{range of-int}$

unfolding *Ints-def* [*symmetric*].

then obtain z where $a: a = \text{of-int } z ..$

show ?thesis

proof

assume $1 + a + a = 0$

with a have $\text{of-int } (1 + z + z) = (\text{of-int } 0 :: 'a)$ by simp

then have $1 + z + z = 0$ by (simp only: *of-int-eq-iff*)

with *odd-nonzero* show False by blast

qed

qed

lemma *Nats-numeral* [*simp*]: numeral $w \in \mathbb{N}$

using *of-nat-in-Nats* [*of numeral w*] by simp

lemma *Ints-odd-less-0*:

fixes $a :: 'a::linordered-idom$

assumes *in-Ints*: $a \in \mathbb{Z}$

shows $1 + a + a < 0 \iff a < 0$

proof –

from *in-Ints* have $a \in \text{range of-int}$

unfolding *Ints-def* [*symmetric*].

then obtain z where $a: a = \text{of-int } z ..$

with a have $1 + a + a < 0 \iff \text{of-int } (1 + z + z) < (\text{of-int } 0 :: 'a)$

by simp

also have $\dots \iff z < 0$

by (simp only: *of-int-less-iff odd-less-0-iff*)

also have $\dots \iff a < 0$

by (simp add: a)

finally show ?thesis .

qed

54.10 sum and prod

context *semiring-1*

begin

lemma *of-nat-sum* [*simp*]:

$\text{of-nat } (\text{sum } f A) = (\sum_{x \in A} \text{of-nat } (f x))$

by (induction A rule: infinite-finite-induct) auto

end

context ring-1

begin

lemma of-int-sum [simp]:

of-int (sum f A) = ($\sum_{x \in A} \text{of-int } (f x)$)

by (induction A rule: infinite-finite-induct) auto

end

context comm-semiring-1

begin

lemma of-nat-prod [simp]:

of-nat (prod f A) = ($\prod_{x \in A} \text{of-nat } (f x)$)

by (induction A rule: infinite-finite-induct) auto

end

context comm-ring-1

begin

lemma of-int-prod [simp]:

of-int (prod f A) = ($\prod_{x \in A} \text{of-int } (f x)$)

by (induction A rule: infinite-finite-induct) auto

end

54.11 Setting up simplification procedures

ML-file $\langle \text{Tools/int-arith.ML} \rangle$

declaration $\langle K$ (

Lin-Arith.add-discrete-type **type-name** $\langle \text{Int.int} \rangle$

#> *Lin-Arith.add-lessD* @{thm zless-imp-add1-zle}

#> *Lin-Arith.add-inj-thms* @{thms of-nat-le-iff [THEN iffD2] of-nat-eq-iff [THEN iffD2]}

#> *Lin-Arith.add-inj-const* (**const-name** $\langle \text{of-nat} \rangle$, **typ** $\langle \text{nat} \Rightarrow \text{int} \rangle$)

#> *Lin-Arith.add-simps*

@{thms of-int-0 of-int-1 of-int-add of-int-mult of-int-numeral of-int-neg-numeral
nat-0 nat-1 diff-nat-numeral nat-numeral

neg-less-iff-less

True-implies-equals

distrib-left [where a = numeral v for v]

distrib-left [where a = - numeral v for v]

div-by-1 div-0

```

    times-divide-eq-right times-divide-eq-left
    minus-divide-left [THEN sym] minus-divide-right [THEN sym]
    add-divide-distrib diff-divide-distrib
    of-int-minus of-int-diff
    of-int-of-nat-eq}
  #> Lin-Arith.add-simprocs [Int-Arith.zero-one-idom-simproc]
)>

```

```

simproc-setup fast-arith
  ((m::'a::linordered-idom) < n |
   (m::'a::linordered-idom) ≤ n |
   (m::'a::linordered-idom) = n) =
  ⟨K Lin-Arith.simproc⟩

```

54.12 More Inequality Reasoning

```

lemma zless-add1-eq:  $w < z + 1 \longleftrightarrow w < z \vee w = z$ 
for  $w z :: \text{int}$ 
by arith

```

```

lemma add1-zle-eq:  $w + 1 \leq z \longleftrightarrow w < z$ 
for  $w z :: \text{int}$ 
by arith

```

```

lemma zle-diff1-eq [simp]:  $w \leq z - 1 \longleftrightarrow w < z$ 
for  $w z :: \text{int}$ 
by arith

```

```

lemma zle-add1-eq-le [simp]:  $w < z + 1 \longleftrightarrow w \leq z$ 
for  $w z :: \text{int}$ 
by arith

```

```

lemma int-one-le-iff-zero-less:  $1 \leq z \longleftrightarrow 0 < z$ 
for  $z :: \text{int}$ 
by arith

```

```

lemma Ints-nonzero-abs-ge1:
fixes  $x::'a :: \text{linordered-idom}$ 
assumes  $x \in \text{Ints } x \neq 0$ 
shows  $1 \leq \text{abs } x$ 

```

```

proof (rule Ints-cases [OF ⟨ $x \in \text{Ints}$ ⟩])
fix  $z::\text{int}$ 
assume  $x = \text{of-int } z$ 
with ⟨ $x \neq 0$ ⟩
show  $1 \leq |x|$ 
apply (auto simp: abs-iff)
by (metis diff-0 of-int-1 of-int-le-iff of-int-minus zle-diff1-eq)
qed

```

lemma *Ints-nonzero-abs-less1*:
fixes $x :: 'a :: \text{linordered-idom}$
shows $\llbracket x \in \text{Ints}; \text{abs } x < 1 \rrbracket \implies x = 0$
using *Ints-nonzero-abs-ge1* [of x] **by** *auto*

lemma *Ints-eq-abs-less1*:
fixes $x :: 'a :: \text{linordered-idom}$
shows $\llbracket x \in \text{Ints}; y \in \text{Ints} \rrbracket \implies x = y \longleftrightarrow \text{abs } (x - y) < 1$
using *eq-iff-diff-eq-0* **by** (*fastforce intro: Ints-nonzero-abs-less1*)

54.13 The functions *nat* and *int*

Simplify the term $w + - z$.

lemma *one-less-nat-eq* [*simp*]: $\text{Suc } 0 < \text{nat } z \longleftrightarrow 1 < z$
using *zless-nat-conj* [of $1 z$] **by** *auto*

lemma *int-eq-iff-numeral* [*simp*]:
 $\text{int } m = \text{numeral } v \longleftrightarrow m = \text{numeral } v$
by (*simp add: int-eq-iff*)

lemma *nat-abs-int-diff*:
 $\text{nat } |\text{int } a - \text{int } b| = (\text{if } a \leq b \text{ then } b - a \text{ else } a - b)$
by *auto*

lemma *nat-int-add*: $\text{nat } (\text{int } a + \text{int } b) = a + b$
by *auto*

context *ring-1*
begin

lemma *of-int-of-nat* [*nitpick-simp*]:
 $\text{of-int } k = (\text{if } k < 0 \text{ then } - \text{of-nat } (\text{nat } (- k)) \text{ else } \text{of-nat } (\text{nat } k))$
proof (*cases k < 0*)
case *True*
then have $0 \leq - k$ **by** *simp*
then have $\text{of-nat } (\text{nat } (- k)) = \text{of-int } (- k)$ **by** (*rule of-nat-nat*)
with *True* **show** *?thesis* **by** *simp*

next
case *False*
then show *?thesis* **by** (*simp add: not-less*)
qed

end

lemma *transfer-rule-of-int*:
includes *lifting-syntax*
fixes $R :: 'a::\text{ring-1} \Rightarrow 'b::\text{ring-1} \Rightarrow \text{bool}$
assumes [*transfer-rule*]: $R \ 0 \ 0 \ R \ 1 \ 1$
 $(R \ ==\ ==\ > \ R \ ==\ ==\ > \ R) \ (+) \ (+)$

$(R \implies R)$ *uminus uminus*
shows $((=) \implies R)$ *of-int of-int*
proof –
note *assms*
note *transfer-rule-of-nat [transfer-rule]*
have *[transfer-rule]: ((=) $\implies R$) of-nat of-nat*
by *transfer-prover*
show *?thesis*
by *(unfold of-int-of-nat [abs-def]) transfer-prover*
qed

lemma *nat-mult-distrib:*

fixes $z z' :: \text{int}$
assumes $0 \leq z$
shows $\text{nat } (z * z') = \text{nat } z * \text{nat } z'$
proof *(cases $0 \leq z'$)*
case *False*
with *assms* **have** $z * z' \leq 0$
by *(simp add: not-le mult-le-0-iff)*
then have $\text{nat } (z * z') = 0$ **by** *simp*
moreover from *False* **have** $\text{nat } z' = 0$ **by** *simp*
ultimately show *?thesis* **by** *simp*
next
case *True*
with *assms* **have** $ge-0: z * z' \geq 0$ **by** *(simp add: zero-le-mult-iff)*
show *?thesis*
by *(rule injD [of of-nat :: nat \Rightarrow int, OF inj-of-nat])*
(simp only: of-nat-mult of-nat-nat [OF True]
of-nat-nat [OF assms] of-nat-nat [OF ge-0], simp)
qed

lemma *nat-mult-distrib-neg:*

assumes $z \leq (0::\text{int})$ **shows** $\text{nat } (z * z') = \text{nat } (-z) * \text{nat } (-z')$ **(is ?L = ?R)**
proof –
have $?L = \text{nat } (-z * -z')$
using *assms* **by** *auto*
also have $\dots = ?R$
by *(rule nat-mult-distrib) (use assms in auto)*
finally show *?thesis* .
qed

lemma *nat-abs-mult-distrib:* $\text{nat } |w * z| = \text{nat } |w| * \text{nat } |z|$

by *(cases $z = 0 \vee w = 0$)*
(auto simp add: abs-if nat-mult-distrib [symmetric]
nat-mult-distrib-neg [symmetric] mult-less-0-iff)

lemma *int-in-range-abs [simp]:* $\text{int } n \in \text{range abs}$

proof *(rule range-eqI)*
show $\text{int } n = |\text{int } n|$ **by** *simp*

qed

lemma *range-abs-Nats* [*simp*]: $\text{range abs} = (\mathbb{N} :: \text{int set})$

proof –

have $|k| \in \mathbb{N}$ **for** $k :: \text{int}$

by (*cases k*) *simp-all*

moreover have $k \in \text{range abs}$ **if** $k \in \mathbb{N}$ **for** $k :: \text{int}$

using that by *induct simp*

ultimately show *?thesis* **by** *blast*

qed

lemma *Suc-nat-eq-nat-zadd1*: $0 \leq z \implies \text{Suc} (\text{nat } z) = \text{nat} (1 + z)$

for $z :: \text{int}$

by (*rule sym*) (*simp add: nat-eq-iff*)

lemma *diff-nat-eq-if*:

$\text{nat } z - \text{nat } z' =$

 (*if* $z' < 0$ *then* $\text{nat } z$

else

$\text{let } d = z - z'$

in if $d < 0$ *then* 0 *else* $\text{nat } d$)

by (*simp add: Let-def nat-diff-distrib [symmetric]*)

lemma *nat-numeral-diff-1* [*simp*]: $\text{numeral } v - (1 :: \text{nat}) = \text{nat} (\text{numeral } v - 1)$

using *diff-nat-numeral [of v Num.One]* **by** *simp*

54.14 Induction principles for int

Well-founded segments of the integers.

definition *int-ge-less-than* :: $\text{int} \Rightarrow (\text{int} \times \text{int}) \text{ set}$

where *int-ge-less-than* $d = \{(z', z). d \leq z' \wedge z' < z\}$

lemma *wf-int-ge-less-than*: *wf* (*int-ge-less-than* d)

proof –

have *int-ge-less-than* $d \subseteq \text{measure} (\lambda z. \text{nat} (z - d))$

by (*auto simp add: int-ge-less-than-def*)

then show *?thesis*

by (*rule wf-subset [OF wf-measure]*)

qed

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition *int-ge-less-than2* :: $\text{int} \Rightarrow (\text{int} \times \text{int}) \text{ set}$

where *int-ge-less-than2* $d = \{(z', z). d \leq z \wedge z' < z\}$

lemma *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2* d)

proof –

have *int-ge-less-than2* $d \subseteq \text{measure} (\lambda z. \text{nat} (1 + z - d))$


```

  by (auto simp add: int-ge-less-than2-def)
  then show ?thesis
  by (rule wf-subset [OF wf-measure])
qed

```

```

theorem int-ge-induct [case-names base step, induct set: int]:
  fixes i :: int
  assumes ge:  $k \leq i$ 
  and base:  $P k$ 
  and step:  $\bigwedge i. k \leq i \implies P i \implies P (i + 1)$ 
  shows  $P i$ 
proof -
  have  $\bigwedge i::int. n = \text{nat } (i - k) \implies k \leq i \implies P i$  for  $n$ 
  proof (induct  $n$ )
  case 0
  then have  $i = k$  by arith
  with base show  $P i$  by simp
  next
  case (Suc  $n$ )
  then have  $n = \text{nat } ((i - 1) - k)$  by arith
  moreover have  $k: k \leq i - 1$  using Suc.prem by arith
  ultimately have  $P (i - 1)$  by (rule Suc.hyps)
  from step [OF  $k$  this] show ?case by simp
  qed
  with ge show ?thesis by fast
qed

```

```

theorem int-gr-induct [case-names base step, induct set: int]:
  fixes i k :: int
  assumes  $k < i$   $P (k + 1)$   $\bigwedge i. k < i \implies P i \implies P (i + 1)$ 
  shows  $P i$ 
proof -
  have  $k+1 \leq i$ 
  using assms by auto
  then show ?thesis
  by (induction i rule: int-ge-induct) (auto simp: assms)
qed

```

```

theorem int-le-induct [consumes 1, case-names base step]:
  fixes i k :: int
  assumes le:  $i \leq k$ 
  and base:  $P k$ 
  and step:  $\bigwedge i. i \leq k \implies P i \implies P (i - 1)$ 
  shows  $P i$ 
proof -
  have  $\bigwedge i::int. n = \text{nat}(k-i) \implies i \leq k \implies P i$  for  $n$ 
  proof (induct  $n$ )

```

```

    case 0
    then have  $i = k$  by arith
    with base show  $P i$  by simp
  next
    case (Suc n)
    then have  $n = \text{nat } (k - (i + 1))$  by arith
    moreover have  $k: i + 1 \leq k$  using Suc.prem1 by arith
    ultimately have  $P (i + 1)$  by (rule Suc.hyps)
    from step[OF k this] show ?case by simp
  qed
  with le show ?thesis by fast
qed

```

```

theorem int-less-induct [consumes 1, case-names base step]:
  fixes  $i k :: \text{int}$ 
  assumes  $i < k \wedge P (k - 1) \wedge i. i < k \implies P i \implies P (i - 1)$ 
  shows  $P i$ 
proof -
  have  $i \leq k - 1$ 
  using assms by auto
  then show ?thesis
  by (induction i rule: int-le-induct) (auto simp: assms)
qed

```

```

theorem int-induct [case-names base step1 step2]:
  fixes  $k :: \text{int}$ 
  assumes base:  $P k$ 
  and step1:  $\wedge i. k \leq i \implies P i \implies P (i + 1)$ 
  and step2:  $\wedge i. k \geq i \implies P i \implies P (i - 1)$ 
  shows  $P i$ 
proof -
  have  $i \leq k \vee i \geq k$  by arith
  then show ?thesis
  proof
    assume  $i \geq k$ 
    then show ?thesis
    using base by (rule int-ge-induct) (fact step1)
  next
    assume  $i \leq k$ 
    then show ?thesis
    using base by (rule int-le-induct) (fact step2)
  qed
qed

```

54.15 Intermediate value theorems

lemma nat-ivt-aux:

```


$$[\forall i < n. |f (\text{Suc } i) - f i| \leq 1; f 0 \leq k; k \leq f n] \implies \exists i \leq n. f i = k$$

  for  $m n :: \text{nat}$  and  $k :: \text{int}$ 

```

```

proof (induct n)
  case (Suc n)
  show ?case
  proof (cases k = f (Suc n))
    case False
    with Suc have  $k \leq f\ n$ 
    by auto
    with Suc show ?thesis
    by (auto simp add: abs-if split: if-split-asm intro: le-SucI)
  qed (use Suc in auto)
qed auto

```

```

lemma nat-intermed-int-val:
  fixes m n :: nat and k :: int
  assumes  $\forall i. m \leq i \wedge i < n \longrightarrow |f\ (Suc\ i) - f\ i| \leq 1$ 
   $m \leq n$ 
   $f\ m \leq k$ 
   $k \leq f\ n$ 
  shows  $\exists i. m \leq i \wedge i \leq n \wedge f\ i = k$ 
proof -
  obtain i where  $i \leq n - m$ 
   $k = f\ (m + i)$ 
  using nat-ivt-aux [of n - m f o plus m k] assms by auto
  with assms show ?thesis
  using exI[of - m + i] by auto
qed

```

```

lemma nat0-intermed-int-val:
   $\exists i \leq n. f\ i = k$ 
  if  $\forall i < n. |f\ (i + 1) - f\ i| \leq 1$ 
   $f\ 0 \leq k$ 
   $k \leq f\ n$ 
  for n :: nat and k :: int
  using nat-intermed-int-val [of 0 n f k] that by auto

```

54.16 Products and 1, by T. M. Rasmussen

```

lemma abs-zmult-eq-1:
  fixes m n :: int
  assumes mn:  $|m * n| = 1$ 
  shows  $|m| = 1$ 
proof -
  from mn have 0:  $m \neq 0$ 
   $n \neq 0$  by auto
  have  $2 \leq |m|$ 
  proof
    assume  $2 \leq |m|$ 
    then have  $2 * |n| \leq |m| * |n|$  by (simp add: mult-mono 0)
    also have  $\dots = |m * n|$  by (simp add: abs-mult)
    also from mn have  $\dots = 1$  by simp
    finally have  $2 * |n| \leq 1$  .
    with 0 show False by arith
  qed
  with 0 show ?thesis by auto
qed

```

lemma *pos-zmult-eq-1-iff-lemma*: $m * n = 1 \implies m = 1 \vee m = -1$
for $m\ n :: \text{int}$
using *abs-zmult-eq-1* [of $m\ n$] **by** *arith*

lemma *pos-zmult-eq-1-iff*:
fixes $m\ n :: \text{int}$
assumes $0 < m$
shows $m * n = 1 \longleftrightarrow m = 1 \wedge n = 1$
proof –
from *assms* **have** $m * n = 1 \implies m = 1$
by (*auto dest: pos-zmult-eq-1-iff-lemma*)
then show *?thesis*
by (*auto dest: pos-zmult-eq-1-iff-lemma*)
qed

lemma *zmult-eq-1-iff*: $m * n = 1 \longleftrightarrow (m = 1 \wedge n = 1) \vee (m = -1 \wedge n = -1)$ (**is** *?L = ?R*)
for $m\ n :: \text{int}$
proof
assume L : *?L* **show** *?R*
using *pos-zmult-eq-1-iff-lemma* [OF L] L **by** *force*
qed *auto*

lemma *infinite-UNIV-int* [*simp*]: $\neg \text{finite } (\text{UNIV} :: \text{int set})$
proof
assume *finite* ($\text{UNIV} :: \text{int set}$)
moreover have *inj* ($\lambda i :: \text{int}. 2 * i$)
by (*rule injI*) *simp*
ultimately have *surj* ($\lambda i :: \text{int}. 2 * i$)
by (*rule finite-UNIV-inj-surj*)
then obtain $i :: \text{int}$ **where** $1 = 2 * i$ **by** (*rule surjE*)
then show *False* **by** (*simp add: pos-zmult-eq-1-iff*)
qed

54.17 The divides relation

lemma *zdvd-antisym-nonneg*: $0 \leq m \implies 0 \leq n \implies m \text{ dvd } n \implies n \text{ dvd } m \implies m = n$
for $m\ n :: \text{int}$
by (*auto simp add: dvd-def mult.assoc zero-le-mult-iff zmult-eq-1-iff*)

lemma *zdvd-antisym-abs*:
fixes $a\ b :: \text{int}$
assumes $a \text{ dvd } b$ **and** $b \text{ dvd } a$
shows $|a| = |b|$
proof (*cases a = 0*)
case *True*
with *assms* **show** *?thesis* **by** *simp*
next

```

case False
from ⟨a dvd b⟩ obtain k where k: b = a * k
  unfolding dvd-def by blast
from ⟨b dvd a⟩ obtain k' where k': a = b * k'
  unfolding dvd-def by blast
from k k' have a = a * k * k' by simp
with mult-cancel-left1 [where c=a and b=k*k'] have kk': k * k' = 1
  using ⟨a ≠ 0⟩ by (simp add: mult.assoc)
then have k = 1 ∧ k' = 1 ∨ k = -1 ∧ k' = -1
  by (simp add: zmult-eq-1-iff)
with k k' show ?thesis by auto
qed

```

```

lemma zdvd-zdiffD: k dvd m - n ⇒ k dvd n ⇒ k dvd m
  for k m n :: int
  using dvd-add-right-iff [of k - n m] by simp

```

```

lemma zdvd-reduce: k dvd n + k * m ⇔ k dvd n
  for k m n :: int
  using dvd-add-times-triv-right-iff [of k n m] by (simp add: ac-simps)

```

```

lemma dvd-imp-le-int:
  fixes d i :: int
  assumes i ≠ 0 and d dvd i
  shows |d| ≤ |i|
proof -
  from ⟨d dvd i⟩ obtain k where i = d * k ..
  with ⟨i ≠ 0⟩ have k ≠ 0 by auto
  then have 1 ≤ |k| and 0 ≤ |d| by auto
  then have |d| * 1 ≤ |d| * |k| by (rule mult-left-mono)
  with ⟨i = d * k⟩ show ?thesis by (simp add: abs-mult)
qed

```

```

lemma zdvd-not-zless:
  fixes m n :: int
  assumes 0 < m and m < n
  shows ¬ n dvd m
proof
  from assms have 0 < n by auto
  assume n dvd m then obtain k where k: m = n * k ..
  with ⟨0 < m⟩ have 0 < n * k by auto
  with ⟨0 < n⟩ have 0 < k by (simp add: zero-less-mult-iff)
  with k ⟨0 < n⟩ ⟨m < n⟩ have n * k < n * 1 by simp
  with ⟨0 < n⟩ ⟨0 < k⟩ show False unfolding mult-less-cancel-left by auto
qed

```

```

lemma zdvd-mult-cancel:
  fixes k m n :: int
  assumes d: k * m dvd k * n

```

```

    and  $k \neq 0$ 
  shows  $m \text{ dvd } n$ 
proof -
  from  $d$  obtain  $h$  where  $h: k * n = k * m * h$ 
    unfolding dvd-def by blast
  have  $n = m * h$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    with  $\langle k \neq 0 \rangle$  have  $k * n \neq k * (m * h)$  by simp
    with  $h$  show False
      by (simp add: mult.assoc)
    qed
  then show ?thesis by simp
qed

lemma int-dvd-int-iff [simp]:
   $int\ m \text{ dvd } int\ n \longleftrightarrow m \text{ dvd } n$ 
proof -
  have  $m \text{ dvd } n$  if  $int\ n = int\ m * k$  for  $k$ 
  proof (cases  $k$ )
    case (nonneg  $q$ )
      with that have  $n = m * q$ 
        by (simp del: of-nat-mult add: of-nat-mult [symmetric])
      then show ?thesis ..
    next
      case (neg  $q$ )
      with that have  $int\ n = int\ m * (- int\ (Suc\ q))$ 
        by simp
      also have  $\dots = - (int\ m * int\ (Suc\ q))$ 
        by (simp only: mult-minus-right)
      also have  $\dots = - int\ (m * Suc\ q)$ 
        by (simp only: of-nat-mult [symmetric])
      finally have  $- int\ (m * Suc\ q) = int\ n$  ..
      then show ?thesis
        by (simp only: negative-eq-positive) auto
    qed
  then show ?thesis by (auto simp add: dvd-def)
qed

lemma dvd-nat-abs-iff [simp]:
   $n \text{ dvd } nat\ |k| \longleftrightarrow int\ n \text{ dvd } k$ 
proof -
  have  $n \text{ dvd } nat\ |k| \longleftrightarrow int\ n \text{ dvd } int\ (nat\ |k|)$ 
    by (simp only: int-dvd-int-iff)
  then show ?thesis
    by simp
  qed

lemma nat-abs-dvd-iff [simp]:

```

```

  nat |k| dvd n  $\longleftrightarrow$  k dvd int n
proof –
  have nat |k| dvd n  $\longleftrightarrow$  int (nat |k|) dvd int n
    by (simp only: int-dvd-int-iff)
  then show ?thesis
    by simp
qed

lemma zdvd1-eq [simp]: x dvd 1  $\longleftrightarrow$  |x| = 1 (is ?lhs  $\longleftrightarrow$  ?rhs)
  for x :: int
proof
  assume ?lhs
  then have nat |x| dvd nat |1|
    by (simp only: nat-abs-dvd-iff) simp
  then have nat |x| = 1
    by simp
  then show ?rhs
    by (cases x < 0) simp-all
next
  assume ?rhs
  then have x = 1  $\vee$  x = - 1
    by auto
  then show ?lhs
    by (auto intro: dvdI)
qed

lemma zdvd-mult-cancel1:
  fixes m :: int
  assumes mp: m  $\neq$  0
  shows m * n dvd m  $\longleftrightarrow$  |n| = 1
    (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs
  then show ?lhs
    by (cases n > 0) (auto simp add: minus-equation-iff)
next
  assume ?lhs
  then have m * n dvd m * 1 by simp
  from zdvd-mult-cancel[OF this mp] show ?rhs
    by (simp only: zdvd1-eq)
qed

lemma nat-dvd-iff: nat z dvd m  $\longleftrightarrow$  (if 0  $\leq$  z then z dvd int m else m = 0)
  using nat-abs-dvd-iff [of z m] by (cases z  $\geq$  0) auto

lemma eq-nat-nat-iff: 0  $\leq$  z  $\implies$  0  $\leq$  z'  $\implies$  nat z = nat z'  $\longleftrightarrow$  z = z'
  by (auto elim: nonneg-int-cases)

lemma nat-power-eq: 0  $\leq$  z  $\implies$  nat (z ^ n) = nat z ^ n

```

by (induct n) (simp-all add: nat-mult-distrib)

lemma numeral-power-eq-nat-cancel-iff [simp]:
 $\text{numeral } x \wedge n = \text{nat } y \longleftrightarrow \text{numeral } x \wedge n = y$
 using nat-eq-iff2 by auto

lemma nat-eq-numeral-power-cancel-iff [simp]:
 $\text{nat } y = \text{numeral } x \wedge n \longleftrightarrow y = \text{numeral } x \wedge n$
 using numeral-power-eq-nat-cancel-iff[of x n y]
 by (metis (mono-tags))

lemma numeral-power-le-nat-cancel-iff [simp]:
 $\text{numeral } x \wedge n \leq \text{nat } a \longleftrightarrow \text{numeral } x \wedge n \leq a$
 using nat-le-eq-zle[of numeral x ^ n a]
 by (auto simp: nat-power-eq)

lemma nat-le-numeral-power-cancel-iff [simp]:
 $\text{nat } a \leq \text{numeral } x \wedge n \longleftrightarrow a \leq \text{numeral } x \wedge n$
 by (simp add: nat-le-iff)

lemma numeral-power-less-nat-cancel-iff [simp]:
 $\text{numeral } x \wedge n < \text{nat } a \longleftrightarrow \text{numeral } x \wedge n < a$
 using nat-less-eq-zless[of numeral x ^ n a]
 by (auto simp: nat-power-eq)

lemma nat-less-numeral-power-cancel-iff [simp]:
 $\text{nat } a < \text{numeral } x \wedge n \longleftrightarrow a < \text{numeral } x \wedge n$
 using nat-less-eq-zless[of a numeral x ^ n]
 by (cases a < 0) (auto simp: nat-power-eq less-le-trans[where y=0])

lemma zdvd-imp-le: $z \leq n$ if $z \text{ dvd } n$ $0 < n$ for $n z :: \text{int}$
proof (cases n)
 case (nonneg n)
 show ?thesis
 by (cases z) (use nonneg dvd-imp-le that in auto)
qed (use that in auto)

lemma zdvd-period:
 fixes a d :: int
 assumes a dvd d
 shows a dvd (x + t) \longleftrightarrow a dvd ((x + c * d) + t)
 (is ?lhs \longleftrightarrow ?rhs)
proof –
 from assms have a dvd (x + t) \longleftrightarrow a dvd ((x + t) + c * d)
 by (simp add: dvd-add-left-iff)
 then show ?thesis
 by (simp add: ac-simps)
qed

54.18 Powers with integer exponents

The following allows writing powers with an integer exponent. While the type signature is very generic, most theorems will assume that the underlying type is a division ring or a field.

The notation ‘powi’ is inspired by the ‘powr’ notation for real/complex exponentiation.

definition *power-int* :: 'a :: {inverse, power} ⇒ int ⇒ 'a (**infixr** powi 80) **where**
power-int x n = (if n ≥ 0 then x ^ nat n else inverse x ^ (nat (-n)))

lemma *power-int-0-right* [simp]: *power-int* x 0 = 1
and *power-int-1-right* [simp]:
power-int (y :: 'a :: {power, inverse, monoid-mult}) 1 = y
and *power-int-minus1-right* [simp]:
power-int (y :: 'a :: {power, inverse, monoid-mult}) (-1) = inverse y
by (simp-all add: *power-int-def*)

lemma *power-int-of-nat* [simp]: *power-int* x (int n) = x ^ n
by (simp add: *power-int-def*)

lemma *power-int-numeral* [simp]: *power-int* x (numeral n) = x ^ numeral n
by (simp add: *power-int-def*)

lemma *int-cases4* [case-names nonneg neg]:
fixes m :: int
obtains n **where** m = int n | n **where** n > 0 m = -int n
proof (cases m ≥ 0)
case True
thus ?thesis **using** that(1)[of nat m] **by** auto
next
case False
thus ?thesis **using** that(2)[of nat (-m)] **by** auto
qed

context
assumes SORT-CONSTRAINT('a::division-ring)
begin

lemma *power-int-minus*: *power-int* (x::'a) (-n) = inverse (*power-int* x n)
by (auto simp: *power-int-def* *power-inverse*)

lemma *power-int-minus-divide*: *power-int* (x::'a) (-n) = 1 / (*power-int* x n)
by (simp add: *divide-inverse* *power-int-minus*)

lemma *power-int-eq-0-iff* [simp]: *power-int* (x::'a) n = 0 ⇔ x = 0 ∧ n ≠ 0
by (auto simp: *power-int-def*)

lemma *power-int-0-left-If*: $\text{power-int } (0 :: 'a) m = (\text{if } m = 0 \text{ then } 1 \text{ else } 0)$
by (*auto simp: power-int-def*)

lemma *power-int-0-left [simp]*: $m \neq 0 \implies \text{power-int } (0 :: 'a) m = 0$
by (*simp add: power-int-0-left-If*)

lemma *power-int-1-left [simp]*: $\text{power-int } 1 n = (1 :: 'a :: \text{division-ring})$
by (*auto simp: power-int-def*)

lemma *power-diff-conv-inverse*: $x \neq 0 \implies m \leq n \implies (x :: 'a) ^ (n - m) = x ^ n * \text{inverse } x ^ m$
by (*simp add: field-simps flip: power-add*)

lemma *power-mult-inverse-distrib*: $x ^ m * \text{inverse } (x :: 'a) = \text{inverse } x * x ^ m$

proof (*cases x = 0*)

case [*simp*]: *False*

show *?thesis*

proof (*cases m*)

case (*Suc m'*)

have $x ^ \text{Suc } m' * \text{inverse } x = x ^ m'$

by (*subst power-Suc2*) (*auto simp: mult.assoc*)

also have $\dots = \text{inverse } x * x ^ \text{Suc } m'$

by (*subst power-Suc*) (*auto simp: mult.assoc [symmetric]*)

finally show *?thesis using Suc by simp*

qed *auto*

qed *auto*

lemma *power-mult-power-inverse-commute*:

$x ^ m * \text{inverse } (x :: 'a) ^ n = \text{inverse } x ^ n * x ^ m$

proof (*induction n*)

case (*Suc n*)

have $x ^ m * \text{inverse } x ^ \text{Suc } n = (x ^ m * \text{inverse } x ^ n) * \text{inverse } x$

by (*simp only: power-Suc2 mult.assoc*)

also have $x ^ m * \text{inverse } x ^ n = \text{inverse } x ^ n * x ^ m$

by (*rule Suc*)

also have $\dots * \text{inverse } x = (\text{inverse } x ^ n * \text{inverse } x) * x ^ m$

by (*simp add: mult.assoc power-mult-inverse-distrib*)

also have $\dots = \text{inverse } x ^ (\text{Suc } n) * x ^ m$

by (*simp only: power-Suc2*)

finally show *?case .*

qed *auto*

lemma *power-int-add*:

assumes $x \neq 0 \vee m + n \neq 0$

shows $\text{power-int } (x :: 'a) (m + n) = \text{power-int } x m * \text{power-int } x n$

proof (*cases x = 0*)

case *True*

thus *?thesis using assms by (auto simp: power-int-0-left-If)*

next

```

case [simp]: False
show ?thesis
proof (cases m n rule: int-cases4 [case-product int-cases4])
  case (nonneg-nonneg a b)
  thus ?thesis
  by (auto simp: power-int-def nat-add-distrib power-add)
next
case (nonneg-neg a b)
thus ?thesis
by (auto simp: power-int-def nat-diff-distrib not-le power-diff-conv-inverse
  power-mult-power-inverse-commute)
next
case (neg-nonneg a b)
thus ?thesis
by (auto simp: power-int-def nat-diff-distrib not-le power-diff-conv-inverse
  power-mult-power-inverse-commute)
next
case (neg-neg a b)
thus ?thesis
by (auto simp: power-int-def nat-add-distrib add.commute simp flip: power-add)
qed
qed

```

```

lemma power-int-add-1:
  assumes  $x \neq 0 \vee m \neq -1$ 
  shows  $\text{power-int } (x::'a) (m + 1) = \text{power-int } x m * x$ 
  using assms by (subst power-int-add) auto

```

```

lemma power-int-add-1':
  assumes  $x \neq 0 \vee m \neq -1$ 
  shows  $\text{power-int } (x::'a) (m + 1) = x * \text{power-int } x m$ 
  using assms by (subst add.commute, subst power-int-add) auto

```

```

lemma power-int-commutes:  $\text{power-int } (x :: 'a) n * x = x * \text{power-int } x n$ 
  by (cases  $x = 0$ ) (auto simp flip: power-int-add-1 power-int-add-1')

```

```

lemma power-int-inverse [field-simps, field-split-simps, divide-simps]:
   $\text{power-int } (\text{inverse } (x :: 'a)) n = \text{inverse } (\text{power-int } x n)$ 
  by (auto simp: power-int-def power-inverse)

```

```

lemma power-int-mult:  $\text{power-int } (x :: 'a) (m * n) = \text{power-int } (\text{power-int } x m) n$ 
  by (auto simp: power-int-def zero-le-mult-iff simp flip: power-mult power-inverse
  nat-mult-distrib)

```

end

```

context
  assumes SORT-CONSTRAINT('a::field)
begin

```

lemma *power-int-diff*:

assumes $x \neq 0 \vee m \neq n$

shows $\text{power-int } (x :: 'a) (m - n) = \text{power-int } x m / \text{power-int } x n$

using *power-int-add*[of $x m - n$] *assms* **by** (*auto simp: field-simps power-int-minus*)

lemma *power-int-minus-mult*: $x \neq 0 \vee n \neq 0 \implies \text{power-int } (x :: 'a) (n - 1) * x = \text{power-int } x n$

by (*auto simp flip: power-int-add-1*)

lemma *power-int-mult-distrib*: $\text{power-int } (x * y :: 'a) m = \text{power-int } x m * \text{power-int } y m$

by (*auto simp: power-int-def power-mult-distrib*)

lemmas *power-int-mult-distrib-numeral1* = *power-int-mult-distrib* [**where** $x = \text{numeral } w$ **for** w , *simp*]

lemmas *power-int-mult-distrib-numeral2* = *power-int-mult-distrib* [**where** $y = \text{numeral } w$ **for** w , *simp*]

lemma *power-int-divide-distrib*: $\text{power-int } (x / y :: 'a) m = \text{power-int } x m / \text{power-int } y m$

using *power-int-mult-distrib*[of $x \text{ inverse } y m$] **unfolding** *power-int-inverse*

by (*simp add: field-simps*)

end

lemma *power-int-add-numeral* [*simp*]:

$\text{power-int } x (\text{numeral } m) * \text{power-int } x (\text{numeral } n) = \text{power-int } x (\text{numeral } (m + n))$

for $x :: 'a :: \text{division-ring}$

by (*simp add: power-int-add [symmetric]*)

lemma *power-int-add-numeral2* [*simp*]:

$\text{power-int } x (\text{numeral } m) * (\text{power-int } x (\text{numeral } n) * b) = \text{power-int } x (\text{numeral } (m + n)) * b$

for $x :: 'a :: \text{division-ring}$

by (*simp add: mult.assoc [symmetric]*)

lemma *power-int-mult-numeral* [*simp*]:

$\text{power-int } (\text{power-int } x (\text{numeral } m)) (\text{numeral } n) = \text{power-int } x (\text{numeral } (m * n))$

for $x :: 'a :: \text{division-ring}$

by (*simp only: numeral-mult power-int-mult*)

lemma *power-int-not-zero*: $(x :: 'a :: \text{division-ring}) \neq 0 \vee n = 0 \implies \text{power-int } x n \neq 0$

by (*subst power-int-eq-0-iff*) *auto*

lemma *power-int-one-over* [*field-simps, field-split-simps, divide-simps*]:
power-int (1 / x :: 'a :: *division-ring*) n = 1 / *power-int* x n
using *power-int-inverse*[of x] **by** (*simp add: divide-inverse*)

context

assumes *SORT-CONSTRAINT*('a :: *linordered-field*)

begin

lemma *power-int-numeral-neg-numeral* [*simp*]:
power-int (numeral m) (–numeral n) = (*inverse* (numeral (Num.pow m n)) :: 'a)
by (*simp add: power-int-minus*)

lemma *zero-less-power-int* [*simp*]: $0 < (x :: 'a) \implies 0 < \text{power-int } x \ n$
by (*auto simp: power-int-def*)

lemma *zero-le-power-int* [*simp*]: $0 \leq (x :: 'a) \implies 0 \leq \text{power-int } x \ n$
by (*auto simp: power-int-def*)

lemma *power-int-mono*: $(x :: 'a) \leq y \implies n \geq 0 \implies 0 \leq x \implies \text{power-int } x \ n \leq \text{power-int } y \ n$
by (*cases n rule: int-cases4*) (*auto intro: power-mono*)

lemma *one-le-power-int* [*simp*]: $1 \leq (x :: 'a) \implies n \geq 0 \implies 1 \leq \text{power-int } x \ n$
using *power-int-mono* [of 1 x n] **by** *simp*

lemma *power-int-le-one*: $0 \leq (x :: 'a) \implies n \geq 0 \implies x \leq 1 \implies \text{power-int } x \ n \leq 1$
using *power-int-mono* [of x 1 n] **by** *simp*

lemma *power-int-le-imp-le-exp*:

assumes *gt1*: $1 < (x :: 'a :: \text{linordered-field})$

assumes *power-int* $x \ m \leq \text{power-int } x \ n \ n \geq 0$

shows $m \leq n$

proof (*cases m < 0*)

case *True*

with $\langle n \geq 0 \rangle$ **show** *?thesis* **by** *simp*

next

case *False*

with *assms* **have** $x \wedge^{\text{nat}} m \leq x \wedge^{\text{nat}} n$

by (*simp add: power-int-def*)

from *gt1* **and** *this* **show** *?thesis*

using *False* $\langle n \geq 0 \rangle$ **by** *auto*

qed

lemma *power-int-le-imp-less-exp*:

assumes *gt1*: $1 < (x :: 'a :: \text{linordered-field})$

assumes *power-int* $x \ m < \text{power-int } x \ n \ n \geq 0$

```

shows  $m < n$ 
proof (cases  $m < 0$ )
  case True
    with  $\langle n \geq 0 \rangle$  show ?thesis by simp
  next
    case False
      with assms have  $x \wedge^{\text{nat}} m < x \wedge^{\text{nat}} n$ 
        by (simp add: power-int-def)
      from gt1 and this show ?thesis
      using False  $\langle n \geq 0 \rangle$  by auto
qed

lemma power-int-strict-mono:
  ( $a :: 'a :: \text{linordered-field}$ )  $< b \implies 0 \leq a \implies 0 < n \implies \text{power-int } a \ n < \text{power-int } b \ n$ 
  by (auto simp: power-int-def intro!: power-strict-mono)

lemma power-int-mono-iff [simp]:
  fixes  $a \ b :: 'a :: \text{linordered-field}$ 
  shows  $\llbracket a \geq 0; b \geq 0; n > 0 \rrbracket \implies \text{power-int } a \ n \leq \text{power-int } b \ n \longleftrightarrow a \leq b$ 
  by (auto simp: power-int-def intro!: power-strict-mono)

lemma power-int-strict-increasing:
  fixes  $a :: 'a :: \text{linordered-field}$ 
  assumes  $n < N \ 1 < a$ 
  shows  $\text{power-int } a \ N > \text{power-int } a \ n$ 
proof –
  have  $*$ :  $a \wedge^{\text{nat}} (N - n) > a \wedge^{\text{nat}} 0$ 
    using assms by (intro power-strict-increasing) auto
  have  $\text{power-int } a \ N = \text{power-int } a \ n * \text{power-int } a \ (N - n)$ 
    using assms by (simp flip: power-int-add)
  also have  $\dots > \text{power-int } a \ n * 1$ 
    using assms  $*$ 
  by (intro mult-strict-left-mono zero-less-power-int) (auto simp: power-int-def)
  finally show ?thesis by simp
qed

lemma power-int-increasing:
  fixes  $a :: 'a :: \text{linordered-field}$ 
  assumes  $n \leq N \ a \geq 1$ 
  shows  $\text{power-int } a \ N \geq \text{power-int } a \ n$ 
proof –
  have  $*$ :  $a \wedge^{\text{nat}} (N - n) \geq a \wedge^{\text{nat}} 0$ 
    using assms by (intro power-increasing) auto
  have  $\text{power-int } a \ N = \text{power-int } a \ n * \text{power-int } a \ (N - n)$ 
    using assms by (simp flip: power-int-add)
  also have  $\dots \geq \text{power-int } a \ n * 1$ 
    using assms  $*$  by (intro mult-left-mono) (auto simp: power-int-def)
  finally show ?thesis by simp

```

qed

lemma *power-int-strict-decreasing*:

fixes $a :: 'a :: \text{linordered-field}$

assumes $n < N \ 0 < a \ a < 1$

shows $\text{power-int } a \ N < \text{power-int } a \ n$

proof –

have $*$: $a^{\text{nat } (N - n)} < a^0$

using *assms* **by** (*intro power-strict-decreasing*) *auto*

have $\text{power-int } a \ N = \text{power-int } a \ n * \text{power-int } a \ (N - n)$

using *assms* **by** (*simp flip: power-int-add*)

also have $\dots < \text{power-int } a \ n * 1$

using *assms* $*$

by (*intro mult-strict-left-mono zero-less-power-int*) (*auto simp: power-int-def*)

finally show *?thesis* **by** *simp*

qed

lemma *power-int-decreasing*:

fixes $a :: 'a :: \text{linordered-field}$

assumes $n \leq N \ 0 \leq a \ a \leq 1 \ a \neq 0 \vee N \neq 0 \vee n = 0$

shows $\text{power-int } a \ N \leq \text{power-int } a \ n$

proof (*cases a = 0*)

case *False*

have $*$: $a^{\text{nat } (N - n)} \leq a^0$

using *assms* **by** (*intro power-decreasing*) *auto*

have $\text{power-int } a \ N = \text{power-int } a \ n * \text{power-int } a \ (N - n)$

using *assms False* **by** (*simp flip: power-int-add*)

also have $\dots \leq \text{power-int } a \ n * 1$

using *assms* $*$ **by** (*intro mult-left-mono*) (*auto simp: power-int-def*)

finally show *?thesis* **by** *simp*

qed (*use assms in <auto simp: power-int-0-left-If>*)

lemma *one-less-power-int*: $1 < (a :: 'a) \implies 0 < n \implies 1 < \text{power-int } a \ n$

using *power-int-strict-increasing[of 0 n a]* **by** *simp*

lemma *power-int-abs*: $|\text{power-int } a \ n :: 'a| = \text{power-int } |a| \ n$

by (*auto simp: power-int-def power-abs*)

lemma *power-int-sgn* [*simp*]: $\text{sgn } (\text{power-int } a \ n :: 'a) = \text{power-int } (\text{sgn } a) \ n$

by (*auto simp: power-int-def*)

lemma *abs-power-int-minus* [*simp*]: $|\text{power-int } (-a) \ n :: 'a| = |\text{power-int } a \ n|$

by (*simp add: power-int-abs*)

lemma *power-int-strict-antimono*:

assumes $(a :: 'a :: \text{linordered-field}) < b \ 0 < a \ n < 0$

shows $\text{power-int } a \ n > \text{power-int } b \ n$

proof –

have $\text{inverse } (\text{power-int } a \ (-n)) > \text{inverse } (\text{power-int } b \ (-n))$

```

using assms by (intro less-imp-inverse-less power-int-strict-mono zero-less-power-int)
auto
thus ?thesis by (simp add: power-int-minus)
qed

```

```

lemma power-int-antimono:
  assumes (a :: 'a :: linordered-field)  $\leq b$   $0 < a$   $n < 0$ 
  shows power-int a n  $\geq$  power-int b n
  using power-int-strict-antimono[of a b n] assms by (cases a = b) auto

```

end

54.19 Finiteness of intervals

```

lemma finite-interval-int1 [iff]: finite {i :: int.  $a \leq i \wedge i \leq b$ }
proof (cases a ≤ b)
  case True
    then show ?thesis
    proof (induct b rule: int-ge-induct)
      case base
        have {i.  $a \leq i \wedge i \leq a$ } = {a} by auto
        then show ?case by simp
      next
        case (step b)
          then have {i.  $a \leq i \wedge i \leq b + 1$ } = {i.  $a \leq i \wedge i \leq b$ }  $\cup$  {b + 1} by auto
          with step show ?case by simp
        qed
      next
        case False
          then show ?thesis
          by (metis (lifting, no-types) Collect-empty-eq finite.emptyI order-trans)
    qed

```

```

lemma finite-interval-int2 [iff]: finite {i :: int.  $a \leq i \wedge i < b$ }
  by (rule rev-finite-subset[OF finite-interval-int1[of a b]]) auto

```

```

lemma finite-interval-int3 [iff]: finite {i :: int.  $a < i \wedge i \leq b$ }
  by (rule rev-finite-subset[OF finite-interval-int1[of a b]]) auto

```

```

lemma finite-interval-int4 [iff]: finite {i :: int.  $a < i \wedge i < b$ }
  by (rule rev-finite-subset[OF finite-interval-int1[of a b]]) auto

```

54.20 Configuration of the code generator

Constructors

```

definition Pos :: num  $\Rightarrow$  int
  where [simp, code-abbrev]: Pos = numeral

```

```

definition Neg :: num  $\Rightarrow$  int

```


where [*simp*, *code-abbrev*]: $Neg\ n = - (Pos\ n)$

code-datatype $0::int\ Pos\ Neg$

Auxiliary operations.

definition $dup :: int \Rightarrow int$

where [*simp*]: $dup\ k = k + k$

lemma *dup-code* [*code*]:

$dup\ 0 = 0$

$dup\ (Pos\ n) = Pos\ (Num.Bit0\ n)$

$dup\ (Neg\ n) = Neg\ (Num.Bit0\ n)$

by (*simp-all add: numeral-Bit0*)

definition $sub :: num \Rightarrow num \Rightarrow int$

where [*simp*]: $sub\ m\ n = numeral\ m - numeral\ n$

lemma *sub-code* [*code*]:

$sub\ Num.One\ Num.One = 0$

$sub\ (Num.Bit0\ m)\ Num.One = Pos\ (Num.BitM\ m)$

$sub\ (Num.Bit1\ m)\ Num.One = Pos\ (Num.Bit0\ m)$

$sub\ Num.One\ (Num.Bit0\ n) = Neg\ (Num.BitM\ n)$

$sub\ Num.One\ (Num.Bit1\ n) = Neg\ (Num.Bit0\ n)$

$sub\ (Num.Bit0\ m)\ (Num.Bit0\ n) = dup\ (sub\ m\ n)$

$sub\ (Num.Bit1\ m)\ (Num.Bit1\ n) = dup\ (sub\ m\ n)$

$sub\ (Num.Bit1\ m)\ (Num.Bit0\ n) = dup\ (sub\ m\ n) + 1$

$sub\ (Num.Bit0\ m)\ (Num.Bit1\ n) = dup\ (sub\ m\ n) - 1$

by (*simp-all only: sub-def dup-def numeral.simps Pos-def Neg-def numeral-BitM*)

lemma *sub-BitM-One-eq*:

$\langle (Num.sub\ (Num.BitM\ n)\ num.One) = 2 * (Num.sub\ n\ Num.One :: int) \rangle$

by (*cases\ n\ simp-all*)

Implementations.

lemma *one-int-code* [*code*]: $1 = Pos\ Num.One$

by *simp*

lemma *plus-int-code* [*code*]:

$k + 0 = k$

$0 + l = l$

$Pos\ m + Pos\ n = Pos\ (m + n)$

$Pos\ m + Neg\ n = sub\ m\ n$

$Neg\ m + Pos\ n = sub\ n\ m$

$Neg\ m + Neg\ n = Neg\ (m + n)$

for $k\ l :: int$

by *simp-all*

lemma *uminus-int-code* [*code*]:

$uminus\ 0 = (0::int)$

$uminus (Pos\ m) = Neg\ m$
 $uminus (Neg\ m) = Pos\ m$
by *simp-all*

lemma *minus-int-code* [*code*]:

$k - 0 = k$
 $0 - l = uminus\ l$
 $Pos\ m - Pos\ n = sub\ m\ n$
 $Pos\ m - Neg\ n = Pos\ (m + n)$
 $Neg\ m - Pos\ n = Neg\ (m + n)$
 $Neg\ m - Neg\ n = sub\ n\ m$
for $k\ l :: int$
by *simp-all*

lemma *times-int-code* [*code*]:

$k * 0 = 0$
 $0 * l = 0$
 $Pos\ m * Pos\ n = Pos\ (m * n)$
 $Pos\ m * Neg\ n = Neg\ (m * n)$
 $Neg\ m * Pos\ n = Neg\ (m * n)$
 $Neg\ m * Neg\ n = Pos\ (m * n)$
for $k\ l :: int$
by *simp-all*

instantiation $int :: equal$
begin

definition $HOL.equal\ k\ l \longleftrightarrow k = (l :: int)$

instance

by *standard (rule equal-int-def)*

end

lemma *equal-int-code* [*code*]:

$HOL.equal\ 0\ (0 :: int) \longleftrightarrow True$
 $HOL.equal\ 0\ (Pos\ l) \longleftrightarrow False$
 $HOL.equal\ 0\ (Neg\ l) \longleftrightarrow False$
 $HOL.equal\ (Pos\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (Pos\ k)\ (Pos\ l) \longleftrightarrow HOL.equal\ k\ l$
 $HOL.equal\ (Pos\ k)\ (Neg\ l) \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ (Pos\ l) \longleftrightarrow False$
 $HOL.equal\ (Neg\ k)\ (Neg\ l) \longleftrightarrow HOL.equal\ k\ l$
by (*auto simp add: equal*)

lemma *equal-int-refl* [*code nbe*]: $HOL.equal\ k\ k \longleftrightarrow True$

for $k :: int$
by (*fact equal-refl*)

lemma *less-eq-int-code* [code]:

$0 \leq (0::int) \longleftrightarrow \text{True}$
 $0 \leq \text{Pos } l \longleftrightarrow \text{True}$
 $0 \leq \text{Neg } l \longleftrightarrow \text{False}$
 $\text{Pos } k \leq 0 \longleftrightarrow \text{False}$
 $\text{Pos } k \leq \text{Pos } l \longleftrightarrow k \leq l$
 $\text{Pos } k \leq \text{Neg } l \longleftrightarrow \text{False}$
 $\text{Neg } k \leq 0 \longleftrightarrow \text{True}$
 $\text{Neg } k \leq \text{Pos } l \longleftrightarrow \text{True}$
 $\text{Neg } k \leq \text{Neg } l \longleftrightarrow l \leq k$
by *simp-all*

lemma *less-int-code* [code]:

$0 < (0::int) \longleftrightarrow \text{False}$
 $0 < \text{Pos } l \longleftrightarrow \text{True}$
 $0 < \text{Neg } l \longleftrightarrow \text{False}$
 $\text{Pos } k < 0 \longleftrightarrow \text{False}$
 $\text{Pos } k < \text{Pos } l \longleftrightarrow k < l$
 $\text{Pos } k < \text{Neg } l \longleftrightarrow \text{False}$
 $\text{Neg } k < 0 \longleftrightarrow \text{True}$
 $\text{Neg } k < \text{Pos } l \longleftrightarrow \text{True}$
 $\text{Neg } k < \text{Neg } l \longleftrightarrow l < k$
by *simp-all*

lemma *nat-code* [code]:

$\text{nat } (\text{Int.Neg } k) = 0$
 $\text{nat } 0 = 0$
 $\text{nat } (\text{Int.Pos } k) = \text{nat-of-num } k$
by (*simp-all add: nat-of-num-numeral*)

lemma (**in** *ring-1*) *of-int-code* [code]:

$\text{of-int } (\text{Int.Neg } k) = - \text{numeral } k$
 $\text{of-int } 0 = 0$
 $\text{of-int } (\text{Int.Pos } k) = \text{numeral } k$
by *simp-all*

Serializer setup.

code-identifier

code-module *Int* \rightarrow (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

quickcheck-params [*default-type = int*]

hide-const (**open**) *Pos Neg sub dup*

De-register *int* as a quotient type:

lifting-update *int.lifting*

lifting-forget *int.lifting*

54.21 Duplicates

```

lemmas int-sum = of-nat-sum [where 'a=int]
lemmas int-prod = of-nat-prod [where 'a=int]
lemmas zle-int = of-nat-le-iff [where 'a=int]
lemmas int-int-eq = of-nat-eq-iff [where 'a=int]
lemmas nonneg-eq-int = nonneg-int-cases
lemmas double-eq-0-iff = double-zero

```

```

lemmas int-distrib =
  distrib-right [of z1 z2 w]
  distrib-left [of w z1 z2]
  left-diff-distrib [of z1 z2 w]
  right-diff-distrib [of w z1 z2]
for z1 z2 w :: int

```

end

55 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets

```

theory Lattices-Big
  imports Groups-Big Option
begin

```

55.1 Generic lattice operations over a set

55.1.1 Without neutral element

```

locale semilattice-set = semilattice
begin

```

```

interpretation comp-fun-idem f
  by standard (simp-all add: fun-eq-iff left-commute)

```

```

definition F :: 'a set  $\Rightarrow$  'a

```

where

```

  eq-fold': F A = the (Finite-Set.fold ( $\lambda x y. \text{Some } (\text{case } y \text{ of } \text{None} \Rightarrow x \mid \text{Some } z \Rightarrow f x z)$ ) None A)

```

```

lemma eq-fold:

```

```

  assumes finite A

```

```

  shows F (insert x A) = Finite-Set.fold f x A

```

```

proof (rule sym)

```

```

  let ?f =  $\lambda x y. \text{Some } (\text{case } y \text{ of } \text{None} \Rightarrow x \mid \text{Some } z \Rightarrow f x z)$ 

```

```

  interpret comp-fun-idem ?f

```

```

  by standard (simp-all add: fun-eq-iff commute left-commute split: option.split)

```

```

  from assms show Finite-Set.fold f x A = F (insert x A)

```

```

  proof induct

```

```

    case empty then show ?case by (simp add: eq-fold')
  next
    case (insert y B) then show ?case by (simp add: insert-commute [of x] eq-fold')
  qed
qed

```

lemma *singleton* [simp]:

```

  F {x} = x
  by (simp add: eq-fold)

```

lemma *insert-not-elem*:

```

  assumes finite A and x ∉ A and A ≠ {}
  shows F (insert x A) = x * F A
  proof -
    from ⟨A ≠ {}⟩ obtain b where b ∈ A by blast
    then obtain B where *: A = insert b B b ∉ B by (blast dest: mk-disjoint-insert)
    with ⟨finite A⟩ and ⟨x ∉ A⟩
      have finite (insert x B) and b ∉ insert x B by auto
    then have F (insert b (insert x B)) = x * F (insert b B)
      by (simp add: eq-fold)
    then show ?thesis by (simp add: * insert-commute)
  qed

```

lemma *in-idem*:

```

  assumes finite A and x ∈ A
  shows x * F A = F A
  proof -
    from assms have A ≠ {} by auto
    with ⟨finite A⟩ show ?thesis using ⟨x ∈ A⟩
      by (induct A rule: finite-ne-induct) (auto simp add: ac-simps insert-not-elem)
  qed

```

lemma *insert* [simp]:

```

  assumes finite A and A ≠ {}
  shows F (insert x A) = x * F A
  using assms by (cases x ∈ A) (simp-all add: insert-absorb in-idem insert-not-elem)

```

lemma *union*:

```

  assumes finite A A ≠ {} and finite B B ≠ {}
  shows F (A ∪ B) = F A * F B
  using assms by (induct A rule: finite-ne-induct) (simp-all add: ac-simps)

```

lemma *remove*:

```

  assumes finite A and x ∈ A
  shows F A = (if A - {x} = {} then x else x * F (A - {x}))
  proof -
    from assms obtain B where A = insert x B and x ∉ B by (blast dest:
mk-disjoint-insert)
    with assms show ?thesis by simp
  qed

```

qed

lemma *insert-remove*:

assumes *finite A*

shows $F (\text{insert } x \ A) = (\text{if } A - \{x\} = \{\} \text{ then } x \text{ else } x * F (A - \{x\}))$

using *assms* by (*cases* $x \in A$) (*simp-all* add: *insert-absorb* *remove*)

lemma *subset*:

assumes *finite A* $B \neq \{\}$ and $B \subseteq A$

shows $F B * F A = F A$

proof –

from *assms* have $A \neq \{\}$ and *finite B* by (*auto* *dest*: *finite-subset*)

with *assms* show *?thesis* by (*simp* add: *union* [*symmetric*] *Un-absorb1*)

qed

lemma *closed*:

assumes *finite A* $A \neq \{\}$ and *elem*: $\bigwedge x \ y. x * y \in \{x, y\}$

shows $F A \in A$

using $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle$ proof (*induct* rule: *finite-ne-induct*)

case *singleton* then show *?case* by *simp*

next

case *insert* with *elem* show *?case* by *force*

qed

lemma *hom-commute*:

assumes *hom*: $\bigwedge x \ y. h (x * y) = h x * h y$

and *N*: *finite N* $N \neq \{\}$

shows $h (F N) = F (h \ ' \ N)$

using *N* proof (*induct* rule: *finite-ne-induct*)

case *singleton* thus *?case* by *simp*

next

case (*insert n N*)

then have $h (F (\text{insert } n \ N)) = h (n * F N)$ by *simp*

also have $\dots = h n * h (F N)$ by (*rule* *hom*)

also have $h (F N) = F (h \ ' \ N)$ by (*rule* *insert*)

also have $h n * \dots = F (\text{insert } (h \ n) (h \ ' \ N))$

using *insert* by *simp*

also have $\text{insert } (h \ n) (h \ ' \ N) = h \ ' \ \text{insert } n \ N$ by *simp*

finally show *?case* .

qed

lemma *infinite*: $\neg \text{finite } A \implies F A = \text{the None}$

unfolding *eq-fold'* by (*cases* *finite* (*UNIV*::'a set)) (*auto* *intro*: *finite-subset* *fold-infinite*)

end

locale *semilattice-order-set* = *binary*? : *semilattice-order* + *semilattice-set*

begin

lemma *bounded-iff*:

assumes *finite A* **and** $A \neq \{\}$

shows $x \leq F A \longleftrightarrow (\forall a \in A. x \leq a)$

using *assms* **by** (*induct rule: finite-ne-induct*) *simp-all*

lemma *boundedI*:

assumes *finite A*

assumes $A \neq \{\}$

assumes $\bigwedge a. a \in A \implies x \leq a$

shows $x \leq F A$

using *assms* **by** (*simp add: bounded-iff*)

lemma *boundedE*:

assumes *finite A* **and** $A \neq \{\}$ **and** $x \leq F A$

obtains $\bigwedge a. a \in A \implies x \leq a$

using *assms* **by** (*simp add: bounded-iff*)

lemma *coboundedI*:

assumes *finite A*

and $a \in A$

shows $F A \leq a$

proof –

from *assms* **have** $A \neq \{\}$ **by** *auto*

from $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle \langle a \in A \rangle$ **show** *?thesis*

proof (*induct rule: finite-ne-induct*)

case *singleton* **thus** *?case* **by** (*simp add: refl*)

next

case (*insert x B*)

from *insert* **have** $a = x \vee a \in B$ **by** *simp*

then show *?case* **using** *insert* **by** (*auto intro: coboundedI2*)

qed

qed

lemma *subset-imp*:

assumes $A \subseteq B$ **and** $A \neq \{\}$ **and** *finite B*

shows $F B \leq F A$

proof (*cases A = B*)

case *True* **then show** *?thesis* **by** (*simp add: refl*)

next

case *False*

have *B*: $B = A \cup (B - A)$ **using** $\langle A \subseteq B \rangle$ **by** *blast*

then have $F B = F (A \cup (B - A))$ **by** *simp*

also have $\dots = F A * F (B - A)$ **using** *False assms* **by** (*subst union*) (*auto intro: finite-subset*)

also have $\dots \leq F A$ **by** *simp*

finally show *?thesis* .

qed

end

55.1.2 With neutral element

locale *semilattice-neutr-set* = *semilattice-neutr*
begin

interpretation *comp-fun-idem* *f*
by *standard* (*simp-all add: fun-eq-iff left-commute*)

definition $F :: 'a \text{ set} \Rightarrow 'a$
where
eq-fold: $F A = \text{Finite-Set.fold } f \mathbf{1} A$

lemma *infinite* [*simp*]:
 $\neg \text{finite } A \Longrightarrow F A = \mathbf{1}$
by (*simp add: eq-fold*)

lemma *empty* [*simp*]:
 $F \{\} = \mathbf{1}$
by (*simp add: eq-fold*)

lemma *insert* [*simp*]:
assumes *finite A*
shows $F (\text{insert } x A) = x * F A$
using *assms* by (*simp add: eq-fold*)

lemma *in-idem*:
assumes *finite A* and $x \in A$
shows $x * F A = F A$
proof –
from *assms* have $A \neq \{\}$ by *auto*
with $\langle \text{finite } A \rangle$ show *?thesis* using $\langle x \in A \rangle$
by (*induct A rule: finite-ne-induct*) (*auto simp add: ac-simps*)
qed

lemma *union*:
assumes *finite A* and *finite B*
shows $F (A \cup B) = F A * F B$
using *assms* by (*induct A*) (*simp-all add: ac-simps*)

lemma *remove*:
assumes *finite A* and $x \in A$
shows $F A = x * F (A - \{x\})$
proof –
from *assms* obtain *B* where $A = \text{insert } x B$ and $x \notin B$ by (*blast dest: mk-disjoint-insert*)
with *assms* show *?thesis* by *simp*
qed

lemma *insert-remove*:

assumes *finite A*

shows $F (\text{insert } x \ A) = x * F (A - \{x\})$

using *assms* **by** (*cases* $x \in A$) (*simp-all* *add: insert-absorb remove*)

lemma *subset*:

assumes *finite A* **and** $B \subseteq A$

shows $F B * F A = F A$

proof –

from *assms* **have** *finite B* **by** (*auto* *dest: finite-subset*)

with *assms* **show** *?thesis* **by** (*simp* *add: union [symmetric] Un-absorb1*)

qed

lemma *closed*:

assumes *finite A* $A \neq \{\}$ **and** *elem*: $\bigwedge x y. x * y \in \{x, y\}$

shows $F A \in A$

using $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle$ **proof** (*induct* *rule: finite-ne-induct*)

case *singleton* **then** **show** *?case* **by** *simp*

next

case *insert* **with** *elem* **show** *?case* **by** *force*

qed

end

locale *semilattice-order-neutr-set* = *binary?*: *semilattice-neutr-order* + *semilattice-neutr-set*

begin

lemma *bounded-iff*:

assumes *finite A*

shows $x \leq F A \longleftrightarrow (\forall a \in A. x \leq a)$

using *assms* **by** (*induct* *A*) *simp-all*

lemma *boundedI*:

assumes *finite A*

assumes $\bigwedge a. a \in A \implies x \leq a$

shows $x \leq F A$

using *assms* **by** (*simp* *add: bounded-iff*)

lemma *boundedE*:

assumes *finite A* **and** $x \leq F A$

obtains $\bigwedge a. a \in A \implies x \leq a$

using *assms* **by** (*simp* *add: bounded-iff*)

lemma *coboundedI*:

assumes *finite A*

and $a \in A$

shows $F A \leq a$

```

proof –
  from assms have  $A \neq \{\}$  by auto
  from  $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle \langle a \in A \rangle$  show ?thesis
  proof (induct rule: finite-ne-induct)
    case singleton thus ?case by (simp add: refl)
  next
    case (insert x B)
    from insert have  $a = x \vee a \in B$  by simp
    then show ?case using insert by (auto intro: coboundedI2)
  qed
qed

lemma subset-imp:
  assumes  $A \subseteq B$  and finite B
  shows  $F B \leq F A$ 
  proof (cases A = B)
    case True then show ?thesis by (simp add: refl)
  next
    case False
    have  $B: B = A \cup (B - A)$  using  $\langle A \subseteq B \rangle$  by blast
    then have  $F B = F (A \cup (B - A))$  by simp
    also have  $\dots = F A * F (B - A)$  using False assms by (subst union) (auto
intro: finite-subset)
    also have  $\dots \leq F A$  by simp
    finally show ?thesis .
  qed

```

end

55.2 Lattice operations on finite sets

context *semilattice-inf*

begin

sublocale *Inf-fin: semilattice-order-set inf less-eq less*

defines

$$\text{Inf-fin } (\sqcap_{fin} - [900] 900) = \text{Inf-fin.F} ..$$

end

context *semilattice-sup*

begin

sublocale *Sup-fin: semilattice-order-set sup greater-eq greater*

defines

$$\text{Sup-fin } (\sqcup_{fin} - [900] 900) = \text{Sup-fin.F} ..$$

end

55.3 Infimum and Supremum over non-empty sets

context *lattice*

begin

lemma *Inf-fin-le-Sup-fin* [*simp*]:

assumes *finite A* and $A \neq \{\}$

shows $\prod_{fin} A \leq \sqcup_{fin} A$

proof –

from $\langle A \neq \{\} \rangle$ obtain *a* where $a \in A$ by *blast*

with $\langle finite\ A \rangle$ have $\prod_{fin} A \leq a$ by (rule *Inf-fin.coboundedI*)

moreover from $\langle finite\ A \rangle \langle a \in A \rangle$ have $a \leq \sqcup_{fin} A$ by (rule *Sup-fin.coboundedI*)

ultimately show *?thesis* by (rule *order-trans*)

qed

lemma *sup-Inf-absorb* [*simp*]:

$finite\ A \implies a \in A \implies \prod_{fin} A \sqcup a = a$

by (rule *sup-absorb2*) (rule *Inf-fin.coboundedI*)

lemma *inf-Sup-absorb* [*simp*]:

$finite\ A \implies a \in A \implies a \sqcap \sqcup_{fin} A = a$

by (rule *inf-absorb1*) (rule *Sup-fin.coboundedI*)

end

context *distrib-lattice*

begin

lemma *sup-Inf1-distrib*:

assumes *finite A*

and $A \neq \{\}$

shows $sup\ x\ (\prod_{fin} A) = \prod_{fin} \{sup\ x\ a\ | a. a \in A\}$

using *assms* by (simp add: *image-def Inf-fin.hom-commute* [where $h=sup\ x$, OF *sup-inf-distrib1*])

(rule *arg-cong* [where $f=Inf-fin$], *blast*)

lemma *sup-Inf2-distrib*:

assumes *A: finite A A ≠ {}* and *B: finite B B ≠ {}*

shows $sup\ (\prod_{fin} A)\ (\prod_{fin} B) = \prod_{fin} \{sup\ a\ b\ | a\ b. a \in A \wedge b \in B\}$

using *A* proof (induct rule: *finite-ne-induct*)

case *singleton* then show *?case*

by (simp add: *sup-Inf1-distrib* [OF *B*])

next

case (*insert x A*)

have *finB: finite {sup x b | b. b ∈ B}*

by (rule *finite-surj* [where $f = sup\ x$, OF *B(1)*], *auto*)

have *finAB: finite {sup a b | a b. a ∈ A ∧ b ∈ B}*

proof –

have $\{sup\ a\ b\ | a\ b. a \in A \wedge b \in B\} = (\bigcup_{a \in A}. \bigcup_{b \in B}. \{sup\ a\ b\})$

by *blast*

thus *?thesis* **by**(*simp add: insert(1) B(1)*)
qed
have *ne*: $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B by blast*
have $sup\ (\prod_{fin}(insert\ x\ A))\ (\prod_{fin}B) = sup\ (inf\ x\ (\prod_{fin}A))\ (\prod_{fin}B)$
using *insert by simp*
also have $\dots = inf\ (sup\ x\ (\prod_{fin}B))\ (sup\ (\prod_{fin}A)\ (\prod_{fin}B))$ **by**(*rule sup-inf-distrib2*)
also have $\dots = inf\ (\prod_{fin}\{sup\ x\ b\ |b.\ b \in B\})\ (\prod_{fin}\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\})$
using *insert by(simp add:sup-Inf1-distrib[OF B])*
also have $\dots = \prod_{fin}(\{sup\ x\ b\ |b.\ b \in B\} \cup \{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\})$
(is - = $\prod_{fin} ?M$ **)**
using *B insert*
by (*simp add: Inf-fin.union [OF finB - finAB ne]*)
also have $?M = \{sup\ a\ b\ |a\ b.\ a \in insert\ x\ A \wedge b \in B\}$
by *blast*
finally show *?case .*
qed

lemma *inf-Sup1-distrib*:

assumes *finite A and A $\neq \{\}$*
shows $inf\ x\ (\bigsqcup_{fin}A) = \bigsqcup_{fin}\{inf\ x\ a\ |a.\ a \in A\}$
using *assms by (simp add: image-def Sup-fin.hom-commute [where h=inf x, OF inf-sup-distrib1])*
(rule arg-cong [where f=Sup-fin], blast)

lemma *inf-Sup2-distrib*:

assumes *A: finite A A $\neq \{\}$ and B: finite B B $\neq \{\}$*
shows $inf\ (\bigsqcup_{fin}A)\ (\bigsqcup_{fin}B) = \bigsqcup_{fin}\{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\}$
using *A proof (induct rule: finite-ne-induct)*
case singleton thus *?case*
by(*simp add: inf-Sup1-distrib [OF B]*)

next

case (*insert x A*)
have *finB: finite {inf x b |b. b $\in B$ }*
by(*rule finite-surj[where f = %b. inf x b, OF B(1)], auto*)
have *finAB: finite {inf a b |a b. a $\in A \wedge b \in B$ }*
proof -
have $\{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} = (\bigcup_{a \in A} \bigcup_{b \in B} \{inf\ a\ b\})$
by *blast*
thus *?thesis by(simp add: insert(1) B(1))*

qed

have *ne*: $\{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B by blast*
have $inf\ (\bigsqcup_{fin}(insert\ x\ A))\ (\bigsqcup_{fin}B) = inf\ (sup\ x\ (\bigsqcup_{fin}A))\ (\bigsqcup_{fin}B)$
using *insert by simp*
also have $\dots = sup\ (inf\ x\ (\bigsqcup_{fin}B))\ (inf\ (\bigsqcup_{fin}A)\ (\bigsqcup_{fin}B))$ **by**(*rule inf-sup-distrib2*)
also have $\dots = sup\ (\bigsqcup_{fin}\{inf\ x\ b\ |b.\ b \in B\})\ (\bigsqcup_{fin}\{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\})$
using *insert by(simp add:inf-Sup1-distrib[OF B])*
also have $\dots = \bigsqcup_{fin}(\{inf\ x\ b\ |b.\ b \in B\} \cup \{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\})$

```

  (is - =  $\bigsqcup_{fin} ?M$ )
  using  $B$  insert
  by (simp add: Sup-fin.union [OF finB - finAB ne])
  also have  $?M = \{inf\ a\ b\ | a\ b.\ a \in insert\ x\ A \wedge b \in B\}$ 
  by blast
  finally show ?case .
qed

```

end

```

context complete-lattice
begin

```

```

lemma Inf-fin-Inf:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\prod_{fin} A = \prod A$ 
proof -
  from assms obtain b B where  $A = insert\ b\ B$  and finite B by auto
  then show ?thesis
  by (simp add: Inf-fin.eq-fold inf-Inf-fold-inf inf.commute [of b])
qed

```

```

lemma Sup-fin-Sup:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\bigsqcup_{fin} A = \bigsqcup A$ 
proof -
  from assms obtain b B where  $A = insert\ b\ B$  and finite B by auto
  then show ?thesis
  by (simp add: Sup-fin.eq-fold sup-Sup-fold-sup sup.commute [of b])
qed

```

end

55.4 Minimum and Maximum over non-empty sets

```

context linorder
begin

```

```

sublocale Min: semilattice-order-set min less-eq less
+ Max: semilattice-order-set max greater-eq greater
defines
  Min = Min.F and Max = Max.F ..

```

end

```

syntax
-MIN1  :: pptrns  $\Rightarrow$  'b  $\Rightarrow$  'b          ((3MIN -./ -) [0, 10] 10)
-MIN   :: pptrm  $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b  ((3MIN - $\in$ -./ -) [0, 0, 10] 10)
-MAX1  :: pptrns  $\Rightarrow$  'b  $\Rightarrow$  'b          ((3MAX -./ -) [0, 10] 10)

```

-MAX :: $pttrn \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b \ ((\exists MAX \ -\in\ -) [0, 0, 10] 10)$

translations

$MIN\ x\ y.\ f \equiv MIN\ x.\ MIN\ y.\ f$
 $MIN\ x.\ f \equiv CONST\ Min\ (CONST\ range\ (\lambda x.\ f))$
 $MIN\ x \in A.\ f \equiv CONST\ Min\ ((\lambda x.\ f) \text{ ' } A)$
 $MAX\ x\ y.\ f \equiv MAX\ x.\ MAX\ y.\ f$
 $MAX\ x.\ f \equiv CONST\ Max\ (CONST\ range\ (\lambda x.\ f))$
 $MAX\ x \in A.\ f \equiv CONST\ Max\ ((\lambda x.\ f) \text{ ' } A)$

An aside: *Min/Max* on linear orders as special case of *Inf-fin/Sup-fin*

lemma *Inf-fin-Min*:

$Inf\text{-}fin = (Min :: 'a::\{semilattice\text{-}inf, linorder\} \text{ set} \Rightarrow 'a)$
by (*simp add: Inf-fin-def Min-def inf-min*)

lemma *Sup-fin-Max*:

$Sup\text{-}fin = (Max :: 'a::\{semilattice\text{-}sup, linorder\} \text{ set} \Rightarrow 'a)$
by (*simp add: Sup-fin-def Max-def sup-max*)

context *linorder***begin****lemma** *dual-min*:

$ord.min\ greater\text{-}eq = max$
by (*auto simp add: ord.min-def max-def fun-eq-iff*)

lemma *dual-max*:

$ord.max\ greater\text{-}eq = min$
by (*auto simp add: ord.max-def min-def fun-eq-iff*)

lemma *dual-Min*:

$linorder.Min\ greater\text{-}eq = Max$

proof –

interpret *dual*: $linorder\ greater\text{-}eq\ greater$ **by** (*fact dual-linorder*)
show *?thesis* **by** (*simp add: dual.Min-def dual-min Max-def*)

qed**lemma** *dual-Max*:

$linorder.Max\ greater\text{-}eq = Min$

proof –

interpret *dual*: $linorder\ greater\text{-}eq\ greater$ **by** (*fact dual-linorder*)
show *?thesis* **by** (*simp add: dual.Max-def dual-max Min-def*)

qed

lemmas *Min-singleton* = *Min.singleton*

lemmas *Max-singleton* = *Max.singleton*

lemmas *Min-insert* = *Min.insert*

lemmas *Max-insert* = *Max.insert*

lemmas *Min-Un* = *Min.union*

lemmas *Max-Un* = *Max.union*

lemmas *hom-Min-commute* = *Min.hom-commute*

lemmas *hom-Max-commute* = *Max.hom-commute*

lemma *Min-in* [*simp*]:

assumes *finite A* **and** $A \neq \{\}$

shows $\text{Min } A \in A$

using *assms* **by** (*auto simp add: min-def Min.closed*)

lemma *Max-in* [*simp*]:

assumes *finite A* **and** $A \neq \{\}$

shows $\text{Max } A \in A$

using *assms* **by** (*auto simp add: max-def Max.closed*)

lemma *Min-insert2*:

assumes *finite A* **and** $\bigwedge b. b \in A \implies a \leq b$

shows $\text{Min } (\text{insert } a \ A) = a$

proof (*cases A = \{\}*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

with $\langle \text{finite } A \rangle$ **have** $\text{Min } (\text{insert } a \ A) = \text{min } a \ (\text{Min } A)$

by *simp*

moreover from $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle$ *min* **have** $a \leq \text{Min } A$ **by** *simp*

ultimately show *?thesis* **by** (*simp add: min.absorb1*)

qed

lemma *Max-insert2*:

assumes *finite A* **and** $\bigwedge b. b \in A \implies b \leq a$

shows $\text{Max } (\text{insert } a \ A) = a$

proof (*cases A = \{\}*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

with $\langle \text{finite } A \rangle$ **have** $\text{Max } (\text{insert } a \ A) = \text{max } a \ (\text{Max } A)$

by *simp*

moreover from $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle$ *max* **have** $\text{Max } A \leq a$ **by** *simp*

ultimately show *?thesis* **by** (*simp add: max.absorb1*)

qed

lemma *Max-const*[*simp*]: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Max } ((\lambda-. c) \ ` A) = c$

using *Max-in image-is-empty* **by** *blast*

lemma *Min-const*[*simp*]: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Min } ((\lambda-. c) \ ` A) = c$

using *Min-in image-is-empty* **by** *blast*

lemma *Min-le* [*simp*]:

assumes *finite A* and $x \in A$
 shows $\text{Min } A \leq x$
 using *assms* by (fact *Min.coboundedI*)

lemma *Max-ge [simp]*:
 assumes *finite A* and $x \in A$
 shows $x \leq \text{Max } A$
 using *assms* by (fact *Max.coboundedI*)

lemma *Min-eqI*:
 assumes *finite A*
 assumes $\bigwedge y. y \in A \implies y \geq x$
 and $x \in A$
 shows $\text{Min } A = x$
 proof (rule *order.antisym*)
 from $\langle x \in A \rangle$ have $A \neq \{\}$ by *auto*
 with *assms* show $\text{Min } A \geq x$ by *simp*
 next
 from *assms* show $x \geq \text{Min } A$ by *simp*
 qed

lemma *Max-eqI*:
 assumes *finite A*
 assumes $\bigwedge y. y \in A \implies y \leq x$
 and $x \in A$
 shows $\text{Max } A = x$
 proof (rule *order.antisym*)
 from $\langle x \in A \rangle$ have $A \neq \{\}$ by *auto*
 with *assms* show $\text{Max } A \leq x$ by *simp*
 next
 from *assms* show $x \leq \text{Max } A$ by *simp*
 qed

lemma *eq-Min-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies m = \text{Min } A \iff m \in A \wedge (\forall a \in A. m \leq a)$
 by (meson *Min-in Min-le order.antisym*)

lemma *Min-eq-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Min } A = m \iff m \in A \wedge (\forall a \in A. m \leq a)$
 by (meson *Min-in Min-le order.antisym*)

lemma *eq-Max-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies m = \text{Max } A \iff m \in A \wedge (\forall a \in A. a \leq m)$
 by (meson *Max-in Max-ge order.antisym*)

lemma *Max-eq-iff*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Max } A = m \iff m \in A \wedge (\forall a \in A. a \leq m)$
 by (meson *Max-in Max-ge order.antisym*)

context

fixes $A :: 'a \text{ set}$

assumes $\text{fin-nonempty}: \text{finite } A \ A \neq \{\}$

begin

lemma Min-ge-iff [*simp*]:

$x \leq \text{Min } A \longleftrightarrow (\forall a \in A. x \leq a)$

using fin-nonempty **by** (*fact* Min.bounded-iff)

lemma Max-le-iff [*simp*]:

$\text{Max } A \leq x \longleftrightarrow (\forall a \in A. a \leq x)$

using fin-nonempty **by** (*fact* Max.bounded-iff)

lemma Min-gr-iff [*simp*]:

$x < \text{Min } A \longleftrightarrow (\forall a \in A. x < a)$

using fin-nonempty **by** (*induct rule:* finite-ne-induct) *simp-all*

lemma Max-less-iff [*simp*]:

$\text{Max } A < x \longleftrightarrow (\forall a \in A. a < x)$

using fin-nonempty **by** (*induct rule:* finite-ne-induct) *simp-all*

lemma Min-le-iff :

$\text{Min } A \leq x \longleftrightarrow (\exists a \in A. a \leq x)$

using fin-nonempty **by** (*induct rule:* finite-ne-induct) (*simp-all add:* min-le-iff-disj)

lemma Max-ge-iff :

$x \leq \text{Max } A \longleftrightarrow (\exists a \in A. x \leq a)$

using fin-nonempty **by** (*induct rule:* finite-ne-induct) (*simp-all add:* le-max-iff-disj)

lemma Min-less-iff :

$\text{Min } A < x \longleftrightarrow (\exists a \in A. a < x)$

using fin-nonempty **by** (*induct rule:* finite-ne-induct) (*simp-all add:* min-less-iff-disj)

lemma Max-gr-iff :

$x < \text{Max } A \longleftrightarrow (\exists a \in A. x < a)$

using fin-nonempty **by** (*induct rule:* finite-ne-induct) (*simp-all add:* less-max-iff-disj)

end

lemma Max-eq-if :

assumes $\text{finite } A \ \text{finite } B \ \forall a \in A. \exists b \in B. a \leq b \ \forall b \in B. \exists a \in A. b \leq a$

shows $\text{Max } A = \text{Max } B$

proof *cases*

assume $A = \{\}$ **thus** *?thesis* **using** *assms* **by** *simp*

next

assume $A \neq \{\}$ **thus** *?thesis* **using** *assms*

by(*blast intro:* $\text{order.antisym Max-in Max-ge-iff}$ [*THEN iffD2*])

qed

lemma *Min-antimono*:

assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $\text{Min } N \leq \text{Min } M$
using *assms* **by** (*fact Min.subset-imp*)

lemma *Max-mono*:

assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $\text{Max } M \leq \text{Max } N$
using *assms* **by** (*fact Max.subset-imp*)

lemma *mono-Min-commute*:

assumes *mono f*
assumes *finite A* **and** $A \neq \{\}$
shows $f (\text{Min } A) = \text{Min } (f \text{ ` } A)$
proof (*rule linorder-class.Min-eqI [symmetric]*)
from $\langle \text{finite } A \rangle$ **show** *finite* $(f \text{ ` } A)$ **by** *simp*
from *assms* **show** $f (\text{Min } A) \in f \text{ ` } A$ **by** *simp*
fix x
assume $x \in f \text{ ` } A$
then obtain y **where** $y \in A$ **and** $x = f y$..
with *assms* **have** $\text{Min } A \leq y$ **by** *auto*
with $\langle \text{mono } f \rangle$ **have** $f (\text{Min } A) \leq f y$ **by** (*rule monoE*)
with $\langle x = f y \rangle$ **show** $f (\text{Min } A) \leq x$ **by** *simp*
qed

lemma *mono-Max-commute*:

assumes *mono f*
assumes *finite A* **and** $A \neq \{\}$
shows $f (\text{Max } A) = \text{Max } (f \text{ ` } A)$
proof (*rule linorder-class.Max-eqI [symmetric]*)
from $\langle \text{finite } A \rangle$ **show** *finite* $(f \text{ ` } A)$ **by** *simp*
from *assms* **show** $f (\text{Max } A) \in f \text{ ` } A$ **by** *simp*
fix x
assume $x \in f \text{ ` } A$
then obtain y **where** $y \in A$ **and** $x = f y$..
with *assms* **have** $y \leq \text{Max } A$ **by** *auto*
with $\langle \text{mono } f \rangle$ **have** $f y \leq f (\text{Max } A)$ **by** (*rule monoE*)
with $\langle x = f y \rangle$ **show** $x \leq f (\text{Max } A)$ **by** *simp*
qed

lemma *finite-linorder-max-induct [consumes 1, case-names empty insert]*:

assumes *fin: finite A*
and *empty: P {}*
and *insert: $\bigwedge b A. \text{finite } A \implies \forall a \in A. a < b \implies P A \implies P (\text{insert } b A)$*
shows $P A$
using *fin empty insert*
proof (*induct rule: finite-psubset-induct*)
case (*psubset A*)
have $IH: \bigwedge B. \llbracket B < A; P \{\} \rrbracket; (\bigwedge A b. \llbracket \text{finite } A; \forall a \in A. a < b; P A \rrbracket \implies P (\text{insert } b A))$

$b A)) \implies P B$ **by fact**
have $fin: finite A$ **by fact**
have $empty: P \{\}$ **by fact**
have $step: \bigwedge b A. \llbracket finite A; \forall a \in A. a < b; P A \rrbracket \implies P (insert b A)$ **by fact**
show $P A$
proof ($cases A = \{\}$)
assume $A = \{\}$
then show $P A$ **using** $\langle P \{\} \rangle$ **by simp**
next
let $?B = A - \{Max A\}$
let $?A = insert (Max A) ?B$
have $finite ?B$ **using** $\langle finite A \rangle$ **by simp**
assume $A \neq \{\}$
with $\langle finite A \rangle$ **have** $Max A \in A$ **by auto**
then have $A: ?A = A$ **using** $insert-Diff-single insert-absorb$ **by auto**
then have $P ?B$ **using** $\langle P \{\} \rangle$ $step IH [of ?B]$ **by blast**
moreover
have $\forall a \in ?B. a < Max A$ **using** $Max-ge [OF \langle finite A \rangle]$ **by fastforce**
ultimately show $P A$ **using** $A insert-Diff-single step [OF \langle finite ?B \rangle]$ **by fastforce**
qed
qed

lemma $finite-linorder-min-induct$ [$consumes 1, case-names empty insert$]:
 $\llbracket finite A; P \{\}; \bigwedge b A. \llbracket finite A; \forall a \in A. b < a; P A \rrbracket \implies P (insert b A) \rrbracket \implies P A$
by ($rule linorder.finite-linorder-max-induct [OF dual-linorder]$)

lemma $finite-ranking-induct$ [$consumes 1, case-names empty insert$]:
fixes $f :: 'b \Rightarrow 'a$
assumes $finite S$
assumes $P \{\}$
assumes $\bigwedge x S. finite S \implies (\bigwedge y. y \in S \implies f y \leq f x) \implies P S \implies P (insert x S)$
shows $P S$
using $\langle finite S \rangle$
proof ($induction rule: finite-psubset-induct$)
case ($psubset A$)
{
assume $A \neq \{\}$
hence $f ' A \neq \{\}$ **and** $finite (f ' A)$
using $psubset finite-image-iff$ **by simp+**
then obtain a **where** $f a = Max (f ' A)$ **and** $a \in A$
by ($metis Max-in[of f ' A] imageE$)
then have $P (A - \{a\})$
using $psubset member-remove$ **by blast**
moreover
have $\bigwedge y. y \in A \implies f y \leq f a$
using $\langle f a = Max (f ' A) \rangle \langle finite (f ' A) \rangle$ **by simp**

```

ultimately
have ?case
  by (metis ‹a ∈ A› DiffD1 insert-Diff assms(3) finite-Diff psubset.hyps)
}
thus ?case
  using assms(2) by blast
qed

```

lemma *Least-Min*:

```

assumes finite {a. P a} and ∃ a. P a
shows (LEAST a. P a) = Min {a. P a}
proof -
{ fix A :: 'a set
  assume A: finite A A ≠ {}
  have (LEAST a. a ∈ A) = Min A
  using A proof (induct A rule: finite-ne-induct)
    case singleton show ?case by (rule Least-equality) simp-all
  next
    case (insert a A)
    have (LEAST b. b = a ∨ b ∈ A) = min a (LEAST a. a ∈ A)
    by (auto intro!: Least-equality simp add: min-def not-le Min-le-iff insert.hyps
dest!: less-imp-le)
    with insert show ?case by simp
  qed
} from this [of {a. P a}] assms show ?thesis by simp
qed

```

lemma *infinite-growing*:

```

assumes X ≠ {}
assumes *: ∧x. x ∈ X ⇒ ∃ y∈X. y > x
shows ¬ finite X
proof
  assume finite X
  with ‹X ≠ {}› have Max X ∈ X ∨ x∈X. x ≤ Max X
  by auto
  with *[of Max X] show False
  by auto
qed

```

end

lemma *sum-le-card-Max*: $\text{finite } A \implies \text{sum } f A \leq \text{card } A * \text{Max } (f \text{ ' } A)$
using *sum-bounded-above*[of A f Max (f ' A)] **by** *simp*

lemma *card-Min-le-sum*: $\text{finite } A \implies \text{card } A * \text{Min } (f \text{ ' } A) \leq \text{sum } f A$
using *sum-bounded-below*[of A Min (f ' A) f] **by** *simp*

context *linordered-ab-semigroup-add*
begin

lemma *Min-add-commute*:

fixes k

assumes *finite S and* $S \neq \{\}$

shows $\text{Min } ((\lambda x. f x + k) ' S) = \text{Min}(f ' S) + k$

proof –

have $m: \bigwedge x y. \text{min } x y + k = \text{min } (x+k) (y+k)$

by (*simp add: min-def order.antisym add-right-mono*)

have $(\lambda x. f x + k) ' S = (\lambda y. y + k) ' (f ' S)$ **by** *auto*

also have $\text{Min } \dots = \text{Min } (f ' S) + k$

using *assms hom-Min-commute [of $\lambda y. y+k f ' S$, OF m , symmetric]* **by** *simp*

finally show *?thesis* **by** *simp*

qed

lemma *Max-add-commute*:

fixes k

assumes *finite S and* $S \neq \{\}$

shows $\text{Max } ((\lambda x. f x + k) ' S) = \text{Max}(f ' S) + k$

proof –

have $m: \bigwedge x y. \text{max } x y + k = \text{max } (x+k) (y+k)$

by (*simp add: max-def order.antisym add-right-mono*)

have $(\lambda x. f x + k) ' S = (\lambda y. y + k) ' (f ' S)$ **by** *auto*

also have $\text{Max } \dots = \text{Max } (f ' S) + k$

using *assms hom-Max-commute [of $\lambda y. y+k f ' S$, OF m , symmetric]* **by** *simp*

finally show *?thesis* **by** *simp*

qed

end

context *linordered-ab-group-add*

begin

lemma *minus-Max-eq-Min* [*simp*]:

finite S $\implies S \neq \{\} \implies - \text{Max } S = \text{Min } (\text{uminus } ' S)$

by (*induct S rule: finite-ne-induct*) (*simp-all add: minus-max-eq-min*)

lemma *minus-Min-eq-Max* [*simp*]:

finite S $\implies S \neq \{\} \implies - \text{Min } S = \text{Max } (\text{uminus } ' S)$

by (*induct S rule: finite-ne-induct*) (*simp-all add: minus-min-eq-max*)

end

context *complete-linorder*

begin

lemma *Min-Inf*:

assumes *finite A and* $A \neq \{\}$

shows $\text{Min } A = \text{Inf } A$

proof –

from *assms* **obtain** $b\ B$ **where** $A = \text{insert } b\ B$ **and** *finite* B **by** *auto*
then show *?thesis*
by (*simp add: Min.eq-fold complete-linorder-inf-min [symmetric] inf-Inf-fold-inf*
inf commute [of b])
qed

lemma *Max-Sup*:

assumes *finite* A **and** $A \neq \{\}$
shows $\text{Max } A = \text{Sup } A$

proof –

from *assms* **obtain** $b\ B$ **where** $A = \text{insert } b\ B$ **and** *finite* B **by** *auto*
then show *?thesis*
by (*simp add: Max.eq-fold complete-linorder-sup-max [symmetric] sup-Sup-fold-sup*
sup commute [of b])
qed

end

lemma *disjnt-ge-max*:

$\langle \text{disjnt } X\ Y \rangle$ **if** $\langle \text{finite } Y \rangle$ $\langle \bigwedge x. x \in X \implies x > \text{Max } Y \rangle$
using that by (*auto simp add: disjnt-def*) (*use Max-less-iff in blast*)

55.5 Arg Min

context *ord*

begin

definition *is-arg-min* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'b \Rightarrow \text{bool}$ **where**
 $\text{is-arg-min } f\ P\ x = (P\ x \wedge \neg(\exists y. P\ y \wedge f\ y < f\ x))$

definition *arg-min* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'b$ **where**
 $\text{arg-min } f\ P = (\text{SOME } x. \text{is-arg-min } f\ P\ x)$

definition *arg-min-on* :: $('b \Rightarrow 'a) \Rightarrow 'b\ \text{set} \Rightarrow 'b$ **where**
 $\text{arg-min-on } f\ S = \text{arg-min } f\ (\lambda x. x \in S)$

end

syntax

$\text{-arg-min} :: ('b \Rightarrow 'a) \Rightarrow \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'b$
 $((\text{ARG}'\text{-MIN} - \text{./ } -) [1000, 0, 10] 10)$

translations

$\text{ARG-MIN } f\ x. P \rightleftharpoons \text{CONST } \text{arg-min } f\ (\lambda x. P)$

lemma *is-arg-min-linorder*: **fixes** $f :: 'a \Rightarrow 'b :: \text{linorder}$
shows $\text{is-arg-min } f\ P\ x = (P\ x \wedge (\forall y. P\ y \longrightarrow f\ x \leq f\ y))$
by(*auto simp add: is-arg-min-def*)

lemma *is-arg-min-antimono*: **fixes** $f :: 'a \Rightarrow ('b :: \text{order})$

shows $\llbracket \text{is-arg-min } f P x; f y \leq f x; P y \rrbracket \implies \text{is-arg-min } f P y$
by (*simp add: order.order-iff-strict is-arg-min-def*)

lemma *arg-minI*:

$\llbracket P x;$
 $\quad \bigwedge y. P y \implies \neg f y < f x;$
 $\quad \bigwedge x. \llbracket P x; \forall y. P y \longrightarrow \neg f y < f x \rrbracket \implies Q x \rrbracket$
 $\implies Q (\text{arg-min } f P)$
apply (*simp add: arg-min-def is-arg-min-def*)
apply (*rule someI2-ex*)
apply *blast*
apply *blast*
done

lemma *arg-min-equality*:

$\llbracket P k; \bigwedge x. P x \implies f k \leq f x \rrbracket \implies f (\text{arg-min } f P) = f k$
for $f :: - \Rightarrow 'a::\text{order}$
apply (*rule arg-minI*)
apply *assumption*
apply (*simp add: less-le-not-le*)
by (*metis le-less*)

lemma *wf-linord-ex-has-least*:

$\llbracket \text{wf } r; \forall x y. (x, y) \in r^+ \longleftrightarrow (y, x) \notin r^*; P k \rrbracket$
 $\implies \exists x. P x \wedge (\forall y. P y \longrightarrow (m x, m y) \in r^*)$
apply (*drule wf-trancl [THEN wf-eq-minimal [THEN iffD1]]*)
apply (*drule-tac x = m ' Collect P in spec*)
by *force*

lemma *ex-has-least-nat*: $P k \implies \exists x. P x \wedge (\forall y. P y \longrightarrow m x \leq m y)$

for $m :: 'a \Rightarrow \text{nat}$
apply (*simp only: pred-nat-trancl-eq-le [symmetric]*)
apply (*rule wf-pred-nat [THEN wf-linord-ex-has-least]*)
apply (*simp add: less-eq linorder-not-le pred-nat-trancl-eq-le*)
by *assumption*

lemma *arg-min-nat-lemma*:

$P k \implies P(\text{arg-min } m P) \wedge (\forall y. P y \longrightarrow m (\text{arg-min } m P) \leq m y)$
for $m :: 'a \Rightarrow \text{nat}$
apply (*simp add: arg-min-def is-arg-min-linorder*)
apply (*rule someI-ex*)
apply (*erule ex-has-least-nat*)
done

lemmas *arg-min-natI = arg-min-nat-lemma [THEN conjunct1]*

lemma *is-arg-min-arg-min-nat*: **fixes** $m :: 'a \Rightarrow \text{nat}$

shows $P x \implies \text{is-arg-min } m P (\text{arg-min } m P)$
by (*metis arg-min-nat-lemma is-arg-min-linorder*)

lemma *arg-min-nat-le*: $P\ x \implies m\ (\text{arg-min}\ m\ P) \leq m\ x$
for $m :: 'a \Rightarrow \text{nat}$
by (rule *arg-min-nat-lemma* [THEN *conjunct2*, THEN *spec*, THEN *mp*])

lemma *ex-min-if-finite*:
 $\llbracket \text{finite}\ S; S \neq \{\} \rrbracket \implies \exists m \in S. \neg(\exists x \in S. x < (m :: 'a :: \text{order}))$
by(*induction rule: finite.induct*) (*auto intro: order.strict-trans*)

lemma *ex-is-arg-min-if-finite*: **fixes** $f :: 'a \Rightarrow 'b :: \text{order}$
shows $\llbracket \text{finite}\ S; S \neq \{\} \rrbracket \implies \exists x. \text{is-arg-min}\ f\ (\lambda x. x \in S)\ x$
unfolding *is-arg-min-def*
using *ex-min-if-finite*[of $f\ 'S$]
by *auto*

lemma *arg-min-SOME-Min*:
 $\text{finite}\ S \implies \text{arg-min-on}\ f\ S = (\text{SOME}\ y. y \in S \wedge f\ y = \text{Min}(f\ 'S))$
unfolding *arg-min-on-def arg-min-def is-arg-min-linorder*
apply(rule *arg-cong*[**where** $f = \text{Eps}$])
apply (*auto simp: fun-eq-iff intro: Min-eqI[symmetric]*)
done

lemma *arg-min-if-finite*: **fixes** $f :: 'a \Rightarrow 'b :: \text{order}$
assumes $\text{finite}\ S\ S \neq \{\}$
shows $\text{arg-min-on}\ f\ S \in S$ **and** $\neg(\exists x \in S. f\ x < f\ (\text{arg-min-on}\ f\ S))$
using *ex-is-arg-min-if-finite*[OF *assms*, of f]
unfolding *arg-min-on-def arg-min-def is-arg-min-def*
by(*auto dest!: someI-ex*)

lemma *arg-min-least*: **fixes** $f :: 'a \Rightarrow 'b :: \text{linorder}$
shows $\llbracket \text{finite}\ S; S \neq \{\}; y \in S \rrbracket \implies f(\text{arg-min-on}\ f\ S) \leq f\ y$
by(*simp add: arg-min-SOME-Min inv-into-def2[symmetric] f-inv-into-f*)

lemma *arg-min-inj-eq*: **fixes** $f :: 'a \Rightarrow 'b :: \text{order}$
shows $\llbracket \text{inj-on}\ f\ \{x. P\ x\}; P\ a; \forall y. P\ y \longrightarrow f\ a \leq f\ y \rrbracket \implies \text{arg-min}\ f\ P = a$
apply(*simp add: arg-min-def is-arg-min-def*)
apply(rule *someI2*[of a])
apply (*simp add: less-le-not-le*)
by (*metis inj-on-eq-iff less-le mem-Collect-eq*)

55.6 Arg Max

context *ord*
begin

definition *is-arg-max* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'b \Rightarrow \text{bool}$ **where**
 $\text{is-arg-max}\ f\ P\ x = (P\ x \wedge \neg(\exists y. P\ y \wedge f\ y > f\ x))$

definition *arg-max* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'b$ **where**

$arg-max f P = (SOME x. is-arg-max f P x)$

definition $arg-max-on :: ('b \Rightarrow 'a) \Rightarrow 'b set \Rightarrow 'b$ **where**
 $arg-max-on f S = arg-max f (\lambda x. x \in S)$

end

syntax

$-arg-max :: ('b \Rightarrow 'a) \Rightarrow pttrn \Rightarrow bool \Rightarrow 'a$
 $((\exists ARG'-MAX - ./ -) [1000, 0, 10] 10)$

translations

$ARG-MAX f x. P \Rightarrow CONST arg-max f (\lambda x. P)$

lemma $is-arg-max-linorder$: **fixes** $f :: 'a \Rightarrow 'b :: linorder$
shows $is-arg-max f P x = (P x \wedge (\forall y. P y \longrightarrow f x \geq f y))$
by($auto simp add: is-arg-max-def$)

lemma $arg-maxI$:

$P x \Longrightarrow$
 $(\bigwedge y. P y \Longrightarrow \neg f y > f x) \Longrightarrow$
 $(\bigwedge x. P x \Longrightarrow \forall y. P y \longrightarrow \neg f y > f x \Longrightarrow Q x) \Longrightarrow$
 $Q (arg-max f P)$

apply ($simp add: arg-max-def is-arg-max-def$)

apply ($rule someI2-ex$)

apply $blast$

apply $blast$

done

lemma $arg-max-equality$:

$\llbracket P k; \bigwedge x. P x \Longrightarrow f x \leq f k \rrbracket \Longrightarrow f (arg-max f P) = f k$
for $f :: - \Rightarrow 'a :: order$

apply ($rule arg-maxI$ [**where** $f = f$])

apply $assumption$

apply ($simp add: less-le-not-le$)

by ($metis le-less$)

lemma $ex-has-greatest-nat-lemma$:

$P k \Longrightarrow \forall x. P x \longrightarrow (\exists y. P y \wedge \neg f y \leq f x) \Longrightarrow \exists y. P y \wedge \neg f y < f k + n$
for $f :: 'a \Rightarrow nat$

by ($induct n$) ($force simp: le-Suc-eq$) $+$

lemma $ex-has-greatest-nat$:

assumes $P k$

and $\forall y. P y \longrightarrow (f :: 'a \Rightarrow nat) y < b$

shows $\exists x. P x \wedge (\forall y. P y \longrightarrow f y \leq f x)$

proof ($rule ccontr$)

assume $\nexists x. P x \wedge (\forall y. P y \longrightarrow f y \leq f x)$

then have $\forall x. P x \longrightarrow (\exists y. P y \wedge \neg f y \leq f x)$

by $auto$

```

then have  $\exists y. P y \wedge \neg f y < f k + (b - f k)$ 
  using assms ex-has-greatest-nat-lemma[of P k f b - f k]
  by blast
then show False
  using assms by auto
qed

```

```

lemma arg-max-nat-lemma:
   $\llbracket P k; \forall y. P y \longrightarrow f y < b \rrbracket$ 
   $\implies P (\text{arg-max } f P) \wedge (\forall y. P y \longrightarrow f y \leq f (\text{arg-max } f P))$ 
  for  $f :: 'a \Rightarrow \text{nat}$ 
apply (simp add: arg-max-def is-arg-max-linorder)
apply (rule someI-ex)
apply (erule (1) ex-has-greatest-nat)
done

```

```

lemmas arg-max-natI = arg-max-nat-lemma [THEN conjunct1]

```

```

lemma arg-max-nat-le:  $P x \implies \forall y. P y \longrightarrow f y < b \implies f x \leq f (\text{arg-max } f P)$ 
  for  $f :: 'a \Rightarrow \text{nat}$ 
by (blast dest: arg-max-nat-lemma [THEN conjunct2, THEN spec, of P])

```

```

end

```

56 Division in euclidean (semi)rings

```

theory Euclidean-Rings
  imports Int Lattices-Big
begin

```

56.1 Euclidean (semi)rings with explicit division and remainder

```

class euclidean-semiring = semidom-modulo +
  fixes euclidean-size :: 'a  $\Rightarrow$  nat
  assumes size-0 [simp]: euclidean-size 0 = 0
  assumes mod-size-less:
     $b \neq 0 \implies \text{euclidean-size } (a \text{ mod } b) < \text{euclidean-size } b$ 
  assumes size-mult-mono:
     $b \neq 0 \implies \text{euclidean-size } a \leq \text{euclidean-size } (a * b)$ 
begin

```

```

lemma euclidean-size-eq-0-iff [simp]:
   $\text{euclidean-size } b = 0 \longleftrightarrow b = 0$ 
proof
  assume  $b = 0$ 
  then show  $\text{euclidean-size } b = 0$ 
    by simp
next

```

```

assume euclidean-size b = 0
show b = 0
proof (rule ccontr)
  assume b ≠ 0
  with mod-size-less have euclidean-size (b mod b) < euclidean-size b .
  with ⟨euclidean-size b = 0⟩ show False
    by simp
qed
qed

```

```

lemma euclidean-size-greater-0-iff [simp]:
  euclidean-size b > 0 ↔ b ≠ 0
  using euclidean-size-eq-0-iff [symmetric, of b] by safe simp

```

```

lemma size-mult-mono': b ≠ 0 ⇒ euclidean-size a ≤ euclidean-size (b * a)
  by (subst mult.commute) (rule size-mult-mono)

```

```

lemma dvd-euclidean-size-eq-imp-dvd:
  assumes a ≠ 0 and euclidean-size a = euclidean-size b
  and b dvd a
  shows a dvd b
proof (rule ccontr)
  assume ¬ a dvd b
  hence b mod a ≠ 0 using mod-0-imp-dvd [of b a] by blast
  then have b mod a ≠ 0 by (simp add: mod-eq-0-iff-dvd)
  from ⟨b dvd a⟩ have b dvd b mod a by (simp add: dvd-mod-iff)
  then obtain c where b mod a = b * c unfolding dvd-def by blast
  with ⟨b mod a ≠ 0⟩ have c ≠ 0 by auto
  with ⟨b mod a = b * c⟩ have euclidean-size (b mod a) ≥ euclidean-size b
  using size-mult-mono by force
  moreover from ⟨¬ a dvd b⟩ and ⟨a ≠ 0⟩
  have euclidean-size (b mod a) < euclidean-size a
  using mod-size-less by blast
  ultimately show False using ⟨euclidean-size a = euclidean-size b⟩
  by simp
qed

```

```

lemma euclidean-size-times-unit:
  assumes is-unit a
  shows euclidean-size (a * b) = euclidean-size b
proof (rule antisym)
  from assms have [simp]: a ≠ 0 by auto
  thus euclidean-size (a * b) ≥ euclidean-size b by (rule size-mult-mono')
  from assms have is-unit (1 div a) by simp
  hence 1 div a ≠ 0 by (intro notI) simp-all
  hence euclidean-size (a * b) ≤ euclidean-size ((1 div a) * (a * b))
  by (rule size-mult-mono')
  also from assms have (1 div a) * (a * b) = b
  by (simp add: algebra-simps unit-div-mult-swap)

```

finally show $\text{euclidean-size } (a * b) \leq \text{euclidean-size } b$.
qed

lemma *euclidean-size-unit*:
 $\text{is-unit } a \implies \text{euclidean-size } a = \text{euclidean-size } 1$
using *euclidean-size-times-unit* [of a 1] **by** *simp*

lemma *unit-iff-euclidean-size*:
 $\text{is-unit } a \iff \text{euclidean-size } a = \text{euclidean-size } 1 \wedge a \neq 0$
proof *safe*
assume $A: a \neq 0$ **and** $B: \text{euclidean-size } a = \text{euclidean-size } 1$
show $\text{is-unit } a$
by (*rule dvd-euclidean-size-eq-imp-dvd* [OF A B]) *simp-all*
qed (*auto intro: euclidean-size-unit*)

lemma *euclidean-size-times-nonunit*:
assumes $a \neq 0 \wedge b \neq 0 \wedge \neg \text{is-unit } a$
shows $\text{euclidean-size } b < \text{euclidean-size } (a * b)$
proof (*rule ccontr*)
assume $\neg \text{euclidean-size } b < \text{euclidean-size } (a * b)$
with *size-mult-mono*'[OF *assms*(1), of b]
have $\text{eq}: \text{euclidean-size } (a * b) = \text{euclidean-size } b$ **by** *simp*
have $a * b \text{ dvd } b$
by (*rule dvd-euclidean-size-eq-imp-dvd* [OF - eq])
(*use* *assms* **in** *simp-all*)
hence $a * b \text{ dvd } 1 * b$ **by** *simp*
with $\langle b \neq 0 \rangle$ **have** $\text{is-unit } a$ **by** (*subst* (*asm*) *dvd-times-right-cancel-iff*)
with *assms*(3) **show** *False* **by** *contradiction*
qed

lemma *dvd-imp-size-le*:
assumes $a \text{ dvd } b \wedge b \neq 0$
shows $\text{euclidean-size } a \leq \text{euclidean-size } b$
using *assms* **by** (*auto simp: size-mult-mono*)

lemma *dvd-proper-imp-size-less*:
assumes $a \text{ dvd } b \wedge \neg b \text{ dvd } a \wedge b \neq 0$
shows $\text{euclidean-size } a < \text{euclidean-size } b$
proof –
from *assms*(1) **obtain** c **where** $b = a * c$ **by** (*erule dvdE*)
hence $z: b = c * a$ **by** (*simp add: mult.commute*)
from z *assms* **have** $\neg \text{is-unit } c$ **by** (*auto simp: mult.commute mult-unit-dvd-iff*)
with z *assms* **show** *?thesis*
by (*auto intro!: euclidean-size-times-nonunit*)
qed

lemma *unit-imp-mod-eq-0*:
 $a \text{ mod } b = 0$ **if** $\text{is-unit } b$
using *that* **by** (*simp add: mod-eq-0-iff-dvd unit-imp-dvd*)

lemma *mod-eq-self-iff-div-eq-0*:
 $a \bmod b = a \iff a \operatorname{div} b = 0$ (**is** $?P \iff ?Q$)

proof

assume $?P$
with *div-mult-mod-eq* [of a b] **show** $?Q$
 by *auto*

next

assume $?Q$
with *div-mult-mod-eq* [of a b] **show** $?P$
 by *simp*

qed

lemma *coprime-mod-left-iff* [*simp*]:

$\operatorname{coprime} (a \bmod b) b \iff \operatorname{coprime} a b$ **if** $b \neq 0$

by (*rule iffI*; *rule coprimeI*)

(*use that in* \langle *auto dest!*: *dvd-mod-imp-dvd coprime-common-divisor simp add*:
dvd-mod-iff \rangle)

lemma *coprime-mod-right-iff* [*simp*]:

$\operatorname{coprime} a (b \bmod a) \iff \operatorname{coprime} a b$ **if** $a \neq 0$

using *that coprime-mod-left-iff* [of a b] **by** (*simp add*: *ac-simps*)

end

class *euclidean-ring* = *idom-modulo* + *euclidean-semiring*

begin

lemma *dvd-diff-commute* [*ac-simps*]:

$a \operatorname{dvd} c - b \iff a \operatorname{dvd} b - c$

proof –

have $a \operatorname{dvd} c - b \iff a \operatorname{dvd} (c - b) * - 1$

by (*subst dvd-mult-unit-iff*) *simp-all*

then show $?thesis$

by *simp*

qed

end

56.2 Euclidean (semi)rings with cancel rules

class *euclidean-semiring-cancel* = *euclidean-semiring* +

assumes *div-mult-self1* [*simp*]: $b \neq 0 \implies (a + c * b) \operatorname{div} b = c + a \operatorname{div} b$

and *div-mult-mult1* [*simp*]: $c \neq 0 \implies (c * a) \operatorname{div} (c * b) = a \operatorname{div} b$

begin

lemma *div-mult-self2* [*simp*]:

assumes $b \neq 0$

shows $(a + b * c) \operatorname{div} b = c + a \operatorname{div} b$

using *assms div-mult-self1 [of b a c]* **by** (*simp add: mult.commute*)

lemma *div-mult-self3 [simp]*:

assumes $b \neq 0$

shows $(c * b + a) \text{ div } b = c + a \text{ div } b$

using *assms* **by** (*simp add: add.commute*)

lemma *div-mult-self4 [simp]*:

assumes $b \neq 0$

shows $(b * c + a) \text{ div } b = c + a \text{ div } b$

using *assms* **by** (*simp add: add.commute*)

lemma *mod-mult-self1 [simp]*: $(a + c * b) \text{ mod } b = a \text{ mod } b$

proof (*cases b = 0*)

case *True* **then show** *?thesis* **by** *simp*

next

case *False*

have $a + c * b = (a + c * b) \text{ div } b * b + (a + c * b) \text{ mod } b$

by (*simp add: div-mult-mod-eq*)

also from *False div-mult-self1 [of b a c]* **have**

$\dots = (c + a \text{ div } b) * b + (a + c * b) \text{ mod } b$

by (*simp add: algebra-simps*)

finally have $a = a \text{ div } b * b + (a + c * b) \text{ mod } b$

by (*simp add: add.commute [of a] add.assoc distrib-right*)

then have $a \text{ div } b * b + (a + c * b) \text{ mod } b = a \text{ div } b * b + a \text{ mod } b$

by (*simp add: div-mult-mod-eq*)

then show *?thesis* **by** *simp*

qed

lemma *mod-mult-self2 [simp]*:

$(a + b * c) \text{ mod } b = a \text{ mod } b$

by (*simp add: mult.commute [of b]*)

lemma *mod-mult-self3 [simp]*:

$(c * b + a) \text{ mod } b = a \text{ mod } b$

by (*simp add: add.commute*)

lemma *mod-mult-self4 [simp]*:

$(b * c + a) \text{ mod } b = a \text{ mod } b$

by (*simp add: add.commute*)

lemma *mod-mult-self1-is-0 [simp]*:

$b * a \text{ mod } b = 0$

using *mod-mult-self2 [of 0 b a]* **by** *simp*

lemma *mod-mult-self2-is-0 [simp]*:

$a * b \text{ mod } b = 0$

using *mod-mult-self1 [of 0 a b]* **by** *simp*

lemma *div-add-self1*:
 assumes $b \neq 0$
 shows $(b + a) \operatorname{div} b = a \operatorname{div} b + 1$
 using *assms div-mult-self1 [of b a 1]* **by** (*simp add: add.commute*)

lemma *div-add-self2*:
 assumes $b \neq 0$
 shows $(a + b) \operatorname{div} b = a \operatorname{div} b + 1$
 using *assms div-add-self1 [of b a]* **by** (*simp add: add.commute*)

lemma *mod-add-self1 [simp]*:
 $(b + a) \operatorname{mod} b = a \operatorname{mod} b$
 using *mod-mult-self1 [of a 1 b]* **by** (*simp add: add.commute*)

lemma *mod-add-self2 [simp]*:
 $(a + b) \operatorname{mod} b = a \operatorname{mod} b$
 using *mod-mult-self1 [of a 1 b]* **by** *simp*

lemma *mod-div-trivial [simp]*:
 $a \operatorname{mod} b \operatorname{div} b = 0$
proof (*cases b = 0*)
 assume $b = 0$
 thus *?thesis* **by** *simp*
next
 assume $b \neq 0$
 hence $a \operatorname{div} b + a \operatorname{mod} b \operatorname{div} b = (a \operatorname{mod} b + a \operatorname{div} b * b) \operatorname{div} b$
 by (*rule div-mult-self1 [symmetric]*)
 also have $\dots = a \operatorname{div} b$
 by (*simp only: mod-div-mult-eq*)
 also have $\dots = a \operatorname{div} b + 0$
 by *simp*
 finally **show** *?thesis*
 by (*rule add-left-imp-eq*)
qed

lemma *mod-mod-trivial [simp]*:
 $a \operatorname{mod} b \operatorname{mod} b = a \operatorname{mod} b$
proof –
 have $a \operatorname{mod} b \operatorname{mod} b = (a \operatorname{mod} b + a \operatorname{div} b * b) \operatorname{mod} b$
 by (*simp only: mod-mult-self1*)
 also have $\dots = a \operatorname{mod} b$
 by (*simp only: mod-div-mult-eq*)
 finally **show** *?thesis* .
qed

lemma *mod-mod-cancel*:
 assumes $c \operatorname{dvd} b$
 shows $a \operatorname{mod} b \operatorname{mod} c = a \operatorname{mod} c$
proof –

from $\langle c \text{ dvd } b \rangle$ **obtain** k **where** $b = c * k$
by (*rule dvdE*)
have $a \text{ mod } b \text{ mod } c = a \text{ mod } (c * k) \text{ mod } c$
by (*simp only: $\langle b = c * k \rangle$*)
also have $\dots = (a \text{ mod } (c * k) + a \text{ div } (c * k) * k * c) \text{ mod } c$
by (*simp only: mod-mult-self1*)
also have $\dots = (a \text{ div } (c * k) * (c * k) + a \text{ mod } (c * k)) \text{ mod } c$
by (*simp only: ac-simps*)
also have $\dots = a \text{ mod } c$
by (*simp only: div-mult-mod-eq*)
finally show *?thesis* .
qed

lemma *div-mult-mult2* [*simp*]:
 $c \neq 0 \implies (a * c) \text{ div } (b * c) = a \text{ div } b$
by (*drule div-mult-mult1*) (*simp add: mult.commute*)

lemma *div-mult-mult1-if* [*simp*]:
 $(c * a) \text{ div } (c * b) = (\text{if } c = 0 \text{ then } 0 \text{ else } a \text{ div } b)$
by *simp-all*

lemma *mod-mult-mult1*:
 $(c * a) \text{ mod } (c * b) = c * (a \text{ mod } b)$
proof (*cases c = 0*)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
from *div-mult-mod-eq*
have $((c * a) \text{ div } (c * b)) * (c * b) + (c * a) \text{ mod } (c * b) = c * a$.
with *False* **have** $c * ((a \text{ div } b) * b + a \text{ mod } b) + (c * a) \text{ mod } (c * b)$
 $= c * a + c * (a \text{ mod } b)$ **by** (*simp add: algebra-simps*)
with *div-mult-mod-eq* **show** *?thesis* **by** *simp*
qed

lemma *mod-mult-mult2*:
 $(a * c) \text{ mod } (b * c) = (a \text{ mod } b) * c$
using *mod-mult-mult1* [*of c a b*] **by** (*simp add: mult.commute*)

lemma *mult-mod-left*: $(a \text{ mod } b) * c = (a * c) \text{ mod } (b * c)$
by (*fact mod-mult-mult2*) [*symmetric*]

lemma *mult-mod-right*: $c * (a \text{ mod } b) = (c * a) \text{ mod } (c * b)$
by (*fact mod-mult-mult1*) [*symmetric*]

lemma *dvd-mod*: $k \text{ dvd } m \implies k \text{ dvd } n \implies k \text{ dvd } (m \text{ mod } n)$
unfolding *dvd-def* **by** (*auto simp add: mod-mult-mult1*)

lemma *div-plus-div-distrib-dvd-left*:
 $c \text{ dvd } a \implies (a + b) \text{ div } c = a \text{ div } c + b \text{ div } c$

by (cases $c = 0$) auto

lemma *div-plus-div-distrib-dvd-right*:

$c \text{ dvd } b \implies (a + b) \text{ div } c = a \text{ div } c + b \text{ div } c$

using *div-plus-div-distrib-dvd-left* [of $c \ b \ a$]

by (*simp add: ac-simps*)

lemma *sum-div-partition*:

$\langle (\sum a \in A. f \ a) \text{ div } b = (\sum a \in A \cap \{a. b \text{ dvd } f \ a\}. f \ a \text{ div } b) + (\sum a \in A \cap \{a. \neg b \text{ dvd } f \ a\}. f \ a) \text{ div } b \rangle$

if $\langle \text{finite } A \rangle$

proof –

have $\langle A = A \cap \{a. b \text{ dvd } f \ a\} \cup A \cap \{a. \neg b \text{ dvd } f \ a\} \rangle$

by *auto*

then have $\langle (\sum a \in A. f \ a) = (\sum a \in A \cap \{a. b \text{ dvd } f \ a\} \cup A \cap \{a. \neg b \text{ dvd } f \ a\}. f \ a) \rangle$

by *simp*

also have $\langle \dots = (\sum a \in A \cap \{a. b \text{ dvd } f \ a\}. f \ a) + (\sum a \in A \cap \{a. \neg b \text{ dvd } f \ a\}. f \ a) \rangle$

using $\langle \text{finite } A \rangle$ by (*auto intro: sum.union-inter-neutral*)

finally have $\ast: \langle \text{sum } f \ A = \text{sum } f \ (A \cap \{a. b \text{ dvd } f \ a\}) + \text{sum } f \ (A \cap \{a. \neg b \text{ dvd } f \ a\}) \rangle$.

define B where $B: \langle B = A \cap \{a. b \text{ dvd } f \ a\} \rangle$

with $\langle \text{finite } A \rangle$ have $\langle \text{finite } B \rangle$ and $\langle a \in B \implies b \text{ dvd } f \ a \rangle$ for a

by *simp-all*

then have $\langle (\sum a \in B. f \ a) \text{ div } b = (\sum a \in B. f \ a \text{ div } b) \rangle$ and $\langle b \text{ dvd } (\sum a \in B. f \ a) \rangle$

by *induction (simp-all add: div-plus-div-distrib-dvd-left)*

then show *?thesis* using \ast

by (*simp add: B div-plus-div-distrib-dvd-left*)

qed

named-theorems *mod-simps*

Addition respects modular equivalence.

lemma *mod-add-left-eq* [*mod-simps*]:

$(a \text{ mod } c + b) \text{ mod } c = (a + b) \text{ mod } c$

proof –

have $(a + b) \text{ mod } c = (a \text{ div } c \ast c + a \text{ mod } c + b) \text{ mod } c$

by (*simp only: div-mult-mod-eq*)

also have $\dots = (a \text{ mod } c + b + a \text{ div } c \ast c) \text{ mod } c$

by (*simp only: ac-simps*)

also have $\dots = (a \text{ mod } c + b) \text{ mod } c$

by (*rule mod-mult-self1*)

finally show *?thesis*

by (*rule sym*)

qed

lemma *mod-add-right-eq* [*mod-simps*]:

$(a + b \text{ mod } c) \text{ mod } c = (a + b) \text{ mod } c$

using *mod-add-left-eq* [of *b c a*] **by** (*simp add: ac-simps*)

lemma *mod-add-eq*:

$(a \bmod c + b \bmod c) \bmod c = (a + b) \bmod c$

by (*simp add: mod-add-left-eq mod-add-right-eq*)

lemma *mod-sum-eq* [*mod-simps*]:

$(\sum_{i \in A} f i \bmod a) \bmod a = \text{sum } f A \bmod a$

proof (*induct A rule: infinite-finite-induct*)

case (*insert i A*)

then have $(\sum_{i \in \text{insert } i A} f i \bmod a) \bmod a$

$= (f i \bmod a + (\sum_{i \in A} f i \bmod a)) \bmod a$

by *simp*

also have $\dots = (f i + (\sum_{i \in A} f i \bmod a) \bmod a) \bmod a$

by (*simp add: mod-simps*)

also have $\dots = (f i + (\sum_{i \in A} f i) \bmod a) \bmod a$

by (*simp add: insert.hyps*)

finally show *?case*

by (*simp add: insert.hyps mod-simps*)

qed *simp-all*

lemma *mod-add-cong*:

assumes $a \bmod c = a' \bmod c$

assumes $b \bmod c = b' \bmod c$

shows $(a + b) \bmod c = (a' + b') \bmod c$

proof –

have $(a \bmod c + b \bmod c) \bmod c = (a' \bmod c + b' \bmod c) \bmod c$

unfolding *assms ..*

then show *?thesis*

by (*simp add: mod-add-eq*)

qed

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq* [*mod-simps*]:

$((a \bmod c) * b) \bmod c = (a * b) \bmod c$

proof –

have $(a * b) \bmod c = ((a \text{ div } c * c + a \bmod c) * b) \bmod c$

by (*simp only: div-mult-mod-eq*)

also have $\dots = (a \bmod c * b + a \text{ div } c * b * c) \bmod c$

by (*simp only: algebra-simps*)

also have $\dots = (a \bmod c * b) \bmod c$

by (*rule mod-mult-self1*)

finally show *?thesis*

by (*rule sym*)

qed

lemma *mod-mult-right-eq* [*mod-simps*]:

$(a * (b \bmod c)) \bmod c = (a * b) \bmod c$

using *mod-mult-left-eq* [of *b c a*] **by** (*simp add: ac-simps*)

lemma *mod-mult-eq*:

$((a \bmod c) * (b \bmod c)) \bmod c = (a * b) \bmod c$
by (*simp add: mod-mult-left-eq mod-mult-right-eq*)

lemma *mod-prod-eq* [*mod-simps*]:

$(\prod_{i \in A} f i \bmod a) \bmod a = \text{prod } f A \bmod a$
proof (*induct A rule: infinite-finite-induct*)
case (*insert i A*)
then have $(\prod_{i \in \text{insert } i A} f i \bmod a) \bmod a$
 $= (f i \bmod a * (\prod_{i \in A} f i \bmod a)) \bmod a$
by *simp*
also have $\dots = (f i * ((\prod_{i \in A} f i \bmod a) \bmod a)) \bmod a$
by (*simp add: mod-simps*)
also have $\dots = (f i * ((\prod_{i \in A} f i) \bmod a)) \bmod a$
by (*simp add: insert.hyps*)
finally show *?case*
by (*simp add: insert.hyps mod-simps*)
qed *simp-all*

lemma *mod-mult-cong*:

assumes $a \bmod c = a' \bmod c$
assumes $b \bmod c = b' \bmod c$
shows $(a * b) \bmod c = (a' * b') \bmod c$
proof –
have $(a \bmod c * (b \bmod c)) \bmod c = (a' \bmod c * (b' \bmod c)) \bmod c$
unfolding *assms ..*
then show *?thesis*
by (*simp add: mod-mult-eq*)
qed

Exponentiation respects modular equivalence.

lemma *power-mod* [*mod-simps*]:

$((a \bmod b) ^ n) \bmod b = (a ^ n) \bmod b$
proof (*induct n*)
case 0
then show *?case* **by** *simp*
next
case (*Suc n*)
have $(a \bmod b) ^ \text{Suc } n \bmod b = (a \bmod b) * ((a \bmod b) ^ n \bmod b) \bmod b$
by (*simp add: mod-mult-right-eq*)
with *Suc* **show** *?case*
by (*simp add: mod-mult-left-eq mod-mult-right-eq*)
qed

lemma *power-diff-power-eq*:

$\langle a ^ m \text{ div } a ^ n = (\text{if } n \leq m \text{ then } a ^ (m - n) \text{ else } 1 \text{ div } a ^ (n - m)) \rangle$
if $\langle a \neq 0 \rangle$
proof (*cases* $\langle n \leq m \rangle$)

```

case True
with that power-diff [symmetric, of a n m] show ?thesis by simp
next
case False
then obtain q where n: ⟨n = m + Suc q⟩
  by (auto simp add: not-le dest: less-imp-Suc-add)
then have ⟨a ^ m div a ^ n = (a ^ m * 1) div (a ^ m * a ^ Suc q)⟩
  by (simp add: power-add ac-simps)
moreover from that have ⟨a ^ m ≠ 0⟩
  by simp
ultimately have ⟨a ^ m div a ^ n = 1 div a ^ Suc q⟩
  by (subst (asm) div-mult-mult1) simp
with False n show ?thesis
  by simp
qed

end

```

```

class euclidean-ring-cancel = euclidean-ring + euclidean-semiring-cancel
begin

```

```

subclass idom-divide ..

```

```

lemma div-minus-minus [simp]:  $(- a) \text{ div } (- b) = a \text{ div } b$ 
  using div-mult-mult1 [of - 1 a b] by simp

```

```

lemma mod-minus-minus [simp]:  $(- a) \text{ mod } (- b) = - (a \text{ mod } b)$ 
  using mod-mult-mult1 [of - 1 a b] by simp

```

```

lemma div-minus-right:  $a \text{ div } (- b) = (- a) \text{ div } b$ 
  using div-minus-minus [of - a b] by simp

```

```

lemma mod-minus-right:  $a \text{ mod } (- b) = - ((- a) \text{ mod } b)$ 
  using mod-minus-minus [of - a b] by simp

```

```

lemma div-minus1-right [simp]:  $a \text{ div } (- 1) = - a$ 
  using div-minus-right [of a 1] by simp

```

```

lemma mod-minus1-right [simp]:  $a \text{ mod } (- 1) = 0$ 
  using mod-minus-right [of a 1] by simp

```

Negation respects modular equivalence.

```

lemma mod-minus-eq [mod-simps]:
   $(- (a \text{ mod } b)) \text{ mod } b = (- a) \text{ mod } b$ 
proof -
  have  $(- a) \text{ mod } b = (- (a \text{ div } b * b + a \text{ mod } b)) \text{ mod } b$ 
    by (simp only: div-mult-mod-eq)
  also have  $\dots = (- (a \text{ mod } b) + - (a \text{ div } b) * b) \text{ mod } b$ 

```

by (*simp add: ac-simps*)
 also have $\dots = (- (a \bmod b)) \bmod b$
 by (*rule mod-mult-self1*)
 finally show *?thesis*
 by (*rule sym*)
 qed

lemma *mod-minus-cong*:
 assumes $a \bmod b = a' \bmod b$
 shows $(- a) \bmod b = (- a') \bmod b$
proof –
 have $(- (a \bmod b)) \bmod b = (- (a' \bmod b)) \bmod b$
 unfolding *assms ..*
 then show *?thesis*
 by (*simp add: mod-minus-eq*)
 qed

Subtraction respects modular equivalence.

lemma *mod-diff-left-eq* [*mod-simps*]:
 $(a \bmod c - b) \bmod c = (a - b) \bmod c$
 using *mod-add-cong* [*of a c a mod c - b - b*]
 by *simp*

lemma *mod-diff-right-eq* [*mod-simps*]:
 $(a - b \bmod c) \bmod c = (a - b) \bmod c$
 using *mod-add-cong* [*of a c a - b - (b mod c)*] *mod-minus-cong* [*of b mod c c b*]
 by *simp*

lemma *mod-diff-eq*:
 $(a \bmod c - b \bmod c) \bmod c = (a - b) \bmod c$
 using *mod-add-cong* [*of a c a mod c - b - (b mod c)*] *mod-minus-cong* [*of b mod c c b*]
 by *simp*

lemma *mod-diff-cong*:
 assumes $a \bmod c = a' \bmod c$
 assumes $b \bmod c = b' \bmod c$
 shows $(a - b) \bmod c = (a' - b') \bmod c$
 using *assms mod-add-cong* [*of a c a' - b - b'*] *mod-minus-cong* [*of b c b'*]
 by *simp*

lemma *minus-mod-self2* [*simp*]:
 $(a - b) \bmod b = a \bmod b$
 using *mod-diff-right-eq* [*of a b b*]
 by (*simp add: mod-diff-right-eq*)

lemma *minus-mod-self1* [*simp*]:
 $(b - a) \bmod b = - a \bmod b$
 using *mod-add-self2* [*of - a b*] by *simp*

```

lemma mod-eq-dvd-iff:
   $a \bmod c = b \bmod c \iff c \text{ dvd } a - b$  (is  $?P \iff ?Q$ )
proof
  assume  $?P$ 
  then have  $(a \bmod c - b \bmod c) \bmod c = 0$ 
    by simp
  then show  $?Q$ 
    by (simp add: dvd-eq-mod-eq-0 mod-simps)
next
  assume  $?Q$ 
  then obtain  $d$  where  $a - b = c * d$  ..
  then have  $a = c * d + b$ 
    by (simp add: algebra-simps)
  then show  $?P$  by simp
qed

```

```

lemma mod-eqE:
  assumes  $a \bmod c = b \bmod c$ 
  obtains  $d$  where  $b = a + c * d$ 
proof -
  from assms have  $c \text{ dvd } a - b$ 
    by (simp add: mod-eq-dvd-iff)
  then obtain  $d$  where  $a - b = c * d$  ..
  then have  $b = a + c * -d$ 
    by (simp add: algebra-simps)
  with that show thesis .
qed

```

```

lemma invertible-coprime:
  coprime  $a$   $c$  if  $a * b \bmod c = 1$ 
  by (rule coprimeI) (use that dvd-mod-iff [of - c a * b] in auto)

end

```

56.3 Uniquely determined division

```

class unique-euclidean-semiring = euclidean-semiring +
  assumes euclidean-size-mult:  $\langle \text{euclidean-size } (a * b) = \text{euclidean-size } a * \text{euclidean-size } b \rangle$ 
  fixes division-segment ::  $\langle 'a \Rightarrow 'a \rangle$ 
  assumes is-unit-division-segment [simp]:  $\langle \text{is-unit } (\text{division-segment } a) \rangle$ 
  and division-segment-mult:
     $\langle a \neq 0 \implies b \neq 0 \implies \text{division-segment } (a * b) = \text{division-segment } a * \text{division-segment } b \rangle$ 
  and division-segment-mod:
     $\langle b \neq 0 \implies \neg b \text{ dvd } a \implies \text{division-segment } (a \bmod b) = \text{division-segment } b \rangle$ 
  assumes div-bounded:
     $\langle b \neq 0 \implies \text{division-segment } r = \text{division-segment } b$ 

```

```

     $\implies \text{euclidean-size } r < \text{euclidean-size } b$ 
     $\implies (q * b + r) \text{ div } b = q$ 
begin
lemma division-segment-not-0 [simp]:
   $\langle \text{division-segment } a \neq 0 \rangle$ 
  using is-unit-division-segment [of a] is-unitE [of  $\langle \text{division-segment } a \rangle$ ] by blast

lemma euclidean-relationI [case-names by0 divides euclidean-relation]:
   $\langle (a \text{ div } b, a \text{ mod } b) = (q, r) \rangle$ 
  if by0:  $\langle b = 0 \implies q = 0 \wedge r = a \rangle$ 
  and divides:  $\langle b \neq 0 \implies b \text{ dvd } a \implies r = 0 \wedge a = q * b \rangle$ 
  and euclidean-relation:  $\langle b \neq 0 \implies \neg b \text{ dvd } a \implies \text{division-segment } r = \text{division-segment } b$ 
     $\wedge \text{euclidean-size } r < \text{euclidean-size } b \wedge a = q * b + r \rangle$ 
proof (cases  $\langle b = 0 \rangle$ )
  case True
  with by0 show ?thesis
  by simp
next
  case False
  show ?thesis
  proof (cases  $\langle b \text{ dvd } a \rangle$ )
    case True
    with  $\langle b \neq 0 \rangle$  divides
    show ?thesis
    by simp
  next
    case False
    with  $\langle b \neq 0 \rangle$  euclidean-relation
    have  $\langle \text{division-segment } r = \text{division-segment } b \rangle$ 
       $\langle \text{euclidean-size } r < \text{euclidean-size } b \rangle \langle a = q * b + r \rangle$ 
      by simp-all
    from  $\langle b \neq 0 \rangle \langle \text{division-segment } r = \text{division-segment } b \rangle$ 
       $\langle \text{euclidean-size } r < \text{euclidean-size } b \rangle$ 
    have  $\langle (q * b + r) \text{ div } b = q \rangle$ 
      by (rule div-bounded)
    with  $\langle a = q * b + r \rangle$ 
    have  $\langle q = a \text{ div } b \rangle$ 
      by simp
    from  $\langle a = q * b + r \rangle$ 
    have  $\langle a \text{ div } b * b + a \text{ mod } b = q * b + r \rangle$ 
      by (simp add: div-mult-mod-eq)
    with  $\langle q = a \text{ div } b \rangle$ 
    have  $\langle q * b + a \text{ mod } b = q * b + r \rangle$ 
      by simp
    then have  $\langle r = a \text{ mod } b \rangle$ 
      by simp
    with  $\langle q = a \text{ div } b \rangle$ 

```

```

    show ?thesis
      by simp
  qed
qed

subclass euclidean-semiring-cancel
proof
  fix a b c
  assume ⟨b ≠ 0⟩
  have ⟨(a + c * b) div b, (a + c * b) mod b⟩ = ⟨c + a div b, a mod b⟩
  proof (induction rule: euclidean-relationI)
    case by0
    with ⟨b ≠ 0⟩
    show ?case
      by simp
  next
    case divides
    then show ?case
      by (simp add: algebra-simps dvd-add-left-iff)
  next
    case euclidean-relation
    then have ⟨¬ b dvd a⟩
      by (simp add: dvd-add-left-iff)
    have ⟨a mod b + (b * c + b * (a div b)) = b * c + ((a div b) * b + a mod b)⟩
      by (simp add: ac-simps)
    with ⟨b ≠ 0⟩ have *: ⟨a mod b + (b * c + b * (a div b)) = b * c + a⟩
      by (simp add: div-mult-mod-eq)
    from ⟨¬ b dvd a⟩ euclidean-relation show ?case
      by (simp-all add: algebra-simps division-segment-mod mod-size-less *)
  qed
  then show ⟨(a + c * b) div b = c + a div b⟩
    by simp
next
  fix a b c
  assume ⟨c ≠ 0⟩
  have ⟨(c * a) div (c * b), (c * a) mod (c * b)⟩ = ⟨a div b, c * (a mod b)⟩
  proof (induction rule: euclidean-relationI)
    case by0
    with ⟨c ≠ 0⟩ show ?case
      by simp
  next
    case divides
    then show ?case
      by (auto simp add: algebra-simps)
  next
    case euclidean-relation
    then have ⟨b ≠ 0⟩ ⟨a mod b ≠ 0⟩
      by (simp-all add: mod-eq-0-iff-dvd)
    have ⟨c * (a mod b) + b * (c * (a div b)) = c * ((a div b) * b + a mod b)⟩

```



```

    by (simp add: algebra-simps)
  with ⟨b ≠ 0⟩ have *: ⟨c * (a mod b) + b * (c * (a div b)) = c * a⟩
    by (simp add: div-mult-mod-eq)
  from ⟨b ≠ 0⟩ ⟨c ≠ 0⟩ have ⟨euclidean-size c * euclidean-size (a mod b)
    < euclidean-size c * euclidean-size b⟩
    using mod-size-less [of b a] by simp
  with euclidean-relation ⟨b ≠ 0⟩ ⟨a mod b ≠ 0⟩ show ?case
    by (simp add: algebra-simps division-segment-mult division-segment-mod eu-
clidean-size-mult *)
  qed
  then show ⟨(c * a) div (c * b) = a div b⟩
    by simp
  qed

```

lemma *div-eq-0-iff*:

```

  ⟨a div b = 0 ⟷ euclidean-size a < euclidean-size b ∨ b = 0⟩ (is - ⟷ ?P)
  if ⟨division-segment a = division-segment b⟩
  proof (cases ⟨a = 0 ∨ b = 0⟩)
    case True
      then show ?thesis by auto
    next
      case False
        then have ⟨a ≠ 0⟩ ⟨b ≠ 0⟩
          by simp-all
        have ⟨a div b = 0 ⟷ euclidean-size a < euclidean-size b⟩
          proof
            assume ⟨a div b = 0⟩
            then have ⟨a mod b = a⟩
              using div-mult-mod-eq [of a b] by simp
            with ⟨b ≠ 0⟩ mod-size-less [of b a]
            show ⟨euclidean-size a < euclidean-size b⟩
              by simp
          next
            assume ⟨euclidean-size a < euclidean-size b⟩
            have ⟨(a div b, a mod b) = (0, a)⟩
              proof (induction rule: euclidean-relationI)
                case by0
                  show ?case
                    by simp
              next
                case divides
                  with ⟨euclidean-size a < euclidean-size b⟩ show ?case
                    using dvd-imp-size-le [of b a] ⟨a ≠ 0⟩ by simp
              next
                case euclidean-relation
                  with ⟨euclidean-size a < euclidean-size b⟩ that
                  show ?case
                    by simp
            qed
          qed

```

```

    then show ⟨a div b = 0⟩
      by simp
  qed
  with ⟨b ≠ 0⟩ show ?thesis
    by simp
  qed

```

lemma *div-mult1-eq*:

$$\langle (a * b) \text{ div } c = a * (b \text{ div } c) + a * (b \text{ mod } c) \text{ div } c \rangle$$

proof –

have *: $\langle (a * b) \text{ mod } c + (a * (c * (b \text{ div } c)) + c * (a * (b \text{ mod } c) \text{ div } c)) = a * b \rangle$ (is $\langle ?A + (?B + ?C) = - \rangle$)

proof –

have $\langle ?A = a * (b \text{ mod } c) \text{ mod } c \rangle$

by (*simp add: mod-mult-right-eq*)

then have $\langle ?C + ?A = a * (b \text{ mod } c) \rangle$

by (*simp add: mult-div-mod-eq*)

then have $\langle ?B + (?C + ?A) = a * (c * (b \text{ div } c) + (b \text{ mod } c)) \rangle$

by (*simp add: algebra-simps*)

also have $\langle \dots = a * b \rangle$

by (*simp add: mult-div-mod-eq*)

finally show ?thesis

by (*simp add: algebra-simps*)

qed

have $\langle ((a * b) \text{ div } c, (a * b) \text{ mod } c) = (a * (b \text{ div } c) + a * (b \text{ mod } c) \text{ div } c, (a * b) \text{ mod } c) \rangle$

proof (*induction rule: euclidean-relationI*)

case *by0*

then show ?case **by** *simp*

next

case *divides*

with * **show** ?case

by (*simp add: algebra-simps*)

next

case *euclidean-relation*

with * **show** ?case

by (*simp add: division-segment-mod mod-size-less algebra-simps*)

qed

then show ?thesis

by *simp*

qed

lemma *div-add1-eq*:

$$\langle (a + b) \text{ div } c = a \text{ div } c + b \text{ div } c + (a \text{ mod } c + b \text{ mod } c) \text{ div } c \rangle$$

proof –

have *: $\langle (a + b) \text{ mod } c + (c * (a \text{ div } c) + (c * (b \text{ div } c) + c * ((a \text{ mod } c + b \text{ mod } c) \text{ div } c))) = a + b \rangle$

(is $\langle ?A + (?B + (?C + ?D)) = - \rangle$)

proof –

```

have ⟨ $?A + (?B + (?C + ?D)) = ?A + ?D + (?B + ?C)$ ⟩
  by (simp add: ac-simps)
also have ⟨ $?A + ?D = (a \text{ mod } c + b \text{ mod } c) \text{ mod } c + ?D$ ⟩
  by (simp add: mod-add-eq)
also have ⟨ $\dots = a \text{ mod } c + b \text{ mod } c$ ⟩
  by (simp add: mod-mult-div-eq)
finally have ⟨ $?A + (?B + (?C + ?D)) = (a \text{ mod } c + ?B) + (b \text{ mod } c + ?C)$ ⟩
  by (simp add: ac-simps)
then show ?thesis
  by (simp add: mod-mult-div-eq)
qed
have ⟨ $((a + b) \text{ div } c, (a + b) \text{ mod } c) = (a \text{ div } c + b \text{ div } c + (a \text{ mod } c + b \text{ mod } c) \text{ div } c, (a + b) \text{ mod } c)$ ⟩
proof (induction rule: euclidean-relationI)
  case by0
  then show ?case
    by simp
  next
  case divides
  with * show ?case
    by (simp add: algebra-simps)
  next
  case euclidean-relation
  with * show ?case
    by (simp add: division-segment-mod mod-size-less algebra-simps)
qed
then show ?thesis
  by simp
qed

end

class unique-euclidean-ring = euclidean-ring + unique-euclidean-semiring
begin

subclass euclidean-ring-cancel ..

end

```

56.4 Division on *nat*

```

instantiation nat :: normalization-semidom
begin

```

```

definition normalize-nat :: ⟨nat ⇒ nat⟩
  where [simp]: ⟨normalize = (id :: nat ⇒ nat)⟩

```

```

definition unit-factor-nat :: ⟨nat ⇒ nat⟩
  where ⟨unit-factor n = of-bool (n > 0)⟩ for n :: nat

```

lemma *unit-factor-simps* [*simp*]:

⟨*unit-factor* 0 = (0::nat)⟩

⟨*unit-factor* (Suc n) = 1⟩

by (*simp-all add: unit-factor-nat-def*)

definition *divide-nat* :: ⟨nat ⇒ nat ⇒ nat⟩

where ⟨*m div n* = (if n = 0 then 0 else Max {k. k * n ≤ m})⟩ **for** m n :: nat

instance

by *standard* (*auto simp add: divide-nat-def ac-simps unit-factor-nat-def intro: Max-eqI*)

end

lemma *coprime-Suc-0-left* [*simp*]:

coprime (Suc 0) n

using *coprime-1-left* [of n] **by** *simp*

lemma *coprime-Suc-0-right* [*simp*]:

coprime n (Suc 0)

using *coprime-1-right* [of n] **by** *simp*

lemma *coprime-common-divisor-nat*: *coprime a b ⇒ x dvd a ⇒ x dvd b ⇒ x = 1*

for a b :: nat

by (*drule coprime-common-divisor* [of - - x]) *simp-all*

instantiation nat :: *unique-euclidean-semiring*

begin

definition *euclidean-size-nat* :: ⟨nat ⇒ nat⟩

where [*simp*]: ⟨*euclidean-size-nat* = *id*⟩

definition *division-segment-nat* :: ⟨nat ⇒ nat⟩

where [*simp*]: ⟨*division-segment* n = 1⟩ **for** n :: nat

definition *modulo-nat* :: ⟨nat ⇒ nat ⇒ nat⟩

where ⟨*m mod n* = m - (m div n * n)⟩ **for** m n :: nat

instance proof

fix m n :: nat

have *ex*: ∃k. k * n ≤ l **for** l :: nat

by (*rule exI* [of - 0]) *simp*

have *fin*: *finite* {k. k * n ≤ l} **if** n > 0 **for** l

proof -

from *that* **have** {k. k * n ≤ l} ⊆ {k. k ≤ l}

by (*cases n*) *auto*

then show *?thesis*

```

    by (rule finite-subset) simp
  qed
  have mult-div-unfold:  $n * (m \text{ div } n) = \text{Max } \{l. l \leq m \wedge n \text{ dvd } l\}$ 
  proof (cases  $n = 0$ )
    case True
      moreover have  $\{l. l = 0 \wedge l \leq m\} = \{0::\text{nat}\}$ 
      by auto
      ultimately show ?thesis
      by simp
    next
      case False
        with ex [of m] fin have  $n * \text{Max } \{k. k * n \leq m\} = \text{Max } (\text{times } n \text{ ` } \{k. k * n \leq m\})$ 
        by (auto simp add: nat-mult-max-right intro: hom-Max-commute)
        also have  $\text{times } n \text{ ` } \{k. k * n \leq m\} = \{l. l \leq m \wedge n \text{ dvd } l\}$ 
        by (auto simp add: ac-simps elim!: dvdE)
        finally show ?thesis
        using False by (simp add: divide-nat-def ac-simps)
      qed
      have less-eq:  $m \text{ div } n * n \leq m$ 
      by (auto simp add: mult-div-unfold ac-simps intro: Max.boundedI)
      then show  $m \text{ div } n * n + m \text{ mod } n = m$ 
      by (simp add: modulo-nat-def)
      assume  $n \neq 0$ 
      show euclidean-size (m mod n) < euclidean-size n
      proof -
        have  $m < \text{Suc } (m \text{ div } n) * n$ 
        proof (rule ccontr)
          assume  $\neg m < \text{Suc } (m \text{ div } n) * n$ 
          then have  $\text{Suc } (m \text{ div } n) * n \leq m$ 
          by (simp add: not-less)
          moreover from  $\langle n \neq 0 \rangle$  have  $\text{Max } \{k. k * n \leq m\} < \text{Suc } (m \text{ div } n)$ 
          by (simp add: divide-nat-def)
          with  $\langle n \neq 0 \rangle$  ex fin have  $\bigwedge k. k * n \leq m \implies k < \text{Suc } (m \text{ div } n)$ 
          by auto
          ultimately have  $\text{Suc } (m \text{ div } n) < \text{Suc } (m \text{ div } n)$ 
          by blast
          then show False
          by simp
        qed
      qed
      with  $\langle n \neq 0 \rangle$  show ?thesis
      by (simp add: modulo-nat-def)
    qed
  show euclidean-size m ≤ euclidean-size (m * n)
  using  $\langle n \neq 0 \rangle$  by (cases n) simp-all
  fix q r :: nat
  show (q * n + r) div n = q if euclidean-size r < euclidean-size n
  proof -
    from that have r < n

```

```

    by simp
  have  $k \leq q$  if  $k * n \leq q * n + r$  for  $k$ 
  proof (rule ccontr)
    assume  $\neg k \leq q$ 
    then have  $q < k$ 
      by simp
    then obtain  $l$  where  $k = \text{Suc } (q + l)$ 
      by (auto simp add: less-iff-Suc-add)
    with  $\langle r < n \rangle$  that show False
      by (simp add: algebra-simps)
  qed
  with  $\langle n \neq 0 \rangle$  ex fn show ?thesis
    by (auto simp add: divide-nat-def Max-eq-iff)
  qed
qed simp-all

end

lemma euclidean-relation-natI [case-names by0 divides euclidean-relation]:
   $\langle (m \text{ div } n, m \text{ mod } n) = (q, r) \rangle$ 
  if by0:  $\langle n = 0 \implies q = 0 \wedge r = m \rangle$ 
  and divides:  $\langle n > 0 \implies n \text{ dvd } m \implies r = 0 \wedge m = q * n \rangle$ 
  and euclidean-relation:  $\langle n > 0 \implies \neg n \text{ dvd } m \implies r < n \wedge m = q * n + r \rangle$ 
for  $m \ n \ q \ r :: \text{nat}$ 
  by (rule euclidean-relationI) (use that in simp-all)

lemma div-nat-eqI:
   $\langle m \text{ div } n = q \rangle$  if  $\langle n * q \leq m \rangle$  and  $\langle m < n * \text{Suc } q \rangle$  for  $m \ n \ q :: \text{nat}$ 
proof -
  have  $\langle (m \text{ div } n, m \text{ mod } n) = (q, m - n * q) \rangle$ 
  proof (induction rule: euclidean-relation-natI)
    case by0
    with that show ?case
      by simp
  next
    case divides
    from  $\langle n \text{ dvd } m \rangle$  obtain  $s$  where  $\langle m = n * s \rangle ..$ 
    with  $\langle n > 0 \rangle$  that have  $\langle s < \text{Suc } q \rangle$ 
      by (simp only: mult-less-cancel1)
    with  $\langle m = n * s \rangle \langle n > 0 \rangle$  that have  $\langle q = s \rangle$ 
      by simp
    with  $\langle m = n * s \rangle$  show ?case
      by (simp add: ac-simps)
  next
    case euclidean-relation
    with that show ?case
      by (simp add: ac-simps)
  qed
then show ?thesis

```

by *simp*
qed

lemma *mod-nat-eqI*:

$\langle m \bmod n = r \rangle$ if $\langle r < n \rangle$ and $\langle r \leq m \rangle$ and $\langle n \text{ dvd } m - r \rangle$ for $m \ n \ r :: \text{nat}$

proof –

have $\langle (m \text{ div } n, m \bmod n) = ((m - r) \text{ div } n, r) \rangle$

proof (*induction rule: euclidean-relation-natI*)

case *by0*

with *that show ?case*

by *simp*

next

case *divides*

from *that dvd-minus-add [of r <m> 1 n]*

have $\langle n \text{ dvd } m + (n - r) \rangle$

by *simp*

with *divides have <n dvd n - r>*

by (*simp add: dvd-add-right-iff*)

then **have** $\langle n \leq n - r \rangle$

by (*rule dvd-imp-le*) (*use <r < n> in simp*)

with $\langle n > 0 \rangle$ **have** $\langle r = 0 \rangle$

by *simp*

with $\langle n > 0 \rangle$ **that show** *?case*

by *simp*

next

case *euclidean-relation*

with *that show ?case*

by (*simp add: ac-simps*)

qed

then show *?thesis*

by *simp*

qed

Tool support

ML <

structure Cancel-Div-Mod-Nat = Cancel-Div-Mod

(

val *div-name* = **const-name** *<divide>*;

val *mod-name* = **const-name** *<modulo>*;

val *mk-binop* = *HOLogic.mk-binop*;

val *dest-plus* = *HOLogic.dest-bin const-name <Groups.plus> HOLogic.natT*;

val *mk-sum* = *Arith-Data.mk-sum*;

fun *dest-sum tm* =

if *HOLogic.is-zero tm* *then* []

else

(*case try HOLogic.dest-Suc tm of*

SOME t => HOLogic.Suc-zero :: dest-sum t

| *NONE =>*

(*case try dest-plus tm of*

```

    SOME (t, u) => dest-sum t @ dest-sum u
  | NONE => [tm]);

  val div-mod-eqs = map mk-meta-eq @ {thms cancel-div-mod-rules};

  val prove-eq-sums = Arith-Data.prove-conv2 all-tac
    (Arith-Data.simp-all-tac @ {thms add-0-left add-0-right ac-simps})
)
>

simproc-setup cancel-div-mod-nat ((m::nat) + n) =
  ⟨K Cancel-Div-Mod-Nat.proc⟩

lemma div-mult-self-is-m [simp]:
  m * n div n = m if n > 0 for m n :: nat
  using that by simp

lemma div-mult-self1-is-m [simp]:
  n * m div n = m if n > 0 for m n :: nat
  using that by simp

lemma mod-less-divisor [simp]:
  m mod n < n if n > 0 for m n :: nat
  using mod-size-less [of n m] that by simp

lemma mod-le-divisor [simp]:
  m mod n ≤ n if n > 0 for m n :: nat
  using that by (auto simp add: le-less)

lemma div-times-less-eq-dividend [simp]:
  m div n * n ≤ m for m n :: nat
  by (simp add: minus-mod-eq-div-mult [symmetric])

lemma times-div-less-eq-dividend [simp]:
  n * (m div n) ≤ m for m n :: nat
  using div-times-less-eq-dividend [of m n]
  by (simp add: ac-simps)

lemma dividend-less-div-times:
  m < n + (m div n) * n if 0 < n for m n :: nat
proof –
  from that have m mod n < n
    by simp
  then show ?thesis
    by (simp add: minus-mod-eq-div-mult [symmetric])
qed

lemma dividend-less-times-div:
  m < n + n * (m div n) if 0 < n for m n :: nat

```


using *dividend-less-div-times* [of $n\ m$] *that*
by (*simp add: ac-simps*)

lemma *mod-Suc-le-divisor* [*simp*]:
 $m \bmod \text{Suc } n \leq n$
using *mod-less-divisor* [of $\text{Suc } n\ m$] **by** *arith*

lemma *mod-less-eq-dividend* [*simp*]:
 $m \bmod n \leq m$ **for** $m\ n :: \text{nat}$
proof (*rule add-leD2*)
from *div-mult-mod-eq* **have** $m \text{ div } n * n + m \bmod n = m$.
then show $m \text{ div } n * n + m \bmod n \leq m$ **by** *auto*
qed

lemma
div-less [*simp*]: $m \text{ div } n = 0$
and *mod-less* [*simp*]: $m \bmod n = m$
if $m < n$ **for** $m\ n :: \text{nat}$
using *that* **by** (*auto intro: div-nat-eqI mod-nat-eqI*)

lemma *split-div*:
 $\langle P\ (m \text{ div } n) \longleftrightarrow$
 $(n = 0 \longrightarrow P\ 0) \wedge$
 $(n \neq 0 \longrightarrow (\forall i\ j. j < n \wedge m = n * i + j \longrightarrow P\ i)) \rangle$ (**is** *?div*)
and *split-mod*:
 $\langle Q\ (m \bmod n) \longleftrightarrow$
 $(n = 0 \longrightarrow Q\ m) \wedge$
 $(n \neq 0 \longrightarrow (\forall i\ j. j < n \wedge m = n * i + j \longrightarrow Q\ j)) \rangle$ (**is** *?mod*)
for $m\ n :: \text{nat}$
proof –
have *: $\langle R\ (m \text{ div } n)\ (m \bmod n) \longleftrightarrow$
 $(n = 0 \longrightarrow R\ 0\ m) \wedge$
 $(n \neq 0 \longrightarrow (\forall i\ j. j < n \wedge m = n * i + j \longrightarrow R\ i\ j)) \rangle$ **for** R
by (*cases* $\langle n = 0 \rangle$) *auto*
from * [of $\langle \lambda q. -. P\ q \rangle$] **show** *?div* .
from * [of $\langle \lambda r. Q\ r \rangle$] **show** *?mod* .
qed

declare *split-div* [of - - $\langle \text{numeral } n \rangle$, *linarith-split*] **for** n
declare *split-mod* [of - - $\langle \text{numeral } n \rangle$, *linarith-split*] **for** n

lemma *split-div'*:
 $P\ (m \text{ div } n) \longleftrightarrow n = 0 \wedge P\ 0 \vee (\exists q. (n * q \leq m \wedge m < n * \text{Suc } q) \wedge P\ q)$
proof (*cases* $n = 0$)
case *True*
then show *?thesis*
by *simp*
next
case *False*

then have $n * q \leq m \wedge m < n * \text{Suc } q \longleftrightarrow m \text{ div } n = q$ **for** q
by (*auto intro: div-nat-eqI dividend-less-times-div*)
then show *?thesis*
by *auto*
qed

lemma *le-div-geq*:
 $m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$ **if** $0 < n$ **and** $n \leq m$ **for** $m \ n :: \text{nat}$
proof –
from $\langle n \leq m \rangle$ **obtain** q **where** $m = n + q$
by (*auto simp add: le-iff-add*)
with $\langle 0 < n \rangle$ **show** *?thesis*
by (*simp add: div-add-self1*)
qed

lemma *le-mod-geq*:
 $m \text{ mod } n = (m - n) \text{ mod } n$ **if** $n \leq m$ **for** $m \ n :: \text{nat}$
proof –
from $\langle n \leq m \rangle$ **obtain** q **where** $m = n + q$
by (*auto simp add: le-iff-add*)
then show *?thesis*
by *simp*
qed

lemma *div-if*:
 $m \text{ div } n = (\text{if } m < n \vee n = 0 \text{ then } 0 \text{ else } \text{Suc } ((m - n) \text{ div } n))$
by (*simp add: le-div-geq*)

lemma *mod-if*:
 $m \text{ mod } n = (\text{if } m < n \text{ then } m \text{ else } (m - n) \text{ mod } n)$ **for** $m \ n :: \text{nat}$
by (*simp add: le-mod-geq*)

lemma *div-eq-0-iff*:
 $m \text{ div } n = 0 \longleftrightarrow m < n \vee n = 0$ **for** $m \ n :: \text{nat}$
by (*simp add: div-eq-0-iff*)

lemma *div-greater-zero-iff*:
 $m \text{ div } n > 0 \longleftrightarrow n \leq m \wedge n > 0$ **for** $m \ n :: \text{nat}$
using *div-eq-0-iff* [of $m \ n$] **by** *auto*

lemma *mod-greater-zero-iff-not-dvd*:
 $m \text{ mod } n > 0 \longleftrightarrow \neg n \text{ dvd } m$ **for** $m \ n :: \text{nat}$
by (*simp add: dvd-eq-mod-eq-0*)

lemma *div-by-Suc-0* [*simp*]:
 $m \text{ div } \text{Suc } 0 = m$
using *div-by-1* [of m] **by** *simp*

lemma *mod-by-Suc-0* [*simp*]:

$m \bmod \text{Suc } 0 = 0$
using *mod-by-1* [of m] **by** *simp*

lemma *div2-Suc-Suc* [*simp*]:
 $\text{Suc } (\text{Suc } m) \text{ div } 2 = \text{Suc } (m \text{ div } 2)$
by (*simp add: numeral-2-eq-2 le-div-geq*)

lemma *Suc-n-div-2-gt-zero* [*simp*]:
 $0 < \text{Suc } n \text{ div } 2$ **if** $n > 0$ **for** $n :: \text{nat}$
using *that* **by** (*cases n*) *simp-all*

lemma *div-2-gt-zero* [*simp*]:
 $0 < n \text{ div } 2$ **if** $\text{Suc } 0 < n$ **for** $n :: \text{nat}$
using *that* *Suc-n-div-2-gt-zero* [of $n - 1$] **by** *simp*

lemma *mod2-Suc-Suc* [*simp*]:
 $\text{Suc } (\text{Suc } m) \bmod 2 = m \bmod 2$
by (*simp add: numeral-2-eq-2 le-mod-geq*)

lemma *add-self-div-2* [*simp*]:
 $(m + m) \text{ div } 2 = m$ **for** $m :: \text{nat}$
by (*simp add: mult-2 [symmetric]*)

lemma *add-self-mod-2* [*simp*]:
 $(m + m) \bmod 2 = 0$ **for** $m :: \text{nat}$
by (*simp add: mult-2 [symmetric]*)

lemma *mod2-gr-0* [*simp*]:
 $0 < m \bmod 2 \longleftrightarrow m \bmod 2 = 1$ **for** $m :: \text{nat}$
proof –
have $m \bmod 2 < 2$
by (*rule mod-less-divisor*) *simp*
then have $m \bmod 2 = 0 \vee m \bmod 2 = 1$
by *arith*
then show *?thesis*
by *auto*
qed

lemma *mod-Suc-eq* [*mod-simps*]:
 $\text{Suc } (m \bmod n) \bmod n = \text{Suc } m \bmod n$
proof –
have $(m \bmod n + 1) \bmod n = (m + 1) \bmod n$
by (*simp only: mod-simps*)
then show *?thesis*
by *simp*
qed

lemma *mod-Suc-Suc-eq* [*mod-simps*]:
 $\text{Suc } (\text{Suc } (m \bmod n)) \bmod n = \text{Suc } (\text{Suc } m) \bmod n$

proof –

have $(m \bmod n + 2) \bmod n = (m + 2) \bmod n$
by (*simp only: mod-simps*)
then show *?thesis*
by *simp*
qed

lemma

Suc-mod-mult-self1 [*simp*]: $\text{Suc } (m + k * n) \bmod n = \text{Suc } m \bmod n$
and *Suc-mod-mult-self2* [*simp*]: $\text{Suc } (m + n * k) \bmod n = \text{Suc } m \bmod n$
and *Suc-mod-mult-self3* [*simp*]: $\text{Suc } (k * n + m) \bmod n = \text{Suc } m \bmod n$
and *Suc-mod-mult-self4* [*simp*]: $\text{Suc } (n * k + m) \bmod n = \text{Suc } m \bmod n$
by (*subst mod-Suc-eq [symmetric], simp add: mod-simps*)**+**

lemma *Suc-0-mod-eq* [*simp*]:

$\text{Suc } 0 \bmod n = \text{of-bool } (n \neq \text{Suc } 0)$
by (*cases n*) *simp-all*

lemma *div-mult2-eq*:

$\langle m \text{ div } (n * q) = (m \text{ div } n) \text{ div } q \rangle$ (**is** *?Q*)
and *mod-mult2-eq*:
 $\langle m \bmod (n * q) = n * (m \text{ div } n \bmod q) + m \bmod n \rangle$ (**is** *?R*)
for $m \ n \ q :: \text{nat}$

proof –

have $\langle (m \text{ div } (n * q), m \bmod (n * q)) = ((m \text{ div } n) \text{ div } q, n * (m \text{ div } n \bmod q) + m \bmod n) \rangle$

proof (*induction rule: euclidean-relation-natI*)

case *by0*

then show *?case*

by *auto*

next

case *divides*

from $\langle n * q \text{ dvd } m \rangle$ **obtain** *t* **where** $\langle m = n * q * t \rangle$..

with $\langle n * q > 0 \rangle$ **show** *?case*

by (*simp add: algebra-simps*)

next

case *euclidean-relation*

then have $\langle n > 0 \rangle \langle q > 0 \rangle$

by *simp-all*

from $\langle n > 0 \rangle$ **have** $\langle m \bmod n < n \rangle$

by (*rule mod-less-divisor*)

from $\langle q > 0 \rangle$ **have** $\langle m \text{ div } n \bmod q < q \rangle$

by (*rule mod-less-divisor*)

then obtain *s* **where** $\langle q = \text{Suc } (m \text{ div } n \bmod q + s) \rangle$

by (*blast dest: less-imp-Suc-add*)

moreover have $\langle m \bmod n + n * (m \text{ div } n \bmod q) < n * \text{Suc } (m \text{ div } n \bmod q) + s \rangle$

using $\langle m \bmod n < n \rangle$ **by** (*simp add: add-mult-distrib2*)

ultimately have $\langle m \bmod n + n * (m \text{ div } n \bmod q) < n * q \rangle$

```

    by simp
  then show ?case
    by (simp add: algebra-simps flip: add-mult-distrib2)
  qed
  then show ?Q and ?R
    by simp-all
  qed

```

```

lemma div-le-mono:
  m div k ≤ n div k if m ≤ n for m n k :: nat
proof -
  from that obtain q where n = m + q
  by (auto simp add: le-iff-add)
  then show ?thesis
    by (simp add: div-add1-eq [of m q k])
  qed

```

Antimonotonicity of (*div*) in second argument

```

lemma div-le-mono2:
  k div n ≤ k div m if 0 < m and m ≤ n for m n k :: nat
using that proof (induct k arbitrary: m rule: less-induct)
  case (less k)
  show ?case
  proof (cases n ≤ k)
    case False
    then show ?thesis
      by simp
  next
  case True
  have (k - n) div n ≤ (k - m) div n
    using less.prem
    by (blast intro: div-le-mono diff-le-mono2)
  also have ... ≤ (k - m) div m
    using ⟨n ≤ k⟩ less.prem less.hyps [of k - m m]
    by simp
  finally show ?thesis
    using ⟨n ≤ k⟩ less.prem
    by (simp add: le-div-geq)
  qed
  qed

```

```

lemma div-le-dividend [simp]:
  m div n ≤ m for m n :: nat
using div-le-mono2 [of 1 n m] by (cases n = 0) simp-all

```

```

lemma div-less-dividend [simp]:
  m div n < m if 1 < n and 0 < m for m n :: nat
using that proof (induct m rule: less-induct)
  case (less m)

```

```

show ?case
proof (cases n < m)
  case False
  with less show ?thesis
  by (cases n = m) simp-all
next
  case True
  then show ?thesis
  using less.hyps [of m - n] less.prem
  by (simp add: le-div-geq)
qed
qed

```

```

lemma div-eq-dividend-iff:
  m div n = m  $\longleftrightarrow$  n = 1 if m > 0 for m n :: nat
proof
  assume n = 1
  then show m div n = m
  by simp
next
  assume P: m div n = m
  show n = 1
  proof (rule ccontr)
    have n  $\neq$  0
    by (rule ccontr) (use that P in auto)
    moreover assume n  $\neq$  1
    ultimately have n > 1
    by simp
    with that have m div n < m
    by simp
    with P show False
    by simp
  qed
qed

```

```

lemma less-mult-imp-div-less:
  m div n < i if m < i * n for m n i :: nat
proof -
  from that have i * n > 0
  by (cases i * n = 0) simp-all
  then have i > 0 and n > 0
  by simp-all
  have m div n * n  $\leq$  m
  by simp
  then have m div n * n < i * n
  using that by (rule le-less-trans)
  with <n > 0 show ?thesis
  by simp
qed

```

lemma *div-less-iff-less-mult*:
 $\langle m \text{ div } q < n \longleftrightarrow m < n * q \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)
if $\langle q > 0 \rangle$ **for** $m \ n \ q :: \text{nat}$
proof
 assume $?Q$ **then show** $?P$
 by (*rule less-mult-imp-div-less*)
next
 assume $?P$
 then obtain h **where** $\langle n = \text{Suc } (m \text{ div } q + h) \rangle$
 using *less-natE* **by** *blast*
 moreover have $\langle m < m + (\text{Suc } h * q - m \text{ mod } q) \rangle$
 using *that* **by** (*simp add: trans-less-add1*)
 ultimately show $?Q$
 by (*simp add: algebra-simps flip: minus-mod-eq-mult-div*)
qed

lemma *less-eq-div-iff-mult-less-eq*:
 $\langle m \leq n \text{ div } q \longleftrightarrow m * q \leq n \rangle$ **if** $\langle q > 0 \rangle$ **for** $m \ n \ q :: \text{nat}$
using *div-less-iff-less-mult* [*of q n m*] **that by** *auto*

lemma *div-Suc*:
 $\langle \text{Suc } m \text{ div } n = (\text{if } \text{Suc } m \text{ mod } n = 0 \text{ then } \text{Suc } (m \text{ div } n) \text{ else } m \text{ div } n) \rangle$
proof (*cases* $\langle n = 0 \vee n = 1 \rangle$)
 case *True*
 then show $?thesis$ **by** *auto*
next
 case *False*
 then have $\langle n > 1 \rangle$
 by *simp*
 then have $\langle \text{Suc } m \text{ div } n = m \text{ div } n + \text{Suc } (m \text{ mod } n) \text{ div } n \rangle$
 using *div-add1-eq* [*of m 1 n*] **by** *simp*
 also have $\langle \text{Suc } (m \text{ mod } n) \text{ div } n = \text{of-bool } (n \text{ dvd } \text{Suc } m) \rangle$
 proof (*cases* $\langle n \text{ dvd } \text{Suc } m \rangle$)
 case *False*
 moreover have $\langle \text{Suc } (m \text{ mod } n) \neq n \rangle$
 proof (*rule ccontr*)
 assume $\langle \neg \text{Suc } (m \text{ mod } n) \neq n \rangle$
 then have $\langle m \text{ mod } n = n - \text{Suc } 0 \rangle$
 by *simp*
 with $\langle n > 1 \rangle$ **have** $\langle (m + 1) \text{ mod } n = 0 \rangle$
 by (*subst mod-add-left-eq* [*symmetric*]) *simp*
 then have $\langle n \text{ dvd } \text{Suc } m \rangle$
 by *auto*
 with *False* **show** *False* ..
qed
 moreover have $\langle \text{Suc } (m \text{ mod } n) \leq n \rangle$
 using $\langle n > 1 \rangle$ **by** (*simp add: Suc-le-eq*)
 ultimately show $?thesis$

```

    by (simp add: div-eq-0-iff)
  next
  case True
  then obtain q where q: ⟨Suc m = n * q⟩ ..
  moreover have ⟨q > 0⟩ by (rule ccontr)
    (use q in simp)
  ultimately have ⟨m mod n = n - Suc 0⟩
    using ⟨n > 1⟩ mult-le-cancel1 [of n ⟨Suc 0⟩ q]
    by (auto intro: mod-nat-eqI)
  with True ⟨n > 1⟩ show ?thesis
    by simp
qed
finally show ?thesis
  by (simp add: mod-greater-zero-iff-not-dvd)
qed

lemma mod-Suc:
  ⟨Suc m mod n = (if Suc (m mod n) = n then 0 else Suc (m mod n))⟩
proof (cases ⟨n = 0⟩)
  case True
  then show ?thesis
    by simp
next
  case False
  moreover have ⟨Suc m mod n = Suc (m mod n) mod n⟩
    by (simp add: mod-simps)
  ultimately show ?thesis
    by (auto intro!: mod-nat-eqI intro: neq-le-trans simp add: Suc-le-eq)
qed

lemma Suc-times-mod-eq:
  Suc (m * n) mod m = 1 if Suc 0 < m
  using that by (simp add: mod-Suc)

lemma Suc-times-numeral-mod-eq [simp]:
  Suc (numeral k * n) mod numeral k = 1 if numeral k ≠ (1::nat)
  by (rule Suc-times-mod-eq) (use that in simp)

lemma Suc-div-le-mono [simp]:
  m div n ≤ Suc m div n
  by (simp add: div-le-mono)

```

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

```

lemma div-Suc-eq-div-add3 [simp]:
  m div Suc (Suc (Suc n)) = m div (3 + n)
  by (simp add: Suc3-eq-add-3)

```


lemma *mod-Suc-eq-mod-add3* [*simp*]:
 $m \bmod \text{Suc} (\text{Suc} (\text{Suc} n)) = m \bmod (3 + n)$
by (*simp add: Suc3-eq-add-3*)

lemma *Suc-div-eq-add3-div*:
 $\text{Suc} (\text{Suc} (\text{Suc} m)) \text{div} n = (3 + m) \text{div} n$
by (*simp add: Suc3-eq-add-3*)

lemma *Suc-mod-eq-add3-mod*:
 $\text{Suc} (\text{Suc} (\text{Suc} m)) \bmod n = (3 + m) \bmod n$
by (*simp add: Suc3-eq-add-3*)

lemmas *Suc-div-eq-add3-div-numeral* [*simp*] =
Suc-div-eq-add3-div [*of - numeral v*] **for** v

lemmas *Suc-mod-eq-add3-mod-numeral* [*simp*] =
Suc-mod-eq-add3-mod [*of - numeral v*] **for** v

lemma (*in field-char-0*) *of-nat-div*:
 $\text{of-nat} (m \text{div} n) = ((\text{of-nat} m - \text{of-nat} (m \bmod n)) / \text{of-nat} n)$

proof –

have $\text{of-nat} (m \text{div} n) = ((\text{of-nat} (m \text{div} n * n + m \bmod n) - \text{of-nat} (m \bmod n)) / \text{of-nat} n :: 'a)$

unfolding *of-nat-add* **by** (*cases n = 0*) *simp-all*

then show *?thesis*

by *simp*

qed

An “induction” law for modulus arithmetic.

lemma *mod-induct* [*consumes 3, case-names step*]:
 $P m$ **if** $P n$ **and** $n < p$ **and** $m < p$
and *step*: $\bigwedge n. n < p \implies P n \implies P (\text{Suc} n \bmod p)$
using $\langle m < p \rangle$ **proof** (*induct m*)
case 0
show *?case*
proof (*rule ccontr*)
assume $\neg P 0$
from $\langle n < p \rangle$ **have** $0 < p$
by *simp*
from $\langle n < p \rangle$ **obtain** m **where** $0 < m$ **and** $p = n + m$
by (*blast dest: less-imp-add-positive*)
with $\langle P n \rangle$ **have** $P (p - m)$
by *simp*
moreover **have** $\neg P (p - m)$
using $\langle 0 < m \rangle$ **proof** (*induct m*)
case 0
then show *?case*
by *simp*
next

```

    case (Suc m)
    show ?case
    proof
      assume P: P (p - Suc m)
      with ⟨¬ P 0⟩ have Suc m < p
        by (auto intro: ccontr)
      then have Suc (p - Suc m) = p - m
        by arith
      moreover from ⟨0 < p⟩ have p - Suc m < p
        by arith
      with P step have P ((Suc (p - Suc m)) mod p)
        by blast
      ultimately show False
        using ⟨¬ P 0⟩ Suc.hyps by (cases m = 0) simp-all
    qed
  qed
  ultimately show False
    by blast
  qed
next
  case (Suc m)
  then have m < p and mod: Suc m mod p = Suc m
    by simp-all
  from ⟨m < p⟩ have P m
    by (rule Suc.hyps)
  with ⟨m < p⟩ have P (Suc m mod p)
    by (rule step)
  with mod show ?case
    by simp
  qed

lemma funpow-mod-eq:
  ⟨(f ^^ (m mod n)) x = (f ^^ m) x⟩ if ⟨(f ^^ n) x = x⟩
proof -
  have ⟨(f ^^ m) x = (f ^^ (m mod n + m div n * n)) x⟩
    by simp
  also have ⟨... = (f ^^ (m mod n)) ((f ^^ n) ^^ (m div n)) x⟩
    by (simp only: funpow-add funpow-mult ac-simps) simp
  also have ⟨((f ^^ n) ^^ q) x = x⟩ for q
    by (induction q) (use ⟨(f ^^ n) x = x⟩ in simp-all)
  finally show ?thesis
    by simp
  qed

lemma mod-eq-dvd-iff-nat:
  ⟨m mod q = n mod q ⟷ q dvd m - n⟩ (is ⟨?P ⟷ ?Q⟩)
  if ⟨m ≥ n⟩ for m n q :: nat
proof
  assume ?Q

```

then obtain s **where** $\langle m - n = q * s \rangle ..$
with that have $\langle m = q * s + n \rangle$
 by *simp*
then show $?P$
 by *simp*
next
assume $?P$
have $\langle m - n = m \text{ div } q * q + m \text{ mod } q - (n \text{ div } q * q + n \text{ mod } q) \rangle$
 by *simp*
also have $\langle .. = q * (m \text{ div } q - n \text{ div } q) \rangle$
 by (*simp only: algebra-simps* $\langle ?P \rangle$)
finally show $?Q ..$
qed

lemma *mod-eq-iff-dvd-symdiff-nat*:
 $\langle m \text{ mod } q = n \text{ mod } q \longleftrightarrow q \text{ dvd nat } | \text{int } m - \text{int } n | \rangle$
 by (*auto simp add: abs-if mod-eq-dvd-iff-nat nat-diff-distrib dest: sym intro: sym*)

lemma *mod-eq-nat1E*:
fixes $m \ n \ q :: \text{nat}$
assumes $m \text{ mod } q = n \text{ mod } q$ **and** $m \geq n$
obtains s **where** $m = n + q * s$
proof –
from *assms* **have** $q \text{ dvd } m - n$
 by (*simp add: mod-eq-dvd-iff-nat*)
then obtain s **where** $m - n = q * s ..$
with $\langle m \geq n \rangle$ **have** $m = n + q * s$
 by *simp*
with that show *thesis* .
qed

lemma *mod-eq-nat2E*:
fixes $m \ n \ q :: \text{nat}$
assumes $m \text{ mod } q = n \text{ mod } q$ **and** $n \geq m$
obtains s **where** $n = m + q * s$
using *assms mod-eq-nat1E* [*of n q m*] **by** (*auto simp add: ac-simps*)

lemma *nat-mod-eq-iff*:
 $(x :: \text{nat}) \text{ mod } n = y \text{ mod } n \longleftrightarrow (\exists q1 \ q2. x + n * q1 = y + n * q2)$ (**is** $?lhs = ?rhs$)
proof
assume $H: x \text{ mod } n = y \text{ mod } n$
 { **assume** $xy: x \leq y$
from H **have** $th: y \text{ mod } n = x \text{ mod } n$ **by** *simp*
from *mod-eq-nat1E* [*OF th xy*] **obtain** q **where** $y = x + n * q .$
then have $x + n * q = y + n * 0$
 by *simp*
then have $\exists q1 \ q2. x + n * q1 = y + n * q2$
 by *blast*

```

}
moreover
{ assume  $xy: y \leq x$ 
  from mod-eq-nat1E [OF H xy] obtain  $q$  where  $x = y + n * q$  .
  then have  $x + n * 0 = y + n * q$ 
    by simp
  then have  $\exists q1\ q2. x + n * q1 = y + n * q2$ 
    by blast
}
ultimately show ?rhs using linear[of x y] by blast
next
assume ?rhs then obtain  $q1\ q2$  where  $q12: x + n * q1 = y + n * q2$  by blast
hence  $(x + n * q1) \bmod n = (y + n * q2) \bmod n$  by simp
thus ?lhs by simp
qed

```

56.5 Division on *int*

The following specification of integer division rounds towards minus infinity and is advocated by Donald Knuth. See [5] for an overview and terminology of different possibilities to specify integer division; there division rounding towards minus infinity is named “F-division”.

56.5.1 Basic instantiation

instantiation *int* :: {*normalization-semidom, idom-modulo*}

begin

definition *normalize-int* :: $\langle int \Rightarrow int \rangle$
where [*simp*]: $\langle normalize = (abs :: int \Rightarrow int) \rangle$

definition *unit-factor-int* :: $\langle int \Rightarrow int \rangle$
where [*simp*]: $\langle unit-factor = (sgn :: int \Rightarrow int) \rangle$

definition *divide-int* :: $\langle int \Rightarrow int \Rightarrow int \rangle$
where $\langle k\ div\ l = (sgn\ k * sgn\ l * int\ (nat\ |k|\ div\ nat\ |l|))$
 – *of-bool* $(l \neq 0 \wedge sgn\ k \neq sgn\ l \wedge \neg l\ dvd\ k) \rangle$

lemma *divide-int-unfold*:
 $\langle (sgn\ k * int\ m)\ div\ (sgn\ l * int\ n) = (sgn\ k * sgn\ l * int\ (m\ div\ n))$
 – *of-bool* $((k = 0 \iff m = 0) \wedge l \neq 0 \wedge n \neq 0 \wedge sgn\ k \neq sgn\ l \wedge \neg n\ dvd\ m) \rangle$
by (*simp add: divide-int-def sgn-mult nat-mult-distrib abs-mult sgn-eq-0-iff ac-simps*)

definition *modulo-int* :: $\langle int \Rightarrow int \Rightarrow int \rangle$
where $\langle k\ mod\ l = sgn\ k * int\ (nat\ |k|\ mod\ nat\ |l|) + l * of-bool\ (sgn\ k \neq sgn\ l$
 $\wedge \neg l\ dvd\ k) \rangle$

lemma *modulo-int-unfold*:

$\langle (sgn\ k * int\ m) \bmod (sgn\ l * int\ n) =$
 $sgn\ k * int\ (m \bmod (of\text{-}bool\ (l \neq 0) * n)) + (sgn\ l * int\ n) * of\text{-}bool\ ((k = 0$
 $\longleftrightarrow m = 0) \wedge sgn\ k \neq sgn\ l \wedge \neg n\ dvd\ m) \rangle$
by (*auto simp add: modulo-int-def sgn-mult abs-mult*)

instance proof

fix $k :: int$ **show** $k\ div\ 0 = 0$
by (*simp add: divide-int-def*)
next
fix $k\ l :: int$
assume $l \neq 0$
obtain $n\ m$ **and** $s\ t$ **where** $k = sgn\ s * int\ n$ **and** $l = sgn\ t * int\ m$
by (*blast intro: int-sgnE elim: that*)
then have $k * l = sgn\ (s * t) * int\ (n * m)$
by (*simp add: ac-simps sgn-mult*)
with $k\ l \langle l \neq 0 \rangle$ **show** $k * l\ div\ l = k$
by (*simp only: divide-int-unfold*)
(auto simp add: algebra-simps sgn-mult sgn-1-pos sgn-0-0)
next
fix $k\ l :: int$
obtain $n\ m$ **and** $s\ t$ **where** $k = sgn\ s * int\ n$ **and** $l = sgn\ t * int\ m$
by (*blast intro: int-sgnE elim: that*)
then show $k\ div\ l * l + k \bmod\ l = k$
by (*simp add: divide-int-unfold modulo-int-unfold algebra-simps modulo-nat-def*
of-nat-diff)
qed (*auto simp add: sgn-mult mult-sgn-abs abs-eq-iff'*)
end

56.5.2 Algebraic foundations

lemma *coprime-int-iff* [*simp*]:

$coprime\ (int\ m)\ (int\ n) \longleftrightarrow coprime\ m\ n$ (**is** $?P \longleftrightarrow ?Q$)

proof

assume $?P$

show $?Q$

proof (*rule coprimeI*)

fix q

assume $q\ dvd\ m\ q\ dvd\ n$

then have $int\ q\ dvd\ int\ m\ int\ q\ dvd\ int\ n$

by *simp-all*

with $\langle ?P \rangle$ **have** *is-unit* $(int\ q)$

by (*rule coprime-common-divisor*)

then show *is-unit* q

by *simp*

qed

next

assume $?Q$

show $?P$

```

proof (rule coprimeI)
  fix k
  assume k dvd int m k dvd int n
  then have nat |k| dvd m nat |k| dvd n
    by simp-all
  with ⟨?Q⟩ have is-unit (nat |k|)
    by (rule coprime-common-divisor)
  then show is-unit k
    by simp
qed
qed

```

```

lemma coprime-abs-left-iff [simp]:
  coprime |k| l  $\longleftrightarrow$  coprime k l for k l :: int
  using coprime-normalize-left-iff [of k l] by simp

```

```

lemma coprime-abs-right-iff [simp]:
  coprime k |l|  $\longleftrightarrow$  coprime k l for k l :: int
  using coprime-abs-left-iff [of l k] by (simp add: ac-simps)

```

```

lemma coprime-nat-abs-left-iff [simp]:
  coprime (nat |k|) n  $\longleftrightarrow$  coprime k (int n)
proof –
  define m where m = nat |k|
  then have |k| = int m
    by simp
  moreover have coprime k (int n)  $\longleftrightarrow$  coprime |k| (int n)
    by simp
  ultimately show ?thesis
    by simp
qed

```

```

lemma coprime-nat-abs-right-iff [simp]:
  coprime n (nat |k|)  $\longleftrightarrow$  coprime (int n) k
  using coprime-nat-abs-left-iff [of k n] by (simp add: ac-simps)

```

```

lemma coprime-common-divisor-int: coprime a b  $\implies$  x dvd a  $\implies$  x dvd b  $\implies$  |x|
= 1
  for a b :: int
  by (drule coprime-common-divisor [of - - x]) simp-all

```

56.5.3 Basic conversions

```

lemma div-abs-eq-div-nat:
  |k| div |l| = int (nat |k| div nat |l|)
  by (auto simp add: divide-int-def)

```

```

lemma div-eq-div-abs:
  ⟨k div l = sgn k * sgn l * (|k| div |l|)

```

– *of-bool* ($l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k$)
for $k \ l :: \text{int}$
by (*simp add: divide-int-def [of k l] div-abs-eq-div-nat*)

lemma *div-abs-eq*:

$\langle |k| \text{ div } |l| = \text{sgn } k * \text{sgn } l * (k \text{ div } l + \text{of-bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k)) \rangle$
for $k \ l :: \text{int}$
by (*simp add: div-eq-div-abs [of k l] ac-simps*)

lemma *mod-abs-eq-div-nat*:

$|k| \text{ mod } |l| = \text{int } (\text{nat } |k| \text{ mod } \text{nat } |l|)$
by (*simp add: modulo-int-def*)

lemma *mod-eq-mod-abs*:

$\langle k \text{ mod } l = \text{sgn } k * (|k| \text{ mod } |l|) + l * \text{of-bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
by (*simp add: modulo-int-def [of k l] mod-abs-eq-div-nat*)

lemma *mod-abs-eq*:

$\langle |k| \text{ mod } |l| = \text{sgn } k * (k \text{ mod } l - l * \text{of-bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k)) \rangle$
for $k \ l :: \text{int}$
by (*auto simp: mod-eq-mod-abs [of k l]*)

lemma *div-sgn-abs-cancel*:

fixes $k \ l \ v :: \text{int}$
assumes $v \neq 0$
shows $(\text{sgn } v * |k|) \text{ div } (\text{sgn } v * |l|) = |k| \text{ div } |l|$
using *assms by (simp add: sgn-mult abs-mult sgn-0-0*
*divide-int-def [of sgn v * |k| sgn v * |l|] flip: div-abs-eq-div-nat)*

lemma *div-eq-sgn-abs*:

fixes $k \ l \ v :: \text{int}$
assumes $\text{sgn } k = \text{sgn } l$
shows $k \text{ div } l = |k| \text{ div } |l|$
using *assms by (auto simp add: div-abs-eq)*

lemma *div-dvd-sgn-abs*:

fixes $k \ l :: \text{int}$
assumes $l \text{ dvd } k$
shows $k \text{ div } l = (\text{sgn } k * \text{sgn } l) * (|k| \text{ div } |l|)$
using *assms by (auto simp add: div-abs-eq ac-simps)*

lemma *div-noneq-sgn-abs*:

fixes $k \ l :: \text{int}$
assumes $l \neq 0$
assumes $\text{sgn } k \neq \text{sgn } l$
shows $k \text{ div } l = - (|k| \text{ div } |l|) - \text{of-bool } (\neg l \text{ dvd } k)$
using *assms by (auto simp add: div-abs-eq ac-simps sgn-0-0 dest!: sgn-not-eq-imp)*

56.5.4 Euclidean division

instantiation *int* :: *unique-euclidean-ring*
begin

definition *euclidean-size-int* :: *int* \Rightarrow *nat*
where [*simp*]: *euclidean-size-int* = (*nat* \circ *abs* :: *int* \Rightarrow *nat*)

definition *division-segment-int* :: *int* \Rightarrow *int*
where *division-segment-int* *k* = (*if* *k* \geq 0 *then* 1 *else* - 1)

lemma *division-segment-eq-sgn*:
division-segment *k* = *sgn* *k* **if** *k* \neq 0 **for** *k* :: *int*
using *that* **by** (*simp* *add*: *division-segment-int-def*)

lemma *abs-division-segment* [*simp*]:
 $|$ *division-segment* *k* $|$ = 1 **for** *k* :: *int*
by (*simp* *add*: *division-segment-int-def*)

lemma *abs-mod-less*:
 $|$ *k mod l* $|$ < $|$ *l* $|$ **if** *l* \neq 0 **for** *k l* :: *int*

proof –

obtain *n m* **and** *s t* **where** *k* = *sgn* *s* * *int* *n* **and** *l* = *sgn* *t* * *int* *m*
by (*blast* *intro*: *int-sgnE* *elim*: *that*)
with *that* **show** ?*thesis*
by (*auto* *simp* *add*: *modulo-int-unfold* *abs-mult* *mod-greater-zero-iff-not-dvd*
simp *flip*: *right-diff-distrib* *dest!*: *sgn-not-eq-imp*)
(*simp* *add*: *sgn-0-0*)

qed

lemma *sgn-mod*:
sgn (*k mod l*) = *sgn* *l* **if** *l* \neq 0 \neg *l dvd k* **for** *k l* :: *int*

proof –

obtain *n m* **and** *s t* **where** *k* = *sgn* *s* * *int* *n* **and** *l* = *sgn* *t* * *int* *m*
by (*blast* *intro*: *int-sgnE* *elim*: *that*)
with *that* **show** ?*thesis*
by (*auto* *simp* *add*: *modulo-int-unfold* *sgn-mult* *mod-greater-zero-iff-not-dvd*
simp *flip*: *right-diff-distrib* *dest!*: *sgn-not-eq-imp*)

qed

instance **proof**

fix *k l* :: *int*
show *division-segment* (*k mod l*) = *division-segment* *l* **if**
l \neq 0 **and** \neg *l dvd k*
using *that* **by** (*simp* *add*: *division-segment-eq-sgn* *dvd-eq-mod-eq-0* *sgn-mod*)

next

fix *l q r* :: *int*
obtain *n m* **and** *s t*
where *l*: *l* = *sgn* *s* * *int* *n* **and** *q*: *q* = *sgn* *t* * *int* *m*
by (*blast* *intro*: *int-sgnE* *elim*: *that*)


```

assume ⟨l ≠ 0⟩
with l have s ≠ 0 and n > 0
  by (simp-all add: sgn-0-0)
assume division-segment r = division-segment l
moreover have r = sgn r * |r|
  by (simp add: sgn-mult-abs)
moreover define u where u = nat |r|
ultimately have r = sgn l * int u
  using division-segment-eq-sgn ⟨l ≠ 0⟩ by (cases r = 0) simp-all
with l ⟨n > 0⟩ have r: r = sgn s * int u
  by (simp add: sgn-mult)
assume euclidean-size r < euclidean-size l
with l r ⟨s ≠ 0⟩ have u < n
  by (simp add: abs-mult)
show (q * l + r) div l = q
proof (cases q = 0 ∨ r = 0)
  case True
  then show ?thesis
  proof
    assume q = 0
    then show ?thesis
      using l r ⟨u < n⟩ by (simp add: divide-int-unfold)
  next
    assume r = 0
    from ⟨r = 0⟩ have *: q * l + r = sgn (t * s) * int (n * m)
      using q l by (simp add: ac-simps sgn-mult)
    from ⟨s ≠ 0⟩ ⟨n > 0⟩ show ?thesis
      by (simp only: *, simp only: * q l divide-int-unfold)
      (auto simp add: sgn-mult ac-simps)
  qed
next
case False
with q r have t ≠ 0 and m > 0 and s ≠ 0 and u > 0
  by (simp-all add: sgn-0-0)
moreover from ⟨0 < m⟩ ⟨u < n⟩ have u ≤ m * n
  using mult-le-less-imp-less [of 1 m u n] by simp
ultimately have *: q * l + r = sgn (s * t)
  * int (if t < 0 then m * n - u else m * n + u)
  using l q r
  by (simp add: sgn-mult algebra-simps of-nat-diff)
have (m * n - u) div n = m - 1 if u > 0
  using ⟨0 < m⟩ ⟨u < n⟩ that
  by (auto intro: div-nat-eqI simp add: algebra-simps)
moreover have n dvd m * n - u ⟷ n dvd u
  using ⟨u ≤ m * n⟩ dvd-diffD1 [of n m * n u]
  by auto
ultimately show ?thesis
  using ⟨s ≠ 0⟩ ⟨m > 0⟩ ⟨u > 0⟩ ⟨u < n⟩ ⟨u ≤ m * n⟩
  by (simp only: *, simp only: l q divide-int-unfold)

```

(*auto simp add: sgn-mult sgn-0-0 sgn-1-pos algebra-simps dest: dvd-imp-le*)
qed
qed (*use mult-le-mono2 [of 1] in <auto simp add: division-segment-int-def not-le zero-less-mult-iff mult-less-0-iff abs-mult sgn-mult abs-mod-less sgn-mod nat-mult-distrib>*)
end

lemma *euclidean-relation-intI* [*case-names by0 divides euclidean-relation*]:

<(k div l, k mod l) = (q, r)>
if *by0'*: <l = 0 \implies q = 0 \wedge r = k>
and *divides'*: <l \neq 0 \implies l dvd k \implies r = 0 \wedge k = q * l>
and *euclidean-relation'*: <l \neq 0 \implies \neg l dvd k \implies sgn r = sgn l
 \wedge |r| < |l| \wedge k = q * l + r> **for** k l :: int

proof (*induction rule: euclidean-relationI*)

case *by0*

then show ?*case*

by (*rule by0'*)

next

case *divides*

then show ?*case*

by (*rule divides'*)

next

case *euclidean-relation*

with *euclidean-relation'* **have** <sgn r = sgn l> <|r| < |l|> <k = q * l + r>

by *simp-all*

from <sgn r = sgn l> <l \neq 0> **have** <division-segment r = division-segment l>

by (*simp add: division-segment-int-def sgn-if-split: if-splits*)

with <|r| < |l|> <k = q * l + r>

show ?*case*

by *simp*

qed

56.5.5 Trivial reduction steps

lemma *div-pos-pos-trivial* [*simp*]:

k div l = 0 **if** k \geq 0 **and** k < l **for** k l :: int

using that by (*simp add: unique-euclidean-semiring-class.div-eq-0-iff division-segment-int-def*)

lemma *mod-pos-pos-trivial* [*simp*]:

k mod l = k **if** k \geq 0 **and** k < l **for** k l :: int

using that by (*simp add: mod-eq-self-iff-div-eq-0*)

lemma *div-neg-neg-trivial* [*simp*]:

k div l = 0 **if** k \leq 0 **and** l < k **for** k l :: int

using that by (*cases k = 0*) (*simp, simp add: unique-euclidean-semiring-class.div-eq-0-iff division-segment-int-def*)

lemma *mod-neg-neg-trivial* [*simp*]:

k mod l = k **if** k \leq 0 **and** l < k **for** k l :: int

using *that* by (*simp add: mod-eq-self-iff-div-eq-0*)

lemma

div-pos-neg-trivial: $\langle k \text{ div } l = -1 \rangle$ (is ?Q)
and *mod-pos-neg-trivial*: $\langle k \text{ mod } l = k + l \rangle$ (is ?R)
if $\langle 0 < k \rangle$ **and** $\langle k + l \leq 0 \rangle$ **for** $k \ l :: \text{int}$

proof –

from *that* **have** $\langle l < 0 \rangle$

by *simp*

have $\langle (k \text{ div } l, k \text{ mod } l) = (-1, k + l) \rangle$

proof (*induction rule: euclidean-relation-intI*)

case *by0*

with $\langle l < 0 \rangle$ **show** ?case

by *simp*

next

case *divides*

from $\langle l \text{ dvd } k \rangle$ **obtain** *j* **where** $\langle k = l * j \rangle$..

with $\langle l < 0 \rangle$ $\langle 0 < k \rangle$ **have** $\langle j < 0 \rangle$

by (*simp add: zero-less-mult-iff*)

moreover from $\langle k + l \leq 0 \rangle$ $\langle k = l * j \rangle$ **have** $\langle l * (j + 1) \leq 0 \rangle$

by (*simp add: algebra-simps*)

with $\langle l < 0 \rangle$ **have** $\langle j + 1 \geq 0 \rangle$

by (*simp add: mult-le-0-iff*)

with $\langle j < 0 \rangle$ **have** $\langle j = -1 \rangle$

by *simp*

with $\langle k = l * j \rangle$ **show** ?case

by *simp*

next

case *euclidean-relation*

with $\langle k + l \leq 0 \rangle$ **have** $\langle k + l < 0 \rangle$

by (*auto simp add: less-le add-eq-0-iff*)

with $\langle 0 < k \rangle$ **show** ?case

by *simp*

qed

then show ?Q **and** ?R

by *simp-all*

qed

There is neither *div-neg-pos-trivial* nor *mod-neg-pos-trivial* because $(0::'a)$ $\text{div } l = (0::'a)$ would supersede it.

56.5.6 More uniqueness rules

lemma

fixes $a \ b \ q \ r :: \text{int}$

assumes $\langle a = b * q + r \rangle$ $\langle 0 \leq r \rangle$ $\langle r < b \rangle$

shows *int-div-pos-eq*:

$\langle a \text{ div } b = q \rangle$ (is ?Q)

and *int-mod-pos-eq*:

$\langle a \bmod b = r \rangle$ (is ?R)
proof –
have $\langle (a \text{ div } b, a \bmod b) = (q, r) \rangle$
by (induction rule: euclidean-relation-intI)
 (use *assms in* $\langle \text{auto simp add: ac-simps dvd-add-left-iff sgn-1-pos le-less dest: zdvd-imp-le} \rangle$)
then show ?Q **and** ?R
by *simp-all*
qed

lemma *int-div-neg-eq*:

$\langle a \text{ div } b = q \rangle$ **if** $\langle a = b * q + r \rangle \langle r \leq 0 \rangle \langle b < r \rangle$ **for** $a \ b \ q \ r :: \text{int}$
using *that int-div-pos-eq [of a <- b> <- q> <- r>]* **by** *simp-all*

lemma *int-mod-neg-eq*:

$\langle a \bmod b = r \rangle$ **if** $\langle a = b * q + r \rangle \langle r \leq 0 \rangle \langle b < r \rangle$ **for** $a \ b \ q \ r :: \text{int}$
using *that int-div-neg-eq [of a b q r]* **by** *simp*

56.5.7 Laws for unary minus

lemma *zmod-zminus1-not-zero*:

fixes $k \ l :: \text{int}$
shows $- k \bmod l \neq 0 \implies k \bmod l \neq 0$
by (*simp add: mod-eq-0-iff-dvd*)

lemma *zmod-zminus2-not-zero*:

fixes $k \ l :: \text{int}$
shows $k \bmod - l \neq 0 \implies k \bmod l \neq 0$
by (*simp add: mod-eq-0-iff-dvd*)

lemma *zdiv-zminus1-eq-if*:

$\langle (- a) \text{ div } b = (\text{if } a \bmod b = 0 \text{ then } - (a \text{ div } b) \text{ else } - (a \text{ div } b) - 1) \rangle$
if $\langle b \neq 0 \rangle$ **for** $a \ b :: \text{int}$
using *that sgn-not-eq-imp [of b <- a>]*
by (*cases <a = 0>*) (*auto simp add: div-eq-div-abs [of <- a> b] div-eq-div-abs [of a b] sgn-eq-0-iff*)

lemma *zdiv-zminus2-eq-if*:

$\langle a \text{ div } (- b) = (\text{if } a \bmod b = 0 \text{ then } - (a \text{ div } b) \text{ else } - (a \text{ div } b) - 1) \rangle$
if $\langle b \neq 0 \rangle$ **for** $a \ b :: \text{int}$
using *that by (auto simp add: zdiv-zminus1-eq-if div-minus-right)*

lemma *zmod-zminus1-eq-if*:

$\langle (- a) \bmod b = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } b - (a \bmod b)) \rangle$
for $a \ b :: \text{int}$
by (*cases <b = 0>*)
 (*auto simp flip: minus-div-mult-eq-mod simp add: zdiv-zminus1-eq-if algebra-simps*)

lemma *zmod-zminus2-eq-if*:

$\langle a \bmod (-b) = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } (a \bmod b) - b) \rangle$
for $a \ b :: \text{int}$
by (*auto simp add: zmod-zminus1-eq-if mod-minus-right*)

56.5.8 Borders

lemma *pos-mod-bound* [*simp*]:
 $k \bmod l < l$ **if** $l > 0$ **for** $k \ l :: \text{int}$
proof –
obtain m **and** s **where** $k = \text{sgn } s * \text{int } m$
by (*rule int-sgnE*)
moreover from that obtain n **where** $l = \text{sgn } 1 * \text{int } n$
by (*cases l simp-all*)
moreover from this that have $n > 0$
by *simp*
ultimately show *?thesis*
by (*simp only: modulo-int-unfold*)
(auto simp add: mod-greater-zero-iff-not-dvd sgn-1-pos)
qed

lemma *neg-mod-bound* [*simp*]:
 $l < k \bmod l$ **if** $l < 0$ **for** $k \ l :: \text{int}$
proof –
obtain m **and** s **where** $k = \text{sgn } s * \text{int } m$
by (*rule int-sgnE*)
moreover from that obtain q **where** $l = \text{sgn } (-1) * \text{int } (\text{Suc } q)$
by (*cases l simp-all*)
moreover define n **where** $n = \text{Suc } q$
then have $\text{Suc } q = n$
by *simp*
ultimately show *?thesis*
by (*simp only: modulo-int-unfold*)
(auto simp add: mod-greater-zero-iff-not-dvd sgn-1-neg)
qed

lemma *pos-mod-sign* [*simp*]:
 $0 \leq k \bmod l$ **if** $l > 0$ **for** $k \ l :: \text{int}$
proof –
obtain m **and** s **where** $k = \text{sgn } s * \text{int } m$
by (*rule int-sgnE*)
moreover from that obtain n **where** $l = \text{sgn } 1 * \text{int } n$
by (*cases l auto*)
moreover from this that have $n > 0$
by *simp*
ultimately show *?thesis*
by (*simp only: modulo-int-unfold*) *(auto simp add: sgn-1-pos)*
qed

lemma *neg-mod-sign* [*simp*]:

$k \bmod l \leq 0$ if $l < 0$ for $k \ l :: \text{int}$
proof –
obtain m and s where $k = \text{sgn } s * \text{int } m$
 by (rule *int-sgnE*)
moreover from that **obtain** q where $l = \text{sgn } (- 1) * \text{int } (\text{Suc } q)$
 by (cases l) *simp-all*
moreover define n where $n = \text{Suc } q$
then have $\text{Suc } q = n$
 by *simp*
moreover have $\langle \text{int } (m \bmod n) \leq \text{int } n \rangle$
 using $\langle \text{Suc } q = n \rangle$ by *simp*
then have $\langle \text{sgn } s * \text{int } (m \bmod n) \leq \text{int } n \rangle$
 by (cases s $\langle 0 :: \text{int} \rangle$ rule: *linorder-cases*) *simp-all*
ultimately show *?thesis*
 by (*simp only: modulo-int-unfold*) *auto*
qed

56.5.9 Splitting Rules for div and mod

lemma *split-zdiv*:

$\langle P (n \text{ div } k) \longleftrightarrow$
 $(k = 0 \longrightarrow P 0) \wedge$
 $(0 < k \longrightarrow (\forall i \ j. 0 \leq j \wedge j < k \wedge n = k * i + j \longrightarrow P i)) \wedge$
 $(k < 0 \longrightarrow (\forall i \ j. k < j \wedge j \leq 0 \wedge n = k * i + j \longrightarrow P i)) \rangle$ (**is** *?div*)

and *split-zmod*:

$\langle Q (n \bmod k) \longleftrightarrow$
 $(k = 0 \longrightarrow Q n) \wedge$
 $(0 < k \longrightarrow (\forall i \ j. 0 \leq j \wedge j < k \wedge n = k * i + j \longrightarrow Q j)) \wedge$
 $(k < 0 \longrightarrow (\forall i \ j. k < j \wedge j \leq 0 \wedge n = k * i + j \longrightarrow Q j)) \rangle$ (**is** *?mod*)

for $n \ k :: \text{int}$

proof –

have *: $\langle R (n \text{ div } k) (n \bmod k) \longleftrightarrow$
 $(k = 0 \longrightarrow R 0 n) \wedge$
 $(0 < k \longrightarrow (\forall i \ j. 0 \leq j \wedge j < k \wedge n = k * i + j \longrightarrow R i j)) \wedge$
 $(k < 0 \longrightarrow (\forall i \ j. k < j \wedge j \leq 0 \wedge n = k * i + j \longrightarrow R i j)) \rangle$ **for** R
 by (cases $\langle k = 0 \rangle$)
 (*auto simp add: linorder-class.neq-iff*)

from * [of $\langle \lambda q \ -. \ P \ q \rangle$] **show** *?div* .

from * [of $\langle \lambda \ -. \ r. \ Q \ r \rangle$] **show** *?mod* .

qed

Enable (lin)arith to deal with (*div*) and (*mod*) when these are applied to some constant that is of the form *numeral k*:

declare *split-zdiv* [of - - $\langle \text{numeral } n \rangle$, *linarith-split*] **for** n

declare *split-zdiv* [of - - $\langle - \text{ numeral } n \rangle$, *linarith-split*] **for** n

declare *split-zmod* [of - - $\langle \text{numeral } n \rangle$, *linarith-split*] **for** n

declare *split-zmod* [of - - $\langle - \text{ numeral } n \rangle$, *linarith-split*] **for** n

lemma *zdiv-eq-0-iff*:

$i \text{ div } k = 0 \iff k = 0 \vee 0 \leq i \wedge i < k \vee i \leq 0 \wedge k < i$ (is ?L = ?R)
for $i \ k :: \text{int}$
proof
 assume ?L
 moreover have ?L \longrightarrow ?R
 by (rule split-zdiv [THEN iffD2]) simp
 ultimately show ?R
 by blast
next
 assume ?R then show ?L
 by auto
qed

lemma *zmod-trivial-iff*:
 fixes $i \ k :: \text{int}$
 shows $i \text{ mod } k = i \iff k = 0 \vee 0 \leq i \wedge i < k \vee i \leq 0 \wedge k < i$
proof –
 have $i \text{ mod } k = i \iff i \text{ div } k = 0$
 using *div-mult-mod-eq* [of $i \ k$] by safe auto
 with *zdiv-eq-0-iff*
 show ?thesis
 by simp
qed

56.5.10 Algebraic rewrites

lemma *zdiv-zmult2-eq*: $\langle a \text{ div } (b * c) = (a \text{ div } b) \text{ div } c \rangle$ (is ?Q)
and *zmod-zmult2-eq*: $\langle a \text{ mod } (b * c) = b * (a \text{ div } b \text{ mod } c) + a \text{ mod } b \rangle$ (is ?P)
if $\langle c \geq 0 \rangle$ **for** $a \ b \ c :: \text{int}$
proof –
 have *: $\langle (a \text{ div } (b * c), a \text{ mod } (b * c)) = ((a \text{ div } b) \text{ div } c, b * (a \text{ div } b \text{ mod } c) + a \text{ mod } b) \rangle$
if $\langle b > 0 \rangle$ **for** $a \ b$
proof (*induction rule: euclidean-relationI*)
 case *by0*
 then show ?case by auto
next
 case *divides*
 then obtain d where $\langle a = b * c * d \rangle$
 by blast
 with *divides that show ?case*
 by (*simp add: ac-simps*)
next
 case *euclidean-relation*
 with $\langle b > 0 \rangle \langle c \geq 0 \rangle$ have $\langle 0 < c \rangle \langle b > 0 \rangle$
 by *simp-all*
 then have $\langle a \text{ mod } b < b \rangle$
 by *simp*
 moreover have $\langle 1 \leq c - a \text{ div } b \text{ mod } c \rangle$

```

    using ⟨c > 0⟩ by (simp add: int-one-le-iff-zero-less)
    ultimately have ⟨a mod b * 1 < b * (c - a div b mod c)⟩
      by (rule mult-less-le-imp-less) (use ⟨b > 0⟩ in simp-all)
    with ⟨0 < b⟩ ⟨0 < c⟩ show ?case
      by (simp add: division-segment-int-def algebra-simps flip: minus-mod-eq-mult-div)
  qed
  show ?Q
  proof (cases ⟨b ≥ 0⟩)
    case True
      with * [of b a] show ?thesis
        by (cases ⟨b = 0⟩) simp-all
    next
      case False
        with * [of ⟨- b⟩ ⟨- a⟩] show ?thesis
          by simp
  qed
  show ?P
  proof (cases ⟨b ≥ 0⟩)
    case True
      with * [of b a] show ?thesis
        by (cases ⟨b = 0⟩) simp-all
    next
      case False
        with * [of ⟨- b⟩ ⟨- a⟩] show ?thesis
          by simp
  qed
  qed

```

lemma *zdiv-zmult2-eq'*:

⟨k div (l * j) = ((sgn j * k) div l) div |j|⟩ for k l j :: int

proof –

have ⟨k div (l * j) = (sgn j * k) div (sgn j * (l * j))⟩

by (simp add: sgn-0-0)

also have ⟨sgn j * (l * j) = l * |j|⟩

by (simp add: mult.left-commute [of - l] abs-*sgn*) (simp add: ac-simps)

also have ⟨(sgn j * k) div (l * |j|) = ((sgn j * k) div l) div |j|⟩

by (simp add: zdiv-zmult2-eq)

finally show ?thesis .

qed

lemma *half-nonnegative-int-iff* [*simp*]:

⟨k div 2 ≥ 0 ⟷ k ≥ 0⟩ for k :: int

by *auto*

lemma *half-negative-int-iff* [*simp*]:

⟨k div 2 < 0 ⟷ k < 0⟩ for k :: int

by *auto*

56.5.11 Distributive laws for conversions.**lemma** *zdiv-int*:

$\langle \text{int } (m \text{ div } n) = \text{int } m \text{ div int } n \rangle$
by (*cases* $\langle m = 0 \rangle$) (*auto simp add: divide-int-def*)

lemma *zmod-int*:

$\langle \text{int } (m \text{ mod } n) = \text{int } m \text{ mod int } n \rangle$
by (*cases* $\langle m = 0 \rangle$) (*auto simp add: modulo-int-def*)

lemma *nat-div-distrib*:

$\langle \text{nat } (x \text{ div } y) = \text{nat } x \text{ div nat } y \rangle$ **if** $\langle 0 \leq x \rangle$
using that by (*simp add: divide-int-def sgn-if*)

lemma *nat-div-distrib'*:

$\langle \text{nat } (x \text{ div } y) = \text{nat } x \text{ div nat } y \rangle$ **if** $\langle 0 \leq y \rangle$
using that by (*simp add: divide-int-def sgn-if*)

lemma *nat-mod-distrib*: — Fails if $y < 0$: the LHS collapses to $(\text{nat } z)$ but the RHS doesn't

$\langle \text{nat } (x \text{ mod } y) = \text{nat } x \text{ mod nat } y \rangle$ **if** $\langle 0 \leq x \rangle \langle 0 \leq y \rangle$
using that by (*simp add: modulo-int-def sgn-if*)

56.5.12 Monotonicity in the First Argument (Dividend)**lemma** *zdiv-mono1*:

$\langle a \text{ div } b \leq a' \text{ div } b \rangle$
if $\langle a \leq a' \rangle \langle 0 < b \rangle$
for $a \ b \ b' :: \text{int}$

proof —

from $\langle a \leq a' \rangle$ **have** $\langle b * (a \text{ div } b) + a \text{ mod } b \leq b * (a' \text{ div } b) + a' \text{ mod } b \rangle$

by *simp*

then have $\langle b * (a \text{ div } b) \leq (a' \text{ mod } b - a \text{ mod } b) + b * (a' \text{ div } b) \rangle$

by (*simp add: algebra-simps*)

moreover have $\langle a' \text{ mod } b < b + a \text{ mod } b \rangle$

by (*rule less-le-trans [of - b]*) (*use* $\langle 0 < b \rangle$) **in** *simp-all*

ultimately have $\langle b * (a \text{ div } b) < b * (1 + a' \text{ div } b) \rangle$

by (*simp add: distrib-left*)

with $\langle 0 < b \rangle$ **have** $\langle a \text{ div } b < 1 + a' \text{ div } b \rangle$

by (*simp add: mult-less-cancel-left*)

then show *?thesis*

by *simp*

qed**lemma** *zdiv-mono1-neg*:

$\langle a' \text{ div } b \leq a \text{ div } b \rangle$
if $\langle a \leq a' \rangle \langle b < 0 \rangle$
for $a \ a' \ b :: \text{int}$

using that *zdiv-mono1* [*of* $\langle - a' \rangle \langle - a \rangle \langle - b \rangle$] **by** *simp*

56.5.13 Monotonicity in the Second Argument (Divisor)**lemma** *zdiv-mono2*: $\langle a \text{ div } b \leq a \text{ div } b' \rangle$ **if** $\langle 0 \leq a \rangle \langle 0 < b' \rangle \langle b' \leq b \rangle$ **for** $a \ b \ b' :: \text{int}$ **proof** –**define** $q \ q' \ r \ r'$ **where** **: $\langle q = a \text{ div } b \rangle \langle q' = a \text{ div } b' \rangle \langle r = a \text{ mod } b \rangle \langle r' = a \text{ mod } b' \rangle$ **then have** *: $\langle b * q + r = b' * q' + r' \rangle \langle 0 \leq b' * q' + r' \rangle$ $\langle r' < b' \rangle \langle 0 \leq r \rangle \langle 0 < b' \rangle \langle b' \leq b \rangle$ **using** *that by simp-all***have** $\langle 0 < b' * (q' + 1) \rangle$ **using** * **by** (*simp add: distrib-left*)**with** * **have** $\langle 0 \leq q' \rangle$ **by** (*simp add: zero-less-mult-iff*)**moreover have** $\langle b * q = r' - r + b' * q' \rangle$ **using** * **by** *linarith***ultimately have** $\langle b * q < b * (q' + 1) \rangle$ **using** *mult-right-mono* * **unfolding** *distrib-left* **by** *fastforce***with** * **have** $\langle q \leq q' \rangle$ **by** (*simp add: mult-less-cancel-left-pos*)**with** ** **show** *?thesis***by** *simp***qed****lemma** *zdiv-mono2-neg*: $\langle a \text{ div } b' \leq a \text{ div } b \rangle$ **if** $\langle a < 0 \rangle \langle 0 < b' \rangle \langle b' \leq b \rangle$ **for** $a \ b \ b' :: \text{int}$ **proof** –**define** $q \ q' \ r \ r'$ **where** **: $\langle q = a \text{ div } b \rangle \langle q' = a \text{ div } b' \rangle \langle r = a \text{ mod } b \rangle \langle r' = a \text{ mod } b' \rangle$ **then have** *: $\langle b * q + r = b' * q' + r' \rangle \langle b' * q' + r' < 0 \rangle$ $\langle r < b \rangle \langle 0 \leq r' \rangle \langle 0 < b' \rangle \langle b' \leq b \rangle$ **using** *that by simp-all***have** $\langle b' * q' < 0 \rangle$ **using** * **by** *linarith***with** * **have** $\langle q' \leq 0 \rangle$ **by** (*simp add: mult-less-0-iff*)**have** $\langle b * q' \leq b' * q' \rangle$ **by** (*simp add: $\langle q' \leq 0 \rangle$ * mult-right-mono-neg*)**then have** $\langle b * q' < b * (q + 1) \rangle$ **using** * **by** (*simp add: distrib-left*)**then have** $\langle q' \leq q \rangle$ **using** * **by** (*simp add: mult-less-cancel-left*)**then show** *?thesis***by** (*simp add: ***)**qed****56.5.14 Quotients of Signs****lemma** *div-eq-minus1*: $\langle 0 < b \implies -1 \text{ div } b = -1 \rangle$ **for** $b :: \text{int}$

by (simp add: divide-int-def)

lemma *zmod-minus1*:

$\langle 0 < b \implies -1 \bmod b = b - 1 \rangle$ for $b :: int$
by (auto simp add: modulo-int-def)

lemma *minus-mod-int-eq*:

$\langle -k \bmod l = l - 1 - (k - 1) \bmod l \rangle$ if $\langle l \geq 0 \rangle$ for $k l :: int$

proof (cases $\langle l = 0 \rangle$)

case *True*

then show ?thesis

by simp

next

case *False*

with that have $\langle l > 0 \rangle$

by simp

then show ?thesis

proof (cases $\langle l \text{ dvd } k \rangle$)

case *True*

then obtain j where $\langle k = l * j \rangle$..

moreover have $\langle (l * j \bmod l - 1) \bmod l = l - 1 \rangle$

using $\langle l > 0 \rangle$ by (simp add: zmod-minus1)

then have $\langle (l * j - 1) \bmod l = l - 1 \rangle$

by (simp only: mod-simps)

ultimately show ?thesis

by simp

next

case *False*

moreover have 1: $\langle 0 < k \bmod l \rangle$

using $\langle 0 < l \rangle$ *False le-less* by fastforce

moreover have 2: $\langle k \bmod l < 1 + l \rangle$

using $\langle 0 < l \rangle$ *pos-mod-bound*[of l k] by *linarith*

from 1 2 $\langle l > 0 \rangle$ have $\langle (k \bmod l - 1) \bmod l = k \bmod l - 1 \rangle$

by (simp add: zmod-trivial-iff)

ultimately show ?thesis

by (simp only: zmod-zminus1-eq-if)

(simp add: mod-eq-0-iff-dvd algebra-simps mod-simps)

qed

qed

lemma *div-neg-pos-less0*:

$\langle a \text{ div } b < 0 \rangle$ if $\langle a < 0 \rangle \langle 0 < b \rangle$ for $a b :: int$

proof –

have $a \text{ div } b \leq -1 \text{ div } b$

using *zdiv-mono1* that by auto

also have $\dots \leq -1$

by (simp add: that(2) div-eq-minus1)

finally show ?thesis

by force

qed

lemma *div-nonneg-neg-le0*:
 $\langle a \text{ div } b \leq 0 \rangle$ **if** $\langle 0 \leq a \rangle \langle b < 0 \rangle$ **for** $a \ b :: \text{int}$
using that by (*auto dest: zdiv-mono1-neg*)

lemma *div-nonpos-pos-le0*:
 $\langle a \text{ div } b \leq 0 \rangle$ **if** $\langle a \leq 0 \rangle \langle 0 < b \rangle$ **for** $a \ b :: \text{int}$
using that by (*auto dest: zdiv-mono1*)

Now for some equivalences of the form $a \text{ div } b \geq 0 \iff \dots$ conditional upon the sign of a or b . There are many more. They should all be simp rules unless that causes too much search.

lemma *pos-imp-zdiv-nonneg-iff*:
 $\langle 0 \leq a \text{ div } b \iff 0 \leq a \rangle$
if $\langle 0 < b \rangle$ **for** $a \ b :: \text{int}$

proof

assume $\langle 0 \leq a \text{ div } b \rangle$
show $\langle 0 \leq a \rangle$
proof (*rule ccontr*)
assume $\langle \neg 0 \leq a \rangle$
then have $\langle a < 0 \rangle$
by *simp*
then have $\langle a \text{ div } b < 0 \rangle$
using that by (*rule div-neg-pos-less0*)
with $\langle 0 \leq a \text{ div } b \rangle$ **show** *False*
by *simp*

qed

next

assume $0 \leq a$
then have $0 \text{ div } b \leq a \text{ div } b$
using *zdiv-mono1* **that by** *blast*
then show $0 \leq a \text{ div } b$
by *auto*

qed

lemma *neg-imp-zdiv-nonneg-iff*:
 $\langle 0 \leq a \text{ div } b \iff a \leq 0 \rangle$ **if** $\langle b < 0 \rangle$ **for** $a \ b :: \text{int}$
using that *pos-imp-zdiv-nonneg-iff* [*of* $\langle - \ b \rangle \langle - \ a \rangle$] **by** *simp*

lemma *pos-imp-zdiv-pos-iff*:
 $\langle 0 < (i::\text{int}) \text{ div } k \iff k \leq i \rangle$ **if** $\langle 0 < k \rangle$ **for** $i \ k :: \text{int}$
using that *pos-imp-zdiv-nonneg-iff* [*of* $k \ i$] *zdiv-eq-0-iff* [*of* $i \ k$] **by** *arith*

lemma *pos-imp-zdiv-neg-iff*:
 $\langle a \text{ div } b < 0 \iff a < 0 \rangle$ **if** $\langle 0 < b \rangle$ **for** $a \ b :: \text{int}$
 — But not $(a \text{ div } b \leq 0) = (a \leq 0)$; consider $a = 1, b = 2$ when $a \text{ div } b = 0$.
using that by (*simp add: pos-imp-zdiv-nonneg-iff flip: linorder-not-le*)

lemma *neg-imp-zdiv-neg-iff*:

— But not $(a \text{ div } b \leq 0) = (0 \leq a)$; consider $a = -1, b = -2$ when $a \text{ div } b = 0$.

$\langle a \text{ div } b < 0 \iff 0 < a \rangle$ **if** $\langle b < 0 \rangle$ **for** $a \ b :: \text{int}$

using that by (*simp add: neg-imp-zdiv-nonneg-iff flip: linorder-not-le*)

lemma *nonneg1-imp-zdiv-pos-iff*:

$\langle a \text{ div } b > 0 \iff a \geq b \wedge b > 0 \rangle$ **if** $\langle 0 \leq a \rangle$ **for** $a \ b :: \text{int}$

proof —

have $0 < a \text{ div } b \implies b \leq a$

using *div-pos-pos-trivial*[of $a \ b$] **that by** *arith*

moreover have $0 < a \text{ div } b \implies b > 0$

using that *div-nonneg-neg-le0*[of $a \ b$] **by** (*cases b=0; force*)

moreover have $b \leq a \wedge 0 < b \implies 0 < a \text{ div } b$

using *int-one-le-iff-zero-less*[of $a \text{ div } b$] *zdiv-mono1*[of $b \ a \ b$] **by** *simp*

ultimately show *?thesis*

by *blast*

qed

lemma *zmod-le-nonneg-dividend*:

$\langle m \text{ mod } k \leq m \rangle$ **if** $\langle m :: \text{int} \rangle \geq 0$ **for** $m \ k :: \text{int}$

proof —

from that have $\langle m > 0 \vee m = 0 \rangle$

by *auto*

then show *?thesis* **proof**

assume $\langle m = 0 \rangle$ **then show** *?thesis*

by *simp*

next

assume $\langle m > 0 \rangle$ **then show** *?thesis*

proof (*cases k < 0 :: int rule: linorder-cases*)

case *less*

moreover define l **where** $\langle l = -k \rangle$

ultimately have $\langle l > 0 \rangle$

by *simp*

with $\langle m > 0 \rangle$ **have** $\langle \text{int } (\text{nat } m \text{ mod } \text{nat } l) \leq m \rangle$

by (*simp flip: le-nat-iff*)

then have $\langle \text{int } (\text{nat } m \text{ mod } \text{nat } l) - l \leq m \rangle$

using $\langle l > 0 \rangle$ **by** *simp*

with $\langle m > 0 \rangle \langle l > 0 \rangle$ **show** *?thesis*

by (*simp add: modulo-int-def l-def flip: le-nat-iff*)

qed (*simp-all add: modulo-int-def flip: le-nat-iff*)

qed

qed

lemma *sgn-div-eq-sgn-mult*:

$\langle \text{sgn } (k \text{ div } l) = \text{of_bool } (k \text{ div } l \neq 0) * \text{sgn } (k * l) \rangle$

for $k \ l :: \text{int}$

proof (*cases <k div l = 0>*)

case *True*

```

then show ?thesis
  by simp
next
  case False
  have  $\langle 0 \leq |k| \text{ div } |l| \rangle$ 
    by (cases  $\langle l = 0 \rangle$ ) (simp-all add: pos-imp-zdiv-nonneg-iff)
  then have  $\langle |k| \text{ div } |l| \neq 0 \iff 0 < |k| \text{ div } |l| \rangle$ 
    by (simp add: less-le)
  also have  $\langle \dots \iff |k| \geq |l| \rangle$ 
    using False nonneg1-imp-zdiv-pos-iff by auto
  finally have *:  $\langle |k| \text{ div } |l| \neq 0 \iff |l| \leq |k| \rangle$  .
  show ?thesis
    using  $\langle 0 \leq |k| \text{ div } |l| \rangle$  False
  by (auto simp add: div-eq-div-abs [of k l] div-eq-sgn-abs [of k l]
    sgn-mult sgn-1-pos sgn-1-neg sgn-eq-0-iff nonneg1-imp-zdiv-pos-iff * dest: sgn-not-eq-imp)
qed

```

56.5.15 Further properties

lemma *div-int-pos-iff*:

$$k \text{ div } l \geq 0 \iff k = 0 \vee l = 0 \vee k \geq 0 \wedge l \geq 0 \\ \vee k < 0 \wedge l < 0$$

for $k \ l :: \text{int}$

proof (cases $k = 0 \vee l = 0$)

case False

then have *: $k \neq 0 \ l \neq 0$

by auto

then have $0 \leq k \text{ div } l \implies \neg k < 0 \implies 0 \leq l$

by (meson neg-imp-zdiv-neg-iff not-le not-less-iff-gr-or-eq)

then show ?thesis

using * **by** (auto simp add: pos-imp-zdiv-nonneg-iff neg-imp-zdiv-nonneg-iff)

qed auto

lemma *mod-int-pos-iff*:

$$\langle k \text{ mod } l \geq 0 \iff l \text{ dvd } k \vee l = 0 \wedge k \geq 0 \vee l > 0 \rangle$$

for $k \ l :: \text{int}$

proof (cases $l > 0$)

case False

then show ?thesis

by (simp add: dvd-eq-mod-eq-0) (use neg-mod-sign [of l k] **in** \langle auto simp add: le-less not-less \rangle)

qed auto

lemma *abs-div*:

$$\langle |x \text{ div } y| = |x| \text{ div } |y| \rangle \text{ if } \langle y \text{ dvd } x \rangle \text{ for } x \ y :: \text{int}$$

using that by (cases $\langle y = 0 \rangle$) (auto simp add: abs-mult)

lemma *int-power-div-base*:

$$\langle k \wedge^m \text{ div } k = k \wedge^{(m - \text{Suc } 0)} \rangle \text{ if } \langle 0 < m \rangle \langle 0 < k \rangle \text{ for } k :: \text{int}$$

using that by (cases m) simp-all

lemma *int-div-less-self*:

$\langle x \text{ div } k < x \rangle$ if $\langle 0 < x \rangle \langle 1 < k \rangle$ for $x k :: \text{int}$

proof –

from that have $\langle \text{nat } (x \text{ div } k) = \text{nat } x \text{ div } \text{nat } k \rangle$

by (simp add: nat-div-distrib)

also from that have $\langle \text{nat } x \text{ div } \text{nat } k < \text{nat } x \rangle$

by simp

finally show ?thesis

by simp

qed

56.5.16 Computing *div* and *mod* by shifting

lemma *div-pos-geq*:

$\langle k \text{ div } l = (k - l) \text{ div } l + 1 \rangle$ if $\langle 0 < l \rangle \langle l \leq k \rangle$ for $k l :: \text{int}$

proof –

have $k = (k - l) + l$ by simp

then obtain j where $k: k = j + l$..

with that show ?thesis by (simp add: div-add-self2)

qed

lemma *mod-pos-geq*:

$\langle k \text{ mod } l = (k - l) \text{ mod } l \rangle$ if $\langle 0 < l \rangle \langle l \leq k \rangle$ for $k l :: \text{int}$

proof –

have $k = (k - l) + l$ by simp

then obtain j where $k: k = j + l$..

with that show ?thesis by simp

qed

lemma *pos-zdiv-mult-2*: $\langle (1 + 2 * b) \text{ div } (2 * a) = b \text{ div } a \rangle$ (is ?Q)

and *pos-zmod-mult-2*: $\langle (1 + 2 * b) \text{ mod } (2 * a) = 1 + 2 * (b \text{ mod } a) \rangle$ (is ?R)

if $\langle 0 \leq a \rangle$ for $a b :: \text{int}$

proof –

have $\langle ((1 + 2 * b) \text{ div } (2 * a), (1 + 2 * b) \text{ mod } (2 * a)) = (b \text{ div } a, 1 + 2 * (b \text{ mod } a)) \rangle$

proof (induction rule: euclidean-relation-int1)

case *by0*

then show ?case

by simp

next

case *divides*

have $\langle 2 \text{ dvd } (2 * a) \rangle$

by simp

then have $\langle 2 \text{ dvd } (1 + 2 * b) \rangle$

using $\langle 2 * a \text{ dvd } 1 + 2 * b \rangle$ by (rule dvd-trans)

then have $\langle 2 \text{ dvd } (1 + b * 2) \rangle$

by (simp add: ac-simps)

```

    then have ‹is-unit (2 :: int)›
      by simp
    then show ?case
      by simp
  next
    case euclidean-relation
    with that have ‹a > 0›
      by simp
    moreover have ‹b mod a < a›
      using ‹a > 0› by simp
    then have ‹1 + 2 * (b mod a) < 2 * a›
      by simp
    moreover have ‹2 * (b mod a) + a * (2 * (b div a)) = 2 * (b div a * a + b
mod a)›
      by (simp only: algebra-simps)
    moreover have ‹0 ≤ 2 * (b mod a)›
      using ‹a > 0› by simp
    ultimately show ?case
      by (simp add: algebra-simps)
  qed
  then show ?Q and ?R
    by simp-all
qed

```

```

lemma neg-zdiv-mult-2: ‹(1 + 2 * b) div (2 * a) = (b + 1) div a› (is ?Q)
  and neg-zmod-mult-2: ‹(1 + 2 * b) mod (2 * a) = 2 * ((b + 1) mod a) - 1›
(is ?R)
  if ‹a ≤ 0› for a b :: int
proof -
  have ‹((1 + 2 * b) div (2 * a), (1 + 2 * b) mod (2 * a)) = ((b + 1) div a, 2
* ((b + 1) mod a) - 1)›
  proof (induction rule: euclidean-relation-intI)
    case by0
    then show ?case
      by simp
  next
    case divides
    have ‹2 dvd (2 * a)›
      by simp
    then have ‹2 dvd (1 + 2 * b)›
      using ‹2 * a dvd 1 + 2 * b› by (rule dvd-trans)
    then have ‹2 dvd (1 + b * 2)›
      by (simp add: ac-simps)
    then have ‹is-unit (2 :: int)›
      by simp
    then show ?case
      by simp
  next
    case euclidean-relation

```



```

with that have ⟨a < 0⟩
  by simp
moreover have ⟨(b + 1) mod a > a⟩
  using ⟨a < 0⟩ by simp
then have ⟨2 * ((b + 1) mod a) > 1 + 2 * a⟩
  by simp
moreover have ⟨((1 + b) mod a) ≤ 0⟩
  using ⟨a < 0⟩ by simp
then have ⟨2 * ((1 + b) mod a) ≤ 0⟩
  by simp
moreover have ⟨2 * ((1 + b) mod a) + a * (2 * ((1 + b) div a)) =
  2 * ((1 + b) div a * a + (1 + b) mod a)⟩
  by (simp only: algebra-simps)
ultimately show ?case
  by (simp add: algebra-simps sgn-mult abs-mult)
qed
then show ?Q and ?R
  by simp-all
qed

```

```

lemma zdiv-numeral-Bit0 [simp]:
  ⟨numeral (Num.Bit0 v) div numeral (Num.Bit0 w) =
  numeral v div (numeral w :: int)⟩
  unfolding numeral.simps unfolding mult-2 [symmetric]
  by (rule div-mult-mult1) simp

```

```

lemma zdiv-numeral-Bit1 [simp]:
  ⟨numeral (Num.Bit1 v) div numeral (Num.Bit0 w) =
  (numeral v div (numeral w :: int))⟩
  unfolding numeral.simps
  unfolding mult-2 [symmetric] add.commute [of - 1]
  by (rule pos-zdiv-mult-2) simp

```

```

lemma zmod-numeral-Bit0 [simp]:
  ⟨numeral (Num.Bit0 v) mod numeral (Num.Bit0 w) =
  (2::int) * (numeral v mod numeral w)⟩
  unfolding numeral-Bit0 [of v] numeral-Bit0 [of w]
  unfolding mult-2 [symmetric] by (rule mod-mult-mult1)

```

```

lemma zmod-numeral-Bit1 [simp]:
  ⟨numeral (Num.Bit1 v) mod numeral (Num.Bit0 w) =
  2 * (numeral v mod numeral w) + (1::int)⟩
  unfolding numeral-Bit1 [of v] numeral-Bit0 [of w]
  unfolding mult-2 [symmetric] add.commute [of - 1]
  by (rule pos-zmod-mult-2) simp

```

56.6 Code generation

context

begin

qualified definition *divmod-nat* :: *nat* ⇒ *nat* ⇒ *nat* × *nat*
where *divmod-nat* *m n* = (*m div n*, *m mod n*)

qualified lemma *divmod-nat-if* [code]:
divmod-nat m n = (if *n* = 0 ∨ *m* < *n* then (0, *m*) else
 let (*q*, *r*) = *divmod-nat* (*m* − *n*) *n* in (Suc *q*, *r*))
by (*simp add: divmod-nat-def prod-eq-iff case-prod-beta not-less le-div-geq le-mod-geq*)

qualified lemma [code]:
m div n = *fst* (*divmod-nat m n*)
m mod n = *snd* (*divmod-nat m n*)
by (*simp-all add: divmod-nat-def*)

end

code-identifier

code-module *Euclidean-Rings* → (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*)
Arith

end

57 Parity in rings and semirings

theory *Parity*
imports *Euclidean-Rings*
begin

57.1 Ring structures with parity and *even/odd* predicates

class *semiring-parity* = *comm-semiring-1* + *semiring-modulo* +
assumes *even-iff-mod-2-eq-zero*: 2 *dvd* *a* ↔ *a mod 2* = 0
and *odd-iff-mod-2-eq-one*: ¬ 2 *dvd* *a* ↔ *a mod 2* = 1
and *odd-one* [*simp*]: ¬ 2 *dvd* 1
begin

abbreviation *even* :: 'a ⇒ bool
where *even a* ≡ 2 *dvd a*

abbreviation *odd* :: 'a ⇒ bool
where *odd a* ≡ ¬ 2 *dvd a*

end

class *ring-parity* = *ring* + *semiring-parity*
begin

subclass *comm-ring-1* ..

end

instance *nat* :: *semiring-parity*
 by *standard* (*simp-all add: dvd-eq-mod-eq-0*)

instance *int* :: *ring-parity*
 by *standard* (*auto simp add: dvd-eq-mod-eq-0*)

context *semiring-parity*
begin

lemma *parity-cases* [*case-names even odd*]:
 assumes *even a* $\implies a \bmod 2 = 0 \implies P$
 assumes *odd a* $\implies a \bmod 2 = 1 \implies P$
 shows *P*
 using *assms* by (*cases even a*)
 (*simp-all add: even-iff-mod-2-eq-zero [symmetric] odd-iff-mod-2-eq-one [symmetric]*)

lemma *odd-of-bool-self* [*simp*]:
 $\langle \text{odd } (\text{of-bool } p) \longleftrightarrow p \rangle$
 by (*cases p*) *simp-all*

lemma *not-mod-2-eq-0-eq-1* [*simp*]:
 $a \bmod 2 \neq 0 \longleftrightarrow a \bmod 2 = 1$
 by (*cases a rule: parity-cases*) *simp-all*

lemma *not-mod-2-eq-1-eq-0* [*simp*]:
 $a \bmod 2 \neq 1 \longleftrightarrow a \bmod 2 = 0$
 by (*cases a rule: parity-cases*) *simp-all*

lemma *evenE* [*elim?*]:
 assumes *even a*
 obtains *b* where $a = 2 * b$
 using *assms* by (*rule dvdE*)

lemma *oddE* [*elim?*]:
 assumes *odd a*
 obtains *b* where $a = 2 * b + 1$
proof –
 have $a = 2 * (a \text{ div } 2) + a \bmod 2$
 by (*simp add: mult-div-mod-eq*)
 with *assms* have $a = 2 * (a \text{ div } 2) + 1$
 by (*simp add: odd-iff-mod-2-eq-one*)
 then show *?thesis* ..
qed

lemma *mod-2-eq-odd*:
 $a \bmod 2 = \text{of-bool } (\text{odd } a)$

by (*auto elim: oddE simp add: even-iff-mod-2-eq-zero*)

lemma *of-bool-odd-eq-mod-2*:

of-bool (odd a) = a mod 2

by (*simp add: mod-2-eq-odd*)

lemma *even-mod-2-iff* [*simp*]:

$\langle \text{even } (a \bmod 2) \longleftrightarrow \text{even } a \rangle$

by (*simp add: mod-2-eq-odd*)

lemma *mod2-eq-if*:

$a \bmod 2 = (\text{if even } a \text{ then } 0 \text{ else } 1)$

by (*simp add: mod-2-eq-odd*)

lemma *even-zero* [*simp*]:

even 0

by (*fact dvd-0-right*)

lemma *odd-even-add*:

even (a + b) if odd a and odd b

proof –

from *that obtain c d where a = 2 * c + 1 and b = 2 * d + 1*

by (*blast elim: oddE*)

then have $a + b = 2 * c + 2 * d + (1 + 1)$

by (*simp only: ac-simps*)

also have $\dots = 2 * (c + d + 1)$

by (*simp add: algebra-simps*)

finally show *?thesis ..*

qed

lemma *even-add* [*simp*]:

$\text{even } (a + b) \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b)$

by (*auto simp add: dvd-add-right-iff dvd-add-left-iff odd-even-add*)

lemma *odd-add* [*simp*]:

$\text{odd } (a + b) \longleftrightarrow \neg (\text{odd } a \longleftrightarrow \text{odd } b)$

by *simp*

lemma *even-plus-one-iff* [*simp*]:

$\text{even } (a + 1) \longleftrightarrow \text{odd } a$

by (*auto simp add: dvd-add-right-iff intro: odd-even-add*)

lemma *even-mult-iff* [*simp*]:

$\text{even } (a * b) \longleftrightarrow \text{even } a \vee \text{even } b$ (**is** $?P \longleftrightarrow ?Q$)

proof

assume *?Q*

then show *?P*

by *auto*

next

```

assume ?P
show ?Q
proof (rule ccontr)
  assume  $\neg$  (even a  $\vee$  even b)
  then have odd a and odd b
  by auto
  then obtain r s where  $a = 2 * r + 1$  and  $b = 2 * s + 1$ 
  by (blast elim: oddE)
  then have  $a * b = (2 * r + 1) * (2 * s + 1)$ 
  by simp
  also have  $\dots = 2 * (2 * r * s + r + s) + 1$ 
  by (simp add: algebra-simps)
  finally have odd (a * b)
  by simp
  with ⟨?P⟩ show False
  by auto
qed
qed

```

```

lemma even-numeral [simp]: even (numeral (Num.Bit0 n))
proof -
  have even (2 * numeral n)
  unfolding even-mult-iff by simp
  then have even (numeral n + numeral n)
  unfolding mult-2 .
  then show ?thesis
  unfolding numeral.simps .
qed

```

```

lemma odd-numeral [simp]: odd (numeral (Num.Bit1 n))
proof
  assume even (numeral (num.Bit1 n))
  then have even (numeral n + numeral n + 1)
  unfolding numeral.simps .
  then have even (2 * numeral n + 1)
  unfolding mult-2 .
  then have 2 dvd numeral n * 2 + 1
  by (simp add: ac-simps)
  then have 2 dvd 1
  using dvd-add-times-triv-left-iff [of 2 numeral n 1] by simp
  then show False by simp
qed

```

```

lemma odd-numeral-BitM [simp]:
  ⟨odd (numeral (Num.BitM w))⟩
  by (cases w) simp-all

```

```

lemma even-power [simp]: even (a ^ n)  $\longleftrightarrow$  even a  $\wedge$  n > 0
  by (induct n) auto

```

lemma *even-prod-iff*:

$\langle \text{even } (\text{prod } f A) \longleftrightarrow (\exists a \in A. \text{even } (f a)) \rangle$ **if** $\langle \text{finite } A \rangle$
using *that* **by** $(\text{induction } A)$ *simp-all*

lemma *mask-eq-sum-exp*:

$\langle 2^n - 1 = (\sum m \in \{q. q < n\}. 2^m) \rangle$

proof –

have $*$: $\langle \{q. q < \text{Suc } m\} = \text{insert } m \ \{q. q < m\} \rangle$ **for** m
by *auto*

have $\langle 2^n = (\sum m \in \{q. q < n\}. 2^m) + 1 \rangle$

by $(\text{induction } n)$ $(\text{simp-all add: ac-simps mult-2 } *)$

then have $\langle 2^n - 1 = (\sum m \in \{q. q < n\}. 2^m) + 1 - 1 \rangle$

by *simp*

then show *?thesis*

by *simp*

qed

lemma $(\text{in } -)$ *mask-eq-sum-exp-nat*:

$\langle 2^n - \text{Suc } 0 = (\sum m \in \{q. q < n\}. 2^m) \rangle$

using *mask-eq-sum-exp* [**where** $?a = \text{nat}$] **by** *simp*

end

context *ring-parity*

begin

lemma *even-minus*:

$\text{even } (- a) \longleftrightarrow \text{even } a$

by $(\text{fact } \text{dvd-minus-iff})$

lemma *even-diff* [*simp*]:

$\text{even } (a - b) \longleftrightarrow \text{even } (a + b)$

using *even-add* [*of* $a - b$] **by** *simp*

end

57.2 Instance for *nat*

lemma *even-Suc-Suc-iff* [*simp*]:

$\text{even } (\text{Suc } (\text{Suc } n)) \longleftrightarrow \text{even } n$

using *dvd-add-triv-right-iff* [*of* $2\ n$] **by** *simp*

lemma *even-Suc* [*simp*]: $\text{even } (\text{Suc } n) \longleftrightarrow \text{odd } n$

using *even-plus-one-iff* [*of* n] **by** *simp*

lemma *even-diff-nat* [*simp*]:

$\text{even } (m - n) \longleftrightarrow m < n \vee \text{even } (m + n)$ **for** $m\ n :: \text{nat}$

proof (*cases* $n \leq m$)

case *True*
then have $m - n + n * 2 = m + n$ **by** (*simp add: mult-2-right*)
moreover have $even (m - n) \longleftrightarrow even (m - n + n * 2)$ **by** *simp*
ultimately have $even (m - n) \longleftrightarrow even (m + n)$ **by** (*simp only:*)
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *odd-pos:*
 $odd\ n \implies 0 < n$ **for** $n :: nat$
by (*auto elim: oddE*)

lemma *Suc-double-not-eq-double:*
 $Suc (2 * m) \neq 2 * n$
proof
assume $Suc (2 * m) = 2 * n$
moreover have $odd (Suc (2 * m))$ **and** $even (2 * n)$
by *simp-all*
ultimately show *False* **by** *simp*
qed

lemma *double-not-eq-Suc-double:*
 $2 * m \neq Suc (2 * n)$
using *Suc-double-not-eq-double [of n m]* **by** *simp*

lemma *odd-Suc-minus-one [simp]:* $odd\ n \implies Suc (n - Suc\ 0) = n$
by (*auto elim: oddE*)

lemma *even-Suc-div-two [simp]:*
 $even\ n \implies Suc\ n\ div\ 2 = n\ div\ 2$
by *auto*

lemma *odd-Suc-div-two [simp]:*
 $odd\ n \implies Suc\ n\ div\ 2 = Suc (n\ div\ 2)$
by (*auto elim: oddE*)

lemma *odd-two-times-div-two-nat [simp]:*
assumes $odd\ n$
shows $2 * (n\ div\ 2) = n - (1 :: nat)$
proof –
from *assms* **have** $2 * (n\ div\ 2) + 1 = n$
by (*auto elim: oddE*)
then have $Suc (2 * (n\ div\ 2)) - 1 = n - 1$
by *simp*
then show *?thesis*
by *simp*
qed

```

lemma not-mod2-eq-Suc-0-eq-0 [simp]:
   $n \bmod 2 \neq \text{Suc } 0 \iff n \bmod 2 = 0$ 
  using not-mod-2-eq-1-eq-0 [of n] by simp

lemma odd-card-imp-not-empty:
   $\langle A \neq \{\} \rangle$  if  $\langle \text{odd } (\text{card } A) \rangle$ 
  using that by auto

lemma nat-induct2 [case-names 0 1 step]:
  assumes  $P\ 0\ P\ 1$  and step:  $\bigwedge n::\text{nat}. P\ n \implies P\ (n + 2)$ 
  shows  $P\ n$ 
proof (induct n rule: less-induct)
  case (less n)
  show ?case
  proof (cases n < Suc (Suc 0))
    case True
    then show ?thesis
    using assms by (auto simp: less-Suc-eq)
  next
  case False
  then obtain  $k$  where  $k = \text{Suc } (\text{Suc } k)$ 
  by (force simp: not-less nat-le-iff-add)
  then have  $k < n$ 
  by simp
  with less assms have  $P\ (k+2)$ 
  by blast
  then show ?thesis
  by (simp add: k)
qed
qed

context semiring-parity
begin

lemma even-sum-iff:
   $\langle \text{even } (\text{sum } f\ A) \iff \text{even } (\text{card } \{a \in A. \text{odd } (f\ a)\}) \rangle$  if  $\langle \text{finite } A \rangle$ 
using that proof (induction A)
  case empty
  then show ?case
  by simp
next
  case (insert a A)
  moreover have  $\langle \{b \in \text{insert } a\ A. \text{odd } (f\ b)\} = (\text{if } \text{odd } (f\ a) \text{ then } \{a\} \text{ else } \{\}) \cup \{b \in A. \text{odd } (f\ b)\} \rangle$ 
  by auto
  ultimately show ?case
  by simp
qed

```


lemma *even-mask-iff* [*simp*]:
 $\langle \text{even } (2^n - 1) \longleftrightarrow n = 0 \rangle$
proof (*cases* $\langle n = 0 \rangle$)
 case *True*
 then show *?thesis*
 by *simp*
next
 case *False*
 then have $\langle \{a. a = 0 \wedge a < n\} = \{0\} \rangle$
 by *auto*
 then show *?thesis*
 by (*auto simp add: mask-eq-sum-exp even-sum-iff*)
qed

lemma *even-of-nat-iff* [*simp*]:
 $\text{even } (\text{of-nat } n) \longleftrightarrow \text{even } n$
by (*induction n simp-all*)

end

57.3 Parity and powers

context *ring-1*
begin

lemma *power-minus-even* [*simp*]: $\text{even } n \implies (-a)^n = a^n$
by (*auto elim: evenE*)

lemma *power-minus-odd* [*simp*]: $\text{odd } n \implies (-a)^n = -(a^n)$
by (*auto elim: oddE*)

lemma *uminus-power-if*:
 $(-a)^n = (\text{if even } n \text{ then } a^n \text{ else } -(a^n))$
by *auto*

lemma *neg-one-even-power* [*simp*]: $\text{even } n \implies (-1)^n = 1$
by *simp*

lemma *neg-one-odd-power* [*simp*]: $\text{odd } n \implies (-1)^n = -1$
by *simp*

lemma *neg-one-power-add-eq-neg-one-power-diff*: $k \leq n \implies (-1)^{n+k} = (-1)^{n-k}$
by (*cases even (n + k) auto*)

lemma *minus-one-power-iff*: $(-1)^n = (\text{if even } n \text{ then } 1 \text{ else } -1)$
by (*induct n auto*)

end

context *linordered-idom*

begin

lemma *zero-le-even-power*: $even\ n \implies 0 \leq a \wedge n$
by (*auto elim: evenE*)

lemma *zero-le-odd-power*: $odd\ n \implies 0 \leq a \wedge n \longleftrightarrow 0 \leq a$
by (*auto simp add: power-even-eq zero-le-mult-iff elim: oddE*)

lemma *zero-le-power-eq*: $0 \leq a \wedge n \longleftrightarrow even\ n \vee odd\ n \wedge 0 \leq a$
by (*auto simp add: zero-le-even-power zero-le-odd-power*)

lemma *zero-less-power-eq*: $0 < a \wedge n \longleftrightarrow n = 0 \vee even\ n \wedge a \neq 0 \vee odd\ n \wedge 0 < a$

proof –

have [*simp*]: $0 = a \wedge n \longleftrightarrow a = 0 \wedge n > 0$
unfolding *power-eq-0-iff* [*of a n, symmetric*] **by** *blast*
show *?thesis*
unfolding *less-le zero-le-power-eq* **by** *auto*

qed

lemma *power-less-zero-eq* [*simp*]: $a \wedge n < 0 \longleftrightarrow odd\ n \wedge a < 0$
unfolding *not-le [symmetric] zero-le-power-eq* **by** *auto*

lemma *power-le-zero-eq*: $a \wedge n \leq 0 \longleftrightarrow n > 0 \wedge (odd\ n \wedge a \leq 0 \vee even\ n \wedge a = 0)$

unfolding *not-less [symmetric] zero-less-power-eq* **by** *auto*

lemma *power-even-abs*: $even\ n \implies |a| \wedge n = a \wedge n$
using *power-abs [of a n]* **by** (*simp add: zero-le-even-power*)

lemma *power-mono-even*:

assumes *even n* **and** $|a| \leq |b|$
shows $a \wedge n \leq b \wedge n$

proof –

have $0 \leq |a|$ **by** *auto*
with $\langle |a| \leq |b| \rangle$ **have** $|a| \wedge n \leq |b| \wedge n$
by (*rule power-mono*)
with $\langle even\ n \rangle$ **show** *?thesis*
by (*simp add: power-even-abs*)

qed

lemma *power-mono-odd*:

assumes *odd n* **and** $a \leq b$
shows $a \wedge n \leq b \wedge n$

proof (*cases b < 0*)

case *True*

```

with ⟨a ≤ b⟩ have - b ≤ - a and 0 ≤ - b by auto
then have (- b) ^ n ≤ (- a) ^ n by (rule power-mono)
with ⟨odd n⟩ show ?thesis by simp
next
case False
then have 0 ≤ b by auto
show ?thesis
proof (cases a < 0)
  case True
  then have n ≠ 0 and a ≤ 0 using ⟨odd n⟩ [THEN odd-pos] by auto
  then have a ^ n ≤ 0 unfolding power-le-zero-eq using ⟨odd n⟩ by auto
  moreover from ⟨0 ≤ b⟩ have 0 ≤ b ^ n by auto
  ultimately show ?thesis by auto
next
case False
then have 0 ≤ a by auto
with ⟨a ≤ b⟩ show ?thesis
  using power-mono by auto
qed
qed

```

Simplify, when the exponent is a numeral

```

lemma zero-le-power-eq-numeral [simp]:
  0 ≤ a ^ numeral w ↔ even (numeral w :: nat) ∨ odd (numeral w :: nat) ∧ 0
≤ a
  by (fact zero-le-power-eq)

```

```

lemma zero-less-power-eq-numeral [simp]:
  0 < a ^ numeral w ↔
  numeral w = (0 :: nat) ∨
  even (numeral w :: nat) ∧ a ≠ 0 ∨
  odd (numeral w :: nat) ∧ 0 < a
  by (fact zero-less-power-eq)

```

```

lemma power-le-zero-eq-numeral [simp]:
  a ^ numeral w ≤ 0 ↔
  (0 :: nat) < numeral w ∧
  (odd (numeral w :: nat) ∧ a ≤ 0 ∨ even (numeral w :: nat) ∧ a = 0)
  by (fact power-le-zero-eq)

```

```

lemma power-less-zero-eq-numeral [simp]:
  a ^ numeral w < 0 ↔ odd (numeral w :: nat) ∧ a < 0
  by (fact power-less-zero-eq)

```

```

lemma power-even-abs-numeral [simp]:
  even (numeral w :: nat) ⇒ |a| ^ numeral w = a ^ numeral w
  by (fact power-even-abs)

```

end

57.4 Instance for *int***lemma** *even-diff-iff*:

$even (k - l) \longleftrightarrow even (k + l)$ **for** $k\ l :: int$
by (*fact even-diff*)

lemma *even-abs-add-iff*:

$even (|k| + l) \longleftrightarrow even (k + l)$ **for** $k\ l :: int$
by *simp*

lemma *even-add-abs-iff*:

$even (k + |l|) \longleftrightarrow even (k + l)$ **for** $k\ l :: int$
by *simp*

lemma *even-nat-iff*: $0 \leq k \implies even (nat\ k) \longleftrightarrow even\ k$

by (*simp add: even-of-nat-iff [of nat k, where ?'a = int, symmetric]*)

context

assumes *SORT-CONSTRAINT*('a::division-ring)

begin**lemma** *power-int-minus-left*:

$power-int (-a :: 'a)\ n = (if\ even\ n\ then\ power-int\ a\ n\ else\ -power-int\ a\ n)$
by (*auto simp: power-int-def minus-one-power-iff even-nat-iff*)

lemma *power-int-minus-left-even* [*simp*]: $even\ n \implies power-int (-a :: 'a)\ n = power-int\ a\ n$

by (*simp add: power-int-minus-left*)

lemma *power-int-minus-left-odd* [*simp*]: $odd\ n \implies power-int (-a :: 'a)\ n = -power-int\ a\ n$

by (*simp add: power-int-minus-left*)

lemma *power-int-minus-left-distrib*:

NO-MATCH $(-1)\ x \implies power-int (-a :: 'a)\ n = power-int (-1)\ n * power-int\ a\ n$

by (*simp add: power-int-minus-left*)

lemma *power-int-minus-one-minus*: $power-int (-1 :: 'a)\ (-n) = power-int (-1)\ n$

by (*simp add: power-int-minus-left*)

lemma *power-int-minus-one-diff-commute*: $power-int (-1 :: 'a)\ (a - b) = power-int (-1)\ (b - a)$

by (*subst power-int-minus-one-minus [symmetric]*) *auto*

lemma *power-int-minus-one-mult-self* [*simp*]:

$power-int (-1 :: 'a)\ m * power-int (-1)\ m = 1$
by (*simp add: power-int-minus-left*)

lemma *power-int-minus-one-mult-self'* [simp]:
 $power-int (-1 :: 'a) m * (power-int (-1) m * b) = b$
 by (simp add: power-int-minus-left)

end

57.5 Special case: euclidean rings containing the natural numbers

class *unique-euclidean-semiring-with-nat* = *semidom* + *semiring-char-0* + *unique-euclidean-semiring*
 +
assumes *of-nat-div*: $of-nat (m \div n) = of-nat m \div of-nat n$
and *division-segment-of-nat* [simp]: $division-segment (of-nat n) = 1$
and *division-segment-euclidean-size* [simp]: $division-segment a * of-nat (euclidean-size a) = a$
begin

lemma *division-segment-eq-iff*:
 $a = b$ **if** $division-segment a = division-segment b$
and $euclidean-size a = euclidean-size b$
using *that division-segment-euclidean-size [of a]* **by** *simp*

lemma *euclidean-size-of-nat* [simp]:
 $euclidean-size (of-nat n) = n$
proof –
have $division-segment (of-nat n) * of-nat (euclidean-size (of-nat n)) = of-nat n$
by (fact *division-segment-euclidean-size*)
then show *?thesis* **by** *simp*
qed

lemma *of-nat-euclidean-size*:
 $of-nat (euclidean-size a) = a \div division-segment a$
proof –
have $of-nat (euclidean-size a) = division-segment a * of-nat (euclidean-size a)$
 $\div division-segment a$
by (subst *nonzero-mult-div-cancel-left*) *simp-all*
also have $\dots = a \div division-segment a$
by *simp*
finally show *?thesis* .
qed

lemma *division-segment-1* [simp]:
 $division-segment 1 = 1$
using *division-segment-of-nat [of 1]* **by** *simp*

lemma *division-segment-numeral* [simp]:
 $division-segment (numeral k) = 1$
using *division-segment-of-nat [of numeral k]* **by** *simp*

```

lemma euclidean-size-1 [simp]:
  euclidean-size 1 = 1
  using euclidean-size-of-nat [of 1] by simp

lemma euclidean-size-numeral [simp]:
  euclidean-size (numeral k) = numeral k
  using euclidean-size-of-nat [of numeral k] by simp

lemma of-nat-dvd-iff:
  of-nat m dvd of-nat n  $\longleftrightarrow$  m dvd n (is ?P  $\longleftrightarrow$  ?Q)
proof (cases m = 0)
  case True
  then show ?thesis
    by simp
next
  case False
  show ?thesis
  proof
    assume ?Q
    then show ?P
      by auto
  next
    assume ?P
    with False have of-nat n = of-nat n div of-nat m * of-nat m
      by simp
    then have of-nat n = of-nat (n div m * m)
      by (simp add: of-nat-div)
    then have n = n div m * m
      by (simp only: of-nat-eq-iff)
    then have n = m * (n div m)
      by (simp add: ac-simps)
    then show ?Q ..
  qed
qed

lemma of-nat-mod:
  of-nat (m mod n) = of-nat m mod of-nat n
proof –
  have of-nat m div of-nat n * of-nat n + of-nat m mod of-nat n = of-nat m
    by (simp add: div-mult-mod-eq)
  also have of-nat m = of-nat (m div n * n + m mod n)
    by simp
  finally show ?thesis
    by (simp only: of-nat-div of-nat-mult of-nat-add) simp
qed

lemma one-div-two-eq-zero [simp]:
  1 div 2 = 0
proof –

```

```

from of-nat-div [symmetric] have of-nat 1 div of-nat 2 = of-nat 0
  by (simp only;) simp
then show ?thesis
  by simp
qed

```

```

lemma one-mod-two-eq-one [simp]:
  1 mod 2 = 1
proof –
  from of-nat-mod [symmetric] have of-nat 1 mod of-nat 2 = of-nat 1
    by (simp only;) simp
  then show ?thesis
    by simp
qed

```

```

lemma one-mod-2-pow-eq [simp]:
  1 mod (2 ^ n) = of-bool (n > 0)
proof –
  have 1 mod (2 ^ n) = of-nat (1 mod (2 ^ n))
    using of-nat-mod [of 1 2 ^ n] by simp
  also have ... = of-bool (n > 0)
    by simp
  finally show ?thesis .
qed

```

```

lemma one-div-2-pow-eq [simp]:
  1 div (2 ^ n) = of-bool (n = 0)
  using div-mult-mod-eq [of 1 2 ^ n] by auto

```

```

lemma div-mult2-eq':
  ⟨a div (of-nat m * of-nat n) = a div of-nat m div of-nat n⟩
proof (cases ⟨m = 0 ∨ n = 0⟩)
  case True
  then show ?thesis
    by auto
next
  case False
  then have ⟨m > 0⟩ ⟨n > 0⟩
    by simp-all
  show ?thesis
  proof (cases ⟨of-nat m * of-nat n dvd a⟩)
  case True
  then obtain b where ⟨a = (of-nat m * of-nat n) * b⟩ ..
  then have ⟨a = of-nat m * (of-nat n * b)⟩
    by (simp add: ac-simps)
  then show ?thesis
    by simp
next
  case False

```

define q **where** $\langle q = a \text{ div } (of\text{-nat } m * of\text{-nat } n) \rangle$
define r **where** $\langle r = a \text{ mod } (of\text{-nat } m * of\text{-nat } n) \rangle$
from $\langle m > 0 \rangle \langle n > 0 \rangle \langle \neg of\text{-nat } m * of\text{-nat } n \text{ dvd } a \rangle$ $r\text{-def}$ **have** $division\text{-segment}$
 $r = 1$
using $division\text{-segment-of-nat}$ [$of\ m * n$] **by** ($simp$ $add: division\text{-segment-mod}$)
with $division\text{-segment-euclidean-size}$ [$of\ r$]
have $of\text{-nat}$ ($euclidean\text{-size } r$) = r
by $simp$
have $a \text{ mod } (of\text{-nat } m * of\text{-nat } n) \text{ div } (of\text{-nat } m * of\text{-nat } n) = 0$
by $simp$
with $\langle m > 0 \rangle \langle n > 0 \rangle$ $r\text{-def}$ **have** $r \text{ div } (of\text{-nat } m * of\text{-nat } n) = 0$
by $simp$
with $\langle of\text{-nat } (euclidean\text{-size } r) = r \rangle$
have $of\text{-nat}$ ($euclidean\text{-size } r$) $\text{div } (of\text{-nat } m * of\text{-nat } n) = 0$
by $simp$
then **have** $of\text{-nat}$ ($euclidean\text{-size } r \text{ div } (m * n)$) = 0
by ($simp$ $add: of\text{-nat-div}$)
then **have** $of\text{-nat}$ ($euclidean\text{-size } r \text{ div } m \text{ div } n$) = 0
by ($simp$ $add: div\text{-mult2-eq}$)
with $\langle of\text{-nat } (euclidean\text{-size } r) = r \rangle$ **have** $r \text{ div } of\text{-nat } m \text{ div } of\text{-nat } n = 0$
by ($simp$ $add: of\text{-nat-div}$)
with $\langle m > 0 \rangle \langle n > 0 \rangle$ $q\text{-def}$
have $q = (r \text{ div } of\text{-nat } m + q * of\text{-nat } n * of\text{-nat } m \text{ div } of\text{-nat } m) \text{ div } of\text{-nat } n$
by $simp$
moreover **have** $\langle a = q * (of\text{-nat } m * of\text{-nat } n) + r \rangle$
by ($simp$ $add: q\text{-def } r\text{-def } div\text{-mult-mod-eq}$)
ultimately **show** $\langle a \text{ div } (of\text{-nat } m * of\text{-nat } n) = a \text{ div } of\text{-nat } m \text{ div } of\text{-nat } n \rangle$
using $q\text{-def}$ [$symmetric$] $div\text{-plus-div-distrib-dvd-right}$ [$of\ \langle of\text{-nat } m \rangle \langle q * (of\text{-nat } m * of\text{-nat } n) \rangle r$]
by ($simp$ $add: ac\text{-simps}$)
qed
qed

lemma $mod\text{-mult2-eq}'$:

$a \text{ mod } (of\text{-nat } m * of\text{-nat } n) = of\text{-nat } m * (a \text{ div } of\text{-nat } m \text{ mod } of\text{-nat } n) + a \text{ mod } of\text{-nat } m$

proof –

have $a \text{ div } (of\text{-nat } m * of\text{-nat } n) * (of\text{-nat } m * of\text{-nat } n) + a \text{ mod } (of\text{-nat } m * of\text{-nat } n) = a \text{ div } of\text{-nat } m \text{ div } of\text{-nat } n * of\text{-nat } n * of\text{-nat } m + (a \text{ div } of\text{-nat } m \text{ mod } of\text{-nat } n * of\text{-nat } m + a \text{ mod } of\text{-nat } m)$

by ($simp$ $add: combine\text{-common-factor } div\text{-mult-mod-eq}$)

moreover **have** $a \text{ div } of\text{-nat } m \text{ div } of\text{-nat } n * of\text{-nat } n * of\text{-nat } m = of\text{-nat } n * of\text{-nat } m * (a \text{ div } of\text{-nat } m \text{ div } of\text{-nat } n)$

by ($simp$ $add: ac\text{-simps}$)

ultimately **show** $?thesis$

by ($simp$ $add: div\text{-mult2-eq}' \text{ mult-commute}$)

qed

lemma $div\text{-mult2-numeral-eq}$:

$a \text{ div numeral } k \text{ div numeral } l = a \text{ div numeral } (k * l)$ (is $?A = ?B$)
proof –
have $?A = a \text{ div of-nat } (numeral\ k) \text{ div of-nat } (numeral\ l)$
by *simp*
also have $\dots = a \text{ div } (of\text{-nat } (numeral\ k) * of\text{-nat } (numeral\ l))$
by (*fact div-mult2-eq' [symmetric]*)
also have $\dots = ?B$
by *simp*
finally show $?thesis$.
qed

lemma *numeral-Bit0-div-2*:
 $numeral\ (num.Bit0\ n) \text{ div } 2 = numeral\ n$
proof –
have $numeral\ (num.Bit0\ n) = numeral\ n + numeral\ n$
by (*simp only: numeral.simps*)
also have $\dots = numeral\ n * 2$
by (*simp add: mult-2-right*)
finally have $numeral\ (num.Bit0\ n) \text{ div } 2 = numeral\ n * 2 \text{ div } 2$
by *simp*
also have $\dots = numeral\ n$
by (*rule nonzero-mult-div-cancel-right*) *simp*
finally show $?thesis$.
qed

lemma *numeral-Bit1-div-2*:
 $numeral\ (num.Bit1\ n) \text{ div } 2 = numeral\ n$
proof –
have $numeral\ (num.Bit1\ n) = numeral\ n + numeral\ n + 1$
by (*simp only: numeral.simps*)
also have $\dots = numeral\ n * 2 + 1$
by (*simp add: mult-2-right*)
finally have $numeral\ (num.Bit1\ n) \text{ div } 2 = (numeral\ n * 2 + 1) \text{ div } 2$
by *simp*
also have $\dots = numeral\ n * 2 \text{ div } 2 + 1 \text{ div } 2$
using *dvd-triv-right* **by** (*rule div-plus-div-distrib-dvd-left*)
also have $\dots = numeral\ n * 2 \text{ div } 2$
by *simp*
also have $\dots = numeral\ n$
by (*rule nonzero-mult-div-cancel-right*) *simp*
finally show $?thesis$.
qed

lemma *exp-mod-exp*:
 $\langle 2 \wedge^m \text{ mod } 2 \wedge^n = of\text{-bool } (m < n) * 2 \wedge^m \rangle$
proof –
have $\langle (2::nat) \wedge^m \text{ mod } 2 \wedge^n = of\text{-bool } (m < n) * 2 \wedge^m \rangle$ (is $\langle ?lhs = ?rhs \rangle$)
by (*auto simp add: not-less monoid-mult-class.power-add dest!: le-Suc-ex*)
then have $\langle of\text{-nat } ?lhs = of\text{-nat } ?rhs \rangle$

```

  by simp
  then show ?thesis
    by (simp add: of-nat-mod)
qed

```

lemma *mask-mod-exp*:

$\langle 2^n - 1 \text{ mod } 2^m = 2^{\min m n} - 1 \rangle$

proof –

have $\langle 2^n - 1 \text{ mod } 2^m = 2^{\min m n} - (1::\text{nat}) \rangle$ (**is** $\langle ?lhs = ?rhs \rangle$)

proof (*cases* $\langle n \leq m \rangle$)

case *True*

then show *?thesis*

by (*simp add: Suc-le-lessD*)

next

case *False*

then have $\langle m < n \rangle$

by *simp*

then obtain *q* **where** $n = \text{Suc } q + m$

by (*auto dest: less-imp-Suc-add*)

then have $\langle \min m n = m \rangle$

by *simp*

moreover have $\langle (2::\text{nat})^m \leq 2 * 2^q * 2^m \rangle$

using *mult-le-mono1* [*of* 1 $\langle 2 * 2^q \rangle$ $\langle 2^m \rangle$] **by** *simp*

with *n* **have** $\langle 2^n - 1 = (2^{\text{Suc } q} - 1) * 2^m + (2^m - (1::\text{nat})) \rangle$

by (*simp add: monoid-mult-class.power-add algebra-simps*)

ultimately show *?thesis*

by (*simp only: euclidean-semiring-cancel-class.mod-mult-self3*) *simp*

qed

then have $\langle \text{of-nat } ?lhs = \text{of-nat } ?rhs \rangle$

by *simp*

then show *?thesis*

by (*simp add: of-nat-mod of-nat-diff*)

qed

lemma *of-bool-half-eq-0* [*simp*]:

$\langle \text{of-bool } b \text{ div } 2 = 0 \rangle$

by *simp*

end

class *unique-euclidean-ring-with-nat* = *ring* + *unique-euclidean-semiring-with-nat*

instance *nat* :: *unique-euclidean-semiring-with-nat*

by *standard* (*simp-all add: dvd-eq-mod-eq-0*)

instance *int* :: *unique-euclidean-ring-with-nat*

by *standard* (*auto simp add: divide-int-def division-segment-int-def elim: contra-pos-np*)

```

context unique-euclidean-semiring-with-nat
begin

subclass semiring-parity
proof
  show  $2 \text{ dvd } a \iff a \text{ mod } 2 = 0$  for  $a$ 
    by (fact dvd-eq-mod-eq-0)
  show  $\neg 2 \text{ dvd } a \iff a \text{ mod } 2 = 1$  for  $a$ 
    proof
      assume  $a \text{ mod } 2 = 1$ 
      then show  $\neg 2 \text{ dvd } a$ 
        by auto
    next
      assume  $\neg 2 \text{ dvd } a$ 
      have eucl:  $\text{euclidean-size } (a \text{ mod } 2) = 1$ 
      proof (rule order-antisym)
        show  $\text{euclidean-size } (a \text{ mod } 2) \leq 1$ 
          using mod-size-less [of 2 a] by simp
        show  $1 \leq \text{euclidean-size } (a \text{ mod } 2)$ 
          using  $\langle \neg 2 \text{ dvd } a \rangle$  by (simp add: Suc-le-eq dvd-eq-mod-eq-0)
      qed
    from  $\langle \neg 2 \text{ dvd } a \rangle$  have  $\neg \text{of-nat } 2 \text{ dvd division-segment } a * \text{of-nat } (\text{euclidean-size } a)$ 
      by simp
    then have  $\neg \text{of-nat } 2 \text{ dvd of-nat } (\text{euclidean-size } a)$ 
      by (auto simp only: dvd-mult-unit-iff' is-unit-division-segment)
    then have  $\neg 2 \text{ dvd euclidean-size } a$ 
      using of-nat-dvd-iff [of 2] by simp
    then have  $\text{euclidean-size } a \text{ mod } 2 = 1$ 
      by (simp add: semidom-modulo-class.dvd-eq-mod-eq-0)
    then have  $\text{of-nat } (\text{euclidean-size } a \text{ mod } 2) = \text{of-nat } 1$ 
      by simp
    then have  $\text{of-nat } (\text{euclidean-size } a) \text{ mod } 2 = 1$ 
      by (simp add: of-nat-mod)
    from  $\langle \neg 2 \text{ dvd } a \rangle$  eucl
    show  $a \text{ mod } 2 = 1$ 
      by (auto intro: division-segment-eq-iff simp add: division-segment-mod)
    qed
  show  $\neg \text{is-unit } 2$ 
  proof (rule notI)
    assume is-unit 2
    then have  $\text{of-nat } 2 \text{ dvd of-nat } 1$ 
      by simp
    then have is-unit ( $2::\text{nat}$ )
      by (simp only: of-nat-dvd-iff)
    then show False
      by simp
  qed

```

qed

lemma *even-succ-div-two* [*simp*]:

even a $\implies (a + 1) \text{ div } 2 = a \text{ div } 2$

by (*cases a = 0*) (*auto elim!*: *evenE dest: mult-not-zero*)

lemma *odd-succ-div-two* [*simp*]:

odd a $\implies (a + 1) \text{ div } 2 = a \text{ div } 2 + 1$

by (*auto elim!*: *oddE simp add: add.assoc*)

lemma *even-two-times-div-two*:

even a $\implies 2 * (a \text{ div } 2) = a$

by (*fact dvd-mult-div-cancel*)

lemma *odd-two-times-div-two-succ* [*simp*]:

odd a $\implies 2 * (a \text{ div } 2) + 1 = a$

using *mult-div-mod-eq* [*of 2 a*]

by (*simp add: even-iff-mod-2-eq-zero*)

lemma *coprime-left-2-iff-odd* [*simp*]:

coprime 2 a \longleftrightarrow *odd a*

proof

assume *odd a*

show *coprime 2 a*

proof (*rule coprimeI*)

fix *b*

assume *b dvd 2 b dvd a*

then have *b dvd a mod 2*

by (*auto intro: dvd-mod*)

with $\langle \text{odd } a \rangle$ **show** *is-unit b*

by (*simp add: mod-2-eq-odd*)

qed

next

assume *coprime 2 a*

show *odd a*

proof (*rule notI*)

assume *even a*

then obtain *b* **where** *a = 2 * b ..*

with $\langle \text{coprime } 2 \ a \rangle$ **have** *coprime 2 (2 * b)*

by *simp*

moreover have $\neg \text{coprime } 2 (2 * b)$

by (*rule not-coprimeI* [*of 2*]) *simp-all*

ultimately show *False*

by *blast*

qed

qed

lemma *coprime-right-2-iff-odd* [*simp*]:

coprime a 2 \longleftrightarrow *odd a*

```

    using coprime-left-2-iff-odd [of a] by (simp add: ac-simps)

end

context unique-euclidean-ring-with-nat
begin

subclass ring-parity ..

lemma minus-1-mod-2-eq [simp]:
  - 1 mod 2 = 1
  by (simp add: mod-2-eq-odd)

lemma minus-1-div-2-eq [simp]:
  - 1 div 2 = - 1
proof -
  from div-mult-mod-eq [of - 1 2]
  have - 1 div 2 * 2 = - 1 * 2
    using add-implies-diff by fastforce
  then show ?thesis
    using mult-right-cancel [of 2 - 1 div 2 - 1] by simp
qed

end

context unique-euclidean-semiring-with-nat
begin

lemma even-mask-div-iff':
  ⟨even ((2 ^ m - 1) div 2 ^ n) ⟷ m ≤ n⟩
proof -
  have ⟨even ((2 ^ m - 1) div 2 ^ n) ⟷ even (of-nat ((2 ^ m - Suc 0) div 2 ^ n))⟩
    by (simp only: of-nat-div) (simp add: of-nat-diff)
  also have ⟨... ⟷ even ((2 ^ m - Suc 0) div 2 ^ n)⟩
    by simp
  also have ⟨... ⟷ m ≤ n⟩
  proof (cases ⟨m ≤ n⟩)
    case True
    then show ?thesis
      by (simp add: Suc-le-lessD)
    next
    case False
    then obtain r where r: ⟨m = n + Suc r⟩
      using less-imp-Suc-add by fastforce
    from r have ⟨{q. q < m} ∩ {q. 2 ^ n dvd (2::nat) ^ q} = {q. n ≤ q ∧ q < m}⟩
      by (auto simp add: dvd-power-iff-le)
    moreover from r have ⟨{q. q < m} ∩ {q. ¬ 2 ^ n dvd (2::nat) ^ q} = {q. q

```

```

< n }
  by (auto simp add: dvd-power-iff-le)
  moreover from False have ⟨{q. n ≤ q ∧ q < m ∧ q ≤ n} = {n}⟩
  by auto
  then have ⟨odd ((∑ a∈{q. n ≤ q ∧ q < m}. 2 ^ a div (2::nat) ^ n) + sum
  ((∩ 2) {q. q < n} div 2 ^ n)⟩
  by (simp-all add: euclidean-semiring-cancel-class.power-diff-power-eq semir-
  ing-parity-class.even-sum-iff not-less mask-eq-sum-exp-nat [symmetric])
  ultimately have ⟨odd (sum ((∩ 2::nat) {q. q < m} div 2 ^ n)⟩
  by (subst euclidean-semiring-cancel-class.sum-div-partition) simp-all
  with False show ?thesis
  by (simp add: mask-eq-sum-exp-nat)
qed
finally show ?thesis .
qed
end

```

57.6 Generic symbolic computations

The following type class contains everything necessary to formulate a division algorithm in ring structures with numerals, restricted to its positive segments.

```

class unique-euclidean-semiring-with-nat-division = unique-euclidean-semiring-with-nat
+
  fixes divmod :: ⟨num ⇒ num ⇒ 'a × 'a⟩
  and divmod-step :: ⟨'a ⇒ 'a × 'a ⇒ 'a × 'a⟩ — These are conceptual definitions
  but force generated code to be monomorphic wrt. particular instances of this class
  which yields a significant speedup.
  assumes divmod-def: ⟨divmod m n = (numeral m div numeral n, numeral m mod
  numeral n)⟩
  and divmod-step-def [simp]: ⟨divmod-step l (q, r) =
  (if euclidean-size l ≤ euclidean-size r then (2 * q + 1, r - l)
  else (2 * q, r))⟩ — This is a formulation of one step (referring to one
  digit position) in school-method division: compare the dividend at the current
  digit position with the remainder from previous division steps and evaluate accordingly.
begin

```

```

lemma fst-divmod:
  ⟨fst (divmod m n) = numeral m div numeral n⟩
  by (simp add: divmod-def)

```

```

lemma snd-divmod:
  ⟨snd (divmod m n) = numeral m mod numeral n⟩
  by (simp add: divmod-def)

```

Following a formulation of school-method division. If the divisor is smaller than the dividend, terminate. If not, shift the dividend to the right until

termination occurs and then reiterate single division steps in the opposite direction.

lemma *divmod-divmod-step*:

$\langle \text{divmod } m \ n = (\text{if } m < n \text{ then } (0, \text{numeral } m) \\ \text{else } \text{divmod-step } (\text{numeral } n) (\text{divmod } m (\text{Num.Bit0 } n))) \rangle$

proof (*cases* $\langle m < n \rangle$)

case *True*

then show *?thesis*

by (*simp add: prod-eq-iff fst-divmod snd-divmod flip: of-nat-numeral of-nat-div of-nat-mod*)

next

case *False*

define *r s t* **where** $\langle r = (\text{numeral } m :: \text{nat}) \rangle \langle s = (\text{numeral } n :: \text{nat}) \rangle \langle t = 2 * s \rangle$

then have $*$: $\langle \text{numeral } m = \text{of-nat } r \rangle \langle \text{numeral } n = \text{of-nat } s \rangle \langle \text{numeral } (\text{num.Bit0 } n) = \text{of-nat } t \rangle$

and $\langle \neg s \leq r \text{ mod } s \rangle$

by (*simp-all add: not-le*)

have *t*: $\langle 2 * (r \text{ div } t) = r \text{ div } s - r \text{ div } s \text{ mod } 2 \rangle$

$\langle r \text{ mod } t = s * (r \text{ div } s \text{ mod } 2) + r \text{ mod } s \rangle$

by (*simp add: Rings.minus-mod-eq-mult-div Groups.mult.commute [of 2] Euclidean-Rings.div-mult2-eq* $\langle t = 2 * s \rangle$)

(*use mod-mult2-eq [of r s 2] in* *simp add: ac-simps* $\langle t = 2 * s \rangle$)

have *rs*: $\langle r \text{ div } s \text{ mod } 2 = 0 \vee r \text{ div } s \text{ mod } 2 = \text{Suc } 0 \rangle$

by *auto*

from $\langle \neg s \leq r \text{ mod } s \rangle$ **have** $\langle s \leq r \text{ mod } t \implies$

$r \text{ div } s = \text{Suc } (2 * (r \text{ div } t)) \wedge$

$r \text{ mod } s = r \text{ mod } t - s \rangle$

using *rs*

by (*auto simp add: t*)

moreover have $\langle r \text{ mod } t < s \implies$

$r \text{ div } s = 2 * (r \text{ div } t) \wedge$

$r \text{ mod } s = r \text{ mod } t \rangle$

using *rs*

by (*auto simp add: t*)

ultimately show *?thesis*

by (*simp add: divmod-def prod-eq-iff split-def Let-def*

*not-less mod-eq-0-iff-dvd Rings.mod-eq-0-iff-dvd False not-le **)

(*simp add: flip: of-nat-numeral of-nat-mult add.commute [of 1] of-nat-div of-nat-mod of-nat-Suc of-nat-diff*)

qed

The division rewrite proper – first, trivial results involving 1

lemma *divmod-trivial* [*simp*]:

$\text{divmod } m \ \text{Num.One} = (\text{numeral } m, 0)$

$\text{divmod } \text{num.One} \ (\text{num.Bit0 } n) = (0, \text{Numeral1})$

$\text{divmod } \text{num.One} \ (\text{num.Bit1 } n) = (0, \text{Numeral1})$

using *divmod-divmod-step [of Num.One]* **by** (*simp-all add: divmod-def*)

Division by an even number is a right-shift

lemma *divmod-cancel* [*simp*]:

$\langle \text{divmod } (\text{Num.Bit0 } m) (\text{Num.Bit0 } n) = (\text{case divmod } m \ n \ \text{of } (q, r) \Rightarrow (q, 2 * r)) \rangle$ (is ?P)

$\langle \text{divmod } (\text{Num.Bit1 } m) (\text{Num.Bit0 } n) = (\text{case divmod } m \ n \ \text{of } (q, r) \Rightarrow (q, 2 * r + 1)) \rangle$ (is ?Q)

proof –

define *r s* **where** $\langle r = (\text{numeral } m :: \text{nat}) \rangle \langle s = (\text{numeral } n :: \text{nat}) \rangle$

then have *: $\langle \text{numeral } m = \text{of-nat } r \rangle \langle \text{numeral } n = \text{of-nat } s \rangle$

$\langle \text{numeral } (\text{num.Bit0 } m) = \text{of-nat } (2 * r) \rangle \langle \text{numeral } (\text{num.Bit0 } n) = \text{of-nat } (2 * s) \rangle$

$\langle \text{numeral } (\text{num.Bit1 } m) = \text{of-nat } (\text{Suc } (2 * r)) \rangle$

by *simp-all*

have **: $\langle \text{Suc } (2 * r) \ \text{div } 2 = r \rangle$

by *simp*

show ?P and ?Q

by (*simp-all add: divmod-def* *)

(*simp-all flip: of-nat-numeral of-nat-div of-nat-mod of-nat-mult add.commute [of 1] of-nat-Suc*

*add: Euclidean-Rings.mod-mult-mult1 div-mult2-eq [of - 2] mod-mult2-eq [of - 2] ***)

qed

The really hard work

lemma *divmod-steps* [*simp*]:

$\text{divmod } (\text{num.Bit0 } m) (\text{num.Bit1 } n) =$

(if $m \leq n$ then $(0, \text{numeral } (\text{num.Bit0 } m))$

else $\text{divmod-step } (\text{numeral } (\text{num.Bit1 } n))$

$(\text{divmod } (\text{num.Bit0 } m)$

$(\text{num.Bit0 } (\text{num.Bit1 } n)))$)

$\text{divmod } (\text{num.Bit1 } m) (\text{num.Bit1 } n) =$

(if $m < n$ then $(0, \text{numeral } (\text{num.Bit1 } m))$

else $\text{divmod-step } (\text{numeral } (\text{num.Bit1 } n))$

$(\text{divmod } (\text{num.Bit1 } m)$

$(\text{num.Bit0 } (\text{num.Bit1 } n)))$)

by (*simp-all add: divmod-divmod-step*)

lemmas *divmod-algorithm-code = divmod-trivial divmod-cancel divmod-steps*

Special case: divisibility

definition *divides-aux* :: $'a \times 'a \Rightarrow \text{bool}$

where

divides-aux *qr* $\longleftrightarrow \text{snd } qr = 0$

lemma *divides-aux-eq* [*simp*]:

divides-aux $(q, r) \longleftrightarrow r = 0$

by (*simp add: divides-aux-def*)

lemma *dvd-numeral-simp* [*simp*]:

numeral m dvd numeral n \longleftrightarrow *divides-aux* (*divmod n m*)
by (*simp add: divmod-def mod-eq-0-iff-dvd*)

Generic computation of quotient and remainder

lemma *numeral-div-numeral* [*simp*]:
numeral k div numeral l = fst (divmod k l)
by (*simp add: fst-divmod*)

lemma *numeral-mod-numeral* [*simp*]:
numeral k mod numeral l = snd (divmod k l)
by (*simp add: snd-divmod*)

lemma *one-div-numeral* [*simp*]:
1 div numeral n = fst (divmod num.One n)
by (*simp add: fst-divmod*)

lemma *one-mod-numeral* [*simp*]:
1 mod numeral n = snd (divmod num.One n)
by (*simp add: snd-divmod*)

end

instantiation *nat* :: *unique-euclidean-semiring-with-nat-division*
begin

definition *divmod-nat* :: *num* \Rightarrow *num* \Rightarrow *nat* \times *nat*

where

divmod'-nat-def: divmod-nat m n = (numeral m div numeral n, numeral m mod numeral n)

definition *divmod-step-nat* :: *nat* \Rightarrow *nat* \times *nat* \Rightarrow *nat* \times *nat*

where

divmod-step-nat l qr = (let (q, r) = qr
*in if r \geq l then (2 * q + 1, r - l)*
*else (2 * q, r))*

instance

by *standard* (*simp-all add: divmod'-nat-def divmod-step-nat-def*)

end

declare *divmod-algorithm-code* [**where** *?a = nat, code*]

lemma *Suc-0-div-numeral* [*simp*]:
 \langle *Suc 0 div numeral Num.One = 1* \rangle
 \langle *Suc 0 div numeral (Num.Bit0 n) = 0* \rangle
 \langle *Suc 0 div numeral (Num.Bit1 n) = 0* \rangle
by *simp-all*

```

lemma Suc-0-mod-numeral [simp]:
  ‹Suc 0 mod numeral Num.One = 0›
  ‹Suc 0 mod numeral (Num.Bit0 n) = 1›
  ‹Suc 0 mod numeral (Num.Bit1 n) = 1›
  by simp-all

instantiation int :: unique-euclidean-semiring-with-nat-division
begin

definition divmod-int :: num ⇒ num ⇒ int × int
where
  divmod-int m n = (numeral m div numeral n, numeral m mod numeral n)

definition divmod-step-int :: int ⇒ int × int ⇒ int × int
where
  divmod-step-int l qr = (let (q, r) = qr
    in if |l| ≤ |r| then (2 * q + 1, r - l)
    else (2 * q, r))

instance
  by standard (auto simp add: divmod-int-def divmod-step-int-def)

end

declare divmod-algorithm-code [where ?'a = int, code]

context
begin

qualified definition adjust-div :: int × int ⇒ int
where
  adjust-div qr = (let (q, r) = qr in q + of-bool (r ≠ 0))

qualified lemma adjust-div-eq [simp, code]:
  adjust-div (q, r) = q + of-bool (r ≠ 0)
  by (simp add: adjust-div-def)

qualified definition adjust-mod :: num ⇒ int ⇒ int
where
  [simp]: adjust-mod l r = (if r = 0 then 0 else numeral l - r)

lemma minus-numeral-div-numeral [simp]:
  – numeral m div numeral n = - (adjust-div (divmod m n) :: int)
proof –
  have int (fst (divmod m n)) = fst (divmod m n)
  by (simp only: fst-divmod divide-int-def) auto
  then show ?thesis
  by (auto simp add: split-def Let-def adjust-div-def divides-aux-def divide-int-def)
qed

```

lemma *minus-numeral-mod-numeral* [*simp*]:
 $- \text{numeral } m \text{ mod numeral } n = \text{adjust-mod } n \text{ (snd (divmod } m \text{ } n) :: \text{int})}$
proof (*cases* $\text{snd (divmod } m \text{ } n) = (0 :: \text{int})$)
 case *True*
 then show *?thesis*
 by (*simp add: mod-eq-0-iff-dvd divides-aux-def*)
next
 case *False*
 then have $\text{int (snd (divmod } m \text{ } n)) = \text{snd (divmod } m \text{ } n) \text{ if } \text{snd (divmod } m \text{ } n) \neq (0 :: \text{int})$
 by (*simp only: snd-divmod modulo-int-def*) *auto*
 then show *?thesis*
 by (*simp add: divides-aux-def adjust-div-def*)
 (*simp add: divides-aux-def modulo-int-def*)
qed

lemma *numeral-div-minus-numeral* [*simp*]:
 $\text{numeral } m \text{ div } - \text{numeral } n = - (\text{adjust-div (divmod } m \text{ } n) :: \text{int})$
proof $-$
 have $\text{int (fst (divmod } m \text{ } n)) = \text{fst (divmod } m \text{ } n)$
 by (*simp only: fst-divmod divide-int-def*) *auto*
 then show *?thesis*
 by (*auto simp add: split-def Let-def adjust-div-def divides-aux-def divide-int-def*)
qed

lemma *numeral-mod-minus-numeral* [*simp*]:
 $\text{numeral } m \text{ mod } - \text{numeral } n = - \text{adjust-mod } n \text{ (snd (divmod } m \text{ } n) :: \text{int})}$
proof (*cases* $\text{snd (divmod } m \text{ } n) = (0 :: \text{int})$)
 case *True*
 then show *?thesis*
 by (*simp add: mod-eq-0-iff-dvd divides-aux-def*)
next
 case *False*
 then have $\text{int (snd (divmod } m \text{ } n)) = \text{snd (divmod } m \text{ } n) \text{ if } \text{snd (divmod } m \text{ } n) \neq (0 :: \text{int})$
 by (*simp only: snd-divmod modulo-int-def*) *auto*
 then show *?thesis*
 by (*simp add: divides-aux-def adjust-div-def*)
 (*simp add: divides-aux-def modulo-int-def*)
qed

lemma *minus-one-div-numeral* [*simp*]:
 $- 1 \text{ div numeral } n = - (\text{adjust-div (divmod Num.One } n) :: \text{int})$
using *minus-numeral-div-numeral* [*of Num.One n*] **by** *simp*

lemma *minus-one-mod-numeral* [*simp*]:
 $- 1 \text{ mod numeral } n = \text{adjust-mod } n \text{ (snd (divmod Num.One } n) :: \text{int})}$
using *minus-numeral-mod-numeral* [*of Num.One n*] **by** *simp*

lemma *one-div-minus-numeral* [*simp*]:

$1 \text{ div } - \text{ numeral } n = - (\text{adjust-div } (\text{divmod Num.One } n) :: \text{int})$
using *numeral-div-minus-numeral* [of *Num.One n*] **by** *simp*

lemma *one-mod-minus-numeral* [*simp*]:

$1 \text{ mod } - \text{ numeral } n = - \text{adjust-mod } n (\text{snd } (\text{divmod Num.One } n) :: \text{int})$
using *numeral-mod-minus-numeral* [of *Num.One n*] **by** *simp*

lemma [*code*]:

fixes $k :: \text{int}$

shows

$k \text{ div } 0 = 0$

$k \text{ mod } 0 = k$

$0 \text{ div } k = 0$

$0 \text{ mod } k = 0$

$k \text{ div Int.Pos Num.One} = k$

$k \text{ mod Int.Pos Num.One} = 0$

$k \text{ div Int.Neg Num.One} = -k$

$k \text{ mod Int.Neg Num.One} = 0$

$\text{Int.Pos } m \text{ div Int.Pos } n = (\text{fst } (\text{divmod } m \ n) :: \text{int})$

$\text{Int.Pos } m \text{ mod Int.Pos } n = (\text{snd } (\text{divmod } m \ n) :: \text{int})$

$\text{Int.Neg } m \text{ div Int.Pos } n = - (\text{adjust-div } (\text{divmod } m \ n) :: \text{int})$

$\text{Int.Neg } m \text{ mod Int.Pos } n = \text{adjust-mod } n (\text{snd } (\text{divmod } m \ n) :: \text{int})$

$\text{Int.Pos } m \text{ div Int.Neg } n = - (\text{adjust-div } (\text{divmod } m \ n) :: \text{int})$

$\text{Int.Pos } m \text{ mod Int.Neg } n = - \text{adjust-mod } n (\text{snd } (\text{divmod } m \ n) :: \text{int})$

$\text{Int.Neg } m \text{ div Int.Neg } n = (\text{fst } (\text{divmod } m \ n) :: \text{int})$

$\text{Int.Neg } m \text{ mod Int.Neg } n = - (\text{snd } (\text{divmod } m \ n) :: \text{int})$

by *simp-all*

end

lemma *divmod-BitM-2-eq* [*simp*]:

$\langle \text{divmod } (\text{Num.BitM } m) (\text{Num.Bit0 Num.One}) = (\text{numeral } m - 1, (1 :: \text{int})) \rangle$
by (*cases m*) *simp-all*

57.6.1 Computation by simplification

lemma *euclidean-size-nat-less-eq-iff*:

$\langle \text{euclidean-size } m \leq \text{euclidean-size } n \iff m \leq n \rangle$ **for** $m \ n :: \text{nat}$
by *simp*

lemma *euclidean-size-int-less-eq-iff*:

$\langle \text{euclidean-size } k \leq \text{euclidean-size } l \iff |k| \leq |l| \rangle$ **for** $k \ l :: \text{int}$
by *auto*

simproc-setup *numeral-divmod*

$(0 \text{ div } 0 :: 'a :: \text{unique-euclidean-semiring-with-nat-division} \mid 0 \text{ mod } 0 :: 'a :: \text{unique-euclidean-semiring-with-nat-division} \mid$

```

0 div 1 :: 'a :: unique-euclidean-semiring-with-nat-division | 0 mod 1 :: 'a ::
unique-euclidean-semiring-with-nat-division |
0 div - 1 :: int | 0 mod - 1 :: int |
0 div numeral b :: 'a :: unique-euclidean-semiring-with-nat-division | 0 mod
numeral b :: 'a :: unique-euclidean-semiring-with-nat-division |
0 div - numeral b :: int | 0 mod - numeral b :: int |
1 div 0 :: 'a :: unique-euclidean-semiring-with-nat-division | 1 mod 0 :: 'a ::
unique-euclidean-semiring-with-nat-division |
1 div 1 :: 'a :: unique-euclidean-semiring-with-nat-division | 1 mod 1 :: 'a ::
unique-euclidean-semiring-with-nat-division |
1 div - 1 :: int | 1 mod - 1 :: int |
1 div numeral b :: 'a :: unique-euclidean-semiring-with-nat-division | 1 mod
numeral b :: 'a :: unique-euclidean-semiring-with-nat-division |
1 div - numeral b :: int | 1 mod - numeral b :: int |
- 1 div 0 :: int | - 1 mod 0 :: int | - 1 div 1 :: int | - 1 mod 1 :: int |
- 1 div - 1 :: int | - 1 mod - 1 :: int | - 1 div numeral b :: int | - 1 mod
numeral b :: int |
- 1 div - numeral b :: int | - 1 mod - numeral b :: int |
numeral a div 0 :: 'a :: unique-euclidean-semiring-with-nat-division | numeral a
mod 0 :: 'a :: unique-euclidean-semiring-with-nat-division |
numeral a div 1 :: 'a :: unique-euclidean-semiring-with-nat-division | numeral a
mod 1 :: 'a :: unique-euclidean-semiring-with-nat-division |
numeral a div - 1 :: int | numeral a mod - 1 :: int |
numeral a div numeral b :: 'a :: unique-euclidean-semiring-with-nat-division |
numeral a mod numeral b :: 'a :: unique-euclidean-semiring-with-nat-division |
numeral a div - numeral b :: int | numeral a mod - numeral b :: int |
- numeral a div 0 :: int | - numeral a mod 0 :: int |
- numeral a div 1 :: int | - numeral a mod 1 :: int |
- numeral a div - 1 :: int | - numeral a mod - 1 :: int |
- numeral a div numeral b :: int | - numeral a mod numeral b :: int |
- numeral a div - numeral b :: int | - numeral a mod - numeral b :: int) = <
let
val if-cong = the (Code.get-case-cong theory const-name <If>);
fun successful-rewrite ctxt ct =
let
val thm = Simplifier.rewrite ctxt ct
in if Thm.is-reflexive thm then NONE else SOME thm end;
val_simps = @{thms div-0 mod-0 div-by-0 mod-by-0 div-by-1 mod-by-1
one-div-numeral one-mod-numeral minus-one-div-numeral minus-one-mod-numeral
one-div-minus-numeral one-mod-minus-numeral
numeral-div-numeral numeral-mod-numeral minus-numeral-div-numeral mi-
nus-numeral-mod-numeral
numeral-div-minus-numeral numeral-mod-minus-numeral
div-minus-minus mod-minus-minus Parity.adjust-div-eq of-bool-eq one-neq-zero
numeral-neq-zero neg-equal-0-iff-equal arith-simps arith-special divmod-trivial
divmod-cancel divmod-steps divmod-step-def fst-conv snd-conv numeral-One
case-prod-beta rel-simps Parity.adjust-mod-def div-minus1-right mod-minus1-right
minus-minus numeral-times-numeral mult-zero-right mult-1-right
euclidean-size-nat-less-eq-iff euclidean-size-int-less-eq-iff diff-nat-numeral nat-numeral}

```

```

    @ [@[lemma 0 = 0  $\longleftrightarrow$  True by simp]];
    val simpset =
      HOL-ss |> Simplifier.simpset-map context
        (Simplifier.add-cong if-cong #> fold Simplifier.add-simp_simps);
    in K (fn ctxt => successful-rewrite (Simplifier.put-simpset simpset ctxt)) end
  › — There is space for improvement here: the calculation itself could be carried out
  outside the logic, and a generic simproc (simplifier setup) for generic calculation
  would be helpful.

```

57.7 Computing congruences modulo 2^q

```

context unique-euclidean-semiring-with-nat-division
begin

```

lemma *cong-exp-iff-simps:*

```

  numeral n mod numeral Num.One = 0
   $\longleftrightarrow$  True
  numeral (Num.Bit0 n) mod numeral (Num.Bit0 q) = 0
   $\longleftrightarrow$  numeral n mod numeral q = 0
  numeral (Num.Bit1 n) mod numeral (Num.Bit0 q) = 0
   $\longleftrightarrow$  False
  numeral m mod numeral Num.One = (numeral n mod numeral Num.One)
   $\longleftrightarrow$  True
  numeral Num.One mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral
  (Num.Bit0 q))
   $\longleftrightarrow$  True
  numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod
  numeral (Num.Bit0 q))
   $\longleftrightarrow$  False
  numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod
  numeral (Num.Bit0 q))
   $\longleftrightarrow$  (numeral n mod numeral q) = 0
  numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral Num.One mod
  numeral (Num.Bit0 q))
   $\longleftrightarrow$  False
  numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n)
  mod numeral (Num.Bit0 q))
   $\longleftrightarrow$  numeral m mod numeral q = (numeral n mod numeral q)
  numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n)
  mod numeral (Num.Bit0 q))
   $\longleftrightarrow$  False
  numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral Num.One mod
  numeral (Num.Bit0 q))
   $\longleftrightarrow$  (numeral m mod numeral q) = 0
  numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n)
  mod numeral (Num.Bit0 q))
   $\longleftrightarrow$  False
  numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n)
  mod numeral (Num.Bit0 q))

```

```

    <=> numeral m mod numeral q = (numeral n mod numeral q)
  by (auto simp add: case-prod-beta dest: arg-cong [of - - even])

end

```

code-identifier

```
code-module Parity → (SML) Arith and (OCaml) Arith and (Haskell) Arith
```

```
lemmas even-of-nat = even-of-nat-iff
```

end

58 Combination and Cancellation Simprocs for Numeral Expressions

theory *Numeral-Simprocs*

imports *Parity*

begin

```
ML-file <~~/src/Provers/Arith/assoc-fold.ML>
```

```
ML-file <~~/src/Provers/Arith/cancel-numerals.ML>
```

```
ML-file <~~/src/Provers/Arith/combine-numerals.ML>
```

```
ML-file <~~/src/Provers/Arith/cancel-numeral-factor.ML>
```

```
ML-file <~~/src/Provers/Arith/extract-common-term.ML>
```

lemmas *semiring-norm* =

```
Let-def arith-simps diff-nat-numeral rel-simps
```

```
if-False if-True
```

```
add-Suc add-numeral-left
```

```
add-neg-numeral-left mult-numeral-left
```

```
numeral-One [symmetric] uminus-numeral-One [symmetric] Suc-eq-plus1
```

```
eq-numeral-iff-iszero not-iszero-Numeral1
```

For *combine-numerals*

lemma *left-add-mult-distrib*: $i*u + (j*u + k) = (i+j)*u + (k::nat)$

by (*simp add: add-mult-distrib*)

For *cancel-numerals*

lemma *nat-diff-add-eq1*:

```
j <= (i::nat) ==> ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)
```

by (*simp split: nat-diff-split add: add-mult-distrib*)

lemma *nat-diff-add-eq2*:

```
i <= (j::nat) ==> ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))
```

by (*simp split: nat-diff-split add: add-mult-distrib*)

lemma *nat-eq-add-iff1*:

$j <= (i::nat) ==> (i*u + m = j*u + n) = ((i-j)*u + m = n)$
by (*auto split: nat-diff-split simp add: add-mult-distrib*)

lemma *nat-eq-add-iff2*:

$i <= (j::nat) ==> (i*u + m = j*u + n) = (m = (j-i)*u + n)$
by (*auto split: nat-diff-split simp add: add-mult-distrib*)

lemma *nat-less-add-iff1*:

$j <= (i::nat) ==> (i*u + m < j*u + n) = ((i-j)*u + m < n)$
by (*auto split: nat-diff-split simp add: add-mult-distrib*)

lemma *nat-less-add-iff2*:

$i <= (j::nat) ==> (i*u + m < j*u + n) = (m < (j-i)*u + n)$
by (*auto split: nat-diff-split simp add: add-mult-distrib*)

lemma *nat-le-add-iff1*:

$j <= (i::nat) ==> (i*u + m <= j*u + n) = ((i-j)*u + m <= n)$
by (*auto split: nat-diff-split simp add: add-mult-distrib*)

lemma *nat-le-add-iff2*:

$i <= (j::nat) ==> (i*u + m <= j*u + n) = (m <= (j-i)*u + n)$
by (*auto split: nat-diff-split simp add: add-mult-distrib*)

For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: $(0::nat) < k ==> (k*m <= k*n) = (m <= n)$
by *auto*

lemma *nat-mult-less-cancel1*: $(0::nat) < k ==> (k*m < k*n) = (m < n)$
by *auto*

lemma *nat-mult-eq-cancel1*: $(0::nat) < k ==> (k*m = k*n) = (m = n)$
by *auto*

lemma *nat-mult-div-cancel1*: $(0::nat) < k ==> (k*m) \text{ div } (k*n) = (m \text{ div } n)$
by *auto*

lemma *nat-mult-dvd-cancel-disj[simp]*:

$(k*m) \text{ dvd } (k*n) = (k=0 \vee m \text{ dvd } (n::nat))$
by (*auto simp: dvd-eq-mod-eq-0 mod-mult-mult1*)

lemma *nat-mult-dvd-cancel1*: $0 < k ==> (k*m) \text{ dvd } (k*n::nat) = (m \text{ dvd } n)$
by(*auto*)

For *cancel-factor*

lemmas *nat-mult-le-cancel-disj = mult-le-cancel1*

lemmas *nat-mult-less-cancel-disj = mult-less-cancel1*

lemma *nat-mult-eq-cancel-disj*:

fixes $k\ m\ n :: \text{nat}$
shows $k * m = k * n \longleftrightarrow k = 0 \vee m = n$
by (fact mult-cancel-left)

lemma *nat-mult-div-cancel-disj*:
fixes $k\ m\ n :: \text{nat}$
shows $(k * m) \text{ div } (k * n) = (\text{if } k = 0 \text{ then } 0 \text{ else } m \text{ div } n)$
by (fact div-mult-mult1-if)

lemma *numeral-times-minus-swap*:
fixes $x :: 'a :: \text{comm-ring-1}$ **shows** $\text{numeral } w * -x = x * - \text{numeral } w$
by (simp add: ac-simps)

ML-file $\langle \text{Tools/numeral-simprocs.ML} \rangle$

simproc-setup *semiring-assoc-fold*
 $((a :: 'a :: \text{comm-semiring-1-cancel}) * b) =$
 $\langle K \text{ Numeral-Simprocs.assoc-fold} \rangle$

simproc-setup *int-combine-numerals*
 $((i :: 'a :: \text{comm-ring-1}) + j \mid (i :: 'a :: \text{comm-ring-1}) - j) =$
 $\langle K \text{ Numeral-Simprocs.combine-numerals} \rangle$

simproc-setup *field-combine-numerals*
 $((i :: 'a :: \{\text{field, ring-char-0}\}) + j$
 $\mid (i :: 'a :: \{\text{field, ring-char-0}\}) - j) =$
 $\langle K \text{ Numeral-Simprocs.field-combine-numerals} \rangle$

simproc-setup *inteq-cancel-numerals*
 $((l :: 'a :: \text{comm-ring-1}) + m = n$
 $\mid (l :: 'a :: \text{comm-ring-1}) = m + n$
 $\mid (l :: 'a :: \text{comm-ring-1}) - m = n$
 $\mid (l :: 'a :: \text{comm-ring-1}) = m - n$
 $\mid (l :: 'a :: \text{comm-ring-1}) * m = n$
 $\mid (l :: 'a :: \text{comm-ring-1}) = m * n$
 $\mid - (l :: 'a :: \text{comm-ring-1}) = m$
 $\mid (l :: 'a :: \text{comm-ring-1}) = - m) =$
 $\langle K \text{ Numeral-Simprocs.eq-cancel-numerals} \rangle$

simproc-setup *intless-cancel-numerals*
 $((l :: 'a :: \text{linordered-idom}) + m < n$
 $\mid (l :: 'a :: \text{linordered-idom}) < m + n$
 $\mid (l :: 'a :: \text{linordered-idom}) - m < n$
 $\mid (l :: 'a :: \text{linordered-idom}) < m - n$
 $\mid (l :: 'a :: \text{linordered-idom}) * m < n$
 $\mid (l :: 'a :: \text{linordered-idom}) < m * n$
 $\mid - (l :: 'a :: \text{linordered-idom}) < m$
 $\mid (l :: 'a :: \text{linordered-idom}) < - m) =$

⟨*K Numeral-Simprocs.less-cancel-numerals*⟩

simproc-setup *intle-cancel-numerals*

$((l::'a::\text{linordered-idom}) + m \leq n$
 $| (l::'a::\text{linordered-idom}) \leq m + n$
 $| (l::'a::\text{linordered-idom}) - m \leq n$
 $| (l::'a::\text{linordered-idom}) \leq m - n$
 $| (l::'a::\text{linordered-idom}) * m \leq n$
 $| (l::'a::\text{linordered-idom}) \leq m * n$
 $| - (l::'a::\text{linordered-idom}) \leq m$
 $| (l::'a::\text{linordered-idom}) \leq - m) =$
 ⟨*K Numeral-Simprocs.le-cancel-numerals*⟩

simproc-setup *ring-eq-cancel-numeral-factor*

$((l::'a::\{\text{idom}, \text{ring-char-0}\}) * m = n$
 $| (l::'a::\{\text{idom}, \text{ring-char-0}\}) = m * n) =$
 ⟨*K Numeral-Simprocs.eq-cancel-numeral-factor*⟩

simproc-setup *ring-less-cancel-numeral-factor*

$((l::'a::\text{linordered-idom}) * m < n$
 $| (l::'a::\text{linordered-idom}) < m * n) =$
 ⟨*K Numeral-Simprocs.less-cancel-numeral-factor*⟩

simproc-setup *ring-le-cancel-numeral-factor*

$((l::'a::\text{linordered-idom}) * m \leq n$
 $| (l::'a::\text{linordered-idom}) \leq m * n) =$
 ⟨*K Numeral-Simprocs.le-cancel-numeral-factor*⟩

simproc-setup *int-div-cancel-numeral-factors*

$((l::'a::\{\text{euclidean-semiring-cancel}, \text{comm-ring-1}, \text{ring-char-0}\}) * m) \text{ div } n$
 $| (l::'a::\{\text{euclidean-semiring-cancel}, \text{comm-ring-1}, \text{ring-char-0}\}) \text{ div } (m * n) =$
 ⟨*K Numeral-Simprocs.div-cancel-numeral-factor*⟩

simproc-setup *divide-cancel-numeral-factor*

$((l::'a::\{\text{field}, \text{ring-char-0}\}) * m) / n$
 $| (l::'a::\{\text{field}, \text{ring-char-0}\}) / (m * n)$
 $| ((\text{numeral } v)::'a::\{\text{field}, \text{ring-char-0}\}) / (\text{numeral } w) =$
 ⟨*K Numeral-Simprocs.divide-cancel-numeral-factor*⟩

simproc-setup *ring-eq-cancel-factor*

$((l::'a::\text{idom}) * m = n | (l::'a::\text{idom}) = m * n) =$
 ⟨*K Numeral-Simprocs.eq-cancel-factor*⟩

simproc-setup *linordered-ring-le-cancel-factor*

$((l::'a::\text{linordered-idom}) * m \leq n$
 $| (l::'a::\text{linordered-idom}) \leq m * n) =$
 ⟨*K Numeral-Simprocs.le-cancel-factor*⟩

simproc-setup *linordered-ring-less-cancel-factor*

$$\begin{aligned} & ((l::'a::\text{linordered-idom}) * m < n \\ & | (l::'a::\text{linordered-idom}) < m * n) = \\ & \langle K \text{ Numeral-Simprocs.less-cancel-factor} \rangle \end{aligned}$$

simproc-setup *int-div-cancel-factor*

$$\begin{aligned} & (((l::'a::\text{euclidean-semiring-cancel}) * m) \text{ div } n \\ & | (l::'a::\text{euclidean-semiring-cancel}) \text{ div } (m * n)) = \\ & \langle K \text{ Numeral-Simprocs.div-cancel-factor} \rangle \end{aligned}$$

simproc-setup *int-mod-cancel-factor*

$$\begin{aligned} & (((l::'a::\text{euclidean-semiring-cancel}) * m) \text{ mod } n \\ & | (l::'a::\text{euclidean-semiring-cancel}) \text{ mod } (m * n)) = \\ & \langle K \text{ Numeral-Simprocs.mod-cancel-factor} \rangle \end{aligned}$$

simproc-setup *dvd-cancel-factor*

$$\begin{aligned} & (((l::'a::\text{idom}) * m) \text{ dvd } n \\ & | (l::'a::\text{idom}) \text{ dvd } (m * n)) = \\ & \langle K \text{ Numeral-Simprocs.dvd-cancel-factor} \rangle \end{aligned}$$

simproc-setup *divide-cancel-factor*

$$\begin{aligned} & (((l::'a::\text{field}) * m) / n \\ & | (l::'a::\text{field}) / (m * n)) = \\ & \langle K \text{ Numeral-Simprocs.divide-cancel-factor} \rangle \end{aligned}$$

ML-file $\langle \text{Tools/nat-numeral-simprocs.ML} \rangle$

simproc-setup *nat-combine-numerals*

$$\begin{aligned} & ((i::\text{nat}) + j | \text{Suc } (i + j)) = \\ & \langle K \text{ Nat-Numeral-Simprocs.combine-numerals} \rangle \end{aligned}$$

simproc-setup *nateq-cancel-numerals*

$$\begin{aligned} & ((l::\text{nat}) + m = n | (l::\text{nat}) = m + n | \\ & (l::\text{nat}) * m = n | (l::\text{nat}) = m * n | \\ & \text{Suc } m = n | m = \text{Suc } n) = \\ & \langle K \text{ Nat-Numeral-Simprocs.eq-cancel-numerals} \rangle \end{aligned}$$

simproc-setup *natless-cancel-numerals*

$$\begin{aligned} & ((l::\text{nat}) + m < n | (l::\text{nat}) < m + n | \\ & (l::\text{nat}) * m < n | (l::\text{nat}) < m * n | \\ & \text{Suc } m < n | m < \text{Suc } n) = \\ & \langle K \text{ Nat-Numeral-Simprocs.less-cancel-numerals} \rangle \end{aligned}$$

simproc-setup *natle-cancel-numerals*

$$\begin{aligned} & ((l::\text{nat}) + m \leq n | (l::\text{nat}) \leq m + n | \\ & (l::\text{nat}) * m \leq n | (l::\text{nat}) \leq m * n | \\ & \text{Suc } m \leq n | m \leq \text{Suc } n) = \\ & \langle K \text{ Nat-Numeral-Simprocs.le-cancel-numerals} \rangle \end{aligned}$$

simproc-setup *natdiff-cancel-numerals*
 $((l::nat) + m) - n \mid (l::nat) - (m + n) \mid$
 $(l::nat) * m - n \mid (l::nat) - m * n \mid$
 $Suc\ m - n \mid m - Suc\ n) =$
 $\langle K\ Nat\ Numeral\ Simprocs.diff\ cancel\ numerals \rangle$

simproc-setup *nat-eq-cancel-numeral-factor*
 $((l::nat) * m = n \mid (l::nat) = m * n) =$
 $\langle K\ Nat\ Numeral\ Simprocs.eq\ cancel\ numeral\ factor \rangle$

simproc-setup *nat-less-cancel-numeral-factor*
 $((l::nat) * m < n \mid (l::nat) < m * n) =$
 $\langle K\ Nat\ Numeral\ Simprocs.less\ cancel\ numeral\ factor \rangle$

simproc-setup *nat-le-cancel-numeral-factor*
 $((l::nat) * m <= n \mid (l::nat) <= m * n) =$
 $\langle K\ Nat\ Numeral\ Simprocs.le\ cancel\ numeral\ factor \rangle$

simproc-setup *nat-div-cancel-numeral-factor*
 $((l::nat) * m\ div\ n \mid (l::nat)\ div\ (m * n)) =$
 $\langle K\ Nat\ Numeral\ Simprocs.div\ cancel\ numeral\ factor \rangle$

simproc-setup *nat-dvd-cancel-numeral-factor*
 $((l::nat) * m\ dvd\ n \mid (l::nat)\ dvd\ (m * n)) =$
 $\langle K\ Nat\ Numeral\ Simprocs.dvd\ cancel\ numeral\ factor \rangle$

simproc-setup *nat-eq-cancel-factor*
 $((l::nat) * m = n \mid (l::nat) = m * n) =$
 $\langle K\ Nat\ Numeral\ Simprocs.eq\ cancel\ factor \rangle$

simproc-setup *nat-less-cancel-factor*
 $((l::nat) * m < n \mid (l::nat) < m * n) =$
 $\langle K\ Nat\ Numeral\ Simprocs.less\ cancel\ factor \rangle$

simproc-setup *nat-le-cancel-factor*
 $((l::nat) * m <= n \mid (l::nat) <= m * n) =$
 $\langle K\ Nat\ Numeral\ Simprocs.le\ cancel\ factor \rangle$

simproc-setup *nat-div-cancel-factor*
 $((l::nat) * m\ div\ n \mid (l::nat)\ div\ (m * n)) =$
 $\langle K\ Nat\ Numeral\ Simprocs.div\ cancel\ factor \rangle$

simproc-setup *nat-dvd-cancel-factor*
 $((l::nat) * m\ dvd\ n \mid (l::nat)\ dvd\ (m * n)) =$
 $\langle K\ Nat\ Numeral\ Simprocs.dvd\ cancel\ factor \rangle$

declaration \langle
 $K\ (Lin\ Arith.add\ simprocs$
 $\ [simproc\ \langle semiring\ assoc\ fold \rangle,$

```

    simpproc <int-combine-numerals>,
    simpproc <inteq-cancel-numerals>,
    simpproc <intless-cancel-numerals>,
    simpproc <intle-cancel-numerals>,
    simpproc <field-combine-numerals>,
    simpproc <nat-combine-numerals>,
    simpproc <nateq-cancel-numerals>,
    simpproc <natless-cancel-numerals>,
    simpproc <natle-cancel-numerals>,
    simpproc <natdiff-cancel-numerals>,
    Numeral-Simprocs.field-divide-cancel-numeral-factor])
  >

```

end

59 Semiring normalization

theory *Semiring-Normalization*

imports *Numeral-Simprocs*

begin

Prelude

```

class comm-semiring-1-cancel-crossproduct = comm-semiring-1-cancel +
  assumes crossproduct-eq:  $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$ 
begin

```

lemma *crossproduct-noteq*:

```

   $a \neq b \wedge c \neq d \longleftrightarrow a * c + b * d \neq a * d + b * c$ 
  by (simp add: crossproduct-eq)

```

lemma *add-scale-eq-noteq*:

```

   $r \neq 0 \implies a = b \wedge c \neq d \implies a + r * c \neq b + r * d$ 

```

proof (*rule notI*)

```

  assume nz:  $r \neq 0$  and cnd:  $a = b \wedge c \neq d$ 

```

```

  and eq:  $a + (r * c) = b + (r * d)$ 

```

```

  have  $(0 * d) + (r * c) = (0 * c) + (r * d)$ 

```

```

  using add-left-imp-eq eq mult-zero-left by (simp add: cnd)

```

```

  then show False using crossproduct-eq [of 0 d] nz cnd by simp

```

qed

lemma *add-0-iff*:

```

   $b = b + a \longleftrightarrow a = 0$ 

```

```

  using add-left-imp-eq [of b a 0] by auto

```

end

subclass (**in** *idom*) *comm-semiring-1-cancel-crossproduct*

proof

```

  fix  $w x y z$ 

```

```

show  $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$ 
proof
  assume  $w * y + x * z = w * z + x * y$ 
  then have  $w * y + x * z - w * z - x * y = 0$  by (simp add: algebra-simps)
  then have  $w * (y - z) - x * (y - z) = 0$  by (simp add: algebra-simps)
  then have  $(y - z) * (w - x) = 0$  by (simp add: algebra-simps)
  then have  $y - z = 0 \vee w - x = 0$  by (rule divisors-zero)
  then show  $w = x \vee y = z$  by auto
qed (auto simp add: ac-simps)
qed

instance nat :: comm-semiring-1-cancel-crossproduct
proof
  fix  $w x y z :: nat$ 
  have  $aux: \bigwedge y z. y < z \implies w * y + x * z = w * z + x * y \implies w = x$ 
  proof -
    fix  $y z :: nat$ 
    assume  $y < z$  then have  $\exists k. z = y + k \wedge k \neq 0$  by (intro exI [of - z - y])
  auto
    then obtain  $k$  where  $z = y + k$  and  $k \neq 0$  by blast
    assume  $w * y + x * z = w * z + x * y$ 
    then have  $(w * y + x * y) + x * k = (w * y + x * y) + w * k$  by (simp add:
     $\langle z = y + k \rangle$  algebra-simps)
    then have  $x * k = w * k$  by simp
    then show  $w = x$  using  $\langle k \neq 0 \rangle$  by simp
  qed
  show  $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$ 
  by (auto simp add: neq-iff dest!: aux)
qed

```

Semiring normalization proper

ML-file $\langle Tools/semiring-normalizer.ML \rangle$

context *comm-semiring-1*

begin

lemma *semiring-normalization-rules* [*no-atp*]:

```

 $(a * m) + (b * m) = (a + b) * m$ 
 $(a * m) + m = (a + 1) * m$ 
 $m + (a * m) = (a + 1) * m$ 
 $m + m = (1 + 1) * m$ 
 $0 + a = a$ 
 $a + 0 = a$ 
 $a * b = b * a$ 
 $(a + b) * c = (a * c) + (b * c)$ 
 $0 * a = 0$ 
 $a * 0 = 0$ 
 $1 * a = a$ 
 $a * 1 = a$ 

```

$$\begin{aligned}
(lx * ly) * (rx * ry) &= (lx * rx) * (ly * ry) \\
(lx * ly) * (rx * ry) &= lx * (ly * (rx * ry)) \\
(lx * ly) * (rx * ry) &= rx * ((lx * ly) * ry) \\
(lx * ly) * rx &= (lx * rx) * ly \\
(lx * ly) * rx &= lx * (ly * rx) \\
lx * (rx * ry) &= (lx * rx) * ry \\
lx * (rx * ry) &= rx * (lx * ry) \\
(a + b) + (c + d) &= (a + c) + (b + d) \\
(a + b) + c &= a + (b + c) \\
a + (c + d) &= c + (a + d) \\
(a + b) + c &= (a + c) + b \\
a + c &= c + a \\
a + (c + d) &= (a + c) + d \\
(x \hat{ } p) * (x \hat{ } q) &= x \hat{ } (p + q) \\
x * (x \hat{ } q) &= x \hat{ } (Suc q) \\
(x \hat{ } q) * x &= x \hat{ } (Suc q) \\
x * x &= x^2 \\
(x * y) \hat{ } q &= (x \hat{ } q) * (y \hat{ } q) \\
(x \hat{ } p) \hat{ } q &= x \hat{ } (p * q) \\
x \hat{ } 0 &= 1 \\
x \hat{ } 1 &= x \\
x * (y + z) &= (x * y) + (x * z) \\
x \hat{ } (Suc q) &= x * (x \hat{ } q) \\
x \hat{ } (2*n) &= (x \hat{ } n) * (x \hat{ } n)
\end{aligned}$$

by (*simp-all add: algebra-simps power-add power2-eq-square*
power-mult-distrib power-mult del: one-add-one)

local-setup <

```

Semiring-Normalizer.declare @{thm comm-semiring-1-axioms}
  {semiring = ([term <x + y>, term <x * y>, term <x \hat{ } n>, term <0>, term <1>],
    @{thms semiring-normalization-rules}),
    ring = ([], []),
    field = ([], []),
    idom = [],
    ideal = []}

```

>

end

context *comm-ring-1*

begin

lemma *ring-normalization-rules* [*no-atp*]:

$$\begin{aligned}
-x &= (-1) * x \\
x - y &= x + (-y)
\end{aligned}$$

by *simp-all*

local-setup <

```

Semiring-Normalizer.declare @{thm comm-ring-1-axioms}

```

```

    {semiring = ([term ⟨x + y⟩, term ⟨x * y⟩, term ⟨x ^ n⟩, term ⟨0⟩, term ⟨1⟩],
      @{thms semiring-normalization-rules}),
      ring = ([term ⟨x - y⟩, term ⟨- x⟩], @{thms ring-normalization-rules}),
      field = ([], []),
      idom = [],
      ideal = []}
  }

end

context comm-semiring-1-cancel-crossproduct
begin

local-setup <
  Semiring-Normalizer.declare @{thm comm-semiring-1-cancel-crossproduct-axioms}
  {semiring = ([term ⟨x + y⟩, term ⟨x * y⟩, term ⟨x ^ n⟩, term ⟨0⟩, term ⟨1⟩],
    @{thms semiring-normalization-rules}),
    ring = ([], []),
    field = ([], []),
    idom = @{thms crossproduct-noteq add-scale-eq-noteq},
    ideal = []}
  }

end

context idom
begin

local-setup <
  Semiring-Normalizer.declare @{thm idom-axioms}
  {semiring = ([term ⟨x + y⟩, term ⟨x * y⟩, term ⟨x ^ n⟩, term ⟨0⟩, term ⟨1⟩],
    @{thms semiring-normalization-rules}),
    ring = ([term ⟨x - y⟩, term ⟨- x⟩], @{thms ring-normalization-rules}),
    field = ([], []),
    idom = @{thms crossproduct-noteq add-scale-eq-noteq},
    ideal = @{thms right-minus-eq add-0-iff}}
  }

end

context field
begin

local-setup <
  Semiring-Normalizer.declare @{thm field-axioms}
  {semiring = ([term ⟨x + y⟩, term ⟨x * y⟩, term ⟨x ^ n⟩, term ⟨0⟩, term ⟨1⟩],
    @{thms semiring-normalization-rules}),
    ring = ([term ⟨x - y⟩, term ⟨- x⟩], @{thms ring-normalization-rules}),
    field = ([term ⟨x / y⟩, term ⟨inverse x⟩], @{thms divide-inverse inverse-eq-divide}),
  }

```



```

    idom = @{thms crossproduct-noteq add-scale-eq-noteq},
    ideal = @{thms right-minus-eq add-0-iff}}
  >

```

end

code-identifier

code-module *Semiring-Normalization* \rightarrow (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

60 Groebner bases

theory *Groebner-Basis*

imports *Semiring-Normalization Parity*

begin

60.1 Groebner Bases

lemmas *bool-simps* = *simp-thms(1-34)* — FIXME move to *HOL.HOL*

lemma *nnf-simps*: — FIXME shadows fact binding in *HOL.HOL*

```

( $\neg(P \wedge Q)$ ) = ( $\neg P \vee \neg Q$ ) ( $\neg(P \vee Q)$ ) = ( $\neg P \wedge \neg Q$ )
( $P \longrightarrow Q$ ) = ( $\neg P \vee Q$ )
( $P = Q$ ) = ( $(P \wedge Q) \vee (\neg P \wedge \neg Q)$ ) ( $\neg \neg(P)$ ) =  $P$ 
by blast+

```

lemma *dnf*:

```

( $P \wedge (Q \vee R)$ ) = ( $(P \wedge Q) \vee (P \wedge R)$ )
( $(Q \vee R) \wedge P$ ) = ( $(Q \wedge P) \vee (R \wedge P)$ )
( $P \wedge Q$ ) = ( $Q \wedge P$ )
( $P \vee Q$ ) = ( $Q \vee P$ )
by blast+

```

lemmas *weak-dnf-simps* = *dnf bool-simps*

lemma *PFalse*:

```

 $P \equiv \text{False} \implies \neg P$ 
 $\neg P \implies (P \equiv \text{False})$ 
by auto

```

named-theorems *algebra pre-simplification rules for algebraic methods*

ML-file \langle *Tools/groebner.ML* \rangle

method-setup *algebra* = \langle

```

  let
    fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ()
    val addN = add

```

```

val delN = del
val any-keyword = keyword addN || keyword delN
val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm);
in
  Scan.optional (keyword addN |-- thms) [] --
  Scan.optional (keyword delN |-- thms) [] >>
  (fn (add-ths, del-ths) => fn ctxt =>
    SIMPLE-METHOD' (Groebner.algebra-tac add-ths del-ths ctxt))
end
› solve polynomial equations over (semi)rings and ideal membership problems using
Groebner bases

```

```

declare dvd-def[algebra]
declare mod-eq-0-iff-dvd[algebra]
declare mod-div-trivial[algebra]
declare mod-mod-trivial[algebra]
declare div-by-0[algebra]
declare mod-by-0[algebra]
declare mult-div-mod-eq[algebra]
declare div-minus-minus[algebra]
declare mod-minus-minus[algebra]
declare div-minus-right[algebra]
declare mod-minus-right[algebra]
declare div-0[algebra]
declare mod-0[algebra]
declare mod-by-1[algebra]
declare div-by-1[algebra]
declare mod-minus1-right[algebra]
declare div-minus1-right[algebra]
declare mod-mult-self2-is-0[algebra]
declare mod-mult-self1-is-0[algebra]

```

```

lemma zmod-eq-0-iff [algebra]:
  ⟨ $m \bmod d = 0 \iff (\exists q. m = d * q)$ ⟩ for  $m d :: int$ 
  by (auto simp add: mod-eq-0-iff-dvd)

```

```

declare dvd-0-left-iff[algebra]
declare zdvd1-eq[algebra]
declare mod-eq-dvd-iff[algebra]
declare nat-mod-eq-iff[algebra]

```

```

context semiring-parity
begin

```

```

declare even-mult-iff [algebra]
declare even-power [algebra]

```

```

end

```

```

context ring-parity
begin

declare even-minus [algebra]

end

declare even-Suc [algebra]
declare even-diff-nat [algebra]

end

```

61 Set intervals

```

theory Set-Interval
imports Parity
begin

```

```

lemma card-2-iff:  $\text{card } S = 2 \iff (\exists x y. S = \{x,y\} \wedge x \neq y)$ 
  by (auto simp: card-Suc-eq numeral-eq-Suc)

```

```

lemma card-2-iff':  $\text{card } S = 2 \iff (\exists x \in S. \exists y \in S. x \neq y \wedge (\forall z \in S. z = x \vee z = y))$ 
  by (auto simp: card-Suc-eq numeral-eq-Suc)

```

```

lemma card-3-iff:  $\text{card } S = 3 \iff (\exists x y z. S = \{x,y,z\} \wedge x \neq y \wedge y \neq z \wedge x \neq z)$ 
  by (fastforce simp: card-Suc-eq numeral-eq-Suc)

```

```

context ord
begin

```

```

definition
  lessThan :: 'a => 'a set ((1{..<-})) where
  {..<u} == {x. x < u}

```

```

definition
  atMost :: 'a => 'a set ((1{..})) where
  {..u} == {x. x ≤ u}

```

```

definition
  greaterThan :: 'a => 'a set ((1{<-..})) where
  {l<..} == {x. l < x}

```

```

definition
  atLeast :: 'a => 'a set ((1{-..})) where
  {l..} == {x. l ≤ x}

```

definition

greaterThanLessThan :: 'a => 'a => 'a set ((1{-<..-}) **where**
 {l<..} == {l<..} Int {..}

definition

atLeastLessThan :: 'a => 'a => 'a set ((1{-..-}) **where**
 {l..} == {l..} Int {..}

definition

greaterThanAtMost :: 'a => 'a => 'a set ((1{-<..-}) **where**
 {l<..} == {l<..} Int {..}

definition

atLeastAtMost :: 'a => 'a => 'a set ((1{-..-}) **where**
 {l..} == {l..} Int {..}

end

A note of warning when using {..} on type *nat*: it is equivalent to {0..} but some lemmas involving {m..} may not exist in {..}–form as well.

syntax (ASCII)

-UNION-le :: 'a => 'a => 'b set => 'b set ((3UN -<=.. / -) [0, 0, 10] 10)
 -UNION-less :: 'a => 'a => 'b set => 'b set ((3UN -<.. / -) [0, 0, 10] 10)
 -INTER-le :: 'a => 'a => 'b set => 'b set ((3INT -<=.. / -) [0, 0, 10] 10)
 -INTER-less :: 'a => 'a => 'b set => 'b set ((3INT -<.. / -) [0, 0, 10] 10)

syntax (latex output)

-UNION-le :: 'a => 'a => 'b set => 'b set ((3U (<unbreakable>- ≤ -) / -) [0, 0, 10] 10)
 -UNION-less :: 'a => 'a => 'b set => 'b set ((3U (<unbreakable>- < -) / -) [0, 0, 10] 10)
 -INTER-le :: 'a => 'a => 'b set => 'b set ((3I (<unbreakable>- ≤ -) / -) [0, 0, 10] 10)
 -INTER-less :: 'a => 'a => 'b set => 'b set ((3I (<unbreakable>- < -) / -) [0, 0, 10] 10)

syntax

-UNION-le :: 'a => 'a => 'b set => 'b set ((3U -≤.. / -) [0, 0, 10] 10)
 -UNION-less :: 'a => 'a => 'b set => 'b set ((3U -<.. / -) [0, 0, 10] 10)
 -INTER-le :: 'a => 'a => 'b set => 'b set ((3I -≤.. / -) [0, 0, 10] 10)
 -INTER-less :: 'a => 'a => 'b set => 'b set ((3I -<.. / -) [0, 0, 10] 10)

translations

$\bigcup i \leq n. A \equiv \bigcup i \in \{..n\}. A$
 $\bigcup i < n. A \equiv \bigcup i \in \{..n\}. A$
 $\bigcap i \leq n. A \equiv \bigcap i \in \{..n\}. A$
 $\bigcap i < n. A \equiv \bigcap i \in \{..n\}. A$

61.1 Various equivalences

lemma (in ord) *lessThan-iff* [iff]: $(i \in \text{lessThan } k) = (i < k)$
 by (simp add: lessThan-def)

lemma *Compl-lessThan* [simp]:
 $!!k:: 'a::\text{linorder}. \neg \text{lessThan } k = \text{atLeast } k$
 by (auto simp add: lessThan-def atLeast-def)

lemma *single-Diff-lessThan* [simp]: $!!k:: 'a::\text{preorder}. \{k\} - \text{lessThan } k = \{k\}$
 by auto

lemma (in ord) *greaterThan-iff* [iff]: $(i \in \text{greaterThan } k) = (k < i)$
 by (simp add: greaterThan-def)

lemma *Compl-greaterThan* [simp]:
 $!!k:: 'a::\text{linorder}. \neg \text{greaterThan } k = \text{atMost } k$
 by (auto simp add: greaterThan-def atMost-def)

lemma *Compl-atMost* [simp]: $!!k:: 'a::\text{linorder}. \neg \text{atMost } k = \text{greaterThan } k$
 by (metis Compl-greaterThan double-complement)

lemma (in ord) *atLeast-iff* [iff]: $(i \in \text{atLeast } k) = (k \leq i)$
 by (simp add: atLeast-def)

lemma *Compl-atLeast* [simp]: $!!k:: 'a::\text{linorder}. \neg \text{atLeast } k = \text{lessThan } k$
 by (auto simp add: lessThan-def atLeast-def)

lemma (in ord) *atMost-iff* [iff]: $(i \in \text{atMost } k) = (i \leq k)$
 by (simp add: atMost-def)

lemma *atMost-Int-atLeast*: $!!n:: 'a::\text{order}. \text{atMost } n \text{ Int } \text{atLeast } n = \{n\}$
 by (blast intro: order-antisym)

lemma (in linorder) *lessThan-Int-lessThan*: $\{a <..\} \cap \{b <..\} = \{\max a b <..\}$
 by auto

lemma (in linorder) *greaterThan-Int-greaterThan*: $\{..< a\} \cap \{..< b\} = \{..< \min a b\}$
 by auto

61.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-empty-triv* [simp]: $\{\{\}\} = \text{UNIV}$
 by auto

lemma *atMost-UNIV-triv* [simp]: $\{..\text{UNIV}\} = \text{UNIV}$
 by auto

lemma *atLeast-subset-iff* [iff]:

(*atLeast* $x \subseteq$ *atLeast* y) = ($y \leq (x::'a::preorder)$)
by (*blast intro: order-trans*)

lemma *atLeast-eq-iff* [*iff*]:
(*atLeast* $x =$ *atLeast* y) = ($x = (y::'a::order)$)
by (*blast intro: order-antisym order-trans*)

lemma *greaterThan-subset-iff* [*iff*]:
(*greaterThan* $x \subseteq$ *greaterThan* y) = ($y \leq (x::'a::linorder)$)
unfolding *greaterThan-def* **by** (*auto simp: linorder-not-less [symmetric]*)

lemma *greaterThan-eq-iff* [*iff*]:
(*greaterThan* $x =$ *greaterThan* y) = ($x = (y::'a::linorder)$)
by (*auto simp: elim!: equalityE*)

lemma *atMost-subset-iff* [*iff*]: (*atMost* $x \subseteq$ *atMost* y) = ($x \leq (y::'a::preorder)$)
by (*blast intro: order-trans*)

lemma *atMost-eq-iff* [*iff*]: (*atMost* $x =$ *atMost* y) = ($x = (y::'a::order)$)
by (*blast intro: order-antisym order-trans*)

lemma *lessThan-subset-iff* [*iff*]:
(*lessThan* $x \subseteq$ *lessThan* y) = ($x \leq (y::'a::linorder)$)
unfolding *lessThan-def* **by** (*auto simp: linorder-not-less [symmetric]*)

lemma *lessThan-eq-iff* [*iff*]:
(*lessThan* $x =$ *lessThan* y) = ($x = (y::'a::linorder)$)
by (*auto simp: elim!: equalityE*)

lemma *lessThan-strict-subset-iff*:
fixes $m\ n :: 'a::linorder$
shows $\{.. m \} < \{.. n \} \longleftrightarrow m < n$
by (*metis leD lessThan-subset-iff linorder-linear not-less-iff-gr-or-eq psubset-eq*)

lemma (**in** *linorder*) *Ici-subset-Ioi-iff*: $\{a ..\} \subseteq \{b <..\} \longleftrightarrow b < a$
by *auto*

lemma (**in** *linorder*) *Iic-subset-Iio-iff*: $\{.. a\} \subseteq \{.. < b\} \longleftrightarrow a < b$
by *auto*

lemma (**in** *preorder*) *Ioi-le-Ico*: $\{a <..\} \subseteq \{a ..\}$
by (*auto intro: less-imp-le*)

61.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [*simp*]: ($i \in \{l < .. < u\}$) = ($l < i \wedge i < u$)

by (simp add: greaterThanLessThan-def)

lemma *atLeastLessThan-iff* [simp]: $(i \in \{l..<u\}) = (l \leq i \wedge i < u)$
 by (simp add: atLeastLessThan-def)

lemma *greaterThanAtMost-iff* [simp]: $(i \in \{l<..u\}) = (l < i \wedge i \leq u)$
 by (simp add: greaterThanAtMost-def)

lemma *atLeastAtMost-iff* [simp]: $(i \in \{l..u\}) = (l \leq i \wedge i \leq u)$
 by (simp add: atLeastAtMost-def)

The above four lemmas could be declared as iffs. Unfortunately this breaks many proofs. Since it only helps blast, it is better to leave them alone.

lemma *greaterThanLessThan-eq*: $\{a <..**b\} = \{a <..\} \cap \{..**b\}****$
 by auto

lemma (in order) *atLeastLessThan-eq-atLeastAtMost-diff*:
 $\{a..**b\} = \{a..b\} - \{b\}**$
 by (auto simp add: atLeastLessThan-def atLeastAtMost-def)

lemma (in order) *greaterThanAtMost-eq-atLeastAtMost-diff*:
 $\{a<..**b\} = \{a..b\} - \{a\}**$
 by (auto simp add: greaterThanAtMost-def atLeastAtMost-def)

end

61.3.1 Emptyness, singletons, subset

context *preorder*
begin

lemma *atLeastatMost-empty-iff*[simp]:
 $\{a..b\} = \{\} \iff (\neg a \leq b)$
 by auto (blast intro: order-trans)

lemma *atLeastatMost-empty-iff2*[simp]:
 $\{\} = \{a..b\} \iff (\neg a \leq b)$
 by auto (blast intro: order-trans)

lemma *atLeastLessThan-empty-iff*[simp]:
 $\{a..**b\} = \{\} \iff (\neg a < b)**$
 by auto (blast intro: le-less-trans)

lemma *atLeastLessThan-empty-iff2*[simp]:
 $\{\} = \{a..**b\} \iff (\neg a < b)**$
 by auto (blast intro: le-less-trans)

lemma *greaterThanAtMost-empty-iff*[simp]: $\{k<..**l\} = \{\} \iff \neg k < l**$
 by auto (blast intro: less-le-trans)

lemma *greaterThanAtMost-empty-iff2[simp]*: $\{\} = \{k < .. l\} \longleftrightarrow \neg k < l$
by *auto (blast intro: less-le-trans)*

lemma *atLeastatMost-subset-iff[simp]*:
 $\{a..b\} \subseteq \{c..d\} \longleftrightarrow (\neg a \leq b) \vee c \leq a \wedge b \leq d$
unfolding *atLeastAtMost-def atLeast-def atMost-def*
by *(blast intro: order-trans)*

lemma *atLeastatMost-psubset-iff*:
 $\{a..b\} < \{c..d\} \longleftrightarrow$
 $((\neg a \leq b) \vee c \leq a \wedge b \leq d \wedge (c < a \vee b < d)) \wedge c \leq d$
by *(simp add: psubset-eq set-eq-iff less-le-not-le)(blast intro: order-trans)*

lemma *atLeastAtMost-subseteq-atLeastLessThan-iff*:
 $\{a..b\} \subseteq \{c ..< d\} \longleftrightarrow (a \leq b \longrightarrow c \leq a \wedge b < d)$
by *auto (blast intro: local.order-trans local.le-less-trans elim:)+*

lemma *Icc-subset-Ici-iff[simp]*:
 $\{l..h\} \subseteq \{l'..'\} = (\neg l \leq h \vee l \geq l')$
by *(auto simp: subset-eq intro: order-trans)*

lemma *Icc-subset-Iic-iff[simp]*:
 $\{l..h\} \subseteq \{..h'\} = (\neg l \leq h \vee h \leq h')$
by *(auto simp: subset-eq intro: order-trans)*

lemma *not-Ici-eq-empty[simp]*: $\{l..'\} \neq \{\}$
by *(auto simp: set-eq-iff)*

lemma *not-Iic-eq-empty[simp]*: $\{..h\} \neq \{\}$
by *(auto simp: set-eq-iff)*

lemmas *not-empty-eq-Ici-eq-empty[simp]* = *not-Ici-eq-empty[symmetric]*
lemmas *not-empty-eq-Iic-eq-empty[simp]* = *not-Iic-eq-empty[symmetric]*

end

context *order*
begin

lemma *atLeastatMost-empty[simp]*: $b < a \implies \{a..b\} = \{\}$
and *atLeastatMost-empty'[simp]*: $\neg a \leq b \implies \{a..b\} = \{\}$
by *(auto simp: atLeastAtMost-def atLeast-def atMost-def)*

lemma *atLeastLessThan-empty[simp]*:
 $b \leq a \implies \{a..<b\} = \{\}$
by *(auto simp: atLeastLessThan-def)*

lemma *greaterThanAtMost-empty[simp]*: $l \leq k \implies \{k < .. l\} = \{\}$

by(*auto simp: greaterThanAtMost-def greaterThan-def atMost-def*)

lemma *greaterThanLessThan-empty[simp]*: $l \leq k \implies \{k < .. < l\} = \{\}$
by(*auto simp: greaterThanLessThan-def greaterThan-def lessThan-def*)

lemma *atLeastAtMost-singleton [simp]*: $\{a .. a\} = \{a\}$
by (*auto simp add: atLeastAtMost-def atMost-def atLeast-def*)

lemma *atLeastAtMost-singleton'*: $a = b \implies \{a .. b\} = \{a\}$ **by** *simp*

lemma *Icc-eq-Icc[simp]*:
 $\{l .. h\} = \{l' .. h'\} = (l=l' \wedge h=h' \vee \neg l \leq h \wedge \neg l' \leq h')$
by (*simp add: order-class.order.eq-iff*) (*auto intro: order-trans*)

lemma (**in** *linorder*) *Ico-eq-Ico*:
 $\{l .. < h\} = \{l' .. < h'\} = (l=l' \wedge h=h' \vee \neg l < h \wedge \neg l' < h')$
by (*metis atLeastLessThan-empty-iff2 nle-le not-less ord.atLeastLessThan-iff*)

lemma *atLeastAtMost-singleton-iff[simp]*:
 $\{a .. b\} = \{c\} \longleftrightarrow a = b \wedge b = c$

proof

assume $\{a .. b\} = \{c\}$

hence $*$: $\neg (\neg a \leq b)$ **unfolding** *atLeastatMost-empty-iff[symmetric]* **by** *simp*

with $\langle \{a .. b\} = \{c\} \rangle$ **have** $c \leq a \wedge b \leq c$ **by** *auto*

with $*$ **show** $a = b \wedge b = c$ **by** *auto*

qed *simp*

end

context *no-top*
begin

lemma *not-UNIV-le-Icc[simp]*: $\neg UNIV \subseteq \{l .. h\}$
using *gt-ex[of h]* **by**(*auto simp: subset-eq less-le-not-le*)

lemma *not-UNIV-le-Iic[simp]*: $\neg UNIV \subseteq \{..h\}$
using *gt-ex[of h]* **by**(*auto simp: subset-eq less-le-not-le*)

lemma *not-Ici-le-Icc[simp]*: $\neg \{l ..\} \subseteq \{l' .. h'\}$
using *gt-ex[of h]*
by(*auto simp: subset-eq less-le*)(*blast dest:antisym-conv intro: order-trans*)

lemma *not-Ici-le-Iic[simp]*: $\neg \{l ..\} \subseteq \{..h'\}$
using *gt-ex[of h]*
by(*auto simp: subset-eq less-le*)(*blast dest:antisym-conv intro: order-trans*)

end

context *no-bot*
begin

lemma *not-UNIV-le-Ici*[simp]: $\neg UNIV \subseteq \{l..\}$
using *lt-ex*[of *l*] **by**(*auto simp: subset-eq less-le-not-le*)

lemma *not-Iic-le-Icc*[simp]: $\neg \{..h\} \subseteq \{l'..h'\}$
using *lt-ex*[of *l'*]
by(*auto simp: subset-eq less-le*)(*blast dest:antisym-conv intro: order-trans*)

lemma *not-Iic-le-Ici*[simp]: $\neg \{..h\} \subseteq \{l'..\}$
using *lt-ex*[of *l'*]
by(*auto simp: subset-eq less-le*)(*blast dest:antisym-conv intro: order-trans*)

end

context *no-top*
begin

lemma *not-UNIV-eq-Icc*[simp]: $\neg UNIV = \{l'..h'\}$
using *gt-ex*[of *h'*] **by**(*auto simp: set-eq-iff less-le-not-le*)

lemmas *not-Icc-eq-UNIV*[simp] = *not-UNIV-eq-Icc*[*symmetric*]

lemma *not-UNIV-eq-Iic*[simp]: $\neg UNIV = \{..h'\}$
using *gt-ex*[of *h'*] **by**(*auto simp: set-eq-iff less-le-not-le*)

lemmas *not-Iic-eq-UNIV*[simp] = *not-UNIV-eq-Iic*[*symmetric*]

lemma *not-Icc-eq-Ici*[simp]: $\neg \{l..h\} = \{l'..\}$
unfolding *atLeastAtMost-def* **using** *not-Ici-le-Iic*[of *l'*] **by** *blast*

lemmas *not-Ici-eq-Icc*[simp] = *not-Icc-eq-Ici*[*symmetric*]

lemma *not-Iic-eq-Ici*[simp]: $\neg \{..h\} = \{l'..\}$
using *not-Ici-le-Iic*[of *l' h*] **by** *blast*

lemmas *not-Ici-eq-Iic*[simp] = *not-Iic-eq-Ici*[*symmetric*]

end

context *no-bot*
begin

lemma *not-UNIV-eq-Ici*[simp]: $\neg UNIV = \{l'..\}$
using *lt-ex*[of *l'*] **by**(*auto simp: set-eq-iff less-le-not-le*)

lemmas *not-Ici-eq-UNIV[simp]* = *not-UNIV-eq-Ici[symmetric]*

lemma *not-Icc-eq-Iic[simp]*: $\neg \{l..h\} = \{..h\}$
unfolding *atLeastAtMost-def* **using** *not-Iic-le-Ici[of h]* **by** *blast*

lemmas *not-Iic-eq-Icc[simp]* = *not-Icc-eq-Iic[symmetric]*

end

context *dense-linorder*
begin

lemma *greaterThanLessThan-empty-iff[simp]*:
 $\{ a <..< b \} = \{ \} \longleftrightarrow b \leq a$
using *dense[of a b]* **by** (*cases a < b*) *auto*

lemma *greaterThanLessThan-empty-iff2[simp]*:
 $\{ \} = \{ a <..< b \} \longleftrightarrow b \leq a$
using *dense[of a b]* **by** (*cases a < b*) *auto*

lemma *atLeastLessThan-subseteq-atLeastAtMost-iff*:
 $\{ a ..< b \} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
using *dense[of max a d b]*
by (*force simp: subset-eq Ball-def not-less[symmetric]*)

lemma *greaterThanAtMost-subseteq-atLeastAtMost-iff*:
 $\{ a <.. b \} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
using *dense[of a min c b]*
by (*force simp: subset-eq Ball-def not-less[symmetric]*)

lemma *greaterThanLessThan-subseteq-atLeastAtMost-iff*:
 $\{ a <..< b \} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
using *dense[of a min c b] dense[of max a d b]*
by (*force simp: subset-eq Ball-def not-less[symmetric]*)

lemma *greaterThanLessThan-subseteq-greaterThanLessThan*:
 $\{ a <..< b \} \subseteq \{ c <..< d \} \longleftrightarrow (a < b \longrightarrow a \geq c \wedge b \leq d)$
using *dense[of a min c b] dense[of max a d b]*
by (*force simp: subset-eq Ball-def not-less[symmetric]*)

lemma *greaterThanAtMost-subseteq-atLeastLessThan-iff*:
 $\{ a <.. b \} \subseteq \{ c ..< d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b < d)$
using *dense[of a min c b]*
by (*force simp: subset-eq Ball-def not-less[symmetric]*)

lemma *greaterThanLessThan-subseteq-atLeastLessThan-iff*:
 $\{ a <..< b \} \subseteq \{ c ..< d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$

```

using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanLessThan-subseteq-greaterThanAtMost-iff:
   $\{a <..< b\} \subseteq \{c <..< d\} \iff (a < b \implies c \leq a \wedge b \leq d)$ 
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

end

context no-top
begin

lemma greaterThan-non-empty[simp]:  $\{x <..< \} \neq \{\}$ 
using gt-ex[of x] by auto

end

context no-bot
begin

lemma lessThan-non-empty[simp]:  $\{..< x\} \neq \{\}$ 
using lt-ex[of x] by auto

end

lemma (in linorder) atLeastLessThan-subset-iff:
   $\{a..<b\} \subseteq \{c..<d\} \implies b \leq a \vee c \leq a \wedge b \leq d$ 
proof (cases a < b)
  case True
    assume assm:  $\{a..<b\} \subseteq \{c..<d\}$ 
    then have 1:  $c \leq a \wedge a \leq d$ 
      using True by (auto simp add: subset-eq Ball-def)
    then have 2:  $b \leq d$ 
      using assm by (auto simp add: subset-eq)
    from 1 2 show ?thesis
      by simp
  qed (auto)

lemma atLeastLessThan-inj:
  fixes a b c d :: 'a::linorder
  assumes eq:  $\{a ..< b\} = \{c ..< d\}$  and a < b c < d
  shows a = c b = d
using assms by (metis atLeastLessThan-subset-iff eq less-le-not-le antisym-conv2
subset-refl)+

lemma atLeastLessThan-eq-iff:
  fixes a b c d :: 'a::linorder
  assumes a < b c < d

```

shows $\{a \dots b\} = \{c \dots d\} \longleftrightarrow a = c \wedge b = d$
using *atLeastLessThan-inj* *assms* **by** *auto*

lemma (**in** *linorder*) *Ioc-inj*:

$\langle \{a < \dots b\} = \{c < \dots d\} \longleftrightarrow (b \leq a \wedge d \leq c) \vee a = c \wedge b = d \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)

proof

assume $?Q$

then show $?P$

by *auto*

next

assume $?P$

then have $\langle a < x \wedge x \leq b \longleftrightarrow c < x \wedge x \leq d \rangle$ **for** x

by (*simp* *add: set-eq-iff*)

from *this* [*of a*] *this* [*of b*] *this* [*of c*] *this* [*of d*] **show** $?Q$

by *auto*

qed

lemma (**in** *order*) *Iio-Int-singleton*: $\{\dots < k\} \cap \{x\} = (\text{if } x < k \text{ then } \{x\} \text{ else } \{\})$

by *auto*

lemma (**in** *linorder*) *Ioc-subset-iff*: $\{a < \dots b\} \subseteq \{c < \dots d\} \longleftrightarrow (b \leq a \vee c \leq a \wedge b \leq d)$

by (*auto simp: subset-eq Ball-def*) (*metis less-le not-less*)

lemma (**in** *order-bot*) *atLeast-eq-UNIV-iff*: $\{x \dots\} = \text{UNIV} \longleftrightarrow x = \text{bot}$

by (*auto simp: set-eq-iff intro: le-bot*)

lemma (**in** *order-top*) *atMost-eq-UNIV-iff*: $\{\dots x\} = \text{UNIV} \longleftrightarrow x = \text{top}$

by (*auto simp: set-eq-iff intro: top-le*)

lemma (**in** *bounded-lattice*) *atLeastAtMost-eq-UNIV-iff*:

$\{x \dots y\} = \text{UNIV} \longleftrightarrow (x = \text{bot} \wedge y = \text{top})$

by (*auto simp: set-eq-iff intro: top-le le-bot*)

lemma *Iio-eq-empty-iff*: $\{\dots < n :: 'a :: \{\text{linorder}, \text{order-bot}\}\} = \{\} \longleftrightarrow n = \text{bot}$

by (*auto simp: set-eq-iff not-less le-bot*)

lemma *lessThan-empty-iff*: $\{\dots < n :: \text{nat}\} = \{\} \longleftrightarrow n = 0$

by (*simp add: Iio-eq-empty-iff bot-nat-def*)

lemma *mono-image-least*:

assumes *f-mono*: *mono f* **and** *f-img*: $f \text{ ` } \{m \dots n\} = \{m' \dots n'\}$ $m < n$

shows $f m = m'$

proof –

from *f-img* **have** $\{m' \dots n'\} \neq \{\}$

by (*metis atLeastLessThan-empty-iff image-is-empty*)

with *f-img* **have** $m' \in f \text{ ` } \{m \dots n\}$ **by** *auto*

then obtain k **where** $f k = m'$ $m \leq k$ **by** *auto*

moreover have $m' \leq f m$ **using** *f-img* **by** *auto*

ultimately show $f m = m'$
 using f -mono by (auto elim: monoE[where $x=m$ and $y=k$])
 qed

61.4 Infinite intervals

context *dense-linorder*
 begin

lemma *infinite-Ioo*:

assumes $a < b$

shows \neg finite $\{a <..< < b\}$

proof

assume fin : finite $\{a <..< < b\}$

moreover have ne : $\{a <..< < b\} \neq \{\}$

using $\langle a < b \rangle$ by auto

ultimately have $a < \text{Max } \{a <..< < b\} \text{Max } \{a <..< < b\} < b$

using *Max-in*[of $\{a <..< < b\}$] by auto

then obtain x where $\text{Max } \{a <..< < b\} < x < b$

using *dense*[of $\text{Max } \{a <..< < b\} b$] by auto

then have $x \in \{a <..< < b\}$

using $\langle a < \text{Max } \{a <..< < b\} \rangle$ by auto

then have $x \leq \text{Max } \{a <..< < b\}$

using fin by auto

with $\langle \text{Max } \{a <..< < b\} < x \rangle$ show *False* by auto

qed

lemma *infinite-Icc*: $a < b \implies \neg$ finite $\{a .. b\}$

using *greaterThanLessThan-subseteq-atLeastAtMost-iff*[of $a b a b$] *infinite-Ioo*[of $a b$]

by (auto dest: *finite-subset*)

lemma *infinite-Ico*: $a < b \implies \neg$ finite $\{a ..< b\}$

using *greaterThanLessThan-subseteq-atLeastLessThan-iff*[of $a b a b$] *infinite-Ioo*[of $a b$]

by (auto dest: *finite-subset*)

lemma *infinite-Ioc*: $a < b \implies \neg$ finite $\{a <.. b\}$

using *greaterThanLessThan-subseteq-greaterThanAtMost-iff*[of $a b a b$] *infinite-Ioo*[of $a b$]

by (auto dest: *finite-subset*)

lemma *infinite-Ioo-iff* [*simp*]: $\text{infinite } \{a <..< < b\} \longleftrightarrow a < b$

using *not-less-iff-gr-or-eq* by (fastforce *simp*: *infinite-Ioo*)

lemma *infinite-Icc-iff* [*simp*]: $\text{infinite } \{a .. b\} \longleftrightarrow a < b$

using *not-less-iff-gr-or-eq* by (fastforce *simp*: *infinite-Icc*)

lemma *infinite-Ico-iff* [*simp*]: $\text{infinite } \{a <..< b\} \longleftrightarrow a < b$

using *not-less-iff-gr-or-eq* by (fastforce simp: *infinite-Ico*)

lemma *infinite-Ioc-iff* [simp]: *infinite* $\{a <.. b\} \longleftrightarrow a < b$
 using *not-less-iff-gr-or-eq* by (fastforce simp: *infinite-Ioc*)

end

lemma *infinite-Iio*: \neg *finite* $\{.. < a :: 'a :: \{no-bot, linorder\}\}$

proof

assume *finite* $\{.. < a\}$

then have *: $\bigwedge x. x < a \implies \text{Min } \{.. < a\} \leq x$

by *auto*

obtain *x* where $x < a$

using *lt-ex* by *auto*

obtain *y* where $y < \text{Min } \{.. < a\}$

using *lt-ex* by *auto*

also have $\text{Min } \{.. < a\} \leq x$

using $\langle x < a \rangle$ by *fact*

also note $\langle x < a \rangle$

finally have $\text{Min } \{.. < a\} \leq y$

by *fact*

with $\langle y < \text{Min } \{.. < a\} \rangle$ show *False* by *auto*

qed

lemma *infinite-Iic*: \neg *finite* $\{.. a :: 'a :: \{no-bot, linorder\}\}$

using *infinite-Iio*[of *a*] *finite-subset*[of $\{.. < a\} \{.. a\}$]

by (*auto* simp: *subset-eq less-imp-le*)

lemma *infinite-Ioi*: \neg *finite* $\{a :: 'a :: \{no-top, linorder\} <..\}$

proof

assume *finite* $\{a <..\}$

then have *: $\bigwedge x. a < x \implies x \leq \text{Max } \{a <..\}$

by *auto*

obtain *y* where $\text{Max } \{a <..\} < y$

using *gt-ex* by *auto*

obtain *x* where $x: a < x$

using *gt-ex* by *auto*

also from *x* have $x \leq \text{Max } \{a <..\}$

by *fact*

also note $\langle \text{Max } \{a <..\} < y \rangle$

finally have $y \leq \text{Max } \{a <..\}$

by *fact*

with $\langle \text{Max } \{a <..\} < y \rangle$ show *False* by *auto*

qed

lemma *infinite-Ici*: \neg *finite* $\{a :: 'a :: \{no-top, linorder\} ..\}$

using *infinite-Ioi*[of *a*] *finite-subset*[of {*a* <..*a* ..}]
by (*auto simp: subset-eq less-imp-le*)

61.4.1 Intersection

context *linorder*
begin

lemma *Int-atLeastAtMost*[*simp*]: {*a*..*b*} *Int* {*c*..*d*} = {*max a c* .. *min b d*}
by *auto*

lemma *Int-atLeastAtMostR1*[*simp*]: {..*b*} *Int* {*c*..*d*} = {*c* .. *min b d*}
by *auto*

lemma *Int-atLeastAtMostR2*[*simp*]: {*a*..*a*} *Int* {*c*..*d*} = {*max a c* .. *d*}
by *auto*

lemma *Int-atLeastAtMostL1*[*simp*]: {*a*..*b*} *Int* {..*d*} = {*a* .. *min b d*}
by *auto*

lemma *Int-atLeastAtMostL2*[*simp*]: {*a*..*b*} *Int* {*c*..*c*} = {*max a c* .. *b*}
by *auto*

lemma *Int-atLeastLessThan*[*simp*]: {*a*..*b*} *Int* {*c*..*d*} = {*max a c* ..< *min b d*}
by *auto*

lemma *Int-greaterThanAtMost*[*simp*]: {*a*<..*b*} *Int* {*c*<..*d*} = {*max a c* <..*min b d*}
by *auto*

lemma *Int-greaterThanLessThan*[*simp*]: {*a*<..*b*} *Int* {*c*<..*d*} = {*max a c* <..*min b d*}
by *auto*

lemma *Int-atMost*[*simp*]: {..*a*} ∩ {..*b*} = {..*min a b*}
by (*auto simp: min-def*)

lemma *Ioc-disjoint*: {*a*<..*b*} ∩ {*c*<..*d*} = {} \longleftrightarrow *b* ≤ *a* ∨ *d* ≤ *c* ∨ *b* ≤ *c* ∨ *d* ≤ *a*
by *auto*

end

context *complete-lattice*
begin

lemma
shows *Sup-atLeast*[*simp*]: *Sup* {*x* ..} = *top*
and *Sup-greaterThanAtLeast*[*simp*]: *x* < *top* \implies *Sup* {*x* <..*a*} = *top*

and $Sup\text{-}atMost[simp]: Sup \{.. y\} = y$
and $Sup\text{-}atLeastAtMost[simp]: x \leq y \implies Sup \{ x .. y\} = y$
and $Sup\text{-}greaterThanAtMost[simp]: x < y \implies Sup \{ x <.. y\} = y$
by (*auto intro!*: $Sup\text{-}eqI$)

lemma

shows $Inf\text{-}atMost[simp]: Inf \{.. x\} = bot$
and $Inf\text{-}atMostLessThan[simp]: top < x \implies Inf \{..< x\} = bot$
and $Inf\text{-}atLeast[simp]: Inf \{x ..\} = x$
and $Inf\text{-}atLeastAtMost[simp]: x \leq y \implies Inf \{ x .. y\} = x$
and $Inf\text{-}atLeastLessThan[simp]: x < y \implies Inf \{ x ..< y\} = x$
by (*auto intro!*: $Inf\text{-}eqI$)

end

lemma

fixes $x y :: 'a :: \{complete\text{-}lattice, dense\text{-}linorder\}$
shows $Sup\text{-}lessThan[simp]: Sup \{..< y\} = y$
and $Sup\text{-}atLeastLessThan[simp]: x < y \implies Sup \{ x ..< y\} = y$
and $Sup\text{-}greaterThanLessThan[simp]: x < y \implies Sup \{ x <..< y\} = y$
and $Inf\text{-}greaterThan[simp]: Inf \{x <.. \} = x$
and $Inf\text{-}greaterThanAtMost[simp]: x < y \implies Inf \{ x <.. y\} = x$
and $Inf\text{-}greaterThanLessThan[simp]: x < y \implies Inf \{ x <..< y\} = x$
by (*auto intro!*: $Inf\text{-}eqI$ $Sup\text{-}eqI$ *intro*: $dense\text{-}le$ $dense\text{-}le\text{-}bounded$ $dense\text{-}ge$ $dense\text{-}ge\text{-}bounded$)

61.5 Intervals of natural numbers

61.5.1 The Constant *lessThan*

lemma $lessThan\text{-}0 [simp]: lessThan (0::nat) = \{\}$
by (*simp add*: $lessThan\text{-}def$)

lemma $lessThan\text{-}Suc: lessThan (Suc k) = insert k (lessThan k)$
by (*simp add*: $lessThan\text{-}def$ $less\text{-}Suc\text{-}eq$, *blast*)

The following proof is convenient in induction proofs where new elements get indices at the beginning. So it is used to transform $\{..<Suc n\}$ to 0 and $\{..<n\}$.

lemma $zero\text{-}notin\text{-}Suc\text{-}image [simp]: 0 \notin Suc \text{ ` } A$
by *auto*

lemma $lessThan\text{-}Suc\text{-}eq\text{-}insert\text{-}0: \{..<Suc n\} = insert 0 (Suc \text{ ` } \{..<n\})$
by (*auto simp*: $image\text{-}iff$ $less\text{-}Suc\text{-}eq\text{-}0\text{-}disj$)

lemma $lessThan\text{-}Suc\text{-}atMost: lessThan (Suc k) = atMost k$
by (*simp add*: $lessThan\text{-}def$ $atMost\text{-}def$ $less\text{-}Suc\text{-}eq\text{-}le$)

lemma $atMost\text{-}Suc\text{-}eq\text{-}insert\text{-}0: \{.. Suc n\} = insert 0 (Suc \text{ ` } \{.. n\})$
unfolding $lessThan\text{-}Suc\text{-}atMost[symmetric]$ $lessThan\text{-}Suc\text{-}eq\text{-}insert\text{-}0[of\ Suc\ n]$

..

lemma *UN-lessThan-UNIV*: $(\bigcup m::nat. lessThan\ m) = UNIV$
by *blast*

61.5.2 The Constant *greaterThan*

lemma *greaterThan-0*: $greaterThan\ 0 = range\ Suc$
unfolding *greaterThan-def*
by (*blast dest: gr0-conv-Suc [THEN iffD1]*)

lemma *greaterThan-Suc*: $greaterThan\ (Suc\ k) = greaterThan\ k - \{Suc\ k\}$
unfolding *greaterThan-def*
by (*auto elim: linorder-neqE*)

lemma *INT-greaterThan-UNIV*: $(\bigcap m::nat. greaterThan\ m) = \{\}$
by *blast*

61.5.3 The Constant *atLeast*

lemma *atLeast-0 [simp]*: $atLeast\ (0::nat) = UNIV$
by (*unfold atLeast-def UNIV-def, simp*)

lemma *atLeast-Suc*: $atLeast\ (Suc\ k) = atLeast\ k - \{k\}$
unfolding *atLeast-def* **by** (*auto simp: order-le-less Suc-le-eq*)

lemma *atLeast-Suc-greaterThan*: $atLeast\ (Suc\ k) = greaterThan\ k$
by (*auto simp add: greaterThan-def atLeast-def less-Suc-eq-le*)

lemma *UN-atLeast-UNIV*: $(\bigcup m::nat. atLeast\ m) = UNIV$
by *blast*

61.5.4 The Constant *atMost*

lemma *atMost-0 [simp]*: $atMost\ (0::nat) = \{0\}$
by (*simp add: atMost-def*)

lemma *atMost-Suc*: $atMost\ (Suc\ k) = insert\ (Suc\ k)\ (atMost\ k)$
unfolding *atMost-def* **by** (*auto simp add: less-Suc-eq order-le-less*)

lemma *UN-atMost-UNIV*: $(\bigcup m::nat. atMost\ m) = UNIV$
by *blast*

61.5.5 The Constant *atLeastLessThan*

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside,

when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

lemma *atLeast0LessThan* [code-abbrev]: $\{0::nat..<n\} = \{..<n\}$
 by (*simp add: lessThan-def atLeastLessThan-def*)

lemma *atLeast0AtMost* [code-abbrev]: $\{0..n::nat\} = \{..n\}$
 by (*simp add: atMost-def atLeastAtMost-def*)

lemma *lessThan-atLeast0*: $\{..<n\} = \{0::nat..<n\}$
 by (*simp add: atLeast0LessThan*)

lemma *atMost-atLeast0*: $\{..n\} = \{0::nat..n\}$
 by (*simp add: atLeast0AtMost*)

lemma *atLeastLessThan0*: $\{m..<0::nat\} = \{\}$
 by (*simp add: atLeastLessThan-def*)

lemma *atLeast0-lessThan-Suc*: $\{0..<Suc\ n\} = insert\ n\ \{0..<n\}$
 by (*simp add: atLeast0LessThan lessThan-Suc*)

lemma *atLeast0-lessThan-Suc-eq-insert-0*: $\{0..<Suc\ n\} = insert\ 0\ (Suc\ '\{0..<n\})$
 by (*simp add: atLeast0LessThan lessThan-Suc-eq-insert-0*)

61.5.6 The Constant *atLeastAtMost*

lemma *Icc-eq-insert-lb-nat*: $m \leq n \implies \{m..n\} = insert\ m\ \{Suc\ m..n\}$
 by *auto*

lemma *atLeast0-atMost-Suc*:
 $\{0..Suc\ n\} = insert\ (Suc\ n)\ \{0..n\}$
 by (*simp add: atLeast0AtMost atMost-Suc*)

lemma *atLeast0-atMost-Suc-eq-insert-0*:
 $\{0..Suc\ n\} = insert\ 0\ (Suc\ '\{0..n\})$
 by (*simp add: atLeast0AtMost atMost-Suc-eq-insert-0*)

61.5.7 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
 $\{m..<Suc\ n\} = (if\ m \leq n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$
 by (*auto simp add: atLeastLessThan-def*)

lemma *atLeastLessThan-singleton* [simp]: $\{m..<Suc\ m\} = \{m\}$
 by (*auto simp add: atLeastLessThan-def*)

lemma *atLeastLessThanSuc-atLeastAtMost*: $\{l..<Suc\ u\} = \{l..u\}$
 by (*simp add: lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def*)

lemma *atLeastSucAtMost-greaterThanAtMost*: $\{Suc\ l..u\} = \{l<..u\}$
by (*simp add: atLeast-Suc-greaterThan atLeastAtMost-def greaterThanAtMost-def*)

lemma *atLeastSucLessThan-greaterThanLessThan*: $\{Suc\ l.<u\} = \{l<..<u\}$
by (*simp add: atLeast-Suc-greaterThan atLeastLessThan-def greaterThanLessThan-def*)

lemma *atLeastAtMostSuc-conv*: $m \leq Suc\ n \implies \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
by *auto*

lemma *atLeastAtMost-insertL*: $m \leq n \implies insert\ m\ \{Suc\ m..n\} = \{m\ ..n\}$
by *auto*

The analogous result is useful on *int*:

lemma *atLeastAtMostPlus1-int-conv*:
 $m \leq 1+n \implies \{m..1+n\} = insert\ (1+n)\ \{m..n::int\}$
by (*auto intro: set-eqI*)

lemma *atLeastLessThan-add-Un*: $i \leq j \implies \{i..<j+k\} = \{i..<j\} \cup \{j..<j+k::nat\}$
by (*induct k (simp-all add: atLeastLessThanSuc)*)

61.5.8 Intervals and numerals

lemma *lessThan-nat-numeral*: — Evaluation for specific numerals
 $lessThan\ (numeral\ k :: nat) = insert\ (pred-numeral\ k)\ (lessThan\ (pred-numeral\ k))$
by (*simp add: numeral-eq-Suc lessThan-Suc*)

lemma *atMost-nat-numeral*: — Evaluation for specific numerals
 $atMost\ (numeral\ k :: nat) = insert\ (numeral\ k)\ (atMost\ (pred-numeral\ k))$
by (*simp add: numeral-eq-Suc atMost-Suc*)

lemma *atLeastLessThan-nat-numeral*: — Evaluation for specific numerals
 $atLeastLessThan\ m\ (numeral\ k :: nat) =$
 (*if* $m \leq (pred-numeral\ k)$ *then* $insert\ (pred-numeral\ k)\ (atLeastLessThan\ m\ (pred-numeral\ k))$
 else $\{\}$)
by (*simp add: numeral-eq-Suc atLeastLessThanSuc*)

61.5.9 Image

context *linordered-semidom*
begin

lemma *image-add-atLeast[simp]*: $plus\ k\ ' \{i..\} = \{k + i..\}$

proof —

have $n = k + (n - k)$ **if** $i + k \leq n$ **for** n

```

proof –
  have  $n = (n - (k + i)) + (k + i)$  using that
    by (metis add-commute le-add-diff-inverse)
  then show  $n = k + (n - k)$ 
    by (metis local.add-diff-cancel-left' add-assoc add-commute)
qed
then show ?thesis
  by (fastforce simp: add-le-imp-le-diff add.commute)
qed

```

```

lemma image-add-atLeastAtMost [simp]:
  plus k ‘ {i..j} = {i + k..j + k} (is ?A = ?B)
proof
  show  $?A \subseteq ?B$ 
    by (auto simp add: ac-simps)
next
  show  $?B \subseteq ?A$ 
    proof
      fix  $n$ 
      assume  $n \in ?B$ 
      then have  $i \leq n - k$ 
        by (simp add: add-le-imp-le-diff)
      have  $n = n - k + k$ 
      proof –
        from  $\langle n \in ?B \rangle$  have  $n = n - (i + k) + (i + k)$ 
          by simp
        also have  $\dots = n - k - i + i + k$ 
          by (simp add: algebra-simps)
        also have  $\dots = n - k + k$ 
          using  $\langle i \leq n - k \rangle$  by simp
        finally show ?thesis .
      qed
      moreover have  $n - k \in \{i..j\}$ 
        using  $\langle n \in ?B \rangle$ 
        by (auto simp: add-le-imp-le-diff add-le-add-imp-diff-le)
      ultimately show  $n \in ?A$ 
        by (simp add: ac-simps)
    qed
qed

```

```

lemma image-add-atLeastAtMost' [simp]:
   $(\lambda n. n + k) ‘ \{i..j\} = \{i + k..j + k\}$ 
  by (simp add: add.commute [of - k])

```

```

lemma image-add-atLeastLessThan [simp]:
  plus k ‘ {i.<j} = {i + k.<j + k}
  by (simp add: image-set-diff atLeastLessThan-eq-atLeastAtMost-diff ac-simps)

```

```

lemma image-add-atLeastLessThan' [simp]:

```

$(\lambda n. n + k) \text{ ' } \{i..<j\} = \{i + k..<j + k\}$
by (*simp add: add commute [of - k]*)

lemma *image-add-greaterThanAtMost[simp]*: $(+) c \text{ ' } \{a<..b\} = \{c + a<..c + b\}$
by (*simp add: image-set-diff greaterThanAtMost-eq-atLeastAtMost-diff ac-simps*)

end

context *ordered-ab-group-add*
begin

lemma

fixes $x :: 'a$
shows *image-uminus-greaterThan[simp]*: $uminus \text{ ' } \{x<.. \} = \{..<-x\}$
and *image-uminus-atLeast[simp]*: $uminus \text{ ' } \{x.. \} = \{..-x\}$

proof *safe*

fix y **assume** $y < -x$
hence $*$: $x < -y$ **using** *neg-less-iff-less[of -y x]* **by** *simp*
have $-(-y) \in uminus \text{ ' } \{x<.. \}$
by (*rule imageI*) (*simp add: **)
thus $y \in uminus \text{ ' } \{x<.. \}$ **by** *simp*

next

fix y **assume** $y \leq -x$
have $-(-y) \in uminus \text{ ' } \{x.. \}$
by (*rule imageI*) (*use* $\langle y \leq -x \rangle$ [*THEN le-imp-neg-le*] **in** $\langle simp \rangle$)
thus $y \in uminus \text{ ' } \{x.. \}$ **by** *simp*

qed *simp-all*

lemma

fixes $x :: 'a$
shows *image-uminus-lessThan[simp]*: $uminus \text{ ' } \{..<x\} = \{-x<.. \}$
and *image-uminus-atMost[simp]*: $uminus \text{ ' } \{..x\} = \{-x.. \}$

proof $-$

have $uminus \text{ ' } \{..<x\} = uminus \text{ ' } uminus \text{ ' } \{-x<.. \}$
and $uminus \text{ ' } \{..x\} = uminus \text{ ' } uminus \text{ ' } \{-x.. \}$ **by** *simp-all*
thus $uminus \text{ ' } \{..<x\} = \{-x<.. \}$ **and** $uminus \text{ ' } \{..x\} = \{-x.. \}$
by (*simp-all add: image-image*
del: image-uminus-greaterThan image-uminus-atLeast)

qed

lemma

fixes $x :: 'a$
shows *image-uminus-atLeastAtMost[simp]*: $uminus \text{ ' } \{x..y\} = \{-y..-x\}$
and *image-uminus-greaterThanAtMost[simp]*: $uminus \text{ ' } \{x<..y\} = \{-y..<-x\}$
and *image-uminus-atLeastLessThan[simp]*: $uminus \text{ ' } \{x..<y\} = \{-y<..-x\}$
and *image-uminus-greaterThanLessThan[simp]*: $uminus \text{ ' } \{x<..<y\} = \{-y<..<-x\}$
by (*simp-all add: atLeastAtMost-def greaterThanAtMost-def atLeastLessThan-def*
greaterThanLessThan-def image-Int[OF inj-uminus] Int-commute)

lemma *image-add-atMost*[simp]: $(+) c \text{ ‘ } \{..a\} = \{..c + a\}$
 by (*auto intro!*: *image-eqI*[**where** $x=x - c$ **for** x] *simp*: *algebra-simps*)

end

lemma *image-Suc-atLeastAtMost* [simp]:
 $Suc \text{ ‘ } \{i..j\} = \{Suc \ i..Suc \ j\}$
 using *image-add-atLeastAtMost* [of 1 $i \ j$]
 by (*simp only*: *plus-1-eq-Suc*) *simp*

lemma *image-Suc-atLeastLessThan* [simp]:
 $Suc \text{ ‘ } \{i..<j\} = \{Suc \ i..<Suc \ j\}$
 using *image-add-atLeastLessThan* [of 1 $i \ j$]
 by (*simp only*: *plus-1-eq-Suc*) *simp*

corollary *image-Suc-atMost*:
 $Suc \text{ ‘ } \{..n\} = \{1..Suc \ n\}$
 by (*simp add*: *atMost-atLeast0 atLeastLessThanSuc-atLeastAtMost*)

corollary *image-Suc-lessThan*:
 $Suc \text{ ‘ } \{..<n\} = \{1..n\}$
 by (*simp add*: *lessThan-atLeast0 atLeastLessThanSuc-atLeastAtMost*)

lemma *image-diff-atLeastAtMost* [simp]:
 fixes $d::'a::linordered-idom$ **shows** $(-) d \text{ ‘ } \{a..b\} = \{d-b..d-a\}$
proof
 show $\{d - b..d - a\} \subseteq (-) d \text{ ‘ } \{a..b\}$
proof
 fix x
 assume $x \in \{d - b..d - a\}$
 then have $d - x \in \{a..b\}$ and $x = d - (d - x)$
 by *auto*
 then show $x \in (-) d \text{ ‘ } \{a..b\}$
 by (*rule rev-image-eqI*)
qed
qed(*auto*)

lemma *image-diff-atLeastLessThan* [simp]:
 fixes $a \ b \ c::'a::linordered-idom$
 shows $(-) c \text{ ‘ } \{a..<b\} = \{c - b<..c - a\}$
proof –
 have $(-) c \text{ ‘ } \{a..<b\} = (+) c \text{ ‘ } uminus \text{ ‘ } \{a \ ..<b\}$
 unfolding *image-image* **by** *simp*
 also have $\dots = \{c - b<..c - a\}$ **by** *simp*
 finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-greaterThanAtMost*[simp]:
 fixes $a \ b \ c::'a::linordered-idom$

shows $(-) c \text{ ‘ } \{a <..b\} = \{c - b..<c - a\}$
proof –
 have $(-) c \text{ ‘ } \{a <..b\} = (+) c \text{ ‘ } \textit{uminus} \text{ ‘ } \{a <..b\}$
 unfolding *image-image* **by** *simp*
 also have $\dots = \{c - b..<c - a\}$ **by** *simp*
 finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-atLeast*[*simp*]:
 fixes $a c :: 'a :: \textit{linordered-idom}$
 shows $(-) c \text{ ‘ } \{a.. \} = \{..c - a\}$
proof –
 have $(-) c \text{ ‘ } \{a.. \} = (+) c \text{ ‘ } \textit{uminus} \text{ ‘ } \{a .. \}$
 unfolding *image-image* **by** *simp*
 also have $\dots = \{..c - a\}$ **by** *simp*
 finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-AtMost*[*simp*]:
 fixes $b c :: 'a :: \textit{linordered-idom}$
 shows $(-) c \text{ ‘ } \{..b\} = \{c - b.. \}$
proof –
 have $(-) c \text{ ‘ } \{..b\} = (+) c \text{ ‘ } \textit{uminus} \text{ ‘ } \{..b\}$
 unfolding *image-image* **by** *simp*
 also have $\dots = \{c - b.. \}$ **by** *simp*
 finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-atLeastAtMost'* [*simp*]:
 $(\lambda t. t-d) \text{ ‘ } \{a..b\} = \{a-d..b-d\}$ **for** $d :: 'a :: \textit{linordered-idom}$
by (*metis* (*no-types*, *lifting*) *diff-conv-add-uminus image-add-atLeastAtMost' image-cong*)

context *linordered-field*
begin

lemma *image-mult-atLeastAtMost* [*simp*]:
 $((*) d \text{ ‘ } \{a..b\}) = \{d*a..d*b\}$ **if** $d > 0$
using *that*
by (*auto simp: field-simps mult-le-cancel-right intro: rev-image-eqI* [**where** $x = x/d$
for x])

lemma *image-divide-atLeastAtMost* [*simp*]:
 $((\lambda c. c / d) \text{ ‘ } \{a..b\}) = \{a/d..b/d\}$ **if** $d > 0$
proof –
from *that* **have** *inverse* $d > 0$
 by *simp*
with *image-mult-atLeastAtMost* [*of inverse d a b*]
have $((*) (\textit{inverse} d) \text{ ‘ } \{a..b\}) = \{\textit{inverse} d * a.. \textit{inverse} d * b\}$

by *blast*
moreover have (*) (*inverse d*) = ($\lambda c. c / d$)
 by (*simp add: fun-eq-iff field-simps*)
ultimately show ?*thesis*
 by *simp*
qed

lemma *image-mult-atLeastAtMost-if*:

(*) $c \cdot \{x .. y\} =$
 (*if* $c > 0$ *then* $\{c * x .. c * y\}$ *else if* $x \leq y$ *then* $\{c * y .. c * x\}$ *else* $\{\}$)
proof (*cases c = 0 \vee x > y*)
 case *True*
 then **show** ?*thesis*
 by *auto*
next
 case *False*
 then **have** $x \leq y$
 by *auto*
from *False* **consider** $c < 0 \mid c > 0$
 by (*auto simp add: neq-iff*)
 then **show** ?*thesis*
proof *cases*
 case 1
 have (*) $c \cdot \{x..y\} = \{c * y..c * x\}$
proof (*rule set-eqI*)
 fix *d*
from 1 **have** *inj* ($\lambda z. z / c$)
 by (*auto intro: injI*)
 then **have** $d \in (*) c \cdot \{x..y\} \longleftrightarrow d / c \in (\lambda z. z \text{ div } c) \cdot (*) c \cdot \{x..y\}$
 by (*subst inj-image-mem-iff*) *simp-all*
 also **have** $\dots \longleftrightarrow d / c \in \{x..y\}$
 using 1 **by** (*simp add: image-image*)
 also **have** $\dots \longleftrightarrow d \in \{c * y..c * x\}$
 by (*auto simp add: field-simps 1*)
 finally **show** $d \in (*) c \cdot \{x..y\} \longleftrightarrow d \in \{c * y..c * x\}$.
qed
with $\langle x \leq y \rangle$ **show** ?*thesis*
 by *auto*
qed (*simp add: mult-left-mono-neg*)
qed

lemma *image-mult-atLeastAtMost-if'*:

($\lambda x. x * c$) $\cdot \{x..y\} =$
 (*if* $x \leq y$ *then if* $c > 0$ *then* $\{x * c .. y * c\}$ *else* $\{y * c .. x * c\}$ *else* $\{\}$)
using *image-mult-atLeastAtMost-if* [*of c x y*] **by** (*auto simp add: ac-simps*)

lemma *image-affinity-atLeastAtMost*:

(($\lambda x. m * x + c$) $\cdot \{a..b\}$) = (*if* $\{a..b\} = \{\}$ *then* $\{\}$
else if $0 \leq m$ *then* $\{m * a + c .. m * b + c\}$)

else $\{m * b + c .. m * a + c\}$)

proof –

have *: $(\lambda x. m * x + c) = ((\lambda x. x + c) \circ (*) m)$

by (*simp add: fun-eq-iff*)

show ?thesis **by** (*simp only: * image-comp [symmetric] image-mult-atLeastAtMost-if*)
(*auto simp add: mult-le-cancel-left*)

qed

lemma *image-affinity-atLeastAtMost-diff*:

$((\lambda x. m*x - c) ' \{a..b\}) = (if \{a..b\}=\{\} then \{\}$
else if $0 \leq m$ then $\{m*a - c .. m*b - c\}$
else $\{m*b - c .. m*a - c\}$)

using *image-affinity-atLeastAtMost [of m -c a b]*

by *simp*

lemma *image-affinity-atLeastAtMost-div*:

$((\lambda x. x/m + c) ' \{a..b\}) = (if \{a..b\}=\{\} then \{\}$
else if $0 \leq m$ then $\{a/m + c .. b/m + c\}$
else $\{b/m + c .. a/m + c\}$)

using *image-affinity-atLeastAtMost [of inverse m c a b]*

by (*simp add: field-class.field-divide-inverse algebra-simps inverse-eq-divide*)

lemma *image-affinity-atLeastAtMost-div-diff*:

$((\lambda x. x/m - c) ' \{a..b\}) = (if \{a..b\}=\{\} then \{\}$
else if $0 \leq m$ then $\{a/m - c .. b/m - c\}$
else $\{b/m - c .. a/m - c\}$)

using *image-affinity-atLeastAtMost-diff [of inverse m c a b]*

by (*simp add: field-class.field-divide-inverse algebra-simps inverse-eq-divide*)

end

lemma *atLeast1-lessThan-eq-remove0*:

$\{Suc\ 0..<n\} = \{..<n\} - \{0\}$

by *auto*

lemma *atLeast1-atMost-eq-remove0*:

$\{Suc\ 0..n\} = \{..n\} - \{0\}$

by *auto*

lemma *image-add-int-atLeastLessThan*:

$(\lambda x. x + (l::int)) ' \{0..<u-l\} = \{l..<u\}$

by *safe auto*

lemma *image-minus-const-atLeastLessThan-nat*:

fixes $c :: nat$

shows $(\lambda i. i - c) ' \{x ..< y\} =$

$(if\ c < y\ then\ \{x - c ..< y - c\}\ else\ if\ x < y\ then\ \{0\}\ else\ \{\})$

$(is\ - = ?right)$

proof *safe*

```

fix a assume a: a ∈ ?right
show a ∈ (λi. i - c) ‘ {x ..< y}
proof cases
  assume c < y with a show ?thesis
  by (auto intro!: image-eqI[of - - a + c])
next
  assume ¬ c < y with a show ?thesis
  by (auto intro!: image-eqI[of - - x] split: if-split-asm)
qed
qed auto

```

```

lemma image-int-atLeastLessThan:
  int ‘ {a..<b} = {int a..<int b}
  by (auto intro!: image-eqI [where x = nat x for x])

```

```

lemma image-int-atLeastAtMost:
  int ‘ {a..b} = {int a..int b}
  by (auto intro!: image-eqI [where x = nat x for x])

```

61.5.10 Finiteness

```

lemma finite-lessThan [iff]: fixes k :: nat shows finite {..<k}
  by (induct k) (simp-all add: lessThan-Suc)

```

```

lemma finite-atMost [iff]: fixes k :: nat shows finite {..k}
  by (induct k) (simp-all add: atMost-Suc)

```

```

lemma finite-greaterThanLessThan [iff]:
  fixes l :: nat shows finite {l<..}
  by (simp add: greaterThanLessThan-def)

```

```

lemma finite-atLeastLessThan [iff]:
  fixes l :: nat shows finite {l..}
  by (simp add: atLeastLessThan-def)

```

```

lemma finite-greaterThanAtMost [iff]:
  fixes l :: nat shows finite {l<..}
  by (simp add: greaterThanAtMost-def)

```

```

lemma finite-atLeastAtMost [iff]:
  fixes l :: nat shows finite {l..}
  by (simp add: atLeastAtMost-def)

```

A bounded set of natural numbers is finite.

```

lemma bounded-nat-set-is-finite: (∀ i ∈ N. i < (n::nat)) ⇒ finite N
  by (rule finite-subset [OF - finite-lessThan]) auto

```

A set of natural numbers is finite iff it is bounded.

```

lemma finite-nat-set-iff-bounded:

```

```

finite(N::nat set) = (∃ m. ∀ n∈N. n < m) (is ?F = ?B)
proof
  assume f: ?F show ?B
    using Max-ge[OF ‹?F›, simplified less-Suc-eq-le[symmetric]] by blast
next
  assume ?B show ?F using ‹?B› by (blast intro: bounded-nat-set-is-finite)
qed

```

```

lemma finite-nat-set-iff-bounded-le: finite(N::nat set) = (∃ m. ∀ n∈N. n ≤ m)
  unfolding finite-nat-set-iff-bounded
  by (blast dest: less-imp-le-nat le-imp-less-Suc)

```

```

lemma finite-less-ub:
  ∧ f::nat ⇒ nat. (!n. n ≤ f n) ⇒ finite {n. f n ≤ u}
  by (rule finite-subset[of - {..u}])
  (auto intro: order-trans)

```

```

lemma bounded-Max-nat:
  fixes P :: nat ⇒ bool
  assumes x: P x and M: ∧ x. P x ⇒ x ≤ M
  obtains m where P m ∧ x. P x ⇒ x ≤ m
proof –
  have finite {x. P x}
    using M finite-nat-set-iff-bounded-le by auto
  then have Max {x. P x} ∈ {x. P x}
    using Max-in x by auto
  then show ?thesis
    by (simp add: ‹finite {x. P x}› that)
qed

```

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

```

lemma subset-card-intvl-is-intvl:
  assumes A ⊆ {k..<k + card A}
  shows A = {k..<k + card A}
proof (cases finite A)
  case True
  from this and assms show ?thesis
  proof (induct A rule: finite-linorder-max-induct)
    case empty thus ?case by auto
  next
    case (insert b A)
    hence *: b ∉ A by auto
    with insert have A ≤ {k..<k + card A} and b = k + card A
      by fastforce+
    with insert * show ?case by auto
  qed
next
  case False

```

with *assms* **show** *?thesis* **by** *simp*
qed

61.5.11 Proving Inclusions and Equalities between Unions

lemma *UN-le-eq-Un0*:

$(\bigcup_{i \leq n :: \text{nat.}} M\ i) = (\bigcup_{i \in \{1..n\}} M\ i) \cup M\ 0$ (**is** *?A = ?B*)

proof

show *?A* \subseteq *?B*

proof

fix *x* **assume** $x \in ?A$

then obtain *i* **where** $i \leq n$ $x \in M\ i$ **by** *auto*

show $x \in ?B$

proof(*cases i*)

case *0* **with** *i* **show** *?thesis* **by** *simp*

next

case (*Suc j*) **with** *i* **show** *?thesis* **by** *auto*

qed

qed

next

show *?B* \subseteq *?A* **by** *fastforce*

qed

lemma *UN-le-add-shift*:

$(\bigcup_{i \leq n :: \text{nat.}} M(i+k)) = (\bigcup_{i \in \{k..n+k\}} M\ i)$ (**is** *?A = ?B*)

proof

show *?A* \subseteq *?B* **by** *fastforce*

next

show *?B* \subseteq *?A*

proof

fix *x* **assume** $x \in ?B$

then obtain *i* **where** $i \in \{k..n+k\}$ $x \in M(i)$ **by** *auto*

hence $i-k \leq n$ \wedge $x \in M((i-k)+k)$ **by** *auto*

thus $x \in ?A$ **by** *blast*

qed

qed

lemma *UN-le-add-shift-strict*:

$(\bigcup_{i < n :: \text{nat.}} M(i+k)) = (\bigcup_{i \in \{k..<n+k\}} M\ i)$ (**is** *?A = ?B*)

proof

show *?B* \subseteq *?A*

proof

fix *x* **assume** $x \in ?B$

then obtain *i* **where** $i \in \{k..<n+k\}$ $x \in M(i)$ **by** *auto*

then have $i-k < n$ \wedge $x \in M((i-k)+k)$ **by** *auto*

then show $x \in ?A$ **using** *UN-le-add-shift* **by** *blast*

qed

qed (*fastforce*)

lemma *UN-UN-finite-eq*: $(\bigcup n::nat. \bigcup i \in \{0..<n\}. A\ i) = (\bigcup n. A\ n)$
by (*auto simp add: atLeast0LessThan*)

lemma *UN-finite-subset*:
 $(\bigwedge n::nat. (\bigcup i \in \{0..<n\}. A\ i) \subseteq C) \implies (\bigcup n. A\ n) \subseteq C$
by (*subst UN-UN-finite-eq [symmetric] blast*)

lemma *UN-finite2-subset*:
assumes $\bigwedge n::nat. (\bigcup i \in \{0..<n\}. A\ i) \subseteq (\bigcup i \in \{0..<n+k\}. B\ i)$
shows $(\bigcup n. A\ n) \subseteq (\bigcup n. B\ n)$
proof (*rule UN-finite-subset, rule subsetI*)
fix n **and** a
from *assms* **have** $(\bigcup i \in \{0..<n\}. A\ i) \subseteq (\bigcup i \in \{0..<n+k\}. B\ i)$.
moreover **assume** $a \in (\bigcup i \in \{0..<n\}. A\ i)$
ultimately **have** $a \in (\bigcup i \in \{0..<n+k\}. B\ i)$ **by** *blast*
then **show** $a \in (\bigcup i. B\ i)$ **by** (*auto simp add: UN-UN-finite-eq*)
qed

lemma *UN-finite2-eq*:
assumes $(\bigwedge n::nat. (\bigcup i \in \{0..<n\}. A\ i) = (\bigcup i \in \{0..<n+k\}. B\ i))$
shows $(\bigcup n. A\ n) = (\bigcup n. B\ n)$
proof (*rule subset-antisym [OF UN-finite-subset UN-finite2-subset]*)
fix n
show $(\bigcup i \in \{0..<n\}. A\ i) \subseteq (\bigcup n. B\ n)$
using *assms* **by** *auto*
next
fix n
show $(\bigcup i \in \{0..<n\}. B\ i) \subseteq (\bigcup i \in \{0..<n+k\}. A\ i)$
using *assms* **by** (*force simp add: atLeastLessThan-add-Un [of 0]*)
qed

61.5.12 Cardinality

lemma *card-lessThan [simp]*: $\text{card } \{..<u\} = u$
by (*induct u, simp-all add: lessThan-Suc*)

lemma *card-atMost [simp]*: $\text{card } \{..u\} = \text{Suc } u$
by (*simp add: lessThan-Suc-atMost [THEN sym]*)

lemma *card-atLeastLessThan [simp]*: $\text{card } \{l..<u\} = u - l$

proof –
have $(\lambda x. x + l) ' \{..<u - l\} \subseteq \{l..<u\}$
by *auto*
moreover **have** $\{l..<u\} \subseteq (\lambda x. x + l) ' \{..<u-l\}$
proof
fix x
assume $*$: $x \in \{l..<u\}$
then **have** $x - l \in \{..<u - l\}$
by *auto*

```

    then have  $(x - l) + l \in (\lambda x. x + l) \cdot \{..< u - l\}$ 
      by auto
    then show  $x \in (\lambda x. x + l) \cdot \{..< u - l\}$ 
      using * by auto
  qed
  ultimately have  $\{l..<u\} = (\lambda x. x + l) \cdot \{..<u-l\}$ 
    by auto
  then have  $\text{card } \{l..<u\} = \text{card } \{..<u-l\}$ 
    by (simp add: card-image inj-on-def)
  then show ?thesis
    by simp
qed

lemma card-atLeastAtMost [simp]:  $\text{card } \{l..u\} = \text{Suc } u - l$ 
  by (subst atLeastLessThanSuc-atLeastAtMost [THEN sym], simp)

lemma card-greaterThanAtMost [simp]:  $\text{card } \{l<..u\} = u - l$ 
  by (subst atLeastSucAtMost-greaterThanAtMost [THEN sym], simp)

lemma card-greaterThanLessThan [simp]:  $\text{card } \{l<..<u\} = u - \text{Suc } l$ 
  by (subst atLeastSucLessThan-greaterThanLessThan [THEN sym], simp)

lemma subset-eq-atLeast0-lessThan-finite:
  fixes  $n :: \text{nat}$ 
  assumes  $N \subseteq \{0..<n\}$ 
  shows finite  $N$ 
  using assms finite-atLeastLessThan by (rule finite-subset)

lemma subset-eq-atLeast0-atMost-finite:
  fixes  $n :: \text{nat}$ 
  assumes  $N \subseteq \{0..n\}$ 
  shows finite  $N$ 
  using assms finite-atLeastAtMost by (rule finite-subset)

lemma ex-bij-betw-nat-finite:
  finite  $M \implies \exists h. \text{bij-betw } h \{0..<\text{card } M\} M$ 
  apply (drule finite-imp-nat-seg-image-inj-on)
  apply (auto simp: atLeast0LessThan[symmetric] lessThan-def[symmetric] card-image
    bij-betw-def)
  done

lemma ex-bij-betw-finite-nat:
  finite  $M \implies \exists h. \text{bij-betw } h M \{0..<\text{card } M\}$ 
  by (blast dest: ex-bij-betw-nat-finite bij-betw-inv)

lemma finite-same-card-bij:
  finite  $A \implies \text{finite } B \implies \text{card } A = \text{card } B \implies \exists h. \text{bij-betw } h A B$ 
  apply (drule ex-bij-betw-finite-nat)
  apply (drule ex-bij-betw-nat-finite)

```

```

apply(auto intro!:bij-betw-trans)
done

```

```

lemma ex-bij-betw-nat-finite-1:
  finite M  $\implies \exists h. \text{bij-betw } h \{1 \dots \text{card } M\} M$ 
  by (rule finite-same-card-bij) auto

```

```

lemma bij-betw-iff-card:
  assumes finite A finite B
  shows  $(\exists f. \text{bij-betw } f A B) \longleftrightarrow (\text{card } A = \text{card } B)$ 
proof
  assume card A = card B
  moreover obtain f where bij-betw f A {0 ..< card A}
    using assms ex-bij-betw-finite-nat by blast
  moreover obtain g where bij-betw g {0 ..< card B} B
    using assms ex-bij-betw-nat-finite by blast
  ultimately have bij-betw (g o f) A B
    by (auto simp: bij-betw-trans)
  thus  $(\exists f. \text{bij-betw } f A B)$  by blast
qed (auto simp: bij-betw-same-card)

```

```

lemma subset-eq-atLeast0-lessThan-card:
  fixes n :: nat
  assumes  $N \subseteq \{0..<n\}$ 
  shows card N  $\leq n$ 
proof –
  from assms finite-lessThan have card N  $\leq \text{card } \{0..<n\}$ 
    using card-mono by blast
  then show ?thesis by simp
qed

```

Relational version of *card-inj-on-le*:

```

lemma card-le-if-inj-on-rel:
assumes finite B
   $\bigwedge a. a \in A \implies \exists b. b \in B \wedge r a b$ 
   $\bigwedge a1 a2 b. [a1 \in A; a2 \in A; b \in B; r a1 b; r a2 b] \implies a1 = a2$ 
shows card A  $\leq \text{card } B$ 
proof –
  let ?P =  $\lambda a b. b \in B \wedge r a b$ 
  let ?f =  $\lambda a. \text{SOME } b. ?P a b$ 
  have 1: ?f ‘  $A \subseteq B$  by (auto intro: someI2-ex[OF assms(2)])
  have inj-on ?f A
    unfolding inj-on-def
  proof safe
    fix a1 a2 assume asms: a1  $\in A$  a2  $\in A$  ?f a1 = ?f a2
    have 0: ?f a1  $\in B$  using 1  $\langle a1 \in A \rangle$  by blast
    have 1: r a1 (?f a1) using someI-ex[OF assms(2)][OF  $\langle a1 \in A \rangle$  ] by blast
    have 2: r a2 (?f a1) using someI-ex[OF assms(2)][OF  $\langle a2 \in A \rangle$  ] asms(3) by
auto

```


show $a1 = a2$ using $assms(3)[OF\ asms(1,2)\ 0\ 1\ 2]$.
 qed
 with 1 show *?thesis* using *card-inj-on-le*[of *?f A B*] *assms(1)* by *simp*
 qed

lemma *inj-on-funpow-least*:

$\langle inj\ on\ (\lambda k. (f \wedge k) s) \{0..<n\} \rangle$

if $\langle (f \wedge n) s = s \rangle \langle \bigwedge m. 0 < m \implies m < n \implies (f \wedge m) s \neq s \rangle$

proof –

{ **fix** $k\ l$ **assume** $A: k < n\ l < n\ k \neq l\ (f \wedge k) s = (f \wedge l) s$

define $k'\ l'$ **where** $k' = \min\ k\ l$ **and** $l' = \max\ k\ l$

with A **have** $A': k' < l'\ (f \wedge k') s = (f \wedge l') s\ l' < n$

by (*auto simp: min-def max-def*)

have $s = (f \wedge ((n - l') + l')) s$ **using** *that* $\langle l' < n \rangle$ **by** *simp*

also have $\dots = (f \wedge (n - l')) ((f \wedge l') s)$ **by** (*simp add: funpow-add*)

also have $(f \wedge l') s = (f \wedge k') s$ **by** (*simp add: A'*)

also have $(f \wedge (n - l')) \dots = (f \wedge (n - l' + k')) s$ **by** (*simp add: funpow-add*)

finally have $(f \wedge (n - l' + k')) s = s$ **by** *simp*

moreover have $n - l' + k' < n\ 0 < n - l' + k'$ **using** A' **by** *linarith+*

ultimately have *False* **using** *that(2)* **by** *auto*

}

then show *?thesis* **by** (*intro inj-onI auto*)

qed

61.6 Intervals of integers

lemma *atLeastLessThanPlusOne-atLeastAtMost-int*: $\{l..<u+1\} = \{l..(u::int)\}$

by (*auto simp add: atLeastAtMost-def atLeastLessThan-def*)

lemma *atLeastPlusOneAtMost-greaterThanAtMost-int*: $\{l+1..u\} = \{l<..(u::int)\}$

by (*auto simp add: atLeastAtMost-def greaterThanAtMost-def*)

lemma *atLeastPlusOneLessThan-greaterThanLessThan-int*:

$\{l+1..<u\} = \{l<..<u::int\}$

by (*auto simp add: atLeastLessThan-def greaterThanLessThan-def*)

61.6.1 Finiteness

lemma *image-atLeastZeroLessThan-int*:

assumes $0 \leq u$

shows $\{(0::int)..<u\} = \text{int } \{..<\text{nat } u\}$

unfolding *image-def lessThan-def*

proof

show $\{0..<u\} \subseteq \{y. \exists x \in \{x. x < \text{nat } u\}. y = \text{int } x\}$

proof

fix x

assume $x \in \{0..<u\}$

then have $x = \text{int } (\text{nat } x)$ **and** $\text{nat } x < \text{nat } u$

by (*auto simp add: zless-nat-eq-int-zless [THEN sym]*)

```

then have  $\exists xa < \text{nat } u. x = \text{int } xa$ 
  using exI[of - (nat x)] by simp
then show  $x \in \{y. \exists x \in \{x. x < \text{nat } u\}. y = \text{int } x\}$ 
  by simp
qed
qed (auto)

```

lemma *finite-atLeastZeroLessThan-int*: *finite* $\{(0::\text{int})..<u\}$

```

proof (cases  $0 \leq u$ )
  case True
  then show ?thesis
    by (auto simp: image-atLeastZeroLessThan-int)
qed auto

```

lemma *finite-atLeastLessThan-int* [*iff*]: *finite* $\{l..<u::\text{int}\}$

```

by (simp only: image-add-int-atLeastLessThan [symmetric, of l] finite-imageI
finite-atLeastZeroLessThan-int)

```

lemma *finite-atLeastAtMost-int* [*iff*]: *finite* $\{l..(u::\text{int})\}$

```

by (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym], simp)

```

lemma *finite-greaterThanAtMost-int* [*iff*]: *finite* $\{l<..(u::\text{int})\}$

```

by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

```

lemma *finite-greaterThanLessThan-int* [*iff*]: *finite* $\{l<..<u::\text{int}\}$

```

by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

```

61.6.2 Cardinality

lemma *card-atLeastZeroLessThan-int*: *card* $\{(0::\text{int})..<u\} = \text{nat } u$

```

proof (cases  $0 \leq u$ )
  case True
  then show ?thesis
    by (auto simp: image-atLeastZeroLessThan-int card-image inj-on-def)
qed auto

```

lemma *card-atLeastLessThan-int* [*simp*]: *card* $\{l..<u\} = \text{nat } (u - l)$

```

proof –
  have card  $\{l..<u\} = \text{card } \{0..<u-l\}$ 
    apply (subst image-add-int-atLeastLessThan [symmetric])
    apply (rule card-image)
    apply (simp add: inj-on-def)
    done
  then show ?thesis
    by (simp add: card-atLeastZeroLessThan-int)
qed

```

lemma *card-atLeastAtMost-int* [*simp*]: *card* $\{l..u\} = \text{nat } (u - l + 1)$

```

apply (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym])
apply (auto simp add: algebra-simps)
done

```

```

lemma card-greaterThanAtMost-int [simp]: card {l<..} = nat (u - l)
by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

```

```

lemma card-greaterThanLessThan-int [simp]: card {l<..} = nat (u - (l + 1))
by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

```

```

lemma finite-M-bounded-by-nat: finite {k. P k ∧ k < (i::nat)}
proof -
  have {k. P k ∧ k < i} ⊆ {..} by auto
  with finite-lessThan[of i] show ?thesis by (simp add: finite-subset)
qed

```

```

lemma card-less:
  assumes zero-in-M: 0 ∈ M
  shows card {k ∈ M. k < Suc i} ≠ 0
proof -
  from zero-in-M have {k ∈ M. k < Suc i} ≠ {} by auto
  with finite-M-bounded-by-nat show ?thesis by (auto simp add: card-eq-0-iff)
qed

```

```

lemma card-less-Suc2:
  assumes 0 ∉ M shows card {k. Suc k ∈ M ∧ k < i} = card {k ∈ M. k < Suc i}
proof -
  have *: [j ∈ M; j < Suc i] ⇒ j - Suc 0 < i ∧ Suc (j - Suc 0) ∈ M ∧ Suc 0
  ≤ j for j
  by (cases j) (use assms in auto)
  show ?thesis
  proof (rule card-bij-eq)
    show inj-on Suc {k. Suc k ∈ M ∧ k < i}
    by force
    show inj-on (λx. x - Suc 0) {k ∈ M. k < Suc i}
    by (rule inj-on-diff-nat) (use * in blast)
  qed (use * in auto)
qed

```

```

lemma card-less-Suc:
  assumes 0 ∈ M
  shows Suc (card {k. Suc k ∈ M ∧ k < i}) = card {k ∈ M. k < Suc i}
proof -
  have Suc (card {k. Suc k ∈ M ∧ k < i}) = Suc (card {k. Suc k ∈ M - {0} ∧
  k < i})
  by simp
  also have ... = Suc (card {k ∈ M - {0}. k < Suc i})
  apply (subst card-less-Suc2)

```

```

using assms by auto
also have ... = Suc (card ({k ∈ M. k < Suc i} - {0}))
  by (force intro: arg-cong [where f=card])
also have ... = card (insert 0 ({k ∈ M. k < Suc i} - {0}))
  by (simp add: card.insert-remove)
also have ... = card {k ∈ M. k < Suc i}
  using assms
  by (force simp add: intro: arg-cong [where f=card])
finally show ?thesis.
qed

```

lemma *card-le-Suc-Max*: *finite S* \implies *card S* \leq *Suc* (*Max S*)

proof (*rule classical*)

assume *finite S* **and** \neg *Suc* (*Max S*) \geq *card S*

then have *Suc* (*Max S*) < *card S*

by *simp*

with \langle *finite S* \rangle **have** *S* \subseteq {0..*Max S*}

by *auto*

hence *card S* \leq *card* {0..*Max S*}

by (*intro card-mono; auto*)

thus *card S* \leq *Suc* (*Max S*)

by *simp*

qed

61.7 Lemmas useful with the summation operator sum

For examples, see Algebra/poly/UnivPoly2.thy

61.7.1 Disjoint Unions

Singletons and open intervals

lemma *ivl-disj-un-singleton*:

{*l*::'*a*::*linorder*} *Un* {*l*<..*u*} = {*l*..*u*}

{..*u*} *Un* {*u*::'*a*::*linorder*} = {*l*..*u*}

(*l*::'*a*::*linorder*) < *u* \implies {*l*} *Un* {*l*<..*u*} = {*l*..*u*}

(*l*::'*a*::*linorder*) < *u* \implies {*l*<..*u*} *Un* {*u*} = {*l*<..*u*}

(*l*::'*a*::*linorder*) \leq *u* \implies {*l*} *Un* {*l*<..*u*} = {*l*..*u*}

(*l*::'*a*::*linorder*) \leq *u* \implies {*l*..*u*} *Un* {*u*} = {*l*..*u*}

by *auto*

One- and two-sided intervals

lemma *ivl-disj-un-one*:

(*l*::'*a*::*linorder*) < *u* \implies {..*l*} *Un* {*l*<..*u*} = {..*u*}

(*l*::'*a*::*linorder*) \leq *u* \implies {..*l*} *Un* {*l*..*u*} = {..*u*}

(*l*::'*a*::*linorder*) \leq *u* \implies {..*l*} *Un* {*l*<..*u*} = {..*u*}

(*l*::'*a*::*linorder*) \leq *u* \implies {..*l*} *Un* {*l*..*u*} = {..*u*}

(*l*::'*a*::*linorder*) \leq *u* \implies {*l*<..*u*} *Un* {*u*<..*u*} = {*l*<..*u*}

(*l*::'*a*::*linorder*) < *u* \implies {*l*<..*u*} *Un* {*u*..*u*} = {*l*<..*u*}

$$\begin{aligned} (l::'a::\text{linorder}) \leq u & \implies \{l..u\} \text{ Un } \{u<..\} = \{l..\} \\ (l::'a::\text{linorder}) \leq u & \implies \{l..<u\} \text{ Un } \{u..\} = \{l..\} \end{aligned}$$

by auto

Two- and two-sided intervals

lemma *ivl-disj-un-two*:

$$\begin{aligned} \llbracket (l::'a::\text{linorder}) < m; m \leq u \rrbracket & \implies \{l<..$$

by auto

lemma *ivl-disj-un-two-touch*:

$$\begin{aligned} \llbracket (l::'a::\text{linorder}) < m; m < u \rrbracket & \implies \{l<..m\} \text{ Un } \{m..<u\} = \{l<..$$

by auto

lemmas *ivl-disj-un* = *ivl-disj-un-singleton* *ivl-disj-un-one* *ivl-disj-un-two* *ivl-disj-un-two-touch*

61.7.2 Disjoint Intersections

One- and two-sided intervals

lemma *ivl-disj-int-one*:

$$\begin{aligned} \{..l::'a::\text{order}\} \text{ Int } \{l<..$$

by auto

Two- and two-sided intervals

lemma *ivl-disj-int-two*:

$$\begin{aligned} \{l::'a::\text{order}<..$$

$\{l..m\} \text{ Int } \{m<..u\} = \{\}$
by *auto*

lemmas *ivl-disj-int = ivl-disj-int-one ivl-disj-int-two*

61.7.3 Some Differences

lemma *ivl-diff[simp]*:
 $i \leq n \implies \{i..<m\} - \{i..<n\} = \{n..<(m::'a::\text{linorder})\}$
by(*auto*)

lemma (**in** *linorder*) *lessThan-minus-lessThan [simp]*:
 $\{..<n\} - \{..<m\} = \{m ..<n\}$
by *auto*

lemma (**in** *linorder*) *atLeastAtMost-diff-ends*:
 $\{a..b\} - \{a, b\} = \{a<..<b\}$
by *auto*

61.7.4 Some Subset Conditions

lemma *ivl-subset [simp]*: $(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \vee m \leq i \wedge j \leq (n::'a::\text{linorder}))$
using *linorder-class.le-less-linear[of i n]*
by *safe (force intro: leI)+*

61.8 Generic big monoid operation over intervals

context *semiring-char-0*
begin

lemma *inj-on-of-nat [simp]*:
inj-on of-nat N
by (*rule inj-onI*) *simp*

lemma *bij-betw-of-nat [simp]*:
bij-betw of-nat N A \longleftrightarrow of-nat ' N = A
by (*simp add: bij-betw-def*)

lemma *Nats-infinite: infinite (N :: 'a set)*
by (*metis Nats-def finite-imageD infinite-UNIV-char-0 inj-on-of-nat*)

end

context *comm-monoid-set*
begin

lemma *atLeastLessThan-reindex*:
 $F g \{h m..<h n\} = F (g \circ h) \{m..<n\}$
if *bij-betw h {m..<n} {h m..<h n}* **for** *m n :: nat*
proof –

from *that have* $\text{inj-on } h \{m..<n\}$ **and** $h \text{ ‘ } \{m..<n\} = \{h \ m..<h \ n\}$
by (*simp-all add: bij-betw-def*)
then show *?thesis*
using *reindex [of h {m..<n} g]* **by** *simp*
qed

lemma *atLeastAtMost-reindex:*
 $F \ g \ \{h \ m..h \ n\} = F \ (g \circ h) \ \{m..n\}$
if *bij-betw h {m..n} {h m..h n}* **for** $m \ n \ :: \text{nat}$
proof –
from *that have* $\text{inj-on } h \{m..n\}$ **and** $h \text{ ‘ } \{m..n\} = \{h \ m..h \ n\}$
by (*simp-all add: bij-betw-def*)
then show *?thesis*
using *reindex [of h {m..n} g]* **by** *simp*
qed

lemma *atLeastLessThan-shift-bounds:*
 $F \ g \ \{m + k..<n + k\} = F \ (g \circ \text{plus } k) \ \{m..<n\}$
for $m \ n \ k \ :: \text{nat}$
using *atLeastLessThan-reindex [of plus k m n g]*
by (*simp add: ac-simps*)

lemma *atLeastAtMost-shift-bounds:*
 $F \ g \ \{m + k..n + k\} = F \ (g \circ \text{plus } k) \ \{m..n\}$
for $m \ n \ k \ :: \text{nat}$
using *atLeastAtMost-reindex [of plus k m n g]*
by (*simp add: ac-simps*)

lemma *atLeast-Suc-lessThan-Suc-shift:*
 $F \ g \ \{\text{Suc } m..<\text{Suc } n\} = F \ (g \circ \text{Suc}) \ \{m..<n\}$
using *atLeastLessThan-shift-bounds [of - - 1]*
by (*simp add: plus-1-eq-Suc*)

lemma *atLeast-Suc-atMost-Suc-shift:*
 $F \ g \ \{\text{Suc } m..n\} = F \ (g \circ \text{Suc}) \ \{m..n\}$
using *atLeastAtMost-shift-bounds [of - - 1]*
by (*simp add: plus-1-eq-Suc*)

lemma *atLeast-atMost-pred-shift:*
 $F \ (g \circ (\lambda n. n - \text{Suc } 0)) \ \{\text{Suc } m..n\} = F \ g \ \{m..n\}$
unfolding *atLeast-Suc-atMost-Suc-shift* **by** *simp*

lemma *atLeast-lessThan-pred-shift:*
 $F \ (g \circ (\lambda n. n - \text{Suc } 0)) \ \{\text{Suc } m..<n\} = F \ g \ \{m..<n\}$
unfolding *atLeast-Suc-lessThan-Suc-shift* **by** *simp*

lemma *atLeast-int-lessThan-int-shift:*
 $F \ g \ \{\text{int } m..<\text{int } n\} = F \ (g \circ \text{int}) \ \{m..<n\}$
by (*rule atLeastLessThan-reindex*)

(simp add: image-int-atLeastLessThan)

lemma *atLeast-int-atMost-int-shift*:

$F g \{ \text{int } m.. \text{int } n \} = F (g \circ \text{int}) \{ m..n \}$

by (rule *atLeastAtMost-reindex*)

(simp add: image-int-atLeastAtMost)

lemma *atLeast0-lessThan-Suc*:

$F g \{ 0..< \text{Suc } n \} = F g \{ 0..<n \} * g n$

by (simp add: *atLeast0-lessThan-Suc ac-simps*)

lemma *atLeast0-atMost-Suc*:

$F g \{ 0.. \text{Suc } n \} = F g \{ 0..n \} * g (\text{Suc } n)$

by (simp add: *atLeast0-atMost-Suc ac-simps*)

lemma *atLeast0-lessThan-Suc-shift*:

$F g \{ 0..< \text{Suc } n \} = g 0 * F (g \circ \text{Suc}) \{ 0..<n \}$

by (simp add: *atLeast0-lessThan-Suc-eq-insert-0 atLeast-Suc-lessThan-Suc-shift*)

lemma *atLeast0-atMost-Suc-shift*:

$F g \{ 0.. \text{Suc } n \} = g 0 * F (g \circ \text{Suc}) \{ 0..n \}$

by (simp add: *atLeast0-atMost-Suc-eq-insert-0 atLeast-Suc-atMost-Suc-shift*)

lemma *atLeast-Suc-lessThan*:

$F g \{ m..<n \} = g m * F g \{ \text{Suc } m..<n \}$ **if** $m < n$

proof –

from *that have* $\{ m..<n \} = \text{insert } m \{ \text{Suc } m..<n \}$

by *auto*

then show *?thesis* **by** *simp*

qed

lemma *atLeast-Suc-atMost*:

$F g \{ m..n \} = g m * F g \{ \text{Suc } m..n \}$ **if** $m \leq n$

proof –

from *that have* $\{ m..n \} = \text{insert } m \{ \text{Suc } m..n \}$

by *auto*

then show *?thesis* **by** *simp*

qed

lemma *ivl-cong*:

$a = c \implies b = d \implies (\bigwedge x. c \leq x \implies x < d \implies g x = h x)$

$\implies F g \{ a..<b \} = F h \{ c..<d \}$

by (rule *cong*) *simp-all*

lemma *atLeastLessThan-shift-0*:

fixes $m n p :: \text{nat}$

shows $F g \{ m..<n \} = F (g \circ \text{plus } m) \{ 0..<n - m \}$

using *atLeastLessThan-shift-bounds* [*of* $g 0 m n - m$]

by (*cases* $m \leq n$) *simp-all*

lemma *atLeastAtMost-shift-0*:

fixes $m\ n\ p :: \text{nat}$

assumes $m \leq n$

shows $F\ g\ \{m..n\} = F\ (g \circ \text{plus}\ m)\ \{0..n - m\}$

using *assms atLeastAtMost-shift-bounds* [of $g\ 0\ m\ n - m$] **by** *simp*

lemma *atLeastLessThan-concat*:

fixes $m\ n\ p :: \text{nat}$

shows $m \leq n \implies n \leq p \implies F\ g\ \{m..<n\} * F\ g\ \{n..<p\} = F\ g\ \{m..<p\}$

by (*simp add: union-disjoint [symmetric] ivl-disj-un*)

lemma *atLeastLessThan-rev*:

$F\ g\ \{n..<m\} = F\ (\lambda i. g\ (m + n - \text{Suc}\ i))\ \{n..<m\}$

by (*rule reindex-bij-witness* [where $i=\lambda i. m + n - \text{Suc}\ i$ and $j=\lambda i. m + n - \text{Suc}\ i$], *auto*)

lemma *atLeastAtMost-rev*:

fixes $n\ m :: \text{nat}$

shows $F\ g\ \{n..m\} = F\ (\lambda i. g\ (m + n - i))\ \{n..m\}$

by (*rule reindex-bij-witness* [where $i=\lambda i. m + n - i$ and $j=\lambda i. m + n - i$])

auto

lemma *atLeastLessThan-rev-at-least-Suc-atMost*:

$F\ g\ \{n..<m\} = F\ (\lambda i. g\ (m + n - i))\ \{\text{Suc}\ n..m\}$

unfolding *atLeastLessThan-rev* [of $g\ n\ m$]

by (*cases m*) (*simp-all add: atLeast-Suc-atMost-Suc-shift atLeastLessThanSuc-atLeastAtMost*)

end

61.9 Summation indexed over intervals

syntax (*ASCII*)

-from-to-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\text{SUM}\ - = \dots / -)\ [0,0,0,10]\ 10)$

-from-upto-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\text{SUM}\ - = \dots < / -)\ [0,0,0,10]\ 10)$

-upt-sum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\text{SUM}\ - < / -)\ [0,0,10]\ 10)$

-upto-sum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\text{SUM}\ - < = / -)\ [0,0,10]\ 10)$

syntax (*latex-sum output*)

-from-to-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

$((\exists \sum \bar{_} = _ -)\ [0,0,0,10]\ 10)$

-from-upto-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

$((\exists \sum \bar{_} < _ -)\ [0,0,0,10]\ 10)$

-upt-sum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

$((\exists \sum \bar{_} < _ -)\ [0,0,10]\ 10)$

-upto-sum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

$((\exists \sum \bar{_} \leq _ -)\ [0,0,10]\ 10)$

syntax

```

-from-to-sum :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∑ - = -..-./ -) [0,0,0,10] 10)
-from-upto-sum :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∑ - = -..<-./ -) [0,0,0,10] 10)
-upt-sum :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∑ -<-./ -) [0,0,10] 10)
-upto-sum :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∑ -≤-./ -) [0,0,10] 10)

```

translations

```

∑ x=a..b. t == CONST sum (λx. t) {a..b}
∑ x=a..<b. t == CONST sum (λx. t) {a..<b}
∑ i≤n. t == CONST sum (λi. t) {..n}
∑ i<n. t == CONST sum (λi. t) {..<n}

```

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	L ^A T _E X
$\sum x \in \{a..b\}. e$	$\sum x = a..b. e$	$\sum_{x=a}^b e$
$\sum x \in \{a..<b\}. e$	$\sum x = a..<b. e$	$\sum_{x=a}^{<b} e$
$\sum x \in \{..b\}. e$	$\sum x \leq b. e$	$\sum_{x \leq b} e$
$\sum x \in \{..<b\}. e$	$\sum x < b. e$	$\sum_{x < b} e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode = latex-sum* in antiquotations). It is not the default L^AT_EX output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use $\sum x = 0..<n. e$ rather than $\sum x <n. e$: *sum* may not provide all lemmas available for $\{m..<n\}$ also in the special form for $\{..<n\}$.

This congruence rule should be used for sums over intervals as the standard theorem *sum.cong* does not work well with the simplifier who adds the unsimplified premise $x \in B$ to the context.

```

context comm-monoid-set
begin

```

```

lemma zero-middle:

```

```

  assumes 1 ≤ p k ≤ p

```

```

  shows F (λj. if j < k then g j else if j = k then 1 else h (j - Suc 0)) {..p}
    = F (λj. if j < k then g j else h j) {..p - Suc 0} (is ?lhs = ?rhs)

```

```

proof -

```

```

  have [simp]: {..p - Suc 0} ∩ {j. j < k} = {..<k} {..p - Suc 0} ∩ - {j. j < k}
= {k..p - Suc 0}

```

```

  using assms by auto

```

```

  have ?lhs = F g {..<k} * F (λj. if j = k then 1 else h (j - Suc 0)) {k..p}

```

```

  using union-disjoint [of {..<k} {k..p}] assms

```

```

  by (simp add: ivl-disj-int-one ivl-disj-un-one)

```

also have $\dots = F g \{..<k\} * F (\lambda j. h (j - Suc 0)) \{Suc k..p\}$
by (*simp add: atLeast-Suc-atMost [of k p] assms*)
also have $\dots = F g \{..<k\} * F h \{k .. p - Suc 0\}$
using *reindex [of Suc {k..p - Suc 0}] assms* **by** *simp*
also have $\dots = ?rhs$
by (*simp add: If-cases*)
finally show *?thesis* .
qed

lemma *atMost-Suc [simp]:*
 $F g \{..Suc n\} = F g \{..n\} * g (Suc n)$
by (*simp add: atMost-Suc ac-simps*)

lemma *lessThan-Suc [simp]:*
 $F g \{..<Suc n\} = F g \{..<n\} * g n$
by (*simp add: lessThan-Suc ac-simps*)

lemma *cl-ivl-Suc [simp]:*
 $F g \{m..Suc n\} = (if\ Suc\ n < m\ then\ \mathbf{1}\ else\ F\ g\ \{m..n\} * g(Suc\ n))$
by (*auto simp: ac-simps atLeastAtMostSuc-conv*)

lemma *op-ivl-Suc [simp]:*
 $F g \{m..<Suc n\} = (if\ n < m\ then\ \mathbf{1}\ else\ F\ g\ \{m..<n\} * g(n))$
by (*auto simp: ac-simps atLeastLessThanSuc*)

lemma *head:*
fixes $n :: nat$
assumes $mn: m \leq n$
shows $F g \{m..n\} = g m * F g \{m<..n\}$ (**is** *?lhs = ?rhs*)
proof –
from mn
have $\{m..n\} = \{m\} \cup \{m<..n\}$
by (*auto intro: ivl-disj-un-singleton*)
hence *?lhs* $= F g (\{m\} \cup \{m<..n\})$
by (*simp add: atLeast0LessThan*)
also have $\dots = ?rhs$ **by** *simp*
finally show *?thesis* .
qed

lemma *last-plus:*
fixes $n::nat$ **shows** $m \leq n \implies F g \{m..n\} = g n * F g \{m..<n\}$
by (*cases n*) (*auto simp: atLeastLessThanSuc-atLeastAtMost commute*)

lemma *head-if:*
fixes $n :: nat$
shows $F g \{m..n\} = (if\ n < m\ then\ \mathbf{1}\ else\ F\ g\ \{m..<n\} * g(n))$
by (*simp add: commute last-plus*)

lemma *ub-add-nat:*

assumes $(m::nat) \leq n + 1$
shows $F g \{m..n + p\} = F g \{m..n\} * F g \{n + 1..n + p\}$
proof –
have $\{m .. n+p\} = \{m..n\} \cup \{n+1..n+p\}$ **using** $\langle m \leq n+1 \rangle$ **by** *auto*
thus *?thesis* **by** (*auto simp: ivl-disj-int union-disjoint atLeastSucAtMost-greaterThanAtMost*)
qed

lemma *nat-group*:
fixes $k::nat$ **shows** $F (\lambda m. F g \{m * k ..< m*k + k\}) \{..<n\} = F g \{..< n * k\}$
proof (*cases k*)
case (*Suc l*)
then have $k > 0$
by *auto*
then show *?thesis*
by (*induct n*) (*simp-all add: atLeastLessThan-concat add commute atLeast0LessThan[symmetric]*)
qed *auto*

lemma *triangle-reindex*:
fixes $n :: nat$
shows $F (\lambda(i,j). g i j) \{(i,j). i+j < n\} = F (\lambda k. F (\lambda i. g i (k - i)) \{..k\}) \{..<n\}$
apply (*simp add: Sigma*)
apply (*rule reindex-bij-witness[where j= $\lambda(i, j). (i+j, i)$ and $i=\lambda(k, i). (i, k - i)$]*)
apply *auto*
done

lemma *triangle-reindex-eq*:
fixes $n :: nat$
shows $F (\lambda(i,j). g i j) \{(i,j). i+j \leq n\} = F (\lambda k. F (\lambda i. g i (k - i)) \{..k\}) \{..n\}$
using *triangle-reindex [of g Suc n]*
by (*simp only: Nat.less-Suc-eq-le lessThan-Suc-atMost*)

lemma *nat-diff-reindex*: $F (\lambda i. g (n - Suc i)) \{..<n\} = F g \{..<n\}$
by (*rule reindex-bij-witness[where $i=\lambda i. n - Suc i$ and $j=\lambda i. n - Suc i$]*) *auto*

lemma *shift-bounds-nat-ivl*:
 $F g \{m+k..<n+k\} = F (\lambda i. g(i + k))\{m..<n::nat\}$
by (*induct n, auto simp: atLeastLessThanSuc*)

lemma *shift-bounds-cl-nat-ivl*:
 $F g \{m+k..n+k\} = F (\lambda i. g(i + k))\{m..n::nat\}$
by (*rule reindex-bij-witness[where $i=\lambda i. i + k$ and $j=\lambda i. i - k$]*) *auto*

corollary *shift-bounds-cl-Suc-ivl*:
 $F g \{Suc m..Suc n\} = F (\lambda i. g(Suc i))\{m..n\}$
by (*simp add: shift-bounds-cl-nat-ivl[where $k=Suc 0$, simplified]*)

corollary *Suc-reindex-ivl*: $m \leq n \implies F g \{m..n\} * g (Suc n) = g m * F (\lambda i. g$

$(\text{Suc } i) \{m..n\}$

by (*simp add: assoc atLeast-Suc-atMost flip: shift-bounds-cl-Suc-ivl*)

corollary *shift-bounds-Suc-ivl*:

$F g \{\text{Suc } m..<\text{Suc } n\} = F (\lambda i. g(\text{Suc } i))\{m..<n\}$

by (*simp add: shift-bounds-nat-ivl[where k=Suc 0, simplified]*)

lemma *atMost-Suc-shift*:

shows $F g \{..\text{Suc } n\} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\}$

proof (*induct n*)

case 0 show ?case by *simp*

next

case (*Suc n*) note *IH = this*

have $F g \{..\text{Suc } (\text{Suc } n)\} = F g \{..\text{Suc } n\} * g (\text{Suc } (\text{Suc } n))$

by (*rule atMost-Suc*)

also have $F g \{..\text{Suc } n\} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\}$

by (*rule IH*)

also have $g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\} * g (\text{Suc } (\text{Suc } n)) =$

$g 0 * (F (\lambda i. g (\text{Suc } i)) \{..n\} * g (\text{Suc } (\text{Suc } n)))$

by (*rule assoc*)

also have $F (\lambda i. g (\text{Suc } i)) \{..n\} * g (\text{Suc } (\text{Suc } n)) = F (\lambda i. g (\text{Suc } i)) \{..\text{Suc } n\}$

by (*rule atMost-Suc [symmetric]*)

finally show ?case .

qed

lemma *lessThan-Suc-shift*:

$F g \{..<\text{Suc } n\} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..<n\}$

by (*induction n*) (*simp-all add: ac-simps*)

lemma *atMost-shift*:

$F g \{..n\} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..<n\}$

by (*metis atLeast0AtMost atLeast0LessThan atLeastLessThanSuc-atLeastAtMost*

atLeastSucAtMost-greaterThanAtMost le0 head shift-bounds-Suc-ivl)

lemma *nested-swap*:

$F (\lambda i. F (\lambda j. a i j) \{0..<i\}) \{0..n\} = F (\lambda j. F (\lambda i. a i j) \{\text{Suc } j..n\}) \{0..<n\}$

by (*induction n*) (*auto simp: distrib*)

lemma *nested-swap'*:

$F (\lambda i. F (\lambda j. a i j) \{..<i\}) \{..n\} = F (\lambda j. F (\lambda i. a i j) \{\text{Suc } j..n\}) \{..<n\}$

by (*induction n*) (*auto simp: distrib*)

lemma *atLeast1-atMost-eq*:

$F g \{\text{Suc } 0..n\} = F (\lambda k. g (\text{Suc } k)) \{..<n\}$

proof –

have $F g \{\text{Suc } 0..n\} = F g (\text{Suc } \{..<n\})$

by (*simp add: image-Suc-lessThan*)

also have $\dots = F (\lambda k. g (Suc k)) \{..<n\}$
 by (*simp add: reindex*)
 finally show *?thesis* .
 qed

lemma *atLeastLessThan-Suc*: $a \leq b \implies F g \{a..<Suc b\} = F g \{a..<b\} * g b$
 by (*simp add: atLeastLessThanSuc commute*)

lemma *nat-ivl-Suc'*:
 assumes $m \leq Suc n$
 shows $F g \{m..Suc n\} = g (Suc n) * F g \{m..n\}$
 proof –
 from *assms* have $\{m..Suc n\} = insert (Suc n) \{m..n\}$ by *auto*
 also have $F g \dots = g (Suc n) * F g \{m..n\}$ by *simp*
 finally show *?thesis* .
 qed

lemma *in-pairs*: $F g \{2*m..Suc(2*n)\} = F (\lambda i. g(2*i) * g(Suc(2*i))) \{m..n\}$
 proof (*induction n*)
 case 0
 show *?case*
 by (*cases m=0*) *auto*
 next
 case (*Suc n*)
 then show *?case*
 by (*auto simp: assoc split: if-split-asm*)
 qed

lemma *in-pairs-0*: $F g \{..Suc(2*n)\} = F (\lambda i. g(2*i) * g(Suc(2*i))) \{..n\}$
 using *in-pairs* [*of - 0 n*] by (*simp add: atLeast0AtMost*)

end

lemma *card-sum-le-nat-sum*: $\sum \{0..<card S\} \leq \sum S$
 proof (*cases finite S*)
 case *True*
 then show *?thesis*
 proof (*induction card S arbitrary: S*)
 case (*Suc x*)
 then have $Max S \geq x$ using *card-le-Suc-Max* by *fastforce*
 let $?S' = S - \{Max S\}$
 from *Suc* have $Max S \in S$ by (*auto intro: Max-in*)
 hence *cards*: $card S = Suc (card ?S')$
 using $\langle finite S \rangle$ by (*intro card.remove; auto*)
 hence $\sum \{0..<card ?S'\} \leq \sum ?S'$
 using *Suc* by (*intro Suc; auto*)
 hence $\sum \{0..<card ?S'\} + x \leq \sum ?S' + Max S$
 using $\langle Max S \geq x \rangle$ by *simp*

also have $\dots = \sum S$
using *sum.remove*[*OF* $\langle \text{finite } S \rangle \langle \text{Max } S \in S \rangle$, **where** $g = \lambda x. x$]
by *simp*
finally show *?case*
using *cards Suc* **by** *auto*
qed *simp*
qed *simp*

lemma *sum-natinterval-diff*:
fixes $f :: \text{nat} \Rightarrow ('a :: \text{ab-group-add})$
shows $\text{sum } (\lambda k. f k - f(k + 1)) \{m :: \text{nat} \dots n\} =$
 $(\text{if } m \leq n \text{ then } f m - f(n + 1) \text{ else } 0)$
by (*induct n, auto simp add: algebra-simps not-le le-Suc-eq*)

lemma *sum-diff-nat-ivl*:
fixes $f :: \text{nat} \Rightarrow ('a :: \text{ab-group-add})$
shows $\llbracket m \leq n; n \leq p \rrbracket \implies \text{sum } f \{m..<p\} - \text{sum } f \{m..<n\} = \text{sum } f \{n..<p\}$
using *sum.atLeastLessThan-concat* [*of m n p f, symmetric*]
by (*simp add: ac-simps*)

lemma *sum-diff-distrib*: $\forall x. Q x \leq P x \implies (\sum x < n. P x) - (\sum x < n. Q x) =$
 $(\sum x < n. P x - Q x :: \text{nat})$
by (*subst sum-subtractf-nat*) *auto*

61.9.1 Shifting bounds

context *comm-monoid-add*
begin

context
fixes $f :: \text{nat} \Rightarrow 'a$
assumes $f 0 = 0$
begin

lemma *sum-shift-lb-Suc0-0-upt*:
 $\text{sum } f \{ \text{Suc } 0 .. < k \} = \text{sum } f \{ 0 .. < k \}$
proof (*cases k*)
case 0
then show *?thesis*
by *simp*
next
case (*Suc k*)
moreover have $\{ 0 .. < \text{Suc } k \} = \text{insert } 0 \{ \text{Suc } 0 .. < \text{Suc } k \}$
by *auto*
ultimately show *?thesis*
using $\langle f 0 = 0 \rangle$ **by** *simp*
qed

lemma *sum-shift-lb-Suc0-0*: $\text{sum } f \{ \text{Suc } 0 .. k \} = \text{sum } f \{ 0 .. k \}$

```

proof (cases k)
  case 0
    with ⟨f 0 = 0⟩ show ?thesis
    by simp
  next
    case (Suc k)
    moreover have {0..Suc k} = insert 0 {Suc 0..Suc k}
    by auto
    ultimately show ?thesis
    using ⟨f 0 = 0⟩ by simp
qed

end

end

```

```

lemma sum-Suc-diff:
  fixes f :: nat ⇒ 'a::ab-group-add
  assumes m ≤ Suc n
  shows (∑ i = m..n. f (Suc i) - f i) = f (Suc n) - f m
using assms by (induct n) (auto simp: le-Suc-eq)

```

```

lemma sum-Suc-diff':
  fixes f :: nat ⇒ 'a::ab-group-add
  assumes m ≤ n
  shows (∑ i = m..<n. f (Suc i) - f i) = f n - f m
using assms by (induct n) (auto simp: le-Suc-eq)

```

61.9.2 Telescoping

```

lemma sum-telescope:
  fixes f::nat ⇒ 'a::ab-group-add
  shows sum (λi. f i - f (Suc i)) {.. i} = f 0 - f (Suc i)
  by (induct i) simp-all

```

```

lemma sum-telescope'':
  assumes m ≤ n
  shows (∑ k∈{Suc m..n}. f k - f (k - 1)) = f n - (f m :: 'a :: ab-group-add)
  by (rule dec-induct[OF assms]) (simp-all add: algebra-simps)

```

```

lemma sum-lessThan-telescope:
  (∑ n<m. f (Suc n) - f n :: 'a :: ab-group-add) = f m - f 0
  by (induction m) (simp-all add: algebra-simps)

```

```

lemma sum-lessThan-telescope':
  (∑ n<m. f n - f (Suc n) :: 'a :: ab-group-add) = f 0 - f m
  by (induction m) (simp-all add: algebra-simps)

```


61.9.3 The formula for geometric sums

lemma *sum-power2*: $(\sum_{i=0..<k. (2::nat) \hat{i}}) = 2^k - 1$
 by (*induction k*) (*auto simp: mult-2*)

lemma *geometric-sum*:

assumes $x \neq 1$

shows $(\sum_{i<n. x \hat{i}}) = (x \hat{n} - 1) / (x - 1 :: 'a::field)$

proof –

from *assms* obtain y where $y = x - 1$ and $y \neq 0$ by *simp-all*

moreover have $(\sum_{i<n. (y + 1) \hat{i}}) = ((y + 1) \hat{n} - 1) / y$

by (*induct n*) (*simp-all add: field-simps* $\langle y \neq 0 \rangle$)

ultimately show *?thesis* by *simp*

qed

lemma *geometric-sum-less*:

assumes $0 < x < 1$ finite S

shows $(\sum_{i \in S. x \hat{i}}) < 1 / (1 - x :: 'a::linordered-field)$

proof –

define n where $n \equiv \text{Suc} (\text{Max } S)$

have $(\sum_{i \in S. x \hat{i}}) \leq (\sum_{i < n. x \hat{i}})$

unfolding *n-def* using *assms* by (*fastforce intro!: sum-mono2 le-imp-less-Suc*)

also have $\dots = (1 - x \hat{n}) / (1 - x)$

using *assms* by (*simp add: geometric-sum field-simps*)

also have $\dots < 1 / (1 - x)$

using *assms* by (*simp add: field-simps power-Suc-less*)

finally show *?thesis* .

qed

lemma *diff-power-eq-sum*:

fixes $y :: 'a::\{comm-ring, monoid-mult\}$

shows

$$x \hat{(\text{Suc } n)} - y \hat{(\text{Suc } n)} = (x - y) * (\sum_{p < \text{Suc } n. (x \hat{p}) * y \hat{(n - p)})}$$

proof (*induct n*)

case (*Suc n*)

have $x \hat{\text{Suc}} (\text{Suc } n) - y \hat{\text{Suc}} (\text{Suc } n) = x * (x * x \hat{n}) - y * (y * y \hat{n})$

by *simp*

also have $\dots = y * (x \hat{(\text{Suc } n)} - y \hat{(\text{Suc } n)}) + (x - y) * (x * x \hat{n})$

by (*simp add: algebra-simps*)

also have $\dots = y * ((x - y) * (\sum_{p < \text{Suc } n. (x \hat{p}) * y \hat{(n - p)})}) + (x - y) * (x * x \hat{n})$

by (*simp only: Suc*)

also have $\dots = (x - y) * (y * (\sum_{p < \text{Suc } n. (x \hat{p}) * y \hat{(n - p)})}) + (x - y) * (x * x \hat{n})$

by (*simp only: mult.left-commute*)

also have $\dots = (x - y) * (\sum_{p < \text{Suc}} (\text{Suc } n). x \hat{p} * y \hat{(\text{Suc } n - p)})$

by (*simp add: field-simps Suc-diff-le sum-distrib-right sum-distrib-left*)

finally show *?case* .

qed *simp*

corollary *power-diff-sumr2*: — *COMPLEX-POLYFUN* in HOL Light

fixes $x :: 'a::\{comm-ring, monoid-mult\}$
shows $x^n - y^n = (x - y) * (\sum i < n. y^{(n - Suc i)} * x^i)$
using *diff-power-eq-sum* [of $x^n - 1$ y]
by (*cases* $n = 0$) (*simp-all* *add: field-simps*)

lemma *power-diff-1-eq*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$
shows $x^n - 1 = (x - 1) * (\sum i < n. (x^i))$
using *diff-power-eq-sum* [of $x - 1$]
by (*cases* n) *auto*

lemma *one-diff-power-eq'*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$
shows $1 - x^n = (1 - x) * (\sum i < n. x^{(n - Suc i)})$
using *diff-power-eq-sum* [of $1 - x$]
by (*cases* n) *auto*

lemma *one-diff-power-eq*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$
shows $1 - x^n = (1 - x) * (\sum i < n. x^i)$
by (*metis one-diff-power-eq' sum.nat-diff-reindex*)

lemma *sum-gp-basic*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$
shows $(1 - x) * (\sum i \leq n. x^i) = 1 - x^{Suc n}$
by (*simp only: one-diff-power-eq lessThan-Suc-atMost*)

lemma *sum-power-shift*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$
assumes $m \leq n$
shows $(\sum i = m..n. x^i) = x^m * (\sum i \leq n - m. x^i)$
proof –
have $(\sum i = m..n. x^i) = x^m * (\sum i = m..n. x^{(i - m)})$
by (*simp add: sum-distrib-left power-add [symmetric]*)
also have $(\sum i = m..n. x^{(i - m)}) = (\sum i \leq n - m. x^i)$
using $\langle m \leq n \rangle$ **by** (*intro sum.reindex-bij-witness* [where $j = \lambda i. i - m$ and $i = \lambda i. i + m$]) *auto*
finally show *?thesis* .
qed

lemma *sum-gp-multiplied*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$
assumes $m \leq n$
shows $(1 - x) * (\sum i = m..n. x^i) = x^m - x^{Suc n}$
proof –
have $(1 - x) * (\sum i = m..n. x^i) = x^m * (1 - x) * (\sum i \leq n - m. x^i)$
by (*metis mult.assoc mult commute assms sum-power-shift*)

also have $\dots = x^m * (1 - x^{Suc(n-m)})$
by (*metis mult.assoc sum-gp-basic*)
also have $\dots = x^m - x^{Suc n}$
using *assms*
by (*simp add: algebra-simps*) (*metis le-add-diff-inverse power-add*)
finally show *?thesis* .
qed

lemma *sum-gp*:

fixes $x :: 'a::\{comm-ring, division-ring\}$
shows $(\sum_{i=m..n}. x^i) =$
 $(if\ n < m\ then\ 0$
 $\quad else\ if\ x = 1\ then\ of-nat((n + 1) - m)$
 $\quad else\ (x^m - x^{Suc\ n}) / (1 - x))$

proof (*cases n < m*)

case *False*

assume $*$: $\neg n < m$

then show *?thesis*

proof (*cases x = 1*)

case *False*

assume $x \neq 1$

then have *not-zero*: $1 - x \neq 0$

by *auto*

have $(1 - x) * (\sum_{i=m..n}. x^i) = x^m - x * x^n$

using *sum-gp-multiplied [of m n x]* * **by** *auto*

then have $(\sum_{i=m..n}. x^i) = (x^m - x * x^n) / (1 - x)$

using *nonzero-divide-eq-eq mult.commute not-zero*

by *metis*

then show *?thesis*

by *auto*

qed (*auto*)

qed (*auto*)

61.9.4 Geometric progressions

lemma *sum-gp0*:

fixes $x :: 'a::\{comm-ring, division-ring\}$

shows $(\sum_{i \leq n}. x^i) = (if\ x = 1\ then\ of-nat(n + 1)\ else\ (1 - x^{Suc\ n}) / (1 - x))$

using *sum-gp-basic [of x n]*

by (*simp add: mult.commute field-split-simps*)

lemma *sum-power-add*:

fixes $x :: 'a::\{comm-ring, monoid-mult\}$

shows $(\sum_{i \in I}. x^{(m+i)}) = x^m * (\sum_{i \in I}. x^i)$

by (*simp add: sum-distrib-left power-add*)

lemma *sum-gp-offset*:

fixes $x :: 'a::\{comm-ring, division-ring\}$

shows $(\sum_{i=m..m+n} x^i) =$
 $(\text{if } x = 1 \text{ then of-nat } n + 1 \text{ else } x^m * (1 - x^{\text{Suc } n}) / (1 - x))$
using *sum-gp* [*of x m m+n*]
by (*auto simp: power-add algebra-simps*)

lemma *sum-gp-strict*:

fixes $x :: 'a::\{\text{comm-ring, division-ring}\}$
shows $(\sum_{i<n} x^i) = (\text{if } x = 1 \text{ then of-nat } n \text{ else } (1 - x^n) / (1 - x))$
by (*induct n*) (*auto simp: algebra-simps field-split-simps*)

61.9.5 The formulae for arithmetic sums

context *comm-semiring-1*

begin

lemma *double-gauss-sum*:

$2 * (\sum_{i=0..n} \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1)$
by (*induct n*) (*simp-all add: sum.atLeast0-atMost-Suc algebra-simps left-add-twice*)

lemma *double-gauss-sum-from-Suc-0*:

$2 * (\sum_{i=\text{Suc } 0..n} \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1)$

proof –

have $\text{sum of-nat } \{\text{Suc } 0..n\} = \text{sum of-nat } (\text{insert } 0 \ \{\text{Suc } 0..n\})$

by *simp*

also have $\dots = \text{sum of-nat } \{0..n\}$

by (*cases n*) (*simp-all add: atLeast0-atMost-Suc-eq-insert-0*)

finally show *?thesis*

by (*simp add: double-gauss-sum*)

qed

lemma *double-arith-series*:

$2 * (\sum_{i=0..n} a + \text{of-nat } i * d) = (\text{of-nat } n + 1) * (2 * a + \text{of-nat } n * d)$

proof –

have $(\sum_{i=0..n} a + \text{of-nat } i * d) = ((\sum_{i=0..n} a) + (\sum_{i=0..n} \text{of-nat } i * d))$

by (*rule sum.distrib*)

also have $\dots = (\text{of-nat } (\text{Suc } n) * a + d * (\sum_{i=0..n} \text{of-nat } i))$

by (*simp add: sum-distrib-left algebra-simps*)

finally show *?thesis*

by (*simp add: algebra-simps double-gauss-sum left-add-twice*)

qed

end

context *unique-euclidean-semiring-with-nat*

begin

lemma *gauss-sum*:

$(\sum_{i=0..n} \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1) \text{ div } 2$

using *double-gauss-sum* [of *n*, *symmetric*] by *simp*

lemma *gauss-sum-from-Suc-0*:

$(\sum i = \text{Suc } 0..n. \text{of-nat } i) = \text{of-nat } n * (\text{of-nat } n + 1) \text{ div } 2$
 using *double-gauss-sum-from-Suc-0* [of *n*, *symmetric*] by *simp*

lemma *arith-series*:

$(\sum i = 0..n. a + \text{of-nat } i * d) = (\text{of-nat } n + 1) * (2 * a + \text{of-nat } n * d) \text{ div } 2$
 using *double-arith-series* [of *a d n*, *symmetric*] by *simp*

end

lemma *gauss-sum-nat*:

$\sum \{0..n\} = (n * \text{Suc } n) \text{ div } 2$
 using *gauss-sum* [of *n*, where *?'a = nat*] by *simp*

lemma *arith-series-nat*:

$(\sum i = 0..n. a + i * d) = \text{Suc } n * (2 * a + n * d) \text{ div } 2$
 using *arith-series* [of *a d n*] by *simp*

lemma *Sum-Icc-int*:

$\sum \{m..n\} = (n * (n + 1) - m * (m - 1)) \text{ div } 2$
 if $m \leq n$ for $m n :: \text{int}$

using *that proof* (induct $i \equiv \text{nat } (n - m)$ arbitrary: $m n$)

case 0

then have $m = n$

by *arith*

then show *?case*

by (*simp add: algebra-simps mult-2 [symmetric]*)

next

case (*Suc i*)

have 0: $i = \text{nat}((n-1) - m) m \leq n-1$ using *Suc(2,3)* by *arith+*

have $\sum \{m..n\} = \sum \{m..1+(n-1)\}$ by *simp*

also have $\dots = \sum \{m..n-1\} + n$ using $\langle m \leq n \rangle$

by (*subst atLeastAtMostPlus1-int-conv*) *simp-all*

also have $\dots = ((n-1)*(n-1+1) - m*(m-1)) \text{ div } 2 + n$

by (*simp add: Suc(1)[OF 0]*)

also have $\dots = ((n-1)*(n-1+1) - m*(m-1) + 2*n) \text{ div } 2$ by *simp*

also have $\dots = (n*(n+1) - m*(m-1)) \text{ div } 2$

by (*simp add: algebra-simps mult-2-right*)

finally show *?case* .

qed

lemma *Sum-Icc-nat*:

$\sum \{m..n\} = (n * (n + 1) - m * (m - 1)) \text{ div } 2$ for $m n :: \text{nat}$

proof (cases $m \leq n$)

case *True*

then have *: $m * (m - 1) \leq n * (n + 1)$

by (*meson diff-le-self order-trans le-add1 mult-le-mono*)

```

have int ( $\sum \{m..n\}$ ) = ( $\sum \{int\ m..int\ n\}$ )
  by (simp add: sum.atLeast-int-atMost-int-shift)
also have ... = (int n * (int n + 1) - int m * (int m - 1)) div 2
  using <m ≤ n> by (simp add: Sum-Icc-int)
also have ... = int ((n * (n + 1) - m * (m - 1)) div 2)
  using le-square * by (simp add: algebra-simps of-nat-div of-nat-diff)
finally show ?thesis
  by (simp only: of-nat-eq-iff)
next
case False
then show ?thesis
  by (auto dest: less-imp-Suc-add simp add: not-le algebra-simps)
qed

```

lemma Sum-Ico-nat:

```

 $\sum \{m..<n\}$  = (n * (n - 1) - m * (m - 1)) div 2 for m n :: nat
  by (cases n) (simp-all add: atLeastLessThanSuc-atLeastAtMost Sum-Icc-nat)

```

61.9.6 Division remainder

lemma range-mod:

```

fixes n :: nat
assumes n > 0
shows range ( $\lambda m. m \bmod n$ ) = {0..<n} (is ?A = ?B)
proof (rule set-eqI)
  fix m
  show m ∈ ?A  $\longleftrightarrow$  m ∈ ?B
  proof
    assume m ∈ ?A
    with assms show m ∈ ?B
      by auto
  next
    assume m ∈ ?B
    moreover have m mod n ∈ ?A
      by (rule rangeI)
    ultimately show m ∈ ?A
      by simp
  qed
qed

```

61.10 Products indexed over intervals

syntax (ASCII)

```

-from-to-prod :: idt  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  'b ((PROD - = -..-/ -) [0,0,0,10] 10)
-from-upto-prod :: idt  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  'b ((PROD - = -..<./ -) [0,0,0,10] 10)
-upt-prod :: idt  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  'b ((PROD -<./ -) [0,0,10] 10)
-upto-prod :: idt  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  'b ((PROD -<=./ -) [0,0,10] 10)

```

syntax (latex-prod output)

$\text{-from-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\exists \prod _ = _ -) [0,0,0,10] 10)$
 $\text{-from-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\exists \prod _ < _ -) [0,0,0,10] 10)$
 $\text{-upt-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\exists \prod _ < _ -) [0,0,10] 10)$
 $\text{-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\exists \prod _ \leq _ -) [0,0,10] 10)$

syntax

$\text{-from-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\exists \prod _ = _ _ / _ -) [0,0,0,10] 10)$
 $\text{-from-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\exists \prod _ = _ _ < _ / _ -) [0,0,0,10] 10)$
 $\text{-upt-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\exists \prod _ < _ / _ -) [0,0,10] 10)$
 $\text{-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\exists \prod _ \leq _ / _ -) [0,0,10] 10)$

translations

$\prod x=a..b. t \equiv \text{CONST prod } (\lambda x. t) \{a..b\}$
 $\prod x=a..<b. t \equiv \text{CONST prod } (\lambda x. t) \{a..<b\}$
 $\prod i \leq n. t \equiv \text{CONST prod } (\lambda i. t) \{..n\}$
 $\prod i < n. t \equiv \text{CONST prod } (\lambda i. t) \{..<n\}$

lemma *prod-int-plus-eq*: $\text{prod int } \{i..i+j\} = \prod \{ \text{int } i.. \text{int } (i+j) \}$

by (*induct j*) (*auto simp add: atLeastAtMostSuc-conv atLeastAtMostPlus1-int-conv*)

lemma *prod-int-eq*: $\text{prod int } \{i..j\} = \prod \{ \text{int } i.. \text{int } j \}$

proof (*cases i ≤ j*)

case *True*

then show *?thesis*

by (*metis le-iff-add prod-int-plus-eq*)

next

case *False*

then show *?thesis*

by *auto*

qed

61.11 Efficient folding over intervals

function *fold-atLeastAtMost-nat* **where**

[*simp del*]: $\text{fold-atLeastAtMost-nat } f a (b::\text{nat}) \text{ acc} =$

(*if a > b then acc else fold-atLeastAtMost-nat f (a+1) b (f a acc)*)

by *pat-completeness auto*

termination **by** (*relation measure* $(\lambda(-,a,b,-). \text{Suc } b - a)$) *auto*

lemma *fold-atLeastAtMost-nat*:

assumes *comp-fun-commute f*

shows $\text{fold-atLeastAtMost-nat } f a b \text{ acc} = \text{Finite-Set.fold } f \text{ acc } \{a..b\}$

using *assms*

proof (*induction f a b acc rule: fold-atLeastAtMost-nat.induct, goal-cases*)

case (*1 f a b acc*)

```

interpret comp-fun-commute f by fact
show ?case
proof (cases a > b)
  case True
    thus ?thesis by (subst fold-atLeastAtMost-nat.simps) auto
  next
    case False
    with 1 show ?thesis
      by (subst fold-atLeastAtMost-nat.simps)
        (auto simp: atLeastAtMost-insertL[symmetric] fold-fun-left-comm)
  qed
qed

```

```

lemma sum-atLeastAtMost-code:
  sum f {a..b} = fold-atLeastAtMost-nat (λa acc. f a + acc) a b 0
proof –
  have comp-fun-commute (λa. (+) (f a))
    by unfold-locales (auto simp: o-def add-ac)
  thus ?thesis
    by (simp add: sum.eq-fold fold-atLeastAtMost-nat o-def)
qed

```

```

lemma prod-atLeastAtMost-code:
  prod f {a..b} = fold-atLeastAtMost-nat (λa acc. f a * acc) a b 1
proof –
  have comp-fun-commute (λa. (*) (f a))
    by unfold-locales (auto simp: o-def mult-ac)
  thus ?thesis
    by (simp add: prod.eq-fold fold-atLeastAtMost-nat o-def)
qed

```

end

62 Decision Procedure for Presburger Arithmetic

```

theory Presburger
imports Groebner-Basis Set-Interval
keywords try0 :: diag
begin

```

```

ML-file <Tools/Qelim/qelim.ML>
ML-file <Tools/Qelim/cooper-procedure.ML>

```

62.1 The $-\infty$ and $+\infty$ Properties

```

lemma minf:
  
$$[\exists (z :: 'a::linorder). \forall x < z. P x = P' x; \exists z. \forall x < z. Q x = Q' x]$$


```


$$\begin{aligned} &\implies \exists z. \forall x < z. (P x \wedge Q x) = (P' x \wedge Q' x) \\ &[[\exists (z :: 'a::linorder). \forall x < z. P x = P' x; \exists z. \forall x < z. Q x = Q' x]] \\ &\implies \exists z. \forall x < z. (P x \vee Q x) = (P' x \vee Q' x) \\ &\exists (z :: 'a::\{linorder\}). \forall x < z. (x = t) = False \\ &\exists (z :: 'a::\{linorder\}). \forall x < z. (x \neq t) = True \\ &\exists (z :: 'a::\{linorder\}). \forall x < z. (x < t) = True \\ &\exists (z :: 'a::\{linorder\}). \forall x < z. (x \leq t) = True \\ &\exists (z :: 'a::\{linorder\}). \forall x < z. (x > t) = False \\ &\exists (z :: 'a::\{linorder\}). \forall x < z. (x \geq t) = False \\ &\exists z. \forall (x :: 'b::\{linorder, plus, Rings.dvd\}) < z. (d dvd x + s) = (d dvd x + s) \\ &\exists z. \forall (x :: 'b::\{linorder, plus, Rings.dvd\}) < z. (\neg d dvd x + s) = (\neg d dvd x + s) \\ &\exists z. \forall x < z. F = F \end{aligned}$$

proof *safe*

fix *z1 z2*

assume $\forall x < z1. P x = P' x$ **and** $\forall x < z2. Q x = Q' x$

then have $\forall x < \min z1 z2. (P x \wedge Q x) = (P' x \wedge Q' x)$

by *simp*

then show $\exists z. \forall x < z. (P x \wedge Q x) = (P' x \wedge Q' x)$

by *blast*

next

fix *z1 z2*

assume $\forall x < z1. P x = P' x$ **and** $\forall x < z2. Q x = Q' x$

then have $\forall x < \min z1 z2. (P x \vee Q x) = (P' x \vee Q' x)$

by *simp*

then show $\exists z. \forall x < z. (P x \vee Q x) = (P' x \vee Q' x)$

by *blast*

next

have $\forall x < t. x \leq t$

by *fastforce*

then show $\exists z. \forall x < z. (x \leq t) = True$

by *auto*

next

have $\forall x < t. \neg t < x$

by *fastforce*

then show $\exists z. \forall x < z. (t < x) = False$

by *auto*

next

have $\forall x < t. \neg t \leq x$

by *fastforce*

then show $\exists z. \forall x < z. (t \leq x) = False$

by *auto*

qed *auto*

lemma *pinf*:

$$[[\exists (z :: 'a::linorder). \forall x > z. P x = P' x; \exists z. \forall x > z. Q x = Q' x]]$$

$$\implies \exists z. \forall x > z. (P x \wedge Q x) = (P' x \wedge Q' x)$$

$$[[\exists (z :: 'a::linorder). \forall x > z. P x = P' x; \exists z. \forall x > z. Q x = Q' x]]$$

$$\implies \exists z. \forall x > z. (P x \vee Q x) = (P' x \vee Q' x)$$

$$\exists (z :: 'a::\{linorder\}). \forall x > z. (x = t) = False$$

$\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \neq t) = \text{True}$
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x < t) = \text{False}$
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \leq t) = \text{False}$
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x > t) = \text{True}$
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \geq t) = \text{True}$
 $\exists z. \forall (x :: 'b::\{\text{linorder, plus, Rings.dvd}\}) > z. (d \text{ dvd } x + s) = (d \text{ dvd } x + s)$
 $\exists z. \forall (x :: 'b::\{\text{linorder, plus, Rings.dvd}\}) > z. (\neg d \text{ dvd } x + s) = (\neg d \text{ dvd } x + s)$
 $\exists z. \forall x > z. F = F$

proof *safe*

fix *z1 z2*

assume $\forall x > z1. P x = P' x$ **and** $\forall x > z2. Q x = Q' x$

then have $\forall x > \max z1 z2. (P x \wedge Q x) = (P' x \wedge Q' x)$

by *simp*

then show $\exists z. \forall x > z. (P x \wedge Q x) = (P' x \wedge Q' x)$

by *blast*

next

fix *z1 z2*

assume $\forall x > z1. P x = P' x$ **and** $\forall x > z2. Q x = Q' x$

then have $\forall x > \max z1 z2. (P x \vee Q x) = (P' x \vee Q' x)$

by *simp*

then show $\exists z. \forall x > z. (P x \vee Q x) = (P' x \vee Q' x)$

by *blast*

next

have $\forall x > t. \neg x < t$

by *fastforce*

then show $\exists z. \forall x > z. x < t = \text{False}$

by *blast*

next

have $\forall x > t. \neg x \leq t$

by *fastforce*

then show $\exists z. \forall x > z. x \leq t = \text{False}$

by *blast*

next

have $\forall x > t. t \leq x$

by *fastforce*

then show $\exists z. \forall x > z. t \leq x = \text{True}$

by *blast*

qed *auto*

lemma *inf-period*:

$\llbracket \forall x k. P x = P (x - k * D); \forall x k. Q x = Q (x - k * D) \rrbracket$

$\implies \forall x k. (P x \wedge Q x) = (P (x - k * D) \wedge Q (x - k * D))$

$\llbracket \forall x k. P x = P (x - k * D); \forall x k. Q x = Q (x - k * D) \rrbracket$

$\implies \forall x k. (P x \vee Q x) = (P (x - k * D) \vee Q (x - k * D))$

$(d :: 'a::\{\text{comm-ring, Rings.dvd}\}) \text{ dvd } D \implies \forall x k. (d \text{ dvd } x + t) = (d \text{ dvd } (x - k * D) + t)$

$(d :: 'a::\{\text{comm-ring, Rings.dvd}\}) \text{ dvd } D \implies \forall x k. (\neg d \text{ dvd } x + t) = (\neg d \text{ dvd } (x - k * D) + t)$

$\forall x k. F = F$

apply (*auto elim!*: *dvdE simp add: algebra-simps*)
unfolding *mult.assoc [symmetric] distrib-right [symmetric] left-diff-distrib [symmetric]*
unfolding *dvd-def mult.commute [of d]*
by *auto*

62.2 The A and B sets

lemma *bset*:

$$\begin{aligned} & \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P x \longrightarrow P(x - D) ; \\ & \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q x \longrightarrow Q(x - D) \rrbracket \Longrightarrow \\ & \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x - D) \wedge Q(x - D)) \\ & \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P x \longrightarrow P(x - D) ; \\ & \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q x \longrightarrow Q(x - D) \rrbracket \Longrightarrow \\ & \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P x \vee Q x) \longrightarrow (P(x - D) \vee Q(x - D)) \\ & \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t)) \\ & \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t)) \\ & D > 0 \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t)) \\ & D > 0 \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t)) \\ & \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)) \\ & \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)) \\ & d \text{ dvd } D \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (d \text{ dvd } x+t) \longrightarrow (d \text{ dvd } (x - D) + t)) \\ & d \text{ dvd } D \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (\neg d \text{ dvd } x+t) \longrightarrow (\neg d \text{ dvd } (x - D) + t)) \\ & \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow F \longrightarrow F \end{aligned}$$

proof (*blast, blast*)

assume *dp: D > 0 and tB: t - 1 ∈ B*

show $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$

apply (*rule allI, rule impI,erule ballE[where x=1],erule ballE[where x=t - 1]*)

apply *algebra using dp tB by simp-all*

next

assume *dp: D > 0 and tB: t ∈ B*

show $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$

apply (*rule allI, rule impI,erule ballE[where x=D],erule ballE[where x=t]*)

apply *algebra*

using *dp tB by simp-all*

next

assume *dp: D > 0 thus* $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$ **by** *arith*

next

assume $dp: D > 0$ **thus** $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow$
 $(x - D \leq t)$ **by** *arith*
next
assume $dp: D > 0$ **and** $tB: t \in B$
{fix x **assume** $nob: \forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j$ **and** $g: x > t$ **and** $ng: \neg (x - D) > t$
hence $x - t \leq D$ **and** $1 \leq x - t$ **by** *simp+*
hence $\exists j \in \{1 \dots D\}. x - t = j$ **by** *auto*
hence $\exists j \in \{1 \dots D\}. x = t + j$ **by** (*simp add: algebra-simps*)
with nob tB **have** *False* **by** *simp*
thus $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)$ **by** *blast*
next
assume $dp: D > 0$ **and** $tB: t - 1 \in B$
{fix x **assume** $nob: \forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j$ **and** $g: x \geq t$ **and** $ng: \neg (x - D) \geq t$
hence $x - (t - 1) \leq D$ **and** $1 \leq x - (t - 1)$ **by** *simp+*
hence $\exists j \in \{1 \dots D\}. x - (t - 1) = j$ **by** *auto*
hence $\exists j \in \{1 \dots D\}. x = (t - 1) + j$ **by** (*simp add: algebra-simps*)
with nob tB **have** *False* **by** *simp*
thus $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)$ **by** *blast*
next
assume $d: d \text{ dvd } D$
{fix x **assume** $H: d \text{ dvd } x + t$ **with** d **have** $d \text{ dvd } (x - D) + t$ **by** *algebra*
thus $\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x - D) + t)$ **by** *simp*
next
assume $d: d \text{ dvd } D$
{fix x **assume** $H: \neg (d \text{ dvd } x + t)$ **with** d **have** $\neg d \text{ dvd } (x - D) + t$
by (*clarsimp simp add: dvd-def,erule-tac x = ka + k in allE,simp add: algebra-simps*)
thus $\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x - D) + t)$ **by** *auto*
qed *blast*

lemma *aset:*

$\llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ;$
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow$
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x + D) \wedge Q(x + D))$
 $\llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ;$
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow$
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \vee Q x) \longrightarrow (P(x + D) \vee Q(x + D))$
 $\llbracket D > 0; t + 1 \in A \rrbracket \Longrightarrow (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x = t) \longrightarrow (x + D = t))$
 $\llbracket D > 0; t \in A \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$
 $\llbracket D > 0; t \in A \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x < t) \longrightarrow (x + D < t))$

$\llbracket D > 0; t + 1 \in A \rrbracket \implies (\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t))$
 $D > 0 \implies (\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t))$
 $D > 0 \implies (\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t))$
 $d \text{ dvd } D \implies (\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (d \text{ dvd } x+t) \longrightarrow (d \text{ dvd } (x + D) + t))$
 $d \text{ dvd } D \implies (\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x+t) \longrightarrow (\neg d \text{ dvd } (x + D) + t))$
 $\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow F \longrightarrow F$
proof (*blast*, *blast*)
assume *dp*: $D > 0$ **and** *tA*: $t + 1 \in A$
show $(\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x = t) \longrightarrow (x + D = t))$
apply (*rule allI*, *rule impI*, *erule ballE*[**where** $x=1$], *erule ballE*[**where** $x=t + 1$])
using *dp tA* **by** *simp-all*
next
assume *dp*: $D > 0$ **and** *tA*: $t \in A$
show $(\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$
apply (*rule allI*, *rule impI*, *erule ballE*[**where** $x=D$], *erule ballE*[**where** $x=t$])
using *dp tA* **by** *simp-all*
next
assume *dp*: $D > 0$ **thus** $(\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t))$ **by** *arith*
next
assume *dp*: $D > 0$ **thus** $\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t)$ **by** *arith*
next
assume *dp*: $D > 0$ **and** *tA*: $t \in A$
{fix *x* **assume** *nob*: $\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j$ **and** *g*: $x < t$ **and** *ng*: $\neg (x + D) < t$
hence $t - x \leq D$ **and** $1 \leq t - x$ **by** *simp+*
hence $\exists j \in \{1 .. D\}. t - x = j$ **by** *auto*
hence $\exists j \in \{1 .. D\}. x = t - j$ **by** (*auto simp add: algebra-simps*)
with *nob tA* **have** *False* **by** *simp*
thus $\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x < t) \longrightarrow (x + D < t)$ **by** *blast*
next
assume *dp*: $D > 0$ **and** *tA*: $t + 1 \in A$
{fix *x* **assume** *nob*: $\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j$ **and** *g*: $x \leq t$ **and** *ng*: $\neg (x + D) \leq t$
hence $(t + 1) - x \leq D$ **and** $1 \leq (t + 1) - x$ **by** (*simp-all add: algebra-simps*)
hence $\exists j \in \{1 .. D\}. (t + 1) - x = j$ **by** *auto*
hence $\exists j \in \{1 .. D\}. x = (t + 1) - j$ **by** (*auto simp add: algebra-simps*)
with *nob tA* **have** *False* **by** *simp*
thus $\forall x. (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t)$ **by** *blast*
next
assume *d*: $d \text{ dvd } D$
have $\bigwedge x. d \text{ dvd } x + t \implies d \text{ dvd } x + D + t$

```

proof –
  fix  $x$ 
  assume  $H: d \text{ dvd } x + t$ 
  then obtain  $ka$  where  $x + t = d * ka$ 
    unfolding  $dvd\text{-def}$  by  $blast$ 
  moreover from  $d$  obtain  $k$  where  $*:D = d * k$ 
    unfolding  $dvd\text{-def}$  by  $blast$ 
  ultimately have  $x + d * k + t = d * (ka + k)$ 
    by  $(simp \text{ add: algebra-simps})$ 
  then show  $d \text{ dvd } (x + D) + t$ 
    using  $*$  unfolding  $dvd\text{-def}$  by  $blast$ 
qed
  thus  $\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (d \text{ dvd } x+t) \longrightarrow (d \text{ dvd } (x + D) + t)$  by  $simp$ 
next
  assume  $d: d \text{ dvd } D$ 
  {fix  $x$  assume  $H: \neg(d \text{ dvd } x + t)$  with  $d$  have  $\neg d \text{ dvd } (x + D) + t$ 
    by  $(clarsimp \text{ simp add: dvd-def,erule-tac } x = ka - k \text{ in allE,simp add: algebra-simps)}$ 
  }
  thus  $\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x+t) \longrightarrow (\neg d \text{ dvd } (x + D) + t)$  by  $auto$ 
qed  $blast$ 

```

62.3 Cooper’s Theorem $-\infty$ and $+\infty$ Version

62.3.1 First some trivial facts about periodic sets or predicates

lemma *periodic-finite-ex*:

```

assumes  $dpos: (0::int) < d$  and  $modd: \forall x k. P x = P(x - k*d)$ 
shows  $(\exists x. P x) = (\exists j \in \{1..d\}. P j)$ 
(is ?LHS = ?RHS)

```

proof

assume $?LHS$

then obtain x **where** $P: P x ..$

have $x \text{ mod } d = x - (x \text{ div } d)*d$ **by** $(simp \text{ add:mult-div-mod-eq [symmetric] ac-simps eq-diff-eq})$

hence $P \text{ mod}: P x = P(x \text{ mod } d)$ **using** $modd$ **by** $simp$

show $?RHS$

proof $(cases)$

assume $x \text{ mod } d = 0$

hence $P 0$ **using** $P \text{ mod}$ **by** $simp$

moreover have $P 0 = P(0 - (-1)*d)$ **using** $modd$ **by** $blast$

ultimately have $P d$ **by** $simp$

moreover have $d \in \{1..d\}$ **using** $dpos$ **by** $simp$

ultimately show $?RHS ..$

next

assume $not0: x \text{ mod } d \neq 0$

have $P(x \text{ mod } d)$ **using** $dpos \text{ } P \text{ mod}$ **by** $simp$

moreover have $x \text{ mod } d \in \{1..d\}$

proof –

```

    from dpos have  $0 \leq x \bmod d$  by(rule pos-mod-sign)
    moreover from dpos have  $x \bmod d < d$  by(rule pos-mod-bound)
    ultimately show ?thesis using not0 by simp
  qed
  ultimately show ?RHS ..
  qed
  qed auto

```

62.3.2 The $-\infty$ Version

```

lemma decr-lemma:  $0 < (d::int) \implies x - (|x - z| + 1) * d < z$ 
  by (induct rule: int-gr-induct) (simp-all add: int-distrib)

```

```

lemma incr-lemma:  $0 < (d::int) \implies z < x + (|x - z| + 1) * d$ 
  by (induct rule: int-gr-induct) (simp-all add: int-distrib)

```

lemma decr-mult-lemma:

```

  assumes dpos:  $(0::int) < d$  and minus:  $\forall x. P x \longrightarrow P(x - d)$  and knneg:  $0$ 
  <=  $k$ 

```

```

  shows  $\forall x. P x \longrightarrow P(x - k*d)$ 

```

```

  using knneg

```

```

  proof (induct rule:int-ge-induct)

```

```

    case base thus ?case by simp

```

```

  next

```

```

    case (step  $i$ )

```

```

    {fix  $x$ 

```

```

      have  $P x \longrightarrow P(x - i * d)$  using step.hyps by blast

```

```

      also have  $\dots \longrightarrow P(x - (i + 1) * d)$  using minus[THEN spec, of  $x - i * d$ ]

```

```

        by (simp add: algebra-simps)

```

```

      ultimately have  $P x \longrightarrow P(x - (i + 1) * d)$  by blast}

```

```

    thus ?case ..

```

```

  qed

```

lemma minusinfinity:

```

  assumes dpos:  $0 < d$  and

```

```

  P1eqP1:  $\forall x k. P1 x = P1(x - k*d)$  and ePeqP1:  $\exists z::int. \forall x. x < z \longrightarrow (P x$ 
  =  $P1 x)$ 

```

```

  shows  $(\exists x. P1 x) \longrightarrow (\exists x. P x)$ 

```

```

  proof

```

```

    assume eP1:  $\exists x. P1 x$ 

```

```

    then obtain  $x$  where  $P1: P1 x ..$ 

```

```

    from ePeqP1 obtain  $z$  where P1eqP:  $\forall x. x < z \longrightarrow (P x = P1 x) ..$ 

```

```

    let ? $w$  =  $x - (|x - z| + 1) * d$ 

```

```

    from dpos have  $w: ?w < z$  by(rule decr-lemma)

```

```

    have  $P1 x = P1 ?w$  using P1eqP1 by blast

```

```

    also have  $\dots = P(?w)$  using w P1eqP by blast

```

```

    finally have  $P ?w$  using P1 by blast

```

```

    thus  $\exists x. P x ..$ 

```

```

  qed

```

lemma *cpmi*:

assumes *dp*: $0 < D$ **and** *p1*: $\exists z. \forall x < z. P\ x = P'\ x$
and *nb*: $\forall x. (\forall j \in \{1..D\}. \forall (b::int) \in B. x \neq b+j) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$
and *pd*: $\forall x\ k. P'\ x = P'\ (x - k * D)$
shows $(\exists x. P\ x) = ((\exists j \in \{1..D\}. P'\ j) \vee (\exists j \in \{1..D\}. \exists b \in B. P\ (b+j)))$
(is $?L = (?R1 \vee ?R2)$ **)**

proof –

{assume $?R2$ **hence** $?L$ **by** *blast***}**
moreover
{assume $H: ?R1$ **hence** $?L$ **using** *minusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp***}**
moreover
{ fix x
assume $P: P\ x$ **and** $H: \neg ?R2$
{fix y **assume** $\neg (\exists j \in \{1..D\}. \exists b \in B. P\ (b + j))$ **and** $P: P\ y$
hence $\neg (\exists (j::int) \in \{1..D\}. \exists (b::int) \in B. y = b+j)$ **by** *auto*
with $nb\ P$ **have** $P\ (y - D)$ **by** *auto* **}**
hence $\forall x. \neg (\exists (j::int) \in \{1..D\}. \exists (b::int) \in B. P\ (b+j)) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$ **by** *blast*
with $H\ P$ **have** $th: \forall x. P\ x \longrightarrow P\ (x - D)$ **by** *auto*
from $p1$ **obtain** z **where** $z: \forall x. x < z \longrightarrow (P\ x = P'\ x)$ **by** *blast*
let $?y = x - (|x - z| + 1) * D$
have $zp: 0 \leq (|x - z| + 1)$ **by** *arith*
from dp **have** $yz: ?y < z$ **using** *decr-lemma*[*OF dp*] **by** *simp*
from z [*rule-format*, *OF yz*] *decr-mult-lemma*[*OF dp th zp*, *rule-format*, *OF P*]
have $th2: P'\ ?y$ **by** *auto*
with *periodic-finite-ex*[*OF dp pd*]
have $?R1$ **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

62.3.3 The $+\infty$ Version

lemma *plusinfinity*:

assumes *dpos*: $(0::int) < d$ **and**
 $P1eqP1: \forall x\ k. P'\ x = P'\ (x - k * d)$ **and** $eP1eqP1: \exists z. \forall x > z. P\ x = P'\ x$
shows $(\exists x. P'\ x) \longrightarrow (\exists x. P\ x)$

proof

assume $eP1: \exists x. P'\ x$
then obtain x **where** $P1: P'\ x ..$
from $eP1eqP1$ **obtain** z **where** $P1eqP: \forall x > z. P\ x = P'\ x ..$
let $?w' = x + (|x - z| + 1) * d$
let $?w = x - (-(|x - z| + 1)) * d$
have ww' [*simp*]: $?w = ?w'$ **by** (*simp add: algebra-simps*)
from *dpos* **have** $w: ?w > z$ **by** (*simp only: ww' incr-lemma*)
hence $P'\ x = P'\ ?w$ **using** $P1eqP1$ **by** *blast*
also have $... = P\ (?w)$ **using** $w\ P1eqP$ **by** *blast*
finally have $P\ ?w$ **using** $P1$ **by** *blast*

thus $\exists x. P x ..$
qed

lemma *incr-mult-lemma*:

assumes *dpos*: $(0::int) < d$ and *plus*: $\forall x::int. P x \longrightarrow P(x + d)$ and *knneg*: $0 \leq k$
shows $\forall x. P x \longrightarrow P(x + k*d)$
using *knneg*
proof (*induct rule:int-ge-induct*)
case *base* thus ?*case* by *simp*
next
case (*step i*)
{fix *x*
have $P x \longrightarrow P(x + i * d)$ using *step.hyps* by *blast*
also have $\dots \longrightarrow P(x + (i + 1) * d)$ using *plus*[*THEN spec, of x + i * d*]
by (*simp add:int-distrib ac-simps*)
ultimately have $P x \longrightarrow P(x + (i + 1) * d)$ by *blast* }
thus ?*case* ..
qed

lemma *cppi*:

assumes *dp*: $0 < D$ and *p1*: $\exists z. \forall x > z. P x = P' x$
and *nb*: $\forall x. (\forall j \in \{1..D\}. \forall (b::int) \in A. x \neq b - j) \longrightarrow P(x) \longrightarrow P(x + D)$
and *pd*: $\forall x k. P' x = P'(x - k*D)$
shows $(\exists x. P x) = ((\exists j \in \{1..D\}. P' j) \vee (\exists j \in \{1..D\}. \exists b \in A. P(b - j)))$
(is ?*L* = (?*R1* \vee ?*R2*))
proof –
{assume ?*R2* hence ?*L* by *blast* }
moreover
{assume *H*: ?*R1* hence ?*L* using *plusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] by *simp* }
moreover
{ fix *x*
assume *P*: $P x$ and *H*: $\neg ?R2$
{fix *y* assume $\neg (\exists j \in \{1..D\}. \exists b \in A. P(b - j))$ and *P*: $P y$
hence $\neg (\exists (j::int) \in \{1..D\}. \exists (b::int) \in A. y = b - j)$ by *auto*
with *nb P* have $P(y + D)$ by *auto* }
hence $\forall x. \neg (\exists (j::int) \in \{1..D\}. \exists (b::int) \in A. P(b - j)) \longrightarrow P(x) \longrightarrow P(x + D)$ by *blast*
with *H P* have *th*: $\forall x. P x \longrightarrow P(x + D)$ by *auto*
from *p1* obtain *z* where $z: \forall x. x > z \longrightarrow (P x = P' x)$ by *blast*
let ?*y* = $x + (|x - z| + 1)*D$
have *zp*: $0 \leq (|x - z| + 1)$ by *arith*
from *dp* have *yz*: ?*y* > *z* using *incr-lemma*[*OF dp*] by *simp*
from *z*[*rule-format, OF yz*] *incr-mult-lemma*[*OF dp th zp, rule-format, OF P*]
have *th2*: $P' ?y$ by *auto*
with *periodic-finite-ex*[*OF dp pd*]
have ?*R1* by *blast* }
ultimately show ?*thesis* by *blast*

qed

lemma *simp-from-to*: $\{i..j::int\} = (if\ j < i\ then\ \{\}\ else\ insert\ i\ \{i+1..j\})$
apply(*simp add:atLeastAtMost-def atLeast-def atMost-def*)
apply(*fastforce*)
done

theorem *unity-coeff-ex*: $(\exists (x::'a::\{semiring-0,Rings.dvd\}). P\ (l * x)) \equiv (\exists x. l\ dvd\ (x + 0) \wedge P\ x)$
unfolding *dvd-def* **by** (*rule eq-reflection, rule iffI*) *auto*

lemma *zdvd-mono*:
fixes $k\ m\ t :: int$
assumes $k \neq 0$
shows $m\ dvd\ t \equiv k * m\ dvd\ k * t$
using *assms* **by** *simp*

lemma *uminus-dvd-conv*:
fixes $d\ t :: int$
shows $d\ dvd\ t \equiv -\ d\ dvd\ t$ **and** $d\ dvd\ t \equiv d\ dvd\ -\ t$
by *simp-all*

Theorems for transforming predicates on nat to predicates on *int*

lemma *zdiff-int-split*: $P\ (int\ (x - y)) = ((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
by (*cases y \leq x*) (*simp-all add: of-nat-diff*)

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*:
 $\llbracket x = x'; 0 \leq x' \implies P = P' \rrbracket \implies (0 \leq (x::int) \longrightarrow P) = (0 \leq x' \longrightarrow P')$
by *simp*

theorem *conj-le-cong*:
 $\llbracket x = x'; 0 \leq x' \implies P = P' \rrbracket \implies (0 \leq (x::int) \wedge P) = (0 \leq x' \wedge P')$
by (*simp cong: conj-cong*)

ML-file $\langle Tools/Qelim/cooper.ML \rangle$

method-setup *presburger* = \langle
let
fun *keyword* $k = Scan.lift\ (Args.\$\$\$ k -- Args.colon) >> K\ ()$
fun *simple-keyword* $k = Scan.lift\ (Args.\$\$\$ k) >> K\ ()$
val *addN* = *add*
val *delN* = *del*
val *elimN* = *elim*
val *any-keyword* = *keyword addN || keyword delN || simple-keyword elimN*
val *thms* = *Scan.repeats (Scan.unless any-keyword Attrib.multi-thm)*
in

```

Scan.optional (simple-keyword elimN >> K false) true --
Scan.optional (keyword addN |-- thms) [] --
Scan.optional (keyword delN |-- thms) [] >>
(fn ((elim, add-ths), del-ths) => fn ctxt =>
  SIMPLE-METHOD' (Cooper.tac elim add-ths del-ths ctxt))
end
› Cooper's algorithm for Presburger arithmetic

declare mod-eq-0-iff-dvd [presburger]
declare mod-by-Suc-0 [presburger]
declare mod-0 [presburger]
declare mod-by-1 [presburger]
declare mod-self [presburger]
declare div-by-0 [presburger]
declare mod-by-0 [presburger]
declare mod-div-trivial [presburger]
declare mult-div-mod-eq [presburger]
declare div-mult-mod-eq [presburger]
declare mod-mult-self1 [presburger]
declare mod-mult-self2 [presburger]
declare mod2-Suc-Suc [presburger]
declare not-mod-2-eq-0-eq-1 [presburger]
declare nat-zero-less-power-iff [presburger]

lemma [presburger, algebra]: m mod 2 = (1::nat)  $\longleftrightarrow$   $\neg$  2 dvd m by presburger
lemma [presburger, algebra]: m mod 2 = Suc 0  $\longleftrightarrow$   $\neg$  2 dvd m by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = (1::nat)  $\longleftrightarrow$   $\neg$  2 dvd m by
presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = Suc 0  $\longleftrightarrow$   $\neg$  2 dvd m by
presburger
lemma [presburger, algebra]: m mod 2 = (1::int)  $\longleftrightarrow$   $\neg$  2 dvd m by presburger

context semiring-parity
begin

declare even-mult-iff [presburger]

declare even-power [presburger]

lemma [presburger]:
  even (a + b)  $\longleftrightarrow$  even a  $\wedge$  even b  $\vee$  odd a  $\wedge$  odd b
  by auto

end

context ring-parity
begin

declare even-minus [presburger]

```

end

context *linordered-idom*
begin

declare *zero-le-power-eq* [*presburger*]

declare *zero-less-power-eq* [*presburger*]

declare *power-less-zero-eq* [*presburger*]

declare *power-le-zero-eq* [*presburger*]

end

declare *even-Suc* [*presburger*]

lemma [*presburger*]:
 $Suc\ n\ div\ Suc\ (Suc\ 0) = n\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ n$
by *presburger*

declare *even-diff-nat* [*presburger*]

lemma [*presburger*]:
fixes $k :: int$
shows $(k + 1)\ div\ 2 = k\ div\ 2 \longleftrightarrow even\ k$
by *presburger*

lemma [*presburger*]:
fixes $k :: int$
shows $(k + 1)\ div\ 2 = k\ div\ 2 + 1 \longleftrightarrow odd\ k$
by *presburger*

lemma [*presburger*]:
 $even\ n \longleftrightarrow even\ (int\ n)$
by *simp*

62.4 Nice facts about division by $4 :: 'a$

lemma *even-even-mod-4-iff*:
 $even\ (n :: nat) \longleftrightarrow even\ (n\ mod\ 4)$
by *presburger*

lemma *odd-mod-4-div-2*:
 $n\ mod\ 4 = (3 :: nat) \implies odd\ ((n - Suc\ 0)\ div\ 2)$
by *presburger*

lemma *even-mod-4-div-2*:

$n \text{ mod } 4 = \text{Suc } 0 \implies \text{even } ((n - \text{Suc } 0) \text{ div } 2)$
 by *presburger*

62.5 Try0

ML-file $\langle \text{Tools/try0.ML} \rangle$

end

63 Misc lemmas on division, to be sorted out finally

theory *Divides*
imports *Parity*
begin

class *unique-euclidean-semiring-numeral* = *unique-euclidean-semiring-with-nat* +
linordered-semidom +

assumes *div-less* [*no-atp*]: $0 \leq a \implies a < b \implies a \text{ div } b = 0$
and *mod-less* [*no-atp*]: $0 \leq a \implies a < b \implies a \text{ mod } b = a$
and *div-positive* [*no-atp*]: $0 < b \implies b \leq a \implies a \text{ div } b > 0$
and *mod-less-eq-dividend* [*no-atp*]: $0 \leq a \implies a \text{ mod } b \leq a$
and *pos-mod-bound* [*no-atp*]: $0 < b \implies a \text{ mod } b < b$
and *pos-mod-sign* [*no-atp*]: $0 < b \implies 0 \leq a \text{ mod } b$
and *mod-mult2-eq* [*no-atp*]: $0 \leq c \implies a \text{ mod } (b * c) = b * (a \text{ div } b \text{ mod } c) +$
 $a \text{ mod } b$
and *div-mult2-eq* [*no-atp*]: $0 \leq c \implies a \text{ div } (b * c) = a \text{ div } b \text{ div } c$
assumes *discrete* [*no-atp*]: $a < b \iff a + 1 \leq b$

hide-fact (**open**) *div-less mod-less mod-less-eq-dividend mod-mult2-eq div-mult2-eq*

context *unique-euclidean-semiring-numeral*
begin

context
begin

lemma *divmod-digit-1* [*no-atp*]:

assumes $0 \leq a$ $0 < b$ **and** $b \leq a \text{ mod } (2 * b)$
shows $2 * (a \text{ div } (2 * b)) + 1 = a \text{ div } b$ (**is** ?*P*)
and $a \text{ mod } (2 * b) - b = a \text{ mod } b$ (**is** ?*Q*)

proof –

from *assms mod-less-eq-dividend* [*of a 2 * b*] **have** $b \leq a$
by (*auto intro: trans*)

with $\langle 0 < b \rangle$ **have** $0 < a \text{ div } b$ **by** (*auto intro: div-positive*)

then have [*simp*]: $1 \leq a \text{ div } b$ **by** (*simp add: discrete*)

with $\langle 0 < b \rangle$ **have** *mod-less*: $a \text{ mod } b < b$ **by** (*simp add: pos-mod-bound*)

define *w* **where** $w = a \text{ div } b \text{ mod } 2$

```

then have w-exhaust:  $w = 0 \vee w = 1$  by auto
have mod-w:  $a \bmod (2 * b) = a \bmod b + b * w$ 
  by (simp add: w-def mod-mult2-eq ac-simps)
from assms w-exhaust have  $w = 1$ 
  using mod-less by (auto simp add: mod-w)
with mod-w have mod:  $a \bmod (2 * b) = a \bmod b + b$  by simp
have  $2 * (a \operatorname{div} (2 * b)) = a \operatorname{div} b - w$ 
  by (simp add: w-def div-mult2-eq minus-mod-eq-mult-div ac-simps)
with  $\langle w = 1 \rangle$  have div:  $2 * (a \operatorname{div} (2 * b)) = a \operatorname{div} b - 1$  by simp
then show ?P and ?Q
  by (simp-all add: div mod add-implies-diff [symmetric])
qed

```

```

lemma divmod-digit-0 [no-atp]:
  assumes  $0 < b$  and  $a \bmod (2 * b) < b$ 
  shows  $2 * (a \operatorname{div} (2 * b)) = a \operatorname{div} b$  (is ?P)
    and  $a \bmod (2 * b) = a \bmod b$  (is ?Q)
proof -
  define w where  $w = a \operatorname{div} b \bmod 2$ 
  then have w-exhaust:  $w = 0 \vee w = 1$  by auto
  have mod-w:  $a \bmod (2 * b) = a \bmod b + b * w$ 
    by (simp add: w-def mod-mult2-eq ac-simps)
  moreover have  $b \leq a \bmod b + b$ 
  proof -
    from  $\langle 0 < b \rangle$  pos-mod-sign have  $0 \leq a \bmod b$  by blast
    then have  $0 + b \leq a \bmod b + b$  by (rule add-right-mono)
    then show ?thesis by simp
  qed
  moreover note assms w-exhaust
  ultimately have  $w = 0$  by auto
  with mod-w have mod:  $a \bmod (2 * b) = a \bmod b$  by simp
  have  $2 * (a \operatorname{div} (2 * b)) = a \operatorname{div} b - w$ 
    by (simp add: w-def div-mult2-eq minus-mod-eq-mult-div ac-simps)
  with  $\langle w = 0 \rangle$  have div:  $2 * (a \operatorname{div} (2 * b)) = a \operatorname{div} b$  by simp
  then show ?P and ?Q
    by (simp-all add: div mod)
qed

```

lemma mod-double-modulus: — This is actually useful and should be transferred to a suitable type class

```

assumes  $m > 0$   $x \geq 0$ 
shows  $x \bmod (2 * m) = x \bmod m \vee x \bmod (2 * m) = x \bmod m + m$ 
proof (cases  $x \bmod (2 * m) < m$ )
  case True
  thus ?thesis using assms using divmod-digit-0(2)[of m x] by auto
next
  case False
  hence *:  $x \bmod (2 * m) - m = x \bmod m$ 
    using assms by (intro divmod-digit-1) auto

```

```

hence  $x \bmod (2 * m) = x \bmod m + m$ 
  by (subst * [symmetric], subst le-add-diff-inverse2) (use False in auto)
thus ?thesis by simp
qed

end

end

instance nat :: unique-euclidean-semiring-numeral
  by standard
  (auto simp add: div-greater-zero-iff div-mult2-eq mod-mult2-eq)

instance int :: unique-euclidean-semiring-numeral
  by standard (auto intro: zmod-le-nonneg-dividend simp add:
    pos-imp-zdiv-pos-iff zmod-zmult2-eq zdiv-zmult2-eq)

lemma zmod-eq-0D [dest!]:  $\exists q. m = d * q$  if  $m \bmod d = 0$  for  $m d :: int$ 
  using that by auto

lemma div-geq [no-atp]:  $m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$  if  $0 < n$  and  $\neg m < n$ 
for m n :: nat
  by (rule le-div-geq) (use that in <simp-all add: not-less>)

lemma mod-geq [no-atp]:  $m \bmod n = (m - n) \bmod n$  if  $\neg m < n$  for  $m n :: nat$ 
by (rule le-mod-geq) (use that in <simp add: not-less>)

lemma mod-eq-0D [no-atp]:  $\exists q. m = d * q$  if  $m \bmod d = 0$  for  $m d :: nat$ 
using that by (auto simp add: mod-eq-0-iff-dvd)

lemma pos-mod-conj [no-atp]:  $0 < b \implies 0 \leq a \bmod b \wedge a \bmod b < b$  for  $a b :: int$ 
by simp

lemma neg-mod-conj [no-atp]:  $b < 0 \implies a \bmod b \leq 0 \wedge b < a \bmod b$  for  $a b :: int$ 
by simp

lemma zmod-eq-0-iff [no-atp]:  $m \bmod d = 0 \iff (\exists q. m = d * q)$  for  $m d :: int$ 
by (auto simp add: mod-eq-0-iff-dvd)

lemma div-positive-int [no-atp]:
  k div l > 0 if k  $\geq$  l and l > 0 for k l :: int
  using that by (simp add: nonneg1-imp-zdiv-pos-iff)

code-identifier
  code-module Divides  $\rightarrow$  (SML) Arith and (OCaml) Arith and (Haskell) Arith

```

end

64 Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2

```
theory SMT
  imports Divides Numeral-Simprocs
  keywords smt-status :: diag
begin
```

64.1 A skolemization tactic and proof method

lemma choices:

$$\begin{aligned} & \bigwedge Q. \forall x. \exists y ya. Q x y ya \implies \exists f fa. \forall x. Q x (f x) (fa x) \\ & \bigwedge Q. \forall x. \exists y ya yb. Q x y ya yb \implies \exists f fa fb. \forall x. Q x (f x) (fa x) (fb x) \\ & \bigwedge Q. \forall x. \exists y ya yb yc. Q x y ya yb yc \implies \exists f fa fb fc. \forall x. Q x (f x) (fa x) (fb x) \\ & \quad (fc x) \\ & \bigwedge Q. \forall x. \exists y ya yb yc yd. Q x y ya yb yc yd \implies \\ & \quad \exists f fa fb fc fd. \forall x. Q x (f x) (fa x) (fb x) (fc x) (fd x) \\ & \bigwedge Q. \forall x. \exists y ya yb yc yd ye. Q x y ya yb yc yd ye \implies \\ & \quad \exists f fa fb fc fd fe. \forall x. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) \\ & \bigwedge Q. \forall x. \exists y ya yb yc yd ye yf. Q x y ya yb yc yd ye yf \implies \\ & \quad \exists f fa fb fc fd fe ff. \forall x. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) (ff x) \\ & \bigwedge Q. \forall x. \exists y ya yb yc yd ye yf yg. Q x y ya yb yc yd ye yf yg \implies \\ & \quad \exists f fa fb fc fd fe ff fg. \forall x. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) (ff x) (fg x) \end{aligned}$$

by metis+

lemma bchoices:

$$\begin{aligned} & \bigwedge Q. \forall x \in S. \exists y ya. Q x y ya \implies \exists f fa. \forall x \in S. Q x (f x) (fa x) \\ & \bigwedge Q. \forall x \in S. \exists y ya yb. Q x y ya yb \implies \exists f fa fb. \forall x \in S. Q x (f x) (fa x) (fb x) \\ & \bigwedge Q. \forall x \in S. \exists y ya yb yc. Q x y ya yb yc \implies \exists f fa fb fc. \forall x \in S. Q x (f x) (fa \\ & \quad x) (fb x) (fc x) \\ & \bigwedge Q. \forall x \in S. \exists y ya yb yc yd. Q x y ya yb yc yd \implies \\ & \quad \exists f fa fb fc fd. \forall x \in S. Q x (f x) (fa x) (fb x) (fc x) (fd x) \\ & \bigwedge Q. \forall x \in S. \exists y ya yb yc yd ye. Q x y ya yb yc yd ye \implies \\ & \quad \exists f fa fb fc fd fe. \forall x \in S. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) \\ & \bigwedge Q. \forall x \in S. \exists y ya yb yc yd ye yf. Q x y ya yb yc yd ye yf \implies \\ & \quad \exists f fa fb fc fd fe ff. \forall x \in S. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) (ff x) \\ & \bigwedge Q. \forall x \in S. \exists y ya yb yc yd ye yf yg. Q x y ya yb yc yd ye yf yg \implies \\ & \quad \exists f fa fb fc fd fe ff fg. \forall x \in S. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) (ff x) \\ & \quad (fg x) \end{aligned}$$

by metis+

ML ‹

fun moura-tac ctxt =

Atomize-Elim.atomize-elim-tac ctxt THEN'

SELECT-GOAL (Clasimp.auto-tac (ctxt addSIs @ {thms choice choices bchoice bchoices}) THEN


```

    ALLGOALS (Metis-Tactic.metis-tac (take 1 ATP-Proof-Reconstruct.partial-type-encs)
              ATP-Proof-Reconstruct.default-metis-lam-trans ctxt [] ORELSE'
              blast-tac ctxt))
  >

```

```

method-setup moura = <
  Scan.succeed (SIMPLE-METHOD' o moura-tac)
  > solve skolemization goals, especially those arising from Z3 proofs

```

```

hide-fact (open) choices bchoices

```

64.2 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by "pat") triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by "nopat") inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.

```

typedecl 'a symb-list

```

```

consts

```

```

  Symb-Nil :: 'a symb-list
  Symb-Cons :: 'a ⇒ 'a symb-list ⇒ 'a symb-list

```

```

typedecl pattern

```

```

consts

```

```

  pat :: 'a ⇒ pattern
  nopat :: 'a ⇒ pattern

```

```

definition trigger :: pattern symb-list symb-list ⇒ bool ⇒ bool where
  trigger - P = P

```

64.3 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.

```

definition fun-app :: 'a ⇒ 'a where fun-app f = f

```

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

lemmas *array-rules* = *ext fun-upd-apply fun-upd-same fun-upd-other fun-upd-upd fun-app-def*

64.4 Normalization

lemma *case-bool-if[abs-def]*: *case-bool x y P = (if P then x else y)*
by *simp*

lemmas *Ex1-def-raw = Ex1-def[abs-def]*
lemmas *Ball-def-raw = Ball-def[abs-def]*
lemmas *Bex-def-raw = Bex-def[abs-def]*
lemmas *abs-if-raw = abs-if[abs-def]*
lemmas *min-def-raw = min-def[abs-def]*
lemmas *max-def-raw = max-def[abs-def]*

lemma *nat-zero-as-int*:
 $0 = \text{nat } 0$
by *simp*

lemma *nat-one-as-int*:
 $1 = \text{nat } 1$
by *simp*

lemma *nat-numeral-as-int*: *numeral = ($\lambda i. \text{nat } (\text{numeral } i)$)* **by** *simp*
lemma *nat-less-as-int*: $(<) = (\lambda a b. \text{int } a < \text{int } b)$ **by** *simp*
lemma *nat-leq-as-int*: $(<=) = (\lambda a b. \text{int } a \leq \text{int } b)$ **by** *simp*
lemma *Suc-as-int*: *Suc = ($\lambda a. \text{nat } (\text{int } a + 1)$)* **by** (*rule ext*) *simp*
lemma *nat-plus-as-int*: $(+)$ = $(\lambda a b. \text{nat } (\text{int } a + \text{int } b))$ **by** (*rule ext*) $+$ *simp*
lemma *nat-minus-as-int*: $(-)$ = $(\lambda a b. \text{nat } (\text{int } a - \text{int } b))$ **by** (*rule ext*) $+$ *simp*
lemma *nat-times-as-int*: $(*)$ = $(\lambda a b. \text{nat } (\text{int } a * \text{int } b))$ **by** (*simp add: nat-mult-distrib*)
lemma *nat-div-as-int*: (div) = $(\lambda a b. \text{nat } (\text{int } a \text{ div } \text{int } b))$ **by** (*simp add: nat-div-distrib*)
lemma *nat-mod-as-int*: (mod) = $(\lambda a b. \text{nat } (\text{int } a \text{ mod } \text{int } b))$ **by** (*simp add: nat-mod-distrib*)

lemma *int-Suc*: *int (Suc n) = int n + 1* **by** *simp*

lemma *int-plus*: *int (n + m) = int n + int m* **by** (*rule of-nat-add*)

lemma *int-minus*: *int (n - m) = int (nat (int n - int m))* **by** *auto*

lemma *nat-int-comparison*:
fixes *a b :: nat*
shows $(a = b) = (\text{int } a = \text{int } b)$
and $(a < b) = (\text{int } a < \text{int } b)$
and $(a \leq b) = (\text{int } a \leq \text{int } b)$
by *simp-all*

lemma *int-ops*:
fixes *a b :: nat*
shows *int 0 = 0*
and *int 1 = 1*

```

and int (numeral n) = numeral n
and int (Suc a) = int a + 1
and int (a + b) = int a + int b
and int (a - b) = (if int a < int b then 0 else int a - int b)
and int (a * b) = int a * int b
and int (a div b) = int a div int b
and int (a mod b) = int a mod int b
by (auto intro: zdiv-int zmod-int)

```

```

lemma int-if:
  fixes a b :: nat
  shows int (if P then a else b) = (if P then int a else int b)
  by simp

```

64.5 Integer division and modulo for Z3

The following Z3-inspired definitions are overspecified for the case where $l = 0$. This Schönheitsfehler is corrected in the *div-as-z3div* and *mod-as-z3mod* theorems.

```

definition z3div :: int  $\Rightarrow$  int  $\Rightarrow$  int where
  z3div k l = (if  $l \geq 0$  then k div l else - (k div - l))

```

```

definition z3mod :: int  $\Rightarrow$  int  $\Rightarrow$  int where
  z3mod k l = k mod (if  $l \geq 0$  then l else - l)

```

```

lemma div-as-z3div:
   $\forall k l. k \text{ div } l = (\text{if } l = 0 \text{ then } 0 \text{ else if } l > 0 \text{ then } z3div \ k \ l \text{ else } z3div \ (-k) \ (-l))$ 
  by (simp add: z3div-def)

```

```

lemma mod-as-z3mod:
   $\forall k l. k \text{ mod } l = (\text{if } l = 0 \text{ then } k \text{ else if } l > 0 \text{ then } z3mod \ k \ l \text{ else } - z3mod \ (-k) \ (-l))$ 
  by (simp add: z3mod-def)

```

64.6 Extra theorems for veriT reconstruction

```

lemma verit-sko-forall:  $\langle (\forall x. P \ x) \longleftrightarrow P \ (\text{SOME } x. \neg P \ x) \rangle$ 
  using someI[of  $\langle \lambda x. \neg P \ x \rangle$ ]
  by auto

```

```

lemma verit-sko-forall':  $\langle P \ (\text{SOME } x. \neg P \ x) = A \implies (\forall x. P \ x) = A \rangle$ 
  by (subst verit-sko-forall)

```

```

lemma verit-sko-forall'':  $\langle B = A \implies (\text{SOME } x. P \ x) = A \equiv (\text{SOME } x. P \ x) = B \rangle$ 
  by auto

```

```

lemma verit-sko-forall-indirect:  $\langle x = (\text{SOME } x. \neg P \ x) \implies (\forall x. P \ x) \longleftrightarrow P \ x \rangle$ 
  using someI[of  $\langle \lambda x. \neg P \ x \rangle$ ]

```

by *auto*

lemma *verit-sko-forall-indirect2*:

$\langle x = (\text{SOME } x. \neg P x) \implies (\bigwedge x :: 'a. (P x = P' x)) \implies (\forall x. P' x) \longleftrightarrow P x \rangle$

using *someI*[of $\langle \lambda x. \neg P x \rangle$]

by *auto*

lemma *verit-sko-ex*: $\langle (\exists x. P x) \longleftrightarrow P (\text{SOME } x. P x) \rangle$

using *someI*[of $\langle \lambda x. P x \rangle$]

by *auto*

lemma *verit-sko-ex'*: $\langle P (\text{SOME } x. P x) = A \implies (\exists x. P x) = A \rangle$

by (*subst verit-sko-ex*)

lemma *verit-sko-ex-indirect*: $\langle x = (\text{SOME } x. P x) \implies (\exists x. P x) \longleftrightarrow P x \rangle$

using *someI*[of $\langle \lambda x. P x \rangle$]

by *auto*

lemma *verit-sko-ex-indirect2*: $\langle x = (\text{SOME } x. P x) \implies (\bigwedge x. P x = P' x) \implies (\exists x. P' x) \longleftrightarrow P x \rangle$

using *someI*[of $\langle \lambda x. P x \rangle$]

by *auto*

lemma *verit-Pure-trans*:

$\langle P \equiv Q \implies Q \implies P \rangle$

by *auto*

lemma *verit-if-cong*:

assumes $\langle b \equiv c \rangle$

and $\langle c \implies x \equiv u \rangle$

and $\langle \neg c \implies y \equiv v \rangle$

shows $\langle (\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } u \text{ else } v) \rangle$

using *assms if-cong*[of $b c x u$] by *auto*

lemma *verit-if-weak-cong'*:

$\langle b \equiv c \implies (\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } x \text{ else } y) \rangle$

by *auto*

lemma *verit-or-neg*:

$\langle (A \implies B) \implies B \vee \neg A \rangle$

$\langle (\neg A \implies B) \implies B \vee A \rangle$

by *auto*

lemma *verit-subst-bool*: $\langle P \implies f \text{ True} \implies f P \rangle$

by *auto*

lemma *verit-and-pos*:

$\langle (a \implies \neg(b \wedge c)) \vee A \implies \neg(a \wedge b \wedge c) \vee A \rangle$

$\langle (a \implies b \implies A) \implies \neg(a \wedge b) \vee A \rangle$

$\langle (a \implies A) \implies \neg a \vee A \rangle$
 $\langle (\neg a \implies A) \implies a \vee A \rangle$
by *blast+*

lemma *verit-or-pos*:

$\langle A \wedge A' \implies (c \wedge A) \vee (\neg c \wedge A') \rangle$
 $\langle A \wedge A' \implies (\neg c \wedge A) \vee (c \wedge A') \rangle$
by *blast+*

lemma *verit-la-generic*:

$\langle (a::int) \leq x \vee a = x \vee a \geq x \rangle$
by *linarith*

lemma *verit-bfun-elim*:

$\langle (if\ b\ then\ P\ True\ else\ P\ False) = P\ b \rangle$
 $\langle (\forall b. P' b) = (P' False \wedge P' True) \rangle$
 $\langle (\exists b. P' b) = (P' False \vee P' True) \rangle$
by (*cases b*) (*auto simp: all-bool-eq ex-bool-eq*)

lemma *verit-eq-true-simplify*:

$\langle (P = True) \equiv P \rangle$
by *auto*

lemma *verit-and-neg*:

$\langle (a \implies \neg b \vee A) \implies \neg(a \wedge b) \vee A \rangle$
 $\langle (a \implies A) \implies \neg a \vee A \rangle$
 $\langle (\neg a \implies A) \implies a \vee A \rangle$
by *blast+*

lemma *verit-forall-inst*:

$\langle A \longleftrightarrow B \implies \neg A \vee B \rangle$
 $\langle \neg A \longleftrightarrow B \implies A \vee B \rangle$
 $\langle A \longleftrightarrow B \implies \neg B \vee A \rangle$
 $\langle A \longleftrightarrow \neg B \implies B \vee A \rangle$
 $\langle A \longrightarrow B \implies \neg A \vee B \rangle$
 $\langle \neg A \longrightarrow B \implies A \vee B \rangle$
by *blast+*

lemma *verit-eq-transitive*:

$\langle A = B \implies B = C \implies A = C \rangle$
 $\langle A = B \implies C = B \implies A = C \rangle$
 $\langle B = A \implies B = C \implies A = C \rangle$
 $\langle B = A \implies C = B \implies A = C \rangle$
by *auto*

lemma *verit-bool-simplify*:

$\langle \neg(P \longrightarrow Q) \longleftrightarrow P \wedge \neg Q \rangle$
 $\langle \neg(P \vee Q) \longleftrightarrow \neg P \wedge \neg Q \rangle$

$\langle \neg(P \wedge Q) \longleftrightarrow \neg P \vee \neg Q \rangle$
 $\langle (P \longrightarrow (Q \longrightarrow R)) \longleftrightarrow ((P \wedge Q) \longrightarrow R) \rangle$
 $\langle ((P \longrightarrow Q) \longrightarrow Q) \longleftrightarrow P \vee Q \rangle$
 $\langle (Q \longleftrightarrow (P \vee Q)) \longleftrightarrow (P \longrightarrow Q) \rangle$ — This rule was inverted
 $\langle P \wedge (P \longrightarrow Q) \longleftrightarrow P \wedge Q \rangle$
 $\langle (P \longrightarrow Q) \wedge P \longleftrightarrow P \wedge Q \rangle$

unfolding *not-imp imp-conjL*
by *auto*

We need the last equation for $\neg (\forall a b. \neg P a b)$

lemma *verit-connective-def*: — the definition of XOR is missing as the operator is not generated by Isabelle

$\langle (A = B) \longleftrightarrow ((A \longrightarrow B) \wedge (B \longrightarrow A)) \rangle$
 $\langle (\text{If } A \ B \ C) = ((A \longrightarrow B) \wedge (\neg A \longrightarrow C)) \rangle$
 $\langle (\exists x. P x) \longleftrightarrow \neg(\forall x. \neg P x) \rangle$
 $\langle \neg(\exists x. P x) \longleftrightarrow (\forall x. \neg P x) \rangle$

by *auto*

lemma *verit-ite-simplify*:

$\langle (\text{If } \text{True} \ B \ C) = B \rangle$
 $\langle (\text{If } \text{False} \ B \ C) = C \rangle$
 $\langle (\text{If } A' \ B \ B) = B \rangle$
 $\langle (\text{If } (\neg A') \ B \ C) = (\text{If } A' \ C \ B) \rangle$
 $\langle (\text{If } c \ (\text{If } c \ A \ B) \ C) = (\text{If } c \ A \ C) \rangle$
 $\langle (\text{If } c \ C \ (\text{If } c \ A \ B)) = (\text{If } c \ C \ B) \rangle$
 $\langle (\text{If } A' \ \text{True} \ \text{False}) = A' \rangle$
 $\langle (\text{If } A' \ \text{False} \ \text{True}) \longleftrightarrow \neg A' \rangle$
 $\langle (\text{If } A' \ \text{True} \ B') \longleftrightarrow A' \vee B' \rangle$
 $\langle (\text{If } A' \ B' \ \text{False}) \longleftrightarrow A' \wedge B' \rangle$
 $\langle (\text{If } A' \ \text{False} \ B') \longleftrightarrow \neg A' \wedge B' \rangle$
 $\langle (\text{If } A' \ B' \ \text{True}) \longleftrightarrow \neg A' \vee B' \rangle$
 $\langle x \wedge \text{True} \longleftrightarrow x \rangle$
 $\langle x \vee \text{False} \longleftrightarrow x \rangle$

for $B \ C :: 'a$ and $A' \ B' \ C' :: \text{bool}$
by *auto*

lemma *verit-and-simplify1*:

$\langle \text{True} \wedge b \longleftrightarrow b \rangle \langle b \wedge \text{True} \longleftrightarrow b \rangle$
 $\langle \text{False} \wedge b \longleftrightarrow \text{False} \rangle \langle b \wedge \text{False} \longleftrightarrow \text{False} \rangle$
 $\langle (c \wedge \neg c) \longleftrightarrow \text{False} \rangle \langle (\neg c \wedge c) \longleftrightarrow \text{False} \rangle$
 $\langle \neg \neg a = a \rangle$

by *auto*

lemmas *verit-and-simplify = conj-ac de-Morgan-conj disj-not1*

lemma *verit-or-simplify-1*:

$\langle \text{False} \vee b \longleftrightarrow b \rangle \langle b \vee \text{False} \longleftrightarrow b \rangle$

$\langle b \vee \neg b \rangle$
 $\langle \neg b \vee b \rangle$
by auto

lemmas *verit-or-simplify* = *disj-ac*

lemma *verit-not-simplify*:

$\langle \neg \neg b \longleftrightarrow b \rangle$ $\langle \neg \text{True} \longleftrightarrow \text{False} \rangle$ $\langle \neg \text{False} \longleftrightarrow \text{True} \rangle$
by auto

lemma *verit-implies-simplify*:

$\langle (\neg a \longrightarrow \neg b) \longleftrightarrow (b \longrightarrow a) \rangle$
 $\langle (\text{False} \longrightarrow a) \longleftrightarrow \text{True} \rangle$
 $\langle (a \longrightarrow \text{True}) \longleftrightarrow \text{True} \rangle$
 $\langle (\text{True} \longrightarrow a) \longleftrightarrow a \rangle$
 $\langle (a \longrightarrow \text{False}) \longleftrightarrow \neg a \rangle$
 $\langle (a \longrightarrow a) \longleftrightarrow \text{True} \rangle$
 $\langle (\neg a \longrightarrow a) \longleftrightarrow a \rangle$
 $\langle (a \longrightarrow \neg a) \longleftrightarrow \neg a \rangle$
 $\langle ((a \longrightarrow b) \longrightarrow b) \longleftrightarrow a \vee b \rangle$
by auto

lemma *verit-equiv-simplify*:

$\langle ((\neg a) = (\neg b)) \longleftrightarrow (a = b) \rangle$
 $\langle (a = a) \longleftrightarrow \text{True} \rangle$
 $\langle (a = (\neg a)) \longleftrightarrow \text{False} \rangle$
 $\langle ((\neg a) = a) \longleftrightarrow \text{False} \rangle$
 $\langle (\text{True} = a) \longleftrightarrow a \rangle$
 $\langle (a = \text{True}) \longleftrightarrow a \rangle$
 $\langle (\text{False} = a) \longleftrightarrow \neg a \rangle$
 $\langle (a = \text{False}) \longleftrightarrow \neg a \rangle$
 $\langle \neg \neg a \longleftrightarrow a \rangle$
 $\langle (\neg \text{False}) = \text{True} \rangle$
for $a\ b :: \text{bool}$
by auto

lemmas *verit-eq-simplify* =

semiring-char-0-class.eq-numeral-simps *eq-refl* *zero-neq-one* *num.simps*
neg-equal-zero *equal-neg-zero* *one-neq-zero* *neg-equal-iff-equal*

lemma *verit-minus-simplify*:

$\langle (a :: 'a :: \text{cancel-comm-monoid-add}) - a = 0 \rangle$
 $\langle (a :: 'a :: \text{cancel-comm-monoid-add}) - 0 = a \rangle$
 $\langle 0 - (b :: 'b :: \{\text{group-add}\}) = -b \rangle$
 $\langle - (- (b :: 'b :: \text{group-add})) = b \rangle$
by auto

lemma *verit-sum-simplify*:

$\langle (a :: 'a :: \text{cancel-comm-monoid-add}) + 0 = a \rangle$
by *auto*

lemmas *verit-prod-simplify* =

mult-1
mult-1-right

lemma *verit-comp-simplify1*:

$\langle (a :: 'a :: \text{order}) < a \longleftrightarrow \text{False} \rangle$
 $\langle a \leq a \rangle$
 $\langle \neg(b' \leq a') \longleftrightarrow (a' :: 'b :: \text{linorder}) < b' \rangle$
by *auto*

lemmas *verit-comp-simplify* =

verit-comp-simplify1
le-numeral-simps
le-num-simps
less-numeral-simps
less-num-simps
zero-less-one
zero-le-one
less-neg-numeral-simps

lemma *verit-la-disequality*:

$\langle (a :: 'a :: \text{linorder}) = b \vee \neg a \leq b \vee \neg b \leq a \rangle$
by *auto*

context

begin

For the reconstruction, we need to keep the order of the arguments.

named-theorems *smt-arith-multiplication* \langle *Theorems to reconstruct arithmetic theorems.* \rangle

named-theorems *smt-arith-combine* \langle *Theorems to reconstruct arithmetic theorems.* \rangle

named-theorems *smt-arith-simplify* \langle *Theorems to combine theorems in the LA procedure* \rangle

lemmas [*smt-arith-simplify*] =

div-add dvd-numeral-simp divmod-steps less-num-simps le-num-simps if-True if-False divmod-cancel
dvd-mult dvd-mult2 less-irrefl prod.case numeral-plus-one divmod-step-def order.refl le-zero-eq
le-numeral-simps less-numeral-simps mult.right-neutral simp-thms divides-aux-eq
mult-nonneg-nonneg dvd-imp-mod-0 dvd-add zero-less-one mod-mult-self4 numeral-mod-numeral


```

  divmod-trivial prod.sel mult.left-neutral div-pos-pos-trivial arith-simps div-add
div-mult-self1
  add-le-cancel-left add-le-same-cancel2 not-one-le-zero le-numeral-simps add-le-same-cancel1
  zero-neq-one zero-le-one le-num-simps add-Suc mod-div-trivial nat.distinct mult-minus-right
  add.inverse-inverse distrib-left-numeral mult-num-simps numeral-times-numeral
add-num-simps
  divmod-steps rel-simps if-True if-False numeral-div-numeral divmod-cancel prod.case
  add-num-simps one-plus-numeral fst-conv arith-simps sub-num-simps dbl-inc-simps
  dbl-simps mult-1 add-le-cancel-right left-diff-distrib-numeral add-uminus-conv-diff
zero-neq-one
  zero-le-one One-nat-def add-Suc mod-div-trivial nat.distinct of-int-1 numerals
numeral-One
  of-int-numeral add-uminus-conv-diff zle-diff1-eq add-less-same-cancel2 minus-add-distrib
  add-uminus-conv-diff mult.left-neutral semiring-class.distrib-right
  add-diff-cancel-left' add-diff-eq ring-distrib mult-minus-left minus-diff-eq

```

lemma [smt-arith-simplify]:

```

<¬ (a' :: 'a :: linorder) < b' ↔ b' ≤ a'>
<¬ (a' :: 'a :: linorder) ≤ b' ↔ b' < a'>
<(c::int) mod Numeral1 = 0>
<(a::nat) mod Numeral1 = 0>
<(c::int) div Numeral1 = c>
<a div Numeral1 = a>
<(c::int) mod 1 = 0>
<a mod 1 = 0>
<(c::int) div 1 = c>
<a div 1 = a>
<¬(a' ≠ b') ↔ a' = b'>
by auto

```

lemma *div-mod-decomp*: $A = (A \text{ div } n) * n + (A \text{ mod } n)$ **for** $A :: \text{nat}$
by *auto*

lemma *div-less-mono*:

```

fixes A B :: nat
assumes A < B 0 < n and
  mod: A mod n = 0 B mod n = 0
shows (A div n) < (B div n)
proof –
  show ?thesis
  using assms(1)
  apply (subst (asm) div-mod-decomp[of A n])
  apply (subst (asm) div-mod-decomp[of B n])
  unfolding mod
  by (use assms(2,3) in <auto simp: ac-simps>)
qed

```

lemma *verit-le-mono-div*:

```

fixes  $A B :: \text{nat}$ 
assumes  $A < B \ 0 < n$ 
shows  $(A \text{ div } n) + (\text{if } B \text{ mod } n = 0 \text{ then } 1 \text{ else } 0) \leq (B \text{ div } n)$ 
by (auto simp: ac-simps Suc-leI assms less-mult-imp-div-less div-le-mono less-imp-le-nat)

```

```

lemmas [smt-arith-multiplication] =
  verit-le-mono-div[THEN mult-le-mono1, unfolded add-mult-distrib]
  div-le-mono[THEN mult-le-mono2, unfolded add-mult-distrib]

```

```

lemma div-mod-decomp-int:  $A = (A \text{ div } n) * n + (A \text{ mod } n)$  for  $A :: \text{int}$ 
by auto

```

```

lemma zdiv-mono-strict:
  fixes  $A B :: \text{int}$ 
  assumes  $A < B \ 0 < n$  and
    mod:  $A \text{ mod } n = 0 \ B \text{ mod } n = 0$ 
  shows  $(A \text{ div } n) < (B \text{ div } n)$ 
proof –
  show ?thesis
    using assms(1)
    apply (subst (asm) div-mod-decomp-int[of A n])
    apply (subst (asm) div-mod-decomp-int[of B n])
    unfolding mod
    by (use assms(2,3) in <auto simp: ac-simps>)
qed

```

```

lemma verit-le-mono-div-int:
   $\langle A \text{ div } n + (\text{if } B \text{ mod } n = 0 \text{ then } 1 \text{ else } 0) \leq B \text{ div } n \rangle$ 
  if  $\langle A < B \rangle \ \langle 0 < n \rangle$ 
  for  $A B n :: \text{int}$ 
proof –
  from  $\langle A < B \rangle \ \langle 0 < n \rangle$  have  $\langle A \text{ div } n \leq B \text{ div } n \rangle$ 
    by (auto intro: zdiv-mono1)
  show ?thesis
  proof (cases <n dvd B>)
    case False
    with  $\langle A \text{ div } n \leq B \text{ div } n \rangle$  show ?thesis
      by auto
  next
    case True
    then obtain  $C$  where  $\langle B = n * C \rangle ..$ 
    then have  $\langle B \text{ div } n = C \rangle$ 
      using  $\langle 0 < n \rangle$  by simp
    from  $\langle 0 < n \rangle$  have  $\langle A \text{ mod } n \geq 0 \rangle$ 
      by simp
    have  $\langle A \text{ div } n < C \rangle$ 
    proof (rule ccontr)
      assume  $\langle \neg A \text{ div } n < C \rangle$ 
      then have  $\langle C \leq A \text{ div } n \rangle$ 

```

```

    by simp
  with ⟨B div n = C⟩ ⟨A div n ≤ B div n⟩
  have ⟨A div n = C⟩
    by simp
  moreover from ⟨A < B⟩ have ⟨n * (A div n) + A mod n < B⟩
    by simp
  ultimately have ⟨n * C + A mod n < n * C⟩
    using ⟨B = n * C⟩ by simp
  moreover have ⟨A mod n ≥ 0⟩
    using ⟨0 < n⟩ by simp
  ultimately show False
    by simp
qed
with ⟨n dvd B⟩ ⟨B div n = C⟩ show ?thesis
  by simp
qed
qed

```

lemma *verit-less-mono-div-int2*:

```

  fixes A B :: int
  assumes A ≤ B 0 < -n
  shows (A div n) ≥ (B div n)
  using assms(1) assms(2) zdiv-mono1-neg by auto

```

lemmas [*smt-arith-multiplication*] =

```

  verit-le-mono-div-int[THEN mult-left-mono, unfolded int-distrib]
  zdiv-mono1[THEN mult-left-mono, unfolded int-distrib]

```

lemmas [*smt-arith-multiplication*] =

```

  arg-cong[of - - ⟨λa :: nat. a div n * p⟩ for n p :: nat, THEN sym]
  arg-cong[of - - ⟨λa :: int. a div n * p⟩ for n p :: int, THEN sym]

```

lemma [*smt-arith-combine*]:

```

  a < b ⇒ c < d ⇒ a + c + 2 ≤ b + d
  a < b ⇒ c ≤ d ⇒ a + c + 1 ≤ b + d
  a ≤ b ⇒ c < d ⇒ a + c + 1 ≤ b + d for a b c :: int
  by auto

```

lemma [*smt-arith-combine*]:

```

  a < b ⇒ c < d ⇒ a + c + 2 ≤ b + d
  a < b ⇒ c ≤ d ⇒ a + c + 1 ≤ b + d
  a ≤ b ⇒ c < d ⇒ a + c + 1 ≤ b + d for a b c :: nat
  by auto

```

lemmas [*smt-arith-combine*] =

```

  add-strict-mono
  add-less-le-mono
  add-mono
  add-le-less-mono

```

lemma [*smt-arith-combine*]:

$\langle m < n \implies c = d \implies m + c < n + d \rangle$
 $\langle m \leq n \implies c = d \implies m + c \leq n + d \rangle$
 $\langle c = d \implies m < n \implies m + c < n + d \rangle$
 $\langle c = d \implies m \leq n \implies m + c \leq n + d \rangle$
for $m :: \langle 'a :: \text{ordered-cancel-ab-semigroup-add} \rangle$
by (*auto intro: ordered-cancel-ab-semigroup-add-class.add-strict-right-mono*
ordered-ab-semigroup-add-class.add-right-mono)

lemma *verit-negate-coefficient*:

$\langle a \leq (b :: 'a :: \{\text{ordered-ab-group-add}\}) \implies -a \geq -b \rangle$
 $\langle a < b \implies -a > -b \rangle$
 $\langle a = b \implies -a = -b \rangle$
by *auto*

end

lemma *verit-ite-intro*:

$\langle (\text{if } a \text{ then } P \text{ (if } a \text{ then } a' \text{ else } b') \text{ else } Q) \longleftrightarrow (\text{if } a \text{ then } P \text{ } a' \text{ else } Q) \rangle$
 $\langle (\text{if } a \text{ then } P' \text{ else } Q' \text{ (if } a \text{ then } a' \text{ else } b')) \longleftrightarrow (\text{if } a \text{ then } P' \text{ else } Q' \text{ } b') \rangle$
 $\langle A = f \text{ (if } a \text{ then } R \text{ else } S) \longleftrightarrow (\text{if } a \text{ then } A = f R \text{ else } A = f S) \rangle$
by *auto*

lemma *verit-ite-if-cong*:

fixes $x \ y :: \text{bool}$
assumes $b=c$
and $c \equiv \text{True} \implies x = u$
and $c \equiv \text{False} \implies y = v$
shows $(\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } u \text{ else } v)$
proof –
have $H: (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$
using *assms* **by** (*auto split: if-splits*)

show $(\text{if } b \text{ then } x \text{ else } y) \equiv (\text{if } c \text{ then } u \text{ else } v)$
by (*subst H*) *auto*
qed

64.7 Setup

ML-file $\langle \text{Tools/SMT/smt-util.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smt-failure.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smt-config.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smt-builtin.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smt-datatypes.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smt-normalize.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smt-translate.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smtlib.ML} \rangle$
ML-file $\langle \text{Tools/SMT/smtlib-interface.ML} \rangle$

ML-file \langle *Tools/SMT/smtlib-proof.ML* \rangle
ML-file \langle *Tools/SMT/smtlib-isar.ML* \rangle
ML-file \langle *Tools/SMT/z3-proof.ML* \rangle
ML-file \langle *Tools/SMT/z3-isar.ML* \rangle
ML-file \langle *Tools/SMT/smt-solver.ML* \rangle
ML-file \langle *Tools/SMT/cvc-interface.ML* \rangle
ML-file \langle *Tools/SMT/lethe-proof.ML* \rangle
ML-file \langle *Tools/SMT/lethe-isar.ML* \rangle
ML-file \langle *Tools/SMT/lethe-proof-parse.ML* \rangle
ML-file \langle *Tools/SMT/cvc-proof-parse.ML* \rangle
ML-file \langle *Tools/SMT/conj-disj-perm.ML* \rangle
ML-file \langle *Tools/SMT/smt-replay-methods.ML* \rangle
ML-file \langle *Tools/SMT/smt-replay.ML* \rangle
ML-file \langle *Tools/SMT/smt-replay-arith.ML* \rangle
ML-file \langle *Tools/SMT/z3-interface.ML* \rangle
ML-file \langle *Tools/SMT/z3-replay-rules.ML* \rangle
ML-file \langle *Tools/SMT/z3-replay-methods.ML* \rangle
ML-file \langle *Tools/SMT/z3-replay.ML* \rangle
ML-file \langle *Tools/SMT/lethe-replay-methods.ML* \rangle
ML-file \langle *Tools/SMT/cvc5-replay-methods.ML* \rangle
ML-file \langle *Tools/SMT/verit-replay-methods.ML* \rangle
ML-file \langle *Tools/SMT/verit-strategies.ML* \rangle
ML-file \langle *Tools/SMT/verit-replay.ML* \rangle
ML-file \langle *Tools/SMT/cvc5-replay.ML* \rangle
ML-file \langle *Tools/SMT/smt-systems.ML* \rangle

64.8 Configuration

The current configuration can be printed by the command *smt-status*, which shows the values of most options.

64.9 General configuration options

The option *smt-solver* can be used to change the target SMT solver. The possible values can be obtained from the *smt-status* command.

```
declare [[smt-solver = z3]]
```

Since SMT solvers are potentially nonterminating, there is a timeout (given in seconds) to restrict their runtime.

```
declare [[smt-timeout = 0]]
```

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

```
declare [[smt-random-seed = 1]]
```

In general, the binding to SMT solvers runs as an oracle, i.e., the SMT solvers are fully trusted without additional checks. The following option can

cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently implemented only for veriT and Z3.

```
declare [[smt-oracle = false]]
```

Each SMT solver provides several command-line options to tweak its behaviour. They can be passed to the solver by setting the following options.

```
declare [[cvc4-options = ]]
declare [[cvc5-options = --proof-format-mode=alethe --proof-granularity=dsl-rewrite]]
declare [[verit-options = ]]
declare [[z3-options = ]]
```

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

```
declare [[smt-infer-triggers = false]]
```

Enable the following option to use built-in support for datatypes, codatatypes, and records in CVC4 and cvc5. Currently, this is implemented only in oracle mode.

```
declare [[cvc-extensions = false]]
```

Enable the following option to use built-in support for div/mod, datatypes, and records in Z3. Currently, this is implemented only in oracle mode.

```
declare [[z3-extensions = false]]
```

64.10 Certificates

By setting the option *smt-certificates* to the name of a file, all following applications of an SMT solver are cached in that file. Any further application of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending *.certs* instead of *.thy*) as the certificates file. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

```
declare [[smt-certificates = ]]
```

The option *smt-read-only-certificates* controls whether only stored certificates should be used or invocation of an SMT solver is allowed. When set to *true*, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to *false* and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

declare $[[\text{smt-read-only-certificates} = \text{false}]]$

64.11 Tracing

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to *false*.

declare $[[\text{smt-verbose} = \text{true}]]$

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option *smt-trace* should be set to *true*.

declare $[[\text{smt-trace} = \text{false}]]$

64.12 Schematic rules for Z3 proof reconstruction

Several proof rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in *z3-rule* are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in *z3-simps* are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

lemmas $[\text{z3-rule}] =$

*refl eq-commute conj-commute disj-commute simp-thms nnf-simps
ring-distrib field-simps times-divide-eq-right times-divide-eq-left
if-True if-False not-not
NO-MATCH-def*

lemma $[\text{z3-rule}]$:

$(P \wedge Q) = (\neg (\neg P \vee \neg Q))$
 $(P \wedge Q) = (\neg (\neg Q \vee \neg P))$
 $(\neg P \wedge Q) = (\neg (P \vee \neg Q))$
 $(\neg P \wedge Q) = (\neg (\neg Q \vee P))$
 $(P \wedge \neg Q) = (\neg (\neg P \vee Q))$
 $(P \wedge \neg Q) = (\neg (Q \vee \neg P))$
 $(\neg P \wedge \neg Q) = (\neg (P \vee Q))$
 $(\neg P \wedge \neg Q) = (\neg (Q \vee P))$

by *auto*

lemma $[\text{z3-rule}]$:

$(P \longrightarrow Q) = (Q \vee \neg P)$
 $(\neg P \longrightarrow Q) = (P \vee Q)$
 $(\neg P \longrightarrow Q) = (Q \vee P)$
 $(\text{True} \longrightarrow P) = P$
 $(P \longrightarrow \text{True}) = \text{True}$
 $(\text{False} \longrightarrow P) = \text{True}$
 $(P \longrightarrow P) = \text{True}$
 $(\neg (A \longleftrightarrow \neg B)) \longleftrightarrow (A \longleftrightarrow B)$

by *auto*

lemma [z3-rule]:

$$((P = Q) \longrightarrow R) = (R \vee (Q = (\neg P)))$$

by auto

lemma [z3-rule]:

$$\begin{aligned} (\neg \text{True}) &= \text{False} \\ (\neg \text{False}) &= \text{True} \\ (x = x) &= \text{True} \\ (P = \text{True}) &= P \\ (\text{True} = P) &= P \\ (P = \text{False}) &= (\neg P) \\ (\text{False} = P) &= (\neg P) \\ ((\neg P) = P) &= \text{False} \\ (P = (\neg P)) &= \text{False} \\ ((\neg P) = (\neg Q)) &= (P = Q) \\ \neg (P = (\neg Q)) &= (P = Q) \\ \neg ((\neg P) = Q) &= (P = Q) \\ (P \neq Q) &= (Q = (\neg P)) \\ (P = Q) &= ((\neg P \vee Q) \wedge (P \vee \neg Q)) \\ (P \neq Q) &= ((\neg P \vee \neg Q) \wedge (P \vee Q)) \end{aligned}$$

by auto

lemma [z3-rule]:

$$\begin{aligned} (\text{if } P \text{ then } P \text{ else } \neg P) &= \text{True} \\ (\text{if } \neg P \text{ then } \neg P \text{ else } P) &= \text{True} \\ (\text{if } P \text{ then } \text{True} \text{ else } \text{False}) &= P \\ (\text{if } P \text{ then } \text{False} \text{ else } \text{True}) &= (\neg P) \\ (\text{if } P \text{ then } Q \text{ else } \text{True}) &= ((\neg P) \vee Q) \\ (\text{if } P \text{ then } Q \text{ else } \text{True}) &= (Q \vee (\neg P)) \\ (\text{if } P \text{ then } Q \text{ else } \neg Q) &= (P = Q) \\ (\text{if } P \text{ then } Q \text{ else } \neg Q) &= (Q = P) \\ (\text{if } P \text{ then } \neg Q \text{ else } Q) &= (P = (\neg Q)) \\ (\text{if } P \text{ then } \neg Q \text{ else } Q) &= ((\neg Q) = P) \\ (\text{if } \neg P \text{ then } x \text{ else } y) &= (\text{if } P \text{ then } y \text{ else } x) \\ (\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } x) &= (\text{if } P \wedge (\neg Q) \text{ then } y \text{ else } x) \\ (\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } x) &= (\text{if } (\neg Q) \wedge P \text{ then } y \text{ else } x) \\ (\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } y) &= (\text{if } P \wedge Q \text{ then } x \text{ else } y) \\ (\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } y) &= (\text{if } Q \wedge P \text{ then } x \text{ else } y) \\ (\text{if } P \text{ then } x \text{ else } \text{if } P \text{ then } y \text{ else } z) &= (\text{if } P \text{ then } x \text{ else } z) \\ (\text{if } P \text{ then } x \text{ else } \text{if } Q \text{ then } x \text{ else } y) &= (\text{if } P \vee Q \text{ then } x \text{ else } y) \\ (\text{if } P \text{ then } x \text{ else } \text{if } Q \text{ then } x \text{ else } y) &= (\text{if } Q \vee P \text{ then } x \text{ else } y) \\ (\text{if } P \text{ then } x = y \text{ else } x = z) &= (x = (\text{if } P \text{ then } y \text{ else } z)) \\ (\text{if } P \text{ then } x = y \text{ else } y = z) &= (y = (\text{if } P \text{ then } x \text{ else } z)) \\ (\text{if } P \text{ then } x = y \text{ else } z = y) &= (y = (\text{if } P \text{ then } x \text{ else } z)) \end{aligned}$$

by auto

lemma [z3-rule]:

$$0 + (x::\text{int}) = x$$


```

 $x + 0 = x$ 
 $x + x = 2 * x$ 
 $0 * x = 0$ 
 $1 * x = x$ 
 $x + y = y + x$ 
by auto

```

lemma [z3-rule]:

```

 $P = Q \vee P \vee Q$ 
 $P = Q \vee \neg P \vee \neg Q$ 
 $(\neg P) = Q \vee \neg P \vee Q$ 
 $(\neg P) = Q \vee P \vee \neg Q$ 
 $P = (\neg Q) \vee \neg P \vee Q$ 
 $P = (\neg Q) \vee P \vee \neg Q$ 
 $P \neq Q \vee P \vee \neg Q$ 
 $P \neq Q \vee \neg P \vee Q$ 
 $P \neq (\neg Q) \vee P \vee Q$ 
 $(\neg P) \neq Q \vee P \vee Q$ 
 $P \vee Q \vee P \neq (\neg Q)$ 
 $P \vee Q \vee (\neg P) \neq Q$ 
 $P \vee \neg Q \vee P \neq Q$ 
 $\neg P \vee Q \vee P \neq Q$ 
 $P \vee y = (\text{if } P \text{ then } x \text{ else } y)$ 
 $P \vee (\text{if } P \text{ then } x \text{ else } y) = y$ 
 $\neg P \vee x = (\text{if } P \text{ then } x \text{ else } y)$ 
 $\neg P \vee (\text{if } P \text{ then } x \text{ else } y) = x$ 
 $P \vee R \vee \neg (\text{if } P \text{ then } Q \text{ else } R)$ 
 $\neg P \vee Q \vee \neg (\text{if } P \text{ then } Q \text{ else } R)$ 
 $\neg (\text{if } P \text{ then } Q \text{ else } R) \vee \neg P \vee Q$ 
 $\neg (\text{if } P \text{ then } Q \text{ else } R) \vee P \vee R$ 
 $(\text{if } P \text{ then } Q \text{ else } R) \vee \neg P \vee \neg Q$ 
 $(\text{if } P \text{ then } Q \text{ else } R) \vee P \vee \neg R$ 
 $(\text{if } P \text{ then } \neg Q \text{ else } R) \vee \neg P \vee Q$ 
 $(\text{if } P \text{ then } Q \text{ else } \neg R) \vee P \vee R$ 
by auto

```

hide-type (open) *symb-list pattern*

hide-const (open) *Symb-Nil Symb-Cons trigger pat nopat fun-app z3div z3mod*

end

65 Sledgehammer: Isabelle–ATP Linkup

theory *Sledgehammer*

imports *Presburger SMT*

keywords

sledgehammer :: *diag* **and**

sledgehammer-params :: *thy-decl*

begin

ML-file \langle Tools/ATP/system-on-ttp.ML \rangle
ML-file \langle Tools/Sledgehammer/async-manager-legacy.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-util.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-fact.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-proof-methods.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-isar-annotate.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-isar-proof.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-isar-preplay.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-isar-compress.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-isar-minimize.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-isar.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-atp-systems.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-prover.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-prover-atp.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-prover-smt.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-prover-minimize.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-mepo.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-mash.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-commands.ML \rangle
ML-file \langle Tools/Sledgehammer/sledgehammer-tactics.ML \rangle

end

66 Setup for Lifting/Transfer for the set type

theory *Lifting-Set*
imports *Lifting Groups-Big*
begin

66.1 Relator and predicator properties

lemma *rel-setD1*: $\llbracket \text{rel-set } R \ A \ B; x \in A \rrbracket \implies \exists y \in B. R \ x \ y$
and *rel-setD2*: $\llbracket \text{rel-set } R \ A \ B; y \in B \rrbracket \implies \exists x \in A. R \ x \ y$
by (*simp-all add: rel-set-def*)

lemma *rel-set-conversep* [*simp*]: $\text{rel-set } A^{-1-1} = (\text{rel-set } A)^{-1-1}$
unfolding *rel-set-def* **by** *auto*

lemma *rel-set-eq* [*relator-eq*]: $\text{rel-set } (=) = (=)$
unfolding *rel-set-def fun-eq-iff* **by** *auto*

lemma *rel-set-mono*[*relator-mono*]:
assumes $A \leq B$
shows $\text{rel-set } A \leq \text{rel-set } B$
using *assms* **unfolding** *rel-set-def* **by** *blast*

lemma *rel-set-OO*[*relator-distr*]: $\text{rel-set } R \ OO \ \text{rel-set } S = \text{rel-set } (R \ OO \ S)$

```

apply (rule sym)
apply (intro ext)
subgoal for  $X Z$ 
  apply (rule iffI)
  apply (rule relcomppI [where  $b=\{y. (\exists x \in X. R x y) \wedge (\exists z \in Z. S y z)\}$ ])
  apply (simp add: rel-set-def, fast)+
done
done

```

```

lemma Domainp-set[relator-domain]:
  Domainp (rel-set T) = ( $\lambda A. \text{Ball } A (\text{Domainp } T)$ )
  unfolding rel-set-def Domainp-iff[abs-def]
  apply (intro ext)
  apply (rule iffI)
  apply blast
  subgoal for  $A$  by (rule exI [where  $x=\{y. \exists x \in A. T x y\}$ ]) fast
done

```

```

lemma left-total-rel-set[transfer-rule]:
  left-total  $A \implies$  left-total (rel-set  $A$ )
  unfolding left-total-def rel-set-def
  apply safe
  subgoal for  $X$  by (rule exI [where  $x=\{y. \exists x \in X. A x y\}$ ]) fast
done

```

```

lemma left-unique-rel-set[transfer-rule]:
  left-unique  $A \implies$  left-unique (rel-set  $A$ )
  unfolding left-unique-def rel-set-def
  by fast

```

```

lemma right-total-rel-set [transfer-rule]:
  right-total  $A \implies$  right-total (rel-set  $A$ )
  using left-total-rel-set[of  $A^{-1-1}$ ] by simp

```

```

lemma right-unique-rel-set [transfer-rule]:
  right-unique  $A \implies$  right-unique (rel-set  $A$ )
  unfolding right-unique-def rel-set-def by fast

```

```

lemma bi-total-rel-set [transfer-rule]:
  bi-total  $A \implies$  bi-total (rel-set  $A$ )
  by(simp add: bi-total-alt-def left-total-rel-set right-total-rel-set)

```

```

lemma bi-unique-rel-set [transfer-rule]:
  bi-unique  $A \implies$  bi-unique (rel-set  $A$ )
  unfolding bi-unique-def rel-set-def by fast

```

```

lemma set-relator-eq-onp [relator-eq-onp]:
  rel-set (eq-onp  $P$ ) = eq-onp ( $\lambda A. \text{Ball } A P$ )
  unfolding fun-eq-iff rel-set-def eq-onp-def Ball-def by fast

```

lemma *bi-unique-rel-set-lemma*:

assumes *bi-unique R* **and** *rel-set R X Y*

obtains *f* **where** $Y = \text{image } f \text{ } X$ **and** *inj-on f X* **and** $\forall x \in X. R \ x \ (f \ x)$

proof

define *f* **where** $f \ x = (\text{THE } y. R \ x \ y)$ **for** *x*

{ **fix** *x* **assume** $x \in X$

with $\langle \text{rel-set } R \ X \ Y \rangle \langle \text{bi-unique } R \rangle$ **have** $R \ x \ (f \ x)$

by (*simp add: bi-unique-def rel-set-def f-def*) (*metis theI*)

with *assms* $\langle x \in X \rangle$

have $R \ x \ (f \ x) \ \forall x' \in X. R \ x' \ (f \ x) \longrightarrow x = x' \ \forall y \in Y. R \ x \ y \longrightarrow y = f \ x \ f \ x \in$

Y

by (*fastforce simp add: bi-unique-def rel-set-def*)**+** }

note $*$ = *this*

moreover

{ **fix** *y* **assume** $y \in Y$

with $\langle \text{rel-set } R \ X \ Y \rangle \ *(\exists) \langle y \in Y \rangle$ **have** $\exists x \in X. y = f \ x$

by (*fastforce simp: rel-set-def*) }

ultimately show $\forall x \in X. R \ x \ (f \ x) \ Y = \text{image } f \ X \ \text{inj-on } f \ X$

by (*auto simp: inj-on-def image-iff*)

qed

66.2 Quotient theorem for the Lifting package

lemma *Quotient-set[quot-map]*:

assumes *Quotient R Abs Rep T*

shows *Quotient (rel-set R) (image Abs) (image Rep) (rel-set T)*

using *assms* **unfolding** *Quotient-alt-def4*

apply (*simp add: rel-set-OO[symmetric]*)

apply (*simp add: rel-set-def*)

apply *fast*

done

66.3 Transfer rules for the Transfer package

66.3.1 Unconditional transfer rules

context **includes** *lifting-syntax*

begin

lemma *empty-transfer [transfer-rule]*: $(\text{rel-set } A) \ \{\} \ \{\}$

unfolding *rel-set-def* **by** *simp*

lemma *insert-transfer [transfer-rule]*:

$(A \ ==\ ==\ ==\ > \ \text{rel-set } A \ ==\ ==\ > \ \text{rel-set } A)$ *insert insert*

unfolding *rel-fun-def rel-set-def* **by** *auto*

lemma *union-transfer [transfer-rule]*:

$(\text{rel-set } A \ ==\ ==\ ==\ > \ \text{rel-set } A \ ==\ ==\ > \ \text{rel-set } A)$ *union union*

unfolding *rel-fun-def rel-set-def* **by** *auto*

lemma *Union-transfer* [*transfer-rule*]:
 $(rel\text{-}set\ (rel\text{-}set\ A) \implies rel\text{-}set\ A) \textit{ Union Union}$
unfolding *rel-fun-def rel-set-def* **by** *simp fast*

lemma *image-transfer* [*transfer-rule*]:
 $((A \implies B) \implies rel\text{-}set\ A \implies rel\text{-}set\ B) \textit{ image image}$
unfolding *rel-fun-def rel-set-def* **by** *simp fast*

lemma *UNION-transfer* [*transfer-rule*]: — TODO deletion candidate
 $(rel\text{-}set\ A \implies (A \implies rel\text{-}set\ B) \implies rel\text{-}set\ B) (\lambda A\ f.\ \bigcup(f\ 'A)) (\lambda A\ f.\ \bigcup(f\ 'A))$
by *transfer-prover*

lemma *Ball-transfer* [*transfer-rule*]:
 $(rel\text{-}set\ A \implies (A \implies (=)) \implies (=)) \textit{ Ball Ball}$
unfolding *rel-set-def rel-fun-def* **by** *fast*

lemma *Bex-transfer* [*transfer-rule*]:
 $(rel\text{-}set\ A \implies (A \implies (=)) \implies (=)) \textit{ Bex Bex}$
unfolding *rel-set-def rel-fun-def* **by** *fast*

lemma *Pow-transfer* [*transfer-rule*]:
 $(rel\text{-}set\ A \implies rel\text{-}set\ (rel\text{-}set\ A)) \textit{ Pow Pow}$
apply (*rule rel-funI*)
apply (*rule rel-setI*)
subgoal for $X\ Y\ X'$
apply (*rule rev-bexI* [**where** $x=\{y\in Y.\ \exists x\in X'.\ A\ x\ y\}$])
apply *clarsimp*
apply (*simp add: rel-set-def*)
apply *fast*
done
subgoal for $X\ Y\ Y'$
apply (*rule rev-bexI* [**where** $x=\{x\in X.\ \exists y\in Y'.\ A\ x\ y\}$])
apply *clarsimp*
apply (*simp add: rel-set-def*)
apply *fast*
done
done

lemma *rel-set-transfer* [*transfer-rule*]:
 $((A \implies B \implies (=)) \implies rel\text{-}set\ A \implies rel\text{-}set\ B \implies (=)) \textit{ rel-set rel-set}$
unfolding *rel-fun-def rel-set-def* **by** *fast*

lemma *bind-transfer* [*transfer-rule*]:
 $(rel\text{-}set\ A \implies (A \implies rel\text{-}set\ B) \implies rel\text{-}set\ B) \textit{ Set.bind Set.bind}$
unfolding *bind-UNION [abs-def]* **by** *transfer-prover*

lemma *INF-parametric* [*transfer-rule*]: — TODO deletion candidate
 (*rel-set* A $====>$ (A $====>$ *HOL.eq*) $====>$ *HOL.eq*) ($\lambda A f. \text{Inf } (f \text{ ' } A)$) ($\lambda A f. \text{Inf } (f \text{ ' } A)$)
 by *transfer-prover*

lemma *SUP-parametric* [*transfer-rule*]: — TODO deletion candidate
 (*rel-set* R $====>$ (R $====>$ *HOL.eq*) $====>$ *HOL.eq*) ($\lambda A f. \text{Sup } (f \text{ ' } A)$) ($\lambda A f. \text{Sup } (f \text{ ' } A)$)
 by *transfer-prover*

66.3.2 Rules requiring bi-unique, bi-total or right-total relations

lemma *member-transfer* [*transfer-rule*]:
 assumes *bi-unique* A
 shows (A $====>$ *rel-set* A $====>$ (=)) (\in) (\in)
 using *assms* **unfolding** *rel-fun-def rel-set-def bi-unique-def* by *fast*

lemma *right-total-Collect-transfer*[*transfer-rule*]:
 assumes *right-total* A
 shows ((A $====>$ (=)) $====>$ *rel-set* A) ($\lambda P. \text{Collect } (\lambda x. P x \wedge \text{Domainp } A x)$) *Collect*
 using *assms* **unfolding** *right-total-def rel-set-def rel-fun-def Domainp-iff* by *fast*

lemma *Collect-transfer* [*transfer-rule*]:
 assumes *bi-total* A
 shows ((A $====>$ (=)) $====>$ *rel-set* A) *Collect Collect*
 using *assms* **unfolding** *rel-fun-def rel-set-def bi-total-def* by *fast*

lemma *inter-transfer* [*transfer-rule*]:
 assumes *bi-unique* A
 shows (*rel-set* A $====>$ *rel-set* A $====>$ *rel-set* A) *inter inter*
 using *assms* **unfolding** *rel-fun-def rel-set-def bi-unique-def* by *fast*

lemma *Diff-transfer* [*transfer-rule*]:
 assumes *bi-unique* A
 shows (*rel-set* A $====>$ *rel-set* A $====>$ *rel-set* A) ($-$) ($-$)
 using *assms* **unfolding** *rel-fun-def rel-set-def bi-unique-def*
unfolding *Ball-def Bex-def Diff-eq*
 by (*safe, simp, metis, simp, metis*)

lemma *subset-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique* A
 shows (*rel-set* A $====>$ *rel-set* A $====>$ (=)) (\subseteq) (\subseteq)
unfolding *subset-eq [abs-def]* by *transfer-prover*

context
 includes *lifting-syntax*
 begin

```

lemma strict-subset-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (rel-set A  $\implies$  rel-set A  $\implies$  (=)) ( $\subset$ ) ( $\subset$ )
  unfolding subset-not-subset-eq by transfer-prover

end

declare right-total-UNIV-transfer[transfer-rule]

lemma UNIV-transfer [transfer-rule]:
  assumes bi-total A
  shows (rel-set A) UNIV UNIV
  using assms unfolding rel-set-def bi-total-def by simp

lemma right-total-Compl-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
  shows (rel-set A  $\implies$  rel-set A) ( $\lambda S. \text{uminus } S \cap \text{Collect } (\text{Domainp } A)$ )
  uminus
  unfolding Compl-eq [abs-def]
  by (subst Collect-conj-eq[symmetric]) transfer-prover

lemma Compl-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
  shows (rel-set A  $\implies$  rel-set A) uminus uminus
  unfolding Compl-eq [abs-def] by transfer-prover

lemma right-total-Inter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
  shows (rel-set (rel-set A)  $\implies$  rel-set A) ( $\lambda S. \bigcap S \cap \text{Collect } (\text{Domainp } A)$ )
  Inter
  unfolding Inter-eq[abs-def]
  by (subst Collect-conj-eq[symmetric]) transfer-prover

lemma Inter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
  shows (rel-set (rel-set A)  $\implies$  rel-set A) Inter Inter
  unfolding Inter-eq [abs-def] by transfer-prover

lemma filter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows ((A  $\implies$  (=))  $\implies$  rel-set A  $\implies$  rel-set A) Set.filter Set.filter
  unfolding Set.filter-def[abs-def] rel-fun-def rel-set-def by blast

lemma finite-transfer [transfer-rule]:
  bi-unique A  $\implies$  (rel-set A  $\implies$  (=)) finite finite
  by (rule rel-funI, erule (1) bi-unique-rel-set-lemma)
  (auto dest: finite-imageD)

lemma card-transfer [transfer-rule]:

```

bi-unique $A \implies (\text{rel-set } A \implies (=)) \text{ card card}$
by (*rule rel-funI*, *erule (1) bi-unique-rel-set-lemma*)
(simp add: card-image)

context

includes *lifting-syntax*

begin

lemma *vimage-right-total-transfer*[*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique B right-total A*

shows $((A \implies B) \implies \text{rel-set } B \implies \text{rel-set } A) (\lambda f X. f -' X \cap \text{Collect } (\text{Domainp } A)) \text{ vimage}$

proof –

let $?vimage = (\lambda f B. \{x. f x \in B \wedge \text{Domainp } A x\})$

have $((A \implies B) \implies \text{rel-set } B \implies \text{rel-set } A) ?vimage \text{ vimage}$

unfolding *vimage-def*

by *transfer-prover*

also have $?vimage = (\lambda f X. f -' X \cap \text{Collect } (\text{Domainp } A))$

by *auto*

finally show *?thesis* .

qed

end

lemma *vimage-parametric* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-total A bi-unique B*

shows $((A \implies B) \implies \text{rel-set } B \implies \text{rel-set } A) \text{ vimage vimage}$

unfolding *vimage-def*[*abs-def*] **by** *transfer-prover*

lemma *Image-parametric* [*transfer-rule*]:

assumes *bi-unique A*

shows $(\text{rel-set } (\text{rel-prod } A B) \implies \text{rel-set } A \implies \text{rel-set } B) (‘) (‘)$

by (*intro rel-funI rel-setI*)

(force dest: rel-setD1 bi-uniqueDr[OF assms], force dest: rel-setD2 bi-uniqueDl[OF assms])

lemma *inj-on-transfer*[*transfer-rule*]:

$((A \implies B) \implies \text{rel-set } A \implies (=)) \text{ inj-on inj-on}$

if [*transfer-rule*]: *bi-unique A bi-unique B*

unfolding *inj-on-def*

by *transfer-prover*

end

lemma (**in** *comm-monoid-set*) *F-parametric* [*transfer-rule*]:

fixes $A :: 'b \Rightarrow 'c \Rightarrow \text{bool}$

assumes *bi-unique A*

shows $\text{rel-fun } (\text{rel-fun } A (=)) (\text{rel-fun } (\text{rel-set } A) (=)) F F$

proof (*rule rel-funI*)+


```

fix f :: 'b ⇒ 'a and g S T
assume rel-fun A (=) f g rel-set A S T
with ⟨bi-unique A⟩ obtain i where bij-betw i S T ∧ x. x ∈ S ⇒ f x = g (i x)
  by (auto elim: bi-unique-rel-set-lemma simp: rel-fun-def bij-betw-def)
then show F f S = F g T
  by (simp add: reindex-bij-betw)
qed

```

```

lemmas sum-parametric = sum.F-parametric
lemmas prod-parametric = prod.F-parametric

```

```

lemma rel-set-UNION:
  assumes [transfer-rule]: rel-set Q A B rel-fun Q (rel-set R) f g
  shows rel-set R (⋃(f ‘ A)) (⋃(g ‘ B))
  by transfer-prover

```

```

context
  includes lifting-syntax
begin

```

```

lemma fold-graph-transfer[transfer-rule]:
  assumes bi-unique R right-total R
  shows ((R =====> (=) =====> (=)) =====> (=) =====> rel-set R =====> (=)
  =====> (=)) fold-graph fold-graph
proof(intro rel-funI)
  fix f1 :: 'a ⇒ 'c ⇒ 'c and f2 :: 'b ⇒ 'c ⇒ 'c
  assume rel-f: (R =====> (=) =====> (=)) f1 f2
  fix z1 z2 :: 'c assume [simp]: z1 = z2
  fix A1 A2 assume rel-A: rel-set R A1 A2
  fix y1 y2 :: 'c assume [simp]: y1 = y2

```

```

from ⟨bi-unique R⟩ ⟨right-total R⟩ have The-y: ∀ y. ∃!x. R x y
  unfolding bi-unique-def right-total-def by auto
define r where r ≡ λy. THE x. R x y

```

```

from The-y have r-y: R (r y) y for y
  unfolding r-def using the-equality by fastforce
with assms rel-A have inj-on r A2 A1 = r ‘ A2
  unfolding r-def rel-set-def inj-on-def bi-unique-def
  apply(auto simp: image-iff) by metis+
with ⟨bi-unique R⟩ rel-f r-y have (f1 o r) y = f2 y for y
  unfolding bi-unique-def rel-fun-def by auto
then have (f1 o r) = f2
  by blast
then show fold-graph f1 z1 A1 y1 = fold-graph f2 z2 A2 y2
  by (fastforce simp: fold-graph-image[OF ⟨inj-on r A2⟩] ⟨A1 = r ‘ A2⟩)
qed

```

```

lemma fold-transfer[transfer-rule]:

```

```

assumes [transfer-rule]: bi-unique R right-total R
shows ((R ==> (=) ==> (=)) ==> (=) ==> rel-set R ==> (=))
Finite-Set.fold Finite-Set.fold
unfolding Finite-Set.fold-def
by transfer-prover

end

end

```

67 The datatype of finite lists

```

theory List
imports Sledgehammer Lifting-Set
begin

datatype (set: 'a) list =
  Nil []
  | Cons (hd: 'a) (tl: 'a list) (infixr # 65)
for
  map: map
  rel: list-all2
  pred: list-all
where
  tl [] = []

datatype-compat list

lemma [case-names Nil Cons, cases type: list]:
  — for backward compatibility – names of variables differ
  (y = [] ==> P) ==> (∧ a list. y = a # list ==> P) ==> P
by (rule list.exhaust)

lemma [case-names Nil Cons, induct type: list]:
  — for backward compatibility – names of variables differ
  P [] ==> (∧ a list. P list ==> P (a # list)) ==> P list
by (rule list.induct)

Compatibility:
setup ‹Sign.mandatory-path list›

lemmas inducts = list.induct
lemmas recs = list.rec
lemmas cases = list.case

setup ‹Sign.parent-path›

lemmas set-simps = list.set

```

syntax

— list Enumeration

 $-list :: args \Rightarrow 'a\ list \quad ([(-)])$ **translations** $[x, xs] == x\#\![xs]$ $[x] == x\#\![]$ **67.1 Basic list processing functions****primrec** (*nonexhaustive*) $last :: 'a\ list \Rightarrow 'a\ \mathbf{where}$ $last\ (x\ \#\ xs) = (if\ xs = []\ then\ x\ else\ last\ xs)$ **primrec** $butlast :: 'a\ list \Rightarrow 'a\ list\ \mathbf{where}$ $butlast\ [] = []\ |$ $butlast\ (x\ \#\ xs) = (if\ xs = []\ then\ []\ else\ x\ \#\ butlast\ xs)$ **lemma** *set-rec*: $set\ xs = rec\text{-}list\ \{\}\ (\lambda x.\ insert\ x)\ xs$ **by** (*induct xs*) *auto***definition** $coset :: 'a\ list \Rightarrow 'a\ set\ \mathbf{where}$ $[simp]:\ coset\ xs = -\ set\ xs$ **primrec** $append :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list\ (\mathbf{infixr}\ @\ 65)\ \mathbf{where}$ *append-Nil*: $[]\ @\ ys = ys\ |$ *append-Cons*: $(x\ \#\ xs)\ @\ ys = x\ \#\ xs\ @\ ys$ **primrec** $rev :: 'a\ list \Rightarrow 'a\ list\ \mathbf{where}$ $rev\ [] = []\ |$ $rev\ (x\ \#\ xs) = rev\ xs\ @\ [x]$ **primrec** $filter :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list\ \mathbf{where}$ *filter P* $[] = []\ |$ *filter P* $(x\ \#\ xs) = (if\ P\ x\ then\ x\ \#\ filter\ P\ xs\ else\ filter\ P\ xs)$

Special input syntax for filter:

syntax (*ASCII*) $-filter :: [pttrn, 'a\ list, bool] \Rightarrow 'a\ list\ ((1[-<--./ -]))$ **syntax** $-filter :: [pttrn, 'a\ list, bool] \Rightarrow 'a\ list\ ((1[-<-- ./ -]))$ **translations** $[x<-xs . P] \rightarrow CONST\ filter\ (\lambda x.\ P)\ xs$ **primrec** $fold :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b\ \mathbf{where}$ *fold-Nil*: $fold\ f\ [] = id\ |$ *fold-Cons*: $fold\ f\ (x\ \#\ xs) = fold\ f\ xs\ o\ f\ x$ **primrec** $foldr :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b\ \mathbf{where}$

foldr-Nil: $\text{foldr } f \ [] = \text{id} \mid$
foldr-Cons: $\text{foldr } f (x \# xs) = f \ x \circ \text{foldr } f \ xs$

primrec *foldl* :: $('b \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ list} \Rightarrow 'b$ **where**
foldl-Nil: $\text{foldl } f \ a \ [] = a \mid$
foldl-Cons: $\text{foldl } f \ a (x \# xs) = \text{foldl } f (f \ a \ x) \ xs$

primrec *concat*:: $'a \text{ list list} \Rightarrow 'a \text{ list}$ **where**
concat $[] = [] \mid$
concat $(x \# xs) = x \ @ \ \text{concat } xs$

primrec *drop*:: $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
drop-Nil: $\text{drop } n \ [] = [] \mid$
drop-Cons: $\text{drop } n (x \# xs) = (\text{case } n \text{ of } 0 \Rightarrow x \# xs \mid \text{Suc } m \Rightarrow \text{drop } m \ xs)$
 — Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = \text{Suc } k$

primrec *take*:: $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
take-Nil: $\text{take } n \ [] = [] \mid$
take-Cons: $\text{take } n (x \# xs) = (\text{case } n \text{ of } 0 \Rightarrow [] \mid \text{Suc } m \Rightarrow x \# \text{take } m \ xs)$
 — Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = \text{Suc } k$

primrec (*nonexhaustive*) *nth* :: $'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a$ (**infixl** ! 100) **where**
nth-Cons: $(x \# xs) ! n = (\text{case } n \text{ of } 0 \Rightarrow x \mid \text{Suc } k \Rightarrow xs ! k)$
 — Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = \text{Suc } k$

primrec *list-update* :: $'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ list}$ **where**
list-update $[] \ i \ v = [] \mid$
list-update $(x \# xs) \ i \ v =$
 $(\text{case } i \text{ of } 0 \Rightarrow v \# xs \mid \text{Suc } j \Rightarrow x \# \text{list-update } xs \ j \ v)$

nonterminal *lupdbinds* and *lupdbind*

syntax

-lupdbind:: $['a, 'a] \Rightarrow \text{lupdbind} \quad ((2- := / -))$
 $:: \text{lupdbind} \Rightarrow \text{lupdbinds} \quad (-)$
-lupdbinds :: $[\text{lupdbind}, \text{lupdbinds}] \Rightarrow \text{lupdbinds} \quad (-, / -)$
-LUpdate :: $['a, \text{lupdbinds}] \Rightarrow 'a \quad (- / [(-)] [1000, 0] 900)$

translations

-LUpdate $xs \ (-\text{lupdbinds } b \ bs) == -\text{LUpdate } (-\text{LUpdate } xs \ b) \ bs$
 $xs[i:=x] == \text{CONST list-update } xs \ i \ x$

primrec *takeWhile* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
takeWhile $P \ [] = [] \mid$
takeWhile $P (x \# xs) = (\text{if } P \ x \text{ then } x \# \text{takeWhile } P \ xs \ \text{else } [])$

primrec *dropWhile* :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list **where**
dropWhile P [] = [] |
dropWhile P (x # xs) = (if P x then *dropWhile* P xs else x # xs)

primrec *zip* :: 'a list ⇒ 'b list ⇒ ('a × 'b) list **where**
zip xs [] = [] |
zip-Cons: *zip* xs (y # ys) =
 (case xs of [] ⇒ [] | z # zs ⇒ (z, y) # *zip* zs ys)
 — Warning: simpset does not contain this definition, but separate theorems for
xs = [] and *xs = z # zs*

abbreviation *map2* :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a list ⇒ 'b list ⇒ 'c list **where**
map2 f xs ys ≡ *map* (λ(x,y). f x y) (*zip* xs ys)

primrec *product* :: 'a list ⇒ 'b list ⇒ ('a × 'b) list **where**
product [] - = [] |
product (x#xs) ys = *map* (Pair x) ys @ *product* xs ys

hide-const (open) *product*

primrec *product-lists* :: 'a list list ⇒ 'a list list **where**
product-lists [] = [[]] |
product-lists (xs # xss) = *concat* (*map* (λx. *map* (Cons x) (*product-lists* xss)) xs)

primrec *upt* :: nat ⇒ nat ⇒ nat list ((1[-..</-])) **where**
upt-0: [i..<0] = [] |
upt-Suc: [i..<(Suc j)] = (if i ≤ j then [i..<j] @ [j] else [])

definition *insert* :: 'a ⇒ 'a list ⇒ 'a list **where**
insert x xs = (if x ∈ set xs then xs else x # xs)

definition *union* :: 'a list ⇒ 'a list ⇒ 'a list **where**
union = *fold insert*

hide-const (open) *insert union*
hide-fact (open) *insert-def union-def*

primrec *find* :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a option **where**
find - [] = None |
find P (x#xs) = (if P x then Some x else *find* P xs)

In the context of multisets, *count-list* is equivalent to *count* ◦ *mset* and it is advisable to use the latter.

primrec *count-list* :: 'a list ⇒ 'a ⇒ nat **where**
count-list [] y = 0 |
count-list (x#xs) y = (if x=y then *count-list* xs y + 1 else *count-list* xs y)

definition
extract :: ('a ⇒ bool) ⇒ 'a list ⇒ ('a list * 'a * 'a list) option

where *extract* P $xs =$
 (*case* *dropWhile* (*Not* $\circ P$) xs of
 $[] \Rightarrow None$ |
 $y\#ys \Rightarrow Some(takeWhile (Not \circ P) xs, y, ys)$)

hide-const (**open**) *extract*

primrec *those* $:: 'a$ option list $\Rightarrow 'a$ list option

where
those $[] = Some []$ |
those ($x \# xs$) = (*case* x of
 $None \Rightarrow None$
 | $Some y \Rightarrow map-option (Cons y) (those xs)$)

primrec *remove1* $:: 'a \Rightarrow 'a$ list $\Rightarrow 'a$ list **where**

remove1 $x [] = []$ |
remove1 $x (y \# xs) = (if x = y then xs else y \# remove1 x xs)$

primrec *removeAll* $:: 'a \Rightarrow 'a$ list $\Rightarrow 'a$ list **where**

removeAll $x [] = []$ |
removeAll $x (y \# xs) = (if x = y then removeAll x xs else y \# removeAll x xs)$

primrec *distinct* $:: 'a$ list $\Rightarrow bool$ **where**

distinct $[] \longleftrightarrow True$ |
distinct ($x \# xs$) $\longleftrightarrow x \notin set xs \wedge distinct xs$

fun *successively* $:: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a$ list $\Rightarrow bool$ **where**

successively $P [] = True$ |
successively $P [x] = True$ |
successively $P (x \# y \# xs) = (P x y \wedge successively P (y\#xs))$

definition *distinct-adj* **where**

distinct-adj = *successively* (\neq)

primrec *remdups* $:: 'a$ list $\Rightarrow 'a$ list **where**

remdups $[] = []$ |
remdups ($x \# xs$) = (*if* $x \in set xs$ then *remdups* xs else $x \# remdups xs$)

fun *remdups-adj* $:: 'a$ list $\Rightarrow 'a$ list **where**

remdups-adj $[] = []$ |
remdups-adj $[x] = [x]$ |
remdups-adj ($x \# y \# xs$) = (*if* $x = y$ then *remdups-adj* ($x \# xs$) else $x \# remdups-adj (y \# xs)$)

primrec *replicate* $:: nat \Rightarrow 'a \Rightarrow 'a$ list **where**

replicate-0: *replicate* $0 x = []$ |
replicate-Suc: *replicate* (*Suc* n) $x = x \# replicate n x$

Function *size* is overloaded for all datatypes. Users may refer to the list

version as *length*.

abbreviation *length* :: 'a list \Rightarrow nat **where**
length \equiv *size*

definition *enumerate* :: nat \Rightarrow 'a list \Rightarrow (nat \times 'a) list **where**
enumerate-eq-zip: *enumerate* n xs = *zip* [n..*n* + *length* xs] xs

primrec *rotate1* :: 'a list \Rightarrow 'a list **where**
rotate1 [] = [] |
rotate1 (x # xs) = xs @ [x]

definition *rotate* :: nat \Rightarrow 'a list \Rightarrow 'a list **where**
rotate n = *rotate1* $\overset{\sim}{\sim}$ n

definition *nths* :: 'a list \Rightarrow nat set \Rightarrow 'a list **where**
nths xs A = *map fst* (*filter* ($\lambda p. \text{snd } p \in A$) (*zip* xs [0..*size* xs]))

primrec *subseqs* :: 'a list \Rightarrow 'a list list **where**
subseqs [] = [[]] |
subseqs (x#xs) = (*let* xss = *subseqs* xs *in map* (*Cons* x) xss @ xss)

primrec *n-lists* :: nat \Rightarrow 'a list \Rightarrow 'a list list **where**
n-lists 0 xs = [[]] |
n-lists (Suc n) xs = *concat* (*map* ($\lambda ys. \text{map } (\lambda y. y \# ys)$) xs) (*n-lists* n xs)

hide-const (open) *n-lists*

function *splice* :: 'a list \Rightarrow 'a list \Rightarrow 'a list **where**
splice [] ys = ys |
splice (x#xs) ys = x # *splice* ys xs
by *pat-completeness auto*

termination
by(*relation measure*($\lambda(xs,ys). \text{size } xs + \text{size } ys$)) *auto*

function *shuffles* **where**
shuffles [] ys = {ys}
| *shuffles* xs [] = {xs}
| *shuffles* (x # xs) (y # ys) = (#) x ‘ *shuffles* xs (y # ys) \cup (#) y ‘ *shuffles* (x # xs) ys
by *pat-completeness simp-all*
termination **by** *lexicographic-order*

Use only if you cannot use *Min* instead:

fun *min-list* :: 'a::ord list \Rightarrow 'a **where**
min-list (x # xs) = (*case* xs *of* [] \Rightarrow x | - \Rightarrow *min* x (*min-list* xs))

Returns first minimum:

fun *arg-min-list* :: ('a \Rightarrow ('b::linorder)) \Rightarrow 'a list \Rightarrow 'a **where**

$arg-min-list\ f\ [x] = x \mid$
 $arg-min-list\ f\ (x\#\#y\#\#zs) = (let\ m = arg-min-list\ f\ (y\#\#zs)\ in\ if\ f\ x \leq f\ m\ then\ x$
 $else\ m)$

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort(ed) functions are intended for proofs, not for efficient implementations.

A sorted predicate w.r.t. a relation:

fun *sorted-wrt* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool **where**
sorted-wrt P [] = True |
sorted-wrt P (x # ys) = ((∀ y ∈ set ys. P x y) ∧ *sorted-wrt* P ys)

A class-based sorted predicate:

context *linorder*
begin

abbreviation *sorted* :: 'a list ⇒ bool **where**
sorted ≡ *sorted-wrt* (≤)

lemma *sorted-simps*: *sorted* [] = True *sorted* (x # ys) = ((∀ y ∈ set ys. x ≤ y) ∧ *sorted* ys)
by *auto*

lemma *strict-sorted-simps*: *sorted-wrt* (<) [] = True *sorted-wrt* (<) (x # ys) = ((∀ y ∈ set ys. x < y) ∧ *sorted-wrt* (<) ys)
by *auto*

primrec *insort-key* :: ('b ⇒ 'a) ⇒ 'b ⇒ 'b list ⇒ 'b list **where**
insort-key f x [] = [x] |
insort-key f x (y # ys) =
 (if f x ≤ f y then (x # y # ys) else y # (*insort-key* f x ys))

definition *sort-key* :: ('b ⇒ 'a) ⇒ 'b list ⇒ 'b list **where**
sort-key f xs = *foldr* (*insort-key* f) xs []

definition *insort-insert-key* :: ('b ⇒ 'a) ⇒ 'b ⇒ 'b list ⇒ 'b list **where**
insort-insert-key f x xs =
 (if f x ∈ f ` set xs then xs else *insort-key* f x xs)

abbreviation *sort* ≡ *sort-key* (λx. x)

abbreviation *insort* ≡ *insort-key* (λx. x)

abbreviation *insort-insert* ≡ *insort-insert-key* (λx. x)

definition *stable-sort-key* :: (('b ⇒ 'a) ⇒ 'b list ⇒ 'b list) ⇒ bool **where**
stable-sort-key sk =
 (∀ f xs k. *filter* (λy. f y = k) (sk f xs) = *filter* (λy. f y = k) xs)


```

[a, b] @ [c, d] = [a, b, c, d]
length [a, b, c] = 3
set [a, b, c] = {a, b, c}
map f [a, b, c] = [f a, f b, f c]
rev [a, b, c] = [c, b, a]
hd [a, b, c, d] = a
tl [a, b, c, d] = [b, c, d]
last [a, b, c, d] = d
butlast [a, b, c, d] = [a, b, c]
filter (λn::nat. n<2) [0,2,1] = [0,1]
concat [[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]
fold f [a, b, c] x = f c (f b (f a x))
foldr f [a, b, c] x = f a (f b (f c x))
foldl f x [a, b, c] = f (f (f x a) b) c
successively (≠) [True, False, True, False]
zip [a, b, c] [x, y, z] = [(a, x), (b, y), (c, z)]
zip [a, b] [x, y, z] = [(a, x), (b, y)]
enumerate 3 [a, b, c] = [(3, a), (4, b), (5, c)]
List.product [a, b] [c, d] = [(a, c), (a, d), (b, c), (b, d)]
product-lists [[a, b], [c], [d, e]] = [[a, c, d], [a, c, e], [b, c, d], [b, c, e]]
splice [a, b, c] [x, y, z] = [a, x, b, y, c, z]
splice [a, b, c, d] [x, y] = [a, x, b, y, c, d]
shuffles [a, b] [c, d] = {[a, b, c, d], [a, c, b, d], [a, c, d, b], [c, a, b, d], [c, a, d, b], [c, d, a, b]}
take 2 [a, b, c, d] = [a, b]
take 6 [a, b, c, d] = [a, b, c, d]
drop 2 [a, b, c, d] = [c, d]
drop 6 [a, b, c, d] = []
takeWhile (λn. n < 3) [1, 2, 3, 0] = [1, 2]
dropWhile (λn. n < 3) [1, 2, 3, 0] = [3, 0]
distinct [2, 0, 1]
remdups [2, 0, 2, 1, 2] = [0, 1, 2]
remdups-adj [2, 2, 3, 1, 1, 2, 1] = [2, 3, 1, 2, 1]
List.insert 2 [0, 1, 2] = [0, 1, 2]
List.insert 3 [0, 1, 2] = [3, 0, 1, 2]
List.union [2, 3, 4] [0, 1, 2] = [4, 3, 0, 1, 2]
find ((<) 0) [0, 0] = None
find ((<) 0) [0, 1, 0, 2] = Some 1
count-list [0, 1, 0, 2] 0 = 2
List.extract ((<) 0) [0, 0] = None
List.extract ((<) 0) [0, 1, 0, 2] = Some ([0], 1, [0, 2])
remove1 2 [2, 0, 2, 1, 2] = [0, 2, 1, 2]
removeAll 2 [2, 0, 2, 1, 2] = [0, 1]
[a, b, c, d] ! 2 = c
[a, b, c, d][2 := x] = [a, b, x, d]
nth [a, b, c, d, e] {0, 2, 3} = [a, c, d]
subseqs [a, b] = [[a, b], [a], [b], []]
List.n-lists 2 [a, b, c] = [[a, a], [b, a], [c, a], [a, b], [b, b], [c, b], [a, c], [b, c], [c, c]]
rotate1 [a, b, c, d] = [b, c, d, a]
rotate 3 [a, b, c, d] = [d, a, b, c]
replicate 4 a = [a, a, a, a]
[2..<5] = [2, 3, 4]
min-list [3, 1, -2] = -2
arg-min-list (λi i * i) [3, -1, 1, -2] = -1

```

lemma *strict-sorted-iff*: *sorted-wrt* ($<$) $l \iff$ *sorted* $l \wedge$ *distinct* l
by (*induction* l) (*auto iff*: *antisym-conv1*)

lemma *strict-sorted-imp-sorted*: *sorted-wrt* ($<$) $xs \implies$ *sorted* xs
by (*auto simp*: *strict-sorted-iff*)

end

67.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: $[(x,y). x \leftarrow xs, y \leftarrow ys, x \neq y]$, the list of all pairs of distinct elements from xs and ys . The syntax is as in Haskell, except that $|$ becomes a dot (like in Isabelle’s set comprehension): $[e. x \leftarrow xs, \dots]$ rather than $[e | x \leftarrow xs, \dots]$.

The qualifiers after the dot are

generators $p \leftarrow xs$, where p is a pattern and xs an expression of list type,
or

guards b , where b is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of $[e. x \leftarrow xs]$ is optimized to $map (\lambda x. e) xs$.

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

nonterminal *lc-qual* and *lc-quals*

syntax

-listcompr :: $'a \Rightarrow$ *lc-qual* \Rightarrow *lc-quals* \Rightarrow $'a$ list $([- . --)$

-lc-gen :: $'a \Rightarrow$ $'a$ list \Rightarrow *lc-qual* $(- \leftarrow -)$

-lc-test :: *bool* \Rightarrow *lc-qual* $(-)$

-lc-end :: *lc-quals* $([])$

-lc-quals :: *lc-qual* \Rightarrow *lc-quals* \Rightarrow *lc-quals* $(, --)$

syntax (ASCII)

-lc-gen :: $'a \Rightarrow$ $'a$ list \Rightarrow *lc-qual* $(- \leftarrow -)$

parse-translation \langle

let

val NilC = *Syntax.const const-syntax* $\langle Nil \rangle$;

```

val ConsC = Syntax.const const-syntax ⟨Cons⟩;
val mapC = Syntax.const const-syntax ⟨map⟩;
val concatC = Syntax.const const-syntax ⟨concat⟩;
val IfC = Syntax.const const-syntax ⟨If⟩;
val dummyC = Syntax.const const-syntax ⟨Pure.dummy-pattern⟩

fun single x = ConsC $ x $ NilC;

fun pat-tr ctxt p e opti = (* %x. case x of p => e | - => [] *)
  let
    (* FIXME proper name context!?! *)
    val x =
      Free (singleton (Name.variant-list (fold Term.add-free-names [p, e] [])) x,
dummyT);
    val e = if opti then single e else e;
    val case1 = Syntax.const syntax-const ⟨-case1⟩ $ p $ e;
    val case2 =
      Syntax.const syntax-const ⟨-case1⟩ $ dummyC $ NilC;
    val cs = Syntax.const syntax-const ⟨-case2⟩ $ case1 $ case2;
  in Syntax-Trans.abs-tr [x, Case-Translation.case-tr false ctxt [x, cs]] end;

fun pair-pat-tr (x as Free -) e = Syntax-Trans.abs-tr [x, e]
| pair-pat-tr (- $ p1 $ p2) e =
  Syntax.const const-syntax ⟨case-prod⟩ $ pair-pat-tr p1 (pair-pat-tr p2 e)
| pair-pat-tr dummy e = Syntax-Trans.abs-tr [Syntax.const -iddummy, e]

fun pair-pat ctxt (Const (const-syntax ⟨Pair⟩, -) $ s $ t) =
  pair-pat ctxt s andalso pair-pat ctxt t
| pair-pat ctxt (Free (s, -)) =
  let
    val thy = Proof-Context.theory-of ctxt;
    val s' = Proof-Context.intern-const ctxt s;
  in not (Sign.declared-const thy s') end
| pair-pat - t = (t = dummyC);

fun abs-tr ctxt p e opti =
  let val p = Term-Position.strip-positions p
  in if pair-pat ctxt p
    then (pair-pat-tr p e, true)
    else (pat-tr ctxt p e opti, false)
  end

fun lc-tr ctxt [e, Const (syntax-const ⟨-lc-test⟩, -) $ b, qs] =
  let
    val res =
      (case qs of
        Const (syntax-const ⟨-lc-end⟩, -) => single e
      | Const (syntax-const ⟨-lc-quals⟩, -) $ q $ qs => lc-tr ctxt [e, q, qs]);
  in IfC $ b $ res $ NilC end

```

```

| lc-tr ctxt
  [e, Const (syntax-const <-lc-gen>, -) $ p $ es,
   Const(syntax-const <-lc-end>, -)] =
  (case abs-tr ctxt p e true of
   (f, true) => mapC $ f $ es
  | (f, false) => concatC $ (mapC $ f $ es))
| lc-tr ctxt
  [e, Const (syntax-const <-lc-gen>, -) $ p $ es,
   Const (syntax-const <-lc-quals>, -) $ q $ qs] =
  let val e' = lc-tr ctxt [e, q, qs];
  in concatC $ (mapC $ (fst (abs-tr ctxt p e' false)) $ es) end;

in [(syntax-const <-listcompr>, lc-tr)] end
>

```

ML-val <

```

let
  val read = Syntax.read-term context o Syntax.implode-input;
  fun check s1 s2 =
    read s1 aconv read s2 orelse
    error (Check failed: ^
           quote (#1 (Input.source-content s1)) ^ Position.here-list [Input.pos-of s1,
Input.pos-of s2]);
in
  check <[(x,y,z). b]> <if b then [(x, y, z)] else []>;
  check <[(x,y,z). (x,-,y)←xs]> <map (λ(x,-,y). (x, y, z)) xs>;
  check <[e x y. (x,-)←xs, y←ys]> <concat (map (λ(x,-). map (λy. e x y) ys) xs)>;
  check <[(x,y,z). x<a, x>b]> <if x < a then if b < x then [(x, y, z)] else [] else
[]>;
  check <[(x,y,z). x←xs, x>b]> <concat (map (λx. if b < x then [(x, y, z)] else []
xs)>;
  check <[(x,y,z). x<a, x←xs]> <if x < a then map (λx. (x, y, z)) xs else []>;
  check <[(x,y). Cons True x ← xs]>
  <concat (map (λxa. case xa of [] => [] | True # x => [(x, y)] | False # x => []
xs)>;
  check <[(x,y,z). Cons x [] ← xs]>
  <concat (map (λxa. case xa of [] => [] | [x] => [(x, y, z)] | x # aa # lista =>
[]) xs)>;
  check <[(x,y,z). x<a, x>b, x=d]>
  <if x < a then if b < x then if x = d then [(x, y, z)] else [] else [] else []>;
  check <[(x,y,z). x<a, x>b, y←ys]>
  <if x < a then if b < x then map (λy. (x, y, z)) ys else [] else []>;
  check <[(x,y,z). x<a, (-,x)←xs,y>b]>
  <if x < a then concat (map (λ(-,x). if b < y then [(x, y, z)] else [] xs) else []>;
  check <[(x,y,z). x<a, x←xs, y←ys]>
  <if x < a then concat (map (λx. map (λy. (x, y, z)) ys) xs) else []>;
  check <[(x,y,z). x←xs, x>b, y<a]>
  <concat (map (λx. if b < x then if y < a then [(x, y, z)] else [] else [] xs)>;
  check <[(x,y,z). x←xs, x>b, y←ys]>

```

```

    <concat (map (λx. if b < x then map (λy. (x, y, z)) ys else []) xs)>;
    check <[(x,y,z). x←xs, (y,-)←ys,y>x]>
    <concat (map (λx. concat (map (λ(y,-). if x < y then [(x, y, z)] else []) ys))
xs)>;
    check <[(x,y,z). x←xs, y←ys,z←zs]>
    <concat (map (λx. concat (map (λy. map (λz. (x, y, z)) zs) ys)) xs)>
  end;
>

```

ML <

(* Simproc for rewriting list comprehensions applied to List.set to set comprehension. *)

signature LIST-TO-SET-COMPREHENSION =

sig

val simproc : Proof.context -> cterm -> thm option

end

structure List-to-Set-Comprehension : LIST-TO-SET-COMPREHENSION =
struct

(* conversion *)

fun all-exists-conv cv ctxt ct =

(case Thm.term-of ct of

Const (**const-name** <Ex>, -) \$ Abs - =>

Conv.arg-conv (Conv.abs-conv (all-exists-conv cv o #2) ctxt) ct

| - => cv ctxt ct)

fun all-but-last-exists-conv cv ctxt ct =

(case Thm.term-of ct of

Const (**const-name** <Ex>, -) \$ Abs (-, -, Const (**const-name** <Ex>, -) \$ -) =>

Conv.arg-conv (Conv.abs-conv (all-but-last-exists-conv cv o #2) ctxt) ct

| - => cv ctxt ct)

fun Collect-conv cv ctxt ct =

(case Thm.term-of ct of

Const (**const-name** <Collect>, -) \$ Abs - => Conv.arg-conv (Conv.abs-conv cv

ctxt) ct

| - => raise CTERM (Collect-conv, [ct]))

fun rewr-conv' th = Conv.rewr-conv (mk-meta-eq th)

fun conjunct-assoc-conv ct =

Conv.try-conv

(rewr-conv' @ {thm conj-assoc} then-conv HOLogic.conj-conv Conv.all-conv conjunct-assoc-conv) ct

fun right-hand-set-comprehension-conv conv ctxt =

```

HOLLogic.Trueprop-conv (HOLLogic.eq-conv Conv.all-conv
  (Collect-conv (all-exists-conv conv o #2) ctxt))

```

(* term abstraction of list comprehension patterns *)

```

datatype termlets = If | Case of typ * int

```

local

```

val set-Nil-I = @{lemma set [] = {x. False} by (simp add: empty-def [symmetric])}
val set-singleton = @{lemma set [a] = {x. x = a} by simp}
val inst-Collect-mem-eq = @{lemma set A = {x. x ∈ set A} by simp}
val del-refl-eq = @{lemma (t = t ∧ P) ≡ P by simp}

```

```

fun mk-set T = Const (const-name ⟨set⟩, HOLLogic.listT T --> HOLLogic.mk-setT
  T)

```

```

fun dest-set (Const (const-name ⟨set⟩, -) $ xs) = xs

```

```

fun dest-singleton-list (Const (const-name ⟨Cons⟩, -) $ t $ (Const (const-name ⟨Nil⟩,
  -))) = t

```

```

  | dest-singleton-list t = raise TERM (dest-singleton-list, [t])

```

(* We check that one case returns a singleton list and all other cases
return [], and return the index of the one singleton list case. *)

```

fun possible-index-of-singleton-case cases =

```

```

  let

```

```

    fun check (i, case-t) s =

```

```

      (case strip-abs-body case-t of

```

```

        (Const (const-name ⟨Nil⟩, -)) => s

```

```

        | - => (case s of SOME NONE => SOME (SOME i) | - => NONE))

```

```

    in

```

```

      fold-index check cases (SOME NONE) |> the-default NONE

```

```

    end

```

(* returns condition continuing term option *)

```

fun dest-if (Const (const-name ⟨If⟩, -) $ cond $ then-t $ Const (const-name ⟨Nil⟩,
  -)) =

```

```

  SOME (cond, then-t)

```

```

  | dest-if - = NONE

```

(* returns (case-expr type index chosen-case constr-name) option *)

```

fun dest-case ctxt case-term =

```

```

  let

```

```

    val (case-const, args) = strip-comb case-term

```

```

  in

```

```

    (case try dest-Const case-const of

```

```

      SOME (c, T) =>

```

```

        (case Ctr-Sugar.ctr-sugar-of-case ctxt c of

```

```

    SOME { ctrs, ... } =>
      (case possible-index-of-singleton-case (fst (split-last args)) of
        SOME i =>
          let
            val constr-names = map (fst o dest-Const) ctrs
            val (Ts, -) = strip-type T
            val T' = List.last Ts
            in SOME (List.last args, T', i, nth args i, nth constr-names i) end
          | NONE => NONE)
        | NONE => NONE)
      | NONE => NONE)
  end

fun tac ctxt [] =
  resolve-tac ctxt [set-singleton] 1 ORELSE
  resolve-tac ctxt [inst-Collect-mem-eq] 1
| tac ctxt (If :: cont) =
  Splitter.split-tac ctxt @ { thms if-split } 1
  THEN resolve-tac ctxt @ { thms conjI } 1
  THEN resolve-tac ctxt @ { thms impI } 1
  THEN Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
    CONVERSION (right-hand-set-comprehension-conv (K
      (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @ { thm Eq-TrueI })))
    Conv.all-conv
      then-conv
        rewr-conv' @ { lemma (True  $\wedge$  P) = P by simp })) ctxt') 1) ctxt 1
  THEN tac ctxt cont
  THEN resolve-tac ctxt @ { thms impI } 1
  THEN Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
    CONVERSION (right-hand-set-comprehension-conv (K
      (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @ { thm
Eq-FalseI })))
    Conv.all-conv
      then-conv rewr-conv' @ { lemma (False  $\wedge$  P) = False by simp })) ctxt') 1)
  ctxt 1
  THEN resolve-tac ctxt [set-Nil-I] 1
| tac ctxt (Case (T, i) :: cont) =
  let
    val SOME {injects, distincts, case-thms, split, ...} =
      Ctr-Sugar.ctr-sugar-of ctxt (fst (dest-Type T))
  in
    (* do case distinction *)
    Splitter.split-tac ctxt [split] 1
    THEN EVERY (map-index (fn (i', -) =>
      (if i' < length case-thms - 1 then resolve-tac ctxt @ { thms conjI } 1 else
all-tac)
    THEN REPEAT-DETERM (resolve-tac ctxt @ { thms allI } 1)
    THEN resolve-tac ctxt @ { thms impI } 1
    THEN (if i' = i then
      (* continue recursively *)

```

```

      Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
        CONVERSION (Thm.eta-conversion then-conv right-hand-set-comprehension-conv
(K
  ((HOLogic.conj-conv
    (HOLogic.eq-conv Conv.all-conv (rewr-conv' (List.last prems))
then-conv
  (Conv.try-conv (Conv.rewrs-conv (map mk-meta-eq injects))))
  Conv.all-conv)
  then-conv (Conv.try-conv (Conv.rewr-conv del-refl-eq))
  then-conv conjunct-assoc-conv)) ctxt'
then-conv
  (HOLogic.Trueprop-conv
  (HOLogic.eq-conv Conv.all-conv (Collect-conv (fn (-, ctxt'') =>
    Conv.repeat-conv
    (all-but-last-exists-conv
      (K (rewr-conv'
        @{\lemma (\exists x. x = t \wedge P x) = P t by simp})) ctxt''))
  ctxt')))) 1) ctxt 1
  THEN tac ctxt cont
else
  Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
    CONVERSION
    (right-hand-set-comprehension-conv (K
      (HOLogic.conj-conv
        ((HOLogic.eq-conv Conv.all-conv
          (rewr-conv' (List.last prems))) then-conv
          (Conv.rewrs-conv (map (fn th => th RS @{\thm Eq-FalseI})
distincts))))
      Conv.all-conv then-conv
      (rewr-conv' @{\lemma (False \wedge P) = False by simp}))) ctxt'
then-conv
  HOLogic.Trueprop-conv
  (HOLogic.eq-conv Conv.all-conv
    (Collect-conv (fn (-, ctxt'') =>
      Conv.repeat-conv
      (Conv.bottom-conv
        (K (rewr-conv' @{\lemma (\exists x. P) = P by simp})) ctxt''))
  ctxt')))) 1) ctxt 1
  THEN resolve-tac ctxt [set-Nil-I] 1)) case-thms)
end

in

fun simproc ctxt redex =
  let
    fun make-inner-eqs bound-vs Tis eqs t =
      (case dest-case ctxt t of
        SOME (x, T, i, cont, constr-name) =>
          let

```



```

    val (vs, body) = strip-abs (Envir.eta-long (map snd bound-vs) cont)
    val x' = incr-boundvars (length vs) x
    val eqs' = map (incr-boundvars (length vs)) eqs
    val constr-t =
      list-comb
        (Const (constr-name, map snd vs ----> T), map Bound (((length
vs) - 1) downto 0))
    val constr-eq = Const (constr-name <HOL.eq>, T --> T --> typ <bool>)
$ constr-t $ x'
  in
    make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs')
body
  end
  | NONE =>
    (case dest-if t of
      SOME (condition, cont) => make-inner-eqs bound-vs (If :: Tis) (condition
:: eqs) cont
    | NONE =>
      if null eqs then NONE (*no rewriting, nothing to be done*)
      else
        let
          val Type (type-name <list>, [rT]) = fastype-of1 (map snd bound-vs, t)
          val pat-eq =
            (case try dest-singleton-list t of
              SOME t' =>
                Const (constr-name <HOL.eq>, rT --> rT --> typ <bool>) $
                Bound (length bound-vs) $ t'
            | NONE =>
                Const (constr-name <Set.member>, rT --> HOLogic.mk-setT
rT --> typ <bool>) $
                Bound (length bound-vs) $ (mk-set rT $ t))
          val reverse-bounds = curry subst-bounds
            ((map Bound ((length bound-vs - 1) downto 0)) @ [Bound (length
bound-vs)])
          val eqs' = map reverse-bounds eqs
          val pat-eq' = reverse-bounds pat-eq
          val inner-t =
            fold (fn (-, T) => fn t => HOLogic.exists-const T $ absdummy T t)
              (rev bound-vs) (fold (curry HOLogic.mk-conj) eqs' pat-eq')
          val lhs = Thm.term-of redex
          val rhs = HOLogic.mk-Collect (x, rT, inner-t)
          val rewrite-rule-t = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs))
        in
          SOME
            ((Goal.prove ctxt [] [] rewrite-rule-t
              (fn {context = ctxt', ...} => tac ctxt' (rev Tis))) RS @{thm
eq-reflection})
            end))
  in

```

```

    make-inner-eqs [] [] [] (dest-set (Thm.term-of redex))
  end

end

end
>

```

```

simproc-setup list-to-set-comprehension (set xs) =
  ⟨K List-to-Set-Comprehension.simproc⟩

```

```

code-datatype set coset
hide-const (open) coset

```

67.1.2 [] and (#)

```

lemma not-Cons-self [simp]:
  xs ≠ x # xs
by (induct xs) auto

```

```

lemma not-Cons-self2 [simp]: x # xs ≠ xs
by (rule not-Cons-self [symmetric])

```

```

lemma neq-Nil-conv: (xs ≠ []) = (∃ y ys. xs = y # ys)
by (induct xs) auto

```

```

lemma tl-Nil: tl xs = [] ⟷ xs = [] ∨ (∃ x. xs = [x])
by (cases xs) auto

```

```

lemmas Nil-tl = tl-Nil[THEN eq-iff-swap]

```

```

lemma length-induct:
  (∧ xs. ∀ ys. length ys < length xs ⟶ P ys ⟹ P xs) ⟹ P xs
by (fact measure-induct)

```

```

lemma induct-list012:
  [P []; ∧ x. P [x]; ∧ x y zs. [P zs; P (y # zs)] ⟹ P (x # y # zs)] ⟹ P xs
by induction-schema (pat-completeness, lexicographic-order)

```

```

lemma list-nonempty-induct [consumes 1, case-names single cons]:
  [xs ≠ []; ∧ x. P [x]; ∧ x xs. xs ≠ [] ⟹ P xs ⟹ P (x # xs)] ⟹ P xs
by(induction xs rule: induct-list012) auto

```

```

lemma inj-split-Cons: inj-on (λ(xs, n). n#xs) X
  by (auto intro!: inj-onI)

```

```

lemma inj-on-Cons1 [simp]: inj-on ((#) x) A
by(simp add: inj-on-def)

```

67.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: $\text{length } (xs \text{ @ } ys) = \text{length } xs + \text{length } ys$
by (*induct xs*) *auto*

lemma *length-map* [*simp*]: $\text{length } (\text{map } f \text{ } xs) = \text{length } xs$
by (*induct xs*) *auto*

lemma *length-rev* [*simp*]: $\text{length } (\text{rev } xs) = \text{length } xs$
by (*induct xs*) *auto*

lemma *length-tl* [*simp*]: $\text{length } (\text{tl } xs) = \text{length } xs - 1$
by (*cases xs*) *auto*

lemma *length-0-conv* [*iff*]: $(\text{length } xs = 0) = (xs = [])$
by (*induct xs*) *auto*

lemma *length-greater-0-conv* [*iff*]: $(0 < \text{length } xs) = (xs \neq [])$
by (*induct xs*) *auto*

lemma *length-pos-if-in-set*: $x \in \text{set } xs \implies \text{length } xs > 0$
by *auto*

lemma *length-Suc-conv*: $(\text{length } xs = \text{Suc } n) = (\exists y \text{ } ys. xs = y \# ys \wedge \text{length } ys = n)$
by (*induct xs*) *auto*

lemmas *Suc-length-conv* = *length-Suc-conv*[*THEN eq-iff-swap*]

lemma *Suc-le-length-iff*:
 $(\text{Suc } n \leq \text{length } xs) = (\exists x \text{ } ys. xs = x \# ys \wedge n \leq \text{length } ys)$
by (*metis Suc-le-D*[*of n*] *Suc-le-mono*[*of n*] *Suc-length-conv*[*of - xs*])

lemma *impossible-Cons*: $\text{length } xs \leq \text{length } ys \implies xs = x \# ys = \text{False}$
by (*induct xs*) *auto*

lemma *list-induct2* [*consumes 1, case-names Nil Cons*]:
 $\text{length } xs = \text{length } ys \implies P [] [] \implies$
 $(\bigwedge x \text{ } xs \text{ } y \text{ } ys. \text{length } xs = \text{length } ys \implies P \text{ } xs \text{ } ys \implies P (x \# xs) (y \# ys))$
 $\implies P \text{ } xs \text{ } ys$

proof (*induct xs arbitrary: ys*)
case (*Cons x xs ys*) **then show** ?*case* **by** (*cases ys*) *simp-all*
qed *simp*

lemma *list-induct3* [*consumes 2, case-names Nil Cons*]:
 $\text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P [] [] [] \implies$
 $(\bigwedge x \text{ } xs \text{ } y \text{ } ys \text{ } z \text{ } zs. \text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P \text{ } xs \text{ } ys \text{ } zs$
 $\implies P (x \# xs) (y \# ys) (z \# zs))$

$\implies P\ xs\ ys\ zs$
proof (induct xs arbitrary: ys zs)
 case Nil then show ?case by simp
next
 case (Cons x xs ys zs) then show ?case by (cases ys, simp-all)
 (cases zs, simp-all)
qed

lemma list-induct4 [consumes 3, case-names Nil Cons]:
 $length\ xs = length\ ys \implies length\ ys = length\ zs \implies length\ zs = length\ ws \implies$
 $P\ []\ []\ [] \implies (\bigwedge x\ xs\ y\ ys\ z\ zs\ w\ ws.\ length\ xs = length\ ys \implies$
 $length\ ys = length\ zs \implies length\ zs = length\ ws \implies P\ xs\ ys\ zs\ ws \implies$
 $P\ (x\#\!xs)\ (y\#\!ys)\ (z\#\!zs)\ (w\#\!ws)) \implies P\ xs\ ys\ zs\ ws$
proof (induct xs arbitrary: ys zs ws)
 case Nil then show ?case by simp
next
 case (Cons x xs ys zs ws) then show ?case by ((cases ys, simp-all), (cases
 zs, simp-all)) (cases ws, simp-all)
qed

lemma list-induct2':
 $[P\ []\ [];$
 $\bigwedge x\ xs.\ P\ (x\#\!xs)\ [];$
 $\bigwedge y\ ys.\ P\ []\ (y\#\!ys);$
 $\bigwedge x\ xs\ y\ ys.\ P\ xs\ ys \implies P\ (x\#\!xs)\ (y\#\!ys)]$
 $\implies P\ xs\ ys$
by (induct xs arbitrary: ys) (case-tac x, auto)+

lemma list-all2-iff:
 $list\ all2\ P\ xs\ ys \iff length\ xs = length\ ys \wedge (\forall (x, y) \in set\ (zip\ xs\ ys).\ P\ x\ y)$
by (induct xs ys rule: list-induct2') auto

lemma neq-if-length-neq: $length\ xs \neq length\ ys \implies (xs = ys) == False$
by (rule Eq-FalseI) auto

67.1.4 @ – append

global-interpretation append: monoid append Nil

proof
 fix xs ys zs :: 'a list
 show (xs @ ys) @ zs = xs @ (ys @ zs)
 by (induct xs) simp-all
 show xs @ [] = xs
 by (induct xs) simp-all
qed simp

lemma append-assoc [simp]: $(xs @ ys) @ zs = xs @ (ys @ zs)$
by (fact append.assoc)

lemma *append-Nil2*: $xs @ [] = xs$
by (*fact append.right-neutral*)

lemma *append-is-Nil-conv* [*iff*]: $(xs @ ys = []) = (xs = [] \wedge ys = [])$
by (*induct xs*) *auto*

lemmas *Nil-is-append-conv* [*iff*] = *append-is-Nil-conv*[*THEN eq-iff-swap*]

lemma *append-self-conv* [*iff*]: $(xs @ ys = xs) = (ys = [])$
by (*induct xs*) *auto*

lemmas *self-append-conv* [*iff*] = *append-self-conv*[*THEN eq-iff-swap*]

lemma *append-eq-append-conv* [*simp*]:
 $length\ xs = length\ ys \vee length\ us = length\ vs$
 $\implies (xs @ us = ys @ vs) = (xs = ys \wedge us = vs)$
by (*induct xs arbitrary: ys; case-tac ys; force*)

lemma *append-eq-append-conv2*: $(xs @ ys = zs @ ts) =$
 $(\exists us. xs = zs @ us \wedge us @ ys = ts \vee xs @ us = zs \wedge ys = us @ ts)$

proof (*induct xs arbitrary: ys zs ts*)

case (*Cons x xs*)

then show *?case*

by (*cases zs*) *auto*

qed *fastforce*

lemma *same-append-eq* [*iff, induct-simp*]: $(xs @ ys = xs @ zs) = (ys = zs)$
by *simp*

lemma *append1-eq-conv* [*iff*]: $(xs @ [x] = ys @ [y]) = (xs = ys \wedge x = y)$
by *simp*

lemma *append-same-eq* [*iff, induct-simp*]: $(ys @ xs = zs @ xs) = (ys = zs)$
by *simp*

lemma *append-self-conv2* [*iff*]: $(xs @ ys = ys) = (xs = [])$
using *append-same-eq* [*of - - []*] **by** *auto*

lemmas *self-append-conv2* [*iff*] = *append-self-conv2*[*THEN eq-iff-swap*]

lemma *hd-Cons-tl*: $xs \neq [] \implies hd\ xs \# tl\ xs = xs$
by (*fact list.collapse*)

lemma *hd-append*: $hd\ (xs @ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
by (*induct xs*) *auto*

lemma *hd-append2* [*simp*]: $xs \neq [] \implies hd\ (xs @ ys) = hd\ xs$
by (*simp add: hd-append split: list.split*)

lemma *tl-append*: $tl (xs @ ys) = (case\ xs\ of\ [] \Rightarrow tl\ ys \mid z\#\!zs \Rightarrow zs @ ys)$
by (*simp split: list.split*)

lemma *tl-append2* [*simp*]: $xs \neq [] \Longrightarrow tl (xs @ ys) = tl\ xs @ ys$
by (*simp add: tl-append split: list.split*)

lemma *tl-append-if*: $tl (xs @ ys) = (if\ xs = []\ then\ tl\ ys\ else\ tl\ xs @ ys)$
by (*simp*)

lemma *Cons-eq-append-conv*: $x\#\!xs = ys@zs =$
 $(ys = [] \wedge x\#\!xs = zs \vee (\exists\ ys'. x\#\!ys' = ys \wedge xs = ys'@zs))$
by(*cases ys*) *auto*

lemma *append-eq-Cons-conv*: $(ys@zs = x\#\!xs) =$
 $(ys = [] \wedge zs = x\#\!xs \vee (\exists\ ys'. ys = x\#\!ys' \wedge ys'@zs = xs))$
by(*cases ys*) *auto*

lemma *longest-common-prefix*:
 $\exists\ ps\ xs'\ ys'. xs = ps @ xs' \wedge ys = ps @ ys'$
 $\wedge (xs' = [] \vee ys' = [] \vee hd\ xs' \neq hd\ ys')$
by (*induct xs ys rule: list-induct2'*)
(blast, blast, blast,
metis (no-types, opaque-lifting) append-Cons append-Nil list.sel(1))

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: $xs = ys \Longrightarrow xs = [] @ ys$
by *simp*

lemma *Cons-eq-appendI*: $[x \# xs1 = ys; xs = xs1 @ zs] \Longrightarrow x \# xs = ys @ zs$
by *auto*

lemma *append-eq-appendI*: $[xs @ xs1 = zs; ys = xs1 @ us] \Longrightarrow xs @ ys = zs @ us$
by *auto*

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

simproc-setup *list-eq* $((xs::'a\ list) = ys) = \langle$
let
 $fun\ last\ (cons\ as\ Const\ (\mathbf{const-name}\ \langle Cons \rangle, -)\ \$ - \$ xs) =$
 $(case\ xs\ of\ Const\ (\mathbf{const-name}\ \langle Nil \rangle, -) \Rightarrow cons \mid - \Rightarrow last\ xs)$
 $\mid last\ (Const\ (\mathbf{const-name}\ \langle append \rangle, -)\ \$ - \$ ys) = last\ ys$
 $\mid last\ t = t;$
 $fun\ list1\ (Const\ (\mathbf{const-name}\ \langle Cons \rangle, -)\ \$ - \$ Const\ (\mathbf{const-name}\ \langle Nil \rangle, -)) =$
 $true$
 $\mid list1\ - = false;$

```

fun butlast ((cons as Const(const-name ⟨Cons⟩, -) $ x) $ xs) =
  (case xs of Const (const-name ⟨Nil⟩, -) => xs | - => cons $ butlast xs)
| butlast ((app as Const (const-name ⟨append⟩, -) $ xs) $ ys) = app $ butlast
ys
| butlast xs = Const(const-name ⟨Nil⟩, fastype-of xs);

val rearr-ss =
  simpset-of (put-simpset HOL-basic-ss context
  addsimps [@[thm append-assoc], @[thm append-Nil], @[thm append-Cons]]);

fun list-eq ctxt (F as (eq as Const(-,eqT)) $ lhs $ rhs) =
  let
    val lastl = last lhs and lastr = last rhs;
    fun rearr conv =
      let
        val lhs1 = butlast lhs and rhs1 = butlast rhs;
        val Type(-,listT::-) = eqT
        val appT = [listT,listT] ----> listT
        val app = Const(const-name ⟨append⟩,appT)
        val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
        val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (F,F2));
        val thm = Goal.prove ctxt [] [] eq
          (K (simp-tac (put-simpset rearr-ss ctxt) 1));
        in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;
      in
        if list1 lastl andalso list1 lastr then rearr @[thm append1-eq-conv]
        else if lastl aconv lastr then rearr @[thm append-same-eq]
        else NONE
      end;
    in K (fn ctxt => fn ct => list-eq ctxt (Thm.term-of ct)) end
  >

```

67.1.5 map

lemma *hd-map*: $xs \neq [] \implies \text{hd} (\text{map } f \text{ } xs) = f (\text{hd } xs)$
by (cases xs) simp-all

lemma *map-tl*: $\text{map } f (\text{tl } xs) = \text{tl} (\text{map } f \text{ } xs)$
by (cases xs) simp-all

lemma *map-ext*: $(\bigwedge x. x \in \text{set } xs \implies f x = g x) \implies \text{map } f \text{ } xs = \text{map } g \text{ } xs$
by (induct xs) simp-all

lemma *map-ident* [simp]: $\text{map } (\lambda x. x) = (\lambda xs. xs)$
by (rule ext, induct-tac xs) auto

lemma *map-append* [simp]: $\text{map } f (xs @ ys) = \text{map } f \text{ } xs @ \text{map } f \text{ } ys$
by (induct xs) auto

lemma *map-map [simp]*: $\text{map } f (\text{map } g \text{ } xs) = \text{map } (f \circ g) \text{ } xs$
by (*induct xs*) *auto*

lemma *map-comp-map [simp]*: $((\text{map } f) \circ (\text{map } g)) = \text{map}(f \circ g)$
by (*rule ext*) *simp*

lemma *rev-map*: $\text{rev } (\text{map } f \text{ } xs) = \text{map } f (\text{rev } xs)$
by (*induct xs*) *auto*

lemma *map-eq-conv [simp]*: $(\text{map } f \text{ } xs = \text{map } g \text{ } xs) = (\forall x \in \text{set } xs. f \text{ } x = g \text{ } x)$
by (*induct xs*) *auto*

lemma *map-cong [fundef-cong]*:
 $xs = ys \implies (\bigwedge x. x \in \text{set } ys \implies f \text{ } x = g \text{ } x) \implies \text{map } f \text{ } xs = \text{map } g \text{ } ys$
by *simp*

lemma *map-is-Nil-conv [iff]*: $(\text{map } f \text{ } xs = []) = (xs = [])$
by (*rule list.map-disc-iff*)

lemmas *Nil-is-map-conv [iff] = map-is-Nil-conv [THEN eq-iff-swap]*

lemma *map-eq-Cons-conv*:
 $(\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (\exists z \text{ } zs. xs = z \# zs \wedge f \text{ } z = f \text{ } z \wedge \text{map } f \text{ } zs = \text{map } f \text{ } zs)$
by (*cases xs*) *auto*

lemma *Cons-eq-map-conv*:
 $(x \# xs = \text{map } f \text{ } ys) = (\exists z \text{ } zs. ys = z \# zs \wedge x = f \text{ } z \wedge xs = \text{map } f \text{ } zs)$
by (*cases ys*) *auto*

lemmas *map-eq-Cons-D = map-eq-Cons-conv [THEN iffD1]*

lemmas *Cons-eq-map-D = Cons-eq-map-conv [THEN iffD1]*

declare *map-eq-Cons-D [dest!] Cons-eq-map-D [dest!]*

lemma *ex-map-conv*:
 $(\exists xs. ys = \text{map } f \text{ } xs) = (\forall y \in \text{set } ys. \exists x. y = f \text{ } x)$
by(*induct ys, auto simp add: Cons-eq-map-conv*)

lemma *map-eq-imp-length-eq*:
assumes $\text{map } f \text{ } xs = \text{map } g \text{ } ys$
shows $\text{length } xs = \text{length } ys$
using *assms*

proof (*induct ys arbitrary: xs*)

case *Nil* **then show** *?case* **by** *simp*

next

case (*Cons y ys*) **then obtain** *z zs* **where** $xs = z \# zs$ **by** *auto*

from *Cons xs* **have** $\text{map } f \text{ } zs = \text{map } g \text{ } ys$ **by** *simp*

with *Cons* **have** $\text{length } zs = \text{length } ys$ **by** *blast*

with *xs* **show** *?case* **by** *simp*

qed

lemma *map-inj-on*:

assumes *map*: $\text{map } f \text{ } xs = \text{map } f \text{ } ys$ **and** *inj*: $\text{inj-on } f \text{ } (\text{set } xs \text{ } Un \text{ } \text{set } ys)$

shows $xs = ys$

using *map-eq-imp-length-eq* [*OF map*] *assms*

proof (*induct rule: list-induct2*)

case (*Cons x xs y ys*)

then show *?case*

by (*auto intro: sym*)

qed *auto*

lemma *inj-on-map-eq-map*:

$\text{inj-on } f \text{ } (\text{set } xs \text{ } Un \text{ } \text{set } ys) \implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$

by (*blast dest:map-inj-on*)

lemma *map-injective*:

$\text{map } f \text{ } xs = \text{map } f \text{ } ys \implies \text{inj } f \implies xs = ys$

by (*induct ys arbitrary: xs*) (*auto dest!:injD*)

lemma *inj-map-eq-map[simp]*: $\text{inj } f \implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$

by (*blast dest:map-injective*)

lemma *inj-mapI*: $\text{inj } f \implies \text{inj } (\text{map } f)$

by (*iprover dest: map-injective injD intro: inj-onI*)

lemma *inj-mapD*: $\text{inj } (\text{map } f) \implies \text{inj } f$

by (*metis (no-types, opaque-lifting) injI list.inject list.simps(9) the-inv-f-f*)

lemma *inj-map[iff]*: $\text{inj } (\text{map } f) = \text{inj } f$

by (*blast dest: inj-mapD intro: inj-mapI*)

lemma *inj-on-mapI*: $\text{inj-on } f \text{ } (\bigcup (\text{set } 'A)) \implies \text{inj-on } (\text{map } f) \text{ } A$

by (*blast intro:inj-onI dest:inj-onD map-inj-on*)

lemma *map-idI*: $(\bigwedge x. x \in \text{set } xs \implies f \text{ } x = x) \implies \text{map } f \text{ } xs = xs$

by (*induct xs, auto*)

lemma *map-fun-upd [simp]*: $y \notin \text{set } xs \implies \text{map } (f(y:=v)) \text{ } xs = \text{map } f \text{ } xs$

by (*induct xs*) *auto*

lemma *map-fst-zip[simp]*:

$\text{length } xs = \text{length } ys \implies \text{map } \text{fst } (\text{zip } xs \text{ } ys) = xs$

by (*induct rule:list-induct2, simp-all*)

lemma *map-snd-zip[simp]*:

$\text{length } xs = \text{length } ys \implies \text{map } \text{snd } (\text{zip } xs \text{ } ys) = ys$

by (*induct rule:list-induct2, simp-all*)

lemma *map-fst-zip-take*:

$map\ fst\ (zip\ xs\ ys) = take\ (min\ (length\ xs)\ (length\ ys))\ xs$
by $(induct\ xs\ ys\ rule:\ list-induct2')$ *simp-all*

lemma *map-snd-zip-take*:

$map\ snd\ (zip\ xs\ ys) = take\ (min\ (length\ xs)\ (length\ ys))\ ys$
by $(induct\ xs\ ys\ rule:\ list-induct2')$ *simp-all*

lemma *map2-map-map*: $map2\ h\ (map\ f\ xs)\ (map\ g\ xs) = map\ (\lambda x.\ h\ (f\ x)\ (g\ x))\ xs$

by $(induction\ xs)\ (auto)$

functor *map*: *map*

by $(simp-all\ add:\ id-def)$

declare *map.id* [*simp*]

67.1.6 *rev*

lemma *rev-append* [*simp*]: $rev\ (xs\ @\ ys) = rev\ ys\ @\ rev\ xs$
by $(induct\ xs)\ auto$

lemma *rev-rev-ident* [*simp*]: $rev\ (rev\ xs) = xs$
by $(induct\ xs)\ auto$

lemma *rev-swap*: $(rev\ xs = ys) = (xs = rev\ ys)$
by *auto*

lemma *rev-is-Nil-conv* [*iff*]: $(rev\ xs = []) = (xs = [])$
by $(induct\ xs)\ auto$

lemmas *Nil-is-rev-conv* [*iff*] = *rev-is-Nil-conv*[*THEN eq-iff-swap*]

lemma *rev-singleton-conv* [*simp*]: $(rev\ xs = [x]) = (xs = [x])$
by $(cases\ xs)\ auto$

lemma *singleton-rev-conv* [*simp*]: $([x] = rev\ xs) = ([x] = xs)$
by $(cases\ xs)\ auto$

lemma *rev-is-rev-conv* [*iff*]: $(rev\ xs = rev\ ys) = (xs = ys)$

proof $(induct\ xs\ arbitrary:\ ys)$

case *Nil*

then show *?case* **by** *force*

next

case *Cons*

then show *?case* **by** $(cases\ ys)\ auto$

qed

lemma *inj-on-rev*[*iff*]: *inj-on* *rev A*

by $(simp\ add:\ inj-on-def)$

lemma *rev-induct* [*case-names Nil snoc*]:
assumes $P []$ **and** $\bigwedge x xs. P xs \implies P (xs @ [x])$
shows $P xs$
proof –
have $P (rev (rev xs))$
by (*rule-tac list = rev xs in list.induct, simp-all add: assms*)
then show *?thesis* **by** *simp*
qed

lemma *rev-exhaust* [*case-names Nil snoc*]:
 $(xs = [] \implies P) \implies (\bigwedge ys y. xs = ys @ [y] \implies P) \implies P$
by (*induct xs rule: rev-induct*) *auto*

lemmas *rev-cases = rev-exhaust*

lemma *rev-nonempty-induct* [*consumes 1, case-names single snoc*]:
assumes $xs \neq []$
and *single*: $\bigwedge x. P [x]$
and *snoc'*: $\bigwedge x xs. xs \neq [] \implies P xs \implies P (xs@[x])$
shows $P xs$
using $\langle xs \neq [] \rangle$ **proof** (*induct xs rule: rev-induct*)
case (*snoc x xs*) **then show** *?case*
proof (*cases xs*)
case *Nil* **thus** *?thesis* **by** (*simp add: single*)
next
case *Cons* **with** *snoc* **show** *?thesis* **by** (*fastforce intro!: snoc'*)
qed
qed *simp*

lemma *rev-eq-Cons-iff*[*iff*]: $(rev xs = y\#ys) = (xs = rev ys @ [y])$
by(*rule rev-cases[of xs]*) *auto*

lemma *length-Suc-conv-rev*: $(length xs = Suc n) = (\exists y ys. xs = ys @ [y] \wedge length ys = n)$
by (*induct xs rule: rev-induct*) *auto*

67.1.7 set

declare *list.set*[*code-post*] — pretty output

lemma *finite-set* [*iff*]: *finite (set xs)*
by (*induct xs*) *auto*

lemma *set-append* [*simp*]: $set (xs @ ys) = (set xs \cup set ys)$
by (*induct xs*) *auto*

lemma *hd-in-set*[*simp*]: $xs \neq [] \implies hd xs \in set xs$
by(*cases xs*) *auto*

lemma *set-subset-Cons*: $set\ xs \subseteq set\ (x \# xs)$
by *auto*

lemma *set-ConsD*: $y \in set\ (x \# xs) \implies y=x \vee y \in set\ xs$
by *auto*

lemma *set-empty [iff]*: $(set\ xs = \{\}) = (xs = [])$
by (*induct xs*) *auto*

lemmas *set-empty2 [iff] = set-empty [THEN eq-iff-swap]*

lemma *set-rev [simp]*: $set\ (rev\ xs) = set\ xs$
by (*induct xs*) *auto*

lemma *set-map [simp]*: $set\ (map\ f\ xs) = f'(set\ xs)$
by (*induct xs*) *auto*

lemma *set-filter [simp]*: $set\ (filter\ P\ xs) = \{x. x \in set\ xs \wedge P\ x\}$
by (*induct xs*) *auto*

lemma *set-upt [simp]*: $set[i..<j] = \{i..<j\}$
by (*induct j*) *auto*

lemma *split-list*: $x \in set\ xs \implies \exists\ ys\ zs. xs = ys @ x \# zs$
proof (*induct xs*)
 case *Nil* **thus** ?*case* **by** *simp*
next
 case *Cons* **thus** ?*case* **by** (*auto intro: Cons-eq-appendI*)
qed

lemma *in-set-conv-decomp*: $x \in set\ xs \longleftrightarrow (\exists\ ys\ zs. xs = ys @ x \# zs)$
by (*auto elim: split-list*)

lemma *split-list-first*: $x \in set\ xs \implies \exists\ ys\ zs. xs = ys @ x \# zs \wedge x \notin set\ ys$
proof (*induct xs*)
 case *Nil* **thus** ?*case* **by** *simp*
next
 case (*Cons a xs*)
 show ?*case*
 proof *cases*
 assume $x = a$ **thus** ?*case* **using** *Cons* **by** *fastforce*
 next
 assume $x \neq a$ **thus** ?*case* **using** *Cons* **by** (*fastforce intro!: Cons-eq-appendI*)
 qed
qed

lemma *in-set-conv-decomp-first*:

$(x \in \text{set } xs) = (\exists ys \ zs. \ xs = ys @ x \# zs \wedge x \notin \text{set } ys)$
by (*auto dest!*: *split-list-first*)

lemma *split-list-last*: $x \in \text{set } xs \implies \exists ys \ zs. \ xs = ys @ x \# zs \wedge x \notin \text{set } zs$
proof (*induct xs rule: rev-induct*)
case Nil thus ?case by simp
next
case (*snoc a xs*)
show ?case
proof cases
assume $x = a$ **thus ?case using snoc by** (*auto intro!*: *exI*)
next
assume $x \neq a$ **thus ?case using snoc by fastforce**
qed
qed

lemma *in-set-conv-decomp-last*:
 $(x \in \text{set } xs) = (\exists ys \ zs. \ xs = ys @ x \# zs \wedge x \notin \text{set } zs)$
by (*auto dest!*: *split-list-last*)

lemma *split-list-prop*: $\exists x \in \text{set } xs. \ P \ x \implies \exists ys \ x \ zs. \ xs = ys @ x \# zs \wedge P \ x$
proof (*induct xs*)
case Nil thus ?case by simp
next
case Cons thus ?case
by(*simp add: Bex-def*)(*metis append-Cons append.simps(1)*)
qed

lemma *split-list-propE*:
assumes $\exists x \in \text{set } xs. \ P \ x$
obtains $ys \ x \ zs$ **where** $xs = ys @ x \# zs$ **and** $P \ x$
using *split-list-prop* [*OF assms*] **by blast**

lemma *split-list-first-prop*:
 $\exists x \in \text{set } xs. \ P \ x \implies$
 $\exists ys \ x \ zs. \ xs = ys @ x \# zs \wedge P \ x \wedge (\forall y \in \text{set } ys. \ \neg P \ y)$
proof (*induct xs*)
case Nil thus ?case by simp
next
case (*Cons x xs*)
show ?case
proof cases
assume $P \ x$
hence $x \# xs = [] @ x \# xs \wedge P \ x \wedge (\forall y \in \text{set } []. \ \neg P \ y)$ **by simp**
thus ?thesis by fast
next
assume $\neg P \ x$
hence $\exists x \in \text{set } xs. \ P \ x$ **using** *Cons(2)* **by simp**
thus ?thesis using $\langle \neg P \ x \rangle$ *Cons(1)* **by** (*metis append-Cons set-ConsD*)

qed
qed

lemma *split-list-first-propE*:
assumes $\exists x \in \text{set } xs. P x$
obtains $ys\ x\ zs$ **where** $xs = ys @ x \# zs$ **and** $P x$ **and** $\forall y \in \text{set } ys. \neg P y$
using *split-list-first-prop* [*OF assms*] **by** *blast*

lemma *split-list-first-prop-iff*:
 $(\exists x \in \text{set } xs. P x) \longleftrightarrow$
 $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P x \wedge (\forall y \in \text{set } ys. \neg P y))$
by (*rule, erule split-list-first-prop*) *auto*

lemma *split-list-last-prop*:
 $\exists x \in \text{set } xs. P x \implies$
 $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P x \wedge (\forall z \in \text{set } zs. \neg P z)$
proof(*induct xs rule:rev-induct*)
case Nil thus *?case by simp*
next
case (*snoc x xs*)
show *?case*
proof *cases*
assume $P x$ **thus** *?thesis by (auto intro!: exI)*
next
assume $\neg P x$
hence $\exists x \in \text{set } xs. P x$ **using** *snoc(2)* **by** *simp*
thus *?thesis using* $\langle \neg P x \rangle$ *snoc(1)* **by** *fastforce*
qed
qed

lemma *split-list-last-propE*:
assumes $\exists x \in \text{set } xs. P x$
obtains $ys\ x\ zs$ **where** $xs = ys @ x \# zs$ **and** $P x$ **and** $\forall z \in \text{set } zs. \neg P z$
using *split-list-last-prop* [*OF assms*] **by** *blast*

lemma *split-list-last-prop-iff*:
 $(\exists x \in \text{set } xs. P x) \longleftrightarrow$
 $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P x \wedge (\forall z \in \text{set } zs. \neg P z))$
by *rule (erule split-list-last-prop, auto)*

lemma *finite-list*: $\text{finite } A \implies \exists xs. \text{set } xs = A$
by (*erule finite-induct*) (*auto simp add: list.set(2)[symmetric] simp del: list.set(2)*)

lemma *card-length*: $\text{card } (\text{set } xs) \leq \text{length } xs$
by (*induct xs*) (*auto simp add: card-insert-if*)

lemma *set-minus-filter-out*:
 $\text{set } xs - \{y\} = \text{set } (\text{filter } (\lambda x. \neg (x = y)) xs)$

by (induct xs) auto

lemma *append-Cons-eq-iff*:

$\llbracket x \notin \text{set } xs; x \notin \text{set } ys \rrbracket \implies$

$xs @ x \# ys = xs' @ x \# ys' \iff (xs = xs' \wedge ys = ys')$

by(auto simp: append-eq-Cons-conv Cons-eq-append-conv append-eq-append-conv2)

67.1.8 concat

lemma *concat-append [simp]*: $\text{concat } (xs @ ys) = \text{concat } xs @ \text{concat } ys$

by (induct xs) auto

lemma *concat-eq-Nil-conv [simp]*: $(\text{concat } xss = []) = (\forall xs \in \text{set } xss. xs = [])$

by (induct xss) auto

lemmas *Nil-eq-concat-conv [simp] = concat-eq-Nil-conv[THEN eq-iff-swap]*

lemma *set-concat [simp]*: $\text{set } (\text{concat } xs) = (\bigcup x \in \text{set } xs. \text{set } x)$

by (induct xs) auto

lemma *concat-map-singleton[simp]*: $\text{concat}(\text{map } (\%x. [f x]) xs) = \text{map } f xs$

by (induct xs) auto

lemma *map-concat*: $\text{map } f (\text{concat } xs) = \text{concat } (\text{map } (map f) xs)$

by (induct xs) auto

lemma *rev-concat*: $\text{rev } (\text{concat } xs) = \text{concat } (\text{map } \text{rev } (\text{rev } xs))$

by (induct xs) auto

lemma *length-concat-rev[simp]*: $\text{length } (\text{concat } (\text{rev } xs)) = \text{length } (\text{concat } xs)$

by (induction xs) auto

lemma *concat-eq-concat-iff*: $\forall (x, y) \in \text{set } (\text{zip } xs \ ys). \text{length } x = \text{length } y \implies \text{length } xs = \text{length } ys \implies (\text{concat } xs = \text{concat } ys) = (xs = ys)$

proof (induct xs arbitrary: ys)

case (Cons x xs ys)

thus ?case by (cases ys) auto

qed (auto)

lemma *concat-injective*: $\text{concat } xs = \text{concat } ys \implies \text{length } xs = \text{length } ys \implies \forall (x, y) \in \text{set } (\text{zip } xs \ ys). \text{length } x = \text{length } y \implies xs = ys$

by (simp add: concat-eq-concat-iff)

lemma *concat-eq-appendD*:

assumes $\text{concat } xss = ys @ zs \ xss \neq []$

shows $\exists xss1 \ xs \ xs' \ xss2. xss = xss1 @ (xs @ xs') \# xss2 \wedge ys = \text{concat } xss1 @ xs \wedge zs = xs' @ \text{concat } xss2$

using *assms*

proof(induction xss arbitrary: ys)

```

case (Cons xs xss)
from Cons.prem consider
  us where xs @ us = ys concat xss = us @ zs |
  us where xs = ys @ us us @ concat xss = zs
  by(auto simp add: append-eq-append-conv2)
then show ?case
proof cases
  case 1
  then show ?thesis using Cons.IH[OF 1(2)]
  by(cases xss)(auto intro: exI[where x=[]], metis append.assoc append-Cons
concat.simps(2))
  qed(auto intro: exI[where x=[]])
qed simp

```

lemma concat-eq-append-conv:

```

concat xss = ys @ zs  $\longleftrightarrow$ 
(if xss = [] then ys = []  $\wedge$  zs = []
 else  $\exists$  xss1 xs xs' xss2. xss = xss1 @ (xs @ xs') # xss2  $\wedge$  ys = concat xss1 @ xs
  $\wedge$  zs = xs' @ concat xss2)
by(auto dest: concat-eq-appendD)

```

lemma hd-concat: $[[xs \neq []]; hd\ xs \neq []] \implies hd\ (concat\ xs) = hd\ (hd\ xs)$
by (metis concat.simps(2) hd-Cons-tl hd-append2)

simproc-setup list-neq ((xs::'a list) = ys) = \langle

(*
Reduces $xs=ys$ to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.
*)

let

```

fun len (Const(const-name <Nil>, -)) acc = acc
| len (Const(const-name <Cons>, -) $ - $ xs) (ts,n) = len xs (ts,n+1)
| len (Const(const-name <append>, -) $ xs $ ys) acc = len xs (len ys acc)
| len (Const(const-name <rev>, -) $ xs) acc = len xs acc
| len (Const(const-name <map>, -) $ - $ xs) acc = len xs acc
| len (Const(const-name <concat>, T) $ (Const(const-name <rev>, -) $ xss)) acc
  = len (Const(const-name <concat>, T) $ xss) acc
| len t (ts,n) = (t::ts,n);

```

val ss = simpset-of **context**;

fun list-neq ctxt ct =

```

  let
    val (Const(-,eqT) $ lhs $ rhs) = Thm.term-of ct;
    val (ls,m) = len lhs ([],0) and (rs,n) = len rhs ([],0);

```



```

fun prove-neq() =
  let
    val Type(-,listT::-) = eqT;
    val size = HOLogic.size-const listT;
    val eq-len = HOLogic.mk-eq (size $ lhs, size $ rhs);
    val neq-len = HOLogic.mk-Trueprop (HOLogic.Not $ eq-len);
    val thm = Goal.prove ctxt [] [] neq-len
      (K (simp-tac (put-simpset ss ctxt) 1));
  in SOME (thm RS @ {thm neq-if-length-neq}) end
in
  if m < n andalso submultiset (op aconv) (ls,rs) orelse
    n < m andalso submultiset (op aconv) (rs,ls)
  then prove-neq() else NONE
end;
in K list-neq end
>

```

67.1.9 filter

lemma *filter-append* [simp]: $\text{filter } P (xs @ ys) = \text{filter } P xs @ \text{filter } P ys$
by (induct xs) auto

lemma *rev-filter*: $\text{rev} (\text{filter } P xs) = \text{filter } P (\text{rev } xs)$
by (induct xs) simp-all

lemma *filter-filter* [simp]: $\text{filter } P (\text{filter } Q xs) = \text{filter } (\lambda x. Q x \wedge P x) xs$
by (induct xs) auto

lemma *filter-concat*: $\text{filter } p (\text{concat } xs) = \text{concat} (\text{map} (\text{filter } p) xs)$
by (induct xs) auto

lemma *length-filter-le* [simp]: $\text{length} (\text{filter } P xs) \leq \text{length } xs$
by (induct xs) (auto simp add: le-SucI)

lemma *sum-length-filter-compl*:
 $\text{length}(\text{filter } P xs) + \text{length}(\text{filter } (\lambda x. \neg P x) xs) = \text{length } xs$
by(induct xs) simp-all

lemma *filter-True* [simp]: $\forall x \in \text{set } xs. P x \implies \text{filter } P xs = xs$
by (induct xs) auto

lemma *filter-False* [simp]: $\forall x \in \text{set } xs. \neg P x \implies \text{filter } P xs = []$
by (induct xs) auto

lemma *filter-empty-conv*: $(\text{filter } P xs = []) = (\forall x \in \text{set } xs. \neg P x)$
by (induct xs) simp-all

lemmas *empty-filter-conv* = *filter-empty-conv*[THEN eq-iff-swap]

```

lemma filter-id-conv: (filter P xs = xs) = (∀ x ∈ set xs. P x)
proof (induct xs)
  case (Cons x xs)
  then show ?case
    using length-filter-le
    by (simp add: impossible-Cons)
qed auto

lemma filter-map: filter P (map f xs) = map f (filter (P ∘ f) xs)
by (induct xs) simp-all

lemma length-filter-map[simp]:
  length (filter P (map f xs)) = length (filter (P ∘ f) xs)
by (simp add: filter-map)

lemma filter-is-subset [simp]: set (filter P xs) ≤ set xs
by auto

lemma length-filter-less:
  [ x ∈ set xs; ¬ P x ] ⇒ length (filter P xs) < length xs
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs) thus ?case
    using Suc-le-eq by fastforce
qed

lemma length-filter-conv-card:
  length (filter p xs) = card {i. i < length xs ∧ p(xs!i)}
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  let ?S = {i. i < length xs ∧ p(xs!i)}
  have fin: finite ?S by (fast intro: bounded-nat-set-is-finite)
  show ?case (is ?l = card ?S')
  proof (cases)
    assume p x
    hence eq: ?S' = insert 0 (Suc ' ?S)
    by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
    have length (filter p (x # xs)) = Suc (card ?S)
    using Cons ⟨p x⟩ by simp
    also have ... = Suc (card (Suc ' ?S)) using fin
    by (simp add: card-image)
    also have ... = card ?S' using eq fin
    by (simp add: card-insert-if)
    finally show ?thesis .
  next
  assume ¬ p x

```

```

hence eq: ?S' = Suc ' ?S
  by(auto simp add: image-def split:nat.split elim:lessE)
have length (filter p (x # xs)) = card ?S
  using Cons <¬ p x> by simp
also have ... = card(Suc ' ?S) using fin
  by (simp add: card-image)
also have ... = card ?S' using eq fin
  by (simp add:card-insert-if)
finally show ?thesis .
qed
qed

lemma Cons-eq-filterD:
  x#xs = filter P ys  $\implies$ 
   $\exists us vs. ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P vs$ 
  (is -  $\implies \exists us vs. ?P ys us vs$ )
proof(induct ys)
  case Nil thus ?case by simp
next
  case (Cons y ys)
  show ?case (is  $\exists x. ?Q x$ )
  proof cases
    assume Py: P y
    show ?thesis
    proof cases
      assume x = y
      with Py Cons.prem1 have ?Q [] by simp
      then show ?thesis ..
    next
      assume x  $\neq$  y
      with Py Cons.prem1 show ?thesis by simp
    qed
  next
  assume  $\neg P y$ 
  with Cons obtain us vs where ?P (y#ys) (y#us) vs by fastforce
  then have ?Q (y#us) by simp
  then show ?thesis ..
  qed
qed

```

```

lemma filter-eq-ConsD:
  filter P ys = x#xs  $\implies$ 
   $\exists us vs. ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P vs$ 
  by(rule Cons-eq-filterD) simp

```

```

lemma filter-eq-Cons-iff:
  (filter P ys = x#xs) =
  ( $\exists us vs. ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P vs$ )
  by(auto dest:filter-eq-ConsD)

```

lemmas *Cons-eq-filter-iff* = *filter-eq-Cons-iff*[*THEN eq-iff-swap*]

lemma *inj-on-filter-key-eq*:
assumes *inj-on f (insert y (set xs))*
shows *filter (λx. f y = f x) xs = filter (HOL.eq y) xs*
using *assms* **by** (*induct xs*) *auto*

lemma *filter-cong*[*fundef-cong*]:
 $xs = ys \implies (\bigwedge x. x \in \text{set } ys \implies P x = Q x) \implies \text{filter } P \text{ } xs = \text{filter } Q \text{ } ys$
by (*induct ys arbitrary: xs*) *auto*

67.1.10 List partitioning

primrec *partition* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a list* \Rightarrow *'a list* \times *'a list* **where**
partition P [] = ([], []) |
partition P (x # xs) =
(let (yes, no) = partition P xs
in if P x then (x # yes, no) else (yes, x # no))

lemma *partition-filter1*: *fst (partition P xs) = filter P xs*
by (*induct xs*) (*auto simp add: Let-def split-def*)

lemma *partition-filter2*: *snd (partition P xs) = filter (Not \circ P) xs*
by (*induct xs*) (*auto simp add: Let-def split-def*)

lemma *partition-P*:
assumes *partition P xs = (yes, no)*
shows $(\forall p \in \text{set } \text{yes}. P p) \wedge (\forall p \in \text{set } \text{no}. \neg P p)$
proof –
from *assms* **have** *yes = fst (partition P xs)* **and** *no = snd (partition P xs)*
by *simp-all*
then show *?thesis* **by** (*simp-all add: partition-filter1 partition-filter2*)
qed

lemma *partition-set*:
assumes *partition P xs = (yes, no)*
shows *set yes \cup set no = set xs*
proof –
from *assms* **have** *yes = fst (partition P xs)* **and** *no = snd (partition P xs)*
by *simp-all*
then show *?thesis* **by** (*auto simp add: partition-filter1 partition-filter2*)
qed

lemma *partition-filter-conv*[*simp*]:
 $\text{partition } f \text{ } xs = (\text{filter } f \text{ } xs, \text{filter } (\text{Not } \circ f) \text{ } xs)$
unfolding *partition-filter2*[*symmetric*]
unfolding *partition-filter1*[*symmetric*] **by** *simp*

declare *partition.simps*[*simp del*]

67.1.11 (!)

lemma *nth-Cons-0* [*simp, code*]: $(x \# xs)!0 = x$
by *auto*

lemma *nth-Cons-Suc* [*simp, code*]: $(x \# xs)!(\text{Suc } n) = xs!n$
by *auto*

declare *nth.simps* [*simp del*]

lemma *nth-Cons-pos*[*simp*]: $0 < n \implies (x \# xs)!n = xs!(n - 1)$
by(*auto simp: Nat.gr0-conv-Suc*)

lemma *nth-append*:

$(xs @ ys)!n = (\text{if } n < \text{length } xs \text{ then } xs!n \text{ else } ys!(n - \text{length } xs))$

proof (*induct xs arbitrary: n*)

case (*Cons x xs*)

then show *?case*

using *less-Suc-eq-0-disj* **by** *auto*

qed *simp*

lemma *nth-append-length* [*simp*]: $(xs @ x \# ys)! \text{length } xs = x$
by (*induct xs*) *auto*

lemma *nth-append-length-plus*[*simp*]: $(xs @ ys)!(\text{length } xs + n) = ys!n$
by (*induct xs*) *auto*

lemma *nth-map* [*simp*]: $n < \text{length } xs \implies (\text{map } f \text{ } xs)!n = f(xs!n)$

proof (*induct xs arbitrary: n*)

case (*Cons x xs*)

then show *?case*

using *less-Suc-eq-0-disj* **by** *auto*

qed *simp*

lemma *nth-tl*: $n < \text{length } (\text{tl } xs) \implies \text{tl } xs!n = xs! \text{Suc } n$
by (*induction xs*) *auto*

lemma *hd-conv-nth*: $xs \neq [] \implies \text{hd } xs = xs!0$
by(*cases xs*) *simp-all*

lemma *list-eq-iff-nth-eq*:

$(xs = ys) = (\text{length } xs = \text{length } ys \wedge (\forall i < \text{length } xs. xs!i = ys!i))$

proof (*induct xs arbitrary: ys*)

case (*Cons x xs ys*)

show *?case*

proof (*cases ys*)

case (*Cons y ys*)

with *Cons.hyps* **show** *?thesis* **by** *fastforce*
qed *simp*
qed *force*

lemma *map-equality-iff*:

$map\ f\ xs = map\ g\ ys \longleftrightarrow length\ xs = length\ ys \wedge (\forall i < length\ ys.\ f\ (xs!i) = g\ (ys!i))$

by (*fastforce simp: list-eq-iff-nth-eq*)

lemma *set-conv-nth*: $set\ xs = \{xs!i \mid i.\ i < length\ xs\}$

proof (*induct xs*)

case (*Cons x xs*)

have $insert\ x\ \{xs!\ i \mid i.\ i < length\ xs\} = \{(x \# xs)!\ i \mid i.\ i < Suc\ (length\ xs)\}$
(is *?L=?R*)

proof

show *?L* \subseteq *?R*

by *force*

show *?R* \subseteq *?L*

using *less-Suc-eq-0-disj* **by** *auto*

qed

with *Cons* **show** *?case*

by *simp*

qed *simp*

lemma *in-set-conv-nth*: $(x \in set\ xs) = (\exists i < length\ xs.\ xs!i = x)$

by (*auto simp:set-conv-nth*)

lemma *nth-equal-first-eq*:

assumes $x \notin set\ xs$

assumes $n \leq length\ xs$

shows $(x \# xs)!\ n = x \longleftrightarrow n = 0$ **(is** *?lhs* \longleftrightarrow *?rhs*)

proof

assume *?lhs*

show *?rhs*

proof (*rule ccontr*)

assume $n \neq 0$

then have $n > 0$ **by** *simp*

with $\langle ?lhs \rangle$ **have** $xs!\ (n - 1) = x$ **by** *simp*

moreover from $\langle n > 0 \rangle$ $\langle n \leq length\ xs \rangle$ **have** $n - 1 < length\ xs$ **by** *simp*

ultimately have $\exists i < length\ xs.\ xs!\ i = x$ **by** *auto*

with $\langle x \notin set\ xs \rangle$ *in-set-conv-nth* [*of x xs*] **show** *False* **by** *simp*

qed

next

assume *?rhs* **then show** *?lhs* **by** *simp*

qed

lemma *nth-non-equal-first-eq*:

assumes $x \neq y$

shows $(x \# xs)!\ n = y \longleftrightarrow xs!\ (n - 1) = y \wedge n > 0$ **(is** *?lhs* \longleftrightarrow *?rhs*)

proof

assume *?lhs with assms have* $n > 0$ **by** (*cases n*) *simp-all*

with $\langle ?lhs \rangle$ **show** *?rhs by simp*

next

assume *?rhs then show ?lhs by simp*

qed

lemma *list-ball-nth*: $\llbracket n < \text{length } xs; \forall x \in \text{set } xs. P x \rrbracket \implies P(xs!n)$
by (*auto simp add: set-conv-nth*)

lemma *nth-mem* [*simp*]: $n < \text{length } xs \implies xs!n \in \text{set } xs$
by (*auto simp add: set-conv-nth*)

lemma *all-nth-imp-all-set*:
 $\llbracket \forall i < \text{length } xs. P(xs!i); x \in \text{set } xs \rrbracket \implies P x$
by (*auto simp add: set-conv-nth*)

lemma *all-set-conv-all-nth*:
 $(\forall x \in \text{set } xs. P x) = (\forall i. i < \text{length } xs \longrightarrow P (xs ! i))$
by (*auto simp add: set-conv-nth*)

lemma *rev-nth*:
 $n < \text{size } xs \implies \text{rev } xs ! n = xs ! (\text{length } xs - \text{Suc } n)$

proof (*induct xs arbitrary: n*)

case Nil thus ?case by simp

next

case (*Cons x xs*)

hence $n < \text{Suc } (\text{length } xs)$ **by simp**

moreover

{ **assume** $n < \text{length } xs$

with n **obtain** n' **where** $n': \text{length } xs - n = \text{Suc } n'$

by (*cases length xs - n, auto*)

moreover

from n' **have** $\text{length } xs - \text{Suc } n = n'$ **by simp**

ultimately

have $xs ! (\text{length } xs - \text{Suc } n) = (x \# xs) ! (\text{length } xs - n)$ **by simp**

}

ultimately

show *?case by (clarsimp simp add: Cons nth-append)*

qed

lemma *Skolem-list-nth*:
 $(\forall i < k. \exists x. P i x) = (\exists xs. \text{size } xs = k \wedge (\forall i < k. P i (xs!i)))$
 $(\text{is } - = (\exists xs. ?P k xs))$

proof (*induct k*)

case 0 show ?case by simp

next

case (*Suc k*)

show *?case (is ?L = ?R is - = ($\exists xs. ?P' xs$))*

```

proof
  assume ?R thus ?L using Suc by auto
next
  assume ?L
  with Suc obtain x xs where ?P k xs  $\wedge$  P k x by (metis less-Suc-eq)
  hence ?P'(xs@[x]) by(simp add:nth-append less-Suc-eq)
  thus ?R ..
qed
qed

```

67.1.12 list-update

lemma length-list-update [simp]: $\text{length}(xs[i:=x]) = \text{length } xs$
by (induct xs arbitrary: i) (auto split: nat.split)

lemma nth-list-update:
 $i < \text{length } xs \implies (xs[i:=x])!j = (\text{if } i = j \text{ then } x \text{ else } xs!j)$
by (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

lemma nth-list-update-eq [simp]: $i < \text{length } xs \implies (xs[i:=x])!i = x$
by (simp add: nth-list-update)

lemma nth-list-update-neq [simp]: $i \neq j \implies xs[i:=x]!j = xs!j$
by (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

lemma list-update-id [simp]: $xs[i := xs!i] = xs$
by (induct xs arbitrary: i) (simp-all split: nat.splits)

lemma list-update-beyond [simp]: $\text{length } xs \leq i \implies xs[i:=x] = xs$

proof (induct xs arbitrary: i)
case (Cons x xs i)
then show ?case
by (metis leD length-list-update list-eq-iff-nth-eq nth-list-update-neq)
qed simp

lemma list-update-nonempty [simp]: $xs[k:=x] = [] \longleftrightarrow xs=[]$
by (simp only: length-0-conv[symmetric] length-list-update)

lemma list-update-same-conv:
 $i < \text{length } xs \implies (xs[i := x] = xs) = (xs!i = x)$
by (induct xs arbitrary: i) (auto split: nat.split)

lemma list-update-append1:
 $i < \text{size } xs \implies (xs @ ys)[i:=x] = xs[i:=x] @ ys$
by (induct xs arbitrary: i)(auto split: nat.split)

lemma list-update-append:
 $(xs @ ys)[n:=x] =$
 $(\text{if } n < \text{length } xs \text{ then } xs[n:=x] @ ys \text{ else } xs @ (ys [n-\text{length } xs:=x]))$

by (*induct xs arbitrary: n*) (*auto split:nat.splits*)

lemma *list-update-length* [*simp*]:

$(xs @ x \# ys)[length\ xs := y] = (xs @ y \# ys)$

by (*induct xs, auto*)

lemma *map-update*: $map\ f\ (xs[k := y]) = (map\ f\ xs)[k := f\ y]$

by (*induct xs arbitrary: k*) (*auto split:nat.splits*)

lemma *rev-update*:

$k < length\ xs \implies rev\ (xs[k := y]) = (rev\ xs)[length\ xs - k - 1 := y]$

by (*induct xs arbitrary: k*) (*auto simp: list-update-append split:nat.splits*)

lemma *update-zip*:

$(zip\ xs\ ys)[i := xy] = zip\ (xs[i := fst\ xy])\ (ys[i := snd\ xy])$

by (*induct ys arbitrary: i xy xs*) (*auto, case-tac xs, auto split: nat.split*)

lemma *set-update-subset-insert*: $set(xs[i := x]) \leq insert\ x\ (set\ xs)$

by (*induct xs arbitrary: i*) (*auto split: nat.split*)

lemma *set-update-subsetI*: $\llbracket set\ xs \subseteq A; x \in A \rrbracket \implies set(xs[i := x]) \subseteq A$

by (*blast dest!: set-update-subset-insert [THEN subsetD]*)

lemma *set-update-memI*: $n < length\ xs \implies x \in set\ (xs[n := x])$

by (*induct xs arbitrary: n*) (*auto split:nat.splits*)

lemma *list-update-overwrite* [*simp*]:

$xs\ [i := x, i := y] = xs\ [i := y]$

by (*induct xs arbitrary: i*) (*simp-all split: nat.split*)

lemma *list-update-swap*:

$i \neq i' \implies xs\ [i := x, i' := x'] = xs\ [i' := x', i := x]$

by (*induct xs arbitrary: i i'*) (*simp-all split: nat.split*)

lemma *list-update-code* [*code*]:

$\llbracket [i := y] = [] \rrbracket$

$(x \# xs)[0 := y] = y \# xs$

$(x \# xs)[Suc\ i := y] = x \# xs[i := y]$

by *simp-all*

67.1.13 *last and butlast*

lemma *hd-Nil-eq-last*: $hd\ Nil = last\ Nil$

unfolding *hd-def last-def* **by** *simp*

lemma *last-snoc* [*simp*]: $last\ (xs @ [x]) = x$

by (*induct xs*) *auto*

lemma *butlast-snoc* [*simp*]: $butlast\ (xs @ [x]) = xs$

by (induct xs) auto

lemma last-ConsL: $xs = [] \implies last(x\#xs) = x$
by simp

lemma last-ConsR: $xs \neq [] \implies last(x\#xs) = last\ xs$
by simp

lemma last-append: $last(xs @ ys) = (if\ ys = []\ then\ last\ xs\ else\ last\ ys)$
by (induct xs) (auto)

lemma last-appendL[simp]: $ys = [] \implies last(xs @ ys) = last\ xs$
by(simp add:last-append)

lemma last-appendR[simp]: $ys \neq [] \implies last(xs @ ys) = last\ ys$
by(simp add:last-append)

lemma last-tl: $xs = [] \vee tl\ xs \neq [] \implies last\ (tl\ xs) = last\ xs$
by (induct xs) simp-all

lemma butlast-tl: $butlast\ (tl\ xs) = tl\ (butlast\ xs)$
by (induct xs) simp-all

lemma hd-rev: $hd(rev\ xs) = last\ xs$
by (metis hd-Cons-tl hd-Nil-eq-last last-snoc rev-eq-Cons-iff rev-is-Nil-conv)

lemma last-rev: $last(rev\ xs) = hd\ xs$
by (metis hd-rev rev-swap)

lemma last-in-set[simp]: $as \neq [] \implies last\ as \in set\ as$
by (induct as) auto

lemma length-butlast [simp]: $length\ (butlast\ xs) = length\ xs - 1$
by (induct xs rule: rev-induct) auto

lemma butlast-append:
 $butlast\ (xs @ ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs @ butlast\ ys)$
by (induct xs arbitrary: ys) auto

lemma append-butlast-last-id [simp]:
 $xs \neq [] \implies butlast\ xs @ [last\ xs] = xs$
by (induct xs) auto

lemma in-set-butlastD: $x \in set\ (butlast\ xs) \implies x \in set\ xs$
by (induct xs) (auto split: if-split-asm)

lemma in-set-butlast-appendI:
 $x \in set\ (butlast\ xs) \vee x \in set\ (butlast\ ys) \implies x \in set\ (butlast\ (xs @ ys))$
by (auto dest: in-set-butlastD simp add: butlast-append)

lemma *last-drop*[simp]: $n < \text{length } xs \implies \text{last } (\text{drop } n \text{ } xs) = \text{last } xs$
by (*induct xs arbitrary: n*)(*auto split:nat.split*)

lemma *nth-butlast*:

assumes $n < \text{length } (\text{butlast } xs)$ **shows** $\text{butlast } xs ! n = xs ! n$

proof (*cases xs*)

case (*Cons y ys*)

moreover from *assms* **have** $\text{butlast } xs ! n = (\text{butlast } xs @ [\text{last } xs]) ! n$

by (*simp add: nth-append*)

ultimately show *?thesis* **using** *append-butlast-last-id* **by** *simp*

qed *simp*

lemma *last-conv-nth*: $xs \neq [] \implies \text{last } xs = xs!(\text{length } xs - 1)$

by(*induct xs*)(*auto simp:neq-Nil-conv*)

lemma *butlast-conv-take*: $\text{butlast } xs = \text{take } (\text{length } xs - 1) \text{ } xs$

by (*induction xs rule: induct-list012*) *simp-all*

lemma *last-list-update*:

$xs \neq [] \implies \text{last}(xs[k:=x]) = (\text{if } k = \text{size } xs - 1 \text{ then } x \text{ else } \text{last } xs)$

by (*auto simp: last-conv-nth*)

lemma *butlast-list-update*:

$\text{butlast}(xs[k:=x]) =$

$(\text{if } k = \text{size } xs - 1 \text{ then } \text{butlast } xs \text{ else } (\text{butlast } xs)[k:=x])$

by(*cases xs rule:rev-cases*)(*auto simp: list-update-append split: nat.splits*)

lemma *last-map*: $xs \neq [] \implies \text{last } (\text{map } f \text{ } xs) = f (\text{last } xs)$

by (*cases xs rule: rev-cases*) *simp-all*

lemma *map-butlast*: $\text{map } f (\text{butlast } xs) = \text{butlast } (\text{map } f \text{ } xs)$

by (*induct xs*) *simp-all*

lemma *snoc-eq-iff-butlast*:

$xs @ [x] = ys \iff (ys \neq [] \wedge \text{butlast } ys = xs \wedge \text{last } ys = x)$

by *fastforce*

corollary *longest-common-suffix*:

$\exists ss \text{ } xs' \text{ } ys'. xs = xs' @ ss \wedge ys = ys' @ ss$

$\wedge (xs' = [] \vee ys' = [] \vee \text{last } xs' \neq \text{last } ys')$

using *longest-common-prefix*[of *rev xs rev ys*]

unfolding *rev-swap rev-append* **by** (*metis last-rev rev-is-Nil-conv*)

lemma *butlast-rev* [simp]: $\text{butlast } (\text{rev } xs) = \text{rev } (\text{tl } xs)$

by (*cases xs*) *simp-all*

67.1.14 *take and drop*

lemma *take-0*: $take\ 0\ xs = []$
by (*induct xs*) *auto*

lemma *drop-0*: $drop\ 0\ xs = xs$
by (*induct xs*) *auto*

lemma *take0[simp]*: $take\ 0 = (\lambda xs. [])$
by(*rule ext*) (*rule take-0*)

lemma *drop0[simp]*: $drop\ 0 = (\lambda x. x)$
by(*rule ext*) (*rule drop-0*)

lemma *take-Suc-Cons [simp]*: $take\ (Suc\ n)\ (x\ \#\ xs) = x\ \# \ take\ n\ xs$
by *simp*

lemma *drop-Suc-Cons [simp]*: $drop\ (Suc\ n)\ (x\ \# \ xs) = drop\ n\ xs$
by *simp*

declare *take-Cons [simp del]* **and** *drop-Cons [simp del]*

lemma *take-Suc*: $xs \neq [] \implies take\ (Suc\ n)\ xs = hd\ xs\ \# \ take\ n\ (tl\ xs)$
by(*clarsimp simp add:neq-Nil-conv*)

lemma *drop-Suc*: $drop\ (Suc\ n)\ xs = drop\ n\ (tl\ xs)$
by(*cases xs, simp-all*)

lemma *hd-take[simp]*: $j > 0 \implies hd\ (take\ j\ xs) = hd\ xs$
by (*metis gr0-conv-Suc list.sel(1) take.simps(1) take-Suc*)

lemma *take-tl*: $take\ n\ (tl\ xs) = tl\ (take\ (Suc\ n)\ xs)$
by (*induct xs arbitrary: n*) *simp-all*

lemma *drop-tl*: $drop\ n\ (tl\ xs) = tl(drop\ n\ xs)$
by(*induct xs arbitrary: n, simp-all add:drop-Cons drop-Suc split:nat.split*)

lemma *tl-take*: $tl\ (take\ n\ xs) = take\ (n - 1)\ (tl\ xs)$
by (*cases n, simp, cases xs, auto*)

lemma *tl-drop*: $tl\ (drop\ n\ xs) = drop\ n\ (tl\ xs)$
by (*simp only: drop-tl*)

lemma *nth-via-drop*: $drop\ n\ xs = y\ \# \ ys \implies xs!n = y$
by (*induct xs arbitrary: n, simp*)(*auto simp: drop-Cons nth-Cons split: nat.splits*)

lemma *take-Suc-conv-app-nth*:
 $i < length\ xs \implies take\ (Suc\ i)\ xs = take\ i\ xs\ @ \ [xs!i]$
proof (*induct xs arbitrary: i*)
case Nil

```

then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

```

```

lemma Cons-nth-drop-Suc:
   $i < \text{length } xs \implies (xs!i) \# (\text{drop } (\text{Suc } i) \text{ } xs) = \text{drop } i \text{ } xs$ 
proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

```

```

lemma length-take [simp]:  $\text{length } (\text{take } n \text{ } xs) = \min (\text{length } xs) \ n$ 
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

```

```

lemma length-drop [simp]:  $\text{length } (\text{drop } n \text{ } xs) = (\text{length } xs - n)$ 
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

```

```

lemma take-all [simp]:  $\text{length } xs \leq n \implies \text{take } n \text{ } xs = xs$ 
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

```

```

lemma drop-all [simp]:  $\text{length } xs \leq n \implies \text{drop } n \text{ } xs = []$ 
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

```

```

lemma take-all-iff [simp]:  $\text{take } n \text{ } xs = xs \iff \text{length } xs \leq n$ 
by (metis length-take min.order-iff take-all)

```

```

lemma take-eq-Nil[simp]:  $(\text{take } n \text{ } xs = []) = (n = 0 \vee xs = [])$ 
by(induct xs arbitrary: n)(auto simp: take-Cons split:nat.split)

```

```

lemmas take-eq-Nil2[simp] = take-eq-Nil[THEN eq-iff-swap]

```

```

lemma drop-eq-Nil [simp]:  $\text{drop } n \text{ } xs = [] \iff \text{length } xs \leq n$ 
by (metis diff-is-0-eq drop-all length-drop list.size(3))

```

```

lemmas drop-eq-Nil2 [simp] = drop-eq-Nil[THEN eq-iff-swap]

```

```

lemma take-append [simp]:
   $\text{take } n \text{ } (xs @ ys) = (\text{take } n \text{ } xs @ \text{take } (n - \text{length } xs) \text{ } ys)$ 
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

```

```

lemma drop-append [simp]:
   $\text{drop } n \text{ } (xs @ ys) = \text{drop } n \text{ } xs @ \text{drop } (n - \text{length } xs) \text{ } ys$ 

```

by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma take-take [simp]: take n (take m xs) = take (min n m) xs

proof (induct m arbitrary: xs n)

case 0

then show ?case by simp

next

case Suc

then show ?case by (cases xs; cases n) simp-all

qed

lemma drop-drop [simp]: drop n (drop m xs) = drop (n + m) xs

proof (induct m arbitrary: xs)

case 0

then show ?case by simp

next

case Suc

then show ?case by (cases xs) simp-all

qed

lemma take-drop: take n (drop m xs) = drop m (take (n + m) xs)

proof (induct m arbitrary: xs n)

case 0

then show ?case by simp

next

case Suc

then show ?case by (cases xs; cases n) simp-all

qed

lemma drop-take: drop n (take m xs) = take (m-n) (drop n xs)

by (induct xs arbitrary: m n) (auto simp: take-Cons drop-Cons split: nat.split)

lemma append-take-drop-id [simp]: take n xs @ drop n xs = xs

proof (induct n arbitrary: xs)

case 0

then show ?case by simp

next

case Suc

then show ?case by (cases xs) simp-all

qed

lemma take-map: take n (map f xs) = map f (take n xs)

proof (induct n arbitrary: xs)

case 0

then show ?case by simp

next

case Suc

then show ?case by (cases xs) simp-all

qed

lemma *drop-map*: $\text{drop } n (\text{map } f \text{ } xs) = \text{map } f (\text{drop } n \text{ } xs)$

proof (*induct n arbitrary: xs*)
 case 0
 then show ?case by simp
 next
 case Suc
 then show ?case by (cases xs) simp-all
 qed

lemma *rev-take*: $\text{rev } (\text{take } i \text{ } xs) = \text{drop } (\text{length } xs - i) (\text{rev } xs)$

proof (*induct xs arbitrary: i*)
 case Nil
 then show ?case by simp
 next
 case Cons
 then show ?case by (cases i) auto
 qed

lemma *rev-drop*: $\text{rev } (\text{drop } i \text{ } xs) = \text{take } (\text{length } xs - i) (\text{rev } xs)$

proof (*induct xs arbitrary: i*)
 case Nil
 then show ?case by simp
 next
 case Cons
 then show ?case by (cases i) auto
 qed

lemma *drop-rev*: $\text{drop } n (\text{rev } xs) = \text{rev } (\text{take } (\text{length } xs - n) \text{ } xs)$
 by (cases length xs < n) (auto simp: rev-take)

lemma *take-rev*: $\text{take } n (\text{rev } xs) = \text{rev } (\text{drop } (\text{length } xs - n) \text{ } xs)$
 by (cases length xs < n) (auto simp: rev-drop)

lemma *nth-take* [*simp*]: $i < n \implies (\text{take } n \text{ } xs)!i = xs!i$

proof (*induct xs arbitrary: i n*)
 case Nil
 then show ?case by simp
 next
 case Cons
 then show ?case by (cases n; cases i) simp-all
 qed

lemma *nth-drop* [*simp*]:

$n \leq \text{length } xs \implies (\text{drop } n \text{ } xs)!i = xs!(n + i)$

proof (*induct n arbitrary: xs*)
 case 0
 then show ?case by simp
 next

case *Suc*
then show *?case* **by** (*cases xs*) *simp-all*
qed

lemma *butlast-take*:
 $n \leq \text{length } xs \implies \text{butlast } (\text{take } n \text{ } xs) = \text{take } (n - 1) \text{ } xs$
by (*simp add: butlast-conv-take*)

lemma *butlast-drop*: $\text{butlast } (\text{drop } n \text{ } xs) = \text{drop } n \text{ } (\text{butlast } xs)$
by (*simp add: butlast-conv-take drop-take ac-simps*)

lemma *take-butlast*: $n < \text{length } xs \implies \text{take } n \text{ } (\text{butlast } xs) = \text{take } n \text{ } xs$
by (*simp add: butlast-conv-take*)

lemma *drop-butlast*: $\text{drop } n \text{ } (\text{butlast } xs) = \text{butlast } (\text{drop } n \text{ } xs)$
by (*simp add: butlast-conv-take drop-take ac-simps*)

lemma *butlast-power*: $(\text{butlast } \overset{\sim}{\sim} n) \text{ } xs = \text{take } (\text{length } xs - n) \text{ } xs$
by (*induct n*) (*auto simp: butlast-take*)

lemma *hd-drop-conv-nth*: $n < \text{length } xs \implies \text{hd}(\text{drop } n \text{ } xs) = xs!n$
by(*simp add: hd-conv-nth*)

lemma *set-take-subset-set-take*:
 $m \leq n \implies \text{set}(\text{take } m \text{ } xs) \subseteq \text{set}(\text{take } n \text{ } xs)$
proof (*induct xs arbitrary: m n*)
case (*Cons x xs m n*) **then show** *?case*
by (*cases n*) (*auto simp: take-Cons*)
qed *simp*

lemma *set-take-subset*: $\text{set}(\text{take } n \text{ } xs) \subseteq \text{set } xs$
by(*induct xs arbitrary: n*)(*auto simp:take-Cons split:nat.split*)

lemma *set-drop-subset*: $\text{set}(\text{drop } n \text{ } xs) \subseteq \text{set } xs$
by(*induct xs arbitrary: n*)(*auto simp:drop-Cons split:nat.split*)

lemma *set-drop-subset-set-drop*:
 $m \geq n \implies \text{set}(\text{drop } m \text{ } xs) \subseteq \text{set}(\text{drop } n \text{ } xs)$
proof (*induct xs arbitrary: m n*)
case (*Cons x xs m n*)
then show *?case*
by (*clarsimp simp: drop-Cons split: nat.split*) (*metis set-drop-subset subset-iff*)
qed *simp*

lemma *in-set-takeD*: $x \in \text{set}(\text{take } n \text{ } xs) \implies x \in \text{set } xs$
using *set-take-subset* **by** *fast*

lemma *in-set-dropD*: $x \in \text{set}(\text{drop } n \text{ } xs) \implies x \in \text{set } xs$
using *set-drop-subset* **by** *fast*

lemma *append-eq-conv-conj*:

$$(xs @ ys = zs) = (xs = take (length xs) zs \wedge ys = drop (length xs) zs)$$

proof (*induct xs arbitrary: zs*)

case (*Cons x xs zs*) **then show** ?case

by (*cases zs, auto*)

qed *auto*

lemma *map-eq-append-conv*:

$$map f xs = ys @ zs \longleftrightarrow (\exists us vs. xs = us @ vs \wedge ys = map f us \wedge zs = map f vs)$$

proof –

have $map f xs \neq ys @ zs \wedge map f xs \neq ys @ zs \vee map f xs \neq ys @ zs \vee map f xs = ys @ zs \wedge$

$$(\exists bs bsa. xs = bs @ bsa \wedge ys = map f bs \wedge zs = map f bsa)$$

by (*metis append-eq-conv-conj append-take-drop-id drop-map take-map*)

then show ?thesis

using *map-append* **by** *blast*

qed

lemmas *append-eq-map-conv = map-eq-append-conv*[*THEN eq-iff-swap*]

lemma *take-add*: $take (i+j) xs = take i xs @ take j (drop i xs)$

proof (*induct xs arbitrary: i*)

case (*Cons x xs i*) **then show** ?case

by (*cases i, auto*)

qed *auto*

lemma *append-eq-append-conv-if*:

$$(xs_1 @ xs_2 = ys_1 @ ys_2) =$$

(*if size xs₁ ≤ size ys₁*

$$then xs_1 = take (size xs_1) ys_1 \wedge xs_2 = drop (size xs_1) ys_1 @ ys_2$$

$$else take (size ys_1) xs_1 = ys_1 \wedge drop (size ys_1) xs_1 @ xs_2 = ys_2)$$

proof (*induct xs₁ arbitrary: ys₁*)

case (*Cons a xs₁ ys₁*) **then show** ?case

by (*cases ys₁, auto*)

qed *auto*

lemma *take-hd-drop*:

$$n < length xs \implies take n xs @ [hd (drop n xs)] = take (Suc n) xs$$

by (*induct xs arbitrary: n*) (*simp-all add:drop-Cons split:nat.split*)

lemma *id-take-nth-drop*:

$$i < length xs \implies xs = take i xs @ xs!i \# drop (Suc i) xs$$

proof –

assume *si: i < length xs*

hence $xs = take (Suc i) xs @ drop (Suc i) xs$ **by** *auto*

moreover

from *si* **have** $take (Suc i) xs = take i xs @ [xs!i]$

using *take-Suc-conv-app-nth* **by** *blast*
ultimately show *?thesis* **by** *auto*
qed

lemma *take-update-cancel[simp]*: $n \leq m \implies \text{take } n \ (xs[m := y]) = \text{take } n \ xs$
by (*simp add: list-eq-iff-nth-eq*)

lemma *drop-update-cancel[simp]*: $n < m \implies \text{drop } m \ (xs[n := x]) = \text{drop } m \ xs$
by (*simp add: list-eq-iff-nth-eq*)

lemma *upd-conv-take-nth-drop*:
 $i < \text{length } xs \implies xs[i:=a] = \text{take } i \ xs \ @ \ a \ \# \ \text{drop } (\text{Suc } i) \ xs$
proof –
assume $i < \text{length } xs$
have $xs[i:=a] = (\text{take } i \ xs \ @ \ xs!i \ \# \ \text{drop } (\text{Suc } i) \ xs)[i:=a]$
by (*rule arg-cong[OF id-take-nth-drop[OF i]]*)
also have $\dots = \text{take } i \ xs \ @ \ a \ \# \ \text{drop } (\text{Suc } i) \ xs$
using i **by** (*simp add: list-update-append*)
finally show *?thesis* .
qed

lemma *take-update-swap*: $\text{take } m \ (xs[n := x]) = (\text{take } m \ xs)[n := x]$
proof (*cases n ≥ length xs*)
case *False*
then show *?thesis*
by (*simp add: upd-conv-take-nth-drop take-Cons drop-take min-def diff-Suc split: nat.split*)
qed *auto*

lemma *drop-update-swap*:
assumes $m \leq n$ **shows** $\text{drop } m \ (xs[n := x]) = (\text{drop } m \ xs)[n-m := x]$
proof (*cases n ≥ length xs*)
case *False*
with *assms* **show** *?thesis*
by (*simp add: upd-conv-take-nth-drop drop-take*)
qed *auto*

lemma *nth-image*: $l \leq \text{size } xs \implies \text{nth } xs \ ' \ \{0..<l\} = \text{set}(\text{take } l \ xs)$
by (*simp add: set-conv-nth*) *force*

67.1.15 *takeWhile* and *dropWhile*

lemma *length-takeWhile-le*: $\text{length } (\text{takeWhile } P \ xs) \leq \text{length } xs$
by (*induct xs*) *auto*

lemma *takeWhile-dropWhile-id [simp]*: $\text{takeWhile } P \ xs \ @ \ \text{dropWhile } P \ xs = xs$
by (*induct xs*) *auto*

lemma *takeWhile-append1 [simp]*:

$\llbracket x \in \text{set } xs; \neg P(x) \rrbracket \Longrightarrow \text{takeWhile } P (xs @ ys) = \text{takeWhile } P xs$
by (induct xs) auto

lemma takeWhile-append2 [simp]:

$(\bigwedge x. x \in \text{set } xs \Longrightarrow P x) \Longrightarrow \text{takeWhile } P (xs @ ys) = xs @ \text{takeWhile } P ys$
by (induct xs) auto

lemma takeWhile-append:

$\text{takeWhile } P (xs @ ys) = (\text{if } \forall x \in \text{set } xs. P x \text{ then } xs @ \text{takeWhile } P ys \text{ else } \text{takeWhile } P xs)$

using takeWhile-append1[of - xs P ys] takeWhile-append2[of xs P ys] **by** auto

lemma takeWhile-tail: $\neg P x \Longrightarrow \text{takeWhile } P (xs @ (x\#l)) = \text{takeWhile } P xs$

by (induct xs) auto

lemma takeWhile-eq-Nil-iff: $\text{takeWhile } P xs = [] \longleftrightarrow xs = [] \vee \neg P (\text{hd } xs)$

by (cases xs) auto

lemma takeWhile-nth: $j < \text{length } (\text{takeWhile } P xs) \Longrightarrow \text{takeWhile } P xs ! j = xs ! j$

by (metis nth-append takeWhile-dropWhile-id)

lemma takeWhile-takeWhile: $\text{takeWhile } Q (\text{takeWhile } P xs) = \text{takeWhile } (\lambda x. P x \wedge Q x) xs$

by(induct xs) simp-all

lemma dropWhile-nth: $j < \text{length } (\text{dropWhile } P xs) \Longrightarrow$

$\text{dropWhile } P xs ! j = xs ! (j + \text{length } (\text{takeWhile } P xs))$

by (metis add.commute nth-append-length-plus takeWhile-dropWhile-id)

lemma length-dropWhile-le: $\text{length } (\text{dropWhile } P xs) \leq \text{length } xs$

by (induct xs) auto

lemma dropWhile-append1 [simp]:

$\llbracket x \in \text{set } xs; \neg P(x) \rrbracket \Longrightarrow \text{dropWhile } P (xs @ ys) = (\text{dropWhile } P xs) @ ys$

by (induct xs) auto

lemma dropWhile-append2 [simp]:

$(\bigwedge x. x \in \text{set } xs \Longrightarrow P(x)) \Longrightarrow \text{dropWhile } P (xs @ ys) = \text{dropWhile } P ys$

by (induct xs) auto

lemma dropWhile-append3:

$\neg P y \Longrightarrow \text{dropWhile } P (xs @ y \# ys) = \text{dropWhile } P xs @ y \# ys$

by (induct xs) auto

lemma dropWhile-append:

$\text{dropWhile } P (xs @ ys) = (\text{if } \forall x \in \text{set } xs. P x \text{ then } \text{dropWhile } P ys \text{ else } \text{dropWhile } P xs @ ys)$

using dropWhile-append1[of - xs P ys] dropWhile-append2[of xs P ys] **by** auto

lemma *dropWhile-last*:

$x \in \text{set } xs \implies \neg P x \implies \text{last } (\text{dropWhile } P \text{ } xs) = \text{last } xs$
by (*auto simp add: dropWhile-append3 in-set-conv-decomp*)

lemma *set-dropWhileD*: $x \in \text{set } (\text{dropWhile } P \text{ } xs) \implies x \in \text{set } xs$

by (*induct xs*) (*auto split: if-split-asm*)

lemma *set-takeWhileD*: $x \in \text{set } (\text{takeWhile } P \text{ } xs) \implies x \in \text{set } xs \wedge P x$

by (*induct xs*) (*auto split: if-split-asm*)

lemma *takeWhile-eq-all-conv[simp]*:

$(\text{takeWhile } P \text{ } xs = xs) = (\forall x \in \text{set } xs. P x)$
by(*induct xs, auto*)

lemma *dropWhile-eq-Nil-conv[simp]*:

$(\text{dropWhile } P \text{ } xs = []) = (\forall x \in \text{set } xs. P x)$
by(*induct xs, auto*)

lemma *dropWhile-eq-Cons-conv*:

$(\text{dropWhile } P \text{ } xs = y \# ys) = (xs = \text{takeWhile } P \text{ } xs @ y \# ys \wedge \neg P y)$
by(*induct xs, auto*)

lemma *dropWhile-eq-self-iff*: $\text{dropWhile } P \text{ } xs = xs \iff xs = [] \vee \neg P (\text{hd } xs)$

by (*cases xs*) (*auto simp: dropWhile-eq-Cons-conv*)

lemma *dropWhile-dropWhile1*: $(\bigwedge x. Q x \implies P x) \implies \text{dropWhile } Q \text{ } (\text{dropWhile } P \text{ } xs) = \text{dropWhile } P \text{ } xs$

by(*induct xs*) *simp-all*

lemma *dropWhile-dropWhile2*: $(\bigwedge x. P x \implies Q x) \implies \text{takeWhile } P \text{ } (\text{takeWhile } Q \text{ } xs) = \text{takeWhile } P \text{ } xs$

by(*induct xs*) *simp-all*

lemma *dropWhile-takeWhile*:

$(\bigwedge x. P x \implies Q x) \implies \text{dropWhile } P \text{ } (\text{takeWhile } Q \text{ } xs) = \text{takeWhile } Q \text{ } (\text{dropWhile } P \text{ } xs)$
by (*induction xs*) *auto*

lemma *distinct-takeWhile[simp]*: $\text{distinct } xs \implies \text{distinct } (\text{takeWhile } P \text{ } xs)$

by (*induct xs*) (*auto dest: set-takeWhileD*)

lemma *distinct-dropWhile[simp]*: $\text{distinct } xs \implies \text{distinct } (\text{dropWhile } P \text{ } xs)$

by (*induct xs*) *auto*

lemma *takeWhile-map*: $\text{takeWhile } P \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{takeWhile } (P \circ f) \text{ } xs)$

by (*induct xs*) *auto*

lemma *dropWhile-map*: $\text{dropWhile } P \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{dropWhile } (P \circ f) \text{ } xs)$

by (induct xs) auto

lemma takeWhile-eq-take: takeWhile P xs = take (length (takeWhile P xs)) xs
by (induct xs) auto

lemma dropWhile-eq-drop: dropWhile P xs = drop (length (takeWhile P xs)) xs
by (induct xs) auto

lemma hd-dropWhile: dropWhile P xs ≠ [] $\implies \neg P$ (hd (dropWhile P xs))
by (induct xs) auto

lemma takeWhile-eq-filter:
assumes $\bigwedge x. x \in \text{set } (\text{dropWhile } P \text{ xs}) \implies \neg P x$
shows takeWhile P xs = filter P xs

proof –

have A: filter P xs = filter P (takeWhile P xs @ dropWhile P xs)
by simp

have B: filter P (dropWhile P xs) = []
unfolding filter-empty-conv using assms by blast

have filter P xs = takeWhile P xs

unfolding A filter-append B

by (auto simp add: filter-id-conv dest: set-takeWhileD)

thus ?thesis ..

qed

lemma takeWhile-eq-take-P-nth:

$\llbracket \bigwedge i. \llbracket i < n ; i < \text{length } xs \rrbracket \implies P (xs ! i) ; n < \text{length } xs \implies \neg P (xs ! n) \rrbracket$
 \implies

takeWhile P xs = take n xs

proof (induct xs arbitrary: n)

case Nil

thus ?case by simp

next

case (Cons x xs)

show ?case

proof (cases n)

case 0

with Cons show ?thesis by simp

next

case [simp]: (Suc n')

have P x using Cons.prem1[of 0] by simp

moreover have takeWhile P xs = take n' xs

proof (rule Cons.hyps)

fix i

assume $i < n' i < \text{length } xs$

thus P (xs ! i) using Cons.prem1[of Suc i] by simp

next

assume $n' < \text{length } xs$

thus $\neg P (xs ! n')$ using Cons by auto

qed
 ultimately show *?thesis* by *simp*
 qed
 qed

lemma *nth-length-takeWhile*:
 $length (takeWhile P xs) < length xs \implies \neg P (xs ! length (takeWhile P xs))$
 by (induct xs) auto

lemma *length-takeWhile-less-P-nth*:
 assumes all: $\bigwedge i. i < j \implies P (xs ! i)$ and $j \leq length xs$
 shows $j \leq length (takeWhile P xs)$
proof (rule classical)
 assume $\neg ?thesis$
 hence $length (takeWhile P xs) < length xs$ using *assms* by *simp*
 thus *?thesis* using all $\langle \neg ?thesis \rangle$ *nth-length-takeWhile*[of *P xs*] by auto
 qed

lemma *takeWhile-neq-rev*: $\llbracket distinct\ xs; x \in set\ xs \rrbracket \implies$
 $takeWhile (\lambda y. y \neq x) (rev xs) = rev (tl (dropWhile (\lambda y. y \neq x) xs))$
 by (induct xs) (auto simp: *takeWhile-tail*[where $l=[]$])

lemma *dropWhile-neq-rev*: $\llbracket distinct\ xs; x \in set\ xs \rrbracket \implies$
 $dropWhile (\lambda y. y \neq x) (rev xs) = x \# rev (takeWhile (\lambda y. y \neq x) xs)$
proof (induct xs)
 case (Cons a xs)
 then show *?case*
 by (auto, subst *dropWhile-append2*, auto)
 qed *simp*

lemma *takeWhile-not-last*:
 $distinct\ xs \implies takeWhile (\lambda y. y \neq last\ xs)\ xs = butlast\ xs$
 by (induction xs rule: *induct-list012*) auto

lemma *takeWhile-cong* [*fundef-cong*]:
 $\llbracket l = k; \bigwedge x. x \in set\ l \implies P\ x = Q\ x \rrbracket$
 $\implies takeWhile\ P\ l = takeWhile\ Q\ k$
 by (induct k arbitrary: l) (*simp-all*)

lemma *dropWhile-cong* [*fundef-cong*]:
 $\llbracket l = k; \bigwedge x. x \in set\ l \implies P\ x = Q\ x \rrbracket$
 $\implies dropWhile\ P\ l = dropWhile\ Q\ k$
 by (induct k arbitrary: l, *simp-all*)

lemma *takeWhile-idem* [*simp*]:
 $takeWhile\ P (takeWhile\ P\ xs) = takeWhile\ P\ xs$
 by (induct xs) auto

lemma *dropWhile-idem* [*simp*]:

dropWhile P (dropWhile P xs) = dropWhile P xs
by (*induct xs*) *auto*

67.1.16 *zip*

lemma *zip-Nil* [*simp*]: *zip [] ys = []*
by (*induct ys*) *auto*

lemma *zip-Cons-Cons* [*simp*]: *zip (x # xs) (y # ys) = (x, y) # zip xs ys*
by *simp*

declare *zip-Cons* [*simp del*]

lemma [*code*]:
zip [] ys = []
zip xs [] = []
zip (x # xs) (y # ys) = (x, y) # zip xs ys
by (*fact zip-Nil zip.simps(1) zip-Cons-Cons*)**+**

lemma *zip-Cons1*:
zip (x#xs) ys = (case ys of [] => [] | y#ys => (x,y)#zip xs ys)
by(*auto split:list.split*)

lemma *length-zip* [*simp*]:
length (zip xs ys) = min (length xs) (length ys)
by (*induct xs ys rule:list-induct2'*) *auto*

lemma *zip-obtain-same-length*:
assumes $\bigwedge zs\ ws\ n.\ length\ zs = length\ ws \implies n = \min (length\ xs) (length\ ys)$
 $\implies zs = take\ n\ xs \implies ws = take\ n\ ys \implies P (zip\ zs\ ws)$
shows $P (zip\ xs\ ys)$

proof –

let $?n = \min (length\ xs) (length\ ys)$
have $P (zip (take\ ?n\ xs) (take\ ?n\ ys))$
by (*rule assms*) *simp-all*

moreover have $zip\ xs\ ys = zip (take\ ?n\ xs) (take\ ?n\ ys)$

proof (*induct xs arbitrary: ys*)

case Nil then show $?case$ **by** *simp*

next

case (Cons x xs) then show $?case$ **by** (*cases ys*) *simp-all*

qed

ultimately show $?thesis$ **by** *simp*

qed

lemma *zip-append1*:
zip (xs @ ys) zs =
zip xs (take (length xs) zs) @ zip ys (drop (length xs) zs)
by (*induct xs zs rule:list-induct2'*) *auto*

lemma *zip-append2*:

```
zip xs (ys @ zs) =
zip (take (length ys) xs) ys @ zip (drop (length ys) xs) zs
by (induct xs ys rule:list-induct2') auto
```

lemma *zip-append* [*simp*]:

```
[[length xs = length us]] ==>
zip (xs@ys) (us@vs) = zip xs us @ zip ys vs
by (simp add: zip-append1)
```

lemma *zip-rev*:

```
length xs = length ys ==> zip (rev xs) (rev ys) = rev (zip xs ys)
by (induct rule:list-induct2, simp-all)
```

lemma *zip-map-map*:

```
zip (map f xs) (map g ys) = map (λ (x, y). (f x, g y)) (zip xs ys)
proof (induct xs arbitrary: ys)
case (Cons x xs) note Cons-x-xs = Cons.hyps
show ?case
proof (cases ys)
case (Cons y ys')
show ?thesis unfolding Cons using Cons-x-xs by simp
qed simp
qed simp
```

lemma *zip-map1*:

```
zip (map f xs) ys = map (λ(x, y). (f x, y)) (zip xs ys)
using zip-map-map[of f xs λx. x ys] by simp
```

lemma *zip-map2*:

```
zip xs (map f ys) = map (λ(x, y). (x, f y)) (zip xs ys)
using zip-map-map[of λx. x xs f ys] by simp
```

lemma *map-zip-map*:

```
map f (zip (map g xs) ys) = map (%(x,y). f(g x, y)) (zip xs ys)
by (auto simp: zip-map1)
```

lemma *map-zip-map2*:

```
map f (zip xs (map g ys)) = map (%(x,y). f(x, g y)) (zip xs ys)
by (auto simp: zip-map2)
```

Courtesy of Andreas Lochbihler:

lemma *zip-same-conv-map*: $zip\ xs\ xs = map\ (\lambda x. (x, x))\ xs$
by(induct xs) auto

lemma *nth-zip* [*simp*]:

```
[[i < length xs; i < length ys]] ==> (zip xs ys)!i = (xs!i, ys!i)
proof (induct ys arbitrary: i xs)
case (Cons y ys)
```



```

then show ?case
  by (cases xs) (simp-all add: nth.simps split: nat.split)
qed auto

```

```

lemma set-zip:
  set (zip xs ys) = {(xs!i, ys!i) | i. i < min (length xs) (length ys)}
  by(simp add: set-conv-nth cong: rev-conj-cong)

```

```

lemma zip-same: ((a,b) ∈ set (zip xs xs)) = (a ∈ set xs ∧ a = b)
  by(induct xs) auto

```

```

lemma zip-update: zip (xs[i:=x]) (ys[i:=y]) = (zip xs ys)[i:=(x,y)]
  by (simp add: update-zip)

```

```

lemma zip-replicate [simp]:
  zip (replicate i x) (replicate j y) = replicate (min i j) (x,y)
proof (induct i arbitrary: j)
  case (Suc i)
  then show ?case
    by (cases j, auto)
qed auto

```

```

lemma zip-replicate1: zip (replicate n x) ys = map (Pair x) (take n ys)
  by(induction ys arbitrary: n)(case-tac [2] n, simp-all)

```

```

lemma take-zip: take n (zip xs ys) = zip (take n xs) (take n ys)
proof (induct n arbitrary: xs ys)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs; cases ys) simp-all
qed

```

```

lemma drop-zip: drop n (zip xs ys) = zip (drop n xs) (drop n ys)
proof (induct n arbitrary: xs ys)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs; cases ys) simp-all
qed

```

```

lemma zip-takeWhile-fst: zip (takeWhile P xs) ys = takeWhile (P ∘ fst) (zip xs
ys)
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by simp
next

```

case *Cons*
then show *?case* **by** (*cases ys*) *auto*
qed

lemma *zip-takeWhile-snd*: $\text{zip } xs \ (\text{takeWhile } P \ ys) = \text{takeWhile } (P \circ \text{snd}) \ (\text{zip } xs \ ys)$

proof (*induct xs arbitrary: ys*)

case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases ys*) *auto*
qed

lemma *set-zip-leftD*: $(x,y) \in \text{set } (\text{zip } xs \ ys) \implies x \in \text{set } xs$
by (*induct xs ys rule:list-induct2'*) *auto*

lemma *set-zip-rightD*: $(x,y) \in \text{set } (\text{zip } xs \ ys) \implies y \in \text{set } ys$
by (*induct xs ys rule:list-induct2'*) *auto*

lemma *in-set-zipE*:

$(x,y) \in \text{set}(\text{zip } xs \ ys) \implies (\llbracket x \in \text{set } xs; y \in \text{set } ys \rrbracket \implies R) \implies R$
by (*blast dest: set-zip-leftD set-zip-rightD*)

lemma *zip-map-fst-snd*: $\text{zip } (\text{map } \text{fst } zs) \ (\text{map } \text{snd } zs) = zs$
by (*induct zs*) *simp-all*

lemma *zip-eq-conv*:

$\text{length } xs = \text{length } ys \implies \text{zip } xs \ ys = zs \iff \text{map } \text{fst } zs = xs \wedge \text{map } \text{snd } zs = ys$
by (*auto simp add: zip-map-fst-snd*)

lemma *in-set-zip*:

$p \in \text{set } (\text{zip } xs \ ys) \iff (\exists n. xs \ ! \ n = \text{fst } p \wedge ys \ ! \ n = \text{snd } p \wedge n < \text{length } xs \wedge n < \text{length } ys)$
by (*cases p*) (*auto simp add: set-zip*)

lemma *in-set-impl-in-set-zip1*:

assumes $\text{length } xs = \text{length } ys$
assumes $x \in \text{set } xs$
obtains y **where** $(x, y) \in \text{set } (\text{zip } xs \ ys)$
proof –
from *assms* **have** $x \in \text{set } (\text{map } \text{fst } (\text{zip } xs \ ys))$ **by** *simp*
from *this* **that** **show** *?thesis* **by** *fastforce*
qed

lemma *in-set-impl-in-set-zip2*:

assumes $\text{length } xs = \text{length } ys$
assumes $y \in \text{set } ys$
obtains x **where** $(x, y) \in \text{set } (\text{zip } xs \ ys)$

proof –
from *assms* **have** $y \in \text{set } (\text{map } \text{snd } (\text{zip } xs \ ys))$ **by** *simp*
from *this* **that** **show** *?thesis* **by** *fastforce*
qed

lemma *zip-eq-Nil-iff*[*simp*]:
 $\text{zip } xs \ ys = [] \longleftrightarrow xs = [] \vee ys = []$
by (*cases xs*; *cases ys*) *simp-all*

lemmas *Nil-eq-zip-iff*[*simp*] = *zip-eq-Nil-iff*[*THEN eq-iff-swap*]

lemma *zip-eq-ConsE*:
assumes $\text{zip } xs \ ys = xy \# \ xys$
obtains $x \ xs' \ y \ ys'$ **where** $xs = x \# \ xs'$
and $ys = y \# \ ys'$ **and** $xy = (x, y)$
and $xys = \text{zip } xs' \ ys'$
proof –
from *assms* **have** $xs \neq []$ **and** $ys \neq []$
using *zip-eq-Nil-iff* [*of xs ys*] **by** *simp-all*
then obtain $x \ xs' \ y \ ys'$ **where** $xs = x \# \ xs'$
and $ys = y \# \ ys'$
by (*cases xs*; *cases ys*) *auto*
with *assms* **have** $xy = (x, y)$ **and** $xys = \text{zip } xs' \ ys'$
by *simp-all*
with $xs \ ys$ **show** *?thesis* ..
qed

lemma *semilattice-map2*:
 $\text{semilattice } (\text{map2 } (*))$ **if** $\text{semilattice } (*)$
for f (*infixl* * 70)
proof –
from *that* **interpret** *semilattice f* .
show *?thesis*
proof
show $\text{map2 } (*) (\text{map2 } (*) \ xs \ ys) \ zs = \text{map2 } (*) \ xs \ (\text{map2 } (*) \ ys \ zs)$
for $xs \ ys \ zs :: 'a \ \text{list}$
proof (*induction zip xs (zip ys zs) arbitrary: xs ys zs*)
case Nil
from Nil [*symmetric*] **show** *?case*
by *auto*
next
case (Cons xyz xys)
from *Cons.hyps*(2) [*symmetric*] **show** *?case*
by (*rule zip-eq-ConsE*) (*erule zip-eq-ConsE*,
auto intro: Cons.hyps(1) *simp add: ac-simps*)
qed
show $\text{map2 } (*) \ xs \ ys = \text{map2 } (*) \ ys \ xs$
for $xs \ ys :: 'a \ \text{list}$
proof (*induction zip xs ys arbitrary: xs ys*)

```

    case Nil
    then show ?case
    by auto
  next
  case (Cons xy xys)
  from Cons.hyps(2) [symmetric] show ?case
  by (rule zip-eq-ConsE) (auto intro: Cons.hyps(1) simp add: ac-simps)
qed
show map2 (*) xs xs = xs
for xs :: 'a list
by (induction xs) simp-all
qed
qed

```

lemma *pair-list-eqI*:

```

  assumes map fst xs = map fst ys and map snd xs = map snd ys
  shows xs = ys
proof -
  from assms(1) have length xs = length ys by (rule map-eq-imp-length-eq)
  from this assms show ?thesis
  by (induct xs ys rule: list-induct2) (simp-all add: prod-eqI)
qed

```

lemma *hd-zip*:

```

⟨hd (zip xs ys) = (hd xs, hd ys)⟩ if ⟨xs ≠ []⟩ and ⟨ys ≠ []⟩
using that by (cases xs; cases ys) simp-all

```

lemma *last-zip*:

```

⟨last (zip xs ys) = (last xs, last ys)⟩ if ⟨xs ≠ []⟩ and ⟨ys ≠ []⟩
and ⟨length xs = length ys⟩
using that by (cases xs rule: rev-cases; cases ys rule: rev-cases) simp-all

```

67.1.17 *list-all2*

lemma *list-all2-lengthD* [*intro?*]:

```

list-all2 P xs ys  $\implies$  length xs = length ys
by (simp add: list-all2-iff)

```

lemma *list-all2-Nil* [*iff, code*]: *list-all2* P [] ys = (ys = [])

```

by (simp add: list-all2-iff)

```

lemma *list-all2-Nil2* [*iff, code*]: *list-all2* P xs [] = (xs = [])

```

by (simp add: list-all2-iff)

```

lemma *list-all2-Cons* [*iff, code*]:

```

list-all2 P (x # xs) (y # ys) = (P x y  $\wedge$  list-all2 P xs ys)
by (auto simp add: list-all2-iff)

```

lemma *list-all2-Cons1*:

$list-all2\ P\ (x\ \#\ xs)\ ys = (\exists z\ zs.\ ys = z\ \#\ zs \wedge P\ x\ z \wedge list-all2\ P\ xs\ zs)$
by (cases ys) auto

lemma list-all2-Cons2:

$list-all2\ P\ xs\ (y\ \#\ ys) = (\exists z\ zs.\ xs = z\ \#\ zs \wedge P\ z\ y \wedge list-all2\ P\ zs\ ys)$
by (cases xs) auto

lemma list-all2-induct

[consumes 1, case-names Nil Cons, induct set: list-all2]:

assumes P: list-all2 P xs ys

assumes Nil: R [] []

assumes Cons: $\bigwedge x\ xs\ y\ ys.$

$\llbracket P\ x\ y; list-all2\ P\ xs\ ys; R\ xs\ ys \rrbracket \implies R\ (x\ \#\ xs)\ (y\ \#\ ys)$

shows R xs ys

using P

by (induct xs arbitrary: ys) (auto simp add: list-all2-Cons1 Nil Cons)

lemma list-all2-rev [iff]:

$list-all2\ P\ (rev\ xs)\ (rev\ ys) = list-all2\ P\ xs\ ys$

by (simp add: list-all2-iff zip-rev cong: conj-cong)

lemma list-all2-rev1:

$list-all2\ P\ (rev\ xs)\ ys = list-all2\ P\ xs\ (rev\ ys)$

by (subst list-all2-rev [symmetric]) simp

lemma list-all2-append1:

$list-all2\ P\ (xs\ @\ ys)\ zs =$

$(\exists us\ vs.\ zs = us\ @\ vs \wedge length\ us = length\ xs \wedge length\ vs = length\ ys \wedge$
 $list-all2\ P\ xs\ us \wedge list-all2\ P\ ys\ vs)$ (is ?lhs = ?rhs)

proof

assume ?lhs

then show ?rhs

apply (rule-tac x = take (length xs) zs in exI)

apply (rule-tac x = drop (length xs) zs in exI)

apply (force split: nat-diff-split simp add: list-all2-iff zip-append1)

done

next

assume ?rhs

then show ?lhs

by (auto simp add: list-all2-iff)

qed

lemma list-all2-append2:

$list-all2\ P\ xs\ (ys\ @\ zs) =$

$(\exists us\ vs.\ xs = us\ @\ vs \wedge length\ us = length\ ys \wedge length\ vs = length\ zs \wedge$
 $list-all2\ P\ us\ ys \wedge list-all2\ P\ vs\ zs)$ (is ?lhs = ?rhs)

proof

assume ?lhs

then show ?rhs

```

apply (rule-tac x = take (length ys) xs in exI)
apply (rule-tac x = drop (length ys) xs in exI)
apply (force split: nat-diff-split simp add: list-all2-iff zip-append2)
done
next
assume ?rhs
then show ?lhs
  by (auto simp add: list-all2-iff)
qed

```

lemma *list-all2-append*:
 $length\ xs = length\ ys \implies$
 $list\text{-}all2\ P\ (xs@us)\ (ys@vs) = (list\text{-}all2\ P\ xs\ ys \wedge list\text{-}all2\ P\ us\ vs)$
by (induct rule: list-induct2, simp-all)

lemma *list-all2-appendI* [*intro?*, *trans*]:
 $\llbracket list\text{-}all2\ P\ a\ b; list\text{-}all2\ P\ c\ d \rrbracket \implies list\text{-}all2\ P\ (a@c)\ (b@d)$
by (simp add: list-all2-append list-all2-lengthD)

lemma *list-all2-conv-all-nth*:
 $list\text{-}all2\ P\ xs\ ys =$
 $(length\ xs = length\ ys \wedge (\forall i < length\ xs.\ P\ (xs!i)\ (ys!i)))$
by (force simp add: list-all2-iff set-zip)

lemma *list-all2-trans*:
assumes *tr*: $!!a\ b\ c.\ P1\ a\ b \implies P2\ b\ c \implies P3\ a\ c$
shows $!!bs\ cs.\ list\text{-}all2\ P1\ as\ bs \implies list\text{-}all2\ P2\ bs\ cs \implies list\text{-}all2\ P3\ as\ cs$
 (**is** $!!bs\ cs.\ PROP\ ?Q\ as\ bs\ cs$)
proof (induct *as*)
fix *x xs bs* **assume** *I1*: $!!bs\ cs.\ PROP\ ?Q\ xs\ bs\ cs$
show $!!cs.\ PROP\ ?Q\ (x \# xs)\ bs\ cs$
proof (induct *bs*)
fix *y ys cs* **assume** *I2*: $!!cs.\ PROP\ ?Q\ (x \# xs)\ ys\ cs$
show $PROP\ ?Q\ (x \# xs)\ (y \# ys)\ cs$
by (induct *cs*) (auto intro: *tr* *I1* *I2*)
qed *simp*
qed *simp*

lemma *list-all2-all-nthI* [*intro?*]:
 $length\ a = length\ b \implies (\bigwedge n.\ n < length\ a \implies P\ (a!n)\ (b!n)) \implies list\text{-}all2\ P\ a\ b$
by (simp add: list-all2-conv-all-nth)

lemma *list-all2I*:
 $\forall x \in set\ (zip\ a\ b).\ case\text{-}prod\ P\ x \implies length\ a = length\ b \implies list\text{-}all2\ P\ a\ b$
by (simp add: list-all2-iff)

lemma *list-all2-nthD*:
 $\llbracket list\text{-}all2\ P\ xs\ ys; p < size\ xs \rrbracket \implies P\ (xs!p)\ (ys!p)$
by (simp add: list-all2-conv-all-nth)

lemma *list-all2-nthD2*:

$\llbracket \text{list-all2 } P \text{ } xs \text{ } ys; p < \text{size } ys \rrbracket \implies P (xs!p) (ys!p)$
by (*frule list-all2-lengthD*) (*auto intro: list-all2-nthD*)

lemma *list-all2-map1*:

$\text{list-all2 } P (\text{map } f \text{ } as) \text{ } bs = \text{list-all2 } (\lambda x y. P (f x) y) \text{ } as \text{ } bs$
by (*simp add: list-all2-conv-all-nth*)

lemma *list-all2-map2*:

$\text{list-all2 } P \text{ } as (\text{map } f \text{ } bs) = \text{list-all2 } (\lambda x y. P x (f y)) \text{ } as \text{ } bs$
by (*auto simp add: list-all2-conv-all-nth*)

lemma *list-all2-refl* [*intro?*]:

$(\bigwedge x. P x x) \implies \text{list-all2 } P \text{ } xs \text{ } xs$
by (*simp add: list-all2-conv-all-nth*)

lemma *list-all2-update-cong*:

$\llbracket \text{list-all2 } P \text{ } xs \text{ } ys; P x y \rrbracket \implies \text{list-all2 } P (xs[i:=x]) (ys[i:=y])$
by (*cases i < length ys*) (*auto simp add: list-all2-conv-all-nth nth-list-update*)

lemma *list-all2-takeI* [*simp,intro?*]:

$\text{list-all2 } P \text{ } xs \text{ } ys \implies \text{list-all2 } P (\text{take } n \text{ } xs) (\text{take } n \text{ } ys)$

proof (*induct xs arbitrary: n ys*)

case (*Cons x xs*)

then show *?case*

by (*cases n*) (*auto simp: list-all2-Cons1*)

qed *auto*

lemma *list-all2-dropI* [*simp,intro?*]:

$\text{list-all2 } P \text{ } xs \text{ } ys \implies \text{list-all2 } P (\text{drop } n \text{ } xs) (\text{drop } n \text{ } ys)$

proof (*induct xs arbitrary: n ys*)

case (*Cons x xs*)

then show *?case*

by (*cases n*) (*auto simp: list-all2-Cons1*)

qed *auto*

lemma *list-all2-mono* [*intro?*]:

$\text{list-all2 } P \text{ } xs \text{ } ys \implies (\bigwedge xs \text{ } ys. P \text{ } xs \text{ } ys \implies Q \text{ } xs \text{ } ys) \implies \text{list-all2 } Q \text{ } xs \text{ } ys$
by (*rule list.rel-mono-strong*)

lemma *list-all2-eq*:

$xs = ys \iff \text{list-all2 } (=) \text{ } xs \text{ } ys$

by (*induct xs ys rule: list-induct2'*) *auto*

lemma *list-eq-iff-zip-eq*:

$xs = ys \iff \text{length } xs = \text{length } ys \wedge (\forall (x,y) \in \text{set } (\text{zip } xs \text{ } ys). x = y)$

by(*auto simp add: set-zip list-all2-eq list-all2-conv-all-nth cong: conj-cong*)

lemma *list-all2-same*: $list\text{-}all2\ P\ xs\ xs \longleftrightarrow (\forall x \in set\ xs.\ P\ x\ x)$
by (*auto simp add: list-all2-conv-all-nth set-conv-nth*)

lemma *zip-assoc*:
 $zip\ xs\ (zip\ ys\ zs) = map\ (\lambda((x, y), z).\ (x, y, z))\ (zip\ (zip\ xs\ ys)\ zs)$
by (*rule list-all2-all-nthI[where P=(=), unfolded list.rel-eq]*) *simp-all*

lemma *zip-commute*: $zip\ xs\ ys = map\ (\lambda(x, y).\ (y, x))\ (zip\ ys\ xs)$
by (*rule list-all2-all-nthI[where P=(=), unfolded list.rel-eq]*) *simp-all*

lemma *zip-left-commute*:
 $zip\ xs\ (zip\ ys\ zs) = map\ (\lambda(y, (x, z)).\ (x, y, z))\ (zip\ ys\ (zip\ xs\ zs))$
by (*rule list-all2-all-nthI[where P=(=), unfolded list.rel-eq]*) *simp-all*

lemma *zip-replicate2*: $zip\ xs\ (replicate\ n\ y) = map\ (\lambda x.\ (x, y))\ (take\ n\ xs)$
by (*subst zip-commute*)(*simp add: zip-replicate1*)

67.1.18 *List.product and product-lists*

lemma *product-concat-map*:
 $List.\text{product}\ xs\ ys = concat\ (map\ (\lambda x.\ map\ (\lambda y.\ (x, y))\ ys)\ xs)$
by (*induction xs*) (*simp*)+

lemma *set-product[simp]*: $set\ (List.\text{product}\ xs\ ys) = set\ xs \times set\ ys$
by (*induct xs*) *auto*

lemma *length-product [simp]*:
 $length\ (List.\text{product}\ xs\ ys) = length\ xs * length\ ys$
by (*induct xs*) *simp-all*

lemma *product-nth*:
assumes $n < length\ xs * length\ ys$
shows $List.\text{product}\ xs\ ys ! n = (xs ! (n \text{ div } length\ ys), ys ! (n \text{ mod } length\ ys))$
using *assms proof* (*induct xs arbitrary: n*)
case Nil then show *?case by simp*
next
case (*Cons x xs n*)
then have $length\ ys > 0$ **by** *auto*
with *Cons show* *?case*
by (*auto simp add: nth-append not-less le-mod-geq le-div-geq*)
qed

lemma *in-set-product-lists-length*:
 $xs \in set\ (product\text{-lists}\ xss) \implies length\ xs = length\ xss$
by (*induct xss arbitrary: xs*) *auto*

lemma *product-lists-set*:
 $set\ (product\text{-lists}\ xss) = \{xs.\ list\text{-}all2\ (\lambda x\ ys.\ x \in set\ ys)\ xs\ xss\}$ (**is** *?L = Collect*
?R)


```

proof (intro equalityI subsetI, unfold mem-Collect-eq)
  fix xs assume xs ∈ ?L
  then have length xs = length xss by (rule in-set-product-lists-length)
  from this ⟨xs ∈ ?L⟩ show ?R xs by (induct xs xss rule: list-induct2) auto
next
  fix xs assume ?R xs
  then show xs ∈ ?L by induct auto
qed

```

67.1.19 fold with natural argument order

lemma *fold-simps* [*code*]: — eta-expanded variant for generated code – enables tail-recursion optimisation in Scala

```

fold f [] s = s
fold f (x # xs) s = fold f xs (f x s)
by simp-all

```

lemma *fold-remove1-split*:

```

[[  $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies f x \circ f y = f y \circ f x;$ 
 $x \in \text{set } xs$  ]]
 $\implies \text{fold } f \text{ xs} = \text{fold } f (\text{remove1 } x \text{ xs}) \circ f x$ 
by (induct xs) (auto simp add: comp-assoc)

```

lemma *fold-cong* [*fundef-cong*]:

```

 $a = b \implies xs = ys \implies (\bigwedge x. x \in \text{set } xs \implies f x = g x)$ 
 $\implies \text{fold } f \text{ xs } a = \text{fold } g \text{ ys } b$ 
by (induct ys arbitrary: a b xs) simp-all

```

lemma *fold-id*: ($\bigwedge x. x \in \text{set } xs \implies f x = \text{id}$) $\implies \text{fold } f \text{ xs} = \text{id}$
by (induct xs) *simp-all*

lemma *fold-commute*:

```

( $\bigwedge x. x \in \text{set } xs \implies h \circ g x = f x \circ h$ )  $\implies h \circ \text{fold } g \text{ xs} = \text{fold } f \text{ xs} \circ h$ 
by (induct xs) (simp-all add: fun-eq-iff)

```

lemma *fold-commute-apply*:

```

assumes  $\bigwedge x. x \in \text{set } xs \implies h \circ g x = f x \circ h$ 
shows  $h (\text{fold } g \text{ xs } s) = \text{fold } f \text{ xs} (h s)$ 

```

proof –

```

from assms have  $h \circ \text{fold } g \text{ xs} = \text{fold } f \text{ xs} \circ h$  by (rule fold-commute)
then show ?thesis by (simp add: fun-eq-iff)

```

qed

lemma *fold-invariant*:

```

[[  $\bigwedge x. x \in \text{set } xs \implies Q x; P s; \bigwedge x s. Q x \implies P s \implies P (f x s)$  ]]
 $\implies P (\text{fold } f \text{ xs } s)$ 
by (induct xs arbitrary: s) simp-all

```

lemma *fold-append* [*simp*]: $\text{fold } f (xs @ ys) = \text{fold } f \text{ ys} \circ \text{fold } f \text{ xs}$

by (*induct xs*) *simp-all*

lemma *fold-map* [*code-unfold*]: $\text{fold } g \text{ (map } f \text{ } xs) = \text{fold } (g \circ f) \text{ } xs$
by (*induct xs*) *simp-all*

lemma *fold-filter*:
 $\text{fold } f \text{ (filter } P \text{ } xs) = \text{fold } (\lambda x. \text{ if } P \text{ } x \text{ then } f \text{ } x \text{ else id) } xs$
by (*induct xs*) *simp-all*

lemma *fold-rev*:
 $(\bigwedge x \ y. x \in \text{set } xs \implies y \in \text{set } xs \implies f \ y \circ f \ x = f \ x \circ f \ y)$
 $\implies \text{fold } f \text{ (rev } xs) = \text{fold } f \text{ } xs$
by (*induct xs*) (*simp-all add: fold-commute-apply fun-eq-iff*)

lemma *fold-Cons-rev*: $\text{fold } \text{Cons } xs = \text{append } (\text{rev } xs)$
by (*induct xs*) *simp-all*

lemma *rev-conv-fold* [*code*]: $\text{rev } xs = \text{fold } \text{Cons } xs \ []$
by (*simp add: fold-Cons-rev*)

lemma *fold-append-concat-rev*: $\text{fold } \text{append } xss = \text{append } (\text{concat } (\text{rev } xss))$
by (*induct xss*) *simp-all*

Finite-Set.fold and *fold*

lemma (**in** *comp-fun-commute-on*) *fold-set-fold-remdups*:
assumes $\text{set } xs \subseteq S$
shows $\text{Finite-Set.fold } f \ y \ (\text{set } xs) = \text{fold } f \ (\text{remdups } xs) \ y$
by (*rule sym, use assms in <induct xs arbitrary: y>*)
(simp-all add: insert-absorb fold-fun-left-comm)

lemma (**in** *comp-fun-idem-on*) *fold-set-fold*:
assumes $\text{set } xs \subseteq S$
shows $\text{Finite-Set.fold } f \ y \ (\text{set } xs) = \text{fold } f \ xs \ y$
by (*rule sym, use assms in <induct xs arbitrary: y>*) (*simp-all add: fold-fun-left-comm*)

lemma *union-set-fold* [*code*]: $\text{set } xs \cup A = \text{fold } \text{Set.insert } xs \ A$
proof –

interpret *comp-fun-idem* *Set.insert*
by (*fact comp-fun-idem-insert*)
show *?thesis* **by** (*simp add: union-fold-insert fold-set-fold*)
qed

lemma *union-coset-filter* [*code*]:
 $\text{List.coset } xs \cup A = \text{List.coset } (\text{List.filter } (\lambda x. x \notin A) \ xs)$
by *auto*

lemma *minus-set-fold* [*code*]: $A - \text{set } xs = \text{fold } \text{Set.remove } xs \ A$
proof –
interpret *comp-fun-idem* *Set.remove*

```

    by (fact comp-fun-idem-remove)
  show ?thesis
    by (simp add: minus-fold-remove [of - A] fold-set-fold)
qed

```

```

lemma minus-coset-filter [code]:
  A - List.coset xs = set (List.filter (λx. x ∈ A) xs)
  by auto

```

```

lemma inter-set-filter [code]:
  A ∩ set xs = set (List.filter (λx. x ∈ A) xs)
  by auto

```

```

lemma inter-coset-fold [code]:
  A ∩ List.coset xs = fold Set.remove xs A
  by (simp add: Diff-eq [symmetric] minus-set-fold)

```

```

lemma (in semilattice-set) set-eq-fold [code]:
  F (set (x # xs)) = fold f xs x
proof -
  interpret comp-fun-idem f
    by standard (simp-all add: fun-eq-iff left-commute)
  show ?thesis by (simp add: eq-fold fold-set-fold)
qed

```

```

lemma (in complete-lattice) Inf-set-fold:
  Inf (set xs) = fold inf xs top
proof -
  interpret comp-fun-idem inf :: 'a ⇒ 'a ⇒ 'a
    by (fact comp-fun-idem-inf)
  show ?thesis by (simp add: Inf-fold-inf fold-set-fold inf-commute)
qed

```

```

declare Inf-set-fold [where 'a = 'a set, code]

```

```

lemma (in complete-lattice) Sup-set-fold:
  Sup (set xs) = fold sup xs bot
proof -
  interpret comp-fun-idem sup :: 'a ⇒ 'a ⇒ 'a
    by (fact comp-fun-idem-sup)
  show ?thesis by (simp add: Sup-fold-sup fold-set-fold sup-commute)
qed

```

```

declare Sup-set-fold [where 'a = 'a set, code]

```

```

lemma (in complete-lattice) INF-set-fold:
  ⌊ (f ' set xs) = fold (inf ∘ f) xs top
  using Inf-set-fold [of map f xs] by (simp add: fold-map)

```

lemma (in *complete-lattice*) *SUP-set-fold*:
 $\sqcup (f \text{ ' set } xs) = \text{fold } (sup \circ f) \text{ xs bot}$
using *Sup-set-fold* [of *map f xs*] **by** (*simp add: fold-map*)

67.1.20 Fold variants: *foldr* and *foldl*

Correspondence

lemma *foldr-conv-fold* [*code-abbrev*]: $\text{foldr } f \text{ xs} = \text{fold } f \text{ (rev xs)}$
by (*induct xs*) *simp-all*

lemma *foldl-conv-fold*: $\text{foldl } f \text{ s xs} = \text{fold } (\lambda x \text{ s. } f \text{ s } x) \text{ xs s}$
by (*induct xs arbitrary: s*) *simp-all*

lemma *foldr-conv-foldl*: — The “Third Duality Theorem” in Bird & Wadler:
 $\text{foldr } f \text{ xs } a = \text{foldl } (\lambda x \text{ y. } f \text{ y } x) \text{ a (rev xs)}$
by (*simp add: foldr-conv-fold foldl-conv-fold*)

lemma *foldl-conv-foldr*:
 $\text{foldl } f \text{ a xs} = \text{foldr } (\lambda x \text{ y. } f \text{ y } x) \text{ (rev xs) a}$
by (*simp add: foldr-conv-fold foldl-conv-fold*)

lemma *foldr-fold*:
 $(\bigwedge x \text{ y. } x \in \text{set } xs \implies y \in \text{set } xs \implies f \text{ y } \circ f \text{ x} = f \text{ x } \circ f \text{ y})$
 $\implies \text{foldr } f \text{ xs} = \text{fold } f \text{ xs}$
unfolding *foldr-conv-fold* **by** (*rule fold-rev*)

lemma *foldr-cong* [*fundef-cong*]:
 $a = b \implies l = k \implies (\bigwedge a \text{ x. } x \in \text{set } l \implies f \text{ x } a = g \text{ x } a) \implies \text{foldr } f \text{ l } a = \text{foldr } g \text{ k } b$
by (*auto simp add: foldr-conv-fold intro!: fold-cong*)

lemma *foldl-cong* [*fundef-cong*]:
 $a = b \implies l = k \implies (\bigwedge a \text{ x. } x \in \text{set } l \implies f \text{ a } x = g \text{ a } x) \implies \text{foldl } f \text{ a } l = \text{foldl } g \text{ b } k$
by (*auto simp add: foldl-conv-fold intro!: fold-cong*)

lemma *foldr-append* [*simp*]: $\text{foldr } f \text{ (xs @ ys) } a = \text{foldr } f \text{ xs } (\text{foldr } f \text{ ys } a)$
by (*simp add: foldr-conv-fold*)

lemma *foldl-append* [*simp*]: $\text{foldl } f \text{ a (xs @ ys)} = \text{foldl } f \text{ (foldl } f \text{ a xs) ys}$
by (*simp add: foldl-conv-fold*)

lemma *foldr-map* [*code-unfold*]: $\text{foldr } g \text{ (map } f \text{ xs) } a = \text{foldr } (g \circ f) \text{ xs } a$
by (*simp add: foldr-conv-fold fold-map rev-map*)

lemma *foldr-filter*:
 $\text{foldr } f \text{ (filter } P \text{ xs)} = \text{foldr } (\lambda x. \text{ if } P \text{ x then } f \text{ x else id) \text{ xs}$
by (*simp add: foldr-conv-fold rev-filter fold-filter*)

lemma *foldl-map* [*code-unfold*]:
 $foldl\ g\ a\ (map\ f\ xs) = foldl\ (\lambda a\ x.\ g\ a\ (f\ x))\ a\ xs$
by (*simp add: foldl-conv-fold fold-map comp-def*)

lemma *concat-conv-foldr* [*code*]:
 $concat\ xss = foldr\ append\ xss\ []$
by (*simp add: fold-append-concat-rev foldr-conv-fold*)

67.1.21 *upt*

lemma *upt-rec*[*code*]: $[i..<j] = (if\ i<j\ then\ i\#\ [Suc\ i..<j]\ else\ [])$
— *simp* does not terminate!
by (*induct j auto*)

lemmas *upt-rec-numeral*[*simp*] = *upt-rec*[*of numeral m numeral n*] **for** $m\ n$

lemma *upt-conv-Nil* [*simp*]: $j \leq i \implies [i..<j] = []$
by (*subst upt-rec simp*)

lemma *upt-eq-Nil-conv*[*simp*]: $([i..<j] = []) = (j = 0 \vee j \leq i)$
by(*induct j simp-all*)

lemma *upt-eq-Cons-conv*:
 $([i..<j] = x\#\ xs) = (i < j \wedge i = x \wedge [i+1..<j] = xs)$
proof (*induct j arbitrary: x xs*)
case (*Suc j*)
then show ?*case*
by (*simp add: upt-rec*)
qed *simp*

lemma *upt-Suc-append*: $i \leq j \implies [i..<(Suc\ j)] = [i..<j]@[j]$
— Only needed if *upt-Suc* is deleted from the simpset.
by *simp*

lemma *upt-conv-Cons*: $i < j \implies [i..<j] = i\#\ [Suc\ i..<j]$
by (*simp add: upt-rec*)

lemma *upt-conv-Cons-Cons*: — no precondition
 $m\ \#\ n\ \#\ ns = [m..<q] \longleftrightarrow n\ \#\ ns = [Suc\ m..<q]$
proof (*cases m < q*)
case *False* **then show** ?*thesis* **by** *simp*
next
case *True* **then show** ?*thesis* **by** (*simp add: upt-conv-Cons*)
qed

lemma *upt-add-eq-append*: $i <= j \implies [i..<j+k] = [i..<j]@[j..<j+k]$
— LOOPS as a simplrule, since $j \leq j$.
by (*induct k auto*)

lemma *length-upt* [*simp*]: $\text{length } [i..<j] = j - i$
by (*induct j*) (*auto simp add: Suc-diff-le*)

lemma *nth-upt* [*simp*]: $i + k < j \implies [i..<j] ! k = i + k$
by (*induct j*) (*auto simp add: less-Suc-eq nth-append split: nat-diff-split*)

lemma *hd-upt*[*simp*]: $i < j \implies \text{hd}[i..<j] = i$
by(*simp add:upt-conv-Cons*)

lemma *tl-upt* [*simp*]: $\text{tl } [m..<n] = [\text{Suc } m..<n]$
by (*simp add: upt-rec*)

lemma *last-upt*[*simp*]: $i < j \implies \text{last}[i..<j] = j - 1$
by(*cases j*)(*auto simp: upt-Suc-append*)

lemma *take-upt* [*simp*]: $i + m \leq n \implies \text{take } m [i..<n] = [i..<i+m]$
proof (*induct m arbitrary: i*)
case (*Suc m*)
then show ?*case*
by (*subst take-Suc-conv-app-nth*) *auto*
qed *simp*

lemma *drop-upt*[*simp*]: $\text{drop } m [i..<j] = [i+m..<j]$
by(*induct j*) *auto*

lemma *map-Suc-upt*: $\text{map } \text{Suc } [m..<n] = [\text{Suc } m..<\text{Suc } n]$
by (*induct n*) *auto*

lemma *map-add-upt*: $\text{map } (\lambda i. i + n) [0..<m] = [n..<m + n]$
by (*induct m*) *simp-all*

lemma *nth-map-upt*: $i < n - m \implies (\text{map } f [m..<n]) ! i = f(m+i)$
proof (*induct n m arbitrary: i rule: diff-induct*)
case ($\exists x y$)
then show ?*case*
by (*metis add.commute length-upt less-diff-conv nth-map nth-upt*)
qed *auto*

lemma *map-decr-upt*: $\text{map } (\lambda n. n - \text{Suc } 0) [\text{Suc } m..<\text{Suc } n] = [m..<n]$
by (*induct n*) *simp-all*

lemma *map-upt-Suc*: $\text{map } f [0 ..< \text{Suc } n] = f 0 \# \text{map } (\lambda i. f (\text{Suc } i)) [0 ..< n]$
by (*induct n arbitrary: f*) *auto*

lemma *nth-take-lemma*:

$k \leq \text{length } xs \implies k \leq \text{length } ys \implies$

$(\bigwedge i. i < k \longrightarrow xs!i = ys!i) \implies \text{take } k xs = \text{take } k ys$

proof (*induct k arbitrary: xs ys*)
case (*Suc k*)

then show *?case*
apply (*simp add: less-Suc-eq-0-disj*)
by (*simp add: Suc.premis(3) take-Suc-conv-app-nth*)
qed *simp*

lemma *nth-equalityI*:
 $\llbracket \text{length } xs = \text{length } ys; \bigwedge i. i < \text{length } xs \implies xs!i = ys!i \rrbracket \implies xs = ys$
by (*frule nth-take-lemma [OF le-refl eq-imp-le] simp-all*)

lemma *map-nth*:
 $\text{map } (\lambda i. xs ! i) [0..<\text{length } xs] = xs$
by (*rule nth-equalityI, auto*)

lemma *list-all2-antisym*:
 $\llbracket (\bigwedge x y. \llbracket P x y; Q y x \rrbracket \implies x = y); \text{list-all2 } P \text{ } xs \text{ } ys; \text{list-all2 } Q \text{ } ys \text{ } xs \rrbracket$
 $\implies xs = ys$
by (*simp add: list-all2-conv-all-nth nth-equalityI*)

lemma *take-equalityI*: $(\forall i. \text{take } i \text{ } xs = \text{take } i \text{ } ys) \implies xs = ys$
— The famous take-lemma.
by (*metis length-take min.commute order-refl take-all*)

lemma *take-Cons'*:
 $\text{take } n \text{ } (x \# xs) = (\text{if } n = 0 \text{ then } [] \text{ else } x \# \text{take } (n - 1) \text{ } xs)$
by (*cases n simp-all*)

lemma *drop-Cons'*:
 $\text{drop } n \text{ } (x \# xs) = (\text{if } n = 0 \text{ then } x \# xs \text{ else } \text{drop } (n - 1) \text{ } xs)$
by (*cases n simp-all*)

lemma *nth-Cons'*: $(x \# xs)!n = (\text{if } n = 0 \text{ then } x \text{ else } xs!(n - 1))$
by (*cases n simp-all*)

lemma *take-Cons-numeral* [*simp*]:
 $\text{take } (\text{numeral } v) \text{ } (x \# xs) = x \# \text{take } (\text{numeral } v - 1) \text{ } xs$
by (*simp add: take-Cons'*)

lemma *drop-Cons-numeral* [*simp*]:
 $\text{drop } (\text{numeral } v) \text{ } (x \# xs) = \text{drop } (\text{numeral } v - 1) \text{ } xs$
by (*simp add: drop-Cons'*)

lemma *nth-Cons-numeral* [*simp*]:
 $(x \# xs) ! \text{numeral } v = xs ! (\text{numeral } v - 1)$
by (*simp add: nth-Cons'*)

lemma *map-upt-eqI*:
 $\langle \text{map } f [m..<n] = xs \rangle \text{ if } \langle \text{length } xs = n - m \rangle$
 $\langle \bigwedge i. i < \text{length } xs \implies xs ! i = f (m + i) \rangle$
proof (*rule nth-equalityI*)

```

from ⟨length xs = n - m⟩ show ⟨length (map f [m.. $n$ ]) = length xs⟩
  by simp
next
  fix i
  assume ⟨i < length (map f [m.. $n$ ])⟩
  then have ⟨i < n - m⟩
    by simp
  with that have ⟨xs ! i = f (m + i)⟩
    by simp
  with ⟨i < n - m⟩ show ⟨map f [m.. $n$ ] ! i = xs ! i⟩
    by simp
qed

```

67.1.22 upto: interval-list on int

```

function upto :: int ⇒ int ⇒ int list ((1[-./-])) where
  upto i j = (if i ≤ j then i # [i+1..j] else [])
by auto
termination
by(relation measure(%(i::int,j). nat(j - i + 1))) auto

```

```

declare upto.simps[simp del]

```

```

lemmas upto-rec-numeral [simp] =
  upto.simps[of numeral m numeral n]
  upto.simps[of numeral m - numeral n]
  upto.simps[of - numeral m numeral n]
  upto.simps[of - numeral m - numeral n] for m n

```

```

lemma upto-empty[simp]: j < i ⇒ [i..j] = []
by(simp add: upto.simps)

```

```

lemma upto-single[simp]: [i..i] = [i]
by(simp add: upto.simps)

```

```

lemma upto-Nil[simp]: [i..j] = [] ⇔ j < i
by (simp add: upto.simps)

```

```

lemmas upto-Nil2[simp] = upto-Nil[THEN eq-iff-swap]

```

```

lemma upto-rec1: i ≤ j ⇒ [i..j] = i # [i+1..j]
by(simp add: upto.simps)

```

```

lemma upto-rec2: i ≤ j ⇒ [i..j] = [i..j - 1]@[j]

```

```

proof(induct nat(j-i) arbitrary: i j)
  case 0 thus ?case by(simp add: upto.simps)

```

```

next

```

```

  case (Suc n)

```

```

  hence n = nat (j - (i + 1)) i < j by linarith+

```


from *this*(2) *Suc.hyps*(1)[*OF this*(1)] *Suc*(2,3) *upto-rec1* **show** *?case* **by** *simp*
qed

lemma *length-upto*[*simp*]: $\text{length } [i..j] = \text{nat}(j - i + 1)$
by(*induction* *i j* *rule: upto.induct*) (*auto simp: upto.simps*)

lemma *set-upto*[*simp*]: $\text{set}[i..j] = \{i..j\}$
proof(*induct* *i j* *rule: upto.induct*)
 case (1 *i j*)
 from *this* **show** *?case*
 unfolding *upto.simps*[*of i j*] **by** *auto*
qed

lemma *nth-upto*[*simp*]: $i + \text{int } k \leq j \implies [i..j] ! k = i + \text{int } k$
proof(*induction* *i j* *arbitrary: k* *rule: upto.induct*)
 case (1 *i j*)
 then **show** *?case*
 by (*auto simp add: upto-rec1* [*of i j*] *nth-Cons'*)
qed

lemma *upto-split1*:
 $i \leq j \implies j \leq k \implies [i..k] = [i..j-1] @ [j..k]$
proof (*induction* *j* *rule: int-ge-induct*)
 case *base* **thus** *?case* **by** (*simp add: upto-rec1*)
next
 case *step* **thus** *?case* **using** *upto-rec1 upto-rec2* **by** *simp*
qed

lemma *upto-split2*:
 $i \leq j \implies j \leq k \implies [i..k] = [i..j] @ [j+1..k]$
using *upto-rec1 upto-rec2 upto-split1* **by** *auto*

lemma *upto-split3*: $\llbracket i \leq j; j \leq k \rrbracket \implies [i..k] = [i..j-1] @ j \# [j+1..k]$
using *upto-rec1 upto-split1* **by** *auto*

Tail recursive version for code generation:

definition *upto-aux* :: *int* \Rightarrow *int* \Rightarrow *int list* \Rightarrow *int list* **where**
 upto-aux *i j js* = $[i..j] @ js$

lemma *upto-aux-rec* [*code*]:
 upto-aux *i j js* = (*if* $j < i$ *then* *js* *else* *upto-aux* *i* ($j - 1$) ($j \# js$))
by (*simp add: upto-aux-def upto-rec2*)

lemma *upto-code*[*code*]: $[i..j] = \text{upto-aux } i j []$
by(*simp add: upto-aux-def*)

67.1.23 successively

lemma *successively-Cons*:

successively $P (x \# xs) \longleftrightarrow xs = [] \vee P x (hd xs) \wedge \text{successively } P xs$
by (*cases xs*) *auto*

lemma *successively-cong* [*cong*]:

assumes $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies P x y \longleftrightarrow Q x y$ $xs = ys$

shows *successively* $P xs \longleftrightarrow \text{successively } Q ys$

unfolding *assms*(2) [*symmetric*] **using** *assms*(1)

by (*induction xs*) (*auto simp: successively-Cons*)

lemma *successively-append-iff*:

successively $P (xs @ ys) \longleftrightarrow$

successively $P xs \wedge \text{successively } P ys \wedge$

$(xs = [] \vee ys = [] \vee P (\text{last } xs) (hd ys))$

by (*induction xs*) (*auto simp: successively-Cons*)

lemma *successively-if-sorted-wrt*: *sorted-wrt* $P xs \implies \text{successively } P xs$

by (*induction xs rule: induct-list012*) *auto*

lemma *successively-iff-sorted-wrt-strong*:

assumes $\bigwedge x y z. x \in \text{set } xs \implies y \in \text{set } xs \implies z \in \text{set } xs \implies$

$P x y \implies P y z \implies P x z$

shows *successively* $P xs \longleftrightarrow \text{sorted-wrt } P xs$

proof

assume *successively* $P xs$

from *this* **and** *assms* **show** *sorted-wrt* $P xs$

proof (*induction xs rule: induct-list012*)

case ($\exists x y xs$)

from *3.prem*s **have** $P x y$

by *auto*

have *IH*: *sorted-wrt* $P (y \# xs)$

using *3.prem*s

by (*intro 3.IH*(2) *list.set-intros*(2))(*simp, blast intro: list.set-intros*(2))

have $P x z$ **if** *asm*: $z \in \text{set } xs$ **for** z

proof –

from *IH* **and** *asm* **have** $P y z$

by *auto*

with $\langle P x y \rangle$ **show** $P x z$

using *3.prem*s *asm* **by** *auto*

qed

with *IH* **and** $\langle P x y \rangle$ **show** ?*case* **by** *auto*

qed *auto*

qed (*use successively-if-sorted-wrt in blast*)

lemma *successively-conv-sorted-wrt*:

assumes *transp* P

shows *successively* $P xs \longleftrightarrow \text{sorted-wrt } P xs$

using *assms* **unfolding** *transp-def*

by (*intro successively-iff-sorted-wrt-strong*) *blast*

lemma *successively-rev* [*simp*]: *successively P (rev xs) \longleftrightarrow successively ($\lambda x y. P y x$) xs*

by (*induction xs rule: remdups-adj.induct*)
(*auto simp: successively-append-iff successively-Cons*)

lemma *successively-map*: *successively P (map f xs) \longleftrightarrow successively ($\lambda x y. P (f x) (f y)$) xs*

by (*induction xs rule: induct-list012*) *auto*

lemma *successively-mono*:

assumes *successively P xs*
assumes $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies P x y \implies Q x y$
shows *successively Q xs*
using *assms* **by** (*induction Q xs rule: successively.induct*) *auto*

lemma *successively-altdef*:

successively = ($\lambda P. \text{rec-list True } (\lambda x xs b. \text{case } xs \text{ of } [] \Rightarrow \text{True} \mid y \# - \Rightarrow P x y \wedge b)$)

proof (*intro ext*)

fix *P and xs :: 'a list*

show *successively P xs = rec-list True ($\lambda x xs b. \text{case } xs \text{ of } [] \Rightarrow \text{True} \mid y \# - \Rightarrow P x y \wedge b$) xs*

by (*induction xs*) (*auto simp: successively-Cons split: list.splits*)

qed

67.1.24 *distinct and remdups and remdups-adj*

lemma *distinct-tl*: *distinct xs \implies distinct (tl xs)*

by (*cases xs*) *simp-all*

lemma *distinct-append* [*simp*]:

distinct (xs @ ys) = (distinct xs \wedge distinct ys \wedge set xs \cap set ys = {})

by (*induct xs*) *auto*

lemma *distinct-rev*[*simp*]: *distinct(rev xs) = distinct xs*

by(*induct xs*) *auto*

lemma *set-remdups* [*simp*]: *set (remdups xs) = set xs*

by (*induct xs*) (*auto simp add: insert-absorb*)

lemma *distinct-remdups* [*iff*]: *distinct (remdups xs)*

by (*induct xs*) *auto*

lemma *distinct-remdups-id*: *distinct xs \implies remdups xs = xs*

by (*induct xs, auto*)

lemma *remdups-id-iff-distinct* [*simp*]: *remdups xs = xs \longleftrightarrow distinct xs*

by (metis distinct-remdups distinct-remdups-id)

lemma *finite-distinct-list*: $\text{finite } A \implies \exists xs. \text{set } xs = A \wedge \text{distinct } xs$
 by (metis distinct-remdups finite-list set-remdups)

lemma *remdups-eq-nil-iff* [simp]: $(\text{remdups } x = []) = (x = [])$
 by (induct x, auto)

lemmas *remdups-eq-nil-right-iff* [simp] = *remdups-eq-nil-iff*[THEN *eq-iff-swap*]

lemma *length-remdups-leq*[iff]: $\text{length}(\text{remdups } xs) \leq \text{length } xs$
 by (induct xs) auto

lemma *length-remdups-eq*[iff]:
 $(\text{length } (\text{remdups } xs) = \text{length } xs) = (\text{remdups } xs = xs)$
proof (induct xs)
 case (Cons a xs)
 then show ?case
 by simp (metis Suc-n-not-le-n impossible-Cons length-remdups-leq)
qed auto

lemma *remdups-filter*: $\text{remdups}(\text{filter } P \text{ } xs) = \text{filter } P (\text{remdups } xs)$
 by (induct xs) auto

lemma *distinct-map*:
 $\text{distinct}(\text{map } f \text{ } xs) = (\text{distinct } xs \wedge \text{inj-on } f (\text{set } xs))$
 by (induct xs) auto

lemma *distinct-map-filter*:
 $\text{distinct } (\text{map } f \text{ } xs) \implies \text{distinct } (\text{map } f (\text{filter } P \text{ } xs))$
 by (induct xs) auto

lemma *distinct-filter* [simp]: $\text{distinct } xs \implies \text{distinct } (\text{filter } P \text{ } xs)$
 by (induct xs) auto

lemma *distinct-upt*[simp]: $\text{distinct}[i..<j]$
 by (induct j) auto

lemma *distinct-upto*[simp]: $\text{distinct}[i..j]$
proof (induction i j rule: upto.induct)
 case (1 i j)
 then show ?case
 by (simp add: upto.simps [of i])
qed

lemma *distinct-take*[simp]: $\text{distinct } xs \implies \text{distinct } (\text{take } i \text{ } xs)$
proof (induct xs arbitrary: i)
 case (Cons a xs)
 then show ?case

by (metis Cons.premis append-take-drop-id distinct-append)
qed auto

lemma *distinct-drop[simp]*: $distinct\ xs \implies distinct\ (drop\ i\ xs)$
proof (induct xs arbitrary: i)
case (Cons a xs)
then show ?case
by (metis Cons.premis append-take-drop-id distinct-append)
qed auto

lemma *distinct-list-update*:
assumes d : $distinct\ xs$ and a : $a \notin set\ xs - \{xs!i\}$
shows $distinct\ (xs[i:=a])$
proof (cases $i < length\ xs$)
case True
with a have anot: $a \notin set\ (take\ i\ xs @ xs ! i \# drop\ (Suc\ i)\ xs) - \{xs!i\}$
by simp (metis in-set-dropD in-set-takeD)
show ?thesis
proof (cases $a = xs!i$)
case True
with d show ?thesis
by auto
next
case False
have $set\ (take\ i\ xs) \cap set\ (drop\ (Suc\ i)\ xs) = \{\}$
by (metis True d disjoint-insert(1) distinct-append id-take-nth-drop list.set(2))
then show ?thesis
using d False anot $\langle i < length\ xs \rangle$ by (simp add: upd-conv-take-nth-drop)
qed
next
case False with d show ?thesis by auto
qed

lemma *distinct-concat*:
[[$distinct\ xs$;
 $\bigwedge ys. ys \in set\ xs \implies distinct\ ys$;
 $\bigwedge ys\ zs. \llbracket ys \in set\ xs ; zs \in set\ xs ; ys \neq zs \rrbracket \implies set\ ys \cap set\ zs = \{\}$
]] $\implies distinct\ (concat\ xs)$
by (induct xs) auto

An iff-version of $\llbracket distinct\ ?xs ; \bigwedge ys. ys \in set\ ?xs \implies distinct\ ys ; \bigwedge ys\ zs. \llbracket ys \in set\ ?xs ; zs \in set\ ?xs ; ys \neq zs \rrbracket \implies set\ ys \cap set\ zs = \{\} \rrbracket \implies distinct\ (concat\ ?xs)$ is available further down as *distinct-concat-iff*.

It is best to avoid the following indexed version of distinct, but sometimes it is useful.

lemma *distinct-conv-nth*: $distinct\ xs = (\forall i < size\ xs. \forall j < size\ xs. i \neq j \longrightarrow xs!i \neq xs!j)$
proof (induct xs)

```

case (Cons x xs)
show ?case
  apply (auto simp add: Cons nth-Cons less-Suc-eq-le split: nat.split-asm)
  apply (metis Suc-leI in-set-conv-nth length-pos-if-in-set lessI less-imp-le-nat
less-nat-zero-code)
  apply (metis Suc-le-eq)
  done
qed auto

```

lemma *nth-eq-iff-index-eq*:
 $\llbracket \text{distinct } xs; i < \text{length } xs; j < \text{length } xs \rrbracket \implies (xs!i = xs!j) = (i = j)$
by(auto simp: distinct-conv-nth)

lemma *distinct-Ex1*:
 $\text{distinct } xs \implies x \in \text{set } xs \implies (\exists !i. i < \text{length } xs \wedge xs ! i = x)$
by (auto simp: in-set-conv-nth nth-eq-iff-index-eq)

lemma *inj-on-nth*: $\text{distinct } xs \implies \forall i \in I. i < \text{length } xs \implies \text{inj-on } (nth \ xs) \ I$
by (rule inj-onI) (simp add: nth-eq-iff-index-eq)

lemma *bij-betw-nth*:
assumes $\text{distinct } xs \ A = \{..<\text{length } xs\} \ B = \text{set } xs$
shows $\text{bij-betw } (!) \ xs \ A \ B$
using *assms* **unfolding** *bij-betw-def*
by (auto intro!: inj-on-nth simp: set-conv-nth)

lemma *set-update-distinct*: $\llbracket \text{distinct } xs; n < \text{length } xs \rrbracket \implies$
 $\text{set}(xs[n := x]) = \text{insert } x \ (\text{set } xs - \{xs!n\})$
by(auto simp: set-eq-iff in-set-conv-nth nth-list-update nth-eq-iff-index-eq)

lemma *distinct-swap[simp]*: $\llbracket i < \text{size } xs; j < \text{size } xs \rrbracket \implies$
 $\text{distinct}(xs[i := xs!j, j := xs!i]) = \text{distinct } xs$
apply (simp add: distinct-conv-nth nth-list-update)
apply (safe; metis)
done

lemma *set-swap[simp]*:
 $\llbracket i < \text{size } xs; j < \text{size } xs \rrbracket \implies \text{set}(xs[i := xs!j, j := xs!i]) = \text{set } xs$
by(simp add: set-conv-nth nth-list-update) metis

lemma *distinct-card*: $\text{distinct } xs \implies \text{card } (\text{set } xs) = \text{size } xs$
by (induct xs) auto

lemma *card-distinct*: $\text{card } (\text{set } xs) = \text{size } xs \implies \text{distinct } xs$
proof (induct xs)
case (Cons x xs)
show ?case
proof (cases $x \in \text{set } xs$)
case False **with** Cons **show** ?thesis **by** simp

```

next
  case True with Cons.prem
  have card (set xs) = Suc (length xs)
    by (simp add: card-insert-if split: if-split-asm)
  moreover have card (set xs) ≤ length xs by (rule card-length)
  ultimately have False by simp
  thus ?thesis ..
qed
qed simp

lemma distinct-length-filter: distinct xs ⇒ length (filter P xs) = card ({x. P x}
Int set xs)
by (induct xs) (auto)

lemma not-distinct-decomp: ¬ distinct ws ⇒ ∃ xs ys zs y. ws = xs@[y]@ys@[y]@zs
proof (induct n == length ws arbitrary: ws)
  case (Suc n ws)
  then show ?case
    using length-Suc-conv [of ws n]
    apply (auto simp: eq-commute)
    apply (metis append-Nil in-set-conv-decomp-first)
    by (metis append-Cons)
qed simp

lemma not-distinct-conv-prefix:
  defines dec as xs y ys ≡ y ∈ set xs ∧ distinct xs ∧ as = xs @ y # ys
  shows ¬distinct as ⇔ (∃ xs y ys. dec as xs y ys) (is ?L = ?R)
proof
  assume ?L then show ?R
  proof (induct length as arbitrary: as rule: less-induct)
    case less
    obtain xs ys zs y where decomp: as = (xs @ y # ys) @ y # zs
      using not-distinct-decomp[OF less.prem] by auto
    show ?case
    proof (cases distinct (xs @ y # ys))
      case True
      with decomp have dec as (xs @ y # ys) y zs by (simp add: dec-def)
      then show ?thesis by blast
    next
      case False
      with less decomp obtain xs' y' ys' where dec (xs @ y # ys) xs' y' ys'
        by atomize-elim auto
      with decomp have dec as xs' y' (ys' @ y # zs) by (simp add: dec-def)
      then show ?thesis by blast
    qed
  qed
qed (auto simp: dec-def)

lemma distinct-product:

```

$distinct\ xs \implies distinct\ ys \implies distinct\ (List.product\ xs\ ys)$
by $(induct\ xs)$ $(auto\ intro:\ inj-onI\ simp\ add:\ distinct-map)$

lemma *distinct-product-lists*:

assumes $\forall\ xs \in set\ xss.\ distinct\ xs$
shows $distinct\ (product-lists\ xss)$
using *assms* **proof** $(induction\ xss)$
case $(Cons\ xs\ xss)$ **note** $* = this$
then show *?case*
proof $(cases\ product-lists\ xss)$
case *Nil* **then show** *?thesis* **by** $(induct\ xs)\ simp-all$
next
case $(Cons\ ps\ pss)$ **with** $*$ **show** *?thesis*
by $(auto\ intro!:\ inj-onI\ distinct-concat\ simp\ add:\ distinct-map)$
qed
qed *simp*

lemma *length-remdups-concat*:

$length\ (remdups\ (concat\ xss)) = card\ (\bigcup\ xs \in set\ xss.\ set\ xs)$
by $(simp\ add:\ distinct-card\ [symmetric])$

lemma *remdups-append2*:

$remdups\ (xs\ @\ remdups\ ys) = remdups\ (xs\ @\ ys)$
by $(induction\ xs)\ auto$

lemma *length-remdups-card-conv*: $length(remdups\ xs) = card(set\ xs)$

proof –

have $xs:\ concat[xs] = xs$ **by** *simp*
from *length-remdups-concat* $[of\ [xs]]$ **show** *?thesis* **unfolding** xs **by** *simp*
qed

lemma *remdups-remdups*: $remdups\ (remdups\ xs) = remdups\ xs$

by $(induct\ xs)\ simp-all$

lemma *distinct-butlast*:

assumes $distinct\ xs$
shows $distinct\ (butlast\ xs)$
proof $(cases\ xs = [])$
case *False*
from $\langle xs \neq [] \rangle$ **obtain** $ys\ y$ **where** $xs = ys\ @\ [y]$ **by** $(cases\ xs\ rule:\ rev-cases)$
auto
with $\langle distinct\ xs \rangle$ **show** *?thesis* **by** *simp*
qed $(auto)$

lemma *remdups-map-remdups*:

$remdups\ (map\ f\ (remdups\ xs)) = remdups\ (map\ f\ xs)$
by $(induct\ xs)\ simp-all$

lemma *distinct-zipI1*:


```

assumes distinct xs
shows distinct (zip xs ys)
proof (rule zip-obtain-same-length)
  fix xs' :: 'a list and ys' :: 'b list and n
  assume length xs' = length ys'
  assume xs' = take n xs
  with assms have distinct xs' by simp
  with  $\langle \text{length } xs' = \text{length } ys' \rangle$  show distinct (zip xs' ys')
    by (induct xs' ys' rule: list-induct2) (auto elim: in-set-zipE)
qed

```

```

lemma distinct-zipI2:
  assumes distinct ys
  shows distinct (zip xs ys)
proof (rule zip-obtain-same-length)
  fix xs' :: 'b list and ys' :: 'a list and n
  assume length xs' = length ys'
  assume ys' = take n ys
  with assms have distinct ys' by simp
  with  $\langle \text{length } xs' = \text{length } ys' \rangle$  show distinct (zip xs' ys')
    by (induct xs' ys' rule: list-induct2) (auto elim: in-set-zipE)
qed

```

```

lemma set-take-disj-set-drop-if-distinct:
  distinct vs  $\implies i \leq j \implies \text{set (take } i \text{ vs)} \cap \text{set (drop } j \text{ vs)} = \{\}$ 
by (auto simp: in-set-conv-nth distinct-conv-nth)

```

```

lemma distinct-singleton: distinct [x] by simp

```

```

lemma distinct-length-2-or-more:
  distinct (a # b # xs)  $\longleftrightarrow (a \neq b \wedge \text{distinct (a # xs)} \wedge \text{distinct (b # xs)})$ 
by force

```

```

lemma remdups-adj-altdef: (remdups-adj xs = ys)  $\longleftrightarrow$ 
  ( $\exists f :: \text{nat} \implies \text{nat. mono } f \wedge f \text{ ` } \{0 \dots \text{size } xs\} = \{0 \dots \text{size } ys\}$ 
   $\wedge (\forall i < \text{size } xs. xs!i = ys!(f i))$ 
   $\wedge (\forall i. i + 1 < \text{size } xs \longrightarrow (xs!i = xs!(i+1) \longleftrightarrow f i = f(i+1)))$ ) (is ?L  $\longleftrightarrow$ 
  ( $\exists f. ?p f xs ys$ ))
proof
  assume ?L
  then show  $\exists f. ?p f xs ys$ 
  proof (induct xs arbitrary: ys rule: remdups-adj.induct)
    case (1 ys)
    thus ?case by (intro exI[of - id]) (auto simp: mono-def)
  next
    case (2 x ys)
    thus ?case by (intro exI[of - id]) (auto simp: mono-def)

```

```

next
  case ( $\exists$   $x1$   $x2$   $xs$   $ys$ )
  let  $?xs = x1 \# x2 \# xs$ 
  let  $?cond = x1 = x2$ 
  define  $zs$  where  $zs = remdups-adj (x2 \# xs)$ 
  from  $\exists(1-2)[of\ zs]$ 
  obtain  $f$  where  $p: ?p\ f\ (x2 \# xs)\ zs$  unfolding  $zs-def$  by (cases  $?cond$ ) auto
  then have  $f0: f\ 0 = 0$ 
    by (intro mono-image-least[where  $f=f$ ]) blast+
    from  $p$  have mono: mono  $f$  and  $f-xs-zs: f\ ' \{0..<length\ (x2 \# xs)\} =$ 
     $\{0..<length\ zs\}$  by auto
  have  $ys: ys = (if\ x1 = x2\ then\ zs\ else\ x1 \# zs)$ 
    unfolding  $\exists(\exists)[symmetric]\ zs-def$  by auto
  have  $zs0: zs\ !\ 0 = x2$  unfolding  $zs-def$  by (induct  $xs$ ) auto
  have  $zsne: zs \neq []$  unfolding  $zs-def$  by (induct  $xs$ ) auto
  let  $?Succ = if\ ?cond\ then\ id\ else\ Suc$ 
  let  $?x1 = if\ ?cond\ then\ id\ else\ Cons\ x1$ 
  let  $?f = \lambda\ i.\ if\ i = 0\ then\ 0\ else\ ?Succ\ (f\ (i - 1))$ 
  have  $ys: ys = ?x1\ zs$  unfolding  $ys$  by (cases  $?cond$ , auto)
  have mono: mono  $?f$  using  $\langle mono\ f \rangle$  unfolding mono-def by auto
  show  $?case$  unfolding  $ys$ 
  proof (intro exI[of -  $?f$ ] conjI allI impI)
    show mono  $?f$  by fact
  next
    fix  $i$  assume  $i: i < length\ ?xs$ 
    with  $p$  show  $?xs\ !\ i = ?x1\ zs\ !\ (?f\ i)$  using  $zs0$  by auto
  next
    fix  $i$  assume  $i: i + 1 < length\ ?xs$ 
    with  $p$  show  $(?xs\ !\ i = ?xs\ !\ (i + 1)) = (?f\ i = ?f\ (i + 1))$ 
      by (cases  $i$ ) (auto simp:  $f0$ )
  next
    have  $id: \{0\ ..< length\ (?x1\ zs)\} = insert\ 0\ (?Succ\ ' \{0\ ..< length\ zs\})$ 
      using  $zsne$  by (cases  $?cond$ , auto)
    { fix  $i$  assume  $i < Suc\ (length\ xs)$ 
      hence  $Suc\ i \in \{0..<Suc\ (Suc\ (length\ xs))\} \cap Collect\ ((<) 0)$  by auto
      from imageI[OF this, of  $\lambda i.\ ?Succ\ (f\ (i - Suc\ 0))$ ]
      have  $?Succ\ (f\ i) \in (\lambda i.\ ?Succ\ (f\ (i - Suc\ 0)))\ ' (\{0..<Suc\ (Suc\ (length\ xs))\} \cap Collect\ ((<) 0))$  by auto
    }
    then show  $?f\ ' \{0\ ..< length\ ?xs\} = \{0\ ..< length\ (?x1\ zs)\}$ 
      unfolding  $id\ f-xs-zs[symmetric]$  by auto
  qed
qed
next
  assume  $\exists\ f.\ ?p\ f\ xs\ ys$ 
  then show  $?L$ 
  proof (induct  $xs$  arbitrary:  $ys$  rule: remdups-adj.induct)
    case 1 then show  $?case$  by auto
  next

```

```

case (2 x) then obtain f where f-img:  $f \text{ ‘ } \{0 \dots \text{size } [x]\} = \{0 \dots \text{size } ys\}$ 
and f-nth:  $\bigwedge i. i < \text{size } [x] \implies [x]!i = ys!(f\ i)$ 
by blast

have  $\text{length } ys = \text{card } (f \text{ ‘ } \{0 \dots \text{size } [x]\})$ 
using f-img by auto
then have  $*: \text{length } ys = 1$  by auto
then have  $f\ 0 = 0$  using f-img by auto
with  $*$  show ?case using f-nth by (cases ys) auto
next
case (3 x1 x2 xs)
from 3.prems obtain f where f-mono: mono f
and f-img:  $f \text{ ‘ } \{0 \dots \text{length } (x1 \# x2 \# xs)\} = \{0 \dots \text{length } ys\}$ 
and f-nth:
 $\bigwedge i. i < \text{length } (x1 \# x2 \# xs) \implies (x1 \# x2 \# xs) ! i = ys ! f\ i$ 
 $\bigwedge i. i + 1 < \text{length } (x1 \# x2 \# xs) \implies$ 
 $((x1 \# x2 \# xs) ! i = (x1 \# x2 \# xs) ! (i + 1)) = (f\ i = f\ (i + 1))$ 
by blast

show ?case
proof cases
assume  $x1 = x2$ 

let  $?f' = f \circ \text{Suc}$ 

have remdups-adj  $(x1 \# xs) = ys$ 
proof (intro 3.hyps exI conjI impI allI)
show mono  $?f'$ 
using f-mono by (simp add: mono-iff-le-Suc)
next
have  $?f' \text{ ‘ } \{0 \dots \text{length } (x1 \# xs)\} = f \text{ ‘ } \{\text{Suc } 0 \dots \text{length } (x1 \# x2 \#$ 
xs)\}
using less-Suc-eq-0-disj by auto
also have  $\dots = f \text{ ‘ } \{0 \dots \text{length } (x1 \# x2 \# xs)\}$ 
proof –
have  $f\ 0 = f\ (\text{Suc } 0)$  using  $\langle x1 = x2 \rangle$  f-nth[of 0] by simp
then show ?thesis
using less-Suc-eq-0-disj by auto
qed
also have  $\dots = \{0 \dots \text{length } ys\}$  by fact
finally show  $?f' \text{ ‘ } \{0 \dots \text{length } (x1 \# xs)\} = \{0 \dots \text{length } ys\}$  .
qed (insert f-nth[of Suc i for i], auto simp:  $\langle x1 = x2 \rangle$ )
then show ?thesis using  $\langle x1 = x2 \rangle$  by simp
next
assume  $x1 \neq x2$ 

have two:  $\text{Suc } (\text{Suc } 0) \leq \text{length } ys$ 
proof –
have  $2 = \text{card } \{f\ 0, f\ 1\}$  using  $\langle x1 \neq x2 \rangle$  f-nth[of 0] by auto

```

```

    also have ... ≤ card (f ‘ {0..< length (x1 # x2 # xs)})
      by (rule card-mono) auto
    finally show ?thesis using f-img by simp
  qed

  have f 0 = 0 using f-mono f-img by (rule mono-image-least) simp

  have f (Suc 0) = Suc 0
  proof (rule ccontr)
    assume f (Suc 0) ≠ Suc 0
    then have Suc 0 < f (Suc 0) using f-nth[of 0] ⟨x1 ≠ x2⟩ ⟨f 0 = 0⟩ by
auto
    then have ∧i. Suc 0 < f (Suc i)
      using f-mono
      by (meson Suc-le-mono le0 less-le-trans monoD)
    then have Suc 0 ≠ f i for i using ⟨f 0 = 0⟩
      by (cases i) fastforce+
    then have Suc 0 ∉ f ‘ {0 ..< length (x1 # x2 # xs)} by auto
    then show False using f-img two by auto
  qed
  obtain ys' where ys = x1 # x2 # ys'
    using two f-nth[of 0] f-nth[of 1]
    by (auto simp: Suc-le-length-iff) ⟨f 0 = 0⟩ ⟨f (Suc 0) = Suc 0⟩

  have Suc0-le-f-Suc: Suc 0 ≤ f (Suc i) for i
    by (metis Suc-le-mono ⟨f (Suc 0) = Suc 0⟩ f-mono le0 mono-def)

  define f' where f' x = f (Suc x) - 1 for x
  have f-Suc: f (Suc i) = Suc (f' i) for i
    using Suc0-le-f-Suc[of i] by (auto simp: f'-def)

  have remdups-adj (x2 # xs) = (x2 # ys')
  proof (intro 3.hyps exI conjI impI allI)
    show mono f'
      using Suc0-le-f-Suc f-mono by (auto simp: f'-def mono-iff-le-Suc le-diff-iff)
  next
    have f' ‘ {0 ..< length (x2 # xs)} = (λx. f x - 1) ‘ {0 ..< length (x1 #
x2 # xs)}
      by (auto simp: f'-def ⟨f 0 = 0⟩ ⟨f (Suc 0) = Suc 0⟩ image-def Bex-def
less-Suc-eq-0-disj)
    also have ... = (λx. x - 1) ‘ f ‘ {0 ..< length (x1 # x2 # xs)}
      by (auto simp: image-comp)
    also have ... = (λx. x - 1) ‘ {0 ..< length ys}
      by (simp only: f-img)
    also have ... = {0 ..< length (x2 # ys')}
      using ⟨ys = ->⟩ by (fastforce intro: rev-image-eqI)
    finally show f' ‘ {0 ..< length (x2 # xs)} = {0 ..< length (x2 # ys')} .
  qed (insert f-nth[of Suc i for i] ⟨x1 ≠ x2⟩, auto simp add: f-Suc ⟨ys = ->⟩)
  then show ?case using ⟨ys = ->⟩ ⟨x1 ≠ x2⟩ by simp

```

qed
 qed
 qed

lemma *hd-remdups-adj[simp]*: $hd (remdups-adj\ xs) = hd\ xs$
 by (induction xs rule: remdups-adj.induct) simp-all

lemma *remdups-adj-Cons*: $remdups-adj\ (x\ \#\ xs) =$
 (case remdups-adj xs of [] \Rightarrow [x] | y # xs \Rightarrow if x = y then y # xs else x # y #
 xs)
 by (induct xs arbitrary: x) (auto split: list.splits)

lemma *remdups-adj-append-two*:
 $remdups-adj\ (xs\ @\ [x,y]) = remdups-adj\ (xs\ @\ [x])\ @\ (if\ x = y\ then\ []\ else\ [y])$
 by (induct xs rule: remdups-adj.induct, simp-all)

lemma *remdups-adj-adjacent*:
 $Suc\ i < length\ (remdups-adj\ xs) \Longrightarrow remdups-adj\ xs\ !\ i \neq remdups-adj\ xs\ !\ Suc\ i$
proof (induction xs arbitrary: i rule: remdups-adj.induct)
 case ($\exists\ x\ y\ xs\ i$)
 thus ?case by (cases i, cases x = y) (simp, auto simp: hd-conv-nth[symmetric])
 qed simp-all

lemma *remdups-adj-rev[simp]*: $remdups-adj\ (rev\ xs) = rev\ (remdups-adj\ xs)$
 by (induct xs rule: remdups-adj.induct, simp-all add: remdups-adj-append-two)

lemma *remdups-adj-length[simp]*: $length\ (remdups-adj\ xs) \leq length\ xs$
 by (induct xs rule: remdups-adj.induct, auto)

lemma *remdups-adj-length-ge1[simp]*: $xs \neq [] \Longrightarrow length\ (remdups-adj\ xs) \geq Suc\ 0$
 by (induct xs rule: remdups-adj.induct, simp-all)

lemma *remdups-adj-Nil-iff[simp]*: $remdups-adj\ xs = [] \longleftrightarrow xs = []$
 by (induct xs rule: remdups-adj.induct, simp-all)

lemma *remdups-adj-set[simp]*: $set\ (remdups-adj\ xs) = set\ xs$
 by (induct xs rule: remdups-adj.induct, simp-all)

lemma *last-remdups-adj [simp]*: $last\ (remdups-adj\ xs) = last\ xs$
 by (induction xs rule: remdups-adj.induct) auto

lemma *remdups-adj-Cons-alt[simp]*: $x\ \#\ tl\ (remdups-adj\ (x\ \#\ xs)) = remdups-adj\ (x\ \#\ xs)$
 by (induct xs rule: remdups-adj.induct, auto)

lemma *remdups-adj-distinct*: $distinct\ xs \Longrightarrow remdups-adj\ xs = xs$
 by (induct xs rule: remdups-adj.induct, simp-all)

lemma *remdups-adj-append*:

$remdups-adj (xs_1 @ x \# xs_2) = remdups-adj (xs_1 @ [x]) @ tl (remdups-adj (x \# xs_2))$

by (*induct* xs_1 *rule*: *remdups-adj.induct*, *simp-all*)

lemma *remdups-adj-singleton*:

$remdups-adj xs = [x] \implies xs = replicate (length xs) x$

by (*induct* xs *rule*: *remdups-adj.induct*, *auto split*: *if-split-asm*)

lemma *remdups-adj-map-injective*:

assumes *inj* f

shows $remdups-adj (map f xs) = map f (remdups-adj xs)$

by (*induct* xs *rule*: *remdups-adj.induct*) (*auto simp add*: *injD[OF assms]*)

lemma *remdups-adj-replicate*:

$remdups-adj (replicate n x) = (if n = 0 then [] else [x])$

by (*induction* n) (*auto simp*: *remdups-adj.Cons*)

lemma *remdups-upt [simp]*: $remdups [m..<n] = [m..<n]$

proof (*cases* $m \leq n$)

case *False* **then show** *?thesis* **by** *simp*

next

case *True* **then obtain** q **where** $n = m + q$

by (*auto simp add*: *le-iff-add*)

moreover have $remdups [m..<m + q] = [m..<m + q]$

by (*induct* q) *simp-all*

ultimately show *?thesis* **by** *simp*

qed

lemma *successively-remdups-adjI*:

$successively P xs \implies successively P (remdups-adj xs)$

by (*induction* xs *rule*: *remdups-adj.induct*) (*auto simp*: *successively-Cons*)

lemma *successively-remdups-adj-iff*:

$(\bigwedge x. x \in set xs \implies P x x) \implies$

$successively P (remdups-adj xs) \longleftrightarrow successively P xs$

by (*induction* xs *rule*: *remdups-adj.induct*)(*auto simp*: *successively-Cons*)

lemma *remdups-adj-Cons'*:

$remdups-adj (x \# xs) = x \# remdups-adj (dropWhile (\lambda y. y = x) xs)$

by (*induction* xs) *auto*

lemma *remdups-adj-singleton-iff*:

$length (remdups-adj xs) = Suc 0 \longleftrightarrow xs \neq [] \wedge xs = replicate (length xs) (hd xs)$

proof *safe*

assume $*$: $xs = replicate (length xs) (hd xs)$ **and** *[simp]*: $xs \neq []$

show $length (remdups-adj xs) = Suc 0$

by (*subst* $*$) (*auto simp*: *remdups-adj-replicate*)

next

assume $\text{length } (\text{remdups-adj } xs) = \text{Suc } 0$
thus $xs = \text{replicate } (\text{length } xs) (\text{hd } xs)$
by (*induction xs rule: remdups-adj.induct*) (*auto split: if-splits*)
qed *auto*

lemma *tl-remdups-adj*:
 $ys \neq [] \implies \text{tl } (\text{remdups-adj } ys) = \text{remdups-adj } (\text{dropWhile } (\lambda x. x = \text{hd } ys) (\text{tl } ys))$
by (*cases ys*) (*simp-all add: remdups-adj-Cons'*)

lemma *remdups-adj-append-dropWhile*:
 $\text{remdups-adj } (xs @ y \# ys) = \text{remdups-adj } (xs @ [y]) @ \text{remdups-adj } (\text{dropWhile } (\lambda x. x = y) ys)$
by (*subst remdups-adj-append*) (*simp add: tl-remdups-adj*)

lemma *remdups-adj-append'*:
assumes $xs = [] \vee ys = [] \vee \text{last } xs \neq \text{hd } ys$
shows $\text{remdups-adj } (xs @ ys) = \text{remdups-adj } xs @ \text{remdups-adj } ys$
proof –
have *?thesis* **if** [*simp*]: $xs \neq [] \text{ } ys \neq []$ **and** $\text{last } xs \neq \text{hd } ys$
proof –
obtain $x \text{ } xs'$ **where** $xs = xs' @ [x]$
by (*cases xs rule: rev-cases*) *auto*
have $\text{remdups-adj } (xs' @ x \# ys) = \text{remdups-adj } (xs' @ [x]) @ \text{remdups-adj } ys$
using $\langle \text{last } xs \neq \text{hd } ys \rangle$ **unfolding** xs
by (*metis (full-types) dropWhile-eq-self-iff last-snoc remdups-adj-append-dropWhile*)
thus *?thesis* **by** (*simp add: xs*)
qed
thus *?thesis* **using** *assms*
by (*cases xs = []*; *cases ys = []*) *auto*
qed

lemma *remdups-adj-append''*: $xs \neq []$
 $\implies \text{remdups-adj } (xs @ ys) = \text{remdups-adj } xs @ \text{remdups-adj } (\text{dropWhile } (\lambda y. y = \text{last } xs) ys)$
by (*induction xs rule: remdups-adj.induct*) (*auto simp: remdups-adj-Cons'*)

67.2 distinct-adj

lemma *distinct-adj-Nil* [*simp*]: *distinct-adj* $[]$
and *distinct-adj-singleton* [*simp*]: *distinct-adj* $[x]$
and *distinct-adj-Cons-Cons* [*simp*]: *distinct-adj* $(x \# y \# xs) \longleftrightarrow x \neq y \wedge \text{distinct-adj } (y \# xs)$
by (*auto simp: distinct-adj-def*)

lemma *distinct-adj-Cons*: *distinct-adj* $(x \# xs) \longleftrightarrow xs = [] \vee x \neq \text{hd } xs \wedge \text{distinct-adj } xs$
by (*cases xs*) *auto*

lemma *distinct-adj-ConsD*: $\text{distinct-adj } (x \# xs) \implies \text{distinct-adj } xs$
by (*cases xs*) *auto*

lemma *distinct-adj-remdups-adj[simp]*: $\text{distinct-adj } (\text{remdups-adj } xs)$
by (*induction xs rule: remdups-adj.induct*) (*auto simp: distinct-adj-Cons*)

lemma *distinct-adj-altdef*: $\text{distinct-adj } xs \longleftrightarrow \text{remdups-adj } xs = xs$

proof

assume $\text{remdups-adj } xs = xs$

with *distinct-adj-remdups-adj[of xs]* **show** $\text{distinct-adj } xs$

by *simp*

next

assume $\text{distinct-adj } xs$

thus $\text{remdups-adj } xs = xs$

by (*induction xs rule: induct-list012*) *auto*

qed

lemma *distinct-adj-rev [simp]*: $\text{distinct-adj } (\text{rev } xs) \longleftrightarrow \text{distinct-adj } xs$
by (*simp add: distinct-adj-def eq-commute*)

lemma *distinct-adj-append-iff*:

$\text{distinct-adj } (xs @ ys) \longleftrightarrow$

$\text{distinct-adj } xs \wedge \text{distinct-adj } ys \wedge (xs = [] \vee ys = [] \vee \text{last } xs \neq \text{hd } ys)$

by (*auto simp: distinct-adj-def successively-append-iff*)

lemma *distinct-adj-appendD1 [dest]*: $\text{distinct-adj } (xs @ ys) \implies \text{distinct-adj } xs$
and *distinct-adj-appendD2 [dest]*: $\text{distinct-adj } (xs @ ys) \implies \text{distinct-adj } ys$
by (*auto simp: distinct-adj-append-iff*)

lemma *distinct-adj-mapI*: $\text{distinct-adj } xs \implies \text{inj-on } f (\text{set } xs) \implies \text{distinct-adj } (\text{map } f \ xs)$

unfolding *distinct-adj-def successively-map*

by (*erule successively-mono*) (*auto simp: inj-on-def*)

lemma *distinct-adj-mapD*: $\text{distinct-adj } (\text{map } f \ xs) \implies \text{distinct-adj } xs$
unfolding *distinct-adj-def successively-map* **by** (*erule successively-mono*) *auto*

lemma *distinct-adj-map-iff*: $\text{inj-on } f (\text{set } xs) \implies \text{distinct-adj } (\text{map } f \ xs) \longleftrightarrow \text{distinct-adj } xs$

using *distinct-adj-mapD distinct-adj-mapI* **by** *blast*

67.2.1 *insert*

lemma *in-set-insert [simp]*:

$x \in \text{set } xs \implies \text{List.insert } x \ xs = xs$

by (*simp add: List.insert-def*)

lemma *not-in-set-insert [simp]*:

$x \notin \text{set } xs \implies \text{List.insert } x \ xs = x \# \ xs$

by (simp add: List.insert-def)

lemma insert-Nil [simp]: List.insert x [] = [x]
by simp

lemma set-insert [simp]: set (List.insert x xs) = insert x (set xs)
by (auto simp add: List.insert-def)

lemma distinct-insert [simp]: distinct (List.insert x xs) = distinct xs
by (simp add: List.insert-def)

lemma insert-remdups:
List.insert x (remdups xs) = remdups (List.insert x xs)
by (simp add: List.insert-def)

67.2.2 List.union

This is all one should need to know about union:

lemma set-union[simp]: set (List.union xs ys) = set xs \cup set ys
unfolding List.union-def
by(induct xs arbitrary: ys) simp-all

lemma distinct-union[simp]: distinct(List.union xs ys) = distinct ys
unfolding List.union-def
by(induct xs arbitrary: ys) simp-all

67.2.3 find

lemma find-None-iff: List.find P xs = None \longleftrightarrow \neg ($\exists x. x \in$ set xs \wedge P x)
proof (induction xs)
 case Nil **thus** ?case by simp
next
 case (Cons x xs) **thus** ?case by (fastforce split: if-splits)
qed

lemmas find-None-iff2 = find-None-iff[THEN eq-iff-swap]

lemma find-Some-iff:
List.find P xs = Some x \longleftrightarrow
($\exists i < \text{length } xs. P (xs!i) \wedge x = xs!i \wedge (\forall j < i. \neg P (xs!j))$)
proof (induction xs)
 case Nil **thus** ?case by simp
next
 case (Cons x xs) **thus** ?case
 apply(auto simp: nth-Cons' split: if-splits)
 using diff-Suc-1[unfolded One-nat-def] less-Suc-eq-0-disj **by** fastforce
qed

lemmas find-Some-iff2 = find-Some-iff[THEN eq-iff-swap]

lemma *find-cong*[*fundef-cong*]:
assumes $xs = ys$ **and** $\bigwedge x. x \in \text{set } ys \implies P x = Q x$
shows $List.find P xs = List.find Q ys$
proof (*cases List.find P xs*)
case *None* **thus** *?thesis* **by** (*metis find-None-iff assms*)
next
case (*Some x*)
hence $List.find Q ys = \text{Some } x$ **using** *assms*
by (*auto simp add: find-Some-iff*)
thus *?thesis* **using** *Some* **by** *auto*
qed

lemma *find-dropWhile*:
 $List.find P xs = (\text{case } \text{dropWhile } (Not \circ P) xs$
of $[] \Rightarrow None$
 $| x \# - \Rightarrow \text{Some } x)$
by (*induct xs*) *simp-all*

67.2.4 count-list

This library is intentionally minimal. See the remark about multisets at the point above where *count-list* is defined.

lemma *count-list-append*[*simp*]: $\text{count-list } (xs @ ys) x = \text{count-list } xs x + \text{count-list } ys x$
by (*induction xs*) *simp-all*

lemma *count-list-0-iff*: $\text{count-list } xs x = 0 \iff x \notin \text{set } xs$
by (*induction xs*) *simp-all*

lemma *count-notin*[*simp*]: $x \notin \text{set } xs \implies \text{count-list } xs x = 0$
by(*simp add: count-list-0-iff*)

lemma *count-le-length*: $\text{count-list } xs x \leq \text{length } xs$
by (*induction xs*) *auto*

lemma *count-list-map-ge*: $\text{count-list } xs x \leq \text{count-list } (\text{map } f xs) (f x)$
by (*induction xs*) *auto*

lemma *count-list-inj-map*:
 $\llbracket \text{inj-on } f (\text{set } xs); x \in \text{set } xs \rrbracket \implies \text{count-list } (\text{map } f xs) (f x) = \text{count-list } xs x$
by (*induction xs*) (*simp, fastforce*)

lemma *count-list-rev*[*simp*]: $\text{count-list } (\text{rev } xs) x = \text{count-list } xs x$
by (*induction xs*) *auto*

lemma *sum-count-set*:
 $\text{set } xs \subseteq X \implies \text{finite } X \implies \text{sum } (\text{count-list } xs) X = \text{length } xs$
proof (*induction xs arbitrary: X*)

```

case (Cons x xs)
then show ?case
  using sum.remove [of X x count-list xs]
  by (auto simp: sum.If-cases simp flip: diff-eq)
qed simp

```

67.2.5 List.extract

lemma *extract-None-iff*: $List.extract\ P\ xs = None \longleftrightarrow \neg (\exists x \in set\ xs. P\ x)$
by(auto simp: extract-def dropWhile-eq-Cons-conv split: list.splits)
 (metis in-set-conv-decomp)

lemma *extract-SomeE*:
 $List.extract\ P\ xs = Some\ (ys, y, zs) \implies$
 $xs = ys @ y \# zs \wedge P\ y \wedge \neg (\exists y \in set\ ys. P\ y)$
by(auto simp: extract-def dropWhile-eq-Cons-conv split: list.splits)

lemma *extract-Some-iff*:
 $List.extract\ P\ xs = Some\ (ys, y, zs) \longleftrightarrow$
 $xs = ys @ y \# zs \wedge P\ y \wedge \neg (\exists y \in set\ ys. P\ y)$
by(auto simp: extract-def dropWhile-eq-Cons-conv dest: set-takeWhileD split: list.splits)

lemma *extract-Nil-code*[code]: $List.extract\ P\ [] = None$
by(simp add: extract-def)

lemma *extract-Cons-code*[code]:
 $List.extract\ P\ (x \# xs) = (if\ P\ x\ then\ Some\ ([], x, xs)\ else$
 (case $List.extract\ P\ xs$ of
 None \Rightarrow None |
 Some $(ys, y, zs) \Rightarrow$ Some $(x \# ys, y, zs)))$
by(auto simp add: extract-def comp-def split: list.splits)
 (metis dropWhile-eq-Nil-conv list.distinct(1))

67.2.6 remove1

lemma *remove1-append*:
 $remove1\ x\ (xs @ ys) =$
 (if $x \in set\ xs$ then $remove1\ x\ xs @ ys$ else $xs @ remove1\ x\ ys$)
by (induct xs) auto

lemma *remove1-commute*: $remove1\ x\ (remove1\ y\ zs) = remove1\ y\ (remove1\ x\ zs)$
by (induct zs) auto

lemma *in-set-remove1*[simp]:
 $a \neq b \implies a \in set(remove1\ b\ xs) = (a \in set\ xs)$
by (induct xs) auto

lemma *set-remove1-subset*: $set(remove1\ x\ xs) \subseteq set\ xs$
by (induct xs) auto

lemma *set-remove1-eq* [simp]: $distinct\ xs \implies set(remove1\ x\ xs) = set\ xs - \{x\}$
by (induct xs) auto

lemma *length-remove1*:
 $length(remove1\ x\ xs) = (if\ x \in set\ xs\ then\ length\ xs - 1\ else\ length\ xs)$
by (induct xs) (auto dest!:length-pos-if-in-set)

lemma *remove1-filter-not*[simp]:
 $\neg P\ x \implies remove1\ x\ (filter\ P\ xs) = filter\ P\ xs$
by(induct xs) auto

lemma *filter-remove1*:
 $filter\ Q\ (remove1\ x\ xs) = remove1\ x\ (filter\ Q\ xs)$
by (induct xs) auto

lemma *notin-set-remove1*[simp]: $x \notin set\ xs \implies x \notin set(remove1\ y\ xs)$
by(insert set-remove1-subset) fast

lemma *distinct-remove1*[simp]: $distinct\ xs \implies distinct(remove1\ x\ xs)$
by (induct xs) simp-all

lemma *remove1-remdups*:
 $distinct\ xs \implies remove1\ x\ (remdups\ xs) = remdups\ (remove1\ x\ xs)$
by (induct xs) simp-all

lemma *remove1-idem*: $x \notin set\ xs \implies remove1\ x\ xs = xs$
by (induct xs) simp-all

lemma *remove1-split*:
 $a \in set\ xs \implies remove1\ a\ xs = ys \iff (\exists\ ls\ rs.\ xs = ls @ a \# rs \wedge a \notin set\ ls \wedge ys = ls @ rs)$
by (metis remove1.simps(2) remove1-append split-list-first)

67.2.7 removeAll

lemma *removeAll-filter-not-eq*:
 $removeAll\ x = filter\ (\lambda y.\ x \neq y)$

proof
fix xs
show $removeAll\ x\ xs = filter\ (\lambda y.\ x \neq y)\ xs$
by (induct xs) auto
qed

lemma *removeAll-append*[simp]:
 $removeAll\ x\ (xs @ ys) = removeAll\ x\ xs @ removeAll\ x\ ys$
by (induct xs) auto

lemma *set-removeAll*[simp]: $set(removeAll\ x\ xs) = set\ xs - \{x\}$
by (induct xs) auto

lemma *removeAll-id[simp]*: $x \notin \text{set } xs \implies \text{removeAll } x \text{ } xs = xs$
by (*induct xs*) *auto*

lemma *removeAll-filter-not[simp]*:
 $\neg P \ x \implies \text{removeAll } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$
by(*induct xs*) *auto*

lemma *distinct-removeAll*:
 $\text{distinct } xs \implies \text{distinct } (\text{removeAll } x \ xs)$
by (*simp add: removeAll-filter-not-eq*)

lemma *distinct-remove1-removeAll*:
 $\text{distinct } xs \implies \text{remove1 } x \ xs = \text{removeAll } x \ xs$
by (*induct xs*) *simp-all*

lemma *map-removeAll-inj-on*: $\text{inj-on } f \ (\text{insert } x \ (\text{set } xs)) \implies$
 $\text{map } f \ (\text{removeAll } x \ xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$
by (*induct xs*) (*simp-all add:inj-on-def*)

lemma *map-removeAll-inj*: $\text{inj } f \implies$
 $\text{map } f \ (\text{removeAll } x \ xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$
by (*rule map-removeAll-inj-on, erule subset-inj-on, rule subset-UNIV*)

lemma *length-removeAll-less-eq [simp]*:
 $\text{length } (\text{removeAll } x \ xs) \leq \text{length } xs$
by (*simp add: removeAll-filter-not-eq*)

lemma *length-removeAll-less [termination-simp]*:
 $x \in \text{set } xs \implies \text{length } (\text{removeAll } x \ xs) < \text{length } xs$
by (*auto dest: length-filter-less simp add: removeAll-filter-not-eq*)

lemma *distinct-concat-iff*: $\text{distinct } (\text{concat } xs) \longleftrightarrow$
 $\text{distinct } (\text{removeAll } [] \ xs) \wedge$
 $(\forall \ ys. \ ys \in \text{set } xs \longrightarrow \text{distinct } ys) \wedge$
 $(\forall \ ys \ zs. \ ys \in \text{set } xs \wedge \ zs \in \text{set } xs \wedge \ ys \neq \ zs \longrightarrow \text{set } ys \cap \text{set } zs = \{\})$
apply (*induct xs*)
apply(*simp-all, safe, auto*)
by (*metis Int-iff UN-I empty-iff equalsOI set-empty*)

67.2.8 replicate

lemma *length-replicate [simp]*: $\text{length } (\text{replicate } n \ x) = n$
by (*induct n*) *auto*

lemma *replicate-eqI*:
assumes $\text{length } xs = n$ **and** $\bigwedge y. y \in \text{set } xs \implies y = x$

```

shows  $xs = replicate\ n\ x$ 
using assms
proof (induct xs arbitrary: n)
  case Nil then show ?case by simp
next
  case (Cons x xs) then show ?case by (cases n) simp-all
qed

```

```

lemma Ex-list-of-length:  $\exists xs. length\ xs = n$ 
by (rule exI[of - replicate n undefined]) simp

```

```

lemma map-replicate [simp]:  $map\ f\ (replicate\ n\ x) = replicate\ n\ (f\ x)$ 
by (induct n) auto

```

```

lemma map-replicate-const:
   $map\ (\lambda\ x. k)\ lst = replicate\ (length\ lst)\ k$ 
by (induct lst) auto

```

```

lemma replicate-app-Cons-same:
   $(replicate\ n\ x)\ @\ (x\ \#\ xs) = x\ \#\ replicate\ n\ x\ @\ xs$ 
by (induct n) auto

```

```

lemma rev-replicate [simp]:  $rev\ (replicate\ n\ x) = replicate\ n\ x$ 
by (induct n) (auto simp: replicate-app-Cons-same)

```

```

lemma replicate-add:  $replicate\ (n + m)\ x = replicate\ n\ x\ @\ replicate\ m\ x$ 
by (induct n) auto

```

Courtesy of Matthias Daum:

```

lemma append-replicate-commute:
   $replicate\ n\ x\ @\ replicate\ k\ x = replicate\ k\ x\ @\ replicate\ n\ x$ 
by (metis add.commute replicate-add)

```

Courtesy of Andreas Lochbihler:

```

lemma filter-replicate:
   $filter\ P\ (replicate\ n\ x) = (if\ P\ x\ then\ replicate\ n\ x\ else\ [])$ 
by(induct n) auto

```

```

lemma hd-replicate [simp]:  $n \neq 0 \implies hd\ (replicate\ n\ x) = x$ 
by (induct n) auto

```

```

lemma tl-replicate [simp]:  $tl\ (replicate\ n\ x) = replicate\ (n - 1)\ x$ 
by (induct n) auto

```

```

lemma last-replicate [simp]:  $n \neq 0 \implies last\ (replicate\ n\ x) = x$ 
by (atomize (full), induct n) auto

```

```

lemma nth-replicate[simp]:  $i < n \implies (replicate\ n\ x)!i = x$ 
by (induct n arbitrary: i)(auto simp: nth-Cons split: nat.split)

```

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[simp]: $\text{take } i \text{ (replicate } k \text{ } x) = \text{replicate (min } i \text{ } k) \text{ } x$

proof (*cases* $k \leq i$)

case *True*

then show *?thesis*

by (*simp add: min-def*)

next

case *False*

then have $\text{replicate } k \text{ } x = \text{replicate } i \text{ } x @ \text{replicate (} k - i \text{) } x$

by (*simp add: replicate-add [symmetric]*)

then show *?thesis*

by (*simp add: min-def*)

qed

lemma *drop-replicate*[simp]: $\text{drop } i \text{ (replicate } k \text{ } x) = \text{replicate (} k - i \text{) } x$

proof (*induct* k *arbitrary: i*)

case (*Suc* k)

then show *?case*

by (*simp add: drop-Cons'*)

qed *simp*

lemma *set-replicate-Suc*: $\text{set (replicate (} \text{Suc } n \text{) } x) = \{x\}$

by (*induct* n) *auto*

lemma *set-replicate* [simp]: $n \neq 0 \implies \text{set (replicate } n \text{ } x) = \{x\}$

by (*fast dest!: not0-implies-Suc intro!: set-replicate-Suc*)

lemma *set-replicate-conv-if*: $\text{set (replicate } n \text{ } x) = (\text{if } n = 0 \text{ then } \{\} \text{ else } \{x\})$

by *auto*

lemma *in-set-replicate*[simp]: $(x \in \text{set (replicate } n \text{ } y)) = (x = y \wedge n \neq 0)$

by (*simp add: set-replicate-conv-if*)

lemma *card-set-1-iff-replicate*:

$\text{card(set } xs) = \text{Suc } 0 \iff xs \neq [] \wedge (\exists x. xs = \text{replicate (length } xs) \text{ } x)$

by (*metis card-1-singleton-iff empty-iff insert-iff replicate-eqI set-empty2 set-replicate*)

lemma *Ball-set-replicate*[simp]:

$(\forall x \in \text{set (replicate } n \text{ } a). P \text{ } x) = (P \text{ } a \vee n = 0)$

by (*simp add: set-replicate-conv-if*)

lemma *Bex-set-replicate*[simp]:

$(\exists x \in \text{set (replicate } n \text{ } a). P \text{ } x) = (P \text{ } a \wedge n \neq 0)$

by (*simp add: set-replicate-conv-if*)

lemma *replicate-append-same*:

$\text{replicate } i \text{ } x @ [x] = x \# \text{replicate } i \text{ } x$

by (*induct* i) *simp-all*

lemma *map-replicate-trivial*:
 $\text{map } (\lambda i. x) [0..<i] = \text{replicate } i x$
by (*induct i*) (*simp-all add: replicate-append-same*)

lemma *concat-replicate-trivial[simp]*:
 $\text{concat } (\text{replicate } i []) = []$
by (*induct i*) (*auto simp add: map-replicate-const*)

lemma *replicate-empty[simp]*: $(\text{replicate } n x = []) \longleftrightarrow n=0$
by (*induct n*) *auto*

lemmas *empty-replicate[simp] = replicate-empty[THEN eq-iff-swap]*

lemma *replicate-eq-replicate[simp]*:
 $(\text{replicate } m x = \text{replicate } n y) \longleftrightarrow (m=n \wedge (m \neq 0 \longrightarrow x=y))$
proof (*induct m arbitrary: n*)
case (*Suc m n*)
then show *?case*
by (*induct n*) *auto*
qed *simp*

lemma *takeWhile-replicate[simp]*:
 $\text{takeWhile } P (\text{replicate } n x) = (\text{if } P x \text{ then replicate } n x \text{ else } [])$
using *takeWhile-eq-Nil-iff* **by** *fastforce*

lemma *dropWhile-replicate[simp]*:
 $\text{dropWhile } P (\text{replicate } n x) = (\text{if } P x \text{ then } [] \text{ else replicate } n x)$
using *dropWhile-eq-self-iff* **by** *fastforce*

lemma *replicate-length-filter*:
 $\text{replicate } (\text{length } (\text{filter } (\lambda y. x = y) xs)) x = \text{filter } (\lambda y. x = y) xs$
by (*induct xs*) *auto*

lemma *comm-append-are-replicate*:
 $xs @ ys = ys @ xs \implies \exists m n zs. \text{concat } (\text{replicate } m zs) = xs \wedge \text{concat } (\text{replicate } n zs) = ys$
proof (*induction length (xs @ ys) + length xs arbitrary: xs ys rule: less-induct*)
case *less*
consider (1) $\text{length } ys < \text{length } xs$ | (2) $xs = []$ | (3) $\text{length } xs \leq \text{length } ys \wedge xs \neq []$
by *linarith*
then show *?case*
proof (*cases*)
case 1
then show *?thesis*
using *less.hyps[OF - less.premis[symmetric]] nat-add-left-cancel-less* **by** *auto*
next
case 2


```

then have  $\text{concat} (\text{replicate } 0 \text{ } ys) = xs \wedge \text{concat} (\text{replicate } 1 \text{ } ys) = ys$ 
  by simp
then show ?thesis
  by blast
next
case  $\mathcal{I}$ 
then have  $\text{length } xs \leq \text{length } ys$  and  $xs \neq []$ 
  by blast+
from  $\langle \text{length } xs \leq \text{length } ys \rangle$  and  $\langle xs @ ys = ys @ xs \rangle$ 
obtain  $ws$  where  $ys = xs @ ws$ 
  by (auto simp: append-eq-append-conv2)
from this and  $\langle xs \neq [] \rangle$ 
have  $\text{length } ws < \text{length } ys$ 
  by simp
from  $\langle xs @ ys = ys @ xs \rangle$  [unfolded  $\langle ys = xs @ ws \rangle$ ]
have  $xs @ ws = ws @ xs$ 
  by simp
from less.hyps [OF - this]  $\langle \text{length } ws < \text{length } ys \rangle$ 
obtain  $m \ n' \ zs$  where  $\text{concat} (\text{replicate } m \ zs) = xs$  and  $\text{concat} (\text{replicate } n' \ zs) = ws$ 
  by auto
then have  $\text{concat} (\text{replicate } (m+n') \ zs) = ys$ 
  using  $\langle ys = xs @ ws \rangle$ 
  by (simp add: replicate-add)
then show ?thesis
  using  $\langle \text{concat} (\text{replicate } m \ zs) = xs \rangle$  by blast
qed
qed

```

lemma *comm-append-is-replicate*:

```

fixes  $xs \ ys :: 'a \text{ list}$ 
assumes  $xs \neq [] \ ys \neq []$ 
assumes  $xs @ ys = ys @ xs$ 
shows  $\exists n \ zs. \ n > 1 \wedge \text{concat} (\text{replicate } n \ zs) = xs @ ys$ 
proof –

```

```

obtain  $m \ n \ zs$  where  $\text{concat} (\text{replicate } m \ zs) = xs$ 
  and  $\text{concat} (\text{replicate } n \ zs) = ys$ 
  using comm-append-are-replicate [OF assms( $\mathcal{I}$ )] by blast
then have  $m + n > 1$  and  $\text{concat} (\text{replicate } (m+n) \ zs) = xs @ ys$ 
  using  $\langle xs \neq [] \rangle$  and  $\langle ys \neq [] \rangle$ 
  by (auto simp: replicate-add)
then show ?thesis by blast
qed

```

lemma *Cons-replicate-eq*:

```

 $x \# xs = \text{replicate } n \ y \longleftrightarrow x = y \wedge n > 0 \wedge xs = \text{replicate } (n - 1) \ x$ 
by (induct n) auto

```

lemma *replicate-length-same*:

$(\forall y \in \text{set } xs. y = x) \implies \text{replicate } (\text{length } xs) \ x = xs$
by *(induct xs) simp-all*

lemma *foldr-replicate* [*simp*]:
 $\text{foldr } f \ (\text{replicate } n \ x) = f \ x \ \overset{\sim}{\sim} \ n$
by *(induct n) (simp-all)*

lemma *fold-replicate* [*simp*]:
 $\text{fold } f \ (\text{replicate } n \ x) = f \ x \ \overset{\sim}{\sim} \ n$
by *(subst foldr-fold [symmetric]) simp-all*

67.2.9 enumerate

lemma *enumerate-simps* [*simp, code*]:
 $\text{enumerate } n \ [] = []$
 $\text{enumerate } n \ (x \# \ xs) = (n, x) \# \ \text{enumerate } (\text{Suc } n) \ xs$
by *(simp-all add: enumerate-eq-zip upt-rec)*

lemma *length-enumerate* [*simp*]:
 $\text{length } (\text{enumerate } n \ xs) = \text{length } xs$
by *(simp add: enumerate-eq-zip)*

lemma *map-fst-enumerate* [*simp*]:
 $\text{map } \text{fst} \ (\text{enumerate } n \ xs) = [n..<n + \text{length } xs]$
by *(simp add: enumerate-eq-zip)*

lemma *map-snd-enumerate* [*simp*]:
 $\text{map } \text{snd} \ (\text{enumerate } n \ xs) = xs$
by *(simp add: enumerate-eq-zip)*

lemma *in-set-enumerate-eq*:
 $p \in \text{set } (\text{enumerate } n \ xs) \iff n \leq \text{fst } p \wedge \text{fst } p < \text{length } xs + n \wedge \text{nth } xs \ (\text{fst } p - n) = \text{snd } p$

proof –

{ **fix** m
assume $n \leq m$
moreover assume $m < \text{length } xs + n$
ultimately have $[n..<n + \text{length } xs] ! (m - n) = m \wedge$
 $xs ! (m - n) = xs ! (m - n) \wedge m - n < \text{length } xs$ **by** *auto*
then have $\exists q. [n..<n + \text{length } xs] ! q = m \wedge$
 $xs ! q = xs ! (m - n) \wedge q < \text{length } xs$..
} then show *?thesis* **by** *(cases p) (auto simp add: enumerate-eq-zip in-set-zip)*

qed

lemma *nth-enumerate-eq*: $m < \text{length } xs \implies \text{enumerate } n \ xs ! m = (n + m, xs ! m)$
by *(simp add: enumerate-eq-zip)*

lemma *enumerate-replicate-eq*:

$enumerate\ n\ (replicate\ m\ a) = map\ (\lambda q.\ (q,\ a))\ [n..<n + m]$
by (rule pair-list-eqI)
 (simp-all add: enumerate-eq-zip comp-def map-replicate-const)

lemma enumerate-Suc-eq:
 $enumerate\ (Suc\ n)\ xs = map\ (apfst\ Suc)\ (enumerate\ n\ xs)$
by (rule pair-list-eqI)
 (simp-all add: not-le, simp del: map-map add: map-Suc-upt map-map [symmetric])

lemma distinct-enumerate [simp]:
 $distinct\ (enumerate\ n\ xs)$
by (simp add: enumerate-eq-zip distinct-zipI1)

lemma enumerate-append-eq:
 $enumerate\ n\ (xs\ @\ ys) = enumerate\ n\ xs\ @\ enumerate\ (n + length\ xs)\ ys$
by (simp add: enumerate-eq-zip add.assoc zip-append2)

lemma enumerate-map-upt:
 $enumerate\ n\ (map\ f\ [n..<m]) = map\ (\lambda k.\ (k,\ f\ k))\ [n..<m]$
by (cases $n \leq m$) (simp-all add: zip-map2 zip-same-conv-map enumerate-eq-zip)

67.2.10 rotate1 and rotate

lemma rotate0[simp]: $rotate\ 0 = id$
by(simp add:rotate-def)

lemma rotate-Suc[simp]: $rotate\ (Suc\ n)\ xs = rotate1\ (rotate\ n\ xs)$
by(simp add:rotate-def)

lemma rotate-add:
 $rotate\ (m+n) = rotate\ m \circ rotate\ n$
by(simp add:rotate-def funpow-add)

lemma rotate-rotate: $rotate\ m\ (rotate\ n\ xs) = rotate\ (m+n)\ xs$
by(simp add:rotate-add)

lemma rotate1-map: $rotate1\ (map\ f\ xs) = map\ f\ (rotate1\ xs)$
by(cases xs) simp-all

lemma rotate1-rotate-swap: $rotate1\ (rotate\ n\ xs) = rotate\ n\ (rotate1\ xs)$
by(simp add:rotate-def funpow-swap1)

lemma rotate1-length01[simp]: $length\ xs \leq 1 \implies rotate1\ xs = xs$
by(cases xs) simp-all

lemma rotate-length01[simp]: $length\ xs \leq 1 \implies rotate\ n\ xs = xs$
by (induct n) (simp-all add:rotate-def)

lemma rotate1-hd-tl: $xs \neq [] \implies rotate1\ xs = tl\ xs\ @\ [hd\ xs]$

by (*cases xs simp-all*)

lemma *rotate-drop-take*:

rotate n xs = drop (n mod length xs) xs @ take (n mod length xs) xs

proof (*induct n*)

case (*Suc n*)

show *?case*

proof (*cases xs = []*)

case *False*

then show *?thesis*

proof (*cases n mod length xs = 0*)

case *True*

then show *?thesis*

by (*auto simp add: mod-Suc False Suc.hyps drop-Suc rotate1-hd-tl take-Suc Suc-length-conv*)

next

case *False*

with $\langle xs \neq [] \rangle$ *Suc*

show *?thesis*

by (*simp add: rotate-def mod-Suc rotate1-hd-tl drop-Suc[symmetric] drop-tl[symmetric] take-hd-drop linorder-not-le*)

qed

qed *simp*

qed *simp*

lemma *rotate-conv-mod*: *rotate n xs = rotate (n mod length xs) xs*

by(*simp add:rotate-drop-take*)

lemma *rotate-id[simp]*: *n mod length xs = 0 \implies rotate n xs = xs*

by(*simp add:rotate-drop-take*)

lemma *length-rotate1[simp]*: *length(rotate1 xs) = length xs*

by (*cases xs simp-all*)

lemma *length-rotate[simp]*: *length(rotate n xs) = length xs*

by (*induct n arbitrary: xs*) (*simp-all add:rotate-def*)

lemma *distinct1-rotate[simp]*: *distinct(rotate1 xs) = distinct xs*

by (*cases xs auto*)

lemma *distinct-rotate[simp]*: *distinct(rotate n xs) = distinct xs*

by (*induct n*) (*simp-all add:rotate-def*)

lemma *rotate-map*: *rotate n (map f xs) = map f (rotate n xs)*

by(*simp add:rotate-drop-take take-map drop-map*)

lemma *set-rotate1[simp]*: *set(rotate1 xs) = set xs*

by (*cases xs auto*)

lemma *set-rotate[simp]*: $set(rotate\ n\ xs) = set\ xs$
by (*induct n*) (*simp-all add:rotate-def*)

lemma *rotate1-replicate[simp]*: $rotate1\ (replicate\ n\ a) = replicate\ n\ a$
by (*cases n*) (*simp-all add:replicate-append-same*)

lemma *rotate1-is-Nil-conv[simp]*: $(rotate1\ xs = []) = (xs = [])$
by (*cases xs*) *auto*

lemma *rotate-is-Nil-conv[simp]*: $(rotate\ n\ xs = []) = (xs = [])$
by (*induct n*) (*simp-all add:rotate-def*)

lemma *rotate-rev*:
 $rotate\ n\ (rev\ xs) = rev(rotate\ (length\ xs - (n\ mod\ length\ xs))\ xs)$
proof (*cases length xs = 0 \vee n mod length xs = 0*)
case *False*
then show *?thesis*
by(*simp add:rotate-drop-take rev-drop rev-take*)
qed *force*

lemma *hd-rotate-conv-nth*:
assumes $xs \neq []$ **shows** $hd(rotate\ n\ xs) = xs!(n\ mod\ length\ xs)$
proof –
have $n\ mod\ length\ xs < length\ xs$
using *assms* **by** *simp*
then show *?thesis*
by (*metis drop-eq-Nil hd-append2 hd-drop-conv-nth leD rotate-drop-take*)
qed

lemma *rotate-append*: $rotate\ (length\ l)\ (l\ @\ q) = q\ @\ l$
by (*induct l arbitrary: q*) (*auto simp add:rotate1-rotate-swap*)

lemma *nth-rotate*:
 $\langle rotate\ m\ xs\ !\ n = xs\ !\ ((m + n)\ mod\ length\ xs) \rangle$ **if** $\langle n < length\ xs \rangle$
using that **apply** (*auto simp add:rotate-drop-take nth-append not-less less-diff-conv ac-simps dest!: le-Suc-ex*)
apply (*metis add commute mod-add-right-eq mod-less*)
apply (*metis (no-types, lifting) Nat.diff-diff-right add commute add-diff-cancel-right' diff-le-self dual-order.strict-trans2 length-greater-0-conv less-nat-zero-code list.size(3) mod-add-right-eq mod-add-self2 mod-le-divisor mod-less*)
done

lemma *nth-rotate1*:
 $\langle rotate1\ xs\ !\ n = xs\ !\ (Suc\ n\ mod\ length\ xs) \rangle$ **if** $\langle n < length\ xs \rangle$
using that *nth-rotate [of n xs 1]* **by** *simp*

lemma *inj-rotate1*: *inj rotate1*
proof
fix $xs\ ys :: 'a\ list$ **show** $rotate1\ xs = rotate1\ ys \implies xs = ys$

by (cases xs; cases ys; simp)
qed

lemma *surj-rotate1*: *surj rotate1*
proof (safe, simp-all)
fix xs :: 'a list **show** xs ∈ range rotate1
proof (cases xs rule: rev-exhaust)
case Nil
hence xs = rotate1 [] by auto
thus ?thesis by fast
next
case (snoc as a)
hence xs = rotate1 (a#as) by force
thus ?thesis by fast
qed
qed

lemma *bij-rotate1*: *bij (rotate1 :: 'a list ⇒ 'a list)*
using *bijI inj-rotate1 surj-rotate1* by blast

lemma *rotate1-fixpoint-card*: *rotate1 xs = xs ⇒ xs = [] ∨ card(set xs) = 1*
by(*induction xs*) (*auto simp: card-insert-if[OF finite-set] append-eq-Cons-conv*)

67.2.11 *nths* — a generalization of (!) to sets

lemma *nths-empty* [*simp*]: *nths xs {} = []*
by (*auto simp add: nths-def*)

lemma *nths-nil* [*simp*]: *nths [] A = []*
by (*auto simp add: nths-def*)

lemma *nths-all*: $\forall i < \text{length } xs. i \in I \implies \text{nths } xs \ I = xs$
apply (*simp add: nths-def*)
apply (*subst filter-True*)
apply (*auto simp: in-set-zip subset-iff*)
done

lemma *length-nths*:
 $\text{length } (\text{nths } xs \ I) = \text{card}\{i. i < \text{length } xs \wedge i \in I\}$
by(*simp add: nths-def length-filter-conv-card cong:conj-cong*)

lemma *nths-shift-lemma-Suc*:
 $\text{map } \text{fst } (\text{filter } (\lambda p. P(\text{Suc}(\text{snd } p))) (\text{zip } xs \ is)) =$
 $\text{map } \text{fst } (\text{filter } (\lambda p. P(\text{snd } p)) (\text{zip } xs \ (\text{map } \text{Suc } is)))$
proof (*induct xs arbitrary: is*)
case (*Cons x xs is*)
show ?case
by (*cases is*) (*auto simp add: Cons.hyps*)
qed *simp*

lemma *nths-shift-lemma*:

$\text{map fst (filter } (\lambda p. \text{snd } p \in A) (\text{zip } xs [i..<i + \text{length } xs])) =$
 $\text{map fst (filter } (\lambda p. \text{snd } p + i \in A) (\text{zip } xs [0..<\text{length } xs]))$
by (*induct xs rule: rev-induct*) (*simp-all add: add commute*)

lemma *nths-append*:

$\text{nths } (l @ l') A = \text{nths } l A @ \text{nths } l' \{j. j + \text{length } l \in A\}$
unfolding *nths-def*
proof (*induct l' rule: rev-induct*)
case (*snoc x xs*)
then show *?case*
by (*simp add: upt-add-eq-append[of 0] nths-shift-lemma add commute*)
qed *auto*

lemma *nths-Cons*:

$\text{nths } (x \# l) A = (\text{if } 0 \in A \text{ then } [x] \text{ else } []) @ \text{nths } l \{j. \text{Suc } j \in A\}$
proof (*induct l rule: rev-induct*)
case (*snoc x xs*)
then show *?case*
by (*simp flip: append-Cons add: nths-append*)
qed (*auto simp: nths-def*)

lemma *nths-map*: $\text{nths } (\text{map } f \text{ } xs) I = \text{map } f (\text{nths } xs I)$

by (*induction xs arbitrary: I*) (*simp-all add: nths-Cons*)

lemma *set-nths*: $\text{set}(\text{nths } xs I) = \{xs!i \mid i < \text{size } xs \wedge i \in I\}$

by (*induct xs arbitrary: I*) (*auto simp: nths-Cons nth-Cons split:nat.split dest!: gr0-implies-Suc*)

lemma *set-nths-subset*: $\text{set}(\text{nths } xs I) \subseteq \text{set } xs$

by (*auto simp add: set-nths*)

lemma *notin-set-nthsI[simp]*: $x \notin \text{set } xs \implies x \notin \text{set}(\text{nths } xs I)$

by (*auto simp add: set-nths*)

lemma *in-set-nthsD*: $x \in \text{set}(\text{nths } xs I) \implies x \in \text{set } xs$

by (*auto simp add: set-nths*)

lemma *nths-singleton [simp]*: $\text{nths } [x] A = (\text{if } 0 \in A \text{ then } [x] \text{ else } [])$

by (*simp add: nths-Cons*)

lemma *distinct-nthsI[simp]*: $\text{distinct } xs \implies \text{distinct } (\text{nths } xs I)$

by (*induct xs arbitrary: I*) (*auto simp: nths-Cons*)

lemma *nths-upt-eq-take [simp]*: $\text{nths } l \{..<n\} = \text{take } n \text{ } l$

by (*induct l rule: rev-induct*) (*simp-all split: nat-diff-split add: nths-append*)

lemma *nths-nths*: $\text{nths } (\text{nths } xs A) B = \text{nths } xs \{i \in A. \exists j \in B. \text{card } \{i' \in A. i' \in B\} = j\}$

$< i \} = j \}$
by (*induction xs arbitrary: A B*) (*auto simp add: nth-Cons card-less-Suc card-less-Suc2*)

lemma *drop-eq-nths*: $\text{drop } n \text{ } xs = \text{nths } xs \{i. i \geq n\}$
by (*induction xs arbitrary: n*) (*auto simp add: nth-Cons nth-all drop-Cons'*)
intro: arg-cong2[where f=nths, OF refl])

lemma *nths-drop*: $\text{nths } (\text{drop } n \text{ } xs) \ I = \text{nths } xs \ ((+) \ n \ ' \ I)$
by(*force simp: drop-eq-nths nth-nths simp flip: atLeastLessThan-iff*)
intro: arg-cong2[where f=nths, OF refl])

lemma *filter-eq-nths*: $\text{filter } P \text{ } xs = \text{nths } xs \ \{i. i < \text{length } xs \wedge P(xs!i)\}$
by(*induction xs*) (*auto simp: nth-Cons*)

lemma *filter-in-nths*:
 $\text{distinct } xs \implies \text{filter } (\%x. x \in \text{set } (\text{nths } xs \ s)) \ xs = \text{nths } xs \ s$
proof (*induct xs arbitrary: s*)
case Nil **thus** ?*case* **by** *simp*
next
case (*Cons a xs*)
then have $\forall x. x \in \text{set } xs \longrightarrow x \neq a$ **by** *auto*
with Cons **show** ?*case* **by**(*simp add: nth-Cons cong:filter-cong*)
qed

67.2.12 subseqs and List.n-lists

lemma *length-subseqs*: $\text{length } (\text{subseqs } xs) = 2 \wedge \text{length } xs$
by (*induct xs*) (*simp-all add: Let-def*)

lemma *subseqs-powset*: $\text{set } ' \ \text{set } (\text{subseqs } xs) = \text{Pow } (\text{set } xs)$

proof –
have *aux*: $\bigwedge x \ A. \ \text{set } ' \ \text{Cons } x \ ' \ A = \text{insert } x \ ' \ \text{set } ' \ A$
by (*auto simp add: image-def*)
have $\text{set } (\text{map } \text{set } (\text{subseqs } xs)) = \text{Pow } (\text{set } xs)$
by (*induct xs*) (*simp-all add: aux Let-def Pow-insert Un-commute comp-def del: map-map*)
then show ?*thesis* **by** *simp*
qed

lemma *distinct-set-subseqs*:
assumes *distinct xs*
shows *distinct (map set (subseqs xs))*
by (*simp add: assms card-Pow card-distinct distinct-card length-subseqs subseqs-powset*)

lemma *n-lists-Nil* [*simp*]: $\text{List.n-lists } n \ [] = (\text{if } n = 0 \ \text{then } [[]] \ \text{else } [])$
by (*induct n*) *simp-all*

lemma *length-n-lists-elem*: $ys \in \text{set } (\text{List.n-lists } n \ xs) \implies \text{length } ys = n$

by (induct n arbitrary: ys) auto

lemma *set-n-lists*: $\text{set } (\text{List.n-lists } n \text{ } xs) = \{ys. \text{length } ys = n \wedge \text{set } ys \subseteq \text{set } xs\}$
proof (rule *set-eqI*)
 fix *ys* :: 'a list
show $ys \in \text{set } (\text{List.n-lists } n \text{ } xs) \longleftrightarrow ys \in \{ys. \text{length } ys = n \wedge \text{set } ys \subseteq \text{set } xs\}$
proof –
 have $ys \in \text{set } (\text{List.n-lists } n \text{ } xs) \implies \text{length } ys = n$
 by (induct n arbitrary: ys) auto
moreover have $\bigwedge x. ys \in \text{set } (\text{List.n-lists } n \text{ } xs) \implies x \in \text{set } ys \implies x \in \text{set } xs$
 by (induct n arbitrary: ys) auto
moreover have $\text{set } ys \subseteq \text{set } xs \implies ys \in \text{set } (\text{List.n-lists } (\text{length } ys) \text{ } xs)$
 by (induct ys) auto
ultimately show *?thesis* by auto
qed
qed

lemma *subseqs-refl*: $xs \in \text{set } (\text{subseqs } xs)$
 by (induct xs) (simp-all add: *Let-def*)

lemma *subset-subseqs*: $X \subseteq \text{set } xs \implies X \in \text{set } ' \text{set } (\text{subseqs } xs)$
 unfolding *subseqs-powset* by *simp*

lemma *Cons-in-subseqsD*: $y \# ys \in \text{set } (\text{subseqs } xs) \implies ys \in \text{set } (\text{subseqs } xs)$
 by (induct xs) (auto simp: *Let-def*)

lemma *subseqs-distinctD*: $\llbracket ys \in \text{set } (\text{subseqs } xs); \text{distinct } xs \rrbracket \implies \text{distinct } ys$
proof (induct xs arbitrary: ys)
 case (*Cons x xs ys*)
 then show *?case*
 by (auto simp: *Let-def*) (metis *Pow-iff contra-subsetD image-eqI subseqs-powset*)
qed *simp*

67.2.13 splice

lemma *splice-Nil2* [*simp*]: $\text{splice } xs \ [] = xs$
 by (cases xs) *simp-all*

lemma *length-splice*[*simp*]: $\text{length}(\text{splice } xs \ ys) = \text{length } xs + \text{length } ys$
 by (induct xs ys rule: *splice.induct*) auto

lemma *split-Nil-iff*[*simp*]: $\text{splice } xs \ ys = [] \longleftrightarrow xs = [] \wedge ys = []$
 by (induct xs ys rule: *splice.induct*) auto

lemma *splice-replicate*[*simp*]: $\text{splice } (\text{replicate } m \ x) \ (\text{replicate } n \ x) = \text{replicate } (m+n) \ x$

proof (induction replicate m x replicate n x arbitrary: m n rule: *splice.induct*)
 case (*2 x xs*)
 then show *?case*

by (auto simp add: Cons-replicate-eq dest: gr0-implies-Suc)
qed auto

67.2.14 shuffles

lemma *shuffles-commutes*: $shuffles\ xs\ ys = shuffles\ ys\ xs$
by (induction xs ys rule: shuffles.induct) (simp-all add: Un-commute)

lemma *Nil-in-shuffles[simp]*: $[] \in shuffles\ xs\ ys \longleftrightarrow xs = [] \wedge ys = []$
by (induct xs ys rule: shuffles.induct) auto

lemma *shufflesE*:

$zs \in shuffles\ xs\ ys \implies$
 $(zs = xs \implies ys = [] \implies P) \implies$
 $(zs = ys \implies xs = [] \implies P) \implies$
 $(\bigwedge x\ xs'\ z\ zs'.\ xs = x \# xs' \implies zs = z \# zs' \implies x = z \implies zs' \in shuffles\ xs'$
 $ys \implies P) \implies$
 $(\bigwedge y\ ys'\ z\ zs'.\ ys = y \# ys' \implies zs = z \# zs' \implies y = z \implies zs' \in shuffles\ xs$
 $ys' \implies P) \implies P$
 by (induct xs ys rule: shuffles.induct) auto

lemma *Cons-in-shuffles-iff*:

$z \# zs \in shuffles\ xs\ ys \longleftrightarrow$
 $(xs \neq [] \wedge hd\ xs = z \wedge zs \in shuffles\ (tl\ xs)\ ys \vee$
 $ys \neq [] \wedge hd\ ys = z \wedge zs \in shuffles\ xs\ (tl\ ys))$
 by (induct xs ys rule: shuffles.induct) auto

lemma *splice-in-shuffles* [simp, intro]: $splice\ xs\ ys \in shuffles\ xs\ ys$

by (induction xs ys rule: splice.induct) (simp-all add: Cons-in-shuffles-iff shuffles-commutes)

lemma *Nil-in-shufflesI*: $xs = [] \implies ys = [] \implies [] \in shuffles\ xs\ ys$
by simp

lemma *Cons-in-shuffles-leftI*: $zs \in shuffles\ xs\ ys \implies z \# zs \in shuffles\ (z \# xs)$
ys
by (cases ys) auto

lemma *Cons-in-shuffles-rightI*: $zs \in shuffles\ xs\ ys \implies z \# zs \in shuffles\ xs\ (z \# ys)$
by (cases xs) auto

lemma *finite-shuffles* [simp, intro]: $finite\ (shuffles\ xs\ ys)$
by (induction xs ys rule: shuffles.induct) simp-all

lemma *length-shuffles*: $zs \in shuffles\ xs\ ys \implies length\ zs = length\ xs + length\ ys$
by (induction xs ys arbitrary: zs rule: shuffles.induct) auto

lemma *set-shuffles*: $zs \in shuffles\ xs\ ys \implies set\ zs = set\ xs \cup set\ ys$

by (induction xs ys arbitrary: zs rule: shuffles.induct) auto

lemma *distinct-disjoint-shuffles*:

assumes *distinct xs distinct ys set xs \cap set ys = {}* *zs \in shuffles xs ys*
shows *distinct zs*

using *assms*

proof (induction xs ys arbitrary: zs rule: shuffles.induct)

case (\exists x xs y ys)

show ?case

proof (cases zs)

case (Cons z zs')

with \exists .prems and \exists .IH[of zs'] show ?thesis by (force dest: set-shuffles)

qed simp-all

qed simp-all

lemma *Cons-shuffles-subset1*: ($\#$) *x ' shuffles xs ys \subseteq shuffles (x # xs) ys*

by (cases ys) auto

lemma *Cons-shuffles-subset2*: ($\#$) *y ' shuffles xs ys \subseteq shuffles xs (y # ys)*

by (cases xs) auto

lemma *filter-shuffles*:

filter P ' shuffles xs ys = shuffles (filter P xs) (filter P ys)

proof –

have *: *filter P ' ($\#$) x ' A = (if P x then ($\#$) x ' filter P ' A else filter P ' A)*

for x A

by (auto simp: image-image)

show ?thesis

by (induction xs ys rule: shuffles.induct)

(simp-all split: if-splits add: image-Un * Un-absorb1 Un-absorb2
Cons-shuffles-subset1 Cons-shuffles-subset2)

qed

lemma *filter-shuffles-disjoint1*:

assumes *set xs \cap set ys = {}* *zs \in shuffles xs ys*

shows *filter (λ x. x \in set xs) zs = xs (is filter ?P - = -)*

and *filter (λ x. x \notin set xs) zs = ys (is filter ?Q - = -)*

using *assms*

proof –

from *assms* have *filter ?P zs \in filter ?P ' shuffles xs ys* by blast

also have *filter ?P ' shuffles xs ys = shuffles (filter ?P xs) (filter ?P ys)*

by (rule filter-shuffles)

also have *filter ?P xs = xs* by (rule filter-True) simp-all

also have *filter ?P ys = []* by (rule filter-False) (insert assms(1), auto)

also have *shuffles xs [] = {xs}* by simp

finally show *filter ?P zs = xs* by simp

next

from *assms* have *filter ?Q zs \in filter ?Q ' shuffles xs ys* by blast

also have *filter ?Q ' shuffles xs ys = shuffles (filter ?Q xs) (filter ?Q ys)*

by (rule filter-shuffles)
 also have filter ?Q ys = ys by (rule filter-True) (insert assms(1), auto)
 also have filter ?Q xs = [] by (rule filter-False) (insert assms(1), auto)
 also have shuffles [] ys = {ys} by simp
 finally show filter ?Q zs = ys by simp
 qed

lemma filter-shuffles-disjoint2:

assumes set xs \cap set ys = {} zs \in shuffles xs ys
 shows filter ($\lambda x. x \in$ set ys) zs = ys filter ($\lambda x. x \notin$ set ys) zs = xs
 using filter-shuffles-disjoint1 [of ys xs zs] assms
 by (simp-all add: shuffles-commutes Int-commute)

lemma partition-in-shuffles:

xs \in shuffles (filter P xs) (filter ($\lambda x. \neg P$ x) xs)
 proof (induction xs)
 case (Cons x xs)
 show ?case
 proof (cases P x)
 case True
 hence x # xs \in (#) x ' shuffles (filter P xs) (filter ($\lambda x. \neg P$ x) xs)
 by (intro imageI Cons.IH)
 also have ... \subseteq shuffles (filter P (x # xs)) (filter ($\lambda x. \neg P$ x) (x # xs))
 by (simp add: True Cons-shuffles-subset1)
 finally show ?thesis .
 next
 case False
 hence x # xs \in (#) x ' shuffles (filter P xs) (filter ($\lambda x. \neg P$ x) xs)
 by (intro imageI Cons.IH)
 also have ... \subseteq shuffles (filter P (x # xs)) (filter ($\lambda x. \neg P$ x) (x # xs))
 by (simp add: False Cons-shuffles-subset2)
 finally show ?thesis .
 qed
 qed auto

lemma inv-image-partition:

assumes $\bigwedge x. x \in$ set xs $\implies P$ x $\bigwedge y. y \in$ set ys $\implies \neg P$ y
 shows partition P - ' {(xs, ys)} = shuffles xs ys
 proof (intro equalityI subsetI)
 fix zs assume zs: zs \in shuffles xs ys
 hence [simp]: set zs = set xs \cup set ys by (rule set-shuffles)
 from assms have filter P zs = filter ($\lambda x. x \in$ set xs) zs
 filter ($\lambda x. \neg P$ x) zs = filter ($\lambda x. x \in$ set ys) zs
 by (intro filter-cong refl; force)+
 moreover from assms have set xs \cap set ys = {} by auto
 ultimately show zs \in partition P - ' {(xs, ys)} using zs
 by (simp add: o-def filter-shuffles-disjoint1 filter-shuffles-disjoint2)
 next
 fix zs assume zs \in partition P - ' {(xs, ys)}

thus $zs \in \text{shuffles } xs \text{ } ys$ **using** *partition-in-shuffles*[of zs] **by** (*auto simp: o-def*)
qed

67.2.15 Transpose

function *transpose* **where**
transpose [] = [] |
transpose ([] # xss) = *transpose* xss |
transpose ($(x\#xs) \# xss$) =
 $(x \# [h. (h\#t) \leftarrow xss]) \# \text{transpose } (xs \# [t. (h\#t) \leftarrow xss])$
by *pat-completeness auto*

lemma *transpose-aux-filter-head*:
 $\text{concat } (\text{map } (\text{case-list } [] (\lambda h t. [h])) \text{ } xss) =$
 $\text{map } (\lambda xs. \text{hd } xs) (\text{filter } (\lambda ys. ys \neq []) \text{ } xss)$
by (*induct* xss) (*auto split: list.split*)

lemma *transpose-aux-filter-tail*:
 $\text{concat } (\text{map } (\text{case-list } [] (\lambda h t. [t])) \text{ } xss) =$
 $\text{map } (\lambda xs. \text{tl } xs) (\text{filter } (\lambda ys. ys \neq []) \text{ } xss)$
by (*induct* xss) (*auto split: list.split*)

lemma *transpose-aux-max*:
 $\text{max } (\text{Suc } (\text{length } xs)) (\text{foldr } (\lambda xs. \text{max } (\text{length } xs)) \text{ } xss \text{ } 0) =$
 $\text{Suc } (\text{max } (\text{length } xs) (\text{foldr } (\lambda x. \text{max } (\text{length } x - \text{Suc } 0)) (\text{filter } (\lambda ys. ys \neq []) \text{ } xss) \text{ } 0))$
(is $\text{max} - ?\text{foldB} = \text{Suc } (\text{max} - ?\text{foldA})$)

proof (*cases* $(\text{filter } (\lambda ys. ys \neq []) \text{ } xss) = []$)
case *True*
hence $\text{foldr } (\lambda xs. \text{max } (\text{length } xs)) \text{ } xss \text{ } 0 = 0$
proof (*induct* xss)
case (*Cons* $x \text{ } xs$)
then **have** $x = []$ **by** (*cases* x) *auto*
with *Cons* **show** *?case* **by** *auto*
qed *simp*
thus *?thesis* **using** *True* **by** *simp*

next

case *False*

have *foldA*: $?\text{foldA} = \text{foldr } (\lambda x. \text{max } (\text{length } x)) (\text{filter } (\lambda ys. ys \neq []) \text{ } xss) \text{ } 0 - 1$
by (*induct* xss) *auto*
have *foldB*: $?\text{foldB} = \text{foldr } (\lambda x. \text{max } (\text{length } x)) (\text{filter } (\lambda ys. ys \neq []) \text{ } xss) \text{ } 0$
by (*induct* xss) *auto*

have $0 < ?\text{foldB}$

proof –

from *False*

obtain $z \text{ } zs$ **where** $zs: (\text{filter } (\lambda ys. ys \neq []) \text{ } xss) = z\#zs$ **by** (*auto simp: neq-Nil-conv*)

```

hence  $z \in \text{set } (\text{filter } (\lambda ys. ys \neq []) xss)$  by auto
hence  $z \neq []$  by auto
thus ?thesis
  unfolding foldB zs
  by (auto simp: max-def intro: less-le-trans)
qed
thus ?thesis
  unfolding foldA foldB max-Suc-Suc[symmetric]
  by simp
qed

termination transpose
by (relation measure  $(\lambda xs. \text{foldr } (\lambda xs. \text{max } (\text{length } xs)) xs 0 + \text{length } xs)$ )
  (auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max
less-Suc-eq-le)

lemma transpose-empty:  $(\text{transpose } xs = []) \longleftrightarrow (\forall x \in \text{set } xs. x = [])$ 
by (induct rule: transpose.induct) simp-all

lemma length-transpose:
fixes  $xs :: 'a \text{ list list}$ 
shows  $\text{length } (\text{transpose } xs) = \text{foldr } (\lambda xs. \text{max } (\text{length } xs)) xs 0$ 
by (induct rule: transpose.induct)
  (auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max
max-Suc-Suc[symmetric] simp del: max-Suc-Suc)

lemma nth-transpose:
fixes  $xs :: 'a \text{ list list}$ 
assumes  $i < \text{length } (\text{transpose } xs)$ 
shows  $\text{transpose } xs ! i = \text{map } (\lambda xs. xs ! i) (\text{filter } (\lambda ys. i < \text{length } ys) xs)$ 
using assms proof (induct arbitrary: i rule: transpose.induct)
case  $(\exists x \ xs \ xss)$ 
define  $XS$  where  $XS = (x \# xs) \# xss$ 
hence [simp]:  $XS \neq []$  by auto
thus ?case
proof (cases i)
  case 0
  thus ?thesis by (simp add: transpose-aux-filter-head hd-conv-nth)
next
  case  $(\text{Suc } j)$ 
  have  $*$ :  $\bigwedge xss. xs \# \text{map } \text{tl } xss = \text{map } \text{tl } ((x \# xs) \# xss)$  by simp
  have  $**$ :  $\bigwedge xss. (x \# xs) \# \text{filter } (\lambda ys. ys \neq []) xss = \text{filter } (\lambda ys. ys \neq [])$ 
 $((x \# xs) \# xss)$  by simp
  { fix  $xs :: 'a \text{ list}$  have  $\text{Suc } j < \text{length } xs \longleftrightarrow xs \neq [] \wedge j < \text{length } xs - \text{Suc } 0$ 
    by (cases xs) simp-all
    } note  $*** = \text{this}$ 

  have  $j\text{-less}$ :  $j < \text{length } (\text{transpose } (xs \# \text{concat } (\text{map } (\text{case-list } [] (\lambda h \ t. [t]))) xss))$ 

```

```

using 3.premis by (simp add: transpose-aux-filter-tail length-transpose Suc)

show ?thesis
  unfolding transpose.simps ⟨i = Suc j⟩ nth-Cons-Suc 3.hyps[OF j-less]
  apply (auto simp: transpose-aux-filter-tail filter-map comp-def length-transpose
* ** ** XS-def[symmetric])
  by (simp add: nth-tl)
qed
qed simp-all

```

lemma transpose-map-map:

```

  transpose (map (map f) xs) = map (map f) (transpose xs)
proof (rule nth-equalityI)
  have [simp]: length (transpose (map (map f) xs)) = length (transpose xs)
  by (simp add: length-transpose foldr-map comp-def)
  show length (transpose (map (map f) xs)) = length (map (map f) (transpose xs))
by simp

```

```

fix i assume i < length (transpose (map (map f) xs))
thus transpose (map (map f) xs) ! i = map (map f) (transpose xs) ! i
  by (simp add: nth-transpose filter-map comp-def)
qed

```

67.2.16 min and arg-min

lemma min-list-Min: $xs \neq [] \implies \text{min-list } xs = \text{Min } (\text{set } xs)$
by (induction xs rule: induct-list012)(auto)

lemma f-arg-min-list-f: $xs \neq [] \implies f (\text{arg-min-list } f \text{ } xs) = \text{Min } (f \text{ ' } (\text{set } xs))$
by(induction f xs rule: arg-min-list.induct) (auto simp: min-def intro!: antisym)

lemma arg-min-list-in: $xs \neq [] \implies \text{arg-min-list } f \text{ } xs \in \text{set } xs$
by(induction xs rule: induct-list012) (auto simp: Let-def)

67.2.17 (In)finiteness

lemma finite-maxlen:

```

  finite (M::'a list set)  $\implies \exists n. \forall s \in M. \text{size } s < n$ 
proof (induct rule: finite.induct)
  case emptyI show ?case by simp
next
  case (insertI M xs)
  then obtain n where  $\forall s \in M. \text{length } s < n$  by blast
  hence  $\forall s \in \text{insert } xs \text{ } M. \text{size } s < \max n (\text{size } xs) + 1$  by auto
  thus ?case ..
qed

```

lemma lists-length-Suc-eq:

```

{xs. set xs  $\subseteq$  A  $\wedge$  length xs = Suc n} =
  ( $\lambda(x, n). n \# xs$ ) ' ({xs. set xs  $\subseteq$  A  $\wedge$  length xs = n}  $\times$  A)

```

by (auto simp: length-Suc-conv)

lemma

assumes *finite A*

shows *finite-lists-length-eq*: *finite* $\{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\}$

and *card-lists-length-eq*: $\text{card } \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\} = (\text{card } A)^{\wedge n}$

using $\langle \text{finite } A \rangle$

by (induct *n*)

(auto simp: card-image inj-split-Cons lists-length-Suc-eq cong: conj-cong)

lemma *finite-lists-length-le*:

assumes *finite A* shows *finite* $\{xs. \text{set } xs \subseteq A \wedge \text{length } xs \leq n\}$

(is *finite ?S*)

proof –

have $?S = (\bigcup n \in \{0..n\}. \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\})$ by auto

thus *?thesis* by (auto intro!: *finite-lists-length-eq*[OF $\langle \text{finite } A \rangle$] simp only:)

qed

lemma *card-lists-length-le*:

assumes *finite A* shows $\text{card } \{xs. \text{set } xs \subseteq A \wedge \text{length } xs \leq n\} = (\sum i \leq n. \text{card } A^{\wedge i})$

proof –

have $(\sum i \leq n. \text{card } A^{\wedge i}) = \text{card } (\bigcup i \leq n. \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = i\})$

using $\langle \text{finite } A \rangle$

by (subst *card-UN-disjoint*)

(auto simp add: *card-lists-length-eq* *finite-lists-length-eq*)

also have $(\bigcup i \leq n. \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = i\}) = \{xs. \text{set } xs \subseteq A \wedge \text{length } xs \leq n\}$

by auto

finally show *?thesis* by simp

qed

lemma *finite-lists-distinct-length-eq* [intro]:

assumes *finite A*

shows *finite* $\{xs. \text{length } xs = n \wedge \text{distinct } xs \wedge \text{set } xs \subseteq A\}$ (is *finite ?S*)

proof –

have *finite* $\{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\}$

using $\langle \text{finite } A \rangle$ by (rule *finite-lists-length-eq*)

moreover have $?S \subseteq \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\}$ by auto

ultimately show *?thesis* using *finite-subset* by auto

qed

lemma *card-lists-distinct-length-eq*:

assumes *finite A* $k \leq \text{card } A$

shows $\text{card } \{xs. \text{length } xs = k \wedge \text{distinct } xs \wedge \text{set } xs \subseteq A\} = \prod \{\text{card } A - k + 1 .. \text{card } A\}$

using *assms*

proof (induct *k*)

case 0

then have $\{xs. \text{length } xs = 0 \wedge \text{distinct } xs \wedge \text{set } xs \subseteq A\} = \{\{\}\}$ **by** *auto*
then show *?case by simp*
next
case $(\text{Suc } k)$
let $?k\text{-list} = \lambda k xs. \text{length } xs = k \wedge \text{distinct } xs \wedge \text{set } xs \subseteq A$
have *inj-Cons*: $\bigwedge A. \text{inj-on } (\lambda(xs, n). n \# xs) A$ **by** *(rule inj-onI) auto*

from *Suc* **have** $k \leq \text{card } A$ **by** *simp*
moreover note $\langle \text{finite } A \rangle$
moreover have *finite* $\{xs. ?k\text{-list } k xs\}$
by *(rule finite-subset) (use finite-lists-length-eq[OF $\langle \text{finite } A \rangle$, of k] in auto)*
moreover have $\bigwedge i j. i \neq j \longrightarrow \{i\} \times (A - \text{set } i) \cap \{j\} \times (A - \text{set } j) = \{\}$
by *auto*
moreover have $\bigwedge i. i \in \{xs. ?k\text{-list } k xs\} \implies \text{card } (A - \text{set } i) = \text{card } A - k$
by *(simp add: card-Diff-subset distinct-card)*
moreover have $\{xs. ?k\text{-list } (\text{Suc } k) xs\} =$
 $(\lambda(xs, n). n \# xs) \cdot \bigcup ((\lambda xs. \{xs\} \times (A - \text{set } xs)) \cdot \{xs. ?k\text{-list } k xs\})$
by *(auto simp: length-Suc-conv)*
moreover have $\text{Suc } (\text{card } A - \text{Suc } k) = \text{card } A - k$ **using** *Suc.prem*s **by** *simp*
then have $(\text{card } A - k) * \prod \{\text{Suc } (\text{card } A - k).. \text{card } A\} = \prod \{\text{Suc } (\text{card } A - \text{Suc } k).. \text{card } A\}$
by *(subst prod.insert[symmetric]) (simp add: atLeastAtMost-insertL)+*
ultimately show *?case*
by *(simp add: card-image inj-Cons card-UN-disjoint Suc.hyps algebra-simps)*
qed

lemma *card-lists-distinct-length-eq'*:

assumes $k < \text{card } A$
shows $\text{card } \{xs. \text{length } xs = k \wedge \text{distinct } xs \wedge \text{set } xs \subseteq A\} = \prod \{\text{card } A - k + 1 .. \text{card } A\}$
proof –
from $\langle k < \text{card } A \rangle$ **have** *finite* A **and** $k \leq \text{card } A$ **using** *card.infinite* **by** *force+*
from *this* **show** *?thesis* **by** *(rule card-lists-distinct-length-eq)*
qed

lemma *infinite-UNIV-listI*: $\neg \text{finite}(\text{UNIV}::'a \text{ list set})$

by *(metis UNIV-I finite-maxlen length-replicate less-irrefl)*

lemma *same-length-different*:

assumes $xs \neq ys$ **and** $\text{length } xs = \text{length } ys$
shows $\exists \text{pre } x xs' y ys'. x \neq y \wedge xs = \text{pre } @ [x] @ xs' \wedge ys = \text{pre } @ [y] @ ys'$
using *assms*
proof *(induction xs arbitrary: ys)*
case *Nil*
then show *?case* **by** *auto*
next
case $(\text{Cons } x xs)$
then obtain $z zs$ **where** $ys = \text{Cons } z zs$
by *(metis length-Suc-conv)*

```

show ?case
proof (cases x=z)
  case True
  then have  $xs \neq zs$  length xs = length zs
    using Cons.premys ys by auto
  then obtain  $pre\ u\ xs'\ v\ ys'$  where  $u \neq v$  and  $xs: xs = pre\ @\ [u]\ @\ xs'$  and  $zs:$ 
 $zs = pre\ @\ [v]\ @\ ys'$ 
    using Cons.IH by meson
  then have  $x \# xs = (z\#\#pre)\ @\ [u]\ @\ xs' \wedge ys = (z\#\#pre)\ @\ [v]\ @\ ys'$ 
    by (simp add: True ys)
  with  $\langle u \neq v \rangle$  show ?thesis
    by blast
next
  case False
  then have  $x \# xs = []\ @\ [x]\ @\ xs \wedge ys = []\ @\ [z]\ @\ zs$ 
    by (simp add: ys)
  then show ?thesis
    using False by blast
qed
qed

```

67.3 Sorting

67.3.1 sorted-wrt

Sometimes the second equation in the definition of *sorted-wrt* is too aggressive because it relates each list element to *all* its successors. Then this equation should be removed and *sorted-wrt2-simps* should be added instead.

lemma *sorted-wrt1*: $sorted-wrt\ P\ [x] = True$
by (simp)

lemma *sorted-wrt2*: $transp\ P \implies sorted-wrt\ P\ (x \# y \# zs) = (P\ x\ y \wedge sorted-wrt\ P\ (y \# zs))$

proof (induction *zs* arbitrary: $x\ y$)

case (Cons $z\ zs$)

then show ?case

by simp (meson transpD)+

qed auto

lemmas *sorted-wrt2-simps* = *sorted-wrt1 sorted-wrt2*

lemma *sorted-wrt-true* [simp]:

$sorted-wrt\ (\lambda\ -. True)\ xs$

by (induction *xs*) simp-all

lemma *sorted-wrt-append*:

$sorted-wrt\ P\ (xs\ @\ ys) \longleftrightarrow$

$sorted-wrt\ P\ xs \wedge sorted-wrt\ P\ ys \wedge (\forall x \in set\ xs. \forall y \in set\ ys. P\ x\ y)$

by (induction *xs*) auto

lemma *sorted-wrt-map*:

sorted-wrt R (map f xs) = sorted-wrt ($\lambda x y. R (f x) (f y)$) xs
by (*induction xs*) *simp-all*

lemma

assumes *sorted-wrt f xs*

shows *sorted-wrt-take: sorted-wrt f (take n xs)*

and *sorted-wrt-drop: sorted-wrt f (drop n xs)*

proof –

from *assms* **have** *sorted-wrt f (take n xs @ drop n xs)* **by** *simp*

thus *sorted-wrt f (take n xs)* **and** *sorted-wrt f (drop n xs)*

unfolding *sorted-wrt-append* **by** *simp-all*

qed

lemma *sorted-wrt-filter*:

sorted-wrt f xs \implies sorted-wrt f (filter P xs)

by (*induction xs*) *auto*

lemma *sorted-wrt-rev*:

sorted-wrt P (rev xs) = sorted-wrt ($\lambda x y. P y x$) xs

by (*induction xs*) (*auto simp add: sorted-wrt-append*)

lemma *sorted-wrt-mono-rel*:

($\bigwedge x y. \llbracket x \in \text{set } xs; y \in \text{set } xs; P x y \rrbracket \implies Q x y$) \implies *sorted-wrt P xs \implies sorted-wrt Q xs*

by(*induction xs*)(*auto*)

lemma *sorted-wrt01: length xs \leq 1 \implies sorted-wrt P xs*

by(*auto simp: le-Suc-eq length-Suc-conv*)

lemma *sorted-wrt-iff-nth-less*:

sorted-wrt P xs = ($\forall i j. i < j \longrightarrow j < \text{length } xs \longrightarrow P (xs ! i) (xs ! j)$)

by (*induction xs*) (*auto simp add: in-set-conv-nth Ball-def nth-Cons split: nat.split*)

lemma *sorted-wrt-nth-less*:

$\llbracket \text{sorted-wrt } P \text{ xs}; i < j; j < \text{length } xs \rrbracket \implies P (xs ! i) (xs ! j)$

by(*auto simp: sorted-wrt-iff-nth-less*)

lemma *sorted-wrt-iff-nth-Suc-transp: assumes transp P*

shows *sorted-wrt P xs \longleftrightarrow ($\forall i. \text{Suc } i < \text{length } xs \longrightarrow P (xs ! i) (xs ! (\text{Suc } i))$)* (**is** *?L = ?R*)

proof

assume *?L*

thus *?R*

by (*simp add: sorted-wrt-iff-nth-less*)

next

assume *?R*

have $i < j \implies j < \text{length } xs \implies P (xs ! i) (xs ! j)$ **for** $i j$

```

  by(induct i j rule: less-Suc-induct)(simp add: ‹?R›, meson assms transpE
transp-on-less)
  thus ?L
  by (simp add: sorted-wrt-iff-nth-less)
qed

```

```

lemma sorted-wrt-upt[simp]: sorted-wrt (<) [m..n]
by(induction n) (auto simp: sorted-wrt-append)

```

```

lemma sorted-wrt-upto[simp]: sorted-wrt (<) [i..j]
proof(induct i j rule:upto.induct)
  case (1 i j)
  from this show ?case
  unfolding upto.simps[of i j] by auto
qed

```

Each element is greater or equal to its index:

```

lemma sorted-wrt-less-idx:
  sorted-wrt (<) ns  $\implies$  i < length ns  $\implies$  i  $\leq$  ns!i
proof (induction ns arbitrary: i rule: rev-induct)
  case Nil thus ?case by simp
next
  case snoc
  thus ?case
  by (simp add: nth-append sorted-wrt-append)
  (metis less-antisym not-less nth-mem)
qed

```

67.3.2 sorted

```

context linorder
begin

```

Sometimes the second equation in the definition of *sorted* is too aggressive because it relates each list element to *all* its successors. Then this equation should be removed and *sorted2-simps* should be added instead. Executable code is one such use case.

```

lemma sorted0: sorted [] = True
  by simp

```

```

lemma sorted1: sorted [x] = True
  by simp

```

```

lemma sorted2: sorted (x # y # zs) = (x  $\leq$  y  $\wedge$  sorted (y # zs))
  by auto

```

```

lemmas sorted2-simps = sorted1 sorted2

```

```

lemma sorted-append:

```

$sorted (xs@ys) = (sorted\ xs \wedge sorted\ ys \wedge (\forall x \in set\ xs. \forall y \in set\ ys. x \leq y))$
by (*simp add: sorted-wrt-append*)

lemma *sorted-map*:

$sorted (map\ f\ xs) = sorted-wrt (\lambda x\ y. f\ x \leq f\ y)\ xs$
by (*simp add: sorted-wrt-map*)

lemma *sorted01*: $length\ xs \leq 1 \implies sorted\ xs$

by (*simp add: sorted-wrt01*)

lemma *sorted-tl*:

$sorted\ xs \implies sorted (tl\ xs)$
by (*cases xs (simp-all)*)

lemma *sorted-iff-nth-mono-less*:

$sorted\ xs = (\forall i\ j. i < j \longrightarrow j < length\ xs \longrightarrow xs!\ i \leq xs!\ j)$
by (*simp add: sorted-wrt-iff-nth-less*)

lemma *sorted-iff-nth-mono*:

$sorted\ xs = (\forall i\ j. i \leq j \longrightarrow j < length\ xs \longrightarrow xs!\ i \leq xs!\ j)$
by (*auto simp: sorted-iff-nth-mono-less nat-less-le*)

lemma *sorted-nth-mono*:

$sorted\ xs \implies i \leq j \implies j < length\ xs \implies xs!\ i \leq xs!\ j$
by (*auto simp: sorted-iff-nth-mono*)

lemma *sorted-iff-nth-Suc*:

$sorted\ xs \longleftrightarrow (\forall i. Suc\ i < length\ xs \longrightarrow xs!\ i \leq xs!\ (Suc\ i))$
by (*simp add: sorted-wrt-iff-nth-Suc-transp*)

lemma *sorted-rev-nth-mono*:

$sorted (rev\ xs) \implies i \leq j \implies j < length\ xs \implies xs!\ j \leq xs!\ i$
by (*metis local.nle-le order-class.antisym-conv1 sorted-wrt-iff-nth-less sorted-wrt-rev*)

lemma *sorted-rev-iff-nth-mono*:

$sorted (rev\ xs) \longleftrightarrow (\forall i\ j. i \leq j \longrightarrow j < length\ xs \longrightarrow xs!\ j \leq xs!\ i)$ (**is** $?L = ?R$)

proof

assume $?L$ **thus** $?R$

by (*blast intro: sorted-rev-nth-mono*)

next

assume $?R$

have $rev\ xs!\ k \leq rev\ xs!\ l$ **if** *asms*: $k \leq l$ $l < length(rev\ xs)$ **for** $k\ l$

proof –

have $k < length\ xs$ $l < length\ xs$

$length\ xs - Suc\ l \leq length\ xs - Suc\ k$ $length\ xs - Suc\ k < length\ xs$

using *asms* **by** *auto*

thus $rev\ xs!\ k \leq rev\ xs!\ l$

by (*simp add: <?R> rev-nth*)

qed

thus ?L **by** (simp add: sorted-iff-nth-mono)
qed

lemma sorted-rev-iff-nth-Suc:

sorted (rev xs) \longleftrightarrow ($\forall i. \text{Suc } i < \text{length } xs \longrightarrow xs!(\text{Suc } i) \leq xs!i$)

proof –

interpret dual: linorder ($\lambda x y. y \leq x$) ($\lambda x y. y < x$)

using dual-linorder .

show ?thesis

using dual-linorder dual.sorted-iff-nth-Suc dual.sorted-iff-nth-mono

unfolding sorted-rev-iff-nth-mono **by** simp

qed

lemma sorted-map-remove1:

sorted (map f xs) \implies sorted (map f (remove1 x xs))

by (induct xs) (auto)

lemma sorted-remove1: sorted xs \implies sorted (remove1 a xs)

using sorted-map-remove1 [of $\lambda x. x$] **by** simp

lemma sorted-butlast:

assumes sorted xs

shows sorted (butlast xs)

by (simp add: assms butlast-conv-take sorted-wrt-take)

lemma sorted-replicate [simp]: sorted(replicate n x)

by(induction n) (auto)

lemma sorted-remdups[simp]:

sorted xs \implies sorted (remdups xs)

by (induct xs) (auto)

lemma sorted-remdups-adj[simp]:

sorted xs \implies sorted (remdups-adj xs)

by (induct xs rule: remdups-adj.induct, simp-all split: if-split-asm)

lemma sorted-nths: sorted xs \implies sorted (nths xs I)

by(induction xs arbitrary: I)(auto simp: nths-Cons)

lemma sorted-distinct-set-unique:

assumes sorted xs distinct xs sorted ys distinct ys set xs = set ys

shows xs = ys

proof –

from assms **have** 1: length xs = length ys **by** (auto dest!: distinct-card)

from assms **show** ?thesis

proof(induct rule:list-induct2[OF 1])

case 1 **show** ?case **by** simp

next

case (2 x xs y ys)

```

then show ?case
  by (cases ⟨x = y⟩) (auto simp add: insert-eq-iff)
qed
qed

```

```

lemma map-sorted-distinct-set-unique:
  assumes inj-on f (set xs ∪ set ys)
  assumes sorted (map f xs) distinct (map f xs)
    sorted (map f ys) distinct (map f ys)
  assumes set xs = set ys
  shows xs = ys
  using assms map-inj-on sorted-distinct-set-unique by fastforce

```

```

lemma sorted-dropWhile: sorted xs  $\implies$  sorted (dropWhile P xs)
  by (auto dest: sorted-wrt-drop simp add: dropWhile-eq-drop)

```

```

lemma sorted-takeWhile: sorted xs  $\implies$  sorted (takeWhile P xs)
  by (subst takeWhile-eq-take) (auto dest: sorted-wrt-take)

```

```

lemma sorted-filter:
  sorted (map f xs)  $\implies$  sorted (map f (filter P xs))
  by (induct xs) simp-all

```

```

lemma foldr-max-sorted:
  assumes sorted (rev xs)
  shows foldr max xs y = (if xs = [] then y else max (xs ! 0) y)
  using assms
proof (induct xs)
  case (Cons x xs)
  then have sorted (rev xs) using sorted-append by auto
  with Cons show ?case
    by (cases xs) (auto simp add: sorted-append max-def)
qed simp

```

```

lemma filter-equals-takeWhile-sorted-rev:
  assumes sorted: sorted (rev (map f xs))
  shows filter (λx. t < f x) xs = takeWhile (λ x. t < f x) xs
    (is filter ?P xs = ?tW)
proof (rule takeWhile-eq-filter[symmetric])
  let ?dW = dropWhile ?P xs
  fix x assume x: x ∈ set ?dW
  then obtain i where i: i < length ?dW and nth-i: x = ?dW ! i
    unfolding in-set-conv-nth by auto
  hence length ?tW + i < length (?tW @ ?dW)
    unfolding length-append by simp
  hence i': length (map f ?tW) + i < length (map f xs) by simp
  have (map f ?tW @ map f ?dW) ! (length (map f ?tW) + i) ≤
    (map f ?tW @ map f ?dW) ! (length (map f ?tW) + 0)
    using sorted-rev-nth-mono[OF sorted - i', of length ?tW]

```

```

  unfolding map-append[symmetric] by simp
  hence  $f x \leq f (?dW ! 0)$ 
  unfolding nth-append-length-plus nth-i
  using  $i$  preorder-class.le-less-trans[OF le0  $i$ ] by simp
  also have  $\dots \leq t$ 
  by (metis hd-conv-nth hd-dropWhile length-greater-0-conv length-pos-if-in-set
  local.leI  $x$ )
  finally show  $\neg t < f x$  by simp
qed

```

lemma *sorted-map-same*:

```

  sorted (map f (filter ( $\lambda x. f x = g xs$ ) xs))
proof (induct xs arbitrary: g)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  then have sorted (map f (filter ( $\lambda y. f y = (\lambda xs. f x) xs$ ) xs)) .
  moreover from Cons have sorted (map f (filter ( $\lambda y. f y = (g \circ Cons x) xs$ )
  xs)) .
  ultimately show ?case by simp-all
qed

```

lemma *sorted-same*:

```

  sorted (filter ( $\lambda x. x = g xs$ ) xs)
using sorted-map-same [of  $\lambda x. x$ ] by simp

```

end

lemma *sorted-upt*[simp]: sorted [$m..<n$]

```

  by (simp add: sorted-wrt-mono-rel[OF - sorted-wrt-upt])

```

lemma *sorted-upto*[simp]: sorted [$m..n$]

```

  by (simp add: sorted-wrt-mono-rel[OF - sorted-wrt-upto])

```

67.3.3 Sorting functions

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not part of Main. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context *linorder*

begin

lemma *set-insort-key*:

```

  set (insort-key f x xs) = insert x (set xs)
by (induct xs) auto

```


lemma *length-insort* [simp]:

$length (insort\text{-}key\ f\ x\ xs) = Suc (length\ xs)$

by (*induct xs*) *simp-all*

lemma *insort-key-left-comm*:

assumes $f\ x \neq f\ y$

shows $insort\text{-}key\ f\ y\ (insort\text{-}key\ f\ x\ xs) = insort\text{-}key\ f\ x\ (insort\text{-}key\ f\ y\ xs)$

by (*induct xs*) (*auto simp add: assms dest: order.antisym*)

lemma *insort-left-comm*:

$insort\ x\ (insort\ y\ xs) = insort\ y\ (insort\ x\ xs)$

by (*cases x = y*) (*auto intro: insort-key-left-comm*)

lemma *comp-fun-commute-insort*: *comp-fun-commute insort*

proof

qed (*simp add: insort-left-comm fun-eq-iff*)

lemma *sort-key-simps* [simp]:

$sort\text{-}key\ f\ [] = []$

$sort\text{-}key\ f\ (x\#\!xs) = insort\text{-}key\ f\ x\ (sort\text{-}key\ f\ xs)$

by (*simp-all add: sort-key-def*)

lemma *sort-key-conv-fold*:

assumes *inj-on f (set xs)*

shows $sort\text{-}key\ f\ xs = fold (insort\text{-}key\ f)\ xs\ []$

proof –

have $fold (insort\text{-}key\ f)\ (rev\ xs) = fold (insort\text{-}key\ f)\ xs$

proof (*rule fold-rev, rule ext*)

fix *zs*

fix *x y*

assume $x \in set\ xs\ y \in set\ xs$

with *assms* **have** $*$: $f\ y = f\ x \implies y = x$ **by** (*auto dest: inj-onD*)

have $**$: $x = y \iff y = x$ **by** *auto*

show $(insort\text{-}key\ f\ y \circ insort\text{-}key\ f\ x)\ zs = (insort\text{-}key\ f\ x \circ insort\text{-}key\ f\ y)\ zs$

by (*induct zs*) (*auto intro: * simp add: ***)

qed

then show *?thesis* **by** (*simp add: sort-key-def foldr-conv-fold*)

qed

lemma *sort-conv-fold*:

$sort\ xs = fold\ insort\ xs\ []$

by (*rule sort-key-conv-fold*) *simp*

lemma *length-sort*[simp]: $length (sort\text{-}key\ f\ xs) = length\ xs$

by (*induct xs, auto*)

lemma *set-sort*[simp]: $set(sort\text{-}key\ f\ xs) = set\ xs$

by (*induct xs*) (*simp-all add: set-insort-key*)

lemma *distinct-insort*: $\text{distinct} (\text{insort-key } f \ x \ xs) = (x \notin \text{set } xs \wedge \text{distinct } xs)$
by (*induct xs*) (*auto simp: set-insort-key*)

lemma *distinct-insort-key*:
 $\text{distinct} (\text{map } f (\text{insort-key } f \ x \ xs)) = (f \ x \notin f \text{' set } xs \wedge (\text{distinct} (\text{map } f \ xs)))$
by (*induct xs*) (*auto simp: set-insort-key*)

lemma *distinct-sort[simp]*: $\text{distinct} (\text{sort-key } f \ xs) = \text{distinct } xs$
by (*induct xs*) (*simp-all add: distinct-insort*)

lemma *sorted-insort-key*: $\text{sorted} (\text{map } f (\text{insort-key } f \ x \ xs)) = \text{sorted} (\text{map } f \ xs)$
by (*induct xs*) (*auto simp: set-insort-key*)

lemma *sorted-insort*: $\text{sorted} (\text{insort } x \ xs) = \text{sorted } xs$
using *sorted-insort-key* [**where** $f = \lambda x. x$] **by** *simp*

theorem *sorted-sort-key [simp]*: $\text{sorted} (\text{map } f (\text{sort-key } f \ xs))$
by (*induct xs*) (*auto simp: sorted-insort-key*)

theorem *sorted-sort [simp]*: $\text{sorted} (\text{sort } xs)$
using *sorted-sort-key* [**where** $f = \lambda x. x$] **by** *simp*

lemma *insort-not-Nil [simp]*:
 $\text{insort-key } f \ a \ xs \neq []$
by (*induction xs*) *simp-all*

lemma *insort-is-Cons*: $\forall x \in \text{set } xs. f \ a \leq f \ x \implies \text{insort-key } f \ a \ xs = a \# \ xs$
by (*cases xs*) *auto*

lemma *sort-key-id-if-sorted*: $\text{sorted} (\text{map } f \ xs) \implies \text{sort-key } f \ xs = xs$
by (*induction xs*) (*auto simp add: insort-is-Cons*)

Subsumed by $\text{sorted} (\text{map } ?f \ ?xs) \implies \text{sort-key } ?f \ ?xs = ?xs$ but easier to find:

lemma *sorted-sort-id*: $\text{sorted } xs \implies \text{sort } xs = xs$
by (*simp add: sort-key-id-if-sorted*)

lemma *insort-key-remove1*:
assumes $a \in \text{set } xs$ **and** $\text{sorted} (\text{map } f \ xs)$ **and** $\text{hd} (\text{filter} (\lambda x. f \ a = f \ x) \ xs) = a$
shows $\text{insort-key } f \ a (\text{remove1 } a \ xs) = xs$
using *assms* **proof** (*induct xs*)
case (*Cons x xs*)
then show *?case*
proof (*cases x = a*)
case *False*
then have $f \ x \neq f \ a$ **using** *Cons.prem1* **by** *auto*
then have $f \ x < f \ a$ **using** *Cons.prem1* **by** *auto*
with $\langle f \ x \neq f \ a \rangle$ **show** *?thesis* **using** *Cons* **by** (*auto simp: insort-is-Cons*)
qed (*auto simp: insort-is-Cons*)

qed *simp*

lemma *insort-remove1*:

assumes $a \in \text{set } xs$ and *sorted* xs

shows $\text{insort } a (\text{remove1 } a \text{ } xs) = xs$

proof (rule *insort-key-remove1*)

define n where $n = \text{length } (\text{filter } ((=) \ a) \ xs) - 1$

from $\langle a \in \text{set } xs \rangle$ show $a \in \text{set } xs$.

from $\langle \text{sorted } xs \rangle$ show *sorted* $(\text{map } (\lambda x. \ x) \ xs)$ by *simp*

from $\langle a \in \text{set } xs \rangle$ have $a \in \text{set } (\text{filter } ((=) \ a) \ xs)$ by *auto*

then have $\text{set } (\text{filter } ((=) \ a) \ xs) \neq \{\}$ by *auto*

then have $\text{filter } ((=) \ a) \ xs \neq []$ by (*auto simp only: set-empty*)

then have $\text{length } (\text{filter } ((=) \ a) \ xs) > 0$ by *simp*

then have $n: \text{Suc } n = \text{length } (\text{filter } ((=) \ a) \ xs)$ by (*simp add: n-def*)

moreover have $\text{replicate } (\text{Suc } n) \ a = a \ \# \ \text{replicate } n \ a$

by *simp*

ultimately show $\text{hd } (\text{filter } ((=) \ a) \ xs) = a$ by (*simp add: replicate-length-filter*)

qed

lemma *finite-sorted-distinct-unique*:

assumes *finite* A shows $\exists! xs. \text{set } xs = A \wedge \text{sorted } xs \wedge \text{distinct } xs$

proof –

obtain xs where *distinct* xs $A = \text{set } xs$

using *finite-distinct-list* [*OF assms*] by *metis*

then show *?thesis*

by (*rule-tac a=sort xs in ex1I*) (*auto simp: sorted-distinct-set-unique*)

qed

lemma *insort-insert-key-triv*:

$f \ x \in f \ ' \ \text{set } xs \implies \text{insort-insert-key } f \ x \ xs = xs$

by (*simp add: insort-insert-key-def*)

lemma *insort-insert-triv*:

$x \in \text{set } xs \implies \text{insort-insert } x \ xs = xs$

using *insort-insert-key-triv* [*of* $\lambda x. \ x$] by *simp*

lemma *insort-insert-insort-key*:

$f \ x \notin f \ ' \ \text{set } xs \implies \text{insort-insert-key } f \ x \ xs = \text{insort-key } f \ x \ xs$

by (*simp add: insort-insert-key-def*)

lemma *insort-insert-insort*:

$x \notin \text{set } xs \implies \text{insort-insert } x \ xs = \text{insort } x \ xs$

using *insort-insert-insort-key* [*of* $\lambda x. \ x$] by *simp*

lemma *set-insort-insert*:

$\text{set } (\text{insort-insert } x \ xs) = \text{insert } x \ (\text{set } xs)$

by (*auto simp add: insort-insert-key-def set-insort-key*)

lemma *distinct-insort-insert*:

assumes *distinct xs*
shows *distinct (insort-insert-key f x xs)*
using *assms* **by** (*induct xs*) (*auto simp add: insort-insert-key-def set-insort-key*)

lemma *sorted-insort-insert-key*:
assumes *sorted (map f xs)*
shows *sorted (map f (insort-insert-key f x xs))*
using *assms* **by** (*simp add: insort-insert-key-def sorted-insort-key*)

lemma *sorted-insort-insert*:
assumes *sorted xs*
shows *sorted (insort-insert x xs)*
using *assms* *sorted-insort-insert-key* [*of λx. x*] **by** *simp*

lemma *filter-insort-triv*:
 $\neg P x \implies \text{filter } P (\text{insort-key } f x xs) = \text{filter } P xs$
by (*induct xs*) *simp-all*

lemma *filter-insort*:
 $\text{sorted } (\text{map } f xs) \implies P x \implies \text{filter } P (\text{insort-key } f x xs) = \text{insort-key } f x (\text{filter } P xs)$
by (*induct xs*) (*auto, subst insort-is-Cons, auto*)

lemma *filter-sort*:
 $\text{filter } P (\text{sort-key } f xs) = \text{sort-key } f (\text{filter } P xs)$
by (*induct xs*) (*simp-all add: filter-insort-triv filter-insort*)

lemma *remove1-insort-key* [*simp*]:
 $\text{remove1 } x (\text{insort-key } f x xs) = xs$
by (*induct xs*) *simp-all*

end

lemma *sort-upt* [*simp*]: $\text{sort } [m..<n] = [m..<n]$
by (*rule sort-key-id-if-sorted*) *simp*

lemma *sort-upto* [*simp*]: $\text{sort } [i..j] = [i..j]$
by (*rule sort-key-id-if-sorted*) *simp*

lemma *sorted-find-Min*:
 $\text{sorted } xs \implies \exists x \in \text{set } xs. P x \implies \text{List.find } P xs = \text{Some } (\text{Min } \{x \in \text{set } xs. P x\})$
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **show** *?case* **proof** (*cases P x*)
case *True*
with *Cons* **show** *?thesis* **by** (*auto intro: Min-eqI [symmetric]*)
next
case *False* **then have** $\{y. (y = x \vee y \in \text{set } xs) \wedge P y\} = \{y \in \text{set } xs. P y\}$

```

    by auto
  with Cons False show ?thesis by (simp-all)
qed
qed

```

lemma *sorted-enumerate* [simp]: *sorted (map fst (enumerate n xs))*
by (*simp add: enumerate-eq-zip*)

lemma *sorted-insort-is-snoc*: *sorted xs $\implies \forall x \in \text{set } xs. a \geq x \implies \text{insort } a \text{ } xs = xs @ [a]$*
by (*induct xs (auto dest!: insort-is-Cons)*)

Stability of *sort-key*:

lemma *sort-key-stable*: *filter ($\lambda y. f y = k$) (sort-key f xs) = filter ($\lambda y. f y = k$) xs*
by (*induction xs (auto simp: filter-insort insort-is-Cons filter-insort-triv)*)

corollary *stable-sort-key-sort-key*: *stable-sort-key sort-key*
by(*simp add: stable-sort-key-def sort-key-stable*)

lemma *sort-key-const*: *sort-key ($\lambda x. c$) xs = xs*
by (*metis (mono-tags) filter-True sort-key-stable*)

67.3.4 transpose on sorted lists

lemma *sorted-transpose*[simp]: *sorted (rev (map length (transpose xs)))*
by (*auto simp: sorted-iff-nth-mono rev-nth nth-transpose*
length-filter-conv-card intro: card-mono)

lemma *transpose-max-length*:
foldr ($\lambda xs. \max (\text{length } xs)$) (transpose xs) 0 = length (filter ($\lambda x. x \neq []$) xs)
(is ?L = ?R)

```

proof (cases transpose xs = [])
  case False
  have ?L = foldr max (map length (transpose xs)) 0
    by (simp add: foldr-map comp-def)
  also have ... = length (transpose xs ! 0)
    using False sorted-transpose by (simp add: foldr-max-sorted)
  finally show ?thesis
    using False by (simp add: nth-transpose)
next
  case True
  hence filter ( $\lambda x. x \neq []$ ) xs = []
    by (auto intro!: filter-False simp: transpose-empty)
  thus ?thesis by (simp add: transpose-empty True)
qed

```

lemma *length-transpose-sorted*:
fixes xs :: 'a list list
assumes sorted: sorted (rev (map length xs))

```

shows length (transpose xs) = (if xs = [] then 0 else length (xs ! 0))
proof (cases xs = [])
  case False
  thus ?thesis
    using foldr-max-sorted[OF sorted] False
    unfolding length-transpose foldr-map comp-def
    by simp
qed simp

```

```

lemma nth-nth-transpose-sorted[simp]:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  and i: i < length (transpose xs)
  and j: j < length (filter (λys. i < length ys) xs)
  shows transpose xs ! i ! j = xs ! j ! i
using j filter-equals-takeWhile-sorted-rev[OF sorted, of i]
  nth-transpose[OF i] nth-map[OF j]
by (simp add: takeWhile-nth)

```

```

lemma transpose-column-length:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs)) and i < length xs
  shows length (filter (λys. i < length ys) (transpose xs)) = length (xs ! i)
proof -
  have xs ≠ [] using ⟨i < length xs⟩ by auto
  note filter-equals-takeWhile-sorted-rev[OF sorted, simp]
  { fix j assume j ≤ i
    note sorted-rev-nth-mono[OF sorted, of j i, simplified, OF this ⟨i < length xs⟩]
    } note sortedE = this[consumes 1]

  have {j. j < length (transpose xs) ∧ i < length (transpose xs ! j)}
    = {.. $\text{length } (xs ! i)$ }
  proof safe
    fix j
    assume j < length (transpose xs) and i < length (transpose xs ! j)
    with this(2) nth-transpose[OF this(1)]
    have i < length (takeWhile (λys. j < length ys) xs) by simp
    from nth-mem[OF this] takeWhile-nth[OF this]
    show j < length (xs ! i) by (auto dest: set-takeWhileD)
  next
    fix j assume j < length (xs ! i)
    thus j < length (transpose xs)
    using foldr-max-sorted[OF sorted] ⟨xs ≠ []⟩ sortedE[OF le0]
    by (auto simp: length-transpose comp-def foldr-map)

  have Suc i ≤ length (takeWhile (λys. j < length ys) xs)
    using ⟨i < length xs⟩ ⟨j < length (xs ! i)⟩ less-Suc-eq-le
    by (auto intro!: length-takeWhile-less-P-nth dest!: sortedE)
    with nth-transpose[OF j < length (transpose xs)]

```

```

  show  $i < \text{length } (\text{transpose } xs ! j)$  by simp
qed
thus ?thesis by (simp add: length-filter-conv-card)
qed

```

lemma *transpose-column:*

```

fixes  $xs :: 'a \text{ list list}$ 
assumes sorted: sorted (rev (map length xs)) and  $i < \text{length } xs$ 
shows map ( $\lambda ys. ys ! i$ ) (filter ( $\lambda ys. i < \text{length } ys$ ) (transpose xs))
  =  $xs ! i$  (is ?R = -)
proof (rule nth-equalityI)
  show length: length ?R = length (xs ! i)
    using transpose-column-length[OF assms] by simp

```

```

fix  $j$  assume  $j: j < \text{length } ?R$ 

```

```

note * = less-le-trans[OF this, unfolded length-map, OF length-filter-le]

```

```

from  $j$  have  $j\text{-less}: j < \text{length } (xs ! i)$  using length by simp

```

```

have  $i\text{-less-tW}: \text{Suc } i \leq \text{length } (\text{takeWhile } (\lambda ys. \text{Suc } j \leq \text{length } ys) xs)$ 

```

```

proof (rule length-takeWhile-less-P-nth)

```

```

  show  $\text{Suc } i \leq \text{length } xs$  using  $\langle i < \text{length } xs \rangle$  by simp

```

```

  fix  $k$  assume  $k < \text{Suc } i$ 

```

```

  hence  $k \leq i$  by auto

```

```

  with sorted-rev-nth-mono[OF sorted this]  $\langle i < \text{length } xs \rangle$ 

```

```

  have  $\text{length } (xs ! i) \leq \text{length } (xs ! k)$  by simp

```

```

  thus  $\text{Suc } j \leq \text{length } (xs ! k)$  using  $j\text{-less}$  by simp

```

```

qed

```

```

have  $i\text{-less-filter}: i < \text{length } (\text{filter } (\lambda ys. j < \text{length } ys) xs)$ 

```

```

  unfolding filter-equals-takeWhile-sorted-rev[OF sorted, of  $j$ ]

```

```

  using  $i\text{-less-tW}$  by (simp-all add: Suc-le-eq)

```

```

from  $j$  show ?R !  $j = xs ! i ! j$ 

```

```

  unfolding filter-equals-takeWhile-sorted-rev[OF sorted-transpose, of  $i$ ]

```

```

  by (simp add: takeWhile-nth nth-nth-transpose-sorted[OF sorted *  $i\text{-less-filter}$ ])

```

```

qed

```

lemma *transpose-transpose:*

```

fixes  $xs :: 'a \text{ list list}$ 

```

```

assumes sorted: sorted (rev (map length xs))

```

```

shows  $\text{transpose } (\text{transpose } xs) = \text{takeWhile } (\lambda x. x \neq []) xs$  (is ?L = ?R)

```

```

proof -

```

```

  have len: length ?L = length ?R

```

```

    unfolding length-transpose transpose-max-length

```

```

    using filter-equals-takeWhile-sorted-rev[OF sorted, of 0]

```

```

    by simp

```

```

{ fix  $i$  assume  $i < \text{length } ?R$ 

```

```

  with less-le-trans[OF - length-takeWhile-le[of - xs]]

```

```

  have  $i < \text{length } xs$  by simp

```

```

} note * = this

```

```

show ?thesis

```

by (rule nth-equalityI)
 (simp-all add: len nth-transpose transpose-column[OF sorted] * takeWhile-nth)
 qed

theorem transpose-rectangle:

assumes $xs = [] \implies n = 0$

assumes $rect: \bigwedge i. i < \text{length } xs \implies \text{length } (xs ! i) = n$

shows $\text{transpose } xs = \text{map } (\lambda i. \text{map } (\lambda j. xs ! j ! i) [0..<\text{length } xs]) [0..<n]$

(is ?trans = ?map)

proof (rule nth-equalityI)

have sorted (rev (map length xs))

by (auto simp: rev-nth rect sorted-iff-nth-mono)

from foldr-max-sorted[OF this] assms

show len: $\text{length } ?trans = \text{length } ?map$

by (simp-all add: length-transpose foldr-map comp-def)

moreover

{ fix i assume $i < n$ hence $\text{filter } (\lambda ys. i < \text{length } ys) xs = xs$

using rect by (auto simp: in-set-conv-nth intro!: filter-True) }

ultimately show $\bigwedge i. i < \text{length } (\text{transpose } xs) \implies ?trans ! i = ?map ! i$

by (auto simp: nth-transpose intro: nth-equalityI)

qed

67.3.5 sorted-key-list-of-set

This function maps (finite) linearly ordered sets to sorted lists. The linear order is obtained by a key function that maps the elements of the set to a type that is linearly ordered. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

Note: this is a generalisation of the older *sorted-list-of-set* that is obtained by setting the key function to the identity. Consequently, new theorems should be added to the locale below. They should also be aliased to more convenient names for use with *sorted-list-of-set* as seen further below.

definition (in *linorder*) *sorted-key-list-of-set* :: $('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ list}$
 where $\text{sorted-key-list-of-set } f \equiv \text{folding-on.F } (\text{insort-key } f) []$

locale *folding-insort-key* = *lo?*: *linorder* *less-eq* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ *less*

for *less-eq* (infix \preceq 50) and *less* (infix $<$ 50) +

fixes *S*

fixes *f* :: $'b \Rightarrow 'a$

assumes *inj-on*: *inj-on* *f* *S*

begin

lemma *insort-key-commute*:

$x \in S \implies y \in S \implies \text{insort-key } f y o \text{insort-key } f x = \text{insort-key } f x o \text{insort-key } f y$

proof(rule *ext*, *goal-cases*)

case (1 *xs*)

with *inj-on* **show** *?case* **by** (*induction xs*) (*auto simp: inj-onD*)
qed

sublocale *fold-insort-key: folding-on S insort-key f* []
rewrites *folding-on.F (insort-key f) [] = sorted-key-list-of-set f*
proof –
show *folding-on S (insort-key f)*
by *standard (simp add: insort-key-commute)*
qed (*simp add: sorted-key-list-of-set-def*)

lemma *idem-if-sorted-distinct:*
assumes *set xs ⊆ S* **and** *sorted (map f xs) distinct xs*
shows *sorted-key-list-of-set f (set xs) = xs*
proof(*cases S = {}*)
case *True*
then show *?thesis* **using** *⟨set xs ⊆ S⟩* **by** *auto*
next
case *False*
with *assms* **show** *?thesis*
proof(*induction xs*)
case (*Cons a xs*)
with *Cons* **show** *?case* **by** (*cases xs*) *auto*
qed *simp*
qed

lemma *sorted-key-list-of-set-empty:*
sorted-key-list-of-set f {} = []
by (*fact fold-insort-key.empty*)

lemma *sorted-key-list-of-set-insert:*
assumes *insert x A ⊆ S* **and** *finite A x ∉ A*
shows *sorted-key-list-of-set f (insert x A)*
 $= \text{insort-key } f \ x \ (\text{sorted-key-list-of-set } f \ A)$
using *assms* **by** (*fact fold-insort-key.insert*)

lemma *sorted-key-list-of-set-insert-remove [simp]:*
assumes *insert x A ⊆ S* **and** *finite A*
shows *sorted-key-list-of-set f (insert x A)*
 $= \text{insort-key } f \ x \ (\text{sorted-key-list-of-set } f \ (A - \{x\}))$
using *assms* **by** (*fact fold-insort-key.insert-remove*)

lemma *sorted-key-list-of-set-eq-Nil-iff [simp]:*
assumes *A ⊆ S* **and** *finite A*
shows *sorted-key-list-of-set f A = []* \longleftrightarrow *A = {}*
using *assms* **by** (*auto simp: fold-insort-key.remove*)

lemma *set-sorted-key-list-of-set [simp]:*
assumes *A ⊆ S* **and** *finite A*
shows *set (sorted-key-list-of-set f A) = A*

using *assms*(2,1)
by (*induct A rule: finite-induct*) (*simp-all add: set-insort-key*)

lemma *sorted-sorted-key-list-of-set* [*simp*]:
assumes $A \subseteq S$
shows *sorted* (*map f (sorted-key-list-of-set f A)*)
proof (*cases finite A*)
case *True* **thus** ?*thesis* **using** $\langle A \subseteq S \rangle$
by (*induction A*) (*simp-all add: sorted-insort-key*)
next
case *False* **thus** ?*thesis* **by** *simp*
qed

lemma *distinct-if-distinct-map*: $\text{distinct} (\text{map } f \text{ } xs) \implies \text{distinct } xs$
using *inj-on* **by** (*simp add: distinct-map*)

lemma *distinct-sorted-key-list-of-set* [*simp*]:
assumes $A \subseteq S$
shows *distinct* (*map f (sorted-key-list-of-set f A)*)
proof (*cases finite A*)
case *True* **thus** ?*thesis* **using** $\langle A \subseteq S \rangle$ *inj-on*
by (*induction A*) (*force simp: distinct-insort-key dest: inj-onD*)
next
case *False* **thus** ?*thesis* **by** *simp*
qed

lemma *length-sorted-key-list-of-set* [*simp*]:
assumes $A \subseteq S$
shows $\text{length} (\text{sorted-key-list-of-set } f \text{ } A) = \text{card } A$
proof (*cases finite A*)
case *True*
with *assms inj-on* **show** ?*thesis*
using *distinct-card[symmetric, OF distinct-sorted-key-list-of-set]*
by (*auto simp: subset-inj-on intro!: card-image*)
qed *auto*

lemmas *sorted-key-list-of-set* =
set-sorted-key-list-of-set sorted-sorted-key-list-of-set distinct-sorted-key-list-of-set

lemma *sorted-key-list-of-set-remove*:
assumes $\text{insert } x \text{ } A \subseteq S$ **and** *finite A*
shows $\text{sorted-key-list-of-set } f \text{ } (A - \{x\}) = \text{remove1 } x \text{ } (\text{sorted-key-list-of-set } f \text{ } A)$
proof (*cases x ∈ A*)
case *False* **with** *assms* **have** $x \notin \text{set} (\text{sorted-key-list-of-set } f \text{ } A)$ **by** *simp*
with *False* **show** ?*thesis* **by** (*simp add: remove1-idem*)
next
case *True* **then obtain** *B* **where** $A = \text{insert } x \text{ } B$ **by** (*rule Set.set-insert*)
with *assms* **show** ?*thesis* **by** *simp*
qed

lemma *strict-sorted-key-list-of-set* [simp]:

$A \subseteq S \implies \text{sorted-wrt } (<) (\text{map } f (\text{sorted-key-list-of-set } f A))$

by (cases finite A) (auto simp: strict-sorted-iff subset-inj-on[OF inj-on])

lemma *finite-set-strict-sorted*:

assumes $A \subseteq S$ **and** finite A

obtains l **where** sorted-wrt (<) (map f l) set l = A length l = card A

using assms

by (meson length-sorted-key-list-of-set set-sorted-key-list-of-set strict-sorted-key-list-of-set)

lemma (in linorder) *strict-sorted-equal*:

assumes sorted-wrt (<) xs

and sorted-wrt (<) ys

and set ys = set xs

shows ys = xs

using assms

proof (induction xs arbitrary: ys)

case (Cons x xs)

show ?case

proof (cases ys)

case Nil

then show ?thesis

using Cons.prem1 **by** auto

next

case (Cons y ys')

then have xs = ys'

by (metis Cons.prem1 list.inject sorted-distinct-set-unique strict-sorted-iff)

moreover have x = y

using Cons.prem1 <xs = ys'> local.Cons **by** fastforce

ultimately show ?thesis

using local.Cons **by** blast

qed

qed auto

lemma (in linorder) *strict-sorted-equal-Uniq*: $\exists \leq_1 xs. \text{sorted-wrt } (<) xs \wedge \text{set } xs = A$

by (simp add: Uniq-def strict-sorted-equal)

lemma *sorted-key-list-of-set-inject*:

assumes $A \subseteq S$ $B \subseteq S$

assumes sorted-key-list-of-set f A = sorted-key-list-of-set f B finite A finite B

shows A = B

using assms set-sorted-key-list-of-set **by** metis

lemma *sorted-key-list-of-set-unique*:

assumes $A \subseteq S$ **and** finite A

shows sorted-wrt (<) (map f l) \wedge set l = A \wedge length l = card A

\longleftrightarrow sorted-key-list-of-set f A = l

```

using assms
by (auto simp: strict-sorted-iff card-distinct idem-if-sorted-distinct)

end

```

```

context linorder
begin

```

```

definition sorted-list-of-set  $\equiv$  sorted-key-list-of-set ( $\lambda x. 'a. x$ )

```

We abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the key function to the identity.

```

sublocale sorted-list-of-set: folding-insort-key ( $\leq$ ) ( $<$ ) UNIV ( $\lambda x. x$ )
rewrites sorted-key-list-of-set ( $\lambda x. x$ ) = sorted-list-of-set
and  $\bigwedge xs. \text{map } (\lambda x. x) xs \equiv xs$ 
and  $\bigwedge X. (X \subseteq \text{UNIV}) \equiv \text{True}$ 
and  $\bigwedge x. x \in \text{UNIV} \equiv \text{True}$ 
and  $\bigwedge P. (\text{True} \implies P) \equiv \text{Trueprop } P$ 
and  $\bigwedge P Q. (\text{True} \implies \text{PROP } P \implies \text{PROP } Q) \equiv (\text{PROP } P \implies \text{True} \implies$ 
PROP Q)

```

```

proof –

```

```

show folding-insort-key ( $\leq$ ) ( $<$ ) UNIV ( $\lambda x. x$ )
by standard simp
qed (simp-all add: sorted-list-of-set-def)

```

Alias theorems for backwards compatibility and ease of use.

```

lemmas sorted-list-of-set = sorted-list-of-set.sorted-key-list-of-set and
sorted-list-of-set-empty = sorted-list-of-set.sorted-key-list-of-set-empty and
sorted-list-of-set-insert = sorted-list-of-set.sorted-key-list-of-set-insert and
sorted-list-of-set-insert-remove = sorted-list-of-set.sorted-key-list-of-set-insert-remove
and
sorted-list-of-set-eq-Nil-iff = sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff
and
set-sorted-list-of-set = sorted-list-of-set.set-sorted-key-list-of-set and
sorted-sorted-list-of-set = sorted-list-of-set.sorted-sorted-key-list-of-set and
distinct-sorted-list-of-set = sorted-list-of-set.distinct-sorted-key-list-of-set and
length-sorted-list-of-set = sorted-list-of-set.length-sorted-key-list-of-set and
sorted-list-of-set-remove = sorted-list-of-set.sorted-key-list-of-set-remove and
strict-sorted-list-of-set = sorted-list-of-set.strict-sorted-key-list-of-set and
sorted-list-of-set-inject = sorted-list-of-set.sorted-key-list-of-set-inject and
sorted-list-of-set-unique = sorted-list-of-set.sorted-key-list-of-set-unique and
finite-set-strict-sorted = sorted-list-of-set.finite-set-strict-sorted

```

```

lemma sorted-list-of-set-sort-remdups [code]:
sorted-list-of-set (set xs) = sort (remdups xs)

```

```

proof –

```

```

interpret comp-fun-commute insort by (fact comp-fun-commute-insort)
show ?thesis
by (simp add: sorted-list-of-set.fold-insort-key.eq-fold sort-conv-fold fold-set-fold-remdups)

```

qed

end

lemma *sorted-list-of-set-range* [simp]:
 $sorted_list_of_set \{m..<n\} = [m..<n]$
by (rule *sorted-distinct-set-unique*) *simp-all*

lemma *sorted-list-of-set-lessThan-Suc* [simp]:
 $sorted_list_of_set \{..<Suc\ k\} = sorted_list_of_set \{..<k\} @ [k]$
using *le0 lessThan-atLeast0 sorted-list-of-set-range upt-Suc-append* **by** *presburger*

lemma *sorted-list-of-set-atMost-Suc* [simp]:
 $sorted_list_of_set \{..Suc\ k\} = sorted_list_of_set \{..k\} @ [Suc\ k]$
using *lessThan-Suc-atMost sorted-list-of-set-lessThan-Suc* **by** *fastforce*

lemma *sorted-list-of-set-nonempty*:
assumes *finite A A ≠ {}*
shows $sorted_list_of_set\ A = Min\ A \# sorted_list_of_set\ (A - \{Min\ A\})$
using *assms*
by (*auto simp: less-le simp flip: sorted-list-of-set.sorted-key-list-of-set-unique intro: Min-in*)

lemma *sorted-list-of-set-greaterThanLessThan*:
assumes $Suc\ i < j$
shows $sorted_list_of_set \{i<..
proof –
have $\{i<..
using *assms* **by** *auto*
then show *?thesis*
by (*metis assms atLeastSucLessThan-greaterThanLessThan sorted-list-of-set-range upt-conv-Cons*)
qed$$

lemma *sorted-list-of-set-greaterThanAtMost*:
assumes $Suc\ i \leq j$
shows $sorted_list_of_set \{i<..
using *sorted-list-of-set-greaterThanLessThan* [of *i Suc j*]
by (*metis assms greaterThanAtMost-def greaterThanLessThan-eq le-imp-less-Suc lessThan-Suc-atMost*)$

lemma *nth-sorted-list-of-set-greaterThanLessThan*:
 $n < j - Suc\ i \implies sorted_list_of_set \{i<..
by (*induction n arbitrary: i*) (*auto simp: sorted-list-of-set-greaterThanLessThan*)$

lemma *nth-sorted-list-of-set-greaterThanAtMost*:
 $n < j - i \implies sorted_list_of_set \{i<..
using *nth-sorted-list-of-set-greaterThanLessThan* [of *n Suc j i*]$

by (simp add: greaterThanAtMost-def greaterThanLessThan-eq lessThan-Suc-atMost)

67.3.6 lists: the list-forming operator over sets

inductive-set

lists :: 'a set => 'a list set
for A :: 'a set

where

Nil [intro!, simp]: [] ∈ lists A
| Cons [intro!, simp]: [a ∈ A; l ∈ lists A] ⇒ a#l ∈ lists A

inductive-cases listsE [elim!]: x#l ∈ lists A

inductive-cases listspE [elim!]: listsp A (x # l)

inductive-simps listsp-simps[code]:

listsp A []
listsp A (x # xs)

lemma listsp-mono [mono]: A ≤ B ⇒ listsp A ≤ listsp B

by (rule predicate1I, erule listsp.induct, blast+)

lemmas lists-mono = listsp-mono [to-set]

lemma listsp-infI:

assumes l: listsp A l shows listsp B l ⇒ listsp (inf A B) l using l
by induct blast+

lemmas lists-IntI = listsp-infI [to-set]

lemma listsp-inf-eq [simp]: listsp (inf A B) = inf (listsp A) (listsp B)

proof (rule mono-inf [where f=listsp, THEN order-antisym])

show mono listsp by (simp add: mono-def listsp-mono)

show inf (listsp A) (listsp B) ≤ listsp (inf A B) by (blast intro!: listsp-infI)
qed

lemmas listsp-conj-eq [simp] = listsp-inf-eq [simplified inf-fun-def inf-bool-def]

lemmas lists-Int-eq [simp] = listsp-inf-eq [to-set]

lemma Cons-in-lists-iff [simp]: x#xs ∈ lists A ↔ x ∈ A ∧ xs ∈ lists A

by auto

lemma append-in-listsp-conv [iff]:

(listsp A (xs @ ys)) = (listsp A xs ∧ listsp A ys)

by (induct xs) auto

lemmas append-in-lists-conv [iff] = append-in-listsp-conv [to-set]

lemma in-listsp-conv-set: (listsp A xs) = (∀ x ∈ set xs. A x)

— eliminate *listsp* in favour of *set*
by (*induct xs*) *auto*

lemmas *in-lists-conv-set* [*code-unfold*] = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest!*]: *listsp A xs* $\implies \forall x \in \text{set } xs. A x$
by (*rule in-listsp-conv-set* [*THEN iffD1*])

lemmas *in-listsD* [*dest!*] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro!*]: $\forall x \in \text{set } xs. A x \implies \text{listsp } A xs$
by (*rule in-listsp-conv-set* [*THEN iffD2*])

lemmas *in-listsI* [*intro!*] = *in-listspI* [*to-set*]

lemma *lists-eq-set*: *lists A* = {*xs. set xs* \leq *A*}
by *auto*

lemma *lists-empty* [*simp*]: *lists {}* = {[]}
by *auto*

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
by *auto*

lemma *lists-image*: *lists (f'A)* = *map f' lists A*

proof —

{ **fix** *xs* **have** $\forall x \in \text{set } xs. x \in f' A \implies x \in \text{map } f' \text{ lists } A$
by (*induct xs*) (*auto simp del: list.map simp add: list.map[symmetric] intro!*
imageI) }

then show *?thesis* **by** *auto*

qed

67.3.7 Inductive definition for membership

inductive *ListMem* :: 'a \Rightarrow 'a *list* \Rightarrow *bool*

where

elem: *ListMem x (x # xs)*
| *insert*: *ListMem x xs* \implies *ListMem x (y # xs)*

lemma *ListMem-iff*: (*ListMem x xs*) = (*x* \in *set xs*)

proof

show *ListMem x xs* \implies *x* \in *set xs*

by (*induct set: ListMem*) *auto*

show *x* \in *set xs* \implies *ListMem x xs*

by (*induct xs*) (*auto intro: ListMem.intros*)

qed

67.3.8 Lists as Cartesian products

set-Cons A XS : the set of lists with head drawn from A and tail drawn from XS .

definition *set-Cons* :: 'a set \Rightarrow 'a list set \Rightarrow 'a list set **where**
set-Cons A $XS = \{z. \exists x xs. z = x \# xs \wedge x \in A \wedge xs \in XS\}$

lemma *set-Cons-sing-Nil* [*simp*]: *set-Cons* A $\{\}\} = (\%x. [x])'A$
by (*auto simp add: set-Cons-def*)

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

primrec *listset* :: 'a set list \Rightarrow 'a list set **where**
listset $\{\} = \{\}\} |$
listset ($A \# As$) = *set-Cons* A (*listset* As)

67.4 Relations on Lists

67.4.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

primrec — The lexicographic ordering for lists of the specified length
lexn :: ('a \times 'a) set \Rightarrow nat \Rightarrow ('a list \times 'a list) set **where**
lexn r $0 = \{\}$ |
lexn r (*Suc* n) =
 (*map-prod* ($\%(x, xs). x \# xs$) ($\%(x, xs). x \# xs$) ' ($r < *lex* >$ *lexn* r n)) *Int*
 $\{(xs, ys). \text{length } xs = \text{Suc } n \wedge \text{length } ys = \text{Suc } n\}$

definition *lex* :: ('a \times 'a) set \Rightarrow ('a list \times 'a list) set **where**
lex $r = (\bigcup n. \text{lexn } r$ $n)$ — Holds only between lists of the same length

definition *lenlex* :: ('a \times 'a) set \Rightarrow ('a list \times 'a list) set **where**
lenlex $r = \text{inv-image}$ (*less-than* $< *lex* >$ *lex* r) ($\lambda xs. (\text{length } xs, xs)$)
 — Compares lists by their length and then lexicographically

lemma *wf-lexn*: **assumes** *wf* r **shows** *wf* (*lexn* r n)

proof (*induct* n)

case (*Suc* n)

have *inj*: *inj* ($\lambda(x, xs). x \# xs$)

using *assms* **by** (*auto simp: inj-on-def*)

have *wf*: *wf* (*map-prod* ($\lambda(x, xs). x \# xs$) ($\lambda(x, xs). x \# xs$) ' ($r < *lex* >$ *lexn* r n))

by (*simp add: Suc.hyps assms wf-lex-prod wf-map-prod-image* [*OF - inj*])

then show ?*case*

by (*rule wf-subset*) *auto*

qed *auto*

lemma *lexn-length*:

$(xs, ys) \in \text{lexn } r \ n \implies \text{length } xs = n \wedge \text{length } ys = n$

by (*induct n arbitrary: xs ys*) *auto*

lemma *wf-lex* [*intro!*]:

assumes *wf r* **shows** *wf (lex r)*

unfolding *lex-def*

proof (*rule wf-UN*)

show *wf (lexn r i)* **for** *i*

by (*simp add: asms wf-lexn*)

show $\bigwedge i j. \text{lexn } r \ i \neq \text{lexn } r \ j \implies \text{Domain } (\text{lexn } r \ i) \cap \text{Range } (\text{lexn } r \ j) = \{\}$

by (*metis DomainE Int-emptyI RangeE lexn-length*)

qed

lemma *lexn-conv*:

lexn r n =

$\{(xs, ys). \text{length } xs = n \wedge \text{length } ys = n \wedge$

$(\exists xys \ x \ y \ xs' \ ys'. xs = xys @ x \# xs' \wedge ys = xys @ y \# ys' \wedge (x, y) \in r)\}$

proof (*induction n*)

case (*Suc n*)

then show *?case*

apply (*simp add: image-Collect lex-prod-def, safe, blast*)

apply (*rule-tac x = ab # xys in exI, simp*)

apply (*case-tac xys; force*)

done

qed *auto*

By Mathias Fleury:

proposition *lexn-transI*:

assumes *trans r* **shows** *trans (lexn r n)*

unfolding *trans-def*

proof (*intro allI impI*)

fix *as bs cs*

assume *asbs: (as, bs) ∈ lexn r n* **and** *bcs: (bs, cs) ∈ lexn r n*

obtain *abs a b as' bs'* **where**

n: length as = n **and** *length bs = n* **and**

as: as = abs @ a # as' **and**

bs: bs = abs @ b # bs' **and**

abr: (a, b) ∈ r

using *asbs* **unfolding** *lexn-conv* **by** *blast*

obtain *bcs b' c' cs' bs'* **where**

n': length cs = n **and** *length bs = n* **and**

bs': bs = bcs @ b' # bs' **and**

cs: cs = bcs @ c' # cs' **and**

b'c'r: (b', c') ∈ r

using *bcs* **unfolding** *lexn-conv* **by** *blast*

consider (*le*) *length bcs < length abs*

| (*eq*) *length bcs = length abs*

| (*ge*) *length bcs > length abs* **by** *linarith*

thus $(as, cs) \in \text{lexn } r \ n$
proof cases
 let $?k = \text{length } bcs$
 case *le*
hence $as ! ?k = bs ! ?k$ **unfolding** $as \ bs$ **by** $(\text{simp add: nth-append})$
hence $(as ! ?k, cs ! ?k) \in r$ **using** $b'c'r$ **unfolding** $bs' \ cs$ **by** *auto*
moreover
have $\text{length } bcs < \text{length } as$ **using** *le* **unfolding** as **by** *simp*
from id-take-nth-drop [*OF this*]
have $as = \text{take } ?k \ as \ @ \ as ! ?k \ \# \ \text{drop } (\text{Suc } ?k) \ as$.
moreover
have $\text{length } bcs < \text{length } cs$ **unfolding** cs **by** *simp*
from id-take-nth-drop [*OF this*]
have $cs = \text{take } ?k \ cs \ @ \ cs ! ?k \ \# \ \text{drop } (\text{Suc } ?k) \ cs$.
moreover **have** $\text{take } ?k \ as = \text{take } ?k \ cs$
 using *le arg-cong* [*OF bs, of take (length bcs)*]
 unfolding $cs \ as \ bs'$ **by** *auto*
ultimately show *?thesis* **using** $n \ n'$ **unfolding** *lexn-conv* **by** *auto*
next
 let $?k = \text{length } abs$
 case *ge*
hence $bs ! ?k = cs ! ?k$ **unfolding** $bs' \ cs$ **by** $(\text{simp add: nth-append})$
hence $(as ! ?k, cs ! ?k) \in r$ **using** *abr* **unfolding** $as \ bs$ **by** *auto*
moreover
have $\text{length } abs < \text{length } as$ **using** *ge* **unfolding** as **by** *simp*
from id-take-nth-drop [*OF this*]
have $as = \text{take } ?k \ as \ @ \ as ! ?k \ \# \ \text{drop } (\text{Suc } ?k) \ as$.
moreover **have** $\text{length } abs < \text{length } cs$ **using** $n \ n'$ **unfolding** as **by** *simp*
from id-take-nth-drop [*OF this*]
have $cs = \text{take } ?k \ cs \ @ \ cs ! ?k \ \# \ \text{drop } (\text{Suc } ?k) \ cs$.
moreover **have** $\text{take } ?k \ as = \text{take } ?k \ cs$
 using *ge arg-cong* [*OF bs', of take (length abs)*]
 unfolding $cs \ as \ bs$ **by** *auto*
ultimately show *?thesis* **using** $n \ n'$ **unfolding** *lexn-conv* **by** *auto*
next
 let $?k = \text{length } abs$
 case *eq*
hence $*: abs = bcs \ b = b'$ **using** $bs \ bs'$ **by** *auto*
hence $(a, c') \in r$
 using *abr b'c'r assms* **unfolding** *trans-def* **by** *blast*
with $*$ **show** *?thesis* **using** $n \ n'$ **unfolding** *lexn-conv* $as \ bs \ cs$ **by** *auto*
qed
qed

corollary *lex-transI*:

assumes *trans r* **shows** $\text{trans } (\text{lex } r)$
using *lexn-transI* [*OF assms*]
by $(\text{clarsimp simp add: lex-def trans-def})$ $(\text{metis lexn-length})$

lemma *lex-conv*:

lex r =
 $\{(xs,ys). \text{length } xs = \text{length } ys \wedge$
 $(\exists xys\ x\ y\ xs'\ ys'. xs = xys @ x \# xs' \wedge ys = xys @ y \# ys' \wedge (x, y) \in r)\}$
by (*force simp add: lex-def lexn-conv*)

lemma *wf-lenlex* [*intro!*]: $wf\ r \implies wf\ (\text{lenlex } r)$

by (*unfold lenlex-def blast*)

lemma *lenlex-conv*:

$\text{lenlex } r = \{(xs,ys). \text{length } xs < \text{length } ys \vee$
 $\text{length } xs = \text{length } ys \wedge (xs, ys) \in \text{lex } r\}$
by (*auto simp add: lenlex-def Id-on-def lex-prod-def inv-image-def*)

lemma *total-lenlex*:

assumes *total r*
shows *total (lenlex r)*

proof –

have $(xs,ys) \in \text{lexn } r \text{ (length } xs) \vee (ys,xs) \in \text{lexn } r \text{ (length } xs)$
if $xs \neq ys$ **and** *len: length xs = length ys* **for** $xs\ ys$

proof –

obtain $pre\ x\ xs'\ y\ ys'$ **where** $x \neq y$ **and** $xs: xs = pre @ [x] @ xs'$ **and** $ys: ys =$
 $pre @ [y] @ ys'$

by (*meson len <xs ≠ ys> same-length-different*)

then consider $(x,y) \in r \mid (y,x) \in r$

by (*meson UNIV-I assms total-on-def*)

then show *?thesis*

by cases (*use len in <(force simp add: lexn-conv xs ys)+>*)

qed

then show *?thesis*

by (*fastforce simp: lenlex-def total-on-def lex-def*)

qed

lemma *lenlex-transI* [*intro*]: $\text{trans } r \implies \text{trans } (\text{lenlex } r)$

unfolding *lenlex-def*

by (*meson lex-transI trans-inv-image trans-less-than trans-lex-prod*)

lemma *Nil-notin-lex* [*iff*]: $([], ys) \notin \text{lex } r$

by (*simp add: lex-conv*)

lemma *Nil2-notin-lex* [*iff*]: $(xs, []) \notin \text{lex } r$

by (*simp add: lex-conv*)

lemma *Cons-in-lex* [*simp*]:

$(x \# xs, y \# ys) \in \text{lex } r \iff (x, y) \in r \wedge \text{length } xs = \text{length } ys \vee x = y \wedge (xs,$
 $ys) \in \text{lex } r$

(is *?lhs = ?rhs*)

proof

assume *?lhs* **then show** *?rhs*

by (*simp add: lex-conv*) (*metis hd-append list.sel(1) list.sel(3) tl-append2*)
next
 assume *?rhs then show ?lhs*
 by (*simp add: lex-conv*) (*blast intro: Cons-eq-appendI*)
qed

lemma Nil-lenlex-iff1 [*simp*]: $([], ns) \in \text{lenlex } r \longleftrightarrow ns \neq []$
and Nil-lenlex-iff2 [*simp*]: $(ns, []) \notin \text{lenlex } r$
 by (*auto simp: lenlex-def*)

lemma Cons-lenlex-iff:
 $((m \# ms, n \# ns) \in \text{lenlex } r) \longleftrightarrow$
 $\text{length } ms < \text{length } ns$
 $\vee \text{length } ms = \text{length } ns \wedge (m, n) \in r$
 $\vee (m = n \wedge (ms, ns) \in \text{lenlex } r)$
 by (*auto simp: lenlex-def*)

lemma lenlex-irreflexive: $(\bigwedge x. (x, x) \notin r) \implies (xs, xs) \notin \text{lenlex } r$
 by (*induction xs*) (*auto simp add: Cons-lenlex-iff*)

lemma lenlex-trans:
 $\llbracket (x, y) \in \text{lenlex } r; (y, z) \in \text{lenlex } r; \text{trans } r \rrbracket \implies (x, z) \in \text{lenlex } r$
 by (*meson lenlex-transI transD*)

lemma lenlex-length: $(ms, ns) \in \text{lenlex } r \implies \text{length } ms \leq \text{length } ns$
 by (*auto simp: lenlex-def*)

lemma lex-append-rightI:
 $(xs, ys) \in \text{lex } r \implies \text{length } vs = \text{length } us \implies (xs @ us, ys @ vs) \in \text{lex } r$
 by (*fastforce simp: lex-def lexn-conv*)

lemma lex-append-leftI:
 $(ys, zs) \in \text{lex } r \implies (xs @ ys, xs @ zs) \in \text{lex } r$
 by (*induct xs*) *auto*

lemma lex-append-leftD:
 $\forall x. (x, x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \implies (ys, zs) \in \text{lex } r$
 by (*induct xs*) *auto*

lemma lex-append-left-iff:
 $\forall x. (x, x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \longleftrightarrow (ys, zs) \in \text{lex } r$
 by(*metis lex-append-leftD lex-append-leftI*)

lemma lex-take-index:
 assumes $(xs, ys) \in \text{lex } r$
 obtains *i where $i < \text{length } xs$ and $i < \text{length } ys$ and $\text{take } i \text{ } xs = \text{take } i \text{ } ys$*
 and $(xs ! i, ys ! i) \in r$

proof –
 obtain $n \text{ us } x \text{ xs}' \text{ y } \text{ ys}'$ where $(xs, ys) \in \text{lexn } r \text{ } n$ and $\text{length } xs = n$ and length

$ys = n$
and $xs = us @ x \# xs'$ **and** $ys = us @ y \# ys'$ **and** $(x, y) \in r$
using *assms* **by** (*fastforce simp: lex-def lexn-conv*)
then show *?thesis* **by** (*intro that [of length us]*) *auto*
qed

lemma *irrefl-lex*: $irrefl\ r \implies irrefl\ (lex\ r)$
by (*meson irrefl-def lex-take-index*)

lemma *lexl-not-refl [simp]*: $irrefl\ r \implies (x, x) \notin lex\ r$
by (*meson irrefl-def lex-take-index*)

67.4.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" < "ab" < "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

definition *lexord* :: ('a × 'a) set ⇒ ('a list × 'a list) set **where**
 $lexord\ r = \{(x, y). \exists a\ v.\ y = x @ a \# v \vee$
 $(\exists u\ a\ b\ v\ w.\ (a, b) \in r \wedge x = u @ (a \# v) \wedge y = u @ (b \# w))\}$

lemma *lexord-Nil-left[simp]*: $([], y) \in lexord\ r = (\exists a\ x.\ y = a \# x)$
by (*unfold lexord-def, induct-tac y, auto*)

lemma *lexord-Nil-right[simp]*: $(x, []) \notin lexord\ r$
by (*unfold lexord-def, induct-tac x, auto*)

lemma *lexord-cons-cons[simp]*:
 $(a \# x, b \# y) \in lexord\ r \longleftrightarrow (a, b) \in r \vee (a = b \wedge (x, y) \in lexord\ r)$ (**is** *?lhs = ?rhs*)

proof
assume *?lhs*
then show *?rhs*
apply (*simp add: lexord-def*)
apply (*metis hd-append list.sel(1) list.sel(3) tl-append2*)
done
qed (*auto simp add: lexord-def; (blast | meson Cons-eq-appendI)*)

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-same-pref-iff*:
 $(xs @ ys, xs @ zs) \in lexord\ r \longleftrightarrow (\exists x \in set\ xs.\ (x, x) \in r) \vee (ys, zs) \in lexord\ r$
by(*induction xs*) *auto*

lemma *lexord-same-pref-if-irrefl[simp]*:
 $irrefl\ r \implies (xs @ ys, xs @ zs) \in lexord\ r \longleftrightarrow (ys, zs) \in lexord\ r$
by (*simp add: irrefl-def lexord-same-pref-iff*)

lemma *lexord-append-rightI*: $\exists b\ z.\ y = b \# z \implies (x, x @ y) \in lexord\ r$
by (*metis append-Nil2 lexord-Nil-left lexord-same-pref-iff*)

lemma *lexord-append-left-rightI*:

$(a, b) \in r \implies (u @ a \# x, u @ b \# y) \in \text{lexord } r$

by (*simp add: lexord-same-pref-iff*)

lemma *lexord-append-leftI*: $(u, v) \in \text{lexord } r \implies (x @ u, x @ v) \in \text{lexord } r$

by (*simp add: lexord-same-pref-iff*)

lemma *lexord-append-leftD*:

$\llbracket (x @ u, x @ v) \in \text{lexord } r; (\forall a. (a, a) \notin r) \rrbracket \implies (u, v) \in \text{lexord } r$

by (*simp add: lexord-same-pref-iff*)

lemma *lexord-take-index-conv*:

$((x, y) \in \text{lexord } r) =$

$((\text{length } x < \text{length } y \wedge \text{take } (\text{length } x) \ y = x) \vee$

$(\exists i. i < \min(\text{length } x)(\text{length } y) \wedge \text{take } i \ x = \text{take } i \ y \wedge (x!i, y!i) \in r))$

proof –

have $(\exists a \ v. y = x @ a \# v) = (\text{length } x < \text{length } y \wedge \text{take } (\text{length } x) \ y = x)$

by (*metis Cons-nth-drop-Suc append-eq-conv-conj drop-all list.simps(3) not-le*)

moreover

have $(\exists u \ a \ b. (a, b) \in r \wedge (\exists v. x = u @ a \# v) \wedge (\exists w. y = u @ b \# w)) =$

$(\exists i < \text{length } x. i < \text{length } y \wedge \text{take } i \ x = \text{take } i \ y \wedge (x!i, y!i) \in r)$

apply *safe*

using *less-iff-Suc-add* **apply** *auto[1]*

by (*metis id-take-nth-drop*)

ultimately show *?thesis*

by (*auto simp: lexord-def Let-def*)

qed

— *lexord* is extension of partial ordering *List.lex*

lemma *lexord-lex*: $(x, y) \in \text{lex } r = ((x, y) \in \text{lexord } r \wedge \text{length } x = \text{length } y)$

proof (*induction x arbitrary: y*)

case (*Cons a x y*) **then show** *?case*

by (*cases y*) (*force+*)

qed *auto*

lemma *lexord-suff*:

assumes $(u, w) \in \text{lexord } r$ $\text{length } w \leq \text{length } u$

shows $(u @ v, w @ z) \in \text{lexord } r$

proof –

from *leD[OF assms(2)] assms(1)[unfolded lexord-take-index-conv[of u w r] min-absorb2[OF assms(2)]]*

obtain *i* **where** $\text{take } i \ u = \text{take } i \ w$ **and** $(u!i, w!i) \in r$ **and** $i < \text{length } w$

by *blast*

hence $((u @ v)!i, (w @ z)!i) \in r$

unfolding *nth-append* **using** *less-le-trans[OF <i < length w> assms(2)] <(u!i, w!i)*

$\in r$

by *presburger*

moreover have $i < \min(\text{length } (u @ v)) (\text{length } (w @ z))$

using $assms(2)$ $\langle i < length\ w \rangle$ **by** *simp*
moreover have $take\ i\ (u@v) = take\ i\ (w@z)$
using $assms(2)$ $\langle i < length\ w \rangle$ $\langle take\ i\ u = take\ i\ w \rangle$ **by** *simp*
ultimately show *?thesis*
using *lexord-take-index-conv* **by** *blast*
qed

lemma *lexord-sufE*:

assumes $(xs@zs, ys@qs) \in lexord\ r$ $xs \neq ys$ $length\ xs = length\ ys$ $length\ zs = length\ qs$
shows $(xs, ys) \in lexord\ r$
proof –
obtain i **where** $i < length\ (xs@zs)$ **and** $i < length\ (ys@qs)$ **and** $take\ i\ (xs@zs) = take\ i\ (ys@qs)$
and $((xs@zs) ! i, (ys@qs) ! i) \in r$
using $assms(1)$ *lex-take-index*[*unfolded lexord-lex, of xs @ zs ys @ qs r*]
length-append[*of xs zs, unfolded assms(3,4), folded length-append*[*of ys qs*]]
by *blast*
have $length\ (take\ i\ xs) = length\ (take\ i\ ys)$
by (*simp add: assms(3)*)
have $i < length\ xs$
using $assms(2,3)$ *le-less-linear take-all*[*of xs i*] *take-all*[*of ys i*]
 $\langle take\ i\ (xs @ zs) = take\ i\ (ys @ qs) \rangle$ *append-eq-append-conv take-append*
by *metis*
hence $(xs ! i, ys ! i) \in r$
using $\langle (xs @ zs) ! i, (ys @ qs) ! i \rangle \in r$ $assms(3)$ **by** (*simp add: nth-append*)
moreover have $take\ i\ xs = take\ i\ ys$
using $assms(3)$ $\langle take\ i\ (xs @ zs) = take\ i\ (ys @ qs) \rangle$ **by** *auto*
ultimately show *?thesis*
unfolding *lexord-take-index-conv* **using** $\langle i < length\ xs \rangle$ $assms(3)$ **by** *fastforce*
qed

lemma *lexord-irreflexive*: $\forall x. (x,x) \notin r \implies (xs,xs) \notin lexord\ r$
by (*induct xs*) *auto*

By René Thiemann:

lemma *lexord-partial-trans*:

$(\bigwedge x\ y\ z. x \in set\ xs \implies (x,y) \in r \implies (y,z) \in r \implies (x,z) \in r)$
 $\implies (xs,ys) \in lexord\ r \implies (ys,zs) \in lexord\ r \implies (xs,zs) \in lexord\ r$

proof (*induct xs arbitrary: ys zs*)

case *Nil*

from *Nil(3)* **show** *?case unfolding lexord-def* **by** (*cases zs, auto*)

next

case (*Cons x xs yys zzs*)

from *Cons(3)* **obtain** $y\ ys$ **where** $yys: yys = y \# ys$ **unfolding** *lexord-def*
by (*cases yys, auto*)

note $Cons = Cons$ [*unfolded yys*]

from *Cons(3)* **have** *one*: $(x,y) \in r \vee x = y \wedge (xs,ys) \in lexord\ r$ **by** *auto*

from *Cons(4)* **obtain** $z\ zs$ **where** $zzs: zzs = z \# zs$ **unfolding** *lexord-def*

```

    by (cases zzs, auto)
  note Cons = Cons[unfolded zzs]
  from Cons(4) have two:  $(y,z) \in r \vee y = z \wedge (ys,zs) \in \text{lexord } r$  by auto
  {
    assume  $(xs,ys) \in \text{lexord } r$  and  $(ys,zs) \in \text{lexord } r$ 
    from Cons(1)[OF - this] Cons(2)
    have  $(xs,zs) \in \text{lexord } r$  by auto
  } note ind1 = this
  {
    assume  $(x,y) \in r$  and  $(y,z) \in r$ 
    from Cons(2)[OF - this] have  $(x,z) \in r$  by auto
  } note ind2 = this
  from one two ind1 ind2
  have  $(x,z) \in r \vee x = z \wedge (xs,zs) \in \text{lexord } r$  by blast
  thus ?case unfolding zzs by auto
qed

```

lemma *lexord-trans*:

$\llbracket (x, y) \in \text{lexord } r; (y, z) \in \text{lexord } r; \text{trans } r \rrbracket \implies (x, z) \in \text{lexord } r$
 by(*auto simp: trans-def intro:lexord-partial-trans*)

lemma *lexord-transI*: $\text{trans } r \implies \text{trans } (\text{lexord } r)$

by (*meson lexord-trans transI*)

lemma *total-lexord*: $\text{total } r \implies \text{total } (\text{lexord } r)$

unfolding *total-on-def*

proof *clarsimp*

fix $x y$

assume $\forall x y. x \neq y \longrightarrow (x, y) \in r \vee (y, x) \in r$
 and $(x::'a \text{ list}) \neq y$
 and $(y, x) \notin \text{lexord } r$

then

show $(x, y) \in \text{lexord } r$

proof (*induction x arbitrary: y*)

case *Nil*

then show ?case

by (*metis lexord-Nil-left list.exhaust*)

next

case (*Cons a x y*) then show ?case

by (*cases y*) (*force+*)

qed

qed

corollary *lexord-linear*: $(\forall a b. (a,b) \in r \vee a = b \vee (b,a) \in r) \implies (x,y) \in \text{lexord } r \vee x = y \vee (y,x) \in \text{lexord } r$

using *total-lexord* by (*metis UNIV-I total-on-def*)

lemma *lexord-irrefl*:

$\text{irrefl } R \implies \text{irrefl } (\text{lexord } R)$

by (*simp add: irrefl-def lexord-irreflexive*)

lemma *lexord-asy*:

assumes *asy* *R*

shows *asy* (*lexord R*)

proof

fix *xs ys*

assume $(xs, ys) \in \text{lexord } R$

then show $(ys, xs) \notin \text{lexord } R$

proof (*induct xs arbitrary: ys*)

case *Nil*

then show *?case* **by** *simp*

next

case (*Cons x xs*)

then obtain *z zs* **where** *ys: ys = z # zs* **by** (*cases ys*) *auto*

with *assms Cons* **show** *?case* **by** (*auto dest: asyD*)

qed

qed

lemma *lexord-asy**metric*:

assumes *asy* *R*

assumes *hyp: (a, b) ∈ lexord R*

shows $(b, a) \notin \text{lexord } R$

proof –

from $\langle \text{asy } R \rangle$ **have** *asy* (*lexord R*) **by** (*rule lexord-asy*)

then show *?thesis* **by** (*auto simp: hyp dest: asyD*)

qed

lemma *asy-lex: asy R ⇒ asy (lex R)*

by (*meson asyI asyD irrefl-lex lexord-asy lexord-lex*)

lemma *asy-lenlex: asy R ⇒ asy (lenlex R)*

by (*simp add: lenlex-def asy-inv-image asy-less-than asy-lex asy-lex-prod*)

lemma *lenlex-append1*:

assumes *len: (us, xs) ∈ lenlex R* **and** *eq: length vs = length ys*

shows $(us @ vs, xs @ ys) \in \text{lenlex } R$

using *len*

proof (*induction us*)

case *Nil*

then show *?case*

by (*simp add: lenlex-def eq*)

next

case (*Cons u us*)

with *lex-append-rightI* **show** *?case*

by (*fastforce simp add: lenlex-def eq*)

qed

lemma *lenlex-append2* [*simp*]:

```

assumes irrefl R
shows  $(us @ xs, us @ ys) \in \text{lenlex } R \longleftrightarrow (xs, ys) \in \text{lenlex } R$ 
proof (induction us)
  case Nil
  then show ?case
    by (simp add: lenlex-def)
  next
  case (Cons u us)
  with assms show ?case
    by (auto simp: lenlex-def irrefl-def)
qed

```

Predicate version of lexicographic order integrated with Isabelle’s order type classes. Author: Andreas Lochbihler

```

context ord
begin

```

```

context
  notes [inductive-internals]
begin

```

```

inductive lexordp :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool
where

```

```

  Nil: lexordp [] (y # ys)
| Cons:  $x < y \Longrightarrow \text{lexordp } (x \# xs) (y \# ys)$ 
| Cons-eq:
  [ $\neg x < y; \neg y < x; \text{lexordp } xs \ ys$ ]  $\Longrightarrow \text{lexordp } (x \# xs) (y \# ys)$ 

```

```

end

```

```

lemma lexordp-simps [simp, code]:

```

```

  lexordp [] ys = (ys # [])
  lexordp xs [] = False
  lexordp (x # xs) (y # ys)  $\longleftrightarrow x < y \vee \neg y < x \wedge \text{lexordp } xs \ ys$ 
by(subst lexordp.simps, fastforce simp add: neq-Nil-conv)+

```

```

inductive lexordp-eq :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool where

```

```

  Nil: lexordp-eq [] ys
| Cons:  $x < y \Longrightarrow \text{lexordp-eq } (x \# xs) (y \# ys)$ 
| Cons-eq: [ $\neg x < y; \neg y < x; \text{lexordp-eq } xs \ ys$ ]  $\Longrightarrow \text{lexordp-eq } (x \# xs) (y \# ys)$ 

```

```

lemma lexordp-eq-simps [simp, code]:

```

```

  lexordp-eq [] ys = True
  lexordp-eq xs []  $\longleftrightarrow xs = []$ 
  lexordp-eq (x # xs) [] = False
  lexordp-eq (x # xs) (y # ys)  $\longleftrightarrow x < y \vee \neg y < x \wedge \text{lexordp-eq } xs \ ys$ 
by(subst lexordp-eq.simps, fastforce)+

```

```

lemma lexordp-append-rightI: ys # Nil  $\Longrightarrow \text{lexordp } xs (xs @ ys)$ 

```

by(*induct xs*)(*auto simp add: neq-Nil-conv*)

lemma *lexordp-append-left-rightI*: $x < y \implies \text{lexordp } (us @ x \# xs) (us @ y \# ys)$
by(*induct us*) *auto*

lemma *lexordp-eq-refl*: *lexordp-eq xs xs*
by(*induct xs*) *simp-all*

lemma *lexordp-append-leftI*: *lexordp us vs \implies lexordp (xs @ us) (xs @ vs)*
by(*induct xs*) *auto*

lemma *lexordp-append-leftD*: $\llbracket \text{lexordp } (xs @ us) (xs @ vs); \forall a. \neg a < a \rrbracket \implies$
lexordp us vs
by(*induct xs*) *auto*

lemma *lexordp-irreflexive*:
assumes *irrefl*: $\forall x. \neg x < x$
shows $\neg \text{lexordp } xs \ xs$
proof
assume *lexordp xs xs*
thus *False* **by**(*induct xs ys \equiv xs*)(*simp-all add: irrefl*)
qed

lemma *lexordp-into-lexordp-eq*:
lexordp xs ys \implies lexordp-eq xs ys
by (*induction rule: lexordp.induct*) *simp-all*

lemma *lexordp-eq-pref*: *lexordp-eq u (u @ v)*
by (*metis append-Nil2 lexordp-append-rightI lexordp-eq-refl lexordp-into-lexordp-eq*)

end

declare *ord.lexordp-simps* [*simp, code*]
declare *ord.lexordp-eq-simps* [*simp, code*]

context *order*
begin

lemma *lexordp-antisym*:
assumes *lexordp xs ys lexordp ys xs*
shows *False*
using *assms* **by** *induct auto*

lemma *lexordp-irreflexive'*: $\neg \text{lexordp } xs \ xs$
by(*rule lexordp-irreflexive*) *simp*

end

context *linorder* **begin**

lemma *lexordp-cases* [*consumes 1, case-names Nil Cons Cons-eq, cases pred: lexordp*]:

assumes *lexordp xs ys*
obtains (*Nil*) *y ys'* **where** $xs = [] \text{ } ys = y \# ys'$
| (*Cons*) *x xs' y ys'* **where** $xs = x \# xs' \text{ } ys = y \# ys' \text{ } x < y$
| (*Cons-eq*) *x xs' ys'* **where** $xs = x \# xs' \text{ } ys = x \# ys' \text{ } \textit{lexordp} \textit{ } xs' \textit{ } ys'$
using *assms* **by** *cases (fastforce simp add: not-less-iff-gr-or-eq)+*

lemma *lexordp-induct* [*consumes 1, case-names Nil Cons Cons-eq, induct pred: lexordp*]:

assumes *major: lexordp xs ys*
and *Nil: $\bigwedge y \textit{ } ys. P \textit{ } [] \textit{ } (y \# ys)$*
and *Cons: $\bigwedge x \textit{ } xs \textit{ } y \textit{ } ys. x < y \implies P \textit{ } (x \# xs) \textit{ } (y \# ys)$*
and *Cons-eq: $\bigwedge x \textit{ } xs \textit{ } ys. [\textit{lexordp} \textit{ } xs \textit{ } ys; P \textit{ } xs \textit{ } ys] \implies P \textit{ } (x \# xs) \textit{ } (x \# ys)$*
shows *P xs ys*
using *major* **by** *induct (simp-all add: Nil Cons not-less-iff-gr-or-eq Cons-eq)*

lemma *lexordp-iff*:

$\textit{lexordp} \textit{ } xs \textit{ } ys \iff (\exists x \textit{ } vs. ys = xs \textit{ } @ \textit{ } x \# vs) \vee (\exists us \textit{ } a \textit{ } b \textit{ } vs \textit{ } ws. a < b \wedge xs = us \textit{ } @ \textit{ } a \# vs \wedge ys = us \textit{ } @ \textit{ } b \# ws)$
(is ?lhs = ?rhs)

proof

assume *?lhs* **thus** *?rhs*
proof *induct*
case *Cons-eq* **thus** *?case* **by** *simp (metis append.simps(2))*
qed(*fastforce intro: disjI2 del: disjCI intro: exI[where x=[]*])+
next
assume *?rhs* **thus** *?lhs*
by(*auto intro: lexordp-append-leftI[where us=[], simplified] lexordp-append-leftI*)
qed

lemma *lexordp-conv-lexord*:

$\textit{lexordp} \textit{ } xs \textit{ } ys \iff (xs, ys) \in \textit{lexord} \textit{ } \{(x, y). x < y\}$
by(*simp add: lexordp-iff lexord-def*)

lemma *lexordp-eq-antisym*:

assumes *lexordp-eq xs ys lexordp-eq ys xs*
shows $xs = ys$
using *assms* **by** *induct simp-all*

lemma *lexordp-eq-trans*:

assumes *lexordp-eq xs ys* **and** *lexordp-eq ys zs*
shows *lexordp-eq xs zs*
using *assms*
by (*induct arbitrary: zs*) (*case-tac zs; auto*)+

lemma *lexordp-trans*:

assumes *lexordp xs ys lexordp ys zs*

shows $\text{lexordp } xs \ zs$
using assms
by $(\text{induct arbitrary: } zs) (\text{case-tac } zs; \text{ auto})+$

lemma $\text{lexordp-linear: } \text{lexordp } xs \ ys \vee \ xs = \ ys \vee \ \text{lexordp } ys \ xs$
by $(\text{induct } xs \ \text{arbitrary: } ys; \text{ case-tac } ys; \text{ fastforce})$

lemma $\text{lexordp-conv-lexordp-eq: } \text{lexordp } xs \ ys \longleftrightarrow \ \text{lexordp-eq } xs \ ys \wedge \ \neg \ \text{lexordp-eq}$
 $\ \ ys \ xs$

(is $?lhs \longleftrightarrow ?rhs$)

proof

assume $?lhs$

hence $\neg \ \text{lexordp-eq } ys \ xs$ **by** induct simp-all

with $\langle ?lhs \rangle$ **show** $?rhs$ **by** $(\text{simp add: } \text{lexordp-into-lexordp-eq})$

next

assume $?rhs$

hence $\text{lexordp-eq } xs \ ys \ \neg \ \text{lexordp-eq } ys \ xs$ **by** simp-all

thus $?lhs$ **by** induct simp-all

qed

lemma $\text{lexordp-eq-conv-lexord: } \text{lexordp-eq } xs \ ys \longleftrightarrow \ xs = \ ys \vee \ \text{lexordp } xs \ ys$
by $(\text{auto simp add: } \text{lexordp-conv-lexordp-eq } \text{lexordp-eq-refl } \text{dest: } \text{lexordp-eq-antisym})$

lemma $\text{lexordp-eq-linear: } \text{lexordp-eq } xs \ ys \vee \ \text{lexordp-eq } ys \ xs$
by $(\text{induct } xs \ \text{arbitrary: } ys) (\text{case-tac } ys; \text{ auto})+$

lemma $\text{lexordp-linorder: } \text{class.linorder } \text{lexordp-eq } \text{lexordp}$
by unfold-locales

$(\text{auto simp add: } \text{lexordp-conv-lexordp-eq } \text{lexordp-eq-refl } \text{lexordp-eq-antisym } \text{intro:}$
 $\ \text{lexordp-eq-trans } \text{del: } \text{disjCI } \text{intro: } \text{lexordp-eq-linear})$

end

67.4.3 Lexicographic combination of measure functions

These are useful for termination proofs

definition $\text{measures } fs = \text{inv-image } (\text{lex less-than}) (\%a. \text{map } (\%f. f \ a) \ fs)$

lemma $\text{wf-measures[simp]: } \text{wf } (\text{measures } fs)$

unfolding measures-def

by blast

lemma $\text{in-measures[simp]:}$

$(x, y) \in \text{measures } [] = \text{False}$

$(x, y) \in \text{measures } (f \ \# \ fs)$

$= (f \ x < f \ y \vee (f \ x = f \ y \wedge (x, y) \in \text{measures } fs))$

unfolding measures-def

by auto

lemma *measures-less*: $f x < f y \implies (x, y) \in \text{measures } (f\#fs)$
by *simp*

lemma *measures-lesseq*: $f x \leq f y \implies (x, y) \in \text{measures } fs \implies (x, y) \in \text{measures } (f\#fs)$
by *auto*

67.4.4 Lifting Relations to Lists: one element

definition *listrel1* :: $('a \times 'a) \text{ set} \implies ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
listrel1 $r = \{(xs, ys).$

$$\exists us z z' vs. xs = us @ z \# vs \wedge (z, z') \in r \wedge ys = us @ z' \# vs\}$$

lemma *listrel1I*:

$$\llbracket (x, y) \in r; xs = us @ x \# vs; ys = us @ y \# vs \rrbracket \implies (xs, ys) \in \text{listrel1 } r$$

unfolding *listrel1-def* **by** *auto*

lemma *listrel1E*:

$$\llbracket (xs, ys) \in \text{listrel1 } r; !!x y us vs. \llbracket (x, y) \in r; xs = us @ x \# vs; ys = us @ y \# vs \rrbracket \implies P \rrbracket \implies P$$

unfolding *listrel1-def* **by** *auto*

lemma *not-Nil-listrel1 [iff]*: $([], xs) \notin \text{listrel1 } r$

unfolding *listrel1-def* **by** *blast*

lemma *not-listrel1-Nil [iff]*: $(xs, []) \notin \text{listrel1 } r$

unfolding *listrel1-def* **by** *blast*

lemma *Cons-listrel1-Cons [iff]*:

$$(x \# xs, y \# ys) \in \text{listrel1 } r \longleftrightarrow (x, y) \in r \wedge xs = ys \vee x = y \wedge (xs, ys) \in \text{listrel1 } r$$

by (*simp add: listrel1-def Cons-eq-append-conv*) (*blast*)

lemma *listrel1I1*: $(x, y) \in r \implies (x \# xs, y \# xs) \in \text{listrel1 } r$

by *fast*

lemma *listrel1I2*: $(xs, ys) \in \text{listrel1 } r \implies (x \# xs, x \# ys) \in \text{listrel1 } r$

by *fast*

lemma *append-listrel1I*:

$$(xs, ys) \in \text{listrel1 } r \wedge us = vs \vee xs = ys \wedge (us, vs) \in \text{listrel1 } r \implies (xs @ us, ys @ vs) \in \text{listrel1 } r$$

unfolding *listrel1-def*

by *auto* (*blast intro: append-eq-appendI*)**+**

lemma *Cons-listrel1E1 [elim!]*:

$$\text{assumes } (x \# xs, ys) \in \text{listrel1 } r$$

and $\bigwedge y. ys = y \# xs \implies (x, y) \in r \implies R$
and $\bigwedge zs. ys = x \# zs \implies (xs, zs) \in listrel1\ r \implies R$
shows R
using *assms* **by** (*cases ys*) *blast+*

lemma *Cons-listrel1E2[elim!]*:
assumes $(xs, y \# ys) \in listrel1\ r$
and $\bigwedge x. xs = x \# ys \implies (x, y) \in r \implies R$
and $\bigwedge zs. xs = y \# zs \implies (zs, ys) \in listrel1\ r \implies R$
shows R
using *assms* **by** (*cases xs*) *blast+*

lemma *snoc-listrel1-snoc-iff*:
 $(xs @ [x], ys @ [y]) \in listrel1\ r$
 $\longleftrightarrow (xs, ys) \in listrel1\ r \wedge x = y \vee xs = ys \wedge (x, y) \in r$ (**is** $?L \longleftrightarrow ?R$)
proof
assume $?L$ **thus** $?R$
by (*fastforce simp: listrel1-def snoc-eq-iff-butlast butlast-append*)
next
assume $?R$ **then show** $?L$ **unfolding** *listrel1-def* **by** *force*
qed

lemma *listrel1-eq-len*: $(xs, ys) \in listrel1\ r \implies length\ xs = length\ ys$
unfolding *listrel1-def* **by** *auto*

lemma *listrel1-mono*:
 $r \subseteq s \implies listrel1\ r \subseteq listrel1\ s$
unfolding *listrel1-def* **by** *blast*

lemma *listrel1-converse*: $listrel1\ (r^{-1}) = (listrel1\ r)^{-1}$
unfolding *listrel1-def* **by** *blast*

lemma *in-listrel1-converse*:
 $(x, y) \in listrel1\ (r^{-1}) \longleftrightarrow (x, y) \in (listrel1\ r)^{-1}$
unfolding *listrel1-def* **by** *blast*

lemma *listrel1-iff-update*:
 $(xs, ys) \in (listrel1\ r)$
 $\longleftrightarrow (\exists y\ n. (xs ! n, y) \in r \wedge n < length\ xs \wedge ys = xs[n:=y])$ (**is** $?L \longleftrightarrow ?R$)
proof
assume $?L$
then obtain $x\ y\ u\ v$ **where** $xs = u @ x \# v$ $ys = u @ y \# v$ $(x, y) \in r$
unfolding *listrel1-def* **by** *auto*
then have $ys = xs[length\ u := y]$ **and** $length\ u < length\ xs$
and $(xs ! length\ u, y) \in r$ **by** *auto*
then show $?R$ **by** *auto*
next
assume $?R$

then obtain $x y n$ **where** $(xs!n, y) \in r$ $n < \text{size } xs$ $ys = xs[n:=y]$ $x = xs!n$
by *auto*
then obtain $u v$ **where** $xs = u @ x \# v$ **and** $ys = u @ y \# v$ **and** $(x, y) \in r$
by (*auto intro: upd-conv-take-nth-drop id-take-nth-drop*)
then show ?L **by** (*auto simp: listrel1-def*)
qed

Accessible part and wellfoundedness:

lemma *Cons-acc-listrel1I* [*intro!*]:

$x \in \text{Wellfounded.acc } r \implies xs \in \text{Wellfounded.acc } (\text{listrel1 } r) \implies (x \# xs) \in \text{Wellfounded.acc } (\text{listrel1 } r)$

proof (*induction arbitrary: xs set: Wellfounded.acc*)

case *outer*: (1 u)

show ?*case*

proof (*induct xs rule: acc-induct*)

case 1

show $xs \in \text{Wellfounded.acc } (\text{listrel1 } r)$

by (*simp add: outer.prem*s)

qed (*metis (no-types, lifting) Cons-listrel1E2 acc.simp*s *outer.IH*)

qed

lemma *lists-accD*: $xs \in \text{lists } (\text{Wellfounded.acc } r) \implies xs \in \text{Wellfounded.acc } (\text{listrel1 } r)$

proof (*induct set: lists*)

case *Nil*

then show ?*case*

by (*meson acc.intros not-listrel1-Nil*)

next

case (*Cons a l*)

then show ?*case*

by *blast*

qed

lemma *lists-accI*: $xs \in \text{Wellfounded.acc } (\text{listrel1 } r) \implies xs \in \text{lists } (\text{Wellfounded.acc } r)$

proof (*induction set: Wellfounded.acc*)

case (1 x)

then have $\bigwedge u v. \llbracket u \in \text{set } x; (v, u) \in r \rrbracket \implies v \in \text{Wellfounded.acc } r$

by (*metis in-lists-conv-set in-set-conv-decomp listrel1I*)

then show ?*case*

by (*meson acc.intros in-listsI*)

qed

lemma *wf-listrel1-iff*[*simp*]: $\text{wf}(\text{listrel1 } r) = \text{wf } r$

by (*auto simp: wf-acc-iff*)

intro: lists-accD lists-accI[*THEN Cons-in-lists-iff*[*THEN iffD1*, *THEN conjunct1*]]

67.4.5 Lifting Relations to Lists: all elements**inductive-set** $listrel :: ('a \times 'b) set \Rightarrow ('a list \times 'b list) set$ **for** $r :: ('a \times 'b) set$ **where** $Nil: ([], []) \in listrel\ r$ $| Cons: [(x,y) \in r; (xs,ys) \in listrel\ r] \Longrightarrow (x\#\!xs, y\#\!ys) \in listrel\ r$ **inductive-cases** $listrel-Nil1$ $[elim!]: ([],xs) \in listrel\ r$ **inductive-cases** $listrel-Nil2$ $[elim!]: (xs,[]) \in listrel\ r$ **inductive-cases** $listrel-Cons1$ $[elim!]: (y\#\!ys,xs) \in listrel\ r$ **inductive-cases** $listrel-Cons2$ $[elim!]: (xs,y\#\!ys) \in listrel\ r$ **lemma** $listrel-eq-len: (xs, ys) \in listrel\ r \Longrightarrow length\ xs = length\ ys$ **by** (*induct rule: listrel.induct*) *auto***lemma** $listrel-iff-zip$ $[code-unfold]: (xs,ys) \in listrel\ r \longleftrightarrow$ $length\ xs = length\ ys \wedge (\forall (x,y) \in set(zip\ xs\ ys). (x,y) \in r)$ (**is** $?L \longleftrightarrow ?R$)**proof****assume** $?L$ **thus** $?R$ **by** *induct* (*auto intro: listrel-eq-len*)**next****assume** $?R$ **thus** $?L$ **apply** (*clarify*)**by** (*induct rule: list-induct2*) (*auto intro: listrel.intros*)**qed****lemma** $listrel-iff-nth: (xs,ys) \in listrel\ r \longleftrightarrow$ $length\ xs = length\ ys \wedge (\forall n < length\ xs. (xs!\!n, ys!\!n) \in r)$ (**is** $?L \longleftrightarrow ?R$)**by** (*auto simp add: all-set-conv-all-nth listrel-iff-zip*)**lemma** $listrel-mono: r \subseteq s \Longrightarrow listrel\ r \subseteq listrel\ s$ **by** (*meson listrel-iff-nth subrelI subset-eq*)**lemma** $listrel-subset:$ **assumes** $r \subseteq A \times A$ **shows** $listrel\ r \subseteq lists\ A \times lists\ A$ **proof** *clarify***show** $a \in lists\ A \wedge b \in lists\ A$ **if** $(a, b) \in listrel\ r$ **for** $a\ b$ **using** *that assms* **by** (*induction rule: listrel.induct, auto*)**qed****lemma** $listrel-refl-on:$ **assumes** $refl-on\ A\ r$ **shows** $refl-on\ (lists\ A)\ (listrel\ r)$ **proof** $-$ **have** $l \in lists\ A \Longrightarrow (l, l) \in listrel\ r$ **for** l **using** *assms* **unfolding** $refl-on-def$ **by** (*induction l, auto intro: listrel.intros*)**then show** $?thesis$ **by** (*meson assms listrel-subset refl-on-def*)

qed

lemma *listrel-sym*: $\text{sym } r \implies \text{sym } (\text{listrel } r)$
 by (*simp add: listrel-iff-nth sym-def*)

lemma *listrel-trans*:

assumes *trans r* shows *trans (listrel r)*

proof –

have $(x, z) \in \text{listrel } r$ if $(x, y) \in \text{listrel } r$ $(y, z) \in \text{listrel } r$ for $x y z$
 using *that*

proof *induction*

case (*Cons x y xs ys*)

then show *?case*

by *clarsimp (metis assms listrel.Cons listrel-iff-nth transD)*

qed *auto*

then show *?thesis*

using *transI* by *blast*

qed

theorem *equiv-listrel*: $\text{equiv } A r \implies \text{equiv } (\text{lists } A) (\text{listrel } r)$
 by (*simp add: equiv-def listrel-refl-on listrel-sym listrel-trans*)

lemma *listrel-rtrancl-refl[iff]*: $(xs, xs) \in \text{listrel}(r^*)$
 using *listrel-refl-on[of UNIV, OF refl-rtrancl]*
 by(*auto simp: refl-on-def*)

lemma *listrel-rtrancl-trans*:

$[(xs, ys) \in \text{listrel}(r^*); (ys, zs) \in \text{listrel}(r^*)] \implies (xs, zs) \in \text{listrel}(r^*)$

by (*metis listrel-trans trans-def trans-rtrancl*)

lemma *listrel-Nil [simp]*: $\text{listrel } r \text{ “ } \{\} = \{\}$
 by (*blast intro: listrel.intros*)

lemma *listrel-Cons*:

$\text{listrel } r \text{ “ } \{x \# xs\} = \text{set-Cons } (r \text{ “ } \{x\}) (\text{listrel } r \text{ “ } \{xs\})$

by (*auto simp add: set-Cons-def intro: listrel.intros*)

Relating *listrel1*, *listrel* and closures:

lemma *listrel1-rtrancl-subset-rtrancl-listrel1*: $\text{listrel1 } (r^*) \subseteq (\text{listrel1 } r)^*$

proof (*rule subrelI*)

fix *xs ys* assume *1*: $(xs, ys) \in \text{listrel1 } (r^*)$

{ fix *x y us vs*

have $(x, y) \in r^* \implies (us @ x \# vs, us @ y \# vs) \in (\text{listrel1 } r)^*$

proof(*induct rule: rtrancl.induct*)

case *rtrancl-refl* show *?case* by *simp*

next

case *rtrancl-into-rtrancl* thus *?case*

by (*metis listrel1I rtrancl.rtrancl-into-rtrancl*)

qed }

thus $(xs,ys) \in (listrel1\ r)^*$ **using** 1 **by**(blast elim: listrel1E)
qed

lemma rtrancl-listrel1-eq-len: $(x,y) \in (listrel1\ r)^* \implies length\ x = length\ y$
by (induct rule: rtrancl.induct) (auto intro: listrel1-eq-len)

lemma rtrancl-listrel1-ConsI1:
 $(xs,ys) \in (listrel1\ r)^* \implies (x\#\!xs,x\#\!ys) \in (listrel1\ r)^*$
proof (induction rule: rtrancl.induct)
case (rtrancl-into-rtrancl a b c)
then show ?case
by (metis listrel1I2 rtrancl.rtrancl-into-rtrancl)
qed auto

lemma rtrancl-listrel1-ConsI2:
 $(x,y) \in r^* \implies (xs,ys) \in (listrel1\ r)^* \implies (x\#\!xs,y\#\!ys) \in (listrel1\ r)^*$
by (meson in-mono listrel1I1 listrel1-rtrancl-subset-rtrancl-listrel1 rtrancl-listrel1-ConsI1
rtrancl-trans)

lemma listrel1-subset-listrel:
 $r \subseteq r' \implies refl\ r' \implies listrel1\ r \subseteq listrel(r')$
by(auto elim!: listrel1E simp add: listrel-iff-zip set-zip refl-on-def)

lemma listrel-reflcl-if-listrel1:
 $(xs,ys) \in listrel1\ r \implies (xs,ys) \in listrel(r^*)$
by(erule listrel1E)(auto simp add: listrel-iff-zip set-zip)

lemma listrel-rtrancl-eq-rtrancl-listrel1: $listrel\ (r^*) = (listrel1\ r)^*$

proof
{ **fix** x y **assume** $(x,y) \in listrel\ (r^*)$
then have $(x,y) \in (listrel1\ r)^*$
by induct (auto intro: rtrancl-listrel1-ConsI2) }
then show $listrel\ (r^*) \subseteq (listrel1\ r)^*$
by (rule subrelI)
next
show $listrel\ (r^*) \supseteq (listrel1\ r)^*$
proof(rule subrelI)
fix xs ys **assume** $(xs,ys) \in (listrel1\ r)^*$
then show $(xs,ys) \in listrel\ (r^*)$
proof induct
case base **show** ?case **by**(auto simp add: listrel-iff-zip set-zip)
next
case (step ys zs)
thus ?case **by** (metis listrel-reflcl-if-listrel1 listrel-rtrancl-trans)
qed
qed
qed

lemma rtrancl-listrel1-if-listrel:

$(xs, ys) \in \text{listrel } r \implies (xs, ys) \in (\text{listrel1 } r)^*$
by (*metis listrel-rtrancl-eq-rtrancl-listrel1 subsetD[OF listrel-mono] r-into-rtrancl subsetI*)

lemma *listrel-subset-rtrancl-listrel1*: $\text{listrel } r \subseteq (\text{listrel1 } r)^*$
by (*fast intro:rtrancl-listrel1-if-listrel*)

67.5 Size function

lemma [*measure-function*]: $\text{is-measure } f \implies \text{is-measure } (\text{size-list } f)$
by (*rule is-measure-trivial*)

lemma [*measure-function*]: $\text{is-measure } f \implies \text{is-measure } (\text{size-option } f)$
by (*rule is-measure-trivial*)

lemma *size-list-estimation*[*termination-simp*]:
 $x \in \text{set } xs \implies y < f x \implies y < \text{size-list } f xs$
by (*induct xs*) *auto*

lemma *size-list-estimation'*[*termination-simp*]:
 $x \in \text{set } xs \implies y \leq f x \implies y \leq \text{size-list } f xs$
by (*induct xs*) *auto*

lemma *size-list-map*[*simp*]: $\text{size-list } f (\text{map } g xs) = \text{size-list } (f \circ g) xs$
by (*induct xs*) *auto*

lemma *size-list-append*[*simp*]: $\text{size-list } f (xs @ ys) = \text{size-list } f xs + \text{size-list } f ys$
by (*induct xs, auto*)

lemma *size-list-pointwise*[*termination-simp*]:
 $(\bigwedge x. x \in \text{set } xs \implies f x \leq g x) \implies \text{size-list } f xs \leq \text{size-list } g xs$
by (*induct xs*) *force+*

67.6 Monad operation

definition *bind* :: 'a list \Rightarrow ('a \Rightarrow 'b list) \Rightarrow 'b list **where**
bind xs f = *concat* (*map* f xs)

hide-const (**open**) *bind*

lemma *bind-simps* [*simp*]:
List.bind [] f = []
List.bind (x # xs) f = f x @ *List.bind* xs f
by (*simp-all add: bind-def*)

lemma *list-bind-cong* [*fundef-cong*]:
assumes $xs = ys$ ($\bigwedge x. x \in \text{set } xs \implies f x = g x$)
shows $\text{List.bind } xs f = \text{List.bind } ys g$

proof –

from *assms*(2) **have** $\text{List.bind } xs f = \text{List.bind } xs g$

by (induction xs) simp-all
 with assms(1) show ?thesis by simp
 qed

lemma *set-list-bind*: $set (List.bind\ xs\ f) = (\bigcup_{x \in set\ xs} set\ (f\ x))$
 by (induction xs) simp-all

67.7 Code generation

Optional tail recursive version of *map*. Can avoid stack overflow in some target languages.

fun *map-tailrec-rev* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
map-tailrec-rev f [] bs = bs |
map-tailrec-rev f (a#as) bs = *map-tailrec-rev* f as (f a # bs)

lemma *map-tailrec-rev*:
 $map\ tailrec\ rev\ f\ as\ bs = rev (map\ f\ as) @ bs$
 by (induction as arbitrary: bs) simp-all

definition *map-tailrec* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
map-tailrec f as = rev (*map-tailrec-rev* f as [])

Code equation:

lemma *map-eq-map-tailrec*: $map = map\ tailrec$
 by (simp add: fun-eq-iff map-tailrec-def map-tailrec-rev)

67.7.1 Counterparts for set-related operations

definition *member* :: $'a\ list \Rightarrow 'a \Rightarrow bool$ **where**
 [code-abbrev]: $member\ xs\ x \longleftrightarrow x \in set\ xs$

Use *member* only for generating executable code. Otherwise use $x \in set\ xs$ instead — it is much easier to reason about.

lemma *member-rec* [code]:
 $member\ (x\ \#\ xs)\ y \longleftrightarrow x = y \vee member\ xs\ y$
 $member\ []\ y \longleftrightarrow False$
 by (auto simp add: member-def)

lemma *in-set-member* :
 $x \in set\ xs \longleftrightarrow member\ xs\ x$
 by (simp add: member-def)

lemmas *list-all-iff* [code-abbrev] = fun-cong[OF list.pred-set]

definition *list-ex* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**
 [code-abbrev]: $list\ ex\ P\ xs \longleftrightarrow Bex\ (set\ xs)\ P$

definition *list-ex1* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**

list-ex1-iff [code-abbrev]: $list-ex1\ P\ xs \longleftrightarrow (\exists! x. x \in set\ xs \wedge P\ x)$

Usually you should prefer $\forall x \in set\ xs, \exists x \in set\ xs$ and $\exists! x. x \in set\ xs \wedge$ - over *list-all*, *list-ex* and *list-ex1* in specifications.

lemma *list-all-simps* [code]:

$list-all\ P\ (x\ \#\ xs) \longleftrightarrow P\ x \wedge list-all\ P\ xs$

$list-all\ P\ [] \longleftrightarrow True$

by (*simp-all add: list-all-iff*)

lemma *list-ex-simps* [simp, code]:

$list-ex\ P\ (x\ \#\ xs) \longleftrightarrow P\ x \vee list-ex\ P\ xs$

$list-ex\ P\ [] \longleftrightarrow False$

by (*simp-all add: list-ex-iff*)

lemma *list-ex1-simps* [simp, code]:

$list-ex1\ P\ [] = False$

$list-ex1\ P\ (x\ \#\ xs) = (if\ P\ x\ then\ list-all\ (\lambda y. \neg P\ y \vee x = y)\ xs\ else\ list-ex1\ P\ xs)$

by (*auto simp add: list-ex1-iff list-all-iff*)

lemma *Ball-set-list-all*:

$Ball\ (set\ xs)\ P \longleftrightarrow list-all\ P\ xs$

by (*simp add: list-all-iff*)

lemma *Bex-set-list-ex*:

$Bex\ (set\ xs)\ P \longleftrightarrow list-ex\ P\ xs$

by (*simp add: list-ex-iff*)

lemma *list-all-append* [simp]:

$list-all\ P\ (xs\ @\ ys) \longleftrightarrow list-all\ P\ xs \wedge list-all\ P\ ys$

by (*auto simp add: list-all-iff*)

lemma *list-ex-append* [simp]:

$list-ex\ P\ (xs\ @\ ys) \longleftrightarrow list-ex\ P\ xs \vee list-ex\ P\ ys$

by (*auto simp add: list-ex-iff*)

lemma *list-all-rev* [simp]:

$list-all\ P\ (rev\ xs) \longleftrightarrow list-all\ P\ xs$

by (*simp add: list-all-iff*)

lemma *list-ex-rev* [simp]:

$list-ex\ P\ (rev\ xs) \longleftrightarrow list-ex\ P\ xs$

by (*simp add: list-ex-iff*)

lemma *list-all-length*:

$list-all\ P\ xs \longleftrightarrow (\forall n < length\ xs. P\ (xs\ !\ n))$

by (*auto simp add: list-all-iff set-conv-nth*)

lemma *list-ex-length*:

$list\text{-}ex\ P\ xs \longleftrightarrow (\exists n < length\ xs. P\ (xs\ !\ n))$
by (*auto simp add: list-ex-iff set-conv-nth*)

lemmas *list-all-cong [fundef-cong] = list.pred-cong*

lemma *list-ex-cong [fundef-cong]:*

$xs = ys \implies (\bigwedge x. x \in set\ ys \implies f\ x = g\ x) \implies list\text{-}ex\ f\ xs = list\text{-}ex\ g\ ys$
by (*simp add: list-ex-iff*)

definition *can-select* :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$ **where**
[code-abbrev]: can-select P A = $(\exists !x \in A. P\ x)$

lemma *can-select-set-list-ex1 [code]:*

$can\text{-}select\ P\ (set\ A) = list\text{-}ex1\ P\ A$
by (*simp add: list-ex1-iff can-select-def*)

Executable checks for relations on sets

definition *listrel1p* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
listrel1p r xs ys = $((xs, ys) \in listrel1\ \{(x, y). r\ x\ y\})$

lemma *[code-unfold]:*

$(xs, ys) \in listrel1\ r = listrel1p\ (\lambda x\ y. (x, y) \in r)\ xs\ ys$
unfolding *listrel1p-def by auto*

lemma *[code]:*

$listrel1p\ r\ []\ xs = False$
 $listrel1p\ r\ xs\ [] = False$
 $listrel1p\ r\ (x\ \#\ xs)\ (y\ \#\ ys) \longleftrightarrow$
 $r\ x\ y \wedge xs = ys \vee x = y \wedge listrel1p\ r\ xs\ ys$
by (*simp add: listrel1p-def*)⁺

definition

lexordp :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
lexordp r xs ys = $((xs, ys) \in lexord\ \{(x, y). r\ x\ y\})$

lemma *[code-unfold]:*

$(xs, ys) \in lexord\ r = lexordp\ (\lambda x\ y. (x, y) \in r)\ xs\ ys$
unfolding *lexordp-def by auto*

lemma *[code]:*

$lexordp\ r\ xs\ [] = False$
 $lexordp\ r\ []\ (y\ \#\ ys) = True$
 $lexordp\ r\ (x\ \#\ xs)\ (y\ \#\ ys) = (r\ x\ y \vee (x = y \wedge lexordp\ r\ xs\ ys))$
unfolding *lexordp-def by auto*

Bounded quantification and summation over nats.

lemma *atMost-upto [code-unfold]:*

$\{..n\} = set\ [0..<Suc\ n]$
by *auto*

lemma *atLeast-upt* [code-unfold]:

$\{..
by *auto*$

lemma *greaterThanLessThan-upt* [code-unfold]:

$\{n<..
by *auto*$

lemmas *atLeastLessThan-upt* [code-unfold] = *set-upt* [symmetric]

lemma *greaterThanAtMost-upt* [code-unfold]:

$\{n<..
by *auto*$

lemma *atLeastAtMost-upt* [code-unfold]:

$\{n..
by *auto*$

lemma *all-nat-less-eq* [code-unfold]:

$(\forall m < n :: \text{nat}. P m) \longleftrightarrow (\forall m \in \{0..
by *auto*$

lemma *ex-nat-less-eq* [code-unfold]:

$(\exists m < n :: \text{nat}. P m) \longleftrightarrow (\exists m \in \{0..
by *auto*$

lemma *all-nat-less* [code-unfold]:

$(\forall m \leq n :: \text{nat}. P m) \longleftrightarrow (\forall m \in \{0..
by *auto*$

lemma *ex-nat-less* [code-unfold]:

$(\exists m \leq n :: \text{nat}. P m) \longleftrightarrow (\exists m \in \{0..
by *auto*$

Bounded *LEAST* operator:

definition *Bleat* $S P = (\text{LEAST } x. x \in S \wedge P x)$

definition *abort-Bleat* $S P = (\text{LEAST } x. x \in S \wedge P x)$

declare [[code abort: abort-Bleat]]

lemma *Bleat-code* [code]:

Bleat (set xs) $P = (\text{case filter } P (\text{sort } xs) \text{ of}$
 $x \# xs \Rightarrow x \mid$
 $[] \Rightarrow \text{abort-Bleat } (\text{set } xs) P)$

proof (cases filter P (sort xs))

case Nil thus *?thesis* **by** (simp add: *Bleat-def* *abort-Bleat-def*)
next


```

case (Cons x ys)
have (LEAST x. x ∈ set xs ∧ P x) = x
proof (rule Least-equality)
  show x ∈ set xs ∧ P x
    by (metis Cons Cons-eq-filter-iff in-set-conv-decomp set-sort)
  next
    fix y assume y ∈ set xs ∧ P y
    hence y ∈ set (filter P xs) by auto
    thus x ≤ y
      by (metis Cons eq-iff filter-sort set-ConsD set-sort sorted-wrt.simps(2)
sorted-sort)
  qed
thus ?thesis using Cons by (simp add: Bleast-def)
qed

```

```

declare Bleast-def[symmetric, code-unfold]

```

Summation over ints.

```

lemma greaterThanLessThan-upto [code-unfold]:
  {i<..j::int} = set [i+1..j - 1]
by auto

```

```

lemma atLeastLessThan-upto [code-unfold]:
  {i..j::int} = set [i..j - 1]
by auto

```

```

lemma greaterThanAtMost-upto [code-unfold]:
  {i<..j::int} = set [i+1..j]
by auto

```

```

lemmas atLeastAtMost-upto [code-unfold] = set-upto [symmetric]

```

67.7.2 Optimizing by rewriting

```

definition null :: 'a list ⇒ bool where
  [code-abbrev]: null xs ↔ xs = []

```

Efficient emptiness check is implemented by *null*.

```

lemma null-rec [code]:
  null (x # xs) ↔ False
  null [] ↔ True
by (simp-all add: null-def)

```

```

lemma eq-Nil-null:
  xs = [] ↔ null xs
by (simp add: null-def)

```

```

lemma equal-Nil-null [code-unfold]:
  HOL.equal xs [] ↔ null xs

```

HOL.equal [] = null
by (*auto simp add: equal null-def*)

definition *maps* :: ('a ⇒ 'b list) ⇒ 'a list ⇒ 'b list **where**
[code-abbrev]: maps f xs = concat (map f xs)

definition *map-filter* :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list **where**
[code-post]: map-filter f xs = map (the ∘ f) (filter (λx. f x ≠ None) xs)

Operations *maps* and *map-filter* avoid intermediate lists on execution – do not use for proving.

lemma *maps-simps* [*code*]:
maps f (x # xs) = f x @ maps f xs
maps f [] = []
by (*simp-all add: maps-def*)

lemma *map-filter-simps* [*code*]:
map-filter f (x # xs) = (case f x of None ⇒ map-filter f xs | Some y ⇒ y # map-filter f xs)
map-filter f [] = []
by (*simp-all add: map-filter-def split: option.split*)

lemma *concat-map-maps*:
concat (map f xs) = maps f xs
by (*simp add: maps-def*)

lemma *map-filter-map-filter* [*code-unfold*]:
map f (filter P xs) = map-filter (λx. if P x then Some (f x) else None) xs
by (*simp add: map-filter-def*)

Optimized code for $\forall i \in \{a..b::int\}$ and $\forall n: \{a..<b::nat\}$ and similiarly for \exists .

definition *all-interval-nat* :: (nat ⇒ bool) ⇒ nat ⇒ nat ⇒ bool **where**
all-interval-nat P i j ⟷ (∀ n ∈ {i..<j}. P n)

lemma [*code*]:
all-interval-nat P i j ⟷ i ≥ j ∨ P i ∧ all-interval-nat P (Suc i) j
proof –
have *: $\bigwedge n. P i \implies \forall n \in \{Suc\ i..<j\}. P n \implies i \leq n \implies n < j \implies P n$
using *le-less-Suc-eq* **by** *fastforce*
show *?thesis* **by** (*auto simp add: all-interval-nat-def intro: **)
qed

lemma *list-all-iff-all-interval-nat* [*code-unfold*]:
list-all P [i..<j] ⟷ all-interval-nat P i j
by (*simp add: list-all-iff all-interval-nat-def*)

lemma *list-ex-iff-not-all-inverval-nat* [*code-unfold*]:
list-ex P [i..<j] ⟷ ¬ (all-interval-nat (Not ∘ P) i j)
by (*simp add: list-ex-iff all-interval-nat-def*)

definition *all-interval-int* :: (int ⇒ bool) ⇒ int ⇒ int ⇒ bool **where**
all-interval-int P i j ⇔ (∀ k ∈ {i..j}. P k)

lemma [code]:

all-interval-int P i j ⇔ i > j ∨ P i ∧ *all-interval-int* P (i + 1) j

proof –

have *: ∧k. P i ⇒ ∀ k ∈ {i+1..j}. P k ⇒ i ≤ k ⇒ k ≤ j ⇒ P k

by (smt (verit, best) atLeastAtMost-iff)

show ?thesis **by** (auto simp add: *all-interval-int-def* intro: *)

qed

lemma *list-all-iff-all-interval-int* [code-unfold]:

list-all P [i..j] ⇔ *all-interval-int* P i j

by (simp add: *list-all-iff-all-interval-int-def*)

lemma *list-ex-iff-not-all-interval-int* [code-unfold]:

list-ex P [i..j] ⇔ ¬ (*all-interval-int* (Not ∘ P) i j)

by (simp add: *list-ex-iff-all-interval-int-def*)

optimized code (tail-recursive) for *length*

definition *gen-length* :: nat ⇒ 'a list ⇒ nat

where *gen-length* n xs = n + *length* xs

lemma *gen-length-code* [code]:

gen-length n [] = n

gen-length n (x # xs) = *gen-length* (Suc n) xs

by(simp-all add: *gen-length-def*)

declare *list.size*(3–4)[code del]

lemma *length-code* [code]: *length* = *gen-length* 0

by(simp add: *gen-length-def* fun-eq-iff)

hide-const (open) *member null maps map-filter all-interval-nat all-interval-int gen-length*

67.7.3 Pretty lists

ML ‹

(* Code generation for list literals. *)

signature LIST-CODE =

sig

val add-literal-list: string → theory → theory

end;

structure List-Code : LIST-CODE =

struct

```
open Basic-Code-Thingol;
```

```
fun implode-list t =
  let
    fun dest-cons (ICnst { sym = Code-Symbol.Constant const-name <Cons>, ...
  } $ t1 $ t2) = SOME (t1, t2)
      | dest-cons - = NONE;
    val (ts, t') = Code-Thingol.unfoldr dest-cons t;
  in case t'
    of ICnst { sym = Code-Symbol.Constant const-name <Nil>, ... } => SOME
  ts
      | - => NONE
  end;
```

```
fun print-list (target-fxy, target-cons) pr fxy t1 t2 =
  Code-Printer.brackify-infix (target-fxy, Code-Printer.R) fxy (
    pr (Code-Printer.INFX (target-fxy, Code-Printer.X)) t1,
    Code-Printer.str target-cons,
    pr (Code-Printer.INFX (target-fxy, Code-Printer.R)) t2
  );
```

```
fun add-literal-list target =
  let
    fun pretty literals pr - vars fxy [(t1, -), (t2, -)] =
      case Option.map (cons t1) (implode-list t2)
      of SOME ts =>
         Code-Printer.literal-list literals (map (pr vars Code-Printer.NOBR) ts)
        | NONE =>
         print-list (Code-Printer.infix-cons literals) (pr vars) fxy t1 t2;
  in
    Code-Target.set-printings (Code-Symbol.Constant (const-name <Cons>,
      [(target, SOME (Code-Printer.complex-const-syntax (2, pretty)))]))
  end
end;
```

code-printing

```
type-constructor list →
  (SML) - list
  and (OCaml) - list
  and (Haskell) ![-]
  and (Scala) List[-]
| constant Nil →
  (SML) []
  and (OCaml) []
  and (Haskell) []
  and (Scala) !Nil
```

```

| class-instance list :: equal →
  (Haskell) –
| constant HOL.equal :: 'a list ⇒ 'a list ⇒ bool →
  (Haskell) infix 4 ==

setup ⟨fold (List-Code.add-literal-list) [SML, OCaml, Haskell, Scala]⟩

code-reserved SML
  list

code-reserved OCaml
  list

```

67.7.4 Use convenient predefined operations

```

code-printing
constant (@) →
  (SML) infixr 7 @
  and (OCaml) infixr 6 @
  and (Haskell) infixr 5 ++
  and (Scala) infixl 7 ++
| constant map →
  (Haskell) map
| constant filter →
  (Haskell) filter
| constant concat →
  (Haskell) concat
| constant List.maps →
  (Haskell) concatMap
| constant rev →
  (Haskell) reverse
| constant zip →
  (Haskell) zip
| constant List.null →
  (Haskell) null
| constant takeWhile →
  (Haskell) takeWhile
| constant dropWhile →
  (Haskell) dropWhile
| constant list-all →
  (Haskell) all
| constant list-ex →
  (Haskell) any

```

67.7.5 Implementation of sets by lists

```

lemma is-empty-set [code]:
  Set.is-empty (set xs) ↔ List.null xs
by (simp add: Set.is-empty-def null-def)

```

lemma *empty-set* [code]:
 $\{\} = \text{set } []$
by *simp*

lemma *UNIV-coset* [code]:
 $\text{UNIV} = \text{List.coset } []$
by *simp*

lemma *compl-set* [code]:
 $\neg \text{set } xs = \text{List.coset } xs$
by *simp*

lemma *compl-coset* [code]:
 $\neg \text{List.coset } xs = \text{set } xs$
by *simp*

lemma [code]:
 $x \in \text{set } xs \longleftrightarrow \text{List.member } xs \ x$
 $x \in \text{List.coset } xs \longleftrightarrow \neg \text{List.member } xs \ x$
by (*simp-all add: member-def*)

lemma *insert-code* [code]:
 $\text{insert } x \ (\text{set } xs) = \text{set } (\text{List.insert } x \ xs)$
 $\text{insert } x \ (\text{List.coset } xs) = \text{List.coset } (\text{removeAll } x \ xs)$
by *simp-all*

lemma *remove-code* [code]:
 $\text{Set.remove } x \ (\text{set } xs) = \text{set } (\text{removeAll } x \ xs)$
 $\text{Set.remove } x \ (\text{List.coset } xs) = \text{List.coset } (\text{List.insert } x \ xs)$
by (*simp-all add: remove-def Compl-insert*)

lemma *filter-set* [code]:
 $\text{Set.filter } P \ (\text{set } xs) = \text{set } (\text{filter } P \ xs)$
by *auto*

lemma *image-set* [code]:
 $\text{image } f \ (\text{set } xs) = \text{set } (\text{map } f \ xs)$
by *simp*

lemma *subset-code* [code]:
 $\text{set } xs \leq B \longleftrightarrow (\forall x \in \text{set } xs. x \in B)$
 $A \leq \text{List.coset } ys \longleftrightarrow (\forall y \in \text{set } ys. y \notin A)$
 $\text{List.coset } [] \subseteq \text{set } [] \longleftrightarrow \text{False}$
by *auto*

A frequent case – avoid intermediate sets

lemma [code-unfold]:
 $\text{set } xs \subseteq \text{set } ys \longleftrightarrow \text{list-all } (\lambda x. x \in \text{set } ys) \ xs$
by (*auto simp: list-all-iff*)

lemma *Ball-set* [code]:

Ball (set xs) P \longleftrightarrow *list-all P xs*
by (*simp add: list-all-iff*)

lemma *Bex-set* [code]:

Bex (set xs) P \longleftrightarrow *list-ex P xs*
by (*simp add: list-ex-iff*)

lemma *card-set* [code]:

card (set xs) = length (remdups xs)
by (*simp add: length-remdups-card-conv*)

lemma *the-elem-set* [code]:

the-elem (set [x]) = x
by *simp*

lemma *Pow-set* [code]:

Pow (set []) = {{}}
Pow (set (x # xs)) = (let A = Pow (set xs) in A \cup insert x ‘ A)
by (*simp-all add: Pow-insert Let-def*)

definition *map-project* :: (*'a* \Rightarrow *'b option*) \Rightarrow *'a set* \Rightarrow *'b set* **where**
map-project f A = {b. \exists a \in A. f a = Some b}

lemma [code]:

map-project f (set xs) = set (List.map-filter f xs)
by (*auto simp add: map-project-def map-filter-def image-def*)

hide-const (**open**) *map-project*

Operations on relations

lemma *product-code* [code]:

Product-Type.product (set xs) (set ys) = set [(x, y). x \leftarrow xs, y \leftarrow ys]
by (*auto simp add: Product-Type.product-def*)

lemma *Id-on-set* [code]:

Id-on (set xs) = set [(x, x). x \leftarrow xs]
by (*auto simp add: Id-on-def*)

lemma [code]:

R “S = List.map-project ($\lambda(x, y).$ if $x \in S$ then Some y else None) R
unfolding *map-project-def* **by** (*auto split: prod.split if-split-asm*)

lemma *trancl-set-ntrancl* [code]:

trancl (set xs) = ntrancl (card (set xs) - 1) (set xs)
by (*simp add: finite-trancl-ntrancl*)

lemma *set-relcomp* [code]:

set xys O set yzs = set ((fst xy, snd yz). xy ← xys, yz ← yzs, snd xy = fst yz)
by *auto (auto simp add: Bex-def image-def)*

lemma *wf-set* [*code*]:
wf (set xs) = acyclic (set xs)
by (*simp add: wf-iff-acyclic-if-finite*)

67.8 Setup for Lifting/Transfer

67.8.1 Transfer rules for the Transfer package

context *includes lifting-syntax*
begin

lemma *tl-transfer* [*transfer-rule*]:
(list-all2 A ==> list-all2 A) tl tl
unfolding *tl-def[abs-def]* **by** *transfer-prover*

lemma *butlast-transfer* [*transfer-rule*]:
(list-all2 A ==> list-all2 A) butlast butlast
by (*rule rel-funI, erule list-all2-induct, auto*)

lemma *map-rec*: *map f xs = rec-list Nil (%x - y. Cons (f x) y) xs*
by (*induct xs*) *auto*

lemma *append-transfer* [*transfer-rule*]:
(list-all2 A ==> list-all2 A ==> list-all2 A) append append
unfolding *List.append-def* **by** *transfer-prover*

lemma *rev-transfer* [*transfer-rule*]:
(list-all2 A ==> list-all2 A) rev rev
unfolding *List.rev-def* **by** *transfer-prover*

lemma *filter-transfer* [*transfer-rule*]:
((A ==> (=)) ==> list-all2 A ==> list-all2 A) filter filter
unfolding *List.filter-def* **by** *transfer-prover*

lemma *fold-transfer* [*transfer-rule*]:
((A ==> B ==> B) ==> list-all2 A ==> B ==> B) fold fold
unfolding *List.fold-def* **by** *transfer-prover*

lemma *foldr-transfer* [*transfer-rule*]:
((A ==> B ==> B) ==> list-all2 A ==> B ==> B) foldr foldr
unfolding *List.foldr-def* **by** *transfer-prover*

lemma *foldl-transfer* [*transfer-rule*]:
((B ==> A ==> B) ==> B ==> list-all2 A ==> B) foldl foldl
unfolding *List.foldl-def* **by** *transfer-prover*

lemma *concat-transfer* [*transfer-rule*]:

$(list\text{-}all2\ (list\text{-}all2\ A) \implies list\text{-}all2\ A)\ concat\ concat$
unfolding *List.concat-def* **by** *transfer-prover*

lemma *drop-transfer* [*transfer-rule*]:
 $((=) \implies list\text{-}all2\ A \implies list\text{-}all2\ A)\ drop\ drop$
unfolding *List.drop-def* **by** *transfer-prover*

lemma *take-transfer* [*transfer-rule*]:
 $((=) \implies list\text{-}all2\ A \implies list\text{-}all2\ A)\ take\ take$
unfolding *List.take-def* **by** *transfer-prover*

lemma *list-update-transfer* [*transfer-rule*]:
 $(list\text{-}all2\ A \implies (=) \implies A \implies list\text{-}all2\ A)\ list\text{-}update\ list\text{-}update$
unfolding *list-update-def* **by** *transfer-prover*

lemma *takeWhile-transfer* [*transfer-rule*]:
 $((A \implies (=)) \implies list\text{-}all2\ A \implies list\text{-}all2\ A)\ takeWhile\ takeWhile$
unfolding *takeWhile-def* **by** *transfer-prover*

lemma *dropWhile-transfer* [*transfer-rule*]:
 $((A \implies (=)) \implies list\text{-}all2\ A \implies list\text{-}all2\ A)\ dropWhile\ dropWhile$
unfolding *dropWhile-def* **by** *transfer-prover*

lemma *zip-transfer* [*transfer-rule*]:
 $(list\text{-}all2\ A \implies list\text{-}all2\ B \implies list\text{-}all2\ (rel\text{-}prod\ A\ B))\ zip\ zip$
unfolding *zip-def* **by** *transfer-prover*

lemma *product-transfer* [*transfer-rule*]:
 $(list\text{-}all2\ A \implies list\text{-}all2\ B \implies list\text{-}all2\ (rel\text{-}prod\ A\ B))\ List.product\ List.product$
unfolding *List.product-def* **by** *transfer-prover*

lemma *product-lists-transfer* [*transfer-rule*]:
 $(list\text{-}all2\ (list\text{-}all2\ A) \implies list\text{-}all2\ (list\text{-}all2\ A))\ product\text{-}lists\ product\text{-}lists$
unfolding *product-lists-def* **by** *transfer-prover*

lemma *insert-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $(A \implies list\text{-}all2\ A \implies list\text{-}all2\ A)\ List.insert\ List.insert$
unfolding *List.insert-def* [*abs-def*] **by** *transfer-prover*

lemma *find-transfer* [*transfer-rule*]:
 $((A \implies (=)) \implies list\text{-}all2\ A \implies rel\text{-}option\ A)\ List.find\ List.find$
unfolding *List.find-def* **by** *transfer-prover*

lemma *those-transfer* [*transfer-rule*]:
 $(list\text{-}all2\ (rel\text{-}option\ P) \implies rel\text{-}option\ (list\text{-}all2\ P))\ those\ those$
unfolding *List.those-def* **by** *transfer-prover*

lemma *remove1-transfer* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique A*
shows ($A \text{ ===> list-all2 } A \text{ ===> list-all2 } A$) *remove1 remove1*
unfolding *remove1-def* **by** *transfer-prover*

lemma *removeAll-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ($A \text{ ===> list-all2 } A \text{ ===> list-all2 } A$) *removeAll removeAll*
unfolding *removeAll-def* **by** *transfer-prover*

lemma *successively-transfer* [*transfer-rule*]:
 $((A \text{ ===> } A \text{ ===> } (=)) \text{ ===> list-all2 } A \text{ ===> } (=))$ *successively successively*
unfolding *successively-altdef* **by** *transfer-prover*

lemma *distinct-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ($\text{list-all2 } A \text{ ===> } (=)$) *distinct distinct*
unfolding *distinct-def* **by** *transfer-prover*

lemma *distinct-adj-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows ($\text{list-all2 } A \text{ ===> } (=)$) *distinct-adj distinct-adj*
unfolding *rel-fun-def*
proof (*intro allI impI*)
fix *xs ys* **assume** $\text{list-all2 } A \text{ } xs \text{ } ys$
thus $\text{distinct-adj } xs \longleftrightarrow \text{distinct-adj } ys$
proof (*induction rule: list-all2-induct*)
case (*Cons x xs y ys*)
show *?case*
by (*metis Cons assms bi-unique-def distinct-adj-Cons list.rel-sel*)
qed *auto*
qed

lemma *remdups-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ($\text{list-all2 } A \text{ ===> list-all2 } A$) *remdups remdups*
unfolding *remdups-def* **by** *transfer-prover*

lemma *remdups-adj-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ($\text{list-all2 } A \text{ ===> list-all2 } A$) *remdups-adj remdups-adj*
proof (*rule rel-funI, erule list-all2-induct*)
qed (*auto simp: remdups-adj-Cons assms[unfolded bi-unique-def] split: list.splits*)

lemma *replicate-transfer* [*transfer-rule*]:
 $((=) \text{ ===> } A \text{ ===> list-all2 } A)$ *replicate replicate*
unfolding *replicate-def* **by** *transfer-prover*

lemma *length-transfer* [*transfer-rule*]:
 $(\text{list-all2 } A \text{ ===> } (=))$ *length length*

unfolding *size-list-overloaded-def size-list-def* **by** *transfer-prover*

lemma *rotate1-transfer* [*transfer-rule*]:
 (*list-all2 A* \implies *list-all2 A*) *rotate1 rotate1*
unfolding *rotate1-def* **by** *transfer-prover*

lemma *rotate-transfer* [*transfer-rule*]:
 (($=$) \implies *list-all2 A* \implies *list-all2 A*) *rotate rotate*
unfolding *rotate-def* [*abs-def*] **by** *transfer-prover*

lemma *nths-transfer* [*transfer-rule*]:
 (*list-all2 A* \implies *rel-set* ($=$) \implies *list-all2 A*) *nths nths*
unfolding *nths-def* [*abs-def*] **by** *transfer-prover*

lemma *subseqs-transfer* [*transfer-rule*]:
 (*list-all2 A* \implies *list-all2* (*list-all2 A*)) *subseqs subseqs*
unfolding *subseqs-def* [*abs-def*] **by** *transfer-prover*

lemma *partition-transfer* [*transfer-rule*]:
 ((*A* \implies ($=$)) \implies *list-all2 A* \implies *rel-prod* (*list-all2 A*) (*list-all2 A*))
partition partition
unfolding *partition-def* **by** *transfer-prover*

lemma *lists-transfer* [*transfer-rule*]:
 (*rel-set A* \implies *rel-set* (*list-all2 A*)) *lists lists*
proof (*rule rel-funI, rule rel-setI*)
show [$l \in$ *lists X*; *rel-set A X Y*] $\implies \exists y \in$ *lists Y. list-all2 A l y* **for** *X Y l*
proof (*induction l rule: lists.induct*)
case (*Cons a l*)
then show *?case*
by (*simp only: rel-set-def list-all2-Cons1, metis lists.Cons*)
qed *auto*
show [$l \in$ *lists Y*; *rel-set A X Y*] $\implies \exists x \in$ *lists X. list-all2 A x l* **for** *X Y l*
proof (*induction l rule: lists.induct*)
case (*Cons a l*)
then show *?case*
by (*simp only: rel-set-def list-all2-Cons2, metis lists.Cons*)
qed *auto*
qed

lemma *set-Cons-transfer* [*transfer-rule*]:
 (*rel-set A* \implies *rel-set* (*list-all2 A*) \implies *rel-set* (*list-all2 A*))
set-Cons set-Cons
unfolding *rel-fun-def rel-set-def set-Cons-def*
by (*fastforce simp add: list-all2-Cons1 list-all2-Cons2*)

lemma *listset-transfer* [*transfer-rule*]:
 (*list-all2* (*rel-set A*) \implies *rel-set* (*list-all2 A*)) *listset listset*
unfolding *listset-def* **by** *transfer-prover*

lemma *null-transfer* [*transfer-rule*]:
(list-all2 A ===> (=)) List.null List.null
unfolding *rel-fun-def List.null-def* **by** *auto*

lemma *list-all-transfer* [*transfer-rule*]:
((A ===> (=)) ===> list-all2 A ===> (=)) list-all list-all
unfolding *list-all-iff [abs-def]* **by** *transfer-prover*

lemma *list-ex-transfer* [*transfer-rule*]:
((A ===> (=)) ===> list-all2 A ===> (=)) list-ex list-ex
unfolding *list-ex-iff [abs-def]* **by** *transfer-prover*

lemma *splice-transfer* [*transfer-rule*]:
(list-all2 A ===> list-all2 A ===> list-all2 A) splice splice
apply (*rule rel-funI, erule list-all2-induct, simp add: rel-fun-def, simp*)
apply (*rule rel-funI*)
apply (*erule-tac xs=x in list-all2-induct, simp, simp add: rel-fun-def*)
done

lemma *shuffles-transfer* [*transfer-rule*]:
(list-all2 A ===> list-all2 A ===> rel-set (list-all2 A)) shuffles shuffles
proof (*intro rel-funI, goal-cases*)
case (*1 xs xs' ys ys'*)
thus *?case*
proof (*induction xs ys arbitrary: xs' ys' rule: shuffles.induct*)
case (*3 x xs y ys xs' ys'*)
from *3.prem1 obtain x' xs'' where xs': xs' = x' # xs'' by (cases xs') auto*
from *3.prem2 obtain y' ys'' where ys': ys' = y' # ys'' by (cases ys') auto*
have [*transfer-rule*]: *A x x' A y y' list-all2 A xs xs'' list-all2 A ys ys''*
using *3.prem3 by (simp-all add: xs' ys')*
have [*transfer-rule*]: *rel-set (list-all2 A) (shuffles xs (y # ys)) (shuffles xs'' ys')*
and
[transfer-rule]: rel-set (list-all2 A) (shuffles (x # xs) ys) (shuffles xs' ys')
using *3.prem4 by (auto intro!: 3.IH simp: xs' ys')*
have *rel-set (list-all2 A) ((#) x ' shuffles xs (y # ys) ∪ (#) y ' shuffles (x # xs) ys)*
((#) x' ' shuffles xs'' ys' ∪ (#) y' ' shuffles xs' ys'') by transfer-prover
thus *?case by (simp add: xs' ys')*
qed (*auto simp: rel-set-def*)
qed

lemma *rtrancl-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A bi-total A*
shows (*rel-set (rel-prod A A) ===> rel-set (rel-prod A A)*) *rtrancl rtrancl*
unfolding *rtrancl-def* **by** *transfer-prover*

lemma *monotone-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*

shows $((A \text{ ==== } A \text{ ==== } (=)) \text{ ==== } (B \text{ ==== } B \text{ ==== } (=)) \text{ ==== } (A \text{ ==== } B) \text{ ==== } (=))$ *monotone monotone*
unfolding *monotone-def[abs-def]* **by** *transfer-prover*

lemma *fun-ord-parametric [transfer-rule]*:
assumes *[transfer-rule]: bi-total C*
shows $((A \text{ ==== } B \text{ ==== } (=)) \text{ ==== } (C \text{ ==== } A) \text{ ==== } (C \text{ ==== } B) \text{ ==== } (=))$ *fun-ord fun-ord*
unfolding *fun-ord-def[abs-def]* **by** *transfer-prover*

lemma *fun-lub-parametric [transfer-rule]*:
assumes *[transfer-rule]: bi-total A bi-unique A*
shows $((\text{rel-set } A \text{ ==== } B) \text{ ==== } \text{rel-set } (C \text{ ==== } A) \text{ ==== } C \text{ ==== } B)$
fun-lub fun-lub
unfolding *fun-lub-def[abs-def]* **by** *transfer-prover*

end

end

68 Sum and product over lists

theory *Groups-List*
imports *List*
begin

locale *monoid-list = monoid*
begin

definition $F :: 'a \text{ list} \Rightarrow 'a$
where
eq-foldr [code]: F xs = foldr f xs 1

lemma *Nil [simp]*:
 $F [] = 1$
by *(simp add: eq-foldr)*

lemma *Cons [simp]*:
 $F (x \# xs) = x * F xs$
by *(simp add: eq-foldr)*

lemma *append [simp]*:
 $F (xs @ ys) = F xs * F ys$
by *(induct xs) (simp-all add: assoc)*

end

locale *comm-monoid-list = comm-monoid + monoid-list*
begin

```

lemma rev [simp]:
   $F (\text{rev } xs) = F xs$ 
  by (simp add: eq-foldr foldr-fold fold-rev fun-eq-iff assoc left-commute)

end

locale comm-monoid-list-set = list: comm-monoid-list + set: comm-monoid-set
begin

lemma distinct-set-conv-list:
   $\text{distinct } xs \implies \text{set.F } g (\text{set } xs) = \text{list.F } (map\ g\ xs)$ 
  by (induct xs simp-all)

lemma set-conv-list [code]:
   $\text{set.F } g (\text{set } xs) = \text{list.F } (map\ g\ (\text{remdups } xs))$ 
  by (simp add: distinct-set-conv-list [symmetric])

end

68.1 List summation

context monoid-add
begin

sublocale sum-list: monoid-list plus 0
defines
   $\text{sum-list} = \text{sum-list.F } ..$ 

end

context comm-monoid-add
begin

sublocale sum-list: comm-monoid-list plus 0
rewrites
   $\text{monoid-list.F plus } 0 = \text{sum-list}$ 
proof –
  show comm-monoid-list plus 0 ..
  then interpret sum-list: comm-monoid-list plus 0 .
  from sum-list-def show  $\text{monoid-list.F plus } 0 = \text{sum-list}$  by simp
qed

sublocale sum: comm-monoid-list-set plus 0
rewrites
   $\text{monoid-list.F plus } 0 = \text{sum-list}$ 
  and  $\text{comm-monoid-set.F plus } 0 = \text{sum}$ 
proof –
  show comm-monoid-list-set plus 0 ..

```

```

then interpret sum: comm-monoid-list-set plus 0 .
from sum-list-def show monoid-list.F plus 0 = sum-list by simp
from sum-def show comm-monoid-set.F plus 0 = sum by (auto intro: sym)
qed

```

end

Some syntactic sugar for summing a function over a list:

```

syntax (ASCII)
  -sum-list :: pttrn => 'a list => 'b => 'b (( $\exists$ SUM  $\leftarrow$ -. -) [0, 51, 10] 10)
syntax
  -sum-list :: pttrn => 'a list => 'b => 'b (( $\exists$  $\sum$   $\leftarrow$ -. -) [0, 51, 10] 10)
translations — Beware of argument permutation!
   $\sum x \leftarrow xs. b == \text{CONST } \textit{sum-list} (\text{CONST } \textit{map} (\lambda x. b) xs)$ 

```

context

```

includes lifting-syntax
begin

```

```

lemma sum-list-transfer [transfer-rule]:
  (list-all2 A ===> A) sum-list sum-list
  if [transfer-rule]: A 0 0 (A ===> A ===> A) (+) (+)
  unfolding sum-list.eq-foldr [abs-def]
  by transfer-prover

```

end

TODO duplicates

```

lemmas sum-list-simps = sum-list.Nil sum-list.Cons
lemmas sum-list-append = sum-list.append
lemmas sum-list-rev = sum-list.rev

```

```

lemma (in monoid-add) fold-plus-sum-list-rev:
  fold plus xs = plus (sum-list (rev xs))

```

proof

```

fix x
have fold plus xs x = sum-list (rev xs @ [x])
  by (simp add: foldr-conv-fold sum-list.eq-foldr)
also have  $\dots = \textit{sum-list} (\textit{rev } xs) + x$ 
  by simp
finally show fold plus xs x = sum-list (rev xs) + x

```

qed

```

lemma (in comm-monoid-add) sum-list-map-remove1:
   $x \in \textit{set } xs \implies \textit{sum-list} (\textit{map } f xs) = f x + \textit{sum-list} (\textit{map } f (\textit{remove1 } x xs))$ 
  by (induct xs) (auto simp add: ac-simps)

```

```

lemma (in monoid-add) size-list-conv-sum-list:

```

size-list f $xs = \text{sum-list } (\text{map } f \text{ } xs) + \text{size } xs$
by (*induct* xs) *auto*

lemma (*in monoid-add*) *length-concat*:
 $\text{length } (\text{concat } xss) = \text{sum-list } (\text{map } \text{length } xss)$
by (*induct* xss) *simp-all*

lemma (*in monoid-add*) *length-product-lists*:
 $\text{length } (\text{product-lists } xss) = \text{foldr } (*) (\text{map } \text{length } xss) 1$
proof (*induct* xss)
case (*Cons* xs xss) **then show** ?*case* **by** (*induct* xs) (*auto simp: length-concat o-def*)
qed *simp*

lemma (*in monoid-add*) *sum-list-map-filter*:
assumes $\bigwedge x. x \in \text{set } xs \implies \neg P \ x \implies f \ x = 0$
shows $\text{sum-list } (\text{map } f (\text{filter } P \ xs)) = \text{sum-list } (\text{map } f \ xs)$
using *assms* **by** (*induct* xs) *auto*

lemma *sum-list-filter-le-nat*:
fixes $f :: 'a \Rightarrow \text{nat}$
shows $\text{sum-list } (\text{map } f (\text{filter } P \ xs)) \leq \text{sum-list } (\text{map } f \ xs)$
by(*induction* xs ; *simp*)

lemma (*in comm-monoid-add*) *distinct-sum-list-conv-Sum*:
 $\text{distinct } xs \implies \text{sum-list } xs = \text{Sum } (\text{set } xs)$
by (*induct* xs) *simp-all*

lemma *sum-list-upt*[*simp*]:
 $m \leq n \implies \text{sum-list } [m..<n] = \sum \{m..<n\}$
by(*simp* *add: distinct-sum-list-conv-Sum*)

context *ordered-comm-monoid-add*
begin

lemma *sum-list-nonneg*: $(\bigwedge x. x \in \text{set } xs \implies 0 \leq x) \implies 0 \leq \text{sum-list } xs$
by (*induction* xs) *auto*

lemma *sum-list-nonpos*: $(\bigwedge x. x \in \text{set } xs \implies x \leq 0) \implies \text{sum-list } xs \leq 0$
by (*induction* xs) (*auto simp: add-nonpos-nonpos*)

lemma *sum-list-nonneg-eq-0-iff*:
 $(\bigwedge x. x \in \text{set } xs \implies 0 \leq x) \implies \text{sum-list } xs = 0 \iff (\forall x \in \text{set } xs. x = 0)$
by (*induction* xs) (*simp-all add: add-nonneg-eq-0-iff sum-list-nonneg*)

end

context *canonically-ordered-monoid-add*
begin

lemma *sum-list-eq-0-iff* [*simp*]:
 $sum\text{-list } ns = 0 \iff (\forall n \in set\ ns. n = 0)$
by (*simp add: sum-list-nonneg-eq-0-iff*)

lemma *member-le-sum-list*:
 $x \in set\ xs \implies x \leq sum\text{-list } xs$
by (*induction xs*) (*auto simp: add-increasing add-increasing2*)

lemma *elem-le-sum-list*:
 $k < size\ ns \implies ns ! k \leq sum\text{-list } (ns)$
by (*rule member-le-sum-list*) *simp*

end

lemma (*in ordered-cancel-comm-monoid-diff*) *sum-list-update*:
 $k < size\ xs \implies sum\text{-list } (xs[k := x]) = sum\text{-list } xs + x - xs ! k$
apply (*induction xs arbitrary:k*)
apply (*auto simp: add-ac split: nat.split*)
apply (*drule elem-le-sum-list*)
by (*simp add: local.add-diff-assoc local.add-increasing*)

lemma (*in monoid-add*) *sum-list-triv*:
 $(\sum x \leftarrow xs. r) = of\text{-nat } (length\ xs) * r$
by (*induct xs*) (*simp-all add: distrib-right*)

lemma (*in monoid-add*) *sum-list-0* [*simp*]:
 $(\sum x \leftarrow xs. 0) = 0$
by (*induct xs*) (*simp-all add: distrib-right*)

For non-Abelian groups *xs* needs to be reversed on one side:

lemma (*in ab-group-add*) *uminus-sum-list-map*:
 $- sum\text{-list } (map\ f\ xs) = sum\text{-list } (map\ (uminus \circ f)\ xs)$
by (*induct xs*) *simp-all*

lemma (*in comm-monoid-add*) *sum-list-addf*:
 $(\sum x \leftarrow xs. f\ x + g\ x) = sum\text{-list } (map\ f\ xs) + sum\text{-list } (map\ g\ xs)$
by (*induct xs*) (*simp-all add: algebra-simps*)

lemma (*in ab-group-add*) *sum-list-subtractf*:
 $(\sum x \leftarrow xs. f\ x - g\ x) = sum\text{-list } (map\ f\ xs) - sum\text{-list } (map\ g\ xs)$
by (*induct xs*) (*simp-all add: algebra-simps*)

lemma (*in semiring-0*) *sum-list-const-mult*:
 $(\sum x \leftarrow xs. c * f\ x) = c * (\sum x \leftarrow xs. f\ x)$
by (*induct xs*) (*simp-all add: algebra-simps*)

lemma (*in semiring-0*) *sum-list-mult-const*:
 $(\sum x \leftarrow xs. f\ x * c) = (\sum x \leftarrow xs. f\ x) * c$

by (induct xs) (simp-all add: algebra-simps)

lemma (in ordered-ab-group-add-abs) sum-list-abs:

|sum-list xs| ≤ sum-list (map abs xs)

by (induct xs) (simp-all add: order-trans [OF abs-triangle-ineq])

lemma sum-list-mono:

fixes f g :: 'a ⇒ 'b::{monoid-add, ordered-ab-semigroup-add}

shows (∧x. x ∈ set xs ⇒ f x ≤ g x) ⇒ (∑ x←xs. f x) ≤ (∑ x←xs. g x)

by (induct xs) (simp, simp add: add-mono)

lemma sum-list-strict-mono:

fixes f g :: 'a ⇒ 'b::{monoid-add, strict-ordered-ab-semigroup-add}

shows [| xs ≠ []; ∧x. x ∈ set xs ⇒ f x < g x |]

⇒ sum-list (map f xs) < sum-list (map g xs)

proof (induction xs)

case Nil thus ?case by simp

next

case C: (Cons - xs)

show ?case

proof (cases xs)

case Nil thus ?thesis using C.prem by simp

next

case Cons thus ?thesis using C by (simp add: add-strict-mono)

qed

qed

A much more general version of this monotonicity lemma can be formulated with multisets and the multiset order

lemma sum-list-mono2: fixes xs :: 'a :: ordered-comm-monoid-add list

shows [| length xs = length ys; ∧i. i < length xs → xs!i ≤ ys!i |]

⇒ sum-list xs ≤ sum-list ys

apply(induction xs ys rule: list-induct2)

by(auto simp: nth-Cons' less-Suc-eq-0-disj imp-ex add-mono)

lemma (in monoid-add) sum-list-distinct-conv-sum-set:

distinct xs ⇒ sum-list (map f xs) = sum f (set xs)

by (induct xs) simp-all

lemma (in monoid-add) interv-sum-list-conv-sum-set-nat:

sum-list (map f [m.. n]) = sum f (set [m.. n])

by (simp add: sum-list-distinct-conv-sum-set)

lemma (in monoid-add) interv-sum-list-conv-sum-set-int:

sum-list (map f [k..l]) = sum f (set [k..l])

by (simp add: sum-list-distinct-conv-sum-set)

General equivalence between *sum-list* and *sum*

lemma (in monoid-add) sum-list-sum-nth:

$sum-list\ xs = (\sum\ i = 0 \ ..< length\ xs.\ xs\ !\ i)$
using *interv-sum-list-conv-sum-set-nat* [of (!) *xs* 0 *length xs*] **by** (*simp add: map-nth*)

lemma *sum-list-map-eq-sum-count*:

$sum-list\ (map\ f\ xs) = sum\ (\lambda x.\ count-list\ xs\ x * f\ x)\ (set\ xs)$

proof(*induction xs*)

case (*Cons x xs*)

show *?case* (**is** *?l = ?r*)

proof *cases*

assume $x \in set\ xs$

have $?l = f\ x + (\sum\ x \in set\ xs.\ count-list\ xs\ x * f\ x)$ **by** (*simp add: Cons.IH*)

also have $set\ xs = insert\ x\ (set\ xs - \{x\})$ **using** $\langle x \in set\ xs \rangle$ **by** *blast*

also have $f\ x + (\sum\ x \in insert\ x\ (set\ xs - \{x\}).\ count-list\ xs\ x * f\ x) = ?r$

by (*simp add: sum.insert-remove eq-commute*)

finally show *?thesis* .

next

assume $x \notin set\ xs$

hence $\bigwedge xa.\ xa \in set\ xs \implies x \neq xa$ **by** *blast*

thus *?thesis* **by** (*simp add: Cons.IH* $\langle x \notin set\ xs \rangle$)

qed

qed *simp*

lemma *sum-list-map-eq-sum-count2*:

assumes $set\ xs \subseteq X$ *finite X*

shows $sum-list\ (map\ f\ xs) = sum\ (\lambda x.\ count-list\ xs\ x * f\ x)\ X$

proof–

let $?F = \lambda x.\ count-list\ xs\ x * f\ x$

have $sum\ ?F\ X = sum\ ?F\ (set\ xs \cup (X - set\ xs))$

using *Un-absorb1*[*OF assms(1)*] **by**(*simp*)

also have $\dots = sum\ ?F\ (set\ xs)$

using *assms(2)*

by(*simp add: sum.union-disjoint*[*OF - - Diff-disjoint*] *del: Un-Diff-cancel*)

finally show *?thesis* **by**(*simp add:sum-list-map-eq-sum-count*)

qed

lemma *sum-list-replicate*: $sum-list\ (replicate\ n\ c) = of-nat\ n * c$

by(*induction n*)(*auto simp add: distrib-right*)

lemma *sum-list-nonneg*:

$(\bigwedge x.\ x \in set\ xs \implies (x :: 'a :: ordered-comm-monoid-add) \geq 0) \implies sum-list\ xs \geq 0$

by (*induction xs*) *simp-all*

lemma *sum-list-Suc*:

$sum-list\ (map\ (\lambda x.\ Suc(f\ x))\ xs) = sum-list\ (map\ f\ xs) + length\ xs$

by(*induction xs; simp*)

lemma (in monoid-add) sum-list-map-filter':
 $sum\text{-list} (map\ f (filter\ P\ xs)) = sum\text{-list} (map\ (\lambda x. if\ P\ x\ then\ f\ x\ else\ 0)\ xs)$
 by (induction xs) simp-all

Summation of a strictly ascending sequence with length n can be upper-bounded by summation over $\{0..<n\}$.

lemma sorted-wrt-less-sum-mono-lowerbound:
 fixes $f :: nat \Rightarrow ('b::ordered-comm-monoid-add)$
 assumes mono: $\bigwedge x\ y. x \leq y \implies f\ x \leq f\ y$
 shows sorted-wrt ($<$) ns \implies
 $(\sum_{i \in \{0..<length\ ns\}} f\ i) \leq (\sum_{i \leftarrow ns} f\ i)$
proof (induction ns rule: rev-induct)
 case Nil
 then show ?case by simp
next
 case (snoc n ns)
 have $sum\ f\ \{0..<length\ (ns\ @\ [n])\}$
 $= sum\ f\ \{0..<length\ ns\} + f\ (length\ ns)$
 by simp
 also have $sum\ f\ \{0..<length\ ns\} \leq sum\text{-list}\ (map\ f\ ns)$
 using snoc by (auto simp: sorted-wrt-append)
 also have $length\ ns \leq n$
 using sorted-wrt-less-idx[OF snoc.prem1, of length ns] by auto
 finally have $sum\ f\ \{0..<length\ (ns\ @\ [n])\} \leq sum\text{-list}\ (map\ f\ ns) + f\ n$
 using mono add-mono by blast
 thus ?case by simp
qed

68.2 Horner sums

context comm-semiring-0
begin

definition horner-sum :: $\langle ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'a \rangle$
 where horner-sum-foldr: $\langle horner\text{-sum}\ f\ a\ xs = foldr\ (\lambda x\ b. f\ x + a * b)\ xs\ 0 \rangle$

lemma horner-sum-simps [simp]:
 $\langle horner\text{-sum}\ f\ a\ [] = 0 \rangle$
 $\langle horner\text{-sum}\ f\ a\ (x \# xs) = f\ x + a * horner\text{-sum}\ f\ a\ xs \rangle$
 by (simp-all add: horner-sum-foldr)

lemma horner-sum-eq-sum-funpow:
 $\langle horner\text{-sum}\ f\ a\ xs = (\sum_{n = 0..<length\ xs} ((*)\ a\ \sim n)\ (f\ (xs\ !\ n))) \rangle$
proof (induction xs)
 case Nil
 then show ?case
 by simp
next
 case (Cons x xs)

```

then show ?case
  by (simp add: sum.atLeast0-lessThan-Suc-shift sum-distrib-left del: sum.op-ivl-Suc)
qed

```

```

end

```

```

context
  includes lifting-syntax
begin

```

```

lemma horner-sum-transfer [transfer-rule]:
  ⟨((B ==> A) ==> A ==> list-all2 B ==> A) horner-sum horner-sum⟩
  if [transfer-rule]: ⟨A 0 0⟩
    and [transfer-rule]: ⟨(A ==> A ==> A) (+) (+)⟩
    and [transfer-rule]: ⟨(A ==> A ==> A) (*) (*)⟩
  by (unfold horner-sum-foldr) transfer-prover

```

```

end

```

```

context comm-semiring-1
begin

```

```

lemma horner-sum-eq-sum:
  ⟨horner-sum f a xs = (∑ n = 0..<length xs. f (xs ! n) * a ^ n)⟩
proof –
  have ⟨(*) a ^ n = (*) (a ^ n)⟩ for n
    by (induction n) (simp-all add: ac-simps)
  then show ?thesis
    by (simp add: horner-sum-eq-sum-funpow ac-simps)
qed

```

```

lemma horner-sum-append:
  ⟨horner-sum f a (xs @ ys) = horner-sum f a xs + a ^ length xs * horner-sum f a
  ys⟩
  using sum.atLeastLessThan-shift-bounds [of - 0 <length xs> <length ys>]
    atLeastLessThan-add-Un [of 0 <length xs> <length ys>]
  by (simp add: horner-sum-eq-sum sum-distrib-left sum.union-disjoint ac-simps
  nth-append power-add)

```

```

end

```

```

context linordered-semidom
begin

```

```

lemma horner-sum-nonnegative:
  ⟨0 ≤ horner-sum of-bool 2 bs⟩
  by (induction bs) simp-all

```

```

end

```

context *unique-euclidean-semiring-numeral*
begin

lemma *horner-sum-bound*:
 $\langle \text{horner-sum of-bool } 2 \text{ } bs < 2 \wedge \text{length } bs \rangle$
proof (*induction bs*)
 case *Nil*
 then show *?case*
 by *simp*
next
 case (*Cons b bs*)
 moreover define *a* **where** $\langle a = 2 \wedge \text{length } bs - \text{horner-sum of-bool } 2 \text{ } bs \rangle$
 ultimately have ***: $\langle 2 \wedge \text{length } bs = \text{horner-sum of-bool } 2 \text{ } bs + a \rangle$
 by *simp*
 have $\langle 1 < a * 2 \rangle$ **if** $\langle 0 < a \rangle$
 using *that add-mono [of 1 a 1 a]*
 by (*simp add: mult-2-right discrete*)
 with *Cons* **show** *?case*
 by (*simp add: algebra-simps **)
qed
end

lemma *nat-horner-sum [simp]*:
 $\langle \text{nat } (\text{horner-sum of-bool } 2 \text{ } bs) = \text{horner-sum of-bool } 2 \text{ } bs \rangle$
 by (*induction bs*) (*auto simp add: nat-add-distrib horner-sum-nonnegative*)

context *unique-euclidean-semiring-numeral*
begin

lemma *horner-sum-less-eq-iff-lexordp-eq*:
 $\langle \text{horner-sum of-bool } 2 \text{ } bs \leq \text{horner-sum of-bool } 2 \text{ } cs \iff \text{lexordp-eq } (\text{rev } bs) (\text{rev } cs) \rangle$
 if $\langle \text{length } bs = \text{length } cs \rangle$
proof –
 have $\langle \text{horner-sum of-bool } 2 \text{ } (\text{rev } bs) \leq \text{horner-sum of-bool } 2 \text{ } (\text{rev } cs) \iff \text{lex-ordp-eq } bs \text{ } cs \rangle$
 if $\langle \text{length } bs = \text{length } cs \rangle$ **for** *bs cs*
 using that proof (*induction bs cs rule: list-induct2*)
 case *Nil*
 then show *?case*
 by *simp*
next
 case (*Cons b bs c cs*)
 with *horner-sum-nonnegative [of <rev bs>]* *horner-sum-nonnegative [of <rev cs>]*
 horner-sum-bound [of <rev bs>] *horner-sum-bound [of <rev cs>]*
 show *?case*
 by (*auto simp add: horner-sum-append not-le Cons intro: add-strict-increasing2*)

```

add-increasing)
  qed
  from that this [of ⟨rev bs⟩ ⟨rev cs⟩] show ?thesis
    by simp
  qed

lemma horner-sum-less-iff-lexordp:
  ⟨horner-sum of-bool 2 bs < horner-sum of-bool 2 cs ⟷ ord-class.lexordp (rev bs)
  (rev cs)⟩
  if ⟨length bs = length cs⟩
  proof –
    have ⟨horner-sum of-bool 2 (rev bs) < horner-sum of-bool 2 (rev cs) ⟷ ord-class.lexordp
    bs cs⟩
      if ⟨length bs = length cs⟩ for bs cs
    using that proof (induction bs cs rule: list-induct2)
      case Nil
      then show ?case
        by simp
    next
      case (Cons b bs c cs)
      with horner-sum-nonnegative [of ⟨rev bs⟩] horner-sum-nonnegative [of ⟨rev cs⟩]
        horner-sum-bound [of ⟨rev bs⟩] horner-sum-bound [of ⟨rev cs⟩]
      show ?case
        by (auto simp add: horner-sum-append not-less Cons intro: add-strict-increasing2
        add-increasing)
    qed
  from that this [of ⟨rev bs⟩ ⟨rev cs⟩] show ?thesis
    by simp
  qed

end

```

68.3 Further facts about *List.n-lists*

```

lemma length-n-lists: length (List.n-lists n xs) = length xs ^ n
  by (induct n) (auto simp add: comp-def length-concat sum-list-triv)

```

```

lemma distinct-n-lists:
  assumes distinct xs
  shows distinct (List.n-lists n xs)
  proof (rule card-distinct)
    from assms have card-length: card (set xs) = length xs by (rule distinct-card)
    have card (set (List.n-lists n xs)) = card (set xs) ^ n
    proof (induct n)
      case 0 then show ?case by simp
    next
      case (Suc n)
      moreover have card (⋃ ys∈set (List.n-lists n xs). (λy. y # ys) ‘ set xs)
        = (∑ ys∈set (List.n-lists n xs). card ((λy. y # ys) ‘ set xs))

```

```

    by (rule card-UN-disjoint) auto
  moreover have  $\bigwedge ys. \text{card } ((\lambda y. y \# ys) \text{ ` set } xs) = \text{card } (\text{set } xs)$ 
    by (rule card-image) (simp add: inj-on-def)
  ultimately show ?case by auto
qed
also have  $\dots = \text{length } xs \wedge n$  by (simp add: card-length)
finally show  $\text{card } (\text{set } (\text{List.n-lists } n \text{ } xs)) = \text{length } (\text{List.n-lists } n \text{ } xs)$ 
  by (simp add: length-n-lists)
qed

```

68.4 Tools setup

lemmas *sum-code = sum.set-conv-list*

lemma *sum-set-upto-conv-sum-list-int* [code-unfold]:
 $\text{sum } f \text{ (set } [i..j::\text{int}]) = \text{sum-list } (\text{map } f [i..j])$
 by (simp add: interv-sum-list-conv-sum-set-int)

lemma *sum-set-upt-conv-sum-list-nat* [code-unfold]:
 $\text{sum } f \text{ (set } [m..<n]) = \text{sum-list } (\text{map } f [m..<n])$
 by (simp add: interv-sum-list-conv-sum-set-nat)

68.5 List product

context *monoid-mult*
begin

sublocale *prod-list: monoid-list times 1*
defines
 $\text{prod-list} = \text{prod-list.F } ..$

end

context *comm-monoid-mult*
begin

sublocale *prod-list: comm-monoid-list times 1*
rewrites
 $\text{monoid-list.F times 1} = \text{prod-list}$

proof –
show *comm-monoid-list times 1* ..
then interpret *prod-list: comm-monoid-list times 1* .
from *prod-list-def* **show** $\text{monoid-list.F times 1} = \text{prod-list}$ **by** *simp*
qed

sublocale *prod: comm-monoid-list-set times 1*
rewrites
 $\text{monoid-list.F times 1} = \text{prod-list}$
and $\text{comm-monoid-set.F times 1} = \text{prod}$
proof –


```

show comm-monoid-list-set times 1 ..
then interpret prod: comm-monoid-list-set times 1 .
from prod-list-def show monoid-list.F times 1 = prod-list by simp
from prod-def show comm-monoid-set.F times 1 = prod by (auto intro: sym)
qed

```

end

Some syntactic sugar:

```

syntax (ASCII)
  -prod-list :: pptrn ==> 'a list ==> 'b ==> 'b ((3PROD -<--. -) [0, 51, 10] 10)
syntax
  -prod-list :: pptrn ==> 'a list ==> 'b ==> 'b ((3[] <--. -) [0, 51, 10] 10)
translations — Beware of argument permutation!
   $\prod x \leftarrow xs. b \iff \text{CONST } prod\text{-list } (\text{CONST } map (\lambda x. b) xs)$ 

```

context

```

includes lifting-syntax
begin

```

```

lemma prod-list-transfer [transfer-rule]:
  (list-all2 A ===> A) prod-list prod-list
  if [transfer-rule]: A 1 1 (A ===> A ===> A) (*) (*)
  unfolding prod-list.eq-foldr [abs-def]
  by transfer-prover

```

end

```

lemma prod-list-zero-iff:
  prod-list xs = 0 <=> (0 :: 'a :: {semiring-no-zero-divisors, semiring-1}) ∈ set xs
  by (induction xs) simp-all

```

end

69 Bit operations in suitable algebraic structures

theory *Bit-Operations*

```

imports Presburger Groups-List
begin

```

69.1 Abstract bit structures

```

class semiring-bits = semiring-parity +
  assumes bits-induct [case-names stable rec]:
     $\langle (\bigwedge a. a \text{ div } 2 = a \implies P a) \implies (\bigwedge a b. P a \implies (\text{of-bool } b + 2 * a) \text{ div } 2 = a \implies P (\text{of-bool } b + 2 * a)) \implies P a \rangle$ 
  assumes bits-div-0 [simp]: <0 div a = 0>
  and bits-div-by-1 [simp]: <a div 1 = a>

```

```

and bits-mod-div-trivial [simp]:  $\langle a \bmod b \text{ div } b = 0 \rangle$ 
and even-succ-div-2 [simp]:  $\langle \text{even } a \implies (1 + a) \text{ div } 2 = a \text{ div } 2 \rangle$ 
and even-mask-div-iff:  $\langle \text{even } ((2^m - 1) \text{ div } 2^n) \iff 2^n = 0 \vee m \leq n \rangle$ 
and exp-div-exp-eq:  $\langle 2^m \text{ div } 2^n = \text{of-bool } (2^m \neq 0 \wedge m \geq n) * 2^{(m - n)} \rangle$ 
and div-exp-eq:  $\langle a \text{ div } 2^m \text{ div } 2^n = a \text{ div } 2^{(m + n)} \rangle$ 
and mod-exp-eq:  $\langle a \bmod 2^m \bmod 2^n = a \bmod 2^{\min m n} \rangle$ 
and mult-exp-mod-exp-eq:  $\langle m \leq n \implies (a * 2^m) \bmod (2^n) = (a \bmod 2^{(n - m)}) * 2^m \rangle$ 
and div-exp-mod-exp-eq:  $\langle a \text{ div } 2^n \bmod 2^m = a \bmod (2^{(n + m)}) \text{ div } 2^n \rangle$ 
and even-mult-exp-div-exp-iff:  $\langle \text{even } (a * 2^m \text{ div } 2^n) \iff m > n \vee 2^n = 0 \vee (m \leq n \wedge \text{even } (a \text{ div } 2^{(n - m)})) \rangle$ 
fixes bit ::  $\langle 'a \implies \text{nat} \implies \text{bool} \rangle$ 
assumes bit-iff-odd:  $\langle \text{bit } a \ n \iff \text{odd } (a \text{ div } 2^n) \rangle$ 
begin

```

Having *bit* as definitional class operation takes into account that specific instances can be implemented differently wrt. code generation.

```

lemma bits-div-by-0 [simp]:
   $\langle a \text{ div } 0 = 0 \rangle$ 
by (metis add-cancel-right-right bits-mod-div-trivial mod-mult-div-eq mult-not-zero)

```

```

lemma bits-1-div-2 [simp]:
   $\langle 1 \text{ div } 2 = 0 \rangle$ 
using even-succ-div-2 [of 0] by simp

```

```

lemma bits-1-div-exp [simp]:
   $\langle 1 \text{ div } 2^n = \text{of-bool } (n = 0) \rangle$ 
using div-exp-eq [of 1 1] by (cases n) simp-all

```

```

lemma even-succ-div-exp [simp]:
   $\langle (1 + a) \text{ div } 2^n = a \text{ div } 2^n \rangle$  if  $\langle \text{even } a \rangle$  and  $\langle n > 0 \rangle$ 

```

```

proof (cases n)
  case 0
    with that show ?thesis
    by simp
next
  case (Suc n)
    with  $\langle \text{even } a \rangle$  have  $\langle (1 + a) \text{ div } 2^{\text{Suc } n} = a \text{ div } 2^{\text{Suc } n} \rangle$ 
    proof (induction n)
      case 0
        then show ?case
        by simp
    next
    case (Suc n)
      then show ?case
      using div-exp-eq [of - 1]  $\langle \text{Suc } n \rangle$ , symmetric

```

```

    by simp
  qed
  with Suc show ?thesis
    by simp
  qed

```

```

lemma even-succ-mod-exp [simp]:
   $\langle (1 + a) \bmod 2^n = 1 + (a \bmod 2^n) \rangle$  if  $\langle \text{even } a \rangle$  and  $\langle n > 0 \rangle$ 
  using div-mult-mod-eq [of  $\langle 1 + a \rangle \langle 2^n \rangle$ ] that
  apply simp
  by (metis local.add.left-commute local.add-left-cancel local.div-mult-mod-eq)

```

```

lemma bits-mod-by-1 [simp]:
   $\langle a \bmod 1 = 0 \rangle$ 
  using div-mult-mod-eq [of  $a \ 1$ ] by simp

```

```

lemma bits-mod-0 [simp]:
   $\langle 0 \bmod a = 0 \rangle$ 
  using div-mult-mod-eq [of  $0 \ a$ ] by simp

```

```

lemma bits-one-mod-two-eq-one [simp]:
   $\langle 1 \bmod 2 = 1 \rangle$ 
  by (simp add: mod2-eq-if)

```

```

lemma bit-0:
   $\langle \text{bit } a \ 0 \longleftrightarrow \text{odd } a \rangle$ 
  by (simp add: bit-iff-odd)

```

```

lemma bit-Suc:
   $\langle \text{bit } a \ (\text{Suc } n) \longleftrightarrow \text{bit } (a \text{ div } 2) \ n \rangle$ 
  using div-exp-eq [of  $a \ 1 \ n$ ] by (simp add: bit-iff-odd)

```

```

lemma bit-rec:
   $\langle \text{bit } a \ n \longleftrightarrow (\text{if } n = 0 \text{ then odd } a \text{ else bit } (a \text{ div } 2) \ (n - 1)) \rangle$ 
  by (cases  $n$ ) (simp-all add: bit-Suc bit-0)

```

```

lemma bit-0-eq [simp]:
   $\langle \text{bit } 0 = \text{bot} \rangle$ 
  by (simp add: fun-eq-iff bit-iff-odd)

```

```

context
  fixes  $a$ 
  assumes stable:  $\langle a \text{ div } 2 = a \rangle$ 
begin

```

```

lemma bits-stable-imp-add-self:
   $\langle a + a \bmod 2 = 0 \rangle$ 
proof –
  have  $\langle a \text{ div } 2 * 2 + a \bmod 2 = a \rangle$ 

```

```

  by (fact div-mult-mod-eq)
  then have ⟨a * 2 + a mod 2 = a⟩
    by (simp add: stable)
  then show ?thesis
    by (simp add: mult-2-right ac-simps)
qed

```

```

lemma stable-imp-bit-iff-odd:
  ⟨bit a n ⟷ odd a⟩
  by (induction n) (simp-all add: stable bit-Suc bit-0)

```

end

```

lemma bit-iff-odd-imp-stable:
  ⟨a div 2 = a⟩ if ⟨∧n. bit a n ⟷ odd a⟩
using that proof (induction a rule: bits-induct)
  case (stable a)
  then show ?case
    by simp
next
  case (rec a b)
  from rec.prem1 [of 1] have [simp]: ⟨b = odd a⟩
    by (simp add: rec.hyps bit-Suc bit-0)
  from rec.hyps have hyp: ⟨of-bool (odd a) + 2 * a div 2 = a⟩
    by simp
  have ⟨bit a n ⟷ odd a⟩ for n
    using rec.prem1 [of ⟨Suc n⟩] by (simp add: hyp bit-Suc)
  then have ⟨a div 2 = a⟩
    by (rule rec.IH)
  then have ⟨of-bool (odd a) + 2 * a = 2 * (a div 2) + of-bool (odd a)⟩
    by (simp add: ac-simps)
  also have ⟨... = a⟩
    using mult-div-mod-eq [of 2 a]
    by (simp add: of-bool-odd-eq-mod-2)
  finally show ?case
    using ⟨a div 2 = a⟩ by (simp add: hyp)
qed

```

```

lemma exp-eq-0-imp-not-bit:
  ⟨¬ bit a n⟩ if ⟨2 ^ n = 0⟩
  using that by (simp add: bit-iff-odd)

```

definition

```

possible-bit :: 'a itself ⇒ nat ⇒ bool
where
  possible-bit tyrep n = (2 ^ n ≠ (0 :: 'a))

```

```

lemma possible-bit-0[simp]:
  possible-bit ty 0

```

by (simp add: possible-bit-def)

lemma fold-possible-bit:

$2 \wedge n = (0 :: 'a) \longleftrightarrow \neg \text{possible-bit TYPE('a)} n$

by (simp add: possible-bit-def)

lemmas impossible-bit = exp-eq-0-imp-not-bit[simplified fold-possible-bit]

lemma bit-imp-possible-bit:

$\text{bit } a \ n \Longrightarrow \text{possible-bit TYPE('a)} n$

by (rule ccontr) (simp add: impossible-bit)

lemma possible-bit-less-imp:

$\text{possible-bit tyrep } i \Longrightarrow j \leq i \Longrightarrow \text{possible-bit tyrep } j$

using power-add[of 2 :: 'a j i - j]

by (clarsimp simp: possible-bit-def eq-commute[where a=0])

lemma possible-bit-min[simp]:

$\text{possible-bit tyrep } (\min i j) \longleftrightarrow \text{possible-bit tyrep } i \vee \text{possible-bit tyrep } j$

by (auto simp: min-def elim: possible-bit-less-imp)

lemma bit-eqI:

$\langle a = b \rangle \text{ if } \langle \bigwedge n. \text{possible-bit TYPE('a)} n \Longrightarrow \text{bit } a \ n \longleftrightarrow \text{bit } b \ n \rangle$

proof -

have $\langle \text{bit } a \ n \longleftrightarrow \text{bit } b \ n \rangle$ for n

proof (cases $\langle 2 \wedge n = 0 \rangle$)

case True

then show ?thesis

by (simp add: exp-eq-0-imp-not-bit)

next

case False

then show ?thesis

by (rule that[unfolded possible-bit-def])

qed

then show ?thesis proof (induction a arbitrary: b rule: bits-induct)

case (stable a)

from stable(2) [of 0] have **: $\langle \text{even } b \longleftrightarrow \text{even } a \rangle$

by (simp add: bit-0)

have $\langle b \text{ div } 2 = a \rangle$

proof (rule bit-iff-idd-imp-stable)

fix n

from stable have *: $\langle \text{bit } b \ n \longleftrightarrow \text{bit } a \ n \rangle$

by simp

also have $\langle \text{bit } a \ n \longleftrightarrow \text{odd } a \rangle$

using stable by (simp add: stable-imp-bit-iff-odd)

finally show $\langle \text{bit } b \ n \longleftrightarrow \text{odd } b \rangle$

by (simp add: **)

qed

from ** have $\langle a \text{ mod } 2 = b \text{ mod } 2 \rangle$

```

    by (simp add: mod2-eq-if)
  then have  $\langle a \bmod 2 + (a + b) = b \bmod 2 + (a + b) \rangle$ 
    by simp
  then have  $\langle a + a \bmod 2 + b = b + b \bmod 2 + a \rangle$ 
    by (simp add: ac-simps)
  with  $\langle a \bmod 2 = a \rangle \langle b \bmod 2 = b \rangle$  show ?case
    by (simp add: bits-stable-imp-add-self)
next
case (rec a p)
from rec.prem1 [of 0] have [simp]:  $\langle p = \text{odd } b \rangle$ 
  by (simp add: bit-0)
from rec.hyps have  $\langle \text{bit } a \ n \longleftrightarrow \text{bit } (b \bmod 2) \ n \rangle$  for  $n$ 
  using rec.prem1 [of  $\langle \text{Suc } n \rangle$ ] by (simp add: bit-Suc)
then have  $\langle a = b \bmod 2 \rangle$ 
  by (rule rec.IH)
then have  $\langle 2 * a = 2 * (b \bmod 2) \rangle$ 
  by simp
then have  $\langle b \bmod 2 + 2 * a = b \bmod 2 + 2 * (b \bmod 2) \rangle$ 
  by simp
also have  $\langle \dots = b \rangle$ 
  by (fact mod-mult-div-eq)
finally show ?case
  by (auto simp add: mod2-eq-if)
qed
qed

```

lemma *bit-eq-iff*:

```

 $\langle a = b \longleftrightarrow (\forall n. \text{possible-bit TYPE('a)} \ n \longrightarrow \text{bit } a \ n \longleftrightarrow \text{bit } b \ n) \rangle$ 
  by (auto intro: bit-eqI)

```

named-theorems *bit-simps* $\langle \text{Simplification rules for } \mathbf{const} \langle \text{bit} \rangle \rangle$

lemma *bit-exp-iff* [*bit-simps*]:

```

 $\langle \text{bit } (2 \wedge m) \ n \longleftrightarrow \text{possible-bit TYPE('a)} \ n \wedge m = n \rangle$ 
  by (auto simp add: bit-iff-odd exp-div-exp-eq possible-bit-def)

```

lemma *bit-1-iff* [*bit-simps*]:

```

 $\langle \text{bit } 1 \ n \longleftrightarrow n = 0 \rangle$ 
  using bit-exp-iff [of 0  $n$ ]
  by auto

```

lemma *bit-2-iff* [*bit-simps*]:

```

 $\langle \text{bit } 2 \ n \longleftrightarrow \text{possible-bit TYPE('a)} \ 1 \wedge n = 1 \rangle$ 
  using bit-exp-iff [of 1  $n$ ] by auto

```

lemma *even-bit-succ-iff*:

```

 $\langle \text{bit } (1 + a) \ n \longleftrightarrow \text{bit } a \ n \vee n = 0 \rangle$  if  $\langle \text{even } a \rangle$ 
  using that by (cases  $\langle n = 0 \rangle$ ) (simp-all add: bit-iff-odd)

```

lemma *bit-double-iff* [*bit-simps*]:

$\langle \text{bit } (2 * a) n \longleftrightarrow \text{bit } a (n - 1) \wedge n \neq 0 \wedge \text{possible-bit TYPE}(a) n \rangle$
using *even-mult-exp-div-exp-iff* [*of a 1 n*]
by (*cases n, auto simp add: bit-iff-odd ac-simps possible-bit-def*)

lemma *odd-bit-iff-bit-pred*:

$\langle \text{bit } a n \longleftrightarrow \text{bit } (a - 1) n \vee n = 0 \rangle$ **if** $\langle \text{odd } a \rangle$

proof –

from $\langle \text{odd } a \rangle$ **obtain** *b* **where** $\langle a = 2 * b + 1 \rangle$..
moreover have $\langle \text{bit } (2 * b) n \vee n = 0 \longleftrightarrow \text{bit } (1 + 2 * b) n \rangle$
using *even-bit-succ-iff* **by** *simp*
ultimately show *?thesis* **by** (*simp add: ac-simps*)

qed

lemma *bit-eq-rec*:

$\langle a = b \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b) \wedge a \text{ div } 2 = b \text{ div } 2 \rangle$ (**is** $\langle ?P = ?Q \rangle$)

proof

assume *?P*
then show *?Q*
by *simp*

next

assume *?Q*
then have $\langle \text{even } a \longleftrightarrow \text{even } b \rangle$ **and** $\langle a \text{ div } 2 = b \text{ div } 2 \rangle$
by *simp-all*

show *?P*

proof (*rule bit-eqI*)

fix *n*

show $\langle \text{bit } a n \longleftrightarrow \text{bit } b n \rangle$

proof (*cases n*)

case *0*

with $\langle \text{even } a \longleftrightarrow \text{even } b \rangle$ **show** *?thesis*

by (*simp add: bit-0*)

next

case (*Suc n*)

moreover from $\langle a \text{ div } 2 = b \text{ div } 2 \rangle$ **have** $\langle \text{bit } (a \text{ div } 2) n = \text{bit } (b \text{ div } 2) n \rangle$

by *simp*

ultimately show *?thesis*

by (*simp add: bit-Suc*)

qed

qed

qed

lemma *bit-mod-2-iff* [*simp*]:

$\langle \text{bit } (a \text{ mod } 2) n \longleftrightarrow n = 0 \wedge \text{odd } a \rangle$

by (*cases a rule: parity-cases*) (*simp-all add: bit-iff-odd*)

lemma *bit-mask-sub-iff*:

$\langle \text{bit } (2 \wedge m - 1) n \longleftrightarrow \text{possible-bit TYPE}(a) n \wedge n < m \rangle$

by (*simp add: bit-iff-odd even-mask-div-iff not-le possible-bit-def*)

lemma *exp-add-not-zero-imp*:

$\langle 2^m \neq 0 \rangle$ and $\langle 2^n \neq 0 \rangle$ if $\langle 2^{m+n} \neq 0 \rangle$

proof –

have $\langle \neg (2^m = 0 \vee 2^n = 0) \rangle$

proof (*rule notI*)

assume $\langle 2^m = 0 \vee 2^n = 0 \rangle$

then have $\langle 2^{m+n} = 0 \rangle$

by (*rule disjE*) (*simp-all add: power-add*)

with that show *False* ..

qed

then show $\langle 2^m \neq 0 \rangle$ and $\langle 2^n \neq 0 \rangle$

by *simp-all*

qed

lemma *bit-disjunctive-add-iff*:

$\langle \text{bit } (a + b) \ n \longleftrightarrow \text{bit } a \ n \vee \text{bit } b \ n \rangle$

if $\langle \bigwedge n. \neg \text{bit } a \ n \vee \neg \text{bit } b \ n \rangle$

proof (*cases* $\langle 2^n = 0 \rangle$)

case *True*

then show *?thesis*

by (*simp add: exp-eq-0-imp-not-bit*)

next

case *False*

with that show *?thesis* **proof** (*induction n arbitrary: a b*)

case *0*

from *0.premis(1)* [*of 0*] **show** *?case*

by (*auto simp add: bit-0*)

next

case (*Suc n*)

from *Suc.premis(1)* [*of 0*] **have** *even*: $\langle \text{even } a \vee \text{even } b \rangle$

by (*auto simp add: bit-0*)

have *bit*: $\langle \neg \text{bit } (a \text{ div } 2) \ n \vee \neg \text{bit } (b \text{ div } 2) \ n \rangle$ **for** *n*

using *Suc.premis(1)* [*of* $\langle \text{Suc } n \rangle$] **by** (*simp add: bit-Suc*)

from *Suc.premis(2)* **have** $\langle 2 * 2^n \neq 0 \rangle$ $\langle 2^n \neq 0 \rangle$

by (*auto simp add: mult-2*)

have $\langle a + b = (a \text{ div } 2 * 2 + a \text{ mod } 2) + (b \text{ div } 2 * 2 + b \text{ mod } 2) \rangle$

using *div-mult-mod-eq* [*of a 2*] *div-mult-mod-eq* [*of b 2*] **by** *simp*

also have $\langle \dots = \text{of-bool } (\text{odd } a \vee \text{odd } b) + 2 * (a \text{ div } 2 + b \text{ div } 2) \rangle$

using *even* **by** (*auto simp add: algebra-simps mod2-eq-if*)

finally have $\langle \text{bit } ((a + b) \text{ div } 2) \ n \longleftrightarrow \text{bit } (a \text{ div } 2 + b \text{ div } 2) \ n \rangle$

using $\langle 2 * 2^n \neq 0 \rangle$ **by** *simp* (*simp-all flip: bit-Suc add: bit-double-iff*

possible-bit-def)

also have $\langle \dots \longleftrightarrow \text{bit } (a \text{ div } 2) \ n \vee \text{bit } (b \text{ div } 2) \ n \rangle$

using *bit* $\langle 2^n \neq 0 \rangle$ **by** (*rule Suc.IH*)

finally show *?case*

by (*simp add: bit-Suc*)

qed

qed

lemma

exp-add-not-zero-imp-left: $\langle 2^m \neq 0 \rangle$
and *exp-add-not-zero-imp-right*: $\langle 2^n \neq 0 \rangle$
if $\langle 2^{m+n} \neq 0 \rangle$

proof –

have $\langle \neg (2^m = 0 \vee 2^n = 0) \rangle$

proof (*rule notI*)

assume $\langle 2^m = 0 \vee 2^n = 0 \rangle$

then have $\langle 2^{m+n} = 0 \rangle$

by (*rule disjE*) (*simp-all add: power-add*)

with that show *False* ..

qed

then show $\langle 2^m \neq 0 \rangle$ **and** $\langle 2^n \neq 0 \rangle$

by *simp-all*

qed

lemma *exp-not-zero-imp-exp-diff-not-zero*:

$\langle 2^{n-m} \neq 0 \rangle$ **if** $\langle 2^n \neq 0 \rangle$

proof (*cases* $\langle m \leq n \rangle$)

case *True*

moreover define *q* **where** $\langle q = n - m \rangle$

ultimately have $\langle n = m + q \rangle$

by *simp*

with that show *?thesis*

by (*simp add: exp-add-not-zero-imp-right*)

next

case *False*

with that show *?thesis*

by *simp*

qed

end

lemma *nat-bit-induct* [*case-names zero even odd*]:

P n **if zero**: *P 0*

and even: $\bigwedge n. P n \implies n > 0 \implies P (2 * n)$

and odd: $\bigwedge n. P n \implies P (Suc (2 * n))$

proof (*induction n rule: less-induct*)

case (*less n*)

show *P n*

proof (*cases n = 0*)

case *True* **with zero show** *?thesis* **by** *simp*

next

case *False*

with less have hyp: *P (n div 2)* **by** *simp*

show *?thesis*

proof (*cases even n*)

case *True*

```

    then have  $n \neq 1$ 
      by auto
    with  $\langle n \neq 0 \rangle$  have  $n \text{ div } 2 > 0$ 
      by simp
    with  $\langle \text{even } n \rangle$  hyp even [of  $n \text{ div } 2$ ] show ?thesis
      by simp
  next
  case False
  with hyp odd [of  $n \text{ div } 2$ ] show ?thesis
    by simp
qed
qed
qed

instantiation nat :: semiring-bits
begin

definition bit-nat ::  $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$ 
  where  $\langle \text{bit-nat } m \ n \longleftrightarrow \text{odd } (m \text{ div } 2 \wedge n) \rangle$ 

instance
proof
  show  $\langle P \ n \rangle$  if stable:  $\langle \bigwedge n. n \text{ div } 2 = n \implies P \ n \rangle$ 
    and rec:  $\langle \bigwedge n \ b. P \ n \implies (\text{of-bool } b + 2 * n) \text{ div } 2 = n \implies P \ (\text{of-bool } b + 2 * n) \rangle$ 
  for  $P$  and  $n :: \text{nat}$ 
  proof (induction n rule: nat-bit-induct)
    case zero
    from stable [of 0] show ?case
      by simp
  next
  case (even n)
  with rec [of n False] show ?case
    by simp
  next
  case (odd n)
  with rec [of n True] show ?case
    by simp
  qed
  show  $\langle q \text{ mod } 2 \wedge m \text{ mod } 2 \wedge n = q \text{ mod } 2 \wedge \min \ m \ n \rangle$ 
    for  $q \ m \ n :: \text{nat}$ 
  apply (auto simp add: less-iff-Suc-add power-add mod-mod-cancel split: split-min-lin)
  apply (metis div-mult2-eq mod-div-trivial mod-eq-self-iff-div-eq-0 mod-mult-self2-is-0 power-commutes)
  done
  show  $\langle (q * 2 \wedge m) \text{ mod } (2 \wedge n) = (q \text{ mod } 2 \wedge (n - m)) * 2 \wedge m \rangle$  if  $\langle m \leq n \rangle$ 
    for  $q \ m \ n :: \text{nat}$ 
  using that
  apply (auto simp add: mod-mod-cancel div-mult2-eq power-add mod-mult2-eq

```

le-iff-add split: split-min-lin

```

done
show  $\langle \text{even } ((2 \wedge m - (1::\text{nat})) \text{ div } 2 \wedge n) \longleftrightarrow 2 \wedge n = (0::\text{nat}) \vee m \leq n) \rangle$ 
for  $m\ n :: \text{nat}$ 
using even-mask-div-iff' [where  $?a = \text{nat}$ , of  $m\ n$ ] by simp
show  $\langle \text{even } (q * 2 \wedge m \text{ div } 2 \wedge n) \longleftrightarrow n < m \vee (2::\text{nat}) \wedge n = 0 \vee m \leq n \wedge$ 
 $\text{even } (q \text{ div } 2 \wedge (n - m)) \rangle$ 
for  $m\ n\ q\ r :: \text{nat}$ 
apply (auto simp add: not-less power-add ac-simps dest!: le-Suc-ex)
apply (metis (full-types) dvd-mult dvd-mult-imp-div dvd-power-iff-le not-less
not-less-eq order-refl power-Suc)
done
qed (auto simp add: div-mult2-eq mod-mult2-eq power-add power-diff bit-nat-def)

end

```

```

lemma possible-bit-nat[simp]:
  possible-bit TYPE(nat) n
by (simp add: possible-bit-def)

```

```

lemma not-bit-Suc-0-Suc [simp]:
   $\langle \neg \text{bit } (\text{Suc } 0) (\text{Suc } n) \rangle$ 
by (simp add: bit-Suc)

```

```

lemma not-bit-Suc-0-numeral [simp]:
   $\langle \neg \text{bit } (\text{Suc } 0) (\text{numeral } n) \rangle$ 
by (simp add: numeral-eq-Suc)

```

```

lemma int-bit-induct [case-names zero minus even odd]:
  P k if zero-int: P 0
  and minus-int: P (- 1)
  and even-int:  $\bigwedge k. P k \implies k \neq 0 \implies P (k * 2)$ 
  and odd-int:  $\bigwedge k. P k \implies k \neq - 1 \implies P (1 + (k * 2))$  for  $k :: \text{int}$ 
proof (cases k  $\geq 0$ )
case True
define  $n$  where  $n = \text{nat } k$ 
with True have  $k = \text{int } n$ 
by simp
then show  $P k$ 
proof (induction n arbitrary: k rule: nat-bit-induct)
case zero
then show ?case
by (simp add: zero-int)
next
case (even n)
have  $P (\text{int } n * 2)$ 
by (rule even-int) (use even in simp-all)
with even show ?case
by (simp add: ac-simps)

```

```

next
  case (odd n)
  have P (1 + (int n * 2))
    by (rule odd-int) (use odd in simp-all)
  with odd show ?case
    by (simp add: ac-simps)
qed
next
case False
define n where n = nat (- k - 1)
with False have k = - int n - 1
  by simp
then show P k
proof (induction n arbitrary: k rule: nat-bit-induct)
  case zero
  then show ?case
    by (simp add: minus-int)
next
case (even n)
have P (1 + (- int (Suc n) * 2))
  by (rule odd-int) (use even in <simp-all add: algebra-simps>)
also have ... = - int (2 * n) - 1
  by (simp add: algebra-simps)
finally show ?case
  using even.premis by simp
next
case (odd n)
have P (- int (Suc n) * 2)
  by (rule even-int) (use odd in <simp-all add: algebra-simps>)
also have ... = - int (Suc (2 * n)) - 1
  by (simp add: algebra-simps)
finally show ?case
  using odd.premis by simp
qed
qed

context semiring-bits
begin

lemma bit-of-bool-iff [bit-simps]:
  <bit (of-bool b) n <math>\longleftrightarrow b \wedge n = 0</math>>
  by (simp add: bit-1-iff)

lemma bit-of-nat-iff [bit-simps]:
  <bit (of-nat m) n <math>\longleftrightarrow \text{possible-bit TYPE('a) } n \wedge \text{bit } m\ n</math>>
proof (cases <math>\langle 2::'a \rangle ^ n = 0</math>)
  case True
  then show ?thesis
    by (simp add: exp-eq-0-imp-not-bit possible-bit-def)

```

```

next
  case False
  then have  $\langle \text{bit } (\text{of-nat } m) \ n \longleftrightarrow \text{bit } m \ n \rangle$ 
  proof (induction m arbitrary: n rule: nat-bit-induct)
    case zero
    then show ?case
    by simp
  next
  case (even m)
  then show ?case
  by (cases n)
    (auto simp add: bit-double-iff Bit-Operations.bit-double-iff possible-bit-def
    bit-0 dest: mult-not-zero)
  next
  case (odd m)
  then show ?case
  by (cases n)
    (auto simp add: bit-double-iff even-bit-succ-iff possible-bit-def
    Bit-Operations.bit-Suc Bit-Operations.bit-0 dest: mult-not-zero)
  qed
  with False show ?thesis
  by (simp add: possible-bit-def)
qed

end

instantiation int :: semiring-bits
begin

definition bit-int ::  $\langle \text{int} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$ 
  where  $\langle \text{bit-int } k \ n \longleftrightarrow \text{odd } (k \ \text{div } 2 \wedge n) \rangle$ 

instance
proof
  show  $\langle P \ k \rangle$  if stable:  $\langle \bigwedge k. k \ \text{div } 2 = k \implies P \ k \rangle$ 
  and rec:  $\langle \bigwedge k \ b. P \ k \implies (\text{of-bool } b + 2 * k) \ \text{div } 2 = k \implies P \ (\text{of-bool } b + 2 * k) \rangle$ 
  for P and k :: int
  proof (induction k rule: int-bit-induct)
    case zero
    from stable [of 0] show ?case
    by simp
  next
  case minus
  from stable [of <- 1] show ?case
  by simp
  next
  case (even k)
  with rec [of k False] show ?case

```

```

    by (simp add: ac-simps)
  next
    case (odd k)
    with rec [of k True] show ?case
      by (simp add: ac-simps)
  qed
  show ⟨(2::int) ^ m div 2 ^ n = of-bool ((2::int) ^ m ≠ 0 ∧ n ≤ m) * 2 ^ (m -
n)⟩
    for m n :: nat
  proof (cases ⟨m < n⟩)
    case True
    then have ⟨n = m + (n - m)⟩
      by simp
    then have ⟨(2::int) ^ m div 2 ^ n = (2::int) ^ m div 2 ^ (m + (n - m))⟩
      by simp
    also have ⟨... = (2::int) ^ m div (2 ^ m * 2 ^ (n - m))⟩
      by (simp add: power-add)
    also have ⟨... = (2::int) ^ m div 2 ^ m div 2 ^ (n - m)⟩
      by (simp add: zdiv-zmult2-eq)
    finally show ?thesis using ⟨m < n⟩ by simp
  next
    case False
    then show ?thesis
      by (simp add: power-diff)
  qed
  show ⟨k mod 2 ^ m mod 2 ^ n = k mod 2 ^ min m n⟩
    for m n :: nat and k :: int
    using mod-exp-eq [of ⟨nat k⟩ m n]
    apply (auto simp add: mod-mod-cancel zdiv-zmult2-eq power-add zmod-zmult2-eq
le-iff-add split: split-min-lin)
    apply (auto simp add: less-iff-Suc-add mod-mod-cancel power-add)
    apply (simp only: flip: mult.left-commute [of ⟨2 ^ m⟩])
    apply (subst zmod-zmult2-eq) apply simp-all
  done
  show ⟨(k * 2 ^ m) mod (2 ^ n) = (k mod 2 ^ (n - m)) * 2 ^ m⟩
    if ⟨m ≤ n⟩ for m n :: nat and k :: int
    using that
    apply (auto simp add: power-add zmod-zmult2-eq le-iff-add split: split-min-lin)
  done
  show ⟨even ((2 ^ m - (1::int)) div 2 ^ n) ⟷ 2 ^ n = (0::int) ∨ m ≤ n⟩
    for m n :: nat
    using even-mask-div-iff' [where ?'a = int, of m n] by simp
  show ⟨even (k * 2 ^ m div 2 ^ n) ⟷ n < m ∨ (2::int) ^ n = 0 ∨ m ≤ n ∧
even (k div 2 ^ (n - m))⟩
    for m n :: nat and k l :: int
    apply (auto simp add: not-less power-add ac-simps dest!: le-Suc-ex)
    apply (metis Suc-leI dvd-mult dvd-mult-imp-div dvd-power-le dvd-refl power.simps(2))
  done
  qed (auto simp add: zdiv-zmult2-eq zmod-zmult2-eq power-add power-diff not-le

```

bit-int-def)

end

lemma *possible-bit-int*[*simp*]:
possible-bit TYPE(int) n
by (*simp add: possible-bit-def*)

lemma *bit-not-int-iff'*:
 $\langle \text{bit } (-k - 1) n \longleftrightarrow \neg \text{bit } k n \rangle$
for $k :: \text{int}$
proof (*induction n arbitrary: k*)
case 0
show ?*case*
by (*simp add: bit-0*)
next
case (*Suc n*)
have $\langle -k - 1 = -(k + 2) + 1 \rangle$
by *simp*
also have $\langle (- (k + 2) + 1) \text{ div } 2 = - (k \text{ div } 2) - 1 \rangle$
proof (*cases even k*)
case *True*
then have $\langle -k \text{ div } 2 = - (k \text{ div } 2) \rangle$
by *rule (simp flip: mult-minus-right)*
with *True* **show** ?*thesis*
by *simp*
next
case *False*
have $\langle 4 = 2 * (2 :: \text{int}) \rangle$
by *simp*
also have $\langle 2 * 2 \text{ div } 2 = (2 :: \text{int}) \rangle$
by (*simp only: nonzero-mult-div-cancel-left*)
finally have *: $\langle 4 \text{ div } 2 = (2 :: \text{int}) \rangle$.
from *False* **obtain** l **where** $k: \langle k = 2 * l + 1 \rangle$..
then have $\langle -k - 2 = 2 * -(l + 2) + 1 \rangle$
by *simp*
then have $\langle (-k - 2) \text{ div } 2 + 1 = - (k \text{ div } 2) - 1 \rangle$
by (*simp flip: mult-minus-right add: **) (*simp add: k*)
with *False* **show** ?*thesis*
by *simp*
qed
finally have $\langle (-k - 1) \text{ div } 2 = - (k \text{ div } 2) - 1 \rangle$.
with *Suc* **show** ?*case*
by (*simp add: bit-Suc*)
qed

lemma *bit-nat-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{nat } k) n \longleftrightarrow k \geq 0 \wedge \text{bit } k n \rangle$
proof (*cases k ≥ 0*)

```

case True
moreover define m where ⟨m = nat k⟩
ultimately have ⟨k = int m⟩
  by simp
then show ?thesis
  by (simp add: bit-simps)
next
case False
then show ?thesis
  by simp
qed

```

69.2 Bit operations

```

class semiring-bit-operations = semiring-bits +
  fixes and :: ⟨'a ⇒ 'a ⇒ 'a⟩ (infixr ⟨AND⟩ 64)
  and or :: ⟨'a ⇒ 'a ⇒ 'a⟩ (infixr ⟨OR⟩ 59)
  and xor :: ⟨'a ⇒ 'a ⇒ 'a⟩ (infixr ⟨XOR⟩ 59)
  and mask :: ⟨nat ⇒ 'a⟩
  and set-bit :: ⟨nat ⇒ 'a ⇒ 'a⟩
  and unset-bit :: ⟨nat ⇒ 'a ⇒ 'a⟩
  and flip-bit :: ⟨nat ⇒ 'a ⇒ 'a⟩
  and push-bit :: ⟨nat ⇒ 'a ⇒ 'a⟩
  and drop-bit :: ⟨nat ⇒ 'a ⇒ 'a⟩
  and take-bit :: ⟨nat ⇒ 'a ⇒ 'a⟩
  assumes bit-and-iff [bit-simps]: ⟨bit (a AND b) n ⟷ bit a n ∧ bit b n⟩
  and bit-or-iff [bit-simps]: ⟨bit (a OR b) n ⟷ bit a n ∨ bit b n⟩
  and bit-xor-iff [bit-simps]: ⟨bit (a XOR b) n ⟷ bit a n ≠ bit b n⟩
  and mask-eq-exp-minus-1: ⟨mask n = 2 ^ n - 1⟩
  and set-bit-eq-or: ⟨set-bit n a = a OR push-bit n 1⟩
  and bit-unset-bit-iff [bit-simps]: ⟨bit (unset-bit m a) n ⟷ bit a n ∧ m ≠ n⟩
  and flip-bit-eq-xor: ⟨flip-bit n a = a XOR push-bit n 1⟩
  and push-bit-eq-mult: ⟨push-bit n a = a * 2 ^ n⟩
  and drop-bit-eq-div: ⟨drop-bit n a = a div 2 ^ n⟩
  and take-bit-eq-mod: ⟨take-bit n a = a mod 2 ^ n⟩
begin

```

We want the bitwise operations to bind slightly weaker than $+$ and $-$.

Logically, *push-bit*, *drop-bit* and *take-bit* are just aliases; having them as separate operations makes proofs easier, otherwise proof automation would fiddle with concrete expressions $(2::'a)^n$ in a way obfuscating the basic algebraic relationships between those operations.

For the sake of code generation operations are specified as definitional class operations, taking into account that specific instances of these can be implemented differently wrt. code generation.

```

sublocale and: semilattice ⟨(AND)⟩
  by standard (auto simp add: bit-eq-iff bit-and-iff)

```


sublocale *or*: *semilattice-neutr* $\langle (OR) \rangle 0$
by *standard* (*auto simp add: bit-eq-iff bit-or-iff*)

sublocale *xor*: *comm-monoid* $\langle (XOR) \rangle 0$
by *standard* (*auto simp add: bit-eq-iff bit-xor-iff*)

lemma *even-and-iff*:
 $\langle \text{even } (a \text{ AND } b) \longleftrightarrow \text{even } a \vee \text{even } b \rangle$
using *bit-and-iff* [*of a b 0*] **by** (*auto simp add: bit-0*)

lemma *even-or-iff*:
 $\langle \text{even } (a \text{ OR } b) \longleftrightarrow \text{even } a \wedge \text{even } b \rangle$
using *bit-or-iff* [*of a b 0*] **by** (*auto simp add: bit-0*)

lemma *even-xor-iff*:
 $\langle \text{even } (a \text{ XOR } b) \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b) \rangle$
using *bit-xor-iff* [*of a b 0*] **by** (*auto simp add: bit-0*)

lemma *zero-and-eq* [*simp*]:
 $\langle 0 \text{ AND } a = 0 \rangle$
by (*simp add: bit-eq-iff bit-and-iff*)

lemma *and-zero-eq* [*simp*]:
 $\langle a \text{ AND } 0 = 0 \rangle$
by (*simp add: bit-eq-iff bit-and-iff*)

lemma *one-and-eq*:
 $\langle 1 \text{ AND } a = a \text{ mod } 2 \rangle$
by (*simp add: bit-eq-iff bit-and-iff*) (*auto simp add: bit-1-iff bit-0*)

lemma *and-one-eq*:
 $\langle a \text{ AND } 1 = a \text{ mod } 2 \rangle$
using *one-and-eq* [*of a*] **by** (*simp add: ac-simps*)

lemma *one-or-eq*:
 $\langle 1 \text{ OR } a = a + \text{of-bool } (\text{even } a) \rangle$
by (*simp add: bit-eq-iff bit-or-iff add.commute* [*of - 1*] *even-bit-succ-iff*)
(*auto simp add: bit-1-iff bit-0*)

lemma *or-one-eq*:
 $\langle a \text{ OR } 1 = a + \text{of-bool } (\text{even } a) \rangle$
using *one-or-eq* [*of a*] **by** (*simp add: ac-simps*)

lemma *one-xor-eq*:
 $\langle 1 \text{ XOR } a = a + \text{of-bool } (\text{even } a) - \text{of-bool } (\text{odd } a) \rangle$
by (*simp add: bit-eq-iff bit-xor-iff add.commute* [*of - 1*] *even-bit-succ-iff*)
(*auto simp add: bit-1-iff odd-bit-iff-bit-pred bit-0 elim: oddE*)

lemma *xor-one-eq*:

⟨ $a \text{ XOR } 1 = a + \text{of-bool (even } a) - \text{of-bool (odd } a)$ ⟩
using *one-xor-eq [of a]* **by** (*simp add: ac-simps*)

lemma *xor-self-eq [simp]*:
 ⟨ $a \text{ XOR } a = 0$ ⟩
by (*rule bit-eqI*) (*simp add: bit-simps*)

lemma *bit-iff-odd-drop-bit*:
 ⟨ $\text{bit } a \ n \longleftrightarrow \text{odd (drop-bit } n \ a)$ ⟩
by (*simp add: bit-iff-odd drop-bit-eq-div*)

lemma *even-drop-bit-iff-not-bit*:
 ⟨ $\text{even (drop-bit } n \ a) \longleftrightarrow \neg \text{bit } a \ n$ ⟩
by (*simp add: bit-iff-odd-drop-bit*)

lemma *div-push-bit-of-1-eq-drop-bit*:
 ⟨ $a \ \text{div} \ \text{push-bit } n \ 1 = \text{drop-bit } n \ a$ ⟩
by (*simp add: push-bit-eq-mult drop-bit-eq-div*)

lemma *bits-ident*:
 $\text{push-bit } n \ (\text{drop-bit } n \ a) + \text{take-bit } n \ a = a$
using *div-mult-mod-eq* **by** (*simp add: push-bit-eq-mult take-bit-eq-mod drop-bit-eq-div*)

lemma *push-bit-push-bit [simp]*:
 $\text{push-bit } m \ (\text{push-bit } n \ a) = \text{push-bit } (m + n) \ a$
by (*simp add: push-bit-eq-mult power-add ac-simps*)

lemma *push-bit-0-id [simp]*:
 $\text{push-bit } 0 = \text{id}$
by (*simp add: fun-eq-iff push-bit-eq-mult*)

lemma *push-bit-of-0 [simp]*:
 $\text{push-bit } n \ 0 = 0$
by (*simp add: push-bit-eq-mult*)

lemma *push-bit-of-1 [simp]*:
 $\text{push-bit } n \ 1 = 2 \ ^n$
by (*simp add: push-bit-eq-mult*)

lemma *push-bit-Suc [simp]*:
 $\text{push-bit } (\text{Suc } n) \ a = \text{push-bit } n \ (a * 2)$
by (*simp add: push-bit-eq-mult ac-simps*)

lemma *push-bit-double*:
 $\text{push-bit } n \ (a * 2) = \text{push-bit } n \ a * 2$
by (*simp add: push-bit-eq-mult ac-simps*)

lemma *push-bit-add*:
 $\text{push-bit } n \ (a + b) = \text{push-bit } n \ a + \text{push-bit } n \ b$

by (*simp add: push-bit-eq-mult algebra-simps*)

lemma *push-bit-numeral* [*simp*]:

$\langle \text{push-bit (numeral } l) \text{ (numeral } k) = \text{push-bit (pred-numeral } l) \text{ (numeral (Num.Bit0 } k))} \rangle$

by (*simp add: numeral-eq-Suc mult-2-right*) (*simp add: numeral-Bit0*)

lemma *take-bit-0* [*simp*]:

take-bit 0 a = 0

by (*simp add: take-bit-eq-mod*)

lemma *take-bit-Suc*:

$\langle \text{take-bit (Suc } n) a = \text{take-bit } n \text{ (} a \text{ div } 2) * 2 + a \text{ mod } 2 \rangle$

proof –

have $\langle \text{take-bit (Suc } n) (a \text{ div } 2 * 2 + \text{of-bool (odd } a)) = \text{take-bit } n \text{ (} a \text{ div } 2) * 2 + \text{of-bool (odd } a) \rangle$

using *even-succ-mod-exp* [*of* $\langle 2 * (a \text{ div } 2) \rangle$ $\langle \text{Suc } n \rangle$]

mult-exp-mod-exp-eq [*of* 1 $\langle \text{Suc } n \rangle$ $\langle a \text{ div } 2 \rangle$]

by (*auto simp add: take-bit-eq-mod ac-simps*)

then show *?thesis*

using *div-mult-mod-eq* [*of* $a \ 2$] **by** (*simp add: mod-2-eq-odd*)

qed

lemma *take-bit-rec*:

$\langle \text{take-bit } n \ a = \text{(if } n = 0 \text{ then } 0 \text{ else take-bit (} n - 1) \text{ (} a \text{ div } 2) * 2 + a \text{ mod } 2) \rangle$

by (*cases n*) (*simp-all add: take-bit-Suc*)

lemma *take-bit-Suc-0* [*simp*]:

$\langle \text{take-bit (Suc } 0) a = a \text{ mod } 2 \rangle$

by (*simp add: take-bit-eq-mod*)

lemma *take-bit-of-0* [*simp*]:

take-bit n 0 = 0

by (*simp add: take-bit-eq-mod*)

lemma *take-bit-of-1* [*simp*]:

take-bit n 1 = of-bool (n > 0)

by (*cases n*) (*simp-all add: take-bit-Suc*)

lemma *drop-bit-of-0* [*simp*]:

drop-bit n 0 = 0

by (*simp add: drop-bit-eq-div*)

lemma *drop-bit-of-1* [*simp*]:

drop-bit n 1 = of-bool (n = 0)

by (*simp add: drop-bit-eq-div*)

lemma *drop-bit-0* [*simp*]:

drop-bit 0 = id

by (simp add: fun-eq-iff drop-bit-eq-div)

lemma drop-bit-Suc:

drop-bit (Suc n) a = drop-bit n (a div 2)

using div-exp-eq [of a 1] by (simp add: drop-bit-eq-div)

lemma drop-bit-rec:

drop-bit n a = (if n = 0 then a else drop-bit (n - 1) (a div 2))

by (cases n) (simp-all add: drop-bit-Suc)

lemma drop-bit-half:

drop-bit n (a div 2) = drop-bit n a div 2

by (induction n arbitrary: a) (simp-all add: drop-bit-Suc)

lemma drop-bit-of-bool [simp]:

drop-bit n (of-bool b) = of-bool (n = 0 \wedge b)

by (cases n) simp-all

lemma even-take-bit-eq [simp]:

$\langle \text{even } (\text{take-bit } n \ a) \longleftrightarrow n = 0 \vee \text{even } a \rangle$

by (simp add: take-bit-rec [of n a])

lemma take-bit-take-bit [simp]:

take-bit m (take-bit n a) = take-bit (min m n) a

by (simp add: take-bit-eq-mod mod-exp-eq ac-simps)

lemma drop-bit-drop-bit [simp]:

drop-bit m (drop-bit n a) = drop-bit (m + n) a

by (simp add: drop-bit-eq-div power-add div-exp-eq ac-simps)

lemma push-bit-take-bit:

push-bit m (take-bit n a) = take-bit (m + n) (push-bit m a)

apply (simp add: push-bit-eq-mult take-bit-eq-mod power-add ac-simps)

using mult-exp-mod-exp-eq [of m $\langle m + n \rangle$ a] apply (simp add: ac-simps power-add)

done

lemma take-bit-push-bit:

take-bit m (push-bit n a) = push-bit n (take-bit (m - n) a)

proof (cases m \leq n)

case True

then show ?thesis

apply (simp add:)

apply (simp-all add: push-bit-eq-mult take-bit-eq-mod)

apply (auto dest!: le-Suc-ex simp add: power-add ac-simps)

using mult-exp-mod-exp-eq [of m m $\langle a * 2 ^ n \rangle$ for n]

apply (simp add: ac-simps)

done

next

case False

then show *?thesis*
using *push-bit-take-bit* [of $n\ m - n\ a$]
by *simp*
qed

lemma *take-bit-drop-bit*:
 $take\ bit\ m\ (drop\ bit\ n\ a) = drop\ bit\ n\ (take\ bit\ (m + n)\ a)$
by (*simp add: drop-bit-eq-div take-bit-eq-mod ac-simps div-exp-mod-exp-eq*)

lemma *drop-bit-take-bit*:
 $drop\ bit\ m\ (take\ bit\ n\ a) = take\ bit\ (n - m)\ (drop\ bit\ m\ a)$
proof (*cases $m \leq n$*)

case *True*
then show *?thesis*
using *take-bit-drop-bit* [of $n - m\ m\ a$] **by** *simp*
next
case *False*
then obtain q **where** $\langle m = n + q \rangle$
by (*auto simp add: not-le dest: less-imp-Suc-add*)
then have $\langle drop\ bit\ m\ (take\ bit\ n\ a) = 0 \rangle$
using *div-exp-eq* [of $\langle a\ mod\ 2^{\wedge}n \rangle\ n\ q$]
by (*simp add: take-bit-eq-mod drop-bit-eq-div*)
with *False* **show** *?thesis*
by *simp*
qed

lemma *even-push-bit-iff* [*simp*]:
 $\langle even\ (push\ bit\ n\ a) \longleftrightarrow n \neq 0 \vee even\ a \rangle$
by (*simp add: push-bit-eq-mult*) *auto*

lemma *bit-push-bit-iff* [*bit-simps*]:
 $\langle bit\ (push\ bit\ m\ a)\ n \longleftrightarrow m \leq n \wedge possible\ bit\ TYPE('a)\ n \wedge bit\ a\ (n - m) \rangle$
by (*auto simp add: bit-iff-odd push-bit-eq-mult even-mult-exp-div-exp-iff possible-bit-def*)

lemma *bit-drop-bit-eq* [*bit-simps*]:
 $\langle bit\ (drop\ bit\ n\ a) = bit\ a\ o\ (+)\ n \rangle$
by (*simp add: bit-iff-odd fun-eq-iff ac-simps flip: drop-bit-eq-div*)

lemma *bit-take-bit-iff* [*bit-simps*]:
 $\langle bit\ (take\ bit\ m\ a)\ n \longleftrightarrow n < m \wedge bit\ a\ n \rangle$
by (*simp add: bit-iff-odd drop-bit-take-bit not-le flip: drop-bit-eq-div*)

lemma *stable-imp-drop-bit-eq*:
 $\langle drop\ bit\ n\ a = a \rangle$
if $\langle a\ div\ 2 = a \rangle$
by (*induction n*) (*simp-all add: that drop-bit-Suc*)

lemma *stable-imp-take-bit-eq*:

$\langle \text{take-bit } n \ a = (\text{if even } a \text{ then } 0 \text{ else } 2^{\wedge} n - 1) \rangle$
 if $\langle a \ \text{div } 2 = a \rangle$
proof (rule bit-eqI[unfolding possible-bit-def])
 fix m
 assume $\langle 2^{\wedge} m \neq 0 \rangle$
 with that show $\langle \text{bit } (\text{take-bit } n \ a) \ m \longleftrightarrow \text{bit } (\text{if even } a \text{ then } 0 \text{ else } 2^{\wedge} n - 1) \ m \rangle$
 by (simp add: bit-take-bit-iff bit-mask-sub-iff possible-bit-def stable-imp-bit-iff-odd)
qed

lemma exp-dvdE:
 assumes $\langle 2^{\wedge} n \ \text{dvd } a \rangle$
 obtains b where $\langle a = \text{push-bit } n \ b \rangle$
proof –
 from *assms* obtain b where $\langle a = 2^{\wedge} n * b \rangle$..
 then have $\langle a = \text{push-bit } n \ b \rangle$
 by (simp add: push-bit-eq-mult ac-simps)
 with that show *thesis* .
qed

lemma take-bit-eq-0-iff:
 $\langle \text{take-bit } n \ a = 0 \longleftrightarrow 2^{\wedge} n \ \text{dvd } a \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)
proof
 assume $?P$
 then show $?Q$
 by (simp add: take-bit-eq-mod mod-0-imp-dvd)
next
 assume $?Q$
 then obtain b where $\langle a = \text{push-bit } n \ b \rangle$
 by (rule exp-dvdE)
 then show $?P$
 by (simp add: take-bit-push-bit)
qed

lemma take-bit-tightened:
 $\langle \text{take-bit } m \ a = \text{take-bit } m \ b \rangle$ if $\langle \text{take-bit } n \ a = \text{take-bit } n \ b \rangle$ and $\langle m \leq n \rangle$
proof –
 from that have $\langle \text{take-bit } m \ (\text{take-bit } n \ a) = \text{take-bit } m \ (\text{take-bit } n \ b) \rangle$
 by *simp*
 then have $\langle \text{take-bit } (\min m \ n) \ a = \text{take-bit } (\min m \ n) \ b \rangle$
 by *simp*
 with that show *?thesis*
 by (simp add: min-def)
qed

lemma take-bit-eq-self-iff-drop-bit-eq-0:
 $\langle \text{take-bit } n \ a = a \longleftrightarrow \text{drop-bit } n \ a = 0 \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)
proof
 assume $?P$

```

show ?Q
proof (rule bit-eqI)
  fix m
  from ⟨?P⟩ have ⟨a = take-bit n a⟩ ..
  also have ⟨¬ bit (take-bit n a) (n + m)⟩
    unfolding bit-simps
    by (simp add: bit-simps)
  finally show ⟨bit (drop-bit n a) m ⟷ bit 0 m⟩
    by (simp add: bit-simps)
qed
next
assume ?Q
show ?P
proof (rule bit-eqI)
  fix m
  from ⟨?Q⟩ have ⟨¬ bit (drop-bit n a) (m - n)⟩
    by simp
  then have ⟨¬ bit a (n + (m - n))⟩
    by (simp add: bit-simps)
  then show ⟨bit (take-bit n a) m ⟷ bit a m⟩
    by (cases ⟨m < n⟩) (auto simp add: bit-simps)
qed
qed

```

lemma *drop-bit-exp-eq*:
 ⟨drop-bit m (2^n) = of-bool (m ≤ n ∧ possible-bit TYPE('a) n) * $2^{(n-m)}$ ⟩
by (auto simp add: bit-eq-iff bit-simps)

lemma *take-bit-and* [simp]:
 ⟨take-bit n (a AND b) = take-bit n a AND take-bit n b⟩
by (auto simp add: bit-eq-iff bit-simps)

lemma *take-bit-or* [simp]:
 ⟨take-bit n (a OR b) = take-bit n a OR take-bit n b⟩
by (auto simp add: bit-eq-iff bit-simps)

lemma *take-bit-xor* [simp]:
 ⟨take-bit n (a XOR b) = take-bit n a XOR take-bit n b⟩
by (auto simp add: bit-eq-iff bit-simps)

lemma *push-bit-and* [simp]:
 ⟨push-bit n (a AND b) = push-bit n a AND push-bit n b⟩
by (auto simp add: bit-eq-iff bit-simps)

lemma *push-bit-or* [simp]:
 ⟨push-bit n (a OR b) = push-bit n a OR push-bit n b⟩
by (auto simp add: bit-eq-iff bit-simps)

lemma *push-bit-xor* [simp]:

⟨*push-bit* n (a XOR b) = *push-bit* n a XOR *push-bit* n b ⟩
by (*auto simp add: bit-eq-iff bit-simps*)

lemma *drop-bit-and* [*simp*]:
 ⟨*drop-bit* n (a AND b) = *drop-bit* n a AND *drop-bit* n b ⟩
by (*auto simp add: bit-eq-iff bit-simps*)

lemma *drop-bit-or* [*simp*]:
 ⟨*drop-bit* n (a OR b) = *drop-bit* n a OR *drop-bit* n b ⟩
by (*auto simp add: bit-eq-iff bit-simps*)

lemma *drop-bit-xor* [*simp*]:
 ⟨*drop-bit* n (a XOR b) = *drop-bit* n a XOR *drop-bit* n b ⟩
by (*auto simp add: bit-eq-iff bit-simps*)

lemma *bit-mask-iff* [*bit-simps*]:
 ⟨*bit* (*mask* m) n \longleftrightarrow *possible-bit* TYPE('a) $n \wedge n < m$ ⟩
by (*simp add: mask-eq-exp-minus-1 bit-mask-sub-iff*)

lemma *even-mask-iff*:
 ⟨*even* (*mask* n) \longleftrightarrow $n = 0$ ⟩
using *bit-mask-iff* [*of* n 0] **by** (*auto simp add: bit-0*)

lemma *mask-0* [*simp*]:
 ⟨*mask* 0 = 0⟩
by (*simp add: mask-eq-exp-minus-1*)

lemma *mask-Suc-0* [*simp*]:
 ⟨*mask* (*Suc* 0) = 1⟩
by (*simp add: mask-eq-exp-minus-1 add-implies-diff sym*)

lemma *mask-Suc-exp*:
 ⟨*mask* (*Suc* n) = 2 \wedge n OR *mask* n ⟩
by (*auto simp add: bit-eq-iff bit-simps*)

lemma *mask-Suc-double*:
 ⟨*mask* (*Suc* n) = 1 OR 2 * *mask* n ⟩
by (*auto simp add: bit-eq-iff bit-simps elim: possible-bit-less-imp*)

lemma *mask-numeral*:
 ⟨*mask* (*numeral* n) = 1 + 2 * *mask* (*pred-numeral* n)⟩
by (*simp add: numeral-eq-Suc mask-Suc-double one-or-eq ac-simps*)

lemma *take-bit-of-mask* [*simp*]:
 ⟨*take-bit* m (*mask* n) = *mask* (*min* m n)⟩
by (*rule bit-eqI*) (*simp add: bit-simps*)

lemma *take-bit-eq-mask*:
 ⟨*take-bit* n a = a AND *mask* n ⟩

by (*auto simp add: bit-eq-iff bit-simps*)

lemma *or-eq-0-iff*:

$\langle a \text{ OR } b = 0 \longleftrightarrow a = 0 \wedge b = 0 \rangle$

by (*auto simp add: bit-eq-iff bit-or-iff*)

lemma *disjunctive-add*:

$\langle a + b = a \text{ OR } b \rangle$ **if** $\langle \bigwedge n. \neg \text{bit } a \ n \vee \neg \text{bit } b \ n \rangle$

by (*rule bit-eqI*) (*use that in* $\langle \text{simp add: bit-disjunctive-add-iff bit-or-iff} \rangle$)

lemma *bit-iff-and-drop-bit-eq-1*:

$\langle \text{bit } a \ n \longleftrightarrow \text{drop-bit } n \ a \ \text{AND } 1 = 1 \rangle$

by (*simp add: bit-iff-odd-drop-bit and-one-eq odd-iff-mod-2-eq-one*)

lemma *bit-iff-and-push-bit-not-eq-0*:

$\langle \text{bit } a \ n \longleftrightarrow a \ \text{AND } \text{push-bit } n \ 1 \neq 0 \rangle$

apply (*cases* $\langle 2 \wedge n = 0 \rangle$)

apply (*simp-all add: bit-eq-iff bit-and-iff bit-push-bit-iff exp-eq-0-imp-not-bit*)

apply (*simp-all add: bit-exp-iff*)

done

lemmas *set-bit-def = set-bit-eq-or*

lemma *bit-set-bit-iff* [*bit-simps*]:

$\langle \text{bit } (\text{set-bit } m \ a) \ n \longleftrightarrow \text{bit } a \ n \vee (m = n \wedge \text{possible-bit } \text{TYPE}(a) \ n) \rangle$

by (*auto simp add: set-bit-def bit-or-iff bit-exp-iff*)

lemma *even-set-bit-iff*:

$\langle \text{even } (\text{set-bit } m \ a) \longleftrightarrow \text{even } a \wedge m \neq 0 \rangle$

using *bit-set-bit-iff* [*of m a 0*] **by** (*auto simp add: bit-0*)

lemma *even-unset-bit-iff*:

$\langle \text{even } (\text{unset-bit } m \ a) \longleftrightarrow \text{even } a \vee m = 0 \rangle$

using *bit-unset-bit-iff* [*of m a 0*] **by** (*auto simp add: bit-0*)

lemma *and-exp-eq-0-iff-not-bit*:

$\langle a \ \text{AND } 2 \wedge n = 0 \longleftrightarrow \neg \text{bit } a \ n \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)

using *bit-imp-possible-bit*[*of a n*]

by (*auto simp add: bit-eq-iff bit-simps*)

lemmas *flip-bit-def = flip-bit-eq-xor*

lemma *bit-flip-bit-iff* [*bit-simps*]:

$\langle \text{bit } (\text{flip-bit } m \ a) \ n \longleftrightarrow (m = n \longleftrightarrow \neg \text{bit } a \ n) \wedge \text{possible-bit } \text{TYPE}(a) \ n \rangle$

by (*auto simp add: bit-eq-iff bit-simps flip-bit-eq-xor bit-imp-possible-bit*)

lemma *even-flip-bit-iff*:

$\langle \text{even } (\text{flip-bit } m \ a) \longleftrightarrow \neg (\text{even } a \longleftrightarrow m = 0) \rangle$

using *bit-flip-bit-iff* [*of m a 0*] **by** (*auto simp: possible-bit-def bit-0*)

lemma *set-bit-0* [*simp*]:

⟨*set-bit 0 a = 1 + 2 * (a div 2)*⟩

by (*auto simp add: bit-eq-iff bit-simps even-bit-succ-iff simp flip: bit-Suc*)

lemma *bit-sum-mult-2-cases*:

assumes *a: ∀j. ¬ bit a (Suc j)*

shows *bit (a + 2 * b) n = (if n = 0 then odd a else bit (2 * b) n)*

proof –

have *a-eq: bit a i ⟷ i = 0 ∧ odd a* **for** *i*

by (*cases i*) (*simp-all add: a bit-0*)

show *?thesis*

by (*simp add: disjunctive-add[simplified disj-imp] a-eq bit-simps*)

qed

lemma *set-bit-Suc*:

⟨*set-bit (Suc n) a = a mod 2 + 2 * set-bit n (a div 2)*⟩

by (*auto simp add: bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip: bit-Suc elim: possible-bit-less-imp*)

lemma *unset-bit-0* [*simp*]:

⟨*unset-bit 0 a = 2 * (a div 2)*⟩

by (*auto simp add: bit-eq-iff bit-simps even-bit-succ-iff simp flip: bit-Suc*)

lemma *unset-bit-Suc*:

⟨*unset-bit (Suc n) a = a mod 2 + 2 * unset-bit n (a div 2)*⟩

by (*auto simp add: bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip: bit-Suc elim: possible-bit-less-imp*)

lemma *flip-bit-0* [*simp*]:

⟨*flip-bit 0 a = of-bool (even a) + 2 * (a div 2)*⟩

by (*auto simp add: bit-eq-iff bit-simps even-bit-succ-iff bit-0 simp flip: bit-Suc*)

lemma *flip-bit-Suc*:

⟨*flip-bit (Suc n) a = a mod 2 + 2 * flip-bit n (a div 2)*⟩

by (*auto simp add: bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip: bit-Suc elim: possible-bit-less-imp*)

lemma *flip-bit-eq-if*:

⟨*flip-bit n a = (if bit a n then unset-bit else set-bit) n a*⟩

by (*rule bit-eqI*) (*auto simp add: bit-set-bit-iff bit-unset-bit-iff bit-flip-bit-iff*)

lemma *take-bit-set-bit-eq*:

⟨*take-bit n (set-bit m a) = (if n ≤ m then take-bit n a else set-bit m (take-bit n a))*⟩

by (*rule bit-eqI*) (*auto simp add: bit-take-bit-iff bit-set-bit-iff*)

lemma *take-bit-unset-bit-eq*:

⟨*take-bit n (unset-bit m a) = (if n ≤ m then take-bit n a else unset-bit m (take-bit n a))*⟩

$n a$) \rangle
by (*rule bit-eqI*) (*auto simp add: bit-take-bit-iff bit-unset-bit-iff*)

lemma *take-bit-flip-bit-eq*:
 $\langle \text{take-bit } n \text{ (flip-bit } m \text{ } a) = (\text{if } n \leq m \text{ then take-bit } n \text{ } a \text{ else flip-bit } m \text{ (take-bit } n \text{ } a)) \rangle$
by (*rule bit-eqI*) (*auto simp add: bit-take-bit-iff bit-flip-bit-iff*)

lemma *bit-1-0* [*simp*]:
 $\langle \text{bit } 1 \ 0 \rangle$
by (*simp add: bit-0*)

lemma *not-bit-1-Suc* [*simp*]:
 $\langle \neg \text{bit } 1 \text{ (Suc } n) \rangle$
by (*simp add: bit-Suc*)

lemma *push-bit-Suc-numeral* [*simp*]:
 $\langle \text{push-bit (Suc } n) \text{ (numeral } k) = \text{push-bit } n \text{ (numeral (Num.Bit0 } k)) \rangle$
by (*simp add: numeral-eq-Suc mult-2-right*) (*simp add: numeral-Bit0*)

lemma *mask-eq-0-iff* [*simp*]:
 $\langle \text{mask } n = 0 \longleftrightarrow n = 0 \rangle$
by (*cases n*) (*simp-all add: mask-Suc-double or-eq-0-iff*)

end

class *ring-bit-operations* = *semiring-bit-operations* + *ring-parity* +
fixes *not* :: $\langle 'a \Rightarrow 'a \rangle$ ($\langle \text{NOT} \rangle$)
assumes *bit-not-iff-eq*: $\langle \bigwedge n. \text{bit (NOT } a) \ n \longleftrightarrow 2 \wedge n \neq 0 \wedge \neg \text{bit } a \ n \rangle$
assumes *minus-eq-not-minus-1*: $\langle - a = \text{NOT } (a - 1) \rangle$
begin

lemmas *bit-not-iff*[*bit-simps*] = *bit-not-iff-eq*[*unfolded fold-possible-bit*]

For the sake of code generation *NOT* is specified as definitional class operation. Note that *NOT* has no sensible definition for unlimited but only positive bit strings (type *nat*).

lemma *bits-minus-1-mod-2-eq* [*simp*]:
 $\langle (- 1) \bmod 2 = 1 \rangle$
by (*simp add: mod-2-eq-odd*)

lemma *not-eq-complement*:
 $\langle \text{NOT } a = - a - 1 \rangle$
using *minus-eq-not-minus-1* [*of* $\langle a + 1 \rangle$] **by** *simp*

lemma *minus-eq-not-plus-1*:
 $\langle - a = \text{NOT } a + 1 \rangle$
using *not-eq-complement* [*of* $\langle a \rangle$] **by** *simp*

lemma *bit-minus-iff* [*bit-simps*]:

$\langle \text{bit } (- a) n \longleftrightarrow \text{possible-bit TYPE}(a) n \wedge \neg \text{bit } (a - 1) n \rangle$

by (*simp add: minus-eq-not-minus-1 bit-not-iff*)

lemma *even-not-iff* [*simp*]:

$\langle \text{even } (\text{NOT } a) \longleftrightarrow \text{odd } a \rangle$

using *bit-not-iff* [*of a 0*] **by** (*auto simp add: bit-0*)

lemma *bit-not-exp-iff* [*bit-simps*]:

$\langle \text{bit } (\text{NOT } (2 \wedge m)) n \longleftrightarrow \text{possible-bit TYPE}(a) n \wedge n \neq m \rangle$

by (*auto simp add: bit-not-iff bit-exp-iff*)

lemma *bit-minus-1-iff* [*simp*]:

$\langle \text{bit } (- 1) n \longleftrightarrow \text{possible-bit TYPE}(a) n \rangle$

by (*simp add: bit-minus-iff*)

lemma *bit-minus-exp-iff* [*bit-simps*]:

$\langle \text{bit } (- (2 \wedge m)) n \longleftrightarrow \text{possible-bit TYPE}(a) n \wedge n \geq m \rangle$

by (*auto simp add: bit-simps simp flip: mask-eq-exp-minus-1*)

lemma *bit-minus-2-iff* [*simp*]:

$\langle \text{bit } (- 2) n \longleftrightarrow \text{possible-bit TYPE}(a) n \wedge n > 0 \rangle$

by (*simp add: bit-minus-iff bit-1-iff*)

lemma *not-one-eq* [*simp*]:

$\langle \text{NOT } 1 = - 2 \rangle$

by (*simp add: bit-eq-iff bit-not-iff*) (*simp add: bit-1-iff*)

sublocale *and: semilattice-neutr* $\langle (\text{AND}) \rangle \langle - 1 \rangle$

by *standard* (*rule bit-eqI, simp add: bit-and-iff*)

sublocale *bit: abstract-boolean-algebra* $\langle (\text{AND}) \rangle \langle (\text{OR}) \rangle \text{NOT } 0 \langle - 1 \rangle$

by *standard* (*auto simp add: bit-and-iff bit-or-iff bit-not-iff intro: bit-eqI*)

sublocale *bit: abstract-boolean-algebra-sym-diff* $\langle (\text{AND}) \rangle \langle (\text{OR}) \rangle \text{NOT } 0 \langle - 1 \rangle$

$\langle (\text{XOR}) \rangle$

apply *standard*

apply (*rule bit-eqI*)

apply (*auto simp add: bit-simps*)

done

lemma *and-eq-not-not-or*:

$\langle a \text{ AND } b = \text{NOT } (\text{NOT } a \text{ OR } \text{NOT } b) \rangle$

by *simp*

lemma *or-eq-not-not-and*:

$\langle a \text{ OR } b = \text{NOT } (\text{NOT } a \text{ AND } \text{NOT } b) \rangle$

by *simp*

lemma *not-add-distrib*:

$\langle \text{NOT } (a + b) = \text{NOT } a - b \rangle$

by (*simp add: not-eq-complement algebra-simps*)

lemma *not-diff-distrib*:

$\langle \text{NOT } (a - b) = \text{NOT } a + b \rangle$

using *not-add-distrib* [*of a <- b*] **by** *simp*

lemma *and-eq-minus-1-iff*:

$\langle a \text{ AND } b = - 1 \iff a = - 1 \wedge b = - 1 \rangle$

by (*auto simp: bit-eq-iff bit-simps*)

lemma *disjunctive-diff*:

$\langle a - b = a \text{ AND } \text{NOT } b \rangle$ **if** $\langle \bigwedge n. \text{bit } b \ n \implies \text{bit } a \ n \rangle$

proof –

have $\langle \text{NOT } a + b = \text{NOT } a \text{ OR } b \rangle$

by (*rule disjunctive-add*) (*auto simp add: bit-not-iff dest: that*)

then have $\langle \text{NOT } (\text{NOT } a + b) = \text{NOT } (\text{NOT } a \text{ OR } b) \rangle$

by *simp*

then show *?thesis*

by (*simp add: not-add-distrib*)

qed

lemma *push-bit-minus*:

$\langle \text{push-bit } n \ (- a) = - \text{push-bit } n \ a \rangle$

by (*simp add: push-bit-eq-mult*)

lemma *take-bit-not-take-bit*:

$\langle \text{take-bit } n \ (\text{NOT } (\text{take-bit } n \ a)) = \text{take-bit } n \ (\text{NOT } a) \rangle$

by (*auto simp add: bit-eq-iff bit-take-bit-iff bit-not-iff*)

lemma *take-bit-not-iff*:

$\langle \text{take-bit } n \ (\text{NOT } a) = \text{take-bit } n \ (\text{NOT } b) \iff \text{take-bit } n \ a = \text{take-bit } n \ b \rangle$

apply (*simp add: bit-eq-iff*)

apply (*simp add: bit-not-iff bit-take-bit-iff bit-exp-iff*)

apply (*use exp-eq-0-imp-not-bit in blast*)

done

lemma *take-bit-not-eq-mask-diff*:

$\langle \text{take-bit } n \ (\text{NOT } a) = \text{mask } n - \text{take-bit } n \ a \rangle$

proof –

have $\langle \text{take-bit } n \ (\text{NOT } a) = \text{take-bit } n \ (\text{NOT } (\text{take-bit } n \ a)) \rangle$

by (*simp add: take-bit-not-take-bit*)

also have $\langle \dots = \text{mask } n \text{ AND } \text{NOT } (\text{take-bit } n \ a) \rangle$

by (*simp add: take-bit-eq-mask ac-simps*)

also have $\langle \dots = \text{mask } n - \text{take-bit } n \ a \rangle$

by (*subst disjunctive-diff*)

(*auto simp add: bit-take-bit-iff bit-mask-iff bit-imp-possible-bit*)

finally show *?thesis*

by *simp*
qed

lemma *mask-eq-take-bit-minus-one*:
 $\langle \text{mask } n = \text{take-bit } n \ (-\ 1) \rangle$
 by (*simp add: bit-eq-iff bit-mask-iff bit-take-bit-iff conj-commute*)

lemma *take-bit-minus-one-eq-mask* [*simp*]:
 $\langle \text{take-bit } n \ (-\ 1) = \text{mask } n \rangle$
 by (*simp add: mask-eq-take-bit-minus-one*)

lemma *minus-exp-eq-not-mask*:
 $\langle -\ (2 \wedge n) = \text{NOT } (\text{mask } n) \rangle$
 by (*rule bit-eqI*) (*simp add: bit-minus-iff bit-not-iff flip: mask-eq-exp-minus-1*)

lemma *push-bit-minus-one-eq-not-mask* [*simp*]:
 $\langle \text{push-bit } n \ (-\ 1) = \text{NOT } (\text{mask } n) \rangle$
 by (*simp add: push-bit-eq-mult minus-exp-eq-not-mask*)

lemma *take-bit-not-mask-eq-0*:
 $\langle \text{take-bit } m \ (\text{NOT } (\text{mask } n)) = 0 \rangle$ if $\langle n \geq m \rangle$
 by (*rule bit-eqI*) (*use that in* *simp add: bit-take-bit-iff bit-not-iff bit-mask-iff*)

lemma *unset-bit-eq-and-not*:
 $\langle \text{unset-bit } n \ a = a \ \text{AND} \ \text{NOT } (\text{push-bit } n \ 1) \rangle$
 by (*rule bit-eqI*) (*auto simp add: bit-simps*)

lemmas *unset-bit-def = unset-bit-eq-and-not*

lemma *push-bit-Suc-minus-numeral* [*simp*]:
 $\langle \text{push-bit } (\text{Suc } n) \ (-\ \text{numeral } k) = \text{push-bit } n \ (-\ \text{numeral } (\text{Num.Bit0 } k)) \rangle$
 apply (*simp only: numeral-Bit0*)
 apply *simp*
 apply (*simp only: numeral-mult mult-2-right numeral-add*)
 done

lemma *push-bit-minus-numeral* [*simp*]:
 $\langle \text{push-bit } (\text{numeral } l) \ (-\ \text{numeral } k) = \text{push-bit } (\text{pred-numeral } l) \ (-\ \text{numeral } (\text{Num.Bit0 } k)) \rangle$
 by (*simp only: numeral-eq-Suc push-bit-Suc-minus-numeral*)

lemma *take-bit-Suc-minus-1-eq*:
 $\langle \text{take-bit } (\text{Suc } n) \ (-\ 1) = 2 \wedge \text{Suc } n - 1 \rangle$
 by (*simp add: mask-eq-exp-minus-1*)

lemma *take-bit-numeral-minus-1-eq*:
 $\langle \text{take-bit } (\text{numeral } k) \ (-\ 1) = 2 \wedge \text{numeral } k - 1 \rangle$
 by (*simp add: mask-eq-exp-minus-1*)

lemma *push-bit-mask-eq*:
 $\langle \text{push-bit } m \text{ (mask } n) = \text{mask } (n + m) \text{ AND NOT (mask } m) \rangle$
apply (*rule bit-eqI*)
apply (*auto simp add: bit-simps not-less possible-bit-def*)
apply (*drule sym [of 0]*)
apply (*simp only:*)
using *exp-not-zero-imp-exp-diff-not-zero* **apply** (*blast dest: exp-not-zero-imp-exp-diff-not-zero*)
done

lemma *slice-eq-mask*:
 $\langle \text{push-bit } n \text{ (take-bit } m \text{ (drop-bit } n \ a)) = a \text{ AND mask } (m + n) \text{ AND NOT (mask } n) \rangle$
by (*rule bit-eqI*) (*auto simp add: bit-simps*)

lemma *push-bit-numeral-minus-1* [*simp*]:
 $\langle \text{push-bit (numeral } n) (- 1) = - (2 \wedge \text{numeral } n) \rangle$
by (*simp add: push-bit-eq-mult*)

end

69.3 Instance *int*

instantiation *int* :: *ring-bit-operations*
begin

definition *not-int* :: $\langle \text{int} \Rightarrow \text{int} \rangle$
where $\langle \text{not-int } k = - k - 1 \rangle$

lemma *not-int-rec*:
 $\langle \text{NOT } k = \text{of-bool (even } k) + 2 * \text{NOT (} k \text{ div } 2) \rangle$ **for** $k :: \text{int}$
by (*auto simp add: not-int-def elim: oddE*)

lemma *even-not-iff-int*:
 $\langle \text{even (NOT } k) \longleftrightarrow \text{odd } k \rangle$ **for** $k :: \text{int}$
by (*simp add: not-int-def*)

lemma *not-int-div-2*:
 $\langle \text{NOT } k \text{ div } 2 = \text{NOT (} k \text{ div } 2) \rangle$ **for** $k :: \text{int}$
by (*simp add: not-int-def*)

lemma *bit-not-int-iff*:
 $\langle \text{bit (NOT } k) \ n \longleftrightarrow \neg \text{bit } k \ n \rangle$
for $k :: \text{int}$
by (*simp add: bit-not-int-iff' not-int-def*)

function *and-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle (k :: \text{int}) \text{ AND } l = (\text{if } k \in \{0, - 1\} \wedge l \in \{0, - 1\} \text{ then } - \text{of-bool (odd } k \wedge \text{odd } l) \text{ else of-bool (odd } k \wedge \text{odd } l) + 2 * ((k \text{ div } 2) \text{ AND } (l \text{ div } 2))) \rangle$

```

by auto

termination proof (relation ⟨measure (λ(k, l). nat (|k| + |l|))⟩)
show ⟨wf (measure (λ(k, l). nat (|k| + |l|)))⟩
  by simp
show ⟨((k div 2, l div 2), k, l) ∈ measure (λ(k, l). nat (|k| + |l|))⟩
  if ⟨¬ (k ∈ {0, - 1} ∧ l ∈ {0, - 1})⟩ for k l
proof -
  have less-eq: ⟨|k div 2| ≤ |k|⟩ for k :: int
    by (cases k) (simp-all add: divide-int-def nat-add-distrib)
  have less: ⟨|k div 2| < |k|⟩ if ⟨k ∉ {0, - 1}⟩ for k :: int
  proof (cases k)
    case (nonneg n)
    with that show ?thesis
      by (simp add: int-div-less-self)
  next
    case (neg n)
    with that have ⟨n ≠ 0⟩
      by simp
    then have ⟨n div 2 < n⟩
      by (simp add: div-less-iff-less-mult)
    with neg that show ?thesis
      by (simp add: divide-int-def nat-add-distrib)
  qed
from that have *: ⟨k ∉ {0, - 1} ∨ l ∉ {0, - 1}⟩
  by simp
then have ⟨0 < |k| + |l|⟩
  by auto
moreover from * have ⟨|k div 2| + |l div 2| < |k| + |l|⟩
proof
  assume ⟨k ∉ {0, - 1}⟩
  then have ⟨|k div 2| < |k|⟩
    by (rule less)
  with less-eq [of l] show ?thesis
    by auto
next
  assume ⟨l ∉ {0, - 1}⟩
  then have ⟨|l div 2| < |l|⟩
    by (rule less)
  with less-eq [of k] show ?thesis
    by auto
qed
ultimately show ?thesis
  by simp
qed
qed

declare and-int.simps [simp del]

```


lemma *and-int-rec*:

$\langle k \text{ AND } l = \text{of-bool } (\text{odd } k \wedge \text{odd } l) + 2 * ((k \text{ div } 2) \text{ AND } (l \text{ div } 2)) \rangle$
for $k \ l :: \text{int}$
proof (*cases* $\langle k \in \{0, -1\} \wedge l \in \{0, -1\} \rangle$)
case *True*
then show *?thesis*
by *auto* (*simp-all add: and-int.simps*)
next
case *False*
then show *?thesis*
by (*auto simp add: ac-simps and-int.simps [of k l]*)
qed

lemma *bit-and-int-iff*:

$\langle \text{bit } (k \text{ AND } l) \ n \longleftrightarrow \text{bit } k \ n \wedge \text{bit } l \ n \rangle$ **for** $k \ l :: \text{int}$
proof (*induction n arbitrary: k l*)
case *0*
then show *?case*
by (*simp add: and-int-rec [of k l] bit-0*)
next
case (*Suc n*)
then show *?case*
by (*simp add: and-int-rec [of k l] bit-Suc*)
qed

lemma *even-and-iff-int*:

$\langle \text{even } (k \text{ AND } l) \longleftrightarrow \text{even } k \vee \text{even } l \rangle$ **for** $k \ l :: \text{int}$
using *bit-and-int-iff [of k l 0]* **by** (*auto simp add: bit-0*)

definition *or-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$

where $\langle k \text{ OR } l = \text{NOT } (\text{NOT } k \text{ AND } \text{NOT } l) \rangle$ **for** $k \ l :: \text{int}$

lemma *or-int-rec*:

$\langle k \text{ OR } l = \text{of-bool } (\text{odd } k \vee \text{odd } l) + 2 * ((k \text{ div } 2) \text{ OR } (l \text{ div } 2)) \rangle$
for $k \ l :: \text{int}$
using *and-int-rec [of <NOT k> <NOT l>]*
by (*simp add: or-int-def even-not-iff-int not-int-div-2*)
(simp-all add: not-int-def)

lemma *bit-or-int-iff*:

$\langle \text{bit } (k \text{ OR } l) \ n \longleftrightarrow \text{bit } k \ n \vee \text{bit } l \ n \rangle$ **for** $k \ l :: \text{int}$
by (*simp add: or-int-def bit-not-int-iff bit-and-int-iff*)

definition *xor-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$

where $\langle k \text{ XOR } l = k \text{ AND } \text{NOT } l \text{ OR } \text{NOT } k \text{ AND } l \rangle$ **for** $k \ l :: \text{int}$

lemma *xor-int-rec*:

$\langle k \text{ XOR } l = \text{of-bool } (\text{odd } k \neq \text{odd } l) + 2 * ((k \text{ div } 2) \text{ XOR } (l \text{ div } 2)) \rangle$
for $k \ l :: \text{int}$

by (*simp add: xor-int-def or-int-rec [of <k AND NOT l> <NOT k AND l>]*
even-and-iff-int even-not-iff-int)
(simp add: and-int-rec [of <NOT k> <l>] and-int-rec [of <k> <NOT l>] not-int-div-2)

lemma *bit-xor-int-iff*:

<bit (k XOR l) n \longleftrightarrow bit k n \neq bit l n> **for** *k l :: int*
by (*auto simp add: xor-int-def bit-or-int-iff bit-and-int-iff bit-not-int-iff*)

definition *mask-int* :: *<nat \Rightarrow int>*

where *<mask n = (2 :: int) $^{\wedge}$ n - 1>*

definition *push-bit-int* :: *<nat \Rightarrow int \Rightarrow int>*

where *<push-bit-int n k = k * 2 $^{\wedge}$ n>*

definition *drop-bit-int* :: *<nat \Rightarrow int \Rightarrow int>*

where *<drop-bit-int n k = k div 2 $^{\wedge}$ n>*

definition *take-bit-int* :: *<nat \Rightarrow int \Rightarrow int>*

where *<take-bit-int n k = k mod 2 $^{\wedge}$ n>*

definition *set-bit-int* :: *<nat \Rightarrow int \Rightarrow int>*

where *<set-bit n k = k OR push-bit n 1>* **for** *k :: int*

definition *unset-bit-int* :: *<nat \Rightarrow int \Rightarrow int>*

where *<unset-bit n k = k AND NOT (push-bit n 1)>* **for** *k :: int*

definition *flip-bit-int* :: *<nat \Rightarrow int \Rightarrow int>*

where *<flip-bit n k = k XOR push-bit n 1>* **for** *k :: int*

instance proof

fix *k l :: int and m n :: nat*

show *<- k = NOT (k - 1)>*

by (*simp add: not-int-def*)

show *<bit (k AND l) n \longleftrightarrow bit k n \wedge bit l n>*

by (*fact bit-and-int-iff*)

show *<bit (k OR l) n \longleftrightarrow bit k n \vee bit l n>*

by (*fact bit-or-int-iff*)

show *<bit (k XOR l) n \longleftrightarrow bit k n \neq bit l n>*

by (*fact bit-xor-int-iff*)

show *<bit (unset-bit m k) n \longleftrightarrow bit k n \wedge m \neq n>*

proof -

have *<unset-bit m k = k AND NOT (push-bit m 1)>*

by (*simp add: unset-bit-int-def*)

also have *<NOT (push-bit m 1 :: int) = - (push-bit m 1 + 1)>*

by (*simp add: not-int-def*)

finally show *?thesis* **by** (*simp only: bit-simps bit-and-int-iff*)

(auto simp add: bit-simps bit-not-int-iff! push-bit-int-def)

qed

qed (*simp-all add: bit-not-int-iff mask-int-def set-bit-int-def flip-bit-int-def*)

push-bit-int-def drop-bit-int-def take-bit-int-def)

end

lemma *bit-push-bit-iff-int*:

$\langle \text{bit } (\text{push-bit } m \ k) \ n \longleftrightarrow m \leq n \wedge \text{bit } k \ (n - m) \rangle$ **for** $k :: \text{int}$
by (*auto simp add: bit-push-bit-iff*)

lemma *take-bit-nonnegative* [*simp*]:

$\langle \text{take-bit } n \ k \geq 0 \rangle$ **for** $k :: \text{int}$
by (*simp add: take-bit-eq-mod*)

lemma *not-take-bit-negative* [*simp*]:

$\langle \neg \text{take-bit } n \ k < 0 \rangle$ **for** $k :: \text{int}$
by (*simp add: not-less*)

lemma *take-bit-int-less-exp* [*simp*]:

$\langle \text{take-bit } n \ k < 2^{\wedge} n \rangle$ **for** $k :: \text{int}$
by (*simp add: take-bit-eq-mod*)

lemma *take-bit-int-eq-self-iff*:

$\langle \text{take-bit } n \ k = k \longleftrightarrow 0 \leq k \wedge k < 2^{\wedge} n \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)
for $k :: \text{int}$

proof

assume $?P$

moreover note *take-bit-int-less-exp* [*of n k*] *take-bit-nonnegative* [*of n k*]

ultimately show $?Q$

by *simp*

next

assume $?Q$

then show $?P$

by (*simp add: take-bit-eq-mod*)

qed

lemma *take-bit-int-eq-self*:

$\langle \text{take-bit } n \ k = k \rangle$ **if** $\langle 0 \leq k \rangle \langle k < 2^{\wedge} n \rangle$ **for** $k :: \text{int}$
using that by (*simp add: take-bit-int-eq-self-iff*)

lemma *mask-half-int*:

$\langle \text{mask } n \ \text{div } 2 = (\text{mask } (n - 1)) :: \text{int} \rangle$
by (*cases n*) (*simp-all add: mask-eq-exp-minus-1 algebra-simps*)

lemma *mask-nonnegative-int* [*simp*]:

$\langle \text{mask } n \geq (0 :: \text{int}) \rangle$
by (*simp add: mask-eq-exp-minus-1*)

lemma *not-mask-negative-int* [*simp*]:

$\langle \neg \text{mask } n < (0 :: \text{int}) \rangle$
by (*simp add: not-less*)

lemma *not-nonnegative-int-iff* [simp]:
 $\langle \text{NOT } k \geq 0 \longleftrightarrow k < 0 \rangle$ **for** $k :: \text{int}$
by (simp add: not-int-def)

lemma *not-negative-int-iff* [simp]:
 $\langle \text{NOT } k < 0 \longleftrightarrow k \geq 0 \rangle$ **for** $k :: \text{int}$
by (subst Not-eq-iff [symmetric]) (simp add: not-less not-le)

lemma *and-nonnegative-int-iff* [simp]:
 $\langle k \text{ AND } l \geq 0 \longleftrightarrow k \geq 0 \vee l \geq 0 \rangle$ **for** $k \ l :: \text{int}$
proof (induction k arbitrary: l rule: int-bit-induct)
case zero
then show ?case
by simp
next
case minus
then show ?case
by simp
next
case (even k)
then show ?case
using and-int-rec [of $\langle k * 2 \rangle$ l]
by (simp add: pos-imp-zdiv-nonneg-iff zero-le-mult-iff)
next
case (odd k)
from odd **have** $\langle 0 \leq k \text{ AND } l \text{ div } 2 \longleftrightarrow 0 \leq k \vee 0 \leq l \text{ div } 2 \rangle$
by simp
then have $\langle 0 \leq (1 + k * 2) \text{ div } 2 \text{ AND } l \text{ div } 2 \longleftrightarrow 0 \leq (1 + k * 2) \text{ div } 2 \vee$
 $0 \leq l \text{ div } 2 \rangle$
by simp
with and-int-rec [of $\langle 1 + k * 2 \rangle$ l]
show ?case
by (auto simp add: zero-le-mult-iff not-le)
qed

lemma *and-negative-int-iff* [simp]:
 $\langle k \text{ AND } l < 0 \longleftrightarrow k < 0 \wedge l < 0 \rangle$ **for** $k \ l :: \text{int}$
by (subst Not-eq-iff [symmetric]) (simp add: not-less)

lemma *and-less-eq*:
 $\langle k \text{ AND } l \leq k \rangle$ **if** $\langle l < 0 \rangle$ **for** $k \ l :: \text{int}$
using that proof (induction k arbitrary: l rule: int-bit-induct)
case zero
then show ?case
by simp
next
case minus
then show ?case

```

    by simp
next
  case (even k)
  from even.IH [of ⟨l div 2⟩] even.hyps even.prem
  show ?case
    by (simp add: and-int-rec [of - l])
next
  case (odd k)
  from odd.IH [of ⟨l div 2⟩] odd.hyps odd.prem
  show ?case
    by (simp add: and-int-rec [of - l])
qed

lemma or-nonnegative-int-iff [simp]:
  ⟨k OR l ≥ 0 ⟷ k ≥ 0 ∧ l ≥ 0⟩ for k l :: int
  by (simp only: or-eq-not-not-and not-nonnegative-int-iff) simp

lemma or-negative-int-iff [simp]:
  ⟨k OR l < 0 ⟷ k < 0 ∨ l < 0⟩ for k l :: int
  by (subst Not-eq-iff [symmetric]) (simp add: not-less)

lemma or-greater-eq:
  ⟨k OR l ≥ k⟩ if ⟨l ≥ 0⟩ for k l :: int
using that proof (induction k arbitrary: l rule: int-bit-induct)
  case zero
  then show ?case
    by simp
next
  case minus
  then show ?case
    by simp
next
  case (even k)
  from even.IH [of ⟨l div 2⟩] even.hyps even.prem
  show ?case
    by (simp add: or-int-rec [of - l])
next
  case (odd k)
  from odd.IH [of ⟨l div 2⟩] odd.hyps odd.prem
  show ?case
    by (simp add: or-int-rec [of - l])
qed

lemma xor-nonnegative-int-iff [simp]:
  ⟨k XOR l ≥ 0 ⟷ (k ≥ 0 ⟷ l ≥ 0)⟩ for k l :: int
  by (simp only: bit.xor-def or-nonnegative-int-iff) auto

lemma xor-negative-int-iff [simp]:
  ⟨k XOR l < 0 ⟷ (k < 0) ≠ (l < 0)⟩ for k l :: int

```

by (subst Not-eq-iff [symmetric]) (auto simp add: not-less)

lemma *OR-upper*:

fixes $x\ y :: \text{int}$

assumes $\langle 0 \leq x \rangle \langle x < 2^n \rangle \langle y < 2^n \rangle$

shows $\langle x \text{ OR } y < 2^n \rangle$

using *assms* **proof** (induction x arbitrary: $y\ n$ rule: *int-bit-induct*)

case *zero*

then show ?case

by *simp*

next

case *minus*

then show ?case

by *simp*

next

case (even x)

from *even.IH* [of $\langle n - 1 \rangle \langle y \text{ div } 2 \rangle$] *even.prem*s *even.hyps*

show ?case

by (cases n) (auto simp add: *or-int-rec* [of $\langle - * 2 \rangle$] *elim*: *oddE*)

next

case (odd x)

from *odd.IH* [of $\langle n - 1 \rangle \langle y \text{ div } 2 \rangle$] *odd.prem*s *odd.hyps*

show ?case

by (cases n) (auto simp add: *or-int-rec* [of $\langle 1 + - * 2 \rangle$], *linarith*)

qed

lemma *XOR-upper*:

fixes $x\ y :: \text{int}$

assumes $\langle 0 \leq x \rangle \langle x < 2^n \rangle \langle y < 2^n \rangle$

shows $\langle x \text{ XOR } y < 2^n \rangle$

using *assms* **proof** (induction x arbitrary: $y\ n$ rule: *int-bit-induct*)

case *zero*

then show ?case

by *simp*

next

case *minus*

then show ?case

by *simp*

next

case (even x)

from *even.IH* [of $\langle n - 1 \rangle \langle y \text{ div } 2 \rangle$] *even.prem*s *even.hyps*

show ?case

by (cases n) (auto simp add: *xor-int-rec* [of $\langle - * 2 \rangle$] *elim*: *oddE*)

next

case (odd x)

from *odd.IH* [of $\langle n - 1 \rangle \langle y \text{ div } 2 \rangle$] *odd.prem*s *odd.hyps*

show ?case

by (cases n) (auto simp add: *xor-int-rec* [of $\langle 1 + - * 2 \rangle$])

qed

lemma *AND-lower* [*simp*]:

fixes $x\ y :: \text{int}$
assumes $\langle 0 \leq x \rangle$
shows $\langle 0 \leq x \text{ AND } y \rangle$
using *assms* **by** *simp*

lemma *OR-lower* [*simp*]:

fixes $x\ y :: \text{int}$
assumes $\langle 0 \leq x \rangle \langle 0 \leq y \rangle$
shows $\langle 0 \leq x \text{ OR } y \rangle$
using *assms* **by** *simp*

lemma *XOR-lower* [*simp*]:

fixes $x\ y :: \text{int}$
assumes $\langle 0 \leq x \rangle \langle 0 \leq y \rangle$
shows $\langle 0 \leq x \text{ XOR } y \rangle$
using *assms* **by** *simp*

lemma *AND-upper1* [*simp*]:

fixes $x\ y :: \text{int}$
assumes $\langle 0 \leq x \rangle$
shows $\langle x \text{ AND } y \leq x \rangle$
using *assms* **proof** (*induction* x *arbitrary*: y *rule*: *int-bit-induct*)
case (*odd* k)
then have $\langle k \text{ AND } y \text{ div } 2 \leq k \rangle$
by *simp*
then show *?case*
by (*simp* *add*: *and-int-rec* [*of* $\langle 1 + - * 2 \rangle$])
qed (*simp-all* *add*: *and-int-rec* [*of* $\langle - * 2 \rangle$])

lemmas *AND-upper1'* [*simp*] = *order-trans* [*OF* *AND-upper1*]

lemmas *AND-upper1''* [*simp*] = *order-le-less-trans* [*OF* *AND-upper1*]

lemma *AND-upper2* [*simp*]:

fixes $x\ y :: \text{int}$
assumes $\langle 0 \leq y \rangle$
shows $\langle x \text{ AND } y \leq y \rangle$
using *assms* *AND-upper1* [*of* $y\ x$] **by** (*simp* *add*: *ac-simps*)

lemmas *AND-upper2'* [*simp*] = *order-trans* [*OF* *AND-upper2*]

lemmas *AND-upper2''* [*simp*] = *order-le-less-trans* [*OF* *AND-upper2*]

lemma *plus-and-or*: $\langle (x \text{ AND } y) + (x \text{ OR } y) = x + y \rangle$ **for** $x\ y :: \text{int}$

proof (*induction* x *arbitrary*: y *rule*: *int-bit-induct*)

case *zero*

then show *?case*

by *simp*

next

```

case minus
then show ?case
  by simp
next
  case (even x)
  from even.IH [of  $\langle y \text{ div } 2 \rangle$ ]
  show ?case
    by (auto simp add: and-int-rec [of - y] or-int-rec [of - y] elim: oddE)
next
  case (odd x)
  from odd.IH [of  $\langle y \text{ div } 2 \rangle$ ]
  show ?case
    by (auto simp add: and-int-rec [of - y] or-int-rec [of - y] elim: oddE)
qed

```

lemma *push-bit-minus-one*:
 $\langle \text{push-bit } n \text{ } (-1 :: \text{int}) = - (2 \wedge n) \rangle$
by (*simp add: push-bit-eq-mult*)

lemma *minus-1-div-exp-eq-int*:
 $\langle -1 \text{ div } (2 :: \text{int}) \wedge n = -1 \rangle$
by (*induction n*) (*use div-exp-eq* [*symmetric, of* $\langle -1 :: \text{int} \rangle 1$] **in** $\langle \text{simp-all add: ac-simps} \rangle$)

lemma *drop-bit-minus-one* [*simp*]:
 $\langle \text{drop-bit } n \text{ } (-1 :: \text{int}) = -1 \rangle$
by (*simp add: drop-bit-eq-div minus-1-div-exp-eq-int*)

lemma *take-bit-Suc-from-most*:
 $\langle \text{take-bit } (\text{Suc } n) \text{ } k = 2 \wedge n * \text{of-bool } (\text{bit } k \text{ } n) + \text{take-bit } n \text{ } k \rangle$ **for** $k :: \text{int}$
by (*simp only: take-bit-eq-mod power-Suc2*) (*simp-all add: bit-iff-odd odd-iff-mod-2-eq-one zmod-zmult2-eq*)

lemma *take-bit-minus*:
 $\langle \text{take-bit } n \text{ } (- \text{take-bit } n \text{ } k) = \text{take-bit } n \text{ } (-k) \rangle$
for $k :: \text{int}$
by (*simp add: take-bit-eq-mod mod-minus-eq*)

lemma *take-bit-diff*:
 $\langle \text{take-bit } n \text{ } (\text{take-bit } n \text{ } k - \text{take-bit } n \text{ } l) = \text{take-bit } n \text{ } (k - l) \rangle$
for $k \text{ } l :: \text{int}$
by (*simp add: take-bit-eq-mod mod-diff-eq*)

lemma *bit-imp-take-bit-positive*:
 $\langle 0 < \text{take-bit } m \text{ } k \rangle$ **if** $\langle n < m \rangle$ **and** $\langle \text{bit } k \text{ } n \rangle$ **for** $k :: \text{int}$
proof (*rule ccontr*)
assume $\langle \neg 0 < \text{take-bit } m \text{ } k \rangle$
then have $\langle \text{take-bit } m \text{ } k = 0 \rangle$
by (*auto simp add: not-less intro: order-antisym*)

then have $\langle \text{bit } (\text{take-bit } m \ k) \ n = \text{bit } 0 \ n \rangle$
by *simp*
with that show *False*
by (*simp add: bit-take-bit-iff*)
qed

lemma *take-bit-mult*:
 $\langle \text{take-bit } n \ (\text{take-bit } n \ k * \text{take-bit } n \ l) = \text{take-bit } n \ (k * l) \rangle$
for $k \ l :: \text{int}$
by (*simp add: take-bit-eq-mod mod-mult-eq*)

lemma (*in ring-1*) *of-nat-nat-take-bit-eq* [*simp*]:
 $\langle \text{of-nat } (\text{nat } (\text{take-bit } n \ k)) = \text{of-int } (\text{take-bit } n \ k) \rangle$
by *simp*

lemma *take-bit-minus-small-eq*:
 $\langle \text{take-bit } n \ (- \ k) = 2^{\wedge} n - k \rangle$ **if** $\langle 0 < k \rangle \langle k \leq 2^{\wedge} n \rangle$ **for** $k :: \text{int}$
proof –
define m **where** $\langle m = \text{nat } k \rangle$
with that have $\langle k = \text{int } m \rangle$ **and** $\langle 0 < m \rangle$ **and** $\langle m \leq 2^{\wedge} n \rangle$
by *simp-all*
have $\langle (2^{\wedge} n - m) \bmod 2^{\wedge} n = 2^{\wedge} n - m \rangle$
using $\langle 0 < m \rangle$ **by** *simp*
then have $\langle \text{int } ((2^{\wedge} n - m) \bmod 2^{\wedge} n) = \text{int } (2^{\wedge} n - m) \rangle$
by *simp*
then have $\langle (2^{\wedge} n - \text{int } m) \bmod 2^{\wedge} n = 2^{\wedge} n - \text{int } m \rangle$
using $\langle m \leq 2^{\wedge} n \rangle$ **by** (*simp only: of-nat-mod of-nat-diff*) *simp*
with $\langle k = \text{int } m \rangle$ **have** $\langle (2^{\wedge} n - k) \bmod 2^{\wedge} n = 2^{\wedge} n - k \rangle$
by *simp*
then show *?thesis*
by (*simp add: take-bit-eq-mod*)
qed

lemma *drop-bit-push-bit-int*:
 $\langle \text{drop-bit } m \ (\text{push-bit } n \ k) = \text{drop-bit } (m - n) \ (\text{push-bit } (n - m) \ k) \rangle$ **for** $k :: \text{int}$
by (*cases* $\langle m \leq n \rangle$) (*auto simp add: mult.left-commute [of - $2^{\wedge} n$] mult.commute [of - $2^{\wedge} n$] mult.assoc*
mult.commute [of k] drop-bit-eq-div push-bit-eq-mult not-le power-add dest!:
le-Suc-ex less-imp-Suc-add)

lemma *push-bit-nonnegative-int-iff* [*simp*]:
 $\langle \text{push-bit } n \ k \geq 0 \iff k \geq 0 \rangle$ **for** $k :: \text{int}$
by (*simp add: push-bit-eq-mult zero-le-mult-iff power-le-zero-eq*)

lemma *push-bit-negative-int-iff* [*simp*]:
 $\langle \text{push-bit } n \ k < 0 \iff k < 0 \rangle$ **for** $k :: \text{int}$
by (*subst Not-eq-iff [symmetric]*) (*simp add: not-less*)

lemma *drop-bit-nonnegative-int-iff* [*simp*]:

⟨*drop-bit* n $k \geq 0 \iff k \geq 0$ ⟩ **for** $k :: \text{int}$
by (*induction* n) (*auto simp add: drop-bit-Suc drop-bit-half*)

lemma *drop-bit-negative-int-iff* [*simp*]:
 ⟨*drop-bit* n $k < 0 \iff k < 0$ ⟩ **for** $k :: \text{int}$
by (*subst Not-eq-iff [symmetric]*) (*simp add: not-less*)

lemma *set-bit-nonnegative-int-iff* [*simp*]:
 ⟨*set-bit* n $k \geq 0 \iff k \geq 0$ ⟩ **for** $k :: \text{int}$
by (*simp add: set-bit-def*)

lemma *set-bit-negative-int-iff* [*simp*]:
 ⟨*set-bit* n $k < 0 \iff k < 0$ ⟩ **for** $k :: \text{int}$
by (*simp add: set-bit-def*)

lemma *unset-bit-nonnegative-int-iff* [*simp*]:
 ⟨*unset-bit* n $k \geq 0 \iff k \geq 0$ ⟩ **for** $k :: \text{int}$
by (*simp add: unset-bit-def*)

lemma *unset-bit-negative-int-iff* [*simp*]:
 ⟨*unset-bit* n $k < 0 \iff k < 0$ ⟩ **for** $k :: \text{int}$
by (*simp add: unset-bit-def*)

lemma *flip-bit-nonnegative-int-iff* [*simp*]:
 ⟨*flip-bit* n $k \geq 0 \iff k \geq 0$ ⟩ **for** $k :: \text{int}$
by (*simp add: flip-bit-def*)

lemma *flip-bit-negative-int-iff* [*simp*]:
 ⟨*flip-bit* n $k < 0 \iff k < 0$ ⟩ **for** $k :: \text{int}$
by (*simp add: flip-bit-def*)

lemma *set-bit-greater-eq*:
 ⟨*set-bit* n $k \geq k$ ⟩ **for** $k :: \text{int}$
by (*simp add: set-bit-def or-greater-eq*)

lemma *unset-bit-less-eq*:
 ⟨*unset-bit* n $k \leq k$ ⟩ **for** $k :: \text{int}$
by (*simp add: unset-bit-def and-less-eq*)

lemma *set-bit-eq*:
 ⟨*set-bit* n $k = k + \text{of-bool } (\neg \text{bit } k \ n) * 2 \wedge n$ ⟩ **for** $k :: \text{int}$

proof (*rule bit-eqI*)

fix m

show ⟨*bit* (*set-bit* n k) $m \iff \text{bit } (k + \text{of-bool } (\neg \text{bit } k \ n) * 2 \wedge n) \ m$ ⟩

proof (*cases* ⟨ $m = n$ ⟩)

case *True*

then show *?thesis*

apply (*simp add: bit-set-bit-iff*)

apply (*simp add: bit-iff-odd div-plus-div-distrib-dvd-right*)

```

done
next
case False
then show ?thesis
  apply (clarsimp simp add: bit-set-bit-iff)
  apply (subst disjunctive-add)
  apply (clarsimp simp add: bit-exp-iff)
  apply (clarsimp simp add: bit-or-iff bit-exp-iff)
done
qed
qed

```

lemma *unset-bit-eq*:

$\langle \text{unset-bit } n \ k = k - \text{of-bool } (\text{bit } k \ n) * 2^{\wedge} n \rangle$ **for** $k :: \text{int}$

proof (rule *bit-eqI*)

fix m

show $\langle \text{bit } (\text{unset-bit } n \ k) \ m \longleftrightarrow \text{bit } (k - \text{of-bool } (\text{bit } k \ n) * 2^{\wedge} n) \ m \rangle$

proof (cases $\langle m = n \rangle$)

case *True*

then show ?thesis

apply (simp add: bit-unset-bit-iff)

apply (simp add: bit-iff-odd)

using *div-plus-div-distrib-dvd-right* [of $\langle 2^{\wedge} n \rangle \langle - (2^{\wedge} n) \rangle k$]

apply (simp add: dvd-neg-div)

done

next

case *False*

then show ?thesis

apply (clarsimp simp add: bit-unset-bit-iff)

apply (subst disjunctive-diff)

apply (clarsimp simp add: bit-exp-iff)

apply (clarsimp simp add: bit-and-iff bit-not-iff bit-exp-iff)

done

qed

qed

lemma *and-int-unfold*:

$\langle k \ \text{AND} \ l = (\text{if } k = 0 \vee l = 0 \text{ then } 0 \text{ else if } k = -1 \text{ then } l \text{ else if } l = -1 \text{ then } k$

$\text{else } (k \bmod 2) * (l \bmod 2) + 2 * ((k \text{ div } 2) \ \text{AND} \ (l \text{ div } 2))) \rangle$ **for** $k \ l :: \text{int}$

by (auto simp add: *and-int-rec* [of $k \ l$] *zmult-eq-1-iff elim: oddE*)

lemma *or-int-unfold*:

$\langle k \ \text{OR} \ l = (\text{if } k = -1 \vee l = -1 \text{ then } -1 \text{ else if } k = 0 \text{ then } l \text{ else if } l = 0 \text{ then } k$

$\text{else } \max (k \bmod 2) (l \bmod 2) + 2 * ((k \text{ div } 2) \ \text{OR} \ (l \text{ div } 2))) \rangle$ **for** $k \ l :: \text{int}$

by (auto simp add: *or-int-rec* [of $k \ l$] *elim: oddE*)

lemma *xor-int-unfold*:

$\langle k \ \text{XOR} \ l = (\text{if } k = -1 \text{ then } \text{NOT } l \text{ else if } l = -1 \text{ then } \text{NOT } k \text{ else if } k = 0$

then l else if $l = 0$ then k
 else $|k \bmod 2 - l \bmod 2| + 2 * ((k \text{ div } 2) \text{ XOR } (l \text{ div } 2))$ for $k \ l :: \text{int}$
 by (auto simp add: xor-int-rec [of $k \ l$] not-int-def elim!: oddE)

lemma bit-minus-int-iff:

$\langle \text{bit } (-k) \ n \longleftrightarrow \text{bit } (\text{NOT } (k - 1)) \ n \rangle$
 for $k :: \text{int}$
 by (simp add: bit-simps)

lemma take-bit-incr-eq:

$\langle \text{take-bit } n \ (k + 1) = 1 + \text{take-bit } n \ k \rangle$ if $\langle \text{take-bit } n \ k \neq 2^n - 1 \rangle$
 for $k :: \text{int}$

proof –

from that have $\langle 2^n \neq k \bmod 2^n + 1 \rangle$

by (simp add: take-bit-eq-mod)

moreover have $\langle k \bmod 2^n < 2^n \rangle$

by simp

ultimately have *: $\langle k \bmod 2^n + 1 < 2^n \rangle$

by linarith

have $\langle (k + 1) \bmod 2^n = (k \bmod 2^n + 1) \bmod 2^n \rangle$

by (simp add: mod-simps)

also have $\langle \dots = k \bmod 2^n + 1 \rangle$

using * by (simp add: zmod-trivial-iff)

finally have $\langle (k + 1) \bmod 2^n = k \bmod 2^n + 1 \rangle$.

then show ?thesis

by (simp add: take-bit-eq-mod)

qed

lemma take-bit-decr-eq:

$\langle \text{take-bit } n \ (k - 1) = \text{take-bit } n \ k - 1 \rangle$ if $\langle \text{take-bit } n \ k \neq 0 \rangle$
 for $k :: \text{int}$

proof –

from that have $\langle k \bmod 2^n \neq 0 \rangle$

by (simp add: take-bit-eq-mod)

moreover have $\langle k \bmod 2^n \geq 0 \rangle \langle k \bmod 2^n < 2^n \rangle$

by simp-all

ultimately have *: $\langle k \bmod 2^n > 0 \rangle$

by linarith

have $\langle (k - 1) \bmod 2^n = (k \bmod 2^n - 1) \bmod 2^n \rangle$

by (simp add: mod-simps)

also have $\langle \dots = k \bmod 2^n - 1 \rangle$

by (simp add: zmod-trivial-iff)

(use $\langle k \bmod 2^n < 2^n \rangle$ * in linarith)

finally have $\langle (k - 1) \bmod 2^n = k \bmod 2^n - 1 \rangle$.

then show ?thesis

by (simp add: take-bit-eq-mod)

qed

lemma take-bit-int-greater-eq:

$\langle k + 2^n \leq \text{take-bit } n \ k \rangle$ if $\langle k < 0 \rangle$ for $k :: \text{int}$
proof –
have $\langle k + 2^n \leq \text{take-bit } n \ (k + 2^n) \rangle$
proof (cases $\langle k > - (2^n) \rangle$)
case *False*
then have $\langle k + 2^n \leq 0 \rangle$
by *simp*
also note *take-bit-nonnegative*
finally show *?thesis* .
next
case *True*
with that have $\langle 0 \leq k + 2^n \rangle$ and $\langle k + 2^n < 2^n \rangle$
by *simp-all*
then show *?thesis*
by (*simp only: take-bit-eq-mod mod-pos-pos-trivial*)
qed
then show *?thesis*
by (*simp add: take-bit-eq-mod*)
qed

lemma *take-bit-int-less-eq*:
 $\langle \text{take-bit } n \ k \leq k - 2^n \rangle$ if $\langle 2^n \leq k \rangle$ and $\langle n > 0 \rangle$ for $k :: \text{int}$
using that *zmod-le-nonneg-dividend* [of $\langle k - 2^n \rangle$ $\langle 2^n \rangle$]
by (*simp add: take-bit-eq-mod*)

lemma *take-bit-int-less-eq-self-iff*:
 $\langle \text{take-bit } n \ k \leq k \iff 0 \leq k \rangle$ (is $\langle ?P \iff ?Q \rangle$)
for $k :: \text{int}$
proof
assume *?P*
show *?Q*
proof (*rule ccontr*)
assume $\langle \neg 0 \leq k \rangle$
then have $\langle k < 0 \rangle$
by *simp*
with $\langle ?P \rangle$
have $\langle \text{take-bit } n \ k < 0 \rangle$
by (*rule le-less-trans*)
then show *False*
by *simp*
qed
next
assume *?Q*
then show *?P*
by (*simp add: take-bit-eq-mod zmod-le-nonneg-dividend*)
qed

lemma *take-bit-int-less-self-iff*:
 $\langle \text{take-bit } n \ k < k \iff 2^n \leq k \rangle$

for $k :: \text{int}$
by (*auto simp add: less-le take-bit-int-less-eq-self-iff take-bit-int-eq-self-iff*
intro: order-trans [of 0 $\langle 2 \wedge n \rangle k$])

lemma *take-bit-int-greater-self-iff*:
 $\langle k < \text{take-bit } n \ k \longleftrightarrow k < 0 \rangle$
for $k :: \text{int}$
using *take-bit-int-less-eq-self-iff [of n k]* **by** *auto*

lemma *take-bit-int-greater-eq-self-iff*:
 $\langle k \leq \text{take-bit } n \ k \longleftrightarrow k < 2 \wedge n \rangle$
for $k :: \text{int}$
by (*auto simp add: le-less take-bit-int-greater-self-iff take-bit-int-eq-self-iff*
dest: sym not-sym intro: less-trans [of k 0 $\langle 2 \wedge n \rangle$])

lemma *not-exp-less-eq-0-int [simp]*:
 $\langle \neg 2 \wedge n \leq (0 :: \text{int}) \rangle$
by (*simp add: power-le-zero-eq*)

lemma *int-bit-bound*:

fixes $k :: \text{int}$
obtains n **where** $\langle \bigwedge m. n \leq m \implies \text{bit } k \ m \longleftrightarrow \text{bit } k \ n \rangle$
and $\langle n > 0 \implies \text{bit } k \ (n - 1) \neq \text{bit } k \ n \rangle$

proof –

obtain q **where** $*$: $\langle \bigwedge m. q \leq m \implies \text{bit } k \ m \longleftrightarrow \text{bit } k \ q \rangle$

proof (*cases $\langle k \geq 0 \rangle$*)

case *True*

moreover from *power-gt-expt [of 2 $\langle \text{nat } k \rangle$]*

have $\langle \text{nat } k < 2 \wedge \text{nat } k \rangle$

by *simp*

then have $\langle \text{int } (\text{nat } k) < \text{int } (2 \wedge \text{nat } k) \rangle$

by (*simp only: of-nat-less-iff*)

ultimately have $*$: $\langle k \text{ div } 2 \wedge \text{nat } k = 0 \rangle$

by *simp*

show *thesis*

proof (*rule that [of $\langle \text{nat } k \rangle$]*)

fix m

assume $\langle \text{nat } k \leq m \rangle$

then show $\langle \text{bit } k \ m \longleftrightarrow \text{bit } k \ (\text{nat } k) \rangle$

by (*auto simp add: * bit-iff-odd power-add zdiv-zmult2-eq dest!: le-Suc-ex*)

qed

next

case *False*

moreover from *power-gt-expt [of 2 $\langle \text{nat } (- k) \rangle$]*

have $\langle \text{nat } (- k) < 2 \wedge \text{nat } (- k) \rangle$

by *simp*

then have $\langle \text{int } (\text{nat } (- k)) < \text{int } (2 \wedge \text{nat } (- k)) \rangle$

by (*simp only: of-nat-less-iff*)

ultimately have $\langle - k \text{ div } - (2 \wedge \text{nat } (- k)) = - 1 \rangle$

```

    by (subst div-pos-neg-trivial) simp-all
  then have *: ⟨k div 2 ^ nat (- k) = - 1⟩
    by simp
  show thesis
  proof (rule that [of ⟨nat (- k)⟩])
    fix m
    assume ⟨nat (- k) ≤ m⟩
    then show ⟨bit k m ↔ bit k (nat (- k))⟩
      by (auto simp add: * bit-iff-odd power-add zdiv-zmult2-eq minus-1-div-exp-eq-int
        dest!: le-Suc-ex)
    qed
  qed
  show thesis
  proof (cases ⟨∀ m. bit k m ↔ bit k q⟩)
    case True
    then have ⟨bit k 0 ↔ bit k q⟩
      by blast
    with True that [of 0] show thesis
      by simp
  next
  case False
  then obtain r where **: ⟨bit k r ≠ bit k q⟩
    by blast
  have ⟨r < q⟩
    by (rule ccontr) (use * [of r] ** in simp)
  define N where ⟨N = {n. n < q ∧ bit k n ≠ bit k q}⟩
  moreover have ⟨finite N⟩ ⟨r ∈ N⟩
    using ** N-def ⟨r < q⟩ by auto
  moreover define n where ⟨n = Suc (Max N)⟩
  ultimately have ⟨∧ m. n ≤ m ⇒ bit k m ↔ bit k n⟩
    apply auto
    apply (metis (full-types, lifting) * Max-ge-iff Suc-n-not-le-n ⟨finite N⟩
      all-not-in-conv mem-Collect-eq not-le)
    apply (metis * Max-ge Suc-n-not-le-n ⟨finite N⟩ linorder-not-less mem-Collect-eq)
    apply (metis * Max-ge Suc-n-not-le-n ⟨finite N⟩ linorder-not-less mem-Collect-eq)
    apply (metis (full-types, lifting) * Max-ge-iff Suc-n-not-le-n ⟨finite N⟩ all-not-in-conv
      mem-Collect-eq not-le)
    done
  have ⟨bit k (Max N) ≠ bit k n⟩
    by (metis (mono-tags, lifting) * Max-in N-def ⟨∧ m. n ≤ m ⇒ bit k m =
      bit k n⟩ ⟨finite N⟩ ⟨r ∈ N⟩ empty-iff le-cases mem-Collect-eq)
  show thesis apply (rule that [of n])
    using ⟨∧ m. n ≤ m ⇒ bit k m = bit k n⟩ apply blast
    using ⟨bit k (Max N) ≠ bit k n⟩ n-def by auto
  qed
  qed

```

lemma *take-bit-tightened-less-eq-int*:

⟨take-bit m k ≤ take-bit n k⟩ **if** ⟨m ≤ n⟩ **for** k :: int

```

proof –
  have  $\langle \text{take-bit } m \ (\text{take-bit } n \ k) \leq \text{take-bit } n \ k \rangle$ 
    by (simp only: take-bit-int-less-eq-self-iff take-bit-nonnegative)
  with that show ?thesis
    by simp
qed

context ring-bit-operations
begin

lemma even-of-int-iff:
   $\langle \text{even } (\text{of-int } k) \longleftrightarrow \text{even } k \rangle$ 
  by (induction k rule: int-bit-induct simp-all)

lemma bit-of-int-iff [bit-simps]:
   $\langle \text{bit } (\text{of-int } k) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } k \ n \rangle$ 
proof (cases  $\langle \text{possible-bit } \text{TYPE}('a) \ n \rangle$ )
  case False
    then show ?thesis
      by (simp add: impossible-bit)
next
  case True
    then have  $\langle \text{bit } (\text{of-int } k) \ n \longleftrightarrow \text{bit } k \ n \rangle$ 
    proof (induction k arbitrary: n rule: int-bit-induct)
      case zero
        then show ?case
          by simp
      next
        case minus
          then show ?case
            by simp
      next
        case (even k)
          then show ?case
            using bit-double-iff [of  $\langle \text{of-int } k \rangle \ n$ ] Bit-Operations.bit-double-iff [of  $k \ n$ ]
            by (cases n) (auto simp add: ac-simps possible-bit-def dest: mult-not-zero)
        next
          case (odd k)
            then show ?case
              using bit-double-iff [of  $\langle \text{of-int } k \rangle \ n$ ]
              by (cases n)
              (auto simp add: ac-simps bit-double-iff even-bit-succ-iff Bit-Operations.bit-0
Bit-Operations.bit-Suc
possible-bit-def dest: mult-not-zero)
    qed
  with True show ?thesis
    by simp
qed

```


lemma *push-bit-of-int*:
 $\langle \text{push-bit } n \text{ (of-int } k) = \text{of-int (push-bit } n \text{ } k) \rangle$
by (*simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *of-int-push-bit*:
 $\langle \text{of-int (push-bit } n \text{ } k) = \text{push-bit } n \text{ (of-int } k) \rangle$
by (*simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *take-bit-of-int*:
 $\langle \text{take-bit } n \text{ (of-int } k) = \text{of-int (take-bit } n \text{ } k) \rangle$
by (*rule bit-eqI*) (*simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-int-iff*)

lemma *of-int-take-bit*:
 $\langle \text{of-int (take-bit } n \text{ } k) = \text{take-bit } n \text{ (of-int } k) \rangle$
by (*rule bit-eqI*) (*simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-int-iff*)

lemma *of-int-not-eq*:
 $\langle \text{of-int (NOT } k) = \text{NOT (of-int } k) \rangle$
by (*rule bit-eqI*) (*simp add: bit-not-iff Bit-Operations.bit-not-iff bit-of-int-iff*)

lemma *of-int-not-numeral*:
 $\langle \text{of-int (NOT (numeral } k)) = \text{NOT (numeral } k) \rangle$
by (*simp add: local.of-int-not-eq*)

lemma *of-int-and-eq*:
 $\langle \text{of-int (} k \text{ AND } l) = \text{of-int } k \text{ AND of-int } l \rangle$
by (*rule bit-eqI*) (*simp add: bit-of-int-iff bit-and-iff Bit-Operations.bit-and-iff*)

lemma *of-int-or-eq*:
 $\langle \text{of-int (} k \text{ OR } l) = \text{of-int } k \text{ OR of-int } l \rangle$
by (*rule bit-eqI*) (*simp add: bit-of-int-iff bit-or-iff Bit-Operations.bit-or-iff*)

lemma *of-int-xor-eq*:
 $\langle \text{of-int (} k \text{ XOR } l) = \text{of-int } k \text{ XOR of-int } l \rangle$
by (*rule bit-eqI*) (*simp add: bit-of-int-iff bit-xor-iff Bit-Operations.bit-xor-iff*)

lemma *of-int-mask-eq*:
 $\langle \text{of-int (mask } n) = \text{mask } n \rangle$
by (*induction n*) (*simp-all add: mask-Suc-double Bit-Operations.mask-Suc-double of-int-or-eq*)

end

69.4 Instance *nat*

instantiation *nat* :: *semiring-bit-operations*
begin

definition *and-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$

where $\langle m \text{ AND } n = \text{nat } (\text{int } m \text{ AND } \text{int } n) \rangle$ **for** $m \ n :: \text{nat}$

definition $\text{or-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle m \text{ OR } n = \text{nat } (\text{int } m \text{ OR } \text{int } n) \rangle$ **for** $m \ n :: \text{nat}$

definition $\text{xor-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle m \text{ XOR } n = \text{nat } (\text{int } m \text{ XOR } \text{int } n) \rangle$ **for** $m \ n :: \text{nat}$

definition $\text{mask-nat} :: \langle \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{mask } n = (2 :: \text{nat}) \wedge n - 1 \rangle$

definition $\text{push-bit-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{push-bit-nat } n \ m = m * 2 \wedge n \rangle$

definition $\text{drop-bit-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{drop-bit-nat } n \ m = m \text{ div } 2 \wedge n \rangle$

definition $\text{take-bit-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{take-bit-nat } n \ m = m \text{ mod } 2 \wedge n \rangle$

definition $\text{set-bit-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{set-bit } m \ n = n \text{ OR } \text{push-bit } m \ 1 \rangle$ **for** $m \ n :: \text{nat}$

definition $\text{unset-bit-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{unset-bit } m \ n = \text{nat } (\text{unset-bit } m \ (\text{int } n)) \rangle$ **for** $m \ n :: \text{nat}$

definition $\text{flip-bit-nat} :: \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$
where $\langle \text{flip-bit } m \ n = n \text{ XOR } \text{push-bit } m \ 1 \rangle$ **for** $m \ n :: \text{nat}$

instance proof

fix $m \ n \ q :: \text{nat}$

show $\langle \text{bit } (m \text{ AND } n) \ q \longleftrightarrow \text{bit } m \ q \wedge \text{bit } n \ q \rangle$

by (*simp add: and-nat-def bit-simps*)

show $\langle \text{bit } (m \text{ OR } n) \ q \longleftrightarrow \text{bit } m \ q \vee \text{bit } n \ q \rangle$

by (*simp add: or-nat-def bit-simps*)

show $\langle \text{bit } (m \text{ XOR } n) \ q \longleftrightarrow \text{bit } m \ q \neq \text{bit } n \ q \rangle$

by (*simp add: xor-nat-def bit-simps*)

show $\langle \text{bit } (\text{unset-bit } m \ n) \ q \longleftrightarrow \text{bit } n \ q \wedge m \neq q \rangle$

by (*simp add: unset-bit-nat-def bit-simps*)

qed (*simp-all add: mask-nat-def set-bit-nat-def flip-bit-nat-def push-bit-nat-def drop-bit-nat-def take-bit-nat-def*)

end

lemma $\text{take-bit-nat-less-exp}$ [*simp*]:
 $\langle \text{take-bit } n \ m < 2 \wedge n \rangle$ **for** $n \ m :: \text{nat}$
by (*simp add: take-bit-eq-mod*)

lemma $\text{take-bit-nat-eq-self-iff}$:

$\langle \text{take-bit } n \ m = m \longleftrightarrow m < 2^n \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)
for $n \ m :: \text{nat}$
proof
 assume $?P$
 moreover **note** $\text{take-bit-nat-less-exp}$ [of $n \ m$]
 ultimately **show** $?Q$
 by *simp*
next
 assume $?Q$
 then **show** $?P$
 by (*simp add: take-bit-eq-mod*)
qed

lemma $\text{take-bit-nat-eq-self}$:
 $\langle \text{take-bit } n \ m = m \rangle$ **if** $\langle m < 2^n \rangle$ **for** $m \ n :: \text{nat}$
using *that* **by** (*simp add: take-bit-nat-eq-self-iff*)

lemma $\text{take-bit-nat-less-eq-self}$ [*simp*]:
 $\langle \text{take-bit } n \ m \leq m \rangle$ **for** $n \ m :: \text{nat}$
by (*simp add: take-bit-eq-mod*)

lemma $\text{take-bit-nat-less-self-iff}$:
 $\langle \text{take-bit } n \ m < m \longleftrightarrow 2^n \leq m \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)
for $m \ n :: \text{nat}$
proof
 assume $?P$
 then **have** $\langle \text{take-bit } n \ m \neq m \rangle$
 by *simp*
 then **show** $\langle ?Q \rangle$
 by (*simp add: take-bit-nat-eq-self-iff*)
next
have $\langle \text{take-bit } n \ m < 2^n \rangle$
 by (*fact take-bit-nat-less-exp*)
 also **assume** $?Q$
 finally **show** $?P$.
qed

lemma $\text{bit-push-bit-iff-nat}$:
 $\langle \text{bit } (\text{push-bit } m \ q) \ n \longleftrightarrow m \leq n \wedge \text{bit } q \ (n - m) \rangle$ **for** $q :: \text{nat}$
by (*auto simp add: bit-push-bit-iff*)

lemma and-nat-rec :
 $\langle m \ \text{AND} \ n = \text{of-bool } (\text{odd } m \wedge \text{odd } n) + 2 * ((m \ \text{div} \ 2) \ \text{AND} \ (n \ \text{div} \ 2)) \rangle$ **for** $m \ n :: \text{nat}$
by (*simp add: and-nat-def and-int-rec* [of $\langle \text{int } m \rangle \langle \text{int } n \rangle$] *zdiv-int nat-add-distrib nat-mult-distrib*)

lemma or-nat-rec :
 $\langle m \ \text{OR} \ n = \text{of-bool } (\text{odd } m \vee \text{odd } n) + 2 * ((m \ \text{div} \ 2) \ \text{OR} \ (n \ \text{div} \ 2)) \rangle$ **for** $m \ n$

:: nat

by (*simp add: or-nat-def or-int-rec [of ⟨int m⟩ ⟨int n⟩] zdiv-int nat-add-distrib nat-mult-distrib*)

lemma *xor-nat-rec*:

$\langle m \text{ XOR } n = \text{of-bool } (\text{odd } m \neq \text{odd } n) + 2 * ((m \text{ div } 2) \text{ XOR } (n \text{ div } 2)) \rangle$ **for** m
 $n :: \text{nat}$

by (*simp add: xor-nat-def xor-int-rec [of ⟨int m⟩ ⟨int n⟩] zdiv-int nat-add-distrib nat-mult-distrib*)

lemma *Suc-0-and-eq [simp]*:

$\langle \text{Suc } 0 \text{ AND } n = n \text{ mod } 2 \rangle$

using *one-and-eq [of n] by simp*

lemma *and-Suc-0-eq [simp]*:

$\langle n \text{ AND } \text{Suc } 0 = n \text{ mod } 2 \rangle$

using *and-one-eq [of n] by simp*

lemma *Suc-0-or-eq*:

$\langle \text{Suc } 0 \text{ OR } n = n + \text{of-bool } (\text{even } n) \rangle$

using *one-or-eq [of n] by simp*

lemma *or-Suc-0-eq*:

$\langle n \text{ OR } \text{Suc } 0 = n + \text{of-bool } (\text{even } n) \rangle$

using *or-one-eq [of n] by simp*

lemma *Suc-0-xor-eq*:

$\langle \text{Suc } 0 \text{ XOR } n = n + \text{of-bool } (\text{even } n) - \text{of-bool } (\text{odd } n) \rangle$

using *one-xor-eq [of n] by simp*

lemma *xor-Suc-0-eq*:

$\langle n \text{ XOR } \text{Suc } 0 = n + \text{of-bool } (\text{even } n) - \text{of-bool } (\text{odd } n) \rangle$

using *xor-one-eq [of n] by simp*

lemma *and-nat-unfold [code]*:

$\langle m \text{ AND } n = (\text{if } m = 0 \vee n = 0 \text{ then } 0 \text{ else } (m \text{ mod } 2) * (n \text{ mod } 2) + 2 * ((m \text{ div } 2) \text{ AND } (n \text{ div } 2))) \rangle$

for $m \ n :: \text{nat}$

by (*auto simp add: and-nat-rec [of m n] elim: oddE*)

lemma *or-nat-unfold [code]*:

$\langle m \text{ OR } n = (\text{if } m = 0 \text{ then } n \text{ else if } n = 0 \text{ then } m$

$\text{else } \max (m \text{ mod } 2) (n \text{ mod } 2) + 2 * ((m \text{ div } 2) \text{ OR } (n \text{ div } 2))) \rangle$ **for** $m \ n ::$

nat

by (*auto simp add: or-nat-rec [of m n] elim: oddE*)

lemma *xor-nat-unfold [code]*:

$\langle m \text{ XOR } n = (\text{if } m = 0 \text{ then } n \text{ else if } n = 0 \text{ then } m$

$\text{else } (m \text{ mod } 2 + n \text{ mod } 2) \text{ mod } 2 + 2 * ((m \text{ div } 2) \text{ XOR } (n \text{ div } 2))) \rangle$ **for** $m \ n$

:: nat

by (auto simp add: xor-nat-rec [of m n] elim!: oddE)

lemma [code]:

⟨unset-bit 0 m = 2 * (m div 2)⟩

⟨unset-bit (Suc n) m = m mod 2 + 2 * unset-bit n (m div 2)⟩ for m n :: nat

by (simp-all add: unset-bit-Suc)

lemma push-bit-of-Suc-0 [simp]:

⟨push-bit n (Suc 0) = 2 ^ n⟩

using push-bit-of-1 [where ?'a = nat] by simp

lemma take-bit-of-Suc-0 [simp]:

⟨take-bit n (Suc 0) = of-bool (0 < n)⟩

using take-bit-of-1 [where ?'a = nat] by simp

lemma drop-bit-of-Suc-0 [simp]:

⟨drop-bit n (Suc 0) = of-bool (n = 0)⟩

using drop-bit-of-1 [where ?'a = nat] by simp

lemma Suc-mask-eq-exp:

⟨Suc (mask n) = 2 ^ n⟩

by (simp add: mask-eq-exp-minus-1)

lemma less-eq-mask:

⟨n ≤ mask n⟩

by (simp add: mask-eq-exp-minus-1 le-diff-conv2)

(metis Suc-mask-eq-exp diff-Suc-1 diff-le-diff-pow diff-zero le-refl not-less-eq-eq power-0)

lemma less-mask:

⟨n < mask n⟩ if ⟨Suc 0 < n⟩

proof –

define m where ⟨m = n - 2⟩

with that have *: ⟨n = m + 2⟩

by simp

have ⟨Suc (Suc (Suc m)) < 4 * 2 ^ m⟩

by (induction m) simp-all

then have ⟨Suc (m + 2) < Suc (mask (m + 2))⟩

by (simp add: Suc-mask-eq-exp)

then have ⟨m + 2 < mask (m + 2)⟩

by (simp add: less-le)

with * show ?thesis

by simp

qed

lemma mask-nat-less-exp [simp]:

⟨(mask n :: nat) < 2 ^ n⟩

by (simp add: mask-eq-exp-minus-1)

```

lemma mask-nat-positive-iff [simp]:
  ⟨(0::nat) < mask n ↔ 0 < n⟩
proof (cases ⟨n = 0⟩)
  case True
    then show ?thesis
    by simp
  next
  case False
    then have ⟨0 < n⟩
    by simp
    then have ⟨(0::nat) < mask n⟩
    using less-eq-mask [of n] by (rule order-less-le-trans)
    with ⟨0 < n⟩ show ?thesis
    by simp
qed

lemma take-bit-tightened-less-eq-nat:
  ⟨take-bit m q ≤ take-bit n q⟩ if ⟨m ≤ n⟩ for q :: nat
proof –
  have ⟨take-bit m (take-bit n q) ≤ take-bit n q⟩
  by (rule take-bit-nat-less-eq-self)
  with that show ?thesis
  by simp
qed

lemma push-bit-nat-eq:
  ⟨push-bit n (nat k) = nat (push-bit n k)⟩
  by (cases ⟨k ≥ 0⟩) (simp-all add: push-bit-eq-mult nat-mult-distrib not-le mult-nonneg-nonpos2)

lemma drop-bit-nat-eq:
  ⟨drop-bit n (nat k) = nat (drop-bit n k)⟩
  apply (cases ⟨k ≥ 0⟩)
  apply (simp-all add: drop-bit-eq-div nat-div-distrib nat-power-eq not-le)
  apply (simp add: divide-int-def)
  done

lemma take-bit-nat-eq:
  ⟨take-bit n (nat k) = nat (take-bit n k)⟩ if ⟨k ≥ 0⟩
  using that by (simp add: take-bit-eq-mod nat-mod-distrib nat-power-eq)

lemma nat-take-bit-eq:
  ⟨nat (take-bit n k) = take-bit n (nat k)⟩
  if ⟨k ≥ 0⟩
  using that by (simp add: take-bit-eq-mod nat-mod-distrib nat-power-eq)

context semiring-bit-operations
begin

```

lemma *push-bit-of-nat*:
 $\langle \text{push-bit } n \text{ (of-nat } m) = \text{of-nat (push-bit } n \text{ } m) \rangle$
by (*simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *of-nat-push-bit*:
 $\langle \text{of-nat (push-bit } m \text{ } n) = \text{push-bit } m \text{ (of-nat } n) \rangle$
by (*simp add: push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *take-bit-of-nat*:
 $\langle \text{take-bit } n \text{ (of-nat } m) = \text{of-nat (take-bit } n \text{ } m) \rangle$
by (*rule bit-eqI*) (*simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-nat-iff*)

lemma *of-nat-take-bit*:
 $\langle \text{of-nat (take-bit } n \text{ } m) = \text{take-bit } n \text{ (of-nat } m) \rangle$
by (*rule bit-eqI*) (*simp add: bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-nat-iff*)

end

context *semiring-bit-operations*
begin

lemma *of-nat-and-eq*:
 $\langle \text{of-nat } (m \text{ AND } n) = \text{of-nat } m \text{ AND of-nat } n \rangle$
by (*rule bit-eqI*) (*simp add: bit-of-nat-iff bit-and-iff Bit-Operations.bit-and-iff*)

lemma *of-nat-or-eq*:
 $\langle \text{of-nat } (m \text{ OR } n) = \text{of-nat } m \text{ OR of-nat } n \rangle$
by (*rule bit-eqI*) (*simp add: bit-of-nat-iff bit-or-iff Bit-Operations.bit-or-iff*)

lemma *of-nat-xor-eq*:
 $\langle \text{of-nat } (m \text{ XOR } n) = \text{of-nat } m \text{ XOR of-nat } n \rangle$
by (*rule bit-eqI*) (*simp add: bit-of-nat-iff bit-xor-iff Bit-Operations.bit-xor-iff*)

lemma *of-nat-mask-eq*:
 $\langle \text{of-nat (mask } n) = \text{mask } n \rangle$
by (*induction n*) (*simp-all add: mask-Suc-double Bit-Operations.mask-Suc-double of-nat-or-eq*)

end

lemma *nat-mask-eq*:
 $\langle \text{nat (mask } n) = \text{mask } n \rangle$
by (*simp add: nat-eq-iff of-nat-mask-eq*)

69.5 Common algebraic structure

class *unique-euclidean-semiring-with-bit-operations* =
unique-euclidean-semiring-with-nat + *semiring-bit-operations*
begin

lemma *possible-bit* [simp]:
 $\langle \text{possible-bit TYPE}(a) n \rangle$
by (simp add: possible-bit-def)

lemma *take-bit-of-exp* [simp]:
 $\langle \text{take-bit } n (2 \wedge n) = \text{of-bool } (n < m) * 2 \wedge n \rangle$
by (simp add: take-bit-eq-mod exp-mod-exp)

lemma *take-bit-of-2* [simp]:
 $\langle \text{take-bit } n 2 = \text{of-bool } (2 \leq n) * 2 \rangle$
using take-bit-of-exp [of n 1] **by** simp

lemma *push-bit-eq-0-iff* [simp]:
 $\text{push-bit } n a = 0 \longleftrightarrow a = 0$
by (simp add: push-bit-eq-mult)

lemma *take-bit-add*:
 $\text{take-bit } n (\text{take-bit } n a + \text{take-bit } n b) = \text{take-bit } n (a + b)$
by (simp add: take-bit-eq-mod mod-simps)

lemma *take-bit-of-1-eq-0-iff* [simp]:
 $\text{take-bit } n 1 = 0 \longleftrightarrow n = 0$
by (simp add: take-bit-eq-mod)

lemma *drop-bit-Suc-bit0* [simp]:
 $\langle \text{drop-bit } (\text{Suc } n) (\text{numeral } (\text{Num.Bit0 } k)) = \text{drop-bit } n (\text{numeral } k) \rangle$
by (simp add: drop-bit-Suc numeral-Bit0-div-2)

lemma *drop-bit-Suc-bit1* [simp]:
 $\langle \text{drop-bit } (\text{Suc } n) (\text{numeral } (\text{Num.Bit1 } k)) = \text{drop-bit } n (\text{numeral } k) \rangle$
by (simp add: drop-bit-Suc numeral-Bit1-div-2)

lemma *drop-bit-numeral-bit0* [simp]:
 $\langle \text{drop-bit } (\text{numeral } l) (\text{numeral } (\text{Num.Bit0 } k)) = \text{drop-bit } (\text{pred-numeral } l) (\text{numeral } k) \rangle$
by (simp add: drop-bit-rec numeral-Bit0-div-2)

lemma *drop-bit-numeral-bit1* [simp]:
 $\langle \text{drop-bit } (\text{numeral } l) (\text{numeral } (\text{Num.Bit1 } k)) = \text{drop-bit } (\text{pred-numeral } l) (\text{numeral } k) \rangle$
by (simp add: drop-bit-rec numeral-Bit1-div-2)

lemma *take-bit-Suc-1* [simp]:
 $\langle \text{take-bit } (\text{Suc } n) 1 = 1 \rangle$
by (simp add: take-bit-Suc)

lemma *take-bit-Suc-bit0*:
 $\langle \text{take-bit } (\text{Suc } n) (\text{numeral } (\text{Num.Bit0 } k)) = \text{take-bit } n (\text{numeral } k) * 2 \rangle$

by (*simp add: take-bit-Suc numeral-Bit0-div-2*)

lemma *take-bit-Suc-bit1*:

$\langle \text{take-bit } (\text{Suc } n) \text{ (numeral (Num.Bit1 } k)) = \text{take-bit } n \text{ (numeral } k) * 2 + 1 \rangle$

by (*simp add: take-bit-Suc numeral-Bit1-div-2 mod-2-eq-odd*)

lemma *take-bit-numeral-1 [simp]*:

$\langle \text{take-bit (numeral } l) 1 = 1 \rangle$

by (*simp add: take-bit-rec [of <numeral l> 1]*)

lemma *take-bit-numeral-bit0*:

$\langle \text{take-bit (numeral } l) \text{ (numeral (Num.Bit0 } k)) = \text{take-bit (pred-numeral } l) \text{ (numeral } k) * 2 \rangle$

by (*simp add: take-bit-rec numeral-Bit0-div-2*)

lemma *take-bit-numeral-bit1*:

$\langle \text{take-bit (numeral } l) \text{ (numeral (Num.Bit1 } k)) = \text{take-bit (pred-numeral } l) \text{ (numeral } k) * 2 + 1 \rangle$

by (*simp add: take-bit-rec numeral-Bit1-div-2 mod-2-eq-odd*)

lemma *bit-of-nat-iff-bit [bit-simps]*:

$\langle \text{bit (of-nat } m) n \longleftrightarrow \text{bit } m n \rangle$

proof –

have $\langle \text{even } (m \text{ div } 2 \wedge n) \longleftrightarrow \text{even } (\text{of-nat } (m \text{ div } 2 \wedge n)) \rangle$

by *simp*

also have $\langle \text{of-nat } (m \text{ div } 2 \wedge n) = \text{of-nat } m \text{ div of-nat } (2 \wedge n) \rangle$

by (*simp add: of-nat-div*)

finally show *?thesis*

by (*simp add: bit-iff-odd semiring-bits-class.bit-iff-odd*)

qed

lemma *drop-bit-mask-eq*:

$\langle \text{drop-bit } m \text{ (mask } n) = \text{mask } (n - m) \rangle$

by (*rule bit-eqI (auto simp add: bit-simps possible-bit-def)*)

lemma *drop-bit-of-nat*:

$\text{drop-bit } n \text{ (of-nat } m) = \text{of-nat } (\text{drop-bit } n m)$

by (*simp add: drop-bit-eq-div Bit-Operations.drop-bit-eq-div of-nat-div [of m 2 \wedge n]*)

lemma *of-nat-drop-bit*:

$\langle \text{of-nat } (\text{drop-bit } m n) = \text{drop-bit } m \text{ (of-nat } n) \rangle$

by (*simp add: drop-bit-eq-div Bit-Operations.drop-bit-eq-div of-nat-div*)

lemma *take-bit-sum*:

$\text{take-bit } n a = (\sum k = 0..<n. \text{push-bit } k \text{ (of-bool (bit } a k)))$

for $n :: \text{nat}$

proof (*induction n arbitrary: a*)

case 0

```

then show ?case
  by simp
next
  case (Suc n)
  have ( $\sum k = 0..<Suc\ n.\ push-bit\ k\ (of-bool\ (bit\ a\ k))$ ) =
     $of-bool\ (odd\ a) + (\sum k = Suc\ 0..<Suc\ n.\ push-bit\ k\ (of-bool\ (bit\ a\ k)))$ 
    by (simp add: sum.atLeast-Suc-lessThan ac-simps bit-0)
  also have ( $\sum k = Suc\ 0..<Suc\ n.\ push-bit\ k\ (of-bool\ (bit\ a\ k))$ )
    = ( $\sum k = 0..<n.\ push-bit\ k\ (of-bool\ (bit\ (a\ div\ 2)\ k))$ ) * 2
    by (simp only: sum.atLeast-Suc-lessThan-Suc-shift) (simp add: sum-distrib-right
push-bit-double drop-bit-Suc bit-Suc)
  finally show ?case
    using Suc [of a div 2] by (simp add: ac-simps take-bit-Suc mod-2-eq-odd)
qed

end

instance nat :: unique-euclidean-semiring-with-bit-operations ..

instance int :: unique-euclidean-semiring-with-bit-operations ..

lemma drop-bit-Suc-minus-bit0 [simp]:
   $\langle drop-bit\ (Suc\ n)\ (-\ numeral\ (Num.Bit0\ k)) = drop-bit\ n\ (-\ numeral\ k\ ::\ int) \rangle$ 
  by (simp add: drop-bit-Suc numeral-Bit0-div-2)

lemma drop-bit-Suc-minus-bit1 [simp]:
   $\langle drop-bit\ (Suc\ n)\ (-\ numeral\ (Num.Bit1\ k)) = drop-bit\ n\ (-\ numeral\ (Num.inc\ k)\ ::\ int) \rangle$ 
  by (simp add: drop-bit-Suc numeral-Bit1-div-2 add-One)

lemma drop-bit-numeral-minus-bit0 [simp]:
   $\langle drop-bit\ (numeral\ l)\ (-\ numeral\ (Num.Bit0\ k)) = drop-bit\ (pred-numeral\ l)\ (-\ numeral\ k\ ::\ int) \rangle$ 
  by (simp add: numeral-eq-Suc numeral-Bit0-div-2)

lemma drop-bit-numeral-minus-bit1 [simp]:
   $\langle drop-bit\ (numeral\ l)\ (-\ numeral\ (Num.Bit1\ k)) = drop-bit\ (pred-numeral\ l)\ (-\ numeral\ (Num.inc\ k)\ ::\ int) \rangle$ 
  by (simp add: numeral-eq-Suc numeral-Bit1-div-2)

lemma take-bit-Suc-minus-bit0:
   $\langle take-bit\ (Suc\ n)\ (-\ numeral\ (Num.Bit0\ k)) = take-bit\ n\ (-\ numeral\ k) * (2\ ::\ int) \rangle$ 
  by (simp add: take-bit-Suc numeral-Bit0-div-2)

lemma take-bit-Suc-minus-bit1:
   $\langle take-bit\ (Suc\ n)\ (-\ numeral\ (Num.Bit1\ k)) = take-bit\ n\ (-\ numeral\ (Num.inc\ k)) * 2 + (1\ ::\ int) \rangle$ 
  by (simp add: take-bit-Suc numeral-Bit1-div-2 add-One)

```

lemma *take-bit-numeral-minus-bit0*:

$\langle \text{take-bit } (\text{numeral } l) \text{ } (- \text{ numeral } (\text{Num.Bit0 } k)) = \text{take-bit } (\text{pred-numeral } l) \text{ } (- \text{ numeral } k) * (2 :: \text{int}) \rangle$

by (*simp add: numeral-eq-Suc numeral-Bit0-div-2 take-bit-Suc-minus-bit0*)

lemma *take-bit-numeral-minus-bit1*:

$\langle \text{take-bit } (\text{numeral } l) \text{ } (- \text{ numeral } (\text{Num.Bit1 } k)) = \text{take-bit } (\text{pred-numeral } l) \text{ } (- \text{ numeral } (\text{Num.inc } k)) * 2 + (1 :: \text{int}) \rangle$

by (*simp add: numeral-eq-Suc numeral-Bit1-div-2 take-bit-Suc-minus-bit1*)

69.6 Symbolic computations on numeral expressions

context *semiring-bits*

begin

lemma *not-bit-numeral-Bit0-0* [*simp*]:

$\langle \neg \text{bit } (\text{numeral } (\text{Num.Bit0 } m)) \text{ } 0 \rangle$

by (*simp add: bit-0*)

lemma *bit-numeral-Bit1-0* [*simp*]:

$\langle \text{bit } (\text{numeral } (\text{Num.Bit1 } m)) \text{ } 0 \rangle$

by (*simp add: bit-0*)

end

context *ring-bit-operations*

begin

lemma *not-bit-minus-numeral-Bit0-0* [*simp*]:

$\langle \neg \text{bit } (- \text{ numeral } (\text{Num.Bit0 } m)) \text{ } 0 \rangle$

by (*simp add: bit-0*)

lemma *bit-minus-numeral-Bit1-0* [*simp*]:

$\langle \text{bit } (- \text{ numeral } (\text{Num.Bit1 } m)) \text{ } 0 \rangle$

by (*simp add: bit-0*)

end

context *unique-euclidean-semiring-with-bit-operations*

begin

lemma *bit-numeral-iff*:

$\langle \text{bit } (\text{numeral } m) \text{ } n \longleftrightarrow \text{bit } (\text{numeral } m :: \text{nat}) \text{ } n \rangle$

using *bit-of-nat-iff-bit* [*of* $\langle \text{numeral } m \rangle \text{ } n$] **by** *simp*

lemma *bit-numeral-Bit0-Suc-iff* [*simp*]:

$\langle \text{bit } (\text{numeral } (\text{Num.Bit0 } m)) \text{ } (\text{Suc } n) \longleftrightarrow \text{bit } (\text{numeral } m) \text{ } n \rangle$

by (*simp add: bit-Suc numeral-Bit0-div-2*)

lemma *bit-numeral-Bit1-Suc-iff* [simp]:

⟨*bit (numeral (Num.Bit1 m)) (Suc n) ⟷ bit (numeral m) n*⟩
by (simp add: bit-Suc numeral-Bit1-div-2)

lemma *bit-numeral-rec*:

⟨*bit (numeral (Num.Bit0 w)) n ⟷ (case n of 0 ⇒ False | Suc m ⇒ bit (numeral w) m)*⟩
 ⟨*bit (numeral (Num.Bit1 w)) n ⟷ (case n of 0 ⇒ True | Suc m ⇒ bit (numeral w) m)*⟩
by (cases n; simp add: bit-0)+

lemma *bit-numeral-simps* [simp]:

⟨ \neg *bit 1 (numeral n)*⟩
 ⟨*bit (numeral (Num.Bit0 w)) (numeral n) ⟷ bit (numeral w) (pred-numeral n)*⟩
 ⟨*bit (numeral (Num.Bit1 w)) (numeral n) ⟷ bit (numeral w) (pred-numeral n)*⟩
by (simp-all add: bit-1-iff numeral-eq-Suc)

lemma *and-numerals* [simp]:

⟨*1 AND numeral (Num.Bit0 y) = 0*⟩
 ⟨*1 AND numeral (Num.Bit1 y) = 1*⟩
 ⟨*numeral (Num.Bit0 x) AND numeral (Num.Bit0 y) = 2 * (numeral x AND numeral y)*⟩
 ⟨*numeral (Num.Bit0 x) AND numeral (Num.Bit1 y) = 2 * (numeral x AND numeral y)*⟩
 ⟨*numeral (Num.Bit0 x) AND 1 = 0*⟩
 ⟨*numeral (Num.Bit1 x) AND numeral (Num.Bit0 y) = 2 * (numeral x AND numeral y)*⟩
 ⟨*numeral (Num.Bit1 x) AND numeral (Num.Bit1 y) = 1 + 2 * (numeral x AND numeral y)*⟩
 ⟨*numeral (Num.Bit1 x) AND 1 = 1*⟩
by (simp-all add: bit-eq-iff) (simp-all add: bit-0 bit-simps bit-Suc bit-numeral-rec split: nat.splits)

fun *and-num* :: ⟨*num ⇒ num ⇒ num option*⟩

where

⟨*and-num num.One num.One = Some num.One*⟩
 | ⟨*and-num num.One (num.Bit0 n) = None*⟩
 | ⟨*and-num num.One (num.Bit1 n) = Some num.One*⟩
 | ⟨*and-num (num.Bit0 m) num.One = None*⟩
 | ⟨*and-num (num.Bit0 m) (num.Bit0 n) = map-option num.Bit0 (and-num m n)*⟩
 | ⟨*and-num (num.Bit0 m) (num.Bit1 n) = map-option num.Bit0 (and-num m n)*⟩
 | ⟨*and-num (num.Bit1 m) num.One = Some num.One*⟩
 | ⟨*and-num (num.Bit1 m) (num.Bit0 n) = map-option num.Bit0 (and-num m n)*⟩
 | ⟨*and-num (num.Bit1 m) (num.Bit1 n) = (case and-num m n of None ⇒ Some num.One | Some n' ⇒ Some (num.Bit1 n'))*⟩

lemma *numeral-and-num*:

⟨*numeral m AND numeral n = (case and-num m n of None ⇒ 0 | Some n' ⇒*

numeral n›)

by (*induction m n rule: and-num.induct*) (*simp-all add: split: option.split*)

lemma *and-num-eq-None-iff*:

⟨*and-num m n = None* \longleftrightarrow *numeral m AND numeral n = 0*›

by (*simp add: numeral-and-num split: option.split*)

lemma *and-num-eq-Some-iff*:

⟨*and-num m n = Some q* \longleftrightarrow *numeral m AND numeral n = numeral q*›

by (*simp add: numeral-and-num split: option.split*)

lemma *or-numerals [simp]*:

⟨*1 OR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)*›

⟨*1 OR numeral (Num.Bit1 y) = numeral (Num.Bit1 y)*›

⟨*numeral (Num.Bit0 x) OR numeral (Num.Bit0 y) = 2 * (numeral x OR numeral y)*›

⟨*numeral (Num.Bit0 x) OR numeral (Num.Bit1 y) = 1 + 2 * (numeral x OR numeral y)*›

⟨*numeral (Num.Bit0 x) OR 1 = numeral (Num.Bit1 x)*›

⟨*numeral (Num.Bit1 x) OR numeral (Num.Bit0 y) = 1 + 2 * (numeral x OR numeral y)*›

⟨*numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = 1 + 2 * (numeral x OR numeral y)*›

⟨*numeral (Num.Bit1 x) OR 1 = numeral (Num.Bit1 x)*›

by (*simp-all add: bit-eq-iff*) (*simp-all add: bit-0 bit-simps bit-Suc bit-numeral-rec split: nat.splits*)

fun *or-num* :: ⟨*num* \Rightarrow *num* \Rightarrow *num*›

where

⟨*or-num num.One num.One = num.One*›

| ⟨*or-num num.One (num.Bit0 n) = num.Bit1 n*›

| ⟨*or-num num.One (num.Bit1 n) = num.Bit1 n*›

| ⟨*or-num (num.Bit0 m) num.One = num.Bit1 m*›

| ⟨*or-num (num.Bit0 m) (num.Bit0 n) = num.Bit0 (or-num m n)*›

| ⟨*or-num (num.Bit0 m) (num.Bit1 n) = num.Bit1 (or-num m n)*›

| ⟨*or-num (num.Bit1 m) num.One = num.Bit1 m*›

| ⟨*or-num (num.Bit1 m) (num.Bit0 n) = num.Bit1 (or-num m n)*›

| ⟨*or-num (num.Bit1 m) (num.Bit1 n) = num.Bit1 (or-num m n)*›

lemma *numeral-or-num*:

⟨*numeral m OR numeral n = numeral (or-num m n)*›

by (*induction m n rule: or-num.induct*) *simp-all*

lemma *numeral-or-num-eq*:

⟨*numeral (or-num m n) = numeral m OR numeral n*›

by (*simp add: numeral-or-num*)

lemma *xor-numerals [simp]*:

⟨*1 XOR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)*›

$\langle 1 \text{ XOR numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit0 } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ XOR numeral } (\text{Num.Bit0 } y) = 2 * (\text{numeral } x \text{ XOR numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ XOR numeral } (\text{Num.Bit1 } y) = 1 + 2 * (\text{numeral } x \text{ XOR numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ XOR } 1 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ XOR numeral } (\text{Num.Bit0 } y) = 1 + 2 * (\text{numeral } x \text{ XOR numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ XOR numeral } (\text{Num.Bit1 } y) = 2 * (\text{numeral } x \text{ XOR numeral } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ XOR } 1 = \text{numeral } (\text{Num.Bit0 } x) \rangle$
by (*simp-all add: bit-eq-iff*) (*simp-all add: bit-0 bit-simps bit-Suc bit-numeral-rec split: nat.splits*)

fun *xor-num* :: $\langle \text{num} \Rightarrow \text{num} \Rightarrow \text{num option} \rangle$

where

$\langle \text{xor-num } \text{num.One } \text{num.One} = \text{None} \rangle$
 $\langle \text{xor-num } \text{num.One } (\text{num.Bit0 } n) = \text{Some } (\text{num.Bit1 } n) \rangle$
 $\langle \text{xor-num } \text{num.One } (\text{num.Bit1 } n) = \text{Some } (\text{num.Bit0 } n) \rangle$
 $\langle \text{xor-num } (\text{num.Bit0 } m) \text{ num.One} = \text{Some } (\text{num.Bit1 } m) \rangle$
 $\langle \text{xor-num } (\text{num.Bit0 } m) (\text{num.Bit0 } n) = \text{map-option } \text{num.Bit0 } (\text{xor-num } m n) \rangle$
 $\langle \text{xor-num } (\text{num.Bit0 } m) (\text{num.Bit1 } n) = \text{Some } (\text{case } \text{xor-num } m n \text{ of } \text{None} \Rightarrow \text{num.One} \mid \text{Some } n' \Rightarrow \text{num.Bit1 } n') \rangle$
 $\langle \text{xor-num } (\text{num.Bit1 } m) \text{ num.One} = \text{Some } (\text{num.Bit0 } m) \rangle$
 $\langle \text{xor-num } (\text{num.Bit1 } m) (\text{num.Bit0 } n) = \text{Some } (\text{case } \text{xor-num } m n \text{ of } \text{None} \Rightarrow \text{num.One} \mid \text{Some } n' \Rightarrow \text{num.Bit1 } n') \rangle$
 $\langle \text{xor-num } (\text{num.Bit1 } m) (\text{num.Bit1 } n) = \text{map-option } \text{num.Bit0 } (\text{xor-num } m n) \rangle$

lemma *numeral-xor-num*:

$\langle \text{numeral } m \text{ XOR numeral } n = (\text{case } \text{xor-num } m n \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$

by (*induction m n rule: xor-num.induct*) (*simp-all split: option.split*)

lemma *xor-num-eq-None-iff*:

$\langle \text{xor-num } m n = \text{None} \iff \text{numeral } m \text{ XOR numeral } n = 0 \rangle$

by (*simp add: numeral-xor-num split: option.split*)

lemma *xor-num-eq-Some-iff*:

$\langle \text{xor-num } m n = \text{Some } q \iff \text{numeral } m \text{ XOR numeral } n = \text{numeral } q \rangle$

by (*simp add: numeral-xor-num split: option.split*)

end

lemma *bit-Suc-0-iff* [*bit-simps*]:

$\langle \text{bit } (\text{Suc } 0) n \iff n = 0 \rangle$

using *bit-1-iff* [*of n, where ?'a = nat*] **by** *simp*

lemma *and-nat-numerals* [*simp*]:

$\langle \text{Suc } 0 \text{ AND numeral } (\text{Num.Bit0 } y) = 0 \rangle$

$\langle \text{Suc } 0 \text{ AND numeral } (\text{Num.Bit1 } y) = 1 \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ AND Suc } 0 = 0 \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ AND Suc } 0 = 1 \rangle$
by (*simp-all only: and-numerals flip: One-nat-def*)

lemma *or-nat-numerals [simp]*:
 $\langle \text{Suc } 0 \text{ OR numeral } (\text{Num.Bit0 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle \text{Suc } 0 \text{ OR numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ OR Suc } 0 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ OR Suc } 0 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
by (*simp-all only: or-numerals flip: One-nat-def*)

lemma *xor-nat-numerals [simp]*:
 $\langle \text{Suc } 0 \text{ XOR numeral } (\text{Num.Bit0 } y) = \text{numeral } (\text{Num.Bit1 } y) \rangle$
 $\langle \text{Suc } 0 \text{ XOR numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit0 } y) \rangle$
 $\langle \text{numeral } (\text{Num.Bit0 } x) \text{ XOR Suc } 0 = \text{numeral } (\text{Num.Bit1 } x) \rangle$
 $\langle \text{numeral } (\text{Num.Bit1 } x) \text{ XOR Suc } 0 = \text{numeral } (\text{Num.Bit0 } x) \rangle$
by (*simp-all only: xor-numerals flip: One-nat-def*)

context *ring-bit-operations*
begin

lemma *minus-numeral-inc-eq*:
 $\langle - \text{ numeral } (\text{Num.inc } n) = \text{NOT } (\text{numeral } n) \rangle$
by (*simp add: not-eq-complement sub-inc-One-eq add-One*)

lemma *sub-one-eq-not-neg*:
 $\langle \text{Num.sub } n \text{ num.One} = \text{NOT } (- \text{ numeral } n) \rangle$
by (*simp add: not-eq-complement*)

lemma *minus-numeral-eq-not-sub-one*:
 $\langle - \text{ numeral } n = \text{NOT } (\text{Num.sub } n \text{ num.One}) \rangle$
by (*simp add: not-eq-complement*)

lemma *not-numeral-eq [simp]*:
 $\langle \text{NOT } (\text{numeral } n) = - \text{ numeral } (\text{Num.inc } n) \rangle$
by (*simp add: minus-numeral-inc-eq*)

lemma *not-minus-numeral-eq [simp]*:
 $\langle \text{NOT } (- \text{ numeral } n) = \text{Num.sub } n \text{ num.One} \rangle$
by (*simp add: sub-one-eq-not-neg*)

lemma *minus-not-numeral-eq [simp]*:
 $\langle - (\text{NOT } (\text{numeral } n)) = \text{numeral } (\text{Num.inc } n) \rangle$
by *simp*

lemma *not-numeral-BitM-eq*:
 $\langle \text{NOT } (\text{numeral } (\text{Num.BitM } n)) = - \text{ numeral } (\text{num.Bit0 } n) \rangle$
by (*simp add: inc-BitM-eq*)

lemma *not-numeral-Bit0-eq*:

⟨*NOT* (numeral (Num.Bit0 n)) = − numeral (num.Bit1 n)⟩
by *simp*

end

lemma *bit-minus-numeral-int* [*simp*]:

⟨*bit* (− numeral (num.Bit0 w) :: int) (numeral n) ⟷ *bit* (− numeral w :: int)
(pred-numeral n)⟩
⟨*bit* (− numeral (num.Bit1 w) :: int) (numeral n) ⟷ ¬ *bit* (numeral w :: int)
(pred-numeral n)⟩
by (*simp-all add: bit-minus-iff bit-not-iff numeral-eq-Suc bit-Suc add-One sub-inc-One-eq*)

lemma *bit-minus-numeral-Bit0-Suc-iff* [*simp*]:

⟨*bit* (− numeral (num.Bit0 w) :: int) (Suc n) ⟷ *bit* (− numeral w :: int) n⟩
by (*simp add: bit-Suc*)

lemma *bit-minus-numeral-Bit1-Suc-iff* [*simp*]:

⟨*bit* (− numeral (num.Bit1 w) :: int) (Suc n) ⟷ ¬ *bit* (numeral w :: int) n⟩
by (*simp add: bit-Suc add-One flip: bit-not-int-iff*)

lemma *and-not-numerals*:

⟨1 AND NOT 1 = (0 :: int)⟩
⟨1 AND NOT (numeral (Num.Bit0 n)) = (1 :: int)⟩
⟨1 AND NOT (numeral (Num.Bit1 n)) = (0 :: int)⟩
⟨numeral (Num.Bit0 m) AND NOT (1 :: int) = numeral (Num.Bit0 m)⟩
⟨numeral (Num.Bit0 m) AND NOT (numeral (Num.Bit0 n)) = (2 :: int) *
(numeral m AND NOT (numeral n))⟩
⟨numeral (Num.Bit0 m) AND NOT (numeral (Num.Bit1 n)) = (2 :: int) *
(numeral m AND NOT (numeral n))⟩
⟨numeral (Num.Bit1 m) AND NOT (1 :: int) = numeral (Num.Bit0 m)⟩
⟨numeral (Num.Bit1 m) AND NOT (numeral (Num.Bit0 n)) = 1 + (2 :: int) *
(numeral m AND NOT (numeral n))⟩
⟨numeral (Num.Bit1 m) AND NOT (numeral (Num.Bit1 n)) = (2 :: int) *
(numeral m AND NOT (numeral n))⟩
by (*simp-all add: bit-eq-iff*) (*auto simp add: bit-0 bit-simps bit-Suc bit-numeral-rec
BitM-inc-eq sub-inc-One-eq split: nat.split*)

fun *and-not-num* :: ⟨num ⇒ num ⇒ num option⟩

where

⟨*and-not-num* num.One num.One = None⟩
| ⟨*and-not-num* num.One (num.Bit0 n) = Some num.One⟩
| ⟨*and-not-num* num.One (num.Bit1 n) = None⟩
| ⟨*and-not-num* (num.Bit0 m) num.One = Some (num.Bit0 m)⟩
| ⟨*and-not-num* (num.Bit0 m) (num.Bit0 n) = map-option num.Bit0 (and-not-num
m n)⟩
| ⟨*and-not-num* (num.Bit0 m) (num.Bit1 n) = map-option num.Bit0 (and-not-num
m n)⟩

$\langle \text{and-not-num } (\text{num.Bit1 } m) \text{ num.One} = \text{Some } (\text{num.Bit0 } m) \rangle$
 $\langle \text{and-not-num } (\text{num.Bit1 } m) (\text{num.Bit0 } n) = (\text{case and-not-num } m \text{ } n \text{ of None} \Rightarrow \text{Some num.One} \mid \text{Some } n' \Rightarrow \text{Some } (\text{num.Bit1 } n')) \rangle$
 $\langle \text{and-not-num } (\text{num.Bit1 } m) (\text{num.Bit1 } n) = \text{map-option num.Bit0 } (\text{and-not-num } m \text{ } n) \rangle$

lemma *int-numeral-and-not-num*:

$\langle \text{numeral } m \text{ AND NOT } (\text{numeral } n) = (\text{case and-not-num } m \text{ } n \text{ of None} \Rightarrow 0 \text{ :: int} \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$

by (*induction* $m \ n$ *rule*: *and-not-num.induct*) (*simp-all del*: *not-numeral-eq not-one-eq add*: *and-not-numerals split*: *option.split*)

lemma *int-numeral-not-and-num*:

$\langle \text{NOT } (\text{numeral } m) \text{ AND numeral } n = (\text{case and-not-num } n \text{ } m \text{ of None} \Rightarrow 0 \text{ :: int} \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$

using *int-numeral-and-not-num* [*of* $n \ m$] **by** (*simp add*: *ac-simps*)

lemma *and-not-num-eq-None-iff*:

$\langle \text{and-not-num } m \text{ } n = \text{None} \iff \text{numeral } m \text{ AND NOT } (\text{numeral } n) = (0 \text{ :: int}) \rangle$

by (*simp del*: *not-numeral-eq add*: *int-numeral-and-not-num split*: *option.split*)

lemma *and-not-num-eq-Some-iff*:

$\langle \text{and-not-num } m \text{ } n = \text{Some } q \iff \text{numeral } m \text{ AND NOT } (\text{numeral } n) = (\text{numeral } q \text{ :: int}) \rangle$

by (*simp del*: *not-numeral-eq add*: *int-numeral-and-not-num split*: *option.split*)

lemma *and-minus-numerals* [*simp*]:

$\langle 1 \text{ AND } - (\text{numeral } (\text{num.Bit0 } n)) = (0 \text{ :: int}) \rangle$

$\langle 1 \text{ AND } - (\text{numeral } (\text{num.Bit1 } n)) = (1 \text{ :: int}) \rangle$

$\langle \text{numeral } m \text{ AND } - (\text{numeral } (\text{num.Bit0 } n)) = (\text{case and-not-num } m \text{ } (\text{Num.BitM } n) \text{ of None} \Rightarrow 0 \text{ :: int} \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$

$\langle \text{numeral } m \text{ AND } - (\text{numeral } (\text{num.Bit1 } n)) = (\text{case and-not-num } m \text{ } (\text{Num.Bit0 } n) \text{ of None} \Rightarrow 0 \text{ :: int} \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$

$\langle - (\text{numeral } (\text{num.Bit0 } n)) \text{ AND } 1 = (0 \text{ :: int}) \rangle$

$\langle - (\text{numeral } (\text{num.Bit1 } n)) \text{ AND } 1 = (1 \text{ :: int}) \rangle$

$\langle - (\text{numeral } (\text{num.Bit0 } n)) \text{ AND numeral } m = (\text{case and-not-num } m \text{ } (\text{Num.BitM } n) \text{ of None} \Rightarrow 0 \text{ :: int} \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$

$\langle - (\text{numeral } (\text{num.Bit1 } n)) \text{ AND numeral } m = (\text{case and-not-num } m \text{ } (\text{Num.Bit0 } n) \text{ of None} \Rightarrow 0 \text{ :: int} \mid \text{Some } n' \Rightarrow \text{numeral } n') \rangle$

by (*simp-all del*: *not-numeral-eq add*: *ac-simps*

and-not-numerals one-and-eq not-numeral-BitM-eq not-numeral-Bit0-eq and-not-num-eq-None-iff and-not-num-eq-Some-iff split: *option.split*)

lemma *and-minus-minus-numerals* [*simp*]:

$\langle - (\text{numeral } m \text{ :: int}) \text{ AND } - (\text{numeral } n \text{ :: int}) = \text{NOT } ((\text{numeral } m - 1) \text{ OR } (\text{numeral } n - 1)) \rangle$

by (*simp add*: *minus-numeral-eq-not-sub-one*)

lemma *or-not-numerals*:

```

⟨1 OR NOT 1 = NOT (0 :: int)⟩
⟨1 OR NOT (numeral (Num.Bit0 n)) = NOT (numeral (Num.Bit0 n) :: int)⟩
⟨1 OR NOT (numeral (Num.Bit1 n)) = NOT (numeral (Num.Bit0 n) :: int)⟩
⟨numeral (Num.Bit0 m) OR NOT (1 :: int) = NOT (1 :: int)⟩
⟨numeral (Num.Bit0 m) OR NOT (numeral (Num.Bit0 n)) = 1 + (2 :: int) *
(numeral m OR NOT (numeral n))⟩
⟨numeral (Num.Bit0 m) OR NOT (numeral (Num.Bit1 n)) = (2 :: int) *
(numeral m OR NOT (numeral n))⟩
⟨numeral (Num.Bit1 m) OR NOT (1 :: int) = NOT (0 :: int)⟩
⟨numeral (Num.Bit1 m) OR NOT (numeral (Num.Bit0 n)) = 1 + (2 :: int) *
(numeral m OR NOT (numeral n))⟩
⟨numeral (Num.Bit1 m) OR NOT (numeral (Num.Bit1 n)) = 1 + (2 :: int) *
(numeral m OR NOT (numeral n))⟩
by (simp-all add: bit-eq-iff)
      (auto simp add: bit-0 bit-simps bit-Suc bit-numeral-rec sub-inc-One-eq split:
nat.split)

```

fun *or-not-num-neg* :: ⟨num ⇒ num ⇒ num⟩

where

```

⟨or-not-num-neg num.One num.One = num.One⟩
| ⟨or-not-num-neg num.One (num.Bit0 m) = num.Bit1 m⟩
| ⟨or-not-num-neg num.One (num.Bit1 m) = num.Bit1 m⟩
| ⟨or-not-num-neg (num.Bit0 n) num.One = num.Bit0 num.One⟩
| ⟨or-not-num-neg (num.Bit0 n) (num.Bit0 m) = Num.BitM (or-not-num-neg n
m)⟩
| ⟨or-not-num-neg (num.Bit0 n) (num.Bit1 m) = num.Bit0 (or-not-num-neg n m)⟩
| ⟨or-not-num-neg (num.Bit1 n) num.One = num.One⟩
| ⟨or-not-num-neg (num.Bit1 n) (num.Bit0 m) = Num.BitM (or-not-num-neg n
m)⟩
| ⟨or-not-num-neg (num.Bit1 n) (num.Bit1 m) = Num.BitM (or-not-num-neg n
m)⟩

```

lemma *int-numeral-or-not-num-neg*:

```

⟨numeral m OR NOT (numeral n :: int) = - numeral (or-not-num-neg m n)⟩
by (induction m n rule: or-not-num-neg.induct) (simp-all del: not-numeral-eq
not-one-eq add: or-not-numerals, simp-all)

```

lemma *int-numeral-not-or-num-neg*:

```

⟨NOT (numeral m) OR (numeral n :: int) = - numeral (or-not-num-neg n m)⟩
using int-numeral-or-not-num-neg [of n m] by (simp add: ac-simps)

```

lemma *numeral-or-not-num-eq*:

```

⟨numeral (or-not-num-neg m n) = - (numeral m OR NOT (numeral n :: int))⟩
using int-numeral-or-not-num-neg [of m n] by simp

```

lemma *or-minus-numerals* [simp]:

```

⟨1 OR - (numeral (num.Bit0 n)) = - (numeral (or-not-num-neg num.One
(Num.BitM n)) :: int)⟩
⟨1 OR - (numeral (num.Bit1 n)) = - (numeral (num.Bit1 n) :: int)⟩

```

$\langle \text{numeral } m \text{ OR } - (\text{numeral } (\text{num.Bit0 } n)) = - (\text{numeral } (\text{or-not-num-neg } m (\text{Num.BitM } n)) :: \text{int}) \rangle$
 $\langle \text{numeral } m \text{ OR } - (\text{numeral } (\text{num.Bit1 } n)) = - (\text{numeral } (\text{or-not-num-neg } m (\text{Num.Bit0 } n)) :: \text{int}) \rangle$
 $\langle - (\text{numeral } (\text{num.Bit0 } n)) \text{ OR } 1 = - (\text{numeral } (\text{or-not-num-neg } \text{num.One } (\text{Num.BitM } n)) :: \text{int}) \rangle$
 $\langle - (\text{numeral } (\text{num.Bit1 } n)) \text{ OR } 1 = - (\text{numeral } (\text{num.Bit1 } n) :: \text{int}) \rangle$
 $\langle - (\text{numeral } (\text{num.Bit0 } n)) \text{ OR numeral } m = - (\text{numeral } (\text{or-not-num-neg } m (\text{Num.BitM } n)) :: \text{int}) \rangle$
 $\langle - (\text{numeral } (\text{num.Bit1 } n)) \text{ OR numeral } m = - (\text{numeral } (\text{or-not-num-neg } m (\text{Num.Bit0 } n)) :: \text{int}) \rangle$
by (*simp-all only: or.commute [of - 1] or.commute [of - <numeral m>] minus-numeral-eq-not-sub-one or-not-numerals numeral-or-not-num-eq arith-simps minus-minus numeral-One*)

lemma *or-minus-minus-numerals [simp]:*

$\langle - (\text{numeral } m :: \text{int}) \text{ OR } - (\text{numeral } n :: \text{int}) = \text{NOT } ((\text{numeral } m - 1) \text{ AND } (\text{numeral } n - 1)) \rangle$
by (*simp add: minus-numeral-eq-not-sub-one*)

lemma *xor-minus-numerals [simp]:*

$\langle - \text{numeral } n \text{ XOR } k = \text{NOT } (\text{neg-numeral-class.sub } n \text{ num.One XOR } k) \rangle$
 $\langle k \text{ XOR } - \text{numeral } n = \text{NOT } (k \text{ XOR } (\text{neg-numeral-class.sub } n \text{ num.One})) \rangle$ **for**
 $k :: \text{int}$
by (*simp-all add: minus-numeral-eq-not-sub-one*)

definition *take-bit-num :: <nat \Rightarrow num \Rightarrow num option>*

where $\langle \text{take-bit-num } n \text{ } m =$
 $(\text{if } \text{take-bit } n (\text{numeral } m :: \text{nat}) = 0 \text{ then } \text{None} \text{ else } \text{Some } (\text{num-of-nat } (\text{take-bit } n (\text{numeral } m :: \text{nat})))) \rangle$

lemma *take-bit-num-simps:*

$\langle \text{take-bit-num } 0 \text{ } m = \text{None} \rangle$
 $\langle \text{take-bit-num } (\text{Suc } n) \text{ Num.One} =$
 $\text{Some Num.One} \rangle$
 $\langle \text{take-bit-num } (\text{Suc } n) (\text{Num.Bit0 } m) =$
 $(\text{case } \text{take-bit-num } n \text{ } m \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } q \Rightarrow \text{Some } (\text{Num.Bit0 } q)) \rangle$
 $\langle \text{take-bit-num } (\text{Suc } n) (\text{Num.Bit1 } m) =$
 $\text{Some } (\text{case } \text{take-bit-num } n \text{ } m \text{ of } \text{None} \Rightarrow \text{Num.One} \mid \text{Some } q \Rightarrow \text{Num.Bit1 } q) \rangle$
 $\langle \text{take-bit-num } (\text{numeral } r) \text{ Num.One} =$
 $\text{Some Num.One} \rangle$
 $\langle \text{take-bit-num } (\text{numeral } r) (\text{Num.Bit0 } m) =$
 $(\text{case } \text{take-bit-num } (\text{pred-numeral } r) \text{ } m \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } q \Rightarrow \text{Some } (\text{Num.Bit0 } q)) \rangle$
 $\langle \text{take-bit-num } (\text{numeral } r) (\text{Num.Bit1 } m) =$
 $\text{Some } (\text{case } \text{take-bit-num } (\text{pred-numeral } r) \text{ } m \text{ of } \text{None} \Rightarrow \text{Num.One} \mid \text{Some } q \Rightarrow$
 $\text{Num.Bit1 } q) \rangle$
by (*auto simp add: take-bit-num-def ac-simps mult-2 num-of-nat-double take-bit-Suc-bit0 take-bit-Suc-bit1 take-bit-numeral-bit0 take-bit-numeral-bit1*)

lemma *take-bit-num-code* [code]:

— Ocaml-style pattern matching is more robust wrt. different representations of *nat*

```

⟨take-bit-num n m = (case (n, m)
  of (0, -) ⇒ None
    | (Suc n, Num.One) ⇒ Some Num.One
    | (Suc n, Num.Bit0 m) ⇒ (case take-bit-num n m of None ⇒ None | Some q
⇒ Some (Num.Bit0 q))
    | (Suc n, Num.Bit1 m) ⇒ Some (case take-bit-num n m of None ⇒ Num.One
| Some q ⇒ Num.Bit1 q))⟩
  by (cases n; cases m) (simp-all add: take-bit-num-simps)

```

context *semiring-bit-operations*

begin

lemma *take-bit-num-eq-None-imp*:

⟨take-bit m (numeral n) = 0⟩ if ⟨take-bit-num m n = None⟩

proof —

from that **have** ⟨take-bit m (numeral n :: nat) = 0⟩

by (simp add: take-bit-num-def split: if-splits)

then have ⟨of-nat (take-bit m (numeral n)) = of-nat 0⟩

by simp

then show ?thesis

by (simp add: of-nat-take-bit)

qed

lemma *take-bit-num-eq-Some-imp*:

⟨take-bit m (numeral n) = numeral q⟩ if ⟨take-bit-num m n = Some q⟩

proof —

from that **have** ⟨take-bit m (numeral n :: nat) = numeral q⟩

by (auto simp add: take-bit-num-def Num.numeral-num-of-nat-unfold split: if-splits)

then have ⟨of-nat (take-bit m (numeral n)) = of-nat (numeral q)⟩

by simp

then show ?thesis

by (simp add: of-nat-take-bit)

qed

lemma *take-bit-numeral-numeral*:

⟨take-bit (numeral m) (numeral n) =

(case take-bit-num (numeral m) n of None ⇒ 0 | Some q ⇒ numeral q)⟩

by (auto split: option.split dest: take-bit-num-eq-None-imp take-bit-num-eq-Some-imp)

end

lemma *take-bit-numeral-minus-numeral-int*:

⟨take-bit (numeral m) (− numeral n :: int) =

(case take-bit-num (numeral m) n of None ⇒ 0 | Some q ⇒ take-bit (numeral

```

m) ( $2 \wedge \text{numeral } m - \text{numeral } q$ ) (is  $\langle ?lhs = ?rhs \rangle$ )
proof (cases  $\langle \text{take-bit-num } (\text{numeral } m) \ n \rangle$ )
  case None
  then show ?thesis
  by (auto dest: take-bit-num-eq-None-imp [where ?'a = int] simp add: take-bit-eq-0-iff)
next
  case (Some q)
  then have q:  $\langle \text{take-bit } (\text{numeral } m) \ (\text{numeral } n :: \text{int}) = \text{numeral } q \rangle$ 
  by (auto dest: take-bit-num-eq-Some-imp)
  let ?T =  $\langle \text{take-bit } (\text{numeral } m) :: \text{int} \Rightarrow \text{int} \rangle$ 
  have *:  $\langle ?T \ (2 \wedge \text{numeral } m) = ?T \ (?T \ 0) \rangle$ 
  by (simp add: take-bit-eq-0-iff)
  have  $\langle ?lhs = ?T \ (0 - \text{numeral } n) \rangle$ 
  by simp
  also have  $\langle \dots = ?T \ (?T \ (?T \ 0) - ?T \ (?T \ (\text{numeral } n))) \rangle$ 
  by (simp only: take-bit-diff)
  also have  $\langle \dots = ?T \ (2 \wedge \text{numeral } m - ?T \ (\text{numeral } n)) \rangle$ 
  by (simp only: take-bit-diff flip: *)
  also have  $\langle \dots = ?rhs \rangle$ 
  by (simp add: q Some)
  finally show ?thesis .
qed

declare take-bit-num-simps [simp]
  take-bit-numeral-numeral [simp]
  take-bit-numeral-minus-numeral-int [simp]

```

69.7 More properties

lemma take-bit-eq-mask-iff:

```

 $\langle \text{take-bit } n \ k = \text{mask } n \iff \text{take-bit } n \ (k + 1) = 0 \rangle$  (is  $\langle ?P \iff ?Q \rangle$ )
for k :: int
proof
  assume ?P
  then have  $\langle \text{take-bit } n \ (\text{take-bit } n \ k + \text{take-bit } n \ 1) = 0 \rangle$ 
  by (simp add: mask-eq-exp-minus-1 take-bit-eq-0-iff)
  then show ?Q
  by (simp only: take-bit-add)
next
  assume ?Q
  then have  $\langle \text{take-bit } n \ (k + 1) - 1 = -1 \rangle$ 
  by simp
  then have  $\langle \text{take-bit } n \ (\text{take-bit } n \ (k + 1) - 1) = \text{take-bit } n \ (-1) \rangle$ 
  by simp
  moreover have  $\langle \text{take-bit } n \ (\text{take-bit } n \ (k + 1) - 1) = \text{take-bit } n \ k \rangle$ 
  by (simp add: take-bit-eq-mod mod-simps)
  ultimately show ?P
  by simp
qed

```

lemma *take-bit-eq-mask-iff-exp-dvd*:
 $\langle \text{take-bit } n \ k = \text{mask } n \iff 2 \wedge^n \text{ dvd } k + 1 \rangle$
for $k :: \text{int}$
by (*simp add: take-bit-eq-mask-iff flip: take-bit-eq-0-iff*)

69.8 Bit concatenation

definition *concat-bit* :: $\langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle \text{concat-bit } n \ k \ l = \text{take-bit } n \ k \ \text{OR} \ \text{push-bit } n \ l \rangle$

lemma *bit-concat-bit-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{concat-bit } m \ k \ l) \ n \iff n < m \wedge \text{bit } k \ n \vee m \leq n \wedge \text{bit } l \ (n - m) \rangle$
by (*simp add: concat-bit-def bit-or-iff bit-and-iff bit-take-bit-iff bit-push-bit-iff ac-simps*)

lemma *concat-bit-eq*:
 $\langle \text{concat-bit } n \ k \ l = \text{take-bit } n \ k + \text{push-bit } n \ l \rangle$
by (*simp add: concat-bit-def take-bit-eq-mask bit-and-iff bit-mask-iff bit-push-bit-iff disjunctive-add*)

lemma *concat-bit-0* [*simp*]:
 $\langle \text{concat-bit } 0 \ k \ l = l \rangle$
by (*simp add: concat-bit-def*)

lemma *concat-bit-Suc*:
 $\langle \text{concat-bit } (\text{Suc } n) \ k \ l = k \bmod 2 + 2 * \text{concat-bit } n \ (k \text{ div } 2) \ l \rangle$
by (*simp add: concat-bit-eq take-bit-Suc push-bit-double*)

lemma *concat-bit-of-zero-1* [*simp*]:
 $\langle \text{concat-bit } n \ 0 \ l = \text{push-bit } n \ l \rangle$
by (*simp add: concat-bit-def*)

lemma *concat-bit-of-zero-2* [*simp*]:
 $\langle \text{concat-bit } n \ k \ 0 = \text{take-bit } n \ k \rangle$
by (*simp add: concat-bit-def take-bit-eq-mask*)

lemma *concat-bit-nonnegative-iff* [*simp*]:
 $\langle \text{concat-bit } n \ k \ l \geq 0 \iff l \geq 0 \rangle$
by (*simp add: concat-bit-def*)

lemma *concat-bit-negative-iff* [*simp*]:
 $\langle \text{concat-bit } n \ k \ l < 0 \iff l < 0 \rangle$
by (*simp add: concat-bit-def*)

lemma *concat-bit-assoc*:
 $\langle \text{concat-bit } n \ k \ (\text{concat-bit } m \ l \ r) = \text{concat-bit } (m + n) \ (\text{concat-bit } n \ k \ l) \ r \rangle$
by (*rule bit-eqI*) (*auto simp add: bit-concat-bit-iff ac-simps*)

lemma *concat-bit-assoc-sym*:

$\langle \text{concat-bit } m (\text{concat-bit } n k l) r = \text{concat-bit } (\text{min } m n) k (\text{concat-bit } (m - n) l r) \rangle$

by (*rule bit-eqI*) (*auto simp add: bit-concat-bit-iff ac-simps min-def*)

lemma *concat-bit-eq-iff*:

$\langle \text{concat-bit } n k l = \text{concat-bit } n r s$

$\longleftrightarrow \text{take-bit } n k = \text{take-bit } n r \wedge l = s \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)

proof

assume $?Q$

then show $?P$

by (*simp add: concat-bit-def*)

next

assume $?P$

then have $*$: $\langle \text{bit } (\text{concat-bit } n k l) m = \text{bit } (\text{concat-bit } n r s) m \rangle$ **for** m

by (*simp add: bit-eq-iff*)

have $\langle \text{take-bit } n k = \text{take-bit } n r \rangle$

proof (*rule bit-eqI*)

fix m

from $*$ [*of* m]

show $\langle \text{bit } (\text{take-bit } n k) m \longleftrightarrow \text{bit } (\text{take-bit } n r) m \rangle$

by (*auto simp add: bit-take-bit-iff bit-concat-bit-iff*)

qed

moreover have $\langle \text{push-bit } n l = \text{push-bit } n s \rangle$

proof (*rule bit-eqI*)

fix m

from $*$ [*of* m]

show $\langle \text{bit } (\text{push-bit } n l) m \longleftrightarrow \text{bit } (\text{push-bit } n s) m \rangle$

by (*auto simp add: bit-push-bit-iff bit-concat-bit-iff*)

qed

then have $\langle l = s \rangle$

by (*simp add: push-bit-eq-mult*)

ultimately show $?Q$

by (*simp add: concat-bit-def*)

qed

lemma *take-bit-concat-bit-eq*:

$\langle \text{take-bit } m (\text{concat-bit } n k l) = \text{concat-bit } (\text{min } m n) k (\text{take-bit } (m - n) l) \rangle$

by (*rule bit-eqI*)

(*auto simp add: bit-take-bit-iff bit-concat-bit-iff min-def*)

lemma *concat-bit-take-bit-eq*:

$\langle \text{concat-bit } n (\text{take-bit } n b) = \text{concat-bit } n b \rangle$

by (*simp add: concat-bit-def [abs-def]*)

69.9 Taking bits with sign propagation

context *ring-bit-operations*

begin

definition *signed-take-bit* :: $\langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle$

where $\langle \text{signed-take-bit } n \ a = \text{take-bit } n \ a \ \text{OR} \ (\text{of-bool } (\text{bit } a \ n) * \text{NOT } (\text{mask } n)) \rangle$

lemma *signed-take-bit-eq-if-positive*:

$\langle \text{signed-take-bit } n \ a = \text{take-bit } n \ a \rangle$ **if** $\langle \neg \text{bit } a \ n \rangle$

using that by (*simp add: signed-take-bit-def*)

lemma *signed-take-bit-eq-if-negative*:

$\langle \text{signed-take-bit } n \ a = \text{take-bit } n \ a \ \text{OR} \ \text{NOT } (\text{mask } n) \rangle$ **if** $\langle \text{bit } a \ n \rangle$

using that by (*simp add: signed-take-bit-def*)

lemma *even-signed-take-bit-iff*:

$\langle \text{even } (\text{signed-take-bit } m \ a) \longleftrightarrow \text{even } a \rangle$

by (*auto simp add: bit-0 signed-take-bit-def even-or-iff even-mask-iff bit-double-iff*)

lemma *bit-signed-take-bit-iff* [*bit-simps*]:

$\langle \text{bit } (\text{signed-take-bit } m \ a) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } a \ (\text{min } m \ n) \rangle$

by (*simp add: signed-take-bit-def bit-take-bit-iff bit-or-iff bit-not-iff bit-mask-iff min-def not-le*)

(*blast dest: bit-imp-possible-bit*)

lemma *signed-take-bit-0* [*simp*]:

$\langle \text{signed-take-bit } 0 \ a = - (a \ \text{mod } 2) \rangle$

by (*simp add: bit-0 signed-take-bit-def odd-iff-mod-2-eq-one*)

lemma *signed-take-bit-Suc*:

$\langle \text{signed-take-bit } (\text{Suc } n) \ a = a \ \text{mod } 2 + 2 * \text{signed-take-bit } n \ (a \ \text{div } 2) \rangle$

by (*simp add: bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 possible-bit-less-imp flip: bit-Suc min-Suc-Suc*)

lemma *signed-take-bit-of-0* [*simp*]:

$\langle \text{signed-take-bit } n \ 0 = 0 \rangle$

by (*simp add: signed-take-bit-def*)

lemma *signed-take-bit-of-minus-1* [*simp*]:

$\langle \text{signed-take-bit } n \ (-1) = -1 \rangle$

by (*simp add: signed-take-bit-def mask-eq-exp-minus-1 possible-bit-def*)

lemma *signed-take-bit-Suc-1* [*simp*]:

$\langle \text{signed-take-bit } (\text{Suc } n) \ 1 = 1 \rangle$

by (*simp add: signed-take-bit-Suc*)

lemma *signed-take-bit-numeral-of-1* [*simp*]:

$\langle \text{signed-take-bit } (\text{numeral } k) \ 1 = 1 \rangle$

by (*simp add: bit-1-iff signed-take-bit-eq-if-positive*)

lemma *signed-take-bit-rec*:

⟨signed-take-bit n $a = (\text{if } n = 0 \text{ then } - (a \text{ mod } 2) \text{ else } a \text{ mod } 2 + 2 * \text{signed-take-bit } (n - 1) (a \text{ div } 2))$ ⟩

by (cases n) (simp-all add: signed-take-bit-Suc)

lemma signed-take-bit-eq-iff-take-bit-eq:

⟨signed-take-bit n $a = \text{signed-take-bit } n$ $b \iff \text{take-bit } (\text{Suc } n) a = \text{take-bit } (\text{Suc } n) b$ ⟩

proof –

have ⟨bit (signed-take-bit n $a) = \text{bit } (\text{signed-take-bit } n$ $b) \iff \text{bit } (\text{take-bit } (\text{Suc } n) a) = \text{bit } (\text{take-bit } (\text{Suc } n) b)$ ⟩

by (simp add: fun-eq-iff bit-signed-take-bit-iff bit-take-bit-iff not-le less-Suc-eq-le min-def)

(use bit-imp-possible-bit in fastforce)

then show ?thesis

by (auto simp add: fun-eq-iff intro: bit-eqI)

qed

lemma signed-take-bit-signed-take-bit [simp]:

⟨signed-take-bit m (signed-take-bit n $a) = \text{signed-take-bit } (\text{min } m$ $n) a$ ⟩

by (auto simp add: bit-eq-iff bit-simps ac-simps)

lemma signed-take-bit-take-bit:

⟨signed-take-bit m (take-bit n $a) = (\text{if } n \leq m \text{ then take-bit } n \text{ else signed-take-bit } m) a$ ⟩

by (rule bit-eqI) (auto simp add: bit-signed-take-bit-iff min-def bit-take-bit-iff)

lemma take-bit-signed-take-bit:

⟨take-bit m (signed-take-bit n $a) = \text{take-bit } m a$ if $\langle m \leq \text{Suc } n \rangle$ ⟩

using that by (rule le-SucE; intro bit-eqI)

(auto simp add: bit-take-bit-iff bit-signed-take-bit-iff min-def less-Suc-eq)

end

Modulus centered around 0

lemma signed-take-bit-eq-concat-bit:

⟨signed-take-bit n $k = \text{concat-bit } n$ k ($- \text{of-bool } (\text{bit } k$ $n)$)⟩

by (simp add: concat-bit-def signed-take-bit-def)

lemma signed-take-bit-add:

⟨signed-take-bit n (signed-take-bit n $k + \text{signed-take-bit } n$ $l) = \text{signed-take-bit } n$ ($k + l$)⟩

for k l :: int

proof –

have ⟨take-bit (Suc n)

(take-bit (Suc n) (signed-take-bit n $k) +$

take-bit (Suc n) (signed-take-bit n $l)) =$

take-bit (Suc n) ($k + l$)⟩

by (simp add: take-bit-signed-take-bit take-bit-add)

then show ?thesis

by (*simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-add*)
qed

lemma *signed-take-bit-diff*:

$\langle \text{signed-take-bit } n \ (\text{signed-take-bit } n \ k - \text{signed-take-bit } n \ l) = \text{signed-take-bit } n \ (k - l) \rangle$

for $k \ l :: \text{int}$

proof –

have $\langle \text{take-bit } (\text{Suc } n)$

$(\text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ k) -$
 $\text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ l)) =$
 $\text{take-bit } (\text{Suc } n) \ (k - l) \rangle$

by (*simp add: take-bit-signed-take-bit take-bit-diff*)

then show *?thesis*

by (*simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-diff*)

qed

lemma *signed-take-bit-minus*:

$\langle \text{signed-take-bit } n \ (- \text{signed-take-bit } n \ k) = \text{signed-take-bit } n \ (- k) \rangle$

for $k :: \text{int}$

proof –

have $\langle \text{take-bit } (\text{Suc } n)$

$(- \text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ k)) =$
 $\text{take-bit } (\text{Suc } n) \ (- k) \rangle$

by (*simp add: take-bit-signed-take-bit take-bit-minus*)

then show *?thesis*

by (*simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-minus*)

qed

lemma *signed-take-bit-mult*:

$\langle \text{signed-take-bit } n \ (\text{signed-take-bit } n \ k * \text{signed-take-bit } n \ l) = \text{signed-take-bit } n \ (k * l) \rangle$

for $k \ l :: \text{int}$

proof –

have $\langle \text{take-bit } (\text{Suc } n)$

$(\text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ k) *$
 $\text{take-bit } (\text{Suc } n) \ (\text{signed-take-bit } n \ l)) =$
 $\text{take-bit } (\text{Suc } n) \ (k * l) \rangle$

by (*simp add: take-bit-signed-take-bit take-bit-mult*)

then show *?thesis*

by (*simp only: signed-take-bit-eq-iff-take-bit-eq take-bit-mult*)

qed

lemma *signed-take-bit-eq-take-bit-minus*:

$\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k - 2 \wedge \text{Suc } n * \text{of-bool } (\text{bit } k \ n) \rangle$

for $k :: \text{int}$

proof (*cases* $\langle \text{bit } k \ n \rangle$)

case *True*

have $\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k \text{ OR NOT } (\text{mask } (\text{Suc } n)) \rangle$

by (*rule bit-eqI*) (*auto simp add: bit-signed-take-bit-iff min-def bit-take-bit-iff bit-or-iff bit-not-iff bit-mask-iff less-Suc-eq True*)
then have $\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ k + \text{NOT } (\text{mask } (\text{Suc } n)) \rangle$
by (*simp add: disjunctive-add bit-take-bit-iff bit-not-iff bit-mask-iff*)
with True show *?thesis*
by (*simp flip: minus-exp-eq-not-mask*)
next
case False
show *?thesis*
by (*rule bit-eqI*) (*simp add: False bit-signed-take-bit-iff bit-take-bit-iff min-def less-Suc-eq*)
qed

lemma *signed-take-bit-eq-take-bit-shift*:

$\langle \text{signed-take-bit } n \ k = \text{take-bit } (\text{Suc } n) \ (k + 2^{\wedge} n) - 2^{\wedge} n \rangle$
for $k :: \text{int}$
proof –
have $\ast: \langle \text{take-bit } n \ k \ \text{OR } 2^{\wedge} n = \text{take-bit } n \ k + 2^{\wedge} n \rangle$
by (*simp add: disjunctive-add bit-exp-iff bit-take-bit-iff*)
have $\langle \text{take-bit } n \ k - 2^{\wedge} n = \text{take-bit } n \ k + \text{NOT } (\text{mask } n) \rangle$
by (*simp add: minus-exp-eq-not-mask*)
also have $\langle \dots = \text{take-bit } n \ k \ \text{OR } \text{NOT } (\text{mask } n) \rangle$
by (*rule disjunctive-add*)
(simp add: bit-exp-iff bit-take-bit-iff bit-not-iff bit-mask-iff)
finally have $\ast\ast: \langle \text{take-bit } n \ k - 2^{\wedge} n = \text{take-bit } n \ k \ \text{OR } \text{NOT } (\text{mask } n) \rangle .$
have $\langle \text{take-bit } (\text{Suc } n) \ (k + 2^{\wedge} n) = \text{take-bit } (\text{Suc } n) \ (\text{take-bit } (\text{Suc } n) \ k + \text{take-bit } (\text{Suc } n) \ (2^{\wedge} n)) \rangle$
by (*simp only: take-bit-add*)
also have $\langle \text{take-bit } (\text{Suc } n) \ k = 2^{\wedge} n \ast \text{of-bool } (\text{bit } k \ n) + \text{take-bit } n \ k \rangle$
by (*simp add: take-bit-Suc-from-most*)
finally have $\langle \text{take-bit } (\text{Suc } n) \ (k + 2^{\wedge} n) = \text{take-bit } (\text{Suc } n) \ (2^{\wedge} (n + \text{of-bool } (\text{bit } k \ n)) + \text{take-bit } n \ k) \rangle$
by (*simp add: ac-simps*)
also have $\langle 2^{\wedge} (n + \text{of-bool } (\text{bit } k \ n)) + \text{take-bit } n \ k = 2^{\wedge} (n + \text{of-bool } (\text{bit } k \ n)) \ \text{OR } \text{take-bit } n \ k \rangle$
by (*rule disjunctive-add*)
(auto simp add: disjunctive-add bit-take-bit-iff bit-double-iff bit-exp-iff)
finally show *?thesis*
using $\ast \ast\ast$ **by** (*simp add: signed-take-bit-def concat-bit-Suc min-def ac-simps*)
qed

lemma *signed-take-bit-nonnegative-iff* [*simp*]:

$\langle 0 \leq \text{signed-take-bit } n \ k \longleftrightarrow \neg \text{bit } k \ n \rangle$
for $k :: \text{int}$
by (*simp add: signed-take-bit-def not-less concat-bit-def*)

lemma *signed-take-bit-negative-iff* [*simp*]:

$\langle \text{signed-take-bit } n \ k < 0 \longleftrightarrow \text{bit } k \ n \rangle$
for $k :: \text{int}$

by (simp add: signed-take-bit-def not-less concat-bit-def)

lemma *signed-take-bit-int-greater-eq-minus-exp* [simp]:
 $\langle - (2 \wedge n) \leq \text{signed-take-bit } n \ k \rangle$
 for $k :: \text{int}$
 by (simp add: signed-take-bit-eq-take-bit-shift)

lemma *signed-take-bit-int-less-exp* [simp]:
 $\langle \text{signed-take-bit } n \ k < 2 \wedge n \rangle$
 for $k :: \text{int}$
 using *take-bit-int-less-exp* [of $\langle \text{Suc } n \rangle$]
 by (simp add: signed-take-bit-eq-take-bit-shift)

lemma *signed-take-bit-int-eq-self-iff*:
 $\langle \text{signed-take-bit } n \ k = k \longleftrightarrow - (2 \wedge n) \leq k \wedge k < 2 \wedge n \rangle$
 for $k :: \text{int}$
 by (auto simp add: signed-take-bit-eq-take-bit-shift take-bit-int-eq-self-iff algebra-simps)

lemma *signed-take-bit-int-eq-self*:
 $\langle \text{signed-take-bit } n \ k = k \ \text{if } \langle - (2 \wedge n) \leq k \rangle \ \langle k < 2 \wedge n \rangle \rangle$
 for $k :: \text{int}$
 using *that* by (simp add: signed-take-bit-int-eq-self-iff)

lemma *signed-take-bit-int-less-eq-self-iff*:
 $\langle \text{signed-take-bit } n \ k \leq k \longleftrightarrow - (2 \wedge n) \leq k \rangle$
 for $k :: \text{int}$
 by (simp add: signed-take-bit-eq-take-bit-shift take-bit-int-less-eq-self-iff algebra-simps) *linarith*

lemma *signed-take-bit-int-less-self-iff*:
 $\langle \text{signed-take-bit } n \ k < k \longleftrightarrow 2 \wedge n \leq k \rangle$
 for $k :: \text{int}$
 by (simp add: signed-take-bit-eq-take-bit-shift take-bit-int-less-self-iff algebra-simps)

lemma *signed-take-bit-int-greater-self-iff*:
 $\langle k < \text{signed-take-bit } n \ k \longleftrightarrow k < - (2 \wedge n) \rangle$
 for $k :: \text{int}$
 by (simp add: signed-take-bit-eq-take-bit-shift take-bit-int-greater-self-iff algebra-simps) *linarith*

lemma *signed-take-bit-int-greater-eq-self-iff*:
 $\langle k \leq \text{signed-take-bit } n \ k \longleftrightarrow k < 2 \wedge n \rangle$
 for $k :: \text{int}$
 by (simp add: signed-take-bit-eq-take-bit-shift take-bit-int-greater-eq-self-iff algebra-simps)

lemma *signed-take-bit-int-greater-eq*:
 $\langle k + 2 \wedge \text{Suc } n \leq \text{signed-take-bit } n \ k \ \text{if } \langle k < - (2 \wedge n) \rangle \rangle$

for $k :: \text{int}$
using *that take-bit-int-greater-eq* [of $\langle k + 2 \wedge n \rangle \langle \text{Suc } n \rangle$]
by (*simp add: signed-take-bit-eq-take-bit-shift*)

lemma *signed-take-bit-int-less-eq*:
 $\langle \text{signed-take-bit } n \ k \leq k - 2 \wedge \text{Suc } n \rangle$ **if** $\langle k \geq 2 \wedge n \rangle$
for $k :: \text{int}$
using *that take-bit-int-less-eq* [of $\langle \text{Suc } n \rangle \langle k + 2 \wedge n \rangle$]
by (*simp add: signed-take-bit-eq-take-bit-shift*)

lemma *signed-take-bit-Suc-bit0* [*simp*]:
 $\langle \text{signed-take-bit } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } n \ (\text{numeral } k) * (2 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-Suc*)

lemma *signed-take-bit-Suc-bit1* [*simp*]:
 $\langle \text{signed-take-bit } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } n \ (\text{numeral } k) * 2 + (1 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-Suc*)

lemma *signed-take-bit-Suc-minus-bit0* [*simp*]:
 $\langle \text{signed-take-bit } (\text{Suc } n) \ (- \text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } n \ (- \text{numeral } k) * (2 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-Suc*)

lemma *signed-take-bit-Suc-minus-bit1* [*simp*]:
 $\langle \text{signed-take-bit } (\text{Suc } n) \ (- \text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } n \ (- \text{numeral } k - 1) * 2 + (1 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-Suc*)

lemma *signed-take-bit-numeral-bit0* [*simp*]:
 $\langle \text{signed-take-bit } (\text{numeral } l) \ (\text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) \ (\text{numeral } k) * (2 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-rec*)

lemma *signed-take-bit-numeral-bit1* [*simp*]:
 $\langle \text{signed-take-bit } (\text{numeral } l) \ (\text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) \ (\text{numeral } k) * 2 + (1 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-rec*)

lemma *signed-take-bit-numeral-minus-bit0* [*simp*]:
 $\langle \text{signed-take-bit } (\text{numeral } l) \ (- \text{numeral } (\text{Num.Bit0 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) \ (- \text{numeral } k) * (2 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-rec*)

lemma *signed-take-bit-numeral-minus-bit1* [*simp*]:
 $\langle \text{signed-take-bit } (\text{numeral } l) \ (- \text{numeral } (\text{Num.Bit1 } k)) = \text{signed-take-bit } (\text{pred-numeral } l) \ (- \text{numeral } k - 1) * 2 + (1 :: \text{int}) \rangle$
by (*simp add: signed-take-bit-rec*)

lemma *signed-take-bit-code* [code]:
 ⟨signed-take-bit n a =
 (let l = take-bit (Suc n) a
 in if bit l n then l + push-bit (Suc n) (- 1) else l)⟩
proof –
have *: ⟨take-bit (Suc n) a + push-bit n (- 2) =
 take-bit (Suc n) a OR NOT (mask (Suc n))⟩
by (auto simp add: bit-take-bit-iff bit-push-bit-iff bit-not-iff bit-mask-iff disjunctive-add
 simp flip: push-bit-minus-one-eq-not-mask)
show ?thesis
by (rule bit-eqI)
 (auto simp add: Let-def * bit-signed-take-bit-iff bit-take-bit-iff min-def less-Suc-eq
 bit-not-iff
 bit-mask-iff bit-or-iff simp del: push-bit-minus-one-eq-not-mask)
qed

69.10 Horner sums

context *semiring-bit-operations*
begin

lemma *horner-sum-bit-eq-take-bit*:
 ⟨horner-sum of-bool 2 (map (bit a) [0.. n]) = take-bit n a⟩
proof (induction a arbitrary: n rule: bits-induct)
case (stable a)
moreover have ⟨bit a = (λ-. odd a)⟩
using stable by (simp add: stable-imp-bit-iff-odd fun-eq-iff)
moreover have ⟨{q. q < n} = {0.. n }⟩
by auto
ultimately show ?case
by (simp add: stable-imp-take-bit-eq horner-sum-eq-sum mask-eq-sum-exp)
next
case (rec a b)
show ?case
proof (cases n)
case 0
then show ?thesis
by simp
next
case (Suc m)
have ⟨map (bit (of-bool b + 2 * a)) [0.. $Suc\ m$] = b # map (bit (of-bool b +
 2 * a)) [Suc 0.. $Suc\ m$]⟩
by (simp only: upt-conv-Cons) (simp add: bit-0)
also have ⟨... = b # map (bit a) [0.. m]⟩
by (simp only: flip: map-Suc-upt) (simp add: bit-Suc rec.hyps)
finally show ?thesis
using Suc rec.IH [of m] by (simp add: take-bit-Suc rec.hyps)

```

      (simp-all add: ac-simps mod-2-eq-odd)
    qed
  qed

end

context unique-euclidean-semiring-with-bit-operations
begin

lemma bit-horner-sum-bit-iff [bit-simps]:
  ⟨bit (horner-sum of-bool 2 bs) n ↔ n < length bs ∧ bs ! n⟩
proof (induction bs arbitrary: n)
  case Nil
  then show ?case
    by simp
next
  case (Cons b bs)
  show ?case
  proof (cases n)
    case 0
    then show ?thesis
      by (simp add: bit-0)
  next
    case (Suc m)
    with bit-rec [of - n] Cons.prem Cons.IH [of m]
    show ?thesis by simp
  qed
qed

lemma take-bit-horner-sum-bit-eq:
  ⟨take-bit n (horner-sum of-bool 2 bs) = horner-sum of-bool 2 (take n bs)⟩
  by (auto simp add: bit-eq-iff bit-take-bit-iff bit-horner-sum-bit-iff)

end

lemma horner-sum-of-bool-2-less:
  ⟨(horner-sum of-bool 2 bs :: int) < 2 ^ length bs⟩
proof -
  have ⟨(∑ n = 0..<length bs. of-bool (bs ! n) * (2::int) ^ n) ≤ (∑ n = 0..<length
  bs. 2 ^ n)⟩
  by (rule sum-mono) simp
  also have ⟨... = 2 ^ length bs - 1⟩
  by (induction bs) simp-all
  finally show ?thesis
  by (simp add: horner-sum-eq-sum)
qed

```

69.11 Symbolic computations for code generation

lemma *bit-int-code* [code]:

```

⟨bit (0::int)          n      ⟷ False⟩
⟨bit (Int.Neg num.One)  n      ⟷ True⟩
⟨bit (Int.Pos num.One)  0      ⟷ True⟩
⟨bit (Int.Pos (num.Bit0 m)) 0    ⟷ False⟩
⟨bit (Int.Pos (num.Bit1 m)) 0    ⟷ True⟩
⟨bit (Int.Neg (num.Bit0 m)) 0    ⟷ False⟩
⟨bit (Int.Neg (num.Bit1 m)) 0    ⟷ True⟩
⟨bit (Int.Pos num.One)   (Suc n) ⟷ False⟩
⟨bit (Int.Pos (num.Bit0 m)) (Suc n) ⟷ bit (Int.Pos m) n⟩
⟨bit (Int.Pos (num.Bit1 m)) (Suc n) ⟷ bit (Int.Pos m) n⟩
⟨bit (Int.Neg (num.Bit0 m)) (Suc n) ⟷ bit (Int.Neg m) n⟩
⟨bit (Int.Neg (num.Bit1 m)) (Suc n) ⟷ bit (Int.Neg (Num.inc m)) n⟩
by (simp-all add: Num.add-One bit-0 bit-Suc)

```

lemma *not-int-code* [code]:

```

⟨NOT (0 :: int) = - 1⟩
⟨NOT (Int.Pos n) = Int.Neg (Num.inc n)⟩
⟨NOT (Int.Neg n) = Num.sub n num.One⟩
by (simp-all add: Num.add-One not-int-def)

```

lemma *and-int-code* [code]:

```

fixes i j :: int shows
⟨0 AND j = 0⟩
⟨i AND 0 = 0⟩
⟨Int.Pos n AND Int.Pos m = (case and-num n m of None ⇒ 0 | Some n' ⇒
Int.Pos n')⟩
⟨Int.Neg n AND Int.Neg m = NOT (Num.sub n num.One OR Num.sub m
num.One)⟩
⟨Int.Pos n AND Int.Neg num.One = Int.Pos n⟩
⟨Int.Pos n AND Int.Neg (num.Bit0 m) = Num.sub (or-not-num-neg (Num.BitM
m) n) num.One⟩
⟨Int.Pos n AND Int.Neg (num.Bit1 m) = Num.sub (or-not-num-neg (num.Bit0
m) n) num.One⟩
⟨Int.Neg num.One AND Int.Pos m = Int.Pos m⟩
⟨Int.Neg (num.Bit0 n) AND Int.Pos m = Num.sub (or-not-num-neg (Num.BitM
n) m) num.One⟩
⟨Int.Neg (num.Bit1 n) AND Int.Pos m = Num.sub (or-not-num-neg (num.Bit0
n) m) num.One⟩
apply (auto simp add: and-num-eq-None-iff [where ?'a = int] and-num-eq-Some-iff
[where ?'a = int]
split: option.split)
apply (simp-all only: sub-one-eq-not-neg numeral-or-not-num-eq minus-minus
and-not-numerals
bit.de-Morgan-disj bit.double-compl and-not-num-eq-None-iff and-not-num-eq-Some-iff
ac-simps)
done

```


lemma *or-int-code* [code]:
fixes $i\ j :: \text{int}$ **shows**
 $\langle 0 \text{ OR } j = j \rangle$
 $\langle i \text{ OR } 0 = i \rangle$
 $\langle \text{Int.Pos } n \text{ OR } \text{Int.Pos } m = \text{Int.Pos } (\text{or-num } n\ m) \rangle$
 $\langle \text{Int.Neg } n \text{ OR } \text{Int.Neg } m = \text{NOT } (\text{Num.sub } n\ \text{num.One AND } \text{Num.sub } m\ \text{num.One}) \rangle$
 $\langle \text{Int.Pos } n \text{ OR } \text{Int.Neg } \text{num.One} = \text{Int.Neg } \text{num.One} \rangle$
 $\langle \text{Int.Pos } n \text{ OR } \text{Int.Neg } (\text{num.Bit0 } m) = (\text{case and-not-num } (\text{Num.BitM } m)\ n \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
 $\langle \text{Int.Pos } n \text{ OR } \text{Int.Neg } (\text{num.Bit1 } m) = (\text{case and-not-num } (\text{num.Bit0 } m)\ n \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
 $\langle \text{Int.Neg } \text{num.One OR } \text{Int.Pos } m = \text{Int.Neg } \text{num.One} \rangle$
 $\langle \text{Int.Neg } (\text{num.Bit0 } n) \text{ OR } \text{Int.Pos } m = (\text{case and-not-num } (\text{Num.BitM } n)\ m \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
 $\langle \text{Int.Neg } (\text{num.Bit1 } n) \text{ OR } \text{Int.Pos } m = (\text{case and-not-num } (\text{num.Bit0 } n)\ m \text{ of } \text{None} \Rightarrow -1 \mid \text{Some } n' \Rightarrow \text{Int.Neg } (\text{Num.inc } n')) \rangle$
apply (*auto simp add: numeral-or-num-eq split: option.splits*)
apply (*simp-all only: and-not-num-eq-None-iff and-not-num-eq-Some-iff and-not-numerals*
numeral-or-not-num-eq or-int-def bit.double-compl ac-simps flip: numeral-eq-iff [where $?'a = \text{int}$])
apply *simp-all*
done

lemma *xor-int-code* [code]:
fixes $i\ j :: \text{int}$ **shows**
 $\langle 0 \text{ XOR } j = j \rangle$
 $\langle i \text{ XOR } 0 = i \rangle$
 $\langle \text{Int.Pos } n \text{ XOR } \text{Int.Pos } m = (\text{case xor-num } n\ m \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } n' \Rightarrow \text{Int.Pos } n') \rangle$
 $\langle \text{Int.Neg } n \text{ XOR } \text{Int.Neg } m = \text{Num.sub } n\ \text{num.One XOR } \text{Num.sub } m\ \text{num.One} \rangle$
 $\langle \text{Int.Neg } n \text{ XOR } \text{Int.Pos } m = \text{NOT } (\text{Num.sub } n\ \text{num.One XOR } \text{Int.Pos } m) \rangle$
 $\langle \text{Int.Pos } n \text{ XOR } \text{Int.Neg } m = \text{NOT } (\text{Int.Pos } n \text{ XOR } \text{Num.sub } m\ \text{num.One}) \rangle$
by (*simp-all add: xor-num-eq-None-iff* [where $?'a = \text{int}$] *xor-num-eq-Some-iff* [where $?'a = \text{int}$] *split: option.split*)

lemma *push-bit-int-code* [code]:
 $\langle \text{push-bit } 0\ i = i \rangle$
 $\langle \text{push-bit } (\text{Suc } n)\ i = \text{push-bit } n\ (\text{Int.dup } i) \rangle$
by (*simp-all add: ac-simps*)

lemma *drop-bit-int-code* [code]:
fixes $i :: \text{int}$ **shows**
 $\langle \text{drop-bit } 0\ i = i \rangle$
 $\langle \text{drop-bit } (\text{Suc } n)\ 0 = (0 :: \text{int}) \rangle$
 $\langle \text{drop-bit } (\text{Suc } n)\ (\text{Int.Pos } \text{num.One}) = 0 \rangle$
 $\langle \text{drop-bit } (\text{Suc } n)\ (\text{Int.Pos } (\text{num.Bit0 } m)) = \text{drop-bit } n\ (\text{Int.Pos } m) \rangle$
 $\langle \text{drop-bit } (\text{Suc } n)\ (\text{Int.Pos } (\text{num.Bit1 } m)) = \text{drop-bit } n\ (\text{Int.Pos } m) \rangle$

$\langle \text{drop-bit } (Suc\ n) (Int.Neg\ num.One) = -\ 1 \rangle$
 $\langle \text{drop-bit } (Suc\ n) (Int.Neg\ (num.Bit0\ m)) = \text{drop-bit } n (Int.Neg\ m) \rangle$
 $\langle \text{drop-bit } (Suc\ n) (Int.Neg\ (num.Bit1\ m)) = \text{drop-bit } n (Int.Neg\ (Num.inc\ m)) \rangle$
by (*simp-all add: drop-bit-Suc add-One*)

69.12 Key ideas of bit operations

When formalizing bit operations, it is tempting to represent bit values as explicit lists over a binary type. This however is a bad idea, mainly due to the inherent ambiguities in representation concerning repeating leading bits.

Hence this approach avoids such explicit lists altogether following an algebraic path:

- Bit values are represented by numeric types: idealized unbounded bit values can be represented by type *int*, bounded bit values by quotient types over *int*.
- (A special case are idealized unbounded bit values ending in 0 which can be represented by type *nat* but only support a restricted set of operations).
- From this idea follows that
 - multiplication by 2 is a bit shift to the left and
 - division by 2 is a bit shift to the right.
- Concerning bounded bit values, iterated shifts to the left may result in eliminating all bits by shifting them all beyond the boundary. The property $2^n \neq 0$ represents that *n* is *not* beyond that boundary.
- The projection on a single bit is then $\text{bit } a\ n = \text{odd } (a\ \text{div } 2^n)$.
- This leads to the most fundamental properties of bit values:
 - Equality rule: $(\bigwedge n. \text{possible-bit TYPE}(int)\ n \implies \text{bit } a\ n = \text{bit } b\ n) \implies a = b$
 - Induction rule: $\llbracket \bigwedge a. a\ \text{div } 2 = a \implies P\ a; \bigwedge a\ b. \llbracket P\ a; (\text{of-bool } b + 2 * a)\ \text{div } 2 = a \rrbracket \implies P\ (\text{of-bool } b + 2 * a) \rrbracket \implies P\ a$
- Typical operations are characterized as follows:
 - Singleton *n*th bit: 2^n
 - Bit mask upto bit *n*: $\text{mask } n = 2^n - 1$
 - Left shift: $\text{push-bit } n\ a = a * 2^n$

- Right shift: $\text{drop-bit } n \ a = a \ \text{div } 2^n$
- Truncation: $\text{take-bit } n \ a = a \ \text{mod } 2^n$
- Negation: $\text{bit } (\text{NOT } a) \ n = (\neg \neg \text{possible-bit } \text{TYPE}(\text{int}) \ n \wedge \neg \text{bit } a \ n)$
- And: $\text{bit } (a \ \text{AND } b) \ n = (\text{bit } a \ n \wedge \text{bit } b \ n)$
- Or: $\text{bit } (a \ \text{OR } b) \ n = (\text{bit } a \ n \vee \text{bit } b \ n)$
- Xor: $\text{bit } (a \ \text{XOR } b) \ n = (\text{bit } a \ n \neq \text{bit } b \ n)$
- Set a single bit: $\text{set-bit } n \ a = a \ \text{OR } \text{push-bit } n \ 1$
- Unset a single bit: $\text{unset-bit } n \ a = a \ \text{AND } \text{NOT } (\text{push-bit } n \ 1)$
- Flip a single bit: $\text{flip-bit } n \ a = a \ \text{XOR } \text{push-bit } n \ 1$
- Signed truncation, or modulus centered around 0: $\text{signed-take-bit } n \ a = \text{take-bit } n \ a \ \text{OR } \text{of-bool } (\text{bit } a \ n) * \text{NOT } (\text{mask } n)$
- Bit concatenation: $\text{concat-bit } n \ k \ l = \text{take-bit } n \ k \ \text{OR } \text{push-bit } n \ l$
- (Bounded) conversion from and to a list of bits: $\text{horner-sum of-bool } 2 \ (\text{map } (\text{bit } a) \ [0..\lt n]) = \text{take-bit } n \ a$

no-notation

```

not (⟨NOT⟩)
and and (infixr ⟨AND⟩ 64)
and or (infixr ⟨OR⟩ 59)
and xor (infixr ⟨XOR⟩ 59)

```

bundle *bit-operations-syntax***begin****notation**

```

not (⟨NOT⟩)
and and (infixr ⟨AND⟩ 64)
and or (infixr ⟨OR⟩ 59)
and xor (infixr ⟨XOR⟩ 59)

```

end**end**

70 Numeric types for code generation onto target language numerals only

```

theory Code-Numeral
imports Divides Lifting Bit-Operations
begin

```

70.1 Type of target language integers

```

typedef integer = UNIV :: int set
  morphisms int-of-integer integer-of-int ..

setup-lifting type-definition-integer

lemma integer-eq-iff:
   $k = l \longleftrightarrow \text{int-of-integer } k = \text{int-of-integer } l$ 
  by transfer rule

lemma integer-eqI:
   $\text{int-of-integer } k = \text{int-of-integer } l \implies k = l$ 
  using integer-eq-iff [of k l] by simp

lemma int-of-integer-integer-of-int [simp]:
   $\text{int-of-integer } (\text{integer-of-int } k) = k$ 
  by transfer rule

lemma integer-of-int-int-of-integer [simp]:
   $\text{integer-of-int } (\text{int-of-integer } k) = k$ 
  by transfer rule

instantiation integer :: ring-1
begin

lift-definition zero-integer :: integer
  is 0 :: int
  .

declare zero-integer.rep-eq [simp]

lift-definition one-integer :: integer
  is 1 :: int
  .

declare one-integer.rep-eq [simp]

lift-definition plus-integer :: integer  $\Rightarrow$  integer  $\Rightarrow$  integer
  is plus :: int  $\Rightarrow$  int  $\Rightarrow$  int
  .

declare plus-integer.rep-eq [simp]

lift-definition uminus-integer :: integer  $\Rightarrow$  integer
  is uminus :: int  $\Rightarrow$  int
  .

declare uminus-integer.rep-eq [simp]

```

```

lift-definition minus-integer :: integer  $\Rightarrow$  integer  $\Rightarrow$  integer
  is minus :: int  $\Rightarrow$  int  $\Rightarrow$  int
  .

declare minus-integer.rep-eq [simp]

lift-definition times-integer :: integer  $\Rightarrow$  integer  $\Rightarrow$  integer
  is times :: int  $\Rightarrow$  int  $\Rightarrow$  int
  .

declare times-integer.rep-eq [simp]

instance proof
qed (transfer, simp add: algebra-simps)+

end

instance integer :: Rings.dvd ..

context
  includes lifting-syntax
  notes transfer-rule-numeral [transfer-rule]
begin

lemma [transfer-rule]:
  (pcr-integer  $\implies$  pcr-integer  $\implies$  ( $\longleftrightarrow$ )) (dvd) (dvd)
  by (unfold dvd-def) transfer-prover

lemma [transfer-rule]:
  (( $\longleftrightarrow$ )  $\implies$  pcr-integer) of-bool of-bool
  by (unfold of-bool-def) transfer-prover

lemma [transfer-rule]:
  ((=)  $\implies$  pcr-integer) int of-nat
  by (rule transfer-rule-of-nat) transfer-prover+

lemma [transfer-rule]:
  ((=)  $\implies$  pcr-integer) ( $\lambda k. k$ ) of-int
proof –
  have ((=)  $\implies$  pcr-integer) of-int of-int
  by (rule transfer-rule-of-int) transfer-prover+
  then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:
  ((=)  $\implies$  pcr-integer) numeral numeral
  by transfer-prover

lemma [transfer-rule]:

```

((=) ==> (=) ==> pcr-integer) Num.sub Num.sub
 by (unfold Num.sub-def) transfer-prover

lemma [transfer-rule]:
 (pcr-integer ==> (=) ==> pcr-integer) (\wedge) (\wedge)
 by (unfold power-def) transfer-prover

end

lemma int-of-integer-of-nat [simp]:
 int-of-integer (of-nat n) = of-nat n
 by transfer rule

lift-definition integer-of-nat :: nat \Rightarrow integer
 is of-nat :: nat \Rightarrow int
 .

lemma integer-of-nat-eq-of-nat [code]:
 integer-of-nat = of-nat
 by transfer rule

lemma int-of-integer-integer-of-nat [simp]:
 int-of-integer (integer-of-nat n) = of-nat n
 by transfer rule

lift-definition nat-of-integer :: integer \Rightarrow nat
 is Int.nat
 .

lemma nat-of-integer-of-nat [simp]:
 nat-of-integer (of-nat n) = n
 by transfer simp

lemma int-of-integer-of-int [simp]:
 int-of-integer (of-int k) = k
 by transfer simp

lemma nat-of-integer-integer-of-nat [simp]:
 nat-of-integer (integer-of-nat n) = n
 by transfer simp

lemma integer-of-int-eq-of-int [simp, code-abbrev]:
 integer-of-int = of-int
 by transfer (simp add: fun-eq-iff)

lemma of-int-integer-of [simp]:
 of-int (int-of-integer k) = (k :: integer)
 by transfer rule

```

lemma int-of-integer-numeral [simp]:
  int-of-integer (numeral k) = numeral k
  by transfer rule

lemma int-of-integer-sub [simp]:
  int-of-integer (Num.sub k l) = Num.sub k l
  by transfer rule

definition integer-of-num :: num ⇒ integer
  where [simp]: integer-of-num = numeral

lemma integer-of-num [code]:
  integer-of-num Num.One = 1
  integer-of-num (Num.Bit0 n) = (let k = integer-of-num n in k + k)
  integer-of-num (Num.Bit1 n) = (let k = integer-of-num n in k + k + 1)
  by (simp-all only: integer-of-num-def numeral.simps Let-def)

lemma integer-of-num-triv:
  integer-of-num Num.One = 1
  integer-of-num (Num.Bit0 Num.One) = 2
  by simp-all

instantiation integer :: {linordered-idom, equal}
begin

lift-definition abs-integer :: integer ⇒ integer
  is abs :: int ⇒ int
  .

declare abs-integer.rep-eq [simp]

lift-definition sgn-integer :: integer ⇒ integer
  is sgn :: int ⇒ int
  .

declare sgn-integer.rep-eq [simp]

lift-definition less-eq-integer :: integer ⇒ integer ⇒ bool
  is less-eq :: int ⇒ int ⇒ bool
  .

lemma integer-less-eq-iff:
   $k \leq l \iff \text{int-of-integer } k \leq \text{int-of-integer } l$ 
  by (fact less-eq-integer.rep-eq)

lift-definition less-integer :: integer ⇒ integer ⇒ bool
  is less :: int ⇒ int ⇒ bool
  .

```

lemma *integer-less-iff*:

$k < l \iff \text{int-of-integer } k < \text{int-of-integer } l$

by (*fact less-integer.rep-eq*)

lift-definition *equal-integer* :: *integer* \Rightarrow *integer* \Rightarrow *bool*

is *HOL.equal* :: *int* \Rightarrow *int* \Rightarrow *bool*

.

instance

by *standard* (*transfer*, *simp add: algebra-simps equal less-le-not-le [symmetric]*
mult-strict-right-mono linear)+

end

context

includes *lifting-syntax*

begin

lemma [*transfer-rule*]:

$\langle \text{pcr-integer} \text{ ===> pcr-integer} \text{ ===> pcr-integer} \rangle \text{ min min}$

by (*unfold min-def*) *transfer-prover*

lemma [*transfer-rule*]:

$\langle \text{pcr-integer} \text{ ===> pcr-integer} \text{ ===> pcr-integer} \rangle \text{ max max}$

by (*unfold max-def*) *transfer-prover*

end

lemma *int-of-integer-min* [*simp*]:

$\text{int-of-integer } (\text{min } k \ l) = \text{min } (\text{int-of-integer } k) \ (\text{int-of-integer } l)$

by *transfer rule*

lemma *int-of-integer-max* [*simp*]:

$\text{int-of-integer } (\text{max } k \ l) = \text{max } (\text{int-of-integer } k) \ (\text{int-of-integer } l)$

by *transfer rule*

lemma *nat-of-integer-non-positive* [*simp*]:

$k \leq 0 \implies \text{nat-of-integer } k = 0$

by *transfer simp*

lemma *of-nat-of-integer* [*simp*]:

$\text{of-nat } (\text{nat-of-integer } k) = \text{max } 0 \ k$

by *transfer auto*

instantiation *integer* :: *unique-euclidean-ring*

begin

lift-definition *divide-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer*

is *divide* :: *int* \Rightarrow *int* \Rightarrow *int*


```

.

declare divide-integer.rep-eq [simp]

lift-definition modulo-integer :: integer  $\Rightarrow$  integer  $\Rightarrow$  integer
  is modulo :: int  $\Rightarrow$  int  $\Rightarrow$  int
.

declare modulo-integer.rep-eq [simp]

lift-definition euclidean-size-integer :: integer  $\Rightarrow$  nat
  is euclidean-size :: int  $\Rightarrow$  nat
.

declare euclidean-size-integer.rep-eq [simp]

lift-definition division-segment-integer :: integer  $\Rightarrow$  integer
  is division-segment :: int  $\Rightarrow$  int
.

declare division-segment-integer.rep-eq [simp]

instance
  apply (standard; transfer)
  apply (use mult-le-mono2 [of 1] in  $\langle$ auto simp add: sgn-mult-abs abs-mult sgn-mult
abs-mod-less sgn-mod nat-mult-distrib
  division-segment-mult division-segment-mod $\rangle$ )
  apply (simp add: division-segment-int-def split: if-splits)
  done

end

lemma [code]:
  euclidean-size = nat-of-integer  $\circ$  abs
  by (simp add: fun-eq-iff nat-of-integer.rep-eq)

lemma [code]:
  division-segment (k :: integer) = (if k  $\geq$  0 then 1 else - 1)
  by transfer (simp add: division-segment-int-def)

instance integer :: unique-euclidean-ring-with-nat
  by (standard; transfer) (simp-all add: of-nat-div division-segment-int-def)

instantiation integer :: ring-bit-operations
begin

lift-definition bit-integer ::  $\langle$ integer  $\Rightarrow$  nat  $\Rightarrow$  bool $\rangle$ 
  is bit .

```

```

lift-definition not-integer :: ⟨integer ⇒ integer⟩
  is not .

lift-definition and-integer :: ⟨integer ⇒ integer ⇒ integer⟩
  is and .

lift-definition or-integer :: ⟨integer ⇒ integer ⇒ integer⟩
  is or .

lift-definition xor-integer :: ⟨integer ⇒ integer ⇒ integer⟩
  is xor .

lift-definition mask-integer :: ⟨nat ⇒ integer⟩
  is mask .

lift-definition set-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is set-bit .

lift-definition unset-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is unset-bit .

lift-definition flip-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is flip-bit .

lift-definition push-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is push-bit .

lift-definition drop-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is drop-bit .

lift-definition take-bit-integer :: ⟨nat ⇒ integer ⇒ integer⟩
  is take-bit .

instance by (standard; transfer)
  (fact bit-eq-rec bits-induct bit-iff-odd push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod
   bits-div-0 bits-div-by-1 bits-mod-div-trivial even-succ-div-2
   exp-div-exp-eq div-exp-eq mod-exp-eq mult-exp-mod-exp-eq div-exp-mod-exp-eq
   even-mask-div-iff even-mult-exp-div-exp-iff
   bit-and-iff bit-or-iff bit-xor-iff mask-eq-exp-minus-1
   set-bit-def bit-unset-bit-iff flip-bit-def bit-not-iff-eq minus-eq-not-minus-1) +

end

instance integer :: unique-euclidean-semiring-with-bit-operations ..

context
  includes bit-operations-syntax
begin

```

lemma [code]:

⟨bit $k\ n \longleftrightarrow \text{odd } (\text{drop-bit } n\ k)\rangle$
 ⟨NOT $k = -k - 1\rangle$
 ⟨mask $n = 2^{\wedge} n - (1 :: \text{integer})\rangle$
 ⟨set-bit $n\ k = k \text{ OR push-bit } n\ 1\rangle$
 ⟨unset-bit $n\ k = k \text{ AND NOT } (\text{push-bit } n\ 1)\rangle$
 ⟨flip-bit $n\ k = k \text{ XOR push-bit } n\ 1\rangle$
 ⟨push-bit $n\ k = k * 2^{\wedge} n\rangle$
 ⟨drop-bit $n\ k = k \text{ div } 2^{\wedge} n\rangle$
 ⟨take-bit $n\ k = k \text{ mod } 2^{\wedge} n\rangle$ **for** $k :: \text{integer}$
by (fact bit-iff-odd-drop-bit not-eq-complement mask-eq-exp-minus-1
 set-bit-eq-or unset-bit-eq-and-not flip-bit-eq-xor push-bit-eq-mult drop-bit-eq-div
 take-bit-eq-mod)+

lemma [code]:

⟨ $k \text{ AND } l = (\text{if } k = 0 \vee l = 0 \text{ then } 0 \text{ else if } k = -1 \text{ then } l \text{ else if } l = -1 \text{ then } k$
 $\text{ else } (k \text{ mod } 2) * (l \text{ mod } 2) + 2 * ((k \text{ div } 2) \text{ AND } (l \text{ div } 2)))\rangle$ **for** $k\ l :: \text{integer}$
by transfer (fact and-int-unfold)

lemma [code]:

⟨ $k \text{ OR } l = (\text{if } k = -1 \vee l = -1 \text{ then } -1 \text{ else if } k = 0 \text{ then } l \text{ else if } l = 0 \text{ then } k$
 $\text{ else } \max (k \text{ mod } 2) (l \text{ mod } 2) + 2 * ((k \text{ div } 2) \text{ OR } (l \text{ div } 2)))\rangle$ **for** $k\ l :: \text{integer}$
by transfer (fact or-int-unfold)

lemma [code]:

⟨ $k \text{ XOR } l = (\text{if } k = -1 \text{ then } \text{NOT } l \text{ else if } l = -1 \text{ then } \text{NOT } k \text{ else if } k = 0$
 $\text{ then } l \text{ else if } l = 0 \text{ then } k$
 $\text{ else } |k \text{ mod } 2 - l \text{ mod } 2| + 2 * ((k \text{ div } 2) \text{ XOR } (l \text{ div } 2)))\rangle$ **for** $k\ l :: \text{integer}$
by transfer (fact xor-int-unfold)

end

instantiation $\text{integer} :: \text{unique-euclidean-semiring-with-nat-division}$

begin

definition $\text{divmod-integer} :: \text{num} \Rightarrow \text{num} \Rightarrow \text{integer} \times \text{integer}$

where

$\text{divmod-integer}'\text{-def}: \text{divmod-integer } m\ n = (\text{numeral } m \text{ div numeral } n, \text{ numeral } m \text{ mod numeral } n)$

definition $\text{divmod-step-integer} :: \text{integer} \Rightarrow \text{integer} \times \text{integer} \Rightarrow \text{integer} \times \text{integer}$

where

$\text{divmod-step-integer } l\ qr = (\text{let } (q, r) = qr$
 $\text{ in if } |l| \leq |r| \text{ then } (2 * q + 1, r - l)$
 $\text{ else } (2 * q, r))$

instance **by** standard

(auto simp add: divmod-integer'-def divmod-step-integer-def integer-less-eq-iff)

end

declare *divmod-algorithm-code* [**where** $?'a = \text{integer}$,
folded integer-of-num-def, *unfolded integer-of-num-triv*,
code]

lemma *integer-of-nat-0*: *integer-of-nat 0 = 0*
by *transfer simp*

lemma *integer-of-nat-1*: *integer-of-nat 1 = 1*
by *transfer simp*

lemma *integer-of-nat-numeral*:
integer-of-nat (numeral n) = numeral n
by *transfer simp*

70.2 Code theorems for target language integers

Constructors

definition *Pos* :: *num* \Rightarrow *integer*
where
 [*simp*, *code-post*]: *Pos = numeral*

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
 $\langle ((=) == => \text{pcr-integer}) \text{ numeral } Pos \rangle$
by *simp transfer-prover*

end

lemma *Pos-fold* [*code-unfold*]:
numeral Num.One = Pos Num.One
numeral (Num.Bit0 k) = Pos (Num.Bit0 k)
numeral (Num.Bit1 k) = Pos (Num.Bit1 k)
by *simp-all*

definition *Neg* :: *num* \Rightarrow *integer*
where
 [*simp*, *code-abbrev*]: *Neg n = - Pos n*

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:

⟨((=) == => pcr-integer) (λn. - numeral n) Neg⟩
 by (unfold Neg-def) transfer-prover

end

code-datatype 0::integer Pos Neg

A further pair of constructors for generated computations

context

begin

qualified definition positive :: num ⇒ integer
 where [simp]: positive = numeral

qualified definition negative :: num ⇒ integer
 where [simp]: negative = uminus ∘ numeral

lemma [code-computation-unfold]:
 numeral = positive
 Pos = positive
 Neg = negative
 by (simp-all add: fun-eq-iff)

end

Auxiliary operations

lift-definition dup :: integer ⇒ integer
 is λk::int. k + k
 .

lemma dup-code [code]:
 dup 0 = 0
 dup (Pos n) = Pos (Num.Bit0 n)
 dup (Neg n) = Neg (Num.Bit0 n)
 by (transfer, simp only: numeral-Bit0 minus-add-distrib)+

lift-definition sub :: num ⇒ num ⇒ integer
 is λm n. numeral m - numeral n :: int
 .

lemma sub-code [code]:
 sub Num.One Num.One = 0
 sub (Num.Bit0 m) Num.One = Pos (Num.BitM m)
 sub (Num.Bit1 m) Num.One = Pos (Num.Bit0 m)
 sub Num.One (Num.Bit0 n) = Neg (Num.BitM n)
 sub Num.One (Num.Bit1 n) = Neg (Num.Bit0 n)
 sub (Num.Bit0 m) (Num.Bit0 n) = dup (sub m n)
 sub (Num.Bit1 m) (Num.Bit1 n) = dup (sub m n)
 sub (Num.Bit1 m) (Num.Bit0 n) = dup (sub m n) + 1

$sub (Num.Bit0 m) (Num.Bit1 n) = dup (sub m n) - 1$
by (transfer, simp add: dbl-def dbl-inc-def dbl-dec-def)+

Implementations

lemma *one-integer-code* [code, code-unfold]:

$1 = Pos\ Num.One$
by simp

lemma *plus-integer-code* [code]:

$k + 0 = (k::integer)$
 $0 + l = (l::integer)$
 $Pos\ m + Pos\ n = Pos\ (m + n)$
 $Pos\ m + Neg\ n = sub\ m\ n$
 $Neg\ m + Pos\ n = sub\ n\ m$
 $Neg\ m + Neg\ n = Neg\ (m + n)$
by (transfer, simp)+

lemma *uminus-integer-code* [code]:

$uminus\ 0 = (0::integer)$
 $uminus\ (Pos\ m) = Neg\ m$
 $uminus\ (Neg\ m) = Pos\ m$
by simp-all

lemma *minus-integer-code* [code]:

$k - 0 = (k::integer)$
 $0 - l = uminus\ (l::integer)$
 $Pos\ m - Pos\ n = sub\ m\ n$
 $Pos\ m - Neg\ n = Pos\ (m + n)$
 $Neg\ m - Pos\ n = Neg\ (m + n)$
 $Neg\ m - Neg\ n = sub\ n\ m$
by (transfer, simp)+

lemma *abs-integer-code* [code]:

$|k| = (if\ (k::integer) < 0\ then\ -\ k\ else\ k)$
by simp

lemma *sgn-integer-code* [code]:

$sgn\ k = (if\ k = 0\ then\ 0\ else\ if\ (k::integer) < 0\ then\ -\ 1\ else\ 1)$
by simp

lemma *times-integer-code* [code]:

$k * 0 = (0::integer)$
 $0 * l = (0::integer)$
 $Pos\ m * Pos\ n = Pos\ (m * n)$
 $Pos\ m * Neg\ n = Neg\ (m * n)$
 $Neg\ m * Pos\ n = Neg\ (m * n)$
 $Neg\ m * Neg\ n = Pos\ (m * n)$
by simp-all

definition *divmod-integer* :: integer \Rightarrow integer \Rightarrow integer \times integer
where

$$\text{divmod-integer } k \ l = (k \ \text{div} \ l, k \ \text{mod} \ l)$$

lemma *fst-divmod-integer* [*simp*]:
fst (divmod-integer *k l*) = *k div l*
by (*simp add: divmod-integer-def*)

lemma *snd-divmod-integer* [*simp*]:
snd (divmod-integer *k l*) = *k mod l*
by (*simp add: divmod-integer-def*)

definition *divmod-abs* :: integer \Rightarrow integer \Rightarrow integer \times integer
where

$$\text{divmod-abs } k \ l = (|k| \ \text{div} \ |l|, |k| \ \text{mod} \ |l|)$$

lemma *fst-divmod-abs* [*simp*]:
fst (divmod-abs *k l*) = *|k| div |l|*
by (*simp add: divmod-abs-def*)

lemma *snd-divmod-abs* [*simp*]:
snd (divmod-abs *k l*) = *|k| mod |l|*
by (*simp add: divmod-abs-def*)

lemma *divmod-abs-code* [*code*]:
divmod-abs (*Pos k*) (*Pos l*) = *divmod k l*
divmod-abs (*Neg k*) (*Neg l*) = *divmod k l*
divmod-abs (*Neg k*) (*Pos l*) = *divmod k l*
divmod-abs (*Pos k*) (*Neg l*) = *divmod k l*
divmod-abs *j 0* = (*0, |j|*)
divmod-abs *0 j* = (*0, 0*)
by (*simp-all add: prod-eq-iff*)

lemma *divmod-integer-eq-cases*:
divmod-integer k l =
 (*if k = 0 then (0, 0) else if l = 0 then (0, k) else*
 (*apsnd o times o sgn*) *l* (*if sgn k = sgn l*
then divmod-abs k l
else (let (r, s) = divmod-abs k l in
if s = 0 then (- r, 0) else (- r - 1, |l| - s))))

proof –

have *: *sgn k = sgn l* \longleftrightarrow *k = 0 \wedge l = 0 \vee 0 < l \wedge 0 < k \vee l < 0 \wedge k < 0*
for *k l :: int*

by (*auto simp add: sgn-if*)

have **: $-k = l * q \longleftrightarrow k = -(l * q)$ **for** *k l q :: int*

by *auto*

show ?thesis

by (*simp add: divmod-integer-def divmod-abs-def*)

(*transfer, auto simp add: * ** not-less zdiv-zminus1-eq-if zmod-zminus1-eq-if*)

div-minus-right mod-minus-right
qed

lemma *divmod-integer-code* [code]:

divmod-integer k l =
 (if $k = 0$ then $(0, 0)$
 else if $l > 0$ then
 (if $k > 0$ then *Code-Numeral.divmod-abs* k l
 else case *Code-Numeral.divmod-abs* k l of $(r, s) \Rightarrow$
 if $s = 0$ then $(-r, 0)$ else $(-r - 1, l - s)$)
 else if $l = 0$ then $(0, k)$
 else *apsnd uminus*
 (if $k < 0$ then *Code-Numeral.divmod-abs* k l
 else case *Code-Numeral.divmod-abs* k l of $(r, s) \Rightarrow$
 if $s = 0$ then $(-r, 0)$ else $(-r - 1, -l - s))$)
by (*cases* l 0 :: *integer rule: linorder-cases*)
 (*auto split: prod.splits simp add: divmod-integer-eq-cases*)

lemma *div-integer-code* [code]:

k *div* l = *fst* (*divmod-integer* k l)
by *simp*

lemma *mod-integer-code* [code]:

k *mod* l = *snd* (*divmod-integer* k l)
by *simp*

definition *bit-cut-integer* :: *integer* \Rightarrow *integer* \times *bool*
where *bit-cut-integer* k = $(k$ *div* 2 , *odd* $k)$

lemma *bit-cut-integer-code* [code]:

bit-cut-integer k = (if $k = 0$ then $(0, \text{False})$
 else let $(r, s) =$ *Code-Numeral.divmod-abs* k 2
 in (if $k > 0$ then r else $-r - s$, $s = 1$))

proof –

have *bit-cut-integer* k = (let $(r, s) =$ *divmod-integer* k 2 in $(r, s = 1)$)
by (*simp add: divmod-integer-def bit-cut-integer-def odd-iff-mod-2-eq-one*)
then show *?thesis*
by (*simp add: divmod-integer-code*) (*auto simp add: split-def*)

qed

lemma *equal-integer-code* [code]:

HOL.equal 0 $(0::\text{integer}) \longleftrightarrow \text{True}$
HOL.equal 0 $(\text{Pos } l) \longleftrightarrow \text{False}$
HOL.equal 0 $(\text{Neg } l) \longleftrightarrow \text{False}$
HOL.equal $(\text{Pos } k)$ $0 \longleftrightarrow \text{False}$
HOL.equal $(\text{Pos } k)$ $(\text{Pos } l) \longleftrightarrow \text{HOL.equal } k$ l
HOL.equal $(\text{Pos } k)$ $(\text{Neg } l) \longleftrightarrow \text{False}$
HOL.equal $(\text{Neg } k)$ $0 \longleftrightarrow \text{False}$
HOL.equal $(\text{Neg } k)$ $(\text{Pos } l) \longleftrightarrow \text{False}$

$HOL.equal (Neg\ k)\ (Neg\ l) \longleftrightarrow HOL.equal\ k\ l$
by (*simp-all add: equal*)

lemma *equal-integer-refl* [*code nbe*]:
 $HOL.equal\ (k::integer)\ k \longleftrightarrow True$
by (*fact equal-refl*)

lemma *less-eq-integer-code* [*code*]:
 $0 \leq (0::integer) \longleftrightarrow True$
 $0 \leq Pos\ l \longleftrightarrow True$
 $0 \leq Neg\ l \longleftrightarrow False$
 $Pos\ k \leq 0 \longleftrightarrow False$
 $Pos\ k \leq Pos\ l \longleftrightarrow k \leq l$
 $Pos\ k \leq Neg\ l \longleftrightarrow False$
 $Neg\ k \leq 0 \longleftrightarrow True$
 $Neg\ k \leq Pos\ l \longleftrightarrow True$
 $Neg\ k \leq Neg\ l \longleftrightarrow l \leq k$
by *simp-all*

lemma *less-integer-code* [*code*]:
 $0 < (0::integer) \longleftrightarrow False$
 $0 < Pos\ l \longleftrightarrow True$
 $0 < Neg\ l \longleftrightarrow False$
 $Pos\ k < 0 \longleftrightarrow False$
 $Pos\ k < Pos\ l \longleftrightarrow k < l$
 $Pos\ k < Neg\ l \longleftrightarrow False$
 $Neg\ k < 0 \longleftrightarrow True$
 $Neg\ k < Pos\ l \longleftrightarrow True$
 $Neg\ k < Neg\ l \longleftrightarrow l < k$
by *simp-all*

lift-definition *num-of-integer* :: *integer* \Rightarrow *num*
is *num-of-nat* \circ *nat*

.

lemma *num-of-integer-code* [*code*]:
 $num-of-integer\ k = (if\ k \leq 1\ then\ Num.One$
else let
 $(l, j) = divmod-integer\ k\ 2;$
 $l' = num-of-integer\ l;$
 $l'' = l' + l'$
in if $j = 0\ then\ l''\ else\ l'' + Num.One)$

proof –

{
assume *int-of-integer* $k\ mod\ 2 = 1$
then have $nat\ (int-of-integer\ k\ mod\ 2) = nat\ 1$ **by** *simp*
moreover assume *: $1 < int-of-integer\ k$
ultimately have **: $nat\ (int-of-integer\ k)\ mod\ 2 = 1$ **by** (*simp add: nat-mod-distrib*)
have $num-of-nat\ (nat\ (int-of-integer\ k)) =$

```

    num-of-nat (2 * (nat (int-of-integer k) div 2) + nat (int-of-integer k) mod 2)
  by simp
  then have num-of-nat (nat (int-of-integer k)) =
    num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + nat
(int-of-integer k) mod 2)
  by (simp add: mult-2)
  with ** have num-of-nat (nat (int-of-integer k)) =
    num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + 1)
  by simp
}
note aux = this
show ?thesis
  by (auto simp add: num-of-integer-def nat-of-integer-def Let-def case-prod-beta
not-le integer-eq-iff less-eq-integer-def
nat-mult-distrib nat-div-distrib num-of-nat-One num-of-nat-plus-distrib
mult-2 [where 'a=nat] aux add-One)
qed

```

```

lemma nat-of-integer-code [code]:
  nat-of-integer k = (if k ≤ 0 then 0
  else let
    (l, j) = divmod-integer k 2;
    l' = nat-of-integer l;
    l'' = l' + l'
  in if j = 0 then l'' else l'' + 1)
proof -
  obtain j where k = integer-of-int j
  proof
    show k = integer-of-int (int-of-integer k) by simp
  qed
  have *: nat j mod 2 = nat-of-integer (of-int j mod 2) if j ≥ 0
  using that by transfer (simp add: nat-mod-distrib)
  from k show ?thesis
  by (auto simp add: split-def Let-def nat-of-integer-def nat-div-distrib mult-2
[symmetric]
minus-mod-eq-mult-div [symmetric] *)
qed

```

```

lemma int-of-integer-code [code]:
  int-of-integer k = (if k < 0 then - (int-of-integer (- k))
  else if k = 0 then 0
  else let
    (l, j) = divmod-integer k 2;
    l' = 2 * int-of-integer l
  in if j = 0 then l' else l' + 1)
  by (auto simp add: split-def Let-def integer-eq-iff minus-mod-eq-mult-div [symmetric])

```

```

lemma integer-of-int-code [code]:
  integer-of-int k = (if k < 0 then - (integer-of-int (- k))

```

```

    else if k = 0 then 0
    else let
      l = 2 * integer-of-int (k div 2);
      j = k mod 2
    in if j = 0 then l else l + 1)
  by (auto simp add: split-def Let-def integer-eq-iff minus-mod-eq-mult-div [symmetric])

```

hide-const (open) Pos Neg sub dup divmod-abs

70.3 Serializer setup for target language integers

code-reserved Eval int Integer abs

code-printing

```

type-constructor integer →
  (SML) IntInf.int
  and (OCaml) Z.t
  and (Haskell) Integer
  and (Scala) BigInt
  and (Eval) int
| class-instance integer :: equal →
  (Haskell) –

```

code-printing

```

constant 0::integer →
  (SML) !(0 / :/ IntInf.int)
  and (OCaml) Z.zero
  and (Haskell) !(0 / ::/ Integer)
  and (Scala) BigInt(0)

```

setup <

```

  fold (fn target =>
    Numeral.add-code const-name <Code-Numeral.Pos> I Code-Printer.literal-numeral
    target
    #> Numeral.add-code const-name <Code-Numeral.Neg> (~) Code-Printer.literal-numeral
    target)
  [SML, OCaml, Haskell, Scala]
>

```

code-printing

```

constant plus :: integer ⇒ - ⇒ - →
  (SML) IntInf.+ ((-), (-))
  and (OCaml) Z.add
  and (Haskell) infixl 6 +
  and (Scala) infixl 7 +
  and (Eval) infixl 8 +
| constant uminus :: integer ⇒ - →
  (SML) IntInf.~
  and (OCaml) Z.neg

```

```

    and (Haskell) negate
    and (Scala) !(- -)
    and (Eval) ~/ -
| constant minus :: integer => - ->
  (SML) IntInf.- ((-), (-))
  and (OCaml) Z.sub
  and (Haskell) infixl 6 -
  and (Scala) infixl 7 -
  and (Eval) infixl 8 -
| constant Code-Numeral.dup ->
  (SML) IntInf.* (2, / (-))
  and (OCaml) Z.shift'-left/ -/ 1
  and (Haskell) !(2 * -)
  and (Scala) !(2 * -)
  and (Eval) !(2 * -)
| constant Code-Numeral.sub ->
  (SML) !(raise/ Fail/ sub)
  and (OCaml) failwith/ sub
  and (Haskell) error/ sub
  and (Scala) !sys.error(sub)
| constant times :: integer => - => - ->
  (SML) IntInf.* ((-), (-))
  and (OCaml) Z.mul
  and (Haskell) infixl 7 *
  and (Scala) infixl 8 *
  and (Eval) infixl 9 *
| constant Code-Numeral.divmod-abs ->
  (SML) IntInf.divMod/ (IntInf.abs -, / IntInf.abs -)
  and (OCaml) !(fun k l -> / if Z.equal Z.zero l then/ (Z.zero, l) else/ Z.div'-rem/
(Z.abs k)/ (Z.abs l))
  and (Haskell) divMod/ (abs -)/ (abs -)
  and (Scala) !((k: BigInt) => (l: BigInt) => / if (l == 0)/ (BigInt(0), k) else/
(k.abs ' / % l.abs))
  and (Eval) Integer.div'-mod/ (abs -)/ (abs -)
| constant HOL.equal :: integer => - => bool ->
  (SML) !((- : IntInf.int) = -)
  and (OCaml) Z.equal
  and (Haskell) infix 4 ==
  and (Scala) infixl 5 ==
  and (Eval) infixl 6 =
| constant less-eq :: integer => - => bool ->
  (SML) IntInf.<= ((-), (-))
  and (OCaml) Z.leq
  and (Haskell) infix 4 <=
  and (Scala) infixl 4 <=
  and (Eval) infixl 6 <=
| constant less :: integer => - => bool ->
  (SML) IntInf.< ((-), (-))
  and (OCaml) Z.lt

```

```

and (Haskell) infix 4 <
and (Scala) infixl 4 <
and (Eval) infixl 6 <
| constant abs :: integer ⇒ - ->
  (SML) IntInf.abs
and (OCaml) Z.abs
and (Haskell) Prelude.abs
and (Scala) -.abs
and (Eval) abs

```

code-identifier

```

code-module Code-Numeral ↪ (SML) Arith and (OCaml) Arith and (Haskell)
Arith

```

70.4 Type of target language naturals

```

typedef natural = UNIV :: nat set
morphisms nat-of-natural natural-of-nat ..

```

```

setup-lifting type-definition-natural

```

```

lemma natural-eq-iff [termination-simp]:
   $m = n \iff \text{nat-of-natural } m = \text{nat-of-natural } n$ 
by transfer rule

```

```

lemma natural-eqI:
   $\text{nat-of-natural } m = \text{nat-of-natural } n \implies m = n$ 
using natural-eq-iff [of m n] by simp

```

```

lemma nat-of-natural-of-nat-inverse [simp]:
   $\text{nat-of-natural } (\text{natural-of-nat } n) = n$ 
by transfer rule

```

```

lemma natural-of-nat-of-natural-inverse [simp]:
   $\text{natural-of-nat } (\text{nat-of-natural } n) = n$ 
by transfer rule

```

```

instantiation natural :: {comm-monoid-diff, semiring-1}
begin

```

```

lift-definition zero-natural :: natural
  is 0 :: nat
  .

```

```

declare zero-natural.rep-eq [simp]

```

```

lift-definition one-natural :: natural
  is 1 :: nat
  .

```

```

declare one-natural.rep-eq [simp]

lift-definition plus-natural :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural
  is plus :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
  .

declare plus-natural.rep-eq [simp]

lift-definition minus-natural :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural
  is minus :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
  .

declare minus-natural.rep-eq [simp]

lift-definition times-natural :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural
  is times :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
  .

declare times-natural.rep-eq [simp]

instance proof
qed (transfer, simp add: algebra-simps) +

end

instance natural :: Rings.dvd ..

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
   $\langle (pcr\text{-}natural == == > pcr\text{-}natural == == > (\longleftrightarrow)) (dvd) (dvd) \rangle$ 
  by (unfold dvd-def) transfer-prover

lemma [transfer-rule]:
   $\langle ((\longleftrightarrow) == == > pcr\text{-}natural) of\text{-}bool of\text{-}bool \rangle$ 
  by (unfold of-bool-def) transfer-prover

lemma [transfer-rule]:
   $\langle ((=) == == > pcr\text{-}natural) (\lambda n. n) of\text{-}nat \rangle$ 
proof –
  have rel-fun HOL.eq pcr-natural (of-nat :: nat  $\Rightarrow$  nat) (of-nat :: nat  $\Rightarrow$  natural)
  by (unfold of-nat-def) transfer-prover
  then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:

```

```

  ⟨((=) ===> pcr-natural) numeral numeral⟩
proof –
  have ⟨((=) ===> pcr-natural) numeral (λn. of-nat (numeral n))⟩
    by transfer-prover
  then show ?thesis by simp
qed

```

```

lemma [transfer-rule]:
  ⟨(pcr-natural ===> (=) ===> pcr-natural) (∧) (∧)⟩
  by (unfold power-def) transfer-prover

```

end

```

lemma nat-of-natural-of-nat [simp]:
  nat-of-natural (of-nat n) = n
  by transfer rule

```

```

lemma natural-of-nat-of-nat [simp, code-abbrev]:
  natural-of-nat = of-nat
  by transfer rule

```

```

lemma of-nat-of-natural [simp]:
  of-nat (nat-of-natural n) = n
  by transfer rule

```

```

lemma nat-of-natural-numeral [simp]:
  nat-of-natural (numeral k) = numeral k
  by transfer rule

```

```

instantiation natural :: {linordered-semiring, equal}
begin

```

```

lift-definition less-eq-natural :: natural ⇒ natural ⇒ bool
  is less-eq :: nat ⇒ nat ⇒ bool
  .

```

```

declare less-eq-natural.rep-eq [termination-simp]

```

```

lift-definition less-natural :: natural ⇒ natural ⇒ bool
  is less :: nat ⇒ nat ⇒ bool
  .

```

```

declare less-natural.rep-eq [termination-simp]

```

```

lift-definition equal-natural :: natural ⇒ natural ⇒ bool
  is HOL.equal :: nat ⇒ nat ⇒ bool
  .

```

instance proof

qed (*transfer*, *simp add: algebra-simps equal less-le-not-le [symmetric] linear*)+

end

context

includes *lifting-syntax*

begin

lemma [*transfer-rule*]:

$\langle \text{pcr-natural} \implies \text{pcr-natural} \implies \text{pcr-natural} \rangle \text{ min min}$
by (*unfold min-def transfer-prover*)

lemma [*transfer-rule*]:

$\langle \text{pcr-natural} \implies \text{pcr-natural} \implies \text{pcr-natural} \rangle \text{ max max}$
by (*unfold max-def transfer-prover*)

end

lemma *nat-of-natural-min* [*simp*]:

$\text{nat-of-natural} (\text{min } k \ l) = \text{min} (\text{nat-of-natural } k) (\text{nat-of-natural } l)$
by *transfer rule*

lemma *nat-of-natural-max* [*simp*]:

$\text{nat-of-natural} (\text{max } k \ l) = \text{max} (\text{nat-of-natural } k) (\text{nat-of-natural } l)$
by *transfer rule*

instantiation *natural* :: *unique-euclidean-semiring*

begin

lift-definition *divide-natural* :: *natural* \Rightarrow *natural* \Rightarrow *natural*

is *divide* :: *nat* \Rightarrow *nat* \Rightarrow *nat*

.

declare *divide-natural.rep-eq* [*simp*]

lift-definition *modulo-natural* :: *natural* \Rightarrow *natural* \Rightarrow *natural*

is *modulo* :: *nat* \Rightarrow *nat* \Rightarrow *nat*

.

declare *modulo-natural.rep-eq* [*simp*]

lift-definition *euclidean-size-natural* :: *natural* \Rightarrow *nat*

is *euclidean-size* :: *nat* \Rightarrow *nat*

.

declare *euclidean-size-natural.rep-eq* [*simp*]

lift-definition *division-segment-natural* :: *natural* \Rightarrow *natural*

is *division-segment* :: *nat* \Rightarrow *nat*


```

.

declare division-segment-natural.rep-eq [simp]

instance
  by (standard; transfer)
      (auto simp add: algebra-simps unit-factor-nat-def gr0-conv-Suc)

end

lemma [code]:
  euclidean-size = nat-of-natural
  by (simp add: fun-eq-iff)

lemma [code]:
  division-segment (n::natural) = 1
  by (simp add: natural-eq-iff)

instance natural :: linordered-semidom
  by (standard; transfer) simp-all

instance natural :: unique-euclidean-semiring-with-nat
  by (standard; transfer) simp-all

instantiation natural :: semiring-bit-operations
begin

lift-definition bit-natural ::  $\langle \text{natural} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$ 
  is bit .

lift-definition and-natural ::  $\langle \text{natural} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$ 
  is  $\langle \text{and} \rangle$  .

lift-definition or-natural ::  $\langle \text{natural} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$ 
  is or .

lift-definition xor-natural ::  $\langle \text{natural} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$ 
  is xor .

lift-definition mask-natural ::  $\langle \text{nat} \Rightarrow \text{natural} \rangle$ 
  is mask .

lift-definition set-bit-natural ::  $\langle \text{nat} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$ 
  is set-bit .

lift-definition unset-bit-natural ::  $\langle \text{nat} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$ 
  is unset-bit .

lift-definition flip-bit-natural ::  $\langle \text{nat} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$ 

```

is *flip-bit* .

lift-definition *push-bit-natural* :: $\langle \text{nat} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$
is *push-bit* .

lift-definition *drop-bit-natural* :: $\langle \text{nat} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$
is *drop-bit* .

lift-definition *take-bit-natural* :: $\langle \text{nat} \Rightarrow \text{natural} \Rightarrow \text{natural} \rangle$
is *take-bit* .

instance by (*standard*; *transfer*)

(*fact bit-eq-rec bits-induct bit-iff-odd push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod*
bits-div-0 bits-div-by-1 bits-mod-div-trivial even-succ-div-2
exp-div-exp-eq div-exp-eq mod-exp-eq mult-exp-mod-exp-eq div-exp-mod-exp-eq
even-mask-div-iff even-mult-exp-div-exp-iff bit-and-iff bit-or-iff bit-xor-iff
mask-eq-exp-minus-1 set-bit-def bit-unset-bit-iff flip-bit-def)+

end

instance *natural* :: *unique-euclidean-semiring-with-bit-operations* ..

context

includes *bit-operations-syntax*

begin

lemma [*code*]:

$\langle \text{bit } m \ n \longleftrightarrow \text{odd } (\text{drop-bit } n \ m) \rangle$
 $\langle \text{mask } n = 2^{\wedge} n - (1 :: \text{integer}) \rangle$
 $\langle \text{set-bit } n \ m = m \ \text{OR} \ \text{push-bit } n \ 1 \rangle$
 $\langle \text{flip-bit } n \ m = m \ \text{XOR} \ \text{push-bit } n \ 1 \rangle$
 $\langle \text{push-bit } n \ m = m * 2^{\wedge} n \rangle$
 $\langle \text{drop-bit } n \ m = m \ \text{div} \ 2^{\wedge} n \rangle$
 $\langle \text{take-bit } n \ m = m \ \text{mod} \ 2^{\wedge} n \rangle$ **for** $m :: \text{natural}$
by (*fact bit-iff-odd-drop-bit mask-eq-exp-minus-1*
set-bit-eq-or flip-bit-eq-xor push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod)+

lemma [*code*]:

$\langle m \ \text{AND} \ n = (\text{if } m = 0 \vee n = 0 \text{ then } 0$
else $(m \ \text{mod} \ 2) * (n \ \text{mod} \ 2) + 2 * ((m \ \text{div} \ 2) \ \text{AND} \ (n \ \text{div} \ 2))) \rangle$ **for** $m \ n ::$
natural
by *transfer (fact and-nat-unfold)*

lemma [*code*]:

$\langle m \ \text{OR} \ n = (\text{if } m = 0 \text{ then } n \ \text{else if } n = 0 \text{ then } m$
else $\max (m \ \text{mod} \ 2) (n \ \text{mod} \ 2) + 2 * ((m \ \text{div} \ 2) \ \text{OR} \ (n \ \text{div} \ 2))) \rangle$ **for** $m \ n ::$
natural
by *transfer (fact or-nat-unfold)*

lemma [code]:
 $\langle m \text{ XOR } n = (\text{if } m = 0 \text{ then } n \text{ else if } n = 0 \text{ then } m$
 $\text{ else } (m \bmod 2 + n \bmod 2) \bmod 2 + 2 * ((m \text{ div } 2) \text{ XOR } (n \text{ div } 2))) \rangle$ **for** $m \ n$
 $:: \text{ natural}$
by *transfer (fact xor-nat-unfold)*

lemma [code]:
 $\langle \text{unset-bit } 0 \ m = 2 * (m \text{ div } 2) \rangle$
 $\langle \text{unset-bit } (\text{Suc } n) \ m = m \bmod 2 + 2 * \text{unset-bit } n \ (m \text{ div } 2) \rangle$ **for** $m :: \text{ natural}$
by (*transfer; simp add: unset-bit-Suc*)**+**

end

lift-definition *natural-of-integer* $:: \text{ integer} \Rightarrow \text{ natural}$
is *nat* $:: \text{ int} \Rightarrow \text{ nat}$

.

lift-definition *integer-of-natural* $:: \text{ natural} \Rightarrow \text{ integer}$
is *of-nat* $:: \text{ nat} \Rightarrow \text{ int}$

.

lemma *natural-of-integer-of-natural* [simp]:
 $\text{natural-of-integer } (\text{integer-of-natural } n) = n$
by *transfer simp*

lemma *integer-of-natural-of-integer* [simp]:
 $\text{integer-of-natural } (\text{natural-of-integer } k) = \text{max } 0 \ k$
by *transfer auto*

lemma *int-of-integer-of-natural* [simp]:
 $\text{int-of-integer } (\text{integer-of-natural } n) = \text{of-nat } (\text{nat-of-natural } n)$
by *transfer rule*

lemma *integer-of-natural-of-nat* [simp]:
 $\text{integer-of-natural } (\text{of-nat } n) = \text{of-nat } n$
by *transfer rule*

lemma [measure-function]:
 $\text{is-measure } \text{nat-of-natural}$
by (*rule is-measure-trivial*)

70.5 Inductive representation of target language naturals

lift-definition *Suc* $:: \text{ natural} \Rightarrow \text{ natural}$
is *Nat.Suc*

.

declare *Suc.rep-eq* [simp]

```

old-rep-datatype 0::natural Suc
  by (transfer, fact nat.induct nat.inject nat.distinct)+

lemma natural-cases [case-names nat, cases type: natural]:
  fixes m :: natural
  assumes  $\bigwedge n. m = \text{of-nat } n \implies P$ 
  shows P
  using assms by transfer blast

instantiation natural :: size
begin

definition size-nat where [simp, code]: size-nat = nat-of-natural

instance ..

end

lemma natural-decr [termination-simp]:
   $n \neq 0 \implies \text{nat-of-natural } n - \text{Nat.Suc } 0 < \text{nat-of-natural } n$ 
  by transfer simp

lemma natural-zero-minus-one: (0::natural) - 1 = 0
  by (rule zero-diff)

lemma Suc-natural-minus-one: Suc n - 1 = n
  by transfer simp

hide-const (open) Suc

```

70.6 Code refinement for target language naturals

```

lift-definition Nat :: integer  $\Rightarrow$  natural
  is nat
  .

lemma [code-post]:
  Nat 0 = 0
  Nat 1 = 1
  Nat (numeral k) = numeral k
  by (transfer, simp)+

lemma [code abstype]:
  Nat (integer-of-natural n) = n
  by transfer simp

lemma [code]:
  natural-of-nat n = natural-of-integer (integer-of-nat n)
  by transfer simp

```

lemma [*code abstract*]:

integer-of-natural (natural-of-integer k) = max 0 k
by *simp*

lemma [*code*]:

⟨integer-of-natural (mask n) = mask n⟩
by *transfer (simp add: mask-eq-exp-minus-1 of-nat-diff)*

lemma [*code-abbrev*]:

natural-of-integer (Code-Numeral.Pos k) = numeral k
by *transfer simp*

lemma [*code abstract*]:

integer-of-natural 0 = 0
by *transfer simp*

lemma [*code abstract*]:

integer-of-natural 1 = 1
by *transfer simp*

lemma [*code abstract*]:

integer-of-natural (Code-Numeral.Suc n) = integer-of-natural n + 1
by *transfer simp*

lemma [*code*]:

nat-of-natural = nat-of-integer ∘ integer-of-natural
by *transfer (simp add: fun-eq-iff)*

lemma [*code, code-unfold*]:

case-natural f g n = (if n = 0 then f else g (n - 1))
by (*cases n rule: natural.exhaust*) (*simp-all, simp add: Suc-def*)

declare *natural.rec* [*code del*]

lemma [*code abstract*]:

integer-of-natural (m + n) = integer-of-natural m + integer-of-natural n
by *transfer simp*

lemma [*code abstract*]:

integer-of-natural (m - n) = max 0 (integer-of-natural m - integer-of-natural n)
by *transfer simp*

lemma [*code abstract*]:

*integer-of-natural (m * n) = integer-of-natural m * integer-of-natural n*
by *transfer simp*

lemma [*code abstract*]:

integer-of-natural ($m \text{ div } n$) = *integer-of-natural* $m \text{ div integer-of-natural } n$
by *transfer* (*simp add: zdiv-int*)

lemma [*code abstract*]:
integer-of-natural ($m \text{ mod } n$) = *integer-of-natural* $m \text{ mod integer-of-natural } n$
by *transfer* (*simp add: zmod-int*)

lemma [*code*]:
HOL.equal $m \longleftrightarrow \text{HOL.equal (integer-of-natural } m) \text{ (integer-of-natural } n)$
by *transfer* (*simp add: equal*)

lemma [*code nbe*]: *HOL.equal* $n (n::\text{natural}) \longleftrightarrow \text{True}$
by (*rule equal-class.equal-refl*)

lemma [*code*]: $m \leq n \longleftrightarrow \text{integer-of-natural } m \leq \text{integer-of-natural } n$
by *transfer simp*

lemma [*code*]: $m < n \longleftrightarrow \text{integer-of-natural } m < \text{integer-of-natural } n$
by *transfer simp*

hide-const (**open**) *Nat*

code-reflect *Code-Numeral*
datatypes *natural*
functions *Code-Numeral.Suc* $0 :: \text{natural } 1 :: \text{natural}$
plus :: *natural* \Rightarrow - *minus* :: *natural* \Rightarrow -
times :: *natural* \Rightarrow - *divide* :: *natural* \Rightarrow -
modulo :: *natural* \Rightarrow -
integer-of-natural natural-of-integer

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

lifting-update *natural.lifting*
lifting-forget *natural.lifting*

end

71 A HOL random engine

theory *Random*
imports *List Groups-List Code-Numeral*
begin

71.1 Auxiliary functions

fun *log* :: *natural* \Rightarrow *natural* \Rightarrow *natural* **where**
log $b \ i = (\text{if } b \leq 1 \vee i < b \text{ then } 1 \text{ else } 1 + \text{log } b \ (i \text{ div } b))$

definition *inc-shift* :: *natural* \Rightarrow *natural* \Rightarrow *natural* **where**
inc-shift *v k* = (if *v* = *k* then 1 else *k* + 1)

definition *minus-shift* :: *natural* \Rightarrow *natural* \Rightarrow *natural* \Rightarrow *natural* **where**
minus-shift *r k l* = (if *k* < *l* then *r* + *k* - *l* else *k* - *l*)

71.2 Random seeds

type-synonym *seed* = *natural* \times *natural*

primrec *next* :: *seed* \Rightarrow *natural* \times *seed* **where**
next (*v*, *w*) = (let
k = *v* div 53668;
v' = *minus-shift* 2147483563 ((*v* mod 53668) * 40014) (*k* * 12211);
l = *w* div 52774;
w' = *minus-shift* 2147483399 ((*w* mod 52774) * 40692) (*l* * 3791);
z = *minus-shift* 2147483562 *v'* (*w'* + 1) + 1
in (*z*, (*v'*, *w'*)))

definition *split-seed* :: *seed* \Rightarrow *seed* \times *seed* **where**
split-seed *s* = (let
(*v*, *w*) = *s*;
(*v'*, *w'*) = *snd* (*next* *s*);
v'' = *inc-shift* 2147483562 *v*;
w'' = *inc-shift* 2147483398 *w*
in ((*v''*, *w'*), (*v'*, *w''*)))

71.3 Base selectors

context

includes *state-combinator-syntax*

begin

fun *iterate* :: *natural* \Rightarrow ('*b* \Rightarrow '*a* \Rightarrow '*b* \times '*a*) \Rightarrow '*b* \Rightarrow '*a* \Rightarrow '*b* \times '*a* **where**
iterate *k f x* = (if *k* = 0 then *Pair* *x* else *f x* $\circ\rightarrow$ *iterate* (*k* - 1) *f*)

definition *range* :: *natural* \Rightarrow *seed* \Rightarrow *natural* \times *seed* **where**
range *k* = *iterate* (*log* 2147483561 *k*)
(λ l. *next* $\circ\rightarrow$ (λ v. *Pair* (*v* + *l* * 2147483561))) 1
 $\circ\rightarrow$ (λ v. *Pair* (*v* mod *k*))

lemma *range*:

k > 0 \implies *fst* (*range* *k s*) < *k*

by (*simp* *add*: *range-def* *split-def* *less-natural-def* *del*: *log.simps* *iterate.simps*)

definition *select* :: '*a* list \Rightarrow *seed* \Rightarrow '*a* \times *seed* **where**
select *xs* = *range* (*natural-of-nat* (*length* *xs*))
 $\circ\rightarrow$ (λ k. *Pair* (*nth* *xs* (*nat-of-natural* *k*)))

lemma *select*:

assumes $xs \neq []$
shows $\text{fst } (\text{select } xs \ s) \in \text{set } xs$
proof –
from *assms* **have** $\text{natural-of-nat } (\text{length } xs) > 0$ **by** (*simp add: less-natural-def*)
with *range* **have**
 $\text{fst } (\text{range } (\text{natural-of-nat } (\text{length } xs)) \ s) < \text{natural-of-nat } (\text{length } xs)$ **by** *best*
then **have**
 $\text{nat-of-natural } (\text{fst } (\text{range } (\text{natural-of-nat } (\text{length } xs)) \ s)) < \text{length } xs$ **by** (*simp add: less-natural-def*)
then **show** *?thesis*
by (*simp add: split-beta select-def*)
qed

primrec *pick* :: $(\text{natural} \times 'a)$ list \Rightarrow $\text{natural} \Rightarrow 'a$ **where**
 $\text{pick } (x \# xs) \ i = (\text{if } i < \text{fst } x \text{ then } \text{snd } x \text{ else } \text{pick } xs \ (i - \text{fst } x))$

lemma *pick-member*:
 $i < \text{sum-list } (\text{map } \text{fst } xs) \Longrightarrow \text{pick } xs \ i \in \text{set } (\text{map } \text{snd } xs)$
by (*induct xs arbitrary: i*) (*simp-all add: less-natural-def*)

lemma *pick-drop-zero*:
 $\text{pick } (\text{filter } (\lambda(k, -). k > 0) \ xs) = \text{pick } xs$
by (*induct xs*) (*auto simp add: fun-eq-iff less-natural-def minus-natural-def*)

lemma *pick-same*:
 $l < \text{length } xs \Longrightarrow \text{Random.pick } (\text{map } (\text{Pair } 1) \ xs) \ (\text{natural-of-nat } l) = \text{nth } xs \ l$
proof (*induct xs arbitrary: l*)
case *Nil* **then** **show** *?case* **by** *simp*
next
case (*Cons x xs*) **then** **show** *?case* **by** (*cases l*) (*simp-all add: less-natural-def*)
qed

definition *select-weight* :: $(\text{natural} \times 'a)$ list \Rightarrow $\text{seed} \Rightarrow 'a \times \text{seed}$ **where**
 $\text{select-weight } xs = \text{range } (\text{sum-list } (\text{map } \text{fst } xs))$
 $\circ \rightarrow (\lambda k. \text{Pair } (\text{pick } xs \ k))$

lemma *select-weight-member*:
assumes $0 < \text{sum-list } (\text{map } \text{fst } xs)$
shows $\text{fst } (\text{select-weight } xs \ s) \in \text{set } (\text{map } \text{snd } xs)$
proof –
from *range assms*
have $\text{fst } (\text{range } (\text{sum-list } (\text{map } \text{fst } xs)) \ s) < \text{sum-list } (\text{map } \text{fst } xs)$.
with *pick-member*
have $\text{pick } xs \ (\text{fst } (\text{range } (\text{sum-list } (\text{map } \text{fst } xs)) \ s)) \in \text{set } (\text{map } \text{snd } xs)$.
then **show** *?thesis* **by** (*simp add: select-weight-def scomp-def split-def*)
qed

lemma *select-weight-cons-zero*:
 $\text{select-weight } ((0, x) \# xs) = \text{select-weight } xs$

by (simp add: select-weight-def less-natural-def)

lemma *select-weight-drop-zero*:

select-weight (filter ($\lambda(k, -). k > 0$) xs) = select-weight xs

proof –

have *sum-list (map fst [(k, -) ← xs . 0 < k]) = sum-list (map fst xs)*

by (induct xs) (auto simp add: less-natural-def natural-eq-iff)

then show *?thesis* **by** (simp only: select-weight-def pick-drop-zero)

qed

lemma *select-weight-select*:

assumes *xs ≠ []*

shows *select-weight (map (Pair 1) xs) = select xs*

proof –

have *less: $\bigwedge s. \text{fst} (\text{range} (\text{natural-of-nat} (\text{length } xs)) s) < \text{natural-of-nat} (\text{length } xs)$*

using *assms* **by** (intro range) (simp add: less-natural-def)

moreover have *sum-list (map fst (map (Pair 1) xs)) = natural-of-nat (length xs)*

by (induct xs) simp-all

ultimately show *?thesis*

by (auto simp add: select-weight-def select-def scomp-def split-def

fun-eq-iff pick-same [symmetric] less-natural-def)

qed

end

71.4 ML interface

code-reflect *Random-Engine*

functions *range select select-weight*

ML ‹

structure Random-Engine =

struct

open Random-Engine;

*type seed = Code-Numeral.natural * Code-Numeral.natural;*

local

val seed = Unsynchronized.ref

(let

val now = Time.toMilliseconds (Time.now ());

val (q, s1) = IntInf.divMod (now, 2147483562);

val s2 = q mod 2147483398;

in apply2 Code-Numeral.natural-of-integer (s1 + 1, s2 + 1) end);

```

in

fun next-seed () =
  let
    val (seed1, seed') = @{code split-seed} (! seed)
    val - = seed := seed'
  in
    seed1
  end

fun run f =
  let
    val (x, seed') = f (! seed);
    val - = seed := seed'
  in x end;

end;

end;
>

hide-type (open) seed
hide-const (open) inc-shift minus-shift log next split-seed
  iterate range select pick select-weight
hide-fact (open) range-def

end

```

72 Maps

```

theory Map
  imports List
  abbrevs (= =  $\subseteq_m$ )
begin

type-synonym ('a, 'b) map = 'a  $\Rightarrow$  'b option (infixr  $\rightarrow$  0)

abbreviation (input)
  empty :: 'a  $\rightarrow$  'b where
  empty  $\equiv$   $\lambda x. \text{None}$ 

definition
  map-comp :: ('b  $\rightarrow$  'c)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'c) (infixl  $\circ_m$  55) where
  f  $\circ_m$  g = ( $\lambda k. \text{case } g \text{ } k \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } v \Rightarrow f \text{ } v$ )

definition
  map-add :: ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) (infixl ++ 100) where
  m1 ++ m2 = ( $\lambda x. \text{case } m2 \text{ } x \text{ of } \text{None} \Rightarrow m1 \text{ } x \mid \text{Some } y \Rightarrow \text{Some } y$ )

```

definition

$restrict\text{-}map :: ('a \rightarrow 'b) \Rightarrow 'a\ set \Rightarrow ('a \rightarrow 'b)$ (**infixl** |‘ 110) **where**
 $m|'A = (\lambda x. \text{if } x \in A \text{ then } m\ x \text{ else } None)$

notation (*latex output*)

$restrict\text{-}map$ (-|· [111,110] 110)

definition

$dom :: ('a \rightarrow 'b) \Rightarrow 'a\ set$ **where**
 $dom\ m = \{a. m\ a \neq None\}$

definition

$ran :: ('a \rightarrow 'b) \Rightarrow 'b\ set$ **where**
 $ran\ m = \{b. \exists a. m\ a = Some\ b\}$

definition

$graph :: ('a \rightarrow 'b) \Rightarrow ('a \times 'b)\ set$ **where**
 $graph\ m = \{(a, b) \mid a\ b. m\ a = Some\ b\}$

definition

$map\text{-}le :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow bool$ (**infix** \subseteq_m 50) **where**
 $(m_1 \subseteq_m m_2) \longleftrightarrow (\forall a \in dom\ m_1. m_1\ a = m_2\ a)$

Function update syntax $f(x := y, \dots)$ is extended with $x \mapsto y$, which is short for $x := Some\ y$. $:=$ and \mapsto can be mixed freely. The syntax $[x \mapsto y, \dots]$ is short for $Map.empty(x \mapsto y, \dots)$ but must only contain \mapsto , not $:=$, because $[x:=y]$ clashes with the list update syntax $xs[i:=x]$.

nonterminal *maplet and maplets***syntax**

$\text{-maplet} :: ['a, 'a] \Rightarrow \text{maplet}$ (- / \mapsto / -)
 $:: \text{maplet} \Rightarrow \text{updbind}$ (-)
 $:: \text{maplet} \Rightarrow \text{maplets}$ (-)
 $\text{-Maplets} :: [\text{maplet}, \text{maplets}] \Rightarrow \text{maplets}$ (-, / -)
 $\text{-Map} :: \text{maplets} \Rightarrow 'a \rightarrow 'b$ ((1[-]))

syntax (*ASCII*)

$\text{-maplet} :: ['a, 'a] \Rightarrow \text{maplet}$ (- /|>/ -)

translations

$\text{-Update } f\ (\text{-maplet } x\ y) \Rightarrow f(x := CONST\ Some\ y)$
 $\text{-Maplets } m\ ms \rightarrow \text{-updbinds } m\ ms$
 $\text{-Map } ms \rightarrow \text{-Update } (CONST\ empty)\ ms$

$\text{-Map } (\text{-maplet } x\ y) \leftarrow \text{-Update } (\lambda u. CONST\ None)\ (\text{-maplet } x\ y)$
 $\text{-Map } (\text{-updbinds } m\ (\text{-maplet } x\ y)) \leftarrow \text{-Update } (\text{-Map } m)\ (\text{-maplet } x\ y)$

Updating with lists:

primrec *map-of* :: ('a × 'b) list ⇒ 'a → 'b **where**
map-of [] = *empty*
| *map-of* (p # ps) = (*map-of* ps)(fst p ↦ snd p)

lemma *map-of-Cons-code* [*code*]:
map-of [] k = *None*
map-of ((l, v) # ps) k = (if l = k then *Some* v else *map-of* ps k)
by *simp-all*

definition *map-upds* :: ('a → 'b) ⇒ 'a list ⇒ 'b list ⇒ 'a → 'b **where**
map-upds m xs ys = m ++ *map-of* (rev (zip xs ys))

There is also the more specialized update syntax *xs* [↦] *ys* for lists *xs* and *ys*.

syntax
-maplets :: ['a, 'a] ⇒ *maplet* (- / [↦] / -)

syntax (*ASCII*)
-maplets :: ['a, 'a] ⇒ *maplet* (- / [→] / -)

translations

-Update m (*-maplets* xs ys) ⇐ *CONST map-upds* m xs ys

-Map (*-maplets* xs ys) ← *-Update* (λu. *CONST None*) (*-maplets* xs ys)

-Map (*-updbinds* m (*-maplets* xs ys)) ← *-Update* (*-Map* m) (*-maplets* xs ys)

72.1 *empty*

lemma *empty-upd-none* [*simp*]: *empty*(x := *None*) = *empty*
by (*rule ext*) *simp*

72.2 *map-upd*

lemma *map-upd-triv*: t k = *Some* x ⇒ t(k ↦ x) = t
by (*rule ext*) *simp*

lemma *map-upd-nonempty* [*simp*]: t(k ↦ x) ≠ *empty*

proof

assume t(k ↦ x) = *empty*

then have (t(k ↦ x)) k = *None* **by** *simp*

then show *False* **by** *simp*

qed

lemma *map-upd-eqD1*:

assumes m(a ↦ x) = n(a ↦ y)

shows x = y

proof –

from *assms* **have** (m(a ↦ x)) a = (n(a ↦ y)) a **by** *simp*

then show *?thesis* **by** *simp*
qed

lemma *map-upd-Some-unfold*:

$((m(a \mapsto b)) x = \text{Some } y) = (x = a \wedge b = y \vee x \neq a \wedge m x = \text{Some } y)$
by *auto*

lemma *image-map-upd* [*simp*]: $x \notin A \implies m(x \mapsto y) ` A = m ` A$
by *auto*

lemma *finite-range-updI*:

assumes *finite* (*range f*) **shows** *finite* (*range (f(a \mapsto b))*)

proof –

have *range (f(a \mapsto b))* \subseteq *insert (Some b) (range f)*
by *auto*

then show *?thesis*

by (*rule finite-subset*) (*use assms in auto*)

qed

72.3 map-of

lemma *map-of-eq-empty-iff* [*simp*]:

$\text{map-of } xys = \text{empty} \longleftrightarrow xys = []$

proof

show $\text{map-of } xys = \text{empty} \implies xys = []$

by (*induction xys*) *simp-all*

qed *simp*

lemma *empty-eq-map-of-iff* [*simp*]:

$\text{empty} = \text{map-of } xys \longleftrightarrow xys = []$

by(*subst eq-commute*) *simp*

lemma *map-of-eq-None-iff*:

$(\text{map-of } xys x = \text{None}) = (x \notin \text{fst} ` (\text{set } xys))$

by (*induct xys*) *simp-all*

lemma *map-of-eq-Some-iff* [*simp*]:

$\text{distinct}(\text{map } \text{fst } xys) \implies (\text{map-of } xys x = \text{Some } y) = ((x,y) \in \text{set } xys)$

proof (*induct xys*)

case (*Cons xy xys*)

then show *?case*

by (*cases xy*) (*auto simp flip: map-of-eq-None-iff*)

qed *auto*

lemma *Some-eq-map-of-iff* [*simp*]:

$\text{distinct}(\text{map } \text{fst } xys) \implies (\text{Some } y = \text{map-of } xys x) = ((x,y) \in \text{set } xys)$

by (*auto simp del: map-of-eq-Some-iff simp: map-of-eq-Some-iff [symmetric]*)

lemma *map-of-is-SomeI* [*simp*]:

$\llbracket \text{distinct}(\text{map fst } xys); (x,y) \in \text{set } xys \rrbracket \implies \text{map-of } xys \ x = \text{Some } y$
by *simp*

lemma *map-of-zip-is-None* [*simp*]:

$\text{length } xs = \text{length } ys \implies (\text{map-of } (\text{zip } xs \ ys) \ x = \text{None}) = (x \notin \text{set } xs)$
by (*induct rule: list-induct2*) *simp-all*

lemma *map-of-zip-is-Some*:

assumes $\text{length } xs = \text{length } ys$
shows $x \in \text{set } xs \longleftrightarrow (\exists y. \text{map-of } (\text{zip } xs \ ys) \ x = \text{Some } y)$
using *assms* **by** (*induct rule: list-induct2*) *simp-all*

lemma *map-of-zip-upd*:

fixes $x :: 'a$ **and** $xs :: 'a \text{ list}$ **and** $ys \ zs :: 'b \text{ list}$
assumes $\text{length } ys = \text{length } xs$
and $\text{length } zs = \text{length } xs$
and $x \notin \text{set } xs$
and $(\text{map-of } (\text{zip } xs \ ys))(x \mapsto y) = (\text{map-of } (\text{zip } xs \ zs))(x \mapsto z)$
shows $\text{map-of } (\text{zip } xs \ ys) = \text{map-of } (\text{zip } xs \ zs)$

proof

fix $x' :: 'a$
show $\text{map-of } (\text{zip } xs \ ys) \ x' = \text{map-of } (\text{zip } xs \ zs) \ x'$
proof (*cases* $x = x'$)
case *True*
from *assms True map-of-zip-is-None* [*of xs ys x'*]
have $\text{map-of } (\text{zip } xs \ ys) \ x' = \text{None}$ **by** *simp*
moreover from *assms True map-of-zip-is-None* [*of xs zs x'*]
have $\text{map-of } (\text{zip } xs \ zs) \ x' = \text{None}$ **by** *simp*
ultimately show *?thesis* **by** *simp*
next
case *False* **from** *assms*
have $((\text{map-of } (\text{zip } xs \ ys))(x \mapsto y)) \ x' = ((\text{map-of } (\text{zip } xs \ zs))(x \mapsto z)) \ x'$ **by**
auto
with *False* **show** *?thesis* **by** *simp*
qed
qed

lemma *map-of-zip-inject*:

assumes $\text{length } ys = \text{length } xs$
and $\text{length } zs = \text{length } xs$
and *dist: distinct xs*
and $\text{map-of: map-of } (\text{zip } xs \ ys) = \text{map-of } (\text{zip } xs \ zs)$
shows $ys = zs$
using *assms(1) assms(2)[symmetric]*
using *dist map-of*
proof (*induct ys xs zs rule: list-induct3*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons y ys x xs z zs*)

from $\langle \text{map-of } (\text{zip } (x\#xs) (y\#ys)) = \text{map-of } (\text{zip } (x\#xs) (z\#zs)) \rangle$
have $\text{map-of: } (\text{map-of } (\text{zip } xs\ ys))(x \mapsto y) = (\text{map-of } (\text{zip } xs\ zs))(x \mapsto z)$ **by**
simp
from *Cons* **have** $\text{length } ys = \text{length } xs$ **and** $\text{length } zs = \text{length } xs$
and $x \notin \text{set } xs$ **by** *simp-all*
then **have** $\text{map-of } (\text{zip } xs\ ys) = \text{map-of } (\text{zip } xs\ zs)$ **using** *map-of* **by** (*rule*
map-of-zip-upd)
with *Cons.hyps* $\langle \text{distinct } (x \# xs) \rangle$ **have** $ys = zs$ **by** *simp*
moreover **from** *map-of* **have** $y = z$ **by** (*rule map-upd-eqD1*)
ultimately **show** *?case* **by** *simp*
qed

lemma *map-of-zip-nth*:
assumes $\text{length } xs = \text{length } ys$
assumes *distinct xs*
assumes $i < \text{length } ys$
shows $\text{map-of } (\text{zip } xs\ ys) (xs ! i) = \text{Some } (ys ! i)$
using *assms* **proof** (*induct arbitrary: i rule: list-induct2*)
case *Nil*
then **show** *?case* **by** *simp*
next
case (*Cons x xs y ys*)
then **show** *?case*
using *less-Suc-eq-0-disj* **by** *auto*
qed

lemma *map-of-zip-map*:
 $\text{map-of } (\text{zip } xs\ (\text{map } f\ xs)) = (\lambda x. \text{if } x \in \text{set } xs \text{ then } \text{Some } (f\ x) \text{ else } \text{None})$
by (*induct xs*) (*simp-all add: fun-eq-iff*)

lemma *finite-range-map-of*: $\text{finite } (\text{range } (\text{map-of } xys))$
proof (*induct xys*)
case (*Cons a xys*)
then **show** *?case*
using *finite-range-updI* **by** *fastforce*
qed *auto*

lemma *map-of-SomeD*: $\text{map-of } xs\ k = \text{Some } y \implies (k, y) \in \text{set } xs$
by (*induct xs*) (*auto split: if-splits*)

lemma *map-of-mapk-SomeI*:
 $\text{inj } f \implies \text{map-of } t\ k = \text{Some } x \implies$
 $\text{map-of } (\text{map } (\text{case-prod } (\lambda k. \text{Pair } (f\ k)))\ t)\ (f\ k) = \text{Some } x$
by (*induct t*) (*auto simp: inj-eq*)

lemma *weak-map-of-SomeI*: $(k, x) \in \text{set } l \implies \exists x. \text{map-of } l\ k = \text{Some } x$
by (*induct l*) *auto*

lemma *map-of-filter-in*:

$map\text{-of } xs \ k = Some \ z \implies P \ k \ z \implies map\text{-of } (filter \ (case\text{-prod } P) \ xs) \ k = Some \ z$
by $(induct \ xs) \ auto$

lemma $map\text{-of-map}$:

$map\text{-of } (map \ (\lambda(k, v). \ (k, f \ v)) \ xs) = map\text{-option } f \circ map\text{-of } xs$
by $(induct \ xs) \ (auto \ simp: fun\text{-eq-iff})$

lemma $dom\text{-map-option}$:

$dom \ (\lambda k. \ map\text{-option } (f \ k) \ (m \ k)) = dom \ m$
by $(simp \ add: dom\text{-def})$

lemma $dom\text{-map-option-comp}$ $[simp]$:

$dom \ (map\text{-option } g \circ m) = dom \ m$
using $dom\text{-map-option}$ $[of \ \lambda\cdot. \ g \ m]$ **by** $(simp \ add: comp\text{-def})$

72.4 map-option related

lemma $map\text{-option-o-empty}$ $[simp]$: $map\text{-option } f \circ empty = empty$
by $(rule \ ext) \ simp$

lemma $map\text{-option-o-map-upd}$ $[simp]$:

$map\text{-option } f \circ m(a \mapsto b) = (map\text{-option } f \circ m)(a \mapsto f \ b)$
by $(rule \ ext) \ simp$

72.5 map-comp related

lemma $map\text{-comp-empty}$ $[simp]$:

$m \circ_m empty = empty$
 $empty \circ_m m = empty$

by $(auto \ simp: map\text{-comp-def} \ split: option.splits)$

lemma $map\text{-comp-simps}$ $[simp]$:

$m2 \ k = None \implies (m1 \circ_m m2) \ k = None$
 $m2 \ k = Some \ k' \implies (m1 \circ_m m2) \ k = m1 \ k'$

by $(auto \ simp: map\text{-comp-def})$

lemma $map\text{-comp-Some-iff}$:

$((m1 \circ_m m2) \ k = Some \ v) = (\exists k'. \ m2 \ k = Some \ k' \wedge m1 \ k' = Some \ v)$

by $(auto \ simp: map\text{-comp-def} \ split: option.splits)$

lemma $map\text{-comp-None-iff}$:

$((m1 \circ_m m2) \ k = None) = (m2 \ k = None \vee (\exists k'. \ m2 \ k = Some \ k' \wedge m1 \ k' = None))$

by $(auto \ simp: map\text{-comp-def} \ split: option.splits)$

72.6 ++

lemma $map\text{-add-empty}$ $[simp]$: $m \ ++ \ empty = m$

by $(simp \ add: map\text{-add-def})$

lemma *empty-map-add*[simp]: $empty ++ m = m$
by (rule ext) (simp add: map-add-def split: option.split)

lemma *map-add-assoc*[simp]: $m1 ++ (m2 ++ m3) = (m1 ++ m2) ++ m3$
by (rule ext) (simp add: map-add-def split: option.split)

lemma *map-add-Some-iff*:
 $((m ++ n) k = Some\ x) = (n\ k = Some\ x \vee n\ k = None \wedge m\ k = Some\ x)$
by (simp add: map-add-def split: option.split)

lemma *map-add-SomeD* [dest!]:
 $(m ++ n) k = Some\ x \implies n\ k = Some\ x \vee n\ k = None \wedge m\ k = Some\ x$
by (rule map-add-Some-iff [THEN iffD1])

lemma *map-add-find-right* [simp]: $n\ k = Some\ xx \implies (m ++ n) k = Some\ xx$
by (subst map-add-Some-iff) fast

lemma *map-add-None* [iff]: $((m ++ n) k = None) = (n\ k = None \wedge m\ k = None)$
by (simp add: map-add-def split: option.split)

lemma *map-add-upd*[simp]: $f ++ g(x \mapsto y) = (f ++ g)(x \mapsto y)$
by (rule ext) (simp add: map-add-def)

lemma *map-add-upds*[simp]: $m1 ++ (m2(xs \mapsto ys)) = (m1 ++ m2)(xs \mapsto ys)$
by (simp add: map-upds-def)

lemma *map-add-upd-left*: $m \notin dom\ e2 \implies e1(m \mapsto u1) ++ e2 = (e1 ++ e2)(m \mapsto u1)$
by (rule ext) (auto simp: map-add-def dom-def split: option.split)

lemma *map-of-append*[simp]: $map-of\ (xs\ @\ ys) = map-of\ ys ++ map-of\ xs$
unfolding map-add-def
proof (induct xs)
case (Cons a xs)
then show ?case
by (force split: option.split)
qed auto

lemma *finite-range-map-of-map-add*:
 $finite\ (range\ f) \implies finite\ (range\ (f ++ map-of\ l))$
proof (induct l)
case (Cons a l)
then show ?case
by (metis finite-range-updI map-add-upd map-of.simps(2))
qed auto

lemma *inj-on-map-add-dom* [iff]:
 $inj-on\ (m ++ m')\ (dom\ m') = inj-on\ m'\ (dom\ m')$

by (*fastforce simp: map-add-def dom-def inj-on-def split: option.splits*)

lemma *map-upds-fold-map-upd*:

$m(k\text{s}\mapsto\text{vs}) = \text{foldl } (\lambda m (k, v). m(k \mapsto v)) m (\text{zip } ks \text{ vs})$

unfolding *map-upds-def* **proof** (*rule sym, rule zip-obtain-same-length*)

fix $ks :: 'a \text{ list}$ **and** $vs :: 'b \text{ list}$

assume $\text{length } ks = \text{length } vs$

then show $\text{foldl } (\lambda m (k, v). m(k \mapsto v)) m (\text{zip } ks \text{ vs}) = m ++ \text{map-of } (\text{rev } (\text{zip } ks \text{ vs}))$

by(*induct arbitrary: m rule: list-induct2*) *simp-all*

qed

lemma *map-add-map-of-foldr*:

$m ++ \text{map-of } ps = \text{foldr } (\lambda(k, v) m. m(k \mapsto v)) ps m$

by (*induct ps*) (*auto simp: fun-eq-iff map-add-def*)

72.7 restrict-map

lemma *restrict-map-to-empty* [*simp*]: $m|'\{\} = \text{empty}$

by (*simp add: restrict-map-def*)

lemma *restrict-map-insert*: $f|'(\text{insert } a \ A) = (f|' \ A)(a := f \ a)$

by (*auto simp: restrict-map-def*)

lemma *restrict-map-empty* [*simp*]: $\text{empty}|'D = \text{empty}$

by (*simp add: restrict-map-def*)

lemma *restrict-in* [*simp*]: $x \in A \implies (m|'A) \ x = m \ x$

by (*simp add: restrict-map-def*)

lemma *restrict-out* [*simp*]: $x \notin A \implies (m|'A) \ x = \text{None}$

by (*simp add: restrict-map-def*)

lemma *ran-restrictD*: $y \in \text{ran } (m|'A) \implies \exists x \in A. m \ x = \text{Some } y$

by (*auto simp: restrict-map-def ran-def split: if-split-asm*)

lemma *dom-restrict* [*simp*]: $\text{dom } (m|'A) = \text{dom } m \cap A$

by (*auto simp: restrict-map-def dom-def split: if-split-asm*)

lemma *restrict-upd-same* [*simp*]: $m(x \mapsto y)|'(-\{x\}) = m|'(-\{x\})$

by (*rule ext*) (*auto simp: restrict-map-def*)

lemma *restrict-restrict* [*simp*]: $m|'A|'B = m|'(A \cap B)$

by (*rule ext*) (*auto simp: restrict-map-def*)

lemma *restrict-fun-upd* [*simp*]:

$m(x := y)|'D = (\text{if } x \in D \text{ then } (m|'(D - \{x\}))(x := y) \text{ else } m|'D)$

by (*simp add: restrict-map-def fun-eq-iff*)

lemma *fun-upd-None-restrict* [simp]:
 $(m|'D)(x := \text{None}) = (\text{if } x \in D \text{ then } m|'(D - \{x\}) \text{ else } m|'D)$
by (*simp add: restrict-map-def fun-eq-iff*)

lemma *fun-upd-restrict*: $(m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$
by (*simp add: restrict-map-def fun-eq-iff*)

lemma *fun-upd-restrict-conv* [simp]:
 $x \in D \implies (m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$
by (*rule fun-upd-restrict*)

lemma *map-of-map-restrict*:
 $\text{map-of } (\text{map } (\lambda k. (k, f k)) ks) = (\text{Some } \circ f) |' \text{ set } ks$
by (*induct ks*) (*simp-all add: fun-eq-iff restrict-map-insert*)

lemma *restrict-complement-singleton-eq*:
 $f |' (- \{x\}) = f(x := \text{None})$
by *auto*

72.8 map-upds

lemma *map-upds-Nil1* [simp]: $m(\ [] \ [\mapsto] bs) = m$
by (*simp add: map-upds-def*)

lemma *map-upds-Nil2* [simp]: $m(as \ [\mapsto] \ []) = m$
by (*simp add: map-upds-def*)

lemma *map-upds-Cons* [simp]: $m(a \# as \ [\mapsto] b \# bs) = (m(a \mapsto b))(as \ [\mapsto] bs)$
by (*simp add: map-upds-def*)

lemma *map-upds-append1* [simp]:
 $\text{size } xs < \text{size } ys \implies m(xs @ [x] \ [\mapsto] ys) = m(xs \ [\mapsto] ys, x \mapsto ys!\text{size } xs)$

proof (*induct xs arbitrary: ys m*)

case *Nil*

then show *?case*

by (*auto simp: neq-Nil-conv*)

next

case (*Cons a xs*)

then show *?case*

by (*cases ys*) *auto*

qed

lemma *map-upds-list-update2-drop* [simp]:
 $\text{size } xs \leq i \implies m(xs \ [\mapsto] ys[i := y]) = m(xs \ [\mapsto] ys)$

proof (*induct xs arbitrary: m ys i*)

case *Nil*

then show *?case*

by *auto*

next

```

case (Cons a xs)
then show ?case
  by (cases ys) (use Cons in ⟨auto split: nat.split⟩)
qed

```

Something weirdly sensitive about this proof, which needs only four lines in apply style

```

lemma map-upd-upds-conv-if:
  (f(x↦y))(xs [↦] ys) =
    (if x ∈ set(take (length ys) xs) then f(xs [↦] ys)
     else (f(xs [↦] ys))(x↦y))
proof (induct xs arbitrary: x y ys f)
  case (Cons a xs)
  show ?case
  proof (cases ys)
    case (Cons z zs)
    then show ?thesis
      using Cons.hyps
      apply (auto split: if-split simp: fun-upd-twist)
      using Cons.hyps apply fastforce+
    done
  qed auto
qed auto

```

```

lemma map-upds-twist [simp]:
  a ∉ set as ⇒ m(a↦b, as[↦]bs) = m(as[↦]bs, a↦b)
using set-take-subset by (fastforce simp add: map-upd-upds-conv-if)

```

```

lemma map-upds-apply-nontin [simp]:
  x ∉ set xs ⇒ (f(xs[↦]ys)) x = f x
proof (induct xs arbitrary: ys)
  case (Cons a xs)
  then show ?case
    by (cases ys) (auto simp: map-upd-upds-conv-if)
qed auto

```

```

lemma fun-upds-append-drop [simp]:
  size xs = size ys ⇒ m(xs@zs[↦]ys) = m(xs[↦]ys)
proof (induct xs arbitrary: ys)
  case (Cons a xs)
  then show ?case
    by (cases ys) (auto simp: map-upd-upds-conv-if)
qed auto

```

```

lemma fun-upds-append2-drop [simp]:
  size xs = size ys ⇒ m(xs[↦]ys@zs) = m(xs[↦]ys)
proof (induct xs arbitrary: ys)
  case (Cons a xs)

```

```

then show ?case
  by (cases ys) (auto simp: map-upd-upds-conv-if)
qed auto

```

```

lemma restrict-map-upds[simp]:
  [[ length xs = length ys; set xs  $\subseteq$  D ]]
   $\implies$  m(xs [map] ys) | `D = (m | `(D - set xs))(xs [map] ys)
proof (induct xs arbitrary: m ys)
  case (Cons a xs)
  then show ?case
  proof (cases ys)
  case (Cons z zs)
  with Cons.hyps Cons.prems show ?thesis
  apply (simp add: insert-absorb flip: Diff-insert)
  apply (auto simp add: map-upd-upds-conv-if)
  done
qed auto
qed auto

```

72.9 dom

```

lemma dom-eq-empty-conv [simp]: dom f = {}  $\longleftrightarrow$  f = empty
  by (auto simp: dom-def)

```

```

lemma domI: m a = Some b  $\implies$  a  $\in$  dom m
  by (simp add: dom-def)

```

```

lemma domD: a  $\in$  dom m  $\implies$   $\exists$  b. m a = Some b
  by (cases m a) (auto simp add: dom-def)

```

```

lemma domIff [iff, simp del, code-unfold]: a  $\in$  dom m  $\longleftrightarrow$  m a  $\neq$  None
  by (simp add: dom-def)

```

```

lemma dom-empty [simp]: dom empty = {}
  by (simp add: dom-def)

```

```

lemma dom-fun-upd [simp]:
  dom(f(x := y)) = (if y = None then dom f - {x} else insert x (dom f))
  by (auto simp: dom-def)

```

```

lemma dom-if:
  dom ( $\lambda$ x. if P x then f x else g x) = dom f  $\cap$  {x. P x}  $\cup$  dom g  $\cap$  {x.  $\neg$  P x}
  by (auto split: if-splits)

```

```

lemma dom-map-of-conv-image-fst:
  dom (map-of xys) = fst ` set xys
  by (induct xys) (auto simp add: dom-if)

```

lemma *dom-map-of-zip* [*simp*]: $\text{length } xs = \text{length } ys \implies \text{dom } (\text{map-of } (\text{zip } xs \ ys)) = \text{set } xs$

by (*induct rule: list-induct2*) (*auto simp: dom-if*)

lemma *finite-dom-map-of*: $\text{finite } (\text{dom } (\text{map-of } l))$

by (*induct l*) (*auto simp: dom-def insert-Collect [symmetric]*)

lemma *dom-map-upds* [*simp*]:

$\text{dom}(m(xs[\mapsto]ys)) = \text{set}(\text{take } (\text{length } ys) \ xs) \cup \text{dom } m$

proof (*induct xs arbitrary: ys*)

case (*Cons a xs*)

then show *?case*

by (*cases ys*) (*auto simp: map-upd-upds-conv-if*)

qed *auto*

lemma *dom-map-add* [*simp*]: $\text{dom } (m \ ++ \ n) = \text{dom } n \cup \text{dom } m$

by (*auto simp: dom-def*)

lemma *dom-override-on* [*simp*]:

$\text{dom } (\text{override-on } f \ g \ A) =$

$(\text{dom } f - \{a. a \in A - \text{dom } g\}) \cup \{a. a \in A \cap \text{dom } g\}$

by (*auto simp: dom-def override-on-def*)

lemma *map-add-comm*: $\text{dom } m1 \cap \text{dom } m2 = \{\} \implies m1 \ ++ \ m2 = m2 \ ++ \ m1$

by (*rule ext*) (*force simp: map-add-def dom-def split: option.split*)

lemma *map-add-dom-app-simps*:

$m \in \text{dom } l2 \implies (l1 \ ++ \ l2) \ m = l2 \ m$

$m \notin \text{dom } l1 \implies (l1 \ ++ \ l2) \ m = l2 \ m$

$m \notin \text{dom } l2 \implies (l1 \ ++ \ l2) \ m = l1 \ m$

by (*auto simp add: map-add-def split: option.split-asm*)

lemma *dom-const* [*simp*]:

$\text{dom } (\lambda x. \text{Some } (f \ x)) = \text{UNIV}$

by *auto*

lemma *finite-map-freshness*:

$\text{finite } (\text{dom } (f :: 'a \rightarrow 'b)) \implies \neg \text{finite } (\text{UNIV} :: 'a \ \text{set}) \implies$

$\exists x. f \ x = \text{None}$

by (*bestsimp dest: ex-new-if-finite*)

lemma *dom-minus*:

$f \ x = \text{None} \implies \text{dom } f - \text{insert } x \ A = \text{dom } f - A$

unfolding *dom-def* **by** *simp*

lemma *insert-dom*:

$f \ x = \text{Some } y \implies \text{insert } x \ (\text{dom } f) = \text{dom } f$

unfolding *dom-def* **by** *auto*

lemma *map-of-map-keys*:

set xs = dom m \implies *map-of* (*map* ($\lambda k. (k, \text{the } (m\ k))$) *xs*) = *m*
by (*rule ext*) (*auto simp add: map-of-map-restrict restrict-map-def*)

lemma *map-of-eqI*:

assumes *set-eq*: *set* (*map fst xs*) = *set* (*map fst ys*)
assumes *map-eq*: $\forall k \in \text{set } (\text{map fst } xs). \text{map-of } xs\ k = \text{map-of } ys\ k$
shows *map-of xs* = *map-of ys*

proof (*rule ext*)

fix *k* **show** *map-of xs k* = *map-of ys k*

proof (*cases map-of xs k*)

case *None*

then have $k \notin \text{set } (\text{map fst } xs)$ **by** (*simp add: map-of-eq-None-iff*)

with *set-eq* **have** $k \notin \text{set } (\text{map fst } ys)$ **by** *simp*

then have *map-of ys k* = *None* **by** (*simp add: map-of-eq-None-iff*)

with *None* **show** *?thesis* **by** *simp*

next

case (*Some v*)

then have $k \in \text{set } (\text{map fst } xs)$ **by** (*auto simp add: dom-map-of-conv-image-fst*
[*symmetric*])

with *map-eq* **show** *?thesis* **by** *auto*

qed

qed

lemma *map-of-eq-dom*:

assumes *map-of xs* = *map-of ys*

shows *fst* ‘ *set xs* = *fst* ‘ *set ys*

proof –

from *assms* **have** *dom* (*map-of xs*) = *dom* (*map-of ys*) **by** *simp*

then show *?thesis* **by** (*simp add: dom-map-of-conv-image-fst*)

qed

lemma *finite-set-of-finite-maps*:

assumes *finite A* *finite B*

shows *finite* {*m. dom m* = *A* \wedge *ran m* \subseteq *B*} (**is** *finite* *?S*)

proof –

let *?S'* = {*m. $\forall x. (x \in A \implies m\ x \in \text{Some } 'B) \wedge (x \notin A \implies m\ x = \text{None})$* }

have *?S* = *?S'*

proof

show *?S* \subseteq *?S'* **by** (*auto simp: dom-def ran-def image-def*)

show *?S'* \subseteq *?S*

proof

fix *m* **assume** $m \in ?S'$

hence 1: *dom m* = *A* **by** *force*

hence 2: *ran m* \subseteq *B* **using** $\langle m \in ?S' \rangle$ **by** (*auto simp: dom-def ran-def*)

from 1 2 **show** $m \in ?S$ **by** *blast*

qed

qed
 with *assms* show ?thesis by (simp add: finite-set-of-finite-funs)
 qed

72.10 ran

lemma *ranI*: $m\ a = \text{Some } b \implies b \in \text{ran } m$
 by (auto simp: ran-def)

lemma *ran-empty* [simp]: $\text{ran } \text{empty} = \{\}$
 by (auto simp: ran-def)

lemma *ran-map-upd* [simp]: $m\ a = \text{None} \implies \text{ran}(m(a \mapsto b)) = \text{insert } b (\text{ran } m)$
 unfolding ran-def
 by force

lemma *fun-upd-None-if-notin-dom* [simp]: $k \notin \text{dom } m \implies m(k := \text{None}) = m$
 by auto

lemma *ran-map-upd-Some*:
 $\llbracket m\ x = \text{Some } y; \text{inj-on } m (\text{dom } m); z \notin \text{ran } m \rrbracket \implies \text{ran}(m(x := \text{Some } z)) =$
 $\text{ran } m - \{y\} \cup \{z\}$
 by (force simp add: ran-def domI inj-onD)

lemma *ran-map-add*:
 assumes $\text{dom } m1 \cap \text{dom } m2 = \{\}$
 shows $\text{ran } (m1 ++ m2) = \text{ran } m1 \cup \text{ran } m2$
 proof
 show $\text{ran } (m1 ++ m2) \subseteq \text{ran } m1 \cup \text{ran } m2$
 unfolding ran-def by auto

next

show $\text{ran } m1 \cup \text{ran } m2 \subseteq \text{ran } (m1 ++ m2)$

proof –

have $(m1 ++ m2)\ x = \text{Some } y$ if $m1\ x = \text{Some } y$ for $x\ y$

using *assms map-add-comm* that by *fastforce*

moreover have $(m1 ++ m2)\ x = \text{Some } y$ if $m2\ x = \text{Some } y$ for $x\ y$

using *assms* that by *auto*

ultimately show ?thesis

unfolding ran-def by *blast*

qed

qed

lemma *finite-ran*:
 assumes *finite* (dom *p*)
 shows *finite* (ran *p*)

proof –

have $\text{ran } p = (\lambda x. \text{the } (p\ x)) \text{ ` dom } p$

unfolding ran-def by *force*

from *this* $\langle \text{finite } (\text{dom } p) \rangle$ **show** *?thesis* **by** *auto*
qed

lemma *ran-distinct*:

assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd* ‘ *set al*
using *assms*
proof (*induct al*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons kv al*)
then have *ran* (*map-of al*) = *snd* ‘ *set al* **by** *simp*
moreover from *Cons.prem*s **have** *map-of al* (*fst kv*) = *None*
by (*simp add: map-of-eq-None-iff*)
ultimately show *?case* **by** (*simp only: map-of.simps ran-map-upd*) *simp*
qed

lemma *ran-map-of-zip*:

assumes *length xs* = *length ys* *distinct xs*
shows *ran* (*map-of* (*zip xs ys*)) = *set ys*
using *assms* **by** (*simp add: ran-distinct set-map[symmetric]*)

lemma *ran-map-option*: *ran* ($\lambda x. \text{map-option } f (m x)$) = *f* ‘ *ran m*
by (*auto simp add: ran-def*)

72.11 *graph*

lemma *graph-empty[simp]*: *graph empty* = {}
unfolding *graph-def* **by** *simp*

lemma *in-graphI*: $m k = \text{Some } v \implies (k, v) \in \text{graph } m$
unfolding *graph-def* **by** *blast*

lemma *in-graphD*: $(k, v) \in \text{graph } m \implies m k = \text{Some } v$
unfolding *graph-def* **by** *blast*

lemma *graph-map-upd[simp]*: *graph* ($m(k \mapsto v)$) = *insert* (*k*, *v*) (*graph* ($m(k := \text{None})$))
unfolding *graph-def* **by** (*auto split: if-splits*)

lemma *graph-fun-upd-None*: *graph* ($m(k := \text{None})$) = {*e* ∈ *graph m*. *fst e* ≠ *k*}
unfolding *graph-def* **by** (*auto split: if-splits*)

lemma *graph-restrictD*:

assumes $(k, v) \in \text{graph } (m \upharpoonright A)$
shows $k \in A$ **and** $m k = \text{Some } v$
using *assms* **unfolding** *graph-def*
by (*auto simp: restrict-map-def split: if-splits*)

lemma *graph-map-comp*[simp]: $\text{graph } (m1 \circ_m m2) = \text{graph } m2 \circ \text{graph } m1$
unfolding *graph-def* **by** (*auto simp: map-comp-Some-iff relcomp-unfold*)

lemma *graph-map-add*: $\text{dom } m1 \cap \text{dom } m2 = \{\} \implies \text{graph } (m1 ++ m2) = \text{graph } m1 \cup \text{graph } m2$
unfolding *graph-def* **using** *map-add-comm* **by** *force*

lemma *graph-eq-to-snd-dom*: $\text{graph } m = (\lambda x. (x, \text{the } (m \ x))) \text{ ` dom } m$
unfolding *graph-def dom-def* **by** *force*

lemma *fst-graph-eq-dom*: $\text{fst ` graph } m = \text{dom } m$
unfolding *graph-eq-to-snd-dom* **by** *force*

lemma *graph-domD*: $x \in \text{graph } m \implies \text{fst } x \in \text{dom } m$
using *fst-graph-eq-dom* **by** (*metis imageI*)

lemma *snd-graph-ran*: $\text{snd ` graph } m = \text{ran } m$
unfolding *graph-def ran-def* **by** *force*

lemma *graph-ranD*: $x \in \text{graph } m \implies \text{snd } x \in \text{ran } m$
using *snd-graph-ran* **by** (*metis imageI*)

lemma *finite-graph-map-of*: $\text{finite } (\text{graph } (\text{map-of } al))$
unfolding *graph-eq-to-snd-dom finite-dom-map-of*
using *finite-dom-map-of* **by** *blast*

lemma *graph-map-of-if-distinct-dom*: $\text{distinct } (\text{map } \text{fst } al) \implies \text{graph } (\text{map-of } al) = \text{set } al$
unfolding *graph-def* **by** *auto*

lemma *finite-graph-iff-finite-dom*[simp]: $\text{finite } (\text{graph } m) = \text{finite } (\text{dom } m)$
by (*metis graph-eq-to-snd-dom finite-imageI fst-graph-eq-dom*)

lemma *inj-on-fst-graph*: $\text{inj-on } \text{fst } (\text{graph } m)$
unfolding *graph-def inj-on-def* **by** *force*

72.12 *map-le*

lemma *map-le-empty* [simp]: $\text{empty} \subseteq_m g$
by (*simp add: map-le-def*)

lemma *upd-None-map-le* [simp]: $f(x := \text{None}) \subseteq_m f$
by (*force simp add: map-le-def*)

lemma *map-le-upd*[simp]: $f \subseteq_m g \implies f(a := b) \subseteq_m g(a := b)$
by (*fastforce simp add: map-le-def*)

lemma *map-le-imp-upd-le* [simp]: $m1 \subseteq_m m2 \implies m1(x := \text{None}) \subseteq_m m2(x \mapsto$

y)
by (*force simp add: map-le-def*)

lemma *map-le-upds* [*simp*]:
 $f \subseteq_m g \implies f(as \mapsto bs) \subseteq_m g(as \mapsto bs)$
proof (*induct as arbitrary: f g bs*)
case (*Cons a as*)
then show *?case*
by (*cases bs*) (*use Cons in auto*)
qed *auto*

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \implies (\text{dom } f \subseteq \text{dom } g)$
by (*fastforce simp add: map-le-def dom-def*)

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
by (*simp add: map-le-def*)

lemma *map-le-trans*[*trans*]: $\llbracket m1 \subseteq_m m2; m2 \subseteq_m m3 \rrbracket \implies m1 \subseteq_m m3$
by (*auto simp add: map-le-def dom-def*)

lemma *map-le-antisym*: $\llbracket f \subseteq_m g; g \subseteq_m f \rrbracket \implies f = g$
unfolding *map-le-def*
by (*metis ext domIff*)

lemma *map-le-map-add* [*simp*]: $f \subseteq_m g \implies f \ ++ \ f$
by (*fastforce simp: map-le-def*)

lemma *map-le-iff-map-add-commute*: $f \subseteq_m f \ ++ \ g \iff f \ ++ \ g = g \ ++ \ f$
by (*fastforce simp: map-add-def map-le-def fun-eq-iff split: option.splits*)

lemma *map-add-le-mapE*: $f \ ++ \ g \subseteq_m h \implies g \subseteq_m h$
by (*fastforce simp: map-le-def map-add-def dom-def*)

lemma *map-add-le-mapI*: $\llbracket f \subseteq_m h; g \subseteq_m h \rrbracket \implies f \ ++ \ g \subseteq_m h$
by (*auto simp: map-le-def map-add-def dom-def split: option.splits*)

lemma *map-add-subsumed1*: $f \subseteq_m g \implies f \ ++ \ g = g$
by (*simp add: map-add-le-mapI map-le-antisym*)

lemma *map-add-subsumed2*: $f \subseteq_m g \implies g \ ++ \ f = g$
by (*metis map-add-subsumed1 map-le-iff-map-add-commute*)

lemma *dom-eq-singleton-conv*: $\text{dom } f = \{x\} \iff (\exists v. f = [x \mapsto v])$
(is *?lhs* \iff *?rhs*)

proof
assume *?rhs*
then show *?lhs* **by** (*auto split: if-split-asm*)
next
assume *?lhs*

```

then obtain  $v$  where  $v: f\ x = \text{Some } v$  by auto
show ?rhs
proof
  show  $f = [x \mapsto v]$ 
  proof (rule map-le-antisym)
    show  $[x \mapsto v] \subseteq_m f$ 
    using  $v$  by (auto simp add: map-le-def)
    show  $f \subseteq_m [x \mapsto v]$ 
    using  $\langle \text{dom } f = \{x\} \rangle \langle f\ x = \text{Some } v \rangle$  by (auto simp add: map-le-def)
  qed
qed
qed

```

```

lemma map-add-eq-empty-iff[simp]:
   $(f++g = \text{empty}) \longleftrightarrow f = \text{empty} \wedge g = \text{empty}$ 
by (metis map-add-None)

```

```

lemma empty-eq-map-add-iff[simp]:
   $(\text{empty} = f++g) \longleftrightarrow f = \text{empty} \wedge g = \text{empty}$ 
by(subst map-add-eq-empty-iff[symmetric])(rule eq-commute)

```

72.13 Various

```

lemma set-map-of-compr:
  assumes distinct: distinct (map fst xs)
  shows  $\text{set } xs = \{(k, v). \text{map-of } xs\ k = \text{Some } v\}$ 
  using assms
proof (induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  obtain  $k\ v$  where  $x = (k, v)$  by (cases x) blast
  with Cons.prems have  $k \notin \text{dom } (\text{map-of } xs)$ 
  by (simp add: dom-map-of-conv-image-fst)
  then have  $*$ :  $\text{insert } (k, v) \{(k, v). \text{map-of } xs\ k = \text{Some } v\} =$ 
     $\{(k', v'). ((\text{map-of } xs)(k \mapsto v))\ k' = \text{Some } v'\}$ 
  by (auto split: if-splits)
  from Cons have  $\text{set } xs = \{(k, v). \text{map-of } xs\ k = \text{Some } v\}$  by simp
  with  $*$   $\langle x = (k, v) \rangle$  show ?case by simp
qed

```

```

lemma eq-key-imp-eq-value:
   $v1 = v2$ 
  if distinct (map fst xs)  $(k, v1) \in \text{set } xs$   $(k, v2) \in \text{set } xs$ 
proof –
  from that have inj-on fst (set xs)
  by (simp add: distinct-map)
  moreover have  $\text{fst } (k, v1) = \text{fst } (k, v2)$ 

```

```

  by simp
  ultimately have  $(k, v1) = (k, v2)$ 
  by (rule inj-onD) (fact that)+
  then show ?thesis
  by simp
qed

```

lemma *map-of-inject-set*:

```

  assumes distinct: distinct (map fst xs) distinct (map fst ys)
  shows map-of xs = map-of ys  $\longleftrightarrow$  set xs = set ys (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  moreover from  $\langle$ distinct (map fst xs) $\rangle$  have set xs =  $\{(k, v). \text{map-of } xs \ k =$ 
  Some v $\}$ 
  by (rule set-map-of-compr)
  moreover from  $\langle$ distinct (map fst ys) $\rangle$  have set ys =  $\{(k, v). \text{map-of } ys \ k =$ 
  Some v $\}$ 
  by (rule set-map-of-compr)
  ultimately show ?rhs by simp
next
  assume ?rhs show ?lhs
proof
  fix k
  show map-of xs k = map-of ys k
proof (cases map-of xs k)
  case None
  with  $\langle$ ?rhs $\rangle$  have map-of ys k = None
  by (simp add: map-of-eq-None-iff)
  with None show ?thesis by simp
next
  case (Some v)
  with distinct  $\langle$ ?rhs $\rangle$  have map-of ys k = Some v
  by simp
  with Some show ?thesis by simp
qed
qed
qed

```

lemma *finite-Map-induct*[consumes 1, case-names empty update]:

```

  assumes finite (dom m)
  assumes P Map.empty
  assumes  $\bigwedge k \ v \ m. \text{finite } (dom \ m) \implies k \notin dom \ m \implies P \ m \implies P \ (m(k \mapsto v))$ 
  shows P m
  using assms(1)
proof (induction dom m arbitrary: m rule: finite-induct)
  case empty
  then show ?case using assms(2) unfolding dom-def by simp
next
  case (insert x F)

```

```

then have finite (dom (m(x:=None))) x ∉ dom (m(x:=None)) P (m(x:=None))
  by (metis Diff-insert-absorb dom-fun-upd)+
with assms( $\mathcal{J}$ )[OF this] show ?case
  by (metis fun-upd-triv fun-upd-upd option.exhaust)
qed

```

```

hide-const (open) Map.empty Map.graph

```

```

end

```

73 Finite types as explicit enumerations

```

theory Enum
imports Map Groups-List
begin

```

73.1 Class *enum*

```

class enum =
  fixes enum :: 'a list
  fixes enum-all :: ('a ⇒ bool) ⇒ bool
  fixes enum-ex :: ('a ⇒ bool) ⇒ bool
  assumes UNIV-enum: UNIV = set enum
    and enum-distinct: distinct enum
  assumes enum-all-UNIV: enum-all P ⟷ Ball UNIV P
  assumes enum-ex-UNIV: enum-ex P ⟷ Bex UNIV P
  — tailored towards simple instantiation
begin

```

```

subclass finite proof
qed (simp add: UNIV-enum)

```

```

lemma enum-UNIV:
  set enum = UNIV
  by (simp only: UNIV-enum)

```

```

lemma in-enum: x ∈ set enum
  by (simp add: enum-UNIV)

```

```

lemma enum-eq-I:
  assumes  $\bigwedge x. x \in \text{set } xs$ 
  shows set enum = set xs
proof —
  from assms UNIV-eq-I have UNIV = set xs by auto
  with enum-UNIV show ?thesis by simp
qed

```

```

lemma card-UNIV-length-enum:
  card (UNIV :: 'a set) = length enum

```

by (*simp add: UNIV-enum distinct-card enum-distinct*)

lemma *enum-all* [*simp*]:
enum-all = *HOL.All*
by (*simp add: fun-eq-iff enum-all-UNIV*)

lemma *enum-ex* [*simp*]:
enum-ex = *HOL.Ex*
by (*simp add: fun-eq-iff enum-ex-UNIV*)

end

73.2 Implementations using *enum*

73.2.1 Unbounded operations and quantifiers

lemma *Collect-code* [*code*]:
Collect P = *set (filter P enum)*
by (*simp add: enum-UNIV*)

lemma *vimage-code* [*code*]:
f -‘ B = *set (filter ($\lambda x. f x \in B$) enum-class.enum)*
unfolding *vimage-def Collect-code ..*

definition *card-UNIV* :: *'a itself* \Rightarrow *nat*
where
[*code del*]: *card-UNIV TYPE('a)* = *card (UNIV :: 'a set)*

lemma [*code*]:
card-UNIV TYPE('a :: enum) = *card (set (Enum.enum :: 'a list))*
by (*simp only: card-UNIV-def enum-UNIV*)

lemma *all-code* [*code*]: $(\forall x. P x) \longleftrightarrow$ *enum-all P*
by *simp*

lemma *exists-code* [*code*]: $(\exists x. P x) \longleftrightarrow$ *enum-ex P*
by *simp*

lemma *exists1-code* [*code*]: $(\exists! x. P x) \longleftrightarrow$ *list-ex1 P enum*
by (*auto simp add: list-ex1-iff enum-UNIV*)

73.2.2 An executable choice operator

definition
[*code del*]: *enum-the* = *The*

lemma [*code*]:
The P = (*case filter P enum of [x] \Rightarrow x | - \Rightarrow enum-the P*)
proof –
{

```

fix a
assume filter-enum: filter P enum = [a]
have The P = a
proof (rule the-equality)
  fix x
  assume P x
  show x = a
  proof (rule ccontr)
    assume x ≠ a
    from filter-enum obtain us vs
      where enum-eq: enum = us @ [a] @ vs
      and ∀ x ∈ set us. ¬ P x
      and ∀ x ∈ set vs. ¬ P x
      and P a
    by (auto simp add: filter-eq-Cons-iff) (simp only: filter-empty-conv[symmetric])
    with ⟨P x⟩ in-enum[of x, unfolded enum-eq] ⟨x ≠ a⟩ show False by auto
  qed
next
  from filter-enum show P a by (auto simp add: filter-eq-Cons-iff)
qed
}
from this show ?thesis
  unfolding enum-the-def by (auto split: list.split)
qed

declare [[code abort: enum-the]]

code-printing
  constant enum-the → (Eval) (fn ' - => raise Match)

```

73.2.3 Equality and order on functions

```

instantiation fun :: (enum, equal) equal
begin

definition
  HOL.equal f g ↔ (∀ x ∈ set enum. f x = g x)

instance proof
qed (simp-all add: equal-fun-def fun-eq-iff enum-UNIV)

end

lemma [code]:
  HOL.equal f g ↔ enum-all (%x. f x = g x)
  by (auto simp add: equal-fun-eq-iff)

lemma [code nbe]:
  HOL.equal (f :: - ⇒ -) f ↔ True

```


by (fact equal-refl)

lemma *order-fun* [code]:
 fixes $f\ g :: 'a::enum \Rightarrow 'b::order$
 shows $f \leq g \iff enum\text{-}all\ (\lambda x. f\ x \leq g\ x)$
 and $f < g \iff f \leq g \wedge enum\text{-}ex\ (\lambda x. f\ x \neq g\ x)$
 by (simp-all add: fun-eq-iff le-fun-def order-less-le)

73.2.4 Operations on relations

lemma [code]:
 $Id = image\ (\lambda x. (x, x))\ (set\ Enum.enum)$
 by (auto intro: imageI in-enum)

lemma *tranclp-unfold* [code]:
 $tranclp\ r\ a\ b \iff (a, b) \in trancl\ \{(x, y). r\ x\ y\}$
 by (simp add: trancl-def)

lemma *rtranclp-rtrancl-eq* [code]:
 $rtranclp\ r\ x\ y \iff (x, y) \in rtrancl\ \{(x, y). r\ x\ y\}$
 by (simp add: rtrancl-def)

lemma *max-ext-eq* [code]:
 $max\text{-}ext\ R = \{(X, Y). finite\ X \wedge finite\ Y \wedge Y \neq \{\} \wedge (\forall x. x \in X \longrightarrow (\exists xa \in Y. (x, xa) \in R))\}$
 by (auto simp add: max-ext.simps)

lemma *max-extp-eq* [code]:
 $max\text{-}extp\ r\ x\ y \iff (x, y) \in max\text{-}ext\ \{(x, y). r\ x\ y\}$
 by (simp add: max-ext-def)

lemma *mlex-eq* [code]:
 $f < *mlex* > R = \{(x, y). f\ x < f\ y \vee (f\ x \leq f\ y \wedge (x, y) \in R)\}$
 by (auto simp add: mlex-prod-def)

73.2.5 Bounded accessible part

primrec *bacc* :: $('a \times 'a)\ set \Rightarrow nat \Rightarrow 'a\ set$

where

$bacc\ r\ 0 = \{x. \forall y. (y, x) \notin r\}$
 $| bacc\ r\ (Suc\ n) = (bacc\ r\ n \cup \{x. \forall y. (y, x) \in r \longrightarrow y \in bacc\ r\ n\})$

lemma *bacc-subseteq-acc*:
 $bacc\ r\ n \subseteq Wellfounded.acc\ r$
 by (induct n) (auto intro: acc.intros)

lemma *bacc-mono*:
 $n \leq m \implies bacc\ r\ n \subseteq bacc\ r\ m$
 by (induct rule: dec-induct) auto

lemma *bacc-upper-bound*:

bacc ($r :: ('a \times 'a)$ set) (card (*UNIV* :: $'a::$ finite set)) = $(\bigcup n. \text{bacc } r \ n)$

proof –

have *mono* (*bacc* r) **unfolding** *mono-def* **by** (*simp add: bacc-mono*)

moreover have $\forall n. \text{bacc } r \ n = \text{bacc } r \ (\text{Suc } n) \longrightarrow \text{bacc } r \ (\text{Suc } n) = \text{bacc } r \ (\text{Suc } n)$ **by** *auto*

moreover have *finite* (*range* (*bacc* r)) **by** *auto*

ultimately show *?thesis*

by (*intro finite-mono-strict-prefix-implies-finite-fixpoint*)

(*auto intro: finite-mono-remains-stable-implies-strict-prefix*)

qed

lemma *acc-subseteq-bacc*:

assumes *finite* r

shows *Wellfounded*.acc $r \subseteq (\bigcup n. \text{bacc } r \ n)$

proof

fix x

assume $x \in \text{Wellfounded}.acc \ r$

then have $\exists n. x \in \text{bacc } r \ n$

proof (*induct* x *arbitrary: rule: acc.induct*)

case (*accI* x)

then have $\forall y. \exists n. (y, x) \in r \longrightarrow y \in \text{bacc } r \ n$ **by** *simp*

from *choice[OF this]* **obtain** n **where** $n: \forall y. (y, x) \in r \longrightarrow y \in \text{bacc } r \ (n \ y)$

..

obtain n **where** $\bigwedge y. (y, x) \in r \implies y \in \text{bacc } r \ n$

proof

fix y **assume** $(y, x) \in r$

with n **have** $y \in \text{bacc } r \ (n \ y)$ **by** *auto*

moreover have $n \ y \leq \text{Max } ((\lambda(y, x). n \ y) \ 'r)$

using $y \ \langle \text{finite } r \rangle$ **by** (*auto intro!: Max-ge*)

note *bacc-mono[OF this, of r]*

ultimately show $y \in \text{bacc } r \ (\text{Max } ((\lambda(y, x). n \ y) \ 'r))$ **by** *auto*

qed

then show *?case*

by (*auto simp add: Let-def intro!: exI[of - Suc n]*)

qed

then show $x \in (\bigcup n. \text{bacc } r \ n)$ **by** *auto*

qed

lemma *acc-bacc-eq*:

fixes $A :: ('a :: \text{finite} \times 'a)$ set

assumes *finite* A

shows *Wellfounded*.acc $A = \text{bacc } A$ (card (*UNIV* :: $'a$ set))

using *assms* **by** (*metis acc-subseteq-bacc bacc-subseteq-acc bacc-upper-bound order-eq-iff*)

lemma [*code*]:

fixes $xs :: ('a::\text{finite} \times 'a)$ list

shows *Wellfounded*.acc (*set* xs) = *bacc* (*set* xs) (card-*UNIV* *TYPE*($'a$))

by (simp add: card-UNIV-def acc-bacc-eq)

73.3 Default instances for *enum*

lemma *map-of-zip-enum-is-Some*:

assumes $\text{length } ys = \text{length } (\text{enum} :: 'a::\text{enum list})$

shows $\exists y. \text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) ys) x = \text{Some } y$

proof –

from *assms* **have** $x \in \text{set } (\text{enum} :: 'a::\text{enum list}) \longleftrightarrow$

$(\exists y. \text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) ys) x = \text{Some } y)$

by (auto intro!: *map-of-zip-is-Some*)

then show *?thesis* **using** *enum-UNIV* **by** auto

qed

lemma *map-of-zip-enum-inject*:

fixes $xs\ ys :: 'b::\text{enum list}$

assumes *length*: $\text{length } xs = \text{length } (\text{enum} :: 'a::\text{enum list})$

$\text{length } ys = \text{length } (\text{enum} :: 'a::\text{enum list})$

and *map-of*: $\text{the } \circ \text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) xs) = \text{the } \circ \text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) ys)$

shows $xs = ys$

proof –

have *map-of* $(\text{zip } (\text{enum} :: 'a\ \text{list}) xs) = \text{map-of } (\text{zip } (\text{enum} :: 'a\ \text{list}) ys)$

proof

fix $x :: 'a$

from *length map-of-zip-enum-is-Some* **obtain** $y1\ y2$

where *map-of* $(\text{zip } (\text{enum} :: 'a\ \text{list}) xs) x = \text{Some } y1$

and *map-of* $(\text{zip } (\text{enum} :: 'a\ \text{list}) ys) x = \text{Some } y2$ **by** *blast*

moreover from *map-of*

have *the* $(\text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) xs) x) = \text{the } (\text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) ys) x)$

by (auto dest: *fun-cong*)

ultimately show *map-of* $(\text{zip } (\text{enum} :: 'a::\text{enum list}) xs) x = \text{map-of } (\text{zip } (\text{enum} :: 'a::\text{enum list}) ys) x$

by *simp*

qed

with *length enum-distinct* **show** $xs = ys$ **by** (rule *map-of-zip-inject*)

qed

definition *all-n-lists* :: $((a :: \text{enum})\ \text{list} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

all-n-lists $P\ n \longleftrightarrow (\forall xs \in \text{set } (\text{List.n-lists } n\ \text{enum}). P\ xs)$

lemma [*code*]:

all-n-lists $P\ n \longleftrightarrow (\text{if } n = 0 \text{ then } P\ [] \text{ else } \text{enum-all } (\%x. \text{all-n-lists } (\%xs. P\ (x \# xs))\ (n - 1)))$

unfolding *all-n-lists-def enum-all*

by (*cases n*) (auto simp add: *enum-UNIV*)

definition *ex-n-lists* :: (('a :: enum) list ⇒ bool) ⇒ nat ⇒ bool

where

ex-n-lists P n \longleftrightarrow (\exists xs \in set (List.n-lists n enum). P xs)

lemma [code]:

ex-n-lists P n \longleftrightarrow (if n = 0 then P [] else enum-ex (%x. *ex-n-lists* (%xs. P (x # xs)) (n - 1)))

unfolding *ex-n-lists-def* enum-ex

by (cases n) (auto simp add: enum-UNIV)

instantiation fun :: (enum, enum) enum

begin

definition

enum = map (λ ys. the \circ map-of (zip (enum::'a list) ys)) (List.n-lists (length (enum::'a::enum list)) enum)

definition

enum-all P = all-n-lists (λ bs. P (the \circ map-of (zip enum bs))) (length (enum :: 'a list))

definition

enum-ex P = *ex-n-lists* (λ bs. P (the \circ map-of (zip enum bs))) (length (enum :: 'a list))

instance proof

show UNIV = set (enum :: ('a ⇒ 'b) list)

proof (rule UNIV-eq-I)

fix f :: 'a ⇒ 'b

have f = the \circ map-of (zip (enum :: 'a::enum list) (map f enum))

by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)

then show f \in set enum

by (auto simp add: enum-fun-def set-n-lists intro: in-enum)

qed

next

from map-of-zip-enum-inject

show distinct (enum :: ('a ⇒ 'b) list)

by (auto intro!: inj-onI simp add: enum-fun-def

distinct-map distinct-n-lists enum-distinct set-n-lists)

next

fix P

show *enum-all* (P :: ('a ⇒ 'b) ⇒ bool) = Ball UNIV P

proof

assume *enum-all* P

show Ball UNIV P

proof

fix f :: 'a ⇒ 'b

have f: f = the \circ map-of (zip (enum :: 'a::enum list) (map f enum))

by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)

```

    from ⟨enum-all P⟩ have P (the ◦ map-of (zip enum (map f enum)))
      unfolding enum-all-fun-def all-n-lists-def
      apply (simp add: set-n-lists)
      apply (erule-tac x=map f enum in allE)
      apply (auto intro!: in-enum)
      done
  from this f show P f by auto
qed
next
  assume Ball UNIV P
  from this show enum-all P
    unfolding enum-all-fun-def all-n-lists-def by auto
  qed
next
fix P
show enum-ex (P :: ('a ⇒ 'b) ⇒ bool) = Bex UNIV P
proof
  assume enum-ex P
  from this show Bex UNIV P
    unfolding enum-ex-fun-def ex-n-lists-def by auto
  next
  assume Bex UNIV P
  from this obtain f where P f ..
  have f: f = the ◦ map-of (zip (enum :: 'a::enum list) (map f enum))
    by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
  from ⟨P f⟩ this have P (the ◦ map-of (zip (enum :: 'a::enum list) (map f
enum)))
    by auto
  from this show enum-ex P
    unfolding enum-ex-fun-def ex-n-lists-def
    apply (auto simp add: set-n-lists)
    apply (rule-tac x=map f enum in exI)
    apply (auto intro!: in-enum)
    done
  qed
qed
end

lemma enum-fun-code [code]: enum = (let enum-a = (enum :: 'a::{enum, equal}
list)
  in map (λys. the ◦ map-of (zip enum-a ys)) (List.n-lists (length enum-a) enum))
  by (simp add: enum-fun-def Let-def)

lemma enum-all-fun-code [code]:
enum-all P = (let enum-a = (enum :: 'a::{enum, equal} list)
  in all-n-lists (λbs. P (the ◦ map-of (zip enum-a bs))) (length enum-a))
  by (simp only: enum-all-fun-def Let-def)

```

lemma *enum-ex-fun-code* [code]:

enum-ex $P = (\text{let } \text{enum-a} = (\text{enum} :: 'a::\{\text{enum}, \text{equal}\} \text{list})$
in *ex-n-lists* ($\lambda bs. P (\text{the } \circ \text{map-of } (\text{zip } \text{enum-a } bs)) (\text{length } \text{enum-a})$)
by (*simp only*: *enum-ex-fun-def Let-def*)

instantiation *set* :: (*enum*) *enum*
begin

definition

enum = *map set* (*subseqs enum*)

definition

enum-all $P \longleftrightarrow (\forall A \in \text{set } \text{enum}. P (A::'a \text{ set}))$

definition

enum-ex $P \longleftrightarrow (\exists A \in \text{set } \text{enum}. P (A::'a \text{ set}))$

instance proof

qed (*simp-all add*: *enum-set-def enum-all-set-def enum-ex-set-def subseqs-powset*
distinct-set-subseqs
enum-distinct enum-UNIV)

end

instantiation *unit* :: *enum*
begin

definition

enum = [()]

definition

enum-all $P = P ()$

definition

enum-ex $P = P ()$

instance proof

qed (*auto simp add*: *enum-unit-def enum-all-unit-def enum-ex-unit-def*)

end

instantiation *bool* :: *enum*
begin

definition

enum = [*False*, *True*]

definition

enum-all $P \longleftrightarrow P \text{ False} \wedge P \text{ True}$

definition

$$\text{enum-ex } P \longleftrightarrow P \text{ False} \vee P \text{ True}$$
instance proof

qed (*simp-all only: enum-bool-def enum-all-bool-def enum-ex-bool-def UNIV-bool, simp-all*)

end

instantiation *prod* :: (*enum*, *enum*) *enum*
begin

definition

$$\text{enum} = \text{List.product } \text{enum } \text{enum}$$
definition

$$\text{enum-all } P = \text{enum-all } (\%x. \text{enum-all } (\%y. P (x, y)))$$
definition

$$\text{enum-ex } P = \text{enum-ex } (\%x. \text{enum-ex } (\%y. P (x, y)))$$
instance**by** *standard*

(*simp-all add: enum-prod-def distinct-product*

enum-UNIV enum-distinct enum-all-prod-def enum-ex-prod-def)

end

instantiation *sum* :: (*enum*, *enum*) *enum*
begin

definition

$$\text{enum} = \text{map } \text{Inl } \text{enum} \text{ @ } \text{map } \text{Inr } \text{enum}$$
definition

$$\text{enum-all } P \longleftrightarrow \text{enum-all } (\lambda x. P (\text{Inl } x)) \wedge \text{enum-all } (\lambda x. P (\text{Inr } x))$$
definition

$$\text{enum-ex } P \longleftrightarrow \text{enum-ex } (\lambda x. P (\text{Inl } x)) \vee \text{enum-ex } (\lambda x. P (\text{Inr } x))$$
instance proof

qed (*simp-all only: enum-sum-def enum-all-sum-def enum-ex-sum-def UNIV-sum, auto simp add: enum-UNIV distinct-map enum-distinct*)

end

instantiation *option* :: (*enum*) *enum*

begin

definition

$enum = None \# \text{map } Some \text{ enum}$

definition

$enum\text{-}all \ P \longleftrightarrow P \ None \wedge \ enum\text{-}all \ (\lambda x. P \ (Some \ x))$

definition

$enum\text{-}ex \ P \longleftrightarrow P \ None \vee \ enum\text{-}ex \ (\lambda x. P \ (Some \ x))$

instance proof

qed (*simp-all only: enum-option-def enum-all-option-def enum-ex-option-def UNIV-option-conv, auto simp add: distinct-map enum-UNIV enum-distinct*)

end

73.4 Small finite types

We define small finite types for use in Quickcheck

datatype (*plugins only: code quickcheck extraction*) *finite-1* =
 a_1

notation (**output**) $a_1 \ (a_1)$

lemma *UNIV-finite-1*:

$UNIV = \{a_1\}$

by (*auto intro: finite-1.exhaust*)

instantiation *finite-1* :: *enum*

begin

definition

$enum = [a_1]$

definition

$enum\text{-}all \ P = P \ a_1$

definition

$enum\text{-}ex \ P = P \ a_1$

instance proof

qed (*simp-all only: enum-finite-1-def enum-all-finite-1-def enum-ex-finite-1-def UNIV-finite-1, simp-all*)

end

instantiation *finite-1* :: *linorder*

begin


```

definition less-finite-1 :: finite-1  $\Rightarrow$  finite-1  $\Rightarrow$  bool
where
   $x < (y :: \textit{finite-1}) \longleftrightarrow \textit{False}$ 

definition less-eq-finite-1 :: finite-1  $\Rightarrow$  finite-1  $\Rightarrow$  bool
where
   $x \leq (y :: \textit{finite-1}) \longleftrightarrow \textit{True}$ 

instance
apply (intro-classes)
apply (auto simp add: less-finite-1-def less-eq-finite-1-def)
apply (metis (full-types) finite-1.exhaust)
done

end

instance finite-1 :: {dense-linorder, wellorder}
by intro-classes (simp-all add: less-finite-1-def)

instantiation finite-1 :: complete-lattice
begin

definition [simp]: Inf = ( $\lambda$ -. a1)
definition [simp]: Sup = ( $\lambda$ -. a1)
definition [simp]: bot = a1
definition [simp]: top = a1
definition [simp]: inf = ( $\lambda$ - -. a1)
definition [simp]: sup = ( $\lambda$ - -. a1)

instance by intro-classes (simp-all add: less-eq-finite-1-def)
end

instance finite-1 :: complete-distrib-lattice
  by standard simp-all

instance finite-1 :: complete-linorder ..

lemma finite-1-eq:  $x = a_1$ 
by (cases x) simp

simproc-setup finite-1-eq ( $x :: \textit{finite-1}$ ) =  $\langle$ 
   $K (K (fn \textit{ct} =>$ 
    (case Thm.term-of ct of
      Const (const-name  $\langle a_1 \rangle$ ,  $-$ )  $=>$  NONE
      |  $- =>$  SOME (mk-meta-eq @{\textit{thm finite-1-eq}}))
    ))
   $\rangle$ 

instantiation finite-1 :: complete-boolean-algebra

```

begin

definition $[simp]: (-) = (\lambda - . a_1)$

definition $[simp]: uminus = (\lambda - . a_1)$

instance by *intro-classes simp-all*

end

instantiation *finite-1* ::

$\{linordered-ring-strict, linordered-comm-semiring-strict, ordered-comm-ring,$
 $ordered-cancel-comm-monoid-diff, comm-monoid-mult, ordered-ring-abs,$
 $one, modulo, sgn, inverse\}$

begin

definition $[simp]: Groups.zero = a_1$

definition $[simp]: Groups.one = a_1$

definition $[simp]: (+) = (\lambda - . a_1)$

definition $[simp]: (*) = (\lambda - . a_1)$

definition $[simp]: (mod) = (\lambda - . a_1)$

definition $[simp]: abs = (\lambda - . a_1)$

definition $[simp]: sgn = (\lambda - . a_1)$

definition $[simp]: inverse = (\lambda - . a_1)$

definition $[simp]: divide = (\lambda - . a_1)$

instance by *intro-classes(simp-all add: less-finite-1-def)*

end

declare $[[simproc del: finite-1-eq]]$

hide-const (**open**) a_1

datatype (*plugins only: code quickcheck extraction*) *finite-2* =

$a_1 \mid a_2$

notation (**output**) a_1 (a_1)

notation (**output**) a_2 (a_2)

lemma *UNIV-finite-2*:

$UNIV = \{a_1, a_2\}$

by (*auto intro: finite-2.exhaust*)

instantiation *finite-2* :: *enum*

begin

definition

$enum = [a_1, a_2]$

definition

$enum-all P \longleftrightarrow P a_1 \wedge P a_2$

definition

$enum-ex P \longleftrightarrow P a_1 \vee P a_2$

instance proof

qed (*simp-all only: enum-finite-2-def enum-all-finite-2-def enum-ex-finite-2-def UNIV-finite-2, simp-all*)

end

instantiation *finite-2* :: *linorder*

begin

definition *less-finite-2* :: *finite-2* \Rightarrow *finite-2* \Rightarrow *bool*

where

$x < y \longleftrightarrow x = a_1 \wedge y = a_2$

definition *less-eq-finite-2* :: *finite-2* \Rightarrow *finite-2* \Rightarrow *bool*

where

$x \leq y \longleftrightarrow x = y \vee x < (y :: \text{finite-2})$

instance

apply (*intro-classes*)

apply (*auto simp add: less-finite-2-def less-eq-finite-2-def*)

apply (*metis finite-2.nchotomy*)+

done

end

instance *finite-2* :: *wellorder*

by(*rule wf-wellorderI*)(*simp add: less-finite-2-def, intro-classes*)

instantiation *finite-2* :: *complete-lattice*

begin

definition $\sqcap A = (\text{if } a_1 \in A \text{ then } a_1 \text{ else } a_2)$

definition $\sqcup A = (\text{if } a_2 \in A \text{ then } a_2 \text{ else } a_1)$

definition [*simp*]: *bot* = a_1

definition [*simp*]: *top* = a_2

definition $x \sqcap y = (\text{if } x = a_1 \vee y = a_1 \text{ then } a_1 \text{ else } a_2)$

definition $x \sqcup y = (\text{if } x = a_2 \vee y = a_2 \text{ then } a_2 \text{ else } a_1)$

lemma *neq-finite-2-a1-iff* [*simp*]: $x \neq a_1 \longleftrightarrow x = a_2$

by(*cases x*) *simp-all*

lemma *neq-finite-2-a1-iff'* [*simp*]: $a_1 \neq x \longleftrightarrow x = a_2$

by(*cases x*) *simp-all*

lemma *neq-finite-2-a2-iff* [*simp*]: $x \neq a_2 \longleftrightarrow x = a_1$

by(*cases x*) *simp-all*

lemma *neq-finite-2-a2-iff'* [*simp*]: $a_2 \neq x \longleftrightarrow x = a_1$

by(*cases x*) *simp-all*

```

instance
proof
  fix  $x :: \text{finite-2}$  and  $A$ 
  assume  $x \in A$ 
  then show  $\sqcap A \leq x \leq \sqcup A$ 
    by(cases  $x$ ; auto simp add: less-eq-finite-2-def less-finite-2-def Inf-finite-2-def
Sup-finite-2-def)+
qed(auto simp add: less-eq-finite-2-def less-finite-2-def inf-finite-2-def sup-finite-2-def
Inf-finite-2-def Sup-finite-2-def)
end

instance  $\text{finite-2} :: \text{complete-linorder} ..$ 

instance  $\text{finite-2} :: \text{complete-distrib-lattice} ..$ 

instantiation  $\text{finite-2} :: \{\text{field}, \text{idom-abs-sgn}, \text{idom-modulo}\}$  begin
definition [simp]:  $0 = a_1$ 
definition [simp]:  $1 = a_2$ 
definition  $x + y = (\text{case } (x, y) \text{ of } (a_1, a_1) \Rightarrow a_1 \mid (a_2, a_2) \Rightarrow a_1 \mid - \Rightarrow a_2)$ 
definition  $\text{uminus} = (\lambda x :: \text{finite-2}. x)$ 
definition  $(-) = ((+) :: \text{finite-2} \Rightarrow -)$ 
definition  $x * y = (\text{case } (x, y) \text{ of } (a_2, a_2) \Rightarrow a_2 \mid - \Rightarrow a_1)$ 
definition  $\text{inverse} = (\lambda x :: \text{finite-2}. x)$ 
definition  $\text{divide} = ((* :: \text{finite-2} \Rightarrow -)$ 
definition  $x \text{ mod } y = (\text{case } (x, y) \text{ of } (a_2, a_1) \Rightarrow a_2 \mid - \Rightarrow a_1)$ 
definition  $\text{abs} = (\lambda x :: \text{finite-2}. x)$ 
definition  $\text{sgn} = (\lambda x :: \text{finite-2}. x)$ 
instance
  by standard
  (subproofs
    ⟨simp-all add: plus-finite-2-def uminus-finite-2-def minus-finite-2-def
    times-finite-2-def
    inverse-finite-2-def divide-finite-2-def modulo-finite-2-def
    abs-finite-2-def sgn-finite-2-def
    split: finite-2.splits⟩)
end

lemma  $\text{two-finite-2}$  [simp]:
   $2 = a_1$ 
  by (simp add: numeral.simps plus-finite-2-def)

lemma  $\text{dvd-finite-2-unfold}$ :
   $x \text{ dvd } y \iff x = a_2 \vee y = a_1$ 
  by (auto simp add: dvd-def times-finite-2-def split: finite-2.splits)

instantiation  $\text{finite-2} :: \{\text{normalization-semidom}, \text{unique-euclidean-semiring}\}$  be-
gin
definition [simp]:  $\text{normalize} = (\text{id} :: \text{finite-2} \Rightarrow -)$ 

```

definition *[simp]*: *unit-factor* = (*id* :: *finite-2* \Rightarrow -)

definition *[simp]*: *eclidean-size* *x* = (*case* *x* of *a*₁ \Rightarrow 0 | *a*₂ \Rightarrow 1)

definition *[simp]*: *division-segment* (*x* :: *finite-2*) = 1

instance

by *standard*

 (*subproofs*

\langle *auto simp add: divide-finite-2-def times-finite-2-def dvd-finite-2-unfold*

split: finite-2.splits \rangle)

end

hide-const (**open**) *a*₁ *a*₂

datatype (*plugins only: code quickcheck extraction*) *finite-3* =

*a*₁ | *a*₂ | *a*₃

notation (**output**) *a*₁ (*a*₁)

notation (**output**) *a*₂ (*a*₂)

notation (**output**) *a*₃ (*a*₃)

lemma *UNIV-finite-3*:

$UNIV = \{a_1, a_2, a_3\}$

by (*auto intro: finite-3.exhaust*)

instantiation *finite-3* :: *enum*

begin

definition

$enum = [a_1, a_2, a_3]$

definition

$enum-all\ P \longleftrightarrow P\ a_1 \wedge P\ a_2 \wedge P\ a_3$

definition

$enum-ex\ P \longleftrightarrow P\ a_1 \vee P\ a_2 \vee P\ a_3$

instance proof

qed (*simp-all only: enum-finite-3-def enum-all-finite-3-def enum-ex-finite-3-def UNIV-finite-3, simp-all*)

end

lemma *finite-3-not-eq-unfold*:

$x \neq a_1 \longleftrightarrow x \in \{a_2, a_3\}$

$x \neq a_2 \longleftrightarrow x \in \{a_1, a_3\}$

$x \neq a_3 \longleftrightarrow x \in \{a_1, a_2\}$

by (*cases x; simp*)⁺

instantiation *finite-3* :: *linorder*

begin

definition *less-finite-3* :: *finite-3* \Rightarrow *finite-3* \Rightarrow *bool*

where

$x < y = (\text{case } x \text{ of } a_1 \Rightarrow y \neq a_1 \mid a_2 \Rightarrow y = a_3 \mid a_3 \Rightarrow \text{False})$

definition *less-eq-finite-3* :: *finite-3* \Rightarrow *finite-3* \Rightarrow *bool*

where

$x \leq y \longleftrightarrow x = y \vee x < (y :: \text{finite-3})$

instance proof (*intro-classes*)

qed (*auto simp add: less-finite-3-def less-eq-finite-3-def split: finite-3.split-asm*)

end

instance *finite-3* :: *wellorder*

proof(*rule wf-wellorderI*)

have *inv-image less-than* (*case-finite-3 0 1 2*) = $\{(x, y). x < y\}$

by(*auto simp add: less-finite-3-def split: finite-3.splits*)

from *this[symmetric]* **show** *wf* ... **by** *simp*

qed *intro-classes*

class *finite-lattice* = *finite* + *lattice* + *Inf* + *Sup* + *bot* + *top* +

assumes *Inf-finite-empty*: *Inf* $\{\}$ = *Sup UNIV*

assumes *Inf-finite-insert*: *Inf* (*insert a A*) = $a \sqcap \text{Inf } A$

assumes *Sup-finite-empty*: *Sup* $\{\}$ = *Inf UNIV*

assumes *Sup-finite-insert*: *Sup* (*insert a A*) = $a \sqcup \text{Sup } A$

assumes *bot-finite-def*: *bot* = *Inf UNIV*

assumes *top-finite-def*: *top* = *Sup UNIV*

begin

subclass *complete-lattice*

proof

fix *x A*

show $x \in A \Longrightarrow \sqcap A \leq x$

by (*metis Set.set-insert abel-semigroup commute local.Inf-finite-insert local.inf.abel-semigroup-axioms local.inf.left-idem local.inf.orderI*)

show $x \in A \Longrightarrow x \leq \sqcup A$

by (*metis Set.set-insert insert-absorb2 local.Sup-finite-insert local.sup.absorb-iff2*)

next

fix *A z*

have $\sqcup \text{UNIV} = z \sqcup \sqcup \text{UNIV}$

by (*subst Sup-finite-insert [symmetric], simp add: insert-UNIV*)

from *this* **have** [*simp*]: $z \leq \sqcup \text{UNIV}$

using *local.le-iff-sup* **by** *auto*

have $(\forall x. x \in A \longrightarrow z \leq x) \longrightarrow z \leq \sqcap A$

by (*rule finite-induct [of A $\lambda A . (\forall x. x \in A \longrightarrow z \leq x) \longrightarrow z \leq \sqcap A]$*)

(*simp-all add: Inf-finite-empty Inf-finite-insert*)

from *this* **show** $(\bigwedge x. x \in A \Longrightarrow z \leq x) \Longrightarrow z \leq \sqcap A$

```

  by simp

  have  $\sqcap UNIV = z \sqcap \sqcap UNIV$ 
    by (subst Inf-finite-insert [symmetric], simp add: insert-UNIV)
  from this have [simp]:  $\sqcap UNIV \leq z$ 
    by (simp add: local.inf.absorb-iff2)
  have  $(\forall x. x \in A \longrightarrow x \leq z) \longrightarrow \sqcup A \leq z$ 
    by (rule finite-induct [of A  $\lambda A. (\forall x. x \in A \longrightarrow x \leq z) \longrightarrow \sqcup A \leq z$ ],
  simp-all add: Sup-finite-empty Sup-finite-insert)
  from this show  $(\bigwedge x. x \in A \implies x \leq z) \implies \sqcup A \leq z$ 
    by blast
next
  show  $\sqcap \{\} = \top$ 
    by (simp add: Inf-finite-empty top-finite-def)
  show  $\sqcup \{\} = \perp$ 
    by (simp add: Sup-finite-empty bot-finite-def)
qed
end

```

class *finite-distrib-lattice* = *finite-lattice* + *distrib-lattice*

begin

lemma *finite-inf-Sup*: $a \sqcap (\text{Sup } A) = \text{Sup } \{a \sqcap b \mid b. b \in A\}$

proof (rule *finite-induct* [of A $\lambda A. a \sqcap (\text{Sup } A) = \text{Sup } \{a \sqcap b \mid b. b \in A\}$],
simp-all)

fix $x::'a$

fix F

assume $x \notin F$

assume [simp]: $a \sqcap \sqcup F = \sqcup \{a \sqcap b \mid b. b \in F\}$

have [simp]: $\text{insert } (a \sqcap x) \{a \sqcap b \mid b. b \in F\} = \{a \sqcap b \mid b. b = x \vee b \in F\}$

by blast

have $a \sqcap (x \sqcup \sqcup F) = a \sqcap x \sqcup a \sqcap \sqcup F$

by (simp add: inf-sup-distrib1)

also have $\dots = a \sqcap x \sqcup \sqcup \{a \sqcap b \mid b. b \in F\}$

by simp

also have $\dots = \sqcup \{a \sqcap b \mid b. b = x \vee b \in F\}$

by (unfold *Sup-insert*[*THEN sym*], simp)

finally show $a \sqcap (x \sqcup \sqcup F) = \sqcup \{a \sqcap b \mid b. b = x \vee b \in F\}$

by simp

qed

lemma *finite-Inf-Sup*: $\sqcap (\text{Sup } 'A) \leq \sqcup (\text{Inf } ' \{f ' A \mid f. \forall Y \in A. f Y \in Y\})$

proof (rule *finite-induct* [of A $\lambda A. \sqcap (\text{Sup } 'A) \leq \sqcup (\text{Inf } ' \{f ' A \mid f. \forall Y \in A. f Y \in Y\})$],
simp-all add: finite-UnionD)

fix $x::'a$ set

fix F

assume $x \notin F$

have [simp]: $\{\sqcup x \sqcap b \mid b. b \in \text{Inf } ' \{f ' F \mid f. \forall Y \in F. f Y \in Y\}\} = \{\sqcup x \sqcap (\text{Inf } (f ' F)) \mid f. (\forall Y \in F. f Y \in Y)\}$

by auto

```

define fa where fa = ( $\lambda (b::'a) f Y . (if\ Y = x\ then\ b\ else\ f\ Y)$ )
have  $\bigwedge f b. \forall Y \in F. f\ Y \in Y \implies b \in x \implies insert\ b\ (f\ ' (F \cap \{Y. Y \neq x\})) =$ 
 $insert\ (fa\ b\ f\ x)\ (fa\ b\ f\ ' F) \wedge fa\ b\ f\ x \in x \wedge (\forall Y \in F. fa\ b\ f\ Y \in Y)$ 
by (auto simp add: fa-def)
from this have B:  $\bigwedge f b. \forall Y \in F. f\ Y \in Y \implies b \in x \implies fa\ b\ f\ ' (\{x\} \cup F) \in$ 
 $\{insert\ (f\ x)\ (f\ ' F) \mid f. f\ x \in x \wedge (\forall Y \in F. f\ Y \in Y)\}$ 
by blast
have [simp]:  $\bigwedge f b. \forall Y \in F. f\ Y \in Y \implies b \in x \implies b \sqcap (\bigsqcap_{x \in F. f\ x}) \leq \bigsqcup (Inf$ 
 $' \{insert\ (f\ x)\ (f\ ' F) \mid f. f\ x \in x \wedge (\forall Y \in F. f\ Y \in Y)\})$ 
using B apply (rule SUP-upper2)
using  $\langle x \notin F \rangle$  apply (simp-all add: fa-def Inf-union-distrib)
apply (simp add: image-mono Inf-superset-mono inf.coboundedI2)
done
assume  $\bigsqcap (Sup\ ' F) \leq \bigsqcup (Inf\ ' \{f\ ' F \mid f. \forall Y \in F. f\ Y \in Y\})$ 

from this have  $\bigsqcup x \sqcap \bigsqcap (Sup\ ' F) \leq \bigsqcup x \sqcap \bigsqcup (Inf\ ' \{f\ ' F \mid f. \forall Y \in F. f\ Y \in Y\})$ 
using inf.coboundedI2 by auto
also have  $\dots = Sup\ \{\bigsqcup x \sqcap (Inf\ (f\ ' F)) \mid f . (\forall Y \in F. f\ Y \in Y)\}$ 
by (simp add: finite-inf-Sup)

also have  $\dots = Sup\ \{Sup\ \{Inf\ (f\ ' F) \sqcap b \mid b . b \in x\} \mid f . (\forall Y \in F. f\ Y \in Y)\}$ 
by (subst inf-commute) (simp add: finite-inf-Sup)

also have  $\dots \leq \bigsqcup (Inf\ ' \{insert\ (f\ x)\ (f\ ' F) \mid f. f\ x \in x \wedge (\forall Y \in F. f\ Y \in Y)\})$ 
apply (rule Sup-least, clarsimp)+
apply (subst inf-commute, simp)
done

finally show  $\bigsqcup x \sqcap \bigsqcap (Sup\ ' F) \leq \bigsqcup (Inf\ ' \{insert\ (f\ x)\ (f\ ' F) \mid f. f\ x \in x \wedge$ 
 $(\forall Y \in F. f\ Y \in Y)\})$ 
by simp
qed

```

```

subclass complete-distrib-lattice
by (standard, rule finite-Inf-Sup)
end

```

```

instantiation finite-3 :: finite-lattice
begin

```

```

definition  $\bigsqcap A = (if\ a_1 \in A\ then\ a_1\ else\ if\ a_2 \in A\ then\ a_2\ else\ a_3)$ 
definition  $\bigsqcup A = (if\ a_3 \in A\ then\ a_3\ else\ if\ a_2 \in A\ then\ a_2\ else\ a_1)$ 
definition [simp]: bot =  $a_1$ 
definition [simp]: top =  $a_3$ 
definition [simp]: inf = (min :: finite-3  $\Rightarrow$  -)
definition [simp]: sup = (max :: finite-3  $\Rightarrow$  -)

```

```

instance
proof

```


qed (*auto simp add: Inf-finite-3-def Sup-finite-3-def max-def min-def less-eq-finite-3-def less-finite-3-def split: finite-3.split*)

end

instance *finite-3* :: *complete-lattice* ..

instance *finite-3* :: *finite-distrib-lattice*

proof

qed (*auto simp add: min-def max-def*)

instance *finite-3* :: *complete-distrib-lattice* ..

instance *finite-3* :: *complete-linorder* ..

instantiation *finite-3* :: {*field, idom-abs-sgn, idom-modulo*} **begin**

definition [*simp*]: $0 = a_1$

definition [*simp*]: $1 = a_2$

definition

$x + y = (\text{case } (x, y) \text{ of}$
 $(a_1, a_1) \Rightarrow a_1 \mid (a_2, a_3) \Rightarrow a_1 \mid (a_3, a_2) \Rightarrow a_1$
 $\mid (a_1, a_2) \Rightarrow a_2 \mid (a_2, a_1) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2$
 $\mid - \Rightarrow a_3)$

definition $- x = (\text{case } x \text{ of } a_1 \Rightarrow a_1 \mid a_2 \Rightarrow a_3 \mid a_3 \Rightarrow a_2)$

definition $x - y = x + (- y :: \text{finite-3})$

definition $x * y = (\text{case } (x, y) \text{ of } (a_2, a_2) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2 \mid (a_2, a_3) \Rightarrow a_3$
 $\mid (a_3, a_2) \Rightarrow a_3 \mid - \Rightarrow a_1)$

definition *inverse* = $(\lambda x :: \text{finite-3}. x)$

definition $x \text{ div } y = x * \text{inverse } (y :: \text{finite-3})$

definition $x \text{ mod } y = (\text{case } y \text{ of } a_1 \Rightarrow x \mid - \Rightarrow a_1)$

definition *abs* = $(\lambda x. \text{case } x \text{ of } a_3 \Rightarrow a_2 \mid - \Rightarrow x)$

definition *sgn* = $(\lambda x :: \text{finite-3}. x)$

instance

by *standard*

(*subproofs*

$\langle \text{simp-all add: plus-finite-3-def uminus-finite-3-def minus-finite-3-def$

times-finite-3-def

inverse-finite-3-def divide-finite-3-def modulo-finite-3-def

abs-finite-3-def sgn-finite-3-def

less-finite-3-def

split: finite-3.splits \rangle)

end

lemma *two-finite-3* [*simp*]:

$2 = a_3$

by (*simp add: numeral.simps plus-finite-3-def*)

lemma *dvd-finite-3-unfold*:

$x \text{ dvd } y \longleftrightarrow x = a_2 \vee x = a_3 \vee y = a_1$

by (*cases x*) (*auto simp add: dvd-def times-finite-3-def split: finite-3.splits*)

instantiation *finite-3* :: {*normalization-semidom, unique-euclidean-semiring*} **begin**

definition [*simp*]: *normalize* $x = (\text{case } x \text{ of } a_3 \Rightarrow a_2 \mid - \Rightarrow x)$

definition [*simp*]: *unit-factor* = (*id* :: *finite-3* \Rightarrow -)

definition [*simp*]: *euclidean-size* $x = (\text{case } x \text{ of } a_1 \Rightarrow 0 \mid - \Rightarrow 1)$

definition [*simp*]: *division-segment* ($x :: \text{finite-3}$) = 1

instance

proof

fix $x :: \text{finite-3}$

assume $x \neq 0$

then show *is-unit* (*unit-factor* x)

by (*cases* x) (*simp-all add: dvd-finite-3-unfold*)

qed

(*subproofs*

$\langle \text{auto simp add: divide-finite-3-def times-finite-3-def}$

$\text{dvd-finite-3-unfold inverse-finite-3-def plus-finite-3-def}$

$\text{split: finite-3.splits} \rangle$)

end

hide-const (**open**) $a_1 a_2 a_3$

datatype (*plugins only: code quickcheck extraction*) *finite-4* =

$a_1 \mid a_2 \mid a_3 \mid a_4$

notation (**output**) a_1 (a_1)

notation (**output**) a_2 (a_2)

notation (**output**) a_3 (a_3)

notation (**output**) a_4 (a_4)

lemma *UNIV-finite-4*:

$UNIV = \{a_1, a_2, a_3, a_4\}$

by (*auto intro: finite-4.exhaust*)

instantiation *finite-4* :: *enum*

begin

definition

enum = [a_1, a_2, a_3, a_4]

definition

enum-all $P \longleftrightarrow P a_1 \wedge P a_2 \wedge P a_3 \wedge P a_4$

definition

enum-ex $P \longleftrightarrow P a_1 \vee P a_2 \vee P a_3 \vee P a_4$

instance proof

qed (*simp-all only: enum-finite-4-def enum-all-finite-4-def enum-ex-finite-4-def UNIV-finite-4, simp-all*)

end

instantiation *finite-4* :: *finite-distrib-lattice* **begin**

$a_1 < a_2, a_3 < a_4$, but a_2 and a_3 are incomparable.

definition

$$x < y \iff (\text{case } (x, y) \text{ of}$$

$$\begin{array}{l} (a_1, a_1) \Rightarrow \text{False} \mid (a_1, -) \Rightarrow \text{True} \\ \mid (a_2, a_4) \Rightarrow \text{True} \\ \mid (a_3, a_4) \Rightarrow \text{True} \mid - \Rightarrow \text{False} \end{array})$$

definition

$$x \leq y \iff (\text{case } (x, y) \text{ of}$$

$$\begin{array}{l} (a_1, -) \Rightarrow \text{True} \\ \mid (a_2, a_2) \Rightarrow \text{True} \mid (a_2, a_4) \Rightarrow \text{True} \\ \mid (a_3, a_3) \Rightarrow \text{True} \mid (a_3, a_4) \Rightarrow \text{True} \\ \mid (a_4, a_4) \Rightarrow \text{True} \mid - \Rightarrow \text{False} \end{array})$$

definition

$\sqcap A = (\text{if } a_1 \in A \vee a_2 \in A \wedge a_3 \in A \text{ then } a_1 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_4)$

definition

$\sqcup A = (\text{if } a_4 \in A \vee a_2 \in A \wedge a_3 \in A \text{ then } a_4 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_1)$

definition [*simp*]: $\text{bot} = a_1$

definition [*simp*]: $\text{top} = a_4$

definition

$$x \sqcap y = (\text{case } (x, y) \text{ of}$$

$$\begin{array}{l} (a_1, -) \Rightarrow a_1 \mid (-, a_1) \Rightarrow a_1 \mid (a_2, a_3) \Rightarrow a_1 \mid (a_3, a_2) \Rightarrow a_1 \\ \mid (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\ \mid (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\ \mid - \Rightarrow a_4 \end{array})$$

definition

$$x \sqcup y = (\text{case } (x, y) \text{ of}$$

$$\begin{array}{l} (a_4, -) \Rightarrow a_4 \mid (-, a_4) \Rightarrow a_4 \mid (a_2, a_3) \Rightarrow a_4 \mid (a_3, a_2) \Rightarrow a_4 \\ \mid (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\ \mid (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\ \mid - \Rightarrow a_1 \end{array})$$

instance

by *standard*

(*subproofs*

$\langle \text{auto simp add: less-finite-4-def less-eq-finite-4-def Inf-finite-4-def Sup-finite-4-def}$

$\text{inf-finite-4-def sup-finite-4-def split: finite-4.splits} \rangle$)

end

instance *finite-4* :: *complete-lattice* ..

instance *finite-4* :: *complete-distrib-lattice* ..

instantiation *finite-4* :: *complete-boolean-algebra* **begin**

definition $- x = (\text{case } x \text{ of } a_1 \Rightarrow a_4 \mid a_2 \Rightarrow a_3 \mid a_3 \Rightarrow a_2 \mid a_4 \Rightarrow a_1)$

definition $x - y = x \sqcap - (y :: \textit{finite-4})$

instance

by *standard*

 (*subproofs*

$\langle \textit{simp-all add: inf-finite-4-def sup-finite-4-def uminus-finite-4-def minus-finite-4-def}$

$\textit{split: finite-4.splits} \rangle$

end

hide-const (**open**) $a_1 a_2 a_3 a_4$

datatype (*plugins only: code quickcheck extraction*) *finite-5* =

$a_1 \mid a_2 \mid a_3 \mid a_4 \mid a_5$

notation (**output**) a_1 (a_1)

notation (**output**) a_2 (a_2)

notation (**output**) a_3 (a_3)

notation (**output**) a_4 (a_4)

notation (**output**) a_5 (a_5)

lemma *UNIV-finite-5*:

$UNIV = \{a_1, a_2, a_3, a_4, a_5\}$

by (*auto intro: finite-5.exhaust*)

instantiation *finite-5* :: *enum*

begin

definition

$enum = [a_1, a_2, a_3, a_4, a_5]$

definition

$enum\text{-}all P \longleftrightarrow P a_1 \wedge P a_2 \wedge P a_3 \wedge P a_4 \wedge P a_5$

definition

$enum\text{-}ex P \longleftrightarrow P a_1 \vee P a_2 \vee P a_3 \vee P a_4 \vee P a_5$

instance proof

qed (*simp-all only: enum-finite-5-def enum-all-finite-5-def enum-ex-finite-5-def UNIV-finite-5, simp-all*)

end

instantiation *finite-5* :: *finite-lattice*

begin

The non-distributive pentagon lattice N_5

definition

$$\begin{aligned}
 x < y &\longleftrightarrow (\text{case } (x, y) \text{ of} \\
 &\quad (a_1, a_1) \Rightarrow \text{False} \mid (a_1, -) \Rightarrow \text{True} \\
 &\quad \mid (a_2, a_3) \Rightarrow \text{True} \mid (a_2, a_5) \Rightarrow \text{True} \\
 &\quad \mid (a_3, a_5) \Rightarrow \text{True} \\
 &\quad \mid (a_4, a_5) \Rightarrow \text{True} \mid - \Rightarrow \text{False})
 \end{aligned}$$

definition

$$\begin{aligned}
 x \leq y &\longleftrightarrow (\text{case } (x, y) \text{ of} \\
 &\quad (a_1, -) \Rightarrow \text{True} \\
 &\quad \mid (a_2, a_2) \Rightarrow \text{True} \mid (a_2, a_3) \Rightarrow \text{True} \mid (a_2, a_5) \Rightarrow \text{True} \\
 &\quad \mid (a_3, a_3) \Rightarrow \text{True} \mid (a_3, a_5) \Rightarrow \text{True} \\
 &\quad \mid (a_4, a_4) \Rightarrow \text{True} \mid (a_4, a_5) \Rightarrow \text{True} \\
 &\quad \mid (a_5, a_5) \Rightarrow \text{True} \mid - \Rightarrow \text{False})
 \end{aligned}$$

definition

$$\begin{aligned}
 \sqcap A &= \\
 &(\text{if } a_1 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A) \text{ then } a_1 \\
 &\quad \text{else if } a_2 \in A \text{ then } a_2 \\
 &\quad \text{else if } a_3 \in A \text{ then } a_3 \\
 &\quad \text{else if } a_4 \in A \text{ then } a_4 \\
 &\quad \text{else } a_5)
 \end{aligned}$$

definition

$$\begin{aligned}
 \sqcup A &= \\
 &(\text{if } a_5 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A) \text{ then } a_5 \\
 &\quad \text{else if } a_3 \in A \text{ then } a_3 \\
 &\quad \text{else if } a_2 \in A \text{ then } a_2 \\
 &\quad \text{else if } a_4 \in A \text{ then } a_4 \\
 &\quad \text{else } a_1)
 \end{aligned}$$

definition [simp]: $\text{bot} = a_1$

definition [simp]: $\text{top} = a_5$

definition

$$\begin{aligned}
 x \sqcap y &= (\text{case } (x, y) \text{ of} \\
 &\quad (a_1, -) \Rightarrow a_1 \mid (-, a_1) \Rightarrow a_1 \mid (a_2, a_4) \Rightarrow a_1 \mid (a_4, a_2) \Rightarrow a_1 \mid (a_3, a_4) \Rightarrow a_1 \mid \\
 &\quad (a_4, a_3) \Rightarrow a_1 \\
 &\quad \mid (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\
 &\quad \mid (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\
 &\quad \mid (a_4, -) \Rightarrow a_4 \mid (-, a_4) \Rightarrow a_4 \\
 &\quad \mid - \Rightarrow a_5)
 \end{aligned}$$

definition

$$\begin{aligned}
 x \sqcup y &= (\text{case } (x, y) \text{ of} \\
 &\quad (a_5, -) \Rightarrow a_5 \mid (-, a_5) \Rightarrow a_5 \mid (a_2, a_4) \Rightarrow a_5 \mid (a_4, a_2) \Rightarrow a_5 \mid (a_3, a_4) \Rightarrow a_5 \mid \\
 &\quad (a_4, a_3) \Rightarrow a_5 \\
 &\quad \mid (a_3, -) \Rightarrow a_3 \mid (-, a_3) \Rightarrow a_3 \\
 &\quad \mid (a_2, -) \Rightarrow a_2 \mid (-, a_2) \Rightarrow a_2 \\
 &\quad \mid (a_4, -) \Rightarrow a_4 \mid (-, a_4) \Rightarrow a_4 \\
 &\quad \mid - \Rightarrow a_1)
 \end{aligned}$$

```

instance
  by standard
    (subproofs
      ⟨auto simp add: less-eq-finite-5-def less-finite-5-def inf-finite-5-def sup-finite-5-def

        Inf-finite-5-def Sup-finite-5-def split: finite-5.splits if-split-asm⟩)
end

```

```

instance finite-5 :: complete-lattice ..

```

```

hide-const (open) a1 a2 a3 a4 a5

```

73.5 Closing up

```

hide-type (open) finite-1 finite-2 finite-3 finite-4 finite-5
hide-const (open) enum enum-all enum-ex all-n-lists ex-n-lists ntranc1
end

```

74 Character and string types

```

theory String
imports Enum Bit-Operations Code-Numeral
begin

```

74.1 Strings as list of bytes

When modelling strings, we follow the approach given in <https://utf8everywhere.org/>:

- Strings are a list of bytes (8 bit).
- Byte values from 0 to 127 are US-ASCII.
- Byte values from 128 to 255 are uninterpreted blobs.

74.1.1 Bytes as datatype

```

datatype char =
  Char (digit0: bool) (digit1: bool) (digit2: bool) (digit3: bool)
    (digit4: bool) (digit5: bool) (digit6: bool) (digit7: bool)

context comm-semiring-1
begin

```

definition *of-char* :: $\langle \text{char} \Rightarrow 'a \rangle$

where $\langle \text{of-char } c = \text{horner-sum of-bool } 2 [\text{digit0 } c, \text{digit1 } c, \text{digit2 } c, \text{digit3 } c, \text{digit4 } c, \text{digit5 } c, \text{digit6 } c, \text{digit7 } c] \rangle$

lemma *of-char-Char* [*simp*]:

$\langle \text{of-char } (\text{Char } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ b7) = \text{horner-sum of-bool } 2 [b0, b1, b2, b3, b4, b5, b6, b7] \rangle$

by (*simp add: of-char-def*)

end

lemma (**in** *comm-semiring-1*) *of-nat-of-char*:

$\langle \text{of-nat } (\text{of-char } c) = \text{of-char } c \rangle$

by (*cases c simp*)

lemma (**in** *comm-ring-1*) *of-int-of-char*:

$\langle \text{of-int } (\text{of-char } c) = \text{of-char } c \rangle$

by (*cases c simp*)

lemma *nat-of-char* [*simp*]:

$\langle \text{nat } (\text{of-char } c) = \text{of-char } c \rangle$

by (*cases c (simp only: of-char-Char nat-horner-sum)*)

context *unique-euclidean-semiring-with-bit-operations*

begin

definition *char-of* :: $\langle 'a \Rightarrow \text{char} \rangle$

where $\langle \text{char-of } n = \text{Char } (\text{bit } n \ 0) (\text{bit } n \ 1) (\text{bit } n \ 2) (\text{bit } n \ 3) (\text{bit } n \ 4) (\text{bit } n \ 5) (\text{bit } n \ 6) (\text{bit } n \ 7) \rangle$

lemma *char-of-take-bit-eq*:

$\langle \text{char-of } (\text{take-bit } n \ m) = \text{char-of } m \rangle$ **if** $\langle n \geq 8 \rangle$

using that by (*simp add: char-of-def bit-take-bit-iff*)

lemma *char-of-char* [*simp*]:

$\langle \text{char-of } (\text{of-char } c) = c \rangle$

by (*simp only: of-char-def char-of-def bit-horner-sum-bit-iff simp*)

lemma *char-of-comp-of-char* [*simp*]:

$\text{char-of} \circ \text{of-char} = \text{id}$

by (*simp add: fun-eq-iff*)

lemma *inj-of-char*:

$\langle \text{inj of-char} \rangle$

proof (*rule injI*)

fix *c d*

assume $\text{of-char } c = \text{of-char } d$

then have $\text{char-of } (\text{of-char } c) = \text{char-of } (\text{of-char } d)$

```

  by simp
  then show  $c = d$ 
  by simp
qed

```

```

lemma of-char-eqI:
   $\langle c = d \rangle$  if  $\langle \text{of-char } c = \text{of-char } d \rangle$ 
  using that inj-of-char by (simp add: inj-eq)

```

```

lemma of-char-eq-iff [simp]:
   $\langle \text{of-char } c = \text{of-char } d \longleftrightarrow c = d \rangle$ 
  by (auto intro: of-char-eqI)

```

```

lemma of-char-of [simp]:
   $\langle \text{of-char } (\text{char-of } a) = a \text{ mod } 256 \rangle$ 
proof -
  have  $\langle [0..<8] = [0, \text{Suc } 0, 2, 3, 4, 5, 6, 7 :: \text{nat}] \rangle$ 
  by (simp add: upt-eq-Cons-conv)
  then have  $\langle [\text{bit } a \ 0, \text{bit } a \ 1, \text{bit } a \ 2, \text{bit } a \ 3, \text{bit } a \ 4, \text{bit } a \ 5, \text{bit } a \ 6, \text{bit } a \ 7] =$ 
  map (bit a) [0..<8]  $\rangle$ 
  by simp
  then have  $\langle \text{of-char } (\text{char-of } a) = \text{take-bit } 8 \ a \rangle$ 
  by (simp only: char-of-def of-char-def char.sel horner-sum-bit-eq-take-bit)
  then show ?thesis
  by (simp add: take-bit-eq-mod)
qed

```

```

lemma char-of-mod-256 [simp]:
   $\langle \text{char-of } (n \text{ mod } 256) = \text{char-of } n \rangle$ 
  by (rule of-char-eqI) simp

```

```

lemma of-char-mod-256 [simp]:
   $\langle \text{of-char } c \text{ mod } 256 = \text{of-char } c \rangle$ 
proof -
  have  $\langle \text{of-char } (\text{char-of } (\text{of-char } c)) \text{ mod } 256 = \text{of-char } (\text{char-of } (\text{of-char } c)) \rangle$ 
  by (simp only: of-char-of) simp
  then show ?thesis
  by simp
qed

```

```

lemma char-of-quasi-inj [simp]:
   $\langle \text{char-of } m = \text{char-of } n \longleftrightarrow m \text{ mod } 256 = n \text{ mod } 256 \rangle$  (is  $\langle ?P \longleftrightarrow ?Q \rangle$ )
proof
  assume ?Q
  then show ?P
  by (auto intro: of-char-eqI)
next
  assume ?P
  then have  $\langle \text{of-char } (\text{char-of } m) = \text{of-char } (\text{char-of } n) \rangle$ 

```



```

  by simp
  then show ?Q
  by simp
qed

```

```

lemma char-of-eq-iff:
  ⟨char-of n = c ⟷ take-bit 8 n = of-char c⟩
  by (auto intro: of-char-eqI simp add: take-bit-eq-mod)

```

```

lemma char-of-nat [simp]:
  ⟨char-of (of-nat n) = char-of n⟩
  by (simp add: char-of-def String.char-of-def drop-bit-of-nat bit-simps possible-bit-def)

```

end

```

lemma inj-on-char-of-nat [simp]:
  inj-on char-of {0::nat..<256}
  by (rule inj-onI) simp

```

```

lemma nat-of-char-less-256 [simp]:
  of-char c < (256 :: nat)
proof -
  have of-char c mod (256 :: nat) < 256
  by arith
  then show ?thesis by simp
qed

```

```

lemma range-nat-of-char:
  range of-char = {0::nat..<256}
proof (rule; rule)
  fix n :: nat
  assume n ∈ range of-char
  then show n ∈ {0..<256}
  by auto
next
  fix n :: nat
  assume n ∈ {0..<256}
  then have n = of-char (char-of n)
  by simp
  then show n ∈ range of-char
  by (rule range-eqI)
qed

```

```

lemma UNIV-char-of-nat:
  UNIV = char-of ‘ {0::nat..<256}
proof -
  have range (of-char :: char ⇒ nat) = of-char ‘ char-of ‘ {0::nat..<256}
  by (auto simp add: range-nat-of-char intro!: image-eqI)
  with inj-of-char [where ?'a = nat] show ?thesis

```

by (*simp add: inj-image-eq-iff*)
qed

lemma *card-UNIV-char*:
card (UNIV :: char set) = 256
 by (*auto simp add: UNIV-char-of-nat card-image*)

context
includes *lifting-syntax integer.lifting natural.lifting*
begin

lemma [*transfer-rule*]:
 $\langle \text{pcr-integer} \implies (=) \text{ char-of char-of} \rangle$
 by (*unfold char-of-def transfer-prover*)

lemma [*transfer-rule*]:
 $\langle (=) \implies \text{pcr-integer} \text{ of-char of-char} \rangle$
 by (*unfold of-char-def transfer-prover*)

lemma [*transfer-rule*]:
 $\langle \text{pcr-natural} \implies (=) \text{ char-of char-of} \rangle$
 by (*unfold char-of-def transfer-prover*)

lemma [*transfer-rule*]:
 $\langle (=) \implies \text{pcr-natural} \text{ of-char of-char} \rangle$
 by (*unfold of-char-def transfer-prover*)

end

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

lifting-update *natural.lifting*
lifting-forget *natural.lifting*

lemma *size-char-eq-0* [*simp, code*]:
 $\langle \text{size } c = 0 \rangle$ **for** $c :: \text{char}$
 by (*cases c simp*)

lemma *size'-char-eq-0* [*simp, code*]:
 $\langle \text{size-char } c = 0 \rangle$
 by (*cases c simp*)

syntax
 $\text{-Char} :: \text{str-position} \Rightarrow \text{char} \quad (\text{CHR } -)$
 $\text{-Char-ord} :: \text{num-const} \Rightarrow \text{char} \quad (\text{CHR } -)$

type-synonym *string = char list*

syntax

-String :: str-position \Rightarrow string (-)

ML-file \langle Tools/string-syntax.ML \rangle

instantiation char :: enum

begin

definition

```
Enum.enum = [
  CHR 0x00, CHR 0x01, CHR 0x02, CHR 0x03,
  CHR 0x04, CHR 0x05, CHR 0x06, CHR 0x07,
  CHR 0x08, CHR 0x09, CHR "\u2194", CHR 0x0B,
  CHR 0x0C, CHR 0x0D, CHR 0x0E, CHR 0x0F,
  CHR 0x10, CHR 0x11, CHR 0x12, CHR 0x13,
  CHR 0x14, CHR 0x15, CHR 0x16, CHR 0x17,
  CHR 0x18, CHR 0x19, CHR 0x1A, CHR 0x1B,
  CHR 0x1C, CHR 0x1D, CHR 0x1E, CHR 0x1F,
  CHR " ", CHR "\u0001", CHR 0x22, CHR "#",
  CHR "$", CHR "%", CHR "&", CHR 0x27,
  CHR "(", CHR ")", CHR "*", CHR "+",
  CHR ",", CHR "-", CHR ".", CHR "/",
  CHR "0", CHR "1", CHR "2", CHR "3",
  CHR "4", CHR "5", CHR "6", CHR "7",
  CHR "8", CHR "9", CHR ":", CHR ";",
  CHR "<", CHR "=", CHR ">", CHR "?",
  CHR "@", CHR "A", CHR "B", CHR "C",
  CHR "D", CHR "E", CHR "F", CHR "G",
  CHR "H", CHR "I", CHR "J", CHR "K",
  CHR "L", CHR "M", CHR "N", CHR "O",
  CHR "P", CHR "Q", CHR "R", CHR "S",
  CHR "T", CHR "U", CHR "V", CHR "W",
  CHR "X", CHR "Y", CHR "Z", CHR "[",
  CHR 0x5C, CHR "\u005D", CHR "~", CHR "-",
  CHR 0x60, CHR "a", CHR "b", CHR "c",
  CHR "d", CHR "e", CHR "f", CHR "g",
  CHR "h", CHR "i", CHR "j", CHR "k",
  CHR "l", CHR "m", CHR "n", CHR "o",
  CHR "p", CHR "q", CHR "r", CHR "s",
  CHR "t", CHR "u", CHR "v", CHR "w",
  CHR "x", CHR "y", CHR "z", CHR "{",
  CHR "\u007C", CHR "}", CHR "~", CHR 0x7F,
  CHR 0x80, CHR 0x81, CHR 0x82, CHR 0x83,
  CHR 0x84, CHR 0x85, CHR 0x86, CHR 0x87,
  CHR 0x88, CHR 0x89, CHR 0x8A, CHR 0x8B,
  CHR 0x8C, CHR 0x8D, CHR 0x8E, CHR 0x8F,
  CHR 0x90, CHR 0x91, CHR 0x92, CHR 0x93,
  CHR 0x94, CHR 0x95, CHR 0x96, CHR 0x97,
  CHR 0x98, CHR 0x99, CHR 0x9A, CHR 0x9B,
```

CHR 0x9C, CHR 0x9D, CHR 0x9E, CHR 0x9F,
 CHR 0xA0, CHR 0xA1, CHR 0xA2, CHR 0xA3,
 CHR 0xA4, CHR 0xA5, CHR 0xA6, CHR 0xA7,
 CHR 0xA8, CHR 0xA9, CHR 0xAA, CHR 0xAB,
 CHR 0xAC, CHR 0xAD, CHR 0xAE, CHR 0xAF,
 CHR 0xB0, CHR 0xB1, CHR 0xB2, CHR 0xB3,
 CHR 0xB4, CHR 0xB5, CHR 0xB6, CHR 0xB7,
 CHR 0xB8, CHR 0xB9, CHR 0xBA, CHR 0xBB,
 CHR 0xBC, CHR 0xBD, CHR 0xBE, CHR 0xBF,
 CHR 0xC0, CHR 0xC1, CHR 0xC2, CHR 0xC3,
 CHR 0xC4, CHR 0xC5, CHR 0xC6, CHR 0xC7,
 CHR 0xC8, CHR 0xC9, CHR 0xCA, CHR 0xCB,
 CHR 0xCC, CHR 0xCD, CHR 0xCE, CHR 0xCF,
 CHR 0xD0, CHR 0xD1, CHR 0xD2, CHR 0xD3,
 CHR 0xD4, CHR 0xD5, CHR 0xD6, CHR 0xD7,
 CHR 0xD8, CHR 0xD9, CHR 0xDA, CHR 0xDB,
 CHR 0xDC, CHR 0xDD, CHR 0xDE, CHR 0xDF,
 CHR 0xE0, CHR 0xE1, CHR 0xE2, CHR 0xE3,
 CHR 0xE4, CHR 0xE5, CHR 0xE6, CHR 0xE7,
 CHR 0xE8, CHR 0xE9, CHR 0xEA, CHR 0xEB,
 CHR 0xEC, CHR 0xED, CHR 0xEE, CHR 0xEF,
 CHR 0xF0, CHR 0xF1, CHR 0xF2, CHR 0xF3,
 CHR 0xF4, CHR 0xF5, CHR 0xF6, CHR 0xF7,
 CHR 0xF8, CHR 0xF9, CHR 0xFA, CHR 0xFB,
 CHR 0xFC, CHR 0xFD, CHR 0xFE, CHR 0xFF]

definition

$Enum.enum-all P \longleftrightarrow list-all P$ ($Enum.enum :: char list$)

definition

$Enum.enum-ex P \longleftrightarrow list-ex P$ ($Enum.enum :: char list$)

lemma *enum-char-unfold*:

$Enum.enum = map char-of [0..<256]$

proof –

have $map (of-char :: char \Rightarrow nat) Enum.enum = [0..<256]$

by (*simp add: enum-char-def of-char-def upt-conv-Cons-Cons numeral-2-eq-2 [symmetric]*)

then have $map char-of (map (of-char :: char \Rightarrow nat) Enum.enum) = map char-of [0..<256]$

by *simp*

then show *?thesis*

by *simp*

qed

instance proof

show $UNIV: UNIV = set (Enum.enum :: char list)$

by (*simp add: enum-char-unfold UNIV-char-of-nat atLeast0LessThan*)

show $distinct (Enum.enum :: char list)$

```

  by (auto simp add: enum-char-unfold distinct-map intro: inj-onI)
show  $\bigwedge P. Enum.enum-all P \longleftrightarrow Ball (UNIV :: char set) P$ 
  by (simp add: UNIV enum-all-char-def list-all-iff)
show  $\bigwedge P. Enum.enum-ex P \longleftrightarrow Bex (UNIV :: char set) P$ 
  by (simp add: UNIV enum-ex-char-def list-ex-iff)
qed

end

```

lemma *linorder-char*:

```

class.linorder ( $\lambda c d. of-char c \leq (of-char d :: nat)$ ) ( $\lambda c d. of-char c < (of-char d :: nat)$ )
  by standard auto

```

Optimized version for execution

```

definition char-of-integer :: integer  $\Rightarrow$  char
  where [code-abbrev]: char-of-integer = char-of

```

```

definition integer-of-char :: char  $\Rightarrow$  integer
  where [code-abbrev]: integer-of-char = of-char

```

lemma *char-of-integer-code* [code]:

```

char-of-integer k = (let
  (q0, b0) = bit-cut-integer k;
  (q1, b1) = bit-cut-integer q0;
  (q2, b2) = bit-cut-integer q1;
  (q3, b3) = bit-cut-integer q2;
  (q4, b4) = bit-cut-integer q3;
  (q5, b5) = bit-cut-integer q4;
  (q6, b6) = bit-cut-integer q5;
  (-, b7) = bit-cut-integer q6
in Char b0 b1 b2 b3 b4 b5 b6 b7)
  by (simp add: bit-cut-integer-def char-of-integer-def char-of-def div-mult2-numeral-eq
bit-iff-odd-drop-bit drop-bit-eq-div)

```

lemma *integer-of-char-code* [code]:

```

integer-of-char (Char b0 b1 b2 b3 b4 b5 b6 b7) =
  ((((((of-bool b7 * 2 + of-bool b6) * 2 +
  of-bool b5) * 2 + of-bool b4) * 2 +
  of-bool b3) * 2 + of-bool b2) * 2 +
  of-bool b1) * 2 + of-bool b0
  by (simp add: integer-of-char-def of-char-def)

```

74.2 Strings as dedicated type for target language code generation

74.2.1 Logical specification

```

context
begin

```

qualified definition *ascii-of* :: *char* ⇒ *char*

where *ascii-of* *c* = *Char* (*digit0* *c*) (*digit1* *c*) (*digit2* *c*) (*digit3* *c*) (*digit4* *c*) (*digit5* *c*) (*digit6* *c*) *False*

qualified lemma *ascii-of-Char* [*simp*]:

ascii-of (*Char* *b0* *b1* *b2* *b3* *b4* *b5* *b6* *b7*) = *Char* *b0* *b1* *b2* *b3* *b4* *b5* *b6* *False*
by (*simp* *add*: *ascii-of-def*)

qualified lemma *digit0-ascii-of-iff* [*simp*]:

digit0 (*String.ascii-of* *c*) ⇔ *digit0* *c*
by (*simp* *add*: *String.ascii-of-def*)

qualified lemma *digit1-ascii-of-iff* [*simp*]:

digit1 (*String.ascii-of* *c*) ⇔ *digit1* *c*
by (*simp* *add*: *String.ascii-of-def*)

qualified lemma *digit2-ascii-of-iff* [*simp*]:

digit2 (*String.ascii-of* *c*) ⇔ *digit2* *c*
by (*simp* *add*: *String.ascii-of-def*)

qualified lemma *digit3-ascii-of-iff* [*simp*]:

digit3 (*String.ascii-of* *c*) ⇔ *digit3* *c*
by (*simp* *add*: *String.ascii-of-def*)

qualified lemma *digit4-ascii-of-iff* [*simp*]:

digit4 (*String.ascii-of* *c*) ⇔ *digit4* *c*
by (*simp* *add*: *String.ascii-of-def*)

qualified lemma *digit5-ascii-of-iff* [*simp*]:

digit5 (*String.ascii-of* *c*) ⇔ *digit5* *c*
by (*simp* *add*: *String.ascii-of-def*)

qualified lemma *digit6-ascii-of-iff* [*simp*]:

digit6 (*String.ascii-of* *c*) ⇔ *digit6* *c*
by (*simp* *add*: *String.ascii-of-def*)

qualified lemma *not-digit7-ascii-of* [*simp*]:

¬ *digit7* (*ascii-of* *c*)
by (*simp* *add*: *ascii-of-def*)

qualified lemma *ascii-of-idem*:

ascii-of *c* = *c* **if** ¬ *digit7* *c*
using *that* **by** (*cases* *c*) *simp*

qualified typedef *literal* = {*cs*. ∀ *c* ∈ *set* *cs*. ¬ *digit7* *c*}

morphisms *explode* *Abs-literal*

proof

show [] ∈ {*cs*. ∀ *c* ∈ *set* *cs*. ¬ *digit7* *c*}

by *simp*
qed

qualified setup-lifting *type-definition-literal*

qualified lift-definition *implode :: string \Rightarrow literal*
is *map ascii-of*
by *auto*

qualified lemma *implode-explode-eq [simp]:*

String.implode (String.explode s) = s

proof *transfer*

fix *cs*

show *map ascii-of cs = cs* if $\forall c \in \text{set } cs. \neg \text{digit7 } c$

using *that*

by (*induction cs*) (*simp-all add: ascii-of-idem*)

qed

qualified lemma *explode-implode-eq [simp]:*

String.explode (String.implode cs) = map ascii-of cs

by *transfer rule*

end

context *unique-euclidean-semiring-with-bit-operations*

begin

context

begin

qualified lemma *char-of-ascii-of [simp]:*

$\langle \text{of-char } (String.\text{ascii-of } c) = \text{take-bit } 7 \text{ (of-char } c) \rangle$

by (*cases c*) (*simp only: String.ascii-of-Char of-char-Char take-bit-horner-sum-bit-eq, simp*)

qualified lemma *ascii-of-char-of:*

$\langle \text{String.ascii-of } (\text{char-of } a) = \text{char-of } (\text{take-bit } 7 \text{ } a) \rangle$

by (*simp add: char-of-def bit-simps*)

end

end

74.2.2 Syntactic representation

Logical ground representations for literals are:

1. 0 for the empty literal;

2. *Literal* $b0 \dots b6$ s for a literal starting with one character and continued by another literal.

Syntactic representations for literals are:

3. Printable text as string prefixed with *STR*;
4. A single ascii value as numerical hexadecimal value prefixed with *STR*.

instantiation *String.literal* :: zero
begin

context
begin

qualified lift-definition *zero-literal* :: *String.literal*
is *Nil*
by *simp*

instance ..

end

end

context
begin

qualified abbreviation (output) *empty-literal* :: *String.literal*
where *empty-literal* $\equiv 0$

qualified lift-definition *Literal* :: $bool \Rightarrow bool \Rightarrow bool \Rightarrow bool \Rightarrow bool \Rightarrow bool \Rightarrow bool \Rightarrow String.literal \Rightarrow String.literal$
is $\lambda b0 b1 b2 b3 b4 b5 b6 cs. Char\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ False\ \# cs$
by *auto*

qualified lemma *Literal-eq-iff* [*simp*]:

$$\begin{aligned} & Literal\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ s = Literal\ c0\ c1\ c2\ c3\ c4\ c5\ c6\ t \\ & \iff (b0 \iff c0) \wedge (b1 \iff c1) \wedge (b2 \iff c2) \wedge (b3 \iff c3) \\ & \quad \wedge (b4 \iff c4) \wedge (b5 \iff c5) \wedge (b6 \iff c6) \wedge s = t \end{aligned}$$

by *transfer simp*

qualified lemma *empty-neq-Literal* [*simp*]:

$$empty-literal \neq Literal\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ s$$

by *transfer simp*

qualified lemma *Literal-neq-empty* [*simp*]:

$$Literal\ b0\ b1\ b2\ b3\ b4\ b5\ b6\ s \neq empty-literal$$

by *transfer simp*

end

code-datatype *0* :: *String.literal* *String.Literal*

syntax

-*Literal* :: *str-position* \Rightarrow *String.literal* (STR -)
 -*Ascii* :: *num-const* \Rightarrow *String.literal* (STR -)

ML-file \langle Tools/literal.ML \rangle

74.2.3 Operations

instantiation *String.literal* :: *plus*

begin

context

begin

qualified lift-definition *plus-literal* :: *String.literal* \Rightarrow *String.literal* \Rightarrow *String.literal*

is (@)

by *auto*

instance ..

end

end

instance *String.literal* :: *monoid-add*

by (*standard*; *transfer*) *simp-all*

instantiation *String.literal* :: *size*

begin

context

includes *literal.lifting*

begin

lift-definition *size-literal* :: *String.literal* \Rightarrow *nat*

is *length* .

end

instance ..

end

instantiation *String.literal* :: *equal*

begin

context

begin

qualified lift-definition *equal-literal* :: *String.literal* \Rightarrow *String.literal* \Rightarrow *bool*
is *HOL.equal* .

instance

by (*standard*; *transfer*) (*simp add: equal*)

end

end

instantiation *String.literal* :: *linorder*

begin

context

begin

qualified lift-definition *less-eq-literal* :: *String.literal* \Rightarrow *String.literal* \Rightarrow *bool*
is *ord.lexordp-eq* ($\lambda c d. \text{of-char } c < (\text{of-char } d :: \text{nat})$)
 .

qualified lift-definition *less-literal* :: *String.literal* \Rightarrow *String.literal* \Rightarrow *bool*
is *ord.lexordp* ($\lambda c d. \text{of-char } c < (\text{of-char } d :: \text{nat})$)
 .

instance proof –

from *linorder-char* **interpret** *linorder ord.lexordp-eq* ($\lambda c d. \text{of-char } c < (\text{of-char } d :: \text{nat})$)

ord.lexordp ($\lambda c d. \text{of-char } c < (\text{of-char } d :: \text{nat})$) :: *string* \Rightarrow *string* \Rightarrow *bool*

by (*rule linorder.lexordp-linorder*)

show *PROP ?thesis*

by (*standard*; *transfer*) (*simp-all add: less-le-not-le linear*)

qed

end

end

lemma *infinite-literal*:

infinite (*UNIV* :: *String.literal* *set*)

proof –

define *S* **where** *S* = *range* ($\lambda n. \text{replicate } n \text{ CHR } "A"$)

have *inj-on String.implode S*

proof (*rule inj-onI*)

fix *cs ds*

assume *String.implode cs = String.implode ds*

then have $String.explode (String.implode cs) = String.explode (String.implode ds)$
by *simp*
moreover assume $cs \in S$ **and** $ds \in S$
ultimately show $cs = ds$
by (*auto simp add: S-def*)
qed
moreover have *infinite S*
by (*auto simp add: S-def dest: finite-range-imageI [of - length]*)
ultimately have *infinite (String.implode ‘ S)*
by (*simp add: finite-image-iff*)
then show *?thesis*
by (*auto intro: finite-subset*)
qed

74.2.4 Executable conversions

context

begin

qualified lift-definition *asciis-of-literal* :: $String.literal \Rightarrow integer\ list$
is map *of-char*

.

qualified lemma *asciis-of-zero* [*simp, code*]:

asciis-of-literal 0 = []

by *transfer simp*

qualified lemma *asciis-of-Literal* [*simp, code*]:

asciis-of-literal (*String.Literal* b0 b1 b2 b3 b4 b5 b6 s) =

of-char (*Char* b0 b1 b2 b3 b4 b5 b6 *False*) # *asciis-of-literal* s

by *transfer simp*

qualified lift-definition *literal-of-asciis* :: $integer\ list \Rightarrow String.literal$

is map (*String.asciï-of* \circ *char-of*)

by *auto*

qualified lemma *literal-of-asciis-Nil* [*simp, code*]:

literal-of-asciis [] = 0

by *transfer simp*

qualified lemma *literal-of-asciis-Cons* [*simp, code*]:

literal-of-asciis (k # ks) = (*case char-of* k

of Char b0 b1 b2 b3 b4 b5 b6 b7 \Rightarrow *String.Literal* b0 b1 b2 b3 b4 b5 b6

(*literal-of-asciis* ks))

by (*simp add: char-of-def*) (*transfer, simp add: char-of-def*)

qualified lemma *literal-of-asciis-of-literal* [*simp*]:

literal-of-asciis (*asciis-of-literal* s) = s

proof *transfer*

fix *cs*

assume $\forall c \in \text{set } cs. \neg \text{digit7 } c$

then show $\text{map } (\text{String.ascii-of} \circ \text{char-of}) (\text{map of-char } cs) = cs$

by (*induction cs*) (*simp-all add: String.ascii-of-idem*)

qed

qualified lemma *explode-code* [*code*]:

$\text{String.explode } s = \text{map char-of } (\text{asciis-of-literal } s)$

by *transfer simp*

qualified lemma *implode-code* [*code*]:

$\text{String.implode } cs = \text{literal-of-asciis } (\text{map of-char } cs)$

by *transfer simp*

qualified lemma *equal-literal* [*code*]:

$\text{HOL.equal } (\text{String.Literal } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ s)$

$(\text{String.Literal } a0 \ a1 \ a2 \ a3 \ a4 \ a5 \ a6 \ r)$

$\longleftrightarrow (b0 \longleftrightarrow a0) \wedge (b1 \longleftrightarrow a1) \wedge (b2 \longleftrightarrow a2) \wedge (b3 \longleftrightarrow a3)$

$\wedge (b4 \longleftrightarrow a4) \wedge (b5 \longleftrightarrow a5) \wedge (b6 \longleftrightarrow a6) \wedge (s = r)$

by (*simp add: equal*)

end

74.2.5 Technical code generation setup

Alternative constructor for generated computations

context

begin

qualified definition $\text{Literal}' :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$

$\Rightarrow \text{String.literal} \Rightarrow \text{String.literal}$

where [*simp*]: $\text{Literal}' = \text{String.Literal}$

lemma [*code*]:

$\langle \text{Literal}' \ b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ s = \text{String.literal-of-asciis}$

$[\text{foldr } (\lambda b \ k. \text{of-bool } b + k * 2) [b0, b1, b2, b3, b4, b5, b6] 0] + s \rangle$

proof –

have $\langle \text{foldr } (\lambda b \ k. \text{of-bool } b + k * 2) [b0, b1, b2, b3, b4, b5, b6] 0 = \text{of-char}$
 $(\text{Char } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ \text{False}) \rangle$

by *simp*

moreover have $\langle \text{Literal}' \ b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ s = \text{String.literal-of-asciis}$

$[\text{of-char } (\text{Char } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ \text{False})] + s \rangle$

by (*unfold Literal'-def*) (*transfer, simp only: list.simps comp-apply char-of-char, simp*)

ultimately show *?thesis*

by *simp*

qed

```

lemma [code-computation-unfold]:
  String.Literal = Literal'
  by simp

end

code-reserved SML string String Char Str-Literal
code-reserved OCaml string String Char Str-Literal
code-reserved Haskell Prelude
code-reserved Scala string

code-identifier
code-module String  $\rightarrow$ 
  (SML) Str and (OCaml) Str and (Haskell) Str and (Scala) Str

code-printing
type-constructor String.literal  $\rightarrow$ 
  (SML) string
  and (OCaml) string
  and (Haskell) String
  and (Scala) String
| constant STR ''''  $\rightarrow$ 
  (SML)
  and (OCaml)
  and (Haskell)
  and (Scala)

setup  $\langle$ 
  fold Literal.add-code [SML, OCaml, Haskell, Scala]
 $\rangle$ 

code-printing
code-module Str-Literal  $\rightarrow$ 
  (SML)  $\langle$ structure Str-Literal =
struct

fun map f [] = []
  | map f (x :: xs) = f x :: map f xs; (* deliberate clone not relying on List.- module *)

fun check-ascii (k : IntInf.int) =
  if  $0 \leq k$  andalso  $k < 128$ 
  then k
  else raise Fail Non-ASCII character in literal;

val char-of-ascii = Char.chr o IntInf.toInt o check-ascii;

val ascii-of-char = check-ascii o IntInf.fromInt o Char.ord;

```

```

val literal-of-asciis = String.implode o map char-of-ascii;

val asciis-of-literal = map ascii-of-char o String.explode;

end;> for constant String.literal-of-asciis String.asciis-of-literal
  and (OCaml) <module Str-Literal =
  struct

let implode f xs =
  let rec length xs = match xs with
    [] -> 0
  | x :: xs -> 1 + length xs in
  let rec nth xs n = match xs with
    (x :: xs) -> if n <= 0 then x else nth xs (n - 1)
  in String.init (length xs) (fun n -> f (nth xs n));

let explode f s =
  let rec map-range f n =
    if n <= 0 then [] else map-range f (n - 1) @ [f n]
  in map-range (fun n -> f (String.get s n)) (String.length s);

let z-128 = Z.of-int 128;;

let check-ascii (k : Z.t) =
  if Z.leq Z.zero k && Z.lt k z-128
  then k
  else failwith Non-ASCII character in literal;;

let char-of-ascii k = Char.chr (Z.to-int (check-ascii k));;

let ascii-of-char c = check-ascii (Z.of-int (Char.code c));;

let literal-of-asciis ks = implode char-of-ascii ks;;

let asciis-of-literal s = explode ascii-of-char s;;

end;> for constant String.literal-of-asciis String.asciis-of-literal
| constant (+) :: String.literal => String.literal => String.literal ->
  (SML) infixl 18 ^
  and (OCaml) infixr 6 ^
  and (Haskell) infixr 5 ++
  and (Scala) infixl 7 +
| constant String.literal-of-asciis ->
  (SML) Str'-Literal.literal'-of'-asciis
  and (OCaml) Str'-Literal.literal'-of'-asciis
  and (Haskell) map/ (let chr k | (0 <= k && k < 128) = Prelude.toEnum k ::
Prelude.Char in chr . Prelude.fromInteger)
  and (Scala) / ++/ -.map((k: BigInt) => if (BigInt(0) <= k && k < Big-
Int(128)) k.charValue else sys.error(Non-ASCII character in literal))

```

```

| constant String.asciis-of-literal  $\rightarrow$ 
  (SML) Str'-Literal.asciis'-of'-literal
  and (OCaml) Str'-Literal.asciis'-of'-literal
  and (Haskell) map/ (let ord k | (k < 128) = Prelude.toInteger k in ord .
(Prelude.fromEnum :: Prelude.Char  $\rightarrow$  Prelude.Int))
  and (Scala) !(-.toList.map(c => { val k: Int = c.toInt; if (k < 128) BigInt(k)
else sys.error(Non-ASCII character in literal) }))
| class-instance String.literal :: equal  $\rightarrow$ 
  (Haskell)  $-$ 
| constant HOL.equal :: String.literal  $\Rightarrow$  String.literal  $\Rightarrow$  bool  $\rightarrow$ 
  (SML) !((- : string) = -)
  and (OCaml) !((- : string) = -)
  and (Haskell) infix 4 ==
  and (Scala) infixl 5 ==
| constant ( $\leq$ ) :: String.literal  $\Rightarrow$  String.literal  $\Rightarrow$  bool  $\rightarrow$ 
  (SML) !((- : string) <= -)
  and (OCaml) !((- : string) <= -)
  and (Haskell) infix 4 <=
   $-$  Order operations for String.literal work in Haskell only if no type class
instance needs to be generated, because String = [Char] in Haskell and char list
need not have the same order as String.literal.
  and (Scala) infixl 4 <=
  and (Eval) infixl 6 <=
| constant (<) :: String.literal  $\Rightarrow$  String.literal  $\Rightarrow$  bool  $\rightarrow$ 
  (SML) !((- : string) < -)
  and (OCaml) !((- : string) < -)
  and (Haskell) infix 4 <
  and (Scala) infixl 4 <
  and (Eval) infixl 6 <

```

74.2.6 Code generation utility

```

setup  $\langle$ Sign.map-naming (Name-Space.mandatory-path Code) $\rangle$ 

```

```

definition abort :: String.literal  $\Rightarrow$  (unit  $\Rightarrow$  'a)  $\Rightarrow$  'a
  where [simp]: abort - f = f ()

```

```

declare [[code drop: Code.abort]]

```

```

lemma abort-cong:

```

```

  msg = msg'  $\implies$  Code.abort msg f = Code.abort msg' f
  by simp

```

```

setup  $\langle$ Sign.map-naming Name-Space.parent-path $\rangle$ 

```

```

setup  $\langle$ Code-Simp.map-ss (Simplifier.add-cong @{thm Code.abort-cong}) $\rangle$ 

```

```

code-printing

```

```

  constant Code.abort  $\rightarrow$ 

```

```

(SML) !(raise/ Fail/ -)
and (OCaml) failwith
and (Haskell) !(error/ ::/ forall a./ String -> (() -> a) -> a)
and (Scala) !{/ sys.error((-);/ ((-).apply())/ }

```

74.2.7 Finally

```

lifting-update literal.lifting
lifting-forget literal.lifting

```

```
end
```

75 Reflecting Pure types into HOL

```

theory Typerep
imports String
begin

```

```
datatype typerep = Typerep String.literal typerep list
```

```

class typerep =
  fixes typerep :: 'a itself ⇒ typerep
begin

```

```

definition typerep-of :: 'a ⇒ typerep where
  [simp]: typerep-of x = typerep TYPE('a)

```

```
end
```

```

syntax
  -TYPEREP :: type => logic ((1TYPEREP/(1'(-))))

```

```
parse-translation <
```

```

  let
    fun typerep-tr (*-TYPEREP*) [ty] =
      Syntax.const const-syntax <typerep> $
        (Syntax.const syntax-const <-constrain> $ Syntax.const const-syn-
tax <Pure.type> $
          (Syntax.const type-syntax <itself> $ ty))
    | typerep-tr (*-TYPEREP*) ts = raise TERM (typerep-tr, ts);
  in [(syntax-const <-TYPEREP>, K typerep-tr)] end
>

```

```
typed-print-translation <
```

```

  let
    fun typerep-tr' ctxt (*typerep*) Type <fun Type <itself T> ->
      (Const (const-syntax <Pure.type>, -) :: ts) =
      Term.list-comb
        (Syntax.const syntax-const <-TYPEREP> $ Syntax-Phases.term-of-typ

```



```

ctxt T, ts)
  | typerep-tr' - T ts = raise Match;
  in [(const-syntax <typerep>, typerep-tr')] end
>

setup <
let

fun add-typerep tyco thy =
  let
    val sorts = replicate (Sign.arity-number thy tyco) sort <typerep>;
    val vs = Name.invent-names Name.context 'a sorts;
    val ty = Type (tyco, map TFree vs);
    val lhs = Const <typerep ty> $ Free (T, Term.itselfT ty);
    val rhs = Const <Typerep> $ HOLogic.mk-literal tyco
      $ HLOLogic.mk-list Type <typerep> (map (HOLogic.mk-typerep o TFree) vs);
    val eq = HLOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
  in
    thy
    |> Class.instantiation ([tyco], vs, sort <typerep>)
    |> '(fn lthy => Syntax.check-term lthy eq)
    |-> (fn eq => Specification.definition NONE [] [] (Binding.empty-atts, eq))
    |> snd
    |> Class.prove-instantiation-exit (fn ctxt => Class.intro-classes-tac ctxt [])
  end;

fun ensure-typerep tyco thy =
  if not (Sorts.has-instance (Sign.classes-of thy) tyco sort <typerep>)
    andalso Sorts.has-instance (Sign.classes-of thy) tyco sort <type>
  then add-typerep tyco thy else thy;

in

add-typerep type-name <fun>
#> Typedef.interpretation (Local-Theory.background-theory o ensure-typerep)
#> Code.type-interpretation ensure-typerep

end
>

lemma [code]:
  HOL.equal (Typerep tyco1 tys1) (Typerep tyco2 tys2) <=> HOL.equal tyco1 tyco2
  ^ list-all2 HOL.equal tys1 tys2
by (auto simp add: eq-equal [symmetric] list-all2-eq [symmetric])

lemma [code nbe]:
  HOL.equal (x :: typerep) x <=> True
by (fact equal-refl)

```

code-printing

type-constructor *typerep* \rightarrow (*Eval*) *Term.typ*
| **constant** *Typerep* \rightarrow (*Eval*) *Term.Type* / (-, -)

code-reserved *Eval Term*

hide-const (**open**) *typerep Typerep*

end

76 Predicates as enumerations

theory *Predicate*

imports *String*

begin

76.1 The type of predicate enumerations (a monad)

datatype (*plugins only: extraction*) (*dead 'a*) *pred* = *Pred* (*eval*: 'a \Rightarrow bool)

lemma *pred-eqI*:

$(\bigwedge w. \text{eval } P \ w \longleftrightarrow \text{eval } Q \ w) \implies P = Q$
by (*cases P, cases Q*) (*auto simp add: fun-eq-iff*)

lemma *pred-eq-iff*:

$P = Q \implies (\bigwedge w. \text{eval } P \ w \longleftrightarrow \text{eval } Q \ w)$
by (*simp add: pred-eqI*)

instantiation *pred* :: (*type*) *complete-lattice*

begin

definition

$P \leq Q \longleftrightarrow \text{eval } P \leq \text{eval } Q$

definition

$P < Q \longleftrightarrow \text{eval } P < \text{eval } Q$

definition

$\perp = \text{Pred } \perp$

lemma *eval-bot* [*simp*]:

$\text{eval } \perp = \perp$
by (*simp add: bot-pred-def*)

definition

$\top = \text{Pred } \top$

lemma *eval-top* [*simp*]:

$\text{eval } \top = \top$

by (*simp add: top-pred-def*)

definition

$P \sqcap Q = \text{Pred } (\text{eval } P \sqcap \text{eval } Q)$

lemma *eval-inf* [*simp*]:

$\text{eval } (P \sqcap Q) = \text{eval } P \sqcap \text{eval } Q$

by (*simp add: inf-pred-def*)

definition

$P \sqcup Q = \text{Pred } (\text{eval } P \sqcup \text{eval } Q)$

lemma *eval-sup* [*simp*]:

$\text{eval } (P \sqcup Q) = \text{eval } P \sqcup \text{eval } Q$

by (*simp add: sup-pred-def*)

definition

$\prod A = \text{Pred } (\prod (\text{eval } ' A))$

lemma *eval-Inf* [*simp*]:

$\text{eval } (\prod A) = \prod (\text{eval } ' A)$

by (*simp add: Inf-pred-def*)

definition

$\bigsqcup A = \text{Pred } (\bigsqcup (\text{eval } ' A))$

lemma *eval-Sup* [*simp*]:

$\text{eval } (\bigsqcup A) = \bigsqcup (\text{eval } ' A)$

by (*simp add: Sup-pred-def*)

instance proof

qed (*auto intro!: pred-eqI simp add: less-eq-pred-def less-pred-def le-fun-def less-fun-def*)

end

lemma *eval-INF* [*simp*]:

$\text{eval } (\prod (f ' A)) = \prod ((\text{eval } \circ f) ' A)$

by (*simp add: image-comp*)

lemma *eval-SUP* [*simp*]:

$\text{eval } (\bigsqcup (f ' A)) = \bigsqcup ((\text{eval } \circ f) ' A)$

by (*simp add: image-comp*)

instantiation *pred* :: (*type*) *complete-boolean-algebra*

begin

definition

$- P = \text{Pred } (- \text{eval } P)$

lemma *eval-compl* [*simp*]:
 $eval (- P) = - eval P$
by (*simp add: uminus-pred-def*)

definition
 $P - Q = Pred (eval P - eval Q)$

lemma *eval-minus* [*simp*]:
 $eval (P - Q) = eval P - eval Q$
by (*simp add: minus-pred-def*)

instance proof

fix $A :: 'a \text{ pred set set}$
show $\sqcap (Sup \text{ ' } A) \leq \sqcup (Inf \text{ ' } \{f \text{ ' } A \mid f. \forall Y \in A. f Y \in Y\})$
proof (*simp add: less-eq-pred-def Sup-fun-def Inf-fun-def, safe*)
fix w
assume $A: \forall x \in A. \exists f \in x. eval f w$
define F **where** $F = (\lambda x. SOME f. f \in x \wedge eval f w)$
have [*simp*]: $(\forall f \in (F \text{ ' } A). eval f w)$
by (*metis (no-types, lifting) A F-def image-iff some-eq-ex*)
have $(\exists f. F \text{ ' } A = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)) \wedge (\forall f \in (F \text{ ' } A). eval f w)$
using A **by** (*simp, metis (no-types, lifting) F-def someI+*)
from this show $\exists x. (\exists f. x = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)) \wedge (\forall f \in x. eval f w)$
by (*rule exI [of - F \text{ ' } A]*)
qed
qed (*auto intro!: pred-eqI*)

end

definition *single* :: $'a \Rightarrow 'a \text{ pred}$ **where**
 $single x = Pred ((=) x)$

lemma *eval-single* [*simp*]:
 $eval (single x) = (=) x$
by (*simp add: single-def*)

definition *bind* :: $'a \text{ pred} \Rightarrow ('a \Rightarrow 'b \text{ pred}) \Rightarrow 'b \text{ pred}$ (**infixl** $\gg=$ 70) **where**
 $P \gg= f = (\sqcup (f \text{ ' } \{x. eval P x\}))$

lemma *eval-bind* [*simp*]:
 $eval (P \gg= f) = eval (\sqcup (f \text{ ' } \{x. eval P x\}))$
by (*simp add: bind-def*)

lemma *bind-bind*:
 $(P \gg= Q) \gg= R = P \gg= (\lambda x. Q x \gg= R)$
by (*rule pred-eqI*) *auto*

lemma *bind-single*:
 $P \gg= single = P$

by (rule pred-eqI) auto

lemma single-bind:

$single\ x \gg= P = P\ x$

by (rule pred-eqI) auto

lemma bottom-bind:

$\perp \gg= P = \perp$

by (rule pred-eqI) auto

lemma sup-bind:

$(P \sqcup Q) \gg= R = P \gg= R \sqcup Q \gg= R$

by (rule pred-eqI) auto

lemma Sup-bind:

$(\sqcup A \gg= f) = \sqcup ((\lambda x. x \gg= f) \text{ ` } A)$

by (rule pred-eqI) auto

lemma pred-iffI:

assumes $\bigwedge x. eval\ A\ x \implies eval\ B\ x$

and $\bigwedge x. eval\ B\ x \implies eval\ A\ x$

shows $A = B$

using *assms* **by** (auto intro: pred-eqI)

lemma singleI: $eval\ (single\ x)\ x$

by *simp*

lemma singleI-unit: $eval\ (single\ ())\ x$

by *simp*

lemma singleE: $eval\ (single\ x)\ y \implies (y = x \implies P) \implies P$

by *simp*

lemma singleE': $eval\ (single\ x)\ y \implies (x = y \implies P) \implies P$

by *simp*

lemma bindI: $eval\ P\ x \implies eval\ (Q\ x)\ y \implies eval\ (P \gg= Q)\ y$

by *auto*

lemma bindE: $eval\ (R \gg= Q)\ y \implies (\bigwedge x. eval\ R\ x \implies eval\ (Q\ x)\ y \implies P) \implies P$

by *auto*

lemma botE: $eval\ \perp\ x \implies P$

by *auto*

lemma supI1: $eval\ A\ x \implies eval\ (A \sqcup B)\ x$

by *auto*

lemma *supI2*: $eval\ B\ x \implies eval\ (A \sqcup B)\ x$
by *auto*

lemma *supE*: $eval\ (A \sqcup B)\ x \implies (eval\ A\ x \implies P) \implies (eval\ B\ x \implies P) \implies P$
by *auto*

lemma *single-not-bot* [*simp*]:
 $single\ x \neq \perp$
by (*auto simp add: single-def bot-pred-def fun-eq-iff*)

lemma *not-bot*:
assumes $A \neq \perp$
obtains x **where** $eval\ A\ x$
using *assms* **by** (*cases A*) (*auto simp add: bot-pred-def*)

76.2 Emptiness check and definite choice

definition *is-empty* :: $'a\ pred \Rightarrow bool$ **where**
 $is_empty\ A \longleftrightarrow A = \perp$

lemma *is-empty-bot*:
 $is_empty\ \perp$
by (*simp add: is-empty-def*)

lemma *not-is-empty-single*:
 $\neg is_empty\ (single\ x)$
by (*auto simp add: is-empty-def single-def bot-pred-def fun-eq-iff*)

lemma *is-empty-sup*:
 $is_empty\ (A \sqcup B) \longleftrightarrow is_empty\ A \wedge is_empty\ B$
by (*auto simp add: is-empty-def*)

definition *singleton* :: $(unit \Rightarrow 'a) \Rightarrow 'a\ pred \Rightarrow 'a$ **where**
 $singleton\ default\ A = (if\ \exists!x. eval\ A\ x\ then\ THE\ x. eval\ A\ x\ else\ default\ ())$ **for**
 $default$

lemma *singleton-eqI*:
 $\exists!x. eval\ A\ x \implies eval\ A\ x \implies singleton\ default\ A = x$ **for** $default$
by (*auto simp add: singleton-def*)

lemma *eval-singletonI*:
 $\exists!x. eval\ A\ x \implies eval\ A\ (singleton\ default\ A)$ **for** $default$

proof –

assume *assm*: $\exists!x. eval\ A\ x$

then obtain x **where** $x: eval\ A\ x$..

with *assm* **have** $singleton\ default\ A = x$ **by** (*rule singleton-eqI*)

with x **show** *?thesis* **by** *simp*

qed

lemma *single-singleton*:

$\exists!x. \text{eval } A \ x \implies \text{single } (\text{singleton default } A) = A$ **for default**

proof –

assume *assm*: $\exists!x. \text{eval } A \ x$

then have $\text{eval } A \ (\text{singleton default } A)$

by (*rule eval-singletonI*)

moreover from *assm* **have** $\bigwedge x. \text{eval } A \ x \implies \text{singleton default } A = x$

by (*rule singleton-eqI*)

ultimately have $\text{eval } (\text{single } (\text{singleton default } A)) = \text{eval } A$

by (*simp (no-asm-use) add: single-def fun-eq-iff*) *blast*

then have $\bigwedge x. \text{eval } (\text{single } (\text{singleton default } A)) \ x = \text{eval } A \ x$

by *simp*

then show *?thesis* **by** (*rule pred-eqI*)

qed

lemma *singleton-undefinedI*:

$\neg (\exists!x. \text{eval } A \ x) \implies \text{singleton default } A = \text{default } ()$ **for default**

by (*simp add: singleton-def*)

lemma *singleton-bot*:

$\text{singleton default } \perp = \text{default } ()$ **for default**

by (*auto simp add: bot-pred-def intro: singleton-undefinedI*)

lemma *singleton-single*:

$\text{singleton default } (\text{single } x) = x$ **for default**

by (*auto simp add: intro: singleton-eqI singleI elim: singleE*)

lemma *singleton-sup-single-single*:

$\text{singleton default } (\text{single } x \sqcup \text{single } y) = (\text{if } x = y \text{ then } x \text{ else default } ())$ **for default**

proof (*cases x = y*)

case *True* **then show** *?thesis* **by** (*simp add: singleton-single*)

next

case *False*

have $\text{eval } (\text{single } x \sqcup \text{single } y) \ x$

and $\text{eval } (\text{single } x \sqcup \text{single } y) \ y$

by (*auto intro: supI1 supI2 singleI*)

with *False* **have** $\neg (\exists!z. \text{eval } (\text{single } x \sqcup \text{single } y) \ z)$

by *blast*

then have $\text{singleton default } (\text{single } x \sqcup \text{single } y) = \text{default } ()$

by (*rule singleton-undefinedI*)

with *False* **show** *?thesis* **by** *simp*

qed

lemma *singleton-sup-aux*:

$\text{singleton default } (A \sqcup B) = (\text{if } A = \perp \text{ then singleton default } B$

$\text{else if } B = \perp \text{ then singleton default } A$

$\text{else singleton default}$

$(\text{single } (\text{singleton default } A) \sqcup \text{single } (\text{singleton default } B)))$ **for default**

```

proof (cases (∃!x. eval A x) ∧ (∃!y. eval B y))
  case True then show ?thesis by (simp add: single-singleton)
next
  case False
  from False have A-or-B:
    singleton default A = default () ∨ singleton default B = default ()
  by (auto intro!: singleton-undefinedI)
  then have rhs: singleton default
    (single (singleton default A) ⊔ single (singleton default B)) = default ()
  by (auto simp add: singleton-sup-single-single singleton-single)
  from False have not-unique:
    ¬ (∃!x. eval A x) ∨ ¬ (∃!y. eval B y) by simp
  show ?thesis proof (cases A ≠ ⊥ ∧ B ≠ ⊥)
    case True
    then obtain a b where a: eval A a and b: eval B b
    by (blast elim: not-bot)
    with True not-unique have ¬ (∃!x. eval (A ⊔ B) x)
    by (auto simp add: sup-pred-def bot-pred-def)
    then have singleton default (A ⊔ B) = default () by (rule singleton-undefinedI)
    with True rhs show ?thesis by simp
  next
  case False then show ?thesis by auto
qed
qed

```

```

lemma singleton-sup:
  singleton default (A ⊔ B) = (if A = ⊥ then singleton default B
    else if B = ⊥ then singleton default A
    else if singleton default A = singleton default B then singleton default A else
  default ()) for default
  using singleton-sup-aux [of default A B] by (simp only: singleton-sup-single-single)

```

76.3 Derived operations

```

definition if-pred :: bool ⇒ unit pred where
  if-pred-eq: if-pred b = (if b then single () else ⊥)

```

```

definition holds :: unit pred ⇒ bool where
  holds-eq: holds P = eval P ()

```

```

definition not-pred :: unit pred ⇒ unit pred where
  not-pred-eq: not-pred P = (if eval P () then ⊥ else single ())

```

```

lemma if-predI: P ⇒ eval (if-pred P) ()
  unfolding if-pred-eq by (auto intro: singleI)

```

```

lemma if-predE: eval (if-pred b) x ⇒ (b ⇒ x = () ⇒ P) ⇒ P
  unfolding if-pred-eq by (cases b) (auto elim: botE)

```


lemma *not-predI*: $\neg P \implies \text{eval } (\text{not-pred } (\text{Pred } (\lambda u. P))) ()$
unfolding *not-pred-eq* **by** (*auto intro: singleI*)

lemma *not-predI'*: $\neg \text{eval } P () \implies \text{eval } (\text{not-pred } P) ()$
unfolding *not-pred-eq* **by** (*auto intro: singleI*)

lemma *not-predE*: $\text{eval } (\text{not-pred } (\text{Pred } (\lambda u. P))) x \implies (\neg P \implies \text{thesis}) \implies \text{thesis}$
unfolding *not-pred-eq*
by (*auto split: if-split-asm elim: botE*)

lemma *not-predE'*: $\text{eval } (\text{not-pred } P) x \implies (\neg \text{eval } P x \implies \text{thesis}) \implies \text{thesis}$
unfolding *not-pred-eq*
by (*auto split: if-split-asm elim: botE*)

lemma $f () = \text{False} \vee f () = \text{True}$
by *simp*

lemma *closure-of-bool-cases* [*no-atp*]:
fixes $f :: \text{unit} \Rightarrow \text{bool}$
assumes $f = (\lambda u. \text{False}) \implies P f$
assumes $f = (\lambda u. \text{True}) \implies P f$
shows $P f$
proof –
have $f = (\lambda u. \text{False}) \vee f = (\lambda u. \text{True})$
apply (*cases f ()*)
apply (*rule disjI2*)
apply (*rule ext*)
apply (*simp add: unit-eq*)
apply (*rule disjI1*)
apply (*rule ext*)
apply (*simp add: unit-eq*)
done
from *this assms* **show** *?thesis* **by** *blast*
qed

lemma *unit-pred-cases*:
assumes $P \perp$
assumes $P (\text{single } ())$
shows $P Q$
using *assms* **unfolding** *bot-pred-def bot-fun-def bot-bool-def empty-def single-def*
proof (*cases Q*)
fix f
assume $P (\text{Pred } (\lambda u. \text{False})) P (\text{Pred } (\lambda u. () = u))$
then have $P (\text{Pred } f)$
by (*cases - f rule: closure-of-bool-cases*) *simp-all*
moreover assume $Q = \text{Pred } f$
ultimately show $P Q$ **by** *simp*
qed

lemma *holds-if-pred*:

holds (if-pred b) = b

unfolding *if-pred-eq holds-eq*

by (*cases b*) (*auto intro: singleI elim: botE*)

lemma *if-pred-holds*:

if-pred (holds P) = P

unfolding *if-pred-eq holds-eq*

by (*rule unit-pred-cases*) (*auto intro: singleI elim: botE*)

lemma *is-empty-holds*:

is-empty P \longleftrightarrow \neg holds P

unfolding *is-empty-def holds-eq*

by (*rule unit-pred-cases*) (*auto elim: botE intro: singleI*)

definition *map* :: (*'a \Rightarrow 'b*) \Rightarrow *'a pred \Rightarrow 'b pred* **where**

map f P = P \gg (single \circ f)

lemma *eval-map [simp]*:

eval (map f P) = (\sqcup $x \in \{x. \text{eval } P \ x\}. (\lambda y. f \ x = y)$)

by (*simp add: map-def comp-def image-comp*)

functor *map*: *map*

by (*rule ext, rule pred-eqI, auto*)⁺

76.4 Implementation

datatype (*plugins only: code extraction*) (*dead 'a*) *seq* =

Empty

| *Insert 'a 'a pred*

| *Join 'a pred 'a seq*

primrec *pred-of-seq* :: *'a seq \Rightarrow 'a pred* **where**

pred-of-seq Empty = \perp

| *pred-of-seq (Insert x P) = single x \sqcup P*

| *pred-of-seq (Join P xq) = P \sqcup pred-of-seq xq*

definition *Seq* :: (*unit \Rightarrow 'a seq*) \Rightarrow *'a pred* **where**

Seq f = pred-of-seq (f ())

code-datatype *Seq*

primrec *member* :: *'a seq \Rightarrow 'a \Rightarrow bool* **where**

member Empty x \longleftrightarrow False

| *member (Insert y P) x \longleftrightarrow x = y \vee eval P x*

| *member (Join P xq) x \longleftrightarrow eval P x \vee member xq x*

lemma *eval-member*:

member xq = eval (pred-of-seq xq)

```

proof (induct xq)
  case Empty show ?case
  by (auto simp add: fun-eq-iff elim: botE)
next
  case Insert show ?case
  by (auto simp add: fun-eq-iff elim: supE singleE intro: supI1 supI2 singleI)
next
  case Join then show ?case
  by (auto simp add: fun-eq-iff elim: supE intro: supI1 supI2)
qed

```

```

lemma eval-code [code]:  $eval (Seq\ f) = member\ (f\ ())$ 
  unfolding Seq-def by (rule sym, rule eval-member)

```

```

lemma single-code [code]:
   $single\ x = Seq\ (\lambda u. Insert\ x\ \perp)$ 
  unfolding Seq-def by simp

```

```

primrec apply :: ('a  $\Rightarrow$  'b pred)  $\Rightarrow$  'a seq  $\Rightarrow$  'b seq where
  apply f Empty = Empty
| apply f (Insert x P) = Join (f x) (Join (P  $\ggg$  f) Empty)
| apply f (Join P xq) = Join (P  $\ggg$  f) (apply f xq)

```

```

lemma apply-bind:
   $pred\text{-of}\text{-seq}\ (apply\ f\ xq) = pred\text{-of}\text{-seq}\ xq\ \ggg\ f$ 
proof (induct xq)
  case Empty show ?case
  by (simp add: bottom-bind)
next
  case Insert show ?case
  by (simp add: single-bind sup-bind)
next
  case Join then show ?case
  by (simp add: sup-bind)
qed

```

```

lemma bind-code [code]:
   $Seq\ g\ \ggg\ f = Seq\ (\lambda u. apply\ f\ (g\ ()))$ 
  unfolding Seq-def by (rule sym, rule apply-bind)

```

```

lemma bot-set-code [code]:
   $\perp = Seq\ (\lambda u. Empty)$ 
  unfolding Seq-def by simp

```

```

primrec adjunct :: 'a pred  $\Rightarrow$  'a seq  $\Rightarrow$  'a seq where
  adjunct P Empty = Join P Empty
| adjunct P (Insert x Q) = Insert x (Q  $\sqcup$  P)
| adjunct P (Join Q xq) = Join Q (adjunct P xq)

```

lemma *adjunct-sup*:
 $pred\text{-of}\text{-seq} (adjunct\ P\ xq) = P \sqcup pred\text{-of}\text{-seq}\ xq$
by (*induct* xq) (*simp-all* *add*: *sup-assoc* *sup-commute* *sup-left-commute*)

lemma *sup-code* [*code*]:
 $Seq\ f \sqcup Seq\ g = Seq\ (\lambda u. case\ f\ ())$
of *Empty* $\Rightarrow g\ ()$
| *Insert* $x\ P \Rightarrow Insert\ x\ (P \sqcup Seq\ g)$
| *Join* $P\ xq \Rightarrow adjunct\ (Seq\ g)\ (Join\ P\ xq)$

proof (*cases* $f\ ()$)
case *Empty*
thus *?thesis*
unfolding *Seq-def* **by** (*simp* *add*: *sup-commute* [*of* \perp])

next
case *Insert*
thus *?thesis*
unfolding *Seq-def* **by** (*simp* *add*: *sup-assoc*)

next
case *Join*
thus *?thesis*
unfolding *Seq-def*
by (*simp* *add*: *adjunct-sup* *sup-assoc* *sup-commute* *sup-left-commute*)

qed

primrec *contained* :: '*a* *seq* \Rightarrow '*a* *pred* \Rightarrow *bool* **where**
contained *Empty* $Q \longleftrightarrow True$
| *contained* (*Insert* $x\ P$) $Q \longleftrightarrow eval\ Q\ x \wedge P \leq Q$
| *contained* (*Join* $P\ xq$) $Q \longleftrightarrow P \leq Q \wedge contained\ xq\ Q$

lemma *single-less-eq-eval*:
 $single\ x \leq P \longleftrightarrow eval\ P\ x$
by (*auto* *simp* *add*: *less-eq-pred-def* *le-fun-def*)

lemma *contained-less-eq*:
 $contained\ xq\ Q \longleftrightarrow pred\text{-of}\text{-seq}\ xq \leq Q$
by (*induct* xq) (*simp-all* *add*: *single-less-eq-eval*)

lemma *less-eq-pred-code* [*code*]:
 $Seq\ f \leq Q = (case\ f\ ())$
of *Empty* $\Rightarrow True$
| *Insert* $x\ P \Rightarrow eval\ Q\ x \wedge P \leq Q$
| *Join* $P\ xq \Rightarrow P \leq Q \wedge contained\ xq\ Q$

by (*cases* $f\ ()$)
(*simp-all* *add*: *Seq-def* *single-less-eq-eval* *contained-less-eq*)

instantiation *pred* :: (*type*) *equal*
begin

definition *equal-pred*

```

where [simp]: HOL.equal P Q  $\longleftrightarrow$  P = (Q :: 'a pred)

instance by standard simp

end

lemma [code]:
  HOL.equal P Q  $\longleftrightarrow$  P  $\leq$  Q  $\wedge$  Q  $\leq$  P for P Q :: 'a pred
  by auto

lemma [code nbe]:
  HOL.equal P P  $\longleftrightarrow$  True for P :: 'a pred
  by (fact equal-refl)

lemma [code]:
  case-pred f P = f (eval P)
  by (fact pred.case-eq-if)

lemma [code]:
  rec-pred f P = f (eval P)
  by (cases P) simp

inductive eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool where eq x x

lemma eq-is-eq: eq x y  $\equiv$  (x = y)
  by (rule eq-reflection) (auto intro: eq.intros elim: eq.cases)

primrec null :: 'a seq  $\Rightarrow$  bool where
  null Empty  $\longleftrightarrow$  True
| null (Insert x P)  $\longleftrightarrow$  False
| null (Join P xq)  $\longleftrightarrow$  is-empty P  $\wedge$  null xq

lemma null-is-empty:
  null xq  $\longleftrightarrow$  is-empty (pred-of-seq xq)
  by (induct xq) (simp-all add: is-empty-bot not-is-empty-single is-empty-sup)

lemma is-empty-code [code]:
  is-empty (Seq f)  $\longleftrightarrow$  null (f ())
  by (simp add: null-is-empty Seq-def)

primrec the-only :: (unit  $\Rightarrow$  'a)  $\Rightarrow$  'a seq  $\Rightarrow$  'a where
  the-only default Empty = default () for default
| the-only default (Insert x P) =
  (if is-empty P then x else let y = singleton default P in if x = y then x else
  default ()) for default
| the-only default (Join P xq) =
  (if is-empty P then the-only default xq else if null xq then singleton default P
  else let x = singleton default P; y = the-only default xq in
  if x = y then x else default ()) for default

```

lemma *the-only-singleton*:

the-only default xq = singleton default (pred-of-seq xq) for default
by (*induct xq*)
(auto simp add: singleton-bot singleton-single is-empty-def
null-is-empty Let-def singleton-sup)

lemma *singleton-code* [*code*]:

singleton default (Seq f) =
(case f () of
Empty \Rightarrow default ())
| Insert x P \Rightarrow if is-empty P then x
else let y = singleton default P in
if x = y then x else default ())
| Join P xq \Rightarrow if is-empty P then the-only default xq
else if null xq then singleton default P
else let x = singleton default P; y = the-only default xq in
if x = y then x else default ()) for default
by (*cases f ()*)
(auto simp add: Seq-def the-only-singleton is-empty-def
null-is-empty singleton-bot singleton-single singleton-sup Let-def)

definition *the* :: 'a pred \Rightarrow 'a **where**

the A = (THE x. eval A x)

lemma *the-eqI*:

(THE x. eval P x) = x \Longrightarrow the P = x
by (*simp add: the-def*)

lemma *the-eq* [*code*]: *the A = singleton (λx . Code.abort (STR "not-unique") (λ -. the A)) A*

by (*rule the-eqI*) (*simp add: singleton-def the-def*)

code-reflect *Predicate*

datatypes *pred = Seq and seq = Empty | Insert | Join*

ML \langle

signature PREDICATE =

sig

*val anamorph: ('a \rightarrow ('b * 'a) option) \rightarrow int \rightarrow 'a \rightarrow 'b list * 'a*

datatype 'a pred = Seq of (unit \rightarrow 'a seq)

*and 'a seq = Empty | Insert of 'a * 'a pred | Join of 'a pred * 'a seq*

val map: ('a \rightarrow 'b) \rightarrow 'a pred \rightarrow 'b pred

*val yield: 'a pred \rightarrow ('a * 'a pred) option*

*val yieldn: int \rightarrow 'a pred \rightarrow 'a list * 'a pred*

end;

structure Predicate : PREDICATE =

struct

```

fun anamorph f k x =
  (if k = 0 then ([], x)
   else case f x
         of NONE => ([], x)
          | SOME (v, y) => let
              val k' = k - 1;
              val (vs, z) = anamorph f k' y
            in (v :: vs, z) end);

datatype pred = datatype Predicate.pred
datatype seq = datatype Predicate.seq

fun map f = @{code Predicate.map} f;

fun yield (Seq f) = next (f ())
and next Empty = NONE
  | next (Insert (x, P)) = SOME (x, P)
  | next (Join (P, xq)) = (case yield P
                          of NONE => next xq
                           | SOME (x, Q) => SOME (x, Seq (fn => Join (Q, xq))));

fun yieldn k = anamorph yield k;

end;
>

```

Conversion from and to sets

definition *pred-of-set* :: 'a set \Rightarrow 'a pred **where**
pred-of-set = Pred \circ ($\lambda A x. x \in A$)

lemma *eval-pred-of-set* [simp]:
eval (pred-of-set A) x \longleftrightarrow x \in A
by (simp add: pred-of-set-def)

definition *set-of-pred* :: 'a pred \Rightarrow 'a set **where**
set-of-pred = Collect \circ *eval*

lemma *member-set-of-pred* [simp]:
x \in set-of-pred P \longleftrightarrow Predicate.eval P x
by (simp add: set-of-pred-def)

definition *set-of-seq* :: 'a seq \Rightarrow 'a set **where**
set-of-seq = set-of-pred \circ pred-of-seq

lemma *member-set-of-seq* [simp]:
x \in set-of-seq xq = Predicate.member xq x
by (simp add: set-of-seq-def eval-member)

lemma *of-pred-code* [*code*]:
set-of-pred (*Predicate.Seq* *f*) = (*case* *f* () of
 Predicate.Empty \Rightarrow {}
 | *Predicate.Insert* *x P* \Rightarrow *insert* *x* (*set-of-pred* *P*)
 | *Predicate.Join* *P xq* \Rightarrow *set-of-pred* *P* \cup *set-of-seq* *xq*)
by (*auto split: seq.split simp add: eval-code*)

lemma *of-seq-code* [*code*]:
set-of-seq *Predicate.Empty* = {}
set-of-seq (*Predicate.Insert* *x P*) = *insert* *x* (*set-of-pred* *P*)
set-of-seq (*Predicate.Join* *P xq*) = *set-of-pred* *P* \cup *set-of-seq* *xq*
by *auto*

Lazy Evaluation of an indexed function

function *iterate-upto* :: (*natural* \Rightarrow 'a) \Rightarrow *natural* \Rightarrow *natural* \Rightarrow 'a *Predicate.pred*
where
 iterate-upto *f n m* =
 Predicate.Seq (%*u*. if *n* > *m* then *Predicate.Empty*
 else *Predicate.Insert* (*f n*) (*iterate-upto* *f* (*n* + 1) *m*))
by *pat-completeness auto*

termination by (*relation measure* (%(*f*, *n*, *m*). *nat-of-natural* (*m* + 1 - *n*)))
 (*auto simp add: less-natural-def*)

Misc

declare *Inf-set-fold* [**where** 'a = 'a *Predicate.pred*, *code*]
declare *Sup-set-fold* [**where** 'a = 'a *Predicate.pred*, *code*]

lemma *pred-of-set-fold-sup*:
 assumes *finite A*
 shows *pred-of-set* *A* = *Finite-Set.fold sup bot* (*Predicate.single* 'A) (**is** ?*lhs* =
 ?*rhs*)
proof (*rule sym*)
 interpret *comp-fun-idem sup* :: 'a *Predicate.pred* \Rightarrow 'a *Predicate.pred* \Rightarrow 'a *Pred-*
icate.pred
 by (*fact comp-fun-idem-sup*)
 from <*finite A*> **show** ?*rhs* = ?*lhs* **by** (*induct A*) (*auto intro!: pred-eqI*)
qed

lemma *pred-of-set-set-fold-sup*:
 pred-of-set (*set xs*) = *fold sup* (*List.map Predicate.single xs*) *bot*
proof –
 interpret *comp-fun-idem sup* :: 'a *Predicate.pred* \Rightarrow 'a *Predicate.pred* \Rightarrow 'a *Pred-*
icate.pred
 by (*fact comp-fun-idem-sup*)
 show ?*thesis* **by** (*simp add: pred-of-set-fold-sup fold-set-fold [symmetric]*)
qed

lemma *pred-of-set-set-foldr-sup* [code]:
pred-of-set (set xs) = foldr sup (List.map Predicate.single xs) bot
by (*simp add: pred-of-set-set-fold-sup ac-simps foldr-fold fun-eq-iff*)

no-notation
bind (infixl $\gg=$ 70)

hide-type (**open**) *pred seq*
hide-const (**open**) *Pred eval single bind is-empty singleton if-pred not-pred holds*
Empty Insert Join Seq member pred-of-seq apply adjunct null the-only eq map the
iterate-upto
hide-fact (**open**) *null-def member-def*

end

77 Lazy sequences

theory *Lazy-Sequence*
imports *Predicate*
begin

77.1 Type of lazy sequences

datatype (*plugins only: code extraction*) (*dead 'a*) *lazy-sequence =*
lazy-sequence-of-list 'a list

primrec *list-of-lazy-sequence :: 'a lazy-sequence \Rightarrow 'a list*
where
list-of-lazy-sequence (lazy-sequence-of-list xs) = xs

lemma *lazy-sequence-of-list-of-lazy-sequence* [simp]:
lazy-sequence-of-list (list-of-lazy-sequence xq) = xq
by (*cases xq*) *simp-all*

lemma *lazy-sequence-eqI*:
list-of-lazy-sequence xq = list-of-lazy-sequence yq \implies xq = yq
by (*cases xq, cases yq*) *simp*

lemma *lazy-sequence-eq-iff*:
xq = yq \longleftrightarrow list-of-lazy-sequence xq = list-of-lazy-sequence yq
by (*auto intro: lazy-sequence-eqI*)

lemma *case-lazy-sequence* [simp]:
case-lazy-sequence f xq = f (list-of-lazy-sequence xq)
by (*cases xq*) *auto*

lemma *rec-lazy-sequence* [simp]:
rec-lazy-sequence f xq = f (list-of-lazy-sequence xq)

by (cases xq) auto

definition *Lazy-Sequence* :: (unit \Rightarrow ('a \times 'a lazy-sequence) option) \Rightarrow 'a lazy-sequence
where

Lazy-Sequence f = lazy-sequence-of-list (case f () of
 None \Rightarrow []
 | Some (x, xq) \Rightarrow x # list-of-lazy-sequence xq)

code-datatype *Lazy-Sequence*

declare list-of-lazy-sequence.simps [code del]
declare lazy-sequence.case [code del]
declare lazy-sequence.rec [code del]

lemma list-of-Lazy-Sequence [simp]:
 list-of-lazy-sequence (Lazy-Sequence f) = (case f () of
 None \Rightarrow []
 | Some (x, xq) \Rightarrow x # list-of-lazy-sequence xq)
by (simp add: Lazy-Sequence-def)

definition *yield* :: 'a lazy-sequence \Rightarrow ('a \times 'a lazy-sequence) option
where

yield xq = (case list-of-lazy-sequence xq of
 [] \Rightarrow None
 | x # xs \Rightarrow Some (x, lazy-sequence-of-list xs))

lemma yield-Seq [simp, code]:
yield (Lazy-Sequence f) = f ()
by (cases f ()) (simp-all add: yield-def split-def)

lemma case-yield-eq [simp]: case-option g h (yield xq) =
 case-list g ($\lambda x.$ curry h x \circ lazy-sequence-of-list) (list-of-lazy-sequence xq)
by (cases list-of-lazy-sequence xq) (simp-all add: yield-def)

lemma equal-lazy-sequence-code [code]:
 HOL.equal xq yq = (case (yield xq, yield yq) of
 (None, None) \Rightarrow True
 | (Some (x, xq'), Some (y, yq')) \Rightarrow HOL.equal x y \wedge HOL.equal xq yq
 | - \Rightarrow False)
by (simp-all add: lazy-sequence-eq-iff equal-eq split: list.splits)

lemma [code nbe]:
 HOL.equal (x :: 'a lazy-sequence) x \longleftrightarrow True
by (fact equal-refl)

definition *empty* :: 'a lazy-sequence
where
empty = lazy-sequence-of-list []

lemma *list-of-lazy-sequence-empty* [*simp*]:
list-of-lazy-sequence empty = []
by (*simp add: empty-def*)

lemma *empty-code* [*code*]:
empty = *Lazy-Sequence* (λ -. *None*)
by (*simp add: lazy-sequence-eq-iff*)

definition *single* :: 'a \Rightarrow 'a *lazy-sequence*
where
single x = *lazy-sequence-of-list* [x]

lemma *list-of-lazy-sequence-single* [*simp*]:
list-of-lazy-sequence (single x) = [x]
by (*simp add: single-def*)

lemma *single-code* [*code*]:
single x = *Lazy-Sequence* (λ -. *Some* (x, *empty*))
by (*simp add: lazy-sequence-eq-iff*)

definition *append* :: 'a *lazy-sequence* \Rightarrow 'a *lazy-sequence* \Rightarrow 'a *lazy-sequence*
where
append xq yq = *lazy-sequence-of-list* (*list-of-lazy-sequence* xq @ *list-of-lazy-sequence* yq)

lemma *list-of-lazy-sequence-append* [*simp*]:
list-of-lazy-sequence (append xq yq) = *list-of-lazy-sequence* xq @ *list-of-lazy-sequence* yq
by (*simp add: append-def*)

lemma *append-code* [*code*]:
append xq yq = *Lazy-Sequence* (λ -. *case yield xq of*
None \Rightarrow *yield yq*
| *Some* (x, xq') \Rightarrow *Some* (x, *append xq' yq*)
by (*simp-all add: lazy-sequence-eq-iff split: list.splits*)

definition *map* :: ('a \Rightarrow 'b) \Rightarrow 'a *lazy-sequence* \Rightarrow 'b *lazy-sequence*
where
map f xq = *lazy-sequence-of-list* (*List.map* f (*list-of-lazy-sequence* xq))

lemma *list-of-lazy-sequence-map* [*simp*]:
list-of-lazy-sequence (map f xq) = *List.map* f (*list-of-lazy-sequence* xq)
by (*simp add: map-def*)

lemma *map-code* [*code*]:
map f xq =
Lazy-Sequence (λ -. *map-option* (λ (x, xq'). (f x, *map f xq'*)) (*yield* xq))
by (*simp-all add: lazy-sequence-eq-iff split: list.splits*)

definition *flat* :: 'a lazy-sequence lazy-sequence \Rightarrow 'a lazy-sequence

where

flat *xqq* = lazy-sequence-of-list (concat (List.map list-of-lazy-sequence (list-of-lazy-sequence *xqq*)))

lemma *list-of-lazy-sequence-flat* [simp]:

list-of-lazy-sequence (*flat* *xqq*) = concat (List.map list-of-lazy-sequence (list-of-lazy-sequence *xqq*))

by (simp add: flat-def)

lemma *flat-code* [code]:

flat *xq* = Lazy-Sequence (λ -. case yield *xq* of

None \Rightarrow None

| Some (*xq*, *xqq'*) \Rightarrow yield (append *xq* (*flat* *xqq'*)))

by (simp add: lazy-sequence-eq-iff split: list.splits)

definition *bind* :: 'a lazy-sequence \Rightarrow ('a \Rightarrow 'b lazy-sequence) \Rightarrow 'b lazy-sequence

where

bind *xq* *f* = flat (map *f* *xq*)

definition *if-seq* :: bool \Rightarrow unit lazy-sequence

where

if-seq *b* = (if *b* then single () else empty)

definition *those* :: 'a option lazy-sequence \Rightarrow 'a lazy-sequence option

where

those *xq* = map-option lazy-sequence-of-list (List.those (list-of-lazy-sequence *xq*))

function *iterate-upto* :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a lazy-sequence

where

iterate-upto *f* *n* *m* =

Lazy-Sequence (λ -. if *n* > *m* then None else Some (*f* *n*, *iterate-upto* *f* (*n* + 1) *m*))

by pat-completeness auto

termination by (relation measure (λ (*f*, *n*, *m*). nat-of-natural (*m* + 1 - *n*)))

(auto simp add: less-natural-def)

definition *not-seq* :: unit lazy-sequence \Rightarrow unit lazy-sequence

where

not-seq *xq* = (case yield *xq* of

None \Rightarrow single ()

| Some ((), *xq*) \Rightarrow empty)

77.2 Code setup

code-reflect *Lazy-Sequence*

datatypes *lazy-sequence* = *Lazy-Sequence*

```

ML <
signature LAZY-SEQUENCE =
sig
  datatype 'a lazy-sequence = Lazy-Sequence of (unit -> ('a * 'a Lazy-Sequence.lazy-sequence)
  option)
  val map: ('a -> 'b) -> 'a lazy-sequence -> 'b lazy-sequence
  val yield: 'a lazy-sequence -> ('a * 'a lazy-sequence) option
  val yieldn: int -> 'a lazy-sequence -> 'a list * 'a lazy-sequence
end;

structure Lazy-Sequence : LAZY-SEQUENCE =
struct

  datatype lazy-sequence = datatype Lazy-Sequence.lazy-sequence;

  fun map f = @{code Lazy-Sequence.map} f;

  fun yield P = @{code Lazy-Sequence.yield} P;

  fun yieldn k = Predicate.anamorph yield k;

end;
>

```

77.3 Generator Sequences

77.3.1 General lazy sequence operation

definition $product :: 'a\ lazy-sequence \Rightarrow 'b\ lazy-sequence \Rightarrow ('a \times 'b)\ lazy-sequence$
where
 $product\ s1\ s2 = bind\ s1\ (\lambda a. bind\ s2\ (\lambda b. single\ (a, b)))$

77.3.2 Small lazy typeclasses

class *small-lazy* =
fixes *small-lazy* :: *natural* \Rightarrow 'a lazy-sequence

instantiation *unit* :: *small-lazy*
begin

definition *small-lazy* *d* = *single* ()

instance ..

end

instantiation *int* :: *small-lazy*
begin

maybe optimise this expression -> append (single x) xs == cons x xs Per-

formance difference?

function *small-lazy'* :: *int* ⇒ *int* ⇒ *int lazy-sequence*

where

small-lazy' d i = (if *d* < *i* then *empty*
 else *append (single i) (small-lazy' d (i + 1))*)
by *pat-completeness auto*

termination

by (*relation measure* (%(*d, i*). *nat (d + 1 - i)*)) *auto*

definition

small-lazy d = *small-lazy' (int (nat-of-natural d)) (- (int (nat-of-natural d)))*

instance ..

end

instantiation *prod* :: (*small-lazy, small-lazy*) *small-lazy*

begin

definition

small-lazy d = *product (small-lazy d) (small-lazy d)*

instance ..

end

instantiation *list* :: (*small-lazy*) *small-lazy*

begin

fun *small-lazy-list* :: *natural* ⇒ 'a *list lazy-sequence*

where

small-lazy-list d = *append (single [])*
 (if *d* > 0 then *bind (product (small-lazy (d - 1))*
(small-lazy (d - 1))) (λ(x, xs). single (x # xs)) else *empty*)

instance ..

end

77.4 With Hit Bound Value

assuming in negative context

type-synonym 'a *hit-bound-lazy-sequence* = 'a *option lazy-sequence*

definition *hit-bound* :: 'a *hit-bound-lazy-sequence*

where

hit-bound = *Lazy-Sequence (λ-. Some (None, empty))*

lemma *list-of-lazy-sequence-hit-bound* [simp]:

list-of-lazy-sequence hit-bound = [None]

by (*simp add: hit-bound-def*)

definition *hb-single* :: 'a ⇒ 'a hit-bound-lazy-sequence

where

hb-single x = *Lazy-Sequence* (λ-. *Some* (*Some* x, *empty*))

definition *hb-map* :: ('a ⇒ 'b) ⇒ 'a hit-bound-lazy-sequence ⇒ 'b hit-bound-lazy-sequence

where

hb-map f xq = *map* (*map-option* f) xq

lemma *hb-map-code* [code]:

hb-map f xq =

Lazy-Sequence (λ-. *map-option* (λ(x, xq'). (*map-option* f x, *hb-map* f xq')) (*yield* xq))

using *map-code* [of *map-option* f xq] **by** (*simp add: hb-map-def*)

definition *hb-flat* :: 'a hit-bound-lazy-sequence hit-bound-lazy-sequence ⇒ 'a hit-bound-lazy-sequence

where

hb-flat xqq = *lazy-sequence-of-list* (*concat*

(*List.map* ((λx. *case* x of *None* ⇒ [None] | *Some* xs ⇒ xs) ∘ *map-option*

list-of-lazy-sequence) (*list-of-lazy-sequence* xqq)))

lemma *list-of-lazy-sequence-hb-flat* [simp]:

list-of-lazy-sequence (*hb-flat* xqq) =

concat (*List.map* ((λx. *case* x of *None* ⇒ [None] | *Some* xs ⇒ xs) ∘ *map-option* *list-of-lazy-sequence*) (*list-of-lazy-sequence* xqq))

by (*simp add: hb-flat-def*)

lemma *hb-flat-code* [code]:

hb-flat xqq = *Lazy-Sequence* (λ-. *case* *yield* xqq of

None ⇒ *None*

| *Some* (xq, xqq') ⇒ *yield*

(*append* (*case* xq of *None* ⇒ *hit-bound* | *Some* xq ⇒ xq) (*hb-flat* xqq')))

by (*simp add: lazy-sequence-eq-iff split: list.splits option.splits*)

definition *hb-bind* :: 'a hit-bound-lazy-sequence ⇒ ('a ⇒ 'b hit-bound-lazy-sequence)

⇒ 'b hit-bound-lazy-sequence

where

hb-bind xq f = *hb-flat* (*hb-map* f xq)

definition *hb-if-seq* :: bool ⇒ unit hit-bound-lazy-sequence

where

hb-if-seq b = (*if* b then *hb-single* () else *empty*)

definition *hb-not-seq* :: unit hit-bound-lazy-sequence ⇒ unit lazy-sequence

where

```

    hb-not-seq xq = (case yield xq of
      None ⇒ single ()
    | Some (x, xq) ⇒ empty)

hide-const (open) yield empty single append flat map bind
  if-seq those iterate-upto not-seq product

hide-fact (open) yield-def empty-def single-def append-def flat-def map-def bind-def
  if-seq-def those-def not-seq-def product-def

end

```

78 Depth-Limited Sequences with failure element

```

theory Limited-Sequence
imports Lazy-Sequence
begin

```

78.1 Depth-Limited Sequence

```

type-synonym 'a dseq = natural ⇒ bool ⇒ 'a lazy-sequence option

```

```

definition empty :: 'a dseq
where
  empty = (λ-. Some Lazy-Sequence.empty)

```

```

definition single :: 'a ⇒ 'a dseq
where
  single x = (λ-. Some (Lazy-Sequence.single x))

```

```

definition eval :: 'a dseq ⇒ natural ⇒ bool ⇒ 'a lazy-sequence option
where
  [simp]: eval f i pol = f i pol

```

```

definition yield :: 'a dseq ⇒ natural ⇒ bool ⇒ ('a × 'a dseq) option
where
  yield f i pol = (case eval f i pol of
    None ⇒ None
  | Some s ⇒ (map-option ∘ apsnd) (λr -. Some r) (Lazy-Sequence.yield s))

```

```

definition map-seq :: ('a ⇒ 'b dseq) ⇒ 'a lazy-sequence ⇒ 'b dseq
where
  map-seq f xq i pol = map-option Lazy-Sequence.flat
    (Lazy-Sequence.those (Lazy-Sequence.map (λx. f x i pol) xq))

```

```

lemma map-seq-code [code]:
  map-seq f xq i pol = (case Lazy-Sequence.yield xq of
    None ⇒ Some Lazy-Sequence.empty
  | Some (x, xq') ⇒ (case eval (f x) i pol of

```



```

    None  $\Rightarrow$  None
  | Some yq  $\Rightarrow$  (case map-seq f xq' i pol of
    None  $\Rightarrow$  None
    | Some zq  $\Rightarrow$  Some (Lazy-Sequence.append yq zq))))
  by (cases xq)
    (auto simp add: map-seq-def Lazy-Sequence.those-def lazy-sequence-eq-iff split:
list.splits option.splits)

```

definition *bind* :: 'a dseq \Rightarrow ('a \Rightarrow 'b dseq) \Rightarrow 'b dseq

where

```

bind x f = ( $\lambda$ i pol.
  if i = 0 then
    (if pol then Some Lazy-Sequence.empty else None)
  else
    (case x (i - 1) pol of
      None  $\Rightarrow$  None
      | Some xq  $\Rightarrow$  map-seq f xq i pol))

```

definition *union* :: 'a dseq \Rightarrow 'a dseq \Rightarrow 'a dseq

where

```

union x y = ( $\lambda$ i pol. case (x i pol, y i pol) of
  (Some xq, Some yq)  $\Rightarrow$  Some (Lazy-Sequence.append xq yq)
  | -  $\Rightarrow$  None)

```

definition *if-seq* :: bool \Rightarrow unit dseq

where

```

if-seq b = (if b then single () else empty)

```

definition *not-seq* :: unit dseq \Rightarrow unit dseq

where

```

not-seq x = ( $\lambda$ i pol. case x i ( $\neg$  pol) of
  None  $\Rightarrow$  Some Lazy-Sequence.empty
  | Some xq  $\Rightarrow$  (case Lazy-Sequence.yield xq of
    None  $\Rightarrow$  Some (Lazy-Sequence.single ())
    | Some -  $\Rightarrow$  Some (Lazy-Sequence.empty)))

```

definition *map* :: ('a \Rightarrow 'b) \Rightarrow 'a dseq \Rightarrow 'b dseq

where

```

map f g = ( $\lambda$ i pol. case g i pol of
  None  $\Rightarrow$  None
  | Some xq  $\Rightarrow$  Some (Lazy-Sequence.map f xq))

```

78.2 Positive Depth-Limited Sequence

type-synonym 'a pos-dseq = natural \Rightarrow 'a Lazy-Sequence.lazy-sequence

definition *pos-empty* :: 'a pos-dseq

where

```

pos-empty = ( $\lambda$ i. Lazy-Sequence.empty)

```

definition *pos-single* :: 'a ⇒ 'a pos-dseq

where

pos-single x = (λi. *Lazy-Sequence.single* x)

definition *pos-bind* :: 'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq

where

pos-bind x f = (λi. *Lazy-Sequence.bind* (x i) (λa. f a i))

definition *pos-decr-bind* :: 'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq

where

pos-decr-bind x f = (λi.
 if i = 0 then
 Lazy-Sequence.empty
 else
 Lazy-Sequence.bind (x (i - 1)) (λa. f a i))

definition *pos-union* :: 'a pos-dseq ⇒ 'a pos-dseq ⇒ 'a pos-dseq

where

pos-union xq yq = (λi. *Lazy-Sequence.append* (xq i) (yq i))

definition *pos-if-seq* :: bool ⇒ unit pos-dseq

where

pos-if-seq b = (if b then *pos-single* () else *pos-empty*)

definition *pos-iterate-upto* :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a pos-dseq

where

pos-iterate-upto f n m = (λi. *Lazy-Sequence.iterate-upto* f n m)

definition *pos-map* :: ('a ⇒ 'b) ⇒ 'a pos-dseq ⇒ 'b pos-dseq

where

pos-map f xq = (λi. *Lazy-Sequence.map* f (xq i))

78.3 Negative Depth-Limited Sequence

type-synonym 'a neg-dseq = natural ⇒ 'a *Lazy-Sequence.hit-bound-lazy-sequence*

definition *neg-empty* :: 'a neg-dseq

where

neg-empty = (λi. *Lazy-Sequence.empty*)

definition *neg-single* :: 'a ⇒ 'a neg-dseq

where

neg-single x = (λi. *Lazy-Sequence.hb-single* x)

definition *neg-bind* :: 'a neg-dseq ⇒ ('a ⇒ 'b neg-dseq) ⇒ 'b neg-dseq

where

neg-bind x f = (λi. *hb-bind* (x i) (λa. f a i))

definition *neg-decr-bind* :: 'a neg-dseq ⇒ ('a ⇒ 'b neg-dseq) ⇒ 'b neg-dseq
where

```
neg-decr-bind x f = (λi.
  if i = 0 then
    Lazy-Sequence.hit-bound
  else
    hb-bind (x (i - 1)) (λa. f a i))
```

definition *neg-union* :: 'a neg-dseq ⇒ 'a neg-dseq ⇒ 'a neg-dseq
where

```
neg-union x y = (λi. Lazy-Sequence.append (x i) (y i))
```

definition *neg-if-seq* :: bool ⇒ unit neg-dseq
where

```
neg-if-seq b = (if b then neg-single () else neg-empty)
```

definition *neg-iterate-upto*
where

```
neg-iterate-upto f n m = (λi. Lazy-Sequence.iterate-upto (λi. Some (f i)) n m)
```

definition *neg-map* :: ('a ⇒ 'b) ⇒ 'a neg-dseq ⇒ 'b neg-dseq
where

```
neg-map f xq = (λi. Lazy-Sequence.hb-map f (xq i))
```

78.4 Negation

definition *pos-not-seq* :: unit neg-dseq ⇒ unit pos-dseq
where

```
pos-not-seq xq = (λi. Lazy-Sequence.hb-not-seq (xq (3 * i)))
```

definition *neg-not-seq* :: unit pos-dseq ⇒ unit neg-dseq
where

```
neg-not-seq x = (λi. case Lazy-Sequence.yield (x i) of
  None ⇒ Lazy-Sequence.hb-single ()
| Some ((), xq) ⇒ Lazy-Sequence.empty)
```

ML ‹

signature LIMITED-SEQUENCE =

sig

```
type 'a dseq = Code-Numeral.natural -> bool -> 'a Lazy-Sequence.lazy-sequence
option
```

```
val map : ('a -> 'b) -> 'a dseq -> 'b dseq
```

```
val yield : 'a dseq -> Code-Numeral.natural -> bool -> ('a * 'a dseq) option
```

```
val yieldn : int -> 'a dseq -> Code-Numeral.natural -> bool -> 'a list * 'a
dseq
```

```
end;
```

structure Limited-Sequence : LIMITED-SEQUENCE =

struct

type 'a dseq = Code-Numeral.natural -> bool -> 'a Lazy-Sequence.lazy-sequence option

fun map f = @{code Limited-Sequence.map} f;

fun yield f = @{code Limited-Sequence.yield} f;

*fun yieldn n f i pol = (case f i pol of
 NONE => ([], fn - => fn - => NONE)
 | SOME s => let val (xs, s') = Lazy-Sequence.yieldn n s in (xs, fn - => fn - =>
 SOME s') end);*

end;

>

code-reserved *Eval Limited-Sequence*

hide-const (**open**) *yield empty single eval map-seq bind union if-seq not-seq map
 pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-seq pos-iterate-upto
 pos-not-seq pos-map
 neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-seq neg-iterate-upto
 neg-not-seq neg-map*

hide-fact (**open**) *yield-def empty-def single-def eval-def map-seq-def bind-def union-def
 if-seq-def not-seq-def map-def
 pos-empty-def pos-single-def pos-bind-def pos-union-def pos-if-seq-def pos-iterate-upto-def
 pos-not-seq-def pos-map-def
 neg-empty-def neg-single-def neg-bind-def neg-union-def neg-if-seq-def neg-iterate-upto-def
 neg-not-seq-def neg-map-def*

end

79 Term evaluation using the generic code generator

theory *Code-Evaluation*
imports *Typerep Limited-Sequence*
keywords *value :: diag*
begin

79.1 Term representation

79.1.1 Terms and class *term-of*

datatype (*plugins only: extraction*) *term = dummy-term*

definition *Const* :: *String.literal* \Rightarrow *typerep* \Rightarrow *term* **where**
Const - - = *dummy-term*

definition *App* :: *term* \Rightarrow *term* \Rightarrow *term* **where**
App - - = *dummy-term*

definition *Abs* :: *String.literal* \Rightarrow *typerep* \Rightarrow *term* \Rightarrow *term* **where**
Abs - - - = *dummy-term*

definition *Free* :: *String.literal* \Rightarrow *typerep* \Rightarrow *term* **where**
Free - - = *dummy-term*

code-datatype *Const App Abs Free*

class *term-of* = *typerep* +
fixes *term-of* :: 'a \Rightarrow *term*

lemma *term-of-anything*: *term-of* *x* \equiv *t*
by (*rule eq-reflection*) (*cases term-of x, cases t, simp*)

definition *valapp* :: ('a \Rightarrow 'b) \times (*unit* \Rightarrow *term*)
 \Rightarrow 'a \times (*unit* \Rightarrow *term*) \Rightarrow 'b \times (*unit* \Rightarrow *term*) **where**
valapp *f* *x* = (*fst* *f* (*fst* *x*), λ u. *App* (*snd* *f* ()) (*snd* *x* ()))

lemma *valapp-code* [*code, code-unfold*]:
valapp (*f*, *tf*) (*x*, *tx*) = (*f* *x*, λ u. *App* (*tf* ()) (*tx* ()))
by (*simp only: valapp-def fst-conv snd-conv*)

79.1.2 Syntax

definition *termify* :: 'a \Rightarrow *term* **where**
[*code del*]: *termify* *x* = *dummy-term*

abbreviation *valtermify* :: 'a \Rightarrow 'a \times (*unit* \Rightarrow *term*) **where**
valtermify *x* \equiv (*x*, λ u. *termify* *x*)

bundle *term-syntax*
begin

notation *App* (**infixl** <.> 70)
and *valapp* (**infixl** {.} 70)

end

79.2 Tools setup and evaluation

context
begin

qualified definition *TERM-OF* :: 'a::*term-of itself*

where

$TERM-OF = snd (Code-Evaluation.term-of :: 'a \Rightarrow -, TYPE('a))$

qualified definition $TERM-OF-EQUAL :: 'a::term-of\ itself$

where

$TERM-OF-EQUAL = snd (\lambda(a::'a). (Code-Evaluation.term-of\ a, HOL.eq\ a), TYPE('a))$

end

lemma $eq-eq-TrueD:$

fixes $x\ y :: 'a::\{\}$

assumes $(x \equiv y) \equiv Trueprop\ True$

shows $x \equiv y$

using $assms$ **by** $simp$

code-printing

type-constructor $term \rightarrow (Eval)\ Term.term$

| **constant** $Const \rightarrow (Eval)\ Term.Const / ((-), (-))$

| **constant** $App \rightarrow (Eval)\ Term.\$ / ((-), (-))$

| **constant** $Abs \rightarrow (Eval)\ Term.Abs / ((-), (-), (-))$

| **constant** $Free \rightarrow (Eval)\ Term.Free / ((-), (-))$

ML-file $\langle Tools/code-evaluation.ML \rangle$

code-reserved $Eval\ Code-Evaluation$

ML-file $\langle \sim\sim / src/HOL/Tools/value-command.ML \rangle$

79.3 Dedicated *term-of* instances

instantiation $fun :: (typerep, typerep)\ term-of$

begin

definition

$term-of\ (f :: 'a \Rightarrow 'b) =$

$Const\ (STR\ "Pure.dummy-pattern")$

$(Typerep.Typeprep\ (STR\ "fun")\ [Typerep.typerep\ TYPE('a), Typerep.typerep\ TYPE('b)])$

instance $..$

end

declare $[[code\ drop:\ rec-term\ case-term$

$term-of :: typerep \Rightarrow -\ term-of :: term \Rightarrow -\ term-of :: String.literal \Rightarrow -$

$term-of :: -\ Predicate.pred \Rightarrow term\ term-of :: -\ Predicate.seq \Rightarrow term]]$

code-printing

constant *term-of* :: *integer* ⇒ *term* → (Eval) *HOLogic.mk'-number* / *HOLogic.code'-integerT*
| **constant** *term-of* :: *String.literal* ⇒ *term* → (Eval) *HOLogic.mk'-literal*

declare [[code drop: *term-of* :: *integer* ⇒ -]]

lemma *term-of-integer* [unfolded *typerep-fun-def typerep-num-def typerep-integer-def*,
code]:

term-of (*i* :: *integer*) =
(if *i* > 0 then
App (Const (STR "Num.numeral-class.numeral") (TYPEREP(*num* ⇒ *integer*)))
(*term-of* (*num-of-integer* *i*))
else if *i* = 0 then Const (STR "Groups.zero-class.zero") TYPEREP(*integer*)
else
App (Const (STR "Groups.uminus-class.uminus") TYPEREP(*integer* ⇒ *integer*))
(*term-of* (- *i*)))
by (rule *term-of-anything* [THEN *meta-eq-to-obj-eq*])

code-reserved *Eval HOLogic*

79.4 Generic reification

ML-file <~/src/HOL/Tools/reification.ML>

79.5 Diagnostic

definition *tracing* :: *String.literal* ⇒ 'a ⇒ 'a **where**
[*code del*]: *tracing* *s* *x* = *x*

code-printing

constant *tracing* :: *String.literal* => 'a => 'a → (Eval) *Code'-Evaluation.tracing*

hide-const *dummy-term valapp*

hide-const (**open**) *Const App Abs Free termify valtermify term-of tracing*

end

80 A simple counterexample generator performing random testing

theory *Quickcheck-Random*

imports *Random Code-Evaluation Enum*

begin

setup <*Code-Target.add-derived-target* (*Quickcheck*, [(*Code-Runtime.target*, *I*)]>

80.1 Catching Match exceptions

axiomatization *catch-match* :: 'a => 'a => 'a

code-printing

constant *catch-match* → (Quickcheck) ((-) handle Match => -)

code-reserved Quickcheck Match

80.2 The random class

class *random* = *typerep* +

fixes *random* :: natural ⇒ Random.seed ⇒ ('a × (unit ⇒ term)) × Random.seed

80.3 Fundamental and numeric types

instantiation *bool* :: *random*

begin

context

includes *state-combinator-syntax*

begin

definition

random i = Random.range 2 ◦→

(λk. Pair (if k = 0 then Code-Evaluation.valtermify False else Code-Evaluation.valtermify True))

instance ..

end

end

instantiation *itself* :: (*typerep*) *random*

begin

definition

random-itself :: natural ⇒ Random.seed ⇒ ('a itself × (unit ⇒ term)) × Random.seed

where *random-itself -* = Pair (Code-Evaluation.valtermify TYPE('a))

instance ..

end

instantiation *char* :: *random*

begin

context

includes *state-combinator-syntax*
begin

definition

random - = *Random.select* (*Enum.enum* :: *char list*) $\circ\rightarrow$ ($\lambda c.$ *Pair* (*c*, $\lambda u.$ *Code-Evaluation.term-of* *c*))

instance ..

end

end

instantiation *String.literal* :: *random*
begin

definition

random - = *Pair* (*STR* ""', $\lambda u.$ *Code-Evaluation.term-of* (*STR* ""'))

instance ..

end

instantiation *nat* :: *random*
begin

context

includes *state-combinator-syntax*
begin

definition *random-nat* :: *natural* \Rightarrow *Random.seed*

\Rightarrow (*nat* \times (*unit* \Rightarrow *Code-Evaluation.term*)) \times *Random.seed*

where

random-nat *i* = *Random.range* (*i* + 1) $\circ\rightarrow$ ($\lambda k.$ *Pair* (
let *n* = *nat-of-natural* *k*
in (*n*, $\lambda.$ *Code-Evaluation.term-of* *n*)))

instance ..

end

end

instantiation *int* :: *random*
begin

context

includes *state-combinator-syntax*
begin

definition

```

random i = Random.range (2 * i + 1) ◦→ (λk. Pair (
  let j = (if k ≥ i then int (nat-of-natural (k - i)) else - (int (nat-of-natural (i
- k))))
  in (j, λ-. Code-Evaluation.term-of j)))

```

instance ..**end****end****instantiation** *natural* :: *random***begin****context****includes** *state-combinator-syntax***begin****definition** *random-natural* :: *natural* ⇒ *Random.seed*⇒ (*natural* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed***where**

```

random-natural i = Random.range (i + 1) ◦→ (λn. Pair (n, λ-. Code-Evaluation.term-of
n))

```

instance ..**end****end****instantiation** *integer* :: *random***begin****context****includes** *state-combinator-syntax***begin****definition** *random-integer* :: *natural* ⇒ *Random.seed*⇒ (*integer* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed***where**

```

random-integer i = Random.range (2 * i + 1) ◦→ (λk. Pair (
  let j = (if k ≥ i then integer-of-natural (k - i) else - (integer-of-natural (i -
k)))
  in (j, λ-. Code-Evaluation.term-of j)))

```

instance ..

end

end

80.4 Complex generators

Towards $'a \Rightarrow 'b$

axiomatization *random-fun-aux* :: *typerep* \Rightarrow *typerep* \Rightarrow ($'a \Rightarrow 'a \Rightarrow \text{bool}$) \Rightarrow ($'a \Rightarrow \text{term}$)
 \Rightarrow (*Random.seed* \Rightarrow ($'b \times (\text{unit} \Rightarrow \text{term})$) \times *Random.seed*)
 \Rightarrow (*Random.seed* \Rightarrow *Random.seed* \times *Random.seed*)
 \Rightarrow *Random.seed* \Rightarrow ($'a \Rightarrow 'b$) \times ($\text{unit} \Rightarrow \text{term}$) \times *Random.seed*

definition *random-fun-lift* :: (*Random.seed* \Rightarrow ($'b \times (\text{unit} \Rightarrow \text{term})$) \times *Random.seed*)
 \Rightarrow *Random.seed* \Rightarrow ($'a::\text{term-of} \Rightarrow 'b::\text{typerep}$) \times ($\text{unit} \Rightarrow \text{term}$) \times *Random.seed*
where
random-fun-lift *f* =
random-fun-aux *TYPEREPA*($'a$) *TYPEREPA*($'b$) (=) *Code-Evaluation.term-of* *f*
Random.split-seed

instantiation *fun* :: ($\{\text{equal}, \text{term-of}\}$, *random*) *random*
begin

definition
random-fun :: *natural* \Rightarrow *Random.seed* \Rightarrow ($'a \Rightarrow 'b$) \times ($\text{unit} \Rightarrow \text{term}$) \times *Random.seed*
where *random* *i* = *random-fun-lift* (*random* *i*)

instance ..

end

Towards type copies and datatypes

context
includes *state-combinator-syntax*
begin

definition *collapse* :: ($'a \Rightarrow ('a \Rightarrow 'b \times 'a) \times 'a$) \Rightarrow $'a \Rightarrow 'b \times 'a$
where *collapse* *f* = (*f* $\circ \rightarrow$ *id*)

end

definition *beyond* :: *natural* \Rightarrow *natural* \Rightarrow *natural*
where *beyond* *k* *l* = (if *l* > *k* then *l* else 0)

lemma *beyond-zero*: *beyond* *k* 0 = 0
by (*simp* *add*: *beyond-def*)

context

includes *term-syntax*

begin

definition [*code-unfold*]:

valterm-emptyset = *Code-Evaluation.valtermify* ($\{\}$:: ('a :: *typerep*) *set*)

definition [*code-unfold*]:

valtermify-insert *x s* = *Code-Evaluation.valtermify insert* $\{\cdot\}$ (*x* :: ('a :: *typerep* * -)) $\{\cdot\}$ *s*

end

instantiation *set* :: (*random*) *random*

begin

context

includes *state-combinator-syntax*

begin

fun *random-aux-set*

where

random-aux-set 0 *j* = *collapse* (*Random.select-weight* [(1, *Pair valterm-emptyset*)])
 | *random-aux-set* (*Code-Numeral.Suc* *i*) *j* =
 collapse (*Random.select-weight*
 [(1, *Pair valterm-emptyset*),
 (*Code-Numeral.Suc* *i*,
 random j $\circ \rightarrow$ (%*x*. *random-aux-set i j* $\circ \rightarrow$ (%*s*. *Pair* (*valtermify-insert* *x s*)))))]

lemma [*code*]:

random-aux-set i j =
 collapse (*Random.select-weight* [(1, *Pair valterm-emptyset*),
 (*i*, *random j* $\circ \rightarrow$ (%*x*. *random-aux-set* (*i* - 1) *j* $\circ \rightarrow$ (%*s*. *Pair* (*valtermify-insert* *x s*)))))]

proof (*induct i rule: natural.induct*)

case *zero*

show ?*case* **by** (*subst select-weight-drop-zero* [*symmetric*])

(*simp add: random-aux-set.simps* [*simplified*] *less-natural-def*)

next

case (*Suc i*)

show ?*case* **by** (*simp only: random-aux-set.simps*(2) [*of i*] *Suc-natural-minus-one*)

qed

definition *random-set i* = *random-aux-set i i*

instance ..

end

end

lemma *random-aux-rec*:

fixes *random-aux* :: *natural* \Rightarrow *'a*

assumes *random-aux* 0 = *rhs* 0

and $\bigwedge k. \text{random-aux } (\text{Code-Numeral.Suc } k) = \text{rhs } (\text{Code-Numeral.Suc } k)$

shows *random-aux* *k* = *rhs* *k*

using *assms* **by** (*rule natural.induct*)

80.5 Deriving random generators for datatypes

ML-file \langle *Tools/Quickcheck/quickcheck-common.ML* \rangle

ML-file \langle *Tools/Quickcheck/random-generators.ML* \rangle

80.6 Code setup

code-printing

constant *random-fun-aux* \rightarrow (*Quickcheck*) *Random'-Generators.random'-fun*

— With enough criminal energy this can be abused to derive *False*; for this reason we use a distinguished target *Quickcheck* not spoiling the regular trusted code generation

code-reserved *Quickcheck Random-Generators*

hide-const (**open**) *catch-match random collapse beyond random-fun-aux random-fun-lift*

hide-fact (**open**) *collapse-def beyond-def random-fun-lift-def*

end

81 The Random-Predicate Monad

theory *Random-Pred*

imports *Quickcheck-Random*

begin

fun *iter'* :: *'a* *itself* \Rightarrow *natural* \Rightarrow *natural* \Rightarrow *Random.seed* \Rightarrow (*'a::random*) *Predicate.pred*

where

iter' *T nrandom sz seed* = (*if nrandom* = 0 *then bot-class.bot* *else*

let ((*x*, -), *seed'*) = *Quickcheck-Random.random* *sz seed*

in Predicate.Seq (*%u. Predicate.Insert* *x* (*iter'* *T* (*nrandom* - 1) *sz seed'*)))

definition *iter* :: *natural* \Rightarrow *natural* \Rightarrow *Random.seed* \Rightarrow (*'a::random*) *Predicate.pred*

where

iter *nrandom sz seed* = *iter'* (*TYPE('a)*) *nrandom sz seed*

lemma [*code*]:

```

iter nrandom sz seed = (if nrandom = 0 then bot-class.bot else
  let ((x, -), seed') = Quickcheck-Random.random sz seed
  in Predicate.Seq (%u. Predicate.Insert x (iter (nrandom - 1) sz seed')))
unfolding iter-def iter'.simps [of - nrandom] ..

```

type-synonym 'a random-pred = Random.seed \Rightarrow ('a Predicate.pred \times Random.seed)

definition empty :: 'a random-pred
where empty = Pair bot

definition single :: 'a \Rightarrow 'a random-pred
where single x = Pair (Predicate.single x)

definition bind :: 'a random-pred \Rightarrow ('a \Rightarrow 'b random-pred) \Rightarrow 'b random-pred
where
 bind R f = (λ s. let
 (P, s') = R s;
 (s1, s2) = Random.split-seed s'
 in (Predicate.bind P (%a. fst (f a s1)), s2))

definition union :: 'a random-pred \Rightarrow 'a random-pred \Rightarrow 'a random-pred
where
 union R1 R2 = (λ s. let
 (P1, s') = R1 s; (P2, s'') = R2 s'
 in (sup-class.sup P1 P2, s''))

definition if-randompred :: bool \Rightarrow unit random-pred
where
 if-randompred b = (if b then single () else empty)

definition iterate-upto :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a random-pred
where
 iterate-upto f n m = Pair (Predicate.iterate-upto f n m)

definition not-randompred :: unit random-pred \Rightarrow unit random-pred
where
 not-randompred P = (λ s. let
 (P', s') = P s
 in if Predicate.eval P' () then (Orderings.bot, s') else (Predicate.single (), s'))

definition Random :: (Random.seed \Rightarrow ('a \times (unit \Rightarrow term)) \times Random.seed) \Rightarrow 'a random-pred
where Random g = scomp g (Pair \circ (Predicate.single \circ fst))

definition map :: ('a \Rightarrow 'b) \Rightarrow 'a random-pred \Rightarrow 'b random-pred
where map f P = bind P (single \circ f)

hide-const (open) iter' iter empty single bind union if-randompred

iterate-upto not-randompred Random map

hide-fact *iter'.simps*

hide-fact (**open**) *iter-def empty-def single-def bind-def union-def
if-randompred-def iterate-upto-def not-randompred-def Random-def map-def*

end

82 Various kind of sequences inside the random monad

theory *Random-Sequence*

imports *Random-Pred*

begin

type-synonym *'a random-dseq = natural \Rightarrow natural \Rightarrow Random.seed \Rightarrow ('a Limited-Sequence.dseq \times Random.seed)*

definition *empty :: 'a random-dseq*

where

empty = (%nrandom size. Pair (Limited-Sequence.empty))

definition *single :: 'a \Rightarrow 'a random-dseq*

where

single x = (%nrandom size. Pair (Limited-Sequence.single x))

definition *bind :: 'a random-dseq \Rightarrow ('a \Rightarrow 'b random-dseq) \Rightarrow 'b random-dseq*

where

*bind R f = (λ nrandom size s. let
(P, s') = R nrandom size s;
(s1, s2) = Random.split-seed s'
in (Limited-Sequence.bind P (%a. fst (f a nrandom size s1)), s2))*

definition *union :: 'a random-dseq \Rightarrow 'a random-dseq \Rightarrow 'a random-dseq*

where

*union R1 R2 = (λ nrandom size s. let
(S1, s') = R1 nrandom size s; (S2, s'') = R2 nrandom size s'
in (Limited-Sequence.union S1 S2, s''))*

definition *if-random-dseq :: bool \Rightarrow unit random-dseq*

where

if-random-dseq b = (if b then single () else empty)

definition *not-random-dseq :: unit random-dseq \Rightarrow unit random-dseq*

where

*not-random-dseq R = (λ nrandom size s. let
(S, s') = R nrandom size s*

in (*Limited-Sequence.not-seq* S, s')

definition $map :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ random-dseq} \Rightarrow 'b \text{ random-dseq}$

where

$map\ f\ P = bind\ P\ (single \circ f)$

fun $Random :: (natural \Rightarrow Random.seed \Rightarrow (('a \times (unit \Rightarrow term)) \times Random.seed)) \Rightarrow 'a \text{ random-dseq}$

where

$Random\ g\ nrandom = (\%size.\ if\ nrandom \leq 0\ then\ (Pair\ Limited-Sequence.empty)$
 $else$

$(scomp\ (g\ size)\ (\%r.\ scomp\ (Random\ g\ (nrandom - 1)\ size)\ (\%rs.\ Pair\ (Limited-Sequence.union\ (Limited-Sequence.single\ (fst\ r))\ rs))))))$

type-synonym $'a \text{ pos-random-dseq} = natural \Rightarrow natural \Rightarrow Random.seed \Rightarrow 'a \text{ Limited-Sequence.pos-dseq}$

definition $pos-empty :: 'a \text{ pos-random-dseq}$

where

$pos-empty = (\%nrandom\ size\ seed.\ Limited-Sequence.pos-empty)$

definition $pos-single :: 'a \Rightarrow 'a \text{ pos-random-dseq}$

where

$pos-single\ x = (\%nrandom\ size\ seed.\ Limited-Sequence.pos-single\ x)$

definition $pos-bind :: 'a \text{ pos-random-dseq} \Rightarrow ('a \Rightarrow 'b \text{ pos-random-dseq}) \Rightarrow 'b \text{ pos-random-dseq}$

where

$pos-bind\ R\ f = (\lambda nrandom\ size\ seed.\ Limited-Sequence.pos-bind\ (R\ nrandom\ size\ seed)\ (\%a.\ f\ a\ nrandom\ size\ seed))$

definition $pos-decr-bind :: 'a \text{ pos-random-dseq} \Rightarrow ('a \Rightarrow 'b \text{ pos-random-dseq}) \Rightarrow 'b \text{ pos-random-dseq}$

where

$pos-decr-bind\ R\ f = (\lambda nrandom\ size\ seed.\ Limited-Sequence.pos-decr-bind\ (R\ nrandom\ size\ seed)\ (\%a.\ f\ a\ nrandom\ size\ seed))$

definition $pos-union :: 'a \text{ pos-random-dseq} \Rightarrow 'a \text{ pos-random-dseq} \Rightarrow 'a \text{ pos-random-dseq}$

where

$pos-union\ R1\ R2 = (\lambda nrandom\ size\ seed.\ Limited-Sequence.pos-union\ (R1\ nrandom\ size\ seed)\ (R2\ nrandom\ size\ seed))$

definition $pos-if-random-dseq :: bool \Rightarrow unit \text{ pos-random-dseq}$

where

$pos-if-random-dseq\ b = (if\ b\ then\ pos-single\ ()\ else\ pos-empty)$

definition $pos-iterate-upto :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a \text{ pos-random-dseq}$

where

pos-iterate-upto f n m = (λnrandom size seed. Limited-Sequence.pos-iterate-upto f n m)

definition *pos-map :: ('a => 'b) => 'a pos-random-dseq => 'b pos-random-dseq*

where

pos-map f P = pos-bind P (pos-single ∘ f)

fun *iter :: (Random.seed => ('a × (unit => term))) × Random.seed*

=> natural => Random.seed => 'a Lazy-Sequence.lazy-sequence

where

iter random nrandom seed =

(if nrandom = 0 then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence (%u. let ((x, -), seed') = random seed in Some (x, iter random (nrandom - 1) seed')))

definition *pos-Random :: (natural => Random.seed => ('a × (unit => term))) × Random.seed*

=> 'a pos-random-dseq

where

pos-Random g = (%nrandom size seed depth. iter (g size) nrandom seed)

type-synonym *'a neg-random-dseq = natural => natural => Random.seed => 'a Limited-Sequence.neg-dseq*

definition *neg-empty :: 'a neg-random-dseq*

where

neg-empty = (%nrandom size seed. Limited-Sequence.neg-empty)

definition *neg-single :: 'a => 'a neg-random-dseq*

where

neg-single x = (%nrandom size seed. Limited-Sequence.neg-single x)

definition *neg-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq*

where

neg-bind R f = (λnrandom size seed. Limited-Sequence.neg-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *neg-decr-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq*

where

neg-decr-bind R f = (λnrandom size seed. Limited-Sequence.neg-decr-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *neg-union :: 'a neg-random-dseq => 'a neg-random-dseq => 'a neg-random-dseq*

where

neg-union R1 R2 = (λnrandom size seed. Limited-Sequence.neg-union (R1 nran-

dom size seed) (*R2 nrandom size seed*))

definition *neg-if-random-dseq* :: *bool* => *unit neg-random-dseq*

where

neg-if-random-dseq *b* = (*if b then neg-single () else neg-empty*)

definition *neg-iterate-upto* :: (*natural* => '*a*) => *natural* => *natural* => '*a*
neg-random-dseq

where

neg-iterate-upto *f n m* = ($\lambda nrandom\ size\ seed.$ *Limited-Sequence.neg-iterate-upto*
f n m)

definition *neg-not-random-dseq* :: *unit pos-random-dseq* => *unit neg-random-dseq*

where

neg-not-random-dseq *R* = ($\lambda nrandom\ size\ seed.$ *Limited-Sequence.neg-not-seq* (*R*
nrandom size seed))

definition *neg-map* :: ('*a* => '*b*) => '*a* *neg-random-dseq* => '*b* *neg-random-dseq*

where

neg-map *f P* = *neg-bind* *P* (*neg-single* \circ *f*)

definition *pos-not-random-dseq* :: *unit neg-random-dseq* => *unit pos-random-dseq*

where

pos-not-random-dseq *R* = ($\lambda nrandom\ size\ seed.$ *Limited-Sequence.pos-not-seq* (*R*
nrandom size seed))

hide-const (**open**)

empty single bind union if-random-dseq not-random-dseq map Random
pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-random-dseq pos-iterate-upto
pos-not-random-dseq pos-map iter pos-Random
neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-random-dseq neg-iterate-upto
neg-not-random-dseq neg-map

hide-fact (**open**) *empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def*

map-def Random.simps
pos-empty-def pos-single-def pos-bind-def pos-decr-bind-def pos-union-def pos-if-random-dseq-def
pos-iterate-upto-def pos-not-random-dseq-def pos-map-def iter.simps pos-Random-def
neg-empty-def neg-single-def neg-bind-def neg-decr-bind-def neg-union-def neg-if-random-dseq-def
neg-iterate-upto-def neg-not-random-dseq-def neg-map-def

end

83 A simple counterexample generator performing exhaustive testing

theory *Quickcheck-Exhaustive*

imports *Quickcheck-Random*

keywords *quickcheck-generator* :: *thy-decl*
begin

83.1 Basic operations for exhaustive generators

definition *orelse* :: *'a option* \Rightarrow *'a option* \Rightarrow *'a option* (**infixr** *orelse* 55)
where [*code-unfold*]: *x orelse y* = (*case x of Some x' \Rightarrow Some x' | None \Rightarrow y*)

83.2 Exhaustive generator type classes

class *exhaustive* = *term-of* +
fixes *exhaustive* :: (*'a* \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*

class *full-exhaustive* = *term-of* +
fixes *full-exhaustive* ::
(*'a* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*

instantiation *natural* :: *full-exhaustive*
begin

function *full-exhaustive-natural'* ::
(*natural* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow
natural \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*
where *full-exhaustive-natural' f d i* =
(*if d < i then None*
else (f (i, λ -. Code-Evaluation.term-of i)) orelse (full-exhaustive-natural' f d (i + 1))))
by *pat-completeness auto*

termination
by (*relation measure* ($\lambda(-, d, i). \text{nat-of-natural } (d + 1 - i)$)) (*auto simp add: less-natural-def*)

definition *full-exhaustive f d* = *full-exhaustive-natural' f d 0*

instance ..

end

instantiation *natural* :: *exhaustive*
begin

function *exhaustive-natural'* ::
(*natural* \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*
where *exhaustive-natural' f d i* =
(*if d < i then None*
else (f i orelse exhaustive-natural' f d (i + 1)))

by *pat-completeness auto*

termination

by (*relation measure* ($\lambda(-, d, i). \text{nat-of-natural } (d + 1 - i)$)) (*auto simp add: less-natural-def*)

definition *exhaustive f d = exhaustive-natural' f d 0*

instance ..

end

instantiation *integer :: exhaustive*

begin

function *exhaustive-integer' ::*

(integer \Rightarrow (bool \times term list) option) \Rightarrow integer \Rightarrow integer \Rightarrow (bool \times term list) option

where *exhaustive-integer' f d i =*

(if d < i then None else (f i orelse exhaustive-integer' f d (i + 1)))

by *pat-completeness auto*

termination

by (*relation measure* ($\lambda(-, d, i). \text{nat-of-integer } (d + 1 - i)$))

(auto simp add: less-integer-def nat-of-integer-def)

definition *exhaustive f d = exhaustive-integer' f (integer-of-natural d) (- (integer-of-natural d))*

instance ..

end

instantiation *integer :: full-exhaustive*

begin

function *full-exhaustive-integer' ::*

(integer \times (unit \Rightarrow term) \Rightarrow (bool \times term list) option) \Rightarrow

integer \Rightarrow integer \Rightarrow (bool \times term list) option

where *full-exhaustive-integer' f d i =*

(if d < i then None

else

(case f (i, $\lambda-. \text{Code-Evaluation.term-of } i$) of

Some t \Rightarrow Some t

| None \Rightarrow full-exhaustive-integer' f d (i + 1)))

by *pat-completeness auto*

termination

by (*relation measure* ($\lambda(-, d, i). \text{nat-of-integer } (d + 1 - i)$))

(*auto simp add: less-integer-def nat-of-integer-def*)

definition *full-exhaustive* $f\ d =$
full-exhaustive-integer' $f\ (\text{integer-of-natural } d)\ (-\ (\text{integer-of-natural } d))$

instance ..

end

instantiation *nat* :: *exhaustive*
begin

definition *exhaustive* $f\ d = \text{exhaustive } (\lambda x. f\ (\text{nat-of-natural } x))\ d$

instance ..

end

instantiation *nat* :: *full-exhaustive*
begin

definition *full-exhaustive* $f\ d =$
full-exhaustive $(\lambda(x, xt). f\ (\text{nat-of-natural } x, \lambda-. \text{Code-Evaluation.term-of } (\text{nat-of-natural } x)))\ d$

instance ..

end

instantiation *int* :: *exhaustive*
begin

function *exhaustive-int'* ::
 $(\text{int} \Rightarrow (\text{bool} \times \text{term list}) \text{ option}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow (\text{bool} \times \text{term list}) \text{ option}$
where *exhaustive-int'* $f\ d\ i =$
 $(\text{if } d < i \text{ then None else } (f\ i \text{ orelse } \text{exhaustive-int}'\ f\ d\ (i + 1)))$
by *pat-completeness auto*

termination

by $(\text{relation measure } (\lambda(-, d, i). \text{nat } (d + 1 - i)))\ \text{auto}$

definition *exhaustive* $f\ d =$
exhaustive-int' $f\ (\text{int-of-integer } (\text{integer-of-natural } d))$
 $(-\ (\text{int-of-integer } (\text{integer-of-natural } d)))$

instance ..

end

instantiation *int* :: *full-exhaustive*
begin

function *full-exhaustive-int'* ::
 (*int* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
int ⇒ *int* ⇒ (*bool* × *term list*) *option*
where *full-exhaustive-int' f d i* =
 (*if d < i then None*
else
 (*case f (i, λ-. Code-Evaluation.term-of i) of*
 Some t ⇒ Some t
 | *None ⇒ full-exhaustive-int' f d (i + 1)*))
by *pat-completeness auto*

termination
by (*relation measure (λ(-, d, i). nat (d + 1 - i)) auto*)

definition *full-exhaustive f d* =
full-exhaustive-int' f (int-of-integer (integer-of-natural d))
 (*- (int-of-integer (integer-of-natural d))*)

instance ..

end

instantiation *prod* :: (*exhaustive, exhaustive*) *exhaustive*
begin

definition *exhaustive f d* = *exhaustive (λx. exhaustive (λy. f ((x, y))) d) d*

instance ..

end

context
includes *term-syntax*
begin

definition
 [*code-unfold*]: *valtermify-pair x y* =
Code-Evaluation.valtermify (Pair :: 'a::typerep ⇒ 'b::typerep ⇒ 'a × 'b) {·} x
 {·} *y*

end

instantiation *prod* :: (*full-exhaustive, full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive f d* =

full-exhaustive ($\lambda x. \text{full-exhaustive } (\lambda y. f \text{ (valtermify-pair } x \ y)) \ d) \ d$

instance ..

end

instantiation *set* :: (*exhaustive*) *exhaustive*
begin

fun *exhaustive-set*

where

exhaustive-set *f* *i* =

(if *i* = 0 then None

else

f {} orelse

exhaustive-set

($\lambda A. f \ A \ \text{orelse} \ \text{exhaustive } (\lambda x. \text{if } x \in A \ \text{then} \ \text{None} \ \text{else} \ f \ (\text{insert } x \ A)) \ (i - 1)) \ (i - 1))$)

instance ..

end

instantiation *set* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-set*

where

full-exhaustive-set *f* *i* =

(if *i* = 0 then None

else

f *valterm-emptyset* orelse

full-exhaustive-set

($\lambda A. f \ A \ \text{orelse} \ \text{Quickcheck-Exhaustive.full-exhaustive}$

($\lambda x. \text{if } \text{fst } x \in \text{fst } A \ \text{then} \ \text{None} \ \text{else} \ f \ (\text{valtermify-insert } x \ A)) \ (i - 1)) \ (i - 1))$)

instance ..

end

instantiation *fun* :: ({*equal*,*exhaustive*}, *exhaustive*) *exhaustive*
begin

fun *exhaustive-fun'* ::

((*a* \Rightarrow *b*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*

where

exhaustive-fun' *f* *i* *d* =

```

    (exhaustive (λb. f (λ-. b)) d) orelse
      (if i > 1 then
        exhaustive-fun'
          (λg. exhaustive (λa. exhaustive (λb. f (g(a := b))) d) d) (i - 1) d else
None)

```

definition *exhaustive-fun* ::

```

((('a ⇒ 'b) ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option
  where exhaustive-fun f d = exhaustive-fun' f d d

```

instance ..

end

definition [*code-unfold*]:

```

valtermify-absdummy =
  (λ(v, t).
    (λ::'a. v,
      λu::unit. Code-Evaluation.Abs (STR "x") (Typerep.typerep TYPE('a::typerep))
    (t ())))

```

context

includes *term-syntax*

begin

definition

```

[code-unfold]: valtermify-fun-upd g a b =
  Code-Evaluation.valtermify
    (fun-upd :: ('a::typerep ⇒ 'b::typerep) ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'b) {·} g {·} a {·} b

```

end

instantiation *fun* :: ({*equal,full-exhaustive*}, *full-exhaustive*) *full-exhaustive*

begin

fun *full-exhaustive-fun'* ::

```

((('a ⇒ 'b) × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
  natural ⇒ natural ⇒ (bool × term list) option

```

where

```

full-exhaustive-fun' f i d =
  full-exhaustive (λv. f (valtermify-absdummy v)) d orelse
    (if i > 1 then
      full-exhaustive-fun'
        (λg. full-exhaustive
          (λa. full-exhaustive (λb. f (valtermify-fun-upd g a b)) d) d) (i - 1) d
    else None)

```

definition *full-exhaustive-fun* ::

```

((('a ⇒ 'b) × (unit ⇒ term) ⇒ (bool × term list) option) ⇒

```



```

    natural ⇒ (bool × term list) option
  where full-exhaustive-fun f d = full-exhaustive-fun' f d d

```

```
instance ..
```

```
end
```

83.2.1 A smarter enumeration scheme for functions over finite datatypes

```

class check-all = enum + term-of +
  fixes check-all :: ('a × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ (bool *
term list) option
  fixes enum-term-of :: 'a itself ⇒ unit ⇒ term list

```

```

fun check-all-n-lists :: ('a::check-all list × (unit ⇒ term list) ⇒
(bool × term list) option) ⇒ natural ⇒ (bool * term list) option

```

```
where
```

```

  check-all-n-lists f n =
    (if n = 0 then f ([], (λ-. []))
     else check-all (λ(x, xt).
      check-all-n-lists (λ(xs, xst). f ((x # xs), (λ-. (xt () # xst ()))) (n - 1)))

```

```
context
```

```
  includes term-syntax
```

```
begin
```

```
definition
```

```

[code-unfold]: termify-fun-upd g a b =
  (Code-Evaluation.termify
   (fun-upd :: ('a::typerep ⇒ 'b::typerep) ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'b) <.> g <.> a
  <.> b)

```

```
end
```

```
definition mk-map-term ::
```

```

(unit ⇒ typerep) ⇒ (unit ⇒ typerep) ⇒
(unit ⇒ term list) ⇒ (unit ⇒ term list) ⇒ unit ⇒ term

```

```
where mk-map-term T1 T2 domm rng =
```

```
(λ-.

```

```
  let
```

```
    T1 = T1 ();
```

```
    T2 = T2 ();
```

```
    update-term =
```

```
      (λg (a, b).
```

```
        Code-Evaluation.App (Code-Evaluation.App (Code-Evaluation.App
```

```
          (Code-Evaluation.Const (STR "Fun.fun-upd"))
```

```
            (Typerep.Typerep (STR "fun")) [Typerep.Typerep (STR "fun") [T1,
```

```
T2],
```

```

      Typerep.Typerep (STR "fun'") [T1,
      Typerep.Typerep (STR "fun'") [T2, Typerep.Typerep (STR "fun'")
[T1, T2]]]]))
      g) a) b)
in
  List.foldl update-term
    (Code-Evaluation.Abs (STR "x'") T1
     (Code-Evaluation.Const (STR "HOL.undefin'" T2)) (zip (domm ())
(rng ())))

```

instantiation *fun* :: ({*equal*,*check-all*}, *check-all*) *check-all*
begin

definition

```

check-all f =
  (let
    mk-term =
      mk-map-term
        (λ-. Typerep.typerep (TYPE('a)))
        (λ-. Typerep.typerep (TYPE('b)))
        (enum-term-of (TYPE('a)));
    enum = (Enum.enum :: 'a list)
  in
    check-all-n-lists
      (λ(ys, yst). f (the ∘ map-of (zip enum ys), mk-term yst))
      (natural-of-nat (length enum)))

```

definition *enum-term-of-fun* :: ('a ⇒ 'b) *itself* ⇒ *unit* ⇒ *term list*

```

where enum-term-of-fun =
  (λ- -.
    let
      enum-term-of-a = enum-term-of (TYPE('a));
      mk-term =
        mk-map-term
          (λ-. Typerep.typerep (TYPE('a)))
          (λ-. Typerep.typerep (TYPE('b)))
          enum-term-of-a
    in
      map (λys. mk-term (λ-. ys) ())
          (List.n-lists (length (enum-term-of-a ())) (enum-term-of (TYPE('b)) ())))

```

instance ..

end

context

includes *term-syntax*

begin

```

fun check-all-subsets ::
  (('a::typerep) set × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
  ('a × (unit ⇒ term)) list ⇒ (bool × term list) option
where
  check-all-subsets f [] = f valterm-emptyset
| check-all-subsets f (x # xs) =
  check-all-subsets (λs. case f s of Some ts ⇒ Some ts | None ⇒ f (valtermify-insert
x s)) xs

```

definition

```
[code-unfold]: term-emptyset = Code-Evaluation.termify ({} :: ('a::typerep) set)
```

definition

```
[code-unfold]: termify-insert x s =
  Code-Evaluation.termify (insert :: ('a::typerep) ⇒ 'a set ⇒ 'a set) <·> x <·>
s
```

```
definition setify :: ('a::typerep) itself ⇒ term list ⇒ term
```

where

```
setify T ts = foldr (termify-insert T) ts (term-emptyset T)
```

end

```
instantiation set :: (check-all) check-all
```

begin**definition**

```
check-all-set f =
  check-all-subsets f
  (zip (Enum.enum :: 'a list)
  (map (λa. λu :: unit. a) (Quickcheck-Exhaustive.enum-term-of (TYPE ('a))
  ())))
```

```
definition enum-term-of-set :: 'a set itself ⇒ unit ⇒ term list
```

```
where enum-term-of-set - - =
```

```
map (setify (TYPE('a))) (subseqs (Quickcheck-Exhaustive.enum-term-of (TYPE('a))
  ()))
```

instance ..**end**

```
instantiation unit :: check-all
```

begin

```
definition check-all f = f (Code-Evaluation.valtermify ())
```

```
definition enum-term-of-unit :: unit itself ⇒ unit ⇒ term list
```

```
where enum-term-of-unit = (λ- -. [Code-Evaluation.term-of ()])
```

instance ..

end

instantiation *bool* :: *check-all*

begin

definition

check-all *f* =
 (case *f* (Code-Evaluation.valtermify False) of
 Some *x'* ⇒ Some *x'*
 | None ⇒ *f* (Code-Evaluation.valtermify True))

definition *enum-term-of-bool* :: *bool* itself ⇒ *unit* ⇒ *term list*

where *enum-term-of-bool* = (λ- -. map Code-Evaluation.term-of (Enum.enum :: *bool list*))

instance ..

end

context

includes *term-syntax*

begin

definition [*code-unfold*]:

termify-pair *x y* =
 Code-Evaluation.termify (Pair :: 'a::typerep ⇒ 'b :: typerep ⇒ 'a * 'b) <·> *x*
 <·> *y*

end

instantiation *prod* :: (*check-all*, *check-all*) *check-all*

begin

definition *check-all* *f* = *check-all* (λ*x*. *check-all* (λ*y*. *f* (valtermify-pair *x y*)))

definition *enum-term-of-prod* :: ('a * 'b) itself ⇒ *unit* ⇒ *term list*

where *enum-term-of-prod* =
 (λ- -.
 map (λ(*x*, *y*). termify-pair TYPE('a) TYPE('b) *x y*)
 (List.product (enum-term-of (TYPE('a)) ()) (enum-term-of (TYPE('b))
 ())))

instance ..

end

context

includes *term-syntax*

begin

definition

[code-unfold]: *valtermify-Inl* $x =$
 $Code-Evaluation.valtermify (Inl :: 'a::typerep \Rightarrow 'a + 'b :: typerep) \{\cdot\} x$

definition

[code-unfold]: *valtermify-Inr* $x =$
 $Code-Evaluation.valtermify (Inr :: 'b::typerep \Rightarrow 'a::typerep + 'b) \{\cdot\} x$

end

instantiation *sum* :: (*check-all*, *check-all*) *check-all*

begin

definition

check-all $f = check-all (\lambda a. f (valtermify-Inl a)) \text{ or else } check-all (\lambda b. f (valtermify-Inr b))$

definition *enum-term-of-sum* :: (*a* + *b*) *itself* $\Rightarrow unit \Rightarrow term\ list$

where *enum-term-of-sum* =

($\lambda - .$

let

$T1 = Typerep.typerep (TYPE('a));$

$T2 = Typerep.typerep (TYPE('b))$

in

map

($Code-Evaluation.App (Code-Evaluation.Const (STR "Sum-Type.Inl")$

($Typerep.Typerep (STR "fun") [T1, Typerep.Typerep (STR "Sum-Type.sum")$

$[T1, T2]]))$)

($enum-term-of (TYPE('a)) () @$

map

($Code-Evaluation.App (Code-Evaluation.Const (STR "Sum-Type.Inr")$

($Typerep.Typerep (STR "fun") [T2, Typerep.Typerep (STR "Sum-Type.sum")$

$[T1, T2]]))$)

($enum-term-of (TYPE('b)) ()$)

instance ..

end

instantiation *char* :: *check-all*

begin

primrec *check-all-char'* ::

($char \times (unit \Rightarrow term) \Rightarrow (bool \times term\ list) \text{ option} \Rightarrow char\ list \Rightarrow (bool \times term$

list) option

where *check-all-char' f []* = None
 | *check-all-char' f (c # cs)* = *f (c, λ-. Code-Evaluation.term-of c)*
 orelse check-all-char' f cs

definition *check-all-char* ::

(*char* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) option) ⇒ (*bool* × *term list*) option
where *check-all f* = *check-all-char' f Enum.enum*

definition *enum-term-of-char* :: *char itself* ⇒ *unit* ⇒ *term list*

where

enum-term-of-char = (λ- -. *map Code-Evaluation.term-of (Enum.enum :: char list)*)

instance ..

end

instantiation *option* :: (*check-all*) *check-all*

begin

definition

check-all f =
f (Code-Evaluation.valtermify (None :: 'a option)) orelse
check-all
 (λ(*x, t*).
 f
 (*Some x,*
 λ-. *Code-Evaluation.App*
 (*Code-Evaluation.Const (STR "Option.option.Some"*)
 (*Typerep.Typerep (STR "fun"*)
 [*Typerep.typerep TYPE('a),*
 Typerep.Typerep (STR "Option.option") [Typerep.typerep TYPE('a)]]))))
 (*t* ()))

definition *enum-term-of-option* :: *'a option itself* ⇒ *unit* ⇒ *term list*

where *enum-term-of-option* =

(λ- -.
Code-Evaluation.term-of (None :: 'a option) #
 (*map*
 (*Code-Evaluation.App*
 (*Code-Evaluation.Const (STR "Option.option.Some"*)
 (*Typerep.Typerep (STR "fun"*)
 [*Typerep.typerep TYPE('a),*
 Typerep.Typerep (STR "Option.option") [Typerep.typerep TYPE('a)]]))))
 (*enum-term-of (TYPE('a))* ())))

instance ..

end

instantiation *Enum.finite-1* :: *check-all*
begin

definition *check-all f* = *f (Code-Evaluation.valtermify Enum.finite-1.a₁)*

definition *enum-term-of-finite-1* :: *Enum.finite-1* itself ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-1* = (λ- -. [*Code-Evaluation.term-of Enum.finite-1.a₁*])

instance ..

end

instantiation *Enum.finite-2* :: *check-all*
begin

definition
check-all f =
(f (Code-Evaluation.valtermify Enum.finite-2.a₁) *orelse*
f (Code-Evaluation.valtermify Enum.finite-2.a₂))

definition *enum-term-of-finite-2* :: *Enum.finite-2* itself ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-2* =
(λ- -. *map Code-Evaluation.term-of (Enum.enum :: Enum.finite-2 list)*)

instance ..

end

instantiation *Enum.finite-3* :: *check-all*
begin

definition
check-all f =
(f (Code-Evaluation.valtermify Enum.finite-3.a₁) *orelse*
f (Code-Evaluation.valtermify Enum.finite-3.a₂) *orelse*
f (Code-Evaluation.valtermify Enum.finite-3.a₃))

definition *enum-term-of-finite-3* :: *Enum.finite-3* itself ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-3* =
(λ- -. *map Code-Evaluation.term-of (Enum.enum :: Enum.finite-3 list)*)

instance ..

end

instantiation *Enum.finite-4* :: *check-all*

begin

definition

```
check-all f =
  f (Code-Evaluation.valtermify Enum.finite-4.a1) orelse
  f (Code-Evaluation.valtermify Enum.finite-4.a2) orelse
  f (Code-Evaluation.valtermify Enum.finite-4.a3) orelse
  f (Code-Evaluation.valtermify Enum.finite-4.a4)
```

definition *enum-term-of-finite-4* :: *Enum.finite-4* itself \Rightarrow unit \Rightarrow term list

```
where enum-term-of-finite-4 =
  ( $\lambda$ - . map Code-Evaluation.term-of (Enum.enum :: Enum.finite-4 list))
```

instance ..

end

83.3 Bounded universal quantifiers

class *bounded-forall* =

```
fixes bounded-forall :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  natural  $\Rightarrow$  bool
```

83.4 Fast exhaustive combinators

class *fast-exhaustive* = term-of +

```
fixes fast-exhaustive :: ('a  $\Rightarrow$  unit)  $\Rightarrow$  natural  $\Rightarrow$  unit
```

axiomatization *throw-Counterexample* :: term list \Rightarrow unit

axiomatization *catch-Counterexample* :: unit \Rightarrow term list option

code-printing

```
constant throw-Counterexample  $\rightarrow$ 
```

```
(Quickcheck) raise (Exhaustive'-Generators.Counterexample -)
```

```
| constant catch-Counterexample  $\rightarrow$ 
```

```
(Quickcheck) (((-); NONE) handle Exhaustive'-Generators.Counterexample ts
 $\Rightarrow$  SOME ts)
```

83.5 Continuation passing style functions as plus monad

type-synonym 'a cps = ('a \Rightarrow term list option) \Rightarrow term list option

definition *cps-empty* :: 'a cps

```
where cps-empty = ( $\lambda$ cont. None)
```

definition *cps-single* :: 'a \Rightarrow 'a cps

```
where cps-single v = ( $\lambda$ cont. cont v)
```

definition *cps-bind* :: 'a cps \Rightarrow ('a \Rightarrow 'b cps) \Rightarrow 'b cps

```
where cps-bind m f = ( $\lambda$ cont. m ( $\lambda$ a. (f a) cont))
```


definition *cps-plus* :: 'a cps ⇒ 'a cps ⇒ 'a cps

where *cps-plus* a b = (λc. case a c of None ⇒ b c | Some x ⇒ Some x)

definition *cps-if* :: bool ⇒ unit cps

where *cps-if* b = (if b then *cps-single* () else *cps-empty*)

definition *cps-not* :: unit cps ⇒ unit cps

where *cps-not* n = (λc. case n (λu. Some []) of None ⇒ c () | Some - ⇒ None)

type-synonym 'a *pos-bound-cps* =

('a ⇒ (bool * term list) option) ⇒ natural ⇒ (bool * term list) option

definition *pos-bound-cps-empty* :: 'a *pos-bound-cps*

where *pos-bound-cps-empty* = (λcont i. None)

definition *pos-bound-cps-single* :: 'a ⇒ 'a *pos-bound-cps*

where *pos-bound-cps-single* v = (λcont i. cont v)

definition *pos-bound-cps-bind* :: 'a *pos-bound-cps* ⇒ ('a ⇒ 'b *pos-bound-cps*) ⇒ 'b *pos-bound-cps*

where *pos-bound-cps-bind* m f = (λcont i. if i = 0 then None else (m (λa. (f a) cont i) (i - 1)))

definition *pos-bound-cps-plus* :: 'a *pos-bound-cps* ⇒ 'a *pos-bound-cps* ⇒ 'a *pos-bound-cps*

where *pos-bound-cps-plus* a b = (λc i. case a c i of None ⇒ b c i | Some x ⇒ Some x)

definition *pos-bound-cps-if* :: bool ⇒ unit *pos-bound-cps*

where *pos-bound-cps-if* b = (if b then *pos-bound-cps-single* () else *pos-bound-cps-empty*)

datatype (*plugins only: code extraction*) (dead 'a) *unknown* =

Unknown | *Known* 'a

datatype (*plugins only: code extraction*) (dead 'a) *three-valued* =

Unknown-value | *Value* 'a | *No-value*

type-synonym 'a *neg-bound-cps* =

('a *unknown* ⇒ term list *three-valued*) ⇒ natural ⇒ term list *three-valued*

definition *neg-bound-cps-empty* :: 'a *neg-bound-cps*

where *neg-bound-cps-empty* = (λcont i. *No-value*)

definition *neg-bound-cps-single* :: 'a ⇒ 'a *neg-bound-cps*

where *neg-bound-cps-single* v = (λcont i. cont (*Known* v))

definition *neg-bound-cps-bind* :: 'a *neg-bound-cps* ⇒ ('a ⇒ 'b *neg-bound-cps*) ⇒ 'b *neg-bound-cps*

where *neg-bound-cps-bind* m f =
(λcont i.

$$\begin{aligned} & \text{if } i = 0 \text{ then cont Unknown} \\ & \text{else } m (\lambda a. \text{ case } a \text{ of Unknown } \Rightarrow \text{ cont Unknown} \mid \text{Known } a' \Rightarrow f a' \text{ cont } i) \\ & (i - 1)) \end{aligned}$$

definition *neg-bound-cps-plus* :: 'a *neg-bound-cps* \Rightarrow 'a *neg-bound-cps* \Rightarrow 'a *neg-bound-cps*
where *neg-bound-cps-plus* a b =
 (λc i.
 case a c i of
 No-value \Rightarrow b c i
 | *Value* x \Rightarrow *Value* x
 | *Unknown-value* \Rightarrow
 (*case* b c i of
 No-value \Rightarrow *Unknown-value*
 | *Value* x \Rightarrow *Value* x
 | *Unknown-value* \Rightarrow *Unknown-value*))

definition *neg-bound-cps-if* :: bool \Rightarrow unit *neg-bound-cps*
where *neg-bound-cps-if* b = (if b then *neg-bound-cps-single* () else *neg-bound-cps-empty*)

definition *neg-bound-cps-not* :: unit *pos-bound-cps* \Rightarrow unit *neg-bound-cps*
where *neg-bound-cps-not* n =
 (λc i. *case* n (λu . *Some* (*True*, [])) i of *None* \Rightarrow c (*Known* ()) | *Some* - \Rightarrow *No-value*)

definition *pos-bound-cps-not* :: unit *neg-bound-cps* \Rightarrow unit *pos-bound-cps*
where *pos-bound-cps-not* n =
 (λc i. *case* n (λu . *Value* []) i of *No-value* \Rightarrow c () | *Value* - \Rightarrow *None* | *Unknown-value* \Rightarrow *None*)

83.6 Defining generators for any first-order data type

axiomatization *unknown* :: 'a

notation (output) *unknown* (?)

ML-file \langle Tools/Quickcheck/exhaustive-generators.ML \rangle

declare [[*quickcheck-batch-tester* = *exhaustive*]]

83.7 Defining generators for abstract types

ML-file \langle Tools/Quickcheck/abstract-generators.ML \rangle

hide-fact (open) *orelse-def*

no-notation *orelse* (infixr *orelse* 55)

hide-const *valtermify-absdummy* *valtermify-fun-upd*

valterm-emptyset *valtermify-insert*

valtermify-pair *valtermify-Inl* *valtermify-Inr*

termify-fun-upd *term-emptyset* *termify-insert* *termify-pair* *setify*

hide-const (open)

*exhaustive full-exhaustive
 exhaustive-int' full-exhaustive-int'
 exhaustive-integer' full-exhaustive-integer'
 exhaustive-natural' full-exhaustive-natural'
 throw-Counterexample catch-Counterexample
 check-all enum-term-of
 orelse unknown mk-map-term check-all-n-lists check-all-subsets*

hide-type (open) *cps pos-bound-cps neg-bound-cps unknown three-valued*

hide-const (open) *cps-empty cps-single cps-bind cps-plus cps-if cps-not
 pos-bound-cps-empty pos-bound-cps-single pos-bound-cps-bind
 pos-bound-cps-plus pos-bound-cps-if pos-bound-cps-not
 neg-bound-cps-empty neg-bound-cps-single neg-bound-cps-bind
 neg-bound-cps-plus neg-bound-cps-if neg-bound-cps-not
 Unknown Known Unknown-value Value No-value*

end

84 A compiler for predicates defined by introduction rules

theory *Predicate-Compile*

imports *Random-Sequence Quickcheck-Exhaustive*

keywords

*code-pred :: thy-goal and
 values :: diag*

begin

ML-file *<Tools/Predicate-Compile/predicate-compile-aux.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile-compilations.ML>*

ML-file *<Tools/Predicate-Compile/core-data.ML>*

ML-file *<Tools/Predicate-Compile/mode-inference.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile-proof.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile-core.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile-data.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile-fun.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile-pred.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile-specialisation.ML>*

ML-file *<Tools/Predicate-Compile/predicate-compile.ML>*

84.1 Set membership as a generator predicate

Introduce a new constant for membership to allow fine-grained control in code equations.

definition *contains :: 'a set => 'a => bool*

where *contains* A $x \longleftrightarrow x \in A$

definition *contains-pred* :: 'a set => 'a => unit Predicate.pred
where *contains-pred* A $x =$ (if $x \in A$ then Predicate.single () else bot)

lemma *pred-of-setE*:
assumes Predicate.eval (pred-of-set A) x
obtains *contains* A x
using *assms* **by**(simp add: contains-def)

lemma *pred-of-setI*: *contains* A $x ==>$ Predicate.eval (pred-of-set A) x
by(simp add: contains-def)

lemma *pred-of-set-eq*: *pred-of-set* $\equiv \lambda A.$ Predicate.Pred (*contains* A)
by(simp add: contains-def[abs-def] pred-of-set-def o-def)

lemma *containsI*: $x \in A ==>$ *contains* A x
by(simp add: contains-def)

lemma *containsE*: **assumes** *contains* A x
obtains A' x' **where** $A = A'$ $x = x'$ $x \in A$
using *assms* **by**(simp add: contains-def)

lemma *contains-predI*: *contains* A $x ==>$ Predicate.eval (*contains-pred* A x) ()
by(simp add: contains-pred-def contains-def)

lemma *contains-predE*:
assumes Predicate.eval (*contains-pred* A x) y
obtains *contains* A x
using *assms* **by**(simp add: contains-pred-def contains-def split: if-split-asm)

lemma *contains-pred-eq*: *contains-pred* $\equiv \lambda A$ $x.$ Predicate.Pred ($\lambda y.$ *contains* A x)
by(rule eq-reflection)(auto simp add: contains-pred-def fun-eq-iff contains-def intro: pred-eqI)

lemma *contains-pred-notI*:
 \neg *contains* A $x ==>$ Predicate.eval (Predicate.not-pred (*contains-pred* A x)) ()
by(simp add: contains-pred-def contains-def not-pred-eq)

setup <
let
val *Fun* = Predicate-Compile-Aux.Fun
val *Input* = Predicate-Compile-Aux.Input
val *Output* = Predicate-Compile-Aux.Output
val *Bool* = Predicate-Compile-Aux.Bool
val *io* = Fun (*Input*, Fun (*Output*, *Bool*))
val *ii* = Fun (*Input*, Fun (*Input*, *Bool*))
in
Core-Data.PredData.map (Graph.new-node

```

(const-name ‹contains›,
 Core-Data.PredData {
   pos = Position.thread-data (),
   intros = [(NONE, @ {thm containsI})],
   elim = SOME @ {thm containsE},
   preprocessed = true,
   function-names = [(Predicate-Compile-Aux.Pred,
     [(io, const-name ‹pred-of-set›), (ii, const-name ‹contains-pred›)]),
   predfun-data = [
     (io, Core-Data.PredfunData {
       elim = @ {thm pred-of-setE}, intro = @ {thm pred-of-setI},
       neg-intro = NONE, definition = @ {thm pred-of-set-eq}
     }),
     (ii, Core-Data.PredfunData {
       elim = @ {thm contains-predE}, intro = @ {thm contains-predI},
       neg-intro = SOME @ {thm contains-pred-notI}, definition = @ {thm
contains-pred-eq}
     })],
   needs-random = []])
end
›

```

```

hide-const (open) contains contains-pred
hide-fact (open) pred-of-setE pred-of-setI pred-of-set-eq
  containsI containsE contains-predI contains-predE contains-pred-eq contains-pred-notI
end

```

85 Counterexample generator performing narrowing-based testing

```

theory Quickcheck-Narrowing
imports Quickcheck-Random
keywords find-unused-assms :: diag
begin

```

85.1 Counterexample generator

85.1.1 Code generation setup

```

setup ‹Code-Target.add-derived-target (Haskell-Quickcheck, [(Code-Haskell.target,
I)])›

```

code-printing

```

code-module Typerep  $\rightarrow$  (Haskell-Quickcheck) ‹
module Typerep(Typerep(..)) where

```

```

data Typerep = Typerep String [Typerep]
› for type-constructor typerep constant Typerep.Typerep

```

```
| type-constructor typerep  $\rightarrow$  (Haskell-Quickcheck) Typerep.TypeRep
| constant Typerep.TypeRep  $\rightarrow$  (Haskell-Quickcheck) Typerep.TypeRep
```

code-reserved *Haskell-Quickcheck Typerep*

code-printing

```
type-constructor integer  $\rightarrow$  (Haskell-Quickcheck) Prelude.Int
| constant 0::integer  $\rightarrow$ 
  (Haskell-Quickcheck) !(0 / :: / Prelude.Int)
```

setup \langle

```
  let
    val target = Haskell-Quickcheck;
    fun print - = Code-Haskell.print-numeral Prelude.Int;
  in
    Numeral.add-code const-name  $\langle$ Code-Numeral.Pos $\rangle$  I print target
    #> Numeral.add-code const-name  $\langle$ Code-Numeral.Neg $\rangle$  (~) print target
  end
 $\rangle$ 
```

85.1.2 Narrowing’s deep representation of types and terms

datatype (*plugins only: code extraction*) *narrowing-type* =
Narrowing-sum-of-products narrowing-type list list

datatype (*plugins only: code extraction*) *narrowing-term* =
Narrowing-variable integer list narrowing-type
| *Narrowing-constructor integer narrowing-term list*

datatype (*plugins only: code extraction*) (*dead 'a*) *narrowing-cons* =
Narrowing-cons narrowing-type (narrowing-term list \Rightarrow 'a) list

primrec *map-cons* :: (*'a* \Rightarrow *'b*) \Rightarrow *'a narrowing-cons* \Rightarrow *'b narrowing-cons*

where

map-cons f (Narrowing-cons ty cs) = Narrowing-cons ty (map ($\lambda c. f \circ c$) cs)

85.1.3 From narrowing’s deep representation of terms to *HOL.Code-Evaluation*’s terms

class *partial-term-of* = *typerep* +
fixes *partial-term-of* :: *'a itself* \Rightarrow *narrowing-term* \Rightarrow *Code-Evaluation.term*

lemma *partial-term-of-anything*: *partial-term-of x nt* \equiv *t*
by (*rule eq-reflection*) (*cases partial-term-of x nt, cases t, simp*)

85.1.4 Auxiliary functions for Narrowing

consts *nth* :: *'a list* \Rightarrow *integer* \Rightarrow *'a*

code-printing constant *nth* \rightarrow (*Haskell-Quickcheck*) **infixl 9 !!**

consts *error* :: *char list* => 'a

code-printing constant *error* → (*Haskell-Quickcheck*) *error*

consts *toEnum* :: *integer* => *char*

code-printing constant *toEnum* → (*Haskell-Quickcheck*) *Prelude.toEnum*

consts *marker* :: *char*

code-printing constant *marker* → (*Haskell-Quickcheck*) '\0'

85.1.5 Narrowing’s basic operations

type-synonym 'a *narrowing* = *integer* => 'a *narrowing-cons*

definition *cons* :: 'a => 'a *narrowing*

where

cons a d = (*Narrowing-cons* (*Narrowing-sum-of-products* [[]]) [(λ-. a)])

fun *conv* :: (*narrowing-term list* => 'a) *list* => *narrowing-term* => 'a

where

conv cs (*Narrowing-variable p* -) = *error* (*marker* # *map toEnum p*)
| *conv cs* (*Narrowing-constructor i xs*) = (*nth cs i*) *xs*

fun *non-empty* :: *narrowing-type* => *bool*

where

non-empty (*Narrowing-sum-of-products ps*) = (¬ (*List.null ps*))

definition *apply* :: ('a => 'b) *narrowing* => 'a *narrowing* => 'b *narrowing*

where

apply f a d = (if *d* > 0 then
 (*case f d* of *Narrowing-cons* (*Narrowing-sum-of-products ps*) *cfs* ⇒
 case a (*d* - 1) of *Narrowing-cons ta cas* ⇒
 let
 shallow = *non-empty ta*;
 cs = [(λ(*x* # *xs*) ⇒ *cf xs* (*conv cas x*)). *shallow*, *cf* ← *cfs*]
 in *Narrowing-cons* (*Narrowing-sum-of-products* [*ta* # *p. shallow*, *p* ← *ps*])
 cs)
 else *Narrowing-cons* (*Narrowing-sum-of-products* [] [])

definition *sum* :: 'a *narrowing* => 'a *narrowing* => 'a *narrowing*

where

sum a b d =
 (*case a d* of *Narrowing-cons* (*Narrowing-sum-of-products ssa*) *ca* ⇒
 case b d of *Narrowing-cons* (*Narrowing-sum-of-products ssb*) *cb* ⇒
 Narrowing-cons (*Narrowing-sum-of-products* (*ssa* @ *ssb*)) (*ca* @ *cb*))

```

lemma [fundef-cong]:
  assumes  $a\ d = a'\ d\ b\ d = b'\ d\ d = d'$ 
  shows  $sum\ a\ b\ d = sum\ a'\ b'\ d'$ 
using assms unfolding sum-def by (auto split: narrowing-cons.split narrowing-type.split)

```

```

lemma [fundef-cong]:
  assumes  $f\ d = f'\ d\ (\bigwedge d'.\ 0 \leq d' \wedge d' < d \implies a\ d' = a'\ d')$ 
  assumes  $d = d'$ 
  shows  $apply\ f\ a\ d = apply\ f'\ a'\ d'$ 
proof –
  note assms
  moreover have  $0 < d' \implies 0 \leq d' - 1$ 
    by (simp add: less-integer-def less-eq-integer-def)
  ultimately show ?thesis
    by (auto simp add: apply-def Let-def
      split: narrowing-cons.split narrowing-type.split)
qed

```

85.1.6 Narrowing generator type class

```

class narrowing =
  fixes narrowing :: integer => 'a narrowing-cons

datatype (plugins only: code extraction) property =
  Universal narrowing-type (narrowing-term => property) narrowing-term =>
  Code-Evaluation.term
| Existential narrowing-type (narrowing-term => property) narrowing-term =>
  Code-Evaluation.term
| Property bool

```

```

definition exists :: ('a :: {narrowing, partial-term-of} => property) => property
where
  exists f = (case narrowing (100 :: integer) of Narrowing-cons ty cs => Existential
  ty ( $\lambda t.\ f\ (conv\ cs\ t)$ ) (partial-term-of (TYPE('a))))

```

```

definition all :: ('a :: {narrowing, partial-term-of} => property) => property
where
  all f = (case narrowing (100 :: integer) of Narrowing-cons ty cs => Universal ty
  ( $\lambda t.\ f\ (conv\ cs\ t)$ ) (partial-term-of (TYPE('a))))

```

85.1.7 class *is-testable*

The class *is-testable* ensures that all necessary type instances are generated.

```

class is-testable

```

```

instance bool :: is-testable ..

```

```

instance fun :: ({term-of, narrowing, partial-term-of}, is-testable) is-testable ..

```


definition *ensure-testable* :: 'a :: is-testable => 'a :: is-testable

where

ensure-testable f = f

85.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way

datatype (*plugins only: code quickcheck-narrowing extraction*) (dead 'a, dead 'b)

ffun =

Constant 'b

| Update 'a 'b ('a, 'b) *ffun*

primrec *eval-ffun* :: ('a, 'b) *ffun* => 'a => 'b

where

eval-ffun (Constant c) x = c

| *eval-ffun* (Update x' y f) x = (if x = x' then y else *eval-ffun* f x)

hide-type (**open**) *ffun*

hide-const (**open**) Constant Update *eval-ffun*

datatype (*plugins only: code quickcheck-narrowing extraction*) (dead 'b) *cfun* =

Constant 'b

primrec *eval-cfun* :: 'b *cfun* => 'a => 'b

where

eval-cfun (Constant c) y = c

hide-type (**open**) *cfun*

hide-const (**open**) Constant *eval-cfun* Abs-*cfun* Rep-*cfun*

85.1.9 Setting up the counterexample generator

external-file <~/src/HOL/Tools/Quickcheck/Narrowing-Engine.hs>

external-file <~/src/HOL/Tools/Quickcheck/PNF-Narrowing-Engine.hs>

ML-file <Tools/Quickcheck/narrowing-generators.ML>

definition *narrowing-dummy-partial-term-of* :: ('a :: partial-term-of) itself =>

narrowing-term => term

where

narrowing-dummy-partial-term-of = *partial-term-of*

definition *narrowing-dummy-narrowing* :: integer => ('a :: narrowing) narrow-

ing-cons

where

narrowing-dummy-narrowing = *narrowing*

lemma [*code*]:

ensure-testable f =

(let

```

    x = narrowing-dummy-narrowing :: integer => bool narrowing-cons;
    y = narrowing-dummy-partial-term-of :: bool itself => narrowing-term =>
term;
    z = (conv :: - => - => unit) in f)
unfolding Let-def ensure-testable-def ..

```

85.2 Narrowing for sets

```

instantiation set :: (narrowing) narrowing
begin

```

```

definition narrowing-set = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons
set) narrowing

```

```

instance ..

```

```

end

```

85.3 Narrowing for integers

```

definition drawn-from :: 'a list => 'a narrowing-cons
where

```

```

    drawn-from xs =
    Narrowing-cons (Narrowing-sum-of-products (map (λ-. []) xs)) (map (λx -. x)
xs)

```

```

function around-zero :: int => int list

```

```

where

```

```

    around-zero i = (if i < 0 then [] else (if i = 0 then [0] else around-zero (i - 1)
@ [i, -i]))

```

```

    by pat-completeness auto

```

```

termination by (relation measure nat) auto

```

```

declare around-zero.simps [simp del]

```

```

lemma length-around-zero:

```

```

    assumes i >= 0

```

```

    shows length (around-zero i) = 2 * nat i + 1

```

```

proof (induct rule: int-ge-induct [OF assms])

```

```

    case 1

```

```

    from 1 show ?case by (simp add: around-zero.simps)

```

```

next

```

```

    case (2 i)

```

```

    from 2 show ?case

```

```

    by (simp add: around-zero.simps [of i + 1])

```

```

qed

```

```

instantiation int :: narrowing

```

```

begin

```

definition

narrowing-int $d = (\text{let } (u :: - \Rightarrow - \Rightarrow \text{unit}) = \text{conv}; i = \text{int-of-integer } d$
in drawn-from (*around-zero* i))

instance ..**end****declare** [[*code drop: partial-term-of* :: *int itself* \Rightarrow -]]**lemma** [*code*]:

partial-term-of ($ty :: \text{int itself}$) (*Narrowing-variable* $p\ t$) \equiv
Code-Evaluation.Free (*STR* "'-") (*Typerep.Typerep* (*STR* "Int.int") [])
partial-term-of ($ty :: \text{int itself}$) (*Narrowing-constructor* i []) \equiv
 (*if* $i \bmod 2 = 0$
then *Code-Evaluation.term-of* ($-(\text{int-of-integer } i) \text{ div } 2$)
else *Code-Evaluation.term-of* ($(\text{int-of-integer } i + 1) \text{ div } 2$))
by (*rule partial-term-of-anything*)+

instantiation *integer* :: *narrowing***begin****definition**

narrowing-integer $d = (\text{let } (u :: - \Rightarrow - \Rightarrow \text{unit}) = \text{conv}; i = \text{int-of-integer } d$
in drawn-from (*map integer-of-int* (*around-zero* i)))

instance ..**end****declare** [[*code drop: partial-term-of* :: *integer itself* \Rightarrow -]]**lemma** [*code*]:

partial-term-of ($ty :: \text{integer itself}$) (*Narrowing-variable* $p\ t$) \equiv
Code-Evaluation.Free (*STR* "'-") (*Typerep.Typerep* (*STR* "Code-Numeral.integer")
 [])
partial-term-of ($ty :: \text{integer itself}$) (*Narrowing-constructor* i []) \equiv
 (*if* $i \bmod 2 = 0$
then *Code-Evaluation.term-of* ($- i \text{ div } 2$)
else *Code-Evaluation.term-of* ($(i + 1) \text{ div } 2$))
by (*rule partial-term-of-anything*)+

code-printing constant *Code-Evaluation.term-of* :: *integer* \Rightarrow *term* \rightarrow (*Haskell-Quickcheck*)

(*let* { $t = \text{Typerep.Typerep Code'-Numeral.integer}$ [];
 $\text{mkFunT } s\ t = \text{Typerep.Typerep fun } [s, t]$;
 $\text{numT} = \text{Typerep.Typerep Num.num}$ [];
 $\text{mkBit } 0 = \text{Generated'-Code.Const Num.num.Bit0 (mkFunT numT numT)}$;
 $\text{mkBit } 1 = \text{Generated'-Code.Const Num.num.Bit1 (mkFunT numT numT)}$;

```

mkNumeral 1 = Generated'-Code.Const Num.num.One numT;
mkNumeral i = let { q = i 'Prelude.div' 2; r = i 'Prelude.mod' 2 }
  in Generated'-Code.App (mkBit r) (mkNumeral q);
mkNumber 0 = Generated'-Code.Const Groups.zero'-class.zero t;
mkNumber 1 = Generated'-Code.Const Groups.one'-class.one t;
mkNumber i = if i > 0 then
  Generated'-Code.App
    (Generated'-Code.Const Num.numeral'-class.numeral
      (mkFunT numT t))
    (mkNumeral i)
else
  Generated'-Code.App
    (Generated'-Code.Const Groups.uminus'-class.uminus (mkFunT t t))
    (mkNumber (- i)); } in mkNumber

```

85.4 The *find-unused-assms* command

ML-file \langle Tools/Quickcheck/find-unused-assms.ML \rangle

85.5 Closing up

hide-type *narrowing-type narrowing-term narrowing-cons property*
hide-const *map-cons nth error toEnum marker empty Narrowing-cons conv non-empty ensure-testable all exists drawn-from around-zero*
hide-const (open) *Narrowing-variable Narrowing-constructor apply sum cons*
hide-fact *empty-def cons-def conv.simps non-empty.simps apply-def sum-def ensure-testable-def all-def exists-def*

end

theory *Mirabelle*

imports *Sledgehammer Predicate-Compile Presburger*
begin

ML-file \langle Tools/Mirabelle/mirabelle-util.ML \rangle

ML-file \langle Tools/Mirabelle/mirabelle.ML \rangle

ML \langle

signature MIRABELLE-ACTION = sig
*val make-action : Mirabelle.action-context -> string * Mirabelle.action*
end

\rangle

ML-file \langle Tools/Mirabelle/mirabelle-arith.ML \rangle

ML-file \langle Tools/Mirabelle/mirabelle-metis.ML \rangle

ML-file \langle Tools/Mirabelle/mirabelle-presburger.ML \rangle

ML-file \langle Tools/Mirabelle/mirabelle-quickcheck.ML \rangle

ML-file \langle Tools/Mirabelle/mirabelle-sledgehammer-filter.ML \rangle

ML-file \langle Tools/Mirabelle/mirabelle-sledgehammer.ML \rangle

ML-file \langle Tools/Mirabelle/mirabelle-try0.ML \rangle

end

86 Program extraction for HOL

theory *Extraction*
 imports *Option*
 begin

86.1 Setup

```
setup  $\langle$ 
  Extraction.add-types
  [(bool, ([], NONE))] #>
  Extraction.set-preprocessor (fn thy =>
    Proofterm.rewrite-proof-notypes
    ([], Rewrite-HOL-Proof.elim-cong :: Proof-Rewrite-Rules.rprocs true) o
    Proofterm.rewrite-proof thy
    (Rewrite-HOL-Proof.rews,
     Proof-Rewrite-Rules.rprocs true @ [Proof-Rewrite-Rules.expand-of-class thy])
  o
  Proof-Rewrite-Rules.elim-vars (curry Const const-name  $\langle$ default $\rangle$ )
 $\rangle$ 
```

```
lemmas [extraction-expand] =
  meta-spec atomize-eq atomize-all atomize-imp atomize-conj
  allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
  notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
  induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
  induct-atomize induct-atomize' induct-rulify induct-rulify'
  induct-rulify-fallback induct-trueI
  True-implies-equals implies-True-equals TrueE
  False-implies-equals implies-False-swap
```

```
lemmas [extraction-expand-def] =
  HOL.induct-forall-def HOL.induct-implies-def HOL.induct-equal-def HOL.induct-conj-def
  HOL.induct-true-def HOL.induct-false-def
```

```
datatype (plugins only: code extraction) sumbool = Left | Right
```

86.2 Type of extracted program

```
extract-type
  typeof (Trueprop P)  $\equiv$  typeof P

  typeof P  $\equiv$  Type (TYPE(Null))  $\implies$  typeof Q  $\equiv$  Type (TYPE('Q))  $\implies$ 
  typeof (P  $\longrightarrow$  Q)  $\equiv$  Type (TYPE('Q))
```

$$\text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } (P \longrightarrow Q) \equiv \text{Type } (\text{TYPE}(\text{Null}))$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \longrightarrow Q) \equiv \text{Type } (\text{TYPE}('P \Rightarrow 'Q)) \end{aligned}$$

$$\begin{aligned} (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\ \text{typeof } (\forall x. P x) \equiv \text{Type } (\text{TYPE}(\text{Null})) \end{aligned}$$

$$\begin{aligned} (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}('P))) \implies \\ \text{typeof } (\forall x::'a. P x) \equiv \text{Type } (\text{TYPE}('a \Rightarrow 'P)) \end{aligned}$$

$$\begin{aligned} (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\ \text{typeof } (\exists x::'a. P x) \equiv \text{Type } (\text{TYPE}('a)) \end{aligned}$$

$$\begin{aligned} (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}('P))) \implies \\ \text{typeof } (\exists x::'a. P x) \equiv \text{Type } (\text{TYPE}('a \times 'P)) \end{aligned}$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}(\text{sumbool})) \end{aligned}$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('Q \text{ option})) \end{aligned}$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('P \text{ option})) \end{aligned}$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('P + 'Q)) \end{aligned}$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('Q)) \end{aligned}$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\ \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P)) \end{aligned}$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P \times 'Q)) \end{aligned}$$

$$\text{typeof } (P = Q) \equiv \text{typeof } ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$$

$$\text{typeof } (x \in P) \equiv \text{typeof } P$$

86.3 Realizability

realizability

$$(\text{realizes } t \text{ (Trueprop } P)) \equiv (\text{Trueprop } (\text{realizes } t P))$$

$$\begin{aligned} (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\ (\text{realizes } t \text{ (} P \longrightarrow Q)) \equiv (\text{realizes } \text{Null } P \longrightarrow \text{realizes } t Q) \end{aligned}$$

$$\begin{aligned}
(\text{typeof } P) &\equiv (\text{Type } (\text{TYPE}('P))) \implies \\
(\text{typeof } Q) &\equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (P \longrightarrow Q)) &\equiv (\forall x::'P. \text{realizes } x P \longrightarrow \text{realizes } \text{Null } Q)
\end{aligned}$$

$$(\text{realizes } t (P \longrightarrow Q)) \equiv (\forall x. \text{realizes } x P \longrightarrow \text{realizes } (t x) Q)$$

$$\begin{aligned}
(\lambda x. \text{typeof } (P x)) &\equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (\forall x. P x)) &\equiv (\forall x. \text{realizes } \text{Null } (P x))
\end{aligned}$$

$$(\text{realizes } t (\forall x. P x)) \equiv (\forall x. \text{realizes } (t x) (P x))$$

$$\begin{aligned}
(\lambda x. \text{typeof } (P x)) &\equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (\exists x. P x)) &\equiv (\text{realizes } \text{Null } (P t))
\end{aligned}$$

$$(\text{realizes } t (\exists x. P x)) \equiv (\text{realizes } (\text{snd } t) (P (\text{fst } t)))$$

$$\begin{aligned}
(\text{typeof } P) &\equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{typeof } Q) &\equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (P \vee Q)) &\equiv \\
(\text{case } t \text{ of } \text{Left} \Rightarrow \text{realizes } \text{Null } P \mid \text{Right} \Rightarrow \text{realizes } \text{Null } Q)
\end{aligned}$$

$$\begin{aligned}
(\text{typeof } P) &\equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (P \vee Q)) &\equiv \\
(\text{case } t \text{ of } \text{None} \Rightarrow \text{realizes } \text{Null } P \mid \text{Some } q \Rightarrow \text{realizes } q Q)
\end{aligned}$$

$$\begin{aligned}
(\text{typeof } Q) &\equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (P \vee Q)) &\equiv \\
(\text{case } t \text{ of } \text{None} \Rightarrow \text{realizes } \text{Null } Q \mid \text{Some } p \Rightarrow \text{realizes } p P)
\end{aligned}$$

$$\begin{aligned}
(\text{realizes } t (P \vee Q)) &\equiv \\
(\text{case } t \text{ of } \text{Inl } p \Rightarrow \text{realizes } p P \mid \text{Inr } q \Rightarrow \text{realizes } q Q)
\end{aligned}$$

$$\begin{aligned}
(\text{typeof } P) &\equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (P \wedge Q)) &\equiv (\text{realizes } \text{Null } P \wedge \text{realizes } t Q)
\end{aligned}$$

$$\begin{aligned}
(\text{typeof } Q) &\equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
(\text{realizes } t (P \wedge Q)) &\equiv (\text{realizes } t P \wedge \text{realizes } \text{Null } Q)
\end{aligned}$$

$$(\text{realizes } t (P \wedge Q)) \equiv (\text{realizes } (\text{fst } t) P \wedge \text{realizes } (\text{snd } t) Q)$$

$$\begin{aligned}
\text{typeof } P &\equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
\text{realizes } t (\neg P) &\equiv \neg \text{realizes } \text{Null } P
\end{aligned}$$

$$\begin{aligned}
\text{typeof } P &\equiv \text{Type } (\text{TYPE}('P)) \implies \\
\text{realizes } t (\neg P) &\equiv (\forall x::'P. \neg \text{realizes } x P)
\end{aligned}$$

$$\begin{aligned}
\text{typeof } (P::\text{bool}) &\equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
\text{typeof } Q &\equiv \text{Type } (\text{TYPE}(\text{Null})) \implies
\end{aligned}$$

realizes t (P = Q) ≡ realizes Null P = realizes Null Q

(realizes t (P = Q)) ≡ (realizes t ((P → Q) ∧ (Q → P)))

86.4 Computational content of basic inference rules

theorem *disjE-realizer:*

assumes *r: case x of Inl p ⇒ P p | Inr q ⇒ Q q*
and *r1: ∧p. P p ⇒ R (f p) and r2: ∧q. Q q ⇒ R (g q)*
shows *R (case x of Inl p ⇒ f p | Inr q ⇒ g q)*

proof (*cases x*)

case *Inl*

with *r show ?thesis by simp (rule r1)*

next

case *Inr*

with *r show ?thesis by simp (rule r2)*

qed

theorem *disjE-realizer2:*

assumes *r: case x of None ⇒ P | Some q ⇒ Q q*
and *r1: P ⇒ R f and r2: ∧q. Q q ⇒ R (g q)*
shows *R (case x of None ⇒ f | Some q ⇒ g q)*

proof (*cases x*)

case *None*

with *r show ?thesis by simp (rule r1)*

next

case *Some*

with *r show ?thesis by simp (rule r2)*

qed

theorem *disjE-realizer3:*

assumes *r: case x of Left ⇒ P | Right ⇒ Q*
and *r1: P ⇒ R f and r2: Q ⇒ R g*
shows *R (case x of Left ⇒ f | Right ⇒ g)*

proof (*cases x*)

case *Left*

with *r show ?thesis by simp (rule r1)*

next

case *Right*

with *r show ?thesis by simp (rule r2)*

qed

theorem *conjI-realizer:*

P p ⇒ Q q ⇒ P (fst (p, q)) ∧ Q (snd (p, q))
by *simp*

theorem *exI-realizer:*

P y x ⇒ P (snd (x, y)) (fst (x, y)) **by** *simp*

theorem *exE-realizer*: $P \text{ (snd } p) \text{ (fst } p) \implies$
 $(\bigwedge x y. P y x \implies Q (f x y)) \implies Q \text{ (let } (x, y) = p \text{ in } f x y)$
by (*cases p*) (*simp add: Let-def*)

theorem *exE-realizer'*: $P \text{ (snd } p) \text{ (fst } p) \implies$
 $(\bigwedge x y. P y x \implies Q) \implies Q$ **by** (*cases p*) *simp*

realizers

impI (P, Q): $\lambda pq. pq$
 $\lambda(c: -) (d: -) P Q pq (h: -). \text{allI } \dots c \cdot (\lambda x. \text{impI } \dots \cdot (h \cdot x))$

impI (P): *Null*
 $\lambda(c: -) P Q (h: -). \text{allI } \dots c \cdot (\lambda x. \text{impI } \dots \cdot (h \cdot x))$

impI (Q): $\lambda q. q \lambda(c: -) P Q q. \text{impI } \dots$

impI: *Null impI*

mp (P, Q): $\lambda pq. pq$
 $\lambda(c: -) (d: -) P Q pq (h: -) p. mp \dots \cdot (\text{spec } \dots p \cdot c \cdot h)$

mp (P): *Null*
 $\lambda(c: -) P Q (h: -) p. mp \dots \cdot (\text{spec } \dots p \cdot c \cdot h)$

mp (Q): $\lambda q. q \lambda(c: -) P Q q. mp \dots$

mp: *Null mp*

allI (P): $\lambda p. p \lambda(c: -) P (d: -) p. \text{allI } \dots d$

allI: *Null allI*

spec (P): $\lambda x p. p x \lambda(c: -) P x (d: -) p. \text{spec } \dots x \cdot d$

spec: *Null spec*

exI (P): $\lambda x p. (x, p) \lambda(c: -) P x (d: -) p. \text{exI-realizer} \cdot P \cdot p \cdot x \cdot c \cdot d$

exI: $\lambda x. x \lambda P x (c: -) (h: -). h$

exE (P, Q): $\lambda p pq. \text{let } (x, y) = p \text{ in } pq x y$
 $\lambda(c: -) (d: -) P Q (e: -) p (h: -) pq. \text{exE-realizer} \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h$

exE (P): *Null*
 $\lambda(c: -) P Q (d: -) p. \text{exE-realizer}' \dots \cdot c \cdot d$

exE (Q): $\lambda x pq. pq x$
 $\lambda(c: -) P Q (d: -) x (h1: -) pq (h2: -). h2 \cdot x \cdot h1$

exE: *Null*

$\lambda P Q (c: -) x (h1: -) (h2: -). h2 \cdot x \cdot h1$

conjI (*P*, *Q*): *Pair*

$\lambda(c: -) (d: -) P Q p (h: -) q. conjI\text{-realizer} \cdot P \cdot p \cdot Q \cdot q \cdot c \cdot d \cdot h$

conjI (*P*): $\lambda p. p$

$\lambda(c: -) P Q p. conjI \cdot \dots$

conjI (*Q*): $\lambda q. q$

$\lambda(c: -) P Q (h: -) q. conjI \cdot \dots \cdot h$

conjI: *Null conjI*

conjunct1 (*P*, *Q*): *fst*

$\lambda(c: -) (d: -) P Q pq. conjunct1 \cdot \dots$

conjunct1 (*P*): $\lambda p. p$

$\lambda(c: -) P Q p. conjunct1 \cdot \dots$

conjunct1 (*Q*): *Null*

$\lambda(c: -) P Q q. conjunct1 \cdot \dots$

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*

$\lambda(c: -) (d: -) P Q pq. conjunct2 \cdot \dots$

conjunct2 (*P*): *Null*

$\lambda(c: -) P Q p. conjunct2 \cdot \dots$

conjunct2 (*Q*): $\lambda p. p$

$\lambda(c: -) P Q p. conjunct2 \cdot \dots$

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*

$\lambda(c: -) (d: -) P Q p. iffD2 \cdot \dots \cdot (sum.case-1 \cdot P \cdot \dots \cdot p \cdot arity\text{-type-bool} \cdot c \cdot d)$

disjI1 (*P*): *Some*

$\lambda(c: -) P Q p. iffD2 \cdot \dots \cdot (option.case-2 \cdot \dots \cdot P \cdot p \cdot arity\text{-type-bool} \cdot c)$

disjI1 (*Q*): *None*

$\lambda(c: -) P Q. iffD2 \cdot \dots \cdot (option.case-1 \cdot \dots \cdot arity\text{-type-bool} \cdot c)$

disjI1: *Left*

$\lambda P Q. iffD2 \cdot \dots \cdot (sumbool.case-1 \cdot \dots \cdot arity\text{-type-bool})$

disjI2 (*P*, *Q*): *Inr*
 $\lambda(d: -) (c: -) Q P q. \text{iffD2} \dots \dots (sum.case-2 \dots Q \cdot q \cdot \text{arity-type-bool} \cdot c \cdot d)$

disjI2 (*P*): *None*
 $\lambda(c: -) Q P. \text{iffD2} \dots \dots (option.case-1 \dots \dots \text{arity-type-bool} \cdot c)$

disjI2 (*Q*): *Some*
 $\lambda(c: -) Q P q. \text{iffD2} \dots \dots (option.case-2 \dots Q \cdot q \cdot \text{arity-type-bool} \cdot c)$

disjI2: *Right*
 $\lambda Q P. \text{iffD2} \dots \dots (sumbool.case-2 \dots \dots \text{arity-type-bool})$

disjE (*P*, *Q*, *R*): $\lambda pq pr qr.$
(case pq of Inl p \Rightarrow pr p | Inr q \Rightarrow qr q)
 $\lambda(c: -) (d: -) (e: -) P Q R pq (h1: -) pr (h2: -) qr.$
 $\text{disjE-realizer} \dots \dots pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot e \cdot h1 \cdot h2$

disjE (*Q*, *R*): $\lambda pq pr qr.$
(case pq of None \Rightarrow pr | Some q \Rightarrow qr q)
 $\lambda(c: -) (d: -) P Q R pq (h1: -) pr (h2: -) qr.$
 $\text{disjE-realizer2} \dots \dots pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot h1 \cdot h2$

disjE (*P*, *R*): $\lambda pq pr qr.$
(case pq of None \Rightarrow qr | Some p \Rightarrow pr p)
 $\lambda(c: -) (d: -) P Q R pq (h1: -) pr (h2: -) qr (h3: -).$
 $\text{disjE-realizer2} \dots \dots pq \cdot R \cdot qr \cdot pr \cdot c \cdot d \cdot h1 \cdot h3 \cdot h2$

disjE (*R*): $\lambda pq pr qr.$
(case pq of Left \Rightarrow pr | Right \Rightarrow qr)
 $\lambda(c: -) P Q R pq (h1: -) pr (h2: -) qr.$
 $\text{disjE-realizer3} \dots \dots pq \cdot R \cdot pr \cdot qr \cdot c \cdot h1 \cdot h2$

disjE (*P*, *Q*): *Null*
 $\lambda(c: -) (d: -) P Q R pq. \text{disjE-realizer} \dots \dots pq \cdot (\lambda x. R) \dots \dots c \cdot d \cdot \text{arity-type-bool}$

disjE (*Q*): *Null*
 $\lambda(c: -) P Q R pq. \text{disjE-realizer2} \dots \dots pq \cdot (\lambda x. R) \dots \dots c \cdot \text{arity-type-bool}$

disjE (*P*): *Null*
 $\lambda(c: -) P Q R pq (h1: -) (h2: -) (h3: -).$
 $\text{disjE-realizer2} \dots \dots pq \cdot (\lambda x. R) \dots \dots c \cdot \text{arity-type-bool} \cdot h1 \cdot h3 \cdot h2$

disjE: *Null*
 $\lambda P Q R pq. \text{disjE-realizer3} \dots \dots pq \cdot (\lambda x. R) \dots \dots \text{arity-type-bool}$

FalseE (*P*): *default*
 $\lambda(c: -) P. \text{FalseE} \dots$

FalseE: *Null FalseE*

notI (*P*): *Null*

$\lambda(c: -) P (h: -). \text{allI} \dots c \cdot (\lambda x. \text{notI} \dots (h \cdot x))$

notI: *Null notI*

notE (*P*, *R*): $\lambda p. \text{default}$

$\lambda(c: -) (d: -) P R (h: -) p. \text{notE} \dots \cdot (\text{spec} \dots p \cdot c \cdot h)$

notE (*P*): *Null*

$\lambda(c: -) P R (h: -) p. \text{notE} \dots \cdot (\text{spec} \dots p \cdot c \cdot h)$

notE (*R*): *default*

$\lambda(c: -) P R. \text{notE} \dots$

notE: *Null notE*

subst (*P*): $\lambda s t ps. ps$

$\lambda(c: -) s t P (d: -) (h: -) ps. \text{subst} \cdot s \cdot t \cdot P ps \cdot d \cdot h$

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*

$\lambda(d: -) (c: -) Q P pq (h: -) p. \text{mp} \dots \cdot (\text{spec} \dots p \cdot d \cdot (\text{conjunct1} \dots \cdot h))$

iffD1 (*P*): $\lambda p. p$

$\lambda(c: -) Q P p (h: -). \text{mp} \dots \cdot (\text{conjunct1} \dots \cdot h)$

iffD1 (*Q*): *Null*

$\lambda(c: -) Q P q1 (h: -) q2. \text{mp} \dots \cdot (\text{spec} \dots q2 \cdot c \cdot (\text{conjunct1} \dots \cdot h))$

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*

$\lambda(c: -) (d: -) P Q pq (h: -) q. \text{mp} \dots \cdot (\text{spec} \dots q \cdot d \cdot (\text{conjunct2} \dots \cdot h))$

iffD2 (*P*): $\lambda p. p$

$\lambda(c: -) P Q p (h: -). \text{mp} \dots \cdot (\text{conjunct2} \dots \cdot h)$

iffD2 (*Q*): *Null*

$\lambda(c: -) P Q q1 (h: -) q2. \text{mp} \dots \cdot (\text{spec} \dots q2 \cdot c \cdot (\text{conjunct2} \dots \cdot h))$

iffD2: *Null iffD2*

```

iffI (P, Q): Pair
  λ(c: -) (d: -) P Q pq (h1 : -) qp (h2 : -). conjI-realizer ·
    (λpq. ∀ x. P x → Q (pq x)) · pq ·
    (λqp. ∀ x. Q x → P (qp x)) · qp ·
    (arity-type-fun · c · d) ·
    (arity-type-fun · d · c) ·
    (allI · · · c · (λx. impI · · · · (h1 · x))) ·
    (allI · · · d · (λx. impI · · · · (h2 · x)))

```

```

iffI (P): λp. p
  λ(c: -) P Q (h1 : -) p (h2 : -). conjI · · · · ·
    (allI · · · c · (λx. impI · · · · · (h1 · x))) ·
    (impI · · · · · h2)

```

```

iffI (Q): λq. q
  λ(c: -) P Q q (h1 : -) (h2 : -). conjI · · · · ·
    (impI · · · · · h1) ·
    (allI · · · c · (λx. impI · · · · · (h2 · x)))

```

```

iffI: Null iffI

```

end

87 Extensible records with structural subtyping

```

theory Record
imports Quickcheck-Exhaustive
keywords
  record :: thy-defn and
  print-record :: diag
begin

```

87.1 Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification $\alpha(\text{beta-update } f \text{ rec}) = \alpha \text{ rec}$ for distinct fields α and β of some record rec with n fields. There are $n \hat{=} 2$ such theorems, which prohibits storage of all of them for large n . The rules can be proved on the fly by case decomposition and simplification in $O(n)$ time. By creating $O(n)$ isomorphic-tuple types while defining the record, however, we can prove the access/update simplification in $O(\log(n) \hat{=} 2)$ time.

The $O(n)$ cost of case decomposition is not because $O(n)$ steps are taken, but rather because the resulting rule must contain $O(n)$ new variables and an $O(n)$ size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields $'a$, $'b$, $'c$ and $'d$ might be introduced as isomorphic to $'a \times ('b \times ('c \times 'd))$. If we balance the tuple tree to $('a \times 'b) \times ('c \times 'd)$ then accessors can be defined by converting to the underlying type then using $O(\log(n))$ fst or snd operations. Updaters can be defined similarly, if we introduce a *fst-update* and *snd-update* function. Furthermore, we can prove the access/update theorem in $O(\log(n))$ steps by using simple rewrites on fst, snd, *fst-update* and *snd-update*.

The catch is that, although $O(\log(n))$ steps were taken, the underlying type we converted to is a tuple tree of size $O(n)$. Processing this term type wastes performance. We avoid this for large n by taking each subtree of size K and defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these $O(n/K)$ new types, or, if $n > K * K$, we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant K .

If we prove the access/update theorem on this type with the analogous steps to the tuple tree, we consume $O(\log(n)^2)$ time as the intermediate terms are $O(\log(n))$ in size and the types needed have size bounded by K . To enable this analogous traversal, we define the functions seen below: *iso-tuple-fst*, *iso-tuple-snd*, *iso-tuple-fst-update* and *iso-tuple-snd-update*. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.

These rewrites are typically used in a structured way. They are here presented as the introduction rule *isomorphic-tuple.intros* rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

87.2 Operators and lemmas for types isomorphic to tuples

datatype (*dead 'a, dead 'b, dead 'c*) *tuple-isomorphism* =
Tuple-Isomorphism 'a \Rightarrow $'b \times 'c$ $'b \times 'c \Rightarrow 'a$

primrec

repr :: ($'a, 'b, 'c$) *tuple-isomorphism* \Rightarrow $'a \Rightarrow 'b \times 'c$ **where**
repr (*Tuple-Isomorphism* r a) = r

primrec

abst :: ($'a, 'b, 'c$) *tuple-isomorphism* \Rightarrow $'b \times 'c \Rightarrow 'a$ **where**
abst (*Tuple-Isomorphism* r a) = a

definition

iso-tuple-fst :: ('a, 'b, 'c) tuple-isomorphism \Rightarrow 'a \Rightarrow 'b **where**
iso-tuple-fst isom = *fst* \circ *repr isom*

definition

iso-tuple-snd :: ('a, 'b, 'c) tuple-isomorphism \Rightarrow 'a \Rightarrow 'c **where**
iso-tuple-snd isom = *snd* \circ *repr isom*

definition

iso-tuple-fst-update ::
('a, 'b, 'c) tuple-isomorphism \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a) **where**
iso-tuple-fst-update isom *f* = *abst isom* \circ *apfst f* \circ *repr isom*

definition

iso-tuple-snd-update ::
('a, 'b, 'c) tuple-isomorphism \Rightarrow ('c \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'a) **where**
iso-tuple-snd-update isom *f* = *abst isom* \circ *apsnd f* \circ *repr isom*

definition

iso-tuple-cons ::
('a, 'b, 'c) tuple-isomorphism \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a **where**
iso-tuple-cons isom = *curry (abst isom)*

87.3 Logical infrastructure for records

definition

iso-tuple-surjective-proof-assist :: 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool **where**
iso-tuple-surjective-proof-assist *x y f* \longleftrightarrow *f x = y*

definition

iso-tuple-update-accessor-cong-assist ::
(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool **where**
iso-tuple-update-accessor-cong-assist *upd ac* \longleftrightarrow
 $(\forall f v. \text{upd } (\lambda x. f (ac v)) v = \text{upd } f v) \wedge (\forall v. \text{upd } id v = v)$

definition

iso-tuple-update-accessor-eq-assist ::
(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool

where

iso-tuple-update-accessor-eq-assist *upd ac v f v' x* \longleftrightarrow
 $\text{upd } f v = v' \wedge ac v = x \wedge \text{iso-tuple-update-accessor-cong-assist } \text{upd } ac$

lemma *update-accessor-congruence-foldE*:

assumes *uac*: *iso-tuple-update-accessor-cong-assist upd ac*

and *r*: *r = r'* **and** *ac* *r' = v'*

and *f*: $\bigwedge v. v' = v \implies f v = f' v$

shows *upd f r = upd f' r'*

using *uac r v [symmetric]*

apply (*subgoal-tac upd* $(\lambda x. f (ac r')) r' = \text{upd } (\lambda x. f' (ac r')) r'$)

apply (*simp add: iso-tuple-update-accessor-cong-assist-def*)
apply (*simp add: f*)
done

lemma *update-accessor-congruence-unfoldE*:
iso-tuple-update-accessor-cong-assist upd ac \implies
 $r = r' \implies ac\ r' = v' \implies (\bigwedge v. v = v' \implies f\ v = f'\ v) \implies$
 $upd\ f\ r = upd\ f'\ r'$
apply (*erule(2) update-accessor-congruence-foldE*)
apply *simp*
done

lemma *iso-tuple-update-accessor-cong-assist-id*:
iso-tuple-update-accessor-cong-assist upd ac $\implies upd\ id = id$
by *rule (simp add: iso-tuple-update-accessor-cong-assist-def)*

lemma *update-accessor-noopE*:
assumes *uac: iso-tuple-update-accessor-cong-assist upd ac*
and *ac: f (ac x) = ac x*
shows $upd\ f\ x = x$
using *uac*
by (*simp add: ac iso-tuple-update-accessor-cong-assist-id [OF uac, unfolded id-def]*
cong: update-accessor-congruence-unfoldE [OF uac])

lemma *update-accessor-noop-compE*:
assumes *uac: iso-tuple-update-accessor-cong-assist upd ac*
and *ac: f (ac x) = ac x*
shows $upd\ (g \circ f)\ x = upd\ g\ x$
by (*simp add: ac cong: update-accessor-congruence-unfoldE [OF uac]*)

lemma *update-accessor-cong-assist-idI*:
iso-tuple-update-accessor-cong-assist id id
by (*simp add: iso-tuple-update-accessor-cong-assist-def*)

lemma *update-accessor-cong-assist-triv*:
iso-tuple-update-accessor-cong-assist upd ac \implies
iso-tuple-update-accessor-cong-assist upd ac
by *assumption*

lemma *update-accessor-accessor-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\implies ac\ v = x$
by (*simp add: iso-tuple-update-accessor-eq-assist-def*)

lemma *update-accessor-updator-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\implies upd\ f\ v = v'$
by (*simp add: iso-tuple-update-accessor-eq-assist-def*)

lemma *iso-tuple-update-accessor-eq-assist-idI*:
 $v' = f\ v \implies iso-tuple-update-accessor-eq-assist\ id\ id\ v\ f\ v'\ v$

by (*simp add: iso-tuple-update-accessor-eq-assist-def update-accessor-cong-assist-idI*)

lemma *iso-tuple-update-accessor-eq-assist-triv*:

iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies
iso-tuple-update-accessor-eq-assist upd ac v f v' x

by *assumption*

lemma *iso-tuple-update-accessor-cong-from-eq*:

iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies
iso-tuple-update-accessor-cong-assist upd ac

by (*simp add: iso-tuple-update-accessor-eq-assist-def*)

lemma *iso-tuple-surjective-proof-assistI*:

f x = y \implies iso-tuple-surjective-proof-assist x y f

by (*simp add: iso-tuple-surjective-proof-assist-def*)

lemma *iso-tuple-surjective-proof-assist-idE*:

iso-tuple-surjective-proof-assist x y id \implies x = y

by (*simp add: iso-tuple-surjective-proof-assist-def*)

locale *isomorphic-tuple =*

fixes *isom :: ('a, 'b, 'c) tuple-isomorphism*

assumes *repr-inv: $\bigwedge x. \text{abst isom (repr isom x) = x}$*

and *abst-inv: $\bigwedge y. \text{repr isom (abst isom y) = y}$*

begin

lemma *repr-inj: repr isom x = repr isom y \longleftrightarrow x = y*

by (*auto dest: arg-cong [of repr isom x repr isom y abst isom]*
simp add: repr-inv)

lemma *abst-inj: abst isom x = abst isom y \longleftrightarrow x = y*

by (*auto dest: arg-cong [of abst isom x abst isom y repr isom]*
simp add: abst-inv)

lemmas *simps = Let-def repr-inv abst-inv repr-inj abst-inj*

lemma *iso-tuple-access-update-fst-fst*:

f \circ h g = j \circ f \implies

(f \circ iso-tuple-fst isom) \circ (iso-tuple-fst-update isom \circ h) g =
j \circ (f \circ iso-tuple-fst isom)

by (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-fst-def simps*
fun-eq-iff)

lemma *iso-tuple-access-update-snd-snd*:

f \circ h g = j \circ f \implies

(f \circ iso-tuple-snd isom) \circ (iso-tuple-snd-update isom \circ h) g =
j \circ (f \circ iso-tuple-snd isom)

by (*clarsimp simp: iso-tuple-snd-update-def iso-tuple-snd-def simps*
fun-eq-iff)

lemma *iso-tuple-access-update-fst-snd*:

$$(f \circ \text{iso-tuple-fst isom}) \circ (\text{iso-tuple-snd-update isom} \circ h) g = \\ \text{id} \circ (f \circ \text{iso-tuple-fst isom})$$

by (*clarsimp simp: iso-tuple-snd-update-def iso-tuple-fst-def_simps fun-eq-iff*)

lemma *iso-tuple-access-update-snd-fst*:

$$(f \circ \text{iso-tuple-snd isom}) \circ (\text{iso-tuple-fst-update isom} \circ h) g = \\ \text{id} \circ (f \circ \text{iso-tuple-snd isom})$$

by (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-def_simps fun-eq-iff*)

lemma *iso-tuple-update-swap-fst-fst*:

$$h f \circ j g = j g \circ h f \implies \\ (\text{iso-tuple-fst-update isom} \circ h) f \circ (\text{iso-tuple-fst-update isom} \circ j) g = \\ (\text{iso-tuple-fst-update isom} \circ j) g \circ (\text{iso-tuple-fst-update isom} \circ h) f$$

by (*clarsimp simp: iso-tuple-fst-update-def_simps apfst-compose fun-eq-iff*)

lemma *iso-tuple-update-swap-snd-snd*:

$$h f \circ j g = j g \circ h f \implies \\ (\text{iso-tuple-snd-update isom} \circ h) f \circ (\text{iso-tuple-snd-update isom} \circ j) g = \\ (\text{iso-tuple-snd-update isom} \circ j) g \circ (\text{iso-tuple-snd-update isom} \circ h) f$$

by (*clarsimp simp: iso-tuple-snd-update-def_simps apsnd-compose fun-eq-iff*)

lemma *iso-tuple-update-swap-fst-snd*:

$$(\text{iso-tuple-snd-update isom} \circ h) f \circ (\text{iso-tuple-fst-update isom} \circ j) g = \\ (\text{iso-tuple-fst-update isom} \circ j) g \circ (\text{iso-tuple-snd-update isom} \circ h) f$$

by (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-update-def_simps fun-eq-iff*)

lemma *iso-tuple-update-swap-snd-fst*:

$$(\text{iso-tuple-fst-update isom} \circ h) f \circ (\text{iso-tuple-snd-update isom} \circ j) g = \\ (\text{iso-tuple-snd-update isom} \circ j) g \circ (\text{iso-tuple-fst-update isom} \circ h) f$$

by (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-update-def_simps fun-eq-iff*)

lemma *iso-tuple-update-compose-fst-fst*:

$$h f \circ j g = k (f \circ g) \implies \\ (\text{iso-tuple-fst-update isom} \circ h) f \circ (\text{iso-tuple-fst-update isom} \circ j) g = \\ (\text{iso-tuple-fst-update isom} \circ k) (f \circ g)$$

by (*clarsimp simp: iso-tuple-fst-update-def_simps apfst-compose fun-eq-iff*)

lemma *iso-tuple-update-compose-snd-snd*:

$$h f \circ j g = k (f \circ g) \implies \\ (\text{iso-tuple-snd-update isom} \circ h) f \circ (\text{iso-tuple-snd-update isom} \circ j) g = \\ (\text{iso-tuple-snd-update isom} \circ k) (f \circ g)$$

by (*clarsimp simp: iso-tuple-snd-update-def_simps apsnd-compose fun-eq-iff*)

lemma *iso-tuple-surjective-proof-assist-step*:

iso-tuple-surjective-proof-assist v a (*iso-tuple-fst isom* $\circ f$) \implies
iso-tuple-surjective-proof-assist v b (*iso-tuple-snd isom* $\circ f$) \implies
iso-tuple-surjective-proof-assist v (*iso-tuple-cons isom* a b) f
by (*clarsimp simp: iso-tuple-surjective-proof-assist-def_simps*
iso-tuple-fst-def iso-tuple-snd-def iso-tuple-cons-def)

lemma *iso-tuple-fst-update-accessor-cong-assist*:

assumes *iso-tuple-update-accessor-cong-assist* f g
shows *iso-tuple-update-accessor-cong-assist*
(*iso-tuple-fst-update isom* $\circ f$) ($g \circ$ *iso-tuple-fst isom*)

proof –

from *assms* **have** f *id* = *id*
by (*rule iso-tuple-update-accessor-cong-assist-id*)
with *assms* **show** ?thesis
by (*clarsimp simp: iso-tuple-update-accessor-cong-assist-def_simps*
iso-tuple-fst-update-def iso-tuple-fst-def)

qed

lemma *iso-tuple-snd-update-accessor-cong-assist*:

assumes *iso-tuple-update-accessor-cong-assist* f g
shows *iso-tuple-update-accessor-cong-assist*
(*iso-tuple-snd-update isom* $\circ f$) ($g \circ$ *iso-tuple-snd isom*)

proof –

from *assms* **have** f *id* = *id*
by (*rule iso-tuple-update-accessor-cong-assist-id*)
with *assms* **show** ?thesis
by (*clarsimp simp: iso-tuple-update-accessor-cong-assist-def_simps*
iso-tuple-snd-update-def iso-tuple-snd-def)

qed

lemma *iso-tuple-fst-update-accessor-eq-assist*:

assumes *iso-tuple-update-accessor-eq-assist* f g a u a' v
shows *iso-tuple-update-accessor-eq-assist*
(*iso-tuple-fst-update isom* $\circ f$) ($g \circ$ *iso-tuple-fst isom*)
(*iso-tuple-cons isom* a b) u (*iso-tuple-cons isom* a' b) v

proof –

from *assms* **have** f *id* = *id*
by (*auto simp add: iso-tuple-update-accessor-eq-assist-def*
intro: iso-tuple-update-accessor-cong-assist-id)
with *assms* **show** ?thesis
by (*clarsimp simp: iso-tuple-update-accessor-eq-assist-def*
iso-tuple-fst-update-def iso-tuple-fst-def
iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def_simps)

qed

lemma *iso-tuple-snd-update-accessor-eq-assist*:

assumes *iso-tuple-update-accessor-eq-assist* f g b u b' v
shows *iso-tuple-update-accessor-eq-assist*

(*iso-tuple-snd-update isom* \circ *f*) (*g* \circ *iso-tuple-snd isom*)
 (*iso-tuple-cons isom* *a b*) *u* (*iso-tuple-cons isom* *a b'*) *v*

proof –

from *assms* **have** *f id* = *id*

by (*auto simp add: iso-tuple-update-accessor-eq-assist-def*
intro: iso-tuple-update-accessor-cong-assist-id)

with *assms* **show** ?*thesis*

by (*clarsimp simp: iso-tuple-update-accessor-eq-assist-def*
iso-tuple-snd-update-def iso-tuple-snd-def
iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def_simps)

qed

lemma *iso-tuple-cons-conj-eqI*:

$a = c \wedge b = d \wedge P \longleftrightarrow Q \implies$

$iso-tuple-cons isom a b = iso-tuple-cons isom c d \wedge P \longleftrightarrow Q$

by (*clarsimp simp: iso-tuple-cons-def_simps*)

lemmas *intros* =

iso-tuple-access-update-fst-fst
iso-tuple-access-update-snd-snd
iso-tuple-access-update-fst-snd
iso-tuple-access-update-snd-fst
iso-tuple-update-swap-fst-fst
iso-tuple-update-swap-snd-snd
iso-tuple-update-swap-fst-snd
iso-tuple-update-swap-snd-fst
iso-tuple-update-compose-fst-fst
iso-tuple-update-compose-snd-snd
iso-tuple-surjective-proof-assist-step
iso-tuple-fst-update-accessor-eq-assist
iso-tuple-snd-update-accessor-eq-assist
iso-tuple-fst-update-accessor-cong-assist
iso-tuple-snd-update-accessor-cong-assist
iso-tuple-cons-conj-eqI

end

lemma *isomorphic-tuple-intro*:

fixes *repr abst*

assumes *repr-inj*: $\bigwedge x y. repr x = repr y \longleftrightarrow x = y$

and *abst-inv*: $\bigwedge z. repr (abst z) = z$

and *v*: $v \equiv Tuple\text{-Isomorphism } repr\ abst$

shows *isomorphic-tuple v*

proof

fix *x* **have** *repr (abst (repr x))* = *repr x*

by (*simp add: abst-inv*)

then show *Record.abst v (Record.repr v x)* = *x*

by (*simp add: v repr-inj*)

next

```

fix y
show Record.repr v (Record.abst v y) = y
  by (simp add: v) (fact abst-inv)
qed

```

definition

tuple-iso-tuple \equiv *Tuple-Isomorphism id id*

lemma *tuple-iso-tuple*:

isomorphic-tuple tuple-iso-tuple

by (simp add: isomorphic-tuple-intro [OF - - reflexive] tuple-iso-tuple-def)

lemma *refl-conj-eq*: $Q = R \implies P \wedge Q \longleftrightarrow P \wedge R$

by *simp*

lemma *iso-tuple-UNIV-I*: $x \in UNIV \equiv True$

by *simp*

lemma *iso-tuple-True-simp*: $(True \implies PROP P) \equiv PROP P$

by *simp*

lemma *prop-subst*: $s = t \implies PROP P t \implies PROP P s$

by *simp*

lemma *K-record-comp*: $(\lambda x. c) \circ f = (\lambda x. c)$

by (simp add: comp-def)

87.4 Concrete record syntax

nonterminal

ident **and**

field-type **and**

field-types **and**

field **and**

fields **and**

field-update **and**

field-updates

syntax

-constify $:: id \Rightarrow ident$ (-)

-constify $:: longid \Rightarrow ident$ (-)

-field-type $:: ident \Rightarrow type \Rightarrow field-type$ ((2- ::/ -))

$:: field-type \Rightarrow field-types$ (-)

-field-types $:: field-type \Rightarrow field-types \Rightarrow field-types$ (-, / -)

-record-type $:: field-types \Rightarrow type$ ((3(|-)))

-record-type-scheme $:: field-types \Rightarrow type \Rightarrow type$ ((3(|-, / (2... ::/ -))))

-field $:: ident \Rightarrow 'a \Rightarrow field$ ((2- =/ -))

```

:: field => fields          (-)
-fields                    :: field => fields => fields      (-, / -)
-record                    :: fields => 'a                  ((3(|-)))
-record-scheme             :: fields => 'a => 'a            ((3(|-, (2... =/ -))))

-field-update              :: ident => 'a => field-update   ((2- :=/ -))
                          :: field-update => field-updates (-)
-field-updates             :: field-update => field-updates => field-updates (-, / -)
-record-update             :: 'a => field-updates => 'b     (-/(3(|-)) [900, 0] 900)

```

syntax (ASCII)

```

-record-type               :: field-types => type           ((3'(| - |')))
-record-type-scheme       :: field-types => type => type    ((3'(| -, / (2... ::/ -) |')))
-record                    :: fields => 'a                  ((3'(| - |')))
-record-scheme             :: fields => 'a => 'a            ((3'(| -, / (2... =/ -) |')))
-record-update             :: 'a => field-updates => 'b     (-/(3'(| - |')) [900, 0] 900)

```

87.5 Record package

ML-file \langle Tools/record.ML \rangle

```

hide-const (open) Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd
  iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons
  iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist
  iso-tuple-update-accessor-eq-assist tuple-iso-tuple

```

end

88 Greatest common divisor and least common multiple

```

theory GCD
  imports Groups-List Code-Numeral
begin

```

88.1 Abstract bounded quasi semilattices as common foundation

```

locale bounded-quasi-semilattice = abel-semigroup +
  fixes top :: 'a ( $\top$ ) and bot :: 'a ( $\perp$ )
  and normalize :: 'a  $\Rightarrow$  'a
  assumes idem-normalize [simp]: a * a = normalize a
  and normalize-left-idem [simp]: normalize a * b = a * b
  and normalize-idem [simp]: normalize (a * b) = a * b
  and normalize-top [simp]: normalize  $\top$  =  $\top$ 
  and normalize-bottom [simp]: normalize  $\perp$  =  $\perp$ 
  and top-left-normalize [simp]:  $\top$  * a = normalize a
  and bottom-left-bottom [simp]:  $\perp$  * a =  $\perp$ 

```

begin

lemma *left-idem* [*simp*]:

$$a * (a * b) = a * b$$

using *assoc* [*of a a b, symmetric*] **by** *simp*

lemma *right-idem* [*simp*]:

$$(a * b) * b = a * b$$

using *left-idem* [*of b a*] **by** (*simp add: ac-simps*)

lemma *comp-fun-idem: comp-fun-idem f*

by *standard* (*simp-all add: fun-eq-iff ac-simps*)

interpretation *comp-fun-idem f*

by (*fact comp-fun-idem*)

lemma *top-right-normalize* [*simp*]:

$$a * \top = \text{normalize } a$$

using *top-left-normalize* [*of a*] **by** (*simp add: ac-simps*)

lemma *bottom-right-bottom* [*simp*]:

$$a * \perp = \perp$$

using *bottom-left-bottom* [*of a*] **by** (*simp add: ac-simps*)

lemma *normalize-right-idem* [*simp*]:

$$a * \text{normalize } b = a * b$$

using *normalize-left-idem* [*of b a*] **by** (*simp add: ac-simps*)

end

locale *bounded-quasi-semilattice-set = bounded-quasi-semilattice*

begin

interpretation *comp-fun-idem f*

by (*fact comp-fun-idem*)

definition *F* :: 'a set \Rightarrow 'a

where

$$\text{eq-fold: } F A = (\text{if finite } A \text{ then } \text{Finite-Set.fold } f \top A \text{ else } \perp)$$

lemma *infinite* [*simp*]:

$$\text{infinite } A \Longrightarrow F A = \perp$$

by (*simp add: eq-fold*)

lemma *set-eq-fold* [*code*]:

$$F (\text{set } xs) = \text{fold } f \top xs$$

by (*simp add: eq-fold fold-set-fold*)

lemma *empty* [*simp*]:

$F \{\} = \top$
by (*simp add: eq-fold*)

lemma *insert* [*simp*]:
 $F (\text{insert } a \ A) = a * F \ A$
by (*cases finite A*) (*simp-all add: eq-fold*)

lemma *normalize* [*simp*]:
 $\text{normalize } (F \ A) = F \ A$
by (*induct A rule: infinite-finite-induct*) *simp-all*

lemma *in-idem*:
assumes $a \in A$
shows $a * F \ A = F \ A$
using *assms* **by** (*induct A rule: infinite-finite-induct*)
(auto simp: left-commute [of a])

lemma *union*:
 $F (A \cup B) = F \ A * F \ B$
by (*induct A rule: infinite-finite-induct*)
(simp-all add: ac-simps)

lemma *remove*:
assumes $a \in A$
shows $F \ A = a * F (A - \{a\})$
proof –
from *assms* **obtain** B **where** $A = \text{insert } a \ B$ **and** $a \notin B$
by (*blast dest: mk-disjoint-insert*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *insert-remove*:
 $F (\text{insert } a \ A) = a * F (A - \{a\})$
by (*cases a ∈ A*) (*simp-all add: insert-absorb remove*)

lemma *subset*:
assumes $B \subseteq A$
shows $F \ B * F \ A = F \ A$
using *assms* **by** (*simp add: union [symmetric] Un-absorb1*)

end

88.2 Abstract GCD and LCM

class *gcd* = *zero* + *one* + *dvd* +
fixes $\text{gcd} :: 'a \Rightarrow 'a \Rightarrow 'a$
and $\text{lcm} :: 'a \Rightarrow 'a \Rightarrow 'a$

class *Gcd* = *gcd* +


```

fixes Gcd :: 'a set  $\Rightarrow$  'a
and Lcm :: 'a set  $\Rightarrow$  'a

```

syntax

```

-GCD1  :: ptrns  $\Rightarrow$  'b  $\Rightarrow$  'b      (( $\exists$ GCD -./ -) [0, 10] 10)
-GCD   :: ptrn   $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b  (( $\exists$ GCD - $\in$ -./ -) [0, 0, 10] 10)
-LCM1  :: ptrns  $\Rightarrow$  'b  $\Rightarrow$  'b      (( $\exists$ LCM -./ -) [0, 10] 10)
-LCM   :: ptrn   $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b  (( $\exists$ LCM - $\in$ -./ -) [0, 0, 10] 10)

```

translations

```

GCD x y. f  $\equiv$  GCD x. GCD y. f
GCD x. f    $\equiv$  CONST Gcd (CONST range ( $\lambda$ x. f))
GCD x $\in$ A. f  $\equiv$  CONST Gcd (( $\lambda$ x. f) ' A)
LCM x y. f  $\equiv$  LCM x. LCM y. f
LCM x. f    $\equiv$  CONST Lcm (CONST range ( $\lambda$ x. f))
LCM x $\in$ A. f  $\equiv$  CONST Lcm (( $\lambda$ x. f) ' A)

```

```

class semiring-gcd = normalization-semidom + gcd +
assumes gcd-dvd1 [iff]: gcd a b dvd a
and gcd-dvd2 [iff]: gcd a b dvd b
and gcd-greatest: c dvd a  $\implies$  c dvd b  $\implies$  c dvd gcd a b
and normalize-gcd [simp]: normalize (gcd a b) = gcd a b
and lcm-gcd: lcm a b = normalize (a * b div gcd a b)
begin

```

```

lemma gcd-greatest-iff [simp]: a dvd gcd b c  $\longleftrightarrow$  a dvd b  $\wedge$  a dvd c
by (blast intro!: gcd-greatest intro: dvd-trans)

```

```

lemma gcd-dvdI1: a dvd c  $\implies$  gcd a b dvd c
by (rule dvd-trans) (rule gcd-dvd1)

```

```

lemma gcd-dvdI2: b dvd c  $\implies$  gcd a b dvd c
by (rule dvd-trans) (rule gcd-dvd2)

```

```

lemma dvd-gcdD1: a dvd gcd b c  $\implies$  a dvd b
using gcd-dvd1 [of b c] by (blast intro: dvd-trans)

```

```

lemma dvd-gcdD2: a dvd gcd b c  $\implies$  a dvd c
using gcd-dvd2 [of b c] by (blast intro: dvd-trans)

```

```

lemma gcd-0-left [simp]: gcd 0 a = normalize a
by (rule associated-eqI) simp-all

```

```

lemma gcd-0-right [simp]: gcd a 0 = normalize a
by (rule associated-eqI) simp-all

```

```

lemma gcd-eq-0-iff [simp]: gcd a b = 0  $\longleftrightarrow$  a = 0  $\wedge$  b = 0
(is ?P  $\longleftrightarrow$  ?Q)

```

```

proof

```

```

assume ?P
then have 0 dvd gcd a b
  by simp
then have 0 dvd a and 0 dvd b
  by (blast intro: dvd-trans)+
then show ?Q
  by simp
next
assume ?Q
then show ?P
  by simp
qed

```

lemma *unit-factor-gcd*: $\text{unit-factor } (\text{gcd } a \ b) = (\text{if } a = 0 \wedge b = 0 \text{ then } 0 \text{ else } 1)$

proof (*cases gcd a b = 0*)

case *True*

then show ?thesis **by** simp

next

case *False*

have $\text{unit-factor } (\text{gcd } a \ b) * \text{normalize } (\text{gcd } a \ b) = \text{gcd } a \ b$

by (*rule unit-factor-mult-normalize*)

then have $\text{unit-factor } (\text{gcd } a \ b) * \text{gcd } a \ b = \text{gcd } a \ b$

by simp

then have $\text{unit-factor } (\text{gcd } a \ b) * \text{gcd } a \ b \ \text{div } \text{gcd } a \ b = \text{gcd } a \ b \ \text{div } \text{gcd } a \ b$

by simp

with *False* **show** ?thesis

by simp

qed

lemma *is-unit-gcd-iff* [*simp*]:

$\text{is-unit } (\text{gcd } a \ b) \longleftrightarrow \text{gcd } a \ b = 1$

by (*cases a = 0 ∧ b = 0*) (*auto simp: unit-factor-gcd dest: is-unit-unit-factor*)

sublocale *gcd*: *abel-semigroup gcd*

proof

fix a b c

show $\text{gcd } a \ b = \text{gcd } b \ a$

by (*rule associated-eqI*) *simp-all*

from *gcd-dvd1* **have** $\text{gcd } (\text{gcd } a \ b) \ c \ \text{dvd } a$

by (*rule dvd-trans*) *simp*

moreover from *gcd-dvd1* **have** $\text{gcd } (\text{gcd } a \ b) \ c \ \text{dvd } b$

by (*rule dvd-trans*) *simp*

ultimately have *P1*: $\text{gcd } (\text{gcd } a \ b) \ c \ \text{dvd } \text{gcd } a \ (\text{gcd } b \ c)$

by (*auto intro!: gcd-greatest*)

from *gcd-dvd2* **have** $\text{gcd } a \ (\text{gcd } b \ c) \ \text{dvd } b$

by (*rule dvd-trans*) *simp*

moreover from *gcd-dvd2* **have** $\text{gcd } a \ (\text{gcd } b \ c) \ \text{dvd } c$

by (*rule dvd-trans*) *simp*

ultimately have *P2*: $\text{gcd } a \ (\text{gcd } b \ c) \ \text{dvd } \text{gcd } (\text{gcd } a \ b) \ c$

by (auto intro!: gcd-greatest)
 from $P1\ P2$ show $\text{gcd} (\text{gcd } a\ b)\ c = \text{gcd } a\ (\text{gcd } b\ c)$
 by (rule associated-eqI) simp-all
 qed

sublocale gcd: bounded-quasi-semilattice gcd 0 1 normalize
proof

show $\text{gcd } a\ a = \text{normalize } a$ for a
proof –
 have $a\ \text{dvd}\ \text{gcd } a\ a$
 by (rule gcd-greatest) simp-all
 then show ?thesis
 by (auto intro: associated-eqI)
 qed
 show $\text{gcd} (\text{normalize } a)\ b = \text{gcd } a\ b$ for $a\ b$
 using gcd-dvd1 [of normalize a b]
 by (auto intro: associated-eqI)
 show $\text{gcd } 1\ a = 1$ for a
 by (rule associated-eqI) simp-all
 qed simp-all

lemma gcd-self: $\text{gcd } a\ a = \text{normalize } a$
 by (fact gcd.idem-normalize)

lemma gcd-left-idem: $\text{gcd } a\ (\text{gcd } a\ b) = \text{gcd } a\ b$
 by (fact gcd.left-idem)

lemma gcd-right-idem: $\text{gcd} (\text{gcd } a\ b)\ b = \text{gcd } a\ b$
 by (fact gcd.right-idem)

lemma gcdI:
 assumes $c\ \text{dvd}\ a$ and $c\ \text{dvd}\ b$
 and greatest: $\bigwedge d. d\ \text{dvd}\ a \implies d\ \text{dvd}\ b \implies d\ \text{dvd}\ c$
 and normalize $c = c$
 shows $c = \text{gcd } a\ b$
 by (rule associated-eqI) (auto simp: assms intro: gcd-greatest)

lemma gcd-unique:
 $d\ \text{dvd}\ a \wedge d\ \text{dvd}\ b \wedge \text{normalize } d = d \wedge (\forall e. e\ \text{dvd}\ a \wedge e\ \text{dvd}\ b \longrightarrow e\ \text{dvd}\ d) \longleftrightarrow$
 $d = \text{gcd } a\ b$
 by rule (auto intro: gcdI simp: gcd-greatest)

lemma gcd-dvd-prod: $\text{gcd } a\ b\ \text{dvd}\ k * b$
 using mult-dvd-mono [of 1] by auto

lemma gcd-proj2-if-dvd: $b\ \text{dvd}\ a \implies \text{gcd } a\ b = \text{normalize } b$
 by (rule gcdI [symmetric]) simp-all

lemma gcd-proj1-if-dvd: $a\ \text{dvd}\ b \implies \text{gcd } a\ b = \text{normalize } a$

by (rule gcdI [symmetric]) simp-all

lemma gcd-proj1-iff: $\text{gcd } m \ n = \text{normalize } m \longleftrightarrow m \ \text{dvd } n$

proof

assume *: $\text{gcd } m \ n = \text{normalize } m$

show $m \ \text{dvd } n$

proof (cases $m = 0$)

case True

with * show ?thesis by simp

next

case [simp]: False

from * have **: $m = \text{gcd } m \ n * \text{unit-factor } m$

by (simp add: unit-eq-div2)

show ?thesis

by (subst **) (simp add: mult-unit-dvd-iff)

qed

next

assume $m \ \text{dvd } n$

then show $\text{gcd } m \ n = \text{normalize } m$

by (rule gcd-proj1-if-dvd)

qed

lemma gcd-proj2-iff: $\text{gcd } m \ n = \text{normalize } n \longleftrightarrow n \ \text{dvd } m$

using gcd-proj1-iff [of $n \ m$] by (simp add: ac-simps)

lemma gcd-mult-left: $\text{gcd } (c * a) \ (c * b) = \text{normalize } (c * \text{gcd } a \ b)$

proof (cases $c = 0$)

case True

then show ?thesis by simp

next

case False

then have *: $c * \text{gcd } a \ b \ \text{dvd } \text{gcd } (c * a) \ (c * b)$

by (auto intro: gcd-greatest)

moreover from False * have $\text{gcd } (c * a) \ (c * b) \ \text{dvd } c * \text{gcd } a \ b$

by (metis div-dvd-iff-mult dvd-mult-left gcd-dvd1 gcd-dvd2 gcd-greatest mult-commute)

ultimately have $\text{normalize } (\text{gcd } (c * a) \ (c * b)) = \text{normalize } (c * \text{gcd } a \ b)$

by (auto intro: associated-eqI)

then show ?thesis

by (simp add: normalize-mult)

qed

lemma gcd-mult-right: $\text{gcd } (a * c) \ (b * c) = \text{normalize } (\text{gcd } b \ a * c)$

using gcd-mult-left [of $c \ a \ b$] by (simp add: ac-simps)

lemma dvd-lcm1 [iff]: $a \ \text{dvd } \text{lcm } a \ b$

by (metis div-mult-swap dvd-mult2 dvd-normalize-iff dvd-refl gcd-dvd2 lcm-gcd)

lemma dvd-lcm2 [iff]: $b \ \text{dvd } \text{lcm } a \ b$

by (metis dvd-div-mult dvd-mult dvd-normalize-iff dvd-refl gcd-dvd1 lcm-gcd)

```

lemma dvd-lcmI1:  $a \text{ dvd } b \implies a \text{ dvd lcm } b \ c$ 
  by (rule dvd-trans) (assumption, blast)

lemma dvd-lcmI2:  $a \text{ dvd } c \implies a \text{ dvd lcm } b \ c$ 
  by (rule dvd-trans) (assumption, blast)

lemma lcm-dvdD1:  $\text{lcm } a \ b \ \text{dvd } c \implies a \ \text{dvd } c$ 
  using dvd-lcm1 [of a b] by (blast intro: dvd-trans)

lemma lcm-dvdD2:  $\text{lcm } a \ b \ \text{dvd } c \implies b \ \text{dvd } c$ 
  using dvd-lcm2 [of a b] by (blast intro: dvd-trans)

lemma lcm-least:
  assumes  $a \ \text{dvd } c$  and  $b \ \text{dvd } c$ 
  shows  $\text{lcm } a \ b \ \text{dvd } c$ 
proof (cases c = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have  $*$ : is-unit (unit-factor c)
    by simp
  show ?thesis
  proof (cases gcd a b = 0)
    case True
    with assms show ?thesis by simp
  next
  case False
  have  $a * b \ \text{dvd normalize } (c * \text{gcd } a \ b)$ 
    using assms by (subst gcd-mult-left [symmetric]) (auto intro!: gcd-greatest
simp: mult-ac)
  with False have  $(a * b \ \text{div } \text{gcd } a \ b) \ \text{dvd } c$ 
    by (subst div-dvd-iff-mult) auto
  thus ?thesis by (simp add: lcm-gcd)
  qed
qed

lemma lcm-least-iff [simp]:  $\text{lcm } a \ b \ \text{dvd } c \iff a \ \text{dvd } c \wedge b \ \text{dvd } c$ 
  by (blast intro!: lcm-least intro: dvd-trans)

lemma normalize-lcm [simp]:  $\text{normalize } (\text{lcm } a \ b) = \text{lcm } a \ b$ 
  by (simp add: lcm-gcd dvd-normalize-div)

lemma lcm-0-left [simp]:  $\text{lcm } 0 \ a = 0$ 
  by (simp add: lcm-gcd)

lemma lcm-0-right [simp]:  $\text{lcm } a \ 0 = 0$ 
  by (simp add: lcm-gcd)

```

lemma *lcm-eq-0-iff*: $\text{lcm } a \ b = 0 \longleftrightarrow a = 0 \vee b = 0$
 (is $?P \longleftrightarrow ?Q$)

proof

assume $?P$
 then have $0 \ \text{dvd} \ \text{lcm } a \ b$
 by *simp*
 also have $\text{lcm } a \ b \ \text{dvd} \ (a * b)$
 by *simp*
 finally show $?Q$
 by *auto*

next

assume $?Q$
 then show $?P$
 by *auto*

qed

lemma *zero-eq-lcm-iff*: $0 = \text{lcm } a \ b \longleftrightarrow a = 0 \vee b = 0$
using *lcm-eq-0-iff*[of $a \ b$] **by** *auto*

lemma *lcm-eq-1-iff* [*simp*]: $\text{lcm } a \ b = 1 \longleftrightarrow \text{is-unit } a \wedge \text{is-unit } b$
by (*auto intro: associated-eqI*)

lemma *unit-factor-lcm*: $\text{unit-factor } (\text{lcm } a \ b) = (\text{if } a = 0 \vee b = 0 \text{ then } 0 \text{ else } 1)$
using *lcm-eq-0-iff*[of $a \ b$] **by** (*cases lcm a b = 0*) (*auto simp: lcm-gcd*)

sublocale *lcm*: *abel-semigroup lcm*

proof

fix $a \ b \ c$
 show $\text{lcm } a \ b = \text{lcm } b \ a$
 by (*simp add: lcm-gcd ac-simps normalize-mult dvd-normalize-div*)
 have $\text{lcm } (\text{lcm } a \ b) \ c \ \text{dvd} \ \text{lcm } a \ (\text{lcm } b \ c)$
 and $\text{lcm } a \ (\text{lcm } b \ c) \ \text{dvd} \ \text{lcm } (\text{lcm } a \ b) \ c$
 by (*auto intro: lcm-least*
 dvd-trans [of b lcm b c lcm a (lcm b c)]
 dvd-trans [of c lcm b c lcm a (lcm b c)]
 dvd-trans [of a lcm a b lcm (lcm a b) c]
 dvd-trans [of b lcm a b lcm (lcm a b) c])
 then show $\text{lcm } (\text{lcm } a \ b) \ c = \text{lcm } a \ (\text{lcm } b \ c)$
 by (*rule associated-eqI*) *simp-all*

qed

sublocale *lcm*: *bounded-quasi-semilattice lcm 1 0 normalize*

proof

show $\text{lcm } a \ a = \text{normalize } a$ **for** a
 proof –
 have $\text{lcm } a \ a \ \text{dvd} \ a$
 by (*rule lcm-least*) *simp-all*
 then show $?thesis$

```

    by (auto intro: associated-eqI)
  qed
  show lcm (normalize a) b = lcm a b for a b
    using dvd-lcm1 [of normalize a b] unfolding normalize-dvd-iff
    by (auto intro: associated-eqI)
  show lcm 1 a = normalize a for a
    by (rule associated-eqI) simp-all
  qed simp-all

```

```

lemma lcm-self: lcm a a = normalize a
  by (fact lcm.idem-normalize)

```

```

lemma lcm-left-idem: lcm a (lcm a b) = lcm a b
  by (fact lcm.left-idem)

```

```

lemma lcm-right-idem: lcm (lcm a b) b = lcm a b
  by (fact lcm.right-idem)

```

```

lemma gcd-lcm:
  assumes a ≠ 0 and b ≠ 0
  shows gcd a b = normalize (a * b div lcm a b)
  proof -
    from assms have [simp]: a * b div gcd a b ≠ 0
      by (subst dvd-div-eq-0-iff) auto
    let ?u = unit-factor (a * b div gcd a b)
    have gcd a b * normalize (a * b div gcd a b) =
      gcd a b * ((a * b div gcd a b) * (1 div ?u))
      by simp
    also have ... = a * b div ?u
      by (subst mult.assoc [symmetric]) auto
    also have ... dvd a * b
      by (subst div-unit-dvd-iff) auto
    finally have gcd a b dvd ((a * b) div lcm a b)
      by (intro dvd-mult-imp-div) (auto simp: lcm-gcd)
    moreover have a * b div lcm a b dvd a and a * b div lcm a b dvd b
      using assms by (subst div-dvd-iff-mult; simp add: lcm-eq-0-iff mult.commute[of
b lcm a b])+
    ultimately have normalize (gcd a b) = normalize (a * b div lcm a b)
      apply -
      apply (rule associated-eqI)
      using assms
      apply (auto simp: div-dvd-iff-mult zero-eq-lcm-iff)
      done
    thus ?thesis by simp
  qed

```

```

lemma lcm-1-left: lcm 1 a = normalize a
  by (fact lcm.top-left-normalize)

```

lemma *lcm-1-right*: $\text{lcm } a \ 1 = \text{normalize } a$
by (*fact lcm.top-right-normalize*)

lemma *lcm-mult-left*: $\text{lcm } (c * a) \ (c * b) = \text{normalize } (c * \text{lcm } a \ b)$
proof (*cases c = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $*$: $\text{lcm } (c * a) \ (c * b) \ \text{dvd } c * \text{lcm } a \ b$
by (*auto intro: lcm-least*)
moreover have $\text{lcm } a \ b \ \text{dvd } \text{lcm } (c * a) \ (c * b) \ \text{div } c$
by (*intro lcm-least*) (*auto intro!: dvd-mult-imp-div simp: mult-ac*)
hence $c * \text{lcm } a \ b \ \text{dvd } \text{lcm } (c * a) \ (c * b)$
using *False* **by** (*subst (asm) dvd-div-iff-mult*) (*auto simp: mult-ac intro: dvd-lcmI1*)
ultimately have $\text{normalize } (\text{lcm } (c * a) \ (c * b)) = \text{normalize } (c * \text{lcm } a \ b)$
by (*auto intro: associated-eqI*)
then show *?thesis*
by (*simp add: normalize-mult*)
qed

lemma *lcm-mult-right*: $\text{lcm } (a * c) \ (b * c) = \text{normalize } (\text{lcm } b \ a * c)$
using *lcm-mult-left* [*of c a b*] **by** (*simp add: ac-simps*)

lemma *lcm-mult-unit1*: $\text{is-unit } a \implies \text{lcm } (b * a) \ c = \text{lcm } b \ c$
by (*rule associated-eqI*) (*simp-all add: mult-unit-dvd-iff dvd-lcmI1*)

lemma *lcm-mult-unit2*: $\text{is-unit } a \implies \text{lcm } b \ (c * a) = \text{lcm } b \ c$
using *lcm-mult-unit1* [*of a c b*] **by** (*simp add: ac-simps*)

lemma *lcm-div-unit1*:
 $\text{is-unit } a \implies \text{lcm } (b \ \text{div } a) \ c = \text{lcm } b \ c$
by (*erule is-unitE* [*of - b*]) (*simp add: lcm-mult-unit1*)

lemma *lcm-div-unit2*: $\text{is-unit } a \implies \text{lcm } b \ (c \ \text{div } a) = \text{lcm } b \ c$
by (*erule is-unitE* [*of - c*]) (*simp add: lcm-mult-unit2*)

lemma *normalize-lcm-left*: $\text{lcm } (\text{normalize } a) \ b = \text{lcm } a \ b$
by (*fact lcm.normalize-left-idem*)

lemma *normalize-lcm-right*: $\text{lcm } a \ (\text{normalize } b) = \text{lcm } a \ b$
by (*fact lcm.normalize-right-idem*)

lemma *comp-fun-idem-gcd*: *comp-fun-idem gcd*
by *standard* (*simp-all add: fun-eq-iff ac-simps*)

lemma *comp-fun-idem-lcm*: *comp-fun-idem lcm*
by *standard* (*simp-all add: fun-eq-iff ac-simps*)

lemma *gcd-dvd-antisym*: $\text{gcd } a \ b \ \text{dvd } \text{gcd } c \ d \implies \text{gcd } c \ d \ \text{dvd } \text{gcd } a \ b \implies \text{gcd } a \ b = \text{gcd } c \ d$

proof (*rule gcdI*)
assume *: $\text{gcd } a \ b \ \text{dvd } \text{gcd } c \ d$
and **: $\text{gcd } c \ d \ \text{dvd } \text{gcd } a \ b$
have $\text{gcd } c \ d \ \text{dvd } c$
by *simp*
with * **show** $\text{gcd } a \ b \ \text{dvd } c$
by (*rule dvd-trans*)
have $\text{gcd } c \ d \ \text{dvd } d$
by *simp*
with * **show** $\text{gcd } a \ b \ \text{dvd } d$
by (*rule dvd-trans*)
show $\text{normalize } (\text{gcd } a \ b) = \text{gcd } a \ b$
by *simp*
fix l **assume** $l \ \text{dvd } c$ **and** $l \ \text{dvd } d$
then have $l \ \text{dvd } \text{gcd } c \ d$
by (*rule gcd-greatest*)
from *this* **and** ** **show** $l \ \text{dvd } \text{gcd } a \ b$
by (*rule dvd-trans*)
qed

declare *unit-factor-lcm* [*simp*]

lemma *lcmI*:
assumes $a \ \text{dvd } c$ **and** $b \ \text{dvd } c$ **and** $\bigwedge d. a \ \text{dvd } d \implies b \ \text{dvd } d \implies c \ \text{dvd } d$
and $\text{normalize } c = c$
shows $c = \text{lcm } a \ b$
by (*rule associated-eqI*) (*auto simp: assms intro: lcm-least*)

lemma *gcd-dvd-lcm* [*simp*]: $\text{gcd } a \ b \ \text{dvd } \text{lcm } a \ b$
using *gcd-dvd2* **by** (*rule dvd-lcmI2*)

lemmas *lcm-0 = lcm-0-right*

lemma *lcm-unique*:
 $a \ \text{dvd } d \wedge b \ \text{dvd } d \wedge \text{normalize } d = d \wedge (\forall e. a \ \text{dvd } e \wedge b \ \text{dvd } e \longrightarrow d \ \text{dvd } e) \longleftrightarrow$
 $d = \text{lcm } a \ b$
by *rule* (*auto intro: lcmI simp: lcm-least lcm-eq-0-iff*)

lemma *lcm-proj1-if-dvd*:
assumes $b \ \text{dvd } a$ **shows** $\text{lcm } a \ b = \text{normalize } a$
proof –
have $\text{normalize } (\text{lcm } a \ b) = \text{normalize } a$
by (*rule associatedI*) (*use assms in auto*)
thus *?thesis* **by** *simp*
qed

lemma *lcm-proj2-if-dvd*: $a \ \text{dvd } b \implies \text{lcm } a \ b = \text{normalize } b$

using *lcm-proj1-if-dvd* [of *a b*] by (*simp add: ac-simps*)

lemma *lcm-proj1-iff*: $\text{lcm } m \ n = \text{normalize } m \longleftrightarrow n \ \text{dvd } m$

proof

assume *: $\text{lcm } m \ n = \text{normalize } m$

show $n \ \text{dvd } m$

proof (*cases m = 0*)

case *True*

then show *?thesis* by *simp*

next

case [*simp*]: *False*

from * have **: $m = \text{lcm } m \ n * \text{unit-factor } m$

by (*simp add: unit-eq-div2*)

show *?thesis* by (*subst ***) *simp*

qed

next

assume $n \ \text{dvd } m$

then show $\text{lcm } m \ n = \text{normalize } m$

by (*rule lcm-proj1-if-dvd*)

qed

lemma *lcm-proj2-iff*: $\text{lcm } m \ n = \text{normalize } n \longleftrightarrow m \ \text{dvd } n$

using *lcm-proj1-iff* [of *n m*] by (*simp add: ac-simps*)

lemma *gcd-mono*: $a \ \text{dvd } c \implies b \ \text{dvd } d \implies \text{gcd } a \ b \ \text{dvd } \text{gcd } c \ d$

by (*simp add: gcd-dvdI1 gcd-dvdI2*)

lemma *lcm-mono*: $a \ \text{dvd } c \implies b \ \text{dvd } d \implies \text{lcm } a \ b \ \text{dvd } \text{lcm } c \ d$

by (*simp add: dvd-lcmI1 dvd-lcmI2*)

lemma *dvd-productE*:

assumes $p \ \text{dvd } a * b$

obtains $x \ y$ where $p = x * y$ $x \ \text{dvd } a$ $y \ \text{dvd } b$

proof (*cases a = 0*)

case *True*

thus *?thesis* by (*intro that*[of *p 1*]) *simp-all*

next

case *False*

define $x \ y$ where $x = \text{gcd } a \ p$ and $y = p \ \text{div } x$

have $p = x * y$ by (*simp add: x-def y-def*)

moreover have $x \ \text{dvd } a$ by (*simp add: x-def*)

moreover from *assms* have $p \ \text{dvd } \text{gcd } (b * a) (b * p)$

by (*intro gcd-greatest*) (*simp-all add: mult.commute*)

hence $p \ \text{dvd } b * \text{gcd } a \ p$ by (*subst (asm) gcd-mult-left*) *auto*

with *False* have $y \ \text{dvd } b$

by (*simp add: x-def y-def div-dvd-iff-mult assms*)

ultimately show *?thesis* by (*rule that*)

qed

```

lemma gcd-mult-unit1:
  assumes is-unit a shows gcd (b * a) c = gcd b c
proof –
  have gcd (b * a) c dvd b
    using assms dvd-mult-unit-iff by blast
  then show ?thesis
    by (rule gcdI) simp-all
qed

lemma gcd-mult-unit2: is-unit a  $\implies$  gcd b (c * a) = gcd b c
  using gcd.commute gcd-mult-unit1 by auto

lemma gcd-div-unit1: is-unit a  $\implies$  gcd (b div a) c = gcd b c
  by (erule is-unitE [of - b]) (simp add: gcd-mult-unit1)

lemma gcd-div-unit2: is-unit a  $\implies$  gcd b (c div a) = gcd b c
  by (erule is-unitE [of - c]) (simp add: gcd-mult-unit2)

lemma normalize-gcd-left: gcd (normalize a) b = gcd a b
  by (fact gcd.normalize-left-idem)

lemma normalize-gcd-right: gcd a (normalize b) = gcd a b
  by (fact gcd.normalize-right-idem)

lemma gcd-add1 [simp]: gcd (m + n) n = gcd m n
  by (rule gcdI [symmetric]) (simp-all add: dvd-add-left-iff)

lemma gcd-add2 [simp]: gcd m (m + n) = gcd m n
  using gcd-add1 [of n m] by (simp add: ac-simps)

lemma gcd-add-mult: gcd m (k * m + n) = gcd m n
  by (rule gcdI [symmetric]) (simp-all add: dvd-add-right-iff)

end

class ring-gcd = comm-ring-1 + semiring-gcd
begin

lemma gcd-neg1 [simp]: gcd (-a) b = gcd a b
  by (rule sym, rule gcdI) (simp-all add: gcd-greatest)

lemma gcd-neg2 [simp]: gcd a (-b) = gcd a b
  by (rule sym, rule gcdI) (simp-all add: gcd-greatest)

lemma gcd-neg-numeral-1 [simp]: gcd (- numeral n) a = gcd (numeral n) a
  by (fact gcd-neg1)

lemma gcd-neg-numeral-2 [simp]: gcd a (- numeral n) = gcd a (numeral n)
  by (fact gcd-neg2)

```

lemma *gcd-diff1*: $\text{gcd } (m - n) \ n = \text{gcd } m \ n$
by (*subst diff-conv-add-uminus*, *subst gcd-neg2[symmetric]*, *subst gcd-add1*, *simp*)

lemma *gcd-diff2*: $\text{gcd } (n - m) \ n = \text{gcd } m \ n$
by (*subst gcd-neg1[symmetric]*) (*simp only: minus-diff-eq gcd-diff1*)

lemma *lcm-neg1* [*simp*]: $\text{lcm } (-a) \ b = \text{lcm } a \ b$
by (*rule sym*, *rule lcmI*) (*simp-all add: lcm-least lcm-eq-0-iff*)

lemma *lcm-neg2* [*simp*]: $\text{lcm } a \ (-b) = \text{lcm } a \ b$
by (*rule sym*, *rule lcmI*) (*simp-all add: lcm-least lcm-eq-0-iff*)

lemma *lcm-neg-numeral-1* [*simp*]: $\text{lcm } (- \text{numeral } n) \ a = \text{lcm } (\text{numeral } n) \ a$
by (*fact lcm-neg1*)

lemma *lcm-neg-numeral-2* [*simp*]: $\text{lcm } a \ (- \text{numeral } n) = \text{lcm } a \ (\text{numeral } n)$
by (*fact lcm-neg2*)

end

class *semiring-Gcd* = *semiring-gcd* + *Gcd* +
assumes *Gcd-dvd*: $a \in A \implies \text{Gcd } A \ \text{dvd } a$
and *Gcd-greatest*: $(\bigwedge b. b \in A \implies a \ \text{dvd } b) \implies a \ \text{dvd } \text{Gcd } A$
and *normalize-Gcd* [*simp*]: $\text{normalize } (\text{Gcd } A) = \text{Gcd } A$
assumes *dvd-Lcm*: $a \in A \implies a \ \text{dvd } \text{Lcm } A$
and *Lcm-least*: $(\bigwedge b. b \in A \implies b \ \text{dvd } a) \implies \text{Lcm } A \ \text{dvd } a$
and *normalize-Lcm* [*simp*]: $\text{normalize } (\text{Lcm } A) = \text{Lcm } A$
begin

lemma *Lcm-Gcd*: $\text{Lcm } A = \text{Gcd } \{b. \forall a \in A. a \ \text{dvd } b\}$
by (*rule associated-eqI*) (*auto intro: Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least*)

lemma *Gcd-Lcm*: $\text{Gcd } A = \text{Lcm } \{b. \forall a \in A. b \ \text{dvd } a\}$
by (*rule associated-eqI*) (*auto intro: Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least*)

lemma *Gcd-empty* [*simp*]: $\text{Gcd } \{\} = 0$
by (*rule dvd-0-left*, *rule Gcd-greatest*) *simp*

lemma *Lcm-empty* [*simp*]: $\text{Lcm } \{\} = 1$
by (*auto intro: associated-eqI Lcm-least*)

lemma *Gcd-insert* [*simp*]: $\text{Gcd } (\text{insert } a \ A) = \text{gcd } a \ (\text{Gcd } A)$
proof –
have $\text{Gcd } (\text{insert } a \ A) \ \text{dvd } \text{gcd } a \ (\text{Gcd } A)$
by (*auto intro: Gcd-dvd Gcd-greatest*)
moreover have $\text{gcd } a \ (\text{Gcd } A) \ \text{dvd } \text{Gcd } (\text{insert } a \ A)$
proof (*rule Gcd-greatest*)
fix b

```

assume  $b \in \text{insert } a \ A$ 
then show  $\text{gcd } a \ (\text{Gcd } A) \ \text{dvd } b$ 
proof
  assume  $b = a$ 
  then show ?thesis
    by simp
next
  assume  $b \in A$ 
  then have  $\text{Gcd } A \ \text{dvd } b$ 
    by (rule Gcd-dvd)
  moreover have  $\text{gcd } a \ (\text{Gcd } A) \ \text{dvd } \text{Gcd } A$ 
    by simp
  ultimately show ?thesis
    by (blast intro: dvd-trans)
qed
qed
ultimately show ?thesis
  by (auto intro: associated-eqI)
qed

lemma Lcm-insert [simp]:  $\text{Lcm } (\text{insert } a \ A) = \text{lcm } a \ (\text{Lcm } A)$ 
proof (rule sym)
  have  $\text{lcm } a \ (\text{Lcm } A) \ \text{dvd } \text{Lcm } (\text{insert } a \ A)$ 
    by (auto intro: dvd-Lcm Lcm-least)
  moreover have  $\text{Lcm } (\text{insert } a \ A) \ \text{dvd } \text{lcm } a \ (\text{Lcm } A)$ 
proof (rule Lcm-least)
  fix  $b$ 
  assume  $b \in \text{insert } a \ A$ 
  then show  $b \ \text{dvd } \text{lcm } a \ (\text{Lcm } A)$ 
proof
  assume  $b = a$ 
  then show ?thesis by simp
next
  assume  $b \in A$ 
  then have  $b \ \text{dvd } \text{Lcm } A$ 
    by (rule dvd-Lcm)
  moreover have  $\text{Lcm } A \ \text{dvd } \text{lcm } a \ (\text{Lcm } A)$ 
    by simp
  ultimately show ?thesis
    by (blast intro: dvd-trans)
qed
qed
ultimately show  $\text{lcm } a \ (\text{Lcm } A) = \text{Lcm } (\text{insert } a \ A)$ 
  by (rule associated-eqI) (simp-all add: lcm-eq-0-iff)
qed

lemma LcmI:
  assumes  $\bigwedge a. a \in A \implies a \ \text{dvd } b$ 
  and  $\bigwedge c. (\bigwedge a. a \in A \implies a \ \text{dvd } c) \implies b \ \text{dvd } c$ 

```

and *normalize* $b = b$
shows $b = \text{Lcm } A$
by (*rule associated-eqI*) (*auto simp: assms dvd-Lcm intro: Lcm-least*)

lemma *Lcm-subset*: $A \subseteq B \implies \text{Lcm } A \text{ dvd } \text{Lcm } B$
by (*blast intro: Lcm-least dvd-Lcm*)

lemma *Lcm-Un*: $\text{Lcm } (A \cup B) = \text{lcm } (\text{Lcm } A) (\text{Lcm } B)$
proof –
have $\bigwedge d. [\text{Lcm } A \text{ dvd } d; \text{Lcm } B \text{ dvd } d] \implies \text{Lcm } (A \cup B) \text{ dvd } d$
by (*meson UnE Lcm-least dvd-Lcm dvd-trans*)
then show *?thesis*
by (*meson Lcm-subset lcm-unique normalize-Lcm sup.cobounded1 sup.cobounded2*)
qed

lemma *Gcd-0-iff* [*simp*]: $\text{Gcd } A = 0 \longleftrightarrow A \subseteq \{0\}$
(is *?P* \longleftrightarrow *?Q*)

proof
assume *?P*
show *?Q*
proof
fix a
assume $a \in A$
then have $\text{Gcd } A \text{ dvd } a$
by (*rule Gcd-dvd*)
with $\langle ?P \rangle$ **have** $a = 0$
by *simp*
then show $a \in \{0\}$
by *simp*

qed

next
assume *?Q*
have $0 \text{ dvd } \text{Gcd } A$
proof (*rule Gcd-greatest*)
fix a
assume $a \in A$
with $\langle ?Q \rangle$ **have** $a = 0$
by *auto*
then show $0 \text{ dvd } a$
by *simp*
qed
then show *?P*
by *simp*

qed

lemma *Lcm-1-iff* [*simp*]: $\text{Lcm } A = 1 \longleftrightarrow (\forall a \in A. \text{is-unit } a)$
(is *?P* \longleftrightarrow *?Q*)

proof
assume *?P*

```

show ?Q
proof
  fix a
  assume  $a \in A$ 
  then have  $a \text{ dvd } \text{Lcm } A$ 
    by (rule dvd-Lcm)
  with  $\langle ?P \rangle$  show  $\text{is-unit } a$ 
    by simp
  qed
next
  assume ?Q
  then have  $\text{is-unit } (\text{Lcm } A)$ 
    by (blast intro: Lcm-least)
  then have  $\text{normalize } (\text{Lcm } A) = 1$ 
    by (rule is-unit-normalize)
  then show ?P
    by simp
  qed

```

lemma *unit-factor-Lcm*: $\text{unit-factor } (\text{Lcm } A) = (\text{if } \text{Lcm } A = 0 \text{ then } 0 \text{ else } 1)$

```

proof (cases  $\text{Lcm } A = 0$ )
  case True
    then show ?thesis
      by simp
  next
    case False
      with unit-factor-normalize have  $\text{unit-factor } (\text{normalize } (\text{Lcm } A)) = 1$ 
        by blast
      with False show ?thesis
        by simp
  qed

```

lemma *unit-factor-Gcd*: $\text{unit-factor } (\text{Gcd } A) = (\text{if } \text{Gcd } A = 0 \text{ then } 0 \text{ else } 1)$
by (simp add: Gcd-Lcm unit-factor-Lcm)

lemma *GcdI*:

```

assumes  $\bigwedge a. a \in A \implies b \text{ dvd } a$ 
  and  $\bigwedge c. (\bigwedge a. a \in A \implies c \text{ dvd } a) \implies c \text{ dvd } b$ 
  and  $\text{normalize } b = b$ 
shows  $b = \text{Gcd } A$ 
by (rule associated-eqI) (auto simp: assms Gcd-dvd intro: Gcd-greatest)

```

lemma *Gcd-eq-1-I*:

```

assumes  $\text{is-unit } a$  and  $a \in A$ 
shows  $\text{Gcd } A = 1$ 
proof –
  from assms have  $\text{is-unit } (\text{Gcd } A)$ 
    by (blast intro: Gcd-dvd dvd-unit-imp-unit)
  then have  $\text{normalize } (\text{Gcd } A) = 1$ 

```

by (rule is-unit-normalize)
 then show ?thesis
 by simp
 qed

lemma *Lcm-eq-0-I*:
 assumes $0 \in A$
 shows $Lcm\ A = 0$
proof –
 from *assms* have $0\ dvd\ Lcm\ A$
 by (rule dvd-Lcm)
 then show ?thesis
 by simp
 qed

lemma *Gcd-UNIV* [simp]: $Gcd\ UNIV = 1$
 using *dvd-refl* by (rule *Gcd-eq-1-I*) simp

lemma *Lcm-UNIV* [simp]: $Lcm\ UNIV = 0$
 by (rule *Lcm-eq-0-I*) simp

lemma *Lcm-0-iff*:
 assumes *finite* A
 shows $Lcm\ A = 0 \longleftrightarrow 0 \in A$
proof (*cases* $A = \{\}$)
 case *True*
 then show ?thesis by simp
next
 case *False*
 with *assms* show ?thesis
 by (*induct* A rule: *finite-ne-induct*) (*auto* simp: *lcm-eq-0-iff*)
 qed

lemma *Gcd-image-normalize* [simp]: $Gcd\ (normalize\ 'A) = Gcd\ A$
proof –
 have $Gcd\ (normalize\ 'A)\ dvd\ a$ if $a \in A$ for a
proof –
 from *that* obtain B where $A = insert\ a\ B$
 by *blast*
 moreover have $gcd\ (normalize\ a)\ (Gcd\ (normalize\ 'B))\ dvd\ normalize\ a$
 by (rule *gcd-dvd1*)
 ultimately show $Gcd\ (normalize\ 'A)\ dvd\ a$
 by *simp*
qed
 then have $Gcd\ (normalize\ 'A)\ dvd\ Gcd\ A$ and $Gcd\ A\ dvd\ Gcd\ (normalize\ 'A)$
 by (*auto* *intro!*: *Gcd-greatest* *intro*: *Gcd-dvd*)
 then show ?thesis
 by (*auto* *intro*: *associated-eqI*)
qed

lemma *Gcd-eqI*:

assumes *normalize* $a = a$

assumes $\bigwedge b. b \in A \implies a \text{ dvd } b$

and $\bigwedge c. (\bigwedge b. b \in A \implies c \text{ dvd } b) \implies c \text{ dvd } a$

shows $\text{Gcd } A = a$

using *assms* **by** (*blast intro: associated-eqI Gcd-greatest Gcd-dvd normalize-Gcd*)

lemma *dvd-GcdD*: $x \text{ dvd } \text{Gcd } A \implies y \in A \implies x \text{ dvd } y$

using *Gcd-dvd dvd-trans* **by** *blast*

lemma *dvd-Gcd-iff*: $x \text{ dvd } \text{Gcd } A \iff (\forall y \in A. x \text{ dvd } y)$

by (*blast dest: dvd-GcdD intro: Gcd-greatest*)

lemma *Gcd-mult*: $\text{Gcd } ((*) c \text{ ' } A) = \text{normalize } (c * \text{Gcd } A)$

proof (*cases* $c = 0$)

case *True*

then show *?thesis* **by** *auto*

next

case [*simp*]: *False*

have $\text{Gcd } ((*) c \text{ ' } A) \text{ div } c \text{ dvd } \text{Gcd } A$

by (*intro Gcd-greatest, subst div-dvd-iff-mult*)

(*auto intro!: Gcd-greatest Gcd-dvd simp: mult.commute[of - c]*)

then have $\text{Gcd } ((*) c \text{ ' } A) \text{ dvd } c * \text{Gcd } A$

by (*subst (asm) div-dvd-iff-mult*) (*auto intro: Gcd-greatest simp: mult-ac*)

moreover have $c * \text{Gcd } A \text{ dvd } \text{Gcd } ((*) c \text{ ' } A)$

by (*intro Gcd-greatest*) (*auto intro: mult-dvd-mono Gcd-dvd*)

ultimately have $\text{normalize } (\text{Gcd } ((*) c \text{ ' } A)) = \text{normalize } (c * \text{Gcd } A)$

by (*rule associatedI*)

then show *?thesis* **by** *simp*

qed

lemma *Lcm-eqI*:

assumes *normalize* $a = a$

and $\bigwedge b. b \in A \implies b \text{ dvd } a$

and $\bigwedge c. (\bigwedge b. b \in A \implies b \text{ dvd } c) \implies a \text{ dvd } c$

shows $\text{Lcm } A = a$

using *assms* **by** (*blast intro: associated-eqI Lcm-least dvd-Lcm normalize-Lcm*)

lemma *Lcm-dvdD*: $\text{Lcm } A \text{ dvd } x \implies y \in A \implies y \text{ dvd } x$

using *dvd-Lcm dvd-trans* **by** *blast*

lemma *Lcm-dvd-iff*: $\text{Lcm } A \text{ dvd } x \iff (\forall y \in A. y \text{ dvd } x)$

by (*blast dest: Lcm-dvdD intro: Lcm-least*)

lemma *Lcm-mult*:

assumes $A \neq \{\}$

shows $\text{Lcm } ((*) c \text{ ' } A) = \text{normalize } (c * \text{Lcm } A)$

proof (*cases* $c = 0$)

```

case True
with assms have  $(*) c \text{ ' } A = \{0\}$ 
  by auto
with True show ?thesis by auto
next
case [simp]: False
from assms obtain x where  $x: x \in A$ 
  by blast
have  $c \text{ dvd } c * x$ 
  by simp
also from x have  $c * x \text{ dvd } \text{Lcm } ((*) c \text{ ' } A)$ 
  by (intro dvd-Lcm) auto
finally have  $\text{dvd}: c \text{ dvd } \text{Lcm } ((*) c \text{ ' } A)$  .
moreover have  $\text{Lcm } A \text{ dvd } \text{Lcm } ((*) c \text{ ' } A) \text{ div } c$ 
  by (intro Lcm-least dvd-mult-imp-div)
  (auto intro!: Lcm-least dvd-Lcm simp: mult.commute[of - c])
ultimately have  $c * \text{Lcm } A \text{ dvd } \text{Lcm } ((*) c \text{ ' } A)$ 
  by auto
moreover have  $\text{Lcm } ((*) c \text{ ' } A) \text{ dvd } c * \text{Lcm } A$ 
  by (intro Lcm-least) (auto intro: mult-dvd-mono dvd-Lcm)
ultimately have  $\text{normalize } (c * \text{Lcm } A) = \text{normalize } (\text{Lcm } ((*) c \text{ ' } A))$ 
  by (rule associatedI)
then show ?thesis by simp
qed

```

lemma *Lcm-no-units*: $\text{Lcm } A = \text{Lcm } (A - \{a. \text{is-unit } a\})$

proof –

have $(A - \{a. \text{is-unit } a\}) \cup \{a \in A. \text{is-unit } a\} = A$

by *blast*

then have $\text{Lcm } A = \text{lcm } (\text{Lcm } (A - \{a. \text{is-unit } a\})) (\text{Lcm } \{a \in A. \text{is-unit } a\})$

by (*simp add: Lcm-Un* [*symmetric*])

also have $\text{Lcm } \{a \in A. \text{is-unit } a\} = 1$

by *simp*

finally show *?thesis*

by *simp*

qed

lemma *Lcm-0-iff'*: $\text{Lcm } A = 0 \iff (\nexists l. l \neq 0 \wedge (\forall a \in A. a \text{ dvd } l))$

by (*metis Lcm-least dvd-0-left dvd-Lcm*)

lemma *Lcm-no-multiple*: $(\forall m. m \neq 0 \implies (\exists a \in A. \neg a \text{ dvd } m)) \implies \text{Lcm } A = 0$

by (*auto simp: Lcm-0-iff'*)

lemma *Lcm-singleton* [*simp*]: $\text{Lcm } \{a\} = \text{normalize } a$

by *simp*

lemma *Lcm-2* [*simp*]: $\text{Lcm } \{a, b\} = \text{lcm } a \ b$

by *simp*

lemma *Gcd-1*: $1 \in A \implies \text{Gcd } A = 1$
by (*auto intro!*: *Gcd-eq-1-I*)

lemma *Gcd-singleton* [*simp*]: $\text{Gcd } \{a\} = \text{normalize } a$
by *simp*

lemma *Gcd-2* [*simp*]: $\text{Gcd } \{a, b\} = \text{gcd } a \ b$
by *simp*

lemma *Gcd-mono*:
assumes $\bigwedge x. x \in A \implies f \ x \ \text{dvd} \ g \ x$
shows $(\text{GCD } x \in A. f \ x) \ \text{dvd} \ (\text{GCD } x \in A. g \ x)$
proof (*intro Gcd-greatest, safe*)
fix x **assume** $x \in A$
hence $(\text{GCD } x \in A. f \ x) \ \text{dvd} \ f \ x$
by (*intro Gcd-dvd*) *auto*
also have $f \ x \ \text{dvd} \ g \ x$
using $\langle x \in A \rangle$ *assms* **by** *blast*
finally show $(\text{GCD } x \in A. f \ x) \ \text{dvd} \ \dots$
qed

lemma *Lcm-mono*:
assumes $\bigwedge x. x \in A \implies f \ x \ \text{dvd} \ g \ x$
shows $(\text{LCM } x \in A. f \ x) \ \text{dvd} \ (\text{LCM } x \in A. g \ x)$
proof (*intro Lcm-least, safe*)
fix x **assume** $x \in A$
hence $f \ x \ \text{dvd} \ g \ x$ **by** (*rule assms*)
also have $g \ x \ \text{dvd} \ (\text{LCM } x \in A. g \ x)$
using $\langle x \in A \rangle$ **by** (*intro dvd-Lcm*) *auto*
finally show $f \ x \ \text{dvd} \ \dots$
qed

end

88.3 An aside: GCD and LCM on finite sets for incomplete gcd rings

context *semiring-gcd*
begin

sublocale *Gcd-fin*: *bounded-quasi-semilattice-set gcd 0 1 normalize*
defines

$\text{Gcd-fin } (\text{Gcd}_{fin}) = \text{Gcd-fin.F} :: 'a \ \text{set} \Rightarrow 'a \ ..$

abbreviation *gcd-list* :: $'a \ \text{list} \Rightarrow 'a$
where $\text{gcd-list } xs \equiv \text{Gcd}_{fin} \ (\text{set } xs)$

sublocale *Lcm-fin*: *bounded-quasi-semilattice-set lcm 1 0 normalize*
defines

$Lcm_fin (Lcm_fin) = Lcm_fin.F ..$

abbreviation $lcm_list :: 'a list \Rightarrow 'a$
where $lcm_list\ xs \equiv Lcm_fin (set\ xs)$

lemma Gcd_fin_dvd :
 $a \in A \implies Gcd_fin\ A\ dvd\ a$
by (*induct A rule: infinite-finite-induct*)
(auto intro: dvd-trans)

lemma dvd_Lcm_fin :
 $a \in A \implies a\ dvd\ Lcm_fin\ A$
by (*induct A rule: infinite-finite-induct*)
(auto intro: dvd-trans)

lemma $Gcd_fin_greatest$:
 $a\ dvd\ Gcd_fin\ A$ **if** $finite\ A$ **and** $\bigwedge b. b \in A \implies a\ dvd\ b$
using that by (*induct A*) *simp-all*

lemma Lcm_fin_least :
 $Lcm_fin\ A\ dvd\ a$ **if** $finite\ A$ **and** $\bigwedge b. b \in A \implies b\ dvd\ a$
using that by (*induct A*) *simp-all*

lemma $gcd_list_greatest$:
 $a\ dvd\ gcd_list\ bs$ **if** $\bigwedge b. b \in set\ bs \implies a\ dvd\ b$
by (*rule Gcd-fin-greatest*) (*simp-all add: that*)

lemma lcm_list_least :
 $lcm_list\ bs\ dvd\ a$ **if** $\bigwedge b. b \in set\ bs \implies b\ dvd\ a$
by (*rule Lcm-fin-least*) (*simp-all add: that*)

lemma $dvd_Gcd_fin_iff$:
 $b\ dvd\ Gcd_fin\ A \iff (\forall a \in A. b\ dvd\ a)$ **if** $finite\ A$
using that by (*auto intro: Gcd-fin-greatest Gcd-fin-dvd dvd-trans [of b Gcd_fin A]*)

lemma $dvd_gcd_list_iff$:
 $b\ dvd\ gcd_list\ xs \iff (\forall a \in set\ xs. b\ dvd\ a)$
by (*simp add: dvd-Gcd-fin-iff*)

lemma $Lcm_fin_dvd_iff$:
 $Lcm_fin\ A\ dvd\ b \iff (\forall a \in A. a\ dvd\ b)$ **if** $finite\ A$
using that by (*auto intro: Lcm-fin-least dvd-Lcm-fin dvd-trans [of - Lcm_fin A b]*)

lemma $lcm_list_dvd_iff$:
 $lcm_list\ xs\ dvd\ b \iff (\forall a \in set\ xs. a\ dvd\ b)$
by (*simp add: Lcm-fin-dvd-iff*)

lemma *Gcd-fin-mult*:

$Gcd_{fin}(\text{image}(\text{times } b) A) = \text{normalize}(b * Gcd_{fin} A)$ **if** *finite A*
using *that by induction (auto simp: gcd-mult-left)*

lemma *Lcm-fin-mult*:

$Lcm_{fin}(\text{image}(\text{times } b) A) = \text{normalize}(b * Lcm_{fin} A)$ **if** $A \neq \{\}$

proof (*cases b = 0*)

case *True*

moreover from *that have times 0 ‘ A = {0}*

by *auto*

ultimately show *?thesis*

by *simp*

next

case *False*

show *?thesis proof (cases finite A)*

case *False*

moreover have *inj-on (times b) A*

using $\langle b \neq 0 \rangle$ **by** (*rule inj-on-mult*)

ultimately have *infinite (times b ‘ A)*

by (*simp add: finite-image-iff*)

with False show *?thesis*

by *simp*

next

case *True*

then show *?thesis using that*

by (*induct A rule: finite-ne-induct (auto simp: lcm-mult-left)*)

qed

qed

lemma *unit-factor-Gcd-fin*:

$\text{unit-factor}(Gcd_{fin} A) = \text{of-bool}(Gcd_{fin} A \neq 0)$

by (*rule normalize-idem-imp-unit-factor-eq simp*)

lemma *unit-factor-Lcm-fin*:

$\text{unit-factor}(Lcm_{fin} A) = \text{of-bool}(Lcm_{fin} A \neq 0)$

by (*rule normalize-idem-imp-unit-factor-eq simp*)

lemma *is-unit-Gcd-fin-iff [simp]*:

$\text{is-unit}(Gcd_{fin} A) \longleftrightarrow Gcd_{fin} A = 1$

by (*rule normalize-idem-imp-is-unit-iff simp*)

lemma *is-unit-Lcm-fin-iff [simp]*:

$\text{is-unit}(Lcm_{fin} A) \longleftrightarrow Lcm_{fin} A = 1$

by (*rule normalize-idem-imp-is-unit-iff simp*)

lemma *Gcd-fin-0-iff*:

$Gcd_{fin} A = 0 \longleftrightarrow A \subseteq \{0\} \wedge \text{finite } A$

by (*induct A rule: infinite-finite-induct simp-all*)

lemma *Lcm-fin-0-iff*:

$Lcm_{fin} A = 0 \iff 0 \in A$ **if** *finite A*
using that by (*induct A*) (*auto simp: lcm-eq-0-iff*)

lemma *Lcm-fin-1-iff*:

$Lcm_{fin} A = 1 \iff (\forall a \in A. is_unit\ a) \wedge finite\ A$
by (*induct A rule: infinite-finite-induct*) *simp-all*

end

context *semiring-Gcd*

begin

lemma *Gcd-fin-eq-Gcd* [*simp*]:

$Gcd_{fin} A = Gcd\ A$ **if** *finite A for A :: 'a set*
using that by *induct simp-all*

lemma *Gcd-set-eq-fold* [*code-unfold*]:

$Gcd\ (set\ xs) = fold\ gcd\ xs\ 0$
by (*simp add: Gcd-fin.set-eq-fold*) [*symmetric*]

lemma *Lcm-fin-eq-Lcm* [*simp*]:

$Lcm_{fin} A = Lcm\ A$ **if** *finite A for A :: 'a set*
using that by *induct simp-all*

lemma *Lcm-set-eq-fold* [*code-unfold*]:

$Lcm\ (set\ xs) = fold\ lcm\ xs\ 1$
by (*simp add: Lcm-fin.set-eq-fold*) [*symmetric*]

end

88.4 Coprimality

context *semiring-gcd*

begin

lemma *coprime-imp-gcd-eq-1* [*simp*]:

$gcd\ a\ b = 1$ **if** *coprime a b*

proof –

define *t r s* **where** $t = gcd\ a\ b$ **and** $r = a\ div\ t$ **and** $s = b\ div\ t$

then have $a = t * r$ **and** $b = t * s$

by *simp-all*

with that have *coprime (t * r) (t * s)*

by *simp*

then show *?thesis*

by (*simp add: t-def*)

qed

lemma *gcd-eq-1-imp-coprime* [*dest!*]:

```

  coprime a b if gcd a b = 1
proof (rule coprimeI)
  fix c
  assume c dvd a and c dvd b
  then have c dvd gcd a b
    by (rule gcd-greatest)
  with that show is-unit c
    by simp
qed

lemma coprime-iff-gcd-eq-1 [presburger, code]:
  coprime a b  $\longleftrightarrow$  gcd a b = 1
  by rule (simp-all add: gcd-eq-1-imp-coprime)

lemma is-unit-gcd [simp]:
  is-unit (gcd a b)  $\longleftrightarrow$  coprime a b
  by (simp add: coprime-iff-gcd-eq-1)

lemma coprime-add-one-left [simp]: coprime (a + 1) a
  by (simp add: gcd-eq-1-imp-coprime ac-simps)

lemma coprime-add-one-right [simp]: coprime a (a + 1)
  using coprime-add-one-left [of a] by (simp add: ac-simps)

lemma coprime-mult-left-iff [simp]:
  coprime (a * b) c  $\longleftrightarrow$  coprime a c  $\wedge$  coprime b c
proof
  assume coprime (a * b) c
  with coprime-common-divisor [of a * b c]
  have *: is-unit d if d dvd a * b and d dvd c for d
    using that by blast
  have coprime a c
    by (rule coprimeI, rule *) simp-all
  moreover have coprime b c
    by (rule coprimeI, rule *) simp-all
  ultimately show coprime a c  $\wedge$  coprime b c ..
next
  assume coprime a c  $\wedge$  coprime b c
  then have coprime a c coprime b c
    by simp-all
  show coprime (a * b) c
proof (rule coprimeI)
  fix d
  assume d dvd a * b
  then obtain r s where d: d = r * s r dvd a s dvd b
    by (rule dvd-productE)
  assume d dvd c
  with d have r * s dvd c
    by simp

```

```

then have  $r \text{ dvd } c \ s \text{ dvd } c$ 
  by (auto intro: dvd-mult-left dvd-mult-right)
from  $\langle \text{coprime } a \ c \rangle \langle r \text{ dvd } a \rangle \langle r \text{ dvd } c \rangle$ 
have  $\text{is-unit } r$ 
  by (rule coprime-common-divisor)
moreover from  $\langle \text{coprime } b \ c \rangle \langle s \text{ dvd } b \rangle \langle s \text{ dvd } c \rangle$ 
have  $\text{is-unit } s$ 
  by (rule coprime-common-divisor)
ultimately show  $\text{is-unit } d$ 
  by (simp add: d-is-unit-mult-iff)
qed

```

```

lemma coprime-mult-right-iff [simp]:
   $\text{coprime } c \ (a * b) \iff \text{coprime } c \ a \wedge \text{coprime } c \ b$ 
  using coprime-mult-left-iff [of a b c] by (simp add: ac-simps)

```

```

lemma coprime-power-left-iff [simp]:
   $\text{coprime } (a \wedge^n) \ b \iff \text{coprime } a \ b \vee n = 0$ 
proof (cases  $n = 0$ )
  case True
    then show ?thesis
      by simp
  next
    case False
      then have  $n > 0$ 
        by simp
      then show ?thesis
        by (induction n rule: nat-induct-non-zero) simp-all
qed

```

```

lemma coprime-power-right-iff [simp]:
   $\text{coprime } a \ (b \wedge^n) \iff \text{coprime } a \ b \vee n = 0$ 
  using coprime-power-left-iff [of b n a] by (simp add: ac-simps)

```

```

lemma prod-coprime-left:
   $\text{coprime } (\prod_{i \in A} f \ i) \ a \ \text{if } \bigwedge i. i \in A \implies \text{coprime } (f \ i) \ a$ 
  using that by (induct A rule: infinite-finite-induct) simp-all

```

```

lemma prod-coprime-right:
   $\text{coprime } a \ (\prod_{i \in A} f \ i) \ \text{if } \bigwedge i. i \in A \implies \text{coprime } a \ (f \ i)$ 
  using that prod-coprime-left [of A f a] by (simp add: ac-simps)

```

```

lemma prod-list-coprime-left:
   $\text{coprime } (\text{prod-list } xs) \ a \ \text{if } \bigwedge x. x \in \text{set } xs \implies \text{coprime } x \ a$ 
  using that by (induct xs) simp-all

```

```

lemma prod-list-coprime-right:
   $\text{coprime } a \ (\text{prod-list } xs) \ \text{if } \bigwedge x. x \in \text{set } xs \implies \text{coprime } a \ x$ 

```


using *that prod-list-coprime-left [of xs a]* by (*simp add: ac-simps*)

lemma *coprime-dvd-mult-left-iff*:

$a \text{ dvd } b * c \longleftrightarrow a \text{ dvd } b \text{ if coprime } a \ c$

proof

assume $a \text{ dvd } b$

then show $a \text{ dvd } b * c$

by *simp*

next

assume $a \text{ dvd } b * c$

show $a \text{ dvd } b$

proof (*cases b = 0*)

case *True*

then show *?thesis*

by *simp*

next

case *False*

then have *unit: is-unit (unit-factor b)*

by *simp*

from $\langle \text{coprime } a \ c \rangle$

have $\text{gcd } (b * a) (b * c) * \text{unit-factor } b = b$

by (*subst gcd-mult-left*) (*simp add: ac-simps*)

moreover from $\langle a \text{ dvd } b * c \rangle$

have $a \text{ dvd } \text{gcd } (b * a) (b * c) * \text{unit-factor } b$

by (*simp add: dvd-mult-unit-iff unit*)

ultimately show *?thesis*

by *simp*

qed

qed

lemma *coprime-dvd-mult-right-iff*:

$a \text{ dvd } c * b \longleftrightarrow a \text{ dvd } b \text{ if coprime } a \ c$

using *that coprime-dvd-mult-left-iff [of a c b]* by (*simp add: ac-simps*)

lemma *divides-mult*:

$a * b \text{ dvd } c \text{ if } a \text{ dvd } c \text{ and } b \text{ dvd } c \text{ and coprime } a \ b$

proof –

from $\langle b \text{ dvd } c \rangle$ obtain b' where $c = b * b' ..$

with $\langle a \text{ dvd } c \rangle$ have $a \text{ dvd } b' * b$

by (*simp add: ac-simps*)

with $\langle \text{coprime } a \ b \rangle$ have $a \text{ dvd } b'$

by (*simp add: coprime-dvd-mult-left-iff*)

then obtain a' where $b' = a * a' ..$

with $\langle c = b * b' \rangle$ have $c = (a * b) * a'$

by (*simp add: ac-simps*)

then show *?thesis ..*

qed

lemma *div-gcd-coprime*:

assumes $a \neq 0 \vee b \neq 0$
shows $\text{coprime } (a \text{ div gcd } a \ b) \ (b \text{ div gcd } a \ b)$
proof –
let $?g = \text{gcd } a \ b$
let $?a' = a \text{ div } ?g$
let $?b' = b \text{ div } ?g$
let $?g' = \text{gcd } ?a' \ ?b'$
have $\text{dvdg}: ?g \text{ dvd } a \ ?g \text{ dvd } b$
by *simp-all*
have $\text{dvdg}': ?g' \text{ dvd } ?a' \ ?g' \text{ dvd } ?b'$
by *simp-all*
from $\text{dvdg } \text{dvdg}'$ **obtain** $ka \ kb \ ka' \ kb'$ **where**
 $ka b: a = ?g * ka \ b = ?g * kb \ ?a' = ?g' * ka' \ ?b' = ?g' * kb'$
unfolding *dvd-def* **by** *blast*
from *this* [*symmetric*] **have** $?g * ?a' = (?g * ?g') * ka' \ ?g * ?b' = (?g * ?g') * kb'$
by (*simp-all add: mult.assoc mult.left-commute [of gcd a b]*)
then **have** $\text{dvdgg}': ?g * ?g' \text{ dvd } a \ ?g * ?g' \text{ dvd } b$
by (*auto simp: dvd-mult-div-cancel [OF dvdg(1)] dvd-mult-div-cancel [OF dvdg(2)] dvd-def*)
have $?g \neq 0$
using *assms* **by** *simp*
moreover **from** *gcd-greatest* [*OF dvdgg'*] **have** $?g * ?g' \text{ dvd } ?g$.
ultimately **show** *?thesis*
using *dvd-times-left-cancel-iff [of gcd a b - 1]*
by *simp (simp only: coprime-iff-gcd-eq-1)*
qed

lemma *gcd-coprime*:

assumes $c: \text{gcd } a \ b \neq 0$
and $a: a = a' * \text{gcd } a \ b$
and $b: b = b' * \text{gcd } a \ b$
shows $\text{coprime } a' \ b'$

proof –

from c **have** $a \neq 0 \vee b \neq 0$
by *simp*
with *div-gcd-coprime* **have** $\text{coprime } (a \text{ div gcd } a \ b) \ (b \text{ div gcd } a \ b)$.
also **from** *assms* **have** $a \text{ div gcd } a \ b = a'$
using *dvd-div-eq-mult gcd-dvd1* **by** *blast*
also **from** *assms* **have** $b \text{ div gcd } a \ b = b'$
using *dvd-div-eq-mult gcd-dvd1* **by** *blast*
finally **show** *?thesis* .

qed

lemma *gcd-coprime-exists*:

assumes $\text{gcd } a \ b \neq 0$
shows $\exists a' \ b'. a = a' * \text{gcd } a \ b \wedge b = b' * \text{gcd } a \ b \wedge \text{coprime } a' \ b'$

proof –

have $\text{coprime } (a \text{ div gcd } a \ b) \ (b \text{ div gcd } a \ b)$

```

    using assms div-gcd-coprime by auto
  then show ?thesis
    by force
qed

lemma pow-divides-pow-iff [simp]:
   $a^n \text{ dvd } b^n \longleftrightarrow a \text{ dvd } b \text{ if } n > 0$ 
proof (cases  $\text{gcd } a \ b = 0$ )
  case True
  then show ?thesis
    by simp
next
  case False
  show ?thesis
  proof
    let  $?d = \text{gcd } a \ b$ 
    from  $\langle n > 0 \rangle$  obtain m where  $m: n = \text{Suc } m$ 
      by (cases n) simp-all
    from False have zn:  $?d^n \neq 0$ 
      by (rule power-not-zero)
    from gcd-coprime-exists [OF False]
    obtain a' b' where ab':  $a = a' * ?d \ b = b' * ?d \ \text{coprime } a' \ b'$ 
      by blast
    assume  $a^n \text{ dvd } b^n$ 
    then have  $(a' * ?d)^n \text{ dvd } (b' * ?d)^n$ 
      by (simp add: ab'(1,2)[symmetric])
    then have  $?d^n * a'^n \text{ dvd } ?d^n * b'^n$ 
      by (simp only: power-mult-distrib ac-simps)
    with zn have  $a'^n \text{ dvd } b'^n$ 
      by simp
    then have  $a' \text{ dvd } b'$ 
      using dvd-trans[of a' a'^n b'^n] by (simp add: m)
    then have  $a' \text{ dvd } b'^m * b'$ 
      by (simp add: m ac-simps)
    moreover have coprime a' (b'^n)
      using  $\langle \text{coprime } a' \ b' \rangle$  by simp
    then have  $a' \text{ dvd } b'$ 
      using  $\langle a' \text{ dvd } b'^n \rangle$  coprime-dvd-mult-left-iff dvd-mult by blast
    then have  $a' * ?d \text{ dvd } b' * ?d$ 
      by (rule mult-dvd-mono) simp
    with ab'(1,2) show  $a \text{ dvd } b$ 
      by simp
  next
    assume  $a \text{ dvd } b$ 
    with  $\langle n > 0 \rangle$  show  $a^n \text{ dvd } b^n$ 
      by (induction rule: nat-induct-non-zero)
        (simp-all add: mult-dvd-mono)
  qed
qed

```

lemma *coprime-crossproduct*:
fixes $a\ b\ c\ d :: 'a$
assumes *coprime* $a\ d$ **and** *coprime* $b\ c$
shows $\text{normalize } a * \text{normalize } c = \text{normalize } b * \text{normalize } d \longleftrightarrow$
 $\text{normalize } a = \text{normalize } b \wedge \text{normalize } c = \text{normalize } d$
(is $?lhs \longleftrightarrow ?rhs)$
proof
assume $?rhs$
then show $?lhs$ **by** *simp*
next
assume $?lhs$
from $\langle ?lhs \rangle$ **have** $\text{normalize } a\ \text{dvd}\ \text{normalize } b * \text{normalize } d$
by (*auto intro: dvdI dest: sym*)
with $\langle \text{coprime } a\ d \rangle$ **have** $a\ \text{dvd}\ b$
by (*simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric]*)
from $\langle ?lhs \rangle$ **have** $\text{normalize } b\ \text{dvd}\ \text{normalize } a * \text{normalize } c$
by (*auto intro: dvdI dest: sym*)
with $\langle \text{coprime } b\ c \rangle$ **have** $b\ \text{dvd}\ a$
by (*simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric]*)
from $\langle ?lhs \rangle$ **have** $\text{normalize } c\ \text{dvd}\ \text{normalize } d * \text{normalize } b$
by (*auto intro: dvdI dest: sym simp add: mult.commute*)
with $\langle \text{coprime } b\ c \rangle$ **have** $c\ \text{dvd}\ d$
by (*simp add: coprime-dvd-mult-left-iff coprime-commute normalize-mult [symmetric]*)
from $\langle ?lhs \rangle$ **have** $\text{normalize } d\ \text{dvd}\ \text{normalize } c * \text{normalize } a$
by (*auto intro: dvdI dest: sym simp add: mult.commute*)
with $\langle \text{coprime } a\ d \rangle$ **have** $d\ \text{dvd}\ c$
by (*simp add: coprime-dvd-mult-left-iff coprime-commute normalize-mult [symmetric]*)
from $\langle a\ \text{dvd}\ b \rangle\ \langle b\ \text{dvd}\ a \rangle$ **have** $\text{normalize } a = \text{normalize } b$
by (*rule associatedI*)
moreover from $\langle c\ \text{dvd}\ d \rangle\ \langle d\ \text{dvd}\ c \rangle$ **have** $\text{normalize } c = \text{normalize } d$
by (*rule associatedI*)
ultimately show $?rhs$ **..**
qed

lemma *gcd-mult-left-left-cancel*:
 $\text{gcd } (c * a)\ b = \text{gcd } a\ b$ **if** *coprime* $b\ c$
proof –
have *coprime* $(\text{gcd } b\ (a * c))\ c$
by (*rule coprimeI*) (*auto intro: that coprime-common-divisor*)
then have $\text{gcd } b\ (a * c)\ \text{dvd}\ a$
using *coprime-dvd-mult-left-iff* [*of gcd b (a * c) c a*]
by *simp*
then show $?thesis$
by (*auto intro: associated-eqI simp add: ac-simps*)
qed

lemma *gcd-mult-left-right-cancel*:
 $\text{gcd } (a * c)\ b = \text{gcd } a\ b$ **if** *coprime* $b\ c$

using *that gcd-mult-left-left-cancel [of b c a]*
by (*simp add: ac-simps*)

lemma *gcd-mult-right-left-cancel:*
 $\text{gcd } a (c * b) = \text{gcd } a b$ **if** *coprime a c*
using *that gcd-mult-left-left-cancel [of a c b]*
by (*simp add: ac-simps*)

lemma *gcd-mult-right-right-cancel:*
 $\text{gcd } a (b * c) = \text{gcd } a b$ **if** *coprime a c*
using *that gcd-mult-right-left-cancel [of a c b]*
by (*simp add: ac-simps*)

lemma *gcd-exp-weak:*
 $\text{gcd } (a \wedge^n) (b \wedge^n) = \text{normalize } (\text{gcd } a b \wedge^n)$
proof (*cases a = 0 \wedge b = 0 \vee n = 0*)
case *True*
then show *?thesis*
by (*cases n*) *simp-all*
next
case *False*
then have *coprime (a div gcd a b) (b div gcd a b) and n > 0*
by (*auto intro: div-gcd-coprime*)
then have *coprime ((a div gcd a b) \wedge^n) ((b div gcd a b) \wedge^n)*
by *simp*
then have $1 = \text{gcd } ((a \text{ div gcd } a b) \wedge^n) ((b \text{ div gcd } a b) \wedge^n)$
by *simp*
then have $\text{normalize } (\text{gcd } a b \wedge^n) = \text{normalize } (\text{gcd } a b \wedge^n * \dots)$
by *simp*
also have $\dots = \text{gcd } (\text{gcd } a b \wedge^n * (a \text{ div gcd } a b) \wedge^n) (\text{gcd } a b \wedge^n * (b \text{ div gcd } a b) \wedge^n)$
by (*rule gcd-mult-left [symmetric]*)
also have $(\text{gcd } a b) \wedge^n * (a \text{ div gcd } a b) \wedge^n = a \wedge^n$
by (*simp add: ac-simps div-power dvd-power-same*)
also have $(\text{gcd } a b) \wedge^n * (b \text{ div gcd } a b) \wedge^n = b \wedge^n$
by (*simp add: ac-simps div-power dvd-power-same*)
finally show *?thesis by simp*
qed

lemma *division-decomp:*
assumes *a dvd b * c*
shows $\exists b' c'. a = b' * c' \wedge b' \text{ dvd } b \wedge c' \text{ dvd } c$
proof (*cases gcd a b = 0*)
case *True*
then have $a = 0 \wedge b = 0$
by *simp*
then have $a = 0 * c \wedge 0 \text{ dvd } b \wedge c \text{ dvd } c$
by *simp*
then show *?thesis by blast*

```

next
  case False
  let  $?d = \text{gcd } a \ b$ 
  from gcd-coprime-exists [OF False]
    obtain  $a' \ b'$  where  $ab': a = a' * ?d \ b = b' * ?d \ \text{coprime } a' \ b'$ 
    by blast
  from  $ab'(1)$  have  $a' \ \text{dvd } a \ ..$ 
  with assms have  $a' \ \text{dvd } b * c$ 
    using dvd-trans [of a' a b * c] by simp
  from assms  $ab'(1,2)$  have  $a' * ?d \ \text{dvd } (b' * ?d) * c$ 
    by simp
  then have  $?d * a' \ \text{dvd } ?d * (b' * c)$ 
    by (simp add: mult-ac)
  with  $\langle ?d \neq 0 \rangle$  have  $a' \ \text{dvd } b' * c$ 
    by simp
  then have  $a' \ \text{dvd } c$ 
    using  $\langle \text{coprime } a' \ b' \rangle$  by (simp add: coprime-dvd-mult-right-iff)
  with  $ab'(1)$  have  $a = ?d * a' \wedge ?d \ \text{dvd } b \wedge a' \ \text{dvd } c$ 
    by (simp add: ac-simps)
  then show ?thesis by blast
qed

```

```

lemma lcm-coprime:  $\text{coprime } a \ b \implies \text{lcm } a \ b = \text{normalize } (a * b)$ 
  by (subst lcm-gcd) simp

```

end

```

context ring-gcd
begin

```

```

lemma coprime-minus-left-iff [simp]:
   $\text{coprime } (- a) \ b \longleftrightarrow \text{coprime } a \ b$ 
  by (rule; rule coprimeI) (auto intro: coprime-common-divisor)

```

```

lemma coprime-minus-right-iff [simp]:
   $\text{coprime } a \ (- b) \longleftrightarrow \text{coprime } a \ b$ 
  using coprime-minus-left-iff [of b a] by (simp add: ac-simps)

```

```

lemma coprime-diff-one-left [simp]:  $\text{coprime } (a - 1) \ a$ 
  using coprime-add-one-right [of a - 1] by simp

```

```

lemma coprime-doff-one-right [simp]:  $\text{coprime } a \ (a - 1)$ 
  using coprime-diff-one-left [of a] by (simp add: ac-simps)

```

end

```

context semiring-Gcd
begin

```

lemma *Lcm-coprime*:
assumes *finite A*
and $A \neq \{\}$
and $\bigwedge a b. a \in A \implies b \in A \implies a \neq b \implies \text{coprime } a \ b$
shows $\text{Lcm } A = \text{normalize } (\prod A)$
using *assms*
proof (*induct rule: finite-ne-induct*)
case *singleton*
then show *?case* **by** *simp*
next
case (*insert a A*)
have $\text{Lcm } (\text{insert } a \ A) = \text{lcm } a \ (\text{Lcm } A)$
by *simp*
also from *insert* **have** $\text{Lcm } A = \text{normalize } (\prod A)$
by *blast*
also have $\text{lcm } a \ \dots = \text{lcm } a \ (\prod A)$
by (*cases* $\prod A = 0$) (*simp-all add: lcm-div-unit2*)
also from *insert* **have** $\text{coprime } a \ (\prod A)$
by (*subst coprime-commute, intro prod-coprime-left*) *auto*
with *insert* **have** $\text{lcm } a \ (\prod A) = \text{normalize } (\prod (\text{insert } a \ A))$
by (*simp add: lcm-coprime*)
finally show *?case* .
qed

lemma *Lcm-coprime'*:
 $\text{card } A \neq 0 \implies$
 $(\bigwedge a b. a \in A \implies b \in A \implies a \neq b \implies \text{coprime } a \ b) \implies$
 $\text{Lcm } A = \text{normalize } (\prod A)$
by (*rule Lcm-coprime*) (*simp-all add: card-eq-0-iff*)

end

And some consequences: cancellation modulo m

lemma *mult-mod-cancel-right*:
fixes $m :: 'a::\{\text{euclidean-ring-cancel, semiring-gcd}\}$
assumes *eq: (a * n) mod m = (b * n) mod m* **and** *coprime m n*
shows $a \ \text{mod } m = b \ \text{mod } m$
proof –
have $m \ \text{dvd } (a*n - b*n)$
using *eq mod-eq-dvd-iff* **by** *blast*
then have $m \ \text{dvd } a-b$
by (*metis <coprime m n> coprime-dvd-mult-left-iff left-diff-distrib'*)
then show *?thesis*
using *mod-eq-dvd-iff* **by** *blast*
qed

lemma *mult-mod-cancel-left*:
fixes $m :: 'a::\{\text{euclidean-ring-cancel, semiring-gcd}\}$
assumes $(n * a) \ \text{mod } m = (n * b) \ \text{mod } m$ **and** *coprime m n*

shows $a \bmod m = b \bmod m$
 by (*metis* *assms* *mult.commute* *mult-mod-cancel-right*)

88.5 GCD and LCM for multiplicative normalisation functions

class *semiring-gcd-mult-normalize* = *semiring-gcd* + *normalization-semidom-multiplicative*
 begin

lemma *mult-gcd-left*: $c * \text{gcd } a \ b = \text{unit-factor } c * \text{gcd } (c * a) \ (c * b)$
 by (*simp* *add*: *gcd-mult-left* *normalize-mult* *mult.assoc* [*symmetric*])

lemma *mult-gcd-right*: $\text{gcd } a \ b * c = \text{gcd } (a * c) \ (b * c) * \text{unit-factor } c$
 using *mult-gcd-left* [of *c a b*] by (*simp* *add*: *ac-simps*)

lemma *gcd-mult-distrib'*: $\text{normalize } c * \text{gcd } a \ b = \text{gcd } (c * a) \ (c * b)$
 by (*subst* *gcd-mult-left*) (*simp-all* *add*: *normalize-mult*)

lemma *gcd-mult-distrib*: $k * \text{gcd } a \ b = \text{gcd } (k * a) \ (k * b) * \text{unit-factor } k$
 proof –

have $\text{normalize } k * \text{gcd } a \ b = \text{gcd } (k * a) \ (k * b)$
 by (*simp* *add*: *gcd-mult-distrib'*)

then have $\text{normalize } k * \text{gcd } a \ b * \text{unit-factor } k = \text{gcd } (k * a) \ (k * b) * \text{unit-factor } k$

by *simp*

then have $\text{normalize } k * \text{unit-factor } k * \text{gcd } a \ b = \text{gcd } (k * a) \ (k * b) * \text{unit-factor } k$

by (*simp* *only*: *ac-simps*)

then show *?thesis*

by *simp*

qed

lemma *gcd-mult-lcm* [*simp*]: $\text{gcd } a \ b * \text{lcm } a \ b = \text{normalize } a * \text{normalize } b$
 by (*simp* *add*: *lcm-gcd* *normalize-mult* *dvd-normalize-div*)

lemma *lcm-mult-gcd* [*simp*]: $\text{lcm } a \ b * \text{gcd } a \ b = \text{normalize } a * \text{normalize } b$
 using *gcd-mult-lcm* [of *a b*] by (*simp* *add*: *ac-simps*)

lemma *mult-lcm-left*: $c * \text{lcm } a \ b = \text{unit-factor } c * \text{lcm } (c * a) \ (c * b)$
 by (*simp* *add*: *lcm-mult-left* *mult.assoc* [*symmetric*] *normalize-mult*)

lemma *mult-lcm-right*: $\text{lcm } a \ b * c = \text{lcm } (a * c) \ (b * c) * \text{unit-factor } c$
 using *mult-lcm-left* [of *c a b*] by (*simp* *add*: *ac-simps*)

lemma *lcm-gcd-prod*: $\text{lcm } a \ b * \text{gcd } a \ b = \text{normalize } (a * b)$
 by (*simp* *add*: *lcm-gcd* *dvd-normalize-div* *normalize-mult*)

lemma *lcm-mult-distrib'*: $\text{normalize } c * \text{lcm } a \ b = \text{lcm } (c * a) \ (c * b)$
 by (*subst* *lcm-mult-left*) (*simp* *add*: *normalize-mult*)

lemma *lcm-mult-distrib*: $k * lcm\ a\ b = lcm\ (k * a)\ (k * b) * unit-factor\ k$

proof –

have *normalize* $k * lcm\ a\ b = lcm\ (k * a)\ (k * b)$

by (*simp add: lcm-mult-distrib'*)

then have *normalize* $k * lcm\ a\ b * unit-factor\ k = lcm\ (k * a)\ (k * b) * unit-factor\ k$

by *simp*

then have *normalize* $k * unit-factor\ k * lcm\ a\ b = lcm\ (k * a)\ (k * b) * unit-factor\ k$

by (*simp only: ac-simps*)

then show *?thesis*

by *simp*

qed

lemma *coprime-crossproduct'*:

fixes $a\ b\ c\ d$

assumes $b \neq 0$

assumes *unit-factors*: $unit-factor\ b = unit-factor\ d$

assumes *coprime*: $coprime\ a\ b\ coprime\ c\ d$

shows $a * d = b * c \longleftrightarrow a = c \wedge b = d$

proof *safe*

assume *eq*: $a * d = b * c$

hence *normalize* $a * normalize\ d = normalize\ c * normalize\ b$

by (*simp only: normalize-mult [symmetric] mult-ac*)

with *coprime* **have** *normalize* $b = normalize\ d$

by (*subst (asm) coprime-crossproduct*) *simp-all*

from *this* **and** *unit-factors* **show** $b = d$

by (*rule normalize-unit-factor-eqI*)

from *eq* **have** $a * d = c * d$ **by** (*simp only: <b = d> mult-ac*)

with $<b \neq 0>$ $<b = d>$ **show** $a = c$ **by** *simp*

qed (*simp-all add: mult-ac*)

lemma *gcd-exp [simp]*:

$gcd\ (a \wedge n)\ (b \wedge n) = gcd\ a\ b \wedge n$

using *gcd-exp-weak[of a n b]* **by** (*simp add: normalize-power*)

end

88.6 GCD and LCM on *nat* and *int*

instantiation *nat* :: *gcd*

begin

fun *gcd-nat* :: $nat \Rightarrow nat \Rightarrow nat$

where *gcd-nat* $x\ y = (if\ y = 0\ then\ x\ else\ gcd\ y\ (x\ mod\ y))$

definition *lcm-nat* :: $nat \Rightarrow nat \Rightarrow nat$

where *lcm-nat* $x\ y = x * y\ div\ (gcd\ x\ y)$

```

instance ..

end

instantiation int :: gcd
begin

definition gcd-int :: int ⇒ int ⇒ int
  where gcd-int x y = int (gcd (nat |x|) (nat |y|))

definition lcm-int :: int ⇒ int ⇒ int
  where lcm-int x y = int (lcm (nat |x|) (nat |y|))

instance ..

end

lemma gcd-int-int-eq [simp]:
  gcd (int m) (int n) = int (gcd m n)
  by (simp add: gcd-int-def)

lemma gcd-nat-abs-left-eq [simp]:
  gcd (nat |k|) n = nat (gcd k (int n))
  by (simp add: gcd-int-def)

lemma gcd-nat-abs-right-eq [simp]:
  gcd n (nat |k|) = nat (gcd (int n) k)
  by (simp add: gcd-int-def)

lemma abs-gcd-int [simp]:
  |gcd x y| = gcd x y
  for x y :: int
  by (simp only: gcd-int-def)

lemma gcd-abs1-int [simp]:
  gcd |x| y = gcd x y
  for x y :: int
  by (simp only: gcd-int-def) simp

lemma gcd-abs2-int [simp]:
  gcd x |y| = gcd x y
  for x y :: int
  by (simp only: gcd-int-def) simp

lemma lcm-int-int-eq [simp]:
  lcm (int m) (int n) = int (lcm m n)
  by (simp add: lcm-int-def)

```

lemma *lcm-nat-abs-left-eq* [*simp*]:
 $lcm (nat |k|) n = nat (lcm k (int n))$
by (*simp add: lcm-int-def*)

lemma *lcm-nat-abs-right-eq* [*simp*]:
 $lcm n (nat |k|) = nat (lcm (int n) k)$
by (*simp add: lcm-int-def*)

lemma *lcm-abs1-int* [*simp*]:
 $lcm |x| y = lcm x y$
for $x y :: int$
by (*simp only: lcm-int-def*) *simp*

lemma *lcm-abs2-int* [*simp*]:
 $lcm x |y| = lcm x y$
for $x y :: int$
by (*simp only: lcm-int-def*) *simp*

lemma *abs-lcm-int* [*simp*]: $|lcm i j| = lcm i j$
for $i j :: int$
by (*simp only: lcm-int-def*)

lemma *gcd-nat-induct* [*case-names base step*]:
fixes $m n :: nat$
assumes $\bigwedge m. P m 0$
and $\bigwedge m n. 0 < n \implies P n (m \bmod n) \implies P m n$
shows $P m n$
proof (*induction m n rule: gcd-nat.induct*)
case (*1 x y*)
then show *?case*
using *assms neq0-conv* **by** *blast*
qed

lemma *gcd-neg1-int* [*simp*]: $gcd (- x) y = gcd x y$
for $x y :: int$
by (*simp only: gcd-int-def*) *simp*

lemma *gcd-neg2-int* [*simp*]: $gcd x (- y) = gcd x y$
for $x y :: int$
by (*simp only: gcd-int-def*) *simp*

lemma *gcd-cases-int*:
fixes $x y :: int$
assumes $x \geq 0 \implies y \geq 0 \implies P (gcd x y)$
and $x \geq 0 \implies y \leq 0 \implies P (gcd x (- y))$
and $x \leq 0 \implies y \geq 0 \implies P (gcd (- x) y)$
and $x \leq 0 \implies y \leq 0 \implies P (gcd (- x) (- y))$
shows $P (gcd x y)$
using *assms* **by** *auto arith*

```

lemma gcd-ge-0-int [simp]:  $\text{gcd } (x::\text{int}) \ y \geq 0$ 
  for  $x \ y :: \text{int}$ 
  by (simp add: gcd-int-def)

lemma lcm-neg1-int:  $\text{lcm } (- \ x) \ y = \text{lcm } x \ y$ 
  for  $x \ y :: \text{int}$ 
  by (simp only: lcm-int-def) simp

lemma lcm-neg2-int:  $\text{lcm } x \ (- \ y) = \text{lcm } x \ y$ 
  for  $x \ y :: \text{int}$ 
  by (simp only: lcm-int-def) simp

lemma lcm-cases-int:
  fixes  $x \ y :: \text{int}$ 
  assumes  $x \geq 0 \implies y \geq 0 \implies P (\text{lcm } x \ y)$ 
    and  $x \geq 0 \implies y \leq 0 \implies P (\text{lcm } x \ (- \ y))$ 
    and  $x \leq 0 \implies y \geq 0 \implies P (\text{lcm } (- \ x) \ y)$ 
    and  $x \leq 0 \implies y \leq 0 \implies P (\text{lcm } (- \ x) \ (- \ y))$ 
  shows  $P (\text{lcm } x \ y)$ 
  using assms by (auto simp: lcm-neg1-int lcm-neg2-int) arith

lemma lcm-ge-0-int [simp]:  $\text{lcm } x \ y \geq 0$ 
  for  $x \ y :: \text{int}$ 
  by (simp only: lcm-int-def)

lemma gcd-0-nat:  $\text{gcd } x \ 0 = x$ 
  for  $x :: \text{nat}$ 
  by simp

lemma gcd-0-int [simp]:  $\text{gcd } x \ 0 = |x|$ 
  for  $x :: \text{int}$ 
  by (auto simp: gcd-int-def)

lemma gcd-0-left-nat:  $\text{gcd } 0 \ x = x$ 
  for  $x :: \text{nat}$ 
  by simp

lemma gcd-0-left-int [simp]:  $\text{gcd } 0 \ x = |x|$ 
  for  $x :: \text{int}$ 
  by (auto simp: gcd-int-def)

lemma gcd-red-nat:  $\text{gcd } x \ y = \text{gcd } y \ (x \bmod y)$ 
  for  $x \ y :: \text{nat}$ 
  by (cases y = 0) auto

```

Weaker, but useful for the simplifier.

```

lemma gcd-non-0-nat:  $y \neq 0 \implies \text{gcd } x \ y = \text{gcd } y \ (x \bmod y)$ 
  for  $x \ y :: \text{nat}$ 

```

by *simp*

lemma *gcd-1-nat* [*simp*]: $\text{gcd } m \ 1 = 1$
 for $m :: \text{nat}$
 by *simp*

lemma *gcd-Suc-0* [*simp*]: $\text{gcd } m \ (\text{Suc } 0) = \text{Suc } 0$
 for $m :: \text{nat}$
 by *simp*

lemma *gcd-1-int* [*simp*]: $\text{gcd } m \ 1 = 1$
 for $m :: \text{int}$
 by (*simp add: gcd-int-def*)

lemma *gcd-idem-nat*: $\text{gcd } x \ x = x$
 for $x :: \text{nat}$
 by *simp*

lemma *gcd-idem-int*: $\text{gcd } x \ x = |x|$
 for $x :: \text{int}$
 by (*auto simp: gcd-int-def*)

declare *gcd-nat.simps* [*simp del*]

gcd m n divides m and n . The conjunctions don't seem provable separately.

instance *nat :: semiring-gcd*

proof

fix $m \ n :: \text{nat}$

show *gcd m n dvd m* and *gcd m n dvd n*

proof (*induct m n rule: gcd-nat-induct*)

case (*step m n*)

then have *gcd n (m mod n) dvd m*

by (*metis dvd-mod-imp-dvd*)

with *step* show *gcd m n dvd m*

by (*simp add: gcd-non-0-nat*)

qed (*simp-all add: gcd-0-nat gcd-non-0-nat*)

next

fix $m \ n \ k :: \text{nat}$

assume *k dvd m* and *k dvd n*

then show *k dvd gcd m n*

by (*induct m n rule: gcd-nat-induct*) (*simp-all add: gcd-non-0-nat dvd-mod gcd-0-nat*)

qed (*simp-all add: lcm-nat-def*)

instance *int :: ring-gcd*

proof

fix $k \ l \ r :: \text{int}$

show [*simp*]: *gcd k l dvd k* *gcd k l dvd l*

using *gcd-dvd1* [*of nat |k| nat |l|*]

```

    gcd-dvd2 [of nat |k| nat |l|]
  by simp-all
show lcm k l = normalize (k * l div gcd k l)
  using lcm-gcd [of nat |k| nat |l|]
  by (simp add: nat-eq-iff of-nat-div abs-mult abs-div)
assume r dvd k r dvd l
then show r dvd gcd k l
  using gcd-greatest [of nat |r| nat |k| nat |l|]
  by simp
qed simp

lemma gcd-le1-nat [simp]: a ≠ 0 ⇒ gcd a b ≤ a
  for a b :: nat
  by (rule dvd-imp-le) auto

lemma gcd-le2-nat [simp]: b ≠ 0 ⇒ gcd a b ≤ b
  for a b :: nat
  by (rule dvd-imp-le) auto

lemma gcd-le1-int [simp]: a > 0 ⇒ gcd a b ≤ a
  for a b :: int
  by (rule zdvd-imp-le) auto

lemma gcd-le2-int [simp]: b > 0 ⇒ gcd a b ≤ b
  for a b :: int
  by (rule zdvd-imp-le) auto

lemma gcd-pos-nat [simp]: gcd m n > 0 ⟷ m ≠ 0 ∨ n ≠ 0
  for m n :: nat
  using gcd-eq-0-iff [of m n] by arith

lemma gcd-pos-int [simp]: gcd m n > 0 ⟷ m ≠ 0 ∨ n ≠ 0
  for m n :: int
  using gcd-eq-0-iff [of m n] gcd-ge-0-int [of m n] by arith

lemma gcd-unique-nat: d dvd a ∧ d dvd b ∧ (∀ e. e dvd a ∧ e dvd b ⟶ e dvd d)
⟷ d = gcd a b
  for d a :: nat
  using gcd-unique by fastforce

lemma gcd-unique-int:
  d ≥ 0 ∧ d dvd a ∧ d dvd b ∧ (∀ e. e dvd a ∧ e dvd b ⟶ e dvd d) ⟷ d = gcd
a b
  for d a :: int
  using zdvd-antisym-nonneg by auto

interpretation gcd-nat:
  semilattice-neutr-order gcd 0::nat Rings.dvd λm n. m dvd n ∧ m ≠ n
  by standard (auto simp: gcd-unique-nat [symmetric] intro: dvd-antisym dvd-trans)

```

lemma *gcd-proj1-if-dvd-int* [*simp*]: $x \text{ dvd } y \implies \text{gcd } x \ y = |x|$
for $x \ y :: \text{int}$
by (*metis abs-dvd-iff gcd-0-left-int gcd-unique-int*)

lemma *gcd-proj2-if-dvd-int* [*simp*]: $y \text{ dvd } x \implies \text{gcd } x \ y = |y|$
for $x \ y :: \text{int}$
by (*metis gcd-proj1-if-dvd-int gcd.commute*)

Multiplication laws.

lemma *gcd-mult-distrib-nat*: $k * \text{gcd } m \ n = \text{gcd } (k * m) \ (k * n)$
for $k \ m \ n :: \text{nat}$
— [1, page 27]
by (*simp add: gcd-mult-left*)

lemma *gcd-mult-distrib-int*: $|k| * \text{gcd } m \ n = \text{gcd } (k * m) \ (k * n)$
for $k \ m \ n :: \text{int}$
by (*simp add: gcd-mult-left abs-mult*)

Addition laws.

lemma *gcd-diff1-nat*: $m \geq n \implies \text{gcd } (m - n) \ n = \text{gcd } m \ n$
for $m \ n :: \text{nat}$
by (*subst gcd-add1 [symmetric]*) *auto*

lemma *gcd-diff2-nat*: $n \geq m \implies \text{gcd } (n - m) \ n = \text{gcd } m \ n$
for $m \ n :: \text{nat}$
by (*metis gcd.commute gcd-add2 gcd-diff1-nat le-add-diff-inverse2*)

lemma *gcd-non-0-int*:
fixes $x \ y :: \text{int}$
assumes $y > 0$ **shows** $\text{gcd } x \ y = \text{gcd } y \ (x \text{ mod } y)$
proof (*cases x mod y = 0*)
case *False*
then have *neg*: $x \text{ mod } y = y - (-x) \text{ mod } y$
by (*simp add: zmod-zminus1-eq-if*)
have *xy*: $0 \leq x \text{ mod } y$
by (*simp add: assms*)
show *?thesis*
proof (*cases x < 0*)
case *True*
have $\text{nat } (-x \text{ mod } y) \leq \text{nat } y$
by (*simp add: assms dual-order.order-iff-strict*)
moreover have $\text{gcd } (\text{nat } (-x)) \ (\text{nat } y) = \text{gcd } (\text{nat } (-x \text{ mod } y)) \ (\text{nat } y)$
using *True assms gcd-non-0-nat nat-mod-distrib* **by** *auto*
ultimately have $\text{gcd } (\text{nat } (-x)) \ (\text{nat } y) = \text{gcd } (\text{nat } y) \ (\text{nat } (x \text{ mod } y))$
using *assms*
by (*simp add: neg nat-diff-distrib'*) (*metis gcd.commute gcd-diff2-nat*)
with *assms <0 ≤ x mod y* **show** *?thesis*

```

    by (simp add: True dual-order.order-iff-strict gcd-int-def)
  next
  case False
  with assms xy have gcd (nat x) (nat y) = gcd (nat y) (nat x mod nat y)
    using gcd-red-nat by blast
  with False assms show ?thesis
    by (simp add: gcd-int-def nat-mod-distrib)
  qed
qed (use assms in auto)

```

```

lemma gcd-red-int: gcd x y = gcd y (x mod y)
  for x y :: int
proof (cases y 0::int rule: linorder-cases)
  case less
  with gcd-non-0-int [of - y - x] show ?thesis
    by auto
  next
  case greater
  with gcd-non-0-int [of y x] show ?thesis
    by auto
qed auto

```

```

lemma finite-divisors-nat [simp]:
  fixes m :: nat
  assumes m > 0
  shows finite {d. d dvd m}
proof -
  from assms have {d. d dvd m}  $\subseteq$  {d. d  $\leq$  m}
    by (auto dest: dvd-imp-le)
  then show ?thesis
    using finite-Collect-le-nat by (rule finite-subset)
qed

```

```

lemma finite-divisors-int [simp]:
  fixes i :: int
  assumes i  $\neq$  0
  shows finite {d. d dvd i}
proof -
  have {d. |d|  $\leq$  |i|} = {- |i|..|i|}
    by (auto simp: abs-if)
  then have finite {d. |d|  $\leq$  |i|}
    by simp
  from finite-subset [OF - this] show ?thesis
    using assms by (simp add: dvd-imp-le-int subset-iff)
qed

```


lemma *Max-divisors-self-nat* [*simp*]: $n \neq 0 \implies \text{Max } \{d::\text{nat}. d \text{ dvd } n\} = n$
by (*fastforce* *intro: antisym Max-le-iff*[*THEN iffD2*] *simp: dvd-imp-le*)

lemma *Max-divisors-self-int* [*simp*]:
assumes $n \neq 0$ **shows** $\text{Max } \{d::\text{int}. d \text{ dvd } n\} = |n|$
proof (*rule antisym*)
show $\text{Max } \{d. d \text{ dvd } n\} \leq |n|$
using *assms* **by** (*auto* *intro: abs-le-D1 dvd-imp-le-int intro!: Max-le-iff* [*THEN iffD2*])
qed (*simp add: assms*)

lemma *gcd-is-Max-divisors-nat*:
fixes $m n :: \text{nat}$
assumes $n > 0$ **shows** $\text{gcd } m n = \text{Max } \{d. d \text{ dvd } m \wedge d \text{ dvd } n\}$
proof (*rule Max-eqI*[*THEN sym*], *simp-all*)
show *finite* $\{d. d \text{ dvd } m \wedge d \text{ dvd } n\}$
by (*simp add: <n > 0*)
show $\bigwedge y. y \text{ dvd } m \wedge y \text{ dvd } n \implies y \leq \text{gcd } m n$
by (*simp add: <n > 0 dvd-imp-le*)
qed

lemma *gcd-is-Max-divisors-int*:
fixes $m n :: \text{int}$
assumes $n \neq 0$ **shows** $\text{gcd } m n = \text{Max } \{d. d \text{ dvd } m \wedge d \text{ dvd } n\}$
proof (*rule Max-eqI*[*THEN sym*], *simp-all*)
show *finite* $\{d. d \text{ dvd } m \wedge d \text{ dvd } n\}$
by (*simp add: <n ≠ 0*)
show $\bigwedge y. y \text{ dvd } m \wedge y \text{ dvd } n \implies y \leq \text{gcd } m n$
by (*simp add: <n ≠ 0 zdvd-imp-le*)
qed

lemma *gcd-code-int* [*code*]: $\text{gcd } k l = \text{if } l = 0 \text{ then } k \text{ else } \text{gcd } l (|k| \bmod |l|)$
for $k l :: \text{int}$
using *gcd-red-int* [*of |k| |l|*] **by** *simp*

lemma *coprime-Suc-left-nat* [*simp*]:
coprime (*Suc* n) n
using *coprime-add-one-left* [*of n*] **by** *simp*

lemma *coprime-Suc-right-nat* [*simp*]:
coprime n (*Suc* n)
using *coprime-Suc-left-nat* [*of n*] **by** (*simp add: ac-simps*)

lemma *coprime-diff-one-left-nat* [*simp*]:
coprime $(n - 1)$ n **if** $n > 0$ **for** $n :: \text{nat}$
using *that coprime-Suc-right-nat* [*of n - 1*] **by** *simp*

lemma *coprime-diff-one-right-nat* [*simp*]:

coprime $n (n - 1)$ **if** $n > 0$ **for** $n :: \text{nat}$
using *that coprime-diff-one-left-nat* [of n] **by** (*simp add: ac-simps*)

lemma *coprime-crossproduct-nat*:
fixes $a b c d :: \text{nat}$
assumes *coprime* $a d$ **and** *coprime* $b c$
shows $a * c = b * d \longleftrightarrow a = b \wedge c = d$
using *assms coprime-crossproduct* [of $a d b c$] **by** *simp*

lemma *coprime-crossproduct-int*:
fixes $a b c d :: \text{int}$
assumes *coprime* $a d$ **and** *coprime* $b c$
shows $|a| * |c| = |b| * |d| \longleftrightarrow |a| = |b| \wedge |c| = |d|$
using *assms coprime-crossproduct* [of $a d b c$] **by** *simp*

88.7 Bezout’s theorem

Function *bezw* returns a pair of witnesses to Bezout’s theorem – see the theorems that follow the definition.

fun *bezw* $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{int} * \text{int}$
where *bezw* $x y =$
 (*if* $y = 0$ *then* $(1, 0)$
 else
 (*snd* (*bezw* $y (x \bmod y)$)),
 (*fst* (*bezw* $y (x \bmod y)$) – *snd* (*bezw* $y (x \bmod y)$) * *int*($x \text{ div } y$)))

lemma *bezw-0* [*simp*]: *bezw* $x 0 = (1, 0)$
by *simp*

lemma *bezw-non-0*:
 $y > 0 \implies \text{bezw } x y =$
 (*snd* (*bezw* $y (x \bmod y)$)), *fst* (*bezw* $y (x \bmod y)$) – *snd* (*bezw* $y (x \bmod y)$) *
int($x \text{ div } y$)
by *simp*

declare *bezw.simps* [*simp del*]

lemma *bezw-aux*: $\text{int } (\text{gcd } x y) = \text{fst } (\text{bezw } x y) * \text{int } x + \text{snd } (\text{bezw } x y) * \text{int } y$
proof (*induct* $x y$ *rule: gcd-nat-induct*)
case (*step* $m n$)
then have $\text{fst } (\text{bezw } m n) * \text{int } m + \text{snd } (\text{bezw } m n) * \text{int } n - \text{int } (\text{gcd } m n)$
 $= \text{int } m * \text{snd } (\text{bezw } n (m \bmod n)) -$
 $(\text{int } (m \bmod n) * \text{snd } (\text{bezw } n (m \bmod n)) + \text{int } n * (\text{int } (m \text{ div } n) * \text{snd } (\text{bezw } n (m \bmod n))))$
by (*simp add: bezw-non-0 gcd-non-0-nat field-simps*)
also have $\dots = \text{int } m * \text{snd } (\text{bezw } n (m \bmod n)) - (\text{int } (m \bmod n) + \text{int } (n * (m \text{ div } n))) * \text{snd } (\text{bezw } n (m \bmod n))$
by (*simp add: distrib-right*)

```

also have ... = 0
  by (metis cancel-comm-monoid-add-class.diff-cancel mod-mult-div-eq of-nat-add)
finally show ?case
  by simp
qed auto

```

```

lemma bezout-int:  $\exists u v. u * x + v * y = \text{gcd } x y$ 
  for  $x y :: \text{int}$ 
proof -
  have aux:  $x \geq 0 \implies y \geq 0 \implies \exists u v. u * x + v * y = \text{gcd } x y$  for  $x y :: \text{int}$ 
    apply (rule-tac  $x = \text{fst } (\text{bezw } (\text{nat } x) (\text{nat } y))$ ) in exI
    apply (rule-tac  $x = \text{snd } (\text{bezw } (\text{nat } x) (\text{nat } y))$ ) in exI
    by (simp add: bezw-aux gcd-int-def)
  consider  $x \geq 0 y \geq 0 \mid x \geq 0 y \leq 0 \mid x \leq 0 y \geq 0 \mid x \leq 0 y \leq 0$ 
    using linear by blast
  then show ?thesis
  proof cases
    case 1
    then show ?thesis by (rule aux)
  next
    case 2
    then show ?thesis
      using aux [of  $x - y$ ]
      by (metis gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le)
  next
    case 3
    then show ?thesis
      using aux [of  $-x y$ ]
      by (metis gcd.commute gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le)
  next
    case 4
    then show ?thesis
      using aux [of  $-x -y$ ]
      by (metis diff-0 diff-ge-0-iff-ge gcd-neg1-int gcd-neg2-int mult.commute mult-minus-right)
  qed
qed

```

Versions of Bezout for *nat*, by Amine Chaieb.

```

lemma Euclid-induct [case-names swap zero add]:
  fixes  $P :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ 
  assumes  $c: \bigwedge a b. P a b \longleftrightarrow P b a$ 
    and  $z: \bigwedge a. P a 0$ 
    and  $\text{add}: \bigwedge a b. P a b \longrightarrow P a (a + b)$ 
  shows  $P a b$ 
proof (induct  $a + b$  arbitrary:  $a b$  rule: less-induct)
  case less
  consider (eq)  $a = b \mid$  (lt)  $a < b \mid$  (gt)  $a > b$ 
    by arith

```

```

show ?case
proof (cases a b rule: linorder-cases)
  case equal
  with add [rule-format, OF z [rule-format, of a]] show ?thesis by simp
next
  case lt: less
  then consider a = 0 | a + b - a < a + b by arith
  then show ?thesis
  proof cases
    case 1
    with z c show ?thesis by blast
  next
    case 2
    also have *: a + b - a = a + (b - a) using lt by arith
    finally have a + (b - a) < a + b .
    then have P a (a + (b - a)) by (rule add [rule-format, OF less])
    then show ?thesis by (simp add: *[symmetric])
  qed
next
  case gt: greater
  then consider b = 0 | b + a - b < a + b by arith
  then show ?thesis
  proof cases
    case 1
    with z c show ?thesis by blast
  next
    case 2
    also have *: b + a - b = b + (a - b) using gt by arith
    finally have b + (a - b) < a + b .
    then have P b (b + (a - b)) by (rule add [rule-format, OF less])
    then have P b a by (simp add: *[symmetric])
    with c show ?thesis by blast
  qed
qed
qed
qed

lemma bezout-lemma-nat:
  fixes d::nat
  shows  $\llbracket d \text{ dvd } a; d \text{ dvd } b; a * x = b * y + d \vee b * x = a * y + d \rrbracket$ 
   $\implies \exists x y. d \text{ dvd } a \wedge d \text{ dvd } a + b \wedge (a * x = (a + b) * y + d \vee (a + b) * x =$ 
 $a * y + d)$ 
  apply auto
  apply (metis add-mult-distrib2 left-add-mult-distrib)
  apply (rule-tac x=x in exI)
  by (metis add-mult-distrib2 mult.commute add.assoc)

lemma bezout-add-nat:
   $\exists (d::nat) x y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge (a * x = b * y + d \vee b * x = a * y + d)$ 
proof (induct a b rule: Euclid-induct)

```

```

    case (swap a b)
    then show ?case
      by blast
  next
    case (zero a)
    then show ?case
      by fastforce
  next
    case (add a b)
    then show ?case
      by (meson bezout-lemma-nat)
qed

```

lemma bezout1-nat: $\exists (d::nat) x y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge (a * x - b * y = d \vee b * x - a * y = d)$
 using bezout-add-nat[of a b] by (metis add-diff-cancel-left')

lemma bezout-add-strong-nat:

```

  fixes a b :: nat
  assumes a: a ≠ 0
  shows  $\exists d x y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge a * x = b * y + d$ 
  proof -
    consider d x y where d dvd a d dvd b a * x = b * y + d
      | d x y where d dvd a d dvd b b * x = a * y + d
      using bezout-add-nat [of a b] by blast
    then show ?thesis
  proof cases
    case 1
    then show ?thesis by blast
  next
    case H: 2
    show ?thesis
    proof (cases b = 0)
      case True
      with H show ?thesis by simp
    next
      case False
      then have bp: b > 0 by simp
      with dvd-imp-le [OF H(2)] consider d = b | d < b
        by atomize-elim auto
      then show ?thesis
    proof cases
      case 1
      with a H show ?thesis
        by (metis Suc-pred add commute mult commute mult-Suc-right neq0-conv)
    next
      case 2
      show ?thesis
      proof (cases x = 0)

```

```

    case True
    with a H show ?thesis by simp
  next
  case x0: False
  then have xp: x > 0 by simp
  from ⟨d < b⟩ have d ≤ b - 1 by simp
  then have d * b ≤ b * (b - 1) by simp
  with xp mult-mono[of 1 x d * b b * (b - 1)]
  have dble: d * b ≤ x * b * (b - 1) using bp by simp
  from H(3) have d + (b - 1) * (b * x) = d + (b - 1) * (a * y + d)
    by simp
  then have d + (b - 1) * a * y + (b - 1) * d = d + (b - 1) * b * x
    by (simp only: mult.assoc distrib-left)
  then have a * ((b - 1) * y) + d * (b - 1 + 1) = d + x * b * (b - 1)
    by algebra
  then have a * ((b - 1) * y) = d + x * b * (b - 1) - d * b
    using bp by simp
  then have a * ((b - 1) * y) = d + (x * b * (b - 1) - d * b)
    by (simp only: diff-add-assoc[OF dble, of d, symmetric])
  then have a * ((b - 1) * y) = b * (x * (b - 1) - d) + d
    by (simp only: diff-mult-distrib2 ac-simps)
  with H(1,2) show ?thesis
    by blast
  qed
  qed
  qed
  qed
  qed

lemma bezout-nat:
  fixes a :: nat
  assumes a: a ≠ 0
  shows ∃ x y. a * x = b * y + gcd a b
proof -
  obtain d x y where d: d dvd a d dvd b and eq: a * x = b * y + d
    using bezout-add-strong-nat [OF a, of b] by blast
  from d have d dvd gcd a b
    by simp
  then obtain k where k: gcd a b = d * k
    unfolding dvd-def by blast
  from eq have a * x * k = (b * y + d) * k
    by auto
  then have a * (x * k) = b * (y * k) + gcd a b
    by (algebra add: k)
  then show ?thesis
    by blast
qed

```

88.8 LCM properties on nat and int

lemma *lcm-altdef-int* [code]: $\text{lcm } a \ b = |a| * |b| \ \text{div} \ \text{gcd } a \ b$
for $a \ b :: \text{int}$
by (*simp add: abs-mult lcm-gcd abs-div*)

lemma *prod-gcd-lcm-nat*: $m * n = \text{gcd } m \ n * \text{lcm } m \ n$
for $m \ n :: \text{nat}$
by (*simp add: lcm-gcd*)

lemma *prod-gcd-lcm-int*: $|m| * |n| = \text{gcd } m \ n * \text{lcm } m \ n$
for $m \ n :: \text{int}$
by (*simp add: lcm-gcd abs-div abs-mult*)

lemma *lcm-pos-nat*: $m > 0 \implies n > 0 \implies \text{lcm } m \ n > 0$
for $m \ n :: \text{nat}$
using *lcm-eq-0-iff* [of $m \ n$] **by** *auto*

lemma *lcm-pos-int*: $m \neq 0 \implies n \neq 0 \implies \text{lcm } m \ n > 0$
for $m \ n :: \text{int}$
by (*simp add: less-le lcm-eq-0-iff*)

lemma *dvd-pos-nat*: $n > 0 \implies m \ \text{dvd} \ n \implies m > 0$
for $m \ n :: \text{nat}$
by *auto*

lemma *lcm-unique-nat*:
 $a \ \text{dvd} \ d \wedge b \ \text{dvd} \ d \wedge (\forall e. a \ \text{dvd} \ e \wedge b \ \text{dvd} \ e \longrightarrow d \ \text{dvd} \ e) \longleftrightarrow d = \text{lcm } a \ b$
for $a \ b \ d :: \text{nat}$
by (*auto intro: dvd-antisym lcm-least*)

lemma *lcm-unique-int*:
 $d \geq 0 \wedge a \ \text{dvd} \ d \wedge b \ \text{dvd} \ d \wedge (\forall e. a \ \text{dvd} \ e \wedge b \ \text{dvd} \ e \longrightarrow d \ \text{dvd} \ e) \longleftrightarrow d = \text{lcm } a \ b$
for $a \ b \ d :: \text{int}$
using *lcm-least zdvd-antisym-nonneg* **by** *auto*

lemma *lcm-proj2-if-dvd-nat* [simp]: $x \ \text{dvd} \ y \implies \text{lcm } x \ y = y$
for $x \ y :: \text{nat}$
by (*simp add: lcm-proj2-if-dvd*)

lemma *lcm-proj2-if-dvd-int* [simp]: $x \ \text{dvd} \ y \implies \text{lcm } x \ y = |y|$
for $x \ y :: \text{int}$
by (*simp add: lcm-proj2-if-dvd*)

lemma *lcm-proj1-if-dvd-nat* [simp]: $x \ \text{dvd} \ y \implies \text{lcm } y \ x = y$
for $x \ y :: \text{nat}$
by (*subst lcm.commute*) (*erule lcm-proj2-if-dvd-nat*)

lemma *lcm-proj1-if-dvd-int* [simp]: $x \ \text{dvd} \ y \implies \text{lcm } y \ x = |y|$

for $x y :: \text{int}$
by (*subst lcm.commute*) (*erule lcm-proj2-if-dvd-int*)

lemma *lcm-proj1-iff-nat* [*simp*]: $\text{lcm } m \ n = m \longleftrightarrow n \ \text{dvd } m$
for $m \ n :: \text{nat}$
by (*metis lcm-proj1-if-dvd-nat lcm-unique-nat*)

lemma *lcm-proj2-iff-nat* [*simp*]: $\text{lcm } m \ n = n \longleftrightarrow m \ \text{dvd } n$
for $m \ n :: \text{nat}$
by (*metis lcm-proj2-if-dvd-nat lcm-unique-nat*)

lemma *lcm-proj1-iff-int* [*simp*]: $\text{lcm } m \ n = |m| \longleftrightarrow n \ \text{dvd } m$
for $m \ n :: \text{int}$
by (*metis dvd-abs-iff lcm-proj1-if-dvd-int lcm-unique-int*)

lemma *lcm-proj2-iff-int* [*simp*]: $\text{lcm } m \ n = |n| \longleftrightarrow m \ \text{dvd } n$
for $m \ n :: \text{int}$
by (*metis dvd-abs-iff lcm-proj2-if-dvd-int lcm-unique-int*)

lemma *lcm-1-iff-nat* [*simp*]: $\text{lcm } m \ n = \text{Suc } 0 \longleftrightarrow m = \text{Suc } 0 \wedge n = \text{Suc } 0$
for $m \ n :: \text{nat}$
using *lcm-eq-1-iff* [*of m n*] **by** *simp*

lemma *lcm-1-iff-int* [*simp*]: $\text{lcm } m \ n = 1 \longleftrightarrow (m = 1 \vee m = -1) \wedge (n = 1 \vee n = -1)$
for $m \ n :: \text{int}$
by *auto*

88.9 The complete divisibility lattice on *nat* and *int*

Lifting *gcd* and *lcm* to sets (*Gcd* / *Lcm*). *Gcd* is defined via *Lcm* to facilitate the proof that we have a complete lattice.

instantiation $\text{nat} :: \text{semiring-Gcd}$
begin

interpretation *semilattice-neutr-set lcm 1::nat*
by *standard simp-all*

definition $\text{Lcm } M = (\text{if } \text{finite } M \text{ then } F \ M \ \text{else } 0)$ **for** $M :: \text{nat set}$

lemma *Lcm-nat-empty*: $\text{Lcm } \{\} = (1::\text{nat})$
by (*simp add: Lcm-nat-def del: One-nat-def*)

lemma *Lcm-nat-insert*: $\text{Lcm } (\text{insert } n \ M) = \text{lcm } n \ (\text{Lcm } M)$ **for** $n :: \text{nat}$
by (*cases finite M*) (*auto simp: Lcm-nat-def simp del: One-nat-def*)

lemma *Lcm-nat-infinite*: $\text{infinite } M \implies \text{Lcm } M = 0$ **for** $M :: \text{nat set}$
by (*simp add: Lcm-nat-def*)

lemma *dvd-Lcm-nat* [*simp*]:

fixes $M :: \text{nat set}$

assumes $m \in M$

shows $m \text{ dvd } \text{Lcm } M$

proof –

from *assms* **have** $\text{insert } m M = M$

by *auto*

moreover **have** $m \text{ dvd } \text{Lcm } (\text{insert } m M)$

by (*simp add: Lcm-nat-insert*)

ultimately **show** *?thesis*

by *simp*

qed

lemma *Lcm-dvd-nat* [*simp*]:

fixes $M :: \text{nat set}$

assumes $\forall m \in M. m \text{ dvd } n$

shows $\text{Lcm } M \text{ dvd } n$

proof (*cases* $n > 0$)

case *False*

then **show** *?thesis* **by** *simp*

next

case *True*

then **have** $\text{finite } \{d. d \text{ dvd } n\}$

by (*rule finite-divisors-nat*)

moreover **have** $M \subseteq \{d. d \text{ dvd } n\}$

using *assms* **by** *fast*

ultimately **have** $\text{finite } M$

by (*rule rev-finite-subset*)

then **show** *?thesis*

using *assms* **by** (*induct* M) (*simp-all add: Lcm-nat-empty Lcm-nat-insert*)

qed

definition $\text{Gcd } M = \text{Lcm } \{d. \forall m \in M. d \text{ dvd } m\}$ **for** $M :: \text{nat set}$

instance

proof

fix $N :: \text{nat set}$

fix $n :: \text{nat}$

show $\text{Gcd } N \text{ dvd } n$ **if** $n \in N$

using *that* **by** (*induct* N *rule: infinite-finite-induct*) (*auto simp: Gcd-nat-def*)

show $n \text{ dvd } \text{Gcd } N$ **if** $\bigwedge m. m \in N \implies n \text{ dvd } m$

using *that* **by** (*induct* N *rule: infinite-finite-induct*) (*auto simp: Gcd-nat-def*)

show $n \text{ dvd } \text{Lcm } N$ **if** $n \in N$

using *that* **by** (*induct* N *rule: infinite-finite-induct*) *auto*

show $\text{Lcm } N \text{ dvd } n$ **if** $\bigwedge m. m \in N \implies m \text{ dvd } n$

using *that* **by** (*induct* N *rule: infinite-finite-induct*) *auto*

show $\text{normalize } (\text{Gcd } N) = \text{Gcd } N$ **and** $\text{normalize } (\text{Lcm } N) = \text{Lcm } N$

by *simp-all*

qed

end

lemma *Gcd-nat-eq-one*: $1 \in N \implies \text{Gcd } N = 1$
for $N :: \text{nat set}$
by (*rule Gcd-eq-1-I*) *auto*

instance *nat* :: *semiring-gcd-mult-normalize*
by *intro-classes (auto simp: unit-factor-nat-def)*

Alternative characterizations of Gcd:

lemma *Gcd-eq-Max*:

fixes $M :: \text{nat set}$
assumes *finite* ($M :: \text{nat set}$) **and** $M \neq \{\}$ **and** $0 \notin M$
shows $\text{Gcd } M = \text{Max } (\bigcap m \in M. \{d. d \text{ dvd } m\})$
proof (*rule antisym*)
from *assms* **obtain** m **where** $m \in M$ **and** $m > 0$
by *auto*
from $\langle m > 0 \rangle$ **have** *finite* $\{d. d \text{ dvd } m\}$
by (*blast intro: finite-divisors-nat*)
with $\langle m \in M \rangle$ **have** *fin*: *finite* $(\bigcap m \in M. \{d. d \text{ dvd } m\})$
by *blast*
from *fin* **show** $\text{Gcd } M \leq \text{Max } (\bigcap m \in M. \{d. d \text{ dvd } m\})$
by (*auto intro: Max-ge Gcd-dvd*)
from *fin* **show** $\text{Max } (\bigcap m \in M. \{d. d \text{ dvd } m\}) \leq \text{Gcd } M$
proof (*rule Max.boundedI, simp-all*)
show $(\bigcap m \in M. \{d. d \text{ dvd } m\}) \neq \{\}$
by *auto*
show $\bigwedge a. \forall x \in M. a \text{ dvd } x \implies a \leq \text{Gcd } M$
by (*meson Gcd-dvd Gcd-greatest* $\langle 0 < m \rangle \langle m \in M \rangle$ *dvd-imp-le dvd-pos-nat*)
qed
qed

lemma *Gcd-remove0-nat*: $\text{finite } M \implies \text{Gcd } M = \text{Gcd } (M - \{0\})$
for $M :: \text{nat set}$
proof (*induct pred: finite*)
case (*insert x M*)
then show *?case*
by (*simp add: insert-Diff-if*)
qed *auto*

lemma *Lcm-in-lcm-closed-set-nat*:

fixes $M :: \text{nat set}$
assumes *finite* M $M \neq \{\}$ $\bigwedge m n. [m \in M; n \in M] \implies \text{lcm } m n \in M$
shows $\text{Lcm } M \in M$
using *assms*
proof (*induction M rule: finite-linorder-min-induct*)
case (*insert x M*)
then have $\bigwedge m n. m \in M \implies n \in M \implies \text{lcm } m n \in M$

by (metis dvd-lcm1 gr0I insert-iff lcm-pos-nat nat-dvd-not-less)
 with insert show ?case
 by simp (metis Lcm-nat-empty One-nat-def dvd-1-left dvd-lcm2)
 qed auto

lemma Lcm-eq-Max-nat:

fixes $M :: \text{nat set}$
 assumes $M: \text{finite } M \ M \neq \{\} \ 0 \notin M$ and $\text{lcm}: \bigwedge m n. \llbracket m \in M; n \in M \rrbracket \implies \text{lcm } m \ n \in M$
 shows $\text{Lcm } M = \text{Max } M$
proof (rule antisym)
 show $\text{Lcm } M \leq \text{Max } M$
 by (simp add: Lcm-in-lcm-closed-set-nat ⟨finite M⟩ ⟨M ≠ {}⟩ lcm)
 show $\text{Max } M \leq \text{Lcm } M$
 by (meson Lcm-0-iff Max-in M dvd-Lcm dvd-imp-le le-0-eq not-le)
 qed

lemma mult-inj-if-coprime-nat:

$\text{inj-on } f \ A \implies \text{inj-on } g \ B \implies (\bigwedge a b. \llbracket a \in A; b \in B \rrbracket \implies \text{coprime } (f \ a) \ (g \ b)) \implies$
 $\text{inj-on } (\lambda(a, b). f \ a * g \ b) \ (A \times B)$
for $f :: 'a \Rightarrow \text{nat}$ and $g :: 'b \Rightarrow \text{nat}$
by (auto simp: inj-on-def coprime-crossproduct-nat simp del: One-nat-def)

88.9.1 Setwise GCD and LCM for integers

instantiation $\text{int} :: \text{Gcd}$
begin

definition $\text{Gcd-int} :: \text{int set} \Rightarrow \text{int}$
 where $\text{Gcd } K = \text{int } (\text{GCD } k \in K. (\text{nat} \circ \text{abs}) \ k)$

definition $\text{Lcm-int} :: \text{int set} \Rightarrow \text{int}$
 where $\text{Lcm } K = \text{int } (\text{LCM } k \in K. (\text{nat} \circ \text{abs}) \ k)$

instance ..

end

lemma Gcd-int-eq [simp]:
 $(\text{GCD } n \in N. \text{int } n) = \text{int } (\text{Gcd } N)$
by (simp add: Gcd-int-def image-image)

lemma Gcd-nat-abs-eq [simp]:
 $(\text{GCD } k \in K. \text{nat } |k|) = \text{nat } (\text{Gcd } K)$
by (simp add: Gcd-int-def)

lemma abs-Gcd-eq [simp]:
 $|\text{Gcd } K| = \text{Gcd } K$ **for** $K :: \text{int set}$
by (simp only: Gcd-int-def)

```

lemma Gcd-int-greater-eq-0 [simp]:
  Gcd K ≥ 0
  for K :: int set
  using abs-ge-zero [of Gcd K] by simp

lemma Gcd-abs-eq [simp]:
  (GCD k ∈ K. |k|) = Gcd K
  for K :: int set
  by (simp only: Gcd-int-def image-image) simp

lemma Lcm-int-eq [simp]:
  (LCM n ∈ N. int n) = int (Lcm N)
  by (simp add: Lcm-int-def image-image)

lemma Lcm-nat-abs-eq [simp]:
  (LCM k ∈ K. nat |k|) = nat (Lcm K)
  by (simp add: Lcm-int-def)

lemma abs-Lcm-eq [simp]:
  |Lcm K| = Lcm K for K :: int set
  by (simp only: Lcm-int-def)

lemma Lcm-int-greater-eq-0 [simp]:
  Lcm K ≥ 0
  for K :: int set
  using abs-ge-zero [of Lcm K] by simp

lemma Lcm-abs-eq [simp]:
  (LCM k ∈ K. |k|) = Lcm K
  for K :: int set
  by (simp only: Lcm-int-def image-image) simp

instance int :: semiring-Gcd
proof
  fix K :: int set and k :: int
  show Gcd K dvd k and k dvd Lcm K if k ∈ K
    using that Gcd-dvd [of nat |k| (nat ∘ abs) 'K]
      dvd-Lcm [of nat |k| (nat ∘ abs) 'K]
    by (simp-all add: comp-def)
  show k dvd Gcd K if  $\bigwedge l. l \in K \implies k \text{ dvd } l$ 
  proof –
    have nat |k| dvd (GCD k ∈ K. nat |k|)
      by (rule Gcd-greatest) (use that in auto)
    then show ?thesis by simp
  qed
  show Lcm K dvd k if  $\bigwedge l. l \in K \implies l \text{ dvd } k$ 
  proof –
    have (LCM k ∈ K. nat |k|) dvd nat |k|

```

```

    by (rule Lcm-least) (use that in auto)
    then show ?thesis by simp
  qed
qed (simp-all add: sgn-mult)

```

```

instance int :: semiring-gcd-mult-normalize
  by intro-classes (auto simp: sgn-mult)

```

88.10 GCD and LCM on *integer*

```

instantiation integer :: gcd
begin

```

```

context
  includes integer.lifting
begin

```

```

lift-definition gcd-integer :: integer  $\Rightarrow$  integer  $\Rightarrow$  integer is gcd .

```

```

lift-definition lcm-integer :: integer  $\Rightarrow$  integer  $\Rightarrow$  integer is lcm .

```

```

end

```

```

instance ..

```

```

end

```

```

lifting-update integer.lifting
lifting-forget integer.lifting

```

```

context
  includes integer.lifting
begin

```

```

lemma gcd-code-integer [code]: gcd k l = |if l = (0::integer) then k else gcd l (|k|
mod |l|)|
  by transfer (fact gcd-code-int)

```

```

lemma lcm-code-integer [code]: lcm a b = |a| * |b| div gcd a b
  for a b :: integer
  by transfer (fact lcm-altdef-int)

```

```

end

```

```

code-printing

```

```

  constant gcd :: integer  $\Rightarrow$  -  $\rightarrow$ 
    (OCaml) !(fun k l  $\rightarrow$  if Z.equal k Z.zero then/ Z.abs l else if Z.equal/ l Z.zero
then Z.abs k else Z.gcd k l)
  and (Haskell) Prelude.gcd

```

and (*Scala*) `-.gcd'((-)')`

— There is no gcd operation in the SML standard library, so no code setup for SML

Some code equations

lemmas `Gcd-nat-set-eq-fold [code] = Gcd-set-eq-fold [where ?'a = nat]`
lemmas `Lcm-nat-set-eq-fold [code] = Lcm-set-eq-fold [where ?'a = nat]`
lemmas `Gcd-int-set-eq-fold [code] = Gcd-set-eq-fold [where ?'a = int]`
lemmas `Lcm-int-set-eq-fold [code] = Lcm-set-eq-fold [where ?'a = int]`

Fact aliases.

lemma `lcm-0-iff-nat [simp]: lcm m n = 0 \longleftrightarrow m = 0 \vee n = 0`
for `m n :: nat`
by (`fact lcm-eq-0-iff`)

lemma `lcm-0-iff-int [simp]: lcm m n = 0 \longleftrightarrow m = 0 \vee n = 0`
for `m n :: int`
by (`fact lcm-eq-0-iff`)

lemma `dvd-lcm-I1-nat [simp]: k dvd m \implies k dvd lcm m n`
for `k m n :: nat`
by (`fact dvd-lcmI1`)

lemma `dvd-lcm-I2-nat [simp]: k dvd n \implies k dvd lcm m n`
for `k m n :: nat`
by (`fact dvd-lcmI2`)

lemma `dvd-lcm-I1-int [simp]: i dvd m \implies i dvd lcm m n`
for `i m n :: int`
by (`fact dvd-lcmI1`)

lemma `dvd-lcm-I2-int [simp]: i dvd n \implies i dvd lcm m n`
for `i m n :: int`
by (`fact dvd-lcmI2`)

lemmas `Gcd-dvd-nat [simp] = Gcd-dvd [where ?'a = nat]`
lemmas `Gcd-dvd-int [simp] = Gcd-dvd [where ?'a = int]`
lemmas `Gcd-greatest-nat [simp] = Gcd-greatest [where ?'a = nat]`
lemmas `Gcd-greatest-int [simp] = Gcd-greatest [where ?'a = int]`

lemma `dvd-Lcm-int [simp]: m \in M \implies m dvd Lcm M`
for `M :: int set`
by (`fact dvd-Lcm`)

lemma `gcd-neg-numeral-1-int [simp]: gcd (- numeral n :: int) x = gcd (numeral n) x`
by (`fact gcd-neg1-int`)

lemma `gcd-neg-numeral-2-int [simp]: gcd x (- numeral n :: int) = gcd x (numeral n)`

```

n)
  by (fact gcd-neg2-int)

lemma gcd-proj1-if-dvd-nat [simp]: x dvd y  $\implies$  gcd x y = x
  for x y :: nat
  by (fact gcd-nat.absorb1)

lemma gcd-proj2-if-dvd-nat [simp]: y dvd x  $\implies$  gcd x y = y
  for x y :: nat
  by (fact gcd-nat.absorb2)

lemma Gcd-in:
  fixes A :: nat set
  assumes  $\bigwedge a b. a \in A \implies b \in A \implies \text{gcd } a \ b \in A$ 
  assumes  $A \neq \{\}$ 
  shows  $\text{Gcd } A \in A$ 
proof (cases A = {0})
  case False
  with assms obtain x where x  $\in$  A x > 0
  by auto
  thus  $\text{Gcd } A \in A$ 
proof (induction x rule: less-induct)
  case (less x)
  show ?case
  proof (cases x = Gcd A)
  case False
  have  $\exists y \in A. \neg x \text{ dvd } y$ 
  using False less.prem1 by (metis Gcd-dvd Gcd-greatest-nat gcd-nat.asym)
  then obtain y where y: y  $\in$  A  $\neg x \text{ dvd } y$ 
  by blast
  have gcd x y  $\in$  A
  by (rule assms(1)) (use  $\langle x \in A \rangle$  y in auto)
  moreover have gcd x y < x
  using  $\langle x > 0 \rangle$  y by (metis gcd-dvd1 gcd-dvd2 nat-dvd-not-less nat-neq-iff)
  moreover have gcd x y > 0
  using  $\langle x > 0 \rangle$  by auto
  ultimately show ?thesis using less.IH by blast
qed (use less in auto)
qed
qed auto

lemma bezout-gcd-nat':
  fixes a b :: nat
  shows  $\exists x y. b * y \leq a * x \wedge a * x - b * y = \text{gcd } a \ b \vee a * y \leq b * x \wedge b * x - a * y = \text{gcd } a \ b$ 
  using bezout-nat[of a b]
  by (metis add-diff-cancel-left' diff-zero gcd commute gcd-0-nat
  le-add-same-cancel1 mult.right-neutral zero-le)

```

lemmas *Lcm-eq-0-I-nat* [simp] = *Lcm-eq-0-I* [where ?'a = nat]
lemmas *Lcm-0-iff-nat* [simp] = *Lcm-0-iff* [where ?'a = nat]
lemmas *Lcm-least-int* [simp] = *Lcm-least* [where ?'a = int]

88.11 Characteristic of a semiring

definition (in *semiring-1*) *semiring-char* :: 'a itself \Rightarrow nat
 where *semiring-char* - = *Gcd* {n. of-nat n = (0 :: 'a)}

context *semiring-1*
begin

context
 fixes *CHAR* :: nat
 defines *CHAR* \equiv *semiring-char* (*Pure.type* :: 'a itself)
begin

lemma *of-nat-CHAR* [simp]: *of-nat CHAR* = (0 :: 'a)

proof –

have *CHAR* \in {n. of-nat n = (0 :: 'a)}

unfolding *CHAR-def semiring-char-def*

proof (*rule Gcd-in, clarify*)

fix a b :: nat

assume *: *of-nat a* = (0 :: 'a) *of-nat b* = (0 :: 'a)

show *of-nat (gcd a b)* = (0 :: 'a)

proof (*cases a = 0*)

case *False*

with bezout-nat obtain x y **where** a * x = b * y + *gcd a b*

by *blast*

hence *of-nat (a * x)* = (*of-nat (b * y + gcd a b)* :: 'a)

by (*rule arg-cong*)

thus *of-nat (gcd a b)* = (0 :: 'a)

using * **by** *simp*

qed (*use * in auto*)

next

have *of-nat 0* = (0 :: 'a)

by *simp*

thus {n. of-nat n = (0 :: 'a)} \neq {}

by *blast*

qed

thus ?thesis

by *simp*

qed

lemma *of-nat-eq-0-iff-char-dvd*: *of-nat n* = (0 :: 'a) \longleftrightarrow *CHAR dvd n*

proof

assume *of-nat n* = (0 :: 'a)

thus *CHAR dvd n*

unfolding *CHAR-def semiring-char-def* **by** (*intro Gcd-dvd*) *auto*


```

next
  assume CHAR dvd n
  then obtain m where n = CHAR * m
    by auto
  thus of-nat n = (0 :: 'a)
    by simp
qed

lemma CHAR-eqI:
  assumes of-nat c = (0 :: 'a)
  assumes  $\bigwedge x. \text{of-nat } x = (0 :: 'a) \implies c \text{ dvd } x$ 
  shows CHAR = c
  using assms by (intro dvd-antisym) (auto simp: of-nat-eq-0-iff-char-dvd)

lemma CHAR-eq0-iff: CHAR = 0  $\longleftrightarrow$  ( $\forall n > 0. \text{of-nat } n \neq (0 :: 'a)$ )
  by (auto simp: of-nat-eq-0-iff-char-dvd)

lemma CHAR-pos-iff: CHAR > 0  $\longleftrightarrow$  ( $\exists n > 0. \text{of-nat } n = (0 :: 'a)$ )
  using CHAR-eq0-iff neq0-conv by blast

lemma CHAR-eq-posI:
  assumes  $c > 0$  of-nat c = (0 :: 'a)  $\bigwedge x. x > 0 \implies x < c \implies \text{of-nat } x \neq (0 :: 'a)$ 
  shows CHAR = c
  proof (rule antisym)
    from assms have CHAR > 0
      by (auto simp: CHAR-pos-iff)
    from assms(3)[OF this] show CHAR  $\geq$  c
      by force
  next
    have CHAR dvd c
      using assms by (auto simp: of-nat-eq-0-iff-char-dvd)
    thus CHAR  $\leq$  c
      using  $\langle c > 0 \rangle$  by (intro dvd-imp-le) auto
  qed

end
end

lemma (in semiring-char-0) CHAR-eq-0 [simp]: semiring-char (Pure.type :: 'a
itself) = 0
  by (simp add: CHAR-eq0-iff)

syntax -type-char :: type  $\Rightarrow$  nat ((1CHAR/(1'(-))))

translations CHAR('t)  $\Rightarrow$  CONST semiring-char (CONST Pure.type :: 't itself)

```

```

print-translation ⟨
  let
    fun char-type-tr' ctxt [Const (const-syntax ⟨Pure.type⟩, Type (-, [T]))] =
      Syntax.const syntax-const ⟨-type-char⟩ $ Syntax-Phases.term-of-typ ctxt T
    in [(const-syntax ⟨semiring-char⟩, char-type-tr') end
  ]
  ⟨
end

```

89 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

```

theory Nitpick
imports Record GCD
keywords
  nitpick :: diag and
  nitpick-params :: thy-decl
begin

datatype (plugins only: extraction) (dead 'a, dead 'b) fun-box = FunBox 'a ⇒ 'b
datatype (plugins only: extraction) (dead 'a, dead 'b) pair-box = PairBox 'a 'b
datatype (plugins only: extraction) (dead 'a) word = Word 'a set

typedecl bisim-iterator
typedecl unsigned-bit
typedecl signed-bit

consts
  unknown :: 'a
  is-unknown :: 'a ⇒ bool
  bisim :: bisim-iterator ⇒ 'a ⇒ 'a ⇒ bool
  bisim-iterator-max :: bisim-iterator
  Quot :: 'a ⇒ 'b
  safe-The :: ('a ⇒ bool) ⇒ 'a

```

Alternative definitions.

```

lemma Ex1-unfold[nitpick-unfold]: Ex1 P ≡ ∃ x. {x. P x} = {x}
apply (rule eq-reflection)
apply (simp add: Ex1-def set-eq-iff)
apply (rule iffI)
apply (erule exE)
apply (erule conjE)
apply (rule-tac x = x in exI)
apply (rule allI)
apply (rename-tac y)
apply (erule-tac x = y in allE)
by auto

```

lemma *rtrancl-unfold*[*nitpick-unfold*]: $r^* \equiv (r^+)^=$
by (*simp only*: *rtrancl-trancl-reflcl*)

lemma *rtranclp-unfold*[*nitpick-unfold*]: $rtranclp\ r\ a\ b \equiv (a = b \vee tranclp\ r\ a\ b)$
by (*rule eq-reflection*) (*auto dest*: *rtranclpD*)

lemma *tranclp-unfold*[*nitpick-unfold*]:
 $tranclp\ r\ a\ b \equiv (a, b) \in trancl\ \{(x, y). r\ x\ y\}$
by (*simp add*: *trancl-def*)

lemma [*nitpick-simp*]:
 $of\text{-}nat\ n = (if\ n = 0\ then\ 0\ else\ 1 + of\text{-}nat\ (n - 1))$
by (*cases n*) *auto*

definition *prod* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \times 'b)\ set$ **where**
 $prod\ A\ B = \{(a, b). a \in A \wedge b \in B\}$

definition *refl'* :: $('a \times 'a)\ set \Rightarrow bool$ **where**
 $refl'\ r \equiv \forall x. (x, x) \in r$

definition *wf'* :: $('a \times 'a)\ set \Rightarrow bool$ **where**
 $wf'\ r \equiv acyclic\ r \wedge (finite\ r \vee unknown)$

definition *card'* :: $'a\ set \Rightarrow nat$ **where**
 $card'\ A \equiv if\ finite\ A\ then\ length\ (SOME\ xs.\ set\ xs = A \wedge distinct\ xs)\ else\ 0$

definition *sum'* :: $('a \Rightarrow 'b::comm\text{-}monoid\text{-}add) \Rightarrow 'a\ set \Rightarrow 'b$ **where**
 $sum'\ f\ A \equiv if\ finite\ A\ then\ sum\text{-}list\ (map\ f\ (SOME\ xs.\ set\ xs = A \wedge distinct\ xs))$
 $else\ 0$

inductive *fold-graph'* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$ **where**
 $fold\text{-}graph'\ f\ z\ \{\} z \mid$
 $[x \in A; fold\text{-}graph'\ f\ z\ (A - \{x\})\ y] \Longrightarrow fold\text{-}graph'\ f\ z\ A\ (f\ x\ y)$

The following lemmas are not strictly necessary but they help the *specialize* optimization.

lemma *The-psimp*[*nitpick-psimp*]: $P = (=)\ x \Longrightarrow The\ P = x$
by *auto*

lemma *Eps-psimp*[*nitpick-psimp*]:
 $\llbracket P\ x; \neg P\ y; Eps\ P = y \rrbracket \Longrightarrow Eps\ P = x$
apply (*cases P (Eps P)*)
apply *auto*
apply (*erule contrapos-np*)
by (*rule someI*)

lemma *case-unit-unfold*[*nitpick-unfold*]:
 $case\text{-}unit\ x\ u \equiv x$

```

apply (subgoal-tac u = ())
apply (simp only: unit.case)
by simp

```

```

declare unit.case[nitpick-simp del]

```

```

lemma case-nat-unfold[nitpick-unfold]:
  case-nat x f n  $\equiv$  if n = 0 then x else f (n - 1)
apply (rule eq-reflection)
by (cases n) auto

```

```

declare nat.case[nitpick-simp del]

```

```

lemma size-list-simp[nitpick-simp]:
  size-list f xs = (if xs = [] then 0 else Suc (f (hd xs) + size-list f (tl xs)))
  size xs = (if xs = [] then 0 else Suc (size (tl xs)))
by (cases xs) auto

```

Auxiliary definitions used to provide an alternative representation for *rat* and *real*.

```

fun nat-gcd :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat where
  nat-gcd x y = (if y = 0 then x else nat-gcd y (x mod y))

```

```

declare nat-gcd.simps [simp del]

```

```

definition nat-lcm :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat where
  nat-lcm x y = x * y div (nat-gcd x y)

```

```

lemma gcd-eq-nitpick-gcd [nitpick-unfold]:
  gcd x y = Nitpick.nat-gcd x y
by (induct x y rule: nat-gcd.induct)
  (simp add: gcd-nat.simps Nitpick.nat-gcd.simps)

```

```

lemma lcm-eq-nitpick-lcm [nitpick-unfold]:
  lcm x y = Nitpick.nat-lcm x y
by (simp only: lcm-nat-def Nitpick.nat-lcm-def gcd-eq-nitpick-gcd)

```

```

definition Frac :: int  $\times$  int  $\Rightarrow$  bool where
  Frac  $\equiv$   $\lambda(a, b).$  b > 0  $\wedge$  coprime a b

```

consts

```

  Abs-Frac :: int  $\times$  int  $\Rightarrow$  'a
  Rep-Frac :: 'a  $\Rightarrow$  int  $\times$  int

```

```

definition zero-frac :: 'a where
  zero-frac  $\equiv$  Abs-Frac (0, 1)

```

```

definition one-frac :: 'a where
  one-frac  $\equiv$  Abs-Frac (1, 1)

```

definition $num :: 'a \Rightarrow int$ **where**

$num \equiv fst \circ Rep-Frac$

definition $denom :: 'a \Rightarrow int$ **where**

$denom \equiv snd \circ Rep-Frac$

function $norm-frac :: int \Rightarrow int \Rightarrow int \times int$ **where**

$norm-frac a b =$

(if $b < 0$ then $norm-frac (- a) (- b)$

else if $a = 0 \vee b = 0$ then $(0, 1)$

else let $c = gcd a b$ in $(a \div c, b \div c)$)

by *pat-completeness auto*

termination by (relation measure $(\lambda(-, b). \text{if } b < 0 \text{ then } 1 \text{ else } 0)$) *auto*

declare $norm-frac.simps[simp del]$

definition $frac :: int \Rightarrow int \Rightarrow 'a$ **where**

$frac a b \equiv Abs-Frac (norm-frac a b)$

definition $plus-frac :: 'a \Rightarrow 'a \Rightarrow 'a$ **where**

[*nitpick-simp*]: $plus-frac q r = (let d = lcm (denom q) (denom r)$ in

$frac (num q * (d \div denom q) + num r * (d \div denom r)) d)$

definition $times-frac :: 'a \Rightarrow 'a \Rightarrow 'a$ **where**

[*nitpick-simp*]: $times-frac q r = frac (num q * num r) (denom q * denom r)$

definition $uminus-frac :: 'a \Rightarrow 'a$ **where**

$uminus-frac q \equiv Abs-Frac (- num q, denom q)$

definition $number-of-frac :: int \Rightarrow 'a$ **where**

$number-of-frac n \equiv Abs-Frac (n, 1)$

definition $inverse-frac :: 'a \Rightarrow 'a$ **where**

$inverse-frac q \equiv frac (denom q) (num q)$

definition $less-frac :: 'a \Rightarrow 'a \Rightarrow bool$ **where**

[*nitpick-simp*]: $less-frac q r \iff num (plus-frac q (uminus-frac r)) < 0$

definition $less-eq-frac :: 'a \Rightarrow 'a \Rightarrow bool$ **where**

[*nitpick-simp*]: $less-eq-frac q r \iff num (plus-frac q (uminus-frac r)) \leq 0$

definition $of-frac :: 'a \Rightarrow 'b::\{inverse,ring-1\}$ **where**

$of-frac q \equiv of-int (num q) / of-int (denom q)$

axiomatization $wf-wfrec :: ('a \times 'a) set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

definition $wf-wfrec' :: ('a \times 'a) set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**

[*nitpick-simp*]: $wf-wfrec' R F x = F (cut (wf-wfrec R F) R x) x$

definition $wfrec' :: ('a \times 'a) \text{ set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
 $wfrec' R F x \equiv \text{if } wf R \text{ then } wf\text{-}wfrec' R F x \text{ else } THE y. wfrec\text{-rel } R (\lambda f x. F (\text{cut } f R x) x) x y$

ML-file $\langle Tools/Nitpick/kodkod.ML \rangle$
ML-file $\langle Tools/Nitpick/kodkod-sat.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-util.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-hol.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-mono.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-preproc.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-scope.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-peephole.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-rep.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-nut.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-kodkod.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-model.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-commands.ML \rangle$
ML-file $\langle Tools/Nitpick/nitpick-tests.ML \rangle$

setup \langle
Nitpick-HOL.register-ersatz-global
 $[(\mathbf{const-name} \langle card \rangle, \mathbf{const-name} \langle card' \rangle),$
 $(\mathbf{const-name} \langle sum \rangle, \mathbf{const-name} \langle sum' \rangle),$
 $(\mathbf{const-name} \langle fold-graph \rangle, \mathbf{const-name} \langle fold-graph' \rangle),$
 $(\mathbf{const-name} \langle wf \rangle, \mathbf{const-name} \langle wf' \rangle),$
 $(\mathbf{const-name} \langle wf\text{-}wfrec \rangle, \mathbf{const-name} \langle wf\text{-}wfrec' \rangle),$
 $(\mathbf{const-name} \langle wfrec \rangle, \mathbf{const-name} \langle wfrec' \rangle)]$
 \rangle

hide-const (open) *unknown is-unknown bisim bisim-iterator-max Quot safe-The FunBox PairBox Word prod*

refl' wf' card' sum' fold-graph' nat-gcd nat-lcm Frac Abs-Frac Rep-Frac
zero-frac one-frac num denom norm-frac frac plus-frac times-frac uminus-frac
number-of-frac
inverse-frac less-frac less-eq-frac of-frac wf-wfrec wf-wfrec' wfrec'

hide-type (open) *bisim-iterator fun-box pair-box unsigned-bit signed-bit word*

hide-fact (open) *Ex1-unfold rtrancl-unfold rtranclp-unfold tranclp-unfold prod-def*
refl'-def wf'-def
card'-def sum'-def The-psimp Eps-psimp case-unit-unfold case-nat-unfold
size-list-simp nat-lcm-def Frac-def zero-frac-def one-frac-def
num-def denom-def frac-def plus-frac-def times-frac-def uminus-frac-def
number-of-frac-def inverse-frac-def less-frac-def less-eq-frac-def of-frac-def wf-wfrec'-def
wfrec'-def

end

```

theory Nunchaku
imports Nitpick
keywords
  nunchaku :: diag and
  nunchaku-params :: thy-decl
begin

consts unreachable :: 'a

definition The-unsafe :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a where
  The-unsafe = The

definition rmember :: 'a set  $\Rightarrow$  'a  $\Rightarrow$  bool where
  rmember A x  $\longleftrightarrow$  x  $\in$  A

ML-file <Tools/Nunchaku/nunchaku-util.ML>
ML-file <Tools/Nunchaku/nunchaku-collect.ML>
ML-file <Tools/Nunchaku/nunchaku-problem.ML>
ML-file <Tools/Nunchaku/nunchaku-translate.ML>
ML-file <Tools/Nunchaku/nunchaku-model.ML>
ML-file <Tools/Nunchaku/nunchaku-reconstruct.ML>
ML-file <Tools/Nunchaku/nunchaku-display.ML>
ML-file <Tools/Nunchaku/nunchaku-tool.ML>
ML-file <Tools/Nunchaku/nunchaku.ML>
ML-file <Tools/Nunchaku/nunchaku-commands.ML>

hide-const (open) unreachable The-unsafe rmember

end

```

90 Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors

```

theory BNF-Greatest-Fixpoint
imports BNF-Fixpoint-Base String
keywords
  codatatype :: thy-defn and
  primcorecursive :: thy-goal-defn and
  primcorec :: thy-defn
begin

alias proj = Equiv-Relations.proj

lemma one-pointE:  $\llbracket \bigwedge x. s = x \implies P \rrbracket \implies P$ 
  by simp

```

lemma *obj-sumE*: $\llbracket \forall x. s = \text{Inl } x \longrightarrow P; \forall x. s = \text{Inr } x \longrightarrow P \rrbracket \Longrightarrow P$
by (*cases s*) *auto*

lemma *not-TrueE*: $\neg \text{True} \Longrightarrow P$
by (*erule notE*, *rule TrueI*)

lemma *neq-eq-eq-contradict*: $\llbracket t \neq u; s = t; s = u \rrbracket \Longrightarrow P$
by *fast*

lemma *converse-Times*: $(A \times B)^{-1} = B \times A$
by *fast*

lemma *equiv-proj*:

assumes *e*: *equiv A R* **and** *m*: $z \in R$

shows $(\text{proj } R \circ \text{fst}) z = (\text{proj } R \circ \text{snd}) z$

proof –

from *m* **have** z : $(\text{fst } z, \text{snd } z) \in R$ **by** *auto*

with *e* **have** $\bigwedge x. (\text{fst } z, x) \in R \Longrightarrow (\text{snd } z, x) \in R \bigwedge x. (\text{snd } z, x) \in R \Longrightarrow (\text{fst } z, x) \in R$

unfolding *equiv-def sym-def trans-def* **by** *blast+*

then show *?thesis* **unfolding** *proj-def[abs-def]* **by** *auto*

qed

definition *image2* **where** $\text{image2 } A f g = \{(f a, g a) \mid a. a \in A\}$

lemma *Id-on-Gr*: $\text{Id-on } A = \text{Gr } A \text{ id}$
unfolding *Id-on-def Gr-def* **by** *auto*

lemma *image2-eqI*: $\llbracket b = f x; c = g x; x \in A \rrbracket \Longrightarrow (b, c) \in \text{image2 } A f g$
unfolding *image2-def* **by** *auto*

lemma *IdD*: $(a, b) \in \text{Id} \Longrightarrow a = b$
by *auto*

lemma *image2-Gr*: $\text{image2 } A f g = (\text{Gr } A f)^{-1} O (\text{Gr } A g)$
unfolding *image2-def Gr-def* **by** *auto*

lemma *GrD1*: $(x, fx) \in \text{Gr } A f \Longrightarrow x \in A$
unfolding *Gr-def* **by** *simp*

lemma *GrD2*: $(x, fx) \in \text{Gr } A f \Longrightarrow f x = fx$
unfolding *Gr-def* **by** *simp*

lemma *Gr-incl*: $\text{Gr } A f \subseteq A \times B \longleftrightarrow f ' A \subseteq B$
unfolding *Gr-def* **by** *auto*

lemma *subset-Collect-iff*: $B \subseteq A \Longrightarrow (B \subseteq \{x \in A. P x\}) = (\forall x \in B. P x)$
by *blast*

lemma *subset-CollectI*: $B \subseteq A \implies (\bigwedge x. x \in B \implies Q\ x \implies P\ x) \implies (\{x \in B. Q\ x\} \subseteq \{x \in A. P\ x\})$
by *blast*

lemma *in-rel-Collect-case-prod-eq*: $in\text{-}rel\ (Collect\ (case\text{-}prod\ X)) = X$
unfolding *fun-eq-iff* **by** *auto*

lemma *Collect-case-prod-in-rel-leI*: $X \subseteq Y \implies X \subseteq Collect\ (case\text{-}prod\ (in\text{-}rel\ Y))$
by *auto*

lemma *Collect-case-prod-in-rel-leE*: $X \subseteq Collect\ (case\text{-}prod\ (in\text{-}rel\ Y)) \implies (X \subseteq Y \implies R) \implies R$
by *force*

lemma *conversep-in-rel*: $(in\text{-}rel\ R)^{-1^{-1}} = in\text{-}rel\ (R^{-1})$
unfolding *fun-eq-iff* **by** *auto*

lemma *relcompp-in-rel*: $in\text{-}rel\ R\ OO\ in\text{-}rel\ S = in\text{-}rel\ (R\ O\ S)$
unfolding *fun-eq-iff* **by** *auto*

lemma *in-rel-Gr*: $in\text{-}rel\ (Gr\ A\ f) = Grp\ A\ f$
unfolding *Gr-def Grp-def fun-eq-iff* **by** *auto*

definition *relImage* **where**
 $relImage\ R\ f \equiv \{(f\ a1, f\ a2) \mid a1\ a2. (a1, a2) \in R\}$

definition *relInvImage* **where**
 $relInvImage\ A\ R\ f \equiv \{(a1, a2) \mid a1\ a2. a1 \in A \wedge a2 \in A \wedge (f\ a1, f\ a2) \in R\}$

lemma *relImage-Gr*:
 $\llbracket R \subseteq A \times A \rrbracket \implies relImage\ R\ f = (Gr\ A\ f)^{-1}\ O\ R\ O\ Gr\ A\ f$
unfolding *relImage-def Gr-def relcomp-def* **by** *auto*

lemma *relInvImage-Gr*: $\llbracket R \subseteq B \times B \rrbracket \implies relInvImage\ A\ R\ f = Gr\ A\ f\ O\ R\ O\ (Gr\ A\ f)^{-1}$
unfolding *Gr-def relcomp-def image-def relInvImage-def* **by** *auto*

lemma *relImage-mono*:
 $R1 \subseteq R2 \implies relImage\ R1\ f \subseteq relImage\ R2\ f$
unfolding *relImage-def* **by** *auto*

lemma *relInvImage-mono*:
 $R1 \subseteq R2 \implies relInvImage\ A\ R1\ f \subseteq relInvImage\ A\ R2\ f$
unfolding *relInvImage-def* **by** *auto*

lemma *relInvImage-Id-on*:
 $(\bigwedge a1\ a2. f\ a1 = f\ a2 \longleftrightarrow a1 = a2) \implies relInvImage\ A\ (Id\text{-}on\ B)\ f \subseteq Id$

unfolding *relInvImage-def Id-on-def* **by** *auto*

lemma *relInvImage-UNIV-relImage*:

$R \subseteq \text{relInvImage UNIV } (\text{relImage } R \ f)$

unfolding *relInvImage-def relImage-def* **by** *auto*

lemma *relImage-proj*:

assumes *equiv A R*

shows $\text{relImage } R \ (\text{proj } R) \subseteq \text{Id-on } (A//R)$

unfolding *relImage-def Id-on-def*

using *proj-iff[OF assms] equiv-class-eq-iff[OF assms]*

by (*auto simp: proj-preserves*)

lemma *relImage-relInvImage*:

assumes $R \subseteq f' A \times f' A$

shows $\text{relImage } (\text{relInvImage } A \ R \ f) \ f = R$

using *assms* **unfolding** *relImage-def relInvImage-def* **by** *fast*

lemma *subst-Pair*: $P \ x \ y \implies a = (x, y) \implies P \ (\text{fst } a) \ (\text{snd } a)$

by *simp*

lemma *fst-diag-id*: $(\text{fst} \circ (\lambda x. (x, x))) \ z = \text{id } z$ **by** *simp*

lemma *snd-diag-id*: $(\text{snd} \circ (\lambda x. (x, x))) \ z = \text{id } z$ **by** *simp*

lemma *fst-diag-fst*: $\text{fst} \circ ((\lambda x. (x, x)) \circ \text{fst}) = \text{fst}$ **by** *auto*

lemma *snd-diag-fst*: $\text{snd} \circ ((\lambda x. (x, x)) \circ \text{fst}) = \text{fst}$ **by** *auto*

lemma *fst-diag-snd*: $\text{fst} \circ ((\lambda x. (x, x)) \circ \text{snd}) = \text{snd}$ **by** *auto*

lemma *snd-diag-snd*: $\text{snd} \circ ((\lambda x. (x, x)) \circ \text{snd}) = \text{snd}$ **by** *auto*

definition *Succ* **where** $\text{Succ } Kl \ kl = \{k . kl \ @ \ [k] \in Kl\}$

definition *Shift* **where** $\text{Shift } Kl \ k = \{kl. k \ \# \ kl \in Kl\}$

definition *shift* **where** $\text{shift } lab \ k = (\lambda kl. lab \ (k \ \# \ kl))$

lemma *empty-Shift*: $\llbracket \rrbracket \in Kl; k \in \text{Succ } Kl \ \llbracket \rrbracket \implies \llbracket \rrbracket \in \text{Shift } Kl \ k$

unfolding *Shift-def Succ-def* **by** *simp*

lemma *SuccD*: $k \in \text{Succ } Kl \ kl \implies kl \ @ \ [k] \in Kl$

unfolding *Succ-def* **by** *simp*

lemmas $\text{SuccE} = \text{SuccD}[\text{elim-format}]$

lemma *SuccI*: $kl \ @ \ [k] \in Kl \implies k \in \text{Succ } Kl \ kl$

unfolding *Succ-def* **by** *simp*

lemma *ShiftD*: $kl \in \text{Shift } Kl \ k \implies k \ \# \ kl \in Kl$

unfolding *Shift-def* **by** *simp*

lemma *Succ-Shift*: $\text{Succ } (\text{Shift } Kl \ k) \ kl = \text{Succ } Kl \ (k \ \# \ kl)$

unfolding *Succ-def Shift-def* **by** *auto*

lemma *length-Cons*: $\text{length } (x \# xs) = \text{Suc } (\text{length } xs)$
by *simp*

lemma *length-append-singleton*: $\text{length } (xs @ [x]) = \text{Suc } (\text{length } xs)$
by *simp*

definition *toCard-pred* $A r f \equiv \text{inj-on } f A \wedge f ' A \subseteq \text{Field } r \wedge \text{Card-order } r$

definition *toCard* $A r \equiv \text{SOME } f. \text{toCard-pred } A r f$

lemma *ex-toCard-pred*:

$\llbracket |A| \leq o r; \text{Card-order } r \rrbracket \Longrightarrow \exists f. \text{toCard-pred } A r f$

unfolding *toCard-pred-def*

using *card-of-ordLeq*[*of A Field r*]

ordLeq-ordIso-trans[*OF - card-of-unique*[*of Field r r*], *of |A|*]

by *blast*

lemma *toCard-pred-toCard*:

$\llbracket |A| \leq o r; \text{Card-order } r \rrbracket \Longrightarrow \text{toCard-pred } A r (\text{toCard } A r)$

unfolding *toCard-def* **using** *someI-ex*[*OF ex-toCard-pred*].

lemma *toCard-inj*: $\llbracket |A| \leq o r; \text{Card-order } r; x \in A; y \in A \rrbracket \Longrightarrow \text{toCard } A r x = \text{toCard } A r y \longleftrightarrow x = y$

using *toCard-pred-toCard* **unfolding** *inj-on-def* *toCard-pred-def* **by** *blast*

definition *fromCard* $A r k \equiv \text{SOME } b. b \in A \wedge \text{toCard } A r b = k$

lemma *fromCard-toCard*:

$\llbracket |A| \leq o r; \text{Card-order } r; b \in A \rrbracket \Longrightarrow \text{fromCard } A r (\text{toCard } A r b) = b$

unfolding *fromCard-def* **by** (*rule some-equality*) (*auto simp add: toCard-inj*)

lemma *Inl-Field-csum*: $a \in \text{Field } r \Longrightarrow \text{Inl } a \in \text{Field } (r + c s)$

unfolding *Field-card-of-csum-def* **by** *auto*

lemma *Inr-Field-csum*: $a \in \text{Field } s \Longrightarrow \text{Inr } a \in \text{Field } (r + c s)$

unfolding *Field-card-of-csum-def* **by** *auto*

lemma *rec-nat-0-imp*: $f = \text{rec-nat } f1 (\lambda n \text{ rec. } f2 n \text{ rec}) \Longrightarrow f 0 = f1$

by *auto*

lemma *rec-nat-Suc-imp*: $f = \text{rec-nat } f1 (\lambda n \text{ rec. } f2 n \text{ rec}) \Longrightarrow f (\text{Suc } n) = f2 n (f n)$

by *auto*

lemma *rec-list-Nil-imp*: $f = \text{rec-list } f1 (\lambda x xs \text{ rec. } f2 x xs \text{ rec}) \Longrightarrow f [] = f1$

by *auto*

lemma *rec-list-Cons-imp*: $f = \text{rec-list } f1 (\lambda x xs \text{ rec. } f2 x xs \text{ rec}) \Longrightarrow f (x \# xs) =$

f2 $x\ xs\ (f\ xs)$
by *auto*

lemma *not-arg-cong-Inr*: $x \neq y \implies \text{Inr } x \neq \text{Inr } y$
by *simp*

definition *image2p* **where**
 $\text{image2p } f\ g\ R = (\lambda x\ y. \exists x'\ y'. R\ x'\ y' \wedge f\ x' = x \wedge g\ y' = y)$

lemma *image2pI*: $R\ x\ y \implies \text{image2p } f\ g\ R\ (f\ x)\ (g\ y)$
unfolding *image2p-def* **by** *blast*

lemma *image2pE*: $\llbracket \text{image2p } f\ g\ R\ fx\ gy; (\bigwedge x\ y. fx = f\ x \implies gy = g\ y \implies R\ x\ y \implies P) \rrbracket \implies P$
unfolding *image2p-def* **by** *blast*

lemma *rel-fun-iff-geq-image2p*: $\text{rel-fun } R\ S\ f\ g = (\text{image2p } f\ g\ R \leq S)$
unfolding *rel-fun-def image2p-def* **by** *auto*

lemma *rel-fun-image2p*: $\text{rel-fun } R\ (\text{image2p } f\ g\ R)\ f\ g$
unfolding *rel-fun-def image2p-def* **by** *auto*

90.1 Equivalence relations, quotients, and Hilbert’s choice

lemma *equiv-Eps-in*:
 $\llbracket \text{equiv } A\ r; X \in A//r \rrbracket \implies \text{Eps } (\lambda x. x \in X) \in X$
apply (*rule someI2-ex*)
using *in-quotient-imp-non-empty* **by** *blast*

lemma *equiv-Eps-preserves*:
assumes *ECH*: *equiv* $A\ r$ **and** $X \in A//r$
shows $\text{Eps } (\lambda x. x \in X) \in A$
apply (*rule in-mono[rule-format]*)
using *assms* **apply** (*rule in-quotient-imp-subset*)
by (*rule equiv-Eps-in*) (*rule assms*)+

lemma *proj-Eps*:
assumes *equiv* $A\ r$ **and** $X \in A//r$
shows $\text{proj } r\ (\text{Eps } (\lambda x. x \in X)) = X$
unfolding *proj-def*
proof *auto*
fix x **assume** $x \in X$
thus $(\text{Eps } (\lambda x. x \in X), x) \in r$ **using** *assms equiv-Eps-in in-quotient-imp-in-rel*
by *fast*
next
fix x **assume** $(\text{Eps } (\lambda x. x \in X), x) \in r$
thus $x \in X$ **using** *in-quotient-imp-closed[OF assms equiv-Eps-in[OF assms]]* **by**
fast
qed

definition *univ* **where** $univ\ f\ X == f\ (Eps\ (\lambda x. x \in X))$

lemma *univ-commute*:

assumes *ECH*: *equiv* $A\ r$ **and** *RES*: f respects r **and** $x: x \in A$

shows $(univ\ f)\ (proj\ r\ x) = f\ x$

proof (*unfold univ-def*)

have *prj*: $proj\ r\ x \in A//r$ **using** x *proj-preserves* **by** *fast*

hence $Eps\ (\lambda y. y \in proj\ r\ x) \in A$ **using** *ECH equiv-Eps-preserves* **by** *fast*

moreover have $proj\ r\ (Eps\ (\lambda y. y \in proj\ r\ x)) = proj\ r\ x$ **using** *ECH prj proj-Eps* **by** *fast*

ultimately have $(x, Eps\ (\lambda y. y \in proj\ r\ x)) \in r$ **using** x *ECH proj-iff* **by** *fast*

thus $f\ (Eps\ (\lambda y. y \in proj\ r\ x)) = f\ x$ **using** *RES unfolding congruent-def* **by** *fastforce*

qed

lemma *univ-preserves*:

assumes *ECH*: *equiv* $A\ r$ **and** *RES*: f respects r **and** *PRES*: $\forall x \in A. f\ x \in B$

shows $\forall X \in A//r. univ\ f\ X \in B$

proof

fix X **assume** $X \in A//r$

then obtain x **where** $x: x \in A$ **and** $X: X = proj\ r\ x$ **using** *ECH proj-image*[*of r A*] **by** *blast*

hence $univ\ f\ X = f\ x$ **using** *ECH RES univ-commute* **by** *fastforce*

thus $univ\ f\ X \in B$ **using** x *PRES* **by** *simp*

qed

lemma *card-suc-ordLess-imp-ordLeq*:

assumes *ORD*: *Card-order* r *Card-order* r' *card-order* r'

and *LESS*: $r <_o\ card-suc\ r'$

shows $r \leq_o\ r'$

proof –

have *Card-order* $(card-suc\ r')$ **by** (*rule Card-order-card-suc*[*OF ORD*](β))

then have $cardSuc\ r \leq_o\ card-suc\ r'$ **using** *cardSuc-least ORD LESS* **by** *blast*

then have $cardSuc\ r \leq_o\ cardSuc\ r'$ **using** *cardSuc-ordIso-card-suc ordIso-symmetric ordLeq-ordIso-trans ORD*(β) **by** *blast*

then show *?thesis* **using** *cardSuc-mono-ordLeq ORD* **by** *blast*

qed

lemma *natLeq-ordLess-cinfinite*: $\llbracket Cinfinite\ r; card-order\ r \rrbracket \implies natLeq\ <_o\ card-suc\ r$

using *natLeq-ordLeq-cinfinite card-suc-greater ordLeq-ordLess-trans* **by** *blast*

corollary *natLeq-ordLess-cinfinite'*: $\llbracket Cinfinite\ r'; card-order\ r'; r \equiv card-suc\ r' \rrbracket \implies natLeq\ <_o\ r$

using *natLeq-ordLess-cinfinite* **by** *blast*

ML-file $\langle Tools/BNF/bnf-gfp-util.ML \rangle$

ML-file $\langle Tools/BNF/bnf-gfp-tactics.ML \rangle$

ML-file \langle Tools/BNF/bnf-gfp.ML \rangle
 ML-file \langle Tools/BNF/bnf-gfp-rec-sugar-tactics.ML \rangle
 ML-file \langle Tools/BNF/bnf-gfp-rec-sugar.ML \rangle

end

91 Filters on predicates

theory Filter
 imports Set-Interval Lifting-Set
 begin

91.1 Filters

This definition also allows non-proper filters.

locale *is-filter* =
 fixes $F :: ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 assumes *True*: $F (\lambda x. \text{True})$
 assumes *conj*: $F (\lambda x. P x) \Longrightarrow F (\lambda x. Q x) \Longrightarrow F (\lambda x. P x \wedge Q x)$
 assumes *mono*: $\forall x. P x \longrightarrow Q x \Longrightarrow F (\lambda x. P x) \Longrightarrow F (\lambda x. Q x)$

typedef 'a filter = $\{F :: ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}. \text{is-filter } F\}$
 proof
 show $(\lambda x. \text{True}) \in ?\text{filter}$ by (auto intro: *is-filter.intro*)
 qed

lemma *is-filter-Rep-filter*: *is-filter* (Rep-filter F)
 using Rep-filter [of F] by simp

lemma *Abs-filter-inverse'*:
 assumes *is-filter* F shows Rep-filter (Abs-filter F) = F
 using *assms* by (simp add: *Abs-filter-inverse*)

91.1.1 Eventually

definition *eventually* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ filter} \Rightarrow \text{bool}$
 where *eventually* $P F \longleftrightarrow \text{Rep-filter } F P$

syntax
 -*eventually* :: $p\text{trn} \Rightarrow 'a \text{ filter} \Rightarrow \text{bool} \Rightarrow \text{bool} \ ((\exists \forall_F \text{ - in -/ -}) [0, 0, 10]$
 10)

translations
 $\forall_F x \text{ in } F. P == \text{CONST eventually } (\lambda x. P) F$

lemma *eventually-Abs-filter*:
 assumes *is-filter* F shows *eventually* $P (\text{Abs-filter } F) = F P$
 unfolding *eventually-def* using *assms* by (simp add: *Abs-filter-inverse*)

lemma *filter-eq-iff*:

shows $F = F' \longleftrightarrow (\forall P. \text{eventually } P F = \text{eventually } P F')$

unfolding *Rep-filter-inject* [*symmetric*] *fun-eq-iff* *eventually-def* ..

lemma *eventually-True* [*simp*]: *eventually* $(\lambda x. \text{True}) F$

unfolding *eventually-def*

by (*rule is-filter.True* [*OF is-filter-Rep-filter*])

lemma *always-eventually*: $\forall x. P x \implies \text{eventually } P F$

proof –

assume $\forall x. P x$ **hence** $P = (\lambda x. \text{True})$ **by** (*simp add: ext*)

thus *eventually* $P F$ **by** *simp*

qed

lemma *eventuallyI*: $(\bigwedge x. P x) \implies \text{eventually } P F$

by (*auto intro: always-eventually*)

lemma *filter-eqI*: $(\bigwedge P. \text{eventually } P F \longleftrightarrow \text{eventually } P G) \implies F = G$

by (*auto simp: filter-eq-iff*)

lemma *eventually-mono*:

$\llbracket \text{eventually } P F; \bigwedge x. P x \implies Q x \rrbracket \implies \text{eventually } Q F$

unfolding *eventually-def*

by (*blast intro: is-filter.mono* [*OF is-filter-Rep-filter*])

lemma *eventually-conj*:

assumes $P: \text{eventually } (\lambda x. P x) F$

assumes $Q: \text{eventually } (\lambda x. Q x) F$

shows *eventually* $(\lambda x. P x \wedge Q x) F$

using *assms* **unfolding** *eventually-def*

by (*rule is-filter.conj* [*OF is-filter-Rep-filter*])

lemma *eventually-mp*:

assumes *eventually* $(\lambda x. P x \longrightarrow Q x) F$

assumes *eventually* $(\lambda x. P x) F$

shows *eventually* $(\lambda x. Q x) F$

proof –

have *eventually* $(\lambda x. (P x \longrightarrow Q x) \wedge P x) F$

using *assms* **by** (*rule eventually-conj*)

then show *?thesis*

by (*blast intro: eventually-mono*)

qed

lemma *eventually-rev-mp*:

assumes *eventually* $(\lambda x. P x) F$

assumes *eventually* $(\lambda x. P x \longrightarrow Q x) F$

shows *eventually* $(\lambda x. Q x) F$

using *assms(2)* *assms(1)* **by** (*rule eventually-mp*)

lemma eventually-conj-iff:
eventually $(\lambda x. P x \wedge Q x) F \longleftrightarrow$ *eventually* $P F \wedge$ *eventually* $Q F$
by (*auto intro: eventually-conj elim: eventually-rev-mp*)

lemma eventually-elim2:
assumes *eventually* $(\lambda i. P i) F$
assumes *eventually* $(\lambda i. Q i) F$
assumes $\bigwedge i. P i \implies Q i \implies R i$
shows *eventually* $(\lambda i. R i) F$
using *assms* **by** (*auto elim!: eventually-rev-mp*)

lemma eventually-cong:
assumes *eventually* $P F$ **and** $\bigwedge x. P x \implies Q x \longleftrightarrow R x$
shows *eventually* $Q F \longleftrightarrow$ *eventually* $R F$
using *assms* *eventually-elim2* **by** *blast*

lemma eventually-ball-finite-distrib:
 $finite A \implies$ (*eventually* $(\lambda x. \forall y \in A. P x y) net$) \longleftrightarrow $(\forall y \in A. \text{eventually } (\lambda x. P x y) net)$
by (*induction A rule: finite-induct*) (*auto simp: eventually-conj-iff*)

lemma eventually-ball-finite:
 $finite A \implies \forall y \in A. \text{eventually } (\lambda x. P x y) net \implies$ *eventually* $(\lambda x. \forall y \in A. P x y) net$
by (*auto simp: eventually-ball-finite-distrib*)

lemma eventually-all-finite:
fixes $P :: 'a \Rightarrow 'b::finite \Rightarrow bool$
assumes $\bigwedge y. \text{eventually } (\lambda x. P x y) net$
shows *eventually* $(\lambda x. \forall y. P x y) net$
using *eventually-ball-finite* [*of UNIV P*] *assms* **by** *simp*

lemma eventually-ex: $(\forall_{Fx \text{ in } F}. \exists y. P x y) \longleftrightarrow (\exists Y. \forall_{Fx \text{ in } F}. P x (Y x))$
proof

assume $\forall_{Fx \text{ in } F}. \exists y. P x y$
then have $\forall_{Fx \text{ in } F}. P x (SOME y. P x y)$
by (*auto intro: someI-ex eventually-mono*)
then show $\exists Y. \forall_{Fx \text{ in } F}. P x (Y x)$
by *auto*
qed (*auto intro: eventually-mono*)

lemma not-eventually-impI: *eventually* $P F \implies \neg$ *eventually* $Q F \implies \neg$ *eventually* $(\lambda x. P x \longrightarrow Q x) F$
by (*auto intro: eventually-mp*)

lemma not-eventuallyD: \neg *eventually* $P F \implies \exists x. \neg P x$
by (*metis always-eventually*)

lemma eventually-subst:

assumes *eventually* $(\lambda n. P\ n = Q\ n)\ F$
shows *eventually* $P\ F = \text{eventually}\ Q\ F$ (**is** $?L = ?R$)
proof –
from *assms* **have** *eventually* $(\lambda x. P\ x \longrightarrow Q\ x)\ F$
and *eventually* $(\lambda x. Q\ x \longrightarrow P\ x)\ F$
by (*auto elim: eventually-mono*)
then show *?thesis* **by** (*auto elim: eventually-elim2*)
qed

91.2 Frequently as dual to eventually

definition *frequently* $:: ('a \Rightarrow \text{bool}) \Rightarrow 'a\ \text{filter} \Rightarrow \text{bool}$
where *frequently* $P\ F \longleftrightarrow \neg\ \text{eventually}\ (\lambda x. \neg\ P\ x)\ F$

syntax

-frequently $:: \text{pttrn} \Rightarrow 'a\ \text{filter} \Rightarrow \text{bool} \Rightarrow \text{bool}\ ((\exists \exists_F\ -\ \text{in}\ -./\ -)\ [0, 0, 10]\ 10)$

translations

$\exists_F x\ \text{in}\ F. P == \text{CONST}\ \text{frequently}\ (\lambda x. P)\ F$

lemma *not-frequently-False* [*simp*]: $\neg\ (\exists_F x\ \text{in}\ F. \text{False})$
by (*simp add: frequently-def*)

lemma *frequently-ex*: $\exists_F x\ \text{in}\ F. P\ x \Longrightarrow \exists x. P\ x$
by (*auto simp: frequently-def dest: not-eventuallyD*)

lemma *frequentlyE*: **assumes** *frequently* $P\ F$ **obtains** x **where** $P\ x$
using *frequently-ex*[*OF assms*] **by** *auto*

lemma *frequently-mp*:

assumes *ev*: $\forall_F x\ \text{in}\ F. P\ x \longrightarrow Q\ x$ **and** P : $\exists_F x\ \text{in}\ F. P\ x$ **shows** $\exists_F x\ \text{in}\ F. Q\ x$

proof –

from *ev* **have** *eventually* $(\lambda x. \neg\ Q\ x \longrightarrow \neg\ P\ x)\ F$
by (*rule eventually-rev-mp*) (*auto intro!: always-eventually*)
from *eventually-mp*[*OF this*] P **show** *?thesis*
by (*auto simp: frequently-def*)

qed

lemma *frequently-rev-mp*:

assumes $\exists_F x\ \text{in}\ F. P\ x$
assumes $\forall_F x\ \text{in}\ F. P\ x \longrightarrow Q\ x$
shows $\exists_F x\ \text{in}\ F. Q\ x$

using *assms*(2) *assms*(1) **by** (*rule frequently-mp*)

lemma *frequently-mono*: $(\forall x. P\ x \longrightarrow Q\ x) \Longrightarrow \text{frequently}\ P\ F \Longrightarrow \text{frequently}\ Q\ F$

using *frequently-mp*[*of P Q*] **by** (*simp add: always-eventually*)

lemma *frequently-elim1*: $\exists_F x\ \text{in}\ F. P\ x \Longrightarrow (\bigwedge i. P\ i \Longrightarrow Q\ i) \Longrightarrow \exists_F x\ \text{in}\ F. Q$

x **by** (*metis frequently-mono*)**lemma** *frequently-disj-iff*: $(\exists_{F x \text{ in } F}. P x \vee Q x) \longleftrightarrow (\exists_{F x \text{ in } F}. P x) \vee (\exists_{F x \text{ in } F}. Q x)$ **by** (*simp add: frequently-def eventually-conj-iff*)**lemma** *frequently-disj*: $\exists_{F x \text{ in } F}. P x \implies \exists_{F x \text{ in } F}. Q x \implies \exists_{F x \text{ in } F}. P x \vee Q x$ **by** (*simp add: frequently-disj-iff*)**lemma** *frequently-bex-finite-distrib*:**assumes** *finite A* **shows** $(\exists_{F x \text{ in } F}. \exists y \in A. P x y) \longleftrightarrow (\exists y \in A. \exists_{F x \text{ in } F}. P x y)$ **using** *assms* **by** *induction (auto simp: frequently-disj-iff)***lemma** *frequently-bex-finite*: *finite A* $\implies \exists_{F x \text{ in } F}. \exists y \in A. P x y \implies \exists y \in A. \exists_{F x \text{ in } F}. P x y$ **by** (*simp add: frequently-bex-finite-distrib*)**lemma** *frequently-all*: $(\exists_{F x \text{ in } F}. \forall y. P x y) \longleftrightarrow (\forall Y. \exists_{F x \text{ in } F}. P x (Y x))$ **using** *eventually-ex[$\text{of } \lambda x y. \neg P x y F$]* **by** (*simp add: frequently-def*)**lemma****shows** *not-eventually*: $\neg \text{eventually } P F \longleftrightarrow (\exists_{F x \text{ in } F}. \neg P x)$ **and** *not-frequently*: $\neg \text{frequently } P F \longleftrightarrow (\forall_{F x \text{ in } F}. \neg P x)$ **by** (*auto simp: frequently-def*)**lemma** *frequently-imp-iff*: $(\exists_{F x \text{ in } F}. P x \longrightarrow Q x) \longleftrightarrow (\text{eventually } P F \longrightarrow \text{frequently } Q F)$ **unfolding** *imp-conv-disj frequently-disj-iff not-eventually[symmetric]* ..**lemma** *frequently-eventually-conj*:**assumes** $\exists_{F x \text{ in } F}. P x$ **assumes** $\forall_{F x \text{ in } F}. Q x$ **shows** $\exists_{F x \text{ in } F}. Q x \wedge P x$ **using** *assms eventually-elim2* **by** (*force simp add: frequently-def*)**lemma** *frequently-cong*:**assumes** *ev: eventually P F* **and** *QR: $\bigwedge x. P x \implies Q x \longleftrightarrow R x$* **shows** *frequently Q F* \longleftrightarrow *frequently R F***unfolding** *frequently-def***using** *QR* **by** (*auto intro!: eventually-cong [OF ev]*)**lemma** *frequently-eventually-frequently*:*frequently P F* \implies *eventually Q F* \implies *frequently ($\lambda x. P x \wedge Q x$) F***using** *frequently-cong [of Q F P $\lambda x. P x \wedge Q x$]* **by** *meson***lemma** *eventually-frequently-const-simps*:

$$\begin{aligned}
(\exists_{Fx \text{ in } F}. P x \wedge C) &\longleftrightarrow (\exists_{Fx \text{ in } F}. P x) \wedge C \\
(\exists_{Fx \text{ in } F}. C \wedge P x) &\longleftrightarrow C \wedge (\exists_{Fx \text{ in } F}. P x) \\
(\forall_{Fx \text{ in } F}. P x \vee C) &\longleftrightarrow (\forall_{Fx \text{ in } F}. P x) \vee C \\
(\forall_{Fx \text{ in } F}. C \vee P x) &\longleftrightarrow C \vee (\forall_{Fx \text{ in } F}. P x) \\
(\forall_{Fx \text{ in } F}. P x \longrightarrow C) &\longleftrightarrow ((\exists_{Fx \text{ in } F}. P x) \longrightarrow C) \\
(\forall_{Fx \text{ in } F}. C \longrightarrow P x) &\longleftrightarrow (C \longrightarrow (\forall_{Fx \text{ in } F}. P x))
\end{aligned}$$

by (cases C ; simp add: not-frequently)+

lemmas eventually-frequently-simps =
eventually-frequently-const-simps
not-eventually
eventually-conj-iff
eventually-ball-finite-distrib
eventually-ex
not-frequently
frequently-disj-iff
frequently-bex-finite-distrib
frequently-all
frequently-imp-iff

ML ‹

```

fun eventually-elim-tac facts =
  CONTEXT-SUBGOAL (fn (goal, i) => fn (ctxt, st) =>
    let
      val mp-facts = facts RL @ {thms eventually-rev-mp}
      val rule =
        @ {thm eventuallyI}
        |> fold (fn mp-fact => fn th => th RS mp-fact) mp-facts
        |> funpow (length facts) (fn th => @ {thm impI} RS th)
      val cases-prop =
        Thm.prop-of (Rule-Cases.internalize-params (rule RS Goal.init (Thm.cterm-of
          ctxt goal)))
      val cases = Rule-Cases.make-common ctxt cases-prop [((elim, []), [])]
    in CONTEXT-CASES cases (resolve-tac ctxt [rule] i) (ctxt, st) end
  )

```

method-setup eventually-elim = ‹

```

Scan.succeed (fn - => CONTEXT-METHOD (fn facts => eventually-elim-tac
  facts 1))
› elimination of eventually quantifiers

```

91.2.1 Finer-than relation

$F \leq F'$ means that filter F is finer than filter F' .

instantiation filter :: (type) complete-lattice
begin

definition le-filter-def:

$$F \leq F' \longleftrightarrow (\forall P. \text{eventually } P \ F' \longrightarrow \text{eventually } P \ F)$$

definition

$$(F :: 'a \text{ filter}) < F' \longleftrightarrow F \leq F' \wedge \neg F' \leq F$$
definition

$$\text{top} = \text{Abs-filter } (\lambda P. \forall x. P x)$$
definition

$$\text{bot} = \text{Abs-filter } (\lambda P. \text{True})$$
definition

$$\text{sup } F F' = \text{Abs-filter } (\lambda P. \text{eventually } P F \wedge \text{eventually } P F')$$
definition

$$\begin{aligned} \text{inf } F F' &= \text{Abs-filter} \\ &(\lambda P. \exists Q R. \text{eventually } Q F \wedge \text{eventually } R F' \wedge (\forall x. Q x \wedge R x \longrightarrow P x)) \end{aligned}$$
definition

$$\text{Sup } S = \text{Abs-filter } (\lambda P. \forall F \in S. \text{eventually } P F)$$
definition

$$\text{Inf } S = \text{Sup } \{F :: 'a \text{ filter}. \forall F' \in S. F \leq F'\}$$

lemma *eventually-top [simp]*: $\text{eventually } P \text{ top} \longleftrightarrow (\forall x. P x)$

unfolding *top-filter-def*

by (*rule eventually-Abs-filter, rule is-filter.intro, auto*)

lemma *eventually-bot [simp]*: $\text{eventually } P \text{ bot}$

unfolding *bot-filter-def*

by (*subst eventually-Abs-filter, rule is-filter.intro, auto*)

lemma *eventually-sup*:

$$\text{eventually } P (\text{sup } F F') \longleftrightarrow \text{eventually } P F \wedge \text{eventually } P F'$$

unfolding *sup-filter-def*

by (*rule eventually-Abs-filter, rule is-filter.intro*)

(*auto elim!: eventually-rev-mp*)

lemma *eventually-inf*:

$$\text{eventually } P (\text{inf } F F') \longleftrightarrow$$

$$(\exists Q R. \text{eventually } Q F \wedge \text{eventually } R F' \wedge (\forall x. Q x \wedge R x \longrightarrow P x))$$

unfolding *inf-filter-def*

apply (*rule eventually-Abs-filter [OF is-filter.intro]*)

apply (*blast intro: eventually-True*)

apply (*force elim!: eventually-conj*)⁺

done

lemma *eventually-Sup*:

$$\text{eventually } P (\text{Sup } S) \longleftrightarrow (\forall F \in S. \text{eventually } P F)$$

unfolding *Sup-filter-def*

```

apply (rule eventually-Abs-filter [OF is-filter.intro])
apply (auto intro: eventually-conj elim!: eventually-rev-mp)
done

```

instance proof

```

fix F F' F'' :: 'a filter and S :: 'a filter set
{ show  $F < F' \longleftrightarrow F \leq F' \wedge \neg F' \leq F$ 
  by (rule less-filter-def) }
{ show  $F \leq F$ 
  unfolding le-filter-def by simp }
{ assume  $F \leq F'$  and  $F' \leq F''$  thus  $F \leq F''$ 
  unfolding le-filter-def by simp }
{ assume  $F \leq F'$  and  $F' \leq F$  thus  $F = F'$ 
  unfolding le-filter-def filter-eq-iff by fast }
{ show  $\inf F F' \leq F$  and  $\inf F F' \leq F'$ 
  unfolding le-filter-def eventually-inf by (auto intro: eventually-True) }
{ assume  $F \leq F'$  and  $F \leq F''$  thus  $F \leq \inf F' F''$ 
  unfolding le-filter-def eventually-inf
  by (auto intro: eventually-mono [OF eventually-conj]) }
{ show  $F \leq \sup F F'$  and  $F' \leq \sup F F'$ 
  unfolding le-filter-def eventually-sup by simp-all }
{ assume  $F \leq F''$  and  $F' \leq F''$  thus  $\sup F F' \leq F''$ 
  unfolding le-filter-def eventually-sup by simp }
{ assume  $F'' \in S$  thus  $\inf S \leq F''$ 
  unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp }
{ assume  $\bigwedge F'. F' \in S \implies F \leq F'$  thus  $F \leq \inf S$ 
  unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp }
{ assume  $F \in S$  thus  $F \leq \sup S$ 
  unfolding le-filter-def eventually-Sup by simp }
{ assume  $\bigwedge F. F \in S \implies F \leq F'$  thus  $\sup S \leq F'$ 
  unfolding le-filter-def eventually-Sup by simp }
{ show  $\inf \{ \} = (\text{top}::'a \text{ filter})$ 
  by (auto simp: top-filter-def Inf-filter-def Sup-filter-def)
  (metis (full-types) top-filter-def always-eventually eventually-top) }
{ show  $\sup \{ \} = (\text{bot}::'a \text{ filter})$ 
  by (auto simp: bot-filter-def Sup-filter-def) }

```

qed

end

instance filter :: (type) distrib-lattice

proof

```

fix F G H :: 'a filter
show  $\sup F (\inf G H) = \inf (\sup F G) (\sup F H)$ 
proof (rule order.antisym)
  show  $\inf (\sup F G) (\sup F H) \leq \sup F (\inf G H)$ 
  unfolding le-filter-def eventually-sup
proof safe
  fix P assume 1: eventually P F and 2: eventually P (inf G H)

```

from 2 obtain $Q R$
 where QR : *eventually* $Q G$ *eventually* $R H \wedge x. Q x \implies R x \implies P x$
 by (*auto simp: eventually-inf*)
define Q' **where** $Q' = (\lambda x. Q x \vee P x)$
define R' **where** $R' = (\lambda x. R x \vee P x)$
from 1 have *eventually* $Q' F$
 by (*elim eventually-mono*) (*auto simp: Q'-def*)
moreover from 1 have *eventually* $R' F$
 by (*elim eventually-mono*) (*auto simp: R'-def*)
moreover from $QR(1)$ **have** *eventually* $Q' G$
 by (*elim eventually-mono*) (*auto simp: Q'-def*)
moreover from $QR(2)$ **have** *eventually* $R' H$
 by (*elim eventually-mono*) (*auto simp: R'-def*)
moreover from QR **have** $P x$ **if** $Q' x R' x$ **for** x
 using *that* **by** (*auto simp: Q'-def R'-def*)
ultimately show *eventually* P (*inf* (*sup* $F G$) (*sup* $F H$))
 by (*auto simp: eventually-inf eventually-sup*)
qed
qed (*auto intro: inf.coboundedI1 inf.coboundedI2*)
qed

lemma *filter-leD*:

$F \leq F' \implies \text{eventually } P F' \implies \text{eventually } P F$
unfolding *le-filter-def* **by** *simp*

lemma *filter-leI*:

$(\bigwedge P. \text{eventually } P F' \implies \text{eventually } P F) \implies F \leq F'$
unfolding *le-filter-def* **by** *simp*

lemma *eventually-False*:

$\text{eventually } (\lambda x. \text{False}) F \longleftrightarrow F = \text{bot}$
unfolding *filter-eq-iff* **by** (*auto elim: eventually-rev-mp*)

lemma *eventually-frequently*: $F \neq \text{bot} \implies \text{eventually } P F \implies \text{frequently } P F$

using *eventually-conj*[*of* $P F \lambda x. \neg P x$]
by (*auto simp add: frequently-def eventually-False*)

lemma *eventually-frequentlyE*:

assumes *eventually* $P F$
assumes *eventually* $(\lambda x. \neg P x \vee Q x) F F \neq \text{bot}$
shows *frequently* $Q F$

proof –

have *eventually* $Q F$
using *eventually-conj*[*OF* *assms*(1,2),*simplified*] **by** (*auto elim: eventually-mono*)
then show *?thesis* **using** *eventually-frequently*[*OF* $\langle F \neq \text{bot} \rangle$] **by** *auto*

qed

lemma *eventually-const-iff*: $\text{eventually } (\lambda x. P) F \longleftrightarrow P \vee F = \text{bot}$

by (*cases P*) (*auto simp: eventually-False*)

lemma *eventually-const[simp]*: $F \neq \text{bot} \implies \text{eventually } (\lambda x. P) F \longleftrightarrow P$
by (*simp add: eventually-const-iff*)

lemma *frequently-const-iff*: $\text{frequently } (\lambda x. P) F \longleftrightarrow P \wedge F \neq \text{bot}$
by (*simp add: frequently-def eventually-const-iff*)

lemma *frequently-const[simp]*: $F \neq \text{bot} \implies \text{frequently } (\lambda x. P) F \longleftrightarrow P$
by (*simp add: frequently-const-iff*)

lemma *eventually-happens*: $\text{eventually } P \text{ net} \implies \text{net} = \text{bot} \vee (\exists x. P x)$
by (*metis frequentlyE eventually-frequently*)

lemma *eventually-happens'*:
assumes $F \neq \text{bot}$ *eventually P F*
shows $\exists x. P x$
using *assms eventually-frequently frequentlyE* **by** *blast*

abbreviation (*input*) *trivial-limit* :: 'a filter \Rightarrow bool
where *trivial-limit* $F \equiv F = \text{bot}$

lemma *trivial-limit-def*: $\text{trivial-limit } F \longleftrightarrow \text{eventually } (\lambda x. \text{False}) F$
by (*rule eventually-False [symmetric]*)

lemma *False-imp-not-eventually*: $(\forall x. \neg P x) \implies \neg \text{trivial-limit } \text{net} \implies \neg \text{eventually } (\lambda x. P x) \text{ net}$
by (*simp add: eventually-False*)

lemma *trivial-limit-eventually*: $\text{trivial-limit } \text{net} \implies \text{eventually } P \text{ net}$
by *simp*

lemma *trivial-limit-eq*: $\text{trivial-limit } \text{net} \longleftrightarrow (\forall P. \text{eventually } P \text{ net})$
by (*simp add: filter-eq-iff*)

lemma *eventually-Inf*: $\text{eventually } P (\text{Inf } B) \longleftrightarrow (\exists X \subseteq B. \text{finite } X \wedge \text{eventually } P (\text{Inf } X))$

proof –

let $?F = \lambda P. \exists X \subseteq B. \text{finite } X \wedge \text{eventually } P (\text{Inf } X)$

have *eventually-F*: $\text{eventually } P (\text{Abs-filter } ?F) \longleftrightarrow ?F P$ **for** P

proof (*rule eventually-Abs-filter is-filter.intro*)+

show $?F (\lambda x. \text{True})$

by (*rule exI[of - {}]*) (*simp add: le-fun-def*)

next

fix $P Q$

assume $?F P ?F Q$

then obtain $X Y$ **where**

$X \subseteq B$ *finite X eventually P* ($\bigcap X$)

```

     $Y \subseteq B$  finite  $Y$  eventually  $Q$  ( $\prod Y$ ) by blast
then show  $?F$  ( $\lambda x. P x \wedge Q x$ )
    by (intro exI[of -  $X \cup Y$ ]) (auto simp: Inf-union-distrib eventually-inf)
next
fix  $P Q$ 
assume  $?F P$ 
then obtain  $X$  where  $X \subseteq B$  finite  $X$  eventually  $P$  ( $\prod X$ )
    by blast
moreover assume  $\forall x. P x \longrightarrow Q x$ 
ultimately show  $?F Q$ 
    by (intro exI[of -  $X$ ]) (auto elim: eventually-mono)
qed

have  $\text{Inf } B = \text{Abs-filter } ?F$ 
proof (intro antisym Inf-greatest)
    show  $\text{Inf } B \leq \text{Abs-filter } ?F$ 
    by (auto simp: le-filter-def eventually-F dest: Inf-superset-mono)
next
fix  $F$  assume  $F \in B$  then show  $\text{Abs-filter } ?F \leq F$ 
    by (auto simp add: le-filter-def eventually-F intro!: exI[of -  $\{F\}$ ])
qed
then show ?thesis
    by (simp add: eventually-F)
qed

lemma eventually-INF:  $\text{eventually } P$  ( $\prod b \in B. F b$ )  $\longleftrightarrow$  ( $\exists X \subseteq B. \text{finite } X \wedge \text{eventually } P$  ( $\prod b \in X. F b$ ))
    unfolding eventually-Inf [of  $P F B$ ]
    by (metis finite-imageI image-mono finite-subset-image)

lemma Inf-filter-not-bot:
fixes  $B :: 'a$  filter set
shows ( $\bigwedge X. X \subseteq B \implies \text{finite } X \implies \text{Inf } X \neq \text{bot}$ )  $\implies \text{Inf } B \neq \text{bot}$ 
unfolding trivial-limit-def eventually-Inf [of -  $B$ ]
    bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp

lemma INF-filter-not-bot:
fixes  $F :: 'i \Rightarrow 'a$  filter
shows ( $\bigwedge X. X \subseteq B \implies \text{finite } X \implies (\prod b \in X. F b) \neq \text{bot}$ )  $\implies (\prod b \in B. F b) \neq \text{bot}$ 
unfolding trivial-limit-def eventually-INF [of - -  $B$ ]
    bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp

lemma eventually-Inf-base:
assumes  $B \neq \{\}$  and base:  $\bigwedge F G. F \in B \implies G \in B \implies \exists x \in B. x \leq \text{inf } F G$ 
shows  $\text{eventually } P$  ( $\text{Inf } B$ )  $\longleftrightarrow$  ( $\exists b \in B. \text{eventually } P b$ )
proof (subst eventually-Inf, safe)
fix  $X$  assume  $\text{finite } X$   $X \subseteq B$ 
then have  $\exists b \in B. \forall x \in X. b \leq x$ 

```


proof *induct*
case *empty* **then show** *?case*
using $\langle B \neq \{\} \rangle$ **by** *auto*
next
case (*insert x X*)
then obtain *b* **where** $b \in B \wedge x. x \in X \implies b \leq x$
by *auto*
with $\langle \text{insert } x \ X \subseteq B \rangle$ *base[of b x]* **show** *?case*
by (*auto intro: order-trans*)
qed
then obtain *b* **where** $b \in B \ b \leq \text{Inf } X$
by (*auto simp: le-Inf-iff*)
then show *eventually P (Inf X) \implies Bex B (eventually P)*
by (*intro bexI[of - b] (auto simp: le-filter-def)*)
qed (*auto intro!: exI[of - {x} for x]*)

lemma *eventually-INF-base:*

$B \neq \{\} \implies (\bigwedge a \ b. a \in B \implies b \in B \implies \exists x \in B. F \ x \leq \text{inf } (F \ a) \ (F \ b)) \implies$
eventually P $(\bigcap b \in B. F \ b) \longleftrightarrow (\exists b \in B. \text{eventually } P \ (F \ b))$
by (*subst eventually-Inf-base*) *auto*

lemma *eventually-INF1: $i \in I \implies \text{eventually } P \ (F \ i) \implies \text{eventually } P \ (\bigcap i \in I. F \ i)$*

using *filter-leD[OF INF-lower]* .

lemma *eventually-INF-finite:*

assumes *finite A*

shows *eventually P* $(\bigcap x \in A. F \ x) \longleftrightarrow$

$(\exists Q. (\forall x \in A. \text{eventually } (Q \ x) \ (F \ x)) \wedge (\forall y. (\forall x \in A. Q \ x \ y) \longrightarrow P \ y))$

using *assms*

proof (*induction arbitrary: P rule: finite-induct*)

case (*insert a A P*)

from *insert.hyps* **have** [*simp*]: $x \neq a$ **if** $x \in A$ **for** *x*

using *that by auto*

have *eventually P* $(\bigcap x \in \text{insert } a \ A. F \ x) \longleftrightarrow$

$(\exists Q \ R \ S. \text{eventually } Q \ (F \ a) \wedge ((\forall x \in A. \text{eventually } (S \ x) \ (F \ x)) \wedge$
 $(\forall y. (\forall x \in A. S \ x \ y) \longrightarrow R \ y)) \wedge (\forall x. Q \ x \wedge R \ x \longrightarrow P \ x))$

unfolding *ex-simps by (simp add: eventually-inf insert.IH)*

also have $\dots \longleftrightarrow (\exists Q. (\forall x \in \text{insert } a \ A. \text{eventually } (Q \ x) \ (F \ x)) \wedge$
 $(\forall y. (\forall x \in \text{insert } a \ A. Q \ x \ y) \longrightarrow P \ y))$

proof (*safe, goal-cases*)

case (*1 Q R S*)

thus *?case using 1 by (intro exI[of - S(a := Q)]) auto*

next

case (*2 Q*)

show *?case*

by (*rule exI[of - Q a], rule exI[of - $\lambda y. \forall x \in A. Q \ x \ y$],*
rule exI[of - $Q(a := (\lambda-. \text{True}))$] (use 2 in auto))

qed

finally show *?case* .
qed *auto*

91.2.2 Map function for filters

definition *filtermap* :: ('a \Rightarrow 'b) \Rightarrow 'a filter \Rightarrow 'b filter
where *filtermap* f F = *Abs-filter* (λP . *eventually* (λx . P (f x)) F)

lemma *eventually-filtermap*:
eventually P (*filtermap* f F) = *eventually* (λx . P (f x)) F
unfolding *filtermap-def*
apply (*rule eventually-Abs-filter* [*OF is-filter.intro*])
apply (*auto elim!*: *eventually-rem-p*)
done

lemma *eventually-comp-filtermap*:
eventually (P \circ f) F \longleftrightarrow *eventually* P (*filtermap* f F)
unfolding *comp-def* **using** *eventually-filtermap* **by** *auto*

lemma *filtermap-compose*: *filtermap* (f \circ g) F = *filtermap* f (*filtermap* g F)
unfolding *filter-eq-iff* **by** (*simp add: eventually-filtermap*)

lemma *filtermap-ident*: *filtermap* (λx . x) F = F
by (*simp add: filter-eq-iff eventually-filtermap*)

lemma *filtermap-filtermap*:
filtermap f (*filtermap* g F) = *filtermap* (λx . f (g x)) F
by (*simp add: filter-eq-iff eventually-filtermap*)

lemma *filtermap-mono*: $F \leq F' \Longrightarrow$ *filtermap* f F \leq *filtermap* f F'
unfolding *le-filter-def* *eventually-filtermap* **by** *simp*

lemma *filtermap-bot* [*simp*]: *filtermap* f bot = bot
by (*simp add: filter-eq-iff eventually-filtermap*)

lemma *filtermap-bot-iff*: *filtermap* f F = bot \longleftrightarrow F = bot
by (*simp add: trivial-limit-def eventually-filtermap*)

lemma *filtermap-sup*: *filtermap* f (*sup* F1 F2) = *sup* (*filtermap* f F1) (*filtermap* f F2)
by (*simp add: filter-eq-iff eventually-filtermap eventually-sup*)

lemma *filtermap-SUP*: *filtermap* f ($\bigsqcup_{b \in B}$ F b) = ($\bigsqcup_{b \in B}$ *filtermap* f (F b))
by (*simp add: filter-eq-iff eventually-Sup eventually-filtermap*)

lemma *filtermap-inf*: *filtermap* f (*inf* F1 F2) \leq *inf* (*filtermap* f F1) (*filtermap* f F2)
by (*intro inf-greatest filtermap-mono inf-sup-ord*)

lemma *filtermap-INF*: $\text{filtermap } f (\prod b \in B. F b) \leq (\prod b \in B. \text{filtermap } f (F b))$
by (*rule INF-greatest, rule filtermap-mono, erule INF-lower*)

lemma *frequently-filtermap*:
 $\text{frequently } P (\text{filtermap } f F) = \text{frequently } (\lambda x. P (f x)) F$
by (*simp add: frequently-def eventually-filtermap*)

91.2.3 Contravariant map function for filters

definition *filtercomap* :: $('a \Rightarrow 'b) \Rightarrow 'b \text{ filter} \Rightarrow 'a \text{ filter}$ **where**
 $\text{filtercomap } f F = \text{Abs-filter } (\lambda P. \exists Q. \text{eventually } Q F \wedge (\forall x. Q (f x) \longrightarrow P x))$

lemma *eventually-filtercomap*:
 $\text{eventually } P (\text{filtercomap } f F) \longleftrightarrow (\exists Q. \text{eventually } Q F \wedge (\forall x. Q (f x) \longrightarrow P x))$

unfolding *filtercomap-def*

proof (*intro eventually-Abs-filter, unfold-locales, goal-cases*)

case 1

show ?case **by** (*auto intro!: exI[of - $\lambda\cdot$. True]*)

next

case (2 $P Q$)

then obtain $P' Q'$ **where** $P' Q'$:

$\text{eventually } P' F \forall x. P' (f x) \longrightarrow P x$

$\text{eventually } Q' F \forall x. Q' (f x) \longrightarrow Q x$

by (*elim exE conjE*)

show ?case

by (*rule exI[of - $\lambda x. P' x \wedge Q' x$] (use $P' Q'$ in $\langle \text{auto intro!}: \text{eventually-conj} \rangle$)*)

next

case (3 $P Q$)

thus ?case **by** *blast*

qed

lemma *filtercomap-ident*: $\text{filtercomap } (\lambda x. x) F = F$
by (*auto simp: filter-eq-iff eventually-filtercomap elim!: eventually-mono*)

lemma *filtercomap-filtercomap*: $\text{filtercomap } f (\text{filtercomap } g F) = \text{filtercomap } (\lambda x. g (f x)) F$

unfolding *filter-eq-iff* **by** (*auto simp: eventually-filtercomap*)

lemma *filtercomap-mono*: $F \leq F' \implies \text{filtercomap } f F \leq \text{filtercomap } f F'$
by (*auto simp: eventually-filtercomap le-filter-def*)

lemma *filtercomap-bot* [*simp*]: $\text{filtercomap } f \text{ bot} = \text{bot}$
by (*auto simp: filter-eq-iff eventually-filtercomap*)

lemma *filtercomap-top* [*simp*]: $\text{filtercomap } f \text{ top} = \text{top}$
by (*auto simp: filter-eq-iff eventually-filtercomap*)

lemma *filtercomap-inf*: $\text{filtercomap } f (\text{inf } F1 F2) = \text{inf } (\text{filtercomap } f F1) (\text{filtercomap } f F2)$

```

f F2)
  unfolding filter-eq-iff
proof safe
  fix P
  assume eventually P (filtercomap f (F1  $\sqcap$  F2))
  then obtain Q R S where *:
    eventually Q F1 eventually R F2  $\wedge$  x. Q x  $\implies$  R x  $\implies$  S x  $\wedge$  x. S (f x)  $\implies$  P
  x
    unfolding eventually-filtercomap eventually-inf by blast
  from * have eventually ( $\lambda$ x. Q (f x)) (filtercomap f F1)
    eventually ( $\lambda$ x. R (f x)) (filtercomap f F2)
    by (auto simp: eventually-filtercomap)
  with * show eventually P (filtercomap f F1  $\sqcap$  filtercomap f F2)
    unfolding eventually-inf by blast
next
  fix P
  assume eventually P (inf (filtercomap f F1) (filtercomap f F2))
  then obtain Q Q' R R' where *:
    eventually Q F1 eventually R F2  $\wedge$  x. Q (f x)  $\implies$  Q' x  $\wedge$  x. R (f x)  $\implies$  R' x
     $\wedge$  x. Q' x  $\implies$  R' x  $\implies$  P x
    unfolding eventually-filtercomap eventually-inf by blast
  from * have eventually ( $\lambda$ x. Q x  $\wedge$  R x) (F1  $\sqcap$  F2) by (auto simp: eventu-
ally-inf)
  with * show eventually P (filtercomap f (F1  $\sqcap$  F2))
    by (auto simp: eventually-filtercomap)
qed

lemma filtercomap-sup: filtercomap f (sup F1 F2)  $\geq$  sup (filtercomap f F1) (filtercomap
f F2)
  by (intro sup-least filtercomap-mono inf-sup-ord)

lemma filtercomap-INF: filtercomap f ( $\sqcap$  b $\in$ B. F b) = ( $\sqcap$  b $\in$ B. filtercomap f (F
b))
proof -
  have *: filtercomap f ( $\sqcap$  b $\in$ B. F b) = ( $\sqcap$  b $\in$ B. filtercomap f (F b)) if finite B for
  B
    using that by induction (simp-all add: filtercomap-inf)
  show ?thesis unfolding filter-eq-iff
  proof
    fix P
    have eventually P ( $\sqcap$  b $\in$ B. filtercomap f (F b))  $\iff$ 
      ( $\exists$  X. (X  $\subseteq$  B  $\wedge$  finite X)  $\wedge$  eventually P ( $\sqcap$  b $\in$ X. filtercomap f (F b)))
    by (subst eventually-INF) blast
    also have ...  $\iff$  ( $\exists$  X. (X  $\subseteq$  B  $\wedge$  finite X)  $\wedge$  eventually P (filtercomap f
( $\sqcap$  b $\in$ X. F b)))
    by (rule ex-cong) (simp add: *)
    also have ...  $\iff$  eventually P (filtercomap f ( $\sqcap$  (F ' B)))
    unfolding eventually-filtercomap by (subst eventually-INF) blast
    finally show eventually P (filtercomap f ( $\sqcap$  (F ' B))) =

```

eventually P ($\prod_{b \in B}. \text{filtercomap } f (F b)$) ..

qed
qed

lemma *filtercomap-SUP*:

filtercomap f ($\sqcup_{b \in B}. F b$) \geq ($\sqcup_{b \in B}. \text{filtercomap } f (F b)$)
by (*intro SUP-least filtercomap-mono SUP-upper*)

lemma *filtermap-le-iff-le-filtercomap*: *filtermap* f $F \leq G \iff F \leq \text{filtercomap } f G$

unfolding *le-filter-def eventually-filtermap eventually-filtercomap*
using *eventually-mono* **by** *auto*

lemma *filtercomap-neq-bot*:

assumes $\bigwedge P. \text{eventually } P F \implies \exists x. P (f x)$
shows *filtercomap* $f F \neq \text{bot}$
using *assms* **by** (*auto simp: trivial-limit-def eventually-filtercomap*)

lemma *filtercomap-neq-bot-surj*:

assumes $F \neq \text{bot}$ **and** *surj* f
shows *filtercomap* $f F \neq \text{bot}$
proof (*rule filtercomap-neq-bot*)
fix P **assume** $*$: *eventually* $P F$
show $\exists x. P (f x)$
proof (*rule ccontr*)
assume $**$: $\neg(\exists x. P (f x))$
from $*$ **have** *eventually* $(\lambda-. \text{False}) F$
proof *eventually-elim*
case (*elim* x)
from $\langle \text{surj } f \rangle$ **obtain** y **where** $x = f y$ **by** *auto*
with *elim* **and** $**$ **show** *False* **by** *auto*
qed
with *assms* **show** *False* **by** (*simp add: trivial-limit-def*)
qed
qed

lemma *eventually-filtercomapI* [*intro*]:

assumes *eventually* $P F$
shows *eventually* $(\lambda x. P (f x)) (\text{filtercomap } f F)$
using *assms* **by** (*auto simp: eventually-filtercomap*)

lemma *filtermap-filtercomap*: *filtermap* $f (\text{filtercomap } f F) \leq F$

by (*auto simp: le-filter-def eventually-filtermap eventually-filtercomap*)

lemma *filtercomap-filtermap*: *filtercomap* $f (\text{filtermap } f F) \geq F$

unfolding *le-filter-def eventually-filtermap eventually-filtercomap*
by (*auto elim!: eventually-mono*)

91.2.4 Standard filters

definition *principal* :: 'a set \Rightarrow 'a filter **where**

principal $S = \text{Abs-filter } (\lambda P. \forall x \in S. P x)$

lemma *eventually-principal*: *eventually* P (*principal* S) $\longleftrightarrow (\forall x \in S. P x)$

unfolding *principal-def*

by (*rule eventually-Abs-filter*, *rule is-filter.intro*) *auto*

lemma *eventually-inf-principal*: *eventually* P (*inf* F (*principal* s)) \longleftrightarrow *eventually* $(\lambda x. x \in s \longrightarrow P x)$ F

unfolding *eventually-inf eventually-principal* **by** (*auto elim: eventually-mono*)

lemma *principal-UNIV[simp]*: *principal* $UNIV = \text{top}$

by (*auto simp: filter-eq-iff eventually-principal*)

lemma *principal-empty[simp]*: *principal* $\{\} = \text{bot}$

by (*auto simp: filter-eq-iff eventually-principal*)

lemma *principal-eq-bot-iff*: *principal* $X = \text{bot} \longleftrightarrow X = \{\}$

by (*auto simp add: filter-eq-iff eventually-principal*)

lemma *principal-le-iff[iff]*: *principal* $A \leq$ *principal* $B \longleftrightarrow A \subseteq B$

by (*auto simp: le-filter-def eventually-principal*)

lemma *le-principal*: $F \leq$ *principal* $A \longleftrightarrow$ *eventually* $(\lambda x. x \in A)$ F

unfolding *le-filter-def eventually-principal*

by (*force elim: eventually-mono*)

lemma *principal-inject[iff]*: *principal* $A =$ *principal* $B \longleftrightarrow A = B$

unfolding *eq-iff* **by** *simp*

lemma *sup-principal[simp]*: *sup* (*principal* A) (*principal* B) = *principal* $(A \cup B)$

unfolding *filter-eq-iff eventually-sup eventually-principal* **by** *auto*

lemma *inf-principal[simp]*: *inf* (*principal* A) (*principal* B) = *principal* $(A \cap B)$

unfolding *filter-eq-iff eventually-inf eventually-principal*

by (*auto intro: exI[of - $\lambda x. x \in A$] exI[of - $\lambda x. x \in B$]*)

lemma *SUP-principal[simp]*: $(\bigsqcup i \in I. \text{principal } (A i)) = \text{principal } (\bigcup i \in I. A i)$

unfolding *filter-eq-iff eventually-Sup* **by** (*auto simp: eventually-principal*)

lemma *INF-principal-finite*: *finite* $X \implies (\prod x \in X. \text{principal } (f x)) = \text{principal } (\prod x \in X. f x)$

by (*induct X rule: finite-induct*) *auto*

lemma *filtermap-principal[simp]*: *filtermap* f (*principal* A) = *principal* $(f \text{ ` } A)$

unfolding *filter-eq-iff eventually-filtermap eventually-principal* **by** *simp*

lemma *filtercomap-principal[simp]*: *filtercomap* f (*principal* A) = *principal* $(f \text{ - ` } A)$

A)

unfolding *filter-eq-iff eventually-filtercomap eventually-principal by fast***91.2.5 Order filters****definition** *at-top* :: ('a::order) filter**where** *at-top* = ($\prod k$. *principal* {*k* ..})**lemma** *at-top-sub*: *at-top* = ($\prod k \in \{c :: 'a :: \text{linorder}..\}$. *principal* {*k* ..})**by** (*auto intro!*: *INF-eq max.cobounded1 max.cobounded2 simp: at-top-def*)**lemma** *eventually-at-top-linorder*: *eventually P at-top* \longleftrightarrow ($\exists N :: 'a :: \text{linorder}$. $\forall n \geq N$. *P n*)**unfolding** *at-top-def***by** (*subst eventually-INF-base*) (*auto simp: eventually-principal intro: max.cobounded1 max.cobounded2*)**lemma** *eventually-filtercomap-at-top-linorder*:*eventually P (filtercomap f at-top)* \longleftrightarrow ($\exists N :: 'a :: \text{linorder}$. $\forall x$. $f x \geq N \longrightarrow P x$)**by** (*auto simp: eventually-filtercomap eventually-at-top-linorder*)**lemma** *eventually-at-top-linorderI*:**fixes** *c* :: 'a :: linorder**assumes** $\bigwedge x$. $c \leq x \implies P x$ **shows** *eventually P at-top***using** *assms by* (*auto simp: eventually-at-top-linorder*)**lemma** *eventually-ge-at-top [simp]*:*eventually* (λx . (*c* :: - :: linorder) $\leq x$) *at-top***unfolding** *eventually-at-top-linorder by auto***lemma** *eventually-at-top-dense*: *eventually P at-top* \longleftrightarrow ($\exists N :: 'a :: \{\text{no-top, linorder}\}$. $\forall n > N$. *P n*)**proof** –**have** *eventually P* ($\prod k$. *principal* {*k* < ..}) \longleftrightarrow ($\exists N :: 'a$. $\forall n > N$. *P n*)**by** (*subst eventually-INF-base*) (*auto simp: eventually-principal intro: max.cobounded1 max.cobounded2*)**also have** ($\prod k$. *principal* {*k* :: 'a < ..}) = *at-top***unfolding** *at-top-def***by** (*intro INF-eq*) (*auto intro: less-imp-le simp: Ici-subset-Ioi-iff gt-ex*)**finally show** *?thesis* .**qed****lemma** *eventually-filtercomap-at-top-dense*:*eventually P (filtercomap f at-top)* \longleftrightarrow ($\exists N :: 'a :: \{\text{no-top, linorder}\}$. $\forall x$. $f x > N \longrightarrow P x$)**by** (*auto simp: eventually-filtercomap eventually-at-top-dense*)**lemma** *eventually-at-top-not-equal [simp]*: *eventually* ($\lambda x :: 'a :: \{\text{no-top, linorder}\}$. *x*)

$\neq c$) *at-top*

unfolding *eventually-at-top-dense* **by** *auto*

lemma *eventually-gt-at-top* [*simp*]: *eventually* $(\lambda x. (c:::\{no-top, linorder\}) < x)$
at-top

unfolding *eventually-at-top-dense* **by** *auto*

lemma *eventually-all-ge-at-top*:

assumes *eventually* P (*at-top* :: ('*a* :: *linorder*) *filter*)

shows *eventually* $(\lambda x. \forall y \geq x. P y)$ *at-top*

proof –

from *assms* **obtain** x **where** $\bigwedge y. y \geq x \implies P y$ **by** (*auto simp: eventually-at-top-linorder*)

hence $\forall z \geq y. P z$ **if** $y \geq x$ **for** y **using** *that* **by** *simp*

thus *?thesis* **by** (*auto simp: eventually-at-top-linorder*)

qed

definition *at-bot* :: ('*a*::*order*) *filter*

where *at-bot* = $(\prod k. \text{principal } \{.. k\})$

lemma *at-bot-sub*: *at-bot* = $(\prod k \in \{.. c::'a::linorder\}. \text{principal } \{.. k\})$

by (*auto intro!: INF-eq min.cobounded1 min.cobounded2 simp: at-bot-def*)

lemma *eventually-at-bot-linorder*:

fixes $P :: 'a::linorder \Rightarrow \text{bool}$ **shows** *eventually* P *at-bot* $\longleftrightarrow (\exists N. \forall n \leq N. P n)$

unfolding *at-bot-def*

by (*subst eventually-INF-base*) (*auto simp: eventually-principal intro: min.cobounded1 min.cobounded2*)

lemma *eventually-filtercomap-at-bot-linorder*:

eventually P (*filtercomap* f *at-bot*) $\longleftrightarrow (\exists N::'a::linorder. \forall x. f x \leq N \longrightarrow P x)$

by (*auto simp: eventually-filtercomap eventually-at-bot-linorder*)

lemma *eventually-le-at-bot* [*simp*]:

eventually $(\lambda x. x \leq (c:::linorder))$ *at-bot*

unfolding *eventually-at-bot-linorder* **by** *auto*

lemma *eventually-at-bot-dense*: *eventually* P *at-bot* $\longleftrightarrow (\exists N::'a::\{no-bot, linorder\}. \forall n < N. P n)$

proof –

have *eventually* P $(\prod k. \text{principal } \{.. < k\}) \longleftrightarrow (\exists N::'a. \forall n < N. P n)$

by (*subst eventually-INF-base*) (*auto simp: eventually-principal intro: min.cobounded1 min.cobounded2*)

also have $(\prod k. \text{principal } \{.. < k::'a\}) = \text{at-bot}$

unfolding *at-bot-def*

by (*intro INF-eq*) (*auto intro: less-imp-le simp: Iic-subset-Iio-iff lt-ex*)

finally show *?thesis* .

qed

lemma *eventually-filtercomap-at-bot-dense*:
eventually P (*filtercomap* f *at-bot*) \longleftrightarrow $(\exists N :: 'a :: \{no-bot, linorder\}. \forall x. f\ x < N \longrightarrow P\ x)$
by (*auto simp: eventually-filtercomap eventually-at-bot-dense*)

lemma *eventually-at-bot-not-equal* [*simp*]: *eventually* $(\lambda x :: 'a :: \{no-bot, linorder\}. x \neq c)$ *at-bot*
unfolding *eventually-at-bot-dense* **by** *auto*

lemma *eventually-gt-at-bot* [*simp*]:
eventually $(\lambda x. x < (c :: :: unbounded-dense-linorder))$ *at-bot*
unfolding *eventually-at-bot-dense* **by** *auto*

lemma *trivial-limit-at-bot-linorder* [*simp*]: \neg *trivial-limit* (*at-bot* :: ('a :: linorder) *filter*)
unfolding *trivial-limit-def*
by (*metis eventually-at-bot-linorder order-refl*)

lemma *trivial-limit-at-top-linorder* [*simp*]: \neg *trivial-limit* (*at-top* :: ('a :: linorder) *filter*)
unfolding *trivial-limit-def*
by (*metis eventually-at-top-linorder order-refl*)

91.3 Sequentially

abbreviation *sequentially* :: *nat filter*
where *sequentially* \equiv *at-top*

lemma *eventually-sequentially*:
eventually P *sequentially* \longleftrightarrow $(\exists N. \forall n \geq N. P\ n)$
by (*rule eventually-at-top-linorder*)

lemma *sequentially-bot* [*simp, intro*]: *sequentially* \neq *bot*
unfolding *filter-eq-iff eventually-sequentially* **by** *auto*

lemmas *trivial-limit-sequentially = sequentially-bot*

lemma *eventually-False-sequentially* [*simp*]:
 \neg *eventually* $(\lambda n. False)$ *sequentially*
by (*simp add: eventually-False*)

lemma *le-sequentially*:
 $F \leq$ *sequentially* \longleftrightarrow $(\forall N. \text{eventually } (\lambda n. N \leq n) F)$
by (*simp add: at-top-def le-INF-iff le-principal*)

lemma *eventually-sequentiallyI* [*intro?*]:
assumes $\bigwedge x. c \leq x \Longrightarrow P\ x$
shows *eventually* P *sequentially*
using *assms* **by** (*auto simp: eventually-sequentially*)

lemma *eventually-sequentially-Suc* [simp]: *eventually* ($\lambda i. P (Suc\ i)$) *sequentially*
 \longleftrightarrow *eventually* *P* *sequentially*

unfolding *eventually-sequentially* **by** (*metis Suc-le-D Suc-le-mono le-Suc-eq*)

lemma *eventually-sequentially-seg* [simp]: *eventually* ($\lambda n. P (n + k)$) *sequentially*
 \longleftrightarrow *eventually* *P* *sequentially*

using *eventually-sequentially-Suc*[of $\lambda n. P (n + k)$ **for** *k*] **by** (*induction k*) *auto*

lemma *filtermap-sequentially-ne-bot*: *filtermap f* *sequentially* \neq *bot*
by (*simp add: filtermap-bot-iff*)

91.4 Increasing finite subsets

definition *finite-subsets-at-top* **where**

finite-subsets-at-top *A* = ($\bigcap X \in \{X. \text{finite } X \wedge X \subseteq A\}. \text{principal } \{Y. \text{finite } Y \wedge X \subseteq Y \wedge Y \subseteq A\}$)

lemma *eventually-finite-subsets-at-top*:

eventually *P* (*finite-subsets-at-top* *A*) \longleftrightarrow

($\exists X. \text{finite } X \wedge X \subseteq A \wedge (\forall Y. \text{finite } Y \wedge X \subseteq Y \wedge Y \subseteq A \longrightarrow P\ Y)$)

unfolding *finite-subsets-at-top-def*

proof (*subst eventually-INF-base, goal-cases*)

show $\{X. \text{finite } X \wedge X \subseteq A\} \neq \{\}$ **by** *auto*

next

case (*2 B C*)

thus *?case* **by** (*intro beXI*[of *- B \cup C*]) *auto*

qed (*simp-all add: eventually-principal*)

lemma *eventually-finite-subsets-at-top-weakI* [*intro*]:

assumes $\bigwedge X. \text{finite } X \Longrightarrow X \subseteq A \Longrightarrow P\ X$

shows *eventually* *P* (*finite-subsets-at-top* *A*)

proof –

have *eventually* ($\lambda X. \text{finite } X \wedge X \subseteq A$) (*finite-subsets-at-top* *A*)

by (*auto simp: eventually-finite-subsets-at-top*)

thus *?thesis* **by** *eventually-elim* (*use assms in auto*)

qed

lemma *finite-subsets-at-top-neq-bot* [simp]: *finite-subsets-at-top* *A* \neq *bot*

proof –

have $\neg \text{eventually } (\lambda x. \text{False})$ (*finite-subsets-at-top* *A*)

by (*auto simp: eventually-finite-subsets-at-top*)

thus *?thesis* **by** *auto*

qed

lemma *filtermap-image-finite-subsets-at-top*:

assumes *inj-on f A*

shows *filtermap* ((\cdot) *f*) (*finite-subsets-at-top* *A*) = *finite-subsets-at-top* (*f* ‘ *A*)

unfolding *filter-eq-iff eventually-filtermap*

proof (*safe, goal-cases*)
case (1 P)
then obtain X **where** X : *finite* $X \subseteq A \wedge Y$. *finite* $Y \implies X \subseteq Y \implies Y \subseteq A \implies P$ ($f \text{ ' } Y$)
unfolding *eventually-finite-subsets-at-top* **by force**
show ?*case* **unfolding** *eventually-finite-subsets-at-top* *eventually-filtermap*
proof (*rule* $\text{exI}[of - f \text{ ' } X]$, *intro* conjI allI impI , *goal-cases*)
case (3 Y)
with *assms* **and** $X(1,2)$ **have** P ($f \text{ ' } (f \text{ - ' } Y \cap A)$) **using** $X(1,2)$
by (*intro* $X(3)$ *finite-vimage-IntI*) *auto*
also have $f \text{ ' } (f \text{ - ' } Y \cap A) = Y$ **using** *assms* 3 **by** *blast*
finally show ?*case* .
qed (*insert* *assms* $X(1,2)$, *auto* *intro!*: *finite-vimage-IntI*)
next
case (2 P)
then obtain X **where** X : *finite* $X \subseteq f \text{ ' } A \wedge Y$. *finite* $Y \implies X \subseteq Y \implies Y \subseteq f \text{ ' } A \implies P$ Y
unfolding *eventually-finite-subsets-at-top* **by force**
show ?*case* **unfolding** *eventually-finite-subsets-at-top* *eventually-filtermap*
proof (*rule* $\text{exI}[of - f \text{ - ' } X \cap A]$, *intro* conjI allI impI , *goal-cases*)
case (3 Y)
with $X(1,2)$ **and** *assms* **show** ?*case* **by** (*intro* $X(3)$) *force+*
qed (*insert* *assms* $X(1)$, *auto* *intro!*: *finite-vimage-IntI*)
qed

lemma *eventually-finite-subsets-at-top-finite*:
assumes *finite* A
shows *eventually* P (*finite-subsets-at-top* A) $\longleftrightarrow P$ A
unfolding *eventually-finite-subsets-at-top* **using** *assms* **by force**

lemma *finite-subsets-at-top-finite*: *finite* $A \implies$ *finite-subsets-at-top* $A =$ *principal* $\{A\}$
by (*auto simp: filter-eq-iff* *eventually-finite-subsets-at-top-finite* *eventually-principal*)

91.5 The cofinite filter

definition *cofinite* = *Abs-filter* (λP . *finite* $\{x. \neg P x\}$)

abbreviation *Inf-many* :: ($'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ (**binder** \exists_{∞} 10)
where *Inf-many* $P \equiv$ *frequently* P *cofinite*

abbreviation *Alm-all* :: ($'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ (**binder** \forall_{∞} 10)
where *Alm-all* $P \equiv$ *eventually* P *cofinite*

notation (*ASCII*)
Inf-many (**binder** *INFM* 10) **and**
Alm-all (**binder** *MOST* 10)

lemma *eventually-cofinite*: *eventually* P *cofinite* \longleftrightarrow *finite* $\{x. \neg P x\}$

unfolding *cofinite-def*
proof (*rule eventually-Abs-filter, rule is-filter.intro*)
fix $P Q :: 'a \Rightarrow \text{bool}$ **assume** $\text{finite } \{x. \neg P x\}$ $\text{finite } \{x. \neg Q x\}$
from *finite-UnI[OF this]* **show** $\text{finite } \{x. \neg (P x \wedge Q x)\}$
by (*rule rev-finite-subset*) *auto*
next
fix $P Q :: 'a \Rightarrow \text{bool}$ **assume** $P: \text{finite } \{x. \neg P x\}$ **and** $*$: $\forall x. P x \longrightarrow Q x$
from $*$ **show** $\text{finite } \{x. \neg Q x\}$
by (*intro finite-subset[OF - P]*) *auto*
qed *simp*

lemma *frequently-cofinite*: $\text{frequently } P \text{ cofinite} \longleftrightarrow \neg \text{finite } \{x. P x\}$
by (*simp add: frequently-def eventually-cofinite*)

lemma *cofinite-bot[simp]*: $\text{cofinite} = (\text{bot}::'a \text{ filter}) \longleftrightarrow \text{finite } (\text{UNIV} :: 'a \text{ set})$
unfolding *trivial-limit-def eventually-cofinite* **by** *simp*

lemma *cofinite-eq-sequentially*: $\text{cofinite} = \text{sequentially}$
unfolding *filter-eq-iff eventually-sequentially eventually-cofinite*
proof *safe*
fix $P :: \text{nat} \Rightarrow \text{bool}$ **assume** [*simp*]: $\text{finite } \{x. \neg P x\}$
show $\exists N. \forall n \geq N. P n$
proof *cases*
assume $\{x. \neg P x\} \neq \{\}$ **then show** *?thesis*
by (*intro exI[of - Suc (Max {x. \neg P x})]*) (*auto simp: Suc-le-eq*)
qed *auto*
next
fix $P :: \text{nat} \Rightarrow \text{bool}$ **and** $N :: \text{nat}$ **assume** $\forall n \geq N. P n$
then have $\{x. \neg P x\} \subseteq \{.. < N\}$
by (*auto simp: not-le*)
then show $\text{finite } \{x. \neg P x\}$
by (*blast intro: finite-subset*)
qed

91.5.1 Product of filters

definition *prod-filter* :: $'a \text{ filter} \Rightarrow 'b \text{ filter} \Rightarrow ('a \times 'b) \text{ filter}$ (**infixr** \times_F 80)
where

prod-filter $F G =$
 $(\bigcap (P, Q) \in \{(P, Q). \text{eventually } P F \wedge \text{eventually } Q G\}. \text{principal } \{(x, y). P x \wedge Q y\})$

lemma *eventually-prod-filter*: $\text{eventually } P (F \times_F G) \longleftrightarrow$
 $(\exists Pf Pg. \text{eventually } Pf F \wedge \text{eventually } Pg G \wedge (\forall x y. Pf x \longrightarrow Pg y \longrightarrow P (x, y)))$

unfolding *prod-filter-def*

proof (*subst eventually-INF-base, goal-cases*)

case 2

moreover have $\text{eventually } Pf F \Longrightarrow \text{eventually } Qf F \Longrightarrow \text{eventually } Pg G \Longrightarrow$

eventually $Qg \ G \implies$
 $\exists P \ Q. \text{eventually } P \ F \wedge \text{eventually } Q \ G \wedge$
 $\text{Collect } P \times \text{Collect } Q \subseteq \text{Collect } Pf \times \text{Collect } Pg \cap \text{Collect } Qf \times \text{Collect } Qg$
for $Pf \ Pg \ Qf \ Qg$
by (*intro conjI exI[of - inf Pf Qf] exI[of - inf Pg Qg]*)
(auto simp: inf-fun-def eventually-conj)
ultimately show *?case*
by *auto*
qed (*auto simp: eventually-principal intro: eventually-True*)

lemma *eventually-prod1*:
assumes $B \neq \text{bot}$
shows $(\forall_F (x, y) \text{ in } A \times_F B. P \ x) \longleftrightarrow (\forall_F x \text{ in } A. P \ x)$
unfolding *eventually-prod-filter*
proof *safe*
fix $R \ Q$
assume $*$: $\forall_F x \text{ in } A. R \ x \ \forall_F x \text{ in } B. Q \ x \ \forall x \ y. R \ x \longrightarrow Q \ y \longrightarrow P \ x$
with $\langle B \neq \text{bot} \rangle$ **obtain** y **where** $Q \ y$ **by** (*auto dest: eventually-happens*)
with $*$ **show** *eventually* $P \ A$
by (*force elim: eventually-mono*)
next
assume *eventually* $P \ A$
then show $\exists Pf \ Pg. \text{eventually } Pf \ A \wedge \text{eventually } Pg \ B \wedge (\forall x \ y. Pf \ x \longrightarrow Pg \ y \longrightarrow P \ x)$
by (*intro exI[of - P] exI[of - $\lambda x. \text{True}$]*) *auto*
qed

lemma *eventually-prod2*:
assumes $A \neq \text{bot}$
shows $(\forall_F (x, y) \text{ in } A \times_F B. P \ y) \longleftrightarrow (\forall_F y \text{ in } B. P \ y)$
unfolding *eventually-prod-filter*
proof *safe*
fix $R \ Q$
assume $*$: $\forall_F x \text{ in } A. R \ x \ \forall_F x \text{ in } B. Q \ x \ \forall x \ y. R \ x \longrightarrow Q \ y \longrightarrow P \ y$
with $\langle A \neq \text{bot} \rangle$ **obtain** x **where** $R \ x$ **by** (*auto dest: eventually-happens*)
with $*$ **show** *eventually* $P \ B$
by (*force elim: eventually-mono*)
next
assume *eventually* $P \ B$
then show $\exists Pf \ Pg. \text{eventually } Pf \ A \wedge \text{eventually } Pg \ B \wedge (\forall x \ y. Pf \ x \longrightarrow Pg \ y \longrightarrow P \ y)$
by (*intro exI[of - P] exI[of - $\lambda x. \text{True}$]*) *auto*
qed

lemma *INF-filter-bot-base*:
fixes $F :: 'a \Rightarrow 'b \text{ filter}$
assumes $*$: $\bigwedge i \ j. i \in I \implies j \in I \implies \exists k \in I. F \ k \leq F \ i \sqcap F \ j$
shows $(\bigcap i \in I. F \ i) = \text{bot} \longleftrightarrow (\exists i \in I. F \ i = \text{bot})$
proof (*cases* $\exists i \in I. F \ i = \text{bot}$)

```

case True
then have  $(\prod_{i \in I}. F i) \leq \text{bot}$ 
  by (auto intro: INF-lower2)
with True show ?thesis
  by (auto simp: bot-unique)
next
case False
moreover have  $(\prod_{i \in I}. F i) \neq \text{bot}$ 
proof (cases  $I = \{\}$ )
  case True
  then show ?thesis
    by (auto simp add: filter-eq-iff)
  next
  case False' : False
  show ?thesis
  proof (rule INF-filter-not-bot)
    fix J
    assume finite J J  $\subseteq$  I
    then have  $\exists k \in I. F k \leq (\prod_{i \in J}. F i)$ 
    proof (induct J)
      case empty
      then show ?case
        using  $\langle I \neq \{\} \rangle$  by auto
    next
      case (insert i J)
      then obtain k where  $k \in I$   $F k \leq (\prod_{i \in J}. F i)$  by auto
      with insert *[of i k] show ?case
        by auto
    qed
  with False show  $(\prod_{i \in J}. F i) \neq \perp$ 
    by (auto simp: bot-unique)
  qed
qed
ultimately show ?thesis
  by auto
qed

lemma Collect-empty-eq-bot:  $\text{Collect } P = \{\} \longleftrightarrow P = \perp$ 
  by auto

lemma prod-filter-eq-bot:  $A \times_F B = \text{bot} \longleftrightarrow A = \text{bot} \vee B = \text{bot}$ 
  unfolding trivial-limit-def
proof
  assume  $\forall_F x$  in  $A \times_F B. \text{False}$ 
  then obtain Pf Pg
    where Pf: eventually  $(\lambda x. Pf x)$  A and Pg: eventually  $(\lambda y. Pg y)$  B
    and *:  $\forall x y. Pf x \longrightarrow Pg y \longrightarrow \text{False}$ 
  unfolding eventually-prod-filter by fast
  from * have  $(\forall x. \neg Pf x) \vee (\forall y. \neg Pg y)$  by fast

```

with $Pf\ Pg$ **show** $(\forall_F x \text{ in } A. \text{False}) \vee (\forall_F x \text{ in } B. \text{False})$ **by** *auto*
next
assume $(\forall_F x \text{ in } A. \text{False}) \vee (\forall_F x \text{ in } B. \text{False})$
then show $\forall_F x \text{ in } A \times_F B. \text{False}$
unfolding *eventually-prod-filter* **by** (*force intro: eventually-True*)
qed

lemma *prod-filter-mono*: $F \leq F' \implies G \leq G' \implies F \times_F G \leq F' \times_F G'$
by (*auto simp: le-filter-def eventually-prod-filter*)

lemma *prod-filter-mono-iff*:
assumes $nAB: A \neq \text{bot } B \neq \text{bot}$
shows $A \times_F B \leq C \times_F D \longleftrightarrow A \leq C \wedge B \leq D$
proof *safe*
assume $*$: $A \times_F B \leq C \times_F D$
with *assms* **have** $A \times_F B \neq \text{bot}$
by (*auto simp: bot-unique prod-filter-eq-bot*)
with $*$ **have** $C \times_F D \neq \text{bot}$
by (*auto simp: bot-unique*)
then have $nCD: C \neq \text{bot } D \neq \text{bot}$
by (*auto simp: prod-filter-eq-bot*)

show $A \leq C$
proof (*rule filter-leI*)
fix P **assume** *eventually* $P\ C$ **with** $*$ [*THEN filter-leD, of* $\lambda(x, y). P\ x$] **show**
eventually $P\ A$
using $nAB\ nCD$ **by** (*simp add: eventually-prod1 eventually-prod2*)
qed

show $B \leq D$
proof (*rule filter-leI*)
fix P **assume** *eventually* $P\ D$ **with** $*$ [*THEN filter-leD, of* $\lambda(x, y). P\ y$] **show**
eventually $P\ B$
using $nAB\ nCD$ **by** (*simp add: eventually-prod1 eventually-prod2*)
qed
qed (*intro prod-filter-mono*)

lemma *eventually-prod-same*: $\text{eventually } P\ (F \times_F F) \longleftrightarrow$
 $(\exists Q. \text{eventually } Q\ F \wedge (\forall x\ y. Q\ x \longrightarrow Q\ y \longrightarrow P\ (x, y)))$
unfolding *eventually-prod-filter* **by** (*blast intro!: eventually-conj*)

lemma *eventually-prod-sequentially*:
 $\text{eventually } P\ (\text{sequentially } \times_F \text{ sequentially}) \longleftrightarrow (\exists N. \forall m \geq N. \forall n \geq N. P\ (n, m))$
unfolding *eventually-prod-same eventually-sequentially* **by** *auto*

lemma *principal-prod-principal*: $\text{principal } A \times_F \text{principal } B = \text{principal } (A \times B)$
unfolding *filter-eq-iff eventually-prod-filter eventually-principal*
by (*fast intro: exI[of - $\lambda x. x \in A$] exI[of - $\lambda x. x \in B$]*)

lemma *le-prod-filterI*:

filtermap fst F ≤ A ⇒ filtermap snd F ≤ B ⇒ F ≤ A ×_F B
unfolding *le-filter-def eventually-filtermap eventually-prod-filter*
by (*force elim: eventually-elim2*)

lemma *filtermap-fst-prod-filter*: *filtermap fst (A ×_F B) ≤ A*

unfolding *le-filter-def eventually-filtermap eventually-prod-filter*
by (*force intro: eventually-True*)

lemma *filtermap-snd-prod-filter*: *filtermap snd (A ×_F B) ≤ B*

unfolding *le-filter-def eventually-filtermap eventually-prod-filter*
by (*force intro: eventually-True*)

lemma *prod-filter-INF*:

assumes *I ≠ {} and J ≠ {}*

shows $(\prod_{i \in I}. A\ i) \times_F (\prod_{j \in J}. B\ j) = (\prod_{i \in I}. \prod_{j \in J}. A\ i \times_F B\ j)$

proof (*rule antisym*)

from $\langle I \neq \{\} \rangle$ **obtain** *i where i ∈ I by auto*

from $\langle J \neq \{\} \rangle$ **obtain** *j where j ∈ J by auto*

show $(\prod_{i \in I}. \prod_{j \in J}. A\ i \times_F B\ j) \leq (\prod_{i \in I}. A\ i) \times_F (\prod_{j \in J}. B\ j)$

by (*fast intro: le-prod-filterI INF-greatest INF-lower2*

order-trans[OF filtermap-INF] ⟨i ∈ I⟩ ⟨j ∈ J⟩

filtermap-fst-prod-filter filtermap-snd-prod-filter)

show $(\prod_{i \in I}. A\ i) \times_F (\prod_{j \in J}. B\ j) \leq (\prod_{i \in I}. \prod_{j \in J}. A\ i \times_F B\ j)$

by (*intro INF-greatest prod-filter-mono INF-lower*)

qed

lemma *filtermap-Pair*: *filtermap (λx. (f x, g x)) F ≤ filtermap f F ×_F filtermap g F*

by (*rule le-prod-filterI, simp-all add: filtermap-filtermap*)

lemma *eventually-prodI*: *eventually P F ⇒ eventually Q G ⇒ eventually (λx. P (fst x) ∧ Q (snd x)) (F ×_F G)*

unfolding *eventually-prod-filter by auto*

lemma *prod-filter-INF1*: *I ≠ {} ⇒ (∏_{i ∈ I}. A i) ×_F B = (∏_{i ∈ I}. A i ×_F B)*

using *prod-filter-INF[of I {B} A λx. x]* **by** *simp*

lemma *prod-filter-INF2*: *J ≠ {} ⇒ A ×_F (∏_{i ∈ J}. B i) = (∏_{i ∈ J}. A ×_F B i)*

using *prod-filter-INF[of {A} J λx. x B]* **by** *simp*

lemma *prod-filtermap1*: *prod-filter (filtermap f F) G = filtermap (apfst f) (prod-filter F G)*

unfolding *filter-eq-iff eventually-filtermap eventually-prod-filter*

apply *safe*

subgoal **by** *auto*

subgoal **for** *P Q R by(rule exI[where x=λy. ∃x. y = f x ∧ Q x])(auto intro:*

eventually-mono)
done

lemma *prod-filtermap2*: *prod-filter* F (*filtermap* g G) = *filtermap* (*apsnd* g) (*prod-filter* F G)

unfolding *filter-eq-iff eventually-filtermap eventually-prod-filter*
apply *safe*
subgoal **by** *auto*
subgoal **for** P Q R **by**(*auto* *intro*: *exI*[**where** $x=\lambda y. \exists x. y = g\ x \wedge R\ x$]
eventually-mono)
done

lemma *prod-filter-assoc*:

prod-filter (*prod-filter* F G) H = *filtermap* ($\lambda(x, y, z). ((x, y), z)$) (*prod-filter* F (*prod-filter* G H))

apply(*clarsimp simp add: filter-eq-iff eventually-filtermap eventually-prod-filter; safe*)
subgoal **for** P Q R S T **by**(*auto* 4 4 *intro*: *exI*[**where** $x=\lambda(a, b). T\ a \wedge S\ b$])
subgoal **for** P Q R S T **by**(*auto* 4 3 *intro*: *exI*[**where** $x=\lambda(a, b). Q\ a \wedge S\ b$])
done

lemma *prod-filter-principal-singleton*: *prod-filter* (*principal* $\{x\}$) F = *filtermap* (*Pair* x) F

by(*fastforce simp add: filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim: eventually-mono intro: exI*[**where** $x=\lambda a. a = x$])

lemma *prod-filter-principal-singleton2*: *prod-filter* F (*principal* $\{x\}$) = *filtermap* ($\lambda a. (a, x)$) F

by(*fastforce simp add: filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim: eventually-mono intro: exI*[**where** $x=\lambda a. a = x$])

lemma *prod-filter-commute*: *prod-filter* F G = *filtermap* *prod.swap* (*prod-filter* G F)

by(*auto simp add: filter-eq-iff eventually-prod-filter eventually-filtermap*)

91.6 Limits

definition *filterlim* :: ($'a \Rightarrow 'b$) \Rightarrow $'b$ *filter* \Rightarrow $'a$ *filter* \Rightarrow *bool* **where**
filterlim f $F2$ $F1$ \longleftrightarrow *filtermap* f $F1$ \leq $F2$

syntax

-LIM :: $p\ ttrns \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow \text{bool}$ ($(\exists LIM\ (-)/\ (-)/\ (-) :> (-)$) [1000, 10, 0, 10] 10)

translations

LIM x $F1. f :> F2$ == *CONST* *filterlim* ($\lambda x. f$) $F2$ $F1$

lemma *filterlim-filtercomapI*: *filterlim* f G \Longrightarrow *filterlim* ($\lambda x. f$ ($g\ x$)) F (*filtercomap* g G)

unfolding *filterlim-def*
by (*metis order-trans filtermap-filtercomap filtermap-filtermap filtermap-mono*)

lemma *filterlim-top* [*simp*]: *filterlim f top F*
by (*simp add: filterlim-def*)

lemma *filterlim-iff*:
 $(LIM\ x\ F1.\ f\ x\ :\>\ F2) \longleftrightarrow (\forall P.\ eventually\ P\ F2 \longrightarrow eventually\ (\lambda x.\ P\ (f\ x))\ F1)$

unfolding *filterlim-def le-filter-def eventually-filtermap ..*

lemma *filterlim-compose*:
 $filterlim\ g\ F3\ F2 \implies filterlim\ f\ F2\ F1 \implies filterlim\ (\lambda x.\ g\ (f\ x))\ F3\ F1$
unfolding *filterlim-def filtermap-filtermap[symmetric]* **by** (*metis filtermap-mono order-trans*)

lemma *filterlim-mono*:
 $filterlim\ f\ F2\ F1 \implies F2 \leq F2' \implies F1' \leq F1 \implies filterlim\ f\ F2'\ F1'$
unfolding *filterlim-def* **by** (*metis filtermap-mono order-trans*)

lemma *filterlim-ident*: $LIM\ x\ F.\ x\ :\>\ F$
by (*simp add: filterlim-def filtermap-ident*)

lemma *filterlim-cong*:
 $F1 = F1' \implies F2 = F2' \implies eventually\ (\lambda x.\ f\ x = g\ x)\ F2 \implies filterlim\ f\ F1\ F2 = filterlim\ g\ F1'\ F2'$
by (*auto simp: filterlim-def le-filter-def eventually-filtermap elim: eventually-elim2*)

lemma *filterlim-mono-eventually*:
assumes *filterlim f F G and ord: F ≤ F' G' ≤ G*
assumes *eq: eventually (λx. f x = f' x) G'*
shows *filterlim f' F' G'*
proof –
have *filterlim f F' G'*
by (*simp add: filterlim-mono[OF - ord] assms*)
then show *?thesis*
by (*rule filterlim-cong[OF refl refl eq, THEN iffD1]*)

qed

lemma *filtermap-mono-strong*: $inj\ f \implies filtermap\ f\ F \leq filtermap\ f\ G \longleftrightarrow F \leq G$
apply (*safe intro!: filtermap-mono*)
apply (*auto simp: le-filter-def eventually-filtermap*)
apply (*erule-tac x=λx. P (inv f x) in allE*)
apply *auto*
done

lemma *eventually-compose-filterlim*:
assumes *eventually P F filterlim f F G*

shows *eventually* $(\lambda x. P (f x)) G$
using *assms* **by** (*simp* *add*: *filterlim-iff*)

lemma *filtermap-eq-strong*: $\text{inj } f \implies \text{filtermap } f F = \text{filtermap } f G \longleftrightarrow F = G$
by (*simp* *add*: *filtermap-mono-strong eq-iff*)

lemma *filtermap-fun-inverse*:

assumes *g*: *filterlim* *g* *F* *G*

assumes *f*: *filterlim* *f* *G* *F*

assumes *ev*: *eventually* $(\lambda x. f (g x) = x) G$

shows $\text{filtermap } f F = G$

proof (*rule antisym*)

show $\text{filtermap } f F \leq G$

using *f* **unfolding** *filterlim-def* .

have $G = \text{filtermap } f (\text{filtermap } g G)$

using *ev* **by** (*auto elim*: *eventually-elim2 simp*: *filter-eq-iff eventually-filtermap*)

also have $\dots \leq \text{filtermap } f F$

using *g* **by** (*intro filtermap-mono*) (*simp* *add*: *filterlim-def*)

finally show $G \leq \text{filtermap } f F$.

qed

lemma *filterlim-principal*:

$(\text{LIM } x F. f x \text{ :> principal } S) \longleftrightarrow (\text{eventually } (\lambda x. f x \in S) F)$

unfolding *filterlim-def eventually-filtermap le-principal* ..

lemma *filterlim-filtercomap* [*intro*]: $\text{filterlim } f F (\text{filtercomap } f F)$

unfolding *filterlim-def* **by** (*rule filtermap-filtercomap*)

lemma *filterlim-inf*:

$(\text{LIM } x F1. f x \text{ :> inf } F2 F3) \longleftrightarrow ((\text{LIM } x F1. f x \text{ :> } F2) \wedge (\text{LIM } x F1. f x \text{ :> } F3))$

unfolding *filterlim-def* **by** *simp*

lemma *filterlim-INF*:

$(\text{LIM } x F. f x \text{ :> } (\bigcap b \in B. G b)) \longleftrightarrow (\forall b \in B. \text{LIM } x F. f x \text{ :> } G b)$

unfolding *filterlim-def le-INF-iff* ..

lemma *filterlim-INF-INF*:

$(\bigwedge m. m \in J \implies \exists i \in I. \text{filtermap } f (F i) \leq G m) \implies \text{LIM } x (\bigcap i \in I. F i). f x \text{ :> } (\bigcap j \in J. G j)$

unfolding *filterlim-def* **by** (*rule order-trans*[*OF filtermap-INF INF-mono*])

lemma *filterlim-INF'*: $x \in A \implies \text{filterlim } f F (G x) \implies \text{filterlim } f F (\bigcap x \in A. G x)$

unfolding *filterlim-def* **by** (*rule order.trans*[*OF filtermap-mono*[*OF INF-lower*]])

lemma *filterlim-filtercomap-iff*: $\text{filterlim } f (\text{filtercomap } g G) F \longleftrightarrow \text{filterlim } (g \circ f) G F$

by (*simp* *add*: *filterlim-def filtermap-le-iff-le-filtercomap filtercomap-filtercomap*)

o-def)

lemma *filterlim-iff-le-filtercomap*: $\text{filterlim } f \ F \ G \longleftrightarrow G \leq \text{filtercomap } f \ F$
by (*simp add: filterlim-def filtermap-le-iff-le-filtercomap*)

lemma *filterlim-base*:

$(\bigwedge m \ x. m \in J \implies i \ m \in I) \implies (\bigwedge m \ x. m \in J \implies x \in F \ (i \ m) \implies f \ x \in G \ m) \implies$
 $LIM \ x \ (\bigcap_{i \in I. \text{principal } (F \ i)}. f \ x \ :> (\bigcap_{j \in J. \text{principal } (G \ j)})$
by (*force intro!: filterlim-INF-INF simp: image-subset-iff*)

lemma *filterlim-base-iff*:

assumes $I \neq \{\}$ **and** *chain*: $\bigwedge i \ j. i \in I \implies j \in I \implies F \ i \subseteq F \ j \vee F \ j \subseteq F \ i$
shows $(LIM \ x \ (\bigcap_{i \in I. \text{principal } (F \ i)}. f \ x \ :> \bigcap_{j \in J. \text{principal } (G \ j)}) \longleftrightarrow$
 $(\forall j \in J. \exists i \in I. \forall x \in F \ i. f \ x \in G \ j)$
unfolding *filterlim-INF filterlim-principal*
proof (*subst eventually-INF-base*)
fix $i \ j$ **assume** $i \in I \ j \in I$
with *chain[OF this]* **show** $\exists x \in I. \text{principal } (F \ x) \leq \text{inf } (\text{principal } (F \ i)) \ (\text{principal } (F \ j))$
by *auto*
qed (*auto simp: eventually-principal <I ≠ {}>*)

lemma *filterlim-filtermap*: $\text{filterlim } f \ F1 \ (\text{filtermap } g \ F2) = \text{filterlim } (\lambda x. f \ (g \ x)) \ F1 \ F2$
unfolding *filterlim-def filtermap-filtermap ..*

lemma *filterlim-sup*:

$\text{filterlim } f \ F \ F1 \implies \text{filterlim } f \ F \ F2 \implies \text{filterlim } f \ F \ (\text{sup } F1 \ F2)$
unfolding *filterlim-def filtermap-sup* **by** *auto*

lemma *filterlim-sequentially-Suc*:

$(LIM \ x \ \text{sequentially. } f \ (\text{Suc } x) \ :> F) \longleftrightarrow (LIM \ x \ \text{sequentially. } f \ x \ :> F)$
unfolding *filterlim-iff* **by** (*subst eventually-sequentially-Suc*) *simp*

lemma *filterlim-Suc*: *filterlim Suc sequentially sequentially*

by (*simp add: filterlim-iff eventually-sequentially*)

lemma *filterlim-If*:

$LIM \ x \ \text{inf } F \ (\text{principal } \{x. P \ x\}). f \ x \ :> G \implies$
 $LIM \ x \ \text{inf } F \ (\text{principal } \{x. \neg P \ x\}). g \ x \ :> G \implies$
 $LIM \ x \ F. \ \text{if } P \ x \ \text{then } f \ x \ \text{else } g \ x \ :> G$
unfolding *filterlim-iff eventually-inf-principal* **by** (*auto simp: eventually-conj-iff*)

lemma *filterlim-Pair*:

$LIM \ x \ F. f \ x \ :> G \implies LIM \ x \ F. g \ x \ :> H \implies LIM \ x \ F. (f \ x, g \ x) \ :> G \times_F H$
unfolding *filterlim-def*
by (*rule order-trans[OF filtermap-Pair prod-filter-mono]*)

91.7 Limits to *at-top* and *at-bot*

lemma *filterlim-at-top*:

fixes $f :: 'a \Rightarrow ('b::\text{linorder})$

shows $(\text{LIM } x F. f x :> \text{at-top}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. Z \leq f x) F)$

by (*auto simp: filterlim-iff eventually-at-top-linorder elim!: eventually-mono*)

lemma *filterlim-at-top-mono*:

$\text{LIM } x F. f x :> \text{at-top} \Longrightarrow \text{eventually } (\lambda x. f x \leq (g x :: 'a::\text{linorder})) F \Longrightarrow$

$\text{LIM } x F. g x :> \text{at-top}$

by (*auto simp: filterlim-at-top elim: eventually-elim2 intro: order-trans*)

lemma *filterlim-at-top-dense*:

fixes $f :: 'a \Rightarrow ('b::\text{unbounded-dense-linorder})$

shows $(\text{LIM } x F. f x :> \text{at-top}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. Z < f x) F)$

by (*metis eventually-mono[of - F] eventually-gt-at-top order-less-imp-le filterlim-at-top[of f F] filterlim-iff[of f at-top F]*)

lemma *filterlim-at-top-ge*:

fixes $f :: 'a \Rightarrow ('b::\text{linorder})$ **and** $c :: 'b$

shows $(\text{LIM } x F. f x :> \text{at-top}) \longleftrightarrow (\forall Z \geq c. \text{eventually } (\lambda x. Z \leq f x) F)$

unfolding *at-top-sub*[of c] *filterlim-INF* **by** (*auto simp add: filterlim-principal*)

lemma *filterlim-at-top-at-top*:

fixes $f :: 'a::\text{linorder} \Rightarrow 'b::\text{linorder}$

assumes *mono*: $\bigwedge x y. Q x \Longrightarrow Q y \Longrightarrow x \leq y \Longrightarrow f x \leq f y$

assumes *bij*: $\bigwedge x. P x \Longrightarrow f (g x) = x \wedge \bigwedge x. P x \Longrightarrow Q (g x)$

assumes *Q*: *eventually* Q *at-top*

assumes *P*: *eventually* P *at-top*

shows *filterlim* f *at-top at-top*

proof –

from *P* **obtain** x **where** $x: \bigwedge y. x \leq y \Longrightarrow P y$

unfolding *eventually-at-top-linorder* **by** *auto*

show *?thesis*

proof (*intro filterlim-at-top-ge*[*THEN iffD2*] *allI impI*)

fix z **assume** $x \leq z$

with x **have** $P z$ **by** *auto*

have *eventually* $(\lambda x. g z \leq x)$ *at-top*

by (*rule eventually-ge-at-top*)

with Q **show** *eventually* $(\lambda x. z \leq f x)$ *at-top*

by *eventually-elim* (*metis mono bij <P z>*)

qed

qed

lemma *filterlim-at-top-gt*:

fixes $f :: 'a \Rightarrow ('b::\text{unbounded-dense-linorder})$ **and** $c :: 'b$

shows $(\text{LIM } x F. f x :> \text{at-top}) \longleftrightarrow (\forall Z > c. \text{eventually } (\lambda x. Z \leq f x) F)$

by (*metis filterlim-at-top order-less-le-trans gt-ex filterlim-at-top-ge*)

lemma *filterlim-at-bot*:

fixes $f :: 'a \Rightarrow ('b::\text{linorder})$
shows $(\text{LIM } x \ F. f \ x \ :> \ \text{at-bot}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. f \ x \ \leq \ Z) \ F)$
by $(\text{auto simp: filterlim-iff eventually-at-bot-linorder elim!: eventually-mono})$

lemma *filterlim-at-bot-dense*:

fixes $f :: 'a \Rightarrow ('b::\{\text{dense-linorder, no-bot}\})$
shows $(\text{LIM } x \ F. f \ x \ :> \ \text{at-bot}) \longleftrightarrow (\forall Z. \text{eventually } (\lambda x. f \ x \ < \ Z) \ F)$
proof $(\text{auto simp add: filterlim-at-bot[of } f \ F])$
fix $Z :: 'b$
from *lt-ex* $[\text{of } Z]$ **obtain** Z' **where** $1: Z' < Z ..$
assume $\forall Z. \text{eventually } (\lambda x. f \ x \ \leq \ Z) \ F$
hence $\text{eventually } (\lambda x. f \ x \ \leq \ Z') \ F$ **by** *auto*
thus $\text{eventually } (\lambda x. f \ x \ < \ Z) \ F$
by $(\text{rule eventually-mono})$ $(\text{use } 1 \ \text{in } \text{auto})$
next
fix $Z :: 'b$
show $\forall Z. \text{eventually } (\lambda x. f \ x \ < \ Z) \ F \implies \text{eventually } (\lambda x. f \ x \ \leq \ Z) \ F$
by $(\text{drule spec } [\text{of } - \ Z], \ \text{erule eventually-mono, auto simp add: less-imp-le})$
qed

lemma *filterlim-at-bot-le*:

fixes $f :: 'a \Rightarrow ('b::\text{linorder})$ **and** $c :: 'b$
shows $(\text{LIM } x \ F. f \ x \ :> \ \text{at-bot}) \longleftrightarrow (\forall Z \leq c. \text{eventually } (\lambda x. Z \geq f \ x) \ F)$
unfolding *filterlim-at-bot*
proof *safe*
fix Z **assume** $*: \forall Z \leq c. \text{eventually } (\lambda x. Z \geq f \ x) \ F$
with $*[\text{THEN spec, of min } Z \ c]$ **show** $\text{eventually } (\lambda x. Z \geq f \ x) \ F$
by $(\text{auto elim!: eventually-mono})$
qed *simp*

lemma *filterlim-at-bot-lt*:

fixes $f :: 'a \Rightarrow ('b::\text{unbounded-dense-linorder})$ **and** $c :: 'b$
shows $(\text{LIM } x \ F. f \ x \ :> \ \text{at-bot}) \longleftrightarrow (\forall Z < c. \text{eventually } (\lambda x. Z \geq f \ x) \ F)$
by $(\text{metis filterlim-at-bot filterlim-at-bot-le lt-ex order-le-less-trans})$

lemma *filterlim-at-top-div-const-nat*:

assumes $c > 0$
shows $\text{filterlim } (\lambda x::\text{nat. } x \ \text{div } \ c) \ \text{at-top at-top}$
unfolding *filterlim-at-top*
proof
fix $C :: \text{nat}$
have $*: n \ \text{div } \ c \geq \ C$ **if** $n \geq C * c$ **for** n
using *assms that* **by** $(\text{metis div-le-mono div-mult-self-is-m})$
have $\text{eventually } (\lambda n. n \geq C * c) \ \text{at-top}$
by $(\text{rule eventually-ge-at-top})$
thus $\text{eventually } (\lambda n. n \ \text{div } \ c \geq \ C) \ \text{at-top}$
by $\text{eventually-elim } (\text{use } * \ \text{in } \text{auto})$
qed

lemma *filterlim-finite-subsets-at-top*:
 $filterlim\ f\ (finite\ subsets\ at\ top\ A)\ F \longleftrightarrow$
 $(\forall X. finite\ X \wedge X \subseteq A \longrightarrow eventually\ (\lambda y. finite\ (f\ y) \wedge X \subseteq f\ y \wedge f\ y \subseteq A)$
 $F)$
(is *?lhs = ?rhs*)
proof
assume *?lhs*
thus *?rhs*
proof (*safe, goal-cases*)
case (*1 X*)
hence *: $(\forall_F\ x\ in\ F. P\ (f\ x))$ **if** *eventually P (finite-subsets-at-top A)* **for** *P*
using *that by (auto simp: filterlim-def le-filter-def eventually-filtermap)*
have $\forall_F\ Y\ in\ finite\ subsets\ at\ top\ A. finite\ Y \wedge X \subseteq Y \wedge Y \subseteq A$
using *1 unfolding eventually-finite-subsets-at-top by force*
thus *?case by (intro *) auto*
qed
next
assume *rhs: ?rhs*
show *?lhs unfolding filterlim-def le-filter-def eventually-finite-subsets-at-top*
proof (*safe, goal-cases*)
case (*1 P X*)
with *rhs have* $\forall_F\ y\ in\ F. finite\ (f\ y) \wedge X \subseteq f\ y \wedge f\ y \subseteq A$ **by** *auto*
thus *eventually P (filtermap f F) unfolding eventually-filtermap*
by *eventually-elim (insert 1, auto)*
qed
qed

lemma *filterlim-atMost-at-top*:
 $filterlim\ (\lambda n. \{..n\})\ (finite\ subsets\ at\ top\ (UNIV :: nat\ set))\ at\ top$
unfolding *filterlim-finite-subsets-at-top*
proof (*safe, goal-cases*)
case (*1 X*)
then obtain *n where* $n: X \subseteq \{..n\}$ **by** (*auto simp: finite-nat-set-iff-bounded-le*)
show *?case using eventually-ge-at-top[of n]*
by *eventually-elim (insert n, auto)*
qed

lemma *filterlim-lessThan-at-top*:
 $filterlim\ (\lambda n. \{..<n\})\ (finite\ subsets\ at\ top\ (UNIV :: nat\ set))\ at\ top$
unfolding *filterlim-finite-subsets-at-top*
proof (*safe, goal-cases*)
case (*1 X*)
then obtain *n where* $n: X \subseteq \{..<n\}$ **by** (*auto simp: finite-nat-set-iff-bounded*)
show *?case using eventually-ge-at-top[of n]*
by *eventually-elim (insert n, auto)*
qed

lemma *filterlim-minus-const-nat-at-top*:
 $filterlim\ (\lambda n. n - c)\ sequentially\ sequentially$

unfolding *filterlim-at-top*
proof
fix $a :: nat$
show *eventually* $(\lambda n. n - c \geq a)$ *at-top*
using *eventually-ge-at-top*[of $a + c$] **by** *eventually-elim auto*
qed

lemma *filterlim-add-const-nat-at-top*:
filterlim $(\lambda n. n + c)$ *sequentially sequentially*
unfolding *filterlim-at-top*
proof
fix $a :: nat$
show *eventually* $(\lambda n. n + c \geq a)$ *at-top*
using *eventually-ge-at-top*[of a] **by** *eventually-elim auto*
qed

91.8 Setup 'a filter for lifting and transfer

lemma *filtermap-id* [*simp, id-simps*]: *filtermap id = id*
by(*simp add: fun-eq-iff id-def filtermap-ident*)

lemma *filtermap-id'* [*simp*]: *filtermap* $(\lambda x. x) = (\lambda F. F)$
using *filtermap-id unfolding id-def* .

context includes *lifting-syntax*
begin

definition *map-filter-on* :: 'a set \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a filter \Rightarrow 'b filter **where**
map-filter-on $X f F = \text{Abs-filter } (\lambda P. \text{eventually } (\lambda x. P (f x) \wedge x \in X) F)$

lemma *is-filter-map-filter-on*:

is-filter $(\lambda P. \forall_F x \text{ in } F. P (f x) \wedge x \in X) \longleftrightarrow \text{eventually } (\lambda x. x \in X) F$

proof(*rule iffI; unfold-locales*)

show $\forall_F x \text{ in } F. \text{True} \wedge x \in X$ **if** *eventually* $(\lambda x. x \in X) F$ **using** *that* **by** *simp*

show $\forall_F x \text{ in } F. (P (f x) \wedge Q (f x)) \wedge x \in X$ **if** $\forall_F x \text{ in } F. P (f x) \wedge x \in X$
 $\forall_F x \text{ in } F. Q (f x) \wedge x \in X$ **for** $P Q$

using *eventually-conj*[*OF that*] **by**(*auto simp add: conj-ac cong: conj-cong*)

show $\forall_F x \text{ in } F. Q (f x) \wedge x \in X$ **if** $\forall x. P x \longrightarrow Q x$ $\forall_F x \text{ in } F. P (f x) \wedge x \in X$ **for** $P Q$

using *that*(2) **by**(*rule eventually-mono*)(*use that*(1) **in** *auto*)

show *eventually* $(\lambda x. x \in X) F$ **if** *is-filter* $(\lambda P. \forall_F x \text{ in } F. P (f x) \wedge x \in X)$

using *is-filter.True*[*OF that*] **by** *simp*

qed

lemma *eventually-map-filter-on*: *eventually* P $(\text{map-filter-on } X f F) = (\forall_F x \text{ in } F. P (f x) \wedge x \in X)$

if *eventually* $(\lambda x. x \in X) F$

by(*simp add: is-filter-map-filter-on map-filter-on-def eventually-Abs-filter that*)

lemma *map-filter-on-UNIV*: *map-filter-on UNIV = filtermap*
by(simp add: map-filter-on-def filtermap-def fun-eq-iff)

lemma *map-filter-on-comp*: *map-filter-on X f (map-filter-on Y g F) = map-filter-on*
Y (f ∘ g) F
if g ‘ Y ⊆ X and eventually (λx. x ∈ Y) F
unfolding *map-filter-on-def* **using** *that(1)*
by(*auto simp add: eventually-Abs-filter that(2) is-filter-map-filter-on intro!*: *arg-cong[where*
f=Abs-filter] arg-cong2[where f=eventually])

inductive *rel-filter* :: (*'a ⇒ 'b ⇒ bool*) ⇒ *'a filter ⇒ 'b filter ⇒ bool* **for** *R F G*
where
rel-filter R F G if eventually (case-prod R) Z map-filter-on {(x, y). R x y} fst Z
= F map-filter-on {(x, y). R x y} snd Z = G

lemma *rel-filter-eq* [*relator-eq*]: *rel-filter (=) = (=)*
proof(*intro ext iffI*)
show *F = G if rel-filter (=) F G for F G using that*
by *cases*(*clarsimp simp add: filter-eq-iff eventually-map-filter-on split-def cong:*
rev-conj-cong)
show *rel-filter (=) F G if F = G for F G unfolding ⟨F = G⟩*
proof
let *?Z = map-filter-on UNIV (λx. (x, x)) G*
have [*simp*]: *range (λx. (x, x)) ⊆ {(x, y). x = y}* **by** *auto*
show *map-filter-on {(x, y). x = y} fst ?Z = G and map-filter-on {(x, y). x =*
y} snd ?Z = G
by(*simp-all add: map-filter-on-comp*)(*simp-all add: map-filter-on-UNIV o-def*)
show *∀_F (x, y) in ?Z. x = y* **by**(*simp add: eventually-map-filter-on*)
qed
qed

lemma *rel-filter-mono* [*relator-mono*]: *rel-filter A ≤ rel-filter B if le: A ≤ B*
proof(*clarify elim!*: *rel-filter.cases*)
show *rel-filter B (map-filter-on {(x, y). A x y} fst Z) (map-filter-on {(x, y). A*
x y} snd Z)
(is rel-filter - ?X ?Y) if ∀_F (x, y) in Z. A x y for Z
proof
let *?Z = map-filter-on {(x, y). A x y} id Z*
show *∀_F (x, y) in ?Z. B x y* **using** *le that*
by(*simp add: eventually-map-filter-on le-fun-def split-def conj-commute cong:*
conj-cong)
have [*simp*]: *{(x, y). A x y} ⊆ {(x, y). B x y}* **using** *le by auto*
show *map-filter-on {(x, y). B x y} fst ?Z = ?X map-filter-on {(x, y). B x y}*
snd ?Z = ?Y
using *le that* **by**(*simp-all add: le-fun-def map-filter-on-comp*)
qed
qed

lemma *rel-filter-conversep*: *rel-filter A⁻¹⁻¹ = (rel-filter A)⁻¹⁻¹*

proof(*safe intro!*: *ext elim!*: *rel-filter.cases*)
show *: *rel-filter* A (*map-filter-on* $\{(x, y). A^{-1-1} x y\}$ *snd* Z) (*map-filter-on* $\{(x, y). A^{-1-1} x y\}$ *fst* Z)
 (*is rel-filter - ?X ?Y*) **if** $\forall_F (x, y)$ *in* Z . $A^{-1-1} x y$ **for** $A Z$
proof
let $?Z = \text{map-filter-on } \{(x, y). A x y\}$ *prod.swap* Z
show $\forall_F (x, y)$ *in* $?Z$. $A x y$ **using** *that* **by**(*simp add: eventually-map-filter-on*)
have [*simp*]: *prod.swap* ‘ $\{(x, y). A x y\} \subseteq \{(x, y). A x y\}$ **by** *auto*
show *map-filter-on* $\{(x, y). A x y\}$ *fst* $?Z = ?X$ *map-filter-on* $\{(x, y). A x y\}$
snd $?Z = ?Y$
using *that* **by**(*simp-all add: map-filter-on-comp o-def*)
qed
show *rel-filter* A^{-1-1} (*map-filter-on* $\{(x, y). A x y\}$ *snd* Z) (*map-filter-on* $\{(x, y). A x y\}$ *fst* Z)
if $\forall_F (x, y)$ *in* Z . $A x y$ **for** Z **using** *[*of* $A^{-1-1} Z$] *that* **by** *simp*
qed

lemma *rel-filter-distr* [*relator-distr*]:

rel-filter A *OO* *rel-filter* $B = \text{rel-filter } (A \text{ OO } B)$

proof(*safe intro!*: *ext elim!*: *rel-filter.cases*)

let $?AB = \{(x, y). (A \text{ OO } B) x y\}$

show (*rel-filter* A *OO* *rel-filter* B)

(*map-filter-on* $\{(x, y). (A \text{ OO } B) x y\}$ *fst* Z) (*map-filter-on* $\{(x, y). (A \text{ OO } B) x y\}$ *snd* Z)

(*is* (- *OO* -) $?F ?H$) **if** $\forall_F (x, y)$ *in* Z . $(A \text{ OO } B) x y$ **for** Z

proof

let $?G = \text{map-filter-on } ?AB (\lambda(x, y). \text{SOME } z. A x z \wedge B z y)$ Z

show *rel-filter* A $?F ?G$

proof

let $?Z = \text{map-filter-on } ?AB (\lambda(x, y). (x, \text{SOME } z. A x z \wedge B z y))$ Z

show $\forall_F (x, y)$ *in* $?Z$. $A x y$ **using** *that*

by(*auto simp add: eventually-map-filter-on split-def elim!: eventually-mono intro: someI2*)

have [*simp*]: $(\lambda p. (\text{fst } p, \text{SOME } z. A (\text{fst } p) z \wedge B z (\text{snd } p)))$ ‘ $\{p. (A \text{ OO } B) (\text{fst } p) (\text{snd } p)\} \subseteq \{p. A (\text{fst } p) (\text{snd } p)\}$ **by**(*auto intro: someI2*)

show *map-filter-on* $\{(x, y). A x y\}$ *fst* $?Z = ?F$ *map-filter-on* $\{(x, y). A x y\}$
snd $?Z = ?G$

using *that* **by**(*simp-all add: map-filter-on-comp split-def o-def*)

qed

show *rel-filter* B $?G ?H$

proof

let $?Z = \text{map-filter-on } ?AB (\lambda(x, y). (\text{SOME } z. A x z \wedge B z y, y))$ Z

show $\forall_F (x, y)$ *in* $?Z$. $B x y$ **using** *that*

by(*auto simp add: eventually-map-filter-on split-def elim!: eventually-mono intro: someI2*)

have [*simp*]: $(\lambda p. (\text{SOME } z. A (\text{fst } p) z \wedge B z (\text{snd } p), \text{snd } p))$ ‘ $\{p. (A \text{ OO } B) (\text{fst } p) (\text{snd } p)\} \subseteq \{p. B (\text{fst } p) (\text{snd } p)\}$ **by**(*auto intro: someI2*)

show *map-filter-on* $\{(x, y). B x y\}$ *fst* $?Z = ?G$ *map-filter-on* $\{(x, y). B x y\}$
snd $?Z = ?H$

```

    using that by(simp-all add: map-filter-on-comp split-def o-def)
  qed
qed

fix F G
assume F:  $\forall_F (x, y)$  in F. A x y and G:  $\forall_F (x, y)$  in G. B x y
and eq: map-filter-on  $\{(x, y). B x y\}$  fst G = map-filter-on  $\{(x, y). A x y\}$  snd
F (is ?Y2 = ?Y1)
let ?X = map-filter-on  $\{(x, y). A x y\}$  fst F
and ?Z = (map-filter-on  $\{(x, y). B x y\}$  snd G)
have step:  $\exists P' \leq P. \exists Q' \leq Q. \text{eventually } P' F \wedge \text{eventually } Q' G \wedge \{y. \exists x. P' (x, y)\} = \{y. \exists z. Q' (y, z)\}$ 
if P: eventually P F and Q: eventually Q G for P Q
proof -
let ?P =  $\lambda(x, y). P (x, y) \wedge A x y$  and ?Q =  $\lambda(y, z). Q (y, z) \wedge B y z$ 
define P' where  $P' \equiv \lambda(x, y). ?P (x, y) \wedge (\exists z. ?Q (y, z))$ 
define Q' where  $Q' \equiv \lambda(y, z). ?Q (y, z) \wedge (\exists x. ?P (x, y))$ 
have  $P' \leq P \ Q' \leq Q \ \{y. \exists x. P' (x, y)\} = \{y. \exists z. Q' (y, z)\}$ 
by(auto simp add: P'-def Q'-def)
moreover
from P Q F G have P': eventually ?P F and Q': eventually ?Q G
by(simp-all add: eventually-conj-iff split-def)
from P' F have  $\forall_F y$  in ?Y1.  $\exists x. P (x, y) \wedge A x y$ 
by(auto simp add: eventually-map-filter-on elim!: eventually-mono)
from this[folded eq] obtain Q'' where Q'': eventually Q'' G
and Q''P:  $\{y. \exists z. Q'' (y, z)\} \subseteq \{y. \exists x. ?P (x, y)\}$ 
using G by(fastforce simp add: eventually-map-filter-on)
have eventually (inf Q'' ?Q) G using Q'' Q' by(auto intro: eventually-conj
simp add: inf-fun-def)
then have eventually Q' G using Q''P by(auto elim!: eventually-mono simp
add: Q'-def)
moreover
from Q' G have  $\forall_F y$  in ?Y2.  $\exists z. Q (y, z) \wedge B y z$ 
by(auto simp add: eventually-map-filter-on elim!: eventually-mono)
from this[unfolded eq] obtain P'' where P'': eventually P'' F
and P''Q:  $\{y. \exists x. P'' (x, y)\} \subseteq \{y. \exists z. ?Q (y, z)\}$ 
using F by(fastforce simp add: eventually-map-filter-on)
have eventually (inf P'' ?P) F using P'' P' by(auto intro: eventually-conj
simp add: inf-fun-def)
then have eventually P' F using P''Q by(auto elim!: eventually-mono simp
add: P'-def)
ultimately show ?thesis by blast
qed

show rel-filter (A OO B) ?X ?Z
proof
let ?Y =  $\lambda Y. \exists X Z. \text{eventually } X ?X \wedge \text{eventually } Z ?Z \wedge (\lambda(x, z). X x \wedge Z z \wedge (A OO B) x z) \leq Y$ 
have Y: is-filter ?Y

```

```

proof
  show ?Y ( $\lambda$ -. True) by(auto simp add: le-fun-def intro: eventually-True)
  show ?Y ( $\lambda$ x. P x  $\wedge$  Q x) if ?Y P ?Y Q for P Q using that
    apply clarify
      apply(intro exI conjI; (elim eventually-rev-mp; fold imp-conjL; intro al-
ways-eventually allI; rule imp-refl)?)
    apply auto
    done
  show ?Y Q if ?Y P  $\forall$ x. P x  $\longrightarrow$  Q x for P Q using that by blast
qed
define Y where Y = Abs-filter ?Y
have eventually-Y: eventually P Y  $\longleftrightarrow$  ?Y P for P
  using eventually-Abs-filter[OF Y, of P] by(simp add: Y-def)
show YY:  $\forall_F$  (x, y) in Y. (A OO B) x y using F G
  by(auto simp add: eventually-Y eventually-map-filter-on eventually-conj-iff
intro!: eventually-True)
  have ?Y ( $\lambda$ (x, z). P x  $\wedge$  (A OO B) x z)  $\longleftrightarrow$  ( $\forall_F$  (x, y) in F. P x  $\wedge$  A x y)
(is ?lhs = ?rhs) for P
proof
  show ?lhs if ?rhs using G F that
    by(auto 4 3 intro: exI[where x= $\lambda$ -. True] simp add: eventually-map-filter-on
split-def)
  assume ?lhs
  then obtain X Z where  $\forall_F$  (x, y) in F. X x  $\wedge$  A x y
    and  $\forall_F$  (x, y) in G. Z y  $\wedge$  B x y
    and ( $\lambda$ (x, z). X x  $\wedge$  Z z  $\wedge$  (A OO B) x z)  $\leq$  ( $\lambda$ (x, z). P x  $\wedge$  (A OO B) x z)
    using F G by(auto simp add: eventually-map-filter-on split-def)
  from step[OF this(1, 2)] this(3)
  show ?rhs by(clarsimp elim!: eventually-rev-mp simp add: le-fun-def)(fastforce
intro: always-eventually)
qed
then show map-filter-on ?AB fst Y = ?X
  by(simp add: filter-eq-iff YY eventually-map-filter-on)(simp add: eventually-Y
eventually-map-filter-on F G; simp add: split-def)

  have ?Y ( $\lambda$ (x, z). P z  $\wedge$  (A OO B) x z)  $\longleftrightarrow$  ( $\forall_F$  (x, y) in G. P y  $\wedge$  B x y)
(is ?lhs = ?rhs) for P
proof
  show ?lhs if ?rhs using G F that
    by(auto 4 3 intro: exI[where x= $\lambda$ -. True] simp add: eventually-map-filter-on
split-def)
  assume ?lhs
  then obtain X Z where  $\forall_F$  (x, y) in F. X x  $\wedge$  A x y
    and  $\forall_F$  (x, y) in G. Z y  $\wedge$  B x y
    and ( $\lambda$ (x, z). X x  $\wedge$  Z z  $\wedge$  (A OO B) x z)  $\leq$  ( $\lambda$ (x, z). P z  $\wedge$  (A OO B) x z)
    using F G by(auto simp add: eventually-map-filter-on split-def)
  from step[OF this(1, 2)] this(3)
  show ?rhs by(clarsimp elim!: eventually-rev-mp simp add: le-fun-def)(fastforce
intro: always-eventually)

```

qed
then show *map-filter-on* ?*AB* *snd* *Y* = ?*Z*
by(*simp add: filter-eq-iff* *YY* *eventually-map-filter-on*)(*simp add: eventually-Y*
eventually-map-filter-on *F* *G*; *simp add: split-def*)
qed
qed

lemma *filtermap-parametric*: ((*A* ==> *B*) ==> *rel-filter* *A* ==> *rel-filter* *B*) *filtermap* *filtermap*

proof(*intro rel-funI*; *erule rel-filter.cases*; *hypsubst*)

fix *f g Z*
assume *fg*: (*A* ==> *B*) *f g* **and** *Z*: $\forall_F (x, y) \text{ in } Z. A \ x \ y$
have *rel-filter* *B* (*map-filter-on* {(*x*, *y*). *A* *x* *y*} (*f* \circ *fst*) *Z*) (*map-filter-on* {(*x*,
y). *A* *x* *y*} (*g* \circ *snd*) *Z*)
(is rel-filter - ?F ?G)

proof

let ?*Z* = *map-filter-on* {(*x*, *y*). *A* *x* *y*} (*map-prod* *f g*) *Z*
show $\forall_F (x, y) \text{ in } ?Z. B \ x \ y$ **using** *fg Z*
by(*auto simp add: eventually-map-filter-on split-def elim!*: *eventually-mono*
rel-funD)

have [*simp*]: *map-prod* *f g* ‘ {*p*. *A* (*fst* *p*) (*snd* *p*)} \subseteq {*p*. *B* (*fst* *p*) (*snd* *p*)}

using *fg* **by**(*auto dest: rel-funD*)

show *map-filter-on* {(*x*, *y*). *B* *x* *y*} *fst* ?*Z* = ?*F* *map-filter-on* {(*x*, *y*). *B* *x* *y*}
snd ?*Z* = ?*G*

using *Z* **by**(*auto simp add: map-filter-on-comp split-def*)

qed

thus *rel-filter* *B* (*filtermap* *f* (*map-filter-on* {(*x*, *y*). *A* *x* *y*} *fst* *Z*)) (*filtermap* *g*
(*map-filter-on* {(*x*, *y*). *A* *x* *y*} *snd* *Z*))

using *Z* **by**(*simp add: map-filter-on-UNIV[symmetric] map-filter-on-comp*)

qed

lemma *rel-filter-Grp*: *rel-filter* (*Grp* *UNIV* *f*) = *Grp* *UNIV* (*filtermap* *f*)

proof((*intro antisym predicate2I*; (*elim GrpE*; *hypsubst*)?), *rule GrpI[OF - UNIV-I]*)

fix *F G*

assume *rel-filter* (*Grp* *UNIV* *f*) *F G*

hence *rel-filter* (=) (*filtermap* *f* *F*) (*filtermap* *id* *G*)

by(*rule filtermap-parametric[THEN rel-funD, THEN rel-funD, rotated]*)(*simp*
add: Grp-def rel-fun-def)

thus *filtermap* *f* *F* = *G* **by**(*simp add: rel-filter-eq*)

next

fix *F* :: ‘*a* *filter*

have *rel-filter* (=) *F* *F* **by**(*simp add: rel-filter-eq*)

hence *rel-filter* (*Grp* *UNIV* *f*) (*filtermap* *id* *F*) (*filtermap* *f* *F*)

by(*rule filtermap-parametric[THEN rel-funD, THEN rel-funD, rotated]*)(*simp*
add: Grp-def rel-fun-def)

thus *rel-filter* (*Grp* *UNIV* *f*) *F* (*filtermap* *f* *F*) **by** *simp*

qed

lemma *Quotient-filter* [*quot-map*]:

Quotient R Abs Rep T \implies *Quotient (rel-filter R) (filtermap Abs) (filtermap Rep)*
(rel-filter T)
unfolding *Quotient-alt-def5 rel-filter-eq[symmetric] rel-filter-Grp[symmetric]*
by(*simp add: rel-filter-distr[symmetric] rel-filter-conversep[symmetric] rel-filter-mono*)

lemma *left-total-rel-filter [transfer-rule]: left-total A \implies left-total (rel-filter A)*
unfolding *left-total-alt-def rel-filter-eq[symmetric] rel-filter-conversep[symmetric]*
rel-filter-distr
by(*rule rel-filter-mono*)

lemma *right-total-rel-filter [transfer-rule]: right-total A \implies right-total (rel-filter A)*
using *left-total-rel-filter[of A⁻¹⁻¹] by(simp add: rel-filter-conversep)*

lemma *bi-total-rel-filter [transfer-rule]: bi-total A \implies bi-total (rel-filter A)*
unfolding *bi-total-alt-def by(simp add: left-total-rel-filter right-total-rel-filter)*

lemma *left-unique-rel-filter [transfer-rule]: left-unique A \implies left-unique (rel-filter A)*
unfolding *left-unique-alt-def rel-filter-eq[symmetric] rel-filter-conversep[symmetric]*
rel-filter-distr
by(*rule rel-filter-mono*)

lemma *right-unique-rel-filter [transfer-rule]:*
right-unique A \implies right-unique (rel-filter A)
using *left-unique-rel-filter[of A⁻¹⁻¹] by(simp add: rel-filter-conversep)*

lemma *bi-unique-rel-filter [transfer-rule]: bi-unique A \implies bi-unique (rel-filter A)*
by(*simp add: bi-unique-alt-def left-unique-rel-filter right-unique-rel-filter*)

lemma *eventually-parametric [transfer-rule]:*
((A \implies (=)) \implies rel-filter A \implies (=)) eventually eventually
by(*auto 4 4 intro!: rel-funI elim!: rel-filter.cases simp add: eventually-map-filter-on*
dest: rel-funD intro: always-eventually elim!: eventually-rev-mp)

lemma *frequently-parametric [transfer-rule]: ((A \implies (=)) \implies rel-filter A \implies (=)) frequently frequently*
unfolding *frequently-def[abs-def] by transfer-prover*

lemma *is-filter-parametric [transfer-rule]:*
assumes *[transfer-rule]: bi-total A*
assumes *[transfer-rule]: bi-unique A*
shows *((A \implies (=)) \implies (=)) \implies (=) is-filter is-filter*
unfolding *is-filter-def by transfer-prover*

lemma *top-filter-parametric [transfer-rule]: rel-filter A top top if bi-total A*

proof

let *?Z = principal {(x, y). A x y}*
show $\forall_F (x, y) \text{ in } ?Z. A x y$ **by**(*simp add: eventually-principal*)

show *map-filter-on* $\{(x, y). A\ x\ y\}$ *fst* $?Z = \text{top}$ *map-filter-on* $\{(x, y). A\ x\ y\}$
snd $?Z = \text{top}$
using *that* **by**(*auto simp add: filter-eq-iff eventually-map-filter-on eventually-principal*
bi-total-def)
qed

lemma *bot-filter-parametric* [*transfer-rule*]: *rel-filter* A *bot* *bot*

proof

show $\forall_F (x, y)$ *in* *bot*. $A\ x\ y$ **by** *simp*
show *map-filter-on* $\{(x, y). A\ x\ y\}$ *fst* *bot* = *bot* *map-filter-on* $\{(x, y). A\ x\ y\}$
snd *bot* = *bot*
by(*simp-all add: filter-eq-iff eventually-map-filter-on*)
qed

lemma *principal-parametric* [*transfer-rule*]: (*rel-set* A $====>$ *rel-filter* A) *principal*
principal

proof(*rule rel-funI rel-filter.intros*)+

fix $S\ S'$

assume $*$: *rel-set* $A\ S\ S'$

define SS' **where** $SS' = S \times S' \cap \{(x, y). A\ x\ y\}$

have SS' : $SS' \subseteq \{(x, y). A\ x\ y\}$ **and** [*simp*]: $S = \text{fst} \text{ ` } SS'$ $S' = \text{snd} \text{ ` } SS'$

using $*$ **by**(*auto 4 3 dest: rel-setD1 rel-setD2 intro: rev-image-eqI simp add:*
SS'-def)

let $?Z = \text{principal } SS'$

show $\forall_F (x, y)$ *in* $?Z$. $A\ x\ y$ **using** SS' **by**(*auto simp add: eventually-principal*)

then show *map-filter-on* $\{(x, y). A\ x\ y\}$ *fst* $?Z = \text{principal } S$

and *map-filter-on* $\{(x, y). A\ x\ y\}$ *snd* $?Z = \text{principal } S'$

by(*auto simp add: filter-eq-iff eventually-map-filter-on eventually-principal*)

qed

lemma *sup-filter-parametric* [*transfer-rule*]:

(*rel-filter* A $====>$ *rel-filter* A $====>$ *rel-filter* A) *sup* *sup*

proof(*intro rel-funI; elim rel-filter.cases; hypsubst*)

show *rel-filter* A

(*map-filter-on* $\{(x, y). A\ x\ y\}$ *fst* $FG \sqcup \text{map-filter-on } \{(x, y). A\ x\ y\}$ *fst* FG')

(*map-filter-on* $\{(x, y). A\ x\ y\}$ *snd* $FG \sqcup \text{map-filter-on } \{(x, y). A\ x\ y\}$ *snd* FG')

(*is rel-filter* - (*sup* $?F\ ?G$) (*sup* $?F'\ ?G'$))

if $\forall_F (x, y)$ *in* FG . $A\ x\ y$ $\forall_F (x, y)$ *in* FG' . $A\ x\ y$ **for** $FG\ FG'$

proof

let $?Z = \text{sup } FG\ FG'$

show $\forall_F (x, y)$ *in* $?Z$. $A\ x\ y$ **by**(*simp add: eventually-sup that*)

then show *map-filter-on* $\{(x, y). A\ x\ y\}$ *fst* $?Z = \text{sup } ?F\ ?G$

and *map-filter-on* $\{(x, y). A\ x\ y\}$ *snd* $?Z = \text{sup } ?F'\ ?G'$

by(*simp-all add: filter-eq-iff eventually-map-filter-on eventually-sup*)

qed

qed

lemma *Sup-filter-parametric* [*transfer-rule*]: (*rel-set* (*rel-filter* A) $====>$ *rel-filter*
 A) *Sup* *Sup*

```

proof(rule rel-funI)
  fix S S'
  define SS' where SS' = S × S' ∩ {(F, G). rel-filter A F G}
  assume rel-set (rel-filter A) S S'
  then have SS': SS' ⊆ {(F, G). rel-filter A F G} and [simp]: S = fst ' SS' S' =
  snd ' SS'
  by(auto 4 3 dest: rel-setD1 rel-setD2 intro: rev-image-eqI simp add: SS'-def)
  from SS' obtain Z where Z: ∧F G. (F, G) ∈ SS' ⇒
  (∀F (x, y) in Z F G. A x y) ∧
  id F = map-filter-on {(x, y). A x y} fst (Z F G) ∧
  id G = map-filter-on {(x, y). A x y} snd (Z F G)
  unfolding rel-filter.simps by atomize-elim((rule choice allI)+; auto)
  have id: eventually P F = eventually P (id F) eventually Q G = eventually Q
  (id G)
  if (F, G) ∈ SS' for P Q F G by simp-all
  show rel-filter A (Sup S) (Sup S')
  proof
    let ?Z = ⋃ (F, G) ∈ SS'. Z F G
    show *: ∀F (x, y) in ?Z. A x y using Z by(auto simp add: eventually-Sup)
    show map-filter-on {(x, y). A x y} fst ?Z = Sup S map-filter-on {(x, y). A x
  y} snd ?Z = Sup S'
    unfolding filter-eq-iff
    by(auto 4 4 simp add: id eventually-Sup eventually-map-filter-on *[simplified
  eventually-Sup] simp del: id-apply dest: Z)
  qed
qed

context
  fixes A :: 'a ⇒ 'b ⇒ bool
  assumes [transfer-rule]: bi-unique A
begin

lemma le-filter-parametric [transfer-rule]:
  (rel-filter A ==> rel-filter A ==> (=)) (≤) (≤)
unfolding le-filter-def[abs-def] by transfer-prover

lemma less-filter-parametric [transfer-rule]:
  (rel-filter A ==> rel-filter A ==> (=)) (<) (<)
unfolding less-filter-def[abs-def] by transfer-prover

context
  assumes [transfer-rule]: bi-total A
begin

lemma Inf-filter-parametric [transfer-rule]:
  (rel-set (rel-filter A) ==> rel-filter A) Inf Inf
unfolding Inf-filter-def[abs-def] by transfer-prover

lemma inf-filter-parametric [transfer-rule]:

```



```

  (rel-filter A ===> rel-filter A ===> rel-filter A) inf inf
proof(intro rel-funI)+
  fix F F' G G'
  assume [transfer-rule]: rel-filter A F F' rel-filter A G G'
  have rel-filter A (Inf {F, G}) (Inf {F', G'}) by transfer-prover
  thus rel-filter A (inf F G) (inf F' G') by simp
qed

end

end

end

end

context
  includes lifting-syntax
begin

lemma prod-filter-parametric [transfer-rule]:
  (rel-filter R ===> rel-filter S ===> rel-filter (rel-prod R S)) prod-filter prod-filter
proof(intro rel-funI; elim rel-filter.cases; hypsubst)
  fix F G
  assume F:  $\forall_F (x, y) \text{ in } F. R \ x \ y$  and G:  $\forall_F (x, y) \text{ in } G. S \ x \ y$ 
  show rel-filter (rel-prod R S)
    (map-filter-on {(x, y). R x y} fst F  $\times_F$  map-filter-on {(x, y). S x y} fst G)
    (map-filter-on {(x, y). R x y} snd F  $\times_F$  map-filter-on {(x, y). S x y} snd G)
    (is rel-filter ?RS ?F ?G)
  proof
  let ?Z = filtermap ( $\lambda((a, b), (a', b')). ((a, a'), (b, b'))$ ) (prod-filter F G)
  show *:  $\forall_F (x, y) \text{ in } ?Z. \text{rel-prod } R \ S \ x \ y$  using F G
    by(auto simp add: eventually-filtermap split-beta eventually-prod-filter)
  show map-filter-on {(x, y). ?RS x y} fst ?Z = ?F
    using F G
    apply(clarsimp simp add: filter-eq-iff eventually-map-filter-on *)
    apply(simp add: eventually-filtermap split-beta eventually-prod-filter)
    apply(subst eventually-map-filter-on; simp)+
    apply(rule iffI; clarsimp)
  subgoal for P P' P''
    apply(rule exI[where  $x=\lambda a. \exists b. P' (a, b) \wedge R \ a \ b$ ]; rule conjI)
    subgoal by(fastforce elim: eventually-rev-mp eventually-mono)
  subgoal
    by(rule exI[where  $x=\lambda a. \exists b. P'' (a, b) \wedge S \ a \ b$ ])(fastforce elim: eventu-
ally-rev-mp eventually-mono)
  done
  subgoal by fastforce
  done
  show map-filter-on {(x, y). ?RS x y} snd ?Z = ?G
    using F G
    apply(clarsimp simp add: filter-eq-iff eventually-map-filter-on *)

```

```

apply(simp add: eventually-filtermap split-beta eventually-prod-filter)
apply(subst eventually-map-filter-on; simp)+
apply(rule iffI; clarsimp)
subgoal for  $P P' P''$ 
  apply(rule exI[where  $x=\lambda b. \exists a. P' (a, b) \wedge R a b$ ]; rule conjI)
  subgoal by(fastforce elim: eventually-rev-mp eventually-mono)
  subgoal
    by(rule exI[where  $x=\lambda b. \exists a. P'' (a, b) \wedge S a b$ ])(fastforce elim: eventu-
ally-rev-mp eventually-mono)
  done
  subgoal by fastforce
  done
qed
qed

end

```

Code generation for filters

```

definition abstract-filter :: (unit  $\Rightarrow$  'a filter)  $\Rightarrow$  'a filter
  where [simp]: abstract-filter f = f ()

```

```

code-datatype principal abstract-filter

```

```

hide-const (open) abstract-filter

```

```

declare [[code drop: filterlim prod-filter filtermap eventually
  inf :: - filter  $\Rightarrow$  - sup :: - filter  $\Rightarrow$  - less-eq :: - filter  $\Rightarrow$  -
  Abs-filter]]

```

```

declare filterlim-principal [code]
declare principal-prod-principal [code]
declare filtermap-principal [code]
declare filtercomap-principal [code]
declare eventually-principal [code]
declare inf-principal [code]
declare sup-principal [code]
declare principal-le-iff [code]

```

```

lemma Rep-filter-iff-eventually [simp, code]:
  Rep-filter F P  $\longleftrightarrow$  eventually P F
  by (simp add: eventually-def)

```

```

lemma bot-eq-principal-empty [code]:
  bot = principal {}
  by simp

```

```

lemma top-eq-principal-UNIV [code]:
  top = principal UNIV
  by simp

```

```

instantiation filter :: (equal) equal
begin

definition equal-filter :: 'a filter  $\Rightarrow$  'a filter  $\Rightarrow$  bool
  where equal-filter F F'  $\longleftrightarrow$  F = F'

lemma equal-filter [code]:
  HOL.equal (principal A) (principal B)  $\longleftrightarrow$  A = B
  by (simp add: equal-filter-def)

instance
  by standard (simp add: equal-filter-def)

end

end

```

92 Conditionally-complete Lattices

```

theory Conditionally-Complete-Lattices
imports Finite-Set Lattices-Big Set-Interval
begin

locale preordering-bdd = preordering
begin

definition bdd :: 'a set  $\Rightarrow$  bool
  where unfold:  $\langle$ bdd A  $\longleftrightarrow$  ( $\exists$  M.  $\forall$  x  $\in$  A. x  $\leq$  M) $\rangle$ 

lemma empty [simp, intro]:
   $\langle$ bdd {} $\rangle$ 
  by (simp add: unfold)

lemma I [intro]:
   $\langle$ bdd A $\rangle$  if  $\langle$  $\bigwedge$ x. x  $\in$  A  $\Longrightarrow$  x  $\leq$  M $\rangle$ 
  using that by (auto simp add: unfold)

lemma E:
  assumes  $\langle$ bdd A $\rangle$ 
  obtains M where  $\langle$  $\bigwedge$ x. x  $\in$  A  $\Longrightarrow$  x  $\leq$  M $\rangle$ 
  using assms that by (auto simp add: unfold)

lemma I2:
   $\langle$ bdd (f ' A) $\rangle$  if  $\langle$  $\bigwedge$ x. x  $\in$  A  $\Longrightarrow$  f x  $\leq$  M $\rangle$ 
  using that by (auto simp add: unfold)

lemma mono:
   $\langle$ bdd A $\rangle$  if  $\langle$ bdd B $\rangle$   $\langle$ A  $\subseteq$  B $\rangle$ 

```

using *that* **by** (*auto simp add: unfold*)

lemma *Int1* [*simp*]:
 $\langle \text{bdd } (A \cap B) \rangle$ **if** $\langle \text{bdd } A \rangle$
using *mono that by auto*

lemma *Int2* [*simp*]:
 $\langle \text{bdd } (A \cap B) \rangle$ **if** $\langle \text{bdd } B \rangle$
using *mono that by auto*

end

92.1 Preorders

context *preorder*
begin

sublocale *bdd-above: preordering-bdd* $\langle (\leq) \rangle$ $\langle (<) \rangle$
defines *bdd-above-primitive-def*: $\text{bdd-above} = \text{bdd-above.bdd} \dots$

sublocale *bdd-below: preordering-bdd* $\langle (\geq) \rangle$ $\langle (>) \rangle$
defines *bdd-below-primitive-def*: $\text{bdd-below} = \text{bdd-below.bdd} \dots$

lemma *bdd-above-def*: $\langle \text{bdd-above } A \longleftrightarrow (\exists M. \forall x \in A. x \leq M) \rangle$
by (*fact bdd-above.unfold*)

lemma *bdd-below-def*: $\langle \text{bdd-below } A \longleftrightarrow (\exists M. \forall x \in A. M \leq x) \rangle$
by (*fact bdd-below.unfold*)

lemma *bdd-aboveI*: $(\bigwedge x. x \in A \implies x \leq M) \implies \text{bdd-above } A$
by (*fact bdd-above.I*)

lemma *bdd-belowI*: $(\bigwedge x. x \in A \implies m \leq x) \implies \text{bdd-below } A$
by (*fact bdd-below.I*)

lemma *bdd-aboveI2*: $(\bigwedge x. x \in A \implies f x \leq M) \implies \text{bdd-above } (f'A)$
by (*fact bdd-above.I2*)

lemma *bdd-belowI2*: $(\bigwedge x. x \in A \implies m \leq f x) \implies \text{bdd-below } (f'A)$
by (*fact bdd-below.I2*)

lemma *bdd-above-empty*: $\text{bdd-above } \{\}$
by (*fact bdd-above.empty*)

lemma *bdd-below-empty*: $\text{bdd-below } \{\}$
by (*fact bdd-below.empty*)

lemma *bdd-above-mono*: $\text{bdd-above } B \implies A \subseteq B \implies \text{bdd-above } A$
by (*fact bdd-above.mono*)

lemma *bdd-below-mono*: $bdd\text{-below } B \implies A \subseteq B \implies bdd\text{-below } A$
by (*fact bdd-below.mono*)

lemma *bdd-above-Int1*: $bdd\text{-above } A \implies bdd\text{-above } (A \cap B)$
by (*fact bdd-above.Int1*)

lemma *bdd-above-Int2*: $bdd\text{-above } B \implies bdd\text{-above } (A \cap B)$
by (*fact bdd-above.Int2*)

lemma *bdd-below-Int1*: $bdd\text{-below } A \implies bdd\text{-below } (A \cap B)$
by (*fact bdd-below.Int1*)

lemma *bdd-below-Int2*: $bdd\text{-below } B \implies bdd\text{-below } (A \cap B)$
by (*fact bdd-below.Int2*)

lemma *bdd-above-Ioo* [*simp, intro*]: $bdd\text{-above } \{a <..< b\}$
by (*auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le*)

lemma *bdd-above-Ico* [*simp, intro*]: $bdd\text{-above } \{a ..< b\}$
by (*auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le*)

lemma *bdd-above-Iio* [*simp, intro*]: $bdd\text{-above } \{..< b\}$
by (*auto simp add: bdd-above-def intro: exI[of - b] less-imp-le*)

lemma *bdd-above-Ioc* [*simp, intro*]: $bdd\text{-above } \{a <.. b\}$
by (*auto simp add: bdd-above-def intro: exI[of - b] less-imp-le*)

lemma *bdd-above-Icc* [*simp, intro*]: $bdd\text{-above } \{a .. b\}$
by (*auto simp add: bdd-above-def intro: exI[of - b] less-imp-le*)

lemma *bdd-above-Iic* [*simp, intro*]: $bdd\text{-above } \{.. b\}$
by (*auto simp add: bdd-above-def intro: exI[of - b] less-imp-le*)

lemma *bdd-below-Ioo* [*simp, intro*]: $bdd\text{-below } \{a <..< b\}$
by (*auto simp add: bdd-below-def intro!: exI[of - a] less-imp-le*)

lemma *bdd-below-Ioc* [*simp, intro*]: $bdd\text{-below } \{a <.. b\}$
by (*auto simp add: bdd-below-def intro!: exI[of - a] less-imp-le*)

lemma *bdd-below-Ioi* [*simp, intro*]: $bdd\text{-below } \{a <.. \}$
by (*auto simp add: bdd-below-def intro: exI[of - a] less-imp-le*)

lemma *bdd-below-Ico* [*simp, intro*]: $bdd\text{-below } \{a ..< b\}$
by (*auto simp add: bdd-below-def intro: exI[of - a] less-imp-le*)

lemma *bdd-below-Icc* [*simp, intro*]: $bdd\text{-below } \{a .. b\}$
by (*auto simp add: bdd-below-def intro: exI[of - a] less-imp-le*)

```

lemma bdd-below-Ici [simp, intro]: bdd-below {a ..}
  by (auto simp add: bdd-below-def intro: exI[of - a] less-imp-le)

end

context order-top
begin

lemma bdd-above-top [simp, intro!]: bdd-above A
  by (rule bdd-aboveI [of - top] simp)

end

context order-bot
begin

lemma bdd-below-bot [simp, intro!]: bdd-below A
  by (rule bdd-belowI [of - bot] simp)

end

lemma bdd-above-image-mono: mono f  $\implies$  bdd-above A  $\implies$  bdd-above (f'A)
  by (auto simp: bdd-above-def mono-def)

lemma bdd-below-image-mono: mono f  $\implies$  bdd-below A  $\implies$  bdd-below (f'A)
  by (auto simp: bdd-below-def mono-def)

lemma bdd-above-image-antimono: antimono f  $\implies$  bdd-below A  $\implies$  bdd-above
(f'A)
  by (auto simp: bdd-above-def bdd-below-def antimono-def)

lemma bdd-below-image-antimono: antimono f  $\implies$  bdd-above A  $\implies$  bdd-below
(f'A)
  by (auto simp: bdd-above-def bdd-below-def antimono-def)

lemma
  fixes X :: 'a::ordered-ab-group-add set
  shows bdd-above-uminus[simp]: bdd-above (uminus 'X)  $\longleftrightarrow$  bdd-below X
    and bdd-below-uminus[simp]: bdd-below (uminus 'X)  $\longleftrightarrow$  bdd-above X
  using bdd-above-image-antimono[of uminus X] bdd-below-image-antimono[of umi-
nus uminus'X]
  using bdd-below-image-antimono[of uminus X] bdd-above-image-antimono[of umi-
nus uminus'X]
  by (auto simp: antimono-def image-image)

```

92.2 Lattices

```

context lattice
begin

```

lemma *bdd-above-insert* [*simp*]: $\text{bdd-above } (\text{insert } a \ A) = \text{bdd-above } A$
by (*auto simp: bdd-above-def intro: le-supI2 sup-ge1*)

lemma *bdd-below-insert* [*simp*]: $\text{bdd-below } (\text{insert } a \ A) = \text{bdd-below } A$
by (*auto simp: bdd-below-def intro: le-infI2 inf-le1*)

lemma *bdd-finite* [*simp*]:
assumes *finite A* **shows** *bdd-above-finite: bdd-above A* **and** *bdd-below-finite: bdd-below A*
using *assms* **by** (*induct rule: finite-induct, auto*)

lemma *bdd-above-Un* [*simp*]: $\text{bdd-above } (A \cup B) = (\text{bdd-above } A \wedge \text{bdd-above } B)$
proof
assume *bdd-above (A ∪ B)*
thus *bdd-above A ∧ bdd-above B* **unfolding** *bdd-above-def* **by** *auto*
next
assume *bdd-above A ∧ bdd-above B*
then obtain *a b* **where** $\forall x \in A. x \leq a \ \forall x \in B. x \leq b$ **unfolding** *bdd-above-def*
by *auto*
hence $\forall x \in A \cup B. x \leq \text{sup } a \ b$ **by** (*auto intro: Un-iff le-supI1 le-supI2*)
thus *bdd-above (A ∪ B)* **unfolding** *bdd-above-def* **..**
qed

lemma *bdd-below-Un* [*simp*]: $\text{bdd-below } (A \cup B) = (\text{bdd-below } A \wedge \text{bdd-below } B)$
proof
assume *bdd-below (A ∪ B)*
thus *bdd-below A ∧ bdd-below B* **unfolding** *bdd-below-def* **by** *auto*
next
assume *bdd-below A ∧ bdd-below B*
then obtain *a b* **where** $\forall x \in A. a \leq x \ \forall x \in B. b \leq x$ **unfolding** *bdd-below-def*
by *auto*
hence $\forall x \in A \cup B. \text{inf } a \ b \leq x$ **by** (*auto intro: Un-iff le-infI1 le-infI2*)
thus *bdd-below (A ∪ B)* **unfolding** *bdd-below-def* **..**
qed

lemma *bdd-above-image-sup*[*simp*]:
 $\text{bdd-above } ((\lambda x. \text{sup } (f \ x) \ (g \ x)) \ 'A) \longleftrightarrow \text{bdd-above } (f \ 'A) \wedge \text{bdd-above } (g \ 'A)$
by (*auto simp: bdd-above-def intro: le-supI1 le-supI2*)

lemma *bdd-below-image-inf*[*simp*]:
 $\text{bdd-below } ((\lambda x. \text{inf } (f \ x) \ (g \ x)) \ 'A) \longleftrightarrow \text{bdd-below } (f \ 'A) \wedge \text{bdd-below } (g \ 'A)$
by (*auto simp: bdd-below-def intro: le-infI1 le-infI2*)

lemma *bdd-below-UN*[*simp*]: $\text{finite } I \implies \text{bdd-below } (\bigcup i \in I. A \ i) = (\forall i \in I. \text{bdd-below } (A \ i))$
by (*induction I rule: finite.induct*) *auto*

lemma *bdd-above-UN*[*simp*]: $\text{finite } I \implies \text{bdd-above } (\bigcup i \in I. A \ i) = (\forall i \in I.$

bdd-above (*A i*)
by (*induction I rule: finite.induct*) *auto*

end

To avoid name classes with the *complete-lattice*-class we prefix *Sup* and *Inf* in theorem names with *c*.

92.3 Conditionally complete lattices

class *conditionally-complete-lattice* = *lattice* + *Sup* + *Inf* +
assumes *cInf-lower*: $x \in X \implies \text{bdd-below } X \implies \text{Inf } X \leq x$
and *cInf-greatest*: $X \neq \{\} \implies (\bigwedge x. x \in X \implies z \leq x) \implies z \leq \text{Inf } X$
assumes *cSup-upper*: $x \in X \implies \text{bdd-above } X \implies x \leq \text{Sup } X$
and *cSup-least*: $X \neq \{\} \implies (\bigwedge x. x \in X \implies x \leq z) \implies \text{Sup } X \leq z$
begin

lemma *cSup-upper2*: $x \in X \implies y \leq x \implies \text{bdd-above } X \implies y \leq \text{Sup } X$
by (*metis cSup-upper order-trans*)

lemma *cInf-lower2*: $x \in X \implies x \leq y \implies \text{bdd-below } X \implies \text{Inf } X \leq y$
by (*metis cInf-lower order-trans*)

lemma *cSup-mono*: $B \neq \{\} \implies \text{bdd-above } A \implies (\bigwedge b. b \in B \implies \exists a \in A. b \leq a) \implies \text{Sup } B \leq \text{Sup } A$
by (*metis cSup-least cSup-upper2*)

lemma *cInf-mono*: $B \neq \{\} \implies \text{bdd-below } A \implies (\bigwedge b. b \in B \implies \exists a \in A. a \leq b) \implies \text{Inf } A \leq \text{Inf } B$
by (*metis cInf-greatest cInf-lower2*)

lemma *cSup-subset-mono*: $A \neq \{\} \implies \text{bdd-above } B \implies A \subseteq B \implies \text{Sup } A \leq \text{Sup } B$
by (*metis cSup-least cSup-upper subsetD*)

lemma *cInf-superset-mono*: $A \neq \{\} \implies \text{bdd-below } B \implies A \subseteq B \implies \text{Inf } B \leq \text{Inf } A$
by (*metis cInf-greatest cInf-lower subsetD*)

lemma *cSup-eq-maximum*: $z \in X \implies (\bigwedge x. x \in X \implies x \leq z) \implies \text{Sup } X = z$
by (*intro order.antisym cSup-upper[of z X] cSup-least[of X z]*) *auto*

lemma *cInf-eq-minimum*: $z \in X \implies (\bigwedge x. x \in X \implies z \leq x) \implies \text{Inf } X = z$
by (*intro order.antisym cInf-lower[of z X] cInf-greatest[of X z]*) *auto*

lemma *cSup-le-iff*: $S \neq \{\} \implies \text{bdd-above } S \implies \text{Sup } S \leq a \longleftrightarrow (\forall x \in S. x \leq a)$
by (*metis order-trans cSup-upper cSup-least*)

lemma *le-cInf-iff*: $S \neq \{\} \implies \text{bdd-below } S \implies a \leq \text{Inf } S \longleftrightarrow (\forall x \in S. a \leq x)$

by (*metis order-trans cInf-lower cInf-greatest*)

lemma *cSup-eq-non-empty*:

assumes 1: $X \neq \{\}$

assumes 2: $\bigwedge x. x \in X \implies x \leq a$

assumes 3: $\bigwedge y. (\bigwedge x. x \in X \implies x \leq y) \implies a \leq y$

shows $\text{Sup } X = a$

by (*intro 3 1 order.antisym cSup-least*) (*auto intro: 2 1 cSup-upper*)

lemma *cInf-eq-non-empty*:

assumes 1: $X \neq \{\}$

assumes 2: $\bigwedge x. x \in X \implies a \leq x$

assumes 3: $\bigwedge y. (\bigwedge x. x \in X \implies y \leq x) \implies y \leq a$

shows $\text{Inf } X = a$

by (*intro 3 1 order.antisym cInf-greatest*) (*auto intro: 2 1 cInf-lower*)

lemma *cInf-cSup*: $S \neq \{\} \implies \text{bdd-below } S \implies \text{Inf } S = \text{Sup } \{x. \forall s \in S. x \leq s\}$

by (*rule cInf-eq-non-empty*) (*auto intro!: cSup-upper cSup-least simp: bdd-below-def*)

lemma *cSup-cInf*: $S \neq \{\} \implies \text{bdd-above } S \implies \text{Sup } S = \text{Inf } \{x. \forall s \in S. s \leq x\}$

by (*rule cSup-eq-non-empty*) (*auto intro!: cInf-lower cInf-greatest simp: bdd-above-def*)

lemma *cSup-insert*: $X \neq \{\} \implies \text{bdd-above } X \implies \text{Sup } (\text{insert } a \ X) = \text{sup } a \ (\text{Sup } X)$

by (*intro cSup-eq-non-empty*) (*auto intro: le-supI2 cSup-upper cSup-least*)

lemma *cInf-insert*: $X \neq \{\} \implies \text{bdd-below } X \implies \text{Inf } (\text{insert } a \ X) = \text{inf } a \ (\text{Inf } X)$

by (*intro cInf-eq-non-empty*) (*auto intro: le-infI2 cInf-lower cInf-greatest*)

lemma *cSup-singleton [simp]*: $\text{Sup } \{x\} = x$

by (*intro cSup-eq-maximum*) *auto*

lemma *cInf-singleton [simp]*: $\text{Inf } \{x\} = x$

by (*intro cInf-eq-minimum*) *auto*

lemma *cSup-insert-If*: $\text{bdd-above } X \implies \text{Sup } (\text{insert } a \ X) = (\text{if } X = \{\} \text{ then } a \text{ else } \text{sup } a \ (\text{Sup } X))$

using *cSup-insert[of X]* **by** *simp*

lemma *cInf-insert-If*: $\text{bdd-below } X \implies \text{Inf } (\text{insert } a \ X) = (\text{if } X = \{\} \text{ then } a \text{ else } \text{inf } a \ (\text{Inf } X))$

using *cInf-insert[of X]* **by** *simp*

lemma *le-cSup-finite*: $\text{finite } X \implies x \in X \implies x \leq \text{Sup } X$

proof (*induct X arbitrary: x rule: finite-induct*)

case (*insert x X y*) **then show** *?case*

by (*cases X = \{\}*) (*auto simp: cSup-insert intro: le-supI2*)

qed *simp*

lemma *cInf-le-finite*: $\text{finite } X \implies x \in X \implies \text{Inf } X \leq x$

proof (*induct* X *arbitrary*: x *rule*: *finite-induct*)

case (*insert* x X y) **then show** *?case*

by (*cases* $X = \{\}$) (*auto simp*: *cInf-insert intro*: *le-infI2*)

qed *simp*

lemma *cSup-eq-Sup-fin*: $\text{finite } X \implies X \neq \{\} \implies \text{Sup } X = \text{Sup-fin } X$

by (*induct* X *rule*: *finite-ne-induct*) (*simp-all add*: *cSup-insert*)

lemma *cInf-eq-Inf-fin*: $\text{finite } X \implies X \neq \{\} \implies \text{Inf } X = \text{Inf-fin } X$

by (*induct* X *rule*: *finite-ne-induct*) (*simp-all add*: *cInf-insert*)

lemma *cSup-atMost[*simp*]*: $\text{Sup } \{..x\} = x$

by (*auto intro!*: *cSup-eq-maximum*)

lemma *cSup-greaterThanAtMost[*simp*]*: $y < x \implies \text{Sup } \{y<..x\} = x$

by (*auto intro!*: *cSup-eq-maximum*)

lemma *cSup-atLeastAtMost[*simp*]*: $y \leq x \implies \text{Sup } \{y..x\} = x$

by (*auto intro!*: *cSup-eq-maximum*)

lemma *cInf-atLeast[*simp*]*: $\text{Inf } \{x.. \} = x$

by (*auto intro!*: *cInf-eq-minimum*)

lemma *cInf-atLeastLessThan[*simp*]*: $y < x \implies \text{Inf } \{y..<x\} = y$

by (*auto intro!*: *cInf-eq-minimum*)

lemma *cInf-atLeastAtMost[*simp*]*: $y \leq x \implies \text{Inf } \{y..x\} = y$

by (*auto intro!*: *cInf-eq-minimum*)

lemma *cINF-lower*: $\text{bdd-below } (f \text{ ' } A) \implies x \in A \implies \bigcap (f \text{ ' } A) \leq f x$

using *cInf-lower [of - $f \text{ ' } A$]* **by** *simp*

lemma *cINF-greatest*: $A \neq \{\} \implies (\bigwedge x. x \in A \implies m \leq f x) \implies m \leq \bigcap (f \text{ ' } A)$

using *cInf-greatest [of $f \text{ ' } A$]* **by** *auto*

lemma *cSUP-upper*: $x \in A \implies \text{bdd-above } (f \text{ ' } A) \implies f x \leq \bigcup (f \text{ ' } A)$

using *cSup-upper [of - $f \text{ ' } A$]* **by** *simp*

lemma *cSUP-least*: $A \neq \{\} \implies (\bigwedge x. x \in A \implies f x \leq M) \implies \bigcup (f \text{ ' } A) \leq M$

using *cSup-least [of $f \text{ ' } A$]* **by** *auto*

lemma *cINF-lower2*: $\text{bdd-below } (f \text{ ' } A) \implies x \in A \implies f x \leq u \implies \bigcap (f \text{ ' } A) \leq u$

by (*auto intro*: *cINF-lower order-trans*)

lemma *cSUP-upper2*: $\text{bdd-above } (f \text{ ' } A) \implies x \in A \implies u \leq f x \implies u \leq \bigcup (f \text{ ' } A)$

by (*auto intro*: *cSUP-upper order-trans*)

lemma *cSUP-const [*simp*]*: $A \neq \{\} \implies (\bigcup x \in A. c) = c$

by (*intro order.antisym cSUP-least*) (*auto intro: cSUP-upper*)

lemma *cINF-const* [*simp*]: $A \neq \{\}$ $\implies (\prod_{x \in A}. c) = c$
by (*intro order.antisym cINF-greatest*) (*auto intro: cINF-lower*)

lemma *le-cINF-iff*: $A \neq \{\}$ $\implies \text{bdd-below } (f \text{ ' } A) \implies u \leq \prod (f \text{ ' } A) \longleftrightarrow (\forall x \in A. u \leq f x)$
by (*metis cINF-greatest cINF-lower order-trans*)

lemma *cSUP-le-iff*: $A \neq \{\}$ $\implies \text{bdd-above } (f \text{ ' } A) \implies \bigsqcup (f \text{ ' } A) \leq u \longleftrightarrow (\forall x \in A. f x \leq u)$
by (*metis cSUP-least cSUP-upper order-trans*)

lemma *less-cINF-D*: $\text{bdd-below } (f \text{ ' } A) \implies y < (\prod_{i \in A}. f i) \implies i \in A \implies y < f i$
by (*metis cINF-lower less-le-trans*)

lemma *cSUP-lessD*: $\text{bdd-above } (f \text{ ' } A) \implies (\bigsqcup_{i \in A}. f i) < y \implies i \in A \implies f i < y$
by (*metis cSUP-upper le-less-trans*)

lemma *cINF-insert*: $A \neq \{\}$ $\implies \text{bdd-below } (f \text{ ' } A) \implies \prod (f \text{ ' } \text{insert } a \text{ } A) = \text{inf } (f a) (\prod (f \text{ ' } A))$
by (*simp add: cInf-insert*)

lemma *cSUP-insert*: $A \neq \{\}$ $\implies \text{bdd-above } (f \text{ ' } A) \implies \bigsqcup (f \text{ ' } \text{insert } a \text{ } A) = \text{sup } (f a) (\bigsqcup (f \text{ ' } A))$
by (*simp add: cSup-insert*)

lemma *cINF-mono*: $B \neq \{\}$ $\implies \text{bdd-below } (f \text{ ' } A) \implies (\bigwedge m. m \in B \implies \exists n \in A. f n \leq g m) \implies \prod (f \text{ ' } A) \leq \prod (g \text{ ' } B)$
using *cInf-mono* [*of g \text{ ' } B f \text{ ' } A*] **by** *auto*

lemma *cSUP-mono*: $A \neq \{\}$ $\implies \text{bdd-above } (g \text{ ' } B) \implies (\bigwedge n. n \in A \implies \exists m \in B. f n \leq g m) \implies \bigsqcup (f \text{ ' } A) \leq \bigsqcup (g \text{ ' } B)$
using *cSup-mono* [*of f \text{ ' } A g \text{ ' } B*] **by** *auto*

lemma *cINF-superset-mono*: $A \neq \{\}$ $\implies \text{bdd-below } (g \text{ ' } B) \implies A \subseteq B \implies (\bigwedge x. x \in B \implies g x \leq f x) \implies \prod (g \text{ ' } B) \leq \prod (f \text{ ' } A)$
by (*rule cINF-mono*) *auto*

lemma *cSUP-subset-mono*:
 $\llbracket A \neq \{\}; \text{bdd-above } (g \text{ ' } B); A \subseteq B; \bigwedge x. x \in A \implies f x \leq g x \rrbracket \implies \bigsqcup (f \text{ ' } A) \leq \bigsqcup (g \text{ ' } B)$
by (*rule cSUP-mono*) *auto*

lemma *less-eq-cInf-inter*: $\text{bdd-below } A \implies \text{bdd-below } B \implies A \cap B \neq \{\} \implies \text{inf } (\text{Inf } A) (\text{Inf } B) \leq \text{Inf } (A \cap B)$
by (*metis cInf-superset-mono lattice-class.inf-sup-ord(1) le-infI1*)

lemma *cSup-inter-less-eq*: $\text{bdd-above } A \implies \text{bdd-above } B \implies A \cap B \neq \{\} \implies \text{Sup } (A \cap B) \leq \text{Sup } A$

$(A \cap B) \leq \text{sup } (\text{Sup } A) (\text{Sup } B)$

by (*metis cSup-subset-mono lattice-class.inf-sup-ord(1) le-supI1*)

lemma *cInf-union-distrib*: $A \neq \{\} \implies \text{bdd-below } A \implies B \neq \{\} \implies \text{bdd-below } B \implies \text{Inf } (A \cup B) = \text{inf } (\text{Inf } A) (\text{Inf } B)$

by (*intro order.antisym le-infI cInf-greatest cInf-lower*) (*auto intro: le-infI1 le-infI2 cInf-lower*)

lemma *cINF-union*: $A \neq \{\} \implies \text{bdd-below } (f \text{ ' } A) \implies B \neq \{\} \implies \text{bdd-below } (f \text{ ' } B) \implies \prod (f \text{ ' } (A \cup B)) = \prod (f \text{ ' } A) \prod \prod (f \text{ ' } B)$

using *cInf-union-distrib [of f ' A f ' B]* **by** (*simp add: image-Un*)

lemma *cSup-union-distrib*: $A \neq \{\} \implies \text{bdd-above } A \implies B \neq \{\} \implies \text{bdd-above } B \implies \text{Sup } (A \cup B) = \text{sup } (\text{Sup } A) (\text{Sup } B)$

by (*intro order.antisym le-supI cSup-least cSup-upper*) (*auto intro: le-supI1 le-supI2 cSup-upper*)

lemma *cSUP-union*: $A \neq \{\} \implies \text{bdd-above } (f \text{ ' } A) \implies B \neq \{\} \implies \text{bdd-above } (f \text{ ' } B) \implies \sqcup (f \text{ ' } (A \cup B)) = \sqcup (f \text{ ' } A) \sqcup \sqcup (f \text{ ' } B)$

using *cSup-union-distrib [of f ' A f ' B]* **by** (*simp add: image-Un*)

lemma *cINF-inf-distrib*: $A \neq \{\} \implies \text{bdd-below } (f \text{ ' } A) \implies \text{bdd-below } (g \text{ ' } A) \implies \prod (f \text{ ' } A) \prod \prod (g \text{ ' } A) = (\prod a \in A. \text{inf } (f \text{ ' } a) (g \text{ ' } a))$

by (*intro order.antisym le-infI cINF-greatest cINF-lower2*)

(*auto intro: le-infI1 le-infI2 cINF-greatest cINF-lower le-infI*)

lemma *SUP-sup-distrib*: $A \neq \{\} \implies \text{bdd-above } (f \text{ ' } A) \implies \text{bdd-above } (g \text{ ' } A) \implies \sqcup (f \text{ ' } A) \sqcup \sqcup (g \text{ ' } A) = (\sqcup a \in A. \text{sup } (f \text{ ' } a) (g \text{ ' } a))$

by (*intro order.antisym le-supI cSUP-least cSUP-upper2*)

(*auto intro: le-supI1 le-supI2 cSUP-least cSUP-upper le-supI*)

lemma *cInf-le-cSup*:

$A \neq \{\} \implies \text{bdd-above } A \implies \text{bdd-below } A \implies \text{Inf } A \leq \text{Sup } A$

by (*auto intro!: cSup-upper2 [of SOME a. a ∈ A] intro: someI cInf-lower*)

context

fixes $f :: 'a \Rightarrow 'b :: \text{conditionally-complete-lattice}$

assumes *mono f*

begin

lemma *mono-cInf*: $\llbracket \text{bdd-below } A; A \neq \{\} \rrbracket \implies f (\text{Inf } A) \leq (\text{INF } x \in A. f \text{ ' } x)$

by (*simp add: <mono f> conditionally-complete-lattice-class.cINF-greatest cInf-lower monoD*)

lemma *mono-cSup*: $\llbracket \text{bdd-above } A; A \neq \{\} \rrbracket \implies (\text{SUP } x \in A. f \text{ ' } x) \leq f (\text{Sup } A)$

by (*simp add: <mono f> conditionally-complete-lattice-class.cSUP-least cSup-upper monoD*)

lemma *mono-cINF*: $\llbracket \text{bdd-below } (A \text{ ' } I); I \neq \{\} \rrbracket \implies f (\text{INF } i \in I. A \text{ ' } i) \leq (\text{INF } x \in I.$

$f (A x)$
by (*simp add: <mono f> conditionally-complete-lattice-class.cINF-greatest cINF-lower monoD*)

lemma *mono-cSUP*: $[[\text{bdd-above } (A\ 'I); I \neq \{\}]] \implies (\text{SUP } x \in I. f (A x)) \leq f (\text{SUP } i \in I. A i)$
by (*simp add: <mono f> conditionally-complete-lattice-class.cSUP-least cSUP-upper monoD*)

end

end

The special case of well-orderings

lemma *wellorder-InfI*:
fixes $k :: 'a :: \{\text{wellorder, conditionally-complete-lattice}\}$
assumes $k \in A$ **shows** $\text{Inf } A \in A$
using *wellorder-class.LeastI [of $\lambda x. x \in A$ k]*
by (*simp add: Least-le assms cInf-eq-minimum*)

lemma *wellorder-Inf-le1*:
fixes $k :: 'a :: \{\text{wellorder, conditionally-complete-lattice}\}$
assumes $k \in A$ **shows** $\text{Inf } A \leq k$
by (*meson Least-le assms bdd-below.I cInf-lower*)

92.4 Complete lattices

instance *complete-lattice* \subseteq *conditionally-complete-lattice*
by *standard (auto intro: Sup-upper Sup-least Inf-lower Inf-greatest)*

lemma *cSup-eq*:
fixes $a :: 'a :: \{\text{conditionally-complete-lattice, no-bot}\}$
assumes *upper*: $\bigwedge x. x \in X \implies x \leq a$
assumes *least*: $\bigwedge y. (\bigwedge x. x \in X \implies x \leq y) \implies a \leq y$
shows $\text{Sup } X = a$

proof *cases*

assume $X = \{\}$ **with** *lt-ex[of a] least* **show** *?thesis* **by** (*auto simp: less-le-not-le*)
qed (*intro cSup-eq-non-empty assms*)

lemma *cSup-unique*:
fixes $b :: 'a :: \{\text{conditionally-complete-lattice, no-bot}\}$
assumes $\bigwedge c. (\forall x \in s. x \leq c) \longleftrightarrow b \leq c$
shows $\text{Sup } s = b$
by (*metis assms cSup-eq order.refl*)

lemma *cInf-eq*:
fixes $a :: 'a :: \{\text{conditionally-complete-lattice, no-top}\}$
assumes *upper*: $\bigwedge x. x \in X \implies a \leq x$
assumes *least*: $\bigwedge y. (\bigwedge x. x \in X \implies y \leq x) \implies y \leq a$

shows $\text{Inf } X = a$
proof cases
assume $X = \{\}$ **with** $gt\text{-ex}[of\ a]$ **least show** $?thesis$ **by** (*auto simp: less-le-not-le*)
qed (*intro cInf-eq-non-empty assms*)

lemma *cInf-unique*:
fixes $b :: 'a :: \{\text{conditionally-complete-lattice, no-top}\}$
assumes $\bigwedge c. (\forall x \in s. x \geq c) \longleftrightarrow b \geq c$
shows $\text{Inf } s = b$
by (*meson assms cInf-eq order.refl*)

class *conditionally-complete-linorder* = *conditionally-complete-lattice* + *linorder*
begin

lemma *less-cSup-iff*:
 $X \neq \{\} \implies \text{bdd-above } X \implies y < \text{Sup } X \longleftrightarrow (\exists x \in X. y < x)$
by (*rule iffI*) (*metis cSup-least not-less, metis cSup-upper less-le-trans*)

lemma *cInf-less-iff*: $X \neq \{\} \implies \text{bdd-below } X \implies \text{Inf } X < y \longleftrightarrow (\exists x \in X. x < y)$
by (*rule iffI*) (*metis cInf-greatest not-less, metis cInf-lower le-less-trans*)

lemma *cINF-less-iff*: $A \neq \{\} \implies \text{bdd-below } (f'A) \implies (\bigcap i \in A. f\ i) < a \longleftrightarrow (\exists x \in A. f\ x < a)$
using *cInf-less-iff*[*of f'A*] **by** *auto*

lemma *less-cSUP-iff*: $A \neq \{\} \implies \text{bdd-above } (f'A) \implies a < (\bigcup i \in A. f\ i) \longleftrightarrow (\exists x \in A. a < f\ x)$
using *less-cSup-iff*[*of f'A*] **by** *auto*

lemma *less-cSupE*:
assumes $y < \text{Sup } X$ $X \neq \{\}$ **obtains** $x \in X$ **where** $x < y$
by (*metis cSup-least assms not-le that*)

lemma *less-cSupD*:
 $X \neq \{\} \implies z < \text{Sup } X \implies \exists x \in X. z < x$
by (*metis less-cSup-iff not-le-imp-less bdd-above-def*)

lemma *cInf-lessD*:
 $X \neq \{\} \implies \text{Inf } X < z \implies \exists x \in X. x < z$
by (*metis cInf-less-iff not-le-imp-less bdd-below-def*)

lemma *complete-interval*:
assumes $a < b$ **and** $P\ a$ **and** $\neg P\ b$
shows $\exists c. a \leq c \wedge c \leq b \wedge (\forall x. a \leq x \wedge x < c \longrightarrow P\ x) \wedge$
 $(\forall d. (\forall x. a \leq x \wedge x < d \longrightarrow P\ x) \longrightarrow d \leq c)$
proof (*rule exI* [**where** $x = \text{Sup } \{d. \forall x. a \leq x \wedge x < d \longrightarrow P\ x\}$], *safe*)
show $a \leq \text{Sup } \{d. \forall x. a \leq x \wedge x < d \longrightarrow P\ x\}$
by (*rule cSup-upper, auto simp: bdd-above-def*)
(metis <a < b> <¬ P b> linear less-le)

```

next
  show  $\text{Sup } \{d. \forall c. a \leq c \wedge c < d \longrightarrow P c\} \leq b$ 
    by (rule cSup-least)
      (use <a<b> <\neg P b> in <auto simp add: less-le-not-le>)
next
  fix  $x$ 
  assume  $x: a \leq x$  and  $lt: x < \text{Sup } \{d. \forall c. a \leq c \wedge c < d \longrightarrow P c\}$ 
  show  $P x$ 
    by (rule less-cSupE [OF lt]) (use less-le-not-le x in <auto>)
next
  fix  $d$ 
  assume  $0: \forall x. a \leq x \wedge x < d \longrightarrow P x$ 
  then have  $d \in \{d. \forall c. a \leq c \wedge c < d \longrightarrow P c\}$ 
    by auto
  moreover have  $\text{bdd-above } \{d. \forall c. a \leq c \wedge c < d \longrightarrow P c\}$ 
    unfolding bdd-above-def using <a<b> <\neg P b> linear
    by (simp add: less-le) blast
  ultimately show  $d \leq \text{Sup } \{d. \forall c. a \leq c \wedge c < d \longrightarrow P c\}$ 
    by (auto simp: cSup-upper)
qed

end

```

92.5 Instances

```

instance complete-linorder < conditionally-complete-linorder
  ..

```

```

lemma cSup-eq-Max: finite (X::'a::conditionally-complete-linorder set)  $\implies X \neq \{\} \implies \text{Sup } X = \text{Max } X$ 
  using cSup-eq-Sup-fin[of X] by (simp add: Sup-fin-Max)

```

```

lemma cInf-eq-Min: finite (X::'a::conditionally-complete-linorder set)  $\implies X \neq \{\} \implies \text{Inf } X = \text{Min } X$ 
  using cInf-eq-Inf-fin[of X] by (simp add: Inf-fin-Min)

```

```

lemma cSup-lessThan[simp]: Sup {.. $x::'a::\{conditionally-complete-linorder, no-bot, dense-linorder\}$ } =  $x$ 
  by (auto intro!: cSup-eq-non-empty intro: dense-le)

```

```

lemma cSup-greaterThanLessThan[simp]:  $y < x \implies \text{Sup } \{y <.. $x::'a::\{conditionally-complete-linorder, dense-linorder\}$ \} = x$ 
  by (auto intro!: cSup-eq-non-empty intro: dense-le-bounded)

```

```

lemma cSup-atLeastLessThan[simp]:  $y < x \implies \text{Sup } \{y..<math>x::'a::\{conditionally-complete-linorder, dense-linorder\}$ \} = x
  by (auto intro!: cSup-eq-non-empty intro: dense-le-bounded)

```

```

lemma cInf-greaterThan[simp]: Inf { $x::'a::\{conditionally-complete-linorder, no-top,$ 

```

dense-linorder } <..*}* = *x*

by (*auto intro!*: *cInf-eq-non-empty intro: dense-ge*)

lemma *cInf-greaterThanAtMost[simp]*: $y < x \implies \text{Inf } \{y < ..x :: 'a :: \{\text{conditionally-complete-linorder, dense-linorder}\}\} = y$

by (*auto intro!*: *cInf-eq-non-empty intro: dense-ge-bounded*)

lemma *cInf-greaterThanLessThan[simp]*: $y < x \implies \text{Inf } \{y < .. < x :: 'a :: \{\text{conditionally-complete-linorder, dense-linorder}\}\} = y$

by (*auto intro!*: *cInf-eq-non-empty intro: dense-ge-bounded*)

lemma *Inf-insert-finite*:

fixes *S* :: 'a :: *conditionally-complete-linorder set*

shows *finite S* $\implies \text{Inf } (\text{insert } x \text{ } S) = (\text{if } S = \{\} \text{ then } x \text{ else } \text{min } x \text{ } (\text{Inf } S))$

by (*simp add: cInf-eq-Min*)

lemma *Sup-insert-finite*:

fixes *S* :: 'a :: *conditionally-complete-linorder set*

shows *finite S* $\implies \text{Sup } (\text{insert } x \text{ } S) = (\text{if } S = \{\} \text{ then } x \text{ else } \text{max } x \text{ } (\text{Sup } S))$

by (*simp add: cSup-insert sup-max*)

lemma *finite-imp-less-Inf*:

fixes *a* :: 'a :: *conditionally-complete-linorder*

shows $\llbracket \text{finite } X; x \in X; \bigwedge x. x \in X \implies a < x \rrbracket \implies a < \text{Inf } X$

by (*induction X rule: finite-induct*) (*simp-all add: cInf-eq-Min Inf-insert-finite*)

lemma *finite-less-Inf-iff*:

fixes *a* :: 'a :: *conditionally-complete-linorder*

shows $\llbracket \text{finite } X; X \neq \{\} \rrbracket \implies a < \text{Inf } X \longleftrightarrow (\forall x \in X. a < x)$

by (*auto simp: cInf-eq-Min*)

lemma *finite-imp-Sup-less*:

fixes *a* :: 'a :: *conditionally-complete-linorder*

shows $\llbracket \text{finite } X; x \in X; \bigwedge x. x \in X \implies a > x \rrbracket \implies a > \text{Sup } X$

by (*induction X rule: finite-induct*) (*simp-all add: cSup-eq-Max Sup-insert-finite*)

lemma *finite-Sup-less-iff*:

fixes *a* :: 'a :: *conditionally-complete-linorder*

shows $\llbracket \text{finite } X; X \neq \{\} \rrbracket \implies a > \text{Sup } X \longleftrightarrow (\forall x \in X. a > x)$

by (*auto simp: cSup-eq-Max*)

class *linear-continuum* = *conditionally-complete-linorder* + *dense-linorder* +

assumes *UNIV-not-singleton*: $\exists a b :: 'a. a \neq b$

begin

lemma *ex-gt-or-lt*: $\exists b. a < b \vee b < a$

by (*metis UNIV-not-singleton neq-iff*)

end

context

fixes $f::'a \Rightarrow 'b::\{\text{conditionally-complete-linorder, ordered-ab-group-add}\}$
begin

lemma *bdd-above-uminus-image*: $\text{bdd-above } ((\lambda x. - f x) ' A) \longleftrightarrow \text{bdd-below } (f ' A)$

by (*metis bdd-above-uminus image-image*)

lemma *bdd-below-uminus-image*: $\text{bdd-below } ((\lambda x. - f x) ' A) \longleftrightarrow \text{bdd-above } (f ' A)$

by (*metis bdd-below-uminus image-image*)

lemma *uminus-cSUP*:

assumes $\text{bdd-above } (f ' A) \ A \neq \{\}$

shows $-(\text{SUP } x \in A. f x) = (\text{INF } x \in A. - f x)$

proof (*rule antisym*)

show $(\text{INF } x \in A. - f x) \leq - \text{Sup } (f ' A)$

by (*metis cINF-lower cSUP-least bdd-below-uminus-image assms le-minus-iff*)

have $*: \bigwedge x. x \in A \implies f x \leq \text{Sup } (f ' A)$

by (*simp add: assms cSup-upper*)

then show $-\text{Sup } (f ' A) \leq (\text{INF } x \in A. - f x)$

by (*simp add: assms cINF-greatest*)

qed

end

context

fixes $f::'a \Rightarrow 'b::\{\text{conditionally-complete-linorder, ordered-ab-group-add}\}$
begin

lemma *uminus-cINF*:

assumes $\text{bdd-below } (f ' A) \ A \neq \{\}$

shows $-(\text{INF } x \in A. f x) = (\text{SUP } x \in A. - f x)$

by (*metis (mono-tags, lifting) INF-cong uminus-cSUP assms bdd-above-uminus-image minus-equation-iff*)

lemma *Sup-add-eq*:

assumes $\text{bdd-above } (f ' A) \ A \neq \{\}$

shows $(\text{SUP } x \in A. a + f x) = a + (\text{SUP } x \in A. f x)$ (**is** $?L=?R$)

proof (*rule antisym*)

have *bdd*: $\text{bdd-above } ((\lambda x. a + f x) ' A)$

by (*metis assms bdd-above-image-mono image-image mono-add*)

with *assms* **show** $?L \leq ?R$

by (*simp add: assms cSup-le-iff cSUP-upper*)

have $\bigwedge x. x \in A \implies f x \leq (\text{SUP } x \in A. a + f x) - a$

by (*simp add: bdd cSup-upper le-diff-eq*)

with $\langle A \neq \{\} \rangle$ **have** $\bigsqcup (f ' A) \leq (\bigsqcup x \in A. a + f x) - a$

```

  by (simp add: cSUP-least)
  then show ?R ≤ ?L
    by (metis add commute le-diff-eq)
qed

```

```

lemma Inf-add-eq: — you don't get a shorter proof by duality
  assumes bdd-below (f ' A) A ≠ {}
  shows (INF x∈A. a + f x) = a + (INF x∈A. f x) (is ?L=?R)
proof (rule antisym)
  show ?R ≤ ?L
    using assms mono-add mono-cINF by blast
  have bdd: bdd-below ((λx. a + f x) ' A)
    by (metis add-left-mono assms(1) bdd-below.E bdd-below.I2 imageI)
  with assms have  $\bigwedge x. x \in A \implies f x \geq (\text{INF } x \in A. a + f x) - a$ 
    by (simp add: cInf-lower diff-le-eq)
  with ⟨A ≠ {}⟩ have  $(\bigcap x \in A. a + f x) - a \leq \bigcap (f ' A)$ 
    by (simp add: cINF-greatest)
  with assms show ?L ≤ ?R
    by (metis add commute diff-le-eq)
qed

```

end

```

instantiation nat :: conditionally-complete-linorder
begin

```

```

definition Sup (X::nat set) = (if X={ } then 0 else Max X)
definition Inf (X::nat set) = (LEAST n. n ∈ X)

```

```

lemma bdd-above-nat: bdd-above X  $\longleftrightarrow$  finite (X::nat set)
proof
  assume bdd-above X
  then obtain z where X ⊆ {.. z}
    by (auto simp: bdd-above-def)
  then show finite X
    by (rule finite-subset) simp
qed simp

```

```

instance

```

```

proof
  fix x :: nat
  fix X :: nat set
  show Inf X ≤ x if x ∈ X bdd-below X
    using that by (simp add: Inf-nat-def Least-le)
  show x ≤ Inf X if X ≠ { }  $\bigwedge y. y \in X \implies x \leq y$ 
    using that unfolding Inf-nat-def ex-in-conv[symmetric] by (rule LeastI2-ex)
  show x ≤ Sup X if x ∈ X bdd-above X
    using that by (auto simp add: Sup-nat-def bdd-above-nat)
  show Sup X ≤ x if X ≠ { }  $\bigwedge y. y \in X \implies y \leq x$ 

```

```

proof –
  from that have bdd-above X
    by (auto simp: bdd-above-def)
  with that show ?thesis
    by (simp add: Sup-nat-def bdd-above-nat)
qed
qed

end

```

```

lemma Inf-nat-def1:
  fixes K::nat set
  assumes K ≠ {}
  shows Inf K ∈ K
by (auto simp add: Min-def Inf-nat-def) (meson LeastI assms bot.extremum-unique subsetI)

```

```

lemma Sup-nat-empty [simp]: Sup {} = (0::nat)
  by (auto simp add: Sup-nat-def)

```

```

instantiation int :: conditionally-complete-linorder
begin

```

```

definition Sup (X::int set) = (THE x. x ∈ X ∧ (∀ y∈X. y ≤ x))
definition Inf (X::int set) = – (Sup (uminus ‘ X))

```

```

instance

```

```

proof
  { fix x :: int and X :: int set assume X ≠ {} bdd-above X
    then obtain x y where X ⊆ {..y} x ∈ X
      by (auto simp: bdd-above-def)
    then have *: finite (X ∩ {x..y}) X ∩ {x..y} ≠ {} and x ≤ y
      by (auto simp: subset-eq)
    have ∃!x∈X. (∀ y∈X. y ≤ x)
      proof
        { fix z assume z ∈ X
          have z ≤ Max (X ∩ {x..y})
            proof cases
              assume x ≤ z with ⟨z ∈ X⟩ ⟨X ⊆ {..y}⟩ *(1) show ?thesis
                by (auto intro!: Max-ge)
            next
              assume ¬ x ≤ z
              then have z < x by simp
              also have x ≤ Max (X ∩ {x..y})
                using ⟨x ∈ X⟩ *(1) ⟨x ≤ y⟩ by (intro Max-ge) auto
              finally show ?thesis by simp
            qed }
        }
      }
  }

```

```

note  $le = this$ 
with  $Max\text{-in}[OF *]$  show  $ex: Max (X \cap \{x..y\}) \in X \wedge (\forall z \in X. z \leq Max (X \cap \{x..y\}))$  by  $auto$ 

fix  $z$  assume  $z: z \in X \wedge (\forall y \in X. y \leq z)$ 
with  $le$  have  $z \leq Max (X \cap \{x..y\})$ 
by  $auto$ 
moreover have  $Max (X \cap \{x..y\}) \leq z$ 
using  $* ex$  by  $auto$ 
ultimately show  $z = Max (X \cap \{x..y\})$ 
by  $auto$ 
qed
then have  $Sup X \in X \wedge (\forall y \in X. y \leq Sup X)$ 
unfolding  $Sup\text{-int-def}$  by  $(rule\ theI')$  }
note  $Sup\text{-int} = this$ 

{ fix  $x :: int$  and  $X :: int\ set$  assume  $x \in X$   $bdd\text{-above}\ X$  then show  $x \leq Sup X$ 
using  $Sup\text{-int}[of\ X]$  by  $auto$  }
note  $le\text{-Sup} = this$ 
{ fix  $x :: int$  and  $X :: int\ set$  assume  $X \neq \{\}$   $\wedge y. y \in X \implies y \leq x$  then show  $Sup X \leq x$ 
using  $Sup\text{-int}[of\ X]$  by  $(auto\ simp: bdd\text{-above}\text{-def})$  }
note  $Sup\text{-le} = this$ 

{ fix  $x :: int$  and  $X :: int\ set$  assume  $x \in X$   $bdd\text{-below}\ X$  then show  $Inf X \leq x$ 
using  $le\text{-Sup}[of\ -x\ uminus\ 'X]$  by  $(auto\ simp: Inf\text{-int}\text{-def})$  }
{ fix  $x :: int$  and  $X :: int\ set$  assume  $X \neq \{\}$   $\wedge y. y \in X \implies x \leq y$  then show  $x \leq Inf X$ 
using  $Sup\text{-le}[of\ uminus\ 'X\ -x]$  by  $(force\ simp: Inf\text{-int}\text{-def})$  }
qed
end

```

lemma *interval-cases:*

```

fixes  $S :: 'a :: conditionally\text{-complete}\text{-linorder}\ set$ 
assumes  $ivl: \bigwedge a\ b\ x. a \in S \implies b \in S \implies a \leq x \implies x \leq b \implies x \in S$ 
shows  $\exists a\ b. S = \{\} \vee$ 
 $S = UNIV \vee$ 
 $S = \{..<b\} \vee$ 
 $S = \{..b\} \vee$ 
 $S = \{a<..\} \vee$ 
 $S = \{a..\} \vee$ 
 $S = \{a<..
 $S = \{a<..b\} \vee$ 
 $S = \{a..
 $S = \{a..b\}$$$ 
```

proof –

```

define  $lower\ upper$  where  $lower = \{x. \exists s \in S. s \leq x\}$  and  $upper = \{x. \exists s \in S. x \leq s\}$ 

```

```

with ivl have  $S = lower \cap upper$ 
  by auto
moreover
have  $\exists a. upper = UNIV \vee upper = \{\} \vee upper = \{.. a\} \vee upper = \{.. < a\}$ 
proof cases
  assume *: bdd-above  $S \wedge S \neq \{\}$ 
  from * have  $upper \subseteq \{.. Sup S\}$ 
    by (auto simp: upper-def intro: cSup-upper2)
  moreover from * have  $\{.. < Sup S\} \subseteq upper$ 
    by (force simp add: less-cSup-iff upper-def subset-eq Ball-def)
  ultimately have  $upper = \{.. Sup S\} \vee upper = \{.. < Sup S\}$ 
    unfolding ivl-disj-un(2)[symmetric] by auto
  then show ?thesis by auto
next
  assume  $\neg (bdd-above S \wedge S \neq \{\})$ 
  then have  $upper = UNIV \vee upper = \{\}$ 
    by (auto simp: upper-def bdd-above-def not-le dest: less-imp-le)
  then show ?thesis
    by auto
qed
moreover
have  $\exists b. lower = UNIV \vee lower = \{\} \vee lower = \{b ..\} \vee lower = \{b < ..\}$ 
proof cases
  assume *: bdd-below  $S \wedge S \neq \{\}$ 
  from * have  $lower \subseteq \{Inf S ..\}$ 
    by (auto simp: lower-def intro: cInf-lower2)
  moreover from * have  $\{Inf S < ..\} \subseteq lower$ 
    by (force simp add: cInf-less-iff lower-def subset-eq Ball-def)
  ultimately have  $lower = \{Inf S ..\} \vee lower = \{Inf S < ..\}$ 
    unfolding ivl-disj-un(1)[symmetric] by auto
  then show ?thesis by auto
next
  assume  $\neg (bdd-below S \wedge S \neq \{\})$ 
  then have  $lower = UNIV \vee lower = \{\}$ 
    by (auto simp: lower-def bdd-below-def not-le dest: less-imp-le)
  then show ?thesis
    by auto
qed
ultimately show ?thesis
  unfolding greaterThanAtMost-def greaterThanLessThan-def atLeastAtMost-def
atLeastLessThan-def
  by (metis inf-bot-left inf-bot-right inf-top.left-neutral inf-top.right-neutral)
qed

lemma cSUP-eq-cINF-D:
fixes  $f :: - \Rightarrow 'b::conditionally-complete-lattice$ 
assumes eq:  $(\bigsqcup x \in A. f x) = (\bigsqcap x \in A. f x)$ 
  and bdd: bdd-above  $(f ' A)$  bdd-below  $(f ' A)$ 
  and  $a: a \in A$ 

```

shows $f a = (\prod_{x \in A}. f x)$
proof (*rule antisym*)
show $f a \leq \prod (f ' A)$
by (*metis a bdd(1) eq cSUP-upper*)
show $\prod (f ' A) \leq f a$
using *a bdd by (auto simp: cINF-lower)*
qed

lemma *cSUP-UNION*:

fixes $f :: - \Rightarrow 'b::\text{conditionally-complete-lattice}$
assumes $ne: A \neq \{\} \wedge x. x \in A \Longrightarrow B(x) \neq \{\}$
and $bdd\text{-}UN: bdd\text{-}above (\bigcup_{x \in A}. f ' B x)$
shows $(\bigsqcup z \in \bigcup_{x \in A}. B x. f z) = (\bigsqcup_{x \in A}. \bigsqcup_{z \in B x}. f z)$
proof –
have $bdd: \bigwedge x. x \in A \Longrightarrow bdd\text{-}above (f ' B x)$
using *bdd-UN by (meson UN-upper bdd-above-mono)*
obtain M **where** $\bigwedge x y. x \in A \Longrightarrow y \in B(x) \Longrightarrow f y \leq M$
using *bdd-UN by (auto simp: bdd-above-def)*
then have $bdd2: bdd\text{-}above ((\lambda x. \bigsqcup_{z \in B x}. f z) ' A)$
unfolding *bdd-above-def by (force simp: bdd cSUP-le-iff ne(2))*
have $(\bigsqcup z \in \bigcup_{x \in A}. B x. f z) \leq (\bigsqcup_{x \in A}. \bigsqcup_{z \in B x}. f z)$
using *assms by (fastforce simp add: intro!: cSUP-least intro: cSUP-upper2 simp: bdd2 bdd)*
moreover have $(\bigsqcup_{x \in A}. \bigsqcup_{z \in B x}. f z) \leq (\bigsqcup z \in \bigcup_{x \in A}. B x. f z)$
using *assms by (fastforce simp add: intro!: cSUP-least intro: cSUP-upper simp: image-UN bdd-UN)*
ultimately show *?thesis*
by (*rule order-antisym*)
qed

lemma *cINF-UNION*:

fixes $f :: - \Rightarrow 'b::\text{conditionally-complete-lattice}$
assumes $ne: A \neq \{\} \wedge x. x \in A \Longrightarrow B(x) \neq \{\}$
and $bdd\text{-}UN: bdd\text{-}below (\bigcup_{x \in A}. f ' B x)$
shows $(\prod z \in \bigcup_{x \in A}. B x. f z) = (\prod_{x \in A}. \prod_{z \in B x}. f z)$
proof –
have $bdd: \bigwedge x. x \in A \Longrightarrow bdd\text{-}below (f ' B x)$
using *bdd-UN by (meson UN-upper bdd-below-mono)*
obtain M **where** $\bigwedge x y. x \in A \Longrightarrow y \in B(x) \Longrightarrow f y \geq M$
using *bdd-UN by (auto simp: bdd-below-def)*
then have $bdd2: bdd\text{-}below ((\lambda x. \prod_{z \in B x}. f z) ' A)$
unfolding *bdd-below-def by (force simp: bdd le-cINF-iff ne(2))*
have $(\prod z \in \bigcup_{x \in A}. B x. f z) \leq (\prod_{x \in A}. \prod_{z \in B x}. f z)$
using *assms by (fastforce simp add: intro!: cINF-greatest intro: cINF-lower simp: bdd2 bdd)*
moreover have $(\prod_{x \in A}. \prod_{z \in B x}. f z) \leq (\prod z \in \bigcup_{x \in A}. B x. f z)$
using *assms by (fastforce simp add: intro!: cINF-greatest intro: cINF-lower2 simp: bdd bdd-UN bdd2)*
ultimately show *?thesis*

by (rule order-antisym)
qed

lemma *cSup-abs-le*:

fixes $S :: ('a::\{\text{linordered-idom, conditionally-complete-linorder}\}) \text{ set}$
shows $S \neq \{\} \implies (\bigwedge x. x \in S \implies |x| \leq a) \implies |\text{Sup } S| \leq a$
apply (auto simp add: abs-le-iff intro: cSup-least)
by (metis bdd-aboveI cSup-upper neg-le-iff-le order-trans)

end

93 Factorial Function, Rising Factorials

theory *Factorial*
imports *Groups-List*
begin

93.1 Factorial Function

context *semiring-char-0*
begin

definition *fact* :: $\text{nat} \Rightarrow 'a$
where *fact-prod*: $\text{fact } n = \text{of-nat } (\prod \{1..n\})$

lemma *fact-prod-Suc*: $\text{fact } n = \text{of-nat } (\text{prod } \text{Suc } \{0..<n\})$
unfolding *fact-prod* using *atLeast0LessThan prod.atLeast1-atMost-eq* by auto

lemma *fact-prod-rev*: $\text{fact } n = \text{of-nat } (\prod i = 0..<n. n - i)$
proof –
have $\text{prod } \text{Suc } \{0..<n\} = \prod \{1..n\}$
by (simp add: *atLeast0LessThan prod.atLeast1-atMost-eq*)
then have $\text{prod } \text{Suc } \{0..<n\} = \text{prod } ((-) (n + 1)) \{1..n\}$
using *prod.atLeastAtMost-rev* [of $\lambda i. i - 1$ n] by *presburger*
then show ?thesis
unfolding *fact-prod-Suc* by (simp add: *atLeast0LessThan prod.atLeast1-atMost-eq*)
qed

lemma *fact-0* [simp]: $\text{fact } 0 = 1$
by (simp add: *fact-prod*)

lemma *fact-1* [simp]: $\text{fact } 1 = 1$
by (simp add: *fact-prod*)

lemma *fact-Suc-0* [simp]: $\text{fact } (\text{Suc } 0) = 1$
by (simp add: *fact-prod*)

lemma *fact-Suc* [simp]: $\text{fact } (\text{Suc } n) = \text{of-nat } (\text{Suc } n) * \text{fact } n$
by (simp add: *fact-prod atLeastAtMostSuc-conv algebra-simps*)

```

lemma fact-2 [simp]: fact 2 = 2
  by (simp add: numeral-2-eq-2)

lemma fact-split:  $k \leq n \implies \text{fact } n = \text{of-nat } (\text{prod } \text{Suc } \{n - k..<n\}) * \text{fact } (n - k)$ 
  by (simp add: fact-prod-Suc prod.union-disjoint [symmetric]
    inv-disj-un ac-simps of-nat-mult [symmetric])

end

lemma of-nat-fact [simp]: of-nat (fact n) = fact n
  by (simp add: fact-prod)

lemma of-int-fact [simp]: of-int (fact n) = fact n
  by (simp only: fact-prod of-int-of-nat-eq)

lemma fact-reduce:  $n > 0 \implies \text{fact } n = \text{of-nat } n * \text{fact } (n - 1)$ 
  by (cases n) auto

lemma fact-nonzero [simp]:  $\text{fact } n \neq (0 :: 'a :: \{\text{semiring-char-0}, \text{semiring-no-zero-divisors}\})$ 
  using of-nat-0-neq by (induct n) auto

lemma fact-mono-nat:  $m \leq n \implies \text{fact } m \leq (\text{fact } n :: \text{nat})$ 
  by (induct n) (auto simp: le-Suc-eq)

lemma fact-in-Nats:  $\text{fact } n \in \mathbb{N}$ 
  by (induct n) auto

lemma fact-in-Ints:  $\text{fact } n \in \mathbb{Z}$ 
  by (induct n) auto

context
  assumes SORT-CONSTRAINT('a::linordered-semidom)
begin

lemma fact-mono:  $m \leq n \implies \text{fact } m \leq (\text{fact } n :: 'a)$ 
  by (metis of-nat-fact of-nat-le-iff fact-mono-nat)

lemma fact-ge-1 [simp]:  $\text{fact } n \geq (1 :: 'a)$ 
  by (metis le0 fact-0 fact-mono)

lemma fact-gt-zero [simp]:  $\text{fact } n > (0 :: 'a)$ 
  using fact-ge-1 less-le-trans zero-less-one by blast

lemma fact-ge-zero [simp]:  $\text{fact } n \geq (0 :: 'a)$ 
  by (simp add: less-imp-le)

lemma fact-not-neg [simp]:  $\neg \text{fact } n < (0 :: 'a)$ 

```


by (simp add: not-less-iff-gr-or-eq)

lemma *fact-le-power*: $\text{fact } n \leq (\text{of-nat } (n \hat{=} n) :: 'a)$

proof (induct n)

case 0

then show ?case by simp

next

case (Suc n)

then have *: $\text{fact } n \leq (\text{of-nat } (Suc\ n \hat{=} n) :: 'a)$

by (rule order-trans) (simp add: power-mono del: of-nat-power)

have $\text{fact } (Suc\ n) = (\text{of-nat } (Suc\ n) * \text{fact } n :: 'a)$

by (simp add: algebra-simps)

also have $\dots \leq \text{of-nat } (Suc\ n) * \text{of-nat } (Suc\ n \hat{=} n)$

by (simp add: * ordered-comm-semiring-class.comm-mult-left-mono del: of-nat-power)

also have $\dots \leq \text{of-nat } (Suc\ n \hat{=} Suc\ n)$

by (metis of-nat-mult order-refl power-Suc)

finally show ?case .

qed

end

lemma *fact-less-mono-nat*: $0 < m \implies m < n \implies \text{fact } m < (\text{fact } n :: \text{nat})$

by (induct n) (auto simp: less-Suc-eq)

lemma *fact-less-mono*: $0 < m \implies m < n \implies \text{fact } m < (\text{fact } n :: 'a::\text{linordered-semidom})$

by (metis of-nat-fact of-nat-less-iff fact-less-mono-nat)

lemma *fact-ge-Suc-0-nat* [simp]: $\text{fact } n \geq \text{Suc } 0$

by (metis One-nat-def fact-ge-1)

lemma *dvd-fact*: $1 \leq m \implies m \leq n \implies m \text{ dvd fact } n$

by (induct n) (auto simp: dvdI le-Suc-eq)

lemma *fact-ge-self*: $\text{fact } n \geq n$

by (cases n = 0) (simp-all add: dvd-imp-le dvd-fact)

lemma *fact-dvd*: $n \leq m \implies \text{fact } n \text{ dvd } (\text{fact } m :: 'a::\text{linordered-semidom})$

by (induct m) (auto simp: le-Suc-eq)

lemma *fact-mod*: $m \leq n \implies \text{fact } n \text{ mod } (\text{fact } m :: 'a::\{\text{semidom-modulo}, \text{linordered-semidom}\}) = 0$

by (simp add: mod-eq-0-iff-dvd fact-dvd)

lemma *fact-eq-fact-times*:

assumes $m \geq n$

shows $\text{fact } m = \text{fact } n * \prod \{\text{Suc } n..m\}$

unfolding *fact-prod*

by (metis add commute assms le-add1 le-add-diff-inverse of-nat-id plus-1-eq-Suc prod.ub-add-nat)

lemma *fact-div-fact*:

assumes $m \geq n$

shows $\text{fact } m \text{ div fact } n = \prod \{n + 1..m\}$

by (*simp add: fact-eq-fact-times [OF assms]*)

lemma *fact-num-eq-if*: $\text{fact } m = (\text{if } m = 0 \text{ then } 1 \text{ else of-nat } m * \text{fact } (m - 1))$

by (*cases m*) *auto*

lemma *fact-div-fact-le-pow*:

assumes $r \leq n$

shows $\text{fact } n \text{ div fact } (n - r) \leq n \wedge r$

proof –

have $r \leq n \implies \prod \{n - r..n\} = (n - r) * \prod \{\text{Suc } (n - r)..n\}$ **for** r

by (*subst prod.insert[symmetric]*) (*auto simp: atLeastAtMost-insertL*)

with *assms* **show** *?thesis*

by (*induct r rule: nat.induct*) (*auto simp add: fact-div-fact Suc-diff-Suc mult-le-mono*)

qed

lemma *prod-Suc-fact*: $\text{prod Suc } \{0..<n\} = \text{fact } n$

by (*simp add: fact-prod-Suc*)

lemma *prod-Suc-Suc-fact*: $\text{prod Suc } \{\text{Suc } 0..<n\} = \text{fact } n$

proof (*cases n = 0*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

have $\text{prod Suc } \{\text{Suc } 0..<n\} = \text{Suc } 0 * \text{prod Suc } \{\text{Suc } 0..<n\}$

by *simp*

also have $\dots = \text{prod Suc } (\text{insert } 0 \{\text{Suc } 0..<n\})$

by *simp*

also have $\text{insert } 0 \{\text{Suc } 0..<n\} = \{0..<n\}$

using *False* **by** *auto*

finally show *?thesis*

by (*simp add: fact-prod-Suc*)

qed

lemma *fact-numeral*: $\text{fact } (\text{numeral } k) = \text{numeral } k * \text{fact } (\text{pred-numeral } k)$

— Evaluation for specific numerals

by (*metis fact-Suc numeral-eq-Suc of-nat-numeral*)

93.2 Pochhammer’s symbol: generalized rising factorial

See https://en.wikipedia.org/wiki/Pochhammer_symbol.

context *comm-semiring-1*

begin

definition *pochhammer* :: $'a \Rightarrow \text{nat} \Rightarrow 'a$

where *pochhammer-prod*: $\text{pochhammer } a \ n = \text{prod } (\lambda i. a + \text{of-nat } i) \ \{0..<n\}$

lemma *pochhammer-prod-rev*: $\text{pochhammer } a \ n = \text{prod } (\lambda i. a + \text{of-nat } (n - i)) \ \{1..n\}$
using *prod.atLeastLessThan-rev-at-least-Suc-atMost* [of $\lambda i. a + \text{of-nat } i \ 0 \ n$]
by (*simp add: pochhammer-prod*)

lemma *pochhammer-Suc-prod*: $\text{pochhammer } a \ (\text{Suc } n) = \text{prod } (\lambda i. a + \text{of-nat } i) \ \{0..n\}$
by (*simp add: pochhammer-prod atLeastLessThanSuc-atLeastAtMost*)

lemma *pochhammer-Suc-prod-rev*: $\text{pochhammer } a \ (\text{Suc } n) = \text{prod } (\lambda i. a + \text{of-nat } (n - i)) \ \{0..n\}$
using *prod.atLeast-Suc-atMost-Suc-shift*
by (*simp add: pochhammer-prod-rev prod.atLeast-Suc-atMost-Suc-shift del: prod.cl-ivl-Suc*)

lemma *pochhammer-0* [*simp*]: $\text{pochhammer } a \ 0 = 1$
by (*simp add: pochhammer-prod*)

lemma *pochhammer-1* [*simp*]: $\text{pochhammer } a \ 1 = a$
by (*simp add: pochhammer-prod lessThan-Suc*)

lemma *pochhammer-Suc0* [*simp*]: $\text{pochhammer } a \ (\text{Suc } 0) = a$
by (*simp add: pochhammer-prod lessThan-Suc*)

lemma *pochhammer-Suc*: $\text{pochhammer } a \ (\text{Suc } n) = \text{pochhammer } a \ n * (a + \text{of-nat } n)$
by (*simp add: pochhammer-prod atLeast0-lessThan-Suc ac-simps*)

end

lemma *pochhammer-nonneg*:
fixes $x :: 'a :: \text{linordered-semidom}$
shows $x > 0 \implies \text{pochhammer } x \ n \geq 0$
by (*induction n*) (*auto simp: pochhammer-Suc intro!: mult-nonneg-nonneg add-nonneg-nonneg*)

lemma *pochhammer-pos*:
fixes $x :: 'a :: \text{linordered-semidom}$
shows $x > 0 \implies \text{pochhammer } x \ n > 0$
by (*induction n*) (*auto simp: pochhammer-Suc intro!: mult-pos-pos add-pos-nonneg*)

context *comm-semiring-1*
begin

lemma *pochhammer-of-nat*: $\text{pochhammer } (\text{of-nat } x) \ n = \text{of-nat } (\text{pochhammer } x \ n)$
by (*simp add: pochhammer-prod Factorial.pochhammer-prod*)

end

context *comm-ring-1*
begin

lemma *pochhammer-of-int*: $\text{pochhammer } (\text{of-int } x) \ n = \text{of-int } (\text{pochhammer } x \ n)$
by (*simp add: pochhammer-prod Factorial.pochhammer-prod*)

end

lemma *pochhammer-rec*: $\text{pochhammer } a \ (\text{Suc } n) = a * \text{pochhammer } (a + 1) \ n$
by (*simp add: pochhammer-prod prod.atLeast0-lessThan-Suc-shift ac-simps del: prod.op-ivl-Suc*)

lemma *pochhammer-rec'*: $\text{pochhammer } z \ (\text{Suc } n) = (z + \text{of-nat } n) * \text{pochhammer } z \ n$
by (*simp add: pochhammer-prod prod.atLeast0-lessThan-Suc ac-simps*)

lemma *pochhammer-fact*: $\text{fact } n = \text{pochhammer } 1 \ n$
by (*simp add: pochhammer-prod fact-prod-Suc*)

lemma *pochhammer-of-nat-eq-0-lemma*: $k > n \implies \text{pochhammer } (- (\text{of-nat } n :: 'a:: \text{idom})) \ k = 0$
by (*auto simp add: pochhammer-prod*)

lemma *pochhammer-of-nat-eq-0-lemma'*:
assumes *kn*: $k \leq n$
shows $\text{pochhammer } (- (\text{of-nat } n :: 'a::\{\text{idom}, \text{ring-char-0}\})) \ k \neq 0$
proof (*cases k*)
case 0
then show *?thesis* **by** *simp*
next
case (*Suc h*)
then show *?thesis*
apply (*simp add: pochhammer-Suc-prod*)
using *Suc kn*
apply (*auto simp add: algebra-simps*)
done

qed

lemma *pochhammer-of-nat-eq-0-iff*:
 $\text{pochhammer } (- (\text{of-nat } n :: 'a::\{\text{idom}, \text{ring-char-0}\})) \ k = 0 \iff k > n$
(is ?l = ?r)
using *pochhammer-of-nat-eq-0-lemma*[*of n k, where ?'a='a*]
pochhammer-of-nat-eq-0-lemma'[*of k n, where ?'a = 'a*]
by (*auto simp add: not-le[symmetric]*)

lemma *pochhammer-0-left*:
 $\text{pochhammer } 0 \ n = (\text{if } n = 0 \ \text{then } 1 \ \text{else } 0)$
by (*induction n*) (*simp-all add: pochhammer-rec*)

lemma *pochhammer-eq-0-iff*: $\text{pochhammer } a \ n = (0 :: 'a :: \text{field-char-0}) \longleftrightarrow (\exists k < n. a = - \text{of-nat } k)$

by (*auto simp add: pochhammer-prod eq-neg-iff-add-eq-0*)

lemma *pochhammer-eq-0-mono*:

$\text{pochhammer } a \ n = (0 :: 'a :: \text{field-char-0}) \implies m \geq n \implies \text{pochhammer } a \ m = 0$

unfolding *pochhammer-eq-0-iff* **by** *auto*

lemma *pochhammer-neq-0-mono*:

$\text{pochhammer } a \ m \neq (0 :: 'a :: \text{field-char-0}) \implies m \geq n \implies \text{pochhammer } a \ n \neq 0$

unfolding *pochhammer-eq-0-iff* **by** *auto*

lemma *pochhammer-minus*:

$\text{pochhammer } (-b) \ k = ((-1) \wedge k :: 'a :: \text{comm-ring-1}) * \text{pochhammer } (b - \text{of-nat } k + 1) \ k$

proof (*cases k*)

case *0*

then show *?thesis* **by** *simp*

next

case (*Suc h*)

have *eq*: $((-1) \wedge \text{Suc } h :: 'a) = (\prod i = 0..h. -1)$

using *prod-constant* [**where** $A = \{0..h\}$ **and** $y = -1 :: 'a$]

by *auto*

with *Suc* **show** *?thesis*

using *pochhammer-Suc-prod-rev* [*of* $b - \text{of-nat } k + 1$]

by (*auto simp add: pochhammer-Suc-prod prod.distrib* [*symmetric*] *eq of-nat-diff simp del: prod-constant*)

qed

lemma *pochhammer-minus'*:

$\text{pochhammer } (b - \text{of-nat } k + 1) \ k = ((-1) \wedge k :: 'a :: \text{comm-ring-1}) * \text{pochhammer } (-b) \ k$

by (*simp add: pochhammer-minus*)

lemma *pochhammer-same*: $\text{pochhammer } (- \text{of-nat } n) \ n =$

$((-1) \wedge n :: 'a :: \{\text{semiring-char-0, comm-ring-1, semiring-no-zero-divisors}\}) * \text{fact } n$

unfolding *pochhammer-minus*

by (*simp add: of-nat-diff pochhammer-fact*)

lemma *pochhammer-product'*: $\text{pochhammer } z \ (n + m) = \text{pochhammer } z \ n * \text{pochhammer } (z + \text{of-nat } n) \ m$

proof (*induct n arbitrary: z*)

case *0*

then show *?case* **by** *simp*

next

case (*Suc n z*)

have $\text{pochhammer } z \ (\text{Suc } n) * \text{pochhammer } (z + \text{of-nat } (\text{Suc } n)) \ m =$

$z * (\text{pochhammer } (z + 1) \ n * \text{pochhammer } (z + 1 + \text{of-nat } n) \ m)$

```

  by (simp add: pochhammer-rec ac-simps)
  also note Suc[symmetric]
  also have z * pochhammer (z + 1) (n + m) = pochhammer z (Suc (n + m))
    by (subst pochhammer-rec) simp
  finally show ?case
    by simp
qed

```

lemma pochhammer-product:

```

  m ≤ n ⇒ pochhammer z n = pochhammer z m * pochhammer (z + of-nat m)
  (n - m)
  using pochhammer-product'[of z m n - m] by simp

```

lemma pochhammer-times-pochhammer-half:

```

  fixes z :: 'a::field-char-0
  shows pochhammer z (Suc n) * pochhammer (z + 1/2) (Suc n) = (∏ k=0..2*n+1.
  z + of-nat k / 2)
  proof (induct n)
    case 0
    then show ?case
      by (simp add: atLeast0-atMost-Suc)
  next
    case (Suc n)
    define n' where n' = Suc n
    have pochhammer z (Suc n') * pochhammer (z + 1 / 2) (Suc n') =
      (pochhammer z n' * pochhammer (z + 1 / 2) n') * ((z + of-nat n') * (z +
  1/2 + of-nat n'))
      (is - = - * ?A)
    by (simp-all add: pochhammer-rec' mult-ac)
    also have ?A = (z + of-nat (Suc (2 * n + 1)) / 2) * (z + of-nat (Suc (Suc (2
  * n + 1)))) / 2)
      (is - = ?B)
    by (simp add: field-simps n'-def)
    also note Suc[folded n'-def]
    also have (∏ k=0..2 * n + 1. z + of-nat k / 2) * ?B = (∏ k=0..2 * Suc n +
  1. z + of-nat k / 2)
      by (simp add: atLeast0-atMost-Suc)
    finally show ?case
      by (simp add: n'-def)
  qed

```

lemma pochhammer-double:

```

  fixes z :: 'a::field-char-0
  shows pochhammer (2 * z) (2 * n) = of-nat (2^(2*n)) * pochhammer z n *
  pochhammer (z+1/2) n
  proof (induct n)
    case 0
    then show ?case by simp
  next

```

```

case (Suc n)
have pochhammer (2 * z) (2 * (Suc n)) = pochhammer (2 * z) (2 * n) *
  (2 * (z + of-nat n)) * (2 * (z + of-nat n) + 1)
by (simp add: pochhammer-rec' ac-simps)
also note Suc
also have of-nat (2 ^ (2 * n)) * pochhammer z n * pochhammer (z + 1/2) n *
  (2 * (z + of-nat n)) * (2 * (z + of-nat n) + 1) =
  of-nat (2 ^ (2 * (Suc n))) * pochhammer z (Suc n) * pochhammer (z + 1/2)
(Suc n)
by (simp add: field-simps pochhammer-rec')
finally show ?case .
qed

```

lemma *fact-double*:

```

fact (2 * n) = (2 ^ (2 * n)) * pochhammer (1 / 2) n * fact n :: 'a::field-char-0)
using pochhammer-double[of 1/2::'a n] by (simp add: pochhammer-fact)

```

lemma *pochhammer-absorb-comp*: $(r - \text{of-nat } k) * \text{pochhammer } (-r) k = r * \text{pochhammer } (-r + 1) k$

(*is ?lhs = ?rhs*)

for $r :: 'a::\text{comm-ring-1}$

proof –

have *?lhs* = – pochhammer (– r) (Suc k)

by (subst pochhammer-rec') (simp add: algebra-simps)

also have ... = *?rhs*

by (subst pochhammer-rec) simp

finally show *?thesis* .

qed

93.3 Misc

lemma *fact-code* [code]:

```

fact n = (of-nat (fold-atLeastAtMost-nat ((*)) 2 n 1)) :: 'a::semiring-char-0)

```

proof –

have *fact* n = (of-nat ($\prod \{1..n\}$)) :: 'a)

by (simp add: fact-prod)

also have $\prod \{1..n\} = \prod \{2..n\}$

by (intro prod.mono-neutral-right) auto

also have ... = fold-atLeastAtMost-nat ((*)) 2 n 1

by (simp add: prod-atLeastAtMost-code)

finally show *?thesis* .

qed

lemma *pochhammer-code* [code]:

```

pochhammer a n =

```

```

  (if n = 0 then 1

```

```

    else fold-atLeastAtMost-nat ( $\lambda n \text{ acc. } (a + \text{of-nat } n) * \text{acc}$ ) 0 (n - 1) 1)

```

by (cases n)

(simp-all add: pochhammer-prod prod-atLeastAtMost-code [symmetric])

atLeastLessThanSuc-atLeastAtMost)

lemma *mult-less-iff1*: $0 < z \implies x * z < y * z \longleftrightarrow x < y$
for $x y z :: 'a::\text{linordered-idom}$
by *simp*

lemma *mult-le-cancel-iff1*: $0 < z \implies x * z \leq y * z \longleftrightarrow x \leq y$
for $x y z :: 'a::\text{linordered-idom}$
by *simp*

lemma *mult-le-cancel-iff2*: $0 < z \implies z * x \leq z * y \longleftrightarrow x \leq y$
for $x y z :: 'a::\text{linordered-idom}$
by *simp*

end

94 Binomial Coefficients and Binomial Theorem

theory *Binomial*
imports *Presburger Factorial*
begin

94.1 Binomial coefficients

This development is based on the work of Andy Gordon and Florian Kam-mueller.

Combinatorial definition

definition *binomial* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ (**infixl** *choose* 65)
where $n \text{ choose } k = \text{card } \{K \in \text{Pow } \{0..<n\}. \text{card } K = k\}$

theorem *n-subsets*:

assumes *finite A*

shows $\text{card } \{B. B \subseteq A \wedge \text{card } B = k\} = \text{card } A \text{ choose } k$

proof –

from *assms* **obtain** *f* **where** *bij*: *bij-betw* f $\{0..<\text{card } A\}$ A

by (*blast dest: ex-bij-betw-nat-finite*)

then have [*simp*]: $\text{card } (f ' C) = \text{card } C$ **if** $C \subseteq \{0..<\text{card } A\}$ **for** C

by (*meson bij-betw-imp-inj-on bij-betw-subset card-image that*)

from *bij* **have** *bij-betw* (*image* f) $(\text{Pow } \{0..<\text{card } A\})$ $(\text{Pow } A)$

by (*rule bij-betw-Pow*)

then have *inj-on* (*image* f) $(\text{Pow } \{0..<\text{card } A\})$

by (*rule bij-betw-imp-inj-on*)

moreover have $\{K. K \subseteq \{0..<\text{card } A\} \wedge \text{card } K = k\} \subseteq \text{Pow } \{0..<\text{card } A\}$

by *auto*

ultimately have *inj-on* (*image* f) $\{K. K \subseteq \{0..<\text{card } A\} \wedge \text{card } K = k\}$

by (*rule inj-on-subset*)

then have $\text{card } \{K. K \subseteq \{0..<\text{card } A\} \wedge \text{card } K = k\} =$
 $\text{card } (\text{image } f ' \{K. K \subseteq \{0..<\text{card } A\} \wedge \text{card } K = k\})$ (**is** - = $\text{card } ?C$)
by (*simp add: card-image*)
also have $?C = \{K. K \subseteq f ' \{0..<\text{card } A\} \wedge \text{card } K = k\}$
by (*auto elim!: subset-imageE*)
also have $f ' \{0..<\text{card } A\} = A$
by (*meson bij bij-betw-def*)
finally show *?thesis*
by (*simp add: binomial-def*)
qed

Recursive characterization

lemma *binomial-n-0 [simp]: n choose 0 = 1*

proof –

have $\{K \in \text{Pow } \{0..<n\}. \text{card } K = 0\} = \{\{\}\}$

by (*auto dest: finite-subset*)

then show *?thesis*

by (*simp add: binomial-def*)

qed

lemma *binomial-0-Suc [simp]: 0 choose Suc k = 0*

by (*simp add: binomial-def*)

lemma *binomial-Suc-Suc [simp]: Suc n choose Suc k = (n choose k) + (n choose Suc k)*

proof –

let $?P = \lambda n k. \{K. K \subseteq \{0..<n\} \wedge \text{card } K = k\}$

let $?Q = ?P$ (*Suc n*) (*Suc k*)

have *inj: inj-on (insert n) (?P n k)*

by *rule (auto; metis atLeastLessThan-iff insert-iff less-irrefl subsetCE)*

have *disjoint: insert n ' ?P n k \cap ?P n (Suc k) = $\{\}$*

by *auto*

have $?Q = \{K \in ?Q. n \in K\} \cup \{K \in ?Q. n \notin K\}$

by *auto*

also have $\{K \in ?Q. n \in K\} = \text{insert } n ' ?P n k$ (**is** $?A = ?B$)

proof (*rule set-eqI*)

fix *K*

have *K-finite: finite K if $K \subseteq \text{insert } n \{0..<n\}$*

using that **by** (*rule finite-subset simp-all*)

have *Suc-card-K: Suc (card K - Suc 0) = card K if $n \in K$*

and *finite K*

proof –

from $\langle n \in K \rangle$ **obtain** *L* **where** $K = \text{insert } n L$ **and** $n \notin L$

by (*blast elim: Set.set-insert*)

with that **show** *?thesis* **by** (*simp add: card.insert-remove*)

qed

show $K \in ?A \iff K \in ?B$

by (*subst in-image-insert-iff*)

(*auto simp add: card.insert-remove subset-eq-atLeast0-lessThan-finite*)

Diff-subset-conv K-finite Suc-card-K)

qed

also have $\{K \in ?Q. n \notin K\} = ?P n (Suc k)$
by (*auto simp add: atLeast0-lessThan-Suc*)

finally show *?thesis using inj disjoint*
by (*simp add: binomial-def card-Un-disjoint card-image*)

qed

lemma *binomial-eq-0*: $n < k \implies n \text{ choose } k = 0$
by (*auto simp add: binomial-def dest: subset-eq-atLeast0-lessThan-card*)

lemma *zero-less-binomial*: $k \leq n \implies n \text{ choose } k > 0$
by (*induct n k rule: diff-induct*) *simp-all*

lemma *binomial-eq-0-iff* [*simp*]: $n \text{ choose } k = 0 \iff n < k$
by (*metis binomial-eq-0 less-numeral-extra(3) not-less zero-less-binomial*)

lemma *zero-less-binomial-iff* [*simp*]: $n \text{ choose } k > 0 \iff k \leq n$
by (*metis binomial-eq-0-iff not-less0 not-less zero-less-binomial*)

lemma *binomial-n-n* [*simp*]: $n \text{ choose } n = 1$
by (*induct n*) (*simp-all add: binomial-eq-0*)

lemma *binomial-Suc-n* [*simp*]: $Suc n \text{ choose } n = Suc n$
by (*induct n*) *simp-all*

lemma *binomial-1* [*simp*]: $n \text{ choose } Suc 0 = n$
by (*induct n*) *simp-all*

lemma *choose-reduce-nat*:
 $0 < n \implies 0 < k \implies$
 $n \text{ choose } k = ((n - 1) \text{ choose } (k - 1)) + ((n - 1) \text{ choose } k)$
using *binomial-Suc-Suc* [*of n - 1 k - 1*] **by** *simp*

lemma *Suc-times-binomial-eq*: $Suc n * (n \text{ choose } k) = (Suc n \text{ choose } Suc k) * Suc k$

proof (*induction n arbitrary: k*)

case 0

then show *?case*
by *auto*

next

case (*Suc n*)

show *?case*

proof (*cases k*)

case (*Suc k'*)

then show *?thesis*
using *Suc.IH*
by (*auto simp add: add-mult-distrib add-mult-distrib2 le-Suc-eq binomial-eq-0*)

qed *auto*

qed

lemma *binomial-le-pow2*: $n \text{ choose } k \leq 2^n$
proof (*induction n arbitrary: k*)
 case 0
 then show ?case
 using *le-less less-le-trans* by *fastforce*
 next
 case (*Suc n*)
 show ?case
 proof (*cases k*)
 case (*Suc k'*)
 then show ?thesis
 using *Suc.IH* by (*simp add: add-le-mono mult-2*)
 qed *auto*
 qed

The absorption property.

lemma *Suc-times-binomial*: $Suc\ k * (Suc\ n \text{ choose } Suc\ k) = Suc\ n * (n \text{ choose } k)$
 using *Suc-times-binomial-eq* by *auto*

This is the well-known version of absorption, but it’s harder to use because of the need to reason about division.

lemma *binomial-Suc-Suc-eq-times*: $(Suc\ n \text{ choose } Suc\ k) = (Suc\ n * (n \text{ choose } k)) \text{ div } Suc\ k$
 by (*simp add: Suc-times-binomial-eq del: mult-Suc mult-Suc-right*)

Another version of absorption, with -1 instead of *Suc*.

lemma *times-binomial-minus1-eq*: $0 < k \implies k * (n \text{ choose } k) = n * ((n - 1) \text{ choose } (k - 1))$
 using *Suc-times-binomial-eq* [where $n = n - 1$ and $k = k - 1$]
 by (*auto split: nat-diff-split*)

94.2 The binomial theorem (courtesy of Tobias Nipkow):

Avigad’s version, generalized to any commutative ring

theorem *binomial-ring*: $(a + b :: 'a::comm-semiring-1)^n = (\sum_{k \leq n}. (of\text{-}nat\ (n \text{ choose } k)) * a^k * b^{(n-k)})$
proof (*induct n*)
 case 0
 then show ?case by *simp*
 next
 case (*Suc n*)
 have *decomp*: $\{0..n+1\} = \{0\} \cup \{n + 1\} \cup \{1..n\}$
 by *auto*
 have *decomp2*: $\{0..n\} = \{0\} \cup \{1..n\}$
 by *auto*

```

have  $(a + b)^{\wedge(n+1)} = (a + b) * (\sum_{k \leq n}. \text{of-nat } (n \text{ choose } k) * a^{\wedge k} * b^{\wedge(n-k)})$ 
using Suc.hyps by simp
also have  $\dots = a * (\sum_{k \leq n}. \text{of-nat } (n \text{ choose } k) * a^{\wedge k} * b^{\wedge(n-k)}) +$ 
 $b * (\sum_{k \leq n}. \text{of-nat } (n \text{ choose } k) * a^{\wedge k} * b^{\wedge(n-k)})$ 
by (rule distrib-right)
also have  $\dots = (\sum_{k \leq n}. \text{of-nat } (n \text{ choose } k) * a^{\wedge(k+1)} * b^{\wedge(n-k)}) +$ 
 $(\sum_{k \leq n}. \text{of-nat } (n \text{ choose } k) * a^{\wedge k} * b^{\wedge(n-k+1)})$ 
by (auto simp add: sum-distrib-left ac-simps)
also have  $\dots = (\sum_{k \leq n}. \text{of-nat } (n \text{ choose } k) * a^{\wedge k} * b^{\wedge(n+1-k)}) +$ 
 $(\sum_{k=1..n+1}. \text{of-nat } (n \text{ choose } (k-1)) * a^{\wedge k} * b^{\wedge(n+1-k)})$ 
by (simp add: atMost-atLeast0 sum.shift-bounds-cl-Suc-ivl Suc-diff-le field-simps
del: sum.cl-ivl-Suc)
also have  $\dots = b^{\wedge(n+1)} +$ 
 $(\sum_{k=1..n}. \text{of-nat } (n \text{ choose } k) * a^{\wedge k} * b^{\wedge(n+1-k)}) + (a^{\wedge(n+1)} +$ 
 $(\sum_{k=1..n}. \text{of-nat } (n \text{ choose } (k-1)) * a^{\wedge k} * b^{\wedge(n+1-k)}))$ 
using sum.nat-ivl-Suc' [of 1 n  $\lambda k. \text{of-nat } (n \text{ choose } (k-1)) * a^{\wedge k} * b^{\wedge(n+1-k)}$ ]
by (simp add: sum.atLeast-Suc-atMost atMost-atLeast0)
also have  $\dots = a^{\wedge(n+1)} + b^{\wedge(n+1)} +$ 
 $(\sum_{k=1..n}. \text{of-nat } (n+1 \text{ choose } k) * a^{\wedge k} * b^{\wedge(n+1-k)})$ 
by (auto simp add: field-simps sum.distrib [symmetric] choose-reduce-nat)
also have  $\dots = (\sum_{k \leq n+1}. \text{of-nat } (n+1 \text{ choose } k) * a^{\wedge k} * b^{\wedge(n+1-k)})$ 
using decomp by (simp add: atMost-atLeast0 field-simps)
finally show ?case
by simp
qed

```

Original version for the naturals.

```

corollary binomial:  $(a + b :: \text{nat})^{\wedge n} = (\sum_{k \leq n}. (\text{of-nat } (n \text{ choose } k)) * a^{\wedge k} * b^{\wedge(n-k)})$ 
using binomial-ring [of int a int b n]
by (simp only: of-nat-add [symmetric] of-nat-mult [symmetric] of-nat-power [symmetric]
of-nat-sum [symmetric] of-nat-eq-iff of-nat-id)

```

lemma *binomial-fact-lemma*: $k \leq n \implies \text{fact } k * \text{fact } (n - k) * (n \text{ choose } k) = \text{fact } n$

proof *(induct n arbitrary: k rule: nat-less-induct)*

fix *n k*

assume *H*: $\forall m < n. \forall x \leq m. \text{fact } x * \text{fact } (m - x) * (m \text{ choose } x) = \text{fact } m$

assume *kn*: $k \leq n$

let *?ths* = $\text{fact } k * \text{fact } (n - k) * (n \text{ choose } k) = \text{fact } n$

consider $n = 0 \vee k = 0 \vee n = k \mid m \text{ h}$ **where** $n = \text{Suc } m \text{ k} = \text{Suc } h \text{ h} < m$

using *kn by atomize-elim presburger*

then show $\text{fact } k * \text{fact } (n - k) * (n \text{ choose } k) = \text{fact } n$

proof *cases*

case *1*

with *kn show ?thesis by auto*

next

```

case 2
note  $n = \langle n = \text{Suc } m \rangle$ 
note  $k = \langle k = \text{Suc } h \rangle$ 
note  $hm = \langle h < m \rangle$ 
have  $mn: m < n$ 
  using  $n$  by arith
have  $hm': h \leq m$ 
  using  $hm$  by arith
have  $km: k \leq m$ 
  using  $hm$   $k$   $n$   $kn$  by arith
have  $m - h = \text{Suc } (m - \text{Suc } h)$ 
  using  $k$   $km$   $hm$  by arith
with  $km$   $k$  have  $\text{fact } (m - h) = (m - h) * \text{fact } (m - k)$ 
  by simp
with  $n$   $k$  have  $\text{fact } k * \text{fact } (n - k) * (n \text{ choose } k) =$ 
   $k * (\text{fact } h * \text{fact } (m - h) * (m \text{ choose } h)) +$ 
   $(m - h) * (\text{fact } k * \text{fact } (m - k) * (m \text{ choose } k))$ 
  by (simp add: field-simps)
also have  $\dots = (k + (m - h)) * \text{fact } m$ 
  using  $H[\text{rule-format, OF } mn \text{ } hm']$   $H[\text{rule-format, OF } mn \text{ } km]$ 
  by (simp add: field-simps)
finally show ?thesis
  using  $k$   $n$   $km$  by simp
qed
qed

```

```

lemma binomial-fact':
assumes  $k \leq n$ 
shows  $n \text{ choose } k = \text{fact } n \text{ div } (\text{fact } k * \text{fact } (n - k))$ 
using binomial-fact-lemma [OF assms]
by (metis fact-nonzero mult-eq-0-iff nonzero-mult-div-cancel-left)

```

```

lemma binomial-fact:
assumes  $kn: k \leq n$ 
shows  $(\text{of-nat } (n \text{ choose } k) :: 'a::\text{field-char-0}) = \text{fact } n / (\text{fact } k * \text{fact } (n - k))$ 
using binomial-fact-lemma [OF  $kn$ ]
by (metis (mono-tags, lifting) fact-nonzero mult-eq-0-iff nonzero-mult-div-cancel-left of-nat-fact of-nat-mult)

```

```

lemma fact-binomial:
assumes  $k \leq n$ 
shows  $\text{fact } k * \text{of-nat } (n \text{ choose } k) = (\text{fact } n / \text{fact } (n - k) :: 'a::\text{field-char-0})$ 
unfolding binomial-fact [OF assms] by (simp add: field-simps)

```

```

lemma choose-two:  $n \text{ choose } 2 = n * (n - 1) \text{ div } 2$ 
proof (cases  $n \geq 2$ )
case False
then have  $n = 0 \vee n = 1$ 
  by auto

```

then show *?thesis* **by** *auto*
next

case *True*
define *m* **where** $m = n - 2$
with *True* **have** $n = m + 2$
by *simp*
then have $\text{fact } n = n * (n - 1) * \text{fact } (n - 2)$
by (*simp add: fact-prod-Suc atLeast0-lessThan-Suc algebra-simps*)
with *True* **show** *?thesis*
by (*simp add: binomial-fact'*)
qed

lemma *choose-row-sum*: $(\sum k \leq n. n \text{ choose } k) = 2^n$
using *binomial [of 1 1 n]* **by** (*simp add: numeral-2-eq-2*)

lemma *sum-choose-lower*: $(\sum k \leq n. (r+k) \text{ choose } k) = \text{Suc } (r+n) \text{ choose } n$
by (*induct n*) *auto*

lemma *sum-choose-upper*: $(\sum k \leq n. k \text{ choose } m) = \text{Suc } n \text{ choose } \text{Suc } m$
by (*induct n*) *auto*

lemma *choose-alternating-sum*:
 $n > 0 \implies (\sum i \leq n. (-1)^i * \text{of-nat } (n \text{ choose } i)) = (0 :: 'a::\text{comm-ring-1})$
using *binomial-ring[of -1 :: 'a 1 n]*
by (*simp add: atLeast0AtMost mult-of-nat-commute zero-power*)

lemma *choose-even-sum*:
assumes $n > 0$
shows $2 * (\sum i \leq n. \text{if even } i \text{ then } \text{of-nat } (n \text{ choose } i) \text{ else } 0) = (2^n :: 'a::\text{comm-ring-1})$
proof –
have $2^n = (\sum i \leq n. \text{of-nat } (n \text{ choose } i)) + (\sum i \leq n. (-1)^i * \text{of-nat } (n \text{ choose } i))$
using *choose-row-sum[of n]*
by (*simp add: choose-alternating-sum assms atLeast0AtMost of-nat-sum[symmetric]*)
also have $\dots = (\sum i \leq n. \text{of-nat } (n \text{ choose } i) + (-1)^i * \text{of-nat } (n \text{ choose } i))$
by (*simp add: sum.distrib*)
also have $\dots = 2 * (\sum i \leq n. \text{if even } i \text{ then } \text{of-nat } (n \text{ choose } i) \text{ else } 0)$
by (*subst sum-distrib-left, intro sum.cong*) *simp-all*
finally show *?thesis ..*
qed

lemma *choose-odd-sum*:
assumes $n > 0$
shows $2 * (\sum i \leq n. \text{if odd } i \text{ then } \text{of-nat } (n \text{ choose } i) \text{ else } 0) = (2^n :: 'a::\text{comm-ring-1})$
proof –
have $2^n = (\sum i \leq n. \text{of-nat } (n \text{ choose } i)) - (\sum i \leq n. (-1)^i * \text{of-nat } (n \text{ choose } i))$
using *choose-row-sum[of n]*

using *choose-row-sum*[*of n*]
by (*simp add: choose-alternating-sum* *assms atLeast0AtMost of-nat-sum*[*symmetric*])
also have $\dots = (\sum_{i \leq n}. \text{of-nat } (n \text{ choose } i) - (-1)^i * \text{of-nat } (n \text{ choose } i))$
by (*simp add: sum-subtractf*)
also have $\dots = 2 * (\sum_{i \leq n}. \text{if odd } i \text{ then of-nat } (n \text{ choose } i) \text{ else } 0)$
by (*subst sum-distrib-left, intro sum.cong*) *simp-all*
finally show *?thesis ..*
qed

NW diagonal sum property

lemma *sum-choose-diagonal*:

assumes $m \leq n$

shows $(\sum_{k \leq m}. (n - k) \text{ choose } (m - k)) = \text{Suc } n \text{ choose } m$

proof –

have $(\sum_{k \leq m}. (n - k) \text{ choose } (m - k)) = (\sum_{k \leq m}. (n - m + k) \text{ choose } k)$

using *sum.atLeastAtMost-rev* [*of $\lambda k. (n - k) \text{ choose } (m - k) 0 m$*] *assms*

by (*simp add: atMost-atLeast0*)

also have $\dots = \text{Suc } (n - m + m) \text{ choose } m$

by (*rule sum-choose-lower*)

also have $\dots = \text{Suc } n \text{ choose } m$

using *assms* **by** *simp*

finally show *?thesis .*

qed

94.3 Generalized binomial coefficients

definition *gbinomial* :: $'a::\{\text{semidom-divide, semiring-char-0}\} \Rightarrow \text{nat} \Rightarrow 'a$ (*infixl gchoose 65*)

where *gbinomial-prod-rev*: $a \text{ gchoose } k = \text{prod } (\lambda i. a - \text{of-nat } i) \{0..<k\} \text{ div fact } k$

lemma *gbinomial-0* [*simp*]:

$a \text{ gchoose } 0 = 1$

$0 \text{ gchoose } (\text{Suc } k) = 0$

by (*simp-all add: gbinomial-prod-rev prod.atLeast0-lessThan-Suc-shift del: prod.op-ivl-Suc*)

lemma *gbinomial-Suc*: $a \text{ gchoose } (\text{Suc } k) = \text{prod } (\lambda i. a - \text{of-nat } i) \{0..k\} \text{ div fact } (\text{Suc } k)$

by (*simp add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost*)

lemma *gbinomial-1* [*simp*]: $a \text{ gchoose } 1 = a$

by (*simp add: gbinomial-prod-rev lessThan-Suc*)

lemma *gbinomial-Suc0* [*simp*]: $a \text{ gchoose } \text{Suc } 0 = a$

by (*simp add: gbinomial-prod-rev lessThan-Suc*)

lemma *gbinomial-mult-fact*: $\text{fact } k * (a \text{ gchoose } k) = (\prod_{i = 0..<k}. a - \text{of-nat } i)$

for $a :: 'a::\text{field-char-0}$

by (*simp-all add: gbinomial-prod-rev field-simps*)

lemma *gbinomial-mult-fact'*: $(a \text{ gchoose } k) * \text{fact } k = (\prod i = 0..<k. a - \text{of-nat } i)$
for $a :: 'a::\text{field-char-0}$
using *gbinomial-mult-fact [of k a]* **by** (*simp add: ac-simps*)

lemma *gbinomial-pochhammer*: $a \text{ gchoose } k = (-1) ^ k * \text{pochhammer } (-a) k$
 $/ \text{fact } k$

for $a :: 'a::\text{field-char-0}$

proof (*cases k*)

case (*Suc k'*)

then have $a \text{ gchoose } k = \text{pochhammer } (a - \text{of-nat } k') (\text{Suc } k') / ((1 + \text{of-nat } k') * \text{fact } k')$

by (*simp add: gbinomial-prod-rev pochhammer-prod-rev atLeastLessThanSuc-atLeastAtMost prod.atLeast-Suc-atMost-Suc-shift of-nat-diff flip: power-mult-distrib prod.cl-ivl-Suc*)

then show *?thesis*

by (*simp add: pochhammer-minus Suc*)

qed *auto*

lemma *gbinomial-pochhammer'*: $a \text{ gchoose } k = \text{pochhammer } (a - \text{of-nat } k + 1) k$
 $/ \text{fact } k$

for $a :: 'a::\text{field-char-0}$

proof –

have $a \text{ gchoose } k = ((-1) ^ k * (-1) ^ k) * \text{pochhammer } (a - \text{of-nat } k + 1) k / \text{fact } k$

by (*simp add: gbinomial-pochhammer pochhammer-minus mult-ac*)

also have $(-1 :: 'a) ^ k * (-1) ^ k = 1$

by (*subst power-add [symmetric]*) *simp*

finally show *?thesis*

by *simp*

qed

lemma *gbinomial-binomial*: $n \text{ gchoose } k = n \text{ choose } k$

proof (*cases k ≤ n*)

case *False*

then have $n < k$

by (*simp add: not-le*)

then have $0 \in ((-) n) ^ \{0..<k\}$

by *auto*

then have $\text{prod } ((-) n) \{0..<k\} = 0$

by (*auto intro: prod-zero*)

with $\langle n < k \rangle$ **show** *?thesis*

by (*simp add: binomial-eq-0 gbinomial-prod-rev prod-zero*)

next

case *True*

from *True* **have** $*$: $\text{prod } ((-) n) \{0..<k\} = \prod \{\text{Suc } (n - k)..n\}$

by (*intro prod.reindex-bij-witness[of - λi. n - i λi. n - i]*) *auto*

from *True* **have** $n \text{ choose } k = \text{fact } n \text{ div } (\text{fact } k * \text{fact } (n - k))$

by (*rule binomial-fact'*)

with $*$ **show** *?thesis*

by (simp add: gbinomial-prod-rev mult.commute [of fact k] div-mult2-eq fact-div-fact)
qed

lemma of-nat-gbinomial: of-nat (n gchoose k) = (of-nat n gchoose k :: 'a::field-char-0)

proof (cases k ≤ n)

case False

then show ?thesis

by (simp add: not-le gbinomial-binomial binomial-eq-0 gbinomial-prod-rev)

next

case True

define m where m = n - k

with True have n: n = m + k

by arith

from n have fact n = ((∏ i = 0..<m + k. of-nat (m + k - i)) :: 'a)

by (simp add: fact-prod-rev)

also have ... = ((∏ i ∈ {0..<k} ∪ {k..<m + k}. of-nat (m + k - i)) :: 'a)

by (simp add: ivl-disj-un)

finally have fact n = (fact m * (∏ i = 0..<k. of-nat m + of-nat k - of-nat i))
:: 'a)

using prod.shift-bounds-nat-ivl [of λi. of-nat (m + k - i) :: 'a 0 k m]

by (simp add: fact-prod-rev [of m] prod.union-disjoint of-nat-diff)

then have fact n / fact (n - k) = ((∏ i = 0..<k. of-nat n - of-nat i) :: 'a)

by (simp add: n)

with True have fact k * of-nat (n gchoose k) = (fact k * (of-nat n gchoose k))
:: 'a)

by (simp only: gbinomial-mult-fact [of k of-nat n] gbinomial-binomial [of n k]
fact-binomial)

then show ?thesis

by simp

qed

lemma binomial-gbinomial: of-nat (n choose k) = (of-nat n gchoose k :: 'a::field-char-0)

by (simp add: gbinomial-binomial [symmetric] of-nat-gbinomial)

setup

⟨Sign.add-const-constraint (const-name ⟨gbinomial⟩, SOME typ ⟨'a::field-char-0
⇒ nat ⇒ 'a⟩)⟩

lemma gbinomial-mult-1:

fixes a :: 'a::field-char-0

shows a * (a gchoose k) = of-nat k * (a gchoose k) + of-nat (Suc k) * (a gchoose
(Suc k))

(is ?l = ?r)

proof -

have ?r = ((- 1) ^k * pochhammer (- a) k / fact k) * (of-nat k - (- a +
of-nat k))

unfolding gbinomial-pochhammer pochhammer-Suc right-diff-distrib power-Suc

by (auto simp add: field-simps simp del: of-nat-Suc)

also have ... = ?l

by (*simp add: field-simps gbinomial-pochhammer*)
 finally show ?thesis ..
 qed

lemma *gbinomial-mult-1'*:

$(a \text{ gchoose } k) * a = \text{of-nat } k * (a \text{ gchoose } k) + \text{of-nat } (\text{Suc } k) * (a \text{ gchoose } (\text{Suc } k))$

for $a :: 'a::\text{field-char-0}$

by (*simp add: mult.commute gbinomial-mult-1*)

lemma *gbinomial-Suc-Suc*: $(a + 1) \text{ gchoose } (\text{Suc } k) = a \text{ gchoose } k + (a \text{ gchoose } (\text{Suc } k))$

for $a :: 'a::\text{field-char-0}$

proof (*cases k*)

case 0

then show ?thesis by *simp*

next

case (*Suc h*)

have *eq0*: $(\prod i \in \{1..k\}. (a + 1) - \text{of-nat } i) = (\prod i \in \{0..h\}. a - \text{of-nat } i)$

proof (*rule prod.reindex-cong*)

show $\{1..k\} = \text{Suc } \{0..h\}$

using *Suc* by (*auto simp add: image-Suc-atMost*)

qed *auto*

have *fact* $(\text{Suc } k) * (a \text{ gchoose } k + (a \text{ gchoose } (\text{Suc } k))) =$

$(a \text{ gchoose } \text{Suc } h) * (\text{fact } (\text{Suc } (\text{Suc } h))) +$

$(a \text{ gchoose } \text{Suc } (\text{Suc } h)) * (\text{fact } (\text{Suc } (\text{Suc } h)))$

by (*simp add: Suc field-simps del: fact-Suc*)

also have ... =

$(a \text{ gchoose } \text{Suc } h) * \text{of-nat } (\text{Suc } (\text{Suc } h) * \text{fact } (\text{Suc } h)) + (\prod i=0.. \text{Suc } h. a - \text{of-nat } i)$

apply (*simp only: gbinomial-mult-fact field-simps mult.left-commute [of - 2]*)

apply (*simp del: fact-Suc add: fact-Suc [of Suc h] field-simps gbinomial-mult-fact mult.left-commute [of - 2] atLeastLessThanSuc-atLeastAtMost*)

done

also have ... =

$(\text{fact } (\text{Suc } h) * (a \text{ gchoose } \text{Suc } h)) * \text{of-nat } (\text{Suc } (\text{Suc } h)) + (\prod i=0.. \text{Suc } h. a - \text{of-nat } i)$

by (*simp only: fact-Suc mult.commute mult.left-commute of-nat-fact of-nat-id of-nat-mult*)

also have ... =

$\text{of-nat } (\text{Suc } (\text{Suc } h)) * (\prod i=0..h. a - \text{of-nat } i) + (\prod i=0.. \text{Suc } h. a - \text{of-nat } i)$

unfolding *gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost* by *auto*

also have ... =

$(\prod i=0.. \text{Suc } h. a - \text{of-nat } i) + (\text{of-nat } h * (\prod i=0..h. a - \text{of-nat } i) + 2 * (\prod i=0..h. a - \text{of-nat } i))$

by (*simp add: field-simps*)

also have ... =

$((a \text{ gchoose } \text{Suc } h) * (\text{fact } (\text{Suc } h)) * \text{of-nat } (\text{Suc } k)) + (\prod i \in \{0.. \text{Suc } h\}. a - \text{of-nat } i)$

unfolding *gbinomial-mult-fact'*
by (*simp add: comm-semiring-class.distrib field-simps Suc atLeastLessThanSuc-atLeastAtMost*)
also have $\dots = (\prod_{i \in \{0..h\}}. a - \text{of-nat } i) * (a + 1)$
unfolding *gbinomial-mult-fact' atLeast0-atMost-Suc*
by (*simp add: field-simps Suc atLeastLessThanSuc-atLeastAtMost*)
also have $\dots = (\prod_{i \in \{0..k\}}. (a + 1) - \text{of-nat } i)$
using *eq0*
by (*simp add: Suc prod.atLeast0-atMost-Suc-shift del: prod.cl-ivl-Suc*)
also have $\dots = (\text{fact } (\text{Suc } k)) * ((a + 1) \text{ gchoose } (\text{Suc } k))$
by (*simp only: gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost*)
finally show *?thesis*
using *fact-nonzero [of Suc k] by auto*
qed

lemma *gbinomial-reduce-nat*: $0 < k \implies a \text{ gchoose } k = (a - 1) \text{ gchoose } (k - 1) + ((a - 1) \text{ gchoose } k)$
for $a :: 'a::\text{field-char-0}$
by (*metis Suc-pred' diff-add-cancel gbinomial-Suc-Suc*)

lemma *gchoose-row-sum-weighted*:
 $(\sum_{k=0..m}. (r \text{ gchoose } k) * (r/2 - \text{of-nat } k)) = \text{of-nat}(\text{Suc } m) / 2 * (r \text{ gchoose } (\text{Suc } m))$
for $r :: 'a::\text{field-char-0}$
by (*induct m*) (*simp-all add: field-simps distrib gbinomial-mult-1*)

lemma *binomial-symmetric*:
assumes $kn: k \leq n$
shows $n \text{ choose } k = n \text{ choose } (n - k)$
proof –
have $kn': n - k \leq n$
using kn **by** *arith*
from *binomial-fact-lemma[OF kn] binomial-fact-lemma[OF kn']*
have $\text{fact } k * \text{fact } (n - k) * (n \text{ choose } k) = \text{fact } (n - k) * \text{fact } (n - (n - k)) * (n \text{ choose } (n - k))$
by *simp*
then show *?thesis*
using kn **by** *simp*
qed

lemma *choose-rising-sum*:
 $(\sum_{j \leq m}. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } (n + 1))$
 $(\sum_{j \leq m}. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } m)$
proof –
show $(\sum_{j \leq m}. ((n + j) \text{ choose } n)) = ((n + m + 1) \text{ choose } (n + 1))$
by (*induct m*) *simp-all*
also have $\dots = (n + m + 1) \text{ choose } m$
by (*subst binomial-symmetric*) *simp-all*
finally show $(\sum_{j \leq m}. ((n + j) \text{ choose } n)) = (n + m + 1) \text{ choose } m .$
qed

lemma *choose-linear-sum*: $(\sum_{i \leq n}. i * (n \text{ choose } i)) = n * 2^{(n - 1)}$
proof (*cases n*)
 case 0
 then show ?thesis **by** *simp*
next
 case (*Suc m*)
 have $(\sum_{i \leq n}. i * (n \text{ choose } i)) = (\sum_{i \leq \text{Suc } m}. i * (\text{Suc } m \text{ choose } i))$
 by (*simp add: Suc*)
 also have $\dots = \text{Suc } m * 2^{m}$
 unfolding *sum.atMost-Suc-shift Suc-times-binomial sum-distrib-left[symmetric]*
 by (*simp add: choose-row-sum*)
 finally show ?thesis
 using *Suc* **by** *simp*
qed

lemma *choose-alternating-linear-sum*:
 assumes $n \neq 1$
 shows $(\sum_{i \leq n}. (-1)^i * \text{of-nat } i * \text{of-nat } (n \text{ choose } i) :: 'a::\text{comm-ring-1}) = 0$
proof (*cases n*)
 case 0
 then show ?thesis **by** *simp*
next
 case (*Suc m*)
 with *assms* **have** $m > 0$
 by *simp*
 have $(\sum_{i \leq n}. (-1)^i * \text{of-nat } i * \text{of-nat } (n \text{ choose } i) :: 'a) =$
 $(\sum_{i \leq \text{Suc } m}. (-1)^i * \text{of-nat } i * \text{of-nat } (\text{Suc } m \text{ choose } i))$
 by (*simp add: Suc*)
 also have $\dots = (\sum_{i \leq m}. (-1)^i * \text{of-nat } (\text{Suc } i) * \text{of-nat } (\text{Suc } i * (\text{Suc } m \text{ choose } \text{Suc } i)))$
 by (*simp only: sum.atMost-Suc-shift sum-distrib-left[symmetric] mult-ac of-nat-mult*)
 simp
 also have $\dots = - \text{of-nat } (\text{Suc } m) * (\sum_{i \leq m}. (-1)^i * \text{of-nat } (m \text{ choose } i))$
 by (*subst sum-distrib-left, rule sum.cong[OF refl], subst Suc-times-binomial*)
 (*simp add: algebra-simps*)
 also have $(\sum_{i \leq m}. (-1)^i * \text{of-nat } ((m \text{ choose } i))) = 0$
 using *choose-alternating-sum[OF ‹m > 0›]* **by** *simp*
 finally show ?thesis
 by *simp*
qed

lemma *vandermonde*: $(\sum_{k \leq r}. (m \text{ choose } k) * (n \text{ choose } (r - k))) = (m + n \text{ choose } r)$
proof (*induct n arbitrary: r*)
 case 0
 have $(\sum_{k \leq r}. (m \text{ choose } k) * (0 \text{ choose } (r - k))) = (\sum_{k \leq r}. \text{if } k = r \text{ then } (m \text{ choose } k) \text{ else } 0)$
 by (*intro sum.cong*) *simp-all*

also have $\dots = m \text{ choose } r$
by *simp*
finally show *?case*
by *simp*
next
case (*Suc n r*)
show *?case*
by (*cases r*) (*simp-all add: Suc [symmetric] algebra-simps sum.distrib Suc-diff-le*)
qed

lemma *choose-square-sum*: $(\sum k \leq n. (n \text{ choose } k)^2) = ((2 * n) \text{ choose } n)$
using *vandermonde[of n n n]*
by (*simp add: power2-eq-square mult-2 binomial-symmetric [symmetric]*)

lemma *pochhammer-binomial-sum*:
fixes $a b :: 'a::\text{comm-ring-1}$
shows $\text{pochhammer } (a + b) n = (\sum k \leq n. \text{of-nat } (n \text{ choose } k) * \text{pochhammer } a k * \text{pochhammer } b (n - k))$
proof (*induction n arbitrary: a b*)
case 0
then show *?case by simp*
next
case (*Suc n a b*)
have $(\sum k \leq \text{Suc } n. \text{of-nat } (\text{Suc } n \text{ choose } k) * \text{pochhammer } a k * \text{pochhammer } b (\text{Suc } n - k)) =$
 $(\sum i \leq n. \text{of-nat } (n \text{ choose } i) * \text{pochhammer } a (\text{Suc } i) * \text{pochhammer } b (n - i)) +$
 $((\sum i \leq n. \text{of-nat } (n \text{ choose } \text{Suc } i) * \text{pochhammer } a (\text{Suc } i) * \text{pochhammer } b (n - i)) +$
 $\text{pochhammer } b (\text{Suc } n))$
by (*subst sum.atMost-Suc-shift*) (*simp add: ring-distrib sum.distrib*)
also have $(\sum i \leq n. \text{of-nat } (n \text{ choose } i) * \text{pochhammer } a (\text{Suc } i) * \text{pochhammer } b (n - i)) =$
 $a * \text{pochhammer } ((a + 1) + b) n$
by (*subst Suc*) (*simp add: sum-distrib-left pochhammer-rec mult-ac*)
also have $(\sum i \leq n. \text{of-nat } (n \text{ choose } \text{Suc } i) * \text{pochhammer } a (\text{Suc } i) * \text{pochhammer } b (n - i)) +$
 $\text{pochhammer } b (\text{Suc } n) =$
 $(\sum i = 0 .. \text{Suc } n. \text{of-nat } (n \text{ choose } i) * \text{pochhammer } a i * \text{pochhammer } b (\text{Suc } n - i))$
apply (*subst sum.atLeast-Suc-atMost, simp*)
apply (*simp add: sum.shift-bounds-cl-Suc-ivl atLeast0AtMost del: sum.cl-ivl-Suc*)
done
also have $\dots = (\sum i \leq n. \text{of-nat } (n \text{ choose } i) * \text{pochhammer } a i * \text{pochhammer } b (\text{Suc } n - i))$
using *Suc by (intro sum.mono-neutral-right) (auto simp: not-le binomial-eq-0)*
also have $\dots = (\sum i \leq n. \text{of-nat } (n \text{ choose } i) * \text{pochhammer } a i * \text{pochhammer } b (\text{Suc } (n - i)))$
by (*intro sum.cong*) (*simp-all add: Suc-diff-le*)

```

also have ... = b * pochhammer (a + (b + 1)) n
  by (subst Suc) (simp add: sum-distrib-left mult-ac pochhammer-rec)
also have a * pochhammer ((a + 1) + b) n + b * pochhammer (a + (b + 1))
n =
  pochhammer (a + b) (Suc n)
  by (simp add: pochhammer-rec algebra-simps)
finally show ?case ..
qed

```

Contributed by Manuel Eberl, generalised by LCP. Alternative definition of the binomial coefficient as $\prod_{i < k}. (n - i) / (k - i)$.

```

lemma gbinomial-altdef-of-nat: a gchoose k = ( $\prod_{i = 0..<k}. (a - \text{of-nat } i) /$ 
of-nat (k - i) :: 'a)
  for k :: nat and a :: 'a::field-char-0
  by (simp add: prod-dividef gbinomial-prod-rev fact-prod-rev)

```

```

lemma gbinomial-ge-n-over-k-pow-k:

```

```

  fixes k :: nat
  and a :: 'a::linordered-field
  assumes of-nat k ≤ a
  shows (a / of-nat k :: 'a) ^ k ≤ a gchoose k
proof -
  have x: 0 ≤ a
    using assms of-nat-0-le-iff order-trans by blast
  have (a / of-nat k :: 'a) ^ k = ( $\prod_{i = 0..<k}. a / \text{of-nat } k :: 'a$ )
    by simp
  also have ... ≤ a gchoose k
proof -
  have  $\bigwedge i. i < k \implies 0 \leq a / \text{of-nat } k$ 
    by (simp add: x zero-le-divide-iff)
  moreover have a / of-nat k ≤ (a - of-nat i) / of-nat (k - i) if i < k for i
proof -
  from assms have a * of-nat i ≥ of-nat (i * k)
    by (metis mult.commute mult-le-cancel-right of-nat-less-0-iff of-nat-mult)
  then have a * of-nat k - a * of-nat i ≤ a * of-nat k - of-nat (i * k)
    by arith
  then have a * of-nat (k - i) ≤ (a - of-nat i) * of-nat k
    using <i < k> by (simp add: algebra-simps zero-less-mult-iff of-nat-diff)
  then have a * of-nat (k - i) ≤ (a - of-nat i) * (of-nat k :: 'a)
    by blast
  with assms show ?thesis
    using <i < k> by (simp add: field-simps)
qed
  ultimately show ?thesis
    unfolding gbinomial-altdef-of-nat
    by (intro prod-mono) auto
qed
finally show ?thesis .
qed

```

lemma *gbinomial-negated-upper*: $(a \text{ gchoose } k) = (-1) ^ k * ((\text{of-nat } k - a - 1) \text{ gchoose } k)$

by (*simp add: gbinomial-pochhammer pochhammer-minus algebra-simps*)

lemma *gbinomial-minus*: $((-a) \text{ gchoose } k) = (-1) ^ k * ((a + \text{of-nat } k - 1) \text{ gchoose } k)$

by (*subst gbinomial-negated-upper (simp add: add-ac)*)

lemma *Suc-times-gbinomial*: $\text{of-nat } (\text{Suc } k) * ((a + 1) \text{ gchoose } (\text{Suc } k)) = (a + 1) * (a \text{ gchoose } k)$

proof (*cases k*)

case 0

then show *?thesis* **by** *simp*

next

case (*Suc b*)

then have $((a + 1) \text{ gchoose } (\text{Suc } (\text{Suc } b))) = (\prod i = 0.. \text{Suc } b. a + (1 - \text{of-nat } i)) / \text{fact } (b + 2)$

by (*simp add: field-simps gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost*)

also have $(\prod i = 0.. \text{Suc } b. a + (1 - \text{of-nat } i)) = (a + 1) * (\prod i = 0..b. a - \text{of-nat } i)$

by (*simp add: prod.atLeast0-atMost-Suc-shift del: prod.cl-ivl-Suc*)

also have $\dots / \text{fact } (b + 2) = (a + 1) / \text{of-nat } (\text{Suc } (\text{Suc } b)) * (a \text{ gchoose } \text{Suc } b)$

by (*simp-all add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost*)

finally show *?thesis* **by** (*simp add: Suc field-simps del: of-nat-Suc*)

qed

lemma *gbinomial-factors*: $((a + 1) \text{ gchoose } (\text{Suc } k)) = (a + 1) / \text{of-nat } (\text{Suc } k) * (a \text{ gchoose } k)$

proof (*cases k*)

case 0

then show *?thesis* **by** *simp*

next

case (*Suc b*)

then have $((a + 1) \text{ gchoose } (\text{Suc } (\text{Suc } b))) = (\prod i = 0 .. \text{Suc } b. a + (1 - \text{of-nat } i)) / \text{fact } (b + 2)$

by (*simp add: field-simps gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost*)

also have $(\prod i = 0 .. \text{Suc } b. a + (1 - \text{of-nat } i)) = (a + 1) * (\prod i = 0..b. a - \text{of-nat } i)$

by (*simp add: prod.atLeast0-atMost-Suc-shift del: prod.cl-ivl-Suc*)

also have $\dots / \text{fact } (b + 2) = (a + 1) / \text{of-nat } (\text{Suc } (\text{Suc } b)) * (a \text{ gchoose } \text{Suc } b)$

by (*simp-all add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost atLeast0AtMost*)

finally show *?thesis*

by (*simp add: Suc*)

qed

lemma *gbinomial-rec*: $((a + 1) \text{ gchoose } (\text{Suc } k)) = (a \text{ gchoose } k) * ((a + 1) / \text{of-nat } (\text{Suc } k))$
using *gbinomial-mult-1*[of a k]
by (*subst gbinomial-Suc-Suc*) (*simp add: field-simps del: of-nat-Suc, simp add: algebra-simps*)

lemma *gbinomial-of-nat-symmetric*: $k \leq n \implies (\text{of-nat } n) \text{ gchoose } k = (\text{of-nat } n) \text{ gchoose } (n - k)$
using *binomial-symmetric*[of k n] **by** (*simp add: binomial-gbinomial [symmetric]*)

The absorption identity (equation 5.5 [3, p. 157]):

$$\binom{r}{k} = \frac{r}{k} \binom{r-1}{k-1}, \quad \text{integer } k \neq 0.$$

lemma *gbinomial-absorption'*: $k > 0 \implies a \text{ gchoose } k = (a / \text{of-nat } k) * (a - 1 \text{ gchoose } (k - 1))$
using *gbinomial-rec*[of a - 1 k - 1]
by (*simp-all add: gbinomial-rec field-simps del: of-nat-Suc*)

The absorption identity is written in the following form to avoid division by k (the lower index) and therefore remove the $k \neq 0$ restriction [3, p. 157]:

$$k \binom{r}{k} = r \binom{r-1}{k-1}, \quad \text{integer } k.$$

lemma *gbinomial-absorption*: $\text{of-nat } (\text{Suc } k) * (a \text{ gchoose } \text{Suc } k) = a * ((a - 1) \text{ gchoose } k)$
using *gbinomial-absorption'*[of Suc k a] **by** (*simp add: field-simps del: of-nat-Suc*)

The absorption identity for natural number binomial coefficients:

lemma *binomial-absorption*: $\text{Suc } k * (n \text{ choose } \text{Suc } k) = n * ((n - 1) \text{ choose } k)$
by (*cases n*) (*simp-all add: binomial-eq-0 Suc-times-binomial del: binomial-Suc-Suc mult-Suc*)

The absorption companion identity for natural number coefficients, following the proof by GKP [3, p. 157]:

lemma *binomial-absorb-comp*: $(n - k) * (n \text{ choose } k) = n * ((n - 1) \text{ choose } k)$
(is ?lhs = ?rhs)

proof (*cases n ≤ k*)

case *True*

then show *?thesis* **by** *auto*

next

case *False*

then have *?rhs = Suc ((n - 1) - k) * (n choose Suc ((n - 1) - k))*

using *binomial-symmetric*[of k n - 1] *binomial-absorption*[of (n - 1) - k n]

by *simp*

also have *Suc ((n - 1) - k) = n - k*

using *False* **by** *simp*
also have $n \text{ choose } \dots = n \text{ choose } k$
using *False* **by** (*intro binomial-symmetric [symmetric]*) *simp-all*
finally show *?thesis ..*
qed

The generalised absorption companion identity:

lemma *gbinomial-absorb-comp*: $(a - \text{of-nat } k) * (a \text{ gchoose } k) = a * ((a - 1) \text{ gchoose } k)$

using *pochhammer-absorb-comp[of a k]* **by** (*simp add: gbinomial-pochhammer*)

lemma *gbinomial-addition-formula*:

$a \text{ gchoose } (\text{Suc } k) = ((a - 1) \text{ gchoose } (\text{Suc } k)) + ((a - 1) \text{ gchoose } k)$

using *gbinomial-Suc-Suc[of a - 1 k]* **by** (*simp add: algebra-simps*)

lemma *binomial-addition-formula*:

$0 < n \implies n \text{ choose } (\text{Suc } k) = ((n - 1) \text{ choose } (\text{Suc } k)) + ((n - 1) \text{ choose } k)$

by (*subst choose-reduce-nat*) *simp-all*

Equation 5.9 of the reference material [3, p. 159] is a useful summation formula, operating on both indices:

$$\sum_{k \leq n} \binom{r+k}{k} = \binom{r+n+1}{n}, \quad \text{integer } n.$$

lemma *gbinomial-parallel-sum*: $(\sum_{k \leq n}. (a + \text{of-nat } k) \text{ gchoose } k) = (a + \text{of-nat } n + 1) \text{ gchoose } n$

proof (*induct n*)

case 0

then show *?case* **by** *simp*

next

case $(\text{Suc } m)$

then show *?case*

using *gbinomial-Suc-Suc[of (a + of-nat m + 1) m]*

by (*simp add: add-ac*)

qed

94.3.1 Summation on the upper index

Another summation formula is equation 5.10 of the reference material [3, p. 160], aptly named *summation on the upper index*:

$$\sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}, \quad \text{integers } m, n \geq 0.$$

lemma *gbinomial-sum-up-index*:

$(\sum_{j = 0..n}. (\text{of-nat } j \text{ gchoose } k) :: 'a::\text{field-char-0}) = (\text{of-nat } n + 1) \text{ gchoose } (k + 1)$

```

proof (induct n)
  case 0
  show ?case
    using gbinomial-Suc-Suc[of 0 k]
    by (cases k) auto
  next
  case (Suc n)
  then show ?case
    using gbinomial-Suc-Suc[of of-nat (Suc n) :: 'a k]
    by (simp add: add-ac)
qed

```

lemma *gbinomial-index-swap*:

$$((-1) ^ k) * ((- (of-nat n) - 1) gchoose k) = ((-1) ^ n) * ((- (of-nat k) - 1) gchoose n)$$

(is ?lhs = ?rhs)

proof –

```

have ?lhs = (of-nat (k + n) gchoose k)
  by (subst gbinomial-negated-upper) (simp add: power-mult-distrib [symmetric])
also have ... = (of-nat (k + n) gchoose n)
  by (subst gbinomial-of-nat-symmetric) simp-all
also have ... = ?rhs
  by (subst gbinomial-negated-upper) simp
finally show ?thesis .

```

qed

lemma *gbinomial-sum-lower-neg*: $(\sum k \leq m. (a gchoose k) * (- 1) ^ k) = (- 1) ^ m * (a - 1 gchoose m)$

(is ?lhs = ?rhs)

proof –

```

have ?lhs =  $(\sum k \leq m. -(a + 1) + of-nat k gchoose k)$ 
  by (intro sum.cong[OF refl]) (subst gbinomial-negated-upper, simp add: power-mult-distrib)
also have ... =  $- a + of-nat m gchoose m$ 
  by (subst gbinomial-parallel-sum) simp
also have ... = ?rhs
  by (subst gbinomial-negated-upper) (simp add: power-mult-distrib)
finally show ?thesis .

```

qed

lemma *gbinomial-partial-row-sum*:

$$(\sum k \leq m. (a gchoose k) * ((a / 2) - of-nat k)) = ((of-nat m + 1) / 2) * (a gchoose (m + 1))$$

proof (induct m)

case 0

then show ?case **by** simp

next

case (Suc mm)

```

then have  $(\sum k \leq \text{Suc } mm. (a gchoose k) * (a / 2 - of-nat k)) =$ 
   $(a - of-nat (\text{Suc } mm)) * (a gchoose \text{Suc } mm) / 2$ 

```

by (*simp add: field-simps*)
 also have ... = $a * (a - 1 \text{ gchoose } \text{Suc } mm) / 2$
 by (*subst gbinomial-absorb-comp*) (*rule refl*)
 also have ... = $(\text{of-nat } (\text{Suc } mm) + 1) / 2 * (a \text{ gchoose } (\text{Suc } mm + 1))$
 by (*subst gbinomial-absorption [symmetric]*) *simp*
 finally show ?*case* .
 qed

lemma *sum-bounds-lt-plus1*: $(\sum k < mm. f (\text{Suc } k)) = (\sum k = 1 .. mm. f k)$
 by (*induct mm*) *simp-all*

lemma *gbinomial-partial-sum-poly*:

$$\begin{aligned}
 & (\sum k \leq m. (\text{of-nat } m + a \text{ gchoose } k) * x^k * y^{m-k}) = \\
 & (\sum k \leq m. (-a \text{ gchoose } k) * (-x)^k * (x + y)^{m-k}) \\
 & \text{(is ?lhs } m = \text{?rhs } m)
 \end{aligned}$$

proof (*induction m*)

case 0

then show ?*case* by *simp*

next

case (*Suc mm*)

define *G* where $G \ i \ k = (\text{of-nat } i + a \text{ gchoose } k) * x^k * y^{i-k}$ for $i \ k$

define *S* where $S = \text{?lhs}$

have *SG-def*: $S = (\lambda i. (\sum k \leq i. (G \ i \ k)))$

unfolding *S-def G-def* ..

have $S (\text{Suc } mm) = G (\text{Suc } mm) \ 0 + (\sum k = \text{Suc } 0 .. \text{Suc } mm. G (\text{Suc } mm) \ k)$
using *SG-def* by (*simp add: sum.atLeast-Suc-atMost atLeast0AtMost [symmetric]*)
also have $(\sum k = \text{Suc } 0 .. \text{Suc } mm. G (\text{Suc } mm) \ k) = (\sum k = 0 .. mm. G (\text{Suc } mm) \ k)$
 (*Suc k*)

by (*subst sum.shift-bounds-cl-Suc-ivl*) *simp*

also have ... = $(\sum k = 0 .. mm. ((\text{of-nat } mm + a \text{ gchoose } (\text{Suc } k)) +$
 $(\text{of-nat } mm + a \text{ gchoose } k)) * x^{(\text{Suc } k)} * y^{(mm - k)})$

unfolding *G-def* by (*subst gbinomial-addition-formula*) *simp*

also have ... = $(\sum k = 0 .. mm. (\text{of-nat } mm + a \text{ gchoose } (\text{Suc } k)) * x^{(\text{Suc } k)} * y^{(mm - k)}) +$
 $(\sum k = 0 .. mm. (\text{of-nat } mm + a \text{ gchoose } k) * x^{(\text{Suc } k)} * y^{(mm - k)})$

by (*subst sum.distrib [symmetric]*) (*simp add: algebra-simps*)

also have $(\sum k = 0 .. mm. (\text{of-nat } mm + a \text{ gchoose } (\text{Suc } k)) * x^{(\text{Suc } k)} * y^{(mm - k)}) =$
 $(\sum k < \text{Suc } mm. (\text{of-nat } mm + a \text{ gchoose } (\text{Suc } k)) * x^{(\text{Suc } k)} * y^{(mm - k)})$

by (*simp only: atLeast0AtMost lessThan-Suc-atMost*)

also have ... = $(\sum k < mm. (\text{of-nat } mm + a \text{ gchoose } \text{Suc } k) * x^{(\text{Suc } k)} * y^{(mm - k)}) +$
 $(\text{of-nat } mm + a \text{ gchoose } (\text{Suc } mm)) * x^{(\text{Suc } mm)}$

(is - = ?*A* + ?*B*)

by (*subst sum.lessThan-Suc*) *simp*

also have ?*A* = $(\sum k = 1 .. mm. (\text{of-nat } mm + a \text{ gchoose } k) * x^k * y^{(mm - k)} + 1)$

proof (*subst sum-bounds-lt-plus1 [symmetric]*, *intro sum.cong[OF refl]*, *clarify*)

fix k
assume $k < mm$
then have $mm - k = mm - Suc\ k + 1$
by *linarith*
then show $(of\text{-}nat\ mm + a\ gchoose\ Suc\ k) * x^{\wedge} Suc\ k * y^{\wedge} (mm - k) =$
 $(of\text{-}nat\ mm + a\ gchoose\ Suc\ k) * x^{\wedge} Suc\ k * y^{\wedge} (mm - Suc\ k + 1)$
by *(simp only:)*
qed
also have $\dots + ?B = y * (\sum k=1..mm. (G\ mm\ k)) + (of\text{-}nat\ mm + a\ gchoose$
 $(Suc\ mm)) * x^{\wedge} (Suc\ mm)$
unfolding *G-def* **by** *(subst sum-distrib-left) (simp add: algebra-simps)*
also have $(\sum k=0..mm. (of\text{-}nat\ mm + a\ gchoose\ k) * x^{\wedge} (Suc\ k) * y^{\wedge} (mm - k))$
 $= x * (S\ mm)$
unfolding *S-def* **by** *(subst sum-distrib-left) (simp add: atLeast0AtMost alge-*
bra-simps)
also have $(G\ (Suc\ mm)\ 0) = y * (G\ mm\ 0)$
by *(simp add: G-def)*
finally have $S\ (Suc\ mm) =$
 $y * (G\ mm\ 0 + (\sum k=1..mm. (G\ mm\ k))) + (of\text{-}nat\ mm + a\ gchoose\ (Suc$
 $mm)) * x^{\wedge} (Suc\ mm) + x * (S\ mm)$
by *(simp add: ring-distrib)*
also have $G\ mm\ 0 + (\sum k=1..mm. (G\ mm\ k)) = S\ mm$
by *(simp add: sum.atLeast-Suc-atMost[symmetric] SG-def atLeast0AtMost)*
finally have $S\ (Suc\ mm) = (x + y) * (S\ mm) + (of\text{-}nat\ mm + a\ gchoose\ (Suc$
 $mm)) * x^{\wedge} (Suc\ mm)$
by *(simp add: algebra-simps)*
also have $(of\text{-}nat\ mm + a\ gchoose\ (Suc\ mm)) = (-1)^{\wedge} (Suc\ mm) * (- a$
 $gchoose\ (Suc\ mm))$
by *(subst gbinomial-negated-upper) simp*
also have $(-1)^{\wedge} Suc\ mm * (- a\ gchoose\ Suc\ mm) * x^{\wedge} Suc\ mm =$
 $(- a\ gchoose\ (Suc\ mm)) * (-x)^{\wedge} Suc\ mm$
by *(simp add: power-minus[of x])*
also have $(x + y) * S\ mm + \dots = (x + y) * ?rhs\ mm + (- a\ gchoose\ (Suc$
 $mm)) * (-x)^{\wedge} Suc\ mm$
unfolding *S-def* **by** *(subst Suc.IH) simp*
also have $(x + y) * ?rhs\ mm = (\sum n \leq mm. ((- a\ gchoose\ n) * (-x)^{\wedge} n * (x$
 $+ y)^{\wedge} (Suc\ mm - n)))$
by *(subst sum-distrib-left, rule sum.cong) (simp-all add: Suc-diff-le)*
also have $\dots + (- a\ gchoose\ (Suc\ mm)) * (-x)^{\wedge} Suc\ mm =$
 $(\sum n \leq Suc\ mm. (- a\ gchoose\ n) * (-x)^{\wedge} n * (x + y)^{\wedge} (Suc\ mm - n))$
by *simp*
finally show *?case*
by *(simp only: S-def)*
qed

lemma *gbinomial-partial-sum-poly-xpos:*

$(\sum k \leq m. (of\text{-}nat\ m + a\ gchoose\ k) * x^{\wedge} k * y^{\wedge} (m - k)) =$
 $(\sum k \leq m. (of\text{-}nat\ k + a - 1\ gchoose\ k) * x^{\wedge} k * (x + y)^{\wedge} (m - k))$ (**is** *?lhs =*
?rhs)

proof –

have $?lhs = (\sum k \leq m. (- a \text{ gchoose } k) * (- x) ^ k * (x + y) ^ (m - k))$
by (*simp add: gbinomial-partial-sum-poly*)
also have $\dots = (\sum k \leq m. (-1) ^ k * (\text{of-nat } k - a - 1 \text{ gchoose } k) * (- x) ^ k * (x + y) ^ (m - k))$
by (*metis (no-types, opaque-lifting) gbinomial-negated-upper*)
also have $\dots = ?rhs$
by (*intro sum.cong*) (*auto simp flip: power-mult-distrib*)
finally show $?thesis$.
qed

lemma *binomial-r-part-sum*: $(\sum k \leq m. (2 * m + 1 \text{ choose } k)) = 2 ^ (2 * m)$

proof –

have $2 * 2 ^ (2 * m) = (\sum k = 0 .. (2 * m + 1). (2 * m + 1 \text{ choose } k))$
using *choose-row-sum* [**where** $n = 2 * m + 1$] **by** (*simp add: atMost-atLeast0*)
also have $(\sum k = 0 .. (2 * m + 1). (2 * m + 1 \text{ choose } k)) =$
 $(\sum k = 0 .. m. (2 * m + 1 \text{ choose } k)) +$
 $(\sum k = m + 1 .. 2 * m + 1. (2 * m + 1 \text{ choose } k))$
using *sum.ub-add-nat* [*of* $0 \ m \ \lambda k. 2 * m + 1 \text{ choose } k \ m + 1$]
by (*simp add: mult-2*)
also have $(\sum k = m + 1 .. 2 * m + 1. (2 * m + 1 \text{ choose } k)) =$
 $(\sum k = 0 .. m. (2 * m + 1 \text{ choose } (k + (m + 1))))$
by (*subst sum.shift-bounds-cl-nat-ivl* [*symmetric*]) (*simp add: mult-2*)
also have $\dots = (\sum k = 0 .. m. (2 * m + 1 \text{ choose } (m - k)))$
by (*intro sum.cong* [*OF refl*], *subst binomial-symmetric*) *simp-all*
also have $\dots = (\sum k = 0 .. m. (2 * m + 1 \text{ choose } k))$
using *sum.atLeastAtMost-rev* [*of* $\lambda k. 2 * m + 1 \text{ choose } (m - k) \ 0 \ m$]
by *simp*
also have $\dots + \dots = 2 * \dots$
by *simp*
finally show $?thesis$
by (*subst (asm) mult-cancel1*) (*simp add: atLeast0AtMost*)
qed

lemma *gbinomial-r-part-sum*: $(\sum k \leq m. (2 * (\text{of-nat } m) + 1 \text{ gchoose } k)) = 2 ^ (2 * m)$

(**is** $?lhs = ?rhs$)

proof –

have $?lhs = \text{of-nat } (\sum k \leq m. (2 * m + 1 \text{ choose } k))$
by (*simp add: binomial-gbinomial add-ac*)
also have $\dots = \text{of-nat } (2 ^ (2 * m))$
by (*subst binomial-r-part-sum*) (*rule refl*)
finally show $?thesis$ **by** *simp*
qed

lemma *gbinomial-sum-nat-pow2*:

$(\sum k \leq m. (\text{of-nat } (m + k) \text{ gchoose } k :: 'a :: \text{field-char-0}) / 2 ^ k) = 2 ^ m$

(**is** $?lhs = ?rhs$)

proof –

have $2^m * 2^m = (2^{(2*m)}) :: 'a$
by (*induct m*) *simp-all*
also have $\dots = (\sum_{k \leq m}. (2 * (of\text{-}nat\ m) + 1\ gchoose\ k))$
using *gbinomial-r-part-sum ..*
also have $\dots = (\sum_{k \leq m}. (of\text{-}nat\ (m + k)\ gchoose\ k) * 2^{(m - k)})$
using *gbinomial-partial-sum-poly-xpos*[**where** $x=1$ **and** $y=1$ **and** $a=of\text{-}nat\ m$
 $+ 1$ **and** $m=m$]
by (*simp add: add-ac*)
also have $\dots = 2^m * (\sum_{k \leq m}. (of\text{-}nat\ (m + k)\ gchoose\ k) / 2^k)$
by (*subst sum-distrib-left*) (*simp add: algebra-simps power-diff*)
finally show *?thesis*
by (*subst (asm) mult-left-cancel*) *simp-all*
qed

lemma *gbinomial-trinomial-revision:*

assumes $k \leq m$
shows $(a\ gchoose\ m) * (of\text{-}nat\ m\ gchoose\ k) = (a\ gchoose\ k) * (a - of\text{-}nat\ k\ gchoose\ (m - k))$
proof –
have $(a\ gchoose\ m) * (of\text{-}nat\ m\ gchoose\ k) = (a\ gchoose\ m) * fact\ m / (fact\ k * fact\ (m - k))$
using *assms* **by** (*simp add: binomial-gbinomial [symmetric] binomial-fact*)
also have $\dots = (a\ gchoose\ k) * (a - of\text{-}nat\ k\ gchoose\ (m - k))$
using *assms* **by** (*simp add: gbinomial-pochhammer power-diff pochhammer-product*)
finally show *?thesis .*
qed

Versions of the theorems above for the natural-number version of "choose"

lemma *binomial-altdef-of-nat:*

$k \leq n \implies of\text{-}nat\ (n\ choose\ k) = (\prod_{i = 0..<k}. of\text{-}nat\ (n - i) / of\text{-}nat\ (k - i))$
 $:: 'a$
for $n\ k :: nat$ **and** $x :: 'a::field\text{-}char\ 0$
by (*simp add: gbinomial-altdef-of-nat binomial-gbinomial of-nat-diff*)

lemma *binomial-ge-n-over-k-pow-k:* $k \leq n \implies (of\text{-}nat\ n / of\text{-}nat\ k :: 'a)^k \leq of\text{-}nat\ (n\ choose\ k)$

for $k\ n :: nat$ **and** $x :: 'a::linordered\text{-}field$
by (*simp add: gbinomial-ge-n-over-k-pow-k binomial-gbinomial of-nat-diff*)

lemma *binomial-le-pow:*

assumes $r \leq n$
shows $n\ choose\ r \leq n^r$
proof –
have $n\ choose\ r \leq fact\ n\ div\ fact\ (n - r)$
using *assms* **by** (*subst binomial-fact-lemma[symmetric]*) *auto*
with *fact-div-fact-le-pow* [*OF assms*] **show** *?thesis*
by *auto*
qed

lemma *binomial-altdef-nat*: $k \leq n \implies n \text{ choose } k = \text{fact } n \text{ div } (\text{fact } k * \text{fact } (n - k))$

for $k \ n :: \text{nat}$

by (*subst binomial-fact-lemma [symmetric]*) *auto*

lemma *choose-dvd*:

assumes $k \leq n$ **shows** $\text{fact } k * \text{fact } (n - k) \text{ dvd } (\text{fact } n :: 'a::\text{linordered-semidom})$

unfolding *dvd-def*

proof

show $\text{fact } n = \text{fact } k * \text{fact } (n - k) * \text{of-nat } (n \text{ choose } k)$

by (*metis assms binomial-fact-lemma of-nat-fact of-nat-mult*)

qed

lemma *fact-fact-dvd-fact*:

$\text{fact } k * \text{fact } n \text{ dvd } (\text{fact } (k + n) :: 'a::\text{linordered-semidom})$

by (*metis add.commute add-diff-cancel-left' choose-dvd le-add2*)

lemma *choose-mult-lemma*:

$((m + r + k) \text{ choose } (m + k)) * ((m + k) \text{ choose } k) = ((m + r + k) \text{ choose } k) * ((m + r) \text{ choose } m)$

(**is** *?lhs = -*)

proof *-*

have *?lhs =*

$\text{fact } (m + r + k) \text{ div } (\text{fact } (m + k) * \text{fact } (m + r - m)) * (\text{fact } (m + k) \text{ div } (\text{fact } k * \text{fact } m))$

by (*simp add: binomial-altdef-nat*)

also have $\dots = \text{fact } (m + r + k) * \text{fact } (m + k) \text{ div}$

$(\text{fact } (m + k) * \text{fact } (m + r - m) * (\text{fact } k * \text{fact } m))$

by (*metis add-implies-diff add-le-mono1 choose-dvd diff-cancel2 div-mult-div-if-dvd le-add1 le-add2*)

also have $\dots = \text{fact } (m + r + k) \text{ div } (\text{fact } r * (\text{fact } k * \text{fact } m))$

by (*auto simp: algebra-simps fact-fact-dvd-fact*)

also have $\dots = (\text{fact } (m + r + k) * \text{fact } (m + r)) \text{ div } (\text{fact } r * (\text{fact } k * \text{fact } m) * \text{fact } (m + r))$

by *simp*

also have $\dots =$

$(\text{fact } (m + r + k) \text{ div } (\text{fact } k * \text{fact } (m + r)) * (\text{fact } (m + r) \text{ div } (\text{fact } r * \text{fact } m)))$

by (*auto simp: div-mult-div-if-dvd fact-fact-dvd-fact algebra-simps*)

finally show *?thesis*

by (*simp add: binomial-altdef-nat mult.commute*)

qed

The "Subset of a Subset" identity.

lemma *choose-mult*:

$k \leq m \implies m \leq n \implies (n \text{ choose } m) * (m \text{ choose } k) = (n \text{ choose } k) * ((n - k) \text{ choose } (m - k))$

using *choose-mult-lemma [of m-k n-m k]* **by** *simp*

lemma *of-nat-binomial-eq-mult-binomial-Suc*:
assumes $k \leq n$
shows $(\text{of-nat} :: (\text{nat} \Rightarrow ('a :: \text{field-char-0}))) (n \text{ choose } k) = \text{of-nat } (n + 1 - k)$
 $/ \text{of-nat } (n + 1) * \text{of-nat } (\text{Suc } n \text{ choose } k)$
proof (*cases k*)
case 0 then show *?thesis*
using *of-nat-neq-0* **by** *auto*
next
case (*Suc l*)
have $\text{of-nat } (n + 1) * (\prod_{i=0..<k.} \text{of-nat } (n - i)) = (\text{of-nat} :: (\text{nat} \Rightarrow 'a)) (n + 1 - k) * (\prod_{i=0..<k.} \text{of-nat } (\text{Suc } n - i))$
using *prod.atLeast0-lessThan-Suc* [**where** *?'a = 'a, symmetric, of $\lambda i. \text{of-nat } (\text{Suc } n - i) k$*]
by (*simp add: ac-simps prod.atLeast0-lessThan-Suc-shift del: prod.op-ivl-Suc*)
also have $\dots = (\text{of-nat} :: (\text{nat} \Rightarrow 'a)) (\text{Suc } n - k) * (\prod_{i=0..<k.} \text{of-nat } (\text{Suc } n - i))$
by (*simp add: Suc atLeast0-atMost-Suc atLeastLessThanSuc-atLeastAtMost*)
also have $\dots = (\text{of-nat} :: (\text{nat} \Rightarrow 'a)) (n + 1 - k) * (\prod_{i=0..<k.} \text{of-nat } (\text{Suc } n - i))$
by (*simp only: Suc-eq-plus1*)
finally have $(\prod_{i=0..<k.} \text{of-nat } (n - i)) = (\text{of-nat} :: (\text{nat} \Rightarrow 'a)) (n + 1 - k)$
 $/ \text{of-nat } (n + 1) * (\prod_{i=0..<k.} \text{of-nat } (\text{Suc } n - i))$
using *of-nat-neq-0* **by** (*auto simp: mult.commute divide-simps*)
with *assms* **show** *?thesis*
by (*simp add: binomial-altdef-of-nat prod-dividef*)
qed

94.4 More on Binomial Coefficients

lemma *choose-one: n choose 1 = n for n :: nat*
by *simp*

The famous inclusion-exclusion formula for the cardinality of a union

lemma *int-card-UNION*:
assumes *finite A*
and $\forall k \in A. \text{finite } k$
shows $\text{int } (\text{card } (\bigcup A)) = (\sum I \mid I \subseteq A \wedge I \neq \{\}). (-1)^{\wedge} (\text{card } I + 1) * \text{int } (\text{card } (\bigcap I))$
(is ?lhs = ?rhs)
proof –
have $?rhs = (\sum I \mid I \subseteq A \wedge I \neq \{\}). (-1)^{\wedge} (\text{card } I + 1) * (\sum - \in \bigcap I. 1)$
by *simp*
also have $\dots = (\sum I \mid I \subseteq A \wedge I \neq \{\}). (\sum - \in \bigcap I. (-1)^{\wedge} (\text{card } I + 1))$
by (*subst sum-distrib-left*) *simp*
also have $\dots = (\sum (I, -) \in \text{Sigma } \{I. I \subseteq A \wedge I \neq \{\}\} \text{Inter. } (-1)^{\wedge} (\text{card } I + 1))$
using *assms* **by** (*subst sum.Sigma*) *auto*
also have $\dots = (\sum (x, I) \in (\text{SIGMA } x: \text{UNIV. } \{I. I \subseteq A \wedge I \neq \{\} \wedge x \in \bigcap I\}). (-1)^{\wedge} (\text{card } I + 1))$


```

    by (rule sum.reindex-cong [where l = λ(x, y). (y, x)]) (auto intro: inj-onI)
    also have ... = (∑ (x, I) ∈ (SIGMA x: ∪ A. {I. I ⊆ A ∧ I ≠ {} ∧ x ∈ ∩ I}).
(- 1) ^ (card I + 1))
    using assms
    by (auto intro!: sum.mono-neutral-cong-right finite-SigmaI2 intro: finite-subset[where
B=∪ A])
    also have ... = (∑ x ∈ ∪ A. (∑ I | I ⊆ A ∧ I ≠ {} ∧ x ∈ ∩ I. (- 1) ^ (card I
+ 1)))
    using assms by (subst sum.Sigma) auto
    also have ... = (∑ - ∈ ∪ A. 1) (is sum ?lhs - = -)
    proof (rule sum.cong[OF refl])
      fix x
      assume x: x ∈ ∪ A
      define K where K = {X ∈ A. x ∈ X}
      with ⟨finite A⟩ have K: finite K
      by auto
      let ?I = λi. {I. I ⊆ A ∧ card I = i ∧ x ∈ ∩ I}
      have inj-on snd (SIGMA i: {1..card A}. ?I i)
      using assms by (auto intro!: inj-onI)
      moreover have [symmetric]: snd ‘ (SIGMA i: {1..card A}. ?I i) = {I. I ⊆ A
∧ I ≠ {} ∧ x ∈ ∩ I}
      using assms
      by (auto intro!: rev-image-eqI [where x=(card a, a) for a]
simp add: card-gt-0-iff [folded Suc-le-eq]
dest: finite-subset intro: card-mono)
      ultimately have ?lhs x = (∑ (i, I) ∈ (SIGMA i: {1..card A}. ?I i). (- 1) ^ (i
+ 1))
      by (rule sum.reindex-cong [where l = snd]) fastforce
      also have ... = (∑ i=1..card A. (∑ I | I ⊆ A ∧ card I = i ∧ x ∈ ∩ I. (- 1)
^ (i + 1)))
      using assms by (subst sum.Sigma) auto
      also have ... = (∑ i=1..card A. (- 1) ^ (i + 1) * (∑ I | I ⊆ A ∧ card I = i
∧ x ∈ ∩ I. 1))
      by (subst sum-distrib-left) simp
      also have ... = (∑ i=1..card K. (- 1) ^ (i + 1) * (∑ I | I ⊆ K ∧ card I =
i. 1))
      (is - = ?rhs)
    proof (rule sum.mono-neutral-cong-right [rule-format])
      show finite {1..card A}
      by simp
      show {1..card K} ⊆ {1..card A}
      using ⟨finite A⟩ by (auto simp add: K-def intro: card-mono)
    next
      fix i
      assume i ∈ {1..card A} - {1..card K}
      then have i: i ≤ card A card K < i
      by auto
      have {I. I ⊆ A ∧ card I = i ∧ x ∈ ∩ I} = {I. I ⊆ K ∧ card I = i}
      by (auto simp add: K-def)

```

```

    also have ... = {}
      using <finite A> i by (auto simp add: K-def dest: card-mono[rotated 1])
    finally show (- 1) ^ (i + 1) * (∑ I | I ⊆ A ∧ card I = i ∧ x ∈ ∩ I. 1 ::
int) = 0
      by (metis mult-zero-right sum.empty)
  next
  fix i
  have (∑ I | I ⊆ A ∧ card I = i ∧ x ∈ ∩ I. 1) = (∑ I | I ⊆ K ∧ card I =
i. 1 :: int)
    (is ?lhs = ?rhs)
    by (rule sum.cong) (auto simp add: K-def)
  then show (- 1) ^ (i + 1) * ?lhs = (- 1) ^ (i + 1) * ?rhs
    by simp
  qed
  also have {I. I ⊆ K ∧ card I = 0} = {{}}
    using assms by (auto simp add: card-eq-0-iff K-def dest: finite-subset)
  then have ?rhs = (∑ i = 0..card K. (- 1) ^ (i + 1) * (∑ I | I ⊆ K ∧ card
I = i. 1 :: int)) + 1
    by (subst (2) sum.atLeast-Suc-atMost) simp-all
  also have ... = (∑ i = 0..card K. (- 1) * ((- 1) ^ i * int (card K choose
i))) + 1
    using K by (subst n-subsets[symmetric]) simp-all
  also have ... = - (∑ i = 0..card K. (- 1) ^ i * int (card K choose i)) + 1
    by (subst sum-distrib-left[symmetric]) simp
  also have ... = - ((- 1 + 1) ^ card K) + 1
    by (subst binomial-ring) (simp add: ac-simps atMost-atLeast0)
  also have ... = 1
    using x K by (auto simp add: K-def card-gt-0-iff)
  finally show ?lhs x = 1 .
  qed
  also have ... = int (card (∪ A))
    by simp
  finally show ?thesis ..
  qed

```

lemma *card-UNION*:

```

assumes finite A
  and ∀ k ∈ A. finite k
shows card (∪ A) = nat (∑ I | I ⊆ A ∧ I ≠ {}. (- 1) ^ (card I + 1) * int
(card (∩ I)))
  by (simp only: flip: int-card-UNION [OF assms])

```

lemma *card-UNION-nonneg*:

```

assumes finite A
  and ∀ k ∈ A. finite k
shows (∑ I | I ⊆ A ∧ I ≠ {}. (- 1) ^ (card I + 1) * int (card (∩ I))) ≥ 0
  using int-card-UNION [OF assms] by presburger

```

The number of nat lists of length m summing to N is $N + m - 1$ choose

N :

lemma *card-length-sum-list-rec*:

assumes $m \geq 1$

shows $\text{card } \{l::\text{nat list. length } l = m \wedge \text{sum-list } l = N\} =$
 $\text{card } \{l. \text{length } l = (m - 1) \wedge \text{sum-list } l = N\} +$
 $\text{card } \{l. \text{length } l = m \wedge \text{sum-list } l + 1 = N\}$
(is $\text{card } ?C = \text{card } ?A + \text{card } ?B)$

proof –

let $?A' = \{l. \text{length } l = m \wedge \text{sum-list } l = N \wedge \text{hd } l = 0\}$

let $?B' = \{l. \text{length } l = m \wedge \text{sum-list } l = N \wedge \text{hd } l \neq 0\}$

let $?f = \lambda l. 0 \# l$

let $?g = \lambda l. (\text{hd } l + 1) \# \text{tl } l$

have $1: xs \neq [] \implies x = \text{hd } xs \implies x \# \text{tl } xs = xs$ **for** $x :: \text{nat}$ **and** xs

by *simp*

have $2: xs \neq [] \implies \text{sum-list}(\text{tl } xs) = \text{sum-list } xs - \text{hd } xs$ **for** $xs :: \text{nat list}$

by (*auto simp add: neq-Nil-conv*)

have $f: \text{bij-betw } ?f ?A ?A'$

by (*rule bij-betw-byWitness[where $f' = \text{tl}$]*) (*use assms in <auto simp: 2 1 simp flip: length-0-conv>*)

have $3: xs \neq [] \implies \text{hd } xs + (\text{sum-list } xs - \text{hd } xs) = \text{sum-list } xs$ **for** $xs :: \text{nat list}$

by (*metis 1 sum-list-simps(2) 2*)

have $g: \text{bij-betw } ?g ?B ?B'$

apply (*rule bij-betw-byWitness[where $f' = \lambda l. (\text{hd } l - 1) \# \text{tl } l$]*)

using *assms*

by (*auto simp: 2 simp flip: length-0-conv intro!: 3*)

have $\text{fin}: \text{finite } \{xs. \text{size } xs = M \wedge \text{set } xs \subseteq \{0..<N\}\}$ **for** $M N :: \text{nat}$

using *finite-lists-length-eq[OF finite-atLeastLessThan] conj-commute* **by** *auto*

have $\text{fin-A}: \text{finite } ?A$ **using** *fin[of - $N+1$]*

by (*intro finite-subset[where $?A = ?A$ and $?B = \{xs. \text{size } xs = m - 1 \wedge \text{set } xs \subseteq \{0..<N+1\}\}$]*)

(*auto simp: member-le-sum-list less-Suc-eq-le*)

have $\text{fin-B}: \text{finite } ?B$

by (*intro finite-subset[where $?A = ?B$ and $?B = \{xs. \text{size } xs = m \wedge \text{set } xs \subseteq \{0..<N\}\}$]*)

(*auto simp: member-le-sum-list less-Suc-eq-le fin*)

have $\text{uni}: ?C = ?A' \cup ?B'$

by *auto*

have $\text{disj}: ?A' \cap ?B' = \{\}$ **by** *blast*

have $\text{card } ?C = \text{card}(?A' \cup ?B')$

using *uni* **by** *simp*

also have $\dots = \text{card } ?A + \text{card } ?B$

using *card-Un-disjoint[OF - - disj] bij-betw-finite[OF f] bij-betw-finite[OF g]*

bij-betw-same-card[OF f] bij-betw-same-card[OF g] fin-A fin-B

by *presburger*

finally show *?thesis* .

qed

lemma *card-length-sum-list*: $\text{card } \{l::\text{nat list. size } l = m \wedge \text{sum-list } l = N\} = (N + m - 1)$ *choose* N

— by Holden Lee, tidied by Tobias Nipkow

```

proof (cases m)
  case 0
  then show ?thesis
    by (cases N) (auto cong: conj-cong)
next
  case (Suc m')
  have m: m ≥ 1
    by (simp add: Suc)
  then show ?thesis
  proof (induct N + m - 1 arbitrary: N m)
    case 0 — In the base case, the only solution is [0].
    have [simp]: {l::nat list. length l = Suc 0 ∧ (∀ n∈set l. n = 0)} = {[0]}
      by (auto simp: length-Suc-conv)
    have m = 1 ∧ N = 0
      using 0 by linarith
    then show ?case
      by simp
  next
  case (Suc k)
  have c1: card {l::nat list. size l = (m - 1) ∧ sum-list l = N} = (N + (m -
1) - 1) choose N
  proof (cases m = 1)
    case True
    with Suc.hyps have N ≥ 1
      by auto
    with True show ?thesis
      by (simp add: binomial-eq-0)
  next
  case False
  then show ?thesis
    using Suc by fastforce
  qed
from Suc have c2: card {l::nat list. size l = m ∧ sum-list l + 1 = N} =
  (if N > 0 then ((N - 1) + m - 1) choose (N - 1) else 0)
proof -
  have *: n > 0 ⇒ Suc m = n ↔ m = n - 1 for m n
    by arith
  from Suc have N > 0 ⇒
  card {l::nat list. size l = m ∧ sum-list l + 1 = N} =
  ((N - 1) + m - 1) choose (N - 1)
    by (simp add: *)
  then show ?thesis
    by auto
  qed
from Suc.prem1 have (card {l::nat list. size l = (m - 1) ∧ sum-list l = N} +
  card {l::nat list. size l = m ∧ sum-list l + 1 = N}) = (N + m - 1)
  choose N
  by (auto simp: c1 c2 choose-choose-nat[of N + m - 1 N] simp del: One-nat-def)

```

then show *?case*
using *card-length-sum-list-rec[OF Suc.premis]* **by** *auto*
qed
qed

lemma *card-disjoint-shuffles:*

assumes *set xs ∩ set ys = {}*
shows *card (shuffles xs ys) = (length xs + length ys) choose length xs*
using *assms*
proof (*induction xs ys rule: shuffles.induct*)
case (*∃ x xs y ys*)
have *shuffles (x # xs) (y # ys) = (#) x ‘ shuffles xs (y # ys) ∪ (#) y ‘ shuffles (x # xs) ys*
by (*rule shuffles.simps*)
also have *card ... = card ((#) x ‘ shuffles xs (y # ys)) + card ((#) y ‘ shuffles (x # xs) ys)*
by (*rule card-Un-disjoint*) (*insert ∃.prems, auto*)
also have *card ((#) x ‘ shuffles xs (y # ys)) = card (shuffles xs (y # ys))*
by (*rule card-image*) *auto*
also have *... = (length xs + length (y # ys)) choose length xs*
using *∃.prems* **by** (*intro ∃.IH*) *auto*
also have *card ((#) y ‘ shuffles (x # xs) ys) = card (shuffles (x # xs) ys)*
by (*rule card-image*) *auto*
also have *... = (length (x # xs) + length ys) choose length (x # xs)*
using *∃.prems* **by** (*intro ∃.IH*) *auto*
also have *length xs + length (y # ys) choose length xs + ... = (length (x # xs) + length (y # ys)) choose length (x # xs)* **by** *simp*
finally show *?case .*
qed *auto*

lemma *Suc-times-binomial-add: Suc a * (Suc (a + b) choose Suc a) = Suc b * (Suc (a + b) choose a)*

— by Lukas Bulwahn

proof —

have *dvd: Suc a * (fact a * fact b) dvd fact (Suc (a + b))* **for** *a b*
using *fact-fact-dvd-fact[of Suc a b, where 'a=nat]*
by (*simp only: fact-Suc add-Suc[symmetric] of-nat-id mult.assoc*)
have *Suc a * (fact (Suc (a + b)) div (Suc a * fact a * fact b)) = Suc a * fact (Suc (a + b)) div (Suc a * (fact a * fact b))*
by (*subst div-mult-swap[symmetric]; simp only: mult.assoc dvd*)
also have *... = Suc b * fact (Suc (a + b)) div (Suc b * (fact a * fact b))*
by (*simp only: div-mult-mult1*)
also have *... = Suc b * (fact (Suc (a + b)) div (Suc b * (fact a * fact b)))*
using *dvd[of b a]* **by** (*subst div-mult-swap[symmetric]; simp only: ac-simps dvd*)
finally show *?thesis*
by (*subst (1 2) binomial-altdef-nat*)
(simp-all only: ac-simps diff-Suc-Suc Suc-diff-le diff-add-inverse fact-Suc of-nat-id)
qed

94.5 Executable code

lemma *gbinomial-code* [*code*]:

a *gchoose* *k* =
 (if *k* = 0 then 1
 else fold-atLeastAtMost-nat (λk acc. (*a* - of-nat *k*) * acc) 0 (*k* - 1) 1 / fact
k)

by (*cases* *k*)

(*simp-all* add: *gbinomial-prod-rev prod-atLeastAtMost-code* [*symmetric*]
atLeastLessThanSuc-atLeastAtMost)

lemma *binomial-code* [*code*]:

n *choose* *k* =

(if *k* > *n* then 0

else if 2 * *k* > *n* then *n* *choose* (*n* - *k*)

else (fold-atLeastAtMost-nat (*) (*n* - *k* + 1) *n* 1 div fact *k*))

proof –

{

assume $k \leq n$

then have $\{1..n\} = \{1..n-k\} \cup \{n-k+1..n\}$ **by** *auto*

then have (fact *n* :: nat) = fact (*n* - *k*) * $\prod \{n-k+1..n\}$

by (*simp* add: *prod.union-disjoint fact-prod*)

}

then show ?*thesis* **by** (*auto simp: binomial-altdef-nat mult-ac prod-atLeastAtMost-code*)

qed

end

95 Main HOL

Classical Higher-order Logic – only “Main”, excluding real and complex numbers etc.

theory *Main*

imports

Predicate-Compile

Quickcheck-Narrowing

Mirabelle

Extraction

Nunchaku

BNF-Greatest-Fixpoint

Filter

Conditionally-Complete-Lattices

Binomial

GCD

Divides

begin

95.1 Namespace cleanup

hide-const (open)

*czero cfinite cfinite csum cone ctwo Csum cprod cexp image2 image2p vimage2p
Gr Grp collect
fstS sndS setL setR convol pick-middlep fstOp sndOp csquare relImage relInvImage
Succ Shift
shift proj id-bnf*

hide-fact (open) *id-bnf-def type-definition-id-bnf-UNIV*

95.2 Syntax cleanup

no-notation

*ordLeq2 (infix <=o 50) and
ordLeq3 (infix ≤o 50) and
ordLess2 (infix <o 50) and
ordIso2 (infix =o 50) and
card-of (|-) and
BNF-Cardinal-Arithmetic.csum (infixr +c 65) and
BNF-Cardinal-Arithmetic.cprod (infixr *c 80) and
BNF-Cardinal-Arithmetic.cexp (infixr ^c 90) and
BNF-Def.convol ((-,/ -))*

bundle *cardinal-syntax*

begin

notation

*ordLeq2 (infix <=o 50) and
ordLeq3 (infix ≤o 50) and
ordLess2 (infix <o 50) and
ordIso2 (infix =o 50) and
card-of (|-) and
BNF-Cardinal-Arithmetic.csum (infixr +c 65) and
BNF-Cardinal-Arithmetic.cprod (infixr *c 80) and
BNF-Cardinal-Arithmetic.cexp (infixr ^c 90)*

alias *cfinite = BNF-Cardinal-Arithmetic.cfinite*

alias *czero = BNF-Cardinal-Arithmetic.czero*

alias *cone = BNF-Cardinal-Arithmetic.cone*

alias *ctwo = BNF-Cardinal-Arithmetic.ctwo*

end

95.3 Lattice syntax

bundle *lattice-syntax*

begin

notation

```

  bot ( $\perp$ ) and
  top ( $\top$ ) and
  inf (infixl  $\sqcap$  70) and
  sup (infixl  $\sqcup$  65) and
  Inf ( $\sqcap$  - [900] 900) and
  Sup ( $\sqcup$  - [900] 900)

syntax
  -INF1   :: ptrns  $\Rightarrow$  'b  $\Rightarrow$  'b      (( $\exists \sqcap$  -./ -) [0, 10] 10)
  -INF    :: ptrn   $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b (( $\exists \sqcap$  - $\in$ ./ -) [0, 0, 10] 10)
  -SUP1   :: ptrns  $\Rightarrow$  'b  $\Rightarrow$  'b      (( $\exists \sqcup$  -./ -) [0, 10] 10)
  -SUP    :: ptrn   $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b (( $\exists \sqcup$  - $\in$ ./ -) [0, 0, 10] 10)

end

bundle no-lattice-syntax
begin

no-notation
  bot ( $\perp$ ) and
  top ( $\top$ ) and
  inf (infixl  $\sqcap$  70) and
  sup (infixl  $\sqcup$  65) and
  Inf ( $\sqcap$  - [900] 900) and
  Sup ( $\sqcup$  - [900] 900)

no-syntax
  -INF1   :: ptrns  $\Rightarrow$  'b  $\Rightarrow$  'b      (( $\exists \sqcap$  -./ -) [0, 10] 10)
  -INF    :: ptrn   $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b (( $\exists \sqcap$  - $\in$ ./ -) [0, 0, 10] 10)
  -SUP1   :: ptrns  $\Rightarrow$  'b  $\Rightarrow$  'b      (( $\exists \sqcup$  -./ -) [0, 10] 10)
  -SUP    :: ptrn   $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b (( $\exists \sqcup$  - $\in$ ./ -) [0, 0, 10] 10)

end

unbundle no-lattice-syntax

end

```

96 Archimedean Fields, Floor and Ceiling Functions

```

theory Archimedean-Field
imports Main
begin

```

```

lemma cInf-abs-ge:

```

```

  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S  $\neq$  {}

```



```

    and bdd:  $\bigwedge x. x \in S \implies |x| \leq a$ 
  shows  $|Inf\ S| \leq a$ 
proof -
  have  $Sup\ (uminus\ 'S) = -\ (Inf\ S)$ 
  proof (rule antisym)
    have  $\bigwedge x. x \in S \implies bdd\text{-above}\ (uminus\ 'S)$ 
    using bdd by (force simp: abs-le-iff bdd-above-def)
  then show  $-\ (Inf\ S) \leq Sup\ (uminus\ 'S)$ 
    by (meson cInf-greatest [OF  $\langle S \neq \{\} \rangle$ ] cSUP-upper minus-le-iff)
  next
    have *:  $\bigwedge x. x \in S \implies Inf\ S \leq x$ 
    by (meson abs-le-iff bdd bdd-below-def cInf-lower minus-le-iff)
  show  $Sup\ (uminus\ 'S) \leq -\ Inf\ S$ 
    using  $\langle S \neq \{\} \rangle$  by (force intro: * cSup-least)
  qed
with cSup-abs-le [of uminus 'S] assms show ?thesis
  by fastforce
qed

```

```

lemma cSup-asclose:
  fixes  $S :: 'a::\{linordered-idom,conditionally-complete-linorder\}$  set
  assumes  $S: S \neq \{\}$ 
    and  $b: \forall x \in S. |x - l| \leq e$ 
  shows  $|Sup\ S - l| \leq e$ 
proof -
  have *:  $|x - l| \leq e \iff l - e \leq x \wedge x \leq l + e$  for  $x\ l\ e :: 'a$ 
    by arith
  have bdd-above S
    using b by (auto intro!: bdd-aboveI[of - l + e])
  with S b show ?thesis
    unfolding * by (auto intro!: cSup-upper2 cSup-least)
qed

```

```

lemma cInf-asclose:
  fixes  $S :: 'a::\{linordered-idom,conditionally-complete-linorder\}$  set
  assumes  $S: S \neq \{\}$ 
    and  $b: \forall x \in S. |x - l| \leq e$ 
  shows  $|Inf\ S - l| \leq e$ 
proof -
  have *:  $|x - l| \leq e \iff l - e \leq x \wedge x \leq l + e$  for  $x\ l\ e :: 'a$ 
    by arith
  have bdd-below S
    using b by (auto intro!: bdd-belowI[of - l - e])
  with S b show ?thesis
    unfolding * by (auto intro!: cInf-lower2 cInf-greatest)
qed

```

96.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

```
class archimedean-field = linordered-field +
  assumes ex-le-of-int:  $\exists z. x \leq \text{of-int } z$ 
```

```
lemma ex-less-of-int:  $\exists z. x < \text{of-int } z$ 
  for  $x :: 'a::\text{archimedean-field}$ 
proof -
  from ex-le-of-int obtain  $z$  where  $x \leq \text{of-int } z ..$ 
  then have  $x < \text{of-int } (z + 1)$  by simp
  then show ?thesis ..
qed
```

```
lemma ex-of-int-less:  $\exists z. \text{of-int } z < x$ 
  for  $x :: 'a::\text{archimedean-field}$ 
proof -
  from ex-less-of-int obtain  $z$  where  $-x < \text{of-int } z ..$ 
  then have  $\text{of-int } (-z) < x$  by simp
  then show ?thesis ..
qed
```

```
lemma reals-Archimedean2:  $\exists n. x < \text{of-nat } n$ 
  for  $x :: 'a::\text{archimedean-field}$ 
proof -
  obtain  $z$  where  $x < \text{of-int } z$ 
  using ex-less-of-int ..
  also have  $\dots \leq \text{of-int } (\text{int } (\text{nat } z))$ 
  by simp
  also have  $\dots = \text{of-nat } (\text{nat } z)$ 
  by (simp only: of-int-of-nat-eq)
  finally show ?thesis ..
qed
```

```
lemma real-arch-simple:  $\exists n. x \leq \text{of-nat } n$ 
  for  $x :: 'a::\text{archimedean-field}$ 
proof -
  obtain  $n$  where  $x < \text{of-nat } n$ 
  using reals-Archimedean2 ..
  then have  $x \leq \text{of-nat } n$ 
  by simp
  then show ?thesis ..
qed
```

Archimedean fields have no infinitesimal elements.

```
lemma reals-Archimedean:
  fixes  $x :: 'a::\text{archimedean-field}$ 
  assumes  $0 < x$ 
  shows  $\exists n. \text{inverse } (\text{of-nat } (\text{Suc } n)) < x$ 
```

proof –

from $\langle 0 < x \rangle$ **have** $0 < \text{inverse } x$
by (rule positive-imp-inverse-positive)
obtain n **where** $\text{inverse } x < \text{of-nat } n$
using reals-Archimedean2 ..
then obtain m **where** $\text{inverse } x < \text{of-nat } (\text{Suc } m)$
using $\langle 0 < \text{inverse } x \rangle$ **by** (cases n) (simp-all del: of-nat-Suc)
then have $\text{inverse } (\text{of-nat } (\text{Suc } m)) < \text{inverse } (\text{inverse } x)$
using $\langle 0 < \text{inverse } x \rangle$ **by** (rule less-imp-inverse-less)
then have $\text{inverse } (\text{of-nat } (\text{Suc } m)) < x$
using $\langle 0 < x \rangle$ **by** (simp add: nonzero-inverse-inverse-eq)
then show ?thesis ..

qed

lemma ex-inverse-of-nat-less:

fixes $x :: 'a::\text{archimedean-field}$
assumes $0 < x$
shows $\exists n > 0. \text{inverse } (\text{of-nat } n) < x$
using reals-Archimedean [OF $\langle 0 < x \rangle$] **by** auto

lemma ex-less-of-nat-mult:

fixes $x :: 'a::\text{archimedean-field}$
assumes $0 < x$
shows $\exists n. y < \text{of-nat } n * x$

proof –

obtain n **where** $y / x < \text{of-nat } n$
using reals-Archimedean2 ..
with $\langle 0 < x \rangle$ **have** $y < \text{of-nat } n * x$
by (simp add: pos-divide-less-eq)
then show ?thesis ..

qed

96.2 Existence and uniqueness of floor function

lemma exists-least-lemma:

assumes $\neg P 0$ **and** $\exists n. P n$
shows $\exists n. \neg P n \wedge P (\text{Suc } n)$

proof –

from $\langle \exists n. P n \rangle$ **have** $P (\text{Least } P)$
by (rule LeastI-ex)
with $\langle \neg P 0 \rangle$ **obtain** n **where** $\text{Least } P = \text{Suc } n$
by (cases Least P) auto
then have $n < \text{Least } P$
by simp
then have $\neg P n$
by (rule not-less-Least)
then have $\neg P n \wedge P (\text{Suc } n)$
using $\langle P (\text{Least } P) \rangle \langle \text{Least } P = \text{Suc } n \rangle$ **by** simp
then show ?thesis ..

qed

lemma *floor-exists*:

fixes $x :: 'a::\text{archimedean-field}$

shows $\exists z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$

proof (*cases* $0 \leq x$)

case *True*

then have $\neg x < \text{of-nat } 0$

by *simp*

then have $\exists n. \neg x < \text{of-nat } n \wedge x < \text{of-nat } (\text{Suc } n)$

using *reals-Archimedean2* **by** (*rule exists-least-lemma*)

then obtain n **where** $\neg x < \text{of-nat } n \wedge x < \text{of-nat } (\text{Suc } n) ..$

then have $\text{of-int } (\text{int } n) \leq x \wedge x < \text{of-int } (\text{int } n + 1)$

by *simp*

then show *?thesis* ..

next

case *False*

then have $\neg -x \leq \text{of-nat } 0$

by *simp*

then have $\exists n. \neg -x \leq \text{of-nat } n \wedge -x \leq \text{of-nat } (\text{Suc } n)$

using *real-arch-simple* **by** (*rule exists-least-lemma*)

then obtain n **where** $\neg -x \leq \text{of-nat } n \wedge -x \leq \text{of-nat } (\text{Suc } n) ..$

then have $\text{of-int } (-\text{int } n - 1) \leq x \wedge x < \text{of-int } (-\text{int } n - 1 + 1)$

by *simp*

then show *?thesis* ..

qed

lemma *floor-exists1*: $\exists! z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$

for $x :: 'a::\text{archimedean-field}$

proof (*rule ex-ex1I*)

show $\exists z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$

by (*rule floor-exists*)

next

fix $y z$

assume $\text{of-int } y \leq x \wedge x < \text{of-int } (y + 1)$

and $\text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$

with *le-less-trans* [*of of-int y x of-int (z + 1)*]

le-less-trans [*of of-int z x of-int (y + 1)*] **show** $y = z$

by (*simp del: of-int-add*)

qed

96.3 Floor function

class *floor-ceiling* = *archimedean-field* +

fixes $\text{floor} :: 'a \Rightarrow \text{int } ([-])$

assumes *floor-correct*: $\text{of-int } [x] \leq x \wedge x < \text{of-int } ([x] + 1)$

lemma *floor-unique*: $\text{of-int } z \leq x \implies x < \text{of-int } z + 1 \implies [x] = z$

using *floor-correct* [*of x*] *floor-exists1* [*of x*] **by** *auto*

lemma *floor-eq-iff*: $\lfloor x \rfloor = a \iff \text{of-int } a \leq x \wedge x < \text{of-int } a + 1$
using *floor-correct floor-unique* **by** *auto*

lemma *of-int-floor-le* [*simp*]: $\text{of-int } \lfloor x \rfloor \leq x$
using *floor-correct ..*

lemma *le-floor-iff*: $z \leq \lfloor x \rfloor \iff \text{of-int } z \leq x$
proof

assume $z \leq \lfloor x \rfloor$
then have $(\text{of-int } z :: 'a) \leq \text{of-int } \lfloor x \rfloor$ **by** *simp*
also have $\text{of-int } \lfloor x \rfloor \leq x$ **by** *(rule of-int-floor-le)*
finally show $\text{of-int } z \leq x$.

next

assume $\text{of-int } z \leq x$
also have $x < \text{of-int } (\lfloor x \rfloor + 1)$ **using** *floor-correct ..*
finally show $z \leq \lfloor x \rfloor$ **by** *(simp del: of-int-add)*

qed

lemma *floor-less-iff*: $\lfloor x \rfloor < z \iff x < \text{of-int } z$
by *(simp add: not-le [symmetric] le-floor-iff)*

lemma *less-floor-iff*: $z < \lfloor x \rfloor \iff \text{of-int } z + 1 \leq x$
using *le-floor-iff [of z + 1 x]* **by** *auto*

lemma *floor-le-iff*: $\lfloor x \rfloor \leq z \iff x < \text{of-int } z + 1$
by *(simp add: not-less [symmetric] less-floor-iff)*

lemma *floor-split*[*linarith-split*]: $P \lfloor t \rfloor \iff (\forall i. \text{of-int } i \leq t \wedge t < \text{of-int } i + 1 \implies P i)$
by *(metis floor-correct floor-unique less-floor-iff not-le order-refl)*

lemma *floor-mono*:

assumes $x \leq y$
shows $\lfloor x \rfloor \leq \lfloor y \rfloor$

proof –

have $\text{of-int } \lfloor x \rfloor \leq x$ **by** *(rule of-int-floor-le)*
also note $\langle x \leq y \rangle$
finally show *?thesis* **by** *(simp add: le-floor-iff)*

qed

lemma *floor-less-cancel*: $\lfloor x \rfloor < \lfloor y \rfloor \implies x < y$
by *(auto simp add: not-le [symmetric] floor-mono)*

lemma *floor-of-int* [*simp*]: $\lfloor \text{of-int } z \rfloor = z$
by *(rule floor-unique) simp-all*

lemma *floor-of-nat* [*simp*]: $\lfloor \text{of-nat } n \rfloor = \text{int } n$
using *floor-of-int [of of-nat n]* **by** *simp*

lemma *le-floor-add*: $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$
by (*simp only: le-floor-iff of-int-add add-mono of-int-floor-le*)

Floor with numerals.

lemma *floor-zero* [*simp*]: $\lfloor 0 \rfloor = 0$
using *floor-of-int [of 0]* **by** *simp*

lemma *floor-one* [*simp*]: $\lfloor 1 \rfloor = 1$
using *floor-of-int [of 1]* **by** *simp*

lemma *floor-numeral* [*simp*]: $\lfloor \text{numeral } v \rfloor = \text{numeral } v$
using *floor-of-int [of numeral v]* **by** *simp*

lemma *floor-neg-numeral* [*simp*]: $\lfloor - \text{numeral } v \rfloor = - \text{numeral } v$
using *floor-of-int [of - numeral v]* **by** *simp*

lemma *zero-le-floor* [*simp*]: $0 \leq \lfloor x \rfloor \iff 0 \leq x$
by (*simp add: le-floor-iff*)

lemma *one-le-floor* [*simp*]: $1 \leq \lfloor x \rfloor \iff 1 \leq x$
by (*simp add: le-floor-iff*)

lemma *numeral-le-floor* [*simp*]: $\text{numeral } v \leq \lfloor x \rfloor \iff \text{numeral } v \leq x$
by (*simp add: le-floor-iff*)

lemma *neg-numeral-le-floor* [*simp*]: $- \text{numeral } v \leq \lfloor x \rfloor \iff - \text{numeral } v \leq x$
by (*simp add: le-floor-iff*)

lemma *zero-less-floor* [*simp*]: $0 < \lfloor x \rfloor \iff 1 \leq x$
by (*simp add: less-floor-iff*)

lemma *one-less-floor* [*simp*]: $1 < \lfloor x \rfloor \iff 2 \leq x$
by (*simp add: less-floor-iff*)

lemma *numeral-less-floor* [*simp*]: $\text{numeral } v < \lfloor x \rfloor \iff \text{numeral } v + 1 \leq x$
by (*simp add: less-floor-iff*)

lemma *neg-numeral-less-floor* [*simp*]: $- \text{numeral } v < \lfloor x \rfloor \iff - \text{numeral } v + 1 \leq x$
by (*simp add: less-floor-iff*)

lemma *floor-le-zero* [*simp*]: $\lfloor x \rfloor \leq 0 \iff x < 1$
by (*simp add: floor-le-iff*)

lemma *floor-le-one* [*simp*]: $\lfloor x \rfloor \leq 1 \iff x < 2$
by (*simp add: floor-le-iff*)

lemma *floor-le-numeral* [*simp*]: $\lfloor x \rfloor \leq \text{numeral } v \iff x < \text{numeral } v + 1$

by (*simp add: floor-le-iff*)

lemma *floor-le-neg-numeral* [*simp*]: $\lfloor x \rfloor \leq - \text{numeral } v \longleftrightarrow x < - \text{numeral } v + 1$

by (*simp add: floor-le-iff*)

lemma *floor-less-zero* [*simp*]: $\lfloor x \rfloor < 0 \longleftrightarrow x < 0$

by (*simp add: floor-less-iff*)

lemma *floor-less-one* [*simp*]: $\lfloor x \rfloor < 1 \longleftrightarrow x < 1$

by (*simp add: floor-less-iff*)

lemma *floor-less-numeral* [*simp*]: $\lfloor x \rfloor < \text{numeral } v \longleftrightarrow x < \text{numeral } v$

by (*simp add: floor-less-iff*)

lemma *floor-less-neg-numeral* [*simp*]: $\lfloor x \rfloor < - \text{numeral } v \longleftrightarrow x < - \text{numeral } v$

by (*simp add: floor-less-iff*)

lemma *le-mult-floor-Ints*:

assumes $0 \leq a$ *a* \in *Ints*

shows $\text{of-int } (\lfloor a \rfloor * \lfloor b \rfloor) \leq (\text{of-int } \lfloor a * b \rfloor :: 'a :: \text{linordered-idom})$

by (*metis Ints-cases assms floor-less-iff floor-of-int linorder-not-less mult-left-mono of-int-floor-le of-int-less-iff of-int-mult*)

Addition and subtraction of integers.

lemma *floor-add-int*: $\lfloor x \rfloor + z = \lfloor x + \text{of-int } z \rfloor$

using *floor-correct* [*of x*] **by** (*simp add: floor-unique[symmetric]*)

lemma *int-add-floor*: $z + \lfloor x \rfloor = \lfloor \text{of-int } z + x \rfloor$

using *floor-correct* [*of x*] **by** (*simp add: floor-unique[symmetric]*)

lemma *one-add-floor*: $\lfloor x \rfloor + 1 = \lfloor x + 1 \rfloor$

using *floor-add-int* [*of x 1*] **by** *simp*

lemma *floor-diff-of-int* [*simp*]: $\lfloor x - \text{of-int } z \rfloor = \lfloor x \rfloor - z$

using *floor-add-int* [*of x - z*] **by** (*simp add: algebra-simps*)

lemma *floor-uminus-of-int* [*simp*]: $\lfloor - (\text{of-int } z) \rfloor = - z$

by (*metis floor-diff-of-int* [*of 0*] *diff-0 floor-zero*)

lemma *floor-diff-numeral* [*simp*]: $\lfloor x - \text{numeral } v \rfloor = \lfloor x \rfloor - \text{numeral } v$

using *floor-diff-of-int* [*of x numeral v*] **by** *simp*

lemma *floor-diff-one* [*simp*]: $\lfloor x - 1 \rfloor = \lfloor x \rfloor - 1$

using *floor-diff-of-int* [*of x 1*] **by** *simp*

lemma *le-mult-floor*:

assumes $0 \leq a$ **and** $0 \leq b$

shows $\lfloor a \rfloor * \lfloor b \rfloor \leq \lfloor a * b \rfloor$

proof –

have $\text{of-int } \lfloor a \rfloor \leq a$ **and** $\text{of-int } \lfloor b \rfloor \leq b$
by (*auto intro: of-int-floor-le*)
then have $\text{of-int } (\lfloor a \rfloor * \lfloor b \rfloor) \leq a * b$
using *assms* **by** (*auto intro!: mult-mono*)
also have $a * b < \text{of-int } (\lfloor a * b \rfloor + 1)$
using *floor-correct*[*of a * b*] **by** *auto*
finally show *?thesis*
unfolding *of-int-less-iff* **by** *simp*

qed

lemma *floor-divide-of-int-eq*: $\lfloor \text{of-int } k / \text{of-int } l \rfloor = k \text{ div } l$

for $k \ l :: \text{int}$

proof (*cases l = 0*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

have $*$: $\lfloor \text{of-int } (k \text{ mod } l) / \text{of-int } l \rfloor :: 'a = 0$

proof (*cases l > 0*)

case *True*

then show *?thesis*

by (*auto intro: floor-unique*)

next

case *False*

obtain r **where** $r = - l$

by *blast*

then have $l: l = - r$

by *simp*

with $\langle l \neq 0 \rangle$ *False* **have** $r > 0$

by *simp*

with l **show** *?thesis*

using *pos-mod-bound* [*of r*]

by (*auto simp add: zmod-zminus2-eq-if less-le field-simps intro: floor-unique*)

qed

have $(\text{of-int } k :: 'a) = \text{of-int } (k \text{ div } l * l + k \text{ mod } l)$

by *simp*

also have $\dots = (\text{of-int } (k \text{ div } l) + \text{of-int } (k \text{ mod } l) / \text{of-int } l) * \text{of-int } l$

using *False* **by** (*simp only: of-int-add*) (*simp add: field-simps*)

finally have $(\text{of-int } k / \text{of-int } l :: 'a) = \dots / \text{of-int } l$

by *simp*

then have $(\text{of-int } k / \text{of-int } l :: 'a) = \text{of-int } (k \text{ div } l) + \text{of-int } (k \text{ mod } l) / \text{of-int } l$

using *False* **by** (*simp only:*) (*simp add: field-simps*)

then have $\lfloor \text{of-int } k / \text{of-int } l \rfloor :: 'a = \lfloor \text{of-int } (k \text{ div } l) + \text{of-int } (k \text{ mod } l) / \text{of-int } l \rfloor :: 'a$

by *simp*

then have $\lfloor \text{of-int } k / \text{of-int } l \rfloor :: 'a = \lfloor \text{of-int } (k \text{ mod } l) / \text{of-int } l + \text{of-int } (k \text{ div } l) \rfloor :: 'a$

by (*simp add: ac-simps*)

then have $\lfloor \text{of-int } k / \text{of-int } l :: 'a \rfloor = \lfloor \text{of-int } (k \text{ mod } l) / \text{of-int } l :: 'a \rfloor + k \text{ div } l$
by (*simp add: floor-add-int*)
with * show ?thesis
by *simp*
qed

lemma *floor-divide-of-nat-eq*: $\lfloor \text{of-nat } m / \text{of-nat } n \rfloor = \text{of-nat } (m \text{ div } n)$
for $m \ n :: \text{nat}$
by (*metis floor-divide-of-int-eq of-int-of-nat-eq unique-euclidean-semiring-with-nat-class.of-nat-div*)

lemma *floor-divide-lower*:
fixes $q :: 'a::\text{floor-ceiling}$
shows $q > 0 \implies \text{of-int } \lfloor p / q \rfloor * q \leq p$
using *of-int-floor-le pos-le-divide-eq* **by** *blast*

lemma *floor-divide-upper*:
fixes $q :: 'a::\text{floor-ceiling}$
shows $q > 0 \implies p < (\text{of-int } \lfloor p / q \rfloor + 1) * q$
by (*meson floor-eq-iff pos-divide-less-eq*)

96.4 Ceiling function

definition *ceiling* :: $'a::\text{floor-ceiling} \Rightarrow \text{int}$ ($\lceil - \rceil$)
where $\lceil x \rceil = - \lfloor - x \rfloor$

lemma *ceiling-correct*: $\text{of-int } \lceil x \rceil - 1 < x \wedge x \leq \text{of-int } \lceil x \rceil$
unfolding *ceiling-def* **using** *floor-correct* [*of - x*]
by (*simp add: le-minus-iff*)

lemma *ceiling-unique*: $\text{of-int } z - 1 < x \implies x \leq \text{of-int } z \implies \lceil x \rceil = z$
unfolding *ceiling-def* **using** *floor-unique* [*of - z - x*] **by** *simp*

lemma *ceiling-eq-iff*: $\lceil x \rceil = a \iff \text{of-int } a - 1 < x \wedge x \leq \text{of-int } a$
using *ceiling-correct ceiling-unique* **by** *auto*

lemma *le-of-int-ceiling* [*simp*]: $x \leq \text{of-int } \lceil x \rceil$
using *ceiling-correct* ..

lemma *ceiling-le-iff*: $\lceil x \rceil \leq z \iff x \leq \text{of-int } z$
unfolding *ceiling-def* **using** *le-floor-iff* [*of - z - x*] **by** *auto*

lemma *less-ceiling-iff*: $z < \lceil x \rceil \iff \text{of-int } z < x$
by (*simp add: not-le [symmetric] ceiling-le-iff*)

lemma *ceiling-less-iff*: $\lceil x \rceil < z \iff x \leq \text{of-int } z - 1$
using *ceiling-le-iff* [*of x z - 1*] **by** *simp*

lemma *le-ceiling-iff*: $z \leq \lceil x \rceil \iff \text{of-int } z - 1 < x$
by (*simp add: not-less [symmetric] ceiling-less-iff*)

lemma *ceiling-mono*: $x \geq y \implies \lceil x \rceil \geq \lceil y \rceil$
unfolding *ceiling-def* **by** (*simp add: floor-mono*)

lemma *ceiling-less-cancel*: $\lceil x \rceil < \lceil y \rceil \implies x < y$
by (*auto simp add: not-le [symmetric] ceiling-mono*)

lemma *ceiling-of-int [simp]*: $\lceil \text{of-int } z \rceil = z$
by (*rule ceiling-unique*) *simp-all*

lemma *ceiling-of-nat [simp]*: $\lceil \text{of-nat } n \rceil = \text{int } n$
using *ceiling-of-int [of of-nat n]* **by** *simp*

lemma *ceiling-add-le*: $\lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil$
by (*simp only: ceiling-le-iff of-int-add add-mono le-of-int-ceiling*)

lemma *mult-ceiling-le*:
assumes $0 \leq a$ **and** $0 \leq b$
shows $\lceil a * b \rceil \leq \lceil a \rceil * \lceil b \rceil$
by (*metis assms ceiling-le-iff ceiling-mono le-of-int-ceiling mult-mono of-int-mult*)

lemma *mult-ceiling-le-Ints*:
assumes $0 \leq a$ $a \in \text{Ints}$
shows $(\text{of-int } \lceil a * b \rceil :: 'a :: \text{linordered-idom}) \leq \text{of-int}(\lceil a \rceil * \lceil b \rceil)$
by (*metis Ints-cases assms ceiling-le-iff ceiling-of-int le-of-int-ceiling mult-left-mono of-int-le-iff of-int-mult*)

lemma *finite-int-segment*:
fixes $a :: 'a :: \text{floor-ceiling}$
shows *finite* $\{x \in \mathbb{Z}. a \leq x \wedge x \leq b\}$
proof –
have *finite* $\{\text{ceiling } a.. \text{floor } b\}$
by *simp*
moreover **have** $\{x \in \mathbb{Z}. a \leq x \wedge x \leq b\} = \text{of-int } \{ \text{ceiling } a.. \text{floor } b \}$
by (*auto simp: le-floor-iff ceiling-le-iff elim!: Ints-cases*)
ultimately show *?thesis*
by *simp*
qed

corollary *finite-abs-int-segment*:
fixes $a :: 'a :: \text{floor-ceiling}$
shows *finite* $\{k \in \mathbb{Z}. |k| \leq a\}$
using *finite-int-segment [of -a a]* **by** (*auto simp add: abs-le-iff conj-commute minus-le-iff*)

96.4.1 Ceiling with numerals.

lemma *ceiling-zero [simp]*: $\lceil 0 \rceil = 0$
using *ceiling-of-int [of 0]* **by** *simp*

lemma *ceiling-one* [simp]: $\lceil 1 \rceil = 1$
using *ceiling-of-int [of 1]* **by** *simp*

lemma *ceiling-numeral* [simp]: $\lceil \text{numeral } v \rceil = \text{numeral } v$
using *ceiling-of-int [of numeral v]* **by** *simp*

lemma *ceiling-neg-numeral* [simp]: $\lceil - \text{numeral } v \rceil = - \text{numeral } v$
using *ceiling-of-int [of - numeral v]* **by** *simp*

lemma *ceiling-le-zero* [simp]: $\lceil x \rceil \leq 0 \iff x \leq 0$
by (*simp add: ceiling-le-iff*)

lemma *ceiling-le-one* [simp]: $\lceil x \rceil \leq 1 \iff x \leq 1$
by (*simp add: ceiling-le-iff*)

lemma *ceiling-le-numeral* [simp]: $\lceil x \rceil \leq \text{numeral } v \iff x \leq \text{numeral } v$
by (*simp add: ceiling-le-iff*)

lemma *ceiling-le-neg-numeral* [simp]: $\lceil x \rceil \leq - \text{numeral } v \iff x \leq - \text{numeral } v$
by (*simp add: ceiling-le-iff*)

lemma *ceiling-less-zero* [simp]: $\lceil x \rceil < 0 \iff x \leq -1$
by (*simp add: ceiling-less-iff*)

lemma *ceiling-less-one* [simp]: $\lceil x \rceil < 1 \iff x \leq 0$
by (*simp add: ceiling-less-iff*)

lemma *ceiling-less-numeral* [simp]: $\lceil x \rceil < \text{numeral } v \iff x \leq \text{numeral } v - 1$
by (*simp add: ceiling-less-iff*)

lemma *ceiling-less-neg-numeral* [simp]: $\lceil x \rceil < - \text{numeral } v \iff x \leq - \text{numeral } v - 1$
by (*simp add: ceiling-less-iff*)

lemma *zero-le-ceiling* [simp]: $0 \leq \lceil x \rceil \iff -1 < x$
by (*simp add: le-ceiling-iff*)

lemma *one-le-ceiling* [simp]: $1 \leq \lceil x \rceil \iff 0 < x$
by (*simp add: le-ceiling-iff*)

lemma *numeral-le-ceiling* [simp]: $\text{numeral } v \leq \lceil x \rceil \iff \text{numeral } v - 1 < x$
by (*simp add: le-ceiling-iff*)

lemma *neg-numeral-le-ceiling* [simp]: $- \text{numeral } v \leq \lceil x \rceil \iff - \text{numeral } v - 1 < x$
by (*simp add: le-ceiling-iff*)

lemma *zero-less-ceiling* [simp]: $0 < \lceil x \rceil \iff 0 < x$

by (simp add: less-ceiling-iff)

lemma one-less-ceiling [simp]: $1 < \lceil x \rceil \iff 1 < x$
 by (simp add: less-ceiling-iff)

lemma numeral-less-ceiling [simp]: numeral $v < \lceil x \rceil \iff$ numeral $v < x$
 by (simp add: less-ceiling-iff)

lemma neg-numeral-less-ceiling [simp]: $-$ numeral $v < \lceil x \rceil \iff -$ numeral $v < x$
 by (simp add: less-ceiling-iff)

lemma ceiling-altdef: $\lceil x \rceil =$ (if $x =$ of-int $\lfloor x \rfloor$ then $\lfloor x \rfloor$ else $\lfloor x \rfloor + 1$)
 by (intro ceiling-unique; simp, linarith?)

lemma floor-le-ceiling [simp]: $\lfloor x \rfloor \leq \lceil x \rceil$
 by (simp add: ceiling-altdef)

96.4.2 Addition and subtraction of integers.

lemma ceiling-add-of-int [simp]: $\lceil x +$ of-int $z \rceil = \lceil x \rceil + z$
 using ceiling-correct [of x] by (simp add: ceiling-def)

lemma ceiling-add-numeral [simp]: $\lceil x +$ numeral $v \rceil = \lceil x \rceil +$ numeral v
 using ceiling-add-of-int [of x numeral v] by simp

lemma ceiling-add-one [simp]: $\lceil x + 1 \rceil = \lceil x \rceil + 1$
 using ceiling-add-of-int [of x 1] by simp

lemma ceiling-diff-of-int [simp]: $\lceil x -$ of-int $z \rceil = \lceil x \rceil - z$
 using ceiling-add-of-int [of $x - z$] by (simp add: algebra-simps)

lemma ceiling-diff-numeral [simp]: $\lceil x -$ numeral $v \rceil = \lceil x \rceil -$ numeral v
 using ceiling-diff-of-int [of x numeral v] by simp

lemma ceiling-diff-one [simp]: $\lceil x - 1 \rceil = \lceil x \rceil - 1$
 using ceiling-diff-of-int [of x 1] by simp

lemma ceiling-split[linarith-split]: $P \lceil t \rceil \iff (\forall i. \text{of-int } i - 1 < t \wedge t \leq \text{of-int } i \implies P i)$
 by (auto simp add: ceiling-unique ceiling-correct)

lemma ceiling-diff-floor-le-1: $\lceil x \rceil - \lfloor x \rfloor \leq 1$

proof –

have of-int $\lceil x \rceil - 1 < x$
 using ceiling-correct[of x] by simp

also have $x < \text{of-int } \lfloor x \rfloor + 1$
 using floor-correct[of x] by simp-all

finally have of-int $(\lceil x \rceil - \lfloor x \rfloor) < (\text{of-int } 2::'a)$
 by simp

then show *?thesis*
unfolding *of-int-less-iff* **by** *simp*
qed

lemma *nat-approx-posE*:
fixes $e :: 'a :: \{\text{archimedean-field, floor-ceiling}\}$
assumes $0 < e$
obtains $n :: \text{nat}$ **where** $1 / \text{of-nat}(\text{Suc } n) < e$
proof
have $(1 :: 'a) / \text{of-nat}(\text{Suc}(\text{nat } \lceil 1/e \rceil)) < 1 / \text{of-int}(\lceil 1/e \rceil)$
proof (*rule divide-strict-left-mono*)
show $(\text{of-int } \lceil 1/e \rceil :: 'a) < \text{of-nat}(\text{Suc}(\text{nat } \lceil 1/e \rceil))$
using *assms* **by** (*simp add: field-simps*)
show $(0 :: 'a) < \text{of-nat}(\text{Suc}(\text{nat } \lceil 1/e \rceil)) * \text{of-int } \lceil 1/e \rceil$
using *assms* **by** (*auto simp: zero-less-mult-iff pos-add-strict*)
qed *auto*
also have $1 / \text{of-int}(\lceil 1/e \rceil) \leq 1 / (1/e)$
by (*rule divide-left-mono*) (*auto simp: ‹0 < e› ceiling-correct*)
also have $\dots = e$ **by** *simp*
finally show $1 / \text{of-nat}(\text{Suc}(\text{nat } \lceil 1/e \rceil)) < e$
by *metis*
qed

lemma *ceiling-divide-upper*:
fixes $q :: 'a :: \text{floor-ceiling}$
shows $q > 0 \implies p \leq \text{of-int}(\text{ceiling}(p / q)) * q$
by (*meson divide-le-eq le-of-int-ceiling*)

lemma *ceiling-divide-lower*:
fixes $q :: 'a :: \text{floor-ceiling}$
shows $q > 0 \implies (\text{of-int } \lceil p / q \rceil - 1) * q < p$
by (*meson ceiling-eq-iff pos-less-divide-eq*)

96.5 Negation

lemma *floor-minus*: $\lfloor -x \rfloor = -\lceil x \rceil$
unfolding *ceiling-def* **by** *simp*

lemma *ceiling-minus*: $\lceil -x \rceil = -\lfloor x \rfloor$
unfolding *ceiling-def* **by** *simp*

96.6 Natural numbers

lemma *of-nat-floor*: $r \geq 0 \implies \text{of-nat}(\text{nat } \lfloor r \rfloor) \leq r$
by *simp*

lemma *of-nat-ceiling*: $\text{of-nat}(\text{nat } \lceil r \rceil) \geq r$
by (*cases* $r \geq 0$) *auto*

96.7 Frac Function

definition $\text{frac} :: 'a \Rightarrow 'a::\text{floor-ceiling}$
where $\text{frac } x \equiv x - \text{of-int } \lfloor x \rfloor$

lemma frac-lt-1 : $\text{frac } x < 1$
by ($\text{simp add: frac-def} \text{ linarith}$)

lemma frac-eq-0-iff [simp]: $\text{frac } x = 0 \longleftrightarrow x \in \mathbf{Z}$
by ($\text{simp add: frac-def} \text{ (metis Ints-cases Ints-of-int floor-of-int)}$)

lemma frac-ge-0 [simp]: $\text{frac } x \geq 0$
unfolding frac-def **by** linarith

lemma frac-gt-0-iff [simp]: $\text{frac } x > 0 \longleftrightarrow x \notin \mathbf{Z}$
by ($\text{metis frac-eq-0-iff frac-ge-0 le-less less-irrefl}$)

lemma frac-of-int [simp]: $\text{frac} (\text{of-int } z) = 0$
by ($\text{simp add: frac-def}$)

lemma frac-frac [simp]: $\text{frac} (\text{frac } x) = \text{frac } x$
by ($\text{simp add: frac-def}$)

lemma floor-add : $\lfloor x + y \rfloor = (\text{if } \text{frac } x + \text{frac } y < 1 \text{ then } \lfloor x \rfloor + \lfloor y \rfloor \text{ else } (\lfloor x \rfloor + \lfloor y \rfloor) + 1)$

proof –

have $x + y < 1 + (\text{of-int } \lfloor x \rfloor + \text{of-int } \lfloor y \rfloor) \implies \lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor$
by ($\text{metis add.commute floor-unique le-floor-add le-floor-iff of-int-add}$)

moreover

have $\neg x + y < 1 + (\text{of-int } \lfloor x \rfloor + \text{of-int } \lfloor y \rfloor) \implies \lfloor x + y \rfloor = 1 + (\lfloor x \rfloor + \lfloor y \rfloor)$

apply ($\text{simp add: floor-eq-iff}$)

apply ($\text{auto simp add: algebra-simps}$)

apply linarith

done

ultimately show $?thesis$ **by** ($\text{auto simp add: frac-def algebra-simps}$)

qed

lemma floor-add2 [simp]: $x \in \mathbf{Z} \vee y \in \mathbf{Z} \implies \lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor$
by ($\text{metis add.commute add.left-neutral frac-lt-1 floor-add frac-eq-0-iff}$)

lemma frac-add :

$\text{frac} (x + y) = (\text{if } \text{frac } x + \text{frac } y < 1 \text{ then } \text{frac } x + \text{frac } y \text{ else } (\text{frac } x + \text{frac } y) - 1)$

by ($\text{simp add: frac-def floor-add}$)

lemma frac-unique-iff : $\text{frac } x = a \longleftrightarrow x - a \in \mathbf{Z} \wedge 0 \leq a \wedge a < 1$

for $x :: 'a::\text{floor-ceiling}$

apply ($\text{auto simp: Ints-def frac-def algebra-simps floor-unique}$)

apply linarith+

done

lemma *frac-eq*: $\text{frac } x = x \longleftrightarrow 0 \leq x \wedge x < 1$
by (*simp add: frac-unique-iff*)

lemma *frac-neg*: $\text{frac } (-x) = (\text{if } x \in \mathbb{Z} \text{ then } 0 \text{ else } 1 - \text{frac } x)$
for $x :: 'a::\text{floor-ceiling}$
apply (*auto simp add: frac-unique-iff*)
apply (*simp add: frac-def*)
apply (*meson frac-lt-1 less-iff-diff-less-0 not-le not-less-iff-gr-or-eq*)
done

lemma *frac-in-Ints-iff* [*simp*]: $\text{frac } x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
proof *safe*
assume $\text{frac } x \in \mathbb{Z}$
hence $\text{of-int } \lfloor x \rfloor + \text{frac } x \in \mathbb{Z}$ **by** *auto*
also have $\text{of-int } \lfloor x \rfloor + \text{frac } x = x$ **by** (*simp add: frac-def*)
finally show $x \in \mathbb{Z}$.
qed (*auto simp: frac-def*)

lemma *frac-1-eq*: $\text{frac } (x+1) = \text{frac } x$
by (*simp add: frac-def*)

96.8 Rounding to the nearest integer

definition *round* :: $'a::\text{floor-ceiling} \Rightarrow \text{int}$
where $\text{round } x = \lfloor x + 1/2 \rfloor$

lemma *of-int-round-ge*: $\text{of-int } (\text{round } x) \geq x - 1/2$
and *of-int-round-le*: $\text{of-int } (\text{round } x) \leq x + 1/2$
and *of-int-round-abs-le*: $|\text{of-int } (\text{round } x) - x| \leq 1/2$
and *of-int-round-gt*: $\text{of-int } (\text{round } x) > x - 1/2$

proof –
from *floor-correct*[$\text{of } x + 1/2$] **have** $x + 1/2 < \text{of-int } (\text{round } x) + 1$
by (*simp add: round-def*)
from *add-strict-right-mono*[*OF this, of -1*] **show** $A: \text{of-int } (\text{round } x) > x - 1/2$
by *simp*
then show $\text{of-int } (\text{round } x) \geq x - 1/2$
by *simp*
from *floor-correct*[$\text{of } x + 1/2$] **show** $\text{of-int } (\text{round } x) \leq x + 1/2$
by (*simp add: round-def*)
with A **show** $|\text{of-int } (\text{round } x) - x| \leq 1/2$
by *linarith*
qed

lemma *round-of-int* [*simp*]: $\text{round } (\text{of-int } n) = n$
unfolding *round-def* **by** (*subst floor-eq-iff*) *force*

lemma *round-0* [*simp*]: $\text{round } 0 = 0$
using *round-of-int*[*of 0*] **by** *simp*

lemma *round-1* [*simp*]: $\text{round } 1 = 1$
using *round-of-int*[*of 1*] **by** *simp*

lemma *round-numeral* [*simp*]: $\text{round } (\text{numeral } n) = \text{numeral } n$
using *round-of-int*[*of numeral n*] **by** *simp*

lemma *round-neg-numeral* [*simp*]: $\text{round } (-\text{numeral } n) = -\text{numeral } n$
using *round-of-int*[*of -numeral n*] **by** *simp*

lemma *round-of-nat* [*simp*]: $\text{round } (\text{of-nat } n) = \text{of-nat } n$
using *round-of-int*[*of int n*] **by** *simp*

lemma *round-mono*: $x \leq y \implies \text{round } x \leq \text{round } y$
unfolding *round-def* **by** (*intro floor-mono*) *simp*

lemma *round-unique*: $\text{of-int } y > x - 1/2 \implies \text{of-int } y \leq x + 1/2 \implies \text{round } x = y$
unfolding *round-def*
proof (*rule floor-unique*)
assume $x - 1 / 2 < \text{of-int } y$
from *add-strict-left-mono*[*OF this, of 1*] **show** $x + 1 / 2 < \text{of-int } y + 1$
by *simp*
qed

lemma *round-unique'*: $|x - \text{of-int } n| < 1/2 \implies \text{round } x = n$
by (*subst (asm) abs-less-iff, rule round-unique*) (*simp-all add: field-simps*)

lemma *round-altdef*: $\text{round } x = (\text{if } \text{frac } x \geq 1/2 \text{ then } \lceil x \rceil \text{ else } \lfloor x \rfloor)$
by (*cases frac x ≥ 1/2*)
(rule round-unique, ((simp add: frac-def field-simps ceiling-altdef; linarith)+)[2])+

lemma *floor-le-round*: $\lfloor x \rfloor \leq \text{round } x$
unfolding *round-def* **by** (*intro floor-mono*) *simp*

lemma *ceiling-ge-round*: $\lceil x \rceil \geq \text{round } x$
unfolding *round-altdef* **by** *simp*

lemma *round-diff-minimal*: $|z - \text{of-int } (\text{round } z)| \leq |z - \text{of-int } m|$
for $z :: 'a::\text{floor-ceiling}$
proof (*cases of-int m ≥ z*)
case *True*
then have $|z - \text{of-int } (\text{round } z)| \leq |\text{of-int } \lceil z \rceil - z|$
unfolding *round-altdef* **by** (*simp add: field-simps ceiling-altdef frac-def*) *linarith*
also have $\text{of-int } \lceil z \rceil - z \geq 0$
by *linarith*
with *True* **have** $|\text{of-int } \lceil z \rceil - z| \leq |z - \text{of-int } m|$
by (*simp add: ceiling-le-iff*)
finally show *?thesis* .


```

next
  case False
  then have  $|z - \text{of-int } (\text{round } z)| \leq |\text{of-int } \lfloor z \rfloor - z|$ 
    unfolding round-altdef by (simp add: field-simps ceiling-altdef frac-def) linarith
    also have  $z - \text{of-int } \lfloor z \rfloor \geq 0$ 
      by linarith
    with False have  $|\text{of-int } \lfloor z \rfloor - z| \leq |z - \text{of-int } m|$ 
      by (simp add: le-floor-iff)
    finally show ?thesis .
qed

end

```

97 Rational numbers

```

theory Rat
  imports Archimedean-Field
begin

```

97.1 Rational numbers as quotient

97.1.1 Construction of the type of rational numbers

```

definition ratrel :: (int × int) ⇒ (int × int) ⇒ bool
  where ratrel = (λx y. snd x ≠ 0 ∧ snd y ≠ 0 ∧ fst x * snd y = fst y * snd x)

```

```

lemma ratrel-iff [simp]: ratrel x y ↔ snd x ≠ 0 ∧ snd y ≠ 0 ∧ fst x * snd y =
fst y * snd x
  by (simp add: ratrel-def)

```

```

lemma exists-ratrel-refl: ∃ x. ratrel x x
  by (auto intro!: one-neq-zero)

```

```

lemma symp-ratrel: symp ratrel
  by (simp add: ratrel-def symp-def)

```

```

lemma transp-ratrel: transp ratrel
proof (rule transpI, unfold split-paired-all)
  fix a b a' b' a'' b'' :: int
  assume *: ratrel (a, b) (a', b')
  assume **: ratrel (a', b') (a'', b'')
  have b' * (a * b'') = b'' * (a * b') by simp
  also from * have a * b' = a' * b by auto
  also have b'' * (a' * b) = b * (a' * b'') by simp
  also from ** have a' * b'' = a'' * b' by auto
  also have b * (a'' * b') = b' * (a'' * b) by simp
  finally have b' * (a * b'') = b' * (a'' * b) .
  moreover from ** have b' ≠ 0 by auto
  ultimately have a * b'' = a'' * b by simp

```

with * ** show *ratrel* (*a*, *b*) (*a''*, *b''*) **by** *auto*
qed

lemma *part-equivp-ratrel*: *part-equivp ratrel*
by (*rule part-equivpI [OF exists-ratrel-refl symp-ratrel transp-ratrel]*)

quotient-type *rat* = *int* × *int* / *partial*: *ratrel*
morphisms *Rep-Rat Abs-Rat*
by (*rule part-equivp-ratrel*)

lemma *Domainp-cr-rat* [*transfer-domain-rule*]: *Domainp pcr-rat* = ($\lambda x. \text{snd } x \neq 0$)
by (*simp add: rat.domain-eq*)

97.1.2 Representation and basic operations

lift-definition *Fract* :: *int* ⇒ *int* ⇒ *rat*
is $\lambda a b. \text{if } b = 0 \text{ then } (0, 1) \text{ else } (a, b)$
by *simp*

lemma *eq-rat*:
 $\bigwedge a b c d. b \neq 0 \implies d \neq 0 \implies \text{Fract } a b = \text{Fract } c d \longleftrightarrow a * d = c * b$
 $\bigwedge a. \text{Fract } a 0 = \text{Fract } 0 1$
 $\bigwedge a c. \text{Fract } 0 a = \text{Fract } 0 c$
by (*transfer, simp*)⁺

lemma *Rat-cases* [*case-names Fract, cases type: rat*]:
assumes that: $\bigwedge a b. q = \text{Fract } a b \implies b > 0 \implies \text{coprime } a b \implies C$
shows *C*

proof –

obtain *a b* :: *int* **where** *q*: *q* = *Fract a b* **and** *b*: *b* ≠ 0

by *transfer simp*

let *?a* = *a div gcd a b*

let *?b* = *b div gcd a b*

from *b* **have** *?b * gcd a b = b*

by *simp*

with *b* **have** *?b* ≠ 0

by *fastforce*

with *q b* **have** *q2*: *q* = *Fract ?a ?b*

by (*simp add: eq-rat dvd-div-mult mult.commute [of a]*)

from *b* **have** *coprime*: *coprime ?a ?b*

by (*auto intro: div-gcd-coprime*)

show *C*

proof (*cases b > 0*)

case *True*

then have *?b* > 0

by (*simp add: nonneg1-imp-zdiv-pos-iff*)

from *q2 this coprime* **show** *C* **by** (*rule that*)

next

```

case False
have  $q = \text{Fract } (- ?a) (- ?b)$ 
  unfolding  $q2$  by transfer simp
moreover from False b have  $- ?b > 0$ 
  by (simp add: pos-imp-zdiv-neg-iff)
moreover from coprime have  $\text{coprime } (- ?a) (- ?b)$ 
  by simp
ultimately show  $C$ 
  by (rule that)
qed
qed

lemma Rat-induct [case-names Fract, induct type: rat]:
  assumes  $\bigwedge a b. b > 0 \implies \text{coprime } a b \implies P (\text{Fract } a b)$ 
  shows  $P q$ 
  using assms by (cases q) simp

instantiation rat :: field
begin

lift-definition zero-rat :: rat is  $(0, 1)$ 
  by simp

lift-definition one-rat :: rat is  $(1, 1)$ 
  by simp

lemma Zero-rat-def:  $0 = \text{Fract } 0 1$ 
  by transfer simp

lemma One-rat-def:  $1 = \text{Fract } 1 1$ 
  by transfer simp

lift-definition plus-rat :: rat  $\Rightarrow$  rat  $\Rightarrow$  rat
  is  $\lambda x y. (\text{fst } x * \text{snd } y + \text{fst } y * \text{snd } x, \text{snd } x * \text{snd } y)$ 
  by (auto simp: distrib-right) (simp add: ac-simps)

lemma add-rat [simp]:
  assumes  $b \neq 0$  and  $d \neq 0$ 
  shows  $\text{Fract } a b + \text{Fract } c d = \text{Fract } (a * d + c * b) (b * d)$ 
  using assms by transfer simp

lift-definition uminus-rat :: rat  $\Rightarrow$  rat is  $\lambda x. (- \text{fst } x, \text{snd } x)$ 
  by simp

lemma minus-rat [simp]:  $-\text{Fract } a b = \text{Fract } (- a) b$ 
  by transfer simp

lemma minus-rat-cancel [simp]:  $\text{Fract } (- a) (- b) = \text{Fract } a b$ 
  by (cases b = 0) (simp-all add: eq-rat)

```

definition *diff-rat-def*: $q - r = q + - r$ for $q r :: rat$

lemma *diff-rat* [*simp*]:

$b \neq 0 \implies d \neq 0 \implies Fract a b - Fract c d = Fract (a * d - c * b) (b * d)$
by (*simp add: diff-rat-def*)

lift-definition *times-rat* :: $rat \Rightarrow rat \Rightarrow rat$

is $\lambda x y. (fst x * fst y, snd x * snd y)$

by (*simp add: ac-simps*)

lemma *mult-rat* [*simp*]: $Fract a b * Fract c d = Fract (a * c) (b * d)$

by *transfer simp*

lemma *mult-rat-cancel*: $c \neq 0 \implies Fract (c * a) (c * b) = Fract a b$

by *transfer simp*

lift-definition *inverse-rat* :: $rat \Rightarrow rat$

is $\lambda x. if\ fst\ x = 0\ then\ (0, 1)\ else\ (snd\ x, fst\ x)$

by (*auto simp add: mult.commute*)

lemma *inverse-rat* [*simp*]: $inverse (Fract a b) = Fract b a$

by *transfer simp*

definition *divide-rat-def*: $q\ div\ r = q * inverse\ r$ for $q r :: rat$

lemma *divide-rat* [*simp*]: $Fract a b\ div\ Fract c d = Fract (a * d) (b * c)$

by (*simp add: divide-rat-def*)

instance

proof

fix $q r s :: rat$

show $(q * r) * s = q * (r * s)$

by *transfer simp*

show $q * r = r * q$

by *transfer simp*

show $1 * q = q$

by *transfer simp*

show $(q + r) + s = q + (r + s)$

by *transfer (simp add: algebra-simps)*

show $q + r = r + q$

by *transfer simp*

show $0 + q = q$

by *transfer simp*

show $- q + q = 0$

by *transfer simp*

show $q - r = q + - r$

by (*fact diff-rat-def*)

show $(q + r) * s = q * s + r * s$

```

    by transfer (simp add: algebra-simps)
  show (0::rat) ≠ 1
    by transfer simp
  show inverse q * q = 1 if q ≠ 0
    using that by transfer simp
  show q div r = q * inverse r
    by (fact divide-rat-def)
  show inverse 0 = (0::rat)
    by transfer simp
qed
end

```

```

lemma div-add-self1-no-field [simp]:
  assumes NO-MATCH (x :: 'b :: field) b (b :: 'a :: euclidean-semiring-cancel) ≠
  0
  shows (b + a) div b = a div b + 1
  using assms(2) by (fact div-add-self1)

```

```

lemma div-add-self2-no-field [simp]:
  assumes NO-MATCH (x :: 'b :: field) b (b :: 'a :: euclidean-semiring-cancel) ≠
  0
  shows (a + b) div b = a div b + 1
  using assms(2) by (fact div-add-self2)

```

```

lemma of-nat-rat: of-nat k = Fract (of-nat k) 1
  by (induct k) (simp-all add: Zero-rat-def One-rat-def)

```

```

lemma of-int-rat: of-int k = Fract k 1
  by (cases k rule: int-diff-cases) (simp add: of-nat-rat)

```

```

lemma Fract-of-nat-eq: Fract (of-nat k) 1 = of-nat k
  by (rule of-nat-rat [symmetric])

```

```

lemma Fract-of-int-eq: Fract k 1 = of-int k
  by (rule of-int-rat [symmetric])

```

```

lemma rat-number-collapse:
  Fract 0 k = 0
  Fract 1 1 = 1
  Fract (numeral w) 1 = numeral w
  Fract (- numeral w) 1 = - numeral w
  Fract (- 1) 1 = - 1
  Fract k 0 = 0
  using Fract-of-int-eq [of numeral w]
  and Fract-of-int-eq [of - numeral w]
  by (simp-all add: Zero-rat-def One-rat-def eq-rat)

```

lemma *rat-number-expand*:

$0 = \text{Fract } 0 \ 1$
 $1 = \text{Fract } 1 \ 1$
 $\text{numeral } k = \text{Fract } (\text{numeral } k) \ 1$
 $- 1 = \text{Fract } (- 1) \ 1$
 $- \text{numeral } k = \text{Fract } (- \text{numeral } k) \ 1$
by (*simp-all add: rat-number-collapse*)

lemma *Rat-cases-nonzero* [*case-names Fract 0*]:

assumes *Fract*: $\bigwedge a \ b. \ q = \text{Fract } a \ b \implies b > 0 \implies a \neq 0 \implies \text{coprime } a \ b \implies C$
and $0: q = 0 \implies C$
shows C
proof (*cases q = 0*)
case *True*
then show C **using** 0 **by** *auto*
next
case *False*
then obtain $a \ b$ **where** $q = \text{Fract } a \ b \ b > 0 \ \text{coprime } a \ b$
by (*cases q*) *auto*
with *False* **have** $0 \neq \text{Fract } a \ b$
by *simp*
with $\langle b > 0 \rangle$ **have** $a \neq 0$
by (*simp add: Zero-rat-def eq-rat*)
with *Fract ** **show** C **by** *blast*
qed

97.1.3 Function *normalize*

lemma *Fract-coprime*: $\text{Fract } (a \ \text{div} \ \text{gcd } a \ b) \ (b \ \text{div} \ \text{gcd } a \ b) = \text{Fract } a \ b$

proof (*cases b = 0*)
case *True*
then show *?thesis*
by (*simp add: eq-rat*)
next
case *False*
moreover have $b \ \text{div} \ \text{gcd } a \ b * \ \text{gcd } a \ b = b$
by (*rule dvd-div-mult-self*) *simp*
ultimately have $b \ \text{div} \ \text{gcd } a \ b * \ \text{gcd } a \ b \neq 0$
by *simp*
then have $b \ \text{div} \ \text{gcd } a \ b \neq 0$
by *fastforce*
with *False* **show** *?thesis*
by (*simp add: eq-rat dvd-div-mult mult.commute [of a]*)
qed

definition *normalize* :: $\text{int} \times \text{int} \Rightarrow \text{int} \times \text{int}$

where *normalize p* =
(if snd p > 0 then (let a = gcd (fst p) (snd p) in (fst p div a, snd p div a))

else if $\text{snd } p = 0$ then $(0, 1)$
 else (let $a = - \text{gcd } (\text{fst } p) (\text{snd } p)$ in $(\text{fst } p \text{ div } a, \text{snd } p \text{ div } a)$))

lemma *normalize-crossproduct*:

assumes $q \neq 0 \ s \neq 0$

assumes $\text{normalize } (p, q) = \text{normalize } (r, s)$

shows $p * s = r * q$

proof –

have $*$: $p * s = q * r$

if $p * \text{gcd } r \ s = \text{sgn } (q * s) * r * \text{gcd } p \ q$ **and** $q * \text{gcd } r \ s = \text{sgn } (q * s) * s * \text{gcd } p \ q$

proof –

from *that* **have** $(p * \text{gcd } r \ s) * (\text{sgn } (q * s) * s * \text{gcd } p \ q) =$
 $(q * \text{gcd } r \ s) * (\text{sgn } (q * s) * r * \text{gcd } p \ q)$

by *simp*

with *assms* **show** *?thesis*

by (*auto simp add: ac-simps sgn-mult sgn-0-0*)

qed

from *assms* **show** *?thesis*

by (*auto simp: normalize-def Let-def dvd-div-div-eq-mult mult.commute sgn-mult split: if-splits intro: **)

qed

lemma *normalize-eq*: $\text{normalize } (a, b) = (p, q) \implies \text{Fract } p \ q = \text{Fract } a \ b$

by (*auto simp: normalize-def Let-def Fract-coprime dvd-div-neg rat-number-collapse split: if-split-asm*)

lemma *normalize-denom-pos*: $\text{normalize } r = (p, q) \implies q > 0$

by (*auto simp: normalize-def Let-def dvd-div-neg pos-imp-zdiv-neg-iff nonneg1-imp-zdiv-pos-iff split: if-split-asm*)

lemma *normalize-coprime*: $\text{normalize } r = (p, q) \implies \text{coprime } p \ q$

by (*auto simp: normalize-def Let-def dvd-div-neg div-gcd-coprime split: if-split-asm*)

lemma *normalize-stable* [*simp*]: $q > 0 \implies \text{coprime } p \ q \implies \text{normalize } (p, q) = (p, q)$

by (*simp add: normalize-def*)

lemma *normalize-denom-zero* [*simp*]: $\text{normalize } (p, 0) = (0, 1)$

by (*simp add: normalize-def*)

lemma *normalize-negative* [*simp*]: $q < 0 \implies \text{normalize } (p, q) = \text{normalize } (- p, - q)$

by (*simp add: normalize-def Let-def dvd-div-neg dvd-neg-div*)

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

definition *quotient-of* :: $\text{rat} \Rightarrow \text{int} \times \text{int}$

where *quotient-of* $x =$

(*THE pair*. $x = \text{Fract } (\text{fst pair}) (\text{snd pair}) \wedge \text{snd pair} > 0 \wedge \text{coprime } (\text{fst pair}) (\text{snd pair})$)

lemma *quotient-of-unique*: $\exists! p. r = \text{Fract } (\text{fst } p) (\text{snd } p) \wedge \text{snd } p > 0 \wedge \text{coprime } (\text{fst } p) (\text{snd } p)$

proof (*cases r*)

case (*Fract a b*)

then have $r = \text{Fract } (\text{fst } (a, b)) (\text{snd } (a, b)) \wedge \text{snd } (a, b) > 0 \wedge \text{coprime } (\text{fst } (a, b)) (\text{snd } (a, b))$

by *auto*

then show *?thesis*

proof (*rule ex1I*)

fix p

assume $r: r = \text{Fract } (\text{fst } p) (\text{snd } p) \wedge \text{snd } p > 0 \wedge \text{coprime } (\text{fst } p) (\text{snd } p)$

obtain $c\ d$ **where** $p = (c, d)$ **by** (*cases p*)

with r **have** $\text{Fract}' : r = \text{Fract } c\ d\ d > 0 \wedge \text{coprime } c\ d$

by *simp-all*

have $(c, d) = (a, b)$

proof (*cases a = 0*)

case *True*

with $\text{Fract } \text{Fract}'$ **show** *?thesis*

by (*simp add: eq-rat*)

next

case *False*

with $\text{Fract } \text{Fract}'$ **have** $*$: $c * b = a * d$ **and** $c \neq 0$

by (*auto simp add: eq-rat*)

then have $c * b > 0 \iff a * d > 0$

by *auto*

with $\langle b > 0 \rangle \langle d > 0 \rangle$ **have** $a > 0 \iff c > 0$

by (*simp add: zero-less-mult-iff*)

with $\langle a \neq 0 \rangle \langle c \neq 0 \rangle$ **have** $\text{sgn}: \text{sgn } a = \text{sgn } c$

by (*auto simp add: not-less*)

from $\langle \text{coprime } a\ b \rangle \langle \text{coprime } c\ d \rangle$ **have** $|a| * |d| = |c| * |b| \iff |a| = |c| \wedge |d| = |b|$

by (*simp add: coprime-crossproduct-int*)

with $\langle b > 0 \rangle \langle d > 0 \rangle$ **have** $|a| * d = |c| * b \iff |a| = |c| \wedge d = b$

by *simp*

then have $a * \text{sgn } a * d = c * \text{sgn } c * b \iff a * \text{sgn } a = c * \text{sgn } c \wedge d = b$

by (*simp add: abs-sgn*)

with $\text{sgn } *$ **show** *?thesis*

by (*auto simp add: sgn-0-0*)

qed

with p **show** $p = (a, b)$

by *simp*

qed

qed

lemma *quotient-of-Fract* [*code*]: $\text{quotient-of } (\text{Fract } a\ b) = \text{normalize } (a, b)$

proof –

have $\text{Fract } a \ b = \text{Fract } (\text{fst } (\text{normalize } (a, b))) \ (\text{snd } (\text{normalize } (a, b)))$ **(is**
 $? \text{Fract})$
by $(\text{rule } \text{sym}) \ (\text{auto } \text{intro: } \text{normalize-eq})$
moreover have $0 < \text{snd } (\text{normalize } (a, b))$ **(is** $? \text{denom-pos})$
by $(\text{cases } \text{normalize } (a, b)) \ (\text{rule } \text{normalize-denom-pos, simp})$
moreover have $\text{coprime } (\text{fst } (\text{normalize } (a, b))) \ (\text{snd } (\text{normalize } (a, b)))$ **(is**
 $? \text{coprime})$
by $(\text{rule } \text{normalize-coprime}) \ \text{simp}$
ultimately have $? \text{Fract } \wedge ? \text{denom-pos} \wedge ? \text{coprime}$ **by** blast
then have $(\text{THE } p. \text{Fract } a \ b = \text{Fract } (\text{fst } p) \ (\text{snd } p) \wedge 0 < \text{snd } p \wedge$
 $\text{coprime } (\text{fst } p) \ (\text{snd } p)) = \text{normalize } (a, b)$
by $(\text{rule } \text{the1-equality } [\text{OF } \text{quotient-of-unique}])$
then show $? \text{thesis}$ **by** $(\text{simp } \text{add: } \text{quotient-of-def})$
qed

lemma $\text{quotient-of-number}$ $[\text{simp}]$:

$\text{quotient-of } 0 = (0, 1)$

$\text{quotient-of } 1 = (1, 1)$

$\text{quotient-of } (\text{numeral } k) = (\text{numeral } k, 1)$

$\text{quotient-of } (- 1) = (- 1, 1)$

$\text{quotient-of } (- \text{numeral } k) = (- \text{numeral } k, 1)$

by $(\text{simp-all } \text{add: } \text{rat-number-expand } \text{quotient-of-Fract})$

lemma quotient-of-eq : $\text{quotient-of } (\text{Fract } a \ b) = (p, q) \implies \text{Fract } p \ q = \text{Fract } a \ b$
by $(\text{simp } \text{add: } \text{quotient-of-Fract } \text{normalize-eq})$

lemma $\text{quotient-of-denom-pos}$: $\text{quotient-of } r = (p, q) \implies q > 0$

by $(\text{cases } r) \ (\text{simp } \text{add: } \text{quotient-of-Fract } \text{normalize-denom-pos})$

lemma $\text{quotient-of-denom-pos}'$: $\text{snd } (\text{quotient-of } r) > 0$

using $\text{quotient-of-denom-pos } [\text{of } r]$ **by** $(\text{simp } \text{add: } \text{prod-eq-iff})$

lemma $\text{quotient-of-coprime}$: $\text{quotient-of } r = (p, q) \implies \text{coprime } p \ q$

by $(\text{cases } r) \ (\text{simp } \text{add: } \text{quotient-of-Fract } \text{normalize-coprime})$

lemma $\text{quotient-of-inject}$:

assumes $\text{quotient-of } a = \text{quotient-of } b$

shows $a = b$

proof –

obtain $p \ q \ r \ s$ **where** $a: a = \text{Fract } p \ q$ **and** $b: b = \text{Fract } r \ s$ **and** $q > 0$ **and** s
 > 0

by $(\text{cases } a, \text{cases } b)$

with assms **show** $? \text{thesis}$

by $(\text{simp } \text{add: } \text{eq-rat } \text{quotient-of-Fract } \text{normalize-crossproduct})$

qed

lemma $\text{quotient-of-inject-eq}$: $\text{quotient-of } a = \text{quotient-of } b \iff a = b$

by $(\text{auto } \text{simp } \text{add: } \text{quotient-of-inject})$

97.1.4 Various

lemma *Fract-of-int-quotient*: $\text{Fract } k \ l = \text{of-int } k \ / \ \text{of-int } l$
by (*simp add: Fract-of-int-eq [symmetric]*)

lemma *Fract-add-one*: $n \neq 0 \implies \text{Fract } (m + n) \ n = \text{Fract } m \ n + 1$
by (*simp add: rat-number-expand*)

lemma *quotient-of-div*:

assumes r : *quotient-of* $r = (n, d)$

shows $r = \text{of-int } n \ / \ \text{of-int } d$

proof –

from *theI* [*OF quotient-of-unique* [*of r*], *unfolded r* [*unfolded quotient-of-def*]]

have $r = \text{Fract } n \ d$ **by** *simp*

then show *?thesis* **using** *Fract-of-int-quotient*

by *simp*

qed

97.1.5 The ordered field of rational numbers

lift-definition *positive* :: *rat* \implies *bool*

is $\lambda x. 0 < \text{fst } x * \text{snd } x$

proof *clarsimp*

fix $a \ b \ c \ d$:: *int*

assume $b \neq 0$ **and** $d \neq 0$ **and** $a * d = c * b$

then have $a * d * b * d = c * b * b * d$

by *simp*

then have $a * b * d^2 = c * d * b^2$

unfolding *power2-eq-square* **by** (*simp add: ac-simps*)

then have $0 < a * b * d^2 \iff 0 < c * d * b^2$

by *simp*

then show $0 < a * b \iff 0 < c * d$

using $\langle b \neq 0 \rangle$ **and** $\langle d \neq 0 \rangle$

by (*simp add: zero-less-mult-iff*)

qed

lemma *positive-zero*: $\neg \text{positive } 0$

by *transfer simp*

lemma *positive-add*: $\text{positive } x \implies \text{positive } y \implies \text{positive } (x + y)$

apply *transfer*

apply (*auto simp add: zero-less-mult-iff add-pos-pos add-neg-neg mult-pos-neg mult-neg-pos mult-neg-neg*)

done

lemma *positive-mult*: $\text{positive } x \implies \text{positive } y \implies \text{positive } (x * y)$

apply *transfer*

by (*metis fst-conv mult.commute mult-pos-neg2 snd-conv zero-less-mult-iff*)

lemma *positive-minus*: $\neg \text{positive } x \implies x \neq 0 \implies \text{positive } (- x)$

by *transfer (auto simp: neq-iff zero-less-mult-iff mult-less-0-iff)*

instantiation *rat* :: *linordered-field*
begin

definition $x < y \longleftrightarrow \text{positive } (y - x)$

definition $x \leq y \longleftrightarrow x < y \vee x = y$ **for** $x y :: \text{rat}$

definition $|a| = (\text{if } a < 0 \text{ then } - a \text{ else } a)$ **for** $a :: \text{rat}$

definition $\text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } - 1)$ **for** $a :: \text{rat}$

instance

proof

fix $a b c :: \text{rat}$

show $|a| = (\text{if } a < 0 \text{ then } - a \text{ else } a)$

by (*rule abs-rat-def*)

show $a < b \longleftrightarrow a \leq b \wedge \neg b \leq a$

unfolding *less-eq-rat-def less-rat-def*

using *positive-add positive-zero* **by force**

show $a \leq a$

unfolding *less-eq-rat-def* **by simp**

show $a \leq b \implies b \leq c \implies a \leq c$

unfolding *less-eq-rat-def less-rat-def*

using *positive-add* **by fastforce**

show $a \leq b \implies b \leq a \implies a = b$

unfolding *less-eq-rat-def less-rat-def*

using *positive-add positive-zero* **by fastforce**

show $a \leq b \implies c + a \leq c + b$

unfolding *less-eq-rat-def less-rat-def* **by auto**

show $\text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } - 1)$

by (*rule sgn-rat-def*)

show $a \leq b \vee b \leq a$

unfolding *less-eq-rat-def less-rat-def*

by (*auto dest!: positive-minus*)

show $a < b \implies 0 < c \implies c * a < c * b$

unfolding *less-rat-def*

by (*metis diff-zero positive-mult right-diff-distrib'*)

qed

end

instantiation *rat* :: *distrib-lattice*

begin

definition (*inf* :: *rat* \Rightarrow *rat* \Rightarrow *rat*) = *min*

definition (*sup* :: *rat* \Rightarrow *rat* \Rightarrow *rat*) = *max*

instance

by *standard* (*auto simp add: inf-rat-def sup-rat-def max-min-distrib2*)

end

lemma *positive-rat*: *positive* (*Fract a b*) $\longleftrightarrow 0 < a * b$

by *transfer simp*

lemma *less-rat* [*simp*]:

$b \neq 0 \implies d \neq 0 \implies \text{Fract } a \ b < \text{Fract } c \ d \longleftrightarrow (a * d) * (b * d) < (c * b) * (b * d)$

by (*simp add: less-rat-def positive-rat algebra-simps*)

lemma *le-rat* [*simp*]:

$b \neq 0 \implies d \neq 0 \implies \text{Fract } a \ b \leq \text{Fract } c \ d \longleftrightarrow (a * d) * (b * d) \leq (c * b) * (b * d)$

by (*simp add: le-less eq-rat*)

lemma *abs-rat* [*simp, code*]: $|\text{Fract } a \ b| = \text{Fract } |a| \ |b|$

by (*auto simp add: abs-rat-def zabs-def Zero-rat-def not-less le-less eq-rat zero-less-mult-iff*)

lemma *sgn-rat* [*simp, code*]: $\text{sgn} (\text{Fract } a \ b) = \text{of-int} (\text{sgn } a * \text{sgn } b)$

unfolding *Fract-of-int-eq*

by (*auto simp: zsgn-def sgn-rat-def Zero-rat-def eq-rat*)

(*auto simp: rat-number-collapse not-less le-less zero-less-mult-iff*)

lemma *Rat-induct-pos* [*case-names Fract, induct type: rat*]:

assumes *step*: $\bigwedge a \ b. 0 < b \implies P (\text{Fract } a \ b)$

shows $P \ q$

proof (*cases q*)

case (*Fract a b*)

have *step'*: $P (\text{Fract } a \ b)$ **if** $b < 0$ **for** $a \ b :: \text{int}$

proof –

from b **have** $0 < -b$

by *simp*

then have $P (\text{Fract } (-a) \ (-b))$

by (*rule step*)

then show $P (\text{Fract } a \ b)$

by (*simp add: order-less-imp-not-eq [OF b]*)

qed

from *Fract* **show** $P \ q$

by (*auto simp add: linorder-neq-iff step step'*)

qed

lemma *zero-less-Fract-iff*: $0 < b \implies 0 < \text{Fract } a \ b \longleftrightarrow 0 < a$

by (*simp add: Zero-rat-def zero-less-mult-iff*)

lemma *Fract-less-zero-iff*: $0 < b \implies \text{Fract } a \ b < 0 \longleftrightarrow a < 0$

by (simp add: Zero-rat-def mult-less-0-iff)

lemma zero-le-Fract-iff: $0 < b \implies 0 \leq \text{Fract } a \ b \longleftrightarrow 0 \leq a$
 by (simp add: Zero-rat-def zero-le-mult-iff)

lemma Fract-le-zero-iff: $0 < b \implies \text{Fract } a \ b \leq 0 \longleftrightarrow a \leq 0$
 by (simp add: Zero-rat-def mult-le-0-iff)

lemma one-less-Fract-iff: $0 < b \implies 1 < \text{Fract } a \ b \longleftrightarrow b < a$
 by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma Fract-less-one-iff: $0 < b \implies \text{Fract } a \ b < 1 \longleftrightarrow a < b$
 by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma one-le-Fract-iff: $0 < b \implies 1 \leq \text{Fract } a \ b \longleftrightarrow b \leq a$
 by (simp add: One-rat-def mult-le-cancel-right)

lemma Fract-le-one-iff: $0 < b \implies \text{Fract } a \ b \leq 1 \longleftrightarrow a \leq b$
 by (simp add: One-rat-def mult-le-cancel-right)

97.1.6 Rationals are an Archimedean field

lemma rat-floor-lemma: $\text{of-int } (a \ \text{div } b) \leq \text{Fract } a \ b \wedge \text{Fract } a \ b < \text{of-int } (a \ \text{div } b + 1)$

proof –

have $\text{Fract } a \ b = \text{of-int } (a \ \text{div } b) + \text{Fract } (a \ \text{mod } b) \ b$

by (cases $b = 0$) (simp, simp add: of-int-rat)

moreover have $0 \leq \text{Fract } (a \ \text{mod } b) \ b \wedge \text{Fract } (a \ \text{mod } b) \ b < 1$

unfolding Fract-of-int-quotient

by (rule linorder-cases [of $b \ 0$]) (simp-all add: divide-nonpos-neg)

ultimately show ?thesis **by** simp

qed

instance rat :: archimedean-field

proof

show $\exists z. r \leq \text{of-int } z$ **for** $r :: \text{rat}$

proof (induct r)

case (Fract $a \ b$)

have $\text{Fract } a \ b \leq \text{of-int } (a \ \text{div } b + 1)$

using rat-floor-lemma [of $a \ b$] **by** simp

then show $\exists z. \text{Fract } a \ b \leq \text{of-int } z$..

qed

qed

instantiation rat :: floor-ceiling

begin

definition floor-rat :: $\text{rat} \Rightarrow \text{int}$

where $[x] = (\text{THE } z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1))$ **for** $x :: \text{rat}$

```

instance
proof
  show of-int  $\lfloor x \rfloor \leq x \wedge x < \text{of-int} (\lfloor x \rfloor + 1)$  for  $x :: \text{rat}$ 
    unfolding floor-rat-def using floor-exists1 by (rule theI')
qed

end

lemma floor-Fract [simp]:  $\lfloor \text{Fract } a \ b \rfloor = a \ \text{div} \ b$ 
  by (simp add: Fract-of-int-quotient floor-divide-of-int-eq)

```

97.2 Linear arithmetic setup

```

declaration <
  K (Lin-Arith.add-inj-thms @ { thms of-int-le-iff [THEN iffD2] of-int-eq-iff [THEN
iffD2] })
  (* not needed because  $x < (y::\text{int})$  can be rewritten as  $x + 1 \leq y$ : of-int-less-iff
RS iffD2 *)
  #> Lin-Arith.add-inj-const (const-name <of-nat>, typ <nat  $\Rightarrow$  rat>)
  #> Lin-Arith.add-inj-const (const-name <of-int>, typ <int  $\Rightarrow$  rat>)
>

```

97.3 Embedding from Rationals to other Fields

```

context field-char-0
begin

lift-definition of-rat :: rat  $\Rightarrow$  'a
  is  $\lambda x. \text{of-int} (\text{fst } x) / \text{of-int} (\text{snd } x)$ 
  by (auto simp: nonzero-divide-eq-eq nonzero-eq-divide-eq) (simp only: of-int-mult
[symmetric])

end

lemma of-rat-rat:  $b \neq 0 \implies \text{of-rat} (\text{Fract } a \ b) = \text{of-int } a / \text{of-int } b$ 
  by transfer simp

lemma of-rat-0 [simp]: of-rat 0 = 0
  by transfer simp

lemma of-rat-1 [simp]: of-rat 1 = 1
  by transfer simp

lemma of-rat-add: of-rat (a + b) = of-rat a + of-rat b
  by transfer (simp add: add-frac-eq)

lemma of-rat-minus: of-rat (- a) = - of-rat a
  by transfer simp

```

lemma *of-rat-neg-one* [*simp*]: $\text{of-rat } (- 1) = - 1$
by (*simp add: of-rat-minus*)

lemma *of-rat-diff*: $\text{of-rat } (a - b) = \text{of-rat } a - \text{of-rat } b$
using *of-rat-add* [*of a - b*] **by** (*simp add: of-rat-minus*)

lemma *of-rat-mult*: $\text{of-rat } (a * b) = \text{of-rat } a * \text{of-rat } b$
by *transfer* (*simp add: divide-inverse nonzero-inverse-mult-distrib ac-simps*)

lemma *of-rat-sum*: $\text{of-rat } (\sum a \in A. f a) = (\sum a \in A. \text{of-rat } (f a))$
by (*induct rule: infinite-finite-induct*) (*auto simp: of-rat-add*)

lemma *of-rat-prod*: $\text{of-rat } (\prod a \in A. f a) = (\prod a \in A. \text{of-rat } (f a))$
by (*induct rule: infinite-finite-induct*) (*auto simp: of-rat-mult*)

lemma *nonzero-of-rat-inverse*: $a \neq 0 \implies \text{of-rat } (\text{inverse } a) = \text{inverse } (\text{of-rat } a)$
by (*rule inverse-unique [symmetric]*) (*simp add: of-rat-mult [symmetric]*)

lemma *of-rat-inverse*: $(\text{of-rat } (\text{inverse } a) :: 'a::\text{field-char-0}) = \text{inverse } (\text{of-rat } a)$
by (*cases a = 0*) (*simp-all add: nonzero-of-rat-inverse*)

lemma *nonzero-of-rat-divide*: $b \neq 0 \implies \text{of-rat } (a / b) = \text{of-rat } a / \text{of-rat } b$
by (*simp add: divide-inverse of-rat-mult nonzero-of-rat-inverse*)

lemma *of-rat-divide*: $(\text{of-rat } (a / b) :: 'a::\text{field-char-0}) = \text{of-rat } a / \text{of-rat } b$
by (*cases b = 0*) (*simp-all add: nonzero-of-rat-divide*)

lemma *of-rat-power*: $(\text{of-rat } (a ^ n) :: 'a::\text{field-char-0}) = \text{of-rat } a ^ n$
by (*induct n*) (*simp-all add: of-rat-mult*)

lemma *of-rat-eq-iff* [*simp*]: $\text{of-rat } a = \text{of-rat } b \iff a = b$
apply *transfer*
apply (*simp add: nonzero-divide-eq-eq nonzero-eq-divide-eq flip: of-int-mult*)
done

lemma *of-rat-eq-0-iff* [*simp*]: $\text{of-rat } a = 0 \iff a = 0$
using *of-rat-eq-iff* [*of - 0*] **by** *simp*

lemma *zero-eq-of-rat-iff* [*simp*]: $0 = \text{of-rat } a \iff 0 = a$
by *simp*

lemma *of-rat-eq-1-iff* [*simp*]: $\text{of-rat } a = 1 \iff a = 1$
using *of-rat-eq-iff* [*of - 1*] **by** *simp*

lemma *one-eq-of-rat-iff* [*simp*]: $1 = \text{of-rat } a \iff 1 = a$
by *simp*

lemma *of-rat-less*: $(\text{of-rat } r :: 'a::\text{linordered-field}) < \text{of-rat } s \iff r < s$
proof (*induct r, induct s*)

```

fix a b c d :: int
assume not-zero: b > 0 d > 0
then have b * d > 0 by simp
have of-int-divide-less-eg:
  (of-int a :: 'a) / of-int b < of-int c / of-int d  $\longleftrightarrow$ 
  (of-int a :: 'a) * of-int d < of-int c * of-int b
  using not-zero by (simp add: pos-less-divide-eg pos-divide-less-eg)
show (of-rat (Fract a b) :: 'a::linordered-field) < of-rat (Fract c d)  $\longleftrightarrow$ 
  Fract a b < Fract c d
  using not-zero <b * d > 0>
  by (simp add: of-rat-rat of-int-divide-less-eg of-int-mult [symmetric] del: of-int-mult)
qed

```

```

lemma of-rat-less-eg: (of-rat r :: 'a::linordered-field)  $\leq$  of-rat s  $\longleftrightarrow$  r  $\leq$  s
  unfolding le-less by (auto simp add: of-rat-less)

```

```

lemma of-rat-le-0-iff [simp]: (of-rat r :: 'a::linordered-field)  $\leq$  0  $\longleftrightarrow$  r  $\leq$  0
  using of-rat-less-eg [of r 0, where 'a = 'a] by simp

```

```

lemma zero-le-of-rat-iff [simp]: 0  $\leq$  (of-rat r :: 'a::linordered-field)  $\longleftrightarrow$  0  $\leq$  r
  using of-rat-less-eg [of 0 r, where 'a = 'a] by simp

```

```

lemma of-rat-le-1-iff [simp]: (of-rat r :: 'a::linordered-field)  $\leq$  1  $\longleftrightarrow$  r  $\leq$  1
  using of-rat-less-eg [of r 1] by simp

```

```

lemma one-le-of-rat-iff [simp]: 1  $\leq$  (of-rat r :: 'a::linordered-field)  $\longleftrightarrow$  1  $\leq$  r
  using of-rat-less-eg [of 1 r] by simp

```

```

lemma of-rat-less-0-iff [simp]: (of-rat r :: 'a::linordered-field) < 0  $\longleftrightarrow$  r < 0
  using of-rat-less [of r 0, where 'a = 'a] by simp

```

```

lemma zero-less-of-rat-iff [simp]: 0 < (of-rat r :: 'a::linordered-field)  $\longleftrightarrow$  0 < r
  using of-rat-less [of 0 r, where 'a = 'a] by simp

```

```

lemma of-rat-less-1-iff [simp]: (of-rat r :: 'a::linordered-field) < 1  $\longleftrightarrow$  r < 1
  using of-rat-less [of r 1] by simp

```

```

lemma one-less-of-rat-iff [simp]: 1 < (of-rat r :: 'a::linordered-field)  $\longleftrightarrow$  1 < r
  using of-rat-less [of 1 r] by simp

```

```

lemma of-rat-eq-id [simp]: of-rat = id

```

```

proof

```

```

  show of-rat a = id a for a

```

```

    by (induct a) (simp add: of-rat-rat Fract-of-int-eq [symmetric])

```

```

qed

```

```

lemma abs-of-rat [simp]:

```

```

  |of-rat r| = (of-rat |r| :: 'a :: linordered-field)

```

```

  by (cases r  $\geq$  0) (simp-all add: not-le of-rat-minus)

```


Collapse nested embeddings.

lemma *of-rat-of-nat-eq* [*simp*]: *of-rat (of-nat n) = of-nat n*
by (*induct n*) (*simp-all add: of-rat-add*)

lemma *of-rat-of-int-eq* [*simp*]: *of-rat (of-int z) = of-int z*
by (*cases z rule: int-diff-cases*) (*simp add: of-rat-diff*)

lemma *of-rat-numeral-eq* [*simp*]: *of-rat (numeral w) = numeral w*
using *of-rat-of-int-eq [of numeral w]* **by** *simp*

lemma *of-rat-neg-numeral-eq* [*simp*]: *of-rat (- numeral w) = - numeral w*
using *of-rat-of-int-eq [of - numeral w]* **by** *simp*

lemma *of-rat-floor* [*simp*]:
 $\lfloor \text{of-rat } r \rfloor = \lfloor r \rfloor$
by (*cases r*) (*simp add: Fract-of-int-quotient of-rat-divide floor-divide-of-int-eq*)

lemma *of-rat-ceiling* [*simp*]:
 $\lceil \text{of-rat } r \rceil = \lceil r \rceil$
using *of-rat-floor [of - r]* **by** (*simp add: of-rat-minus ceiling-def*)

lemmas *zero-rat = Zero-rat-def*

lemmas *one-rat = One-rat-def*

abbreviation *rat-of-nat* :: *nat* \Rightarrow *rat*
where *rat-of-nat* \equiv *of-nat*

abbreviation *rat-of-int* :: *int* \Rightarrow *rat*
where *rat-of-int* \equiv *of-int*

97.4 The Set of Rational Numbers

context *field-char-0*

begin

definition *Rats* :: 'a set (\mathbb{Q})
where $\mathbb{Q} = \text{range of-rat}$

end

lemma *Rats-cases* [*cases set: Rats*]:
assumes $q \in \mathbb{Q}$
obtains (*of-rat*) *r* **where** $q = \text{of-rat } r$
proof –
from $\langle q \in \mathbb{Q} \rangle$ **have** $q \in \text{range of-rat}$
by (*simp only: Rats-def*)
then obtain *r* **where** $q = \text{of-rat } r$..
then show *thesis* ..
qed

lemma *Rats-cases'*:

assumes $(x :: 'a :: \text{field-char-0}) \in \mathbb{Q}$

obtains $a\ b$ **where** $b > 0$ *coprime* $a\ b$ $x = \text{of-int } a / \text{of-int } b$

proof –

from *assms* **obtain** r **where** $x = \text{of-rat } r$

by (*auto simp: Rats-def*)

obtain $a\ b$ **where** *quot: quotient-of* $r = (a,b)$ **by force**

have $b > 0$ **using** *quotient-of-denom-pos*[*OF quot*].

moreover have *coprime* $a\ b$ **using** *quotient-of-coprime*[*OF quot*].

moreover have $x = \text{of-int } a / \text{of-int } b$ **unfolding** $\langle x = \text{of-rat } r \rangle$

quotient-of-div[*OF quot*] **by** (*simp add: of-rat-divide*)

ultimately show *?thesis* **using that by blast**

qed

lemma *Rats-of-rat* [*simp*]: $\text{of-rat } r \in \mathbb{Q}$

by (*simp add: Rats-def*)

lemma *Rats-of-int* [*simp*]: $\text{of-int } z \in \mathbb{Q}$

by (*subst of-rat-of-int-eq* [*symmetric*]) (*rule Rats-of-rat*)

lemma *Ints-subset-Rats*: $\mathbb{Z} \subseteq \mathbb{Q}$

using *Ints-cases Rats-of-int* **by blast**

lemma *Rats-of-nat* [*simp*]: $\text{of-nat } n \in \mathbb{Q}$

by (*subst of-rat-of-nat-eq* [*symmetric*]) (*rule Rats-of-rat*)

lemma *Nats-subset-Rats*: $\mathbb{N} \subseteq \mathbb{Q}$

using *Ints-subset-Rats Nats-subset-Ints* **by blast**

lemma *Rats-number-of* [*simp*]: *numeral* $w \in \mathbb{Q}$

by (*subst of-rat-numeral-eq* [*symmetric*]) (*rule Rats-of-rat*)

lemma *Rats-0* [*simp*]: $0 \in \mathbb{Q}$

unfolding *Rats-def* **by** (*rule range-eqI*) (*rule of-rat-0* [*symmetric*])

lemma *Rats-1* [*simp*]: $1 \in \mathbb{Q}$

unfolding *Rats-def* **by** (*rule range-eqI*) (*rule of-rat-1* [*symmetric*])

lemma *Rats-add* [*simp*]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a + b \in \mathbb{Q}$

by (*metis Rats-cases Rats-of-rat of-rat-add*)

lemma *Rats-minus-iff* [*simp*]: $- a \in \mathbb{Q} \iff a \in \mathbb{Q}$

by (*metis Rats-cases Rats-of-rat add.inverse-inverse of-rat-minus*)

lemma *Rats-diff* [*simp*]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a - b \in \mathbb{Q}$

by (*metis Rats-add Rats-minus-iff diff-conv-add-uminus*)

lemma *Rats-mult* [*simp*]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a * b \in \mathbb{Q}$

by (metis Rats-cases Rats-of-rat of-rat-mult)

lemma *Rats-inverse* [simp]: $a \in \mathbb{Q} \implies \text{inverse } a \in \mathbb{Q}$
 for $a :: 'a::\text{field-char-0}$
 by (metis Rats-cases Rats-of-rat of-rat-inverse)

lemma *Rats-divide* [simp]: $a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a / b \in \mathbb{Q}$
 for $a b :: 'a::\text{field-char-0}$
 by (simp add: divide-inverse)

lemma *Rats-power* [simp]: $a \in \mathbb{Q} \implies a ^ n \in \mathbb{Q}$
 for $a :: 'a::\text{field-char-0}$
 by (metis Rats-cases Rats-of-rat of-rat-power)

lemma *Rats-induct* [case-names of-rat, induct set: Rats]: $q \in \mathbb{Q} \implies (\bigwedge r. P (\text{of-rat } r)) \implies P q$
 by (rule Rats-cases) auto

lemma *Rats-infinite*: $\neg \text{finite } \mathbb{Q}$
 by (auto dest!: finite-imageD simp: inj-on-def infinite-UNIV-char-0 Rats-def)

lemma *Rats-add-iff*: $a \in \mathbb{Q} \vee b \in \mathbb{Q} \implies a+b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 by (metis Rats-add Rats-diff add-diff-cancel add-diff-cancel-left')

lemma *Rats-diff-iff*: $a \in \mathbb{Q} \vee b \in \mathbb{Q} \implies a-b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 by (metis Rats-add-iff diff-add-cancel)

lemma *Rats-mult-iff*: $a \in \mathbb{Q} - \{0\} \vee b \in \mathbb{Q} - \{0\} \implies a*b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 by (metis Diff-iff Rats-divide Rats-mult insertI1 mult.commute nonzero-divide-eq-eq)

lemma *Rats-inverse-iff* [simp]: $\text{inverse } a \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q}$
 using *Rats-inverse* by force

lemma *Rats-divide-iff*: $a \in \mathbb{Q} - \{0\} \vee b \in \mathbb{Q} - \{0\} \implies a/b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
 by (metis Rats-divide Rats-mult-iff divide-eq-0-iff divide-inverse nonzero-mult-div-cancel-right)

97.5 Implementation of rational numbers as pairs of integers

Formal constructor

definition *Frct* :: $\text{int} \times \text{int} \Rightarrow \text{rat}$
 where [simp]: $\text{Frct } p = \text{Fract } (\text{fst } p) (\text{snd } p)$

lemma [code abstype]: $\text{Frct } (\text{quotient-of } q) = q$
 by (cases q) (auto intro: quotient-of-eq)

Numerals

declare *quotient-of-Fract* [code abstract]

definition *of-int* :: *int* ⇒ *rat*
where [*code-abbrev*]: *of-int* = *Int.of-int*

hide-const (**open**) *of-int*

lemma *quotient-of-int* [*code abstract*]: *quotient-of* (*Rat.of-int* *a*) = (*a*, 1)
by (*simp add: of-int-def of-int-rat quotient-of-Fract*)

lemma [*code-unfold*]: *numeral k* = *Rat.of-int* (*numeral k*)
by (*simp add: Rat.of-int-def*)

lemma [*code-unfold*]: $- \text{numeral } k = \text{Rat.of-int } (- \text{numeral } k)$
by (*simp add: Rat.of-int-def*)

lemma *Frct-code-post* [*code-post*]:
 $\text{Frct } (0, a) = 0$
 $\text{Frct } (a, 0) = 0$
 $\text{Frct } (1, 1) = 1$
 $\text{Frct } (\text{numeral } k, 1) = \text{numeral } k$
 $\text{Frct } (1, \text{numeral } k) = 1 / \text{numeral } k$
 $\text{Frct } (\text{numeral } k, \text{numeral } l) = \text{numeral } k / \text{numeral } l$
 $\text{Frct } (- a, b) = - \text{Frct } (a, b)$
 $\text{Frct } (a, - b) = - \text{Frct } (a, b)$
 $- (- \text{Frct } q) = \text{Frct } q$
by (*simp-all add: Fract-of-int-quotient*)

Operations

lemma *rat-zero-code* [*code abstract*]: *quotient-of* 0 = (0, 1)
by (*simp add: Zero-rat-def quotient-of-Fract normalize-def*)

lemma *rat-one-code* [*code abstract*]: *quotient-of* 1 = (1, 1)
by (*simp add: One-rat-def quotient-of-Fract normalize-def*)

lemma *rat-plus-code* [*code abstract*]:
 $\text{quotient-of } (p + q) = (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q$
 $\text{in normalize } (a * d + b * c, c * d))$
by (*cases p, cases q*) (*simp add: quotient-of-Fract*)

lemma *rat-uminus-code* [*code abstract*]:
 $\text{quotient-of } (- p) = (\text{let } (a, b) = \text{quotient-of } p \text{ in } (- a, b))$
by (*cases p*) (*simp add: quotient-of-Fract*)

lemma *rat-minus-code* [*code abstract*]:
 $\text{quotient-of } (p - q) =$
 $(\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q$
 $\text{in normalize } (a * d - b * c, c * d))$
by (*cases p, cases q*) (*simp add: quotient-of-Fract*)

lemma *rat-times-code* [*code abstract*]:

$\text{quotient-of } (p * q) =$
 $(\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q$
 $\text{in normalize } (a * b, c * d))$
by (cases p, cases q) (simp add: quotient-of-Fract)

lemma *rat-inverse-code* [code abstract]:

$\text{quotient-of } (\text{inverse } p) =$
 $(\text{let } (a, b) = \text{quotient-of } p$
 $\text{in if } a = 0 \text{ then } (0, 1) \text{ else } (\text{sgn } a * b, |a|))$

proof (cases p)

case (Fract a b)

then show ?thesis

by (cases 0::int a rule: linorder-cases) (simp-all add: quotient-of-Fract ac-simps)

qed

lemma *rat-divide-code* [code abstract]:

$\text{quotient-of } (p / q) =$
 $(\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q$
 $\text{in normalize } (a * d, c * b))$

by (cases p, cases q) (simp add: quotient-of-Fract)

lemma *rat-abs-code* [code abstract]:

$\text{quotient-of } |p| = (\text{let } (a, b) = \text{quotient-of } p \text{ in } (|a|, b))$

by (cases p) (simp add: quotient-of-Fract)

lemma *rat-sgn-code* [code abstract]: $\text{quotient-of } (\text{sgn } p) = (\text{sgn } (\text{fst } (\text{quotient-of } p)), 1)$

proof (cases p)

case (Fract a b)

then show ?thesis

by (cases 0::int a rule: linorder-cases) (simp-all add: quotient-of-Fract)

qed

lemma *rat-floor-code* [code]: $\lfloor p \rfloor = (\text{let } (a, b) = \text{quotient-of } p \text{ in } a \text{ div } b)$

by (cases p) (simp add: quotient-of-Fract floor-Fract)

instantiation *rat* :: equal

begin

definition [code]: $\text{HOL.equal } a \ b \longleftrightarrow \text{quotient-of } a = \text{quotient-of } b$

instance

by standard (simp add: equal-rat-def quotient-of-inject-eq)

lemma *rat-eq-refl* [code nbe]: $\text{HOL.equal } (r::\text{rat}) \ r \longleftrightarrow \text{True}$

by (rule equal-refl)

end

lemma *rat-less-eq-code* [code]:

$p \leq q \iff (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a * d \leq c * b)$
by (cases p, cases q) (simp add: quotient-of-Fract mult.commute)

lemma *rat-less-code* [code]:

$p < q \iff (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a * d < c * b)$
by (cases p, cases q) (simp add: quotient-of-Fract mult.commute)

lemma [code]: *of-rat* p = (let (a, b) = quotient-of p in of-int a / of-int b)

by (cases p) (simp add: quotient-of-Fract of-rat-rat)

Quickcheck

context

includes *term-syntax*

begin

definition

valterm-fract :: int × (unit ⇒ Code-Evaluation.term) ⇒
 int × (unit ⇒ Code-Evaluation.term) ⇒
 rat × (unit ⇒ Code-Evaluation.term)

where [code-unfold]: *valterm-fract* k l = Code-Evaluation.valtermify Fract {·} k {·} l

end

instantiation *rat* :: random

begin

context

includes *state-combinator-syntax*

begin

definition

Quickcheck-Random.random i =
Quickcheck-Random.random i ◦→ (λnum. Random.range i ◦→ (λdenom. Pair
 (let j = int-of-integer (integer-of-natural (denom + 1))
 in valterm-fract num (j, λu. Code-Evaluation.term-of j))))

instance ..

end

end

instantiation *rat* :: exhaustive

begin

definition

exhaustive-rat f d =

```

Quickcheck-Exhaustive.exhaustive
  (λl. Quickcheck-Exhaustive.exhaustive
    (λk. f (Fract k (int-of-integer (integer-of-natural l) + 1))) d) d

```

instance ..

end

instantiation rat :: full-exhaustive

begin

definition

```

full-exhaustive-rat f d =
  Quickcheck-Exhaustive.full-exhaustive
    (λ(l, -). Quickcheck-Exhaustive.full-exhaustive
      (λk. f
        (let j = int-of-integer (integer-of-natural l) + 1
          in valterm-fract k (j, λ-. Code-Evaluation.term-of j))) d) d

```

instance ..

end

instance rat :: partial-term-of ..

lemma [code]:

```

partial-term-of (ty :: rat itself) (Quickcheck-Narrowing.Narrowing-variable p tt)
≡
  Code-Evaluation.Free (STR "-") (Typerep.Typerep (STR "Rat.rat") [])
partial-term-of (ty :: rat itself) (Quickcheck-Narrowing.Narrowing-constructor 0
[l, k]) ≡
  Code-Evaluation.App
    (Code-Evaluation.Const (STR "Rat.Frct")
      (Typerep.Typerep (STR "fun")
        [Typerep.Typerep (STR "Product-Type.prod")
          [Typerep.Typerep (STR "Int.int") [], Typerep.Typerep (STR "Int.int") []],
            Typerep.Typerep (STR "Rat.rat") []]))
    (Code-Evaluation.App
      (Code-Evaluation.App
        (Code-Evaluation.Const (STR "Product-Type.Pair")
          (Typerep.Typerep (STR "fun")
            [Typerep.Typerep (STR "Int.int") [],
              Typerep.Typerep (STR "fun")
                [Typerep.Typerep (STR "Int.int") [],
                  Typerep.Typerep (STR "Product-Type.prod")
                    [Typerep.Typerep (STR "Int.int") [], Typerep.Typerep (STR "Int.int")
]]))
          []))
        (partial-term-of (TYPE(int)) l)) (partial-term-of (TYPE(int)) k))
    []))
  by (rule partial-term-of-anything)+

```

```
instantiation rat :: narrowing
begin
```

```
definition
```

```
  narrowing =
    Quickcheck-Narrowing.apply
      (Quickcheck-Narrowing.apply
        (Quickcheck-Narrowing.cons (λnom denom. Fract nom denom)) narrowing)
  narrowing
```

```
instance ..
```

```
end
```

97.6 Setup for Nitpick

```
declaration <
```

```
  Nitpick-HOL.register-frac-type type-name <rat>
    [(const-name <Abs-Rat>, const-name <Nitpick.Abs-Frac>),
     (const-name <zero-rat-inst.zero-rat>, const-name <Nitpick.zero-frac>),
     (const-name <one-rat-inst.one-rat>, const-name <Nitpick.one-frac>),
     (const-name <plus-rat-inst.plus-rat>, const-name <Nitpick.plus-frac>),
     (const-name <times-rat-inst.times-rat>, const-name <Nitpick.times-frac>),
     (const-name <uminus-rat-inst.uminus-rat>, const-name <Nitpick.uminus-frac>),
     (const-name <inverse-rat-inst.inverse-rat>, const-name <Nitpick.inverse-frac>),
     (const-name <ord-rat-inst.less-rat>, const-name <Nitpick.less-frac>),
     (const-name <ord-rat-inst.less-eq-rat>, const-name <Nitpick.less-eq-frac>),
     (const-name <field-char-0-class.of-rat>, const-name <Nitpick.of-frac>)]
  >
```

```
lemmas [nitpick-unfold] =
```

```
  inverse-rat-inst.inverse-rat
  one-rat-inst.one-rat ord-rat-inst.less-rat
  ord-rat-inst.less-eq-rat plus-rat-inst.plus-rat times-rat-inst.times-rat
  uminus-rat-inst.uminus-rat zero-rat-inst.zero-rat
```

97.7 Float syntax

```
syntax -Float :: float-const ⇒ 'a (-)
```

```
parse-translation <
```

```
  let
    fun mk-frac str =
      let
        val {mant = i, exp = n} = Lexicon.read-float str;
        val exp = Syntax.const const-syntax <Power.power>;
        val ten = Numeral.mk-number-syntax 10;
        val exp10 = if n = 1 then ten else exp $ ten $ Numeral.mk-number-syntax
          n;

```



```

in Syntax.const const-syntax ⟨Fields.inverse-divide⟩ $ Numeral.mk-number-syntax
i $ exp10 end;

```

```

fun float-tr [(c as Const (syntax-const ⟨-constrain⟩, -)) $ t $ u] = c $ float-tr
[t] $ u
| float-tr [t as Const (str, -)] = mk-frac str
| float-tr ts = raise TERM (float-tr, ts);
in [(syntax-const ⟨-Float⟩, K float-tr)] end
>

```

Test:

```

lemma 123.456 = -111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::rat)
by simp

```

97.8 Hiding implementation details

```

hide-const (open) normalize positive

```

```

lifting-update rat.lifting

```

```

lifting-forget rat.lifting

```

end

98 Development of the Reals using Cauchy Sequences

```

theory Real

```

```

imports Rat

```

```

begin

```

This theory contains a formalization of the real numbers as equivalence classes of Cauchy sequences of rationals. See the AFP entry *Dedekind-Real* for an alternative construction using Dedekind cuts.

98.1 Preliminary lemmas

Useful in convergence arguments

```

lemma inverse-of-nat-le:

```

```

fixes n::nat shows  $[[n \leq m; n \neq 0]] \implies 1 / \text{of-nat } m \leq (1 :: 'a :: \text{linordered-field}) / \text{of-nat } n$ 

```

```

by (simp add: frac-le)

```

```

lemma add-diff-add:  $(a + c) - (b + d) = (a - b) + (c - d)$ 

```

```

for a b c d :: 'a::ab-group-add

```

```

by simp

```

```

lemma minus-diff-minus:  $- a - - b = - (a - b)$ 

```

for $a\ b :: 'a::ab\text{-group-add}$
by *simp*

lemma *mult-diff-mult*: $(x * y - a * b) = x * (y - b) + (x - a) * b$
for $x\ y\ a\ b :: 'a::ring$
by (*simp add: algebra-simps*)

lemma *inverse-diff-inverse*:
fixes $a\ b :: 'a::division\text{-ring}$
assumes $a \neq 0$ **and** $b \neq 0$
shows $inverse\ a - inverse\ b = - (inverse\ a * (a - b) * inverse\ b)$
using *assms* **by** (*simp add: algebra-simps*)

lemma *obtain-pos-sum*:
fixes $r :: rat$ **assumes** $r: 0 < r$
obtains $s\ t$ **where** $0 < s$ **and** $0 < t$ **and** $r = s + t$
proof
from r **show** $0 < r/2$ **by** *simp*
from r **show** $0 < r/2$ **by** *simp*
show $r = r/2 + r/2$ **by** *simp*
qed

98.2 Sequences that converge to zero

definition *vanishes* :: $(nat \Rightarrow rat) \Rightarrow bool$
where *vanishes* $X \longleftrightarrow (\forall r > 0. \exists k. \forall n \geq k. |X\ n| < r)$

lemma *vanishesI*: $(\bigwedge r. 0 < r \implies \exists k. \forall n \geq k. |X\ n| < r) \implies vanishes\ X$
unfolding *vanishes-def* **by** *simp*

lemma *vanishesD*: $vanishes\ X \implies 0 < r \implies \exists k. \forall n \geq k. |X\ n| < r$
unfolding *vanishes-def* **by** *simp*

lemma *vanishes-const* [*simp*]: $vanishes\ (\lambda n. c) \longleftrightarrow c = 0$

proof (*cases* $c = 0$)
case *True*
then show *?thesis*
by (*simp add: vanishesI*)
next
case *False*
then show *?thesis*
unfolding *vanishes-def*
using *zero-less-abs-iff* **by** *blast*
qed

lemma *vanishes-minus*: $vanishes\ X \implies vanishes\ (\lambda n. - X\ n)$
unfolding *vanishes-def* **by** *simp*

lemma *vanishes-add*:

```

assumes  $X$ : vanishes  $X$ 
and  $Y$ : vanishes  $Y$ 
shows vanishes  $(\lambda n. X\ n + Y\ n)$ 
proof (rule vanishesI)
  fix  $r :: \text{rat}$ 
  assume  $0 < r$ 
  then obtain  $s\ t$  where  $s: 0 < s$  and  $t: 0 < t$  and  $r: r = s + t$ 
    by (rule obtain-pos-sum)
  obtain  $i$  where  $i: \forall n \geq i. |X\ n| < s$ 
    using vanishesD [OF X s] ..
  obtain  $j$  where  $j: \forall n \geq j. |Y\ n| < t$ 
    using vanishesD [OF Y t] ..
  have  $\forall n \geq \max\ i\ j. |X\ n + Y\ n| < r$ 
proof clarsimp
  fix  $n$ 
  assume  $n: i \leq n\ j \leq n$ 
  have  $|X\ n + Y\ n| \leq |X\ n| + |Y\ n|$ 
    by (rule abs-triangle-ineq)
  also have  $\dots < s + t$ 
    by (simp add: add-strict-mono i j n)
  finally show  $|X\ n + Y\ n| < r$ 
    by (simp only: r)
qed
then show  $\exists k. \forall n \geq k. |X\ n + Y\ n| < r$  ..
qed

```

lemma *vanishes-diff*:

```

assumes vanishes  $X$  vanishes  $Y$ 
shows vanishes  $(\lambda n. X\ n - Y\ n)$ 
unfolding diff-conv-add-uminus by (intro vanishes-add vanishes-minus assms)

```

lemma *vanishes-mult-bounded*:

```

assumes  $X: \exists a > 0. \forall n. |X\ n| < a$ 
assumes  $Y: \text{vanishes } (\lambda n. Y\ n)$ 
shows vanishes  $(\lambda n. X\ n * Y\ n)$ 
proof (rule vanishesI)
  fix  $r :: \text{rat}$ 
  assume  $r: 0 < r$ 
  obtain  $a$  where  $a: 0 < a \forall n. |X\ n| < a$ 
    using  $X$  by blast
  obtain  $b$  where  $b: 0 < b\ r = a * b$ 
proof
  show  $0 < r / a$  using  $r\ a$  by simp
  show  $r = a * (r / a)$  using  $a$  by simp
qed
obtain  $k$  where  $k: \forall n \geq k. |Y\ n| < b$ 
  using vanishesD [OF Y b(1)] ..
have  $\forall n \geq k. |X\ n * Y\ n| < r$ 
  by (simp add: b(2) abs-mult mult-strict-mono' a k)

```

then show $\exists k. \forall n \geq k. |X n * Y n| < r ..$
qed

98.3 Cauchy sequences

definition *cauchy* :: (nat \Rightarrow rat) \Rightarrow bool
where *cauchy* X $\longleftrightarrow (\forall r > 0. \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r)$

lemma *cauchyI*: $(\bigwedge r. 0 < r \implies \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r) \implies \text{cauchy } X$

unfolding *cauchy-def* **by** *simp*

lemma *cauchyD*: *cauchy* X $\implies 0 < r \implies \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r$
unfolding *cauchy-def* **by** *simp*

lemma *cauchy-const* [*simp*]: *cauchy* ($\lambda n. x$)
unfolding *cauchy-def* **by** *simp*

lemma *cauchy-add* [*simp*]:
assumes X: *cauchy* X **and** Y: *cauchy* Y
shows *cauchy* ($\lambda n. X n + Y n$)

proof (*rule cauchyI*)

fix r :: rat

assume 0 < r

then obtain s t **where** s: 0 < s **and** t: 0 < t **and** r: r = s + t

by (*rule obtain-pos-sum*)

obtain i **where** i: $\forall m \geq i. \forall n \geq i. |X m - X n| < s$

using *cauchyD* [*OF X s*] ..

obtain j **where** j: $\forall m \geq j. \forall n \geq j. |Y m - Y n| < t$

using *cauchyD* [*OF Y t*] ..

have $\forall m \geq \max i j. \forall n \geq \max i j. |(X m + Y m) - (X n + Y n)| < r$

proof *clarsimp*

fix m n

assume *: i \leq m j \leq m i \leq n j \leq n

have $|(X m + Y m) - (X n + Y n)| \leq |X m - X n| + |Y m - Y n|$

unfolding *add-diff-add* **by** (*rule abs-triangle-ineq*)

also have ... < s + t

by (*rule add-strict-mono*) (*simp-all add: i j **)

finally show $|(X m + Y m) - (X n + Y n)| < r$ **by** (*simp only: r*)

qed

then show $\exists k. \forall m \geq k. \forall n \geq k. |(X m + Y m) - (X n + Y n)| < r ..$

qed

lemma *cauchy-minus* [*simp*]:
assumes X: *cauchy* X
shows *cauchy* ($\lambda n. - X n$)
using *assms* **unfolding** *cauchy-def*
unfolding *minus-diff-minus abs-minus-cancel* .

lemma *cauchy-diff* [*simp*]:
assumes *cauchy X cauchy Y*
shows *cauchy* ($\lambda n. X\ n - Y\ n$)
using *assms unfolding diff-conv-add-uminus* **by** (*simp del: add-uminus-conv-diff*)

lemma *cauchy-imp-bounded*:
assumes *cauchy X*
shows $\exists b > 0. \forall n. |X\ n| < b$
proof –
obtain *k* **where** *k*: $\forall m \geq k. \forall n \geq k. |X\ m - X\ n| < 1$
using *cauchyD [OF assms zero-less-one]* ..
show $\exists b > 0. \forall n. |X\ n| < b$
proof (*intro exI conjI allI*)
have $0 \leq |X\ 0|$ **by** *simp*
also have $|X\ 0| \leq \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\})$ **by** *simp*
finally have $0 \leq \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\})$.
then show $0 < \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\}) + 1$ **by** *simp*
next
fix *n* :: *nat*
show $|X\ n| < \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\}) + 1$
proof (*rule linorder-le-cases*)
assume $n \leq k$
then have $|X\ n| \leq \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\})$ **by** *simp*
then show $|X\ n| < \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\}) + 1$ **by** *simp*
next
assume $k \leq n$
have $|X\ n| = |X\ k + (X\ n - X\ k)|$ **by** *simp*
also have $|X\ k + (X\ n - X\ k)| \leq |X\ k| + |X\ n - X\ k|$
by (*rule abs-triangle-ineq*)
also have $\dots < \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\}) + 1$
by (*rule add-le-less-mono*) (*simp-all add: k < k ≤ n*)
finally show $|X\ n| < \text{Max} (\text{abs} \text{ ` } X \text{ ` } \{..k\}) + 1$.
qed
qed
qed

lemma *cauchy-mult* [*simp*]:
assumes *X: cauchy X and Y: cauchy Y*
shows *cauchy* ($\lambda n. X\ n * Y\ n$)
proof (*rule cauchyI*)
fix *r* :: *rat* **assume** $0 < r$
then obtain *u v* **where** $0 < u$ **and** $0 < v$ **and** $r = u + v$
by (*rule obtain-pos-sum*)
obtain *a* **where** $0 < a \forall n. |X\ n| < a$
using *cauchy-imp-bounded [OF X]* **by** *blast*
obtain *b* **where** $0 < b \forall n. |Y\ n| < b$
using *cauchy-imp-bounded [OF Y]* **by** *blast*
obtain *s t* **where** $0 < s$ **and** $0 < t$ **and** $r = a * t + s * b$
proof

```

  show  $0 < v/b$  using  $v\ b(1)$  by simp
  show  $0 < u/a$  using  $u\ a(1)$  by simp
  show  $r = a * (u/a) + (v/b) * b$ 
    using  $a(1)\ b(1)\ \langle r = u + v \rangle$  by simp
qed
obtain  $i$  where  $i: \forall m \geq i. \forall n \geq i. |X\ m - X\ n| < s$ 
  using cauchyD [OF  $X\ s$ ] ..
obtain  $j$  where  $j: \forall m \geq j. \forall n \geq j. |Y\ m - Y\ n| < t$ 
  using cauchyD [OF  $Y\ t$ ] ..
have  $\forall m \geq \max\ i\ j. \forall n \geq \max\ i\ j. |X\ m * Y\ m - X\ n * Y\ n| < r$ 
proof clarsimp
  fix  $m\ n$ 
  assume *:  $i \leq m\ j \leq n\ i \leq n\ j \leq n$ 
  have  $|X\ m * Y\ m - X\ n * Y\ n| = |X\ m * (Y\ m - Y\ n) + (X\ m - X\ n) * Y\ n|$ 
n]
    unfolding mult-diff-mult ..
  also have  $\dots \leq |X\ m * (Y\ m - Y\ n)| + |(X\ m - X\ n) * Y\ n|$ 
    by (rule abs-triangle-ineq)
  also have  $\dots = |X\ m| * |Y\ m - Y\ n| + |X\ m - X\ n| * |Y\ n|$ 
    unfolding abs-mult ..
  also have  $\dots < a * t + s * b$ 
    by (simp-all add: add-strict-mono mult-strict-mono' a\ b\ i\ j\ *)
  finally show  $|X\ m * Y\ m - X\ n * Y\ n| < r$ 
    by (simp only: r)
qed
then show  $\exists k. \forall m \geq k. \forall n \geq k. |X\ m * Y\ m - X\ n * Y\ n| < r ..$ 
qed

```

lemma *cauchy-not-vanishes-cases*:

```

  assumes  $X$ : cauchy  $X$ 
  assumes  $nz$ :  $\neg$  vanishes  $X$ 
  shows  $\exists b > 0. \exists k. (\forall n \geq k. b < -\ X\ n) \vee (\forall n \geq k. b < X\ n)$ 
proof -
  obtain  $r$  where  $0 < r$  and  $r: \forall k. \exists n \geq k. r \leq |X\ n|$ 
    using  $nz$  unfolding vanishes-def by (auto simp add: not-less)
  obtain  $s\ t$  where  $s: 0 < s$  and  $t: 0 < t$  and  $r = s + t$ 
    using  $\langle 0 < r \rangle$  by (rule obtain-pos-sum)
  obtain  $i$  where  $i: \forall m \geq i. \forall n \geq i. |X\ m - X\ n| < s$ 
    using cauchyD [OF  $X\ s$ ] ..
  obtain  $k$  where  $i \leq k$  and  $r \leq |X\ k|$ 
    using  $r$  by blast
  have  $k: \forall n \geq k. |X\ n - X\ k| < s$ 
    using  $i\ \langle i \leq k \rangle$  by auto
  have  $X\ k \leq -\ r \vee r \leq X\ k$ 
    using  $\langle r \leq |X\ k| \rangle$  by auto
  then have  $(\forall n \geq k. t < -\ X\ n) \vee (\forall n \geq k. t < X\ n)$ 
    unfolding  $\langle r = s + t \rangle$  using  $k$  by auto
  then have  $\exists k. (\forall n \geq k. t < -\ X\ n) \vee (\forall n \geq k. t < X\ n) ..$ 
  then show  $\exists t > 0. \exists k. (\forall n \geq k. t < -\ X\ n) \vee (\forall n \geq k. t < X\ n)$ 

```

using t by *auto*
qed

lemma *cauchy-not-vanishes*:
assumes X : *cauchy* X
and nz : \neg *vanishes* X
shows $\exists b > 0. \exists k. \forall n \geq k. b < |X\ n|$
using *cauchy-not-vanishes-cases* [*OF* *assms*]
by (*elim ex-forward conj-forward asm-rl*) *auto*

lemma *cauchy-inverse* [*simp*]:
assumes X : *cauchy* X
and nz : \neg *vanishes* X
shows *cauchy* $(\lambda n. \text{inverse } (X\ n))$
proof (*rule cauchyI*)
fix $r :: \text{rat}$
assume $0 < r$
obtain $b\ i$ where $b: 0 < b$ and $i: \forall n \geq i. b < |X\ n|$
using *cauchy-not-vanishes* [*OF* $X\ nz$] by *blast*
from $b\ i$ have $nz: \forall n \geq i. X\ n \neq 0$ by *auto*
obtain s where $s: 0 < s$ and $r: r = \text{inverse } b * s * \text{inverse } b$
proof
show $0 < b * r * b$ by (*simp add: <0 < r> b*)
show $r = \text{inverse } b * (b * r * b) * \text{inverse } b$
using b by *simp*
qed
obtain j where $j: \forall m \geq j. \forall n \geq j. |X\ m - X\ n| < s$
using *cauchyD* [*OF* $X\ s$] ..
have $\forall m \geq \max\ i\ j. \forall n \geq \max\ i\ j. |\text{inverse } (X\ m) - \text{inverse } (X\ n)| < r$
proof *clarsimp*
fix $m\ n$
assume $*: i \leq m\ j \leq m\ i \leq n\ j \leq n$
have $|\text{inverse } (X\ m) - \text{inverse } (X\ n)| = \text{inverse } |X\ m| * |X\ m - X\ n| * \text{inverse } |X\ n|$
by (*simp add: inverse-diff-inverse nz * abs-mult*)
also have $\dots < \text{inverse } b * s * \text{inverse } b$
by (*simp add: mult-strict-mono less-imp-inverse-less i j b * s*)
finally show $|\text{inverse } (X\ m) - \text{inverse } (X\ n)| < r$ by (*simp only: r*)
qed
then show $\exists k. \forall m \geq k. \forall n \geq k. |\text{inverse } (X\ m) - \text{inverse } (X\ n)| < r$..
qed

lemma *vanishes-diff-inverse*:
assumes X : *cauchy* X \neg *vanishes* X
and Y : *cauchy* Y \neg *vanishes* Y
and XY : *vanishes* $(\lambda n. X\ n - Y\ n)$
shows *vanishes* $(\lambda n. \text{inverse } (X\ n) - \text{inverse } (Y\ n))$
proof (*rule vanishesI*)
fix $r :: \text{rat}$

```

assume  $r: 0 < r$ 
obtain  $a\ i$  where  $a: 0 < a$  and  $i: \forall n \geq i. a < |X\ n|$ 
  using cauchy-not-vanishes [OF X] by blast
obtain  $b\ j$  where  $b: 0 < b$  and  $j: \forall n \geq j. b < |Y\ n|$ 
  using cauchy-not-vanishes [OF Y] by blast
obtain  $s$  where  $s: 0 < s$  and  $\text{inverse } a * s * \text{inverse } b = r$ 
proof
  show  $0 < a * r * b$ 
    using a r b by simp
  show  $\text{inverse } a * (a * r * b) * \text{inverse } b = r$ 
    using a r b by simp
qed
obtain  $k$  where  $k: \forall n \geq k. |X\ n - Y\ n| < s$ 
  using vanishesD [OF XY s] ..
have  $\forall n \geq \max(\max\ i\ j)\ k. |\text{inverse } (X\ n) - \text{inverse } (Y\ n)| < r$ 
proof clarsimp
  fix  $n$ 
  assume  $n: i \leq n\ j \leq n\ k \leq n$ 
  with  $i\ j\ a\ b$  have  $X\ n \neq 0$  and  $Y\ n \neq 0$ 
  by auto
  then have  $|\text{inverse } (X\ n) - \text{inverse } (Y\ n)| = \text{inverse } |X\ n| * |X\ n - Y\ n| * \text{inverse } |Y\ n|$ 
    by (simp add: inverse-diff-inverse abs-mult)
  also have  $\dots < \text{inverse } a * s * \text{inverse } b$ 
    by (intro mult-strict-mono' less-imp-inverse-less) (simp-all add: a b i j k n)
  also note  $\langle \text{inverse } a * s * \text{inverse } b = r \rangle$ 
  finally show  $|\text{inverse } (X\ n) - \text{inverse } (Y\ n)| < r$  .
qed
then show  $\exists k. \forall n \geq k. |\text{inverse } (X\ n) - \text{inverse } (Y\ n)| < r$  ..
qed

```

98.4 Equivalence relation on Cauchy sequences

```

definition realrel ::  $(\text{nat} \Rightarrow \text{rat}) \Rightarrow (\text{nat} \Rightarrow \text{rat}) \Rightarrow \text{bool}$ 
  where  $\text{realrel} = (\lambda X\ Y. \text{cauchy } X \wedge \text{cauchy } Y \wedge \text{vanishes } (\lambda n. X\ n - Y\ n))$ 

```

```

lemma realrelI [intro?]:  $\text{cauchy } X \Longrightarrow \text{cauchy } Y \Longrightarrow \text{vanishes } (\lambda n. X\ n - Y\ n) \Longrightarrow \text{realrel } X\ Y$ 
  by (simp add: realrel-def)

```

```

lemma realrel-refl:  $\text{cauchy } X \Longrightarrow \text{realrel } X\ X$ 
  by (simp add: realrel-def)

```

```

lemma symp-realrel:  $\text{symp } \text{realrel}$ 
  by (simp add: abs-minus-commute realrel-def symp-def vanishes-def)

```

```

lemma transp-realrel:  $\text{transp } \text{realrel}$ 
  unfolding realrel-def
  by (rule transpI) (force simp add: dest: vanishes-add)

```


lemma *part-equivp-realrel*: *part-equivp realrel*
by (*blast intro: part-equivpI symp-realrel transp-realrel realrel-refl cauchy-const*)

98.5 The field of real numbers

quotient-type *real* = *nat* \Rightarrow *rat* / *partial*: *realrel*
morphisms *rep-real Real*
by (*rule part-equivp-realrel*)

lemma *cr-real-eq*: *pcr-real* = ($\lambda x y. \text{cauchy } x \wedge \text{Real } x = y$)
unfolding *real.pcr-cr-eq cr-real-def realrel-def* **by** *auto*

lemma *Real-induct* [*induct type: real*]:
assumes $\bigwedge X. \text{cauchy } X \Longrightarrow P (\text{Real } X)$
shows $P x$
proof (*induct x*)
case (*1 X*)
then have *cauchy X* **by** (*simp add: realrel-def*)
then show $P (\text{Real } X)$ **by** (*rule assms*)
qed

lemma *eq-Real*: $\text{cauchy } X \Longrightarrow \text{cauchy } Y \Longrightarrow \text{Real } X = \text{Real } Y \longleftrightarrow \text{vanishes } (\lambda n. X n - Y n)$
using *real.rel-eq-transfer*
unfolding *real.pcr-cr-eq cr-real-def rel-fun-def realrel-def* **by** *simp*

lemma *Domainp-pcr-real* [*transfer-domain-rule*]: *Domainp pcr-real* = *cauchy*
by (*simp add: real.domain-eq realrel-def*)

instantiation *real* :: *field*
begin

lift-definition *zero-real* :: *real* **is** $\lambda n. 0$
by (*simp add: realrel-refl*)

lift-definition *one-real* :: *real* **is** $\lambda n. 1$
by (*simp add: realrel-refl*)

lift-definition *plus-real* :: *real* \Rightarrow *real* \Rightarrow *real* **is** $\lambda X Y n. X n + Y n$
unfolding *realrel-def add-diff-add*
by (*simp only: cauchy-add vanishes-add simp-thms*)

lift-definition *uminus-real* :: *real* \Rightarrow *real* **is** $\lambda X n. - X n$
unfolding *realrel-def minus-diff-minus*
by (*simp only: cauchy-minus vanishes-minus simp-thms*)

lift-definition *times-real* :: *real* \Rightarrow *real* \Rightarrow *real* **is** $\lambda X Y n. X n * Y n$
proof –

```

fix f1 f2 f3 f4
have  $\llbracket$ cauchy f1; cauchy f4; vanishes  $(\lambda n. f1\ n - f2\ n)$ ; vanishes  $(\lambda n. f3\ n - f4\ n)$  $\rrbracket$ 
   $\implies$  vanishes  $(\lambda n. f1\ n * (f3\ n - f4\ n) + f4\ n * (f1\ n - f2\ n))$ 
  by (simp add: vanishes-add vanishes-mult-bounded cauchy-imp-bounded)
then show  $\llbracket$ realrel f1 f2; realrel f3 f4 $\rrbracket \implies$  realrel  $(\lambda n. f1\ n * f3\ n)$   $(\lambda n. f2\ n * f4\ n)$ 
  by (simp add: mult.commute realrel-def mult-diff-mult)
qed

```

```

lift-definition inverse-real :: real  $\implies$  real
is  $\lambda X. \text{if vanishes } X \text{ then } (\lambda n. 0) \text{ else } (\lambda n. \text{inverse } (X\ n))$ 
proof -
fix X Y
assume realrel X Y
then have X: cauchy X and Y: cauchy Y and XY: vanishes  $(\lambda n. X\ n - Y\ n)$ 
  by (simp-all add: realrel-def)
have vanishes X  $\longleftrightarrow$  vanishes Y
proof
  assume vanishes X
  from vanishes-diff [OF this XY] show vanishes Y
  by simp
next
  assume vanishes Y
  from vanishes-add [OF this XY] show vanishes X
  by simp
qed
then show ?thesis X Y
  by (simp add: vanishes-diff-inverse X Y XY realrel-def)
qed

```

definition $x - y = x + -\ y$ **for** $x\ y :: \text{real}$

definition $x \text{ div } y = x * \text{inverse } y$ **for** $x\ y :: \text{real}$

lemma add-Real: cauchy X \implies cauchy Y \implies Real X + Real Y = Real $(\lambda n. X\ n + Y\ n)$
using plus-real.transfer **by** (simp add: cr-real-eq rel-fun-def)

lemma minus-Real: cauchy X \implies $-$ Real X = Real $(\lambda n. -\ X\ n)$
using uminus-real.transfer **by** (simp add: cr-real-eq rel-fun-def)

lemma diff-Real: cauchy X \implies cauchy Y \implies Real X $-$ Real Y = Real $(\lambda n. X\ n - Y\ n)$
by (simp add: minus-Real add-Real minus-real-def)

lemma mult-Real: cauchy X \implies cauchy Y \implies Real X * Real Y = Real $(\lambda n. X\ n * Y\ n)$
using times-real.transfer **by** (simp add: cr-real-eq rel-fun-def)

lemma *inverse-Real*:

cauchy $X \implies \text{inverse} (\text{Real } X) = (\text{if vanishes } X \text{ then } 0 \text{ else } \text{Real } (\lambda n. \text{inverse} (X n)))$

using *inverse-real.transfer zero-real.transfer*

unfolding *cr-real-eq rel-fun-def* **by** (*simp split: if-split-asm, metis*)

instance

proof

fix $a b c :: \text{real}$

show $a + b = b + a$

by *transfer (simp add: ac-simps realrel-def)*

show $(a + b) + c = a + (b + c)$

by *transfer (simp add: ac-simps realrel-def)*

show $0 + a = a$

by *transfer (simp add: realrel-def)*

show $- a + a = 0$

by *transfer (simp add: realrel-def)*

show $a - b = a + - b$

by (*rule minus-real-def*)

show $(a * b) * c = a * (b * c)$

by *transfer (simp add: ac-simps realrel-def)*

show $a * b = b * a$

by *transfer (simp add: ac-simps realrel-def)*

show $1 * a = a$

by *transfer (simp add: ac-simps realrel-def)*

show $(a + b) * c = a * c + b * c$

by *transfer (simp add: distrib-right realrel-def)*

show $(0::\text{real}) \neq (1::\text{real})$

by *transfer (simp add: realrel-def)*

have *vanishes* $(\lambda n. \text{inverse} (X n) * X n - 1)$ **if** X : *cauchy* $X \neg \text{vanishes } X$ **for** X

proof (*rule vanishesI*)

fix $r::\text{rat}$

assume $0 < r$

obtain $b k$ **where** $b > 0 \forall n \geq k. b < |X n|$

using X *cauchy-not-vanishes* **by** *blast*

then show $\exists k. \forall n \geq k. |\text{inverse} (X n) * X n - 1| < r$

using $\langle 0 < r \rangle$ **by** *force*

qed

then show $a \neq 0 \implies \text{inverse } a * a = 1$

by *transfer (simp add: realrel-def)*

show $a \text{ div } b = a * \text{inverse } b$

by (*rule divide-real-def*)

show $\text{inverse} (0::\text{real}) = 0$

by *transfer (simp add: realrel-def)*

qed

end

98.6 Positive reals

lift-definition *positive* :: *real* \Rightarrow *bool*

is $\lambda X. \exists r > 0. \exists k. \forall n \geq k. r < X\ n$

proof –

have *1*: $\exists r > 0. \exists k. \forall n \geq k. r < Y\ n$

if ***: *realrel* *X Y* **and** ****: $\exists r > 0. \exists k. \forall n \geq k. r < X\ n$ **for** *X Y*

proof –

from *** **have** *XY*: *vanishes* ($\lambda n. X\ n - Y\ n$)

by (*simp-all add: realrel-def*)

from **** **obtain** *r i* **where** $0 < r$ **and** *i*: $\forall n \geq i. r < X\ n$

by *blast*

obtain *s t* **where** $s: 0 < s$ **and** $t: 0 < t$ **and** *r*: $r = s + t$

using $\langle 0 < r \rangle$ **by** (*rule obtain-pos-sum*)

obtain *j* **where** *j*: $\forall n \geq j. |X\ n - Y\ n| < s$

using *vanishesD [OF XY s]* ..

have $\forall n \geq \max\ i\ j. t < Y\ n$

proof *clarsimp*

fix *n*

assume *n*: $i \leq n\ j \leq n$

have $|X\ n - Y\ n| < s$ **and** $r < X\ n$

using *i j n* **by** *simp-all*

then show $t < Y\ n$ **by** (*simp add: r*)

qed

then show *?thesis* **using** *t* **by** *blast*

qed

fix *X Y* **assume** *realrel X Y*

then have *realrel X Y* **and** *realrel Y X*

using *symp-realrel* **by** (*auto simp: symp-def*)

then show *?thesis X Y*

by (*safe elim!: 1*)

qed

lemma *positive-Real: cauchy X \implies positive (Real X) \longleftrightarrow ($\exists r > 0. \exists k. \forall n \geq k. r < X\ n$)*

using *positive.transfer* **by** (*simp add: cr-real-eq rel-fun-def*)

lemma *positive-zero: \neg positive 0*

by *transfer auto*

lemma *positive-add:*

assumes *positive x positive y* **shows** *positive (x + y)*

proof –

have ***: $\llbracket \forall n \geq i. a < x\ n; \forall n \geq j. b < y\ n; 0 < a; 0 < b; n \geq \max\ i\ j \rrbracket$

$\implies a + b < x\ n + y\ n$ **for** *x y* **and** *a b::rat* **and** *i j n::nat*

by (*simp add: add-strict-mono*)

show *?thesis*

using *assms*

by *transfer (blast intro: * pos-add-strict)*

qed

lemma *positive-mult*:

assumes *positive x positive y* **shows** *positive (x * y)*

proof –

have *: $\llbracket \forall n \geq i. a < x \ n; \forall n \geq j. b < y \ n; 0 < a; 0 < b; n \geq \max \ i \ j \rrbracket$
 $\implies a * b < x \ n * y \ n$ **for** *x y* **and** *a b::rat* **and** *i j n::nat*

by (*simp add: mult-strict-mono'*)

show *?thesis*

using *assms*

by *transfer (blast intro: * mult-pos-pos)*

qed

lemma *positive-minus*: $\neg \text{positive } x \implies x \neq 0 \implies \text{positive } (- x)$

apply *transfer*

apply (*simp add: realrel-def*)

apply (*blast dest: cauchy-not-vanishes-cases*)

done

instantiation *real :: linordered-field*

begin

definition $x < y \longleftrightarrow \text{positive } (y - x)$

definition $x \leq y \longleftrightarrow x < y \vee x = y$ **for** *x y :: real*

definition $|a| = (\text{if } a < 0 \text{ then } - a \text{ else } a)$ **for** *a :: real*

definition $\text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } - 1)$ **for** *a :: real*

instance

proof

fix *a b c :: real*

show $|a| = (\text{if } a < 0 \text{ then } - a \text{ else } a)$

by (*rule abs-real-def*)

show $a < b \longleftrightarrow a \leq b \wedge \neg b \leq a$

$a \leq b \implies b \leq c \implies a \leq c$ $a \leq a$

$a \leq b \implies b \leq a \implies a = b$

$a \leq b \implies c + a \leq c + b$

unfolding *less-eq-real-def less-real-def*

by (*force simp add: positive-zero dest: positive-add*)+

show $\text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } - 1)$

by (*rule sgn-real-def*)

show $a \leq b \vee b \leq a$

by (*auto dest!: positive-minus simp: less-eq-real-def less-real-def*)

show $a < b \implies 0 < c \implies c * a < c * b$

unfolding *less-real-def*

by (*force simp add: algebra-simps dest: positive-mult*)

qed

end

instantiation *real* :: *distrib-lattice*
begin

definition (*inf* :: *real* \Rightarrow *real* \Rightarrow *real*) = *min*

definition (*sup* :: *real* \Rightarrow *real* \Rightarrow *real*) = *max*

instance

by *standard* (*auto simp add: inf-real-def sup-real-def max-min-distrib2*)

end

lemma *of-nat-Real*: *of-nat* *x* = *Real* ($\lambda n.$ *of-nat* *x*)

by (*induct* *x*) (*simp-all add: zero-real-def one-real-def add-Real*)

lemma *of-int-Real*: *of-int* *x* = *Real* ($\lambda n.$ *of-int* *x*)

by (*cases* *x* *rule: int-diff-cases*) (*simp add: of-nat-Real diff-Real*)

lemma *of-rat-Real*: *of-rat* *x* = *Real* ($\lambda n.$ *x*)

proof (*induct* *x*)

case (*Fract* *a* *b*)

then show ?*case*

apply (*simp add: Fract-of-int-quotient of-rat-divide*)

apply (*simp add: of-int-Real divide-inverse inverse-Real mult-Real*)

done

qed

instance *real* :: *archimedean-field*

proof

show $\exists z. x \leq \text{of-int } z$ **for** *x* :: *real*

proof (*induct* *x*)

case (*1* *X*)

then obtain *b* **where** $0 < b$ **and** $b: \bigwedge n. |X\ n| < b$

by (*blast dest: cauchy-imp-bounded*)

then have *Real* *X* < *of-int* ($\lceil b \rceil + 1$)

using *1*

apply (*simp add: of-int-Real less-real-def diff-Real positive-Real*)

apply (*rule-tac* *x=1* **in** *exI*)

apply (*simp add: algebra-simps*)

by (*metis abs-ge-self le-less-trans le-of-int-ceiling less-le*)

then show ?*case*

using *less-eq-real-def* **by** *blast*

qed

qed

instantiation *real* :: *floor-ceiling*

begin

definition [code del]: $\lfloor x :: \text{real} \rfloor = (\text{THE } z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1))$

instance

proof

show $\text{of-int } \lfloor x \rfloor \leq x \wedge x < \text{of-int } (\lfloor x \rfloor + 1)$ for $x :: \text{real}$
 unfolding floor-real-def using floor-exists1 by (rule theI')

qed

end

98.7 Completeness

lemma not-positive-Real:

assumes *cauchy* X shows $\neg \text{positive } (\text{Real } X) \iff (\forall r > 0. \exists k. \forall n \geq k. X n \leq r)$ (is ?lhs = ?rhs)

unfolding positive-Real [OF *assms*]

proof (intro iffI allI notI impI)

show $\exists k. \forall n \geq k. X n \leq r$ if $r: \neg (\exists r > 0. \exists k. \forall n \geq k. r < X n)$ and $0 < r$ for r

proof –

obtain $s t$ where $s > 0 t > 0 r = s + t$

using $\langle r > 0 \rangle$ obtain-pos-sum by blast

obtain k where $k: \bigwedge m n. \llbracket m \geq k; n \geq k \rrbracket \implies |X m - X n| < t$

using *cauchyD* [OF *assms* $\langle t > 0 \rangle$] by blast

obtain n where $n \geq k X n \leq s$

by (meson $r < 0 < s$) not-less)

then have $X l \leq r$ if $l \geq n$ for l

using k [OF $\langle n \geq k \rangle$, of l] that $\langle r = s + t \rangle$ by linarith

then show ?thesis

by blast

qed

qed (meson le-cases not-le)

lemma le-Real:

assumes *cauchy* X *cauchy* Y

shows $\text{Real } X \leq \text{Real } Y = (\forall r > 0. \exists k. \forall n \geq k. X n \leq Y n + r)$

unfolding not-less [symmetric, where 'a=real] less-real-def

apply (simp add: diff-Real not-positive-Real *assms*)

apply (simp add: diff-le-eq ac-simps)

done

lemma le-RealI:

assumes $Y: \text{cauchy } Y$

shows $\forall n. x \leq \text{of-rat } (Y n) \implies x \leq \text{Real } Y$

proof (induct x)

fix X

assume $X: \text{cauchy } X$ and $\forall n. \text{Real } X \leq \text{of-rat } (Y n)$

then have $le: \bigwedge m r. 0 < r \implies \exists k. \forall n \geq k. X n \leq Y m + r$

by (simp add: of-rat-Real le-Real)

then have $\exists k. \forall n \geq k. X n \leq Y n + r$ **if** $0 < r$ **for** $r :: \text{rat}$
proof –
from that obtain $s t$ **where** $s: 0 < s$ **and** $t: 0 < t$ **and** $r: r = s + t$
by (*rule obtain-pos-sum*)
obtain i **where** $i: \forall m \geq i. \forall n \geq i. |Y m - Y n| < s$
using *cauchyD [OF Y s]* ..
obtain j **where** $j: \forall n \geq j. X n \leq Y i + t$
using *le [OF t]* ..
have $\forall n \geq \max i j. X n \leq Y n + r$
proof *clarsimp*
fix n
assume $n: i \leq n \wedge j \leq n$
have $X n \leq Y i + t$
using $n j$ **by** *simp*
moreover have $|Y i - Y n| < s$
using $n i$ **by** *simp*
ultimately show $X n \leq Y n + r$
unfolding r **by** *simp*
qed
then show *?thesis* ..
qed
then show $\text{Real } X \leq \text{Real } Y$
by (*simp add: of-rat-Real le-Real X Y*)
qed

lemma *Real-leI*:
assumes $X: \text{cauchy } X$
assumes $le: \forall n. \text{of-rat } (X n) \leq y$
shows $\text{Real } X \leq y$
proof –
have $-y \leq -\text{Real } X$
by (*simp add: minus-Real X le-RealI of-rat-minus le*)
then show *?thesis* **by** *simp*
qed

lemma *less-RealD*:
assumes $\text{cauchy } Y$
shows $x < \text{Real } Y \implies \exists n. x < \text{of-rat } (Y n)$
apply (*erule contrapos-pp*)
apply (*simp add: not-less*)
apply (*erule Real-leI [OF assms]*)
done

lemma *of-nat-less-two-power [simp]*: $\text{of-nat } n < (2::'a::\text{linordered-idom}) ^ n$
apply (*induct n*)
apply *simp*
apply (*metis add-le-less-mono mult-2 of-nat-Suc one-le-numeral one-le-power power-Suc*)
done

lemma *complete-real*:

fixes $S :: \text{real set}$

assumes $\exists x. x \in S$ **and** $\exists z. \forall x \in S. x \leq z$

shows $\exists y. (\forall x \in S. x \leq y) \wedge (\forall z. (\forall x \in S. x \leq z) \longrightarrow y \leq z)$

proof –

obtain x **where** $x: x \in S$ **using** *assms(1)* ..

obtain z **where** $z: \forall x \in S. x \leq z$ **using** *assms(2)* ..

define P **where** $P x \longleftrightarrow (\forall y \in S. y \leq \text{of-rat } x)$ **for** x

obtain a **where** $a: \neg P a$

proof

have $\text{of-int } \lfloor x - 1 \rfloor \leq x - 1$ **by** (*rule of-int-floor-le*)

also have $x - 1 < x$ **by** *simp*

finally have $\text{of-int } \lfloor x - 1 \rfloor < x$.

then have $\neg x \leq \text{of-int } \lfloor x - 1 \rfloor$ **by** (*simp only: not-le*)

then show $\neg P (\text{of-int } \lfloor x - 1 \rfloor)$

unfolding *P-def of-rat-of-int-eq* **using** x **by** *blast*

qed

obtain b **where** $b: P b$

proof

show $P (\text{of-int } \lceil z \rceil)$

unfolding *P-def of-rat-of-int-eq*

proof

fix y **assume** $y \in S$

then have $y \leq z$ **using** z **by** *simp*

also have $z \leq \text{of-int } \lceil z \rceil$ **by** (*rule le-of-int-ceiling*)

finally show $y \leq \text{of-int } \lceil z \rceil$.

qed

qed

define *avg* **where** $\text{avg } x y = x/2 + y/2$ **for** $x y :: \text{rat}$

define *bisect* **where** $\text{bisect} = (\lambda(x, y). \text{if } P (\text{avg } x y) \text{ then } (x, \text{avg } x y) \text{ else } (\text{avg } x y, y))$

define A **where** $A n = \text{fst } ((\text{bisect } \hat{\sim} n) (a, b))$ **for** n

define B **where** $B n = \text{snd } ((\text{bisect } \hat{\sim} n) (a, b))$ **for** n

define C **where** $C n = \text{avg } (A n) (B n)$ **for** n

have $A\ 0$ [*simp*]: $A\ 0 = a$ **unfolding** *A-def* **by** *simp*

have $B\ 0$ [*simp*]: $B\ 0 = b$ **unfolding** *B-def* **by** *simp*

have $A\ \text{Suc}$ [*simp*]: $\bigwedge n. A (\text{Suc } n) = (\text{if } P (C n) \text{ then } A n \text{ else } C n)$

unfolding *A-def B-def C-def bisect-def split-def* **by** *simp*

have $B\ \text{Suc}$ [*simp*]: $\bigwedge n. B (\text{Suc } n) = (\text{if } P (C n) \text{ then } C n \text{ else } B n)$

unfolding *A-def B-def C-def bisect-def split-def* **by** *simp*

have *width*: $B n - A n = (b - a) / 2^n$ **for** n

proof (*induct n*)

case (*Suc n*)

then show *?case*

by (*simp add: C-def eq-divide-eq avg-def algebra-simps*)

```

qed simp
have twos:  $\exists n. y / 2^n < r$  if  $0 < r$  for  $y r :: rat$ 
proof -
  obtain  $n$  where  $y / r < rat-of-nat n$ 
  using  $\langle 0 < r \rangle$  reals-Archimedean2 by blast
  then have  $\exists n. y < r * 2^n$ 
  by (metis divide-less-eq less-trans mult.commute of-nat-less-two-power that)
  then show ?thesis
  by (simp add: field-split-simps)
qed
have PA:  $\neg P (A n)$  for  $n$ 
  by (induct  $n$ ) (simp-all add: a)
have PB:  $P (B n)$  for  $n$ 
  by (induct  $n$ ) (simp-all add: b)
have ab:  $a < b$ 
  using a b unfolding P-def
  by (meson leI less-le-trans of-rat-less)
have AB:  $A n < B n$  for  $n$ 
  by (induct  $n$ ) (simp-all add: ab C-def avg-def)
have  $A i \leq A j \wedge B j \leq B i$  if  $i < j$  for  $i j$ 
  using that
proof (induction rule: less-Suc-induct)
  case (1 i)
  then show ?case
  apply (clarsimp simp add: C-def avg-def add-divide-distrib [symmetric])
  apply (rule AB [THEN less-imp-le])
  done
qed simp
then have A-mono:  $A i \leq A j$  and B-mono:  $B j \leq B i$  if  $i \leq j$  for  $i j$ 
  by (metis eq-refl le-neg-implies-less that)+
have cauchy-lemma:  $cauchy X$  if *:  $\bigwedge n i. i \geq n \implies A n \leq X i \wedge X i \leq B n$  for
X
proof (rule cauchyI)
  fix  $r :: rat$ 
  assume  $0 < r$ 
  then obtain  $k$  where  $k: (b - a) / 2^k < r$ 
  using twos by blast
  have  $|X m - X n| < r$  if  $m \geq k$   $n \geq k$  for  $m n$ 
  proof -
    have  $|X m - X n| \leq B k - A k$ 
    by (simp add: * abs-rat-def diff-mono that)
    also have  $\dots < r$ 
    by (simp add: k width)
    finally show ?thesis .
  qed
  then show  $\exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r$ 
  by blast
qed
have cauchy A

```

```

    by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order
less-le-trans)
    have cauchy B
    by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order
le-less-trans)
    have  $\forall x \in S. x \leq \text{Real } B$ 
    proof
      fix x
      assume  $x \in S$ 
      then show  $x \leq \text{Real } B$ 
        using PB [unfolded P-def] ‹cauchy B›
        by (simp add: le-RealI)
    qed
    moreover have  $\forall z. (\forall x \in S. x \leq z) \longrightarrow \text{Real } A \leq z$ 
      by (meson PA Real-leI P-def ‹cauchy A› le-cases order.trans)
    moreover have vanishes ( $\lambda n. (b - a) / 2^n$ )
    proof (rule vanishesI)
      fix r :: rat
      assume  $0 < r$ 
      then obtain k where  $k: |b - a| / 2^k < r$ 
        using twos by blast
      have  $\forall n \geq k. |(b - a) / 2^n| < r$ 
      proof clarify
        fix n
        assume  $n: k \leq n$ 
        have  $|(b - a) / 2^n| = |b - a| / 2^n$ 
          by simp
        also have  $\dots \leq |b - a| / 2^k$ 
          using n by (simp add: divide-left-mono)
        also note k
        finally show  $|(b - a) / 2^n| < r$  .
      qed
      then show  $\exists k. \forall n \geq k. |(b - a) / 2^n| < r$  ..
    qed
    then have  $\text{Real } B = \text{Real } A$ 
      by (simp add: eq-Real ‹cauchy A› ‹cauchy B› width)
    ultimately show  $\exists y. (\forall x \in S. x \leq y) \wedge (\forall z. (\forall x \in S. x \leq z) \longrightarrow y \leq z)$ 
      by force
  qed

```

instantiation *real* :: *linear-continuum*

begin

98.8 Supremum of a set of reals

definition *Sup* $X = (\text{LEAST } z :: \text{real}. \forall x \in X. x \leq z)$

definition *Inf* $X = - \text{Sup} (\text{uminus } X)$ for $X :: \text{real set}$

instance

proof
show *Sup-upper*: $x \leq \text{Sup } X$
if $x \in X$ *bdd-above* X
for $x :: \text{real}$ **and** $X :: \text{real set}$
proof –
from *that* **obtain** s **where** $s: \forall y \in X. y \leq s \wedge z. \forall y \in X. y \leq z \implies s \leq z$
using *complete-real*[*of* X] **unfolding** *bdd-above-def* **by** *blast*
then show *?thesis*
unfolding *Sup-real-def* **by** (*rule LeastI2-order*) (*auto simp: that*)
qed
show *Sup-least*: $\text{Sup } X \leq z$
if $X \neq \{\}$ **and** $z: \bigwedge x. x \in X \implies x \leq z$
for $z :: \text{real}$ **and** $X :: \text{real set}$
proof –
from *that* **obtain** s **where** $s: \forall y \in X. y \leq s \wedge z. \forall y \in X. y \leq z \implies s \leq z$
using *complete-real* [*of* X] **by** *blast*
then have $\text{Sup } X = s$
unfolding *Sup-real-def* **by** (*best intro: Least-equality*)
also from $s \ z$ **have** $\dots \leq z$
by *blast*
finally show *?thesis* .
qed
show *Inf* $X \leq x$ **if** $x \in X$ *bdd-below* X
for $x :: \text{real}$ **and** $X :: \text{real set}$
using *Sup-upper* [*of* $-x$ *uminus* ‘ X ’] **by** (*auto simp: Inf-real-def that*)
show $z \leq \text{Inf } X$ **if** $X \neq \{\}$ $\bigwedge x. x \in X \implies z \leq x$
for $z :: \text{real}$ **and** $X :: \text{real set}$
using *Sup-least* [*of* *uminus* ‘ $X - z$ ’] **by** (*force simp: Inf-real-def that*)
show $\exists a \ b :: \text{real}. a \neq b$
using *zero-neq-one* **by** *blast*
qed
end

98.9 Hiding implementation details

hide-const (*open*) *vanishes* *cauchy* *positive* *Real*

declare *Real-induct* [*induct del*]
declare *Abs-real-induct* [*induct del*]
declare *Abs-real-cases* [*cases del*]

lifting-update *real.lifting*
lifting-forget *real.lifting*

98.10 Embedding numbers into the Reals

abbreviation *real-of-nat* $:: \text{nat} \Rightarrow \text{real}$
where *real-of-nat* \equiv *of-nat*

abbreviation $real :: nat \Rightarrow real$
where $real \equiv of\text{-}nat$

abbreviation $real\text{-}of\text{-}int :: int \Rightarrow real$
where $real\text{-}of\text{-}int \equiv of\text{-}int$

abbreviation $real\text{-}of\text{-}rat :: rat \Rightarrow real$
where $real\text{-}of\text{-}rat \equiv of\text{-}rat$

declare $[[coercion\text{-}enabled]]$

declare $[[coercion\ of\ nat :: nat \Rightarrow int]]$
declare $[[coercion\ of\ nat :: nat \Rightarrow real]]$
declare $[[coercion\ of\ int :: int \Rightarrow real]]$

declare $[[coercion\text{-}map\ map]]$
declare $[[coercion\text{-}map\ \lambda f\ g\ h\ x.\ g\ (h\ (f\ x))]]$
declare $[[coercion\text{-}map\ \lambda f\ g\ (x,y).\ (f\ x,\ g\ y)]]$

declare $of\text{-}int\text{-}eq\text{-}0\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}eq\text{-}1\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}eq\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}less\text{-}0\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}less\text{-}1\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}less\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}le\text{-}0\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}le\text{-}1\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}le\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}0\text{-}less\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}0\text{-}le\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}1\text{-}less\text{-}iff\ [algebra,\ presburger]$
declare $of\text{-}int\text{-}1\text{-}le\text{-}iff\ [algebra,\ presburger]$

lemma $int\text{-}less\text{-}real\text{-}le: n < m \longleftrightarrow real\text{-}of\text{-}int\ n + 1 \leq real\text{-}of\text{-}int\ m$

proof –

have $(0::real) \leq 1$
by $(metis\ less\text{-}eq\text{-}real\text{-}def\ zero\text{-}less\text{-}one)$
then show $?thesis$
by $(metis\ floor\text{-}of\text{-}int\ less\text{-}floor\text{-}iff)$

qed

lemma $int\text{-}le\text{-}real\text{-}less: n \leq m \longleftrightarrow real\text{-}of\text{-}int\ n < real\text{-}of\text{-}int\ m + 1$
by $(meson\ int\text{-}less\text{-}real\text{-}le\ not\text{-}le)$

lemma $real\text{-}of\text{-}int\text{-}div\text{-}aux:$

$(real\text{-}of\text{-}int\ x) / (real\text{-}of\text{-}int\ d) =$
 $real\text{-}of\text{-}int\ (x\ div\ d) + (real\text{-}of\text{-}int\ (x\ mod\ d)) / (real\text{-}of\text{-}int\ d)$

proof –
have $x = (x \text{ div } d) * d + x \text{ mod } d$
by *auto*
then have $\text{real-of-int } x = \text{real-of-int } (x \text{ div } d) * \text{real-of-int } d + \text{real-of-int}(x \text{ mod } d)$
by (*metis of-int-add of-int-mult*)
then have $\text{real-of-int } x / \text{real-of-int } d = \dots / \text{real-of-int } d$
by *simp*
then show *?thesis*
by (*auto simp add: add-divide-distrib algebra-simps*)
qed

lemma *real-of-int-div*:
 $d \text{ dvd } n \implies \text{real-of-int } (n \text{ div } d) = \text{real-of-int } n / \text{real-of-int } d$ **for** $d \ n :: \text{int}$
by (*simp add: real-of-int-div-aux*)

lemma *real-of-int-div2*: $0 \leq \text{real-of-int } n / \text{real-of-int } x - \text{real-of-int } (n \text{ div } x)$
proof (*cases x = 0*)
case *False*
then show *?thesis*
by (*metis diff-ge-0-iff-ge floor-divide-of-int-eq of-int-floor-le*)
qed *simp*

lemma *real-of-int-div3*: $\text{real-of-int } n / \text{real-of-int } x - \text{real-of-int } (n \text{ div } x) \leq 1$
apply (*simp add: algebra-simps*)
by (*metis add.commute floor-correct floor-divide-of-int-eq less-eq-real-def of-int-1 of-int-add*)

lemma *real-of-int-div4*: $\text{real-of-int } (n \text{ div } x) \leq \text{real-of-int } n / \text{real-of-int } x$
using *real-of-int-div2 [of n x]* **by** *simp*

98.11 Embedding the Naturals into the Reals

lemma *real-of-card*: $\text{real } (\text{card } A) = \text{sum } (\lambda x. 1) A$
by *simp*

lemma *nat-less-real-le*: $n < m \iff \text{real } n + 1 \leq \text{real } m$
by (*metis discrete of-nat-1 of-nat-add of-nat-le-iff*)

lemma *nat-le-real-less*: $n \leq m \iff \text{real } n < \text{real } m + 1$
for $m \ n :: \text{nat}$
by (*meson nat-less-real-le not-le*)

lemma *real-of-nat-div-aux*: $\text{real } x / \text{real } d = \text{real } (x \text{ div } d) + \text{real } (x \text{ mod } d) / \text{real } d$

proof –
have $x = (x \text{ div } d) * d + x \text{ mod } d$
by *auto*
then have $\text{real } x = \text{real } (x \text{ div } d) * \text{real } d + \text{real}(x \text{ mod } d)$

by (metis of-nat-add of-nat-mult)
 then have $\text{real } x / \text{real } d = \dots / \text{real } d$
 by simp
 then show ?thesis
 by (auto simp add: add-divide-distrib algebra-simps)
 qed

lemma *real-of-nat-div*: $d \text{ dvd } n \implies \text{real}(n \text{ div } d) = \text{real } n / \text{real } d$
 by (subst *real-of-nat-div-aux*) (auto simp add: dvd-eq-mod-eq-0 [symmetric])

lemma *real-of-nat-div2*: $0 \leq \text{real } n / \text{real } x - \text{real } (n \text{ div } x)$ for $n \ x :: \text{nat}$
 apply (simp add: algebra-simps)
 by (metis floor-divide-of-nat-eq of-int-floor-le of-int-of-nat-eq)

lemma *real-of-nat-div3*: $\text{real } n / \text{real } x - \text{real } (n \text{ div } x) \leq 1$ for $n \ x :: \text{nat}$
 by (metis of-int-of-nat-eq *real-of-int-div3* of-nat-div)

lemma *real-of-nat-div4*: $\text{real } (n \text{ div } x) \leq \text{real } n / \text{real } x$ for $n \ x :: \text{nat}$
 using *real-of-nat-div2* [of $n \ x$] by simp

lemma *real-binomial-eq-mult-binomial-Suc*:
 assumes $k \leq n$
 shows $\text{real}(n \text{ choose } k) = (n + 1 - k) / (n + 1) * (\text{Suc } n \text{ choose } k)$
 using *assms*
 by (simp add: of-nat-binomial-eq-mult-binomial-Suc [of $k \ n$] add.commute of-nat-diff)

98.12 The Archimedean Property of the Reals

lemma *real-arch-inverse*: $0 < e \longleftrightarrow (\exists n :: \text{nat}. n \neq 0 \wedge 0 < \text{inverse } (\text{real } n) \wedge \text{inverse } (\text{real } n) < e)$
 using *reals-Archimedean*[of e] *less-trans*[of $0 \ 1 / \text{real } n \ e$ for $n :: \text{nat}$]
 by (auto simp add: field-simps cong: conj-cong simp del: of-nat-Suc)

lemma *reals-Archimedean3*: $0 < x \implies \forall y. \exists n. y < \text{real } n * x$
 by (auto intro: *ex-less-of-nat-mult*)

lemma *real-archimedian-rdiv-eq-0*:
 assumes $x0: x \geq 0$
 and $c: c \geq 0$
 and $xc: \bigwedge m :: \text{nat}. m > 0 \implies \text{real } m * x \leq c$
 shows $x = 0$
 by (metis *reals-Archimedean3* dual-order.order-iff-strict le0 le-less-trans not-le x0 xc)

lemma *inverse-Suc*: $\text{inverse } (\text{Suc } n) > 0$
 using *of-nat-0-less-iff positive-imp-inverse-positive zero-less-Suc* by blast

lemma *Archimedean-eventually-inverse*:
 fixes $\varepsilon :: \text{real}$ shows $(\forall_F n \text{ in sequentially. } \text{inverse } (\text{real } (\text{Suc } n)) < \varepsilon) \longleftrightarrow 0 < \varepsilon$

ε
 (is ?lhs=?rhs)
proof
 assume ?lhs
 then show ?rhs
 unfolding eventually-at-top-dense using inverse-Suc order-less-trans by blast
next
 assume ?rhs
 then obtain N where $\text{inverse}(\text{Suc } N) < \varepsilon$
 using reals-Archimedean by blast
 moreover have $\text{inverse}(\text{Suc } n) \leq \text{inverse}(\text{Suc } N)$ if $n \geq N$ for n
 using inverse-Suc that by fastforce
 ultimately show ?lhs
 unfolding eventually-sequentially
 using order-le-less-trans by blast
qed

On the relationship between two different ways of converting to 0

lemma *Inter-eq-Inter-inverse-Suc*:
 assumes $\bigwedge r' r. r' < r \implies A r' \subseteq A r$
 shows $\bigcap (A \text{ ‘ } \{0<..\}) = (\bigcap n. A(\text{inverse}(\text{Suc } n)))$
proof
 have $x \in A \varepsilon$
 if $x: \forall n. x \in A (\text{inverse}(\text{Suc } n))$ and $\varepsilon > 0$ for x and $\varepsilon :: \text{real}$
proof –
 obtain n where $\text{inverse}(\text{Suc } n) < \varepsilon$
 using $\langle \varepsilon > 0 \rangle$ reals-Archimedean by blast
 with *assms* x show ?thesis
 by blast
qed
 then show $(\bigcap n. A(\text{inverse}(\text{Suc } n))) \subseteq (\bigcap \varepsilon \in \{0<..\}. A \varepsilon)$
 by *auto*
qed (use *inverse-Suc* in *fastforce*)

98.13 Rationals

lemma *Rats-abs-iff[simp]*:
 $|(x::\text{real})| \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
 by(*simp add: abs-real-def split: if-splits*)

lemma *Rats-eq-int-div-int*: $\mathbb{Q} = \{\text{real-of-int } i / \text{real-of-int } j \mid i j. j \neq 0\}$ (is - = ?S)

proof
 show $\mathbb{Q} \subseteq ?S$
proof
 fix $x :: \text{real}$
 assume $x \in \mathbb{Q}$
 then obtain r where $x = \text{of-rat } r$
 unfolding *Rats-def* ..


```

    have of-rat  $r \in ?S$ 
      by (cases r) (auto simp add: of-rat-rat)
    then show  $x \in ?S$ 
      using ⟨ $x = \text{of-rat } r$ ⟩ by simp
  qed
next
show  $?S \subseteq \mathbf{Q}$ 
proof (auto simp: Rats-def)
  fix  $i j :: \text{int}$ 
  assume  $j \neq 0$ 
  then have  $\text{real-of-int } i / \text{real-of-int } j = \text{of-rat } (\text{Fract } i j)$ 
    by (simp add: of-rat-rat)
  then show  $\text{real-of-int } i / \text{real-of-int } j \in \text{range of-rat}$ 
    by blast
  qed
qed

lemma Rats-eq-int-div-nat:  $\mathbf{Q} = \{ \text{real-of-int } i / \text{real } n \mid i \text{ int. } n \neq 0 \}$ 
proof (auto simp: Rats-eq-int-div-int)
  fix  $i j :: \text{int}$ 
  assume  $j \neq 0$ 
  show  $\exists (i'::\text{int}) (n::\text{nat}). \text{real-of-int } i / \text{real-of-int } j = \text{real-of-int } i' / \text{real } n \wedge 0 < n$ 
  proof (cases  $j > 0$ )
    case True
    then have  $\text{real-of-int } i / \text{real-of-int } j = \text{real-of-int } i / \text{real } (\text{nat } j) \wedge 0 < \text{nat } j$ 
      by simp
    then show ?thesis by blast
  next
  case False
  with ⟨ $j \neq 0$ ⟩
  have  $\text{real-of-int } i / \text{real-of-int } j = \text{real-of-int } (- i) / \text{real } (\text{nat } (- j)) \wedge 0 < \text{nat } (- j)$ 
    by simp
  then show ?thesis by blast
  qed
  qed
next
fix  $i :: \text{int}$  and  $n :: \text{nat}$ 
assume  $0 < n$ 
then have  $\text{real-of-int } i / \text{real } n = \text{real-of-int } i / \text{real-of-int}(\text{int } n) \wedge \text{int } n \neq 0$ 
  by simp
then show  $\exists i' j. \text{real-of-int } i / \text{real } n = \text{real-of-int } i' / \text{real-of-int } j \wedge j \neq 0$ 
  by blast
qed

lemma Rats-abs-nat-div-natE:
  assumes  $x \in \mathbf{Q}$ 
  obtains  $m n :: \text{nat}$  where  $n \neq 0$  and  $|x| = \text{real } m / \text{real } n$  and coprime  $m n$ 
proof -

```

```

from  $\langle x \in \mathbb{Q} \rangle$  obtain  $i :: \text{int}$  and  $n :: \text{nat}$  where  $n \neq 0$  and  $x = \text{real-of-int } i$ 
/  $\text{real } n$ 
  by (auto simp add: Rats-eq-int-div-nat)
then have  $|x| = \text{real } (\text{nat } |i|) / \text{real } n$  by simp
then obtain  $m :: \text{nat}$  where  $x\text{-rat}: |x| = \text{real } m / \text{real } n$  by blast
let  $?gcd = \text{gcd } m \ n$ 
from  $\langle n \neq 0 \rangle$  have  $\text{gcd}: ?gcd \neq 0$  by simp
let  $?k = m \ \text{div } ?gcd$ 
let  $?l = n \ \text{div } ?gcd$ 
let  $?gcd' = \text{gcd } ?k \ ?l$ 
have  $?gcd \ \text{dvd } m \ ..$ 
then have  $\text{gcd-k}: ?gcd * ?k = m$ 
  by (rule dvd-mult-div-cancel)
have  $?gcd \ \text{dvd } n \ ..$ 
then have  $\text{gcd-l}: ?gcd * ?l = n$ 
  by (rule dvd-mult-div-cancel)
from  $\langle n \neq 0 \rangle$  and  $\text{gcd-l}$  have  $?gcd * ?l \neq 0$  by simp
then have  $?l \neq 0$  by (blast dest!: mult-not-zero)
moreover
have  $|x| = \text{real } ?k / \text{real } ?l$ 
proof –
  from  $\text{gcd}$  have  $\text{real } ?k / \text{real } ?l = \text{real } (?gcd * ?k) / \text{real } (?gcd * ?l)$ 
  by (simp add: real-of-nat-div)
  also from  $\text{gcd-k}$  and  $\text{gcd-l}$  have  $\dots = \text{real } m / \text{real } n$  by simp
  also from  $x\text{-rat}$  have  $\dots = |x| \ ..$ 
  finally show  $?thesis \ ..$ 
qed
moreover
have  $?gcd' = 1$ 
proof –
  have  $?gcd * ?gcd' = \text{gcd } (?gcd * ?k) \ (?gcd * ?l)$ 
  by (rule gcd-mult-distrib-nat)
  with  $\text{gcd-k}$   $\text{gcd-l}$  have  $?gcd * ?gcd' = ?gcd$  by simp
  with  $\text{gcd}$  show  $?thesis$  by auto
qed
then have  $\text{coprime } ?k \ ?l$ 
  by (simp only: coprime-iff-gcd-eq-1)
ultimately show  $?thesis \ ..$ 
qed

```

98.14 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is *Real Analysis* by H.L. Royden. It employs the Archimedean property of the reals.

lemma *Rats-dense-in-real:*

fixes $x :: \text{real}$

assumes $x < y$

shows $\exists r \in \mathbb{Q}. x < r \wedge r < y$

proof –
from $\langle x < y \rangle$ **have** $0 < y - x$ **by** *simp*
with *reals-Archimedean* **obtain** $q :: \text{nat}$ **where** $q: \text{inverse}(\text{real } q) < y - x$ **and**
 $0 < q$
by *blast*
define p **where** $p = \lceil y * \text{real } q \rceil - 1$
define r **where** $r = \text{of-int } p / \text{real } q$
from q **have** $x < y - \text{inverse}(\text{real } q)$
by *simp*
also from $\langle 0 < q \rangle$ **have** $y - \text{inverse}(\text{real } q) \leq r$
by (*simp add: r-def p-def le-divide-eq left-diff-distrib*)
finally have $x < r$.
moreover from $\langle 0 < q \rangle$ **have** $r < y$
by (*simp add: r-def p-def divide-less-eq diff-less-eq less-ceiling-iff [symmetric]*)
moreover have $r \in \mathbb{Q}$
by (*simp add: r-def*)
ultimately show *?thesis* **by** *blast*
qed

lemma *of-rat-dense*:
fixes $x y :: \text{real}$
assumes $x < y$
shows $\exists q :: \text{rat}. x < \text{of-rat } q \wedge \text{of-rat } q < y$
using *Rats-dense-in-real [OF $\langle x < y \rangle$]*
by (*auto elim: Rats-cases*)

98.15 Numerals and Arithmetic

declaration \langle
 K (*Lin-Arith.add-inj-const* (**const-name** $\langle \text{of-nat} \rangle$, **typ** $\langle \text{nat} \Rightarrow \text{real} \rangle$)
 $\#>$ *Lin-Arith.add-inj-const* (**const-name** $\langle \text{of-int} \rangle$, **typ** $\langle \text{int} \Rightarrow \text{real} \rangle$))
 \rangle

98.16 Simprules combining $x + y$ and 0

lemma *real-add-minus-iff [simp]*: $x + - a = 0 \longleftrightarrow x = a$
for $x a :: \text{real}$
by *arith*

lemma *real-add-less-0-iff*: $x + y < 0 \longleftrightarrow y < - x$
for $x y :: \text{real}$
by *auto*

lemma *real-0-less-add-iff*: $0 < x + y \longleftrightarrow - x < y$
for $x y :: \text{real}$
by *auto*

lemma *real-add-le-0-iff*: $x + y \leq 0 \longleftrightarrow y \leq - x$
for $x y :: \text{real}$
by *auto*

lemma *real-0-le-add-iff*: $0 \leq x + y \longleftrightarrow -x \leq y$
for $x\ y :: \text{real}$
by *auto*

98.17 Lemmas about powers

lemma *two-realpow-ge-one*: $(1 :: \text{real}) \leq 2 \wedge n$
by *simp*

declare *sum-squares-eq-zero-iff* [*simp*] *sum-power2-eq-zero-iff* [*simp*]

lemma *real-minus-mult-self-le* [*simp*]: $-(u * u) \leq x * x$
for $u\ x :: \text{real}$
by (*rule order-trans* [**where** $y = 0$]) *auto*

lemma *realpow-square-minus-le* [*simp*]: $-u^2 \leq x^2$
for $u\ x :: \text{real}$
by (*auto simp add: power2-eq-square*)

98.18 Density of the Reals

lemma *field-lbound-gt-zero*: $0 < d1 \implies 0 < d2 \implies \exists e. 0 < e \wedge e < d1 \wedge e < d2$
for $d1\ d2 :: 'a::\text{linordered-field}$
by (*rule exI* [**where** $x = \min\ d1\ d2 / 2$]) (*simp add: min-def*)

lemma *field-less-half-sum*: $x < y \implies x < (x + y) / 2$
for $x\ y :: 'a::\text{linordered-field}$
by *auto*

lemma *field-sum-of-halves*: $x / 2 + x / 2 = x$
for $x :: 'a::\text{linordered-field}$
by *simp*

98.19 Archimedean properties and useful consequences

Bernoulli’s inequality

proposition *Bernoulli-inequality*:

fixes $x :: \text{real}$
assumes $-1 \leq x$
shows $1 + n * x \leq (1 + x) \wedge n$

proof (*induct n*)

case 0

then show *?case* **by** *simp*

next

case (*Suc n*)

have $1 + \text{Suc } n * x \leq 1 + (\text{Suc } n) * x + n * x \wedge 2$

```

  by (simp add: algebra-simps)
  also have ... = (1 + x) * (1 + n*x)
  by (auto simp: power2-eq-square algebra-simps)
  also have ... ≤ (1 + x) ^ Suc n
  using Suc.hyps assms mult-left-mono by fastforce
  finally show ?case .
qed

```

corollary *Bernoulli-inequality-even:*

```

  fixes x :: real
  assumes even n
  shows 1 + n * x ≤ (1 + x) ^ n
proof (cases -1 ≤ x ∨ n=0)
  case True
  then show ?thesis
  by (auto simp: Bernoulli-inequality)
next
  case False
  then have real n ≥ 1
  by simp
  with False have n * x ≤ -1
  by (metis linear minus-zero mult.commute mult.left-neutral mult-left-mono-neg
neg-le-iff-le order-trans zero-le-one)
  then have 1 + n * x ≤ 0
  by auto
  also have ... ≤ (1 + x) ^ n
  using assms
  using zero-le-even-power by blast
  finally show ?thesis .
qed

```

corollary *real-arch-pow:*

```

  fixes x :: real
  assumes x: 1 < x
  shows ∃ n. y < x ^ n
proof -
  from x have x0: x - 1 > 0
  by arith
  from reals-Archimedean3[OF x0, rule-format, of y]
  obtain n :: nat where n: y < real n * (x - 1) by metis
  from x0 have x00: x - 1 ≥ -1 by arith
  from Bernoulli-inequality[OF x00, of n] n
  have y < x ^ n by auto
  then show ?thesis by metis
qed

```

corollary *real-arch-pow-inv:*

```

  fixes x y :: real
  assumes y: y > 0

```

and $x1: x < 1$
shows $\exists n. x^{\wedge}n < y$
proof (cases $x > 0$)
case *True*
with $x1$ **have** $ix: 1 < 1/x$ **by** (simp add: field-simps)
from real-arch-pow[OF ix , of $1/y$]
obtain n **where** $n: 1/y < (1/x)^{\wedge}n$ **by** blast
then show ?thesis **using** $y < x > 0$
by (auto simp add: field-simps)
next
case *False*
with y $x1$ **show** ?thesis
by (metis less-le-trans not-less power-one-right)
qed

lemma forall-pos-mono:
 $(\bigwedge d e::real. d < e \implies P d \implies P e) \implies$
 $(\bigwedge n::nat. n \neq 0 \implies P (\text{inverse } (\text{real } n))) \implies (\bigwedge e. 0 < e \implies P e)$
by (metis real-arch-inverse)

lemma forall-pos-mono-1:
 $(\bigwedge d e::real. d < e \implies P d \implies P e) \implies$
 $(\bigwedge n. P (\text{inverse } (\text{real } (\text{Suc } n)))) \implies 0 < e \implies P e$
using reals-Archimedean **by** blast

lemma Archimedean-eventually-pow:
fixes $x::real$
assumes $1 < x$
shows $\forall_F n$ in sequentially. $b < x^{\wedge}n$
proof –
obtain N **where** $\bigwedge n. n \geq N \implies b < x^{\wedge}n$
by (metis assms le-less order-less-trans power-strict-increasing-iff real-arch-pow)
then show ?thesis
using eventually-sequentially **by** blast
qed

lemma Archimedean-eventually-pow-inverse:
fixes $x::real$
assumes $|x| < 1$ $\varepsilon > 0$
shows $\forall_F n$ in sequentially. $|x^{\wedge}n| < \varepsilon$
proof (cases $x = 0$)
case *True*
then show ?thesis
by (simp add: assms eventually-at-top-dense zero-power)
next
case *False*
then have $\forall_F n$ in sequentially. $\text{inverse } \varepsilon < \text{inverse } |x|^{\wedge}n$
by (simp add: Archimedean-eventually-pow assms(1) one-less-inverse)
then show ?thesis

by *eventually-elim* (*metis* $\langle \varepsilon > 0 \rangle$ *inverse-less-imp-less power-abs power-inverse*)
qed

98.20 Floor and Ceiling Functions from the Reals to the Integers

lemma *real-of-nat-less-numeral-iff* [*simp*]: $\text{real } n < \text{numeral } w \longleftrightarrow n < \text{numeral } w$

for $n :: \text{nat}$
by (*metis of-nat-less-iff of-nat-numeral*)

lemma *numeral-less-real-of-nat-iff* [*simp*]: $\text{numeral } w < \text{real } n \longleftrightarrow \text{numeral } w < n$

for $n :: \text{nat}$
by (*metis of-nat-less-iff of-nat-numeral*)

lemma *numeral-le-real-of-nat-iff* [*simp*]: $\text{numeral } n \leq \text{real } m \longleftrightarrow \text{numeral } n \leq m$

for $m :: \text{nat}$
by (*metis not-le real-of-nat-less-numeral-iff*)

lemma *of-int-floor-cancel* [*simp*]: $\text{of-int } \lfloor x \rfloor = x \longleftrightarrow (\exists n :: \text{int}. x = \text{of-int } n)$

by (*metis floor-of-int*)

lemma *of-int-floor* [*simp*]: $a \in \mathbb{Z} \implies \text{of-int } (\text{floor } a) = a$

by (*metis Ints-cases of-int-floor-cancel*)

lemma *floor-frac* [*simp*]: $\lfloor \text{frac } r \rfloor = 0$

by (*simp add: frac-def*)

lemma *frac-1* [*simp*]: $\text{frac } 1 = 0$

by (*simp add: frac-def*)

lemma *frac-in-Rats-iff* [*simp*]:

fixes $r :: 'a :: \{\text{floor-ceiling, field-char-0}\}$

shows $\text{frac } r \in \mathbb{Q} \longleftrightarrow r \in \mathbb{Q}$

by (*metis Rats-add Rats-diff Rats-of-int diff-add-cancel frac-def*)

lemma *floor-eq*: $\text{real-of-int } n < x \implies x < \text{real-of-int } n + 1 \implies \lfloor x \rfloor = n$

by *linarith*

lemma *floor-eq2*: $\text{real-of-int } n \leq x \implies x < \text{real-of-int } n + 1 \implies \lfloor x \rfloor = n$

by (*fact floor-unique*)

lemma *floor-eq3*: $\text{real } n < x \implies x < \text{real } (\text{Suc } n) \implies \text{nat } \lfloor x \rfloor = n$

by *linarith*

lemma *floor-eq4*: $\text{real } n \leq x \implies x < \text{real } (\text{Suc } n) \implies \text{nat } \lfloor x \rfloor = n$

by *linarith*

lemma *real-of-int-floor-ge-diff-one* [*simp*]: $r - 1 \leq \text{real-of-int } \lfloor r \rfloor$
by *linarith*

lemma *real-of-int-floor-gt-diff-one* [*simp*]: $r - 1 < \text{real-of-int } \lfloor r \rfloor$
by *linarith*

lemma *real-of-int-floor-add-one-ge* [*simp*]: $r \leq \text{real-of-int } \lfloor r \rfloor + 1$
by *linarith*

lemma *real-of-int-floor-add-one-gt* [*simp*]: $r < \text{real-of-int } \lfloor r \rfloor + 1$
by *linarith*

lemma *floor-divide-real-eq-div*:

assumes $0 \leq b$

shows $\lfloor a / \text{real-of-int } b \rfloor = \lfloor a \rfloor \text{ div } b$

proof (*cases* $b = 0$)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

with *assms* **have** $b > 0$ **by** *simp*

have $j = i \text{ div } b$

if $\text{real-of-int } i \leq a$ $a < 1 + \text{real-of-int } i$

$\text{real-of-int } j * \text{real-of-int } b \leq a$ $a < \text{real-of-int } b + \text{real-of-int } j * \text{real-of-int } b$

for $i j :: \text{int}$

proof –

from *that* **have** $i < b + j * b$

by (*metis* *le-less-trans* *of-int-add* *of-int-less-iff* *of-int-mult*)

moreover **have** $j * b < 1 + i$

proof –

have $\text{real-of-int } (j * b) < \text{real-of-int } i + 1$

using $\langle a < 1 + \text{real-of-int } i \rangle \langle \text{real-of-int } j * \text{real-of-int } b \leq a \rangle$ **by** *force*

then show $j * b < 1 + i$ **by** *linarith*

qed

ultimately **have** $(j - i \text{ div } b) * b \leq i \text{ mod } b$ $i \text{ mod } b < ((j - i \text{ div } b) + 1) * b$

by (*auto* *simp*: *field-simps*)

then **have** $(j - i \text{ div } b) * b < 1 * b$ $0 * b < ((j - i \text{ div } b) + 1) * b$

using *pos-mod-bound* [*OF* b , *of* i] *pos-mod-sign* [*OF* b , *of* i]

by *linarith+*

then show *?thesis* **using** b **unfolding** *mult-less-cancel-right* **by** *auto*

qed

with b **show** *?thesis* **by** (*auto* *split*: *floor-split* *simp*: *field-simps*)

qed

lemma *floor-one-divide-eq-div-numeral* [*simp*]:

$\lfloor 1 / \text{numeral } b :: \text{real} \rfloor = 1 \text{ div numeral } b$

by (*metis* *floor-divide-of-int-eq-of-int-1* *of-int-numeral*)

lemma *floor-minus-one-divide-eq-div-numeral* [*simp*]:

$\lfloor - (1 / \text{numeral } b)::\text{real} \rfloor = - 1 \text{ div numeral } b$
by (*metis* (*mono-tags*, *opaque-lifting*) *div-minus-right minus-divide-right*
floor-divide-of-int-eq of-int-neg-numeral of-int-1)

lemma *floor-divide-eq-div-numeral* [*simp*]:
 $\lfloor \text{numeral } a / \text{numeral } b::\text{real} \rfloor = \text{numeral } a \text{ div numeral } b$
by (*metis* *floor-divide-of-int-eq of-int-numeral*)

lemma *floor-minus-divide-eq-div-numeral* [*simp*]:
 $\lfloor - (\text{numeral } a / \text{numeral } b)::\text{real} \rfloor = - \text{numeral } a \text{ div numeral } b$
by (*metis* *divide-minus-left floor-divide-of-int-eq of-int-neg-numeral of-int-numeral*)

lemma *of-int-ceiling-cancel* [*simp*]: $\text{of-int } \lceil x \rceil = x \iff (\exists n::\text{int}. x = \text{of-int } n)$
using *ceiling-of-int* **by** *metis*

lemma *of-int-ceiling* [*simp*]: $a \in \mathbb{Z} \implies \text{of-int } (\text{ceiling } a) = a$
by (*metis* *Ints-cases of-int-ceiling-cancel*)

lemma *ceiling-eq*: $\text{of-int } n < x \implies x \leq \text{of-int } n + 1 \implies \lceil x \rceil = n + 1$
by (*simp* *add: ceiling-unique*)

lemma *of-int-ceiling-diff-one-le* [*simp*]: $\text{of-int } \lceil r \rceil - 1 \leq r$
by *linarith*

lemma *of-int-ceiling-le-add-one* [*simp*]: $\text{of-int } \lceil r \rceil \leq r + 1$
by *linarith*

lemma *ceiling-le*: $x \leq \text{of-int } a \implies \lceil x \rceil \leq a$
by (*simp* *add: ceiling-le-iff*)

lemma *ceiling-divide-eq-div*: $\lceil \text{of-int } a / \text{of-int } b \rceil = - (- a \text{ div } b)$
by (*metis* *ceiling-def floor-divide-of-int-eq minus-divide-left of-int-minus*)

lemma *ceiling-divide-eq-div-numeral* [*simp*]:
 $\lceil \text{numeral } a / \text{numeral } b :: \text{real} \rceil = - (- \text{numeral } a \text{ div numeral } b)$
using *ceiling-divide-eq-div*[*of numeral a numeral b*] **by** *simp*

lemma *ceiling-minus-divide-eq-div-numeral* [*simp*]:
 $\lceil - (\text{numeral } a / \text{numeral } b :: \text{real}) \rceil = - (\text{numeral } a \text{ div numeral } b)$
using *ceiling-divide-eq-div*[*of - numeral a numeral b*] **by** *simp*

The following lemmas are remnants of the erstwhile functions *natfloor* and *natceiling*.

lemma *nat-floor-neg*: $x \leq 0 \implies \text{nat } \lfloor x \rfloor = 0$
for $x :: \text{real}$
by *linarith*

lemma *le-nat-floor*: $\text{real } x \leq a \implies x \leq \text{nat } \lfloor a \rfloor$
by *linarith*

lemma *le-mult-nat-floor*: $\text{nat } \lfloor a \rfloor * \text{nat } \lfloor b \rfloor \leq \text{nat } \lfloor a * b \rfloor$
by (*cases* $0 \leq a \wedge 0 \leq b$)
(auto simp add: nat-mult-distrib[symmetric] nat-mono le-mult-floor)

lemma *nat-ceiling-le-eq* [*simp*]: $\text{nat } \lceil x \rceil \leq a \iff x \leq \text{real } a$
by *linarith*

lemma *real-nat-ceiling-ge*: $x \leq \text{real } (\text{nat } \lceil x \rceil)$
by *linarith*

lemma *Rats-no-top-le*: $\exists q \in \mathbf{Q}. x \leq q$
for $x :: \text{real}$
by (*auto intro!*: *beI*[*of* - *of-nat* ($\text{nat } \lceil x \rceil$)]) *linarith*

lemma *Rats-no-bot-less*: $\exists q \in \mathbf{Q}. q < x$ **for** $x :: \text{real}$
by (*auto intro!*: *beI*[*of* - *of-int* ($\lfloor x \rfloor - 1$)]) *linarith*

98.21 Exponentiation with floor

lemma *floor-power*:
assumes $x = \text{of-int } \lfloor x \rfloor$
shows $\lfloor x \wedge n \rfloor = \lfloor x \rfloor \wedge n$
proof –
have $x \wedge n = \text{of-int } (\lfloor x \rfloor \wedge n)$
using *assms* **by** (*induct* n *arbitrary*: x) *simp-all*
then show *?thesis* **by** (*metis* *floor-of-int*)
qed

lemma *floor-numeral-power* [*simp*]: $\lfloor \text{numeral } x \wedge n \rfloor = \text{numeral } x \wedge n$
by (*metis* *floor-of-int of-int-numeral of-int-power*)

lemma *ceiling-numeral-power* [*simp*]: $\lceil \text{numeral } x \wedge n \rceil = \text{numeral } x \wedge n$
by (*metis* *ceiling-of-int of-int-numeral of-int-power*)

98.22 Implementation of rational real numbers

Formal constructor

definition *Ratreal* :: $\text{rat} \Rightarrow \text{real}$
where [*code-abbrev*, *simp*]: *Ratreal* = *real-of-rat*

code-datatype *Ratreal*

Quasi-Numerals

lemma [*code-abbrev*]:
real-of-rat (*numeral* k) = *numeral* k
real-of-rat ($-$ *numeral* k) = $-$ *numeral* k
real-of-rat (*rat-of-int* a) = *real-of-int* a
by *simp-all*

lemma [code-post]:

real-of-rat 0 = 0

real-of-rat 1 = 1

real-of-rat (- 1) = - 1

real-of-rat (1 / numeral k) = 1 / numeral k

real-of-rat (numeral k / numeral l) = numeral k / numeral l

real-of-rat (- (1 / numeral k)) = - (1 / numeral k)

real-of-rat (- (numeral k / numeral l)) = - (numeral k / numeral l)

by (*simp-all add: of-rat-divide of-rat-minus*)

Operations

lemma *zero-real-code* [code]: $0 = \text{Ratreal } 0$

by *simp*

lemma *one-real-code* [code]: $1 = \text{Ratreal } 1$

by *simp*

instantiation *real* :: *equal*

begin

definition *HOL.equal* $x y \longleftrightarrow x - y = 0$ **for** $x :: \text{real}$

instance **by** *standard* (*simp add: equal-real-def*)

lemma *real-equal-code* [code]: $\text{HOL.equal } (\text{Ratreal } x) (\text{Ratreal } y) \longleftrightarrow \text{HOL.equal } x$
 y

by (*simp add: equal-real-def equal*)

lemma [code nbe]: $\text{HOL.equal } x x \longleftrightarrow \text{True}$

for $x :: \text{real}$

by (*rule equal-refl*)

end

lemma *real-less-eq-code* [code]: $\text{Ratreal } x \leq \text{Ratreal } y \longleftrightarrow x \leq y$

by (*simp add: of-rat-less-eq*)

lemma *real-less-code* [code]: $\text{Ratreal } x < \text{Ratreal } y \longleftrightarrow x < y$

by (*simp add: of-rat-less*)

lemma *real-plus-code* [code]: $\text{Ratreal } x + \text{Ratreal } y = \text{Ratreal } (x + y)$

by (*simp add: of-rat-add*)

lemma *real-times-code* [code]: $\text{Ratreal } x * \text{Ratreal } y = \text{Ratreal } (x * y)$

by (*simp add: of-rat-mult*)

lemma *real-uminus-code* [code]: $-\text{Ratreal } x = \text{Ratreal } (- x)$

by (*simp add: of-rat-minus*)

lemma *real-minus-code* [code]: $\text{Ratreal } x - \text{Ratreal } y = \text{Ratreal } (x - y)$
by (*simp add: of-rat-diff*)

lemma *real-inverse-code* [code]: $\text{inverse } (\text{Ratreal } x) = \text{Ratreal } (\text{inverse } x)$
by (*simp add: of-rat-inverse*)

lemma *real-divide-code* [code]: $\text{Ratreal } x / \text{Ratreal } y = \text{Ratreal } (x / y)$
by (*simp add: of-rat-divide*)

lemma *real-floor-code* [code]: $\lfloor \text{Ratreal } x \rfloor = \lfloor x \rfloor$
by (*metis Ratreal-def floor-le-iff floor-unique le-floor-iff
of-int-floor-le of-rat-of-int-eq real-less-eq-code*)

Quickcheck

context
includes *term-syntax*
begin

definition
valterm-ratreal :: $\text{rat} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow \text{real} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term})$
where [code-unfold]: *valterm-ratreal* $k = \text{Code-Evaluation.valtermify } \text{Ratreal } \{ \cdot \}$
k

end

instantiation *real* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition
Quickcheck-Random.random $i = \text{Quickcheck-Random.random } i \circ \rightarrow (\lambda r. \text{Pair } (\text{valterm-ratreal } r))$

instance ..

end

end

instantiation *real* :: *exhaustive*
begin

definition
exhaustive-real $f d = \text{Quickcheck-Exhaustive.exhaustive } (\lambda r. f (\text{Ratreal } r)) d$

instance ..

end

instantiation *real* :: *full-exhaustive*
begin

definition

full-exhaustive-real *f d* = *Quickcheck-Exhaustive.full-exhaustive* ($\lambda r. f$ (*valterm-ratreal* *r*)) *d*

instance ..

end

instantiation *real* :: *narrowing*
begin

definition

narrowing-real = *Quickcheck-Narrowing.apply* (*Quickcheck-Narrowing.cons* *Ra-treal*) *narrowing*

instance ..

end

98.23 Setup for Nitpick

declaration \langle

Nitpick-HOL.register-frac-type **type-name** \langle *real* \rangle
 [(**const-name** \langle *zero-real-inst.zero-real* \rangle , **const-name** \langle *Nitpick.zero-frac* \rangle),
 (**const-name** \langle *one-real-inst.one-real* \rangle , **const-name** \langle *Nitpick.one-frac* \rangle),
 (**const-name** \langle *plus-real-inst.plus-real* \rangle , **const-name** \langle *Nitpick.plus-frac* \rangle),
 (**const-name** \langle *times-real-inst.times-real* \rangle , **const-name** \langle *Nitpick.times-frac* \rangle),
 (**const-name** \langle *uminus-real-inst.uminus-real* \rangle , **const-name** \langle *Nitpick.uminus-frac* \rangle),
 (**const-name** \langle *inverse-real-inst.inverse-real* \rangle , **const-name** \langle *Nitpick.inverse-frac* \rangle),
 (**const-name** \langle *ord-real-inst.less-real* \rangle , **const-name** \langle *Nitpick.less-frac* \rangle),
 (**const-name** \langle *ord-real-inst.less-eq-real* \rangle , **const-name** \langle *Nitpick.less-eq-frac* \rangle)]

\rangle

lemmas [*nitpick-unfold*] = *inverse-real-inst.inverse-real one-real-inst.one-real*
ord-real-inst.less-real ord-real-inst.less-eq-real plus-real-inst.plus-real
times-real-inst.times-real uminus-real-inst.uminus-real
zero-real-inst.zero-real

98.24 Setup for SMT

ML-file \langle *Tools/SMT/smt-real.ML* \rangle

ML-file \langle *Tools/SMT/z3-real.ML* \rangle

lemma [z3-rule]:

```

0 + x = x
x + 0 = x
0 * x = 0
1 * x = x
-x = -1 * x
x + y = y + x
for x y :: real
by auto

```

lemma [smt-arith-multiplication]:

```

fixes A B :: real and p n :: real
assumes A ≤ B 0 < n p > 0
shows (A / n) * p ≤ (B / n) * p
using assms by (auto simp: field-simps)

```

lemma [smt-arith-multiplication]:

```

fixes A B :: real and p n :: real
assumes A < B 0 < n p > 0
shows (A / n) * p < (B / n) * p
using assms by (auto simp: field-simps)

```

lemma [smt-arith-multiplication]:

```

fixes A B :: real and p n :: int
assumes A ≤ B 0 < n p > 0
shows (A / n) * p ≤ (B / n) * p
using assms by (auto simp: field-simps)

```

lemma [smt-arith-multiplication]:

```

fixes A B :: real and p n :: int
assumes A < B 0 < n p > 0
shows (A / n) * p < (B / n) * p
using assms by (auto simp: field-simps)

```

lemmas [smt-arith-multiplication] =

```

  verit-le-mono-div[THEN mult-left-mono, unfolded int-distrib, of - - ⟨nat (floor (-
  :: real))⟩ ⟨nat (floor (- :: real))⟩]
  div-le-mono[THEN mult-left-mono, unfolded int-distrib, of - - ⟨nat (floor (- ::
  real))⟩ ⟨nat (floor (- :: real))⟩]
  verit-le-mono-div-int[THEN mult-left-mono, unfolded int-distrib, of - - ⟨floor (-
  :: real)⟩ ⟨floor (- :: real)⟩]
  zdiv-mono1[THEN mult-left-mono, unfolded int-distrib, of - - ⟨floor (- :: real)⟩
  ⟨floor (- :: real)⟩]
  arg-cong[of - - ⟨ $\lambda a :: \text{real}. a / \text{real } (n::\text{nat}) * \text{real } (p::\text{nat})$ ⟩ for n p :: nat, THEN
  sym]
  arg-cong[of - - ⟨ $\lambda a :: \text{real}. a / \text{real-of-int } n * \text{real-of-int } p$ ⟩ for n p :: int, THEN
  sym]
  arg-cong[of - - ⟨ $\lambda a :: \text{real}. a / n * p$ ⟩ for n p :: real, THEN sym]

```

```

lemmas [smt-arith-simplify] =
  floor-one floor-numeral div-by-1 times-divide-eq-right
  nonzero-mult-div-cancel-left division-ring-divide-zero div-0
  divide-minus-left zero-less-divide-iff

```

98.25 Setup for Argo

ML-file \langle Tools/Argo/argo-real.ML \rangle

end

99 Topological Spaces

theory *Topological-Spaces*

imports *Main*

begin

named-theorems *continuous-intros structural introduction rules for continuity*

99.1 Topological space

class *open* =

fixes *open* :: 'a set \Rightarrow bool

class *topological-space* = *open* +

assumes *open-UNIV* [*simp*, *intro*]: *open UNIV*

assumes *open-Int* [*intro*]: *open S* \Longrightarrow *open T* \Longrightarrow *open (S \cap T)*

assumes *open-Union* [*intro*]: $\forall S \in K. \textit{open } S \Longrightarrow \textit{open } (\bigcup K)$

begin

definition *closed* :: 'a set \Rightarrow bool

where *closed S* \longleftrightarrow *open (- S)*

lemma *open-empty* [*continuous-intros*, *intro*, *simp*]: *open {}*

using *open-Union* [*of {}*] **by** *simp*

lemma *open-Un* [*continuous-intros*, *intro*]: *open S* \Longrightarrow *open T* \Longrightarrow *open (S \cup T)*

using *open-Union* [*of {S, T}*] **by** *simp*

lemma *open-UN* [*continuous-intros*, *intro*]: $\forall x \in A. \textit{open } (B \ x) \Longrightarrow \textit{open } (\bigcup_{x \in A} B \ x)$

using *open-Union* [*of B ' A*] **by** *simp*

lemma *open-Inter* [*continuous-intros*, *intro*]: *finite S* \Longrightarrow $\forall T \in S. \textit{open } T \Longrightarrow \textit{open } (\bigcap S)$

by (*induction set: finite*) *auto*

lemma *open-INT* [*continuous-intros, intro*]: $\text{finite } A \implies \forall x \in A. \text{open } (B x) \implies \text{open } (\bigcap_{x \in A} B x)$

using *open-Inter* [*of B ‘ A*] **by** *simp*

lemma *openI*:

assumes $\bigwedge x. x \in S \implies \exists T. \text{open } T \wedge x \in T \wedge T \subseteq S$

shows *open S*

proof –

have *open* $(\bigcup \{T. \text{open } T \wedge T \subseteq S\})$ **by** *auto*

moreover have $\bigcup \{T. \text{open } T \wedge T \subseteq S\} = S$ **by** (*auto dest!: assms*)

ultimately show *open S* **by** *simp*

qed

lemma *open-subopen*: $\text{open } S \iff (\forall x \in S. \exists T. \text{open } T \wedge x \in T \wedge T \subseteq S)$

by (*auto intro: openI*)

lemma *closed-empty* [*continuous-intros, intro, simp*]: *closed* $\{\}$

unfolding *closed-def* **by** *simp*

lemma *closed-Un* [*continuous-intros, intro*]: $\text{closed } S \implies \text{closed } T \implies \text{closed } (S \cup T)$

unfolding *closed-def* **by** *auto*

lemma *closed-UNIV* [*continuous-intros, intro, simp*]: *closed* *UNIV*

unfolding *closed-def* **by** *simp*

lemma *closed-Int* [*continuous-intros, intro*]: $\text{closed } S \implies \text{closed } T \implies \text{closed } (S \cap T)$

unfolding *closed-def* **by** *auto*

lemma *closed-INT* [*continuous-intros, intro*]: $\forall x \in A. \text{closed } (B x) \implies \text{closed } (\bigcap_{x \in A} B x)$

unfolding *closed-def* **by** *auto*

lemma *closed-Inter* [*continuous-intros, intro*]: $\forall S \in K. \text{closed } S \implies \text{closed } (\bigcap K)$

unfolding *closed-def* *uminus-Inf* **by** *auto*

lemma *closed-Union* [*continuous-intros, intro*]: $\text{finite } S \implies \forall T \in S. \text{closed } T \implies \text{closed } (\bigcup S)$

by (*induct set: finite*) *auto*

lemma *closed-UN* [*continuous-intros, intro*]:

$\text{finite } A \implies \forall x \in A. \text{closed } (B x) \implies \text{closed } (\bigcup_{x \in A} B x)$

using *closed-Union* [*of B ‘ A*] **by** *simp*

lemma *open-closed*: $\text{open } S \iff \text{closed } (- S)$

by (*simp add: closed-def*)

lemma *closed-open*: $\text{closed } S \iff \text{open } (- S)$

by (*rule closed-def*)

lemma *open-Diff* [*continuous-intros, intro*]: $open\ S \implies closed\ T \implies open\ (S - T)$

by (*simp add: closed-open Diff-eq open-Int*)

lemma *closed-Diff* [*continuous-intros, intro*]: $closed\ S \implies open\ T \implies closed\ (S - T)$

by (*simp add: open-closed Diff-eq closed-Int*)

lemma *open-Compl* [*continuous-intros, intro*]: $closed\ S \implies open\ (-\ S)$

by (*simp add: closed-open*)

lemma *closed-Compl* [*continuous-intros, intro*]: $open\ S \implies closed\ (-\ S)$

by (*simp add: open-closed*)

lemma *open-Collect-neg*: $closed\ \{x.\ P\ x\} \implies open\ \{x.\ \neg\ P\ x\}$

unfolding *Collect-neg-eq* **by** (*rule open-Compl*)

lemma *open-Collect-conj*:

assumes $open\ \{x.\ P\ x\}$ $open\ \{x.\ Q\ x\}$

shows $open\ \{x.\ P\ x \wedge Q\ x\}$

using *open-Int[OF assms]* **by** (*simp add: Int-def*)

lemma *open-Collect-disj*:

assumes $open\ \{x.\ P\ x\}$ $open\ \{x.\ Q\ x\}$

shows $open\ \{x.\ P\ x \vee Q\ x\}$

using *open-Un[OF assms]* **by** (*simp add: Un-def*)

lemma *open-Collect-ex*: $(\bigwedge i.\ open\ \{x.\ P\ i\ x\}) \implies open\ \{x.\ \exists i.\ P\ i\ x\}$

using *open-UN[of UNIV $\lambda i.\ \{x.\ P\ i\ x\}$]* **unfolding** *Collect-ex-eq* **by** *simp*

lemma *open-Collect-imp*: $closed\ \{x.\ P\ x\} \implies open\ \{x.\ Q\ x\} \implies open\ \{x.\ P\ x \longrightarrow Q\ x\}$

unfolding *imp-conv-disj* **by** (*intro open-Collect-disj open-Collect-neg*)

lemma *open-Collect-const*: $open\ \{x.\ P\}$

by (*cases P*) *auto*

lemma *closed-Collect-neg*: $open\ \{x.\ P\ x\} \implies closed\ \{x.\ \neg\ P\ x\}$

unfolding *Collect-neg-eq* **by** (*rule closed-Compl*)

lemma *closed-Collect-conj*:

assumes $closed\ \{x.\ P\ x\}$ $closed\ \{x.\ Q\ x\}$

shows $closed\ \{x.\ P\ x \wedge Q\ x\}$

using *closed-Int[OF assms]* **by** (*simp add: Int-def*)

lemma *closed-Collect-disj*:

assumes $closed\ \{x.\ P\ x\}$ $closed\ \{x.\ Q\ x\}$

shows *closed* $\{x. P x \vee Q x\}$
using *closed-Un*[*OF assms*] **by** (*simp add: Un-def*)

lemma *closed-Collect-all*: $(\bigwedge i. \text{closed } \{x. P i x\}) \implies \text{closed } \{x. \forall i. P i x\}$
using *closed-INT*[*of UNIV* $\lambda i. \{x. P i x\}$] **by** (*simp add: Collect-all-eq*)

lemma *closed-Collect-imp*: $\text{open } \{x. P x\} \implies \text{closed } \{x. Q x\} \implies \text{closed } \{x. P x \longrightarrow Q x\}$
unfolding *imp-conv-disj* **by** (*intro closed-Collect-disj closed-Collect-neg*)

lemma *closed-Collect-const*: $\text{closed } \{x. P\}$
by (*cases P*) *auto*

end

99.2 Hausdorff and other separation properties

class *t0-space* = *topological-space* +
assumes *t0-space*: $x \neq y \implies \exists U. \text{open } U \wedge \neg (x \in U \longleftrightarrow y \in U)$

class *t1-space* = *topological-space* +
assumes *t1-space*: $x \neq y \implies \exists U. \text{open } U \wedge x \in U \wedge y \notin U$

instance *t1-space* \subseteq *t0-space*
by *standard* (*fast dest: t1-space*)

context *t1-space* **begin**

lemma *separation-t1*: $x \neq y \longleftrightarrow (\exists U. \text{open } U \wedge x \in U \wedge y \notin U)$
using *t1-space*[*of x y*] **by** *blast*

lemma *closed-singleton* [*iff*]: $\text{closed } \{a\}$
proof –
let $?T = \bigcup \{S. \text{open } S \wedge a \notin S\}$
have *open* $?T$
by (*simp add: open-Union*)
also have $?T = - \{a\}$
by (*auto simp add: set-eq-iff separation-t1*)
finally show $\text{closed } \{a\}$
by (*simp only: closed-def*)

qed

lemma *closed-insert* [*continuous-intros, simp*]:
assumes *closed S*
shows *closed* (*insert a S*)

proof –
from *closed-singleton assms* **have** $\text{closed } (\{a\} \cup S)$
by (*rule closed-Un*)
then show *closed* (*insert a S*)

by *simp*
qed

lemma *finite-imp-closed*: $finite\ S \implies closed\ S$
by (*induct pred: finite*) *simp-all*

end

T2 spaces are also known as Hausdorff spaces.

class *t2-space* = *topological-space* +
assumes *hausdorff*: $x \neq y \implies \exists U\ V. open\ U \wedge open\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$

instance *t2-space* \subseteq *t1-space*
by *standard* (*fast dest: hausdorff*)

lemma (*in t2-space*) *separation-t2*: $x \neq y \longleftrightarrow (\exists U\ V. open\ U \wedge open\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\})$
using *hausdorff* [*of x y*] **by** *blast*

lemma (*in t0-space*) *separation-t0*: $x \neq y \longleftrightarrow (\exists U. open\ U \wedge \neg (x \in U \longleftrightarrow y \in U))$
using *t0-space* [*of x y*] **by** *blast*

A classical separation axiom for topological space, the T3 axiom – also called regularity: if a point is not in a closed set, then there are open sets separating them.

class *t3-space* = *t2-space* +
assumes *t3-space*: $closed\ S \implies y \notin S \implies \exists U\ V. open\ U \wedge open\ V \wedge y \in U \wedge S \subseteq V \wedge U \cap V = \{\}$

A classical separation axiom for topological space, the T4 axiom – also called normality: if two closed sets are disjoint, then there are open sets separating them.

class *t4-space* = *t2-space* +
assumes *t4-space*: $closed\ S \implies closed\ T \implies S \cap T = \{\} \implies \exists U\ V. open\ U \wedge open\ V \wedge S \subseteq U \wedge T \subseteq V \wedge U \cap V = \{\}$

T4 is stronger than T3, and weaker than metric.

instance *t4-space* \subseteq *t3-space*

proof

fix *S* and *y*::'a **assume** *closed S y* $y \notin S$
then show $\exists U\ V. open\ U \wedge open\ V \wedge y \in U \wedge S \subseteq V \wedge U \cap V = \{\}$
using *t4-space*[*of {y} S*] **by** *auto*

qed

A perfect space is a topological space with no isolated points.

class *perfect-space* = *topological-space* +

assumes *not-open-singleton*: $\neg \text{open } \{x\}$

lemma (*in perfect-space*) *UNIV-not-singleton*: $UNIV \neq \{x\}$
for $x::'a$
by (*metis (no-types) open-UNIV not-open-singleton*)

99.3 Generators for topologies

inductive *generate-topology* :: $'a \text{ set set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ **for** $S :: 'a \text{ set set}$
where

UNIV: *generate-topology* S *UNIV*
| *Int*: *generate-topology* S $(a \cap b)$ **if** *generate-topology* S a **and** *generate-topology* S b
| *UN*: *generate-topology* S $(\bigcup K)$ **if** $(\bigwedge k. k \in K \Longrightarrow \text{generate-topology } S \ k)$
| *Basis*: *generate-topology* S s **if** $s \in S$

hide-fact (**open**) *UNIV Int UN Basis*

lemma *generate-topology-Union*:
 $(\bigwedge k. k \in I \Longrightarrow \text{generate-topology } S \ (K \ k)) \Longrightarrow \text{generate-topology } S \ (\bigcup k \in I. K \ k)$
using *generate-topology.UN [of K ' I]* **by** *auto*

lemma *topological-space-generate-topology*: *class.topological-space* (*generate-topology* S)
by *standard (auto intro: generate-topology.intros)*

99.4 Order topologies

class *order-topology* = *order* + *open* +
assumes *open-generated-order*: $\text{open} = \text{generate-topology} \ (\text{range } (\lambda a. \{..< a\}) \cup \text{range } (\lambda a. \{a <..\}))$
begin

subclass *topological-space*
unfolding *open-generated-order*
by (*rule topological-space-generate-topology*)

lemma *open-greaterThan* [*continuous-intros, simp*]: $\text{open } \{a <..\}$
unfolding *open-generated-order* **by** (*auto intro: generate-topology.Basis*)

lemma *open-lessThan* [*continuous-intros, simp*]: $\text{open } \{..< a\}$
unfolding *open-generated-order* **by** (*auto intro: generate-topology.Basis*)

lemma *open-greaterThanLessThan* [*continuous-intros, simp*]: $\text{open } \{a <..< b\}$
unfolding *greaterThanLessThan-eq* **by** (*simp add: open-Int*)

end

class *linorder-topology* = *linorder* + *order-topology*

lemma *closed-atMost* [*continuous-intros, simp*]: *closed* $\{..a\}$
for $a :: 'a::linorder-topology$
by (*simp add: closed-open*)

lemma *closed-atLeast* [*continuous-intros, simp*]: *closed* $\{a.. \}$
for $a :: 'a::linorder-topology$
by (*simp add: closed-open*)

lemma *closed-atLeastAtMost* [*continuous-intros, simp*]: *closed* $\{a..b\}$
for $a b :: 'a::linorder-topology$
proof –
have $\{a .. b\} = \{a .. \} \cap \{.. b\}$
by *auto*
then show *?thesis*
by (*simp add: closed-Int*)
qed

lemma (*in order*) *less-separate*:
assumes $x < y$
shows $\exists a b. x \in \{..< a\} \wedge y \in \{b <..\} \wedge \{..< a\} \cap \{b <..\} = \{\}$
proof (*cases* $\exists z. x < z \wedge z < y$)
case *True*
then obtain z **where** $x < z \wedge z < y ..$
then have $x \in \{..< z\} \wedge y \in \{z <..\} \wedge \{z <..\} \cap \{..< z\} = \{\}$
by *auto*
then show *?thesis* **by** *blast*
next
case *False*
with $\langle x < y \rangle$ **have** $x \in \{..< y\} \wedge y \in \{x <..\} \wedge \{x <..\} \cap \{..< y\} = \{\}$
by *auto*
then show *?thesis* **by** *blast*
qed

instance *linorder-topology* \subseteq *t2-space*

proof
fix $x y :: 'a$
show $x \neq y \implies \exists U V. \text{open } U \wedge \text{open } V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$
using *less-separate* [*of x y*] *less-separate* [*of y x*]
by (*elim neqE;metis open-lessThan open-greaterThan Int-commute*)
qed

lemma (*in linorder-topology*) *open-right*:
assumes *open* S $x \in S$
and *gt-ex*: $x < y$
shows $\exists b > x. \{x ..< b\} \subseteq S$
using *assms unfolding open-generated-order*
proof *induct*
case *UNIV*
then show *?case* **by** *blast*

```

next
  case (Int A B)
  then obtain a b where a > x {x ..< a} ⊆ A b > x {x ..< b} ⊆ B
    by auto
  then show ?case
    by (auto intro!: exI[of - min a b])
next
  case UN
  then show ?case by blast
next
  case Basis
  then show ?case
    by (fastforce intro: exI[of - y] gt-ex)
qed

```

```

lemma (in linorder-topology) open-left:
  assumes open S x ∈ S
    and lt-ex: y < x
  shows ∃ b < x. {b <.. x} ⊆ S
  using assms unfolding open-generated-order
proof induction
  case UNIV
  then show ?case by blast
next
  case (Int A B)
  then obtain a b where a < x {a <.. x} ⊆ A b < x {b <.. x} ⊆ B
    by auto
  then show ?case
    by (auto intro!: exI[of - max a b])
next
  case UN
  then show ?case by blast
next
  case Basis
  then show ?case
    by (fastforce intro: exI[of - y] lt-ex)
qed

```

99.5 Setup some topologies

99.5.1 Boolean is an order topology

```

class discrete-topology = topological-space +
  assumes open-discrete: ⋀ A. open A

```

```

instance discrete-topology < t2-space

```

```

proof

```

```

  fix x y :: 'a

```

```

  assume x ≠ y

```

```

  then show ∃ U V. open U ∧ open V ∧ x ∈ U ∧ y ∈ V ∧ U ∩ V = {}

```

by (*intro exI*[of - {-}]) (*auto intro!*: *open-discrete*)
qed

instantiation *bool* :: *linorder-topology*
begin

definition *open-bool* :: *bool set* \Rightarrow *bool*
where *open-bool* = *generate-topology* (*range* ($\lambda a. \{..< a\}$) \cup *range* ($\lambda a. \{a <..\}$))

instance
by *standard* (*rule open-bool-def*)

end

instance *bool* :: *discrete-topology*

proof

fix *A* :: *bool set*

have *: $\{False <..\} = \{True\} \{..< True\} = \{False\}$

by *auto*

have $A = UNIV \vee A = \{\} \vee A = \{False <..\} \vee A = \{..< True\}$

using *subset-UNIV*[of *A*] **unfolding** *UNIV-bool* * by *blast*

then show *open A*

by *auto*

qed

instantiation *nat* :: *linorder-topology*
begin

definition *open-nat* :: *nat set* \Rightarrow *bool*
where *open-nat* = *generate-topology* (*range* ($\lambda a. \{..< a\}$) \cup *range* ($\lambda a. \{a <..\}$))

instance
by *standard* (*rule open-nat-def*)

end

instance *nat* :: *discrete-topology*

proof

fix *A* :: *nat set*

have *open* $\{n\}$ for *n* :: *nat*

proof (*cases n*)

case 0

moreover have $\{0\} = \{..< 1::nat\}$

by *auto*

ultimately show *?thesis*

by *auto*

next

case (*Suc n'*)

then have $\{n\} = \{..< Suc\ n\} \cap \{n' <..\}$

```

    by auto
  with Suc show ?thesis
    by (auto intro: open-lessThan open-greaterThan)
qed
then have open ( $\bigcup a \in A. \{a\}$ )
  by (intro open-UN) auto
then show open A
  by simp
qed

instantiation int :: linorder-topology
begin

definition open-int :: int set  $\Rightarrow$  bool
  where open-int = generate-topology (range ( $\lambda a. \{..< a\}$ )  $\cup$  range ( $\lambda a. \{a <..\}$ ))

instance
  by standard (rule open-int-def)

end

instance int :: discrete-topology
proof
  fix A :: int set
  have  $\{..<i + 1\} \cap \{i-1 <..\} = \{i\}$  for i :: int
    by auto
  then have open  $\{i\}$  for i :: int
    using open-Int[OF open-lessThan[of i + 1] open-greaterThan[of i - 1]] by
  auto
  then have open ( $\bigcup a \in A. \{a\}$ )
    by (intro open-UN) auto
  then show open A
    by simp
qed

```

99.5.2 Topological filters

definition (in *topological-space*) *nhds* :: 'a \Rightarrow 'a filter
 where *nhds* a = (INF S \in {S. open S \wedge a \in S}. principal S)

definition (in *topological-space*) *at-within* :: 'a \Rightarrow 'a set \Rightarrow 'a filter
 (at (-)/ within (-) [1000, 60] 60)
 where *at* a within s = inf (nhds a) (principal (s - {a}))

abbreviation (in *topological-space*) *at* :: 'a \Rightarrow 'a filter (at)
 where *at* x \equiv at x within (CONST UNIV)

abbreviation (in *order-topology*) *at-right* :: 'a \Rightarrow 'a filter
 where *at-right* x \equiv at x within {x <..}

abbreviation (in *order-topology*) *at-left* :: 'a \Rightarrow 'a filter
where *at-left* $x \equiv$ *at* x within $\{..< x\}$

lemma (in *topological-space*) *nhds-generated-topology*:
open = *generate-topology* $T \Longrightarrow$ *nhds* $x = (\text{INF } S \in \{S \in T. x \in S\}. \text{principal } S)$
unfolding *nhds-def*

proof (*safe intro!*: *antisym INF-greatest*)

fix S

assume *generate-topology* $T S x \in S$

then show $(\text{INF } S \in \{S \in T. x \in S\}. \text{principal } S) \leq \text{principal } S$

by *induct*

(*auto intro!*: *INF-lower order-trans simp: inf-principal[symmetric] simp del: inf-principal*)

qed (*auto intro!*: *INF-lower intro: generate-topology.intros*)

lemma (in *topological-space*) *eventually-nhds*:
eventually P (*nhds* a) $\longleftrightarrow (\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. P x))$
unfolding *nhds-def* **by** (*subst eventually-INF-base*) (*auto simp: eventually-principal*)

lemma *eventually-eventually*:
eventually $(\lambda y. \text{eventually } P (\text{nhds } y)) (\text{nhds } x) = \text{eventually } P (\text{nhds } x)$
by (*auto simp: eventually-nhds*)

lemma (in *topological-space*) *eventually-nhds-in-open*:
open $s \Longrightarrow x \in s \Longrightarrow \text{eventually } (\lambda y. y \in s) (\text{nhds } x)$
by (*subst eventually-nhds*) *blast*

lemma (in *topological-space*) *eventually-nhds-x-imp-x*: *eventually* P (*nhds* x) \Longrightarrow $P x$
by (*subst (asm) eventually-nhds*) *blast*

lemma (in *topological-space*) *nhds-neq-bot [simp]*: *nhds* $a \neq \text{bot}$
by (*simp add: trivial-limit-def eventually-nhds*)

lemma (in *t1-space*) *t1-space-nhds*: $x \neq y \Longrightarrow (\forall_F x \text{ in } \text{nhds } x. x \neq y)$
by (*drule t1-space*) (*auto simp: eventually-nhds*)

lemma (in *topological-space*) *nhds-discrete-open*: *open* $\{x\} \Longrightarrow \text{nhds } x = \text{principal } \{x\}$
by (*auto simp: nhds-def intro!: antisym INF-greatest INF-lower2[of \{x\}]*)

lemma (in *discrete-topology*) *nhds-discrete*: *nhds* $x = \text{principal } \{x\}$
by (*simp add: nhds-discrete-open open-discrete*)

lemma (in *discrete-topology*) *at-discrete*: *at* x within $S = \text{bot}$
unfolding *at-within-def nhds-discrete* **by** *simp*

lemma (in *discrete-topology*) *tendsto-discrete*:

filterlim ($f :: 'b \Rightarrow 'a$) (*nhds* y) $F \longleftrightarrow$ *eventually* ($\lambda x. f x = y$) F
by (*auto simp: nhds-discrete filterlim-principal*)

lemma (*in topological-space*) *at-within-eq*:
at x within s = (INF S ∈ {S. open S ∧ x ∈ S}. principal (S ∩ s - {x}))
unfolding *nhds-def at-within-def*
by (*subst INF-inf-const2[symmetric]*) (*auto simp: Diff-Int-distrib*)

lemma (*in topological-space*) *eventually-at-filter*:
eventually P (at a within s) ↔ eventually (λx. x ≠ a → x ∈ s → P x) (*nhds a*)
by (*simp add: at-within-def eventually-inf-principal imp-conjL[symmetric] conj-commute*)

lemma (*in topological-space*) *at-le*: $s \subseteq t \implies$ *at x within s ≤ at x within t*
unfolding *at-within-def* **by** (*intro inf-mono*) *auto*

lemma (*in topological-space*) *eventually-at-topological*:
eventually P (at a within s) ↔ (∃ S. open S ∧ a ∈ S ∧ (∀ x ∈ S. x ≠ a → x ∈ s → P x))
by (*simp add: eventually-nhds eventually-at-filter*)

lemma *eventually-at-in-open*:
assumes *open A x ∈ A*
shows *eventually (λy. y ∈ A - {x}) (at x)*
using *assms eventually-at-topological* **by** *blast*

lemma *eventually-at-in-open'*:
assumes *open A x ∈ A*
shows *eventually (λy. y ∈ A) (at x)*
using *assms eventually-at-topological* **by** *blast*

lemma (*in topological-space*) *at-within-open*: $a \in S \implies$ *open S* \implies *at a within S = at a*
unfolding *filter-eq-iff eventually-at-topological* **by** (*metis open-Int Int-iff UNIV-I*)

lemma (*in topological-space*) *at-within-open-NO-MATCH*:
 $a \in s \implies$ *open s* \implies *NO-MATCH UNIV s* \implies *at a within s = at a*
by (*simp only: at-within-open*)

lemma (*in topological-space*) *at-within-open-subset*:
 $a \in S \implies$ *open S* \implies $S \subseteq T \implies$ *at a within T = at a*
by (*metis at-le at-within-open dual-order.antisym subset-UNIV*)

lemma (*in topological-space*) *at-within-nhd*:
assumes $x \in S$ *open S T ∩ S - {x} = U ∩ S - {x}*
shows *at x within T = at x within U*
unfolding *filter-eq-iff eventually-at-filter*
proof (*intro allI eventually-subst*)
have *eventually (λx. x ∈ S) (nhds x)*

using $\langle x \in S \rangle \langle \text{open } S \rangle$ **by** (*auto simp: eventually-nhds*)
then show $\forall_F n \text{ in nhds } x. (n \neq x \longrightarrow n \in T \longrightarrow P n) = (n \neq x \longrightarrow n \in U \longrightarrow P n)$ **for** P
by *eventually-elim (insert $\langle T \cap S - \{x\} = U \cap S - \{x\} \rangle$, blast)*
qed

lemma (*in topological-space*) *at-within-empty [simp]: at a within $\{\}$ = bot*
unfolding *at-within-def* **by** *simp*

lemma (*in topological-space*) *at-within-union:*
at x within $(S \cup T) = \text{sup} (\text{at } x \text{ within } S) (\text{at } x \text{ within } T)$
unfolding *filter-eq-iff eventually-sup eventually-at-filter*
by (*auto elim!: eventually-rev-mp*)

lemma (*in topological-space*) *at-eq-bot-iff: at a = bot \longleftrightarrow open $\{a\}$*
unfolding *trivial-limit-def eventually-at-topological*
by (*metis UNIV-I empty-iff is-singletonE is-singletonI' singleton-iff*)

lemma (*in t1-space*) *eventually-neq-at-within:*
eventually $(\lambda w. w \neq x)$ (at z within A)
by (*smt (verit, ccfv-threshold) eventually-True eventually-at-topological separation-t1*)

lemma (*in perfect-space*) *at-neq-bot [simp]: at a \neq bot*
by (*simp add: at-eq-bot-iff not-open-singleton*)

lemma (*in order-topology*) *nhds-order:*
nhds x = inf (INF a $\in \{x <..\}$. principal $\{.. < a\}$) (INF a $\in \{.. < x\}$. principal $\{a <..\}$)
proof –
have $1: \{S \in \text{range lessThan} \cup \text{range greaterThan}. x \in S\} =$
 $(\lambda a. \{.. < a\}) ' \{x <..\} \cup (\lambda a. \{a <..\}) ' \{.. < x\}$
by *auto*
show *?thesis*
by (*simp only: nhds-generated-topology[OF open-generated-order] INF-union 1 INF-image comp-def*)
qed

lemma (*in topological-space*) *filterlim-at-within-If:*
assumes *filterlim f G (at x within $(A \cap \{x. P x\})$)*
and *filterlim g G (at x within $(A \cap \{x. \neg P x\})$)*
shows *filterlim $(\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) G$ (at x within A)*
proof (*rule filterlim-If*)
note *assms(1)*
also have *at x within $(A \cap \{x. P x\}) = \text{inf} (\text{nhds } x) (\text{principal } (A \cap \text{Collect } P - \{x\}))$*
by (*simp add: at-within-def*)
also have *$A \cap \text{Collect } P - \{x\} = (A - \{x\}) \cap \text{Collect } P$*
by *blast*

also have $\text{inf } (\text{nhds } x) (\text{principal } \dots) = \text{inf } (\text{at } x \text{ within } A) (\text{principal } (\text{Collect } P))$
by (*simp add: at-within-def inf-assoc*)
finally show $\text{filterlim } f G (\text{inf } (\text{at } x \text{ within } A) (\text{principal } (\text{Collect } P)))$.
next
note *assms(2)*
also have $\text{at } x \text{ within } (A \cap \{x. \neg P x\}) = \text{inf } (\text{nhds } x) (\text{principal } (A \cap \{x. \neg P x\} - \{x\}))$
by (*simp add: at-within-def*)
also have $A \cap \{x. \neg P x\} - \{x\} = (A - \{x\}) \cap \{x. \neg P x\}$
by *blast*
also have $\text{inf } (\text{nhds } x) (\text{principal } \dots) = \text{inf } (\text{at } x \text{ within } A) (\text{principal } \{x. \neg P x\})$
by (*simp add: at-within-def inf-assoc*)
finally show $\text{filterlim } g G (\text{inf } (\text{at } x \text{ within } A) (\text{principal } \{x. \neg P x\}))$.
qed

lemma (*in topological-space*) *filterlim-at-If*:

assumes $\text{filterlim } f G (\text{at } x \text{ within } \{x. P x\})$
and $\text{filterlim } g G (\text{at } x \text{ within } \{x. \neg P x\})$
shows $\text{filterlim } (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) G (\text{at } x)$
using *assms* **by** (*intro filterlim-at-within-If*) *simp-all*

lemma (*in linorder-topology*) *at-within-order*:

assumes $UNIV \neq \{x\}$

shows $\text{at } x \text{ within } s =$

$\text{inf } (INF a \in \{x <..\}. \text{principal } (\{.. < a\} \cap s - \{x\}))$
 $(INF a \in \{.. < x\}. \text{principal } (\{a <..\} \cap s - \{x\}))$

proof (*cases* $\{x <..\} = \{\}$ $\{.. < x\} = \{\}$ *rule: case-split [case-product case-split]*)

case *True-True*

have $UNIV = \{.. < x\} \cup \{x\} \cup \{x <..\}$

by *auto*

with *assms True-True* **show** *?thesis*

by *auto*

qed (*auto simp del: inf-principal simp: at-within-def nhds-order Int-Diff*

inf-principal[symmetric] INF-inf-const2 inf-sup-aci[where 'a='a filter])

lemma (*in linorder-topology*) *at-left-eq*:

$y < x \implies \text{at-left } x = (INF a \in \{.. < x\}. \text{principal } \{a <..\ < x\})$

by (*subst at-within-order*)

(*auto simp: greaterThan-Int-greaterThan greaterThanLessThan-eq[symmetric]*)

min.absorb2 INF-constant

intro!: INF-lower2 inf-absorb2)

lemma (*in linorder-topology*) *eventually-at-left*:

$y < x \implies \text{eventually } P (\text{at-left } x) \longleftrightarrow (\exists b < x. \forall y > b. y < x \implies P y)$

unfolding *at-left-eq*

by (*subst eventually-INF-base*) (*auto simp: eventually-principal Ball-def*)

lemma (*in linorder-topology*) *at-right-eq*:

$x < y \implies \text{at-right } x = (\text{INF } a \in \{x <..\}. \text{principal } \{x <..< a\})$
by (*subst at-within-order*)
 (*auto simp: lessThan-Int-lessThan greaterThanLessThan-eq[symmetric] max.absorb2*
INF-constant Int-commute
intro!: INF-lower2 inf-absorb1)

lemma (*in linorder-topology*) *eventually-at-right*:
 $x < y \implies \text{eventually } P (\text{at-right } x) \longleftrightarrow (\exists b > x. \forall y > x. y < b \implies P y)$
unfolding *at-right-eq*
by (*subst eventually-INF-base*) (*auto simp: eventually-principal Ball-def*)

lemma *eventually-at-right-less*: $\forall_F y \text{ in } \text{at-right } (x::'a::\{\text{linorder-topology, no-top}\}).$
 $x < y$
using *gt-ex[of x] eventually-at-right[of x]* **by** *auto*

lemma *trivial-limit-at-right-top*: $\text{at-right } (\text{top}:::\{\text{order-top, linorder-topology}\}) =$
 bot
by (*auto simp: filter-eq-iff eventually-at-topological*)

lemma *trivial-limit-at-left-bot*: $\text{at-left } (\text{bot}:::\{\text{order-bot, linorder-topology}\}) = \text{bot}$
by (*auto simp: filter-eq-iff eventually-at-topological*)

lemma *trivial-limit-at-left-real* [*simp*]: $\neg \text{trivial-limit } (\text{at-left } x)$
for $x :: 'a::\{\text{no-bot, dense-order, linorder-topology}\}$
using *lt-ex [of x]*
by *safe* (*auto simp add: trivial-limit-def eventually-at-left dest: dense*)

lemma *trivial-limit-at-right-real* [*simp*]: $\neg \text{trivial-limit } (\text{at-right } x)$
for $x :: 'a::\{\text{no-top, dense-order, linorder-topology}\}$
using *gt-ex [of x]*
by *safe* (*auto simp add: trivial-limit-def eventually-at-right dest: dense*)

lemma (*in linorder-topology*) *at-eq-sup-left-right*: $\text{at } x = \text{sup } (\text{at-left } x) (\text{at-right } x)$
by (*auto simp: eventually-at-filter filter-eq-iff eventually-sup*
elim: eventually-elim2 eventually-mono)

lemma (*in linorder-topology*) *eventually-at-split*:
 $\text{eventually } P (\text{at } x) \longleftrightarrow \text{eventually } P (\text{at-left } x) \wedge \text{eventually } P (\text{at-right } x)$
by (*subst at-eq-sup-left-right*) (*simp add: eventually-sup*)

lemma (*in order-topology*) *eventually-at-leftI*:
assumes $\bigwedge x. x \in \{a <..< b\} \implies P x a < b$
shows $\text{eventually } P (\text{at-left } b)$
using *assms* **unfolding** *eventually-at-topological* **by** (*intro exI [of - \{a <..\}]*) *auto*

lemma (*in order-topology*) *eventually-at-rightI*:
assumes $\bigwedge x. x \in \{a <..< b\} \implies P x a < b$
shows $\text{eventually } P (\text{at-right } a)$

using *assms* **unfolding** *eventually-at-topological* **by** (*intro exI[of - {..<b}]*) *auto*

lemma *eventually-filtercomap-nhds*:

eventually P (filtercomap f (nhds x)) \longleftrightarrow ($\exists S. \text{open } S \wedge x \in S \wedge (\forall x. f x \in S \longrightarrow P x)$)

unfolding *eventually-filtercomap eventually-nhds* **by** *auto*

lemma *eventually-filtercomap-at-topological*:

eventually P (filtercomap f (at A within B)) \longleftrightarrow

($\exists S. \text{open } S \wedge A \in S \wedge (\forall x. f x \in S \cap B - \{A\} \longrightarrow P x)$) (**is** *?lhs = ?rhs*)

unfolding *at-within-def filtercomap-inf eventually-inf-principal filtercomap-principal*

eventually-filtercomap-nhds eventually-principal **by** *blast*

lemma *eventually-at-right-field*:

eventually P (at-right x) \longleftrightarrow ($\exists b > x. \forall y > x. y < b \longrightarrow P y$)

for $x :: 'a :: \{\text{linordered-field, linorder-topology}\}$

using *linordered-field-no-ub[rule-format, of x]*

by (*auto simp: eventually-at-right*)

lemma *eventually-at-left-field*:

eventually P (at-left x) \longleftrightarrow ($\exists b < x. \forall y > b. y < x \longrightarrow P y$)

for $x :: 'a :: \{\text{linordered-field, linorder-topology}\}$

using *linordered-field-no-lb[rule-format, of x]*

by (*auto simp: eventually-at-left*)

lemma *filtermap-nhds-eq-imp-filtermap-at-eq*:

assumes *filtermap f (nhds z) = nhds (f z)*

assumes *eventually* ($\lambda x. f x = f z \longrightarrow x = z$) (*at z*)

shows *filtermap f (at z) = at (f z)*

proof (*rule filter-eqI*)

fix $P :: 'a \Rightarrow \text{bool}$

have *eventually P (filtermap f (at z))* \longleftrightarrow ($\forall_F x \text{ in } \text{nhds } z. x \neq z \longrightarrow P (f x)$)

by (*simp add: eventually-filtermap eventually-at-filter*)

also have $\dots \longleftrightarrow$ ($\forall_F x \text{ in } \text{nhds } z. f x \neq f z \longrightarrow P (f x)$)

by (*rule eventually-cong [OF assms(2)[unfolded eventually-at-filter]]*) *auto*

also have $\dots \longleftrightarrow$ ($\forall_F x \text{ in } \text{filtermap } f (\text{nhds } z). x \neq f z \longrightarrow P x$)

by (*simp add: eventually-filtermap*)

also have *filtermap f (nhds z) = nhds (f z)*

by (*rule assms*)

also have ($\forall_F x \text{ in } \text{nhds } (f z). x \neq f z \longrightarrow P x$) \longleftrightarrow ($\forall_F x \text{ in } \text{at } (f z). P x$)

by (*simp add: eventually-at-filter*)

finally show *eventually P (filtermap f (at z)) = eventually P (at (f z))* .

qed

99.5.3 Tendsto

abbreviation (*in topological-space*)

tendsto :: ($'b \Rightarrow 'a$) \Rightarrow $'a \Rightarrow 'b \text{ filter} \Rightarrow \text{bool}$ (**infixr** \longrightarrow 55)

where $(f \longrightarrow l) F \equiv \text{filterlim } f \text{ (nhds } l) F$

definition (in *t2-space*) $\text{Lim} :: 'f \text{ filter} \Rightarrow ('f \Rightarrow 'a) \Rightarrow 'a$
where $\text{Lim } A f = (\text{THE } l. (f \longrightarrow l) A)$

lemma (in *topological-space*) $\text{tendsto-eq-rhs}: (f \longrightarrow x) F \Longrightarrow x = y \Longrightarrow (f \longrightarrow y) F$
by *simp*

named-theorems *tendsto-intros* introduction rules for *tendsto*
setup <

Global-Theory.add-thms-dynamic (**binding** <*tendsto-eq-intros*>,
fn context =>
Named-Theorems.get (*Context.proof-of context*) **named-theorems** <*tendsto-intros*>
 |> *map-filter* (*try* (*fn thm* => @{*thm tendsto-eq-rhs*} *OF* [*thm*])))
 >

context *topological-space* **begin**

lemma *tendsto-def*:

$(f \longrightarrow l) F \longleftrightarrow (\forall S. \text{open } S \longrightarrow l \in S \longrightarrow \text{eventually } (\lambda x. f x \in S) F)$
unfolding *nhds-def filterlim-INF filterlim-principal* **by** *auto*

lemma *tendsto-cong*: $(f \longrightarrow c) F \longleftrightarrow (g \longrightarrow c) F$ **if** *eventually* $(\lambda x. f x = g x) F$
by (*rule filterlim-cong [OF refl refl that]*)

lemma *tendsto-mono*: $F \leq F' \Longrightarrow (f \longrightarrow l) F' \Longrightarrow (f \longrightarrow l) F$
unfolding *tendsto-def le-filter-def* **by** *fast*

lemma *tendsto-ident-at* [*tendsto-intros, simp, intro*]: $((\lambda x. x) \longrightarrow a) (at a \text{ within } s)$
by (*auto simp: tendsto-def eventually-at-topological*)

lemma *tendsto-const* [*tendsto-intros, simp, intro*]: $((\lambda x. k) \longrightarrow k) F$
by (*simp add: tendsto-def*)

lemma *filterlim-at*:

$(\text{LIM } x F. f x \text{ :> } at b \text{ within } s) \longleftrightarrow \text{eventually } (\lambda x. f x \in s \wedge f x \neq b) F \wedge (f \longrightarrow b) F$
by (*simp add: at-within-def filterlim-inf filterlim-principal conj-commute*)

lemma (in $-$)

assumes *filterlim* f (nhds L) F

shows *tendsto-imp-filterlim-at-right*:

eventually $(\lambda x. f x > L) F \Longrightarrow \text{filterlim } f \text{ (at-right } L) F$

and *tendsto-imp-filterlim-at-left*:

eventually $(\lambda x. f x < L) F \Longrightarrow \text{filterlim } f \text{ (at-left } L) F$

using *assms* **by** (*auto simp: filterlim-at elim: eventually-mono*)

lemma *filterlim-at-withinI*:

assumes *filterlim* f (*nhds* c) F
assumes *eventually* $(\lambda x. f\ x \in A - \{c\})\ F$
shows *filterlim* f (*at* c *within* A) F
using *assms* **by** (*simp* *add*: *filterlim-at*)

lemma *filterlim-atI*:

assumes *filterlim* f (*nhds* c) F
assumes *eventually* $(\lambda x. f\ x \neq c)\ F$
shows *filterlim* f (*at* c) F
using *assms* **by** (*intro* *filterlim-at-withinI*) *simp-all*

lemma *topological-tendstoI*:

$(\bigwedge S. \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. f\ x \in S)\ F) \implies (f \longrightarrow l)\ F$
by (*auto* *simp*: *tendsto-def*)

lemma *topological-tendstoD*:

$(f \longrightarrow l)\ F \implies \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. f\ x \in S)\ F$
by (*auto* *simp*: *tendsto-def*)

lemma *tendsto-bot* [*simp*]: $(f \longrightarrow a)\ \text{bot}$

by (*simp* *add*: *tendsto-def*)

lemma *tendsto-eventually*: *eventually* $(\lambda x. f\ x = l)\ \text{net} \implies ((\lambda x. f\ x) \longrightarrow l)\ \text{net}$

by (*rule* *topological-tendstoI*) (*auto* *elim*: *eventually-mono*)

lemma *tendsto-principal-singleton*[*simp*]:

shows $(f \longrightarrow f\ x)$ (*principal* $\{x\}$)
unfolding *tendsto-def* *eventually-principal* **by** *simp*

end

lemma (*in* *topological-space*) *filterlim-within-subset*:

filterlim $f\ l$ (*at* x *within* S) $\implies T \subseteq S \implies \text{filterlim } f\ l$ (*at* x *within* T)
by (*blast* *intro*: *filterlim-mono* *at-le*)

lemmas *tendsto-within-subset* = *filterlim-within-subset*

lemma (*in* *order-topology*) *order-tendsto-iff*:

$(f \longrightarrow x)\ F \iff (\forall l < x. \text{eventually } (\lambda x. l < f\ x)\ F) \wedge (\forall u > x. \text{eventually } (\lambda x. f\ x < u)\ F)$

by (*auto* *simp*: *nhds-order* *filterlim-inf* *filterlim-INF* *filterlim-principal*)

lemma (*in* *order-topology*) *order-tendstoI*:

$(\bigwedge a. a < y \implies \text{eventually } (\lambda x. a < f\ x)\ F) \implies (\bigwedge a. y < a \implies \text{eventually } (\lambda x. f\ x < a)\ F) \implies$

$(f \longrightarrow y)\ F$

by (auto simp: order-tendsto-iff)

lemma (in order-topology) order-tendstoD:

assumes $f \longrightarrow y$ F

shows $a < y \implies$ eventually $(\lambda x. a < f x)$ F

and $y < a \implies$ eventually $(\lambda x. f x < a)$ F

using assms by (auto simp: order-tendsto-iff)

lemma (in linorder-topology) tendsto-max[tendsto-intros]:

assumes $X: (X \longrightarrow x)$ net

and $Y: (Y \longrightarrow y)$ net

shows $((\lambda x. \max (X x) (Y x)) \longrightarrow \max x y)$ net

proof (rule order-tendstoI)

fix a

assume $a < \max x y$

then show eventually $(\lambda x. a < \max (X x) (Y x))$ net

using order-tendstoD(1)[OF X, of a] order-tendstoD(1)[OF Y, of a]

by (auto simp: less-max-iff-disj elim: eventually-mono)

next

fix a

assume $\max x y < a$

then show eventually $(\lambda x. \max (X x) (Y x) < a)$ net

using order-tendstoD(2)[OF X, of a] order-tendstoD(2)[OF Y, of a]

by (auto simp: eventually-conj-iff)

qed

lemma (in linorder-topology) tendsto-min[tendsto-intros]:

assumes $X: (X \longrightarrow x)$ net

and $Y: (Y \longrightarrow y)$ net

shows $((\lambda x. \min (X x) (Y x)) \longrightarrow \min x y)$ net

proof (rule order-tendstoI)

fix a

assume $a < \min x y$

then show eventually $(\lambda x. a < \min (X x) (Y x))$ net

using order-tendstoD(1)[OF X, of a] order-tendstoD(1)[OF Y, of a]

by (auto simp: eventually-conj-iff)

next

fix a

assume $\min x y < a$

then show eventually $(\lambda x. \min (X x) (Y x) < a)$ net

using order-tendstoD(2)[OF X, of a] order-tendstoD(2)[OF Y, of a]

by (auto simp: min-less-iff-disj elim: eventually-mono)

qed

lemma (in order-topology)

assumes $a < b$

shows at-within-Icc-at-right: at a within $\{a..b\} =$ at-right a

and at-within-Icc-at-left: at b within $\{a..b\} =$ at-left b

using order-tendstoD(2)[OF tendsto-ident-at assms, of $\{a<..\}$]

using *order-tendstoD(1)[OF tendsto-ident-at assms, of {..**<b**}*]
by (*auto intro!*: *order-class.order-antisym filter-leI*
simp: eventually-at-filter less-le
elim: eventually-elim2)

lemma (**in** *order-topology*) *at-within-Icc-at: $a < x \implies x < b \implies \text{at } x \text{ within } \{a..b\} = \text{at } x$*
by (*rule at-within-open-subset[where $S=\{a<..**<b**\}$]*) *auto*

lemma (**in** *t2-space*) *tendsto-unique*:

assumes $F \neq \text{bot}$
and ($f \longrightarrow a$) F
and ($f \longrightarrow b$) F
shows $a = b$
proof (*rule ccontr*)
assume $a \neq b$
obtain $U V$ **where** *open U open V $a \in U b \in V U \cap V = \{\}$*
using *hausdorff [OF $\langle a \neq b \rangle$]* **by** *fast*
have *eventually* ($\lambda x. f x \in U$) F
using $\langle f \longrightarrow a \rangle F \langle \text{open } U \rangle \langle a \in U \rangle$ **by** (*rule topological-tendstoD*)
moreover
have *eventually* ($\lambda x. f x \in V$) F
using $\langle f \longrightarrow b \rangle F \langle \text{open } V \rangle \langle b \in V \rangle$ **by** (*rule topological-tendstoD*)
ultimately
have *eventually* ($\lambda x. \text{False}$) F
proof *eventually-elim*
case (*elim x*)
then **have** $f x \in U \cap V$ **by** *simp*
with $\langle U \cap V = \{\} \rangle$ **show** *?case* **by** *simp*
qed
with $\langle \neg \text{trivial-limit } F \rangle$ **show** *False*
by (*simp add: trivial-limit-def*)
qed

lemma (**in** *t2-space*) *tendsto-const-iff*:

fixes $a b :: 'a$
assumes $\neg \text{trivial-limit } F$
shows $((\lambda x. a) \longrightarrow b) F \iff a = b$
by (*auto intro!*: *tendsto-unique [OF assms tendsto-const]*)

lemma (**in** *t2-space*) *tendsto-unique'*:

assumes $F \neq \text{bot}$
shows $\exists \leq_1 l. (f \longrightarrow l) F$
using *Uniq-def assms local.tendsto-unique* **by** *fastforce*

lemma *Lim-in-closed-set*:

assumes *closed S eventually* ($\lambda x. f(x) \in S$) $F F \neq \text{bot}$ ($f \longrightarrow l$) F
shows $l \in S$
proof (*rule ccontr*)

```

assume  $l \notin S$ 
with  $\langle \text{closed } S \rangle$  have  $\text{open } (- S) \text{ } l \in - S$ 
  by (simp-all add: open-Compl)
with assms(4) have  $\text{eventually } (\lambda x. f x \in - S) F$ 
  by (rule topological-tendstoD)
with assms(2) have  $\text{eventually } (\lambda x. \text{False}) F$ 
  by (rule eventually-elim2) simp
with assms(3) show False
  by (simp add: eventually-False)
qed

```

lemma (*in t3-space*) *nhds-closed*:

```

assumes  $x \in A$  and open A
shows  $\exists A'. x \in A' \wedge \text{closed } A' \wedge A' \subseteq A \wedge \text{eventually } (\lambda y. y \in A') (\text{nhds } x)$ 
proof –
  from assms have  $\exists U V. \text{open } U \wedge \text{open } V \wedge x \in U \wedge - A \subseteq V \wedge U \cap V = \{\}$ 
  by (intro t3-space) auto
  then obtain  $U V$  where  $UV: \text{open } U \text{ open } V \text{ } x \in U \text{ } -A \subseteq V \text{ } U \cap V = \{\}$ 
  by auto
  have  $\text{eventually } (\lambda y. y \in U) (\text{nhds } x)$ 
  using  $\langle \text{open } U \rangle$  and  $\langle x \in U \rangle$  by (intro eventually-nhds-in-open)
  hence  $\text{eventually } (\lambda y. y \in -V) (\text{nhds } x)$ 
  by eventually-elim (use UV in auto)
  with  $UV$  show ?thesis by (intro exI[of - -V]) auto
qed

```

lemma (*in order-topology*) *increasing-tendsto*:

```

assumes bdd:  $\text{eventually } (\lambda n. f n \leq l) F$ 
  and en:  $\bigwedge x. x < l \implies \text{eventually } (\lambda n. x < f n) F$ 
shows  $(f \longrightarrow l) F$ 
using assms by (intro order-tendstoI) (auto elim!: eventually-mono)

```

lemma (*in order-topology*) *decreasing-tendsto*:

```

assumes bdd:  $\text{eventually } (\lambda n. l \leq f n) F$ 
  and en:  $\bigwedge x. l < x \implies \text{eventually } (\lambda n. f n < x) F$ 
shows  $(f \longrightarrow l) F$ 
using assms by (intro order-tendstoI) (auto elim!: eventually-mono)

```

lemma (*in order-topology*) *tendsto-sandwich*:

```

assumes ev:  $\text{eventually } (\lambda n. f n \leq g n) \text{ net}$   $\text{eventually } (\lambda n. g n \leq h n) \text{ net}$ 
assumes lim:  $(f \longrightarrow c) \text{ net}$   $(h \longrightarrow c) \text{ net}$ 
shows  $(g \longrightarrow c) \text{ net}$ 
proof (rule order-tendstoI)
  fix  $a$ 
  show  $a < c \implies \text{eventually } (\lambda x. a < g x) \text{ net}$ 
  using order-tendstoD[OF lim(1), of a] ev by (auto elim: eventually-elim2)
next
  fix  $a$ 

```

show $c < a \implies \text{eventually } (\lambda x. g x < a)$ *net*
using *order-tendstoD[OF lim(2), of a] ev* **by** (*auto elim: eventually-elim2*)
qed

lemma (*in t1-space*) *limit-frequently-eq*:
assumes $F \neq \text{bot}$
and *frequently* $(\lambda x. f x = c)$ F
and $(f \longrightarrow d)$ F
shows $d = c$
proof (*rule ccontr*)
assume $d \neq c$
from *t1-space[OF this]* **obtain** U **where** *open* U $d \in U$ $c \notin U$
by *blast*
with *assms* **have** *eventually* $(\lambda x. f x \in U)$ F
unfolding *tendsto-def* **by** *blast*
then **have** *eventually* $(\lambda x. f x \neq c)$ F
by *eventually-elim (insert <c $\notin U$, blast)*
with *assms(2)* **show** *False*
unfolding *frequently-def* **by** *contradiction*
qed

lemma (*in t1-space*) *tendsto-imp-eventually-ne*:
assumes $(f \longrightarrow c)$ F $c \neq c'$
shows *eventually* $(\lambda z. f z \neq c')$ F
proof (*cases F=bot*)
case *True*
thus *?thesis* **by** *auto*
next
case *False*
show *?thesis*
proof (*rule ccontr*)
assume $\neg \text{eventually } (\lambda z. f z \neq c')$ F
then **have** *frequently* $(\lambda z. f z = c')$ F
by (*simp add: frequently-def*)
from *limit-frequently-eq[OF False this <(f $\longrightarrow c$) F]* **and** $\langle c \neq c' \rangle$ **show**
False
by *contradiction*
qed
qed

lemma (*in linorder-topology*) *tendsto-le*:
assumes $F: \neg \text{trivial-limit } F$
and $x: (f \longrightarrow x)$ F
and $y: (g \longrightarrow y)$ F
and $ev: \text{eventually } (\lambda x. g x \leq f x)$ F
shows $y \leq x$
proof (*rule ccontr*)
assume $\neg y \leq x$
with *less-separate[of x y]* **obtain** a b **where** $xy: x < a$ $b < y$ $\{..<a\} \cap \{b<..\} =$

```

{}
  by (auto simp: not-le)
  then have eventually ( $\lambda x. f x < a$ )  $F$  eventually ( $\lambda x. b < g x$ )  $F$ 
    using  $x y$  by (auto intro: order-tendstoD)
  with  $ev$  have eventually ( $\lambda x. False$ )  $F$ 
    by eventually-elim (insert  $xy$ , fastforce)
  with  $F$  show  $False$ 
    by (simp add: eventually-False)
qed

```

```

lemma (in linorder-topology) tendsto-lowerbound:
  assumes  $x: (f \longrightarrow x) F$ 
    and  $ev: eventually (\lambda i. a \leq f i) F$ 
    and  $F: \neg \text{trivial-limit } F$ 
  shows  $a \leq x$ 
  using  $F x$  tendsto-const  $ev$  by (rule tendsto-le)

```

```

lemma (in linorder-topology) tendsto-upperbound:
  assumes  $x: (f \longrightarrow x) F$ 
    and  $ev: eventually (\lambda i. a \geq f i) F$ 
    and  $F: \neg \text{trivial-limit } F$ 
  shows  $a \geq x$ 
  by (rule tendsto-le [OF  $F$  tendsto-const  $x ev$ ])

```

```

lemma filterlim-at-within-not-equal:
  fixes  $f::'a \Rightarrow 'b::t2\text{-space}$ 
  assumes filterlim  $f$  (at  $a$  within  $s$ )  $F$ 
  shows eventually ( $\lambda w. f w \in s \wedge f w \neq b$ )  $F$ 
proof (cases  $a=b$ )
  case True
    then show ?thesis using  $assms$  by (simp add: filterlim-at)
  next
  case False
    from hausdorff[OF this] obtain  $U V$  where  $UV: \text{open } U \text{ open } V a \in U b \in V$ 
       $U \cap V = \{\}$ 
      by auto
    have  $(f \longrightarrow a) F$  using  $assms$  filterlim-at by auto
    then have  $\forall_F x \text{ in } F. f x \in U$  using  $UV$  unfolding tendsto-def by auto
    moreover have  $\forall_F x \text{ in } F. f x \in s \wedge f x \neq a$  using  $assms$  filterlim-at by auto
    ultimately show ?thesis
      apply eventually-elim
      using  $UV$  by auto
qed

```

99.5.4 Rules about Lim

```

lemma tendsto-Lim:  $\neg \text{trivial-limit } net \implies (f \longrightarrow l) \text{ net} \implies Lim \text{ net } f = l$ 
  unfolding Lim-def using tendsto-unique [of net  $f$ ] by auto

```

lemma *Lim-ident-at*: $\neg \text{trivial-limit (at } x \text{ within } s) \implies \text{Lim (at } x \text{ within } s) (\lambda x. x) = x$

by (*rule tendsto-Lim[OF - tendsto-ident-at]*) *auto*

lemma *Lim-cong*:

assumes *eventually* $(\lambda x. f x = g x) F F = G$

shows $\text{Lim } F f = \text{Lim } G g$

proof (*cases* $(\exists c. (f \longrightarrow c) F) \wedge F \neq \text{bot}$)

case *True*

then obtain *c* **where** $c: (f \longrightarrow c) F$

by *blast*

hence $\text{Lim } F f = c$

using *True* **by** (*intro tendsto-Lim*) *auto*

moreover have $(f \longrightarrow c) F \longleftrightarrow (g \longrightarrow c) G$

using *assms* **by** (*intro filterlim-cong*) *auto*

with *True c assms* **have** $\text{Lim } G g = c$

by (*intro tendsto-Lim*) *auto*

ultimately show *?thesis*

by *simp*

next

case *False*

show *?thesis*

proof (*cases* $F = \text{bot}$)

case *True*

thus *?thesis* **using** *assms*

by (*auto simp: Topological-Spaces.Lim-def*)

next

case *False*

have $(f \longrightarrow c) F \longleftrightarrow (g \longrightarrow c) G$ **for** *c*

using *assms* **by** (*intro filterlim-cong*) *auto*

thus *?thesis*

by (*auto simp: Topological-Spaces.Lim-def*)

qed

qed

lemma *eventually-Lim-ident-at*:

$(\forall_F y \text{ in at } x \text{ within } X. P (\text{Lim (at } x \text{ within } X) (\lambda x. x)) y) \longleftrightarrow$

$(\forall_F y \text{ in at } x \text{ within } X. P x y)$ **for** $x::'a::t2\text{-space}$

by (*cases at x within X = bot*) (*auto simp: Lim-ident-at*)

lemma *filterlim-at-bot-at-right*:

fixes $f :: 'a::\text{linorder-topology} \Rightarrow 'b::\text{linorder}$

assumes *mono*: $\bigwedge x y. Q x \implies Q y \implies x \leq y \implies f x \leq f y$

and *bij*: $\bigwedge x. P x \implies f (g x) = x \wedge x. P x \implies Q (g x)$

and *Q*: *eventually* *Q* (*at-right a*)

and *bound*: $\bigwedge b. Q b \implies a < b$

and *P*: *eventually* *P* *at-bot*

shows *filterlim f at-bot (at-right a)*

proof –

from P **obtain** x **where** $x: \bigwedge y. y \leq x \implies P y$
unfolding *eventually-at-bot-linorder* **by** *auto*
show *?thesis*
proof (*intro filterlim-at-bot-le[THEN iffD2] allI impI*)
fix z
assume $z \leq x$
with x **have** $P z$ **by** *auto*
have *eventually* $(\lambda x. x \leq g z)$ (*at-right a*)
using *bound[OF bij(2)[OF ‹P z›]*
unfolding *eventually-at-right[OF bound[OF bij(2)[OF ‹P z›]]*
by (*auto intro!: exI[of - g z]*)
with Q **show** *eventually* $(\lambda x. f x \leq z)$ (*at-right a*)
by *eventually-elim (metis bij ‹P z› mono)*
qed
qed

lemma *filterlim-at-top-at-left*:
fixes $f :: 'a::linorder-topology \Rightarrow 'b::linorder$
assumes *mono*: $\bigwedge x y. Q x \implies Q y \implies x \leq y \implies f x \leq f y$
and *bij*: $\bigwedge x. P x \implies f (g x) = x \wedge \bigwedge x. P x \implies Q (g x)$
and *Q*: *eventually* Q (*at-left a*)
and *bound*: $\bigwedge b. Q b \implies b < a$
and *P*: *eventually* P *at-top*
shows *filterlim f at-top (at-left a)*
proof –
from P **obtain** x **where** $x: \bigwedge y. x \leq y \implies P y$
unfolding *eventually-at-top-linorder* **by** *auto*
show *?thesis*
proof (*intro filterlim-at-top-ge[THEN iffD2] allI impI*)
fix z
assume $x \leq z$
with x **have** $P z$ **by** *auto*
have *eventually* $(\lambda x. g z \leq x)$ (*at-left a*)
using *bound[OF bij(2)[OF ‹P z›]*
unfolding *eventually-at-left[OF bound[OF bij(2)[OF ‹P z›]]*
by (*auto intro!: exI[of - g z]*)
with Q **show** *eventually* $(\lambda x. z \leq f x)$ (*at-left a*)
by *eventually-elim (metis bij ‹P z› mono)*
qed
qed

lemma *filterlim-split-at*:
 $filterlim f F (at-left x) \implies filterlim f F (at-right x) \implies$
 $filterlim f F (at x)$
for $x :: 'a::linorder-topology$
by (*subst at-eq-sup-left-right*) (*rule filterlim-sup*)

lemma *filterlim-at-split*:
 $filterlim f F (at x) \iff filterlim f F (at-left x) \wedge filterlim f F (at-right x)$

for $x :: 'a::linorder-topology$
 by (subst at-eq-sup-left-right) (simp add: filterlim-def filtermap-sup)

lemma eventually-nhds-top:

fixes $P :: 'a :: \{order-top, linorder-topology\} \Rightarrow bool$

and $b :: 'a$

assumes $b < top$

shows eventually P (nhds top) $\longleftrightarrow (\exists b < top. (\forall z. b < z \longrightarrow P z))$

unfolding eventually-nhds

proof safe

fix $S :: 'a$ set

assume open S top $\in S$

note open-left[OF this $\langle b < top \rangle$]

moreover assume $\forall s \in S. P s$

ultimately show $\exists b < top. \forall z > b. P z$

by (auto simp: subset-eq Ball-def)

next

fix b

assume $b < top \forall z > b. P z$

then show $\exists S. open S \wedge top \in S \wedge (\forall xa \in S. P xa)$

by (intro exI[of - $\{b < ..\}$]) auto

qed

lemma tendsto-at-within-iff-tendsto-nhds:

$(g \longrightarrow g l)$ (at l within S) $\longleftrightarrow (g \longrightarrow g l)$ (inf (nhds l) (principal S))

unfolding tendsto-def eventually-at-filter eventually-inf-principal

by (intro ext all-cong imp-cong) (auto elim!: eventually-mono)

99.6 Limits on sequences

abbreviation (in topological-space)

LIMSEQ :: $[nat \Rightarrow 'a, 'a] \Rightarrow bool$ $(((-)/ \longrightarrow (-)) [60, 60] 60)$

where $X \longrightarrow L \equiv (X \longrightarrow L)$ sequentially

abbreviation (in t2-space) $lim :: (nat \Rightarrow 'a) \Rightarrow 'a$

where $lim X \equiv Lim$ sequentially X

definition (in topological-space) $convergent :: (nat \Rightarrow 'a) \Rightarrow bool$

where $convergent X = (\exists L. X \longrightarrow L)$

lemma lim-def: $lim X = (THE L. X \longrightarrow L)$

unfolding Lim-def ..

lemma lim-explicit:

$f \longrightarrow f0 \longleftrightarrow (\forall S. open S \longrightarrow f0 \in S \longrightarrow (\exists N. \forall n \geq N. f n \in S))$

unfolding tendsto-def eventually-sequentially **by** auto

99.7 Monotone sequences and subsequences

Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing first.

definition *monoseq* :: (nat \Rightarrow 'a::order) \Rightarrow bool
where *monoseq* X \longleftrightarrow ($\forall m. \forall n \geq m. X m \leq X n$) \vee ($\forall m. \forall n \geq m. X n \leq X m$)

abbreviation *incseq* :: (nat \Rightarrow 'a::order) \Rightarrow bool
where *incseq* X \equiv *mono* X

lemma *incseq-def*: *incseq* X \longleftrightarrow ($\forall m. \forall n \geq m. X n \geq X m$)
unfolding *mono-def* ..

abbreviation *decseq* :: (nat \Rightarrow 'a::order) \Rightarrow bool
where *decseq* X \equiv *antimono* X

lemma *decseq-def*: *decseq* X \longleftrightarrow ($\forall m. \forall n \geq m. X n \leq X m$)
unfolding *antimono-def* ..

99.7.1 Definition of subsequence.

lemma *strict-mono-leD*: *strict-mono* r \Longrightarrow $m \leq n \Longrightarrow r m \leq r n$
by (*erule* (1) *monoD* [OF *strict-mono-mono*])

lemma *strict-mono-id*: *strict-mono* id
by (*simp* *add*: *strict-mono-def*)

lemma *incseq-SucI*: ($\bigwedge n. X n \leq X (Suc n)$) \Longrightarrow *incseq* X
using *lift-Suc-mono-le*[of X] **by** (*auto* *simp*: *incseq-def*)

lemma *incseqD*: *incseq* f \Longrightarrow $i \leq j \Longrightarrow f i \leq f j$
by (*auto* *simp*: *incseq-def*)

lemma *incseq-SucD*: *incseq* A \Longrightarrow $A i \leq A (Suc i)$
using *incseqD*[of A i Suc i] **by** *auto*

lemma *incseq-Suc-iff*: *incseq* f \longleftrightarrow ($\forall n. f n \leq f (Suc n)$)
by (*auto* *intro*: *incseq-SucI* *dest*: *incseq-SucD*)

lemma *incseq-const*[*simp*, *intro*]: *incseq* ($\lambda x. k$)
unfolding *incseq-def* **by** *auto*

lemma *decseq-SucI*: ($\bigwedge n. X (Suc n) \leq X n$) \Longrightarrow *decseq* X
using *order.lift-Suc-mono-le*[OF *dual-order*, of X] **by** (*auto* *simp*: *decseq-def*)

lemma *decseqD*: *decseq* f \Longrightarrow $i \leq j \Longrightarrow f j \leq f i$
by (*auto* *simp*: *decseq-def*)

lemma *decseq-SucD*: $\text{decseq } A \implies A (\text{Suc } i) \leq A i$
using *decseqD*[of *A i Suc i*] **by** *auto*

lemma *decseq-Suc-iff*: $\text{decseq } f \iff (\forall n. f (\text{Suc } n) \leq f n)$
by (*auto intro: decseq-SucI dest: decseq-SucD*)

lemma *decseq-const*[*simp, intro*]: $\text{decseq } (\lambda x. k)$
unfolding *decseq-def* **by** *auto*

lemma *monoseq-iff*: $\text{monoseq } X \iff \text{incseq } X \vee \text{decseq } X$
unfolding *monoseq-def incseq-def decseq-def* **..**

lemma *monoseq-Suc*: $\text{monoseq } X \iff (\forall n. X n \leq X (\text{Suc } n)) \vee (\forall n. X (\text{Suc } n) \leq X n)$
unfolding *monoseq-iff incseq-Suc-iff decseq-Suc-iff* **..**

lemma *monoI1*: $\forall m. \forall n \geq m. X m \leq X n \implies \text{monoseq } X$
by (*simp add: monoseq-def*)

lemma *monoI2*: $\forall m. \forall n \geq m. X n \leq X m \implies \text{monoseq } X$
by (*simp add: monoseq-def*)

lemma *mono-SucI1*: $\forall n. X n \leq X (\text{Suc } n) \implies \text{monoseq } X$
by (*simp add: monoseq-Suc*)

lemma *mono-SucI2*: $\forall n. X (\text{Suc } n) \leq X n \implies \text{monoseq } X$
by (*simp add: monoseq-Suc*)

lemma *monoseq-minus*:
fixes *a* :: *nat* \Rightarrow '*a*::*ordered-ab-group-add*
assumes *monoseq a*
shows $\text{monoseq } (\lambda n. - a n)$
proof (*cases* $\forall m. \forall n \geq m. a m \leq a n$)
case *True*
then have $\forall m. \forall n \geq m. - a n \leq - a m$ **by** *auto*
then show *?thesis* **by** (*rule monoI2*)
next
case *False*
then have $\forall m. \forall n \geq m. - a m \leq - a n$
using $\langle \text{monoseq } a \rangle$ [*unfolded monoseq-def*] **by** *auto*
then show *?thesis* **by** (*rule monoI1*)
qed

99.7.2 Subsequence (alternative definition, (e.g. Hoskins))

For any sequence, there is a monotonic subsequence.

lemma *seq-monosub*:
fixes *s* :: *nat* \Rightarrow '*a*::*linorder*

shows $\exists f. \text{strict-mono } f \wedge \text{monoseq } (\lambda n. (s (f n)))$
proof (*cases* $\forall n. \exists p > n. \forall m \geq p. s m \leq s p$)
case *True*
then have $\exists f. \forall n. (\forall m \geq f n. s m \leq s (f n)) \wedge f n < f (Suc n)$
by (*intro dependent-nat-choice*) (*auto simp: conj-commute*)
then obtain $f :: \text{nat} \Rightarrow \text{nat}$
where $f: \text{strict-mono } f$ **and** $\text{mono}: \bigwedge n m. f n \leq m \implies s m \leq s (f n)$
by (*auto simp: strict-mono-Suc-iff*)
then have *incseq f*
unfolding *strict-mono-Suc-iff incseq-Suc-iff* **by** (*auto intro: less-imp-le*)
then have *monoseq* $(\lambda n. s (f n))$
by (*auto simp add: incseq-def intro!: mono monoI2*)
with f show *?thesis*
by *auto*
next
case *False*
then obtain N **where** $N: p > N \implies \exists m > p. s p < s m$ **for** p
by (*force simp: not-le le-less*)
have $\exists f. \forall n. N < f n \wedge f n < f (Suc n) \wedge s (f n) \leq s (f (Suc n))$
proof (*intro dependent-nat-choice*)
fix x
assume $N < x$ **with** $N[\text{of } x]$
show $\exists y > N. x < y \wedge s x \leq s y$
by (*auto intro: less-trans*)
qed *auto*
then show *?thesis*
by (*auto simp: monoseq-iff incseq-Suc-iff strict-mono-Suc-iff*)
qed

lemma *seq-suble*:
assumes $sf: \text{strict-mono } (f :: \text{nat} \Rightarrow \text{nat})$
shows $n \leq f n$
proof (*induct n*)
case 0
show *?case* **by** *simp*
next
case $(Suc n)$
with sf [*unfolded strict-mono-Suc-iff, rule-format, of n*] **have** $n < f (Suc n)$
by *arith*
then show *?case* **by** *arith*
qed

lemma *eventually-subseq*:
 $\text{strict-mono } r \implies \text{eventually } P \text{ sequentially} \implies \text{eventually } (\lambda n. P (r n)) \text{ sequentially}$
unfolding *eventually-sequentially* **by** (*metis seq-suble le-trans*)

lemma *not-eventually-sequentiallyD*:
assumes $\neg \text{eventually } P \text{ sequentially}$

shows $\exists r :: \text{nat} \Rightarrow \text{nat}. \text{strict-mono } r \wedge (\forall n. \neg P (r n))$

proof –

from *assms* **have** $\forall n. \exists m \geq n. \neg P m$

unfolding *eventually-sequentially* **by** (*simp add: not-less*)

then obtain *r* **where** $\bigwedge n. r n \geq n \wedge \bigwedge n. \neg P (r n)$

by (*auto simp: choice-iff*)

then show *?thesis*

by (*auto intro!: exI[of - $\lambda n. r ((\text{Suc } \circ r) \sim \text{Suc } n) 0]$*)

simp: less-eq-Suc-le strict-mono-Suc-iff)

qed

lemma *sequentially-offset*:

assumes *eventually* $(\lambda i. P i)$ *sequentially*

shows *eventually* $(\lambda i. P (i + k))$ *sequentially*

using *assms* **by** (*rule eventually-sequentially-seg [THEN iffD2]*)

lemma *seq-offset-neg*:

$(f \longrightarrow l)$ *sequentially* $\implies ((\lambda i. f(i - k)) \longrightarrow l)$ *sequentially*

apply (*erule filterlim-compose*)

apply (*simp add: filterlim-def le-sequentially eventually-filtermap eventually-sequentially, arith*)

done

lemma *filterlim-subseq*: *strict-mono* *f* \implies *filterlim* *f* *sequentially sequentially*

unfolding *filterlim-iff* **by** (*metis eventually-subseq*)

lemma *strict-mono-o*: *strict-mono* *r* \implies *strict-mono* *s* \implies *strict-mono* $(r \circ s)$

unfolding *strict-mono-def* **by** *simp*

lemma *strict-mono-compose*: *strict-mono* *r* \implies *strict-mono* *s* \implies *strict-mono* $(\lambda x. r (s x))$

using *strict-mono-o[of r s]* **by** (*simp add: o-def*)

lemma *incseq-imp-monoseq*: *incseq* *X* \implies *monoseq* *X*

by (*simp add: incseq-def monoseq-def*)

lemma *decseq-imp-monoseq*: *decseq* *X* \implies *monoseq* *X*

by (*simp add: decseq-def monoseq-def*)

lemma *decseq-eq-incseq*: *decseq* *X* = *incseq* $(\lambda n. - X n)$

for *X* :: *nat* \Rightarrow '*a*::*ordered-ab-group-add*

by (*simp add: decseq-def incseq-def*)

lemma *INT-decseq-offset*:

assumes *decseq* *F*

shows $(\bigcap i. F i) = (\bigcap i \in \{n.. \}. F i)$

proof *safe*

fix *x i*

assume *x*: $x \in (\bigcap i \in \{n.. \}. F i)$

show $x \in F i$

proof *cases*

from x **have** $x \in F n$ **by** *auto*

also assume $i \leq n$ **with** $\langle \text{decseq } F \rangle$ **have** $F n \subseteq F i$

unfolding *decseq-def* **by** *simp*

finally show *?thesis* .

qed (*insert x, simp*)

qed *auto*

lemma *LIMSEQ-const-iff*: $(\lambda n. k) \longrightarrow l \longleftrightarrow k = l$

for $k l :: 'a::t2\text{-space}$

using *trivial-limit-sequentially* **by** (*rule tendsto-const-iff*)

lemma *LIMSEQ-SUP*: $\text{incseq } X \Longrightarrow X \longrightarrow (\text{SUP } i. X i :: 'a::\{\text{complete-linorder, linorder-topology}\})$

by (*intro increasing-tendsto*)

(*auto simp: SUP-upper less-SUP-iff incseq-def eventually-sequentially intro:*

less-le-trans)

lemma *LIMSEQ-INF*: $\text{decseq } X \Longrightarrow X \longrightarrow (\text{INF } i. X i :: 'a::\{\text{complete-linorder, linorder-topology}\})$

by (*intro decreasing-tendsto*)

(*auto simp: INF-lower INF-less-iff decseq-def eventually-sequentially intro:*

le-less-trans)

lemma *LIMSEQ-ignore-initial-segment*: $f \longrightarrow a \Longrightarrow (\lambda n. f (n + k)) \longrightarrow a$

unfolding *tendsto-def* **by** (*subst eventually-sequentially-seg[where k=k]*)

lemma *LIMSEQ-offset*: $(\lambda n. f (n + k)) \longrightarrow a \Longrightarrow f \longrightarrow a$

unfolding *tendsto-def*

by (*subst (asm) eventually-sequentially-seg[where k=k]*)

lemma *LIMSEQ-Suc*: $f \longrightarrow l \Longrightarrow (\lambda n. f (\text{Suc } n)) \longrightarrow l$

by (*drule LIMSEQ-ignore-initial-segment [where k=Suc 0]*) *simp*

lemma *LIMSEQ-imp-Suc*: $(\lambda n. f (\text{Suc } n)) \longrightarrow l \Longrightarrow f \longrightarrow l$

by (*rule LIMSEQ-offset [where k=Suc 0]*) *simp*

lemma *LIMSEQ-lessThan-iff-atMost*:

shows $(\lambda n. f \{.. $n\}) \longrightarrow x \longleftrightarrow (\lambda n. f \{.. $n\}) \longrightarrow x$$$

apply (*subst filterlim-sequentially-Suc [symmetric]*)

apply (*simp only: lessThan-Suc-atMost*)

done

lemma (**in** *t2-space*) *LIMSEQ-Uniq*: $\exists \leq_1 l. X \longrightarrow l$

by (*simp add: tendsto-unique'*)

lemma (**in** *t2-space*) *LIMSEQ-unique*: $X \longrightarrow a \Longrightarrow X \longrightarrow b \Longrightarrow a = b$

using *trivial-limit-sequentially* **by** (*rule tendsto-unique*)

lemma *LIMSEQ-le-const*: $X \longrightarrow x \Longrightarrow \exists N. \forall n \geq N. a \leq X n \Longrightarrow a \leq x$

for $a\ x :: 'a::\text{linorder-topology}$
by (*simp add: eventually-at-top-linorder tendsto-lowerbound*)

lemma *LIMSEQ-le*: $X \longrightarrow x \implies Y \longrightarrow y \implies \exists N. \forall n \geq N. X\ n \leq Y\ n \implies x \leq y$

for $x\ y :: 'a::\text{linorder-topology}$
using *tendsto-le[of sequentially Y y X x]* **by** (*simp add: eventually-sequentially*)

lemma *LIMSEQ-le-const2*: $X \longrightarrow x \implies \exists N. \forall n \geq N. X\ n \leq a \implies x \leq a$

for $a\ x :: 'a::\text{linorder-topology}$
by (*rule LIMSEQ-le[of X x $\lambda n. a$]*) *auto*

lemma *Lim-bounded*: $f \longrightarrow l \implies \forall n \geq M. f\ n \leq C \implies l \leq C$

for $l :: 'a::\text{linorder-topology}$
by (*intro LIMSEQ-le-const2*) *auto*

lemma *Lim-bounded2*:

fixes $f :: \text{nat} \Rightarrow 'a::\text{linorder-topology}$
assumes $\text{lim}: f \longrightarrow l$ **and** $\text{ge}: \forall n \geq N. f\ n \geq C$
shows $l \geq C$

using *ge*
by (*intro tendsto-le[OF trivial-limit-sequentially lim tendsto-const]*)
(auto simp: eventually-sequentially)

lemma *lim-mono*:

fixes $X\ Y :: \text{nat} \Rightarrow 'a::\text{linorder-topology}$
assumes $\bigwedge n. N \leq n \implies X\ n \leq Y\ n$
and $X \longrightarrow x$
and $Y \longrightarrow y$
shows $x \leq y$
using *assms(1)* **by** (*intro LIMSEQ-le[OF assms(2,3)]*) *auto*

lemma *Sup-lim*:

fixes $a :: 'a::\{\text{complete-linorder, linorder-topology}\}$
assumes $\bigwedge n. b\ n \in s$
and $b \longrightarrow a$
shows $a \leq \text{Sup } s$
by (*metis Lim-bounded assms complete-lattice-class.Sup-upper*)

lemma *Inf-lim*:

fixes $a :: 'a::\{\text{complete-linorder, linorder-topology}\}$
assumes $\bigwedge n. b\ n \in s$
and $b \longrightarrow a$
shows $\text{Inf } s \leq a$
by (*metis Lim-bounded2 assms complete-lattice-class.Inf-lower*)

lemma *SUP-Lim*:

fixes $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder, linorder-topology}\}$
assumes *inc: incseq X*

and $l: X \longrightarrow l$
shows $(SUP\ n.\ X\ n) = l$
using $LIMSEQ-SUP[OF\ inc]\ tendsto-unique[OF\ trivial-limit-sequentially\ l]$
by $simp$

lemma $INF-Lim$:

fixes $X :: nat \Rightarrow 'a::\{complete-linorder,linorder-topology\}$
assumes $dec: decseq\ X$
and $l: X \longrightarrow l$
shows $(INF\ n.\ X\ n) = l$
using $LIMSEQ-INF[OF\ dec]\ tendsto-unique[OF\ trivial-limit-sequentially\ l]$
by $simp$

lemma $convergentD$: $convergent\ X \implies \exists L.\ X \longrightarrow L$
by $(simp\ add: convergent-def)$

lemma $convergentI$: $X \longrightarrow L \implies convergent\ X$
by $(auto\ simp\ add: convergent-def)$

lemma $convergent-LIMSEQ-iff$: $convergent\ X \longleftrightarrow X \longrightarrow lim\ X$
by $(auto\ intro: theI\ LIMSEQ-unique\ simp\ add: convergent-def\ lim-def)$

lemma $convergent-const$: $convergent\ (\lambda n.\ c)$
by $(rule\ convergentI)\ (rule\ tendsto-const)$

lemma $monoseq-le$:

$monoseq\ a \implies a \longrightarrow x \implies$
 $(\forall n.\ a\ n \leq x) \wedge (\forall m.\ \forall n \geq m.\ a\ m \leq a\ n) \vee$
 $(\forall n.\ x \leq a\ n) \wedge (\forall m.\ \forall n \geq m.\ a\ n \leq a\ m)$
for $x :: 'a::linorder-topology$
by $(metis\ LIMSEQ-le-const\ LIMSEQ-le-const2\ decseq-def\ incseq-def\ monoseq-iff)$

lemma $LIMSEQ-subseq-LIMSEQ$: $X \longrightarrow L \implies strict-mono\ f \implies (X \circ f) \longrightarrow L$
unfolding $comp-def$ **by** $(rule\ filterlim-compose\ [of\ X,\ OF - filterlim-subseq])$

lemma $convergent-subseq-convergent$: $convergent\ X \implies strict-mono\ f \implies convergent\ (X \circ f)$
by $(auto\ simp: convergent-def\ intro: LIMSEQ-subseq-LIMSEQ)$

lemma $limI$: $X \longrightarrow L \implies lim\ X = L$
by $(rule\ tendsto-Lim)\ (rule\ trivial-limit-sequentially)$

lemma $lim-le$: $convergent\ f \implies (\bigwedge n.\ f\ n \leq x) \implies lim\ f \leq x$
for $x :: 'a::linorder-topology$
using $LIMSEQ-le-const2[of\ f\ lim\ f\ x]$ **by** $(simp\ add: convergent-LIMSEQ-iff)$

lemma $lim-const$ $[simp]$: $lim\ (\lambda m.\ a) = a$
by $(simp\ add: limI)$

99.7.3 Increasing and Decreasing Series

lemma *incseq-le*: $\text{incseq } X \Longrightarrow X \longrightarrow L \Longrightarrow X n \leq L$
 for $L :: 'a::\text{linorder-topology}$
 by (*metis incseq-def LIMSEQ-le-const*)

lemma *decseq-ge*: $\text{decseq } X \Longrightarrow X \longrightarrow L \Longrightarrow L \leq X n$
 for $L :: 'a::\text{linorder-topology}$
 by (*metis decseq-def LIMSEQ-le-const2*)

99.8 First countable topologies

class *first-countable-topology* = *topological-space* +
assumes *first-countable-basis*:

$\exists A::\text{nat} \Rightarrow 'a \text{ set. } (\forall i. x \in A i \wedge \text{open } (A i)) \wedge (\forall S. \text{open } S \wedge x \in S \longrightarrow (\exists i. A i \subseteq S))$

lemma (in *first-countable-topology*) *countable-basis-at-decseq*:

obtains $A :: \text{nat} \Rightarrow 'a \text{ set}$ **where**

$\bigwedge i. \text{open } (A i) \bigwedge i. x \in (A i)$

$\bigwedge S. \text{open } S \Longrightarrow x \in S \Longrightarrow \text{eventually } (\lambda i. A i \subseteq S) \text{ sequentially}$

proof *atomize-elim*

from *first-countable-basis*[of x] **obtain** $A :: \text{nat} \Rightarrow 'a \text{ set}$

where *nhds*: $\bigwedge i. \text{open } (A i) \bigwedge i. x \in A i$

and *incl*: $\bigwedge S. \text{open } S \Longrightarrow x \in S \Longrightarrow \exists i. A i \subseteq S$

by *auto*

define F **where** $F n = (\bigcap_{i \leq n}. A i)$ **for** n

show $\exists A. (\forall i. \text{open } (A i)) \wedge (\forall i. x \in A i) \wedge$

$(\forall S. \text{open } S \longrightarrow x \in S \longrightarrow \text{eventually } (\lambda i. A i \subseteq S) \text{ sequentially})$

proof (*safe intro!*: *exI*[of F])

fix i

show $\text{open } (F i)$

using *nhds(1)* **by** (*auto simp: F-def*)

show $x \in F i$

using *nhds(2)* **by** (*auto simp: F-def*)

next

fix S

assume $\text{open } S \wedge x \in S$

from *incl*[OF *this*] **obtain** i **where** $F i \subseteq S$

unfolding *F-def* **by** *auto*

moreover **have** $\bigwedge j. i \leq j \Longrightarrow F j \subseteq F i$

by (*simp add: Inf-superset-mono F-def image-mono*)

ultimately **show** $\text{eventually } (\lambda i. F i \subseteq S) \text{ sequentially}$

by (*auto simp: eventually-sequentially*)

qed

qed

lemma (in *first-countable-topology*) *nhds-countable*:

obtains $X :: \text{nat} \Rightarrow 'a \text{ set}$

where $\text{decseq } X \bigwedge n. \text{open } (X n) \bigwedge n. x \in X n$ *nhds* $x = (\text{INF } n. \text{principal } (X$

n)

proof –

from *first-countable-basis* **obtain** $A :: \text{nat} \Rightarrow 'a \text{ set}$

where $*$: $\bigwedge n. x \in A \ n \ \bigwedge n. \text{open} \ (A \ n) \ \bigwedge S. \text{open} \ S \implies x \in S \implies \exists i. A \ i \subseteq S$

by *metis*

show *thesis*

proof

show *decseq* $(\lambda n. \bigcap_{i \leq n}. A \ i)$

by (*simp add: antimono-iff-le-Suc atMost-Suc*)

show $x \in (\bigcap_{i \leq n}. A \ i) \ \bigwedge n. \text{open} \ (\bigcap_{i \leq n}. A \ i)$ **for** n

using $*$ **by** *auto*

with $*$ **show** *nhds* $x = (\text{INF } n. \text{principal} \ (\bigcap_{i \leq n}. A \ i))$

unfolding *nhds-def*

apply (*intro INF-eq*)

apply *fastforce*

apply *blast*

done

qed

qed

lemma (*in first-countable-topology*) *countable-basis*:

obtains $A :: \text{nat} \Rightarrow 'a \text{ set}$ **where**

$\bigwedge i. \text{open} \ (A \ i) \ \bigwedge i. x \in A \ i$

$\bigwedge F. (\forall n. F \ n \in A \ n) \implies F \longrightarrow x$

proof *atomize-elim*

obtain $A :: \text{nat} \Rightarrow 'a \text{ set}$ **where** $*$:

$\bigwedge i. \text{open} \ (A \ i)$

$\bigwedge i. x \in A \ i$

$\bigwedge S. \text{open} \ S \implies x \in S \implies \text{eventually} \ (\lambda i. A \ i \subseteq S)$ *sequentially*

by (*rule countable-basis-at-decseq*) *blast*

have *eventually* $(\lambda n. F \ n \in S)$ *sequentially*

if $\forall n. F \ n \in A \ n \ \text{open} \ S \ x \in S$ **for** $F \ S$

using $*$ (β)[*of S*] **that** **by** (*auto elim: eventually-mono simp: subset-eq*)

with $*$ **show** $\exists A. (\forall i. \text{open} \ (A \ i)) \wedge (\forall i. x \in A \ i) \wedge (\forall F. (\forall n. F \ n \in A \ n) \longrightarrow F \longrightarrow x)$

by (*intro exI[of - A]*) (*auto simp: tendsto-def*)

qed

lemma (*in first-countable-topology*) *sequentially-imp-eventually-nhds-within*:

assumes $\forall f. (\forall n. f \ n \in s) \wedge f \longrightarrow a \longrightarrow \text{eventually} \ (\lambda n. P \ (f \ n))$ *sequentially*

shows *eventually* $P \ (\text{inf} \ (\text{nhds} \ a) \ (\text{principal} \ s))$

proof (*rule ccontr*)

obtain $A :: \text{nat} \Rightarrow 'a \text{ set}$ **where** $*$:

$\bigwedge i. \text{open} \ (A \ i)$

$\bigwedge i. a \in A \ i$

$\bigwedge F. \forall n. F \ n \in A \ n \implies F \longrightarrow a$

by (*rule countable-basis*) *blast*

assume \neg *thesis*

with * have $\exists F. \forall n. F\ n \in s \wedge F\ n \in A\ n \wedge \neg P\ (F\ n)$
unfolding *eventually-inf-principal eventually-nhds*
by (*intro choice*) *fastforce*
then obtain F **where** $F: \forall n. F\ n \in s$ **and** $\forall n. F\ n \in A\ n$ **and** $F': \forall n. \neg P\ (F\ n)$
by *blast*
with * have $F \longrightarrow a$
by *auto*
then have *eventually* $(\lambda n. P\ (F\ n))$ *sequentially*
using *assms F* **by** *simp*
then show *False*
by (*simp add: F'*)
qed

lemma (*in first-countable-topology*) *eventually-nhds-within-iff-sequentially*:
eventually $P\ (inf\ (nhds\ a)\ (principal\ s)) \longleftrightarrow$
 $(\forall f. (\forall n. f\ n \in s) \wedge f \longrightarrow a \longrightarrow \text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially})$
proof (*safe intro!: sequentially-imp-eventually-nhds-within*)
assume *eventually* $P\ (inf\ (nhds\ a)\ (principal\ s))$
then obtain S **where** *open* $S\ a \in S\ \forall x \in S. x \in s \longrightarrow P\ x$
by (*auto simp: eventually-inf-principal eventually-nhds*)
moreover
fix f
assume $\forall n. f\ n \in s\ f \longrightarrow a$
ultimately show *eventually* $(\lambda n. P\ (f\ n))$ *sequentially*
by (*auto dest!: topological-tendstoD elim: eventually-mono*)
qed

lemma (*in first-countable-topology*) *eventually-nhds-iff-sequentially*:
eventually $P\ (nhds\ a) \longleftrightarrow (\forall f. f \longrightarrow a \longrightarrow \text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially})$
using *eventually-nhds-within-iff-sequentially[of P a UNIV]* **by** *simp*

lemma *Inf-as-limit*:
fixes $A::'a::\{linorder-topology, first-countable-topology, complete-linorder\}$ *set*
assumes $A \neq \{\}$
shows $\exists u. (\forall n. u\ n \in A) \wedge u \longrightarrow Inf\ A$
proof (*cases Inf A ∈ A*)
case *True*
show *?thesis*
by (*rule exI[of - λn. Inf A], auto simp add: True*)
next
case *False*
obtain y **where** $y \in A$ **using** *assms* **by** *auto*
then have $Inf\ A < y$ **using** *False Inf-lower less-le* **by** *auto*
obtain $F :: nat \Rightarrow 'a$ **set** **where** $F: \bigwedge i. open\ (F\ i) \bigwedge i. Inf\ A \in F\ i$
 $\bigwedge u. (\forall n. u\ n \in F\ n) \Longrightarrow u \longrightarrow Inf\ A$
by (*metis first-countable-topology-class.countable-basis*)

define u **where** $u = (\lambda n. \text{SOME } z. z \in F\ n \wedge z \in A)$
have $\exists z. z \in U \wedge z \in A$ **if** $\text{Inf } A \in U$ **open** U **for** U
proof –
obtain b **where** $b > \text{Inf } A \{ \text{Inf } A .. < b \} \subseteq U$
using $\text{open-right}[OF \langle \text{open } U \rangle \langle \text{Inf } A \in U \rangle \langle \text{Inf } A < y \rangle]$ **by** *auto*
obtain z **where** $z < b \ z \in A$
using $\langle \text{Inf } A < b \rangle$ *Inf-less-iff* **by** *auto*
then have $z \in \{ \text{Inf } A .. < b \}$
by (*simp add: Inf-lower*)
then show *?thesis* **using** $\langle z \in A \rangle \langle \{ \text{Inf } A .. < b \} \subseteq U \rangle$ **by** *auto*
qed
then have $*$: $u\ n \in F\ n \wedge u\ n \in A$ **for** n
using $\langle \text{Inf } A \in F\ n \rangle \langle \text{open } (F\ n) \rangle$ **unfolding** $u\text{-def}$ **by** (*metis (no-types, lifting)*
someI-ex)
then have $u \longrightarrow \text{Inf } A$ **using** $F(3)$ **by** *simp*
then show *?thesis* **using** $*$ **by** *auto*
qed

lemma *tendsto-at-iff-sequentially*:

$(f \longrightarrow a) \text{ (at } x \text{ within } s) \longleftrightarrow (\forall X. (\forall i. X\ i \in s - \{x\}) \longrightarrow X \longrightarrow x \longrightarrow$
 $((f \circ X) \longrightarrow a))$
for $f :: 'a :: \text{first-countable-topology} \Rightarrow -$
unfolding *filterlim-def[of - nhds a]* *le-filter-def* *eventually-filtermap*
at-within-def *eventually-nhds-within-iff-sequentially comp-def*
by *metis*

lemma *approx-from-above-dense-linorder*:

fixes $x :: 'a :: \{ \text{dense-linorder, linorder-topology, first-countable-topology} \}$
assumes $x < y$
shows $\exists u. (\forall n. u\ n > x) \wedge (u \longrightarrow x)$

proof –

obtain $A :: \text{nat} \Rightarrow 'a$ **set** **where** $A: \bigwedge i. \text{open } (A\ i) \bigwedge i. x \in A\ i$
 $\bigwedge F. (\forall n. F\ n \in A\ n) \Longrightarrow F \longrightarrow x$
by (*metis first-countable-topology-class.countable-basis*)
define u **where** $u = (\lambda n. \text{SOME } z. z \in A\ n \wedge z > x)$
have $\exists z. z \in U \wedge x < z$ **if** $x \in U$ **open** U **for** U
using $\text{open-right}[OF \langle \text{open } U \rangle \langle x \in U \rangle \langle x < y \rangle]$
by (*meson atLeastLessThan-iff dense less-imp-le subset-eq*)
then have $*$: $u\ n \in A\ n \wedge x < u\ n$ **for** n
using $\langle x \in A\ n \rangle \langle \text{open } (A\ n) \rangle$ **unfolding** $u\text{-def}$ **by** (*metis (no-types, lifting)*
someI-ex)
then have $u \longrightarrow x$ **using** $A(3)$ **by** *simp*
then show *?thesis* **using** $*$ **by** *auto*
qed

lemma *approx-from-below-dense-linorder*:

fixes $x :: 'a :: \{ \text{dense-linorder, linorder-topology, first-countable-topology} \}$
assumes $x > y$
shows $\exists u. (\forall n. u\ n < x) \wedge (u \longrightarrow x)$

proof –

obtain $A :: \text{nat} \Rightarrow 'a \text{ set}$ **where** $A: \bigwedge i. \text{open } (A \ i) \wedge i. x \in A \ i$
 $\bigwedge F. (\forall n. F \ n \in A \ n) \Longrightarrow F \longrightarrow x$
by (*metis first-countable-topology-class.countable-basis*)
define u **where** $u = (\lambda n. \text{SOME } z. z \in A \ n \wedge z < x)$
have $\exists z. z \in U \wedge z < x$ **if** $x \in U$ **open** U **for** U
using *open-left[OF <open U> <x ∈ U> <x > y>]*
by (*meson dense greaterThanAtMost-iff less-imp-le subset-eq*)
then have $*$: $u \ n \in A \ n \wedge u \ n < x$ **for** n
using $\langle x \in A \ n \rangle \langle \text{open } (A \ n) \rangle$ **unfolding** $u\text{-def}$ **by** (*metis (no-types, lifting)*
someI-ex)
then have $u \longrightarrow x$ **using** $A(3)$ **by** *simp*
then show *?thesis* **using** $*$ **by** *auto*
qed

99.9 Function limit at a point

abbreviation $LIM :: ('a::\text{topological-space} \Rightarrow 'b::\text{topological-space}) \Rightarrow 'a \Rightarrow 'b \Rightarrow$
 bool
 $(((-)/ -(-)/\rightarrow (-)) [60, 0, 60] 60)$
where $f \ -a \rightarrow L \equiv (f \longrightarrow L) \text{ (at } a \text{)}$

lemma *tendsto-within-open*: $a \in S \Longrightarrow \text{open } S \Longrightarrow (f \longrightarrow l) \text{ (at } a \text{ within } S)$
 $\longleftrightarrow (f \ -a \rightarrow l)$
by (*simp add: tendsto-def at-within-open[where S = S]*)

lemma *tendsto-within-open-NO-MATCH*:
 $a \in S \Longrightarrow \text{NO-MATCH UNIV } S \Longrightarrow \text{open } S \Longrightarrow (f \longrightarrow l) \text{ (at } a \text{ within } S) \longleftrightarrow$
 $(f \longrightarrow l) \text{ (at } a \text{)}$
for $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{topological-space}$
using *tendsto-within-open* **by** *blast*

lemma *LIM-const-not-eq[tendsto-intros]*: $k \neq L \Longrightarrow \neg (\lambda x. k) \ -a \rightarrow L$
for $a :: 'a::\text{perfect-space}$ **and** $k \ L :: 'b::\text{t2-space}$
by (*simp add: tendsto-const-iff*)

lemmas *LIM-not-zero* = *LIM-const-not-eq* [**where** $L = 0$]

lemma *LIM-const-eq*: $(\lambda x. k) \ -a \rightarrow L \Longrightarrow k = L$
for $a :: 'a::\text{perfect-space}$ **and** $k \ L :: 'b::\text{t2-space}$
by (*simp add: tendsto-const-iff*)

lemma *LIM-unique*: $f \ -a \rightarrow L \Longrightarrow f \ -a \rightarrow M \Longrightarrow L = M$
for $a :: 'a::\text{perfect-space}$ **and** $L \ M :: 'b::\text{t2-space}$
using *at-neq-bot* **by** (*rule tendsto-unique*)

lemma *LIM-Uniq*: $\exists \leq_1 L :: 'b::\text{t2-space}. f \ -a \rightarrow L$
for $a :: 'a::\text{perfect-space}$
by (*auto simp add: Uniq-def LIM-unique*)

Limits are equal for functions equal except at limit point.

lemma *LIM-equal*: $\forall x. x \neq a \longrightarrow f x = g x \Longrightarrow (f -a \rightarrow l) \longleftrightarrow (g -a \rightarrow l)$
by (*simp add: tendsto-def eventually-at-topological*)

lemma *LIM-cong*: $a = b \Longrightarrow (\bigwedge x. x \neq b \Longrightarrow f x = g x) \Longrightarrow l = m \Longrightarrow (f -a \rightarrow l) \longleftrightarrow (g -b \rightarrow m)$
by (*simp add: LIM-equal*)

lemma *tendsto-cong-limit*: $(f \longrightarrow l) F \Longrightarrow k = l \Longrightarrow (f \longrightarrow k) F$
by *simp*

lemma *tendsto-at-iff-tendsto-nhds*: $g -l \rightarrow g l \longleftrightarrow (g \longrightarrow g l) (nhds l)$
unfolding *tendsto-def eventually-at-filter*
by (*intro ext all-cong imp-cong*) (*auto elim!: eventually-mono*)

lemma *tendsto-compose*: $g -l \rightarrow g l \Longrightarrow (f \longrightarrow l) F \Longrightarrow ((\lambda x. g (f x)) \longrightarrow g l) F$
unfolding *tendsto-at-iff-tendsto-nhds* **by** (*rule filterlim-compose[of g]*)

lemma *tendsto-compose-eventually*:
 $g -l \rightarrow m \Longrightarrow (f \longrightarrow l) F \Longrightarrow \text{eventually } (\lambda x. f x \neq l) F \Longrightarrow ((\lambda x. g (f x)) \longrightarrow m) F$
by (*rule filterlim-compose[of g - at l]*) (*auto simp add: filterlim-at*)

lemma *LIM-compose-eventually*:
assumes $f -a \rightarrow b$
and $g -b \rightarrow c$
and *eventually* $(\lambda x. f x \neq b)$ (*at a*)
shows $(\lambda x. g (f x)) -a \rightarrow c$
using *assms(2,1,3)* **by** (*rule tendsto-compose-eventually*)

lemma *tendsto-compose-filtermap*: $((g \circ f) \longrightarrow T) F \longleftrightarrow (g \longrightarrow T) (filtermap f F)$
by (*simp add: filterlim-def filtermap-filtermap comp-def*)

lemma *tendsto-compose-at*:
assumes $f: (f \longrightarrow y) F$ **and** $g: (g \longrightarrow z)$ (*at y*) **and** $fg: \text{eventually } (\lambda w. f w = y \longrightarrow g w = z) F$
shows $((g \circ f) \longrightarrow z) F$
proof –
have $(\forall_F a \text{ in } F. f a \neq y) \vee g y = z$
using fg **by** *force*
moreover have $(g \longrightarrow z) (filtermap f F) \vee \neg (\forall_F a \text{ in } F. f a \neq y)$
by (*metis (no-types) filterlim-atI filterlim-def tendsto-mono f g*)
ultimately show *?thesis*
by (*metis (no-types) f filterlim-compose filterlim-filtermap g tendsto-at-iff-tendsto-nhds tendsto-compose-filtermap*)
qed

lemma *tendsto-nhds-iff*: $(f \longrightarrow (c :: 'a :: t1\text{-space})) (nhds\ x) \longleftrightarrow f \text{-}x \rightarrow c \wedge f\ x = c$
proof *safe*
assume *lim*: $(f \longrightarrow c) (nhds\ x)$
show $f\ x = c$
proof (*rule ccontr*)
assume $f\ x \neq c$
hence $c \neq f\ x$
by *auto*
then obtain *A where* $A: \text{open } A\ c \in A\ f\ x \notin A$
by (*subst (asm) separation-t1*) *auto*
with *lim obtain* *B where* $\text{open } B\ x \in B \wedge x. x \in B \implies f\ x \in A$
unfolding *tendsto-def eventually-nhds* **by** *metis*
with $\langle f\ x \notin A \rangle$ **show** *False*
by *blast*
qed
show $(f \longrightarrow c) (at\ x)$
using *lim* **by** (*rule filterlim-mono*) (*auto simp: at-within-def*)
next
assume $f \text{-}x \rightarrow f\ x\ c = f\ x$
thus $(f \longrightarrow f\ x) (nhds\ x)$
unfolding *tendsto-def eventually-at-filter* **by** (*fast elim: eventually-mono*)
qed

99.9.1 Relation of LIM and LIMSEQ

lemma (*in first-countable-topology*) *sequentially-imp-eventually-within*:
 $(\forall f. (\forall n. f\ n \in s \wedge f\ n \neq a) \wedge f \longrightarrow a \longrightarrow \text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially}) \implies$
 $\text{eventually } P\ (at\ a\ \text{within } s)$
unfolding *at-within-def*
by (*intro sequentially-imp-eventually-nhds-within*) *auto*

lemma (*in first-countable-topology*) *sequentially-imp-eventually-at*:
 $(\forall f. (\forall n. f\ n \neq a) \wedge f \longrightarrow a \longrightarrow \text{eventually } (\lambda n. P\ (f\ n)) \text{ sequentially}) \implies$
 $\text{eventually } P\ (at\ a)$
using *sequentially-imp-eventually-within* [**where** $s = UNIV$] **by** *simp*

lemma *LIMSEQ-SEQ-conv*:
 $(\forall S. (\forall n. S\ n \neq a) \wedge S \longrightarrow a \longrightarrow (\lambda n. X\ (S\ n)) \longrightarrow L) \longleftrightarrow X \text{-}a \rightarrow L$ (*is ?lhs=?rhs*)
for $a :: 'a::\text{first-countable-topology}$ **and** $L :: 'b::\text{topological-space}$
proof
assume *?lhs* **then show** *?rhs*
by (*simp add: sequentially-imp-eventually-within tendsto-def*)
next
assume *?rhs* **then show** *?lhs*
using *tendsto-compose-eventually eventuallyI* **by** *blast*
qed

lemma *sequentially-imp-eventually-at-left*:
fixes $a :: 'a::\{\text{linorder-topology,first-countable-topology}\}$
assumes $b[\text{simp}] : b < a$
and $*$: $\bigwedge f. (\bigwedge n. b < f\ n) \implies (\bigwedge n. f\ n < a) \implies \text{incseq}\ f \implies f \longrightarrow a \implies$
eventually $(\lambda n. P\ (f\ n))$ *sequentially*
shows *eventually* P (*at-left* a)
proof (*safe intro!*: *sequentially-imp-eventually-within*)
fix X
assume $X : \forall n. X\ n \in \{.. < a\} \wedge X\ n \neq a \implies X \longrightarrow a$
show *eventually* $(\lambda n. P\ (X\ n))$ *sequentially*
proof (*rule ccontr*)
assume $\text{neg} : \neg\ ?thesis$
have $\exists s. \forall n. (\neg\ P\ (X\ (s\ n)) \wedge b < X\ (s\ n)) \wedge (X\ (s\ n) \leq X\ (s\ (Suc\ n))) \wedge$
 $Suc\ (s\ n) \leq s\ (Suc\ n)$
(is $\exists s. ?P\ s$ *)*
proof (*rule dependent-nat-choice*)
have $\neg\ \text{eventually}\ (\lambda n. b < X\ n \longrightarrow P\ (X\ n))$ *sequentially*
by (*intro not-eventually-impI neg order-tendstoD(1) [OF X(2) b]*)
then show $\exists x. \neg\ P\ (X\ x) \wedge b < X\ x$
by (*auto dest!*: *not-eventuallyD*)
next
fix $x\ n$
have $\neg\ \text{eventually}\ (\lambda n. Suc\ x \leq n \longrightarrow b < X\ n \longrightarrow X\ x < X\ n \longrightarrow P\ (X\ n))$ *sequentially*
using X
by (*intro not-eventually-impI order-tendstoD(1)[OF X(2)] eventually-ge-at-top*
neg) auto
then show $\exists n. (\neg\ P\ (X\ n) \wedge b < X\ n) \wedge (X\ x \leq X\ n \wedge Suc\ x \leq n)$
by (*auto dest!*: *not-eventuallyD*)
qed
then obtain s **where** $?P\ s ..$
with X **have** $b < X\ (s\ n)$
and $X\ (s\ n) < a$
and $\text{incseq}\ (\lambda n. X\ (s\ n))$
and $(\lambda n. X\ (s\ n)) \longrightarrow a$
and $\neg\ P\ (X\ (s\ n))$
for n
by (*auto simp: strict-mono-Suc-iff Suc-le-eq incseq-Suc-iff*
intro!: *LIMSEQ-subseq-LIMSEQ[OF X(2) a], unfolded comp-def*)
from $*[\text{OF this(1,2,3,4)}]$ this(5) **show** False
by *auto*
qed
qed

lemma *tendsto-at-left-sequentially*:

fixes $a\ b :: 'b::\{\text{linorder-topology,first-countable-topology}\}$

assumes $b < a$

assumes $*$: $\bigwedge S. (\bigwedge n. S\ n < a) \implies (\bigwedge n. b < S\ n) \implies \text{incseq}\ S \implies S \longrightarrow$

$a \implies$
 $(\lambda n. X (S n)) \longrightarrow L$
shows $(X \longrightarrow L)$ (at-left a)
using *assms* **by** (*simp add: tendsto-def* [where $l=L$] *sequentially-imp-eventually-at-left*)

lemma *sequentially-imp-eventually-at-right*:

fixes $a b :: 'a :: \{linorder-topology, first-countable-topology\}$

assumes $b[*simp*]$: $a < b$

assumes $*$: $\bigwedge f. (\bigwedge n. a < f n) \implies (\bigwedge n. f n < b) \implies \text{decseq } f \implies f \longrightarrow a$

\implies

eventually $(\lambda n. P (f n))$ *sequentially*

shows *eventually* P (at-right a)

proof (*safe intro!*: *sequentially-imp-eventually-within*)

fix X

assume X : $\forall n. X n \in \{a <..\} \wedge X n \neq a \implies X \longrightarrow a$

show *eventually* $(\lambda n. P (X n))$ *sequentially*

proof (*rule ccontr*)

assume *neg*: $\neg ?thesis$

have $\exists s. \forall n. (\neg P (X (s n)) \wedge X (s n) < b) \wedge (X (s (Suc n)) \leq X (s n) \wedge$
 $Suc (s n) \leq s (Suc n))$

(**is** $\exists s. ?P s$)

proof (*rule dependent-nat-choice*)

have \neg *eventually* $(\lambda n. X n < b \longrightarrow P (X n))$ *sequentially*

by (*intro not-eventually-impI neg order-tendstoD(2)* [*OF* $X(2)$ b])

then show $\exists x. \neg P (X x) \wedge X x < b$

by (*auto dest!*: *not-eventuallyD*)

next

fix $x n$

have \neg *eventually* $(\lambda n. Suc x \leq n \longrightarrow X n < b \longrightarrow X n < X x \longrightarrow P (X$
 $n))$ *sequentially*

using X

by (*intro not-eventually-impI order-tendstoD(2)*[*OF* $X(2)$] *eventually-ge-at-top*
neg) *auto*

then show $\exists n. (\neg P (X n) \wedge X n < b) \wedge (X n \leq X x \wedge Suc x \leq n)$

by (*auto dest!*: *not-eventuallyD*)

qed

then obtain s **where** $?P s ..$

with X **have** $a < X (s n)$

and $X (s n) < b$

and *decseq* $(\lambda n. X (s n))$

and $(\lambda n. X (s n)) \longrightarrow a$

and $\neg P (X (s n))$

for n

by (*auto simp: strict-mono-Suc-iff Suc-le-eq decseq-Suc-iff*

intro!: *LIMSEQ-subseq-LIMSEQ*[*OF* $\langle X \longrightarrow a \rangle$, *unfolded comp-def*])

from $*[*OF* \text{this}(1,2,3,4)] \text{this}(5)$ **show** *False*

by *auto*

qed

qed

lemma *tendsto-at-right-sequentially*:

fixes $a :: - :: \{\text{linorder-topology, first-countable-topology}\}$

assumes $a < b$

and $*$: $\bigwedge S. (\bigwedge n. a < S\ n) \implies (\bigwedge n. S\ n < b) \implies \text{decseq } S \implies S \longrightarrow a$

$\implies (\lambda n. X\ (S\ n)) \longrightarrow L$

shows $(X \longrightarrow L)$ (*at-right* a)

using *assms* **by** (*simp add: tendsto-def [where l=L] sequentially-imp-eventually-at-right*)

99.10 Continuity

99.10.1 Continuity on a set

definition *continuous-on* $:: 'a\ \text{set} \Rightarrow ('a::\text{topological-space} \Rightarrow 'b::\text{topological-space}) \Rightarrow \text{bool}$

where *continuous-on* $s\ f \longleftrightarrow (\forall x \in s. (f \longrightarrow f\ x))$ (*at* x *within* s)

lemma *continuous-on-cong [cong]*:

$s = t \implies (\bigwedge x. x \in t \implies f\ x = g\ x) \implies \text{continuous-on } s\ f \longleftrightarrow \text{continuous-on } t\ g$

unfolding *continuous-on-def*

by (*intro ball-cong filterlim-cong*) (*auto simp: eventually-at-filter*)

lemma *continuous-on-cong-simp*:

$s = t \implies (\bigwedge x. x \in t \implies \text{simp} \Rightarrow f\ x = g\ x) \implies \text{continuous-on } s\ f \longleftrightarrow \text{continuous-on } t\ g$

unfolding *simp-implies-def* **by** (*rule continuous-on-cong*)

lemma *continuous-on-topological*:

continuous-on $s\ f \longleftrightarrow$

$(\forall x \in s. \forall B. \text{open } B \longrightarrow f\ x \in B \longrightarrow (\exists A. \text{open } A \wedge x \in A \wedge (\forall y \in s. y \in A \longrightarrow f\ y \in B)))$

unfolding *continuous-on-def tendsto-def eventually-at-topological* **by** *metis*

lemma *continuous-on-open-invariant*:

continuous-on $s\ f \longleftrightarrow (\forall B. \text{open } B \longrightarrow (\exists A. \text{open } A \wedge A \cap s = f^{-1} B \cap s))$

proof *safe*

fix $B :: 'b\ \text{set}$

assume *continuous-on* $s\ f\ \text{open } B$

then have $\forall x \in f^{-1} B \cap s. (\exists A. \text{open } A \wedge x \in A \wedge s \cap A \subseteq f^{-1} B)$

by (*auto simp: continuous-on-topological subset-eq Ball-def imp-conjL*)

then obtain A **where** $\forall x \in f^{-1} B \cap s. \text{open } (A\ x) \wedge x \in A\ x \wedge s \cap A\ x \subseteq f^{-1} B$

unfolding *bchoice-iff ..*

then show $\exists A. \text{open } A \wedge A \cap s = f^{-1} B \cap s$

by (*intro exI[of - $\bigcup x \in f^{-1} B \cap s. A\ x$]*) *auto*

next

assume $B: \forall B. \text{open } B \longrightarrow (\exists A. \text{open } A \wedge A \cap s = f^{-1} B \cap s)$

show *continuous-on* $s\ f$

unfolding *continuous-on-topological*
proof *safe*
fix $x B$
assume $x \in s \text{ open } B f x \in B$
with B **obtain** A **where** $A: \text{open } A \ A \cap s = f - ' B \cap s$
by *auto*
with $\langle x \in s \rangle \langle f x \in B \rangle$ **show** $\exists A. \text{open } A \wedge x \in A \wedge (\forall y \in s. y \in A \longrightarrow f y \in B)$
by (*intro exI[of - A]*) *auto*
qed
qed

lemma *continuous-on-open-vimage*:
 $\text{open } s \implies \text{continuous-on } s f \iff (\forall B. \text{open } B \longrightarrow \text{open } (f - ' B \cap s))$
unfolding *continuous-on-open-invariant*
by (*metis open-Int Int-absorb Int-commute[of s] Int-assoc[of - - s]*)

corollary *continuous-imp-open-vimage*:
assumes *continuous-on s f open s open B f - ' B \subseteq s*
shows *open (f - ' B)*
by (*metis assms continuous-on-open-vimage le-iff-inf*)

corollary *open-vimage[continuous-intros]*:
assumes *open s*
and *continuous-on UNIV f*
shows *open (f - ' s)*
using *assms by (simp add: continuous-on-open-vimage [OF open-UNIV])*

lemma *continuous-on-closed-invariant*:
 $\text{continuous-on } s f \iff (\forall B. \text{closed } B \longrightarrow (\exists A. \text{closed } A \wedge A \cap s = f - ' B \cap s))$
proof –
have $*$: $(\bigwedge A. P A \iff Q (- A)) \implies (\forall A. P A) \iff (\forall A. Q A)$
for $P Q :: 'b \text{ set} \Rightarrow \text{bool}$
by (*metis double-compl*)
show *?thesis*
unfolding *continuous-on-open-invariant*
by (*intro **) (*auto simp: open-closed[symmetric]*)
qed

lemma *continuous-on-closed-vimage*:
 $\text{closed } s \implies \text{continuous-on } s f \iff (\forall B. \text{closed } B \longrightarrow \text{closed } (f - ' B \cap s))$
unfolding *continuous-on-closed-invariant*
by (*metis closed-Int Int-absorb Int-commute[of s] Int-assoc[of - - s]*)

corollary *closed-vimage-Int[continuous-intros]*:
assumes *closed s*
and *continuous-on t f*
and *t: closed t*
shows *closed (f - ' s \cap t)*

using *assms* **by** (*simp add: continuous-on-closed-vimage [OF t]*)

corollary *closed-vimage[continuous-intros]*:

assumes *closed s*
and *continuous-on UNIV f*
shows *closed (f -‘ s)*
using *closed-vimage-Int [OF assms]* **by** *simp*

lemma *continuous-on-empty [simp]: continuous-on {} f*
by (*simp add: continuous-on-def*)

lemma *continuous-on-sing [simp]: continuous-on {x} f*
by (*simp add: continuous-on-def at-within-def*)

lemma *continuous-on-open-Union*:

$(\bigwedge s. s \in S \implies \text{open } s) \implies (\bigwedge s. s \in S \implies \text{continuous-on } s f) \implies \text{continuous-on } (\bigcup S) f$
unfolding *continuous-on-def*
by *safe (metis open-Union at-within-open UnionI)*

lemma *continuous-on-open-UN*:

$(\bigwedge s. s \in S \implies \text{open } (A s)) \implies (\bigwedge s. s \in S \implies \text{continuous-on } (A s) f) \implies \text{continuous-on } (\bigcup_{s \in S} A s) f$
by (*rule continuous-on-open-Union*) *auto*

lemma *continuous-on-open-Un*:

$\text{open } s \implies \text{open } t \implies \text{continuous-on } s f \implies \text{continuous-on } t f \implies \text{continuous-on } (s \cup t) f$
using *continuous-on-open-Union [of {s,t}]* **by** *auto*

lemma *continuous-on-closed-Un*:

$\text{closed } s \implies \text{closed } t \implies \text{continuous-on } s f \implies \text{continuous-on } t f \implies \text{continuous-on } (s \cup t) f$
by (*auto simp add: continuous-on-closed-vimage closed-Un Int-Un-distrib*)

lemma *continuous-on-closed-Union*:

assumes *finite I*
 $\bigwedge i. i \in I \implies \text{closed } (U i)$
 $\bigwedge i. i \in I \implies \text{continuous-on } (U i) f$
shows *continuous-on* $(\bigcup i \in I. U i) f$
using *assms*
by (*induction I*) (*auto intro!: continuous-on-closed-Un*)

lemma *continuous-on-If*:

assumes *closed: closed s closed t*
and *cont: continuous-on s f continuous-on t g*
and *P: $\bigwedge x. x \in s \implies \neg P x \implies f x = g x \bigwedge x. x \in t \implies P x \implies f x = g x$*
shows *continuous-on* $(s \cup t) (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x)$
(is continuous-on - ?h)

proof–

from P **have** $\forall x \in s. f\ x = ?h\ x \ \forall x \in t. g\ x = ?h\ x$
by *auto*
with *cont* **have** *continuous-on s ?h continuous-on t ?h*
by *simp-all*
with *closed* **show** *?thesis*
by (*rule continuous-on-closed-Un*)

qed

lemma *continuous-on-cases*:

*closed s \implies closed t \implies continuous-on s f \implies continuous-on t g \implies
 $\forall x. (x \in s \wedge \neg P\ x) \vee (x \in t \wedge P\ x) \longrightarrow f\ x = g\ x \implies$
continuous-on (s \cup t) ($\lambda x. \text{if } P\ x \text{ then } f\ x \text{ else } g\ x)$
by (*rule continuous-on-If*) *auto**

lemma *continuous-on-id*[*continuous-intros, simp*]: *continuous-on s ($\lambda x. x$)*
unfolding *continuous-on-def* **by** *fast*

lemma *continuous-on-id'*[*continuous-intros, simp*]: *continuous-on s id*
unfolding *continuous-on-def id-def* **by** *fast*

lemma *continuous-on-const*[*continuous-intros, simp*]: *continuous-on s ($\lambda x. c$)*
unfolding *continuous-on-def* **by** *auto*

lemma *continuous-on-subset*: *continuous-on s f \implies t \subseteq s \implies continuous-on t f*
unfolding *continuous-on-def*
by (*metis subset-eq tendsto-within-subset*)

lemma *continuous-on-compose*[*continuous-intros*]:
continuous-on s f \implies continuous-on (f ' s) g \implies continuous-on s (g \circ f)
unfolding *continuous-on-topological* **by** *simp metis*

lemma *continuous-on-compose2*:

continuous-on t g \implies continuous-on s f \implies f ' s \subseteq t \implies continuous-on s ($\lambda x. g\ (f\ x)$)
using *continuous-on-compose[of s f g] continuous-on-subset* **by** (*force simp add: comp-def*)

lemma *continuous-on-generate-topology*:

assumes *: *open = generate-topology X*
and **: $\bigwedge B. B \in X \implies \exists C. \text{open } C \wedge C \cap A = f\ -' B \cap A$
shows *continuous-on A f*
unfolding *continuous-on-open-invariant*

proof *safe*

fix $B :: 'a\ \text{set}$
assume *open B*
then show $\exists C. \text{open } C \wedge C \cap A = f\ -' B \cap A$
unfolding *
proof *induct*

```

    case (UN K)
  then obtain C where  $\bigwedge k. k \in K \implies \text{open } (C k) \bigwedge k. k \in K \implies C k \cap A =$ 
 $f - ' k \cap A$ 
    by metis
  then show ?case
    by (intro exI[of -  $\bigcup k \in K. C k$ ]) blast
  qed (auto intro: **)
qed

```

lemma *continuous-onI-mono*:

```

  fixes f :: 'a::linorder-topology  $\Rightarrow$  'b::{dense-order,linorder-topology}
  assumes open (f'A)
  and mono:  $\bigwedge x y. x \in A \implies y \in A \implies x \leq y \implies f x \leq f y$ 
  shows continuous-on A f
proof (rule continuous-on-generate-topology[OF open-generated-order], safe)
  have monoD:  $\bigwedge x y. x \in A \implies y \in A \implies f x < f y \implies x < y$ 
    by (auto simp: not-le[symmetric] mono)
  have  $\exists x. x \in A \wedge f x < b \wedge a < x$  if a:  $a \in A$  and fa:  $f a < b$  for a b
  proof -
    obtain y where  $f a < y \{f a .. < y\} \subseteq f'A$ 
      using open-right[OF  $\langle \text{open } (f'A) \rangle$ , of f a b] a fa
      by auto
    obtain z where  $z: f a < z z < \min b y$ 
      using dense[of f a min b y]  $\langle f a < y \rangle \langle f a < b \rangle$  by auto
    then obtain c where  $z = f c c \in A$ 
      using  $\langle \{f a .. < y\} \subseteq f'A \rangle$  [THEN subsetD, of z] by (auto simp: less-imp-le)
    with a z show ?thesis
      by (auto intro!: exI[of - c] simp: monoD)
  qed
  then show  $\exists C. \text{open } C \wedge C \cap A = f - ' \{.. < b\} \cap A$  for b
    by (intro exI[of - ( $\bigcup x \in \{x \in A. f x < b\}. \{.. < x\}$ )])
      (auto intro: le-less-trans[OF mono] less-imp-le)

  have  $\exists x. x \in A \wedge b < f x \wedge x < a$  if a:  $a \in A$  and fa:  $b < f a$  for a b
  proof -
    note a fa
    moreover
    obtain y where  $y < f a \{y <.. f a\} \subseteq f'A$ 
      using open-left[OF  $\langle \text{open } (f'A) \rangle$ , of f a b] a fa
      by auto
    then obtain z where  $z: \max b y < z z < f a$ 
      using dense[of max b y f a]  $\langle y < f a \rangle \langle b < f a \rangle$  by auto
    then obtain c where  $z = f c c \in A$ 
      using  $\langle \{y <.. f a\} \subseteq f'A \rangle$  [THEN subsetD, of z] by (auto simp: less-imp-le)
    with a z show ?thesis
      by (auto intro!: exI[of - c] simp: monoD)
  qed
  then show  $\exists C. \text{open } C \wedge C \cap A = f - ' \{b <.. \} \cap A$  for b
    by (intro exI[of - ( $\bigcup x \in \{x \in A. b < f x\}. \{x <.. \}$ )])

```

(*auto intro: less-le-trans*[*OF - mono*] *less-imp-le*)
qed

lemma *continuous-on-IccI*:

[[*f* \longrightarrow *f a*] (*at-right a*);
f \longrightarrow *f b*] (*at-left b*);
 $(\bigwedge x. a < x \implies x < b \implies f \text{ -}x \rightarrow f x); a < b$] \implies
continuous-on {*a .. b*} *f*

for *a::'a::linorder-topology*

using *at-within-open*[*of - {a<..**b**}*]

by (*auto simp: continuous-on-def at-within-Icc-at-right at-within-Icc-at-left le-less at-within-Icc-at*)

lemma

fixes *a b::'a::linorder-topology*

assumes *continuous-on* {*a .. b*} *f a < b*

shows *continuous-on-Icc-at-rightD*: (*f* \longrightarrow *f a*) (*at-right a*)

and *continuous-on-Icc-at-leftD*: (*f* \longrightarrow *f b*) (*at-left b*)

using *assms*

by (*auto simp: at-within-Icc-at-right at-within-Icc-at-left continuous-on-def dest: bspec[**where** *x=a*] bspec[**where** *x=b*]*)

lemma *continuous-on-discrete* [*simp*]:

continuous-on A (f :: 'a :: discrete-topology \Rightarrow -)

by (*auto simp: continuous-on-def at-discrete*)

lemma *continuous-on-of-nat* [*continuous-intros*]:

assumes *continuous-on A f*

shows *continuous-on A* ($\lambda n. \text{of-nat } (f n)$)

using *continuous-on-compose*[*OF assms continuous-on-discrete*[*of - of-nat*]]

by (*simp add: o-def*)

lemma *continuous-on-of-int* [*continuous-intros*]:

assumes *continuous-on A f*

shows *continuous-on A* ($\lambda n. \text{of-int } (f n)$)

using *continuous-on-compose*[*OF assms continuous-on-discrete*[*of - of-int*]]

by (*simp add: o-def*)

99.10.2 Continuity at a point

definition *continuous* :: '*a::t2-space filter* \Rightarrow (*'a* \Rightarrow '*b::topological-space*) \Rightarrow *bool*

where *continuous F f* \longleftrightarrow (*f* \longrightarrow *f* (*Lim F* ($\lambda x. x$))) *F*

lemma *continuous-bot*[*continuous-intros, simp*]: *continuous bot f*

unfolding *continuous-def* **by** *auto*

lemma *continuous-trivial-limit*: *trivial-limit net* \implies *continuous net f*

by *simp*

lemma *continuous-within*: $\text{continuous (at } x \text{ within } s) f \longleftrightarrow (f \longrightarrow f x) \text{ (at } x \text{ within } s)$

by (*cases trivial-limit (at x within s)*) (*auto simp add: Lim-ident-at continuous-def*)

lemma *continuous-within-topological*:

$\text{continuous (at } x \text{ within } s) f \longleftrightarrow$
 $(\forall B. \text{open } B \longrightarrow f x \in B \longrightarrow (\exists A. \text{open } A \wedge x \in A \wedge (\forall y \in s. y \in A \longrightarrow f y \in B)))$

unfolding *continuous-within tendsto-def eventually-at-topological* **by** *metis*

lemma *continuous-within-compose*[*continuous-intros*]:

$\text{continuous (at } x \text{ within } s) f \Longrightarrow \text{continuous (at } (f x) \text{ within } f ' s) g \Longrightarrow$
 $\text{continuous (at } x \text{ within } s) (g \circ f)$

by (*simp add: continuous-within-topological*) *metis*

lemma *continuous-within-compose2*:

$\text{continuous (at } x \text{ within } s) f \Longrightarrow \text{continuous (at } (f x) \text{ within } f ' s) g \Longrightarrow$
 $\text{continuous (at } x \text{ within } s) (\lambda x. g (f x))$

using *continuous-within-compose*[*of x s f g*] **by** (*simp add: comp-def*)

lemma *continuous-at*: $\text{continuous (at } x) f \longleftrightarrow f -x \rightarrow f x$

using *continuous-within*[*of x UNIV f*] **by** *simp*

lemma *continuous-ident*[*continuous-intros, simp*]: $\text{continuous (at } x \text{ within } S) (\lambda x. x)$

unfolding *continuous-within* **by** (*rule tendsto-ident-at*)

lemma *continuous-id*[*continuous-intros, simp*]: $\text{continuous (at } x \text{ within } S) \text{id}$

by (*simp add: id-def*)

lemma *continuous-const*[*continuous-intros, simp*]: $\text{continuous } F (\lambda x. c)$

unfolding *continuous-def* **by** (*rule tendsto-const*)

lemma *continuous-on-eq-continuous-within*:

$\text{continuous-on } s f \longleftrightarrow (\forall x \in s. \text{continuous (at } x \text{ within } s) f)$

unfolding *continuous-on-def continuous-within ..*

lemma *continuous-discrete* [*simp*]:

$\text{continuous (at } x \text{ within } A) (f :: 'a :: \text{discrete-topology} \Rightarrow -)$

by (*auto simp: continuous-def at-discrete*)

abbreviation *isCont* :: $('a :: t2\text{-space} \Rightarrow 'b :: \text{topological-space}) \Rightarrow 'a \Rightarrow \text{bool}$

where $\text{isCont } f a \equiv \text{continuous (at } a) f$

lemma *isCont-def*: $\text{isCont } f a \longleftrightarrow f -a \rightarrow f a$

by (*rule continuous-at*)

lemma *isContD*: $\text{isCont } f x \Longrightarrow f -x \rightarrow f x$

by (*simp add: isCont-def*)

lemma *isCont-cong*:

assumes *eventually* $(\lambda x. f x = g x)$ (*nhds x*)

shows $isCont f x \longleftrightarrow isCont g x$

proof –

from *assms* **have** [*simp*]: $f x = g x$

by (*rule eventually-nhds-x-imp-x*)

from *assms* **have** *eventually* $(\lambda x. f x = g x)$ (*at x*)

by (*auto simp: eventually-at-filter elim!: eventually-mono*)

with *assms* **have** $isCont f x \longleftrightarrow isCont g x$ **unfolding** *isCont-def*

by (*intro filterlim-cong*) (*auto elim!: eventually-mono*)

with *assms* **show** *?thesis* **by** *simp*

qed

lemma *continuous-at-imp-continuous-at-within*: $isCont f x \implies continuous (at x \text{ within } s) f$

by (*auto intro: tendsto-mono at-le simp: continuous-at continuous-within*)

lemma *continuous-on-eq-continuous-at*: $open s \implies continuous-on s f \longleftrightarrow (\forall x \in s. isCont f x)$

by (*simp add: continuous-on-def continuous-at at-within-open[of - s]*)

lemma *continuous-within-open*: $a \in A \implies open A \implies continuous (at a \text{ within } A) f \longleftrightarrow isCont f a$

by (*simp add: at-within-open-NO-MATCH*)

lemma *continuous-at-imp-continuous-on*: $\forall x \in s. isCont f x \implies continuous-on s f$

by (*auto intro: continuous-at-imp-continuous-at-within simp: continuous-on-eq-continuous-within*)

lemma *isCont-o2*: $isCont f a \implies isCont g (f a) \implies isCont (\lambda x. g (f x)) a$

unfolding *isCont-def* **by** (*rule tendsto-compose*)

lemma *continuous-at-compose[continuous-intros]*: $isCont f a \implies isCont g (f a) \implies isCont (g \circ f) a$

unfolding *o-def* **by** (*rule isCont-o2*)

lemma *isCont-tendsto-compose*: $isCont g l \implies (f \longrightarrow l) F \implies ((\lambda x. g (f x)) \longrightarrow g l) F$

unfolding *isCont-def* **by** (*rule tendsto-compose*)

lemma *continuous-on-tendsto-compose*:

assumes *f-cont*: *continuous-on s f*

and *g*: $(g \longrightarrow l) F$

and *l*: $l \in s$

and *ev*: $\forall_F x \text{ in } F. g x \in s$

shows $((\lambda x. f (g x)) \longrightarrow f l) F$

proof –

from *f-cont l* **have** *f*: $(f \longrightarrow f l)$ (*at l within s*)

by (*simp add: continuous-on-def*)
 have $i: ((\lambda x. \text{if } g \ x = l \ \text{then } f \ l \ \text{else } f \ (g \ x)) \longrightarrow f \ l) \ F$
 by (*rule filterlim-I_f*)
 (auto *intro!*: *filterlim-compose*[*OF f*] *eventually-conj tendsto-mono*[*OF - g*]
simp: filterlim-at eventually-inf-principal eventually-mono[*OF ev*])
 show *?thesis*
 by (*rule filterlim-cong*[*THEN iffD1*[*OF - i*]]) *auto*
 qed

lemma *continuous-within-compose3*:

$\text{isCont } g \ (f \ x) \implies \text{continuous } (\text{at } x \ \text{within } s) \ f \implies \text{continuous } (\text{at } x \ \text{within } s) \ (\lambda x. g \ (f \ x))$
 using *continuous-at-imp-continuous-at-within continuous-within-compose2* by *blast*

lemma *at-within-isCont-imp-nhds*:

fixes $f:: 'a:: \{t2\text{-space}, \text{perfect-space}\} \Rightarrow 'b:: t2\text{-space}$
 assumes $\forall_F w \ \text{in } \text{at } z. f \ w = g \ w \ \text{isCont } f \ z \ \text{isCont } g \ z$
 shows $\forall_F w \ \text{in } \text{nhds } z. f \ w = g \ w$

proof –

have $g \ -z \rightarrow f \ z$
 using *assms isContD tendsto-cong* by *blast*
 moreover have $g \ -z \rightarrow g \ z$ using $\langle \text{isCont } g \ z \rangle$ using *isCont-def* by *blast*
 ultimately have $f \ z = g \ z$ using *LIM-unique* by *auto*
 moreover have $\forall_F x \ \text{in } \text{nhds } z. x \neq z \longrightarrow f \ x = g \ x$
 using *assms unfolding eventually-at-filter* by *auto*
 ultimately show *?thesis*
 by (*auto elim: eventually-mono*)

qed

lemma *filtermap-nhds-open-map'*:

assumes *cont: isCont f a*
 and *open A a ∈ A*
 and *open-map: $\bigwedge S. \text{open } S \implies S \subseteq A \implies \text{open } (f \ ' S)$*
 shows *filtermap f (nhds a) = nhds (f a)*
 unfolding *filter-eq-iff*

proof *safe*

fix P
 assume *eventually P (filtermap f (nhds a))*
 then obtain S where $S: \text{open } S \ a \in S \ \forall x \in S. P \ (f \ x)$
 by (*auto simp: eventually-filtermap eventually-nhds*)
 show *eventually P (nhds (f a))*
 unfolding *eventually-nhds*
proof (*rule exI* [*of - f ' (A ∩ S)*], *safe*)
 show *open (f ' (A ∩ S))*
 using S by (*intro open-Int assms*) *auto*
 show $f \ a \in f \ ' (A \cap S)$
 using *assms S* by *auto*
 show $P \ (f \ x)$ if $x \in A \ x \in S$ for x

using S that by auto
 qed
 qed (metis filterlim-iff tendsto-at-iff-tendsto-nhds isCont-def eventually-filtermap cont)

lemma filtermap-nhds-open-map:
 assumes cont: isCont f a
 and open-map: $\bigwedge S. \text{open } S \implies \text{open } (f'S)$
 shows filtermap f (nhds a) = nhds ($f a$)
 using cont filtermap-nhds-open-map' open-map by blast

lemma continuous-at-split:
 continuous (at x) $f \iff \text{continuous } (\text{at-left } x) f \wedge \text{continuous } (\text{at-right } x) f$
 for $x :: 'a::\text{linorder-topology}$
 by (simp add: continuous-within filterlim-at-split)

lemma continuous-on-max [continuous-intros]:
 fixes $f g :: 'a::\text{topological-space} \Rightarrow 'b::\text{linorder-topology}$
 shows continuous-on A $f \implies \text{continuous-on } A$ $g \implies \text{continuous-on } A$ ($\lambda x. \max (f x) (g x)$)
 by (auto simp: continuous-on-def intro!: tendsto-max)

lemma continuous-on-min [continuous-intros]:
 fixes $f g :: 'a::\text{topological-space} \Rightarrow 'b::\text{linorder-topology}$
 shows continuous-on A $f \implies \text{continuous-on } A$ $g \implies \text{continuous-on } A$ ($\lambda x. \min (f x) (g x)$)
 by (auto simp: continuous-on-def intro!: tendsto-min)

lemma continuous-max [continuous-intros]:
 fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{linorder-topology}$
 shows $\llbracket \text{continuous } F f; \text{continuous } F g \rrbracket \implies \text{continuous } F$ ($\lambda x. (\max (f x) (g x))$)
 by (simp add: tendsto-max continuous-def)

lemma continuous-min [continuous-intros]:
 fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{linorder-topology}$
 shows $\llbracket \text{continuous } F f; \text{continuous } F g \rrbracket \implies \text{continuous } F$ ($\lambda x. (\min (f x) (g x))$)
 by (simp add: tendsto-min continuous-def)

The following open/closed Collect lemmas are ported from Sébastien Gouëzel’s *Ergodic-Theory*.

lemma open-Collect-neq:
 fixes $f g :: 'a::\text{topological-space} \Rightarrow 'b::t2\text{-space}$
 assumes f : continuous-on UNIV f and g : continuous-on UNIV g
 shows open $\{x. f x \neq g x\}$
 proof (rule openI)
 fix t
 assume $t \in \{x. f x \neq g x\}$
 then obtain $U V$ where *: open U open V $f t \in U$ $g t \in V$ $U \cap V = \{\}$

```

  by (auto simp add: separation-t2)
  with open-vimage[OF ‹open U› f] open-vimage[OF ‹open V› g]
  show  $\exists T. \text{open } T \wedge t \in T \wedge T \subseteq \{x. f x \neq g x\}$ 
    by (intro exI[of - f - ‹U ∩ g - ‹V]) auto
qed

```

lemma *closed-Collect-eq*:

```

  fixes f g :: 'a::topological-space  $\Rightarrow$  'b::t2-space
  assumes f: continuous-on UNIV f and g: continuous-on UNIV g
  shows closed  $\{x. f x = g x\}$ 
  using open-Collect-neq[OF f g] by (simp add: closed-def Collect-neg-eq)

```

lemma *open-Collect-less*:

```

  fixes f g :: 'a::topological-space  $\Rightarrow$  'b::linorder-topology
  assumes f: continuous-on UNIV f and g: continuous-on UNIV g
  shows open  $\{x. f x < g x\}$ 
proof (rule openI)
  fix t
  assume t:  $t \in \{x. f x < g x\}$ 
  show  $\exists T. \text{open } T \wedge t \in T \wedge T \subseteq \{x. f x < g x\}$ 
  proof (cases  $\exists z. f t < z \wedge z < g t$ )
    case True
    then obtain z where  $f t < z \wedge z < g t$  by blast
    then show ?thesis
      using open-vimage[OF - f, of  $\{.. < z\}$ ] open-vimage[OF - g, of  $\{z < ..\}$ ]
      by (intro exI[of - f - ‹ $\{.. < z\} \cap g - ‹\{z < ..\}$ ]) auto
  next
    case False
    then have *:  $\{g t ..\} = \{f t < ..\} \{.. < g t\} = \{.. f t\}$ 
      using t by (auto intro: leI)
    show ?thesis
      using open-vimage[OF - f, of  $\{.. < g t\}$ ] open-vimage[OF - g, of  $\{f t < ..\}$ ] t
      apply (intro exI[of - f - ‹ $\{.. < g t\} \cap g - ‹\{f t < ..\}$ ])
      apply (simp add: open-Int)
      apply (auto simp add: *)
      done
  qed
qed

```

lemma *closed-Collect-le*:

```

  fixes f g :: 'a :: topological-space  $\Rightarrow$  'b::linorder-topology
  assumes f: continuous-on UNIV f
  and g: continuous-on UNIV g
  shows closed  $\{x. f x \leq g x\}$ 
  using open-Collect-less [OF g f]
  by (simp add: closed-def Collect-neg-eq[symmetric] not-le)

```

99.10.3 Open-cover compactness**context** *topological-space***begin****definition** *compact* :: 'a set \Rightarrow bool **where***compact-eq-Heine-Borel*:
$$\text{compact } S \iff (\forall C. (\forall c \in C. \text{open } c) \wedge S \subseteq \bigcup C \longrightarrow (\exists D \subseteq C. \text{finite } D \wedge S \subseteq \bigcup D))$$
lemma *compactI*:**assumes** $\bigwedge C. \forall t \in C. \text{open } t \implies s \subseteq \bigcup C \implies \exists C'. C' \subseteq C \wedge \text{finite } C' \wedge s \subseteq \bigcup C'$ **shows** *compact s***unfolding** *compact-eq-Heine-Borel* **using** *assms* **by** *metis***lemma** *compact-empty[simp]*: *compact {}***by** (*auto intro!*: *compactI*)**lemma** *compactE*:**assumes** *compact S* $S \subseteq \bigcup \mathcal{T} \wedge B. B \in \mathcal{T} \implies \text{open } B$ **obtains** \mathcal{T}' **where** $\mathcal{T}' \subseteq \mathcal{T}$ *finite* \mathcal{T}' $S \subseteq \bigcup \mathcal{T}'$ **by** (*meson* *assms compact-eq-Heine-Borel*)**lemma** *compactE-image*:**assumes** *compact S***and** *opn*: $\bigwedge T. T \in C \implies \text{open } (f T)$ **and** *S*: $S \subseteq (\bigcup c \in C. f c)$ **obtains** C' **where** $C' \subseteq C$ **and** *finite* C' **and** $S \subseteq (\bigcup c \in C'. f c)$ **apply** (*rule compactE[OF <compact S> S]*)**using** *opn* **apply** *force***by** (*metis finite-subset-image*)**lemma** *compact-Int-closed [intro]*:**assumes** *compact S***and** *closed T***shows** *compact (S \cap T)***proof** (*rule compactI*)**fix** C **assume** $C: \forall c \in C. \text{open } c$ **assume** *cover*: $S \cap T \subseteq \bigcup C$ **from** C *<closed T>* **have** $\forall c \in C \cup \{- T\}. \text{open } c$ **by** *auto***moreover from** *cover* **have** $S \subseteq \bigcup (C \cup \{- T\})$ **by** *auto***ultimately have** $\exists D \subseteq C \cup \{- T\}. \text{finite } D \wedge S \subseteq \bigcup D$ **using** *<compact S>* **unfolding** *compact-eq-Heine-Borel* **by** *auto***then obtain** D **where** $D \subseteq C \cup \{- T\} \wedge \text{finite } D \wedge S \subseteq \bigcup D$..**then show** $\exists D \subseteq C. \text{finite } D \wedge S \cap T \subseteq \bigcup D$ **by** (*intro exI[of - D - \{-T\}]*) *auto*

qed

lemma *compact-diff*: $\llbracket \text{compact } S; \text{ open } T \rrbracket \implies \text{compact}(S - T)$
 by (*simp add: Diff-eq compact-Int-closed open-closed*)

lemma *inj-setminus*: *inj-on uminus* ($A::'a \text{ set set}$)
 by (*auto simp: inj-on-def*)

99.11 Finite intersection property

lemma *compact-fip*:

compact $U \iff$
 $(\forall A. (\forall a \in A. \text{closed } a) \longrightarrow (\forall B \subseteq A. \text{finite } B \longrightarrow U \cap \bigcap B \neq \{\})) \longrightarrow U \cap \bigcap A \neq \{\}$
 (is - \iff ?R)

proof (*safe intro!: compact-eq-Heine-Borel[THEN iffD2]*)

fix A

assume *compact* U

assume $A: \forall a \in A. \text{closed } a \implies U \cap \bigcap A = \{\}$

assume *fin*: $\forall B \subseteq A. \text{finite } B \longrightarrow U \cap \bigcap B \neq \{\}$

from A have $(\forall a \in \text{uminus } A. \text{open } a) \wedge U \subseteq \bigcup (\text{uminus } A)$

by *auto*

with $\langle \text{compact } U \rangle$ obtain B where $B \subseteq A$ *finite* $(\text{uminus } B) \implies U \subseteq \bigcup (\text{uminus } B)$

unfolding *compact-eq-Heine-Borel* by (*metis subset-image-iff*)

with *fin*[*THEN spec, of B*] show *False*

by (*auto dest: finite-imageD intro: inj-setminus*)

next

fix A

assume ?R

assume $\forall a \in A. \text{open } a \implies U \subseteq \bigcup A$

then have $U \cap \bigcap (\text{uminus } A) = \{\} \implies \forall a \in \text{uminus } A. \text{closed } a$

by *auto*

with $\langle ?R \rangle$ obtain B where $B \subseteq A$ *finite* $(\text{uminus } B) \implies U \cap \bigcap (\text{uminus } B) = \{\}$

by (*metis subset-image-iff*)

then show $\exists T \subseteq A. \text{finite } T \wedge U \subseteq \bigcup T$

by (*auto intro!: exI[of - B] inj-setminus dest: finite-imageD*)

qed

lemma *compact-imp-fip*:

assumes *compact* S

and $\bigwedge T. T \in F \implies \text{closed } T$

and $\bigwedge F'. \text{finite } F' \implies F' \subseteq F \implies S \cap (\bigcap F') \neq \{\}$

shows $S \cap (\bigcap F) \neq \{\}$

using *assms* unfolding *compact-fip* by *auto*

lemma *compact-imp-fip-image*:

assumes *compact* s

and $P: \bigwedge i. i \in I \implies \text{closed } (f i)$

and $Q: \bigwedge I'. \text{finite } I' \implies I' \subseteq I \implies (s \cap (\bigcap i \in I'. f i)) \neq \{\}$

shows $s \cap (\bigcap_{i \in I} f i) \neq \{\}$
proof –
from P **have** $\forall i \in f ' I. \text{closed } i$
by *blast*
moreover have $\forall A. \text{finite } A \wedge A \subseteq f ' I \longrightarrow (s \cap (\bigcap A) \neq \{\})$
by (*metis Q finite-subset-image*)
ultimately show $s \cap (\bigcap (f ' I)) \neq \{\}$
by (*metis <compact s> compact-imp-fip*)
qed

end

lemma (*in t2-space*) *compact-imp-closed*:
assumes *compact s*
shows *closed s*
unfolding *closed-def*
proof (*rule openI*)
fix y
assume $y \in - s$
let $?C = \bigcup_{x \in s} \{u. \text{open } u \wedge x \in u \wedge \text{eventually } (\lambda y. y \notin u) (\text{nhds } y)\}$
have $s \subseteq \bigcup ?C$
proof
fix x
assume $x \in s$
with $\langle y \in - s \rangle$ **have** $x \neq y$ **by** *clarsimp*
then have $\exists u v. \text{open } u \wedge \text{open } v \wedge x \in u \wedge y \in v \wedge u \cap v = \{\}$
by (*rule hausdorff*)
with $\langle x \in s \rangle$ **show** $x \in \bigcup ?C$
unfolding *eventually-nhds by auto*
qed
then obtain D **where** $D \subseteq ?C$ **and** *finite D* **and** $s \subseteq \bigcup D$
by (*rule compactE [OF <compact s>]*) *auto*
from $\langle D \subseteq ?C \rangle$ **have** $\forall x \in D. \text{eventually } (\lambda y. y \notin x) (\text{nhds } y)$
by *auto*
with $\langle \text{finite } D \rangle$ **have** $\text{eventually } (\lambda y. y \notin \bigcup D) (\text{nhds } y)$
by (*simp add: eventually-ball-finite*)
with $\langle s \subseteq \bigcup D \rangle$ **have** $\text{eventually } (\lambda y. y \notin s) (\text{nhds } y)$
by (*auto elim!: eventually-mono*)
then show $\exists t. \text{open } t \wedge y \in t \wedge t \subseteq - s$
by (*simp add: eventually-nhds subset-eq*)
qed

lemma *compact-continuous-image*:
assumes $f: \text{continuous-on } s$
and $s: \text{compact } s$
shows $\text{compact } (f ' s)$
proof (*rule compactI*)
fix C
assume $\forall c \in C. \text{open } c$ **and** *cover: f's $\subseteq \bigcup C$*

with f have $\forall c \in C. \exists A. \text{open } A \wedge A \cap s = f^{-1} c \cap s$
unfolding *continuous-on-open-invariant* **by** *blast*
then obtain A where $A: \forall c \in C. \text{open } (A \ c) \wedge A \ c \cap s = f^{-1} c \cap s$
unfolding *bchoice-iff* ..
with cover have $\bigwedge c. c \in C \implies \text{open } (A \ c) \ s \subseteq (\bigcup c \in C. A \ c)$
by (*fastforce simp add: subset-eq set-eq-iff*)+
from compactE-image[OF s this] obtain D where $D \subseteq C$ *finite* $D \ s \subseteq (\bigcup c \in D. A \ c)$.
with A show $\exists D \subseteq C. \text{finite } D \wedge f's \subseteq \bigcup D$
by (*intro exI[of - D] (fastforce simp add: subset-eq set-eq-iff)*)+
qed

lemma *continuous-on-inv*:

fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::t2\text{-space}$
assumes *continuous-on s f*
and *compact s*
and $\forall x \in s. g \ (f \ x) = x$
shows *continuous-on $(f^{-1} \ s) \ g$*
unfolding *continuous-on-topological*
proof (*clarsimp simp add: assms(3)*)
fix $x :: 'a$ **and** $B :: 'a \text{ set}$
assume $x \in s$ **and** *open B* **and** $x \in B$
have $1: \forall x \in s. f \ x \in f^{-1} (s - B) \longleftrightarrow x \in s - B$
using *assms(3)* **by** (*auto, metis*)
have *continuous-on $(s - B) \ f$*
using $\langle \text{continuous-on } s \ f \rangle$ *Diff-subset*
by (*rule continuous-on-subset*)
moreover have *compact $(s - B)$*
using $\langle \text{open } B \rangle$ **and** $\langle \text{compact } s \rangle$
unfolding *Diff-eq* **by** (*intro compact-Int-closed closed-Compl*)
ultimately have *compact $(f^{-1} (s - B))$*
by (*rule compact-continuous-image*)
then have *closed $(f^{-1} (s - B))$*
by (*rule compact-imp-closed*)
then have *open $(- f^{-1} (s - B))$*
by (*rule open-Compl*)
moreover have $f \ x \in - f^{-1} (s - B)$
using $\langle x \in s \rangle$ **and** $\langle x \in B \rangle$ **by** (*simp add: 1*)
moreover have $\forall y \in s. f \ y \in - f^{-1} (s - B) \longrightarrow y \in B$
by (*simp add: 1*)
ultimately show $\exists A. \text{open } A \wedge f \ x \in A \wedge (\forall y \in s. f \ y \in A \longrightarrow y \in B)$
by *fast*
qed

lemma *continuous-on-inv-into*:

fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::t2\text{-space}$
assumes s : *continuous-on s f compact s*
and f : *inj-on $f \ s$*
shows *continuous-on $(f^{-1} \ s)$ (the-inv-into $s \ f)$*

by (rule continuous-on-inv[OF s]) (auto simp: the-inv-into-f-f[OF f])

lemma (in linorder-topology) compact-attains-sup:

assumes compact S S ≠ {}

shows $\exists s \in S. \forall t \in S. t \leq s$

proof (rule classical)

assume $\neg (\exists s \in S. \forall t \in S. t \leq s)$

then obtain t where t: $\forall s \in S. t s \in S$ and $\forall s \in S. s < t s$

by (metis not-le)

then have $\bigwedge s. s \in S \implies \text{open } \{.. < t s\} S \subseteq (\bigcup s \in S. \{.. < t s\})$

by auto

with $\langle \text{compact } S \rangle$ obtain C where $C \subseteq S$ finite C and $C: S \subseteq (\bigcup s \in C. \{.. < t s\})$

by (metis compactE-image)

with $\langle S \neq \{\} \rangle$ have Max: $\text{Max } (t'C) \in t'C$ and $\forall s \in t'C. s \leq \text{Max } (t'C)$

by (auto intro!: Max-in)

with C have $S \subseteq \{.. < \text{Max } (t'C)\}$

by (auto intro: less-le-trans simp: subset-eq)

with t Max $\langle C \subseteq S \rangle$ show ?thesis

by fastforce

qed

lemma (in linorder-topology) compact-attains-inf:

assumes compact S S ≠ {}

shows $\exists s \in S. \forall t \in S. s \leq t$

proof (rule classical)

assume $\neg (\exists s \in S. \forall t \in S. s \leq t)$

then obtain t where t: $\forall s \in S. t s \in S$ and $\forall s \in S. t s < s$

by (metis not-le)

then have $\bigwedge s. s \in S \implies \text{open } \{t s <..\} S \subseteq (\bigcup s \in S. \{t s <..\})$

by auto

with $\langle \text{compact } S \rangle$ obtain C where $C \subseteq S$ finite C and $C: S \subseteq (\bigcup s \in C. \{t s <..\})$

by (metis compactE-image)

with $\langle S \neq \{\} \rangle$ have Min: $\text{Min } (t'C) \in t'C$ and $\forall s \in t'C. \text{Min } (t'C) \leq s$

by (auto intro!: Min-in)

with C have $S \subseteq \{\text{Min } (t'C) <..\}$

by (auto intro: le-less-trans simp: subset-eq)

with t Min $\langle C \subseteq S \rangle$ show ?thesis

by fastforce

qed

lemma continuous-attains-sup:

fixes f :: 'a::topological-space \Rightarrow 'b::linorder-topology

shows compact s $\implies s \neq \{\} \implies \text{continuous-on } s f \implies (\exists x \in s. \forall y \in s. f y \leq f x)$

using compact-attains-sup[of f 's] compact-continuous-image[of s f] by auto

lemma continuous-attains-inf:

fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{linorder-topology}$
shows $\text{compact } s \Longrightarrow s \neq \{\} \Longrightarrow \text{continuous-on } s f \Longrightarrow (\exists x \in s. \forall y \in s. f x \leq f y)$
using $\text{compact-attains-inf}[of f ' s]$ $\text{compact-continuous-image}[of s f]$ **by** *auto*

99.12 Connectedness

context *topological-space*

begin

definition $\text{connected } S \longleftrightarrow$

$\neg (\exists A B. \text{open } A \wedge \text{open } B \wedge S \subseteq A \cup B \wedge A \cap B \cap S = \{\} \wedge A \cap S \neq \{\} \wedge B \cap S \neq \{\})$

lemma *connectedI*:

$(\bigwedge A B. \text{open } A \Longrightarrow \text{open } B \Longrightarrow A \cap U \neq \{\} \Longrightarrow B \cap U \neq \{\} \Longrightarrow A \cap B \cap U = \{\} \Longrightarrow U \subseteq A \cup B \Longrightarrow \text{False})$
 $\Longrightarrow \text{connected } U$

by (*auto simp: connected-def*)

lemma *connected-empty [simp]: connected {}*

by (*auto intro!: connectedI*)

lemma *connected-sing [simp]: connected {x}*

by (*auto intro!: connectedI*)

lemma *connectedD*:

$\text{connected } A \Longrightarrow \text{open } U \Longrightarrow \text{open } V \Longrightarrow U \cap V \cap A = \{\} \Longrightarrow A \subseteq U \cup V$
 $\Longrightarrow U \cap A = \{\} \vee V \cap A = \{\}$

by (*auto simp: connected-def*)

end

lemma *connected-closed*:

$\text{connected } s \longleftrightarrow$

$\neg (\exists A B. \text{closed } A \wedge \text{closed } B \wedge s \subseteq A \cup B \wedge A \cap B \cap s = \{\} \wedge A \cap s \neq \{\} \wedge B \cap s \neq \{\})$

apply (*simp add: connected-def del: ex-simps, safe*)

apply (*drule-tac x=-A in spec*)

apply (*drule-tac x=-B in spec*)

apply (*fastforce simp add: closed-def [symmetric]*)

apply (*drule-tac x=-A in spec*)

apply (*drule-tac x=-B in spec*)

apply (*fastforce simp add: open-closed [symmetric]*)

done

lemma *connected-closedD*:

$\llbracket \text{connected } s; A \cap B \cap s = \{\}; s \subseteq A \cup B; \text{closed } A; \text{closed } B \rrbracket \Longrightarrow A \cap s = \{\} \vee B \cap s = \{\}$

by (*simp add: connected-closed*)

lemma *connected-Union*:
assumes $cs: \bigwedge s. s \in S \implies \text{connected } s$
and $ne: \bigcap S \neq \{\}$
shows $\text{connected}(\bigcup S)$
proof (*rule connectedI*)
fix $A B$
assume $A: \text{open } A$ **and** $B: \text{open } B$ **and** $Alap: A \cap \bigcup S \neq \{\}$ **and** $Blap: B \cap \bigcup S \neq \{\}$
and $disj: A \cap B \cap \bigcup S = \{\}$ **and** $cover: \bigcup S \subseteq A \cup B$
have $disjs: \bigwedge s. s \in S \implies A \cap B \cap s = \{\}$
using *disj* **by** *auto*
obtain sa **where** $sa: sa \in S \ A \cap sa \neq \{\}$
using *Alap* **by** *auto*
obtain sb **where** $sb: sb \in S \ B \cap sb \neq \{\}$
using *Blap* **by** *auto*
obtain x **where** $x: \bigwedge s. s \in S \implies x \in s$
using *ne* **by** *auto*
then **have** $x \in \bigcup S$
using $\langle sa \in S \rangle$ **by** *blast*
then **have** $x \in A \vee x \in B$
using *cover* **by** *auto*
then **show** *False*
using *cs* [*unfolded connected-def*]
by (*metis A B IntI Sup-upper sa sb disjs x cover empty-iff subset-trans*)
qed

lemma *connected-Un*: $\text{connected } s \implies \text{connected } t \implies s \cap t \neq \{\} \implies \text{connected } (s \cup t)$
using *connected-Union* [*of {s,t}*] **by** *auto*

lemma *connected-diff-open-from-closed*:
assumes $st: s \subseteq t$
and $tu: t \subseteq u$
and $s: \text{open } s$
and $t: \text{closed } t$
and $u: \text{connected } u$
and $ts: \text{connected } (t - s)$
shows $\text{connected}(u - s)$
proof (*rule connectedI*)
fix $A B$
assume $AB: \text{open } A \ \text{open } B \ A \cap (u - s) \neq \{\} \ B \cap (u - s) \neq \{\}$
and $disj: A \cap B \cap (u - s) = \{\}$
and $cover: u - s \subseteq A \cup B$
then **consider** $A \cap (t - s) = \{\} \mid B \cap (t - s) = \{\}$
using $st \ ts \ tu$ *connectedD* [*of t-s A B*] **by** *auto*
then **show** *False*
proof *cases*
case 1

```

then have  $(A - t) \cap (B \cup s) \cap u = \{\}$ 
  using disj st by auto
moreover have  $u \subseteq (A - t) \cup (B \cup s)$ 
  using 1 cover by auto
ultimately show False
  using connectedD [of u A - t B \cup s] AB s t 1 u by auto
next
case 2
then have  $(A \cup s) \cap (B - t) \cap u = \{\}$ 
  using disj st by auto
moreover have  $u \subseteq (A \cup s) \cup (B - t)$ 
  using 2 cover by auto
ultimately show False
  using connectedD [of u A \cup s B - t] AB s t 2 u by auto
qed
qed

lemma connected-iff-const:
  fixes  $S :: 'a::\text{topological-space set}$ 
  shows  $\text{connected } S \longleftrightarrow (\forall P::'a \Rightarrow \text{bool. continuous-on } S P \longrightarrow (\exists c. \forall s \in S. P s = c))$ 
proof safe
  fix  $P :: 'a \Rightarrow \text{bool}$ 
  assume connected S continuous-on S P
  then have  $\bigwedge b. \exists A. \text{open } A \wedge A \cap S = P - \{b\} \cap S$ 
    unfolding continuous-on-open-invariant by (simp add: open-discrete)
  from this[of True] this[of False]
  obtain  $t f$  where open t open f and  $*: f \cap S = P - \{False\} \cap S \wedge t \cap S = P - \{True\} \cap S$ 
    by meson
  then have  $t \cap S = \{\} \vee f \cap S = \{\}$ 
    by (intro connectedD[OF <connected S>] auto)
  then show  $\exists c. \forall s \in S. P s = c$ 
  proof (rule disjE)
    assume  $t \cap S = \{\}$ 
    then show ?thesis
      unfolding  $*$  by (intro exI[of - False] auto)
  next
    assume  $f \cap S = \{\}$ 
    then show ?thesis
      unfolding  $*$  by (intro exI[of - True] auto)
  qed
next
assume  $P: \forall P::'a \Rightarrow \text{bool. continuous-on } S P \longrightarrow (\exists c. \forall s \in S. P s = c)$ 
show connected S
proof (rule connectedI)
  fix  $A B$ 
  assume  $*: \text{open } A \text{ open } B \wedge A \cap S \neq \{\} \wedge B \cap S \neq \{\} \wedge A \cap B \cap S = \{\} \wedge S \subseteq A \cup B$ 

```

```

have continuous-on S ( $\lambda x. x \in A$ )
  unfolding continuous-on-open-invariant
proof safe
  fix C :: bool set
  have  $C = UNIV \vee C = \{True\} \vee C = \{False\} \vee C = \{\}$ 
    using subset-UNIV[of C] unfolding UNIV-bool by auto
  with * show  $\exists T. \text{open } T \wedge T \cap S = (\lambda x. x \in A) - \langle C \cap S$ 
    by (intro exI[of - (if True  $\in C$  then A else  $\{\}$ )  $\cup$  (if False  $\in C$  then B else
   $\{\}$ )] auto
  qed
from  $P[\text{rule-format, OF this}]$  obtain c where  $\bigwedge s. s \in S \implies (s \in A) = c$ 
  by blast
with * show False
  by (cases c) auto
qed
qed

```

```

lemma connectedD-const:  $\text{connected } S \implies \text{continuous-on } S P \implies \exists c. \forall s \in S. P s$ 
 $= c$ 
for  $P :: 'a::\text{topological-space} \Rightarrow \text{bool}$ 
by (auto simp: connected-iff-const)

```

```

lemma connectedI-const:
  ( $\bigwedge P::'a::\text{topological-space} \Rightarrow \text{bool}. \text{continuous-on } S P \implies \exists c. \forall s \in S. P s = c$ )
 $\implies \text{connected } S$ 
by (auto simp: connected-iff-const)

```

```

lemma connected-local-const:
  assumes connected A a  $a \in A$   $b \in A$ 
  and *:  $\forall a \in A. \text{eventually } (\lambda b. f a = f b)$  (at a within A)
  shows  $f a = f b$ 
proof -
  obtain S where  $S: \bigwedge a. a \in A \implies a \in S$   $a \in A \implies \text{open } (S a)$ 
     $\bigwedge a x. a \in A \implies x \in S a \implies x \in A \implies f a = f x$ 
    using * unfolding eventually-at-topological by metis
  let  $?P = \bigcup b \in \{b \in A. f a = f b\}. S b$  and  $?N = \bigcup b \in \{b \in A. f a \neq f b\}. S b$ 
  have  $?P \cap A = \{\} \vee ?N \cap A = \{\}$ 
    using  $\langle \text{connected } A \rangle S \langle a \in A \rangle$ 
    by (intro connectedD) (auto, metis)
  then show  $f a = f b$ 
proof
  assume  $?N \cap A = \{\}$ 
  then have  $\forall x \in A. f a = f x$ 
    using  $S(1)$  by auto
  with  $\langle b \in A \rangle$  show ?thesis by auto
next
  assume  $?P \cap A = \{\}$  then show ?thesis
    using  $\langle a \in A \rangle S(1)[\text{of } a]$  by auto
qed

```

qed

lemma (in *linorder-topology*) *connectedD-interval*:

assumes *connected U*

and *xy: x ∈ U y ∈ U*

and $x \leq z \leq y$

shows $z \in U$

proof –

have *eq: $\{..<z\} \cup \{z<..\} = - \{z\}$*

by *auto*

have $\neg \text{connected } U$ **if** $z \notin U$ $x < z < y$

using *xy that*

apply (*simp only: connected-def simp-thms*)

apply (*rule-tac exI[of - $\{..<z\}$]*)

apply (*rule-tac exI[of - $\{z<..\}$]*)

apply (*auto simp add: eq*)

done

with *assms* **show** $z \in U$

by (*metis less-le*)

qed

lemma (in *linorder-topology*) *not-in-connected-cases*:

assumes *conn: connected S*

assumes *nbdd: $x \notin S$*

assumes *ne: $S \neq \{\}$*

obtains *bdd-above S $\wedge y. y \in S \implies x \geq y$ | bdd-below S $\wedge y. y \in S \implies x \leq y$*

proof –

obtain *s* **where** $s \in S$ **using** *ne* **by** *blast*

{

assume $s \leq x$

have *False* **if** $x \leq y$ $y \in S$ **for** *y*

using *connectedD-interval[OF conn $\langle s \in S \rangle \langle y \in S \rangle \langle s \leq x \rangle \langle x \leq y \rangle \langle x \notin S \rangle$*

by *simp*

then have *wit: $y \in S \implies x \geq y$* **for** *y*

using *le-cases* **by** *blast*

then have *bdd-above S*

by (*rule local.bdd-aboveI*)

note *this wit*

} **moreover** {

assume $x \leq s$

have *False* **if** $x \geq y$ $y \in S$ **for** *y*

using *connectedD-interval[OF conn $\langle y \in S \rangle \langle s \in S \rangle \langle x \geq y \rangle \langle s \geq x \rangle$*

S

by *simp*

then have *wit: $y \in S \implies x \leq y$* **for** *y*

using *le-cases* **by** *blast*

then have *bdd-below S*

by (*rule bdd-belowI*)

note *this wit*

```

} ultimately show ?thesis
  by (meson le-cases that)
qed

```

lemma *connected-continuous-image:*

```

assumes *: continuous-on s f
  and connected s
shows connected (f ` s)
proof (rule connectedI-const)
  fix P :: 'b  $\Rightarrow$  bool
  assume continuous-on (f ` s) P
  then have continuous-on s (P  $\circ$  f)
    by (rule continuous-on-compose[OF *])
  from connectedD-const[OF `connected s` this] show  $\exists c. \forall s \in f ` s. P s = c$ 
    by auto
qed

```

lemma *connected-Un-UN:*

```

assumes connected A  $\wedge$  X. X  $\in$  B  $\implies$  connected X  $\wedge$  X. X  $\in$  B  $\implies$  A  $\cap$  X  $\neq$ 
{}
shows connected (A  $\cup$   $\bigcup$  B)
proof (rule connectedI-const)
  fix f :: 'a  $\Rightarrow$  bool
  assume f: continuous-on (A  $\cup$   $\bigcup$  B) f
  have connected A continuous-on A f
    by (auto intro: assms continuous-on-subset[OF f(1)])
  from connectedD-const[OF this] obtain c where c:  $\bigwedge x. x \in A \implies f x = c$ 
    by metis
  have f x = c if x  $\in$  X X  $\in$  B for x X
  proof -
    have connected X continuous-on X f
      using that by (auto intro: assms continuous-on-subset[OF f])
    from connectedD-const[OF this] obtain c' where c':  $\bigwedge x. x \in X \implies f x = c'$ 
      by metis
    from assms(3) and that obtain y where y  $\in$  A  $\cap$  X
      by auto
    with c[of y] c'[of y] c'[of x] that show ?thesis
      by auto
  qed
  with c show  $\exists c. \forall x \in A \cup \bigcup B. f x = c$ 
    by (intro exI[of - c]) auto
qed

```

100 Linear Continuum Topologies

```

class linear-continuum-topology = linorder-topology + linear-continuum
begin

```

lemma *Inf-notin-open:*

```

assumes  $A$ : open  $A$ 
  and  $bnd$ :  $\forall a \in A. x < a$ 
shows  $Inf\ A \notin A$ 
proof
  assume  $Inf\ A \in A$ 
  then obtain  $b$  where  $b < Inf\ A$   $\{b <.. Inf\ A\} \subseteq A$ 
    using open-left[of  $A$   $Inf\ A$   $x$ ] assms by auto
  with dense[of  $b$   $Inf\ A$ ] obtain  $c$  where  $c < Inf\ A$   $c \in A$ 
    by (auto simp: subset-eq)
  then show False
    using cInf-lower[OF  $\langle c \in A \rangle$ ]  $bnd$ 
    by (metis not-le less-imp-le bdd-belowI)
qed

```

```

lemma Sup-notin-open:
  assumes  $A$ : open  $A$ 
    and  $bnd$ :  $\forall a \in A. a < x$ 
  shows  $Sup\ A \notin A$ 
proof
  assume  $Sup\ A \in A$ 
  with assms obtain  $b$  where  $Sup\ A < b$   $\{Sup\ A ..< b\} \subseteq A$ 
    using open-right[of  $A$   $Sup\ A$   $x$ ] by auto
  with dense[of  $Sup\ A$   $b$ ] obtain  $c$  where  $Sup\ A < c$   $c \in A$ 
    by (auto simp: subset-eq)
  then show False
    using cSup-upper[OF  $\langle c \in A \rangle$ ]  $bnd$ 
    by (metis less-imp-le not-le bdd-aboveI)
qed

```

end

```

instance linear-continuum-topology  $\subseteq$  perfect-space
proof
  fix  $x :: 'a$ 
  obtain  $y$  where  $x < y \vee y < x$ 
    using ex-gt-or-lt [of  $x$ ] ..
  with Inf-notin-open[of  $\{x\}$   $y$ ] Sup-notin-open[of  $\{x\}$   $y$ ] show  $\neg$  open  $\{x\}$ 
    by auto
qed

```

```

lemma connectedI-interval:
  fixes  $U :: 'a ::$  linear-continuum-topology set
  assumes  $*$ :  $\bigwedge x\ y\ z. x \in U \implies y \in U \implies x \leq z \implies z \leq y \implies z \in U$ 
  shows connected  $U$ 
proof (rule connectedI)
  {
    fix  $A\ B$ 
    assume open  $A$  open  $B$   $A \cap B \cap U = \{\}$   $U \subseteq A \cup B$ 
    fix  $x\ y$ 

```

```

assume  $x < y$   $x \in A$   $y \in B$   $x \in U$   $y \in U$ 

let  $?z = \text{Inf} (B \cap \{x <..\})$ 

have  $x \leq ?z$   $?z \leq y$ 
  using  $\langle y \in B \rangle \langle x < y \rangle$  by (auto intro: cInf-lower cInf-greatest)
with  $\langle x \in U \rangle \langle y \in U \rangle$  have  $?z \in U$ 
  by (rule *)
moreover have  $?z \notin B \cap \{x <..\}$ 
  using  $\langle \text{open } B \rangle$  by (intro Inf-notin-open) auto
ultimately have  $?z \in A$ 
  using  $\langle x \leq ?z \rangle \langle A \cap B \cap U = \{\} \rangle \langle x \in A \rangle \langle U \subseteq A \cup B \rangle$  by auto
have  $\exists b \in B. b \in A \wedge b \in U$  if  $?z < y$ 
proof -
  obtain  $a$  where  $?z < a$   $\{?z ..< a\} \subseteq A$ 
    using open-right[OF  $\langle \text{open } A \rangle \langle ?z \in A \rangle \langle ?z < y \rangle$ ] by auto
  moreover obtain  $b$  where  $b \in B$   $x < b$   $b < \text{min } a$   $y$ 
    using cInf-less-iff[of  $B \cap \{x <..\}$  min  $a$   $y$ ]  $\langle ?z < a \rangle \langle ?z < y \rangle \langle x < y \rangle \langle y \in B \rangle$ 
    by auto
  moreover have  $?z \leq b$ 
    using  $\langle b \in B \rangle \langle x < b \rangle$ 
    by (intro cInf-lower) auto
  moreover have  $b \in U$ 
    using  $\langle x \leq ?z \rangle \langle ?z \leq b \rangle \langle b < \text{min } a$   $y \rangle$ 
    by (intro *[OF  $\langle x \in U \rangle \langle y \in U \rangle$ ]) (auto simp: less-imp-le)
  ultimately show ?thesis
    by (intro bexI[of -  $b$ ]) auto
qed
then have False
  using  $\langle ?z \leq y \rangle \langle ?z \in A \rangle \langle y \in B \rangle \langle y \in U \rangle \langle A \cap B \cap U = \{\} \rangle$ 
  unfolding le-less by blast
}
note not-disjoint = this

fix  $A$   $B$  assume  $AB$ : open  $A$  open  $B$   $U \subseteq A \cup B$   $A \cap B \cap U = \{\}$ 
moreover assume  $A \cap U \neq \{\}$  then obtain  $x$  where  $x: x \in U$   $x \in A$  by auto
moreover assume  $B \cap U \neq \{\}$  then obtain  $y$  where  $y: y \in U$   $y \in B$  by auto
moreover note not-disjoint[of  $B$   $A$   $y$   $x$ ] not-disjoint[of  $A$   $B$   $x$   $y$ ]
ultimately show False
  by (cases  $x$   $y$  rule: linorder-cases) auto
qed

lemma connected-iff-interval: connected  $U \longleftrightarrow (\forall x \in U. \forall y \in U. \forall z. x \leq z \longrightarrow z \leq y \longrightarrow z \in U)$ 
  for  $U :: 'a::\text{linear-continuum-topology set}$ 
  by (auto intro: connectedI-interval dest: connectedD-interval)

lemma connected-UNIV[simp]: connected (UNIV::'a::linear-continuum-topology set)

```


by (simp add: connected-iff-interval)

lemma *connected-Ioi*[simp]: *connected* $\{a<..\}$
 for $a :: 'a::linear-continuum-topology$
 by (auto simp: connected-iff-interval)

lemma *connected-Ici*[simp]: *connected* $\{a..\}$
 for $a :: 'a::linear-continuum-topology$
 by (auto simp: connected-iff-interval)

lemma *connected-Iio*[simp]: *connected* $\{..<a\}$
 for $a :: 'a::linear-continuum-topology$
 by (auto simp: connected-iff-interval)

lemma *connected-Iic*[simp]: *connected* $\{..a\}$
 for $a :: 'a::linear-continuum-topology$
 by (auto simp: connected-iff-interval)

lemma *connected-Ioo*[simp]: *connected* $\{a<..**b\}**$
 for $a b :: 'a::linear-continuum-topology$
 unfolding *connected-iff-interval* by auto

lemma *connected-Ioc*[simp]: *connected* $\{a<..**b\}**$
 for $a b :: 'a::linear-continuum-topology$
 by (auto simp: connected-iff-interval)

lemma *connected-Ico*[simp]: *connected* $\{a..**b\}**$
 for $a b :: 'a::linear-continuum-topology$
 by (auto simp: connected-iff-interval)

lemma *connected-Icc*[simp]: *connected* $\{a..**b\}**$
 for $a b :: 'a::linear-continuum-topology$
 by (auto simp: connected-iff-interval)

lemma *connected-contains-Ioo*:
 fixes $A :: 'a :: linorder-topology set$
 assumes *connected* A $a \in A$ $b \in A$ shows $\{a <..**b\} \subseteq A**$
 using *connectedD-interval*[OF *assms*] by (simp add: *subset-eq Ball-def less-imp-le*)

lemma *connected-contains-Icc*:
 fixes $A :: 'a::linorder-topology set$
 assumes *connected* A $a \in A$ $b \in A$
 shows $\{a..**b\} \subseteq A**$

proof

fix x assume $x \in \{a..**b\}**$

then have $x = a \vee x = b \vee x \in \{a<..**b\}**$

by *auto*

then show $x \in A$

using *assms connected-contains-Ioo*[of A a b] by *auto*

qed

100.1 Intermediate Value Theorem

lemma *IVT'*:

fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
 assumes $y: f\ a \leq y \wedge y \leq f\ b \wedge a \leq b$
 and $*$: *continuous-on* $\{a .. b\}$ f
 shows $\exists x. a \leq x \wedge x \leq b \wedge f\ x = y$
 proof –
 have *connected* $\{a..b\}$
 unfolding *connected-iff-interval* by *auto*
 from *connected-continuous-image*[*OF* * *this*, *THEN* *connectedD-interval*, of $f\ a$
 $f\ b\ y$] y
 show ?*thesis*
 by (*auto simp add: atLeastAtMost-def atLeast-def atMost-def*)
 qed

lemma *IVT2'*:

fixes $f :: 'a :: linear-continuum-topology \Rightarrow 'b :: linorder-topology$
 assumes $y: f\ b \leq y \wedge y \leq f\ a \wedge a \leq b$
 and $*$: *continuous-on* $\{a .. b\}$ f
 shows $\exists x. a \leq x \wedge x \leq b \wedge f\ x = y$
 proof –
 have *connected* $\{a..b\}$
 unfolding *connected-iff-interval* by *auto*
 from *connected-continuous-image*[*OF* * *this*, *THEN* *connectedD-interval*, of $f\ b$
 $f\ a\ y$] y
 show ?*thesis*
 by (*auto simp add: atLeastAtMost-def atLeast-def atMost-def*)
 qed

lemma *IVT*:

fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
 shows $f\ a \leq y \Longrightarrow y \leq f\ b \Longrightarrow a \leq b \Longrightarrow (\forall x. a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x)$
 \Longrightarrow
 $\exists x. a \leq x \wedge x \leq b \wedge f\ x = y$
 by (*rule IVT'*) (*auto intro: continuous-at-imp-continuous-on*)

lemma *IVT2*:

fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
 shows $f\ b \leq y \Longrightarrow y \leq f\ a \Longrightarrow a \leq b \Longrightarrow (\forall x. a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x)$
 \Longrightarrow
 $\exists x. a \leq x \wedge x \leq b \wedge f\ x = y$
 by (*rule IVT2'*) (*auto intro: continuous-at-imp-continuous-on*)

lemma *continuous-inj-imp-mono*:

fixes $f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology$
 assumes $x: a < x < b$

```

    and cont: continuous-on {a..b} f
    and inj: inj-on f {a..b}
  shows (f a < f x ∧ f x < f b) ∨ (f b < f x ∧ f x < f a)
proof -
  note I = inj-on-eq-iff[OF inj]
  {
    assume f x < f a f x < f b
    then obtain s t where x ≤ s s ≤ b a ≤ t t ≤ x f s = f t f x < f s
      using IVT'[of f x min (f a) (f b) b] IVT2'[of f x min (f a) (f b) a] x
      by (auto simp: continuous-on-subset[OF cont] less-imp-le)
    with x I have False by auto
  }
  moreover
  {
    assume f a < f x f b < f x
    then obtain s t where x ≤ s s ≤ b a ≤ t t ≤ x f s = f t f s < f x
      using IVT'[of f a max (f a) (f b) x] IVT2'[of f b max (f a) (f b) x] x
      by (auto simp: continuous-on-subset[OF cont] less-imp-le)
    with x I have False by auto
  }
  ultimately show ?thesis
    using I[of a x] I[of x b] x less-trans[OF x]
    by (auto simp add: le-less less-imp-neq neq-iff)
qed

```

lemma continuous-at-Sup-mono:

```

fixes f :: 'a::{linorder-topology, conditionally-complete-linorder} ⇒
  'b::{linorder-topology, conditionally-complete-linorder}
assumes mono f
  and cont: continuous (at-left (Sup S)) f
  and S: S ≠ {} bdd-above S
shows f (Sup S) = (SUP s∈S. f s)
proof (rule antisym)
  have f: (f ⟶ f (Sup S)) (at-left (Sup S))
    using cont unfolding continuous-within .
  show f (Sup S) ≤ (SUP s∈S. f s)
  proof cases
    assume Sup S ∈ S
    then show ?thesis
      by (rule cSUP-upper) (auto intro: bdd-above-image-mono S ‹mono f›)
  next
    assume Sup S ∉ S
    from ‹S ≠ {}› obtain s where s ∈ S
      by auto
    with ‹Sup S ∉ S› S have s < Sup S
      unfolding less-le by (blast intro: cSup-upper)
    show ?thesis
  proof (rule ccontr)
    assume ¬ ?thesis

```

```

with order-tendstoD(1)[OF f, of SUP s∈S. f s] obtain b where b < Sup S
and *:  $\bigwedge y. b < y \implies y < \text{Sup } S \implies (\text{SUP } s \in S. f s) < f y$ 
by (auto simp: not-le eventually-at-left[OF  $\langle s < \text{Sup } S \rangle$ ])
with  $\langle S \neq \{\} \rangle$  obtain c where  $c \in S$  b < c
using less-cSupD[of S b] by auto
with  $\langle \text{Sup } S \notin S \rangle$  S have c < Sup S
unfolding less-le by (blast intro: cSup-upper)
from *[OF  $\langle b < c \rangle \langle c < \text{Sup } S \rangle$ ] cSUP-upper[OF  $\langle c \in S \rangle$  bdd-above-image-mono[of
f]]
show False
by (auto simp: assms)
qed
qed
qed (intro cSUP-least  $\langle \text{mono } f \rangle$ [THEN monoD] cSup-upper S)

```

lemma continuous-at-Sup-antimono:

```

fixes f :: 'a::{\linorder-topology, conditionally-complete-linorder}  $\Rightarrow$ 
'b::{\linorder-topology, conditionally-complete-linorder}
assumes antimono f
and cont: continuous (at-left (Sup S)) f
and S:  $S \neq \{\}$  bdd-above S
shows f (Sup S) = (INF s∈S. f s)
proof (rule antisym)
have f:  $(f \longrightarrow f (\text{Sup } S))$  (at-left (Sup S))
using cont unfolding continuous-within .
show  $(\text{INF } s \in S. f s) \leq f (\text{Sup } S)$ 
proof cases
assume  $\text{Sup } S \in S$ 
then show ?thesis
by (intro cINF-lower) (auto intro: bdd-below-image-antimono S  $\langle \text{antimono } f \rangle$ )
next
assume  $\text{Sup } S \notin S$ 
from  $\langle S \neq \{\} \rangle$  obtain s where  $s \in S$ 
by auto
with  $\langle \text{Sup } S \notin S \rangle$  S have s < Sup S
unfolding less-le by (blast intro: cSup-upper)
show ?thesis
proof (rule ccontr)
assume  $\neg ?thesis$ 
with order-tendstoD(2)[OF f, of INF s∈S. f s] obtain b where b < Sup S
and *:  $\bigwedge y. b < y \implies y < \text{Sup } S \implies f y < (\text{INF } s \in S. f s)$ 
by (auto simp: not-le eventually-at-left[OF  $\langle s < \text{Sup } S \rangle$ ])
with  $\langle S \neq \{\} \rangle$  obtain c where  $c \in S$  b < c
using less-cSupD[of S b] by auto
with  $\langle \text{Sup } S \notin S \rangle$  S have c < Sup S
unfolding less-le by (blast intro: cSup-upper)
from *[OF  $\langle b < c \rangle \langle c < \text{Sup } S \rangle$ ] cINF-lower[OF bdd-below-image-antimono,
of f S c]  $\langle c \in S \rangle$ 
show False

```

```

    by (auto simp: assms)
  qed
  qed
  qed (intro cINF-greatest ⟨antimono f⟩[THEN antimonoD] cSup-upper S)

lemma continuous-at-Inf-mono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒
    'b::{linorder-topology,conditionally-complete-linorder}
  assumes mono f
    and cont: continuous (at-right (Inf S)) f
    and S: S ≠ {} bdd-below S
  shows f (Inf S) = (INF s∈S. f s)
proof (rule antisym)
  have f: (f ⟶ f (Inf S)) (at-right (Inf S))
    using cont unfolding continuous-within .
  show (INF s∈S. f s) ≤ f (Inf S)
proof cases
  assume Inf S ∈ S
  then show ?thesis
    by (rule cINF-lower[rotated]) (auto intro: bdd-below-image-mono S ⟨mono f⟩)
next
  assume Inf S ∉ S
  from ⟨S ≠ {}⟩ obtain s where s ∈ S
    by auto
  with ⟨Inf S ∉ S⟩ S have Inf S < s
    unfolding less-le by (blast intro: cInf-lower)
  show ?thesis
proof (rule ccontr)
  assume ¬ ?thesis
  with order-tendstoD(2)[OF f, of INF s∈S. f s] obtain b where Inf S < b
    and *: ∧y. Inf S < y ⟹ y < b ⟹ f y < (INF s∈S. f s)
    by (auto simp: not-le eventually-at-right[OF ⟨Inf S < s⟩])
  with ⟨S ≠ {}⟩ obtain c where c ∈ S c < b
    using cInf-lessD[of S b] by auto
  with ⟨Inf S ∉ S⟩ S have Inf S < c
    unfolding less-le by (blast intro: cInf-lower)
  from *[OF ⟨Inf S < c⟩ ⟨c < b⟩] cINF-lower[OF bdd-below-image-mono[of f]
    ⟨c ∈ S⟩]
  show False
    by (auto simp: assms)
  qed
  qed
  qed (intro cINF-greatest ⟨mono f⟩[THEN monoD] cInf-lower ⟨bdd-below S⟩ ⟨S ≠
    {}⟩)

```

lemma *continuous-at-Inf-antimono*:

```

  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒
    'b::{linorder-topology,conditionally-complete-linorder}
  assumes antimono f

```

```

    and cont: continuous (at-right (Inf S)) f
    and S: S ≠ {} bdd-below S
  shows f (Inf S) = (SUP s∈S. f s)
proof (rule antisym)
  have f: (f ⟶ f (Inf S)) (at-right (Inf S))
    using cont unfolding continuous-within .
  show f (Inf S) ≤ (SUP s∈S. f s)
proof cases
  assume Inf S ∈ S
  then show ?thesis
    by (rule cSUP-upper) (auto intro: bdd-above-image-antimono S ⟨antimono f⟩)
next
  assume Inf S ∉ S
  from ⟨S ≠ {}⟩ obtain s where s ∈ S
    by auto
  with ⟨Inf S ∉ S⟩ S have Inf S < s
    unfolding less-le by (blast intro: cInf-lower)
  show ?thesis
proof (rule ccontr)
  assume ¬ ?thesis
  with order-tendstoD(1)[OF f, of SUP s∈S. f s] obtain b where Inf S < b
    and *: ∧y. Inf S < y ⟹ y < b ⟹ (SUP s∈S. f s) < f y
    by (auto simp: not-le eventually-at-right[OF ⟨Inf S < s⟩])
  with ⟨S ≠ {}⟩ obtain c where c ∈ S c < b
    using cInf-lessD[of S b] by auto
  with ⟨Inf S ∉ S⟩ S have Inf S < c
    unfolding less-le by (blast intro: cInf-lower)
  from *[OF ⟨Inf S < c⟩ ⟨c < b⟩] cSUP-upper[OF ⟨c ∈ S⟩ bdd-above-image-antimono[of
f]]
    show False
    by (auto simp: assms)
qed
qed (intro cSUP-least ⟨antimono f⟩[THEN antimonoD] cInf-lower S)

```

100.2 Uniform spaces

```

class uniformity =
  fixes uniformity :: ('a × 'a) filter
begin

abbreviation uniformity-on :: 'a set ⇒ ('a × 'a) filter
  where uniformity-on s ≡ inf uniformity (principal (s×s))

end

```

lemma *uniformity-Abort:*

```

uniformity =
  Filter.abstract-filter (λu. Code.abort (STR "uniformity is not executable") (λu.

```

uniformity))
by *simp*

class *open-uniformity* = *open* + *uniformity* +
assumes *open-uniformity*:
 $\bigwedge U. \text{open } U \longleftrightarrow (\forall x \in U. \text{eventually } (\lambda(x', y). x' = x \longrightarrow y \in U) \text{uniformity})$
begin

subclass *topological-space*
by *standard* (*force elim: eventually-mono eventually-elim2 simp: split-beta' open-uniformity*) +
end

class *uniform-space* = *open-uniformity* +
assumes *uniformity-refl*: *eventually E uniformity* $\implies E(x, x)$
and *uniformity-sym*: *eventually E uniformity* $\implies \text{eventually } (\lambda(x, y). E(y, x))$
uniformity
and *uniformity-trans*:
eventually E uniformity \implies
 $\exists D. \text{eventually } D \text{uniformity} \wedge (\forall x y z. D(x, y) \longrightarrow D(y, z) \longrightarrow E(x, z))$
begin

lemma *uniformity-bot*: *uniformity* \neq *bot*
using *uniformity-refl* **by** *auto*

lemma *uniformity-trans'*:
eventually E uniformity \implies
 $\text{eventually } (\lambda((x, y), (y', z)). y = y' \longrightarrow E(x, z)) (\text{uniformity} \times_F \text{uniformity})$
by (*drule uniformity-trans*) (*auto simp add: eventually-prod-same*)

lemma *uniformity-transE*:
assumes *eventually E uniformity*
obtains *D* **where** *eventually D uniformity* $\bigwedge x y z. D(x, y) \implies D(y, z) \implies E(x, z)$
using *uniformity-trans* [*OF assms*] **by** *auto*

lemma *eventually-nhds-uniformity*:
 $\text{eventually } P (\text{nhds } x) \longleftrightarrow \text{eventually } (\lambda(x', y). x' = x \longrightarrow P y) \text{uniformity}$
(is - $\longleftrightarrow ?N P x$)
unfolding *eventually-nhds*
proof *safe*
assume *: $?N P x$
have $?N (?N P) x$ **if** $?N P x$ **for** *x*
proof –
from *that* **obtain** *D* **where** *ev: eventually D uniformity*
and *D*: $D(a, b) \implies D(b, c) \implies \text{case } (a, c) \text{ of } (x', y) \Rightarrow x' = x \longrightarrow P y$
for *a b c*
by (*rule uniformity-transE*) *simp*
from *ev* **show** *?thesis*

by *eventually-elim* (*insert ev D*, *force elim: eventually-mono split: prod.split*)
qed
then have *open* {*x*. *?N P x*}
 by (*simp add: open-uniformity*)
then show $\exists S. \text{open } S \wedge x \in S \wedge (\forall x \in S. P x)$
 by (*intro exI[of - {x. ?N P x}]*) (*auto dest: uniformity-refl simp: **)
qed (*force simp add: open-uniformity elim: eventually-mono*)

100.2.1 Totally bounded sets

definition *totally-bounded* :: 'a set \Rightarrow bool

where *totally-bounded* $S \longleftrightarrow$

$(\forall E. \text{eventually } E \text{ uniformity} \longrightarrow (\exists X. \text{finite } X \wedge (\forall s \in S. \exists x \in X. E (x, s))))$

lemma *totally-bounded-empty[iff]*: *totally-bounded* {}

by (*auto simp add: totally-bounded-def*)

lemma *totally-bounded-subset*: *totally-bounded* $S \Longrightarrow T \subseteq S \Longrightarrow \text{totally-bounded } T$

by (*fastforce simp add: totally-bounded-def*)

lemma *totally-bounded-Union[intro]*:

assumes $M: \text{finite } M \wedge S. S \in M \Longrightarrow \text{totally-bounded } S$

shows *totally-bounded* $(\bigcup M)$

unfolding *totally-bounded-def*

proof *safe*

fix E

assume *eventually* E *uniformity*

with M **obtain** X **where** $\forall S \in M. \text{finite } (X S) \wedge (\forall s \in S. \exists x \in X S. E (x, s))$

by (*metis totally-bounded-def*)

with $\langle \text{finite } M \rangle$ **show** $\exists X. \text{finite } X \wedge (\forall s \in \bigcup M. \exists x \in X. E (x, s))$

by (*intro exI[of - $\bigcup S \in M. X S$] force*)

qed

100.2.2 Cauchy filter

definition *cauchy-filter* :: 'a filter \Rightarrow bool

where *cauchy-filter* $F \longleftrightarrow F \times_F F \leq \text{uniformity}$

definition *Cauchy* :: (nat \Rightarrow 'a) \Rightarrow bool

where *Cauchy-uniform*: *Cauchy* $X = \text{cauchy-filter } (\text{filtermap } X \text{ sequentially})$

lemma *Cauchy-uniform-iff*:

Cauchy $X \longleftrightarrow (\forall P. \text{eventually } P \text{ uniformity} \longrightarrow (\exists N. \forall n \geq N. \forall m \geq N. P (X n, X m)))$

unfolding *Cauchy-uniform cauchy-filter-def le-filter-def eventually-prod-same eventually-filtermap eventually-sequentially*

proof *safe*

let $?U = \lambda P. \text{eventually } P \text{ uniformity}$

{


```

fix P
  assume ?U P  $\forall P$ . ?U P  $\longrightarrow$  ( $\exists Q$ . ( $\exists N$ .  $\forall n \geq N$ .  $Q(X n)$ )  $\wedge$  ( $\forall x y$ .  $Q x \longrightarrow Q y \longrightarrow P(x, y)$ ))
  then obtain Q N where  $\bigwedge n$ .  $n \geq N \implies Q(X n) \bigwedge x y$ .  $Q x \implies Q y \implies P(x, y)$ 
    by metis
    then show  $\exists N$ .  $\forall n \geq N$ .  $\forall m \geq N$ .  $P(X n, X m)$ 
      by blast
  next
  fix P
  assume ?U P and P:  $\forall P$ . ?U P  $\longrightarrow$  ( $\exists N$ .  $\forall n \geq N$ .  $\forall m \geq N$ .  $P(X n, X m)$ )
  then obtain Q where ?U Q and Q:  $\bigwedge x y z$ .  $Q(x, y) \implies Q(y, z) \implies P(x, z)$ 
    by (auto elim: uniformity-transE)
  then have ?U ( $\lambda x$ .  $Q x \wedge (\lambda(x, y)$ .  $Q(y, x)) x$ )
    unfolding eventually-conj-iff by (simp add: uniformity-sym)
  from P[rule-format, OF this]
  obtain N where N:  $\bigwedge n m$ .  $n \geq N \implies m \geq N \implies Q(X n, X m) \wedge Q(X m, X n)$ 
    by auto
  show  $\exists Q$ . ( $\exists N$ .  $\forall n \geq N$ .  $Q(X n)$ )  $\wedge$  ( $\forall x y$ .  $Q x \longrightarrow Q y \longrightarrow P(x, y)$ )
  proof (safe intro!: exI[of -  $\lambda x$ .  $\forall n \geq N$ .  $Q(x, X n) \wedge Q(X n, x)$ ] exI[of - N])
    fix x y
    assume  $\forall n \geq N$ .  $Q(x, X n) \wedge Q(X n, x) \forall n \geq N$ .  $Q(y, X n) \wedge Q(X n, y)$ 
    then have  $Q(x, X N) Q(X N, y)$  by auto
    then show  $P(x, y)$ 
      by (rule Q)
    qed
  }
qed

```

lemma nhds-imp-cauchy-filter:

assumes *: $F \leq \text{nhds } x$

shows *cauchy-filter* F

proof –

have $F \times_F F \leq \text{nhds } x \times_F \text{nhds } x$

by (intro prod-filter-mono *)

also have $\dots \leq \text{uniformity}$

unfolding le-filter-def eventually-nhds-uniformity eventually-prod-same

proof safe

fix P

assume eventually P *uniformity*

then obtain Ql **where** ev: eventually Ql *uniformity*

and $Ql(x, y) \implies Ql(y, z) \implies P(x, z)$ **for** x y z

by (rule uniformity-transE) simp

with ev[THEN uniformity-sym]

show $\exists Q$. eventually ($\lambda(x', y)$. $x' = x \longrightarrow Q y$) *uniformity* \wedge
 $(\forall x y$. $Q x \longrightarrow Q y \longrightarrow P(x, y))$

by (rule-tac exI[of - λy. Ql (y, x) ∧ Ql (x, y)]) (fastforce elim: eventually-elim2)
 qed
 finally show ?thesis
 by (simp add: cauchy-filter-def)
 qed

lemma LIMSEQ-imp-Cauchy: $X \longrightarrow x \implies \text{Cauchy } X$
unfolding Cauchy-uniform filterlim-def **by** (intro nhds-imp-cauchy-filter)

lemma Cauchy-subseq-Cauchy:
assumes Cauchy X strict-mono f
shows Cauchy (X ◦ f)
unfolding Cauchy-uniform comp-def filtermap-filtermap[symmetric] cauchy-filter-def
by (rule order-trans[OF - ⟨Cauchy X⟩[unfolded Cauchy-uniform cauchy-filter-def]])
 (intro prod-filter-mono filtermap-mono filterlim-subseq[OF ⟨strict-mono f⟩,
 unfolded filterlim-def])

lemma convergent-Cauchy: convergent X \implies Cauchy X
unfolding convergent-def **by** (erule exE, erule LIMSEQ-imp-Cauchy)

definition complete :: 'a set \implies bool
where complete-uniform: complete S \longleftrightarrow
 ($\forall F \leq$ principal S. $F \neq$ bot \longrightarrow cauchy-filter F \longrightarrow ($\exists x \in S. F \leq$ nhds x))

lemma (in uniform-space) cauchy-filter-complete-converges:
assumes cauchy-filter F complete A $F \leq$ principal A $F \neq$ bot
shows $\exists c. F \leq$ nhds c
using assms **unfolding** complete-uniform **by** blast

end

100.2.3 Uniformly continuous functions

definition uniformly-continuous-on :: 'a set \implies ('a::uniform-space \implies 'b::uniform-space)
 \implies bool

where uniformly-continuous-on-uniformity: uniformly-continuous-on s f \longleftrightarrow
 (LIM (x, y) (uniformity-on s). (f x, f y) :> uniformity)

lemma uniformly-continuous-onD:
 uniformly-continuous-on s f \implies eventually E uniformity \implies
 eventually (λ(x, y). x ∈ s \longrightarrow y ∈ s \longrightarrow E (f x, f y)) uniformity
by (simp add: uniformly-continuous-on-uniformity filterlim-iff
 eventually-inf-principal split-beta' mem-Times-iff imp-conjL)

lemma uniformly-continuous-on-const[continuous-intros]: uniformly-continuous-on
 s (λx. c)
by (auto simp: uniformly-continuous-on-uniformity filterlim-iff uniformity-refl)

lemma uniformly-continuous-on-id[continuous-intros]: uniformly-continuous-on s

($\lambda x. x$)
by (*auto simp: uniformly-continuous-on-uniformity filterlim-def*)

lemma *uniformly-continuous-on-compose*:
uniformly-continuous-on s g \implies uniformly-continuous-on (g's) f \implies
uniformly-continuous-on s ($\lambda x. f (g x)$)
using *filterlim-compose[$\text{of } \lambda(x, y). (f x, f y)$ uniformity*
uniformity-on (g's) $\lambda(x, y). (g x, g y)$ uniformity-on s]
by (*simp add: split-beta' uniformly-continuous-on-uniformity*
filterlim-inf filterlim-principal eventually-inf-principal mem-Times-iff)

lemma *uniformly-continuous-imp-continuous*:
assumes *f: uniformly-continuous-on s f*
shows *continuous-on s f*
by (*auto simp: filterlim-iff eventually-at-filter eventually-nhds-uniformity contin-*
uous-on-def
elim: eventually-mono dest!: uniformly-continuous-onD[OF f])

101 Product Topology

101.1 Product is a topological space

instantiation *prod :: (topological-space, topological-space) topological-space*
begin

definition *open-prod-def[code del]*:
open (S :: ('a \times 'b) set) \longleftrightarrow
($\forall x \in S. \exists A B. \text{open } A \wedge \text{open } B \wedge x \in A \times B \wedge A \times B \subseteq S$)

lemma *open-prod-elim*:
assumes *open S and $x \in S$*
obtains *A B where open A and open B and $x \in A \times B$ and $A \times B \subseteq S$*
using *assms unfolding open-prod-def by fast*

lemma *open-prod-intro*:
assumes *$\bigwedge x. x \in S \implies \exists A B. \text{open } A \wedge \text{open } B \wedge x \in A \times B \wedge A \times B \subseteq S$*
shows *open S*
using *assms unfolding open-prod-def by fast*

instance

proof

show *open (UNIV :: ('a \times 'b) set)*
unfolding *open-prod-def by auto*

next

fix *S T :: ('a \times 'b) set*
assume *open S open T*
show *open (S \cap T)*
proof (*rule open-prod-intro*)
fix *x*

```

assume  $x: x \in S \cap T$ 
from  $x$  have  $x \in S$  by simp
obtain  $Sa\ Sb$  where  $A: open\ Sa\ open\ Sb\ x \in Sa \times Sb\ Sa \times Sb \subseteq S$ 
  using  $\langle open\ S \rangle$  and  $\langle x \in S \rangle$  by (rule open-prod-elim)
from  $x$  have  $x \in T$  by simp
obtain  $Ta\ Tb$  where  $B: open\ Ta\ open\ Tb\ x \in Ta \times Tb\ Ta \times Tb \subseteq T$ 
  using  $\langle open\ T \rangle$  and  $\langle x \in T \rangle$  by (rule open-prod-elim)
let  $?A = Sa \cap Ta$  and  $?B = Sb \cap Tb$ 
have  $open\ ?A \wedge open\ ?B \wedge x \in ?A \times ?B \wedge ?A \times ?B \subseteq S \cap T$ 
  using  $A\ B$  by (auto simp add: open-Int)
then show  $\exists A\ B. open\ A \wedge open\ B \wedge x \in A \times B \wedge A \times B \subseteq S \cap T$ 
  by fast
qed
next
fix  $K :: ('a \times 'b)\ set\ set$ 
assume  $\forall S \in K. open\ S$ 
then show  $open\ (\bigcup K)$ 
  unfolding open-prod-def by fast
qed

end

declare [[code abort: open :: ('a::topological-space \times 'b::topological-space)\ set \Rightarrow bool]]

lemma open-Times:  $open\ S \Longrightarrow open\ T \Longrightarrow open\ (S \times T)$ 
  unfolding open-prod-def by auto

lemma fst-vimage-eq-Times:  $fst\ -' S = S \times UNIV$ 
  by auto

lemma snd-vimage-eq-Times:  $snd\ -' S = UNIV \times S$ 
  by auto

lemma open-vimage-fst:  $open\ S \Longrightarrow open\ (fst\ -' S)$ 
  by (simp add: fst-vimage-eq-Times open-Times)

lemma open-vimage-snd:  $open\ S \Longrightarrow open\ (snd\ -' S)$ 
  by (simp add: snd-vimage-eq-Times open-Times)

lemma closed-vimage-fst:  $closed\ S \Longrightarrow closed\ (fst\ -' S)$ 
  unfolding closed-open vimage-Compl [symmetric]
  by (rule open-vimage-fst)

lemma closed-vimage-snd:  $closed\ S \Longrightarrow closed\ (snd\ -' S)$ 
  unfolding closed-open vimage-Compl [symmetric]
  by (rule open-vimage-snd)

lemma closed-Times:  $closed\ S \Longrightarrow closed\ T \Longrightarrow closed\ (S \times T)$ 

```

proof –

have $S \times T = (fst \text{ -' } S) \cap (snd \text{ -' } T)$
by *auto*
then show $closed\ S \implies closed\ T \implies closed\ (S \times T)$
by (*simp add: closed-vimage-fst closed-vimage-snd closed-Int*)
qed

lemma *subset-fst-imageI*: $A \times B \subseteq S \implies y \in B \implies A \subseteq fst \text{ -' } S$
unfolding *image-def subset-eq* **by** *force*

lemma *subset-snd-imageI*: $A \times B \subseteq S \implies x \in A \implies B \subseteq snd \text{ -' } S$
unfolding *image-def subset-eq* **by** *force*

lemma *open-image-fst*:

assumes *open S*
shows *open (fst -' S)*
proof (*rule openI*)
fix *x*
assume $x \in fst \text{ -' } S$
then obtain *y* **where** $(x, y) \in S$
by *auto*
then obtain *A B* **where** *open A open B x \in A y \in B A \times B \subseteq S*
using $\langle open\ S \rangle$ **unfolding** *open-prod-def* **by** *auto*
from $\langle A \times B \subseteq S \rangle \langle y \in B \rangle$ **have** $A \subseteq fst \text{ -' } S$
by (*rule subset-fst-imageI*)
with $\langle open\ A \rangle \langle x \in A \rangle$ **have** $open\ A \wedge x \in A \wedge A \subseteq fst \text{ -' } S$
by *simp*
then show $\exists T. open\ T \wedge x \in T \wedge T \subseteq fst \text{ -' } S ..$
qed

lemma *open-image-snd*:

assumes *open S*
shows *open (snd -' S)*
proof (*rule openI*)
fix *y*
assume $y \in snd \text{ -' } S$
then obtain *x* **where** $(x, y) \in S$
by *auto*
then obtain *A B* **where** *open A open B x \in A y \in B A \times B \subseteq S*
using $\langle open\ S \rangle$ **unfolding** *open-prod-def* **by** *auto*
from $\langle A \times B \subseteq S \rangle \langle x \in A \rangle$ **have** $B \subseteq snd \text{ -' } S$
by (*rule subset-snd-imageI*)
with $\langle open\ B \rangle \langle y \in B \rangle$ **have** $open\ B \wedge y \in B \wedge B \subseteq snd \text{ -' } S$
by *simp*
then show $\exists T. open\ T \wedge y \in T \wedge T \subseteq snd \text{ -' } S ..$
qed

lemma *nhds-prod*: $nhds\ (a, b) = nhds\ a \times_F nhds\ b$
unfolding *nhds-def*

proof (*subst prod-filter-INF, auto intro!: antisym INF-greatest simp: principal-prod-principal*)
fix $S T$
assume $\text{open } S \ a \in S \ \text{open } T \ b \in T$
then show $(\text{INF } x \in \{S. \text{open } S \wedge (a, b) \in S\}. \text{principal } x) \leq \text{principal } (S \times T)$
by (*intro INF-lower*) (*auto intro!: open-Times*)
next
fix S'
assume $\text{open } S' \ (a, b) \in S'$
then obtain $S T$ **where** $\text{open } S \ a \in S \ \text{open } T \ b \in T \ S \times T \subseteq S'$
by (*auto elim: open-prod-elim*)
then show $(\text{INF } x \in \{S. \text{open } S \wedge a \in S\}. \text{INF } y \in \{S. \text{open } S \wedge b \in S\}. \text{principal } (x \times y)) \leq \text{principal } S'$
by (*auto intro!: INF-lower2*)
qed

101.1.1 Continuity of operations

lemma *tendsto-fst* [*tendsto-intros*]:
assumes $(f \longrightarrow a) F$
shows $((\lambda x. \text{fst } (f x)) \longrightarrow \text{fst } a) F$
proof (*rule topological-tendstoI*)
fix S
assume $\text{open } S$ **and** $\text{fst } a \in S$
then have $\text{open } (\text{fst } -' S)$ **and** $a \in \text{fst } -' S$
by (*simp-all add: open-vimage-fst*)
with *assms* **have** *eventually* $(\lambda x. f x \in \text{fst } -' S) F$
by (*rule topological-tendstoD*)
then show *eventually* $(\lambda x. \text{fst } (f x) \in S) F$
by *simp*
qed

lemma *tendsto-snd* [*tendsto-intros*]:
assumes $(f \longrightarrow a) F$
shows $((\lambda x. \text{snd } (f x)) \longrightarrow \text{snd } a) F$
proof (*rule topological-tendstoI*)
fix S
assume $\text{open } S$ **and** $\text{snd } a \in S$
then have $\text{open } (\text{snd } -' S)$ **and** $a \in \text{snd } -' S$
by (*simp-all add: open-vimage-snd*)
with *assms* **have** *eventually* $(\lambda x. f x \in \text{snd } -' S) F$
by (*rule topological-tendstoD*)
then show *eventually* $(\lambda x. \text{snd } (f x) \in S) F$
by *simp*
qed

lemma *tendsto-Pair* [*tendsto-intros*]:
assumes $(f \longrightarrow a) F$ **and** $(g \longrightarrow b) F$
shows $((\lambda x. (f x, g x)) \longrightarrow (a, b)) F$

unfolding *nhds-prod* **using** *assms* **by** (*rule filterlim-Pair*)

lemma *continuous-fst*[*continuous-intros*]: *continuous F f* \implies *continuous F* ($\lambda x.$ *fst* (*f x*))

unfolding *continuous-def* **by** (*rule tendsto-fst*)

lemma *continuous-snd*[*continuous-intros*]: *continuous F f* \implies *continuous F* ($\lambda x.$ *snd* (*f x*))

unfolding *continuous-def* **by** (*rule tendsto-snd*)

lemma *continuous-Pair*[*continuous-intros*]:

continuous F f \implies *continuous F g* \implies *continuous F* ($\lambda x.$ (*f x*, *g x*))

unfolding *continuous-def* **by** (*rule tendsto-Pair*)

lemma *continuous-on-fst*[*continuous-intros*]:

continuous-on s f \implies *continuous-on s* ($\lambda x.$ *fst* (*f x*))

unfolding *continuous-on-def* **by** (*auto intro: tendsto-fst*)

lemma *continuous-on-snd*[*continuous-intros*]:

continuous-on s f \implies *continuous-on s* ($\lambda x.$ *snd* (*f x*))

unfolding *continuous-on-def* **by** (*auto intro: tendsto-snd*)

lemma *continuous-on-Pair*[*continuous-intros*]:

continuous-on s f \implies *continuous-on s g* \implies *continuous-on s* ($\lambda x.$ (*f x*, *g x*))

unfolding *continuous-on-def* **by** (*auto intro: tendsto-Pair*)

lemma *continuous-on-swap*[*continuous-intros*]: *continuous-on A prod.swap*

by (*simp add: prod.swap-def continuous-on-fst continuous-on-snd continuous-on-Pair continuous-on-id*)

lemma *continuous-on-swap-args*:

assumes *continuous-on* ($A \times B$) ($\lambda(x,y).$ *d x y*)

shows *continuous-on* ($B \times A$) ($\lambda(x,y).$ *d y x*)

proof –

have ($\lambda(x,y).$ *d y x*) = ($\lambda(x,y).$ *d x y*) \circ *prod.swap*

by *force*

then show *?thesis*

by (*metis assms continuous-on-compose continuous-on-swap product-swap*)

qed

lemma *isCont-fst* [*simp*]: *isCont f a* \implies *isCont* ($\lambda x.$ *fst* (*f x*)) *a*

by (*fact continuous-fst*)

lemma *isCont-snd* [*simp*]: *isCont f a* \implies *isCont* ($\lambda x.$ *snd* (*f x*)) *a*

by (*fact continuous-snd*)

lemma *isCont-Pair* [*simp*]: [*isCont f a*; *isCont g a*] \implies *isCont* ($\lambda x.$ (*f x*, *g x*)) *a*

by (*fact continuous-Pair*)

lemma *continuous-on-compose-Pair*:

assumes *f*: *continuous-on* (*Sigma* *A B*) ($\lambda(a, b). f\ a\ b$)
assumes *g*: *continuous-on* *C g*
assumes *h*: *continuous-on* *C h*
assumes *subset*: $\bigwedge c. c \in C \implies g\ c \in A \bigwedge c. c \in C \implies h\ c \in B\ (g\ c)$
shows *continuous-on* *C* ($\lambda c. f\ (g\ c)\ (h\ c)$)
using *continuous-on-compose2*[*OF f continuous-on-Pair*[*OF g h*]] *subset*
by *auto*

101.1.2 Connectedness of products

proposition *connected-Times*:

assumes *S*: *connected* *S* **and** *T*: *connected* *T*

shows *connected* (*S* \times *T*)

proof (*rule connectedI-const*)

fix *P*::*'a* \times *'b* \implies *bool*

assume *P*[*THEN continuous-on-compose2, continuous-intros*]: *continuous-on* (*S* \times *T*) *P*

have *continuous-on* *S* ($\lambda s. P\ (s, t)$) **if** *t* \in *T* **for** *t*

by (*auto intro!*: *continuous-intros that*)

from *connectedD-const*[*OF S this*]

obtain *c1* **where** $\bigwedge s\ t. t \in T \implies s \in S \implies P\ (s, t) = c1\ t$

by *metis*

moreover

have *continuous-on* *T* ($\lambda t. P\ (s, t)$) **if** *s* \in *S* **for** *s*

by (*auto intro!*: *continuous-intros that*)

from *connectedD-const*[*OF T this*]

obtain *c2* **where** $\bigwedge s\ t. t \in T \implies s \in S \implies P\ (s, t) = c2\ s$

by *metis*

ultimately show $\exists c. \forall s \in S \times T. P\ s = c$

by *auto*

qed

corollary *connected-Times-eq* [*simp*]:

connected (*S* \times *T*) \longleftrightarrow *S* = {} \vee *T* = {} \vee *connected* *S* \wedge *connected* *T* (**is** *?lhs* = *?rhs*)

proof

assume *L*: *?lhs*

show *?rhs*

proof *cases*

assume *S* \neq {} \wedge *T* \neq {}

moreover

have *connected* (*fst* ‘ (*S* \times *T*)) *connected* (*snd* ‘ (*S* \times *T*))

using *continuous-on-fst continuous-on-snd continuous-on-id*

by (*blast intro*: *connected-continuous-image* [*OF - L*])+

ultimately show *?thesis*

by *auto*

qed *auto*

qed (*auto simp*: *connected-Times*)

101.1.3 Separation axioms

instance *prod* :: (*t0-space*, *t0-space*) *t0-space*

proof

fix *x y* :: '*a* × '*b*

assume *x* ≠ *y*

then have *fst x* ≠ *fst y* ∨ *snd x* ≠ *snd y*

by (*simp add: prod-eq-iff*)

then show ∃ *U*. *open U* ∧ (*x* ∈ *U*) ≠ (*y* ∈ *U*)

by (*fast dest: t0-space elim: open-vimage-fst open-vimage-snd*)

qed

instance *prod* :: (*t1-space*, *t1-space*) *t1-space*

proof

fix *x y* :: '*a* × '*b*

assume *x* ≠ *y*

then have *fst x* ≠ *fst y* ∨ *snd x* ≠ *snd y*

by (*simp add: prod-eq-iff*)

then show ∃ *U*. *open U* ∧ *x* ∈ *U* ∧ *y* ∉ *U*

by (*fast dest: t1-space elim: open-vimage-fst open-vimage-snd*)

qed

instance *prod* :: (*t2-space*, *t2-space*) *t2-space*

proof

fix *x y* :: '*a* × '*b*

assume *x* ≠ *y*

then have *fst x* ≠ *fst y* ∨ *snd x* ≠ *snd y*

by (*simp add: prod-eq-iff*)

then show ∃ *U V*. *open U* ∧ *open V* ∧ *x* ∈ *U* ∧ *y* ∈ *V* ∧ *U* ∩ *V* = {}

by (*fast dest: hausdorff elim: open-vimage-fst open-vimage-snd*)

qed

lemma *isCont-swap[continuous-intros]*: *isCont prod.swap a*

using *continuous-on-eq-continuous-within continuous-on-swap* **by** *blast*

lemma *open-diagonal-complement*:

open {(*x,y*) | *x y*. *x* ≠ (*y*::'*a*::*t2-space*)}

proof –

have *open* {(*x, y*). *x* ≠ (*y*::'*a*)}

unfolding *split-def* **by** (*intro open-Collect-neq continuous-intros*)

also have {(*x, y*). *x* ≠ (*y*::'*a*)} = {(*x, y*) | *x y*. *x* ≠ (*y*::'*a*)}

by *auto*

finally show *?thesis* .

qed

lemma *closed-diagonal*:

closed {*y*. ∃ *x*::'*a*::*t2-space*. *y* = (*x,x*)}

proof –

have {*y*. ∃ *x*::'*a*. *y* = (*x,x*)} = *UNIV* – {(*x,y*) | *x y*. *x* ≠ *y*} **by** *auto*

then show *?thesis* **using** *open-diagonal-complement closed-Diff* **by** *auto*

qed

lemma *open-superdiagonal*:

open $\{(x,y) \mid x \ y. \ x > (y::'a::\{\text{linorder-topology}\})\}$

proof –

have *open* $\{(x, y). \ x > (y::'a)\}$

unfolding *split-def* **by** (*intro open-Collect-less continuous-intros*)

also have $\{(x, y). \ x > (y::'a)\} = \{(x, y) \mid x \ y. \ x > (y::'a)\}$

by *auto*

finally show *?thesis* .

qed

lemma *closed-subdiagonal*:

closed $\{(x,y) \mid x \ y. \ x \leq (y::'a::\{\text{linorder-topology}\})\}$

proof –

have $\{(x,y) \mid x \ y. \ x \leq (y::'a)\} = \text{UNIV} - \{(x,y) \mid x \ y. \ x > (y::'a)\}$ **by** *auto*

then show *?thesis* **using** *open-superdiagonal closed-Diff* **by** *auto*

qed

lemma *open-subdiagonal*:

open $\{(x,y) \mid x \ y. \ x < (y::'a::\{\text{linorder-topology}\})\}$

proof –

have *open* $\{(x, y). \ x < (y::'a)\}$

unfolding *split-def* **by** (*intro open-Collect-less continuous-intros*)

also have $\{(x, y). \ x < (y::'a)\} = \{(x, y) \mid x \ y. \ x < (y::'a)\}$

by *auto*

finally show *?thesis* .

qed

lemma *closed-superdiagonal*:

closed $\{(x,y) \mid x \ y. \ x \geq (y::('a::\{\text{linorder-topology}\}))\}$

proof –

have $\{(x,y) \mid x \ y. \ x \geq (y::'a)\} = \text{UNIV} - \{(x,y) \mid x \ y. \ x < y\}$ **by** *auto*

then show *?thesis* **using** *open-subdiagonal closed-Diff* **by** *auto*

qed

end

theory *Hull*

imports *Main*

begin

101.2 A generic notion of the convex, affine, conic hull, or closed "hull".

definition *hull* :: (*'a set* \Rightarrow *bool*) \Rightarrow *'a set* \Rightarrow *'a set* (**infixl** *hull* 75)

where *S hull s* = $\bigcap \{t. \ S \ t \wedge \ s \subseteq t\}$

lemma *hull-same*: $S\ s \Longrightarrow S\ \text{hull}\ s = s$
unfolding *hull-def* **by** *auto*

lemma *hull-in*: $(\bigwedge T. \text{Ball}\ T\ S \Longrightarrow S\ (\bigcap T)) \Longrightarrow S\ (S\ \text{hull}\ s)$
unfolding *hull-def* *Ball-def* **by** *auto*

lemma *hull-eq*: $(\bigwedge T. \text{Ball}\ T\ S \Longrightarrow S\ (\bigcap T)) \Longrightarrow (S\ \text{hull}\ s) = s \longleftrightarrow S\ s$
using *hull-same*[*of S s*] *hull-in*[*of S s*] **by** *metis*

lemma *hull-hull* [*simp*]: $S\ \text{hull}\ (S\ \text{hull}\ s) = S\ \text{hull}\ s$
unfolding *hull-def* **by** *blast*

lemma *hull-subset*[*intro*]: $s \subseteq (S\ \text{hull}\ s)$
unfolding *hull-def* **by** *blast*

lemma *hull-mono*: $s \subseteq t \Longrightarrow (S\ \text{hull}\ s) \subseteq (S\ \text{hull}\ t)$
unfolding *hull-def* **by** *blast*

lemma *hull-antimono*: $\forall x. S\ x \longrightarrow T\ x \Longrightarrow (T\ \text{hull}\ s) \subseteq (S\ \text{hull}\ s)$
unfolding *hull-def* **by** *blast*

lemma *hull-minimal*: $s \subseteq t \Longrightarrow S\ t \Longrightarrow (S\ \text{hull}\ s) \subseteq t$
unfolding *hull-def* **by** *blast*

lemma *subset-hull*: $S\ t \Longrightarrow S\ \text{hull}\ s \subseteq t \longleftrightarrow s \subseteq t$
unfolding *hull-def* **by** *blast*

lemma *hull-UNIV* [*simp*]: $S\ \text{hull}\ \text{UNIV} = \text{UNIV}$
unfolding *hull-def* **by** *auto*

lemma *hull-unique*: $s \subseteq t \Longrightarrow S\ t \Longrightarrow (\bigwedge t'. s \subseteq t' \Longrightarrow S\ t' \Longrightarrow t \subseteq t') \Longrightarrow (S\ \text{hull}\ s = t)$
unfolding *hull-def* **by** *auto*

lemma *hull-induct*: $\llbracket a \in Q\ \text{hull}\ S; \bigwedge x. x \in S \Longrightarrow P\ x; Q\ \{x. P\ x\} \rrbracket \Longrightarrow P\ a$
using *hull-minimal*[*of S {x. P x} Q*]
by (*auto simp add: subset-eq*)

lemma *hull-inc*: $x \in S \Longrightarrow x \in P\ \text{hull}\ S$
by (*metis hull-subset subset-eq*)

lemma *hull-Un-subset*: $(S\ \text{hull}\ s) \cup (S\ \text{hull}\ t) \subseteq (S\ \text{hull}\ (s \cup t))$
unfolding *Un-subset-iff* **by** (*metis hull-mono Un-upper1 Un-upper2*)

lemma *hull-Un*:
assumes *T*: $\bigwedge T. \text{Ball}\ T\ S \Longrightarrow S\ (\bigcap T)$
shows $S\ \text{hull}\ (s \cup t) = S\ \text{hull}\ (S\ \text{hull}\ s \cup S\ \text{hull}\ t)$
apply (*rule equalityI*)
apply (*meson hull-mono hull-subset sup.mono*)

by (metis hull-Un-subset hull-hull hull-mono)

lemma hull-Un-left: $P \text{ hull } (S \cup T) = P \text{ hull } (P \text{ hull } S \cup T)$
apply (rule equalityI)
apply (simp add: Un-commute hull-mono hull-subset sup.coboundedI2)
by (metis Un-subset-iff hull-hull hull-mono hull-subset)

lemma hull-Un-right: $P \text{ hull } (S \cup T) = P \text{ hull } (S \cup P \text{ hull } T)$
by (metis hull-Un-left sup commute)

lemma hull-insert:
 $P \text{ hull } (\text{insert } a \ S) = P \text{ hull } (\text{insert } a \ (P \text{ hull } S))$
by (metis hull-Un-right insert-is-Un)

lemma hull-redundant-eq: $a \in (S \text{ hull } s) \iff S \text{ hull } (\text{insert } a \ s) = S \text{ hull } s$
unfolding hull-def **by** blast

lemma hull-redundant: $a \in (S \text{ hull } s) \implies S \text{ hull } (\text{insert } a \ s) = S \text{ hull } s$
by (metis hull-redundant-eq)

end

102 Modules

Bases of a linear algebra based on modules (i.e. vector spaces of rings).

theory Modules
imports Hull
begin

102.1 Locale for additive functions

locale additive =
fixes $f :: 'a::\text{ab-group-add} \Rightarrow 'b::\text{ab-group-add}$
assumes add: $f (x + y) = f x + f y$
begin

lemma zero: $f 0 = 0$
proof –
have $f 0 = f (0 + 0)$ **by** simp
also have $\dots = f 0 + f 0$ **by** (rule add)
finally show $f 0 = 0$ **by** simp
qed

lemma minus: $f (-x) = -f x$
proof –
have $f (-x) + f x = f (-x + x)$ **by** (rule add [symmetric])
also have $\dots = -f x + f x$ **by** (simp add: zero)
finally show $f (-x) = -f x$ **by** (rule add-right-imp-eq)

qed

lemma *diff*: $f (x - y) = f x - f y$
using *add [of x - y]* **by** (*simp add: minus*)

lemma *sum*: $f (\text{sum } g A) = (\sum x \in A. f (g x))$
by (*induct A rule: infinite-finite-induct*) (*simp-all add: zero add*)

end

Modules form the central spaces in linear algebra. They are a generalization from vector spaces by replacing the scalar field by a scalar ring.

locale *module* =

fixes *scale* :: 'a::comm-ring-1 \Rightarrow 'b::ab-group-add \Rightarrow 'b (**infixr** *s 75)

assumes *scale-right-distrib* [*algebra-simps, algebra-split-simps*]:

$a *s (x + y) = a *s x + a *s y$

and *scale-left-distrib* [*algebra-simps, algebra-split-simps*]:

$(a + b) *s x = a *s x + b *s x$

and *scale-scale [simp]*: $a *s (b *s x) = (a * b) *s x$

and *scale-one [simp]*: $1 *s x = x$

begin

lemma *scale-left-commute*: $a *s (b *s x) = b *s (a *s x)$
by (*simp add: mult.commute*)

lemma *scale-zero-left [simp]*: $0 *s x = 0$
and *scale-minus-left [simp]*: $(- a) *s x = - (a *s x)$
and *scale-left-diff-distrib* [*algebra-simps, algebra-split-simps*]:
 $(a - b) *s x = a *s x - b *s x$
and *scale-sum-left*: $(\text{sum } f A) *s x = (\sum a \in A. (f a) *s x)$

proof –

interpret *s*: *additive* $\lambda a. a *s x$

by *standard* (*rule scale-left-distrib*)

show $0 *s x = 0$ **by** (*rule s.zero*)

show $(- a) *s x = - (a *s x)$ **by** (*rule s.minus*)

show $(a - b) *s x = a *s x - b *s x$ **by** (*rule s.diff*)

show $(\text{sum } f A) *s x = (\sum a \in A. (f a) *s x)$ **by** (*rule s.sum*)

qed

lemma *scale-zero-right [simp]*: $a *s 0 = 0$
and *scale-minus-right [simp]*: $a *s (- x) = - (a *s x)$
and *scale-right-diff-distrib* [*algebra-simps, algebra-split-simps*]:
 $a *s (x - y) = a *s x - a *s y$
and *scale-sum-right*: $a *s (\text{sum } f A) = (\sum x \in A. a *s (f x))$

proof –

interpret *s*: *additive* $\lambda x. a *s x$

by *standard* (*rule scale-right-distrib*)

show $a *s 0 = 0$ **by** (*rule s.zero*)

show $a *s (- x) = - (a *s x)$ **by** (*rule s.minus*)

show $a *s (x - y) = a *s x - a *s y$ **by** (*rule s.diff*)
show $a *s (\text{sum } f A) = (\sum x \in A. a *s (f x))$ **by** (*rule s.sum*)
qed

lemma *sum-constant-scale*: $(\sum x \in A. y) = \text{scale } (\text{of-nat } (\text{card } A)) y$
by (*induct A rule: infinite-finite-induct*) (*simp-all add: algebra-simps*)

end

setup $\langle \text{Sign.add-const-constraint } (\text{const-name } \langle \text{divide} \rangle, \text{SOME } \text{typ } \langle 'a \Rightarrow 'a \Rightarrow 'a \rangle) \rangle$

context *module*
begin

lemma [*field-simps, field-split-simps*]:

shows *scale-left-distrib-NO-MATCH*: $\text{NO-MATCH } (x \text{ div } y) c \Longrightarrow (a + b) *s x = a *s x + b *s x$

and *scale-right-distrib-NO-MATCH*: $\text{NO-MATCH } (x \text{ div } y) a \Longrightarrow a *s (x + y) = a *s x + a *s y$

and *scale-left-diff-distrib-NO-MATCH*: $\text{NO-MATCH } (x \text{ div } y) c \Longrightarrow (a - b) *s x = a *s x - b *s x$

and *scale-right-diff-distrib-NO-MATCH*: $\text{NO-MATCH } (x \text{ div } y) a \Longrightarrow a *s (x - y) = a *s x - a *s y$

by (*rule scale-left-distrib scale-right-distrib scale-left-diff-distrib scale-right-diff-distrib*)**+**

end

setup $\langle \text{Sign.add-const-constraint } (\text{const-name } \langle \text{divide} \rangle, \text{SOME } \text{typ } \langle 'a::\text{divide} \Rightarrow 'a \Rightarrow 'a \rangle) \rangle$

103 Subspace

context *module*
begin

definition *subspace* :: $'b \text{ set} \Rightarrow \text{bool}$

where *subspace* $S \longleftrightarrow 0 \in S \wedge (\forall x \in S. \forall y \in S. x + y \in S) \wedge (\forall c. \forall x \in S. c *s x \in S)$

lemma *subspaceI*:

$0 \in S \Longrightarrow (\bigwedge x y. x \in S \Longrightarrow y \in S \Longrightarrow x + y \in S) \Longrightarrow (\bigwedge c x. x \in S \Longrightarrow c *s x \in S) \Longrightarrow \text{subspace } S$

by (*auto simp: subspace-def*)

lemma *subspace-UNIV*[*simp*]: *subspace UNIV*

by (*simp add: subspace-def*)

lemma *subspace-single-0*[*simp*]: *subspace* $\{0\}$

by (*simp add: subspace-def*)

lemma *subspace-0*: $\text{subspace } S \implies 0 \in S$
by (*metis subspace-def*)

lemma *subspace-add*: $\text{subspace } S \implies x \in S \implies y \in S \implies x + y \in S$
by (*metis subspace-def*)

lemma *subspace-scale*: $\text{subspace } S \implies x \in S \implies c * s x \in S$
by (*metis subspace-def*)

lemma *subspace-neg*: $\text{subspace } S \implies x \in S \implies -x \in S$
by (*metis scale-minus-left scale-one subspace-scale*)

lemma *subspace-diff*: $\text{subspace } S \implies x \in S \implies y \in S \implies x - y \in S$
by (*metis diff-conv-add-uminus subspace-add subspace-neg*)

lemma *subspace-sum*: $\text{subspace } A \implies (\bigwedge x. x \in B \implies f x \in A) \implies \text{sum } f B \in A$
by (*induct B rule: infinite-finite-induct*) (*auto simp add: subspace-add subspace-0*)

lemma *subspace-Int*: $(\bigwedge i. i \in I \implies \text{subspace } (s i)) \implies \text{subspace } (\bigcap_{i \in I} s i)$
by (*auto simp: subspace-def*)

lemma *subspace-Inter*: $\forall s \in f. \text{subspace } s \implies \text{subspace } (\bigcap f)$
unfolding *subspace-def* by *auto*

lemma *subspace-inter*: $\text{subspace } A \implies \text{subspace } B \implies \text{subspace } (A \cap B)$
by (*simp add: subspace-def*)

104 Span: subspace generated by a set

definition *span* :: 'b set \Rightarrow 'b set

where *span-explicit*: $\text{span } b = \{(\sum a \in t. r a * s a) \mid t r. \text{finite } t \wedge t \subseteq b\}$

lemma *span-explicit'*:

$\text{span } b = \{(\sum v \mid f v \neq 0. f v * s v) \mid f. \text{finite } \{v. f v \neq 0\} \wedge (\forall v. f v \neq 0 \longrightarrow v \in b)\}$

unfolding *span-explicit*

proof *safe*

fix *t r* assume *finite t t* $\subseteq b$

then show $\exists f. (\sum a \in t. r a * s a) = (\sum v \mid f v \neq 0. f v * s v) \wedge \text{finite } \{v. f v \neq 0\} \wedge (\forall v. f v \neq 0 \longrightarrow v \in b)$

by (*intro exI[of - $\lambda v. \text{if } v \in t \text{ then } r v \text{ else } 0$]*) (*auto intro!: sum.mono-neutral-cong-right*)

next

fix *f* :: 'b \Rightarrow 'a assume *finite* $\{v. f v \neq 0\}$ $(\forall v. f v \neq 0 \longrightarrow v \in b)$

then show $\exists t r. (\sum v \mid f v \neq 0. f v * s v) = (\sum a \in t. r a * s a) \wedge \text{finite } t \wedge t \subseteq b$

by (*intro exI[of - $\{v. f v \neq 0\}$]* *exI[of - f]*) *auto*

qed

lemma *span-alt*:

span $B = \{(\sum x \mid f x \neq 0. f x *s x) \mid f. \{x. f x \neq 0\} \subseteq B \wedge \text{finite } \{x. f x \neq 0\}\}$

unfolding *span-explicit'* **by** *auto*

lemma *span-finite*:

assumes *fS*: *finite S*

shows *span S = range* $(\lambda u. \sum v \in S. u v *s v)$

unfolding *span-explicit*

proof *safe*

fix *t r* **assume** $t \subseteq S$ **then show** $(\sum a \in t. r a *s a) \in \text{range } (\lambda u. \sum v \in S. u v *s v)$

by (*intro image-eqI*[*of - -* $\lambda a. \text{if } a \in t \text{ then } r a \text{ else } 0$])

(*auto simp: if-distrib*[*of* $\lambda r. r *s a$ **for** *a*] *sum.If-cases fS Int-absorb1*)

next

show $\exists t r. (\sum v \in S. u v *s v) = (\sum a \in t. r a *s a) \wedge \text{finite } t \wedge t \subseteq S$ **for** *u*

by (*intro exI*[*of - u*] *exI*[*of - S*]) (*auto intro: fS*)

qed

lemma *span-induct-alt* [*consumes 1, case-names base step, induct set: span*]:

assumes *x*: $x \in \text{span } S$

assumes *h0*: $h 0$ **and** *hS*: $\bigwedge c x y. x \in S \implies h y \implies h (c *s x + y)$

shows $h x$

using *x* **unfolding** *span-explicit*

proof *safe*

fix *t r* **assume** *finite t* $t \subseteq S$ **then show** $h (\sum a \in t. r a *s a)$

by (*induction t*) (*auto intro: hS h0*)

qed

lemma *span-mono*: $A \subseteq B \implies \text{span } A \subseteq \text{span } B$

by (*auto simp: span-explicit*)

lemma *span-base*: $a \in S \implies a \in \text{span } S$

by (*auto simp: span-explicit intro!: exI*[*of - {a}*] *exI*[*of -* $\lambda-. 1$])

lemma *span-superset*: $S \subseteq \text{span } S$

by (*auto simp: span-base*)

lemma *span-zero*: $0 \in \text{span } S$

by (*auto simp: span-explicit intro!: exI*[*of - {}*])

lemma *span-UNIV*[*simp*]: $\text{span } UNIV = UNIV$

by (*auto intro: span-base*)

lemma *span-add*: $x \in \text{span } S \implies y \in \text{span } S \implies x + y \in \text{span } S$

unfolding *span-explicit*

proof *safe*

fix *tx ty rx ry* **assume** $*$: *finite tx* *finite ty* $tx \subseteq S$ $ty \subseteq S$

have [*simp*]: $(tx \cup ty) \cap tx = tx$ $(tx \cup ty) \cap ty = ty$

by *auto*

show $\exists t r. (\sum a \in tx. rx\ a * s\ a) + (\sum a \in ty. ry\ a * s\ a) = (\sum a \in t. r\ a * s\ a) \wedge$
finite $t \wedge t \subseteq S$
apply (*intro* *exI*[*of* - $tx \cup ty$])
apply (*intro* *exI*[*of* - $\lambda a. (if\ a \in tx\ then\ rx\ a\ else\ 0) + (if\ a \in ty\ then\ ry\ a\ else\ 0)$])
apply (*auto simp*: ** scale-left-distrib sum.distrib if-distrib*[*of* $\lambda r. r * s\ a$ **for** a]
sum.If-cases)
done
qed

lemma *span-scale*: $x \in span\ S \implies c * s\ x \in span\ S$

unfolding *span-explicit*

proof *safe*

fix $t\ r$ **assume** $*$: *finite* $t\ t \subseteq S$

show $\exists t' r'. c * s\ (\sum a \in t. r\ a * s\ a) = (\sum a \in t'. r'\ a * s\ a) \wedge$ *finite* $t' \wedge t' \subseteq S$

by (*intro* *exI*[*of* - t] *exI*[*of* - $\lambda a. c * r\ a$]) (*auto simp*: ** scale-sum-right*)

qed

lemma *subspace-span* [*iff*]: *subspace* (*span* S)

by (*auto simp*: *subspace-def span-zero span-add span-scale*)

lemma *span-neg*: $x \in span\ S \implies -\ x \in span\ S$

by (*metis* *subspace-neg subspace-span*)

lemma *span-diff*: $x \in span\ S \implies y \in span\ S \implies x - y \in span\ S$

by (*metis* *subspace-span subspace-diff*)

lemma *span-sum*: $(\bigwedge x. x \in A \implies f\ x \in span\ S) \implies sum\ f\ A \in span\ S$

by (*rule* *subspace-sum*, *rule* *subspace-span*)

lemma *span-minimal*: $S \subseteq T \implies$ *subspace* $T \implies span\ S \subseteq T$

by (*auto simp*: *span-explicit intro!*: *subspace-sum subspace-scale*)

lemma *span-def*: $span\ S =$ *subspace* *hull* S

by (*intro* *hull-unique*[*symmetric*] *span-superset subspace-span span-minimal*)

lemma *span-unique*:

$S \subseteq T \implies$ *subspace* $T \implies (\bigwedge T'. S \subseteq T' \implies$ *subspace* $T' \implies T \subseteq T') \implies$
 $span\ S = T$

unfolding *span-def* **by** (*rule* *hull-unique*)

lemma *span-subspace-induct*[*consumes* 2]:

assumes x : $x \in span\ S$

and P : *subspace* P

and SP : $\bigwedge x. x \in S \implies x \in P$

shows $x \in P$

proof –

from SP **have** SP' : $S \subseteq P$

by (*simp add*: *subset-eq*)

from x *hull-minimal*[**where** $S = \text{subspace}$, *OF SP' P*, *unfolded span-def[symmetric]*]
show $x \in P$
by (*metis subset-eq*)
qed

lemma (**in** *module*) *span-induct*[*consumes 1*, *case-names base step*, *induct set: span*]:
assumes $x: x \in \text{span } S$
and $P: \text{subspace } (\text{Collect } P)$
and $SP: \bigwedge x. x \in S \implies P x$
shows $P x$
using $P SP \text{span-subspace-induct } x$ **by** *fastforce*

lemma *span-empty[simp]*: $\text{span } \{\} = \{0\}$
by (*rule span-unique*) (*auto simp add: subspace-def*)

lemma *span-subspace*: $A \subseteq B \implies B \subseteq \text{span } A \implies \text{subspace } B \implies \text{span } A = B$
by (*metis order-antisym span-def hull-minimal*)

lemma *span-span*: $\text{span } (\text{span } A) = \text{span } A$
unfolding *span-def hull-hull ..*

lemma *span-add-eq*: **assumes** $x: x \in \text{span } S$ **shows** $x + y \in \text{span } S \longleftrightarrow y \in \text{span } S$
proof
assume $*$: $x + y \in \text{span } S$
have $(x + y) - x \in \text{span } S$ **using** $*$ **by** (*rule span-diff*)
then show $y \in \text{span } S$ **by** *simp*
qed (*intro span-add x*)

lemma *span-add-eq2*: **assumes** $y: y \in \text{span } S$ **shows** $x + y \in \text{span } S \longleftrightarrow x \in \text{span } S$
using *span-add-eq*[*of y S x*] y **by** (*auto simp: ac-simps*)

lemma *span-singleton*: $\text{span } \{x\} = \text{range } (\lambda k. k * x)$
by (*auto simp: span-finite*)

lemma *span-Un*: $\text{span } (S \cup T) = \{x + y \mid x y. x \in \text{span } S \wedge y \in \text{span } T\}$
proof *safe*
fix x **assume** $x \in \text{span } (S \cup T)$
then obtain $t r$ **where** $t: \text{finite } t \subseteq S \cup T$ **and** $x: x = (\sum a \in t. r a * a)$
by (*auto simp: span-explicit*)
moreover have $t \cap S \cup (t - S) = t$ **by** *auto*
ultimately show $\exists xa y. x = xa + y \wedge xa \in \text{span } S \wedge y \in \text{span } T$
unfolding x
apply (*rule-tac exI*[*of -* $\sum a \in t \cap S. r a * a$])
apply (*rule-tac exI*[*of -* $\sum a \in t - S. r a * a$])
apply (*subst sum.union-inter-neutral[symmetric]*)

```

    apply (auto intro!: span-sum span-scale intro: span-base)
  done
next
fix x y assume x ∈ span S y ∈ span T then show x + y ∈ span (S ∪ T)
  using span-mono[of S S ∪ T] span-mono[of T S ∪ T]
  by (auto intro!: span-add)
qed

```

```

lemma span-insert: span (insert a S) = {x. ∃ k. (x - k * s a) ∈ span S}
proof -
  have span ({a} ∪ S) = {x. ∃ k. (x - k * s a) ∈ span S}
  unfolding span-Un span-singleton
  apply (auto simp add: set-eq-iff)
  subgoal for y k by (auto intro!: exI[of - k])
  subgoal for y k by (rule exI[of - k * s a], rule exI[of - y - k * s a]) auto
  done
  then show ?thesis by simp
qed

```

```

lemma span-breakdown:
  assumes bS: b ∈ S
  and aS: a ∈ span S
  shows ∃ k. a - k * s b ∈ span (S - {b})
  using assms span-insert [of b S - {b}]
  by (simp add: insert-absorb)

```

```

lemma span-breakdown-eq: x ∈ span (insert a S) ⟷ (∃ k. x - k * s a ∈ span S)
  by (simp add: span-insert)

```

```

lemmas span-clauses = span-base span-zero span-add span-scale

```

```

lemma span-eq-iff[simp]: span s = s ⟷ subspace s
  unfolding span-def by (rule hull-eq) (rule subspace-Inter)

```

```

lemma span-eq: span S = span T ⟷ S ⊆ span T ∧ T ⊆ span S
  by (metis span-minimal span-subspace span-superset subspace-span)

```

```

lemma eq-span-insert-eq:
  assumes (x - y) ∈ span S
  shows span(insert x S) = span(insert y S)
proof -
  have *: span(insert x S) ⊆ span(insert y S) if (x - y) ∈ span S for x y
  proof -
    have 1: (r * s x - r * s y) ∈ span S for r
      by (metis scale-right-diff-distrib span-scale that)
    have 2: (z - k * s y) - k * s (x - y) = z - k * s x for z k
      by (simp add: scale-right-diff-distrib)
    show ?thesis
  apply (clarsimp simp add: span-breakdown-eq)

```

```

  by (metis 1 2 diff-add-cancel scale-right-diff-distrib span-add-eq)
qed
show ?thesis
  apply (intro subset-antisym * assms)
  using assms subspace-neg subspace-span minus-diff-eq by force
qed

```

105 Dependent and independent sets

definition *dependent* :: 'b set \Rightarrow bool

where *dependent-explicit*: $\text{dependent } s \iff (\exists t u. \text{finite } t \wedge t \subseteq s \wedge (\sum_{v \in t}. u \ v * s \ v) = 0 \wedge (\exists v \in t. u \ v \neq 0))$

abbreviation *independent* $s \equiv \neg \text{dependent } s$

lemma *dependent-mono*: $\text{dependent } B \implies B \subseteq A \implies \text{dependent } A$

by (*auto simp add: dependent-explicit*)

lemma *independent-mono*: $\text{independent } A \implies B \subseteq A \implies \text{independent } B$

by (*auto intro: dependent-mono*)

lemma *dependent-zero*: $0 \in A \implies \text{dependent } A$

by (*auto simp: dependent-explicit intro!: exI[of - $\lambda i. 1$] exI[of - $\{0\}$]*)

lemma *independent-empty*[*intro*]: *independent* $\{\}$

by (*simp add: dependent-explicit*)

lemma *independent-explicit-module*:

$\text{independent } s \iff (\forall t u v. \text{finite } t \longrightarrow t \subseteq s \longrightarrow (\sum_{v \in t}. u \ v * s \ v) = 0 \longrightarrow v \in t \longrightarrow u \ v = 0)$

unfolding *dependent-explicit* **by** *auto*

lemma *independentD*: $\text{independent } s \implies \text{finite } t \implies t \subseteq s \implies (\sum_{v \in t}. u \ v * s \ v) = 0 \implies v \in t \implies u \ v = 0$

by (*simp add: independent-explicit-module*)

lemma *independent-Union-directed*:

assumes *directed*: $\bigwedge c d. c \in C \implies d \in C \implies c \subseteq d \vee d \subseteq c$

assumes *indep*: $\bigwedge c. c \in C \implies \text{independent } c$

shows *independent* $(\bigcup C)$

proof

assume *dependent* $(\bigcup C)$

then obtain $u \ v \ S$ **where** $S: \text{finite } S \ S \subseteq \bigcup C \ v \in S \ u \ v \neq 0 \ (\sum_{v \in S}. u \ v * s \ v) = 0$

by (*auto simp: dependent-explicit*)

have $S \neq \{\}$

using $\langle v \in S \rangle$ **by** *auto*

have $\exists c \in C. S \subseteq c$

```

  using ⟨finite S⟩ ⟨S ≠ {}⟩ ⟨S ⊆ ⋃ C⟩
proof (induction rule: finite-ne-induct)
  case (insert i I)
  then obtain c d where cd: c ∈ C d ∈ C and iI: I ⊆ c i ∈ d
  by blast
  from directed[OF cd] cd have c ∪ d ∈ C
  by (auto simp: sup.absorb1 sup.absorb2)
  with iI show ?case
  by (intro bexI[of - c ∪ d]) auto
qed auto
then obtain c where c ∈ C S ⊆ c
  by auto
have dependent c
  unfolding dependent-explicit
  by (intro exI[of - S] exI[of - u] bexI[of - v] conjI) fact+
with indep[OF ⟨c ∈ C⟩] show False
  by auto
qed

```

lemma *dependent-finite:*

```

  assumes finite S
  shows dependent S ⟷ (∃ u. (∃ v ∈ S. u v ≠ 0) ∧ (∑ v ∈ S. u v *s v) = 0)
  (is ?lhs = ?rhs)
proof
  assume ?lhs
  then obtain T u v
  where finite T T ⊆ S v ∈ T u v ≠ 0 (∑ v ∈ T. u v *s v) = 0
  by (force simp: dependent-explicit)
  with assms show ?rhs
  apply (rule-tac x=λv. if v ∈ T then u v else 0 in exI)
  apply (auto simp: sum.mono-neutral-right)
  done
next
  assume ?rhs with assms show ?lhs
  by (fastforce simp add: dependent-explicit)
qed

```

lemma *dependent-alt:*

```

  dependent B ⟷
  (∃ X. finite {x. X x ≠ 0} ∧ {x. X x ≠ 0} ⊆ B ∧ (∑ x | X x ≠ 0. X x *s x) =
  0 ∧ (∃ x. X x ≠ 0))
  unfolding dependent-explicit
  apply safe
  subgoal for S u v
  apply (intro exI[of - λx. if x ∈ S then u x else 0])
  apply (subst sum.mono-neutral-cong-left[where T=S])
  apply (auto intro!: sum.mono-neutral-cong-right cong: rev-conj-cong)
  done
  apply auto

```

done

lemma *independent-alt:*

independent $B \longleftrightarrow$
 $(\forall X. \text{finite } \{x. X x \neq 0\} \longrightarrow \{x. X x \neq 0\} \subseteq B \longrightarrow (\sum x | X x \neq 0. X x * s x) = 0 \longrightarrow (\forall x. X x = 0))$
unfolding *dependent-alt by auto*

lemma *independentD-alt:*

independent $B \Longrightarrow \text{finite } \{x. X x \neq 0\} \Longrightarrow \{x. X x \neq 0\} \subseteq B \Longrightarrow (\sum x | X x \neq 0. X x * s x) = 0 \Longrightarrow X x = 0$
unfolding *independent-alt by blast*

lemma *independentD-unique:*

assumes B : *independent* B
and X : *finite* $\{x. X x \neq 0\} \{x. X x \neq 0\} \subseteq B$
and Y : *finite* $\{x. Y x \neq 0\} \{x. Y x \neq 0\} \subseteq B$
and $(\sum x | X x \neq 0. X x * s x) = (\sum x | Y x \neq 0. Y x * s x)$
shows $X = Y$

proof –

have $X x - Y x = 0$ **for** x
using B
proof (*rule independentD-alt*)
have $\{x. X x - Y x \neq 0\} \subseteq \{x. X x \neq 0\} \cup \{x. Y x \neq 0\}$
by *auto*
then show *finite* $\{x. X x - Y x \neq 0\} \{x. X x - Y x \neq 0\} \subseteq B$
using $X Y$ **by** (*auto dest: finite-subset*)
then have $(\sum x | X x - Y x \neq 0. (X x - Y x) * s x) = (\sum v \in \{S. X S \neq 0\} \cup \{S. Y S \neq 0\}. (X v - Y v) * s v)$
using $X Y$ **by** (*intro sum.mono-neutral-cong-left*) *auto*
also have $\dots = (\sum v \in \{S. X S \neq 0\} \cup \{S. Y S \neq 0\}. X v * s v) - (\sum v \in \{S. X S \neq 0\} \cup \{S. Y S \neq 0\}. Y v * s v)$
by (*simp add: scale-left-diff-distrib sum-subtractf assms*)
also have $(\sum v \in \{S. X S \neq 0\} \cup \{S. Y S \neq 0\}. X v * s v) = (\sum v \in \{S. X S \neq 0\}. X v * s v)$
using $X Y$ **by** (*intro sum.mono-neutral-cong-right*) *auto*
also have $(\sum v \in \{S. X S \neq 0\} \cup \{S. Y S \neq 0\}. Y v * s v) = (\sum v \in \{S. Y S \neq 0\}. Y v * s v)$
using $X Y$ **by** (*intro sum.mono-neutral-cong-right*) *auto*
finally show $(\sum x | X x - Y x \neq 0. (X x - Y x) * s x) = 0$
using *assms* **by** *simp*
qed
then show *?thesis*
by *auto*
qed

106 Representation of a vector on a specific basis

definition *representation* :: 'b set \Rightarrow 'b \Rightarrow 'b \Rightarrow 'a

where *representation basis* $v =$
 (if independent basis $\wedge v \in \text{span basis}$ then
 SOME $f. (\forall v. f v \neq 0 \longrightarrow v \in \text{basis}) \wedge \text{finite } \{v. f v \neq 0\} \wedge (\sum v \in \{v. f v \neq 0\}. f v * s v) = v$
 else $(\lambda b. 0)$)

lemma *unique-representation:*

assumes *basis: independent basis*
and *in-basis:* $\bigwedge v. f v \neq 0 \implies v \in \text{basis} \wedge v. g v \neq 0 \implies v \in \text{basis}$
and [*simp*]: $\text{finite } \{v. f v \neq 0\} \text{ finite } \{v. g v \neq 0\}$
and *eq:* $(\sum v \in \{v. f v \neq 0\}. f v * s v) = (\sum v \in \{v. g v \neq 0\}. g v * s v)$
shows $f = g$
proof (*rule ext, rule ccontr*)
fix v **assume** *ne:* $f v \neq g v$
have *dependent basis*
unfolding *dependent-explicit*
proof (*intro exI conjI*)
have $*$: $\{v. f v - g v \neq 0\} \subseteq \{v. f v \neq 0\} \cup \{v. g v \neq 0\}$
by *auto*
show $\text{finite } \{v. f v - g v \neq 0\}$
by (*rule finite-subset[OF *]*) *simp*
show $\exists v \in \{v. f v - g v \neq 0\}. f v - g v \neq 0$
by (*rule exI[of - v]*) (*auto simp: ne*)
have $(\sum v \mid f v - g v \neq 0. (f v - g v) * s v) =$
 $(\sum v \in \{v. f v \neq 0\} \cup \{v. g v \neq 0\}. (f v - g v) * s v)$
by (*intro sum.mono-neutral-cong-left **) *auto*
also have $\dots =$
 $(\sum v \in \{v. f v \neq 0\} \cup \{v. g v \neq 0\}. f v * s v) - (\sum v \in \{v. f v \neq 0\} \cup \{v. g v \neq 0\}. g v * s v)$
by (*simp add: algebra-simps sum-subtractf*)
also have $\dots = (\sum v \mid f v \neq 0. f v * s v) - (\sum v \mid g v \neq 0. g v * s v)$
by (*intro arg-cong2[where f = (-)] sum.mono-neutral-cong-right*) *auto*
finally show $(\sum v \mid f v - g v \neq 0. (f v - g v) * s v) = 0$
by (*simp add: eq*)
show $\{v. f v - g v \neq 0\} \subseteq \text{basis}$
using *in-basis ** **by** *auto*
qed
with *basis* **show** *False* **by** *auto*
qed

lemma

shows *representation-ne-zero:* $\bigwedge b. \text{representation basis } v b \neq 0 \implies b \in \text{basis}$
and *finite-representation:* $\text{finite } \{b. \text{representation basis } v b \neq 0\}$
and *sum-nonzero-representation-eq:*
 $\text{independent basis} \implies v \in \text{span basis} \implies (\sum b \mid \text{representation basis } v b \neq 0. \text{representation basis } v b * s b) = v$
proof –
 { **assume** *basis: independent basis* **and** $v: v \in \text{span basis}$
define p **where** $p f \longleftrightarrow$

$(\forall v. f v \neq 0 \longrightarrow v \in \text{basis}) \wedge \text{finite } \{v. f v \neq 0\} \wedge (\sum v \in \{v. f v \neq 0\}. f v * s v) = v$ **for** f
obtain $t r$ **where** $*$: $\text{finite } t \subseteq \text{basis } (\sum b \in t. r b * s b) = v$
using $\langle v \in \text{span basis} \rangle$ **by** $(\text{auto simp: span-explicit})$
define f **where** $f b = (\text{if } b \in t \text{ then } r b \text{ else } 0)$ **for** b
have $p f$
using $*$ **by** $(\text{auto simp: p-def f-def intro!: sum.mono-neutral-cong-left})$
have $*$: $\text{representation basis } v = \text{Eps } p$ **by** $(\text{simp add: p-def[abs-def] representation-def basis } v)$
from $\text{someI[of } p f, OF \langle p f \rangle]$ **have** p $(\text{representation basis } v)$
unfolding $* . \}$
note $* = \text{this}$

show $\text{representation basis } v b \neq 0 \implies b \in \text{basis for } b$
using $*$ **by** $(\text{cases independent basis } \wedge v \in \text{span basis}) (\text{auto simp: representation-def})$

show $\text{finite } \{b. \text{representation basis } v b \neq 0\}$
using $*$ **by** $(\text{cases independent basis } \wedge v \in \text{span basis}) (\text{auto simp: representation-def})$

show $\text{independent basis} \implies v \in \text{span basis} \implies (\sum b \mid \text{representation basis } v b \neq 0. \text{representation basis } v b * s b) = v$
using $*$ **by** auto

qed

lemma *sum-representation-eq*:

$(\sum b \in B. \text{representation basis } v b * s b) = v$
if $\text{independent basis } v \in \text{span basis finite } B \text{ basis } \subseteq B$
proof –
have $(\sum b \in B. \text{representation basis } v b * s b) =$
 $(\sum b \mid \text{representation basis } v b \neq 0. \text{representation basis } v b * s b)$
apply $(\text{rule sum.mono-neutral-cong})$
apply $(\text{rule finite-representation})$
apply fact
subgoal for b
using $\text{that representation-ne-zero[of basis } v b]$
by auto
subgoal by auto
subgoal by simp
done
also have $\dots = v$
by $(\text{rule sum-nonzero-representation-eq; fact})$
finally show $?thesis .$

qed

lemma *representation-eqI*:

assumes $\text{basis: independent basis}$ **and** $b: v \in \text{span basis}$
and $\text{ne-zero: } \bigwedge b. f b \neq 0 \implies b \in \text{basis}$

and *finite*: $\text{finite } \{b. f b \neq 0\}$
and *eq*: $(\sum b \mid f b \neq 0. f b *s b) = v$
shows *representation basis* $v = f$
by (*rule unique-representation*[*OF basis*])
(auto simp: representation-ne-zero finite-representation
sum-nonzero-representation-eq[*OF basis b*] *ne-zero finite eq*)

lemma *representation-basis*:

assumes *basis: independent basis* **and** $b: b \in \text{basis}$
shows *representation basis* $b = (\lambda v. \text{if } v = b \text{ then } 1 \text{ else } 0)$
proof (*rule unique-representation*[*OF basis*])
show *representation basis* $b v \neq 0 \implies v \in \text{basis}$ **for** v
using *representation-ne-zero* .
show *finite* $\{v. \text{representation basis } b v \neq 0\}$
using *finite-representation* .
show $(\text{if } v = b \text{ then } 1 \text{ else } 0) \neq 0 \implies v \in \text{basis}$ **for** v
by (*cases* $v = b$) (*auto simp: b*)
have $*$: $\{v. (\text{if } v = b \text{ then } 1 \text{ else } 0 :: 'a) \neq 0\} = \{b\}$
by *auto*
show *finite* $\{v. (\text{if } v = b \text{ then } 1 \text{ else } 0) \neq 0\}$ **unfolding** $*$ **by** *auto*
show $(\sum v \mid \text{representation basis } b v \neq 0. \text{representation basis } b v *s v) =$
 $(\sum v \mid (\text{if } v = b \text{ then } 1 \text{ else } 0 :: 'a) \neq 0. (\text{if } v = b \text{ then } 1 \text{ else } 0) *s v)$
unfolding $*$ *sum-nonzero-representation-eq*[*OF basis span-base*[*OF b*]] **by** *auto*
qed

lemma *representation-zero*: *representation basis* $0 = (\lambda b. 0)$

proof *cases*

assume *basis: independent basis* **show** *?thesis*
by (*rule representation-eqI*[*OF basis span-zero*]) *auto*
qed (*simp add: representation-def*)

lemma *representation-diff*:

assumes *basis: independent basis* **and** $v: v \in \text{span basis}$ **and** $u: u \in \text{span basis}$
shows *representation basis* $(u - v) = (\lambda b. \text{representation basis } u b - \text{representation basis } v b)$
proof (*rule representation-eqI*[*OF basis span-diff*[*OF u v*]])
let $?R = \text{representation basis}$
note *finite-representation*[*simp*] u [*simp*] v [*simp*]
have $*$: $\{b. ?R u b - ?R v b \neq 0\} \subseteq \{b. ?R u b \neq 0\} \cup \{b. ?R v b \neq 0\}$
by *auto*
then show $?R u b - ?R v b \neq 0 \implies b \in \text{basis}$ **for** b
by (*auto dest: representation-ne-zero*)
show *finite* $\{b. ?R u b - ?R v b \neq 0\}$
by (*intro finite-subset*[*OF **]) *simp-all*
have $(\sum b \mid ?R u b - ?R v b \neq 0. (?R u b - ?R v b) *s b) =$
 $(\sum b \in \{b. ?R u b \neq 0\} \cup \{b. ?R v b \neq 0\}. (?R u b - ?R v b) *s b)$
by (*intro sum.mono-neutral-cong-left* $*$) *auto*
also have $\dots =$
 $(\sum b \in \{b. ?R u b \neq 0\} \cup \{b. ?R v b \neq 0\}. ?R u b *s b) - (\sum b \in \{b. ?R v b \neq 0\}. ?R v b *s b)$

$\neq 0\} \cup \{b. ?R v b \neq 0\}. ?R v b *s b)$
 by (simp add: algebra-simps sum-subtractf)
 also have ... = $(\sum b \mid ?R u b \neq 0. ?R u b *s b) - (\sum b \mid ?R v b \neq 0. ?R v b *s b)$
 by (intro arg-cong2[where f= (-)] sum.mono-neutral-cong-right) auto
 finally show $(\sum b \mid ?R u b - ?R v b \neq 0. (?R u b - ?R v b) *s b) = u - v$
 by (simp add: sum-nonzero-representation-eq[OF basis])
 qed

lemma representation-neg:
independent basis $\implies v \in \text{span basis} \implies \text{representation basis } (- v) = (\lambda b. - \text{representation basis } v b)$
 using representation-diff[of basis v 0] by (simp add: representation-zero span-zero)

lemma representation-add:
independent basis $\implies v \in \text{span basis} \implies u \in \text{span basis} \implies \text{representation basis } (u + v) = (\lambda b. \text{representation basis } u b + \text{representation basis } v b)$
 using representation-diff[of basis -v u] by (simp add: representation-neg representation-diff span-neg)

lemma representation-sum:
independent basis $\implies (\bigwedge i. i \in I \implies v i \in \text{span basis}) \implies \text{representation basis } (\text{sum } v I) = (\lambda b. \sum i \in I. \text{representation basis } (v i) b)$
 by (induction I rule: infinite-finite-induct)
 (auto simp: representation-zero representation-add span-sum)

lemma representation-scale:
assumes basis: *independent basis and $v: v \in \text{span basis}$*
shows *representation basis $(r *s v) = (\lambda b. r * \text{representation basis } v b)$*
proof (rule representation-eqI[OF basis span-scale[OF v]])
 let $?R = \text{representation basis}$
note *finite-representation[simp] v[simp]*
have *: $\{b. r * ?R v b \neq 0\} \subseteq \{b. ?R v b \neq 0\}$
 by auto
then show $r * \text{representation basis } v b \neq 0 \implies b \in \text{basis for } b$
 using representation-ne-zero by auto
show *finite $\{b. r * ?R v b \neq 0\}$*
 by (intro finite-subset[OF *]) simp-all
have $(\sum b \mid r * ?R v b \neq 0. (r * ?R v b) *s b) = (\sum b \in \{b. ?R v b \neq 0\}. (r * ?R v b) *s b)$
 by (intro sum.mono-neutral-cong-left *) auto
also have ... = $r *s (\sum b \mid ?R v b \neq 0. ?R v b *s b)$
 by (simp add: scale-scale[symmetric] scale-sum-right del: scale-scale)
finally show $(\sum b \mid r * ?R v b \neq 0. (r * ?R v b) *s b) = r *s v$
 by (simp add: sum-nonzero-representation-eq[OF basis])
 qed

lemma representation-extend:

assumes *basis*: independent *basis* **and** *v*: $v \in \text{span } \textit{basis}'$ **and** *basis'*: $\textit{basis}' \subseteq \textit{basis}$
shows *representation basis* $v = \textit{representation basis}' v$
proof (rule *representation-eqI*[*OF basis*])
show *v'*: $v \in \text{span } \textit{basis}$ **using** *span-mono*[*OF basis'*] *v* **by** *auto*
have ***: independent *basis'* **using** *basis'* *basis* **by** (*auto intro*: *dependent-mono*)
show *representation basis'* $v \neq 0 \implies b \in \textit{basis}$ **for** *b*
using *representation-ne-zero basis'* **by** *auto*
show *finite* {*b*. *representation basis'* $v \neq 0$ }
using *finite-representation* .
show $(\sum b \mid \textit{representation basis}' v \neq 0. \textit{representation basis}' v \neq 0) = v$
using *sum-nonzero-representation-eq*[*OF * v*] .
qed

The set B is the maximal independent set for $\text{span } B$, or A is the minimal spanning set

lemma *spanning-subset-independent*:

assumes *BA*: $B \subseteq A$
and *iA*: *independent A*
and *AsB*: $A \subseteq \text{span } B$
shows $A = B$
proof (*intro antisym*[*OF - BA*] *subsetI*)
have *iB*: *independent B* **using** *independent-mono* [*OF iA BA*] .
fix *v* **assume** $v \in A$
with *AsB* **have** $v \in \text{span } B$ **by** *auto*
let *?RB* = *representation B v* **and** *?RA* = *representation A v*
have *?RB v* = 1
unfolding *representation-extend*[*OF iA* $\langle v \in \text{span } B \rangle$ *BA*, *symmetric*] *representation-basis*[*OF iA* $\langle v \in A \rangle$] **by** *simp*
then show $v \in B$
using *representation-ne-zero*[*of B v v*] **by** *auto*
qed

end

A linear function is a mapping between two modules over the same ring.

locale *module-hom* = *m1*: *module s1* + *m2*: *module s2*
for *s1* :: '*a*::*comm-ring-1* \Rightarrow '*b*::*ab-group-add* \Rightarrow '*b* (**infixr** **a* 75)
and *s2* :: '*a*::*comm-ring-1* \Rightarrow '*c*::*ab-group-add* \Rightarrow '*c* (**infixr** **b* 75) +
fixes *f* :: '*b* \Rightarrow '*c*
assumes *add*: $f (b1 + b2) = f b1 + f b2$
and *scale*: $f (r * a \ b) = r * b \ f \ b$
begin

lemma *zero*[*simp*]: $f \ 0 = 0$
using *scale*[*of 0 0*] **by** *simp*

lemma *neg*: $f (- x) = - f x$
using *scale* [**where** $r = -1$] **by** (*metis add add-eq-0-iff zero*)

lemma *diff*: $f (x - y) = f x - f y$
by (*metis diff-conv-add-uminus add neg*)

lemma *sum*: $f (\text{sum } g S) = (\sum a \in S. f (g a))$
proof (*induct S rule: infinite-finite-induct*)
case (*insert x F*)
have $f (\text{sum } g (\text{insert } x F)) = f (g x + \text{sum } g F)$
using *insert.hyps by simp*
also have $\dots = f (g x) + f (\text{sum } g F)$
using *add by simp*
also have $\dots = (\sum a \in \text{insert } x F. f (g a))$
using *insert.hyps by simp*
finally show *?case .*
qed *simp-all*

lemma *inj-on-iff-eq-0*:
assumes *s: m1.subspace s*
shows $\text{inj-on } f s \longleftrightarrow (\forall x \in s. f x = 0 \longrightarrow x = 0)$
proof –
have $\text{inj-on } f s \longleftrightarrow (\forall x \in s. \forall y \in s. f x - f y = 0 \longrightarrow x - y = 0)$
by (*simp add: inj-on-def*)
also have $\dots \longleftrightarrow (\forall x \in s. \forall y \in s. f (x - y) = 0 \longrightarrow x - y = 0)$
by (*simp add: diff*)
also have $\dots \longleftrightarrow (\forall x \in s. f x = 0 \longrightarrow x = 0)$ (**is** *?l = ?r*)
proof *safe*
fix *x* **assume** *?l* **assume** $x \in s$ $f x = 0$ **with** $\langle ?l \rangle [\text{rule-format, of } x 0]$ **show**
 $x = 0$
by (*auto simp: m1.subspace-0*)
next
fix *x y* **assume** *?r* **assume** $x \in s$ $y \in s$ $f (x - y) = 0$
with $\langle ?r \rangle [\text{rule-format, of } x - y]$ **show** $x - y = 0$
by (*auto simp: m1.subspace-diff*)
qed
finally show *?thesis*
by *auto*
qed

lemma *inj-iff-eq-0*: $\text{inj } f = (\forall x. f x = 0 \longrightarrow x = 0)$
by (*rule inj-on-iff-eq-0[OF m1.subspace-UNIV, unfolded ball-UNIV]*)

lemma *subspace-image*: **assumes** *S: m1.subspace S* **shows** $m2.\text{subspace } (f \text{ ` } S)$
unfolding *m2.subspace-def*
proof *safe*
show $0 \in f \text{ ` } S$
by (*rule image-eqI[of - - 0]*) (*auto simp: S m1.subspace-0*)
show $x \in S \implies y \in S \implies f x + f y \in f \text{ ` } S$ **for** *x y*
by (*rule image-eqI[of - - x + y]*) (*auto simp: S m1.subspace-add add*)

show $x \in S \implies r * b f x \in f \text{ ' } S$ **for** $r x$
by (rule image-eqI[of - - r * a x]) (auto simp: S m1.subspace-scale scale)
qed

lemma *subspace-vimage*: $m2.subspace S \implies m1.subspace (f \text{ ' } S)$
by (simp add: vimage-def add scale m1.subspace-def m2.subspace-0 m2.subspace-add m2.subspace-scale)

lemma *subspace-kernel*: $m1.subspace \{x. f x = 0\}$
using *subspace-vimage*[OF m2.subspace-single-0] **by** (simp add: vimage-def)

lemma *span-image*: $m2.span (f \text{ ' } S) = f \text{ ' } (m1.span S)$

proof (rule m2.span-unique)

show $f \text{ ' } S \subseteq f \text{ ' } m1.span S$

by (rule image-mono, rule m1.span-superset)

show $m2.subspace (f \text{ ' } m1.span S)$

using *m1.subspace-span* **by** (rule *subspace-image*)

next

fix T **assume** $f \text{ ' } S \subseteq T$ **and** $m2.subspace T$ **then show** $f \text{ ' } m1.span S \subseteq T$

unfolding *image-subset-iff-subset-vimage* **by** (metis *subspace-vimage m1.span-minimal*)

qed

lemma *dependent-inj-imageD*:

assumes $d: m2.dependent (f \text{ ' } s)$ **and** $i: inj\text{-on } f (m1.span s)$

shows $m1.dependent s$

proof –

have [intro]: $inj\text{-on } f s$

using $\langle inj\text{-on } f (m1.span s) \rangle$ *m1.span-superset* **by** (rule *inj-on-subset*)

from d **obtain** $s' r v$ **where** $*$: $finite\ s' s' \subseteq s$ ($\sum v \in f \text{ ' } s'. r v * b v = 0$) $v \in s'$
 $r (f v) \neq 0$

by (auto simp: *m2.dependent-explicit subset-image-iff dest!*: *finite-imageD intro: inj-on-subset*)

have $f (\sum v \in s'. r (f v) * a v) = (\sum v \in s'. r (f v) * b f v)$

by (simp add: *sum scale*)

also have $\dots = (\sum v \in f \text{ ' } s'. r v * b v)$

using $\langle s' \subseteq s \rangle$ **by** (*subst sum.reindex*) (auto *dest!*: *finite-imageD intro: inj-on-subset*)

finally have $f (\sum v \in s'. r (f v) * a v) = 0$

by (simp add: $*$)

with $\langle s' \subseteq s \rangle$ **have** ($\sum v \in s'. r (f v) * a v = 0$)

by (*intro inj-onD*[OF i] *m1.span-zero m1.span-sum m1.span-scale*) (auto *intro: m1.span-base*)

then show $m1.dependent s$

using $\langle finite\ s' \rangle \langle s' \subseteq s \rangle \langle v \in s' \rangle \langle r (f v) \neq 0 \rangle$ **by** (*force simp add: m1.dependent-explicit*)

qed

lemma *eq-0-on-span*:

assumes $f0: \bigwedge x. x \in b \implies f x = 0$ **and** $x: x \in m1.span b$ **shows** $f x = 0$

using *m1.span-induct*[OF x *subspace-kernel*] $f0$ **by** *simp*

lemma *independent-injective-image*: $m1.independent\ s \implies inj\ on\ f\ (m1.span\ s) \implies m2.independent\ (f\ 's)$

using *dependent-inj-imageD*[of *s*] **by** *auto*

lemma *inj-on-span-independent-image*:

assumes *ifB*: $m2.independent\ (f\ 'B)$ **and** *f*: *inj-on f B* **shows** *inj-on f (m1.span B)*

unfolding *inj-on-iff-eq-0*[OF *m1.subspace-span*] **unfolding** *m1.span-explicit'*

proof *safe*

fix *r* **assume** *fr*: *finite* $\{v. r\ v \neq 0\}$ **and** *r*: $\forall v. r\ v \neq 0 \implies v \in B$

and *eq0*: $f\ (\sum v \mid r\ v \neq 0. r\ v * a\ v) = 0$

have $0 = (\sum v \mid r\ v \neq 0. r\ v * b\ f\ v)$

using *eq0* **by** (*simp add: sum scale*)

also have $\dots = (\sum v \in f\ ' \{v. r\ v \neq 0\}. r\ (the\ inv\ into\ B\ f\ v) * b\ v)$

using *r* **by** (*subst sum.reindex*) (*auto simp: the-inv-into-f-f*[OF *f*] *intro!*:

inj-on-subset[OF *f*] *sum.cong*)

finally have $r\ v \neq 0 \implies r\ (the\ inv\ into\ B\ f\ (f\ v)) = 0$ **for** *v*

using *fr r ifB*[*unfolded m2.independent-explicit-module, rule-format,*
of f ' {v. r v ≠ 0} λv. r (the-inv-into B f v)]

by *auto*

then have $r\ v = 0$ **for** *v*

using *the-inv-into-f-f*[OF *f*] *r* **by** *auto*

then show $(\sum v \mid r\ v \neq 0. r\ v * a\ v) = 0$ **by** *auto*

qed

lemma *inj-on-span-iff-independent-image*: $m2.independent\ (f\ 'B) \implies inj\ on\ f\ (m1.span\ B) \iff inj\ on\ f\ B$

using *inj-on-span-independent-image*[of *B*] *inj-on-subset*[OF - *m1.span-superset,*
of f B] **by** *auto*

lemma *subspace-linear-preimage*: $m2.subspace\ S \implies m1.subspace\ \{x. f\ x \in S\}$

by (*simp add: add scale m1.subspace-def m2.subspace-def*)

lemma *spans-image*: $V \subseteq m1.span\ B \implies f\ 'V \subseteq m2.span\ (f\ 'B)$

by (*metis image-mono span-image*)

Relation between bases and injectivity/surjectivity of map.

lemma *spanning-surjective-image*:

assumes *us*: $UNIV \subseteq m1.span\ S$

and *sf*: *surj f*

shows $UNIV \subseteq m2.span\ (f\ 'S)$

proof –

have $UNIV \subseteq f\ 'UNIV$

using *sf* **by** (*auto simp add: surj-def*)

also have $\dots \subseteq m2.span\ (f\ 'S)$

using *spans-image*[OF *us*] .

finally show *?thesis* .

qed

lemmas *independent-inj-on-image = independent-injective-image*

lemma *independent-inj-image:*

m1.independent S \implies inj f \implies m2.independent (f ‘ S)

using *independent-inj-on-image[of S]* **by** (*auto simp: subset-inj-on*)

end

lemma *module-hom-iff:*

module-hom s1 s2 f \longleftrightarrow

module s1 \wedge module s2 \wedge

($\forall x y. f (x + y) = f x + f y$) \wedge ($\forall c x. f (s1 c x) = s2 c (f x)$)

by (*simp add: module-hom-def module-hom-axioms-def*)

locale *module-pair = m1: module s1 + m2: module s2*

for *s1 :: 'a :: comm-ring-1 \Rightarrow 'b \Rightarrow 'b :: ab-group-add*

and *s2 :: 'a :: comm-ring-1 \Rightarrow 'c \Rightarrow 'c :: ab-group-add*

begin

lemma *module-hom-zero: module-hom s1 s2 ($\lambda x. 0$)*

by (*simp add: module-hom-iff m1.module-axioms m2.module-axioms*)

lemma *module-hom-add: module-hom s1 s2 f \implies module-hom s1 s2 g \implies module-hom s1 s2 ($\lambda x. f x + g x$)*

by (*simp add: module-hom-iff module.scale-right-distrib*)

lemma *module-hom-sub: module-hom s1 s2 f \implies module-hom s1 s2 g \implies module-hom s1 s2 ($\lambda x. f x - g x$)*

by (*simp add: module-hom-iff module.scale-right-diff-distrib*)

lemma *module-hom-neg: module-hom s1 s2 f \implies module-hom s1 s2 ($\lambda x. - f x$)*

by (*simp add: module-hom-iff module.scale-minus-right*)

lemma *module-hom-scale: module-hom s1 s2 f \implies module-hom s1 s2 ($\lambda x. s2 c (f x)$)*

by (*simp add: module-hom-iff module.scale-scale module.scale-right-distrib ac-simps*)

lemma *module-hom-compose-scale:*

module-hom s1 s2 ($\lambda x. s2 (f x) (c)$)

if *module-hom s1 (*) f*

proof –

interpret *mh: module-hom s1 (*) f by fact*

show *?thesis*

by *unfold-locale (simp-all add: mh.add mh.scale m2.scale-left-distrib)*

qed

lemma *bij-module-hom-imp-inv-module-hom: module-hom scale1 scale2 f \implies bij f \implies*

module-hom scale2 scale1 (inv f)

by (*auto simp: module-hom-iff bij-is-surj bij-is-inj surj-f-inv-f*
intro!: Hilbert-Choice.inv-f-eq)

lemma *module-hom-sum*: $(\bigwedge i. i \in I \implies \text{module-hom } s1 \ s2 \ (f \ i)) \implies (I = \{\} \implies \text{module } s1 \wedge \text{module } s2) \implies \text{module-hom } s1 \ s2 \ (\lambda x. \sum_{i \in I}. f \ i \ x)$
apply (*induction I rule: infinite-finite-induct*)
apply (*auto intro!: module-hom-zero module-hom-add*)
using *m1.module-axioms m2.module-axioms* **by** *blast*

lemma *module-hom-eq-on-span*: $f \ x = g \ x$
if *module-hom s1 s2 f module-hom s1 s2 g*
and $(\bigwedge x. x \in B \implies f \ x = g \ x) \ x \in m1.\text{span } B$
proof –
interpret *module-hom s1 s2* $\lambda x. f \ x - g \ x$
by (*rule module-hom-sub that*)
from *eq-0-on-span[OF - that(4)] that(3)* **show** *?thesis* **by** *auto*
qed

end

context *module* **begin**

lemma *module-hom-scale-self[simp]*:
module-hom scale scale $(\lambda x. \text{scale } c \ x)$
using *module-axioms module-hom-iff scale-left-commute scale-right-distrib* **by**
blast

lemma *module-hom-scale-left[simp]*:
module-hom $(*)$ *scale* $(\lambda r. \text{scale } r \ x)$
by *unfold-locales (auto simp: algebra-simps)*

lemma *module-hom-id*: *module-hom scale scale id*
by (*simp add: module-hom-iff module-axioms*)

lemma *module-hom-ident*: *module-hom scale scale* $(\lambda x. x)$
by (*simp add: module-hom-iff module-axioms*)

lemma *module-hom-uminus*: *module-hom scale scale uminus*
by (*simp add: module-hom-iff module-axioms*)

end

lemma *module-hom-compose*: *module-hom s1 s2 f* \implies *module-hom s2 s3 g* \implies
module-hom s1 s3 $(g \ o \ f)$
by (*auto simp: module-hom-iff*)

end

107 Vector Spaces

```
theory Vector-Spaces
  imports Modules
begin
```

lemma *isomorphism-expand*:

```
 $f \circ g = id \wedge g \circ f = id \iff (\forall x. f (g x) = x) \wedge (\forall x. g (f x) = x)$ 
by (simp add: fun-eq-iff o-def id-def)
```

lemma *left-right-inverse-eq*:

```
assumes fg:  $f \circ g = id$ 
and gh:  $g \circ h = id$ 
shows  $f = h$ 
proof –
  have  $f = f \circ (g \circ h)$ 
    unfolding gh by simp
  also have  $\dots = (f \circ g) \circ h$ 
    by (simp add: o-assoc)
  finally show  $f = h$ 
    unfolding fg by simp
qed
```

lemma *ordLeq3-finite-infinite*:

```
assumes A: finite A and B: infinite B shows ordLeq3 (card-of A) (card-of B)
proof –
  have  $\langle \text{ordLeq3 } (\text{card-of } A) (\text{card-of } B) \vee \text{ordLeq3 } (\text{card-of } B) (\text{card-of } A) \rangle$ 
    by (intro ordLeq-total card-of-Well-order)
  moreover have  $\neg \text{ordLeq3 } (\text{card-of } B) (\text{card-of } A)$ 
    using B A card-of-ordLeq-finite[of B A] by auto
  ultimately show ?thesis by auto
qed
```

locale *vector-space* =

```
fixes scale :: 'a::field  $\Rightarrow$  'b::ab-group-add  $\Rightarrow$  'b (infixr *s 75)
assumes vector-space-assms:— re-stating the assumptions of module instead of
extending module allows us to rewrite in the sublocale.
   $a *s (x + y) = a *s x + a *s y$ 
   $(a + b) *s x = a *s x + b *s x$ 
   $a *s (b *s x) = (a * b) *s x$ 
   $1 *s x = x$ 
```

lemma *module-iff-vector-space*: *module* *s* \iff *vector-space* *s*

```
unfolding module-def vector-space-def ..
```

locale *linear* = *vs1*: *vector-space* *s1* + *vs2*: *vector-space* *s2* + *module-hom* *s1* *s2* *f*

```
for s1 :: 'a::field  $\Rightarrow$  'b::ab-group-add  $\Rightarrow$  'b (infixr *a 75)
and s2 :: 'a::field  $\Rightarrow$  'c::ab-group-add  $\Rightarrow$  'c (infixr *b 75)
and f :: 'b  $\Rightarrow$  'c
```

lemma *module-hom-iff-linear*: $\text{module-hom } s1 \ s2 \ f \longleftrightarrow \text{linear } s1 \ s2 \ f$
unfolding *module-hom-def linear-def module-iff-vector-space* **by** *auto*
lemmas *module-hom-eq-linear* = *module-hom-iff-linear*[*abs-def*, *THEN meta-eq-to-obj-eq*]
lemmas *linear-iff-module-hom* = *module-hom-iff-linear*[*symmetric*]
lemmas *linear-module-homI* = *module-hom-iff-linear*[*THEN iffD1*]
and *module-hom-linearI* = *module-hom-iff-linear*[*THEN iffD2*]

context *vector-space* **begin**

sublocale *module scale* **rewrites** *module-hom* = *linear*
by *unfold-locales (fact vector-space-assms module-hom-eq-linear)+*

lemmas— *from module*
linear-id = *module-hom-id*
and *linear-ident* = *module-hom-ident*
and *linear-scale-self* = *module-hom-scale-self*
and *linear-scale-left* = *module-hom-scale-left*
and *linear-uminus* = *module-hom-uminus*

lemma *linear-imp-scale*:
fixes $D::'a \Rightarrow 'b$
assumes *linear* (*) *scale D*
obtains d **where** $D = (\lambda x. \text{scale } x \ d)$
proof —
interpret *linear* (*) *scale D* **by** *fact*
show *?thesis*
by (*metis mult.commute mult.left-neutral scale that*)
qed

lemma *scale-eq-0-iff* [*simp*]: $\text{scale } a \ x = 0 \longleftrightarrow a = 0 \vee x = 0$
by (*metis scale-left-commute right-inverse scale-one scale-scale scale-zero-left*)

lemma *scale-left-imp-eq*:
assumes *nonzero*: $a \neq 0$
and *scale*: $\text{scale } a \ x = \text{scale } a \ y$
shows $x = y$
proof —
from *scale* **have** $\text{scale } a \ (x - y) = 0$
by (*simp add: scale-right-diff-distrib*)
with *nonzero* **have** $x - y = 0$ **by** *simp*
then show $x = y$ **by** (*simp only: right-minus-eq*)
qed

lemma *scale-right-imp-eq*:
assumes *nonzero*: $x \neq 0$
and *scale*: $\text{scale } a \ x = \text{scale } b \ x$
shows $a = b$
proof —

from *scale* **have** $\text{scale } (a - b) x = 0$
by (*simp add: scale-left-diff-distrib*)
with *nonzero* **have** $a - b = 0$ **by** *simp*
then show $a = b$ **by** (*simp only: right-minus-eq*)
qed

lemma *scale-cancel-left* [*simp*]: $\text{scale } a x = \text{scale } a y \longleftrightarrow x = y \vee a = 0$
by (*auto intro: scale-left-imp-eq*)

lemma *scale-cancel-right* [*simp*]: $\text{scale } a x = \text{scale } b x \longleftrightarrow a = b \vee x = 0$
by (*auto intro: scale-right-imp-eq*)

lemma *injective-scale*: $c \neq 0 \implies \text{inj } (\lambda x. \text{scale } c x)$
by (*simp add: inj-on-def*)

lemma *dependent-def*: $\text{dependent } P \longleftrightarrow (\exists a \in P. a \in \text{span } (P - \{a\}))$
unfolding *dependent-explicit*

proof *safe*

fix a **assume** $aP: a \in P$ **and** $a \in \text{span } (P - \{a\})$
then obtain $a S u$
where $aP: a \in P$ **and** $fS: \text{finite } S$ **and** $SP: S \subseteq P$ $a \notin S$ **and** $ua: (\sum_{v \in S. u v * s v} = a$
 $v * s v) = a$
unfolding *span-explicit* **by** *blast*
let $?S = \text{insert } a S$
let $?u = \lambda y. \text{if } y = a \text{ then } -1 \text{ else } u y$
from $fS SP$ **have** $(\sum_{v \in ?S. ?u v * s v} = 0$
by (*simp add: if-distrib[of $\lambda r. r * s a$ for a] sum.If-cases field-simps Diff-eq[symmetric]*
 $ua)$
moreover have $\text{finite } ?S$ $?S \subseteq P$ $a \in ?S$ $?u a \neq 0$
using $fS SP aP$ **by** *auto*
ultimately show $\exists t u. \text{finite } t \wedge t \subseteq P \wedge (\sum_{v \in t. u v * s v} = 0 \wedge (\exists v \in t. u v$
 $\neq 0))$ **by** *fast*

next

fix $S u v$
assume $fS: \text{finite } S$ **and** $SP: S \subseteq P$ **and** $vS: v \in S$
and $uv: u v \neq 0$ **and** $u: (\sum_{v \in S. u v * s v} = 0$
let $?a = v$
let $?S = S - \{v\}$
let $?u = \lambda i. (- u i) / u v$
have $th0: ?a \in P$ $\text{finite } ?S$ $?S \subseteq P$
using $fS SP vS$ **by** *auto*
have $(\sum_{v \in ?S. ?u v * s v} = (\sum_{v \in S. (- (\text{inverse } (u ?a))) * s (u v * s v)} - ?u$
 $v * s v$
using $fS vS uv$ **by** (*simp add: sum-diff1 field-simps*)
also have $\dots = ?a$
unfolding *scale-sum-right[symmetric]* u **using** uv **by** *simp*
finally have $(\sum_{v \in ?S. ?u v * s v} = ?a$
with $th0$ **show** $\exists a \in P. a \in \text{span } (P - \{a\})$
unfolding *span-explicit* **by** (*auto intro!: bexI[where $x=?a$] exI[where $x=?S$]*)

exI[where $x=?u$]
qed

lemma *dependent-single*[*simp*]: *dependent* $\{x\} \longleftrightarrow x = 0$
unfolding *dependent-def* **by** *auto*

lemma *in-span-insert*:

assumes $a: a \in \text{span } (\text{insert } b \ S)$

and $na: a \notin \text{span } S$

shows $b \in \text{span } (\text{insert } a \ S)$

proof –

from *span-breakdown*[of b insert b S a , *OF* *insertI1* a]

obtain k **where** $k: a - k *s b \in \text{span } (S - \{b\})$ **by** *auto*

have $k \neq 0$

proof

assume $k = 0$

with k *span-mono*[of $S - \{b\}$ S] **have** $a \in \text{span } S$ **by** *auto*

with na **show** *False* **by** *blast*

qed

then have *eq*: $b = (1/k) *s a - (1/k) *s (a - k *s b)$

by (*simp* *add*: *algebra-simps*)

from k **have** $(1/k) *s (a - k *s b) \in \text{span } (S - \{b\})$

by (*rule* *span-scale*)

also have $\dots \subseteq \text{span } (\text{insert } a \ S)$

by (*rule* *span-mono*) *auto*

finally show *?thesis*

using k **by** (*subst* *eq*) (*blast* *intro*: *span-diff* *span-scale* *span-base*)

qed

lemma *dependent-insertD*: **assumes** $a: a \notin \text{span } S$ **and** $S: \text{dependent } (\text{insert } a \ S)$
shows *dependent* S

proof –

have $a \notin S$ **using** a **by** (*auto* *dest*: *span-base*)

obtain b **where** $b: b = a \vee b \in S$ $b \in \text{span } (\text{insert } a \ S - \{b\})$

using S **unfolding** *dependent-def* **by** *blast*

have $b \neq a$ $b \in S$

using $b \langle a \notin S \rangle a$ **by** *auto*

with b **have** $*$: $b \in \text{span } (\text{insert } a \ (S - \{b\}))$

by (*auto* *simp*: *insert-Diff-if*)

show *dependent* S

proof *cases*

assume $b \in \text{span } (S - \{b\})$ **with** $\langle b \in S \rangle$ **show** *?thesis*

by (*auto* *simp* *add*: *dependent-def*)

next

assume $b \notin \text{span } (S - \{b\})$

with $*$ **have** $a \in \text{span } (\text{insert } b \ (S - \{b\}))$ **by** (*rule* *in-span-insert*)

with a **show** *?thesis*

using $\langle b \in S \rangle$ **by** (*auto* *simp*: *insert-absorb*)

qed
qed

lemma *independent-insertI*: $a \notin \text{span } S \implies \text{independent } S \implies \text{independent } (\text{insert } a \ S)$
by (*auto dest: dependent-insertD*)

lemma *independent-insert*:
 $\text{independent } (\text{insert } a \ S) \longleftrightarrow (\text{if } a \in S \text{ then independent } S \text{ else independent } S \wedge a \notin \text{span } S)$
proof –
have $a \notin S \implies a \in \text{span } S \implies \text{dependent } (\text{insert } a \ S)$
by (*auto simp: dependent-def*)
then show *?thesis*
by (*auto intro: dependent-mono simp: independent-insertI*)
qed

lemma *maximal-independent-subset-extend*:
assumes $S \subseteq V$ *independent* S
obtains B **where** $S \subseteq B$ $B \subseteq V$ *independent* B $V \subseteq \text{span } B$
proof –
let $?C = \{B. S \subseteq B \wedge \text{independent } B \wedge B \subseteq V\}$
have $\exists M \in ?C. \forall X \in ?C. M \subseteq X \longrightarrow X = M$
proof (*rule subset-Zorn*)
fix $C :: 'b \text{ set set}$ **assume** *subset.chain* $?C \ C$
then have $C: \bigwedge c. c \in C \implies c \subseteq V \wedge c. c \in C \implies S \subseteq c \wedge c. c \in C \implies$
independent c
 $\bigwedge c \ d. c \in C \implies d \in C \implies c \subseteq d \vee d \subseteq c$
unfolding *subset.chain-def* **by** *blast+*

show $\exists U \in ?C. \forall X \in C. X \subseteq U$
proof *cases*
assume $C = \{\}$ **with** *assms* **show** *?thesis*
by (*auto intro!: exI[of - S]*)

next
assume $C \neq \{\}$
with $C(2)$ **have** $S \subseteq \bigcup C$
by *auto*
moreover **have** *independent* $(\bigcup C)$
by (*intro independent-Union-directed C*)
moreover **have** $\bigcup C \subseteq V$
using C **by** *auto*
ultimately show *?thesis*
by *auto*

qed
qed
then obtain B **where** B : *independent* B $B \subseteq V$ $S \subseteq B$
and *max*: $\bigwedge S. \text{independent } S \implies S \subseteq V \implies B \subseteq S \implies S = B$
by *auto*

moreover
{ assume $\neg V \subseteq \text{span } B$
then obtain v **where** $v \in V \ v \notin \text{span } B$
by *auto*
with B **have** *independent (insert v B)* **by** (*auto intro: dependent-insertD*)
from *max[OF this] $\langle v \in V \rangle \langle B \subseteq V \rangle$*
have $v \in B$
by *auto*
with $\langle v \notin \text{span } B \rangle$ **have** *False*
by (*auto intro: span-base*) }
ultimately show *?thesis*
by (*meson that*)
qed

lemma *maximal-independent-subset:*

obtains B **where** $B \subseteq V$ *independent* $B \subseteq \text{span } B$
by (*metis maximal-independent-subset-extend[of {}] empty-subsetI independent-empty*)

Extends a basis from B to a basis of the entire space.

definition *extend-basis* :: $'b \text{ set} \Rightarrow 'b \text{ set}$

where *extend-basis* $B = (\text{SOME } B'. B \subseteq B' \wedge \text{independent } B' \wedge \text{span } B' = \text{UNIV})$

lemma

assumes B : *independent* B
shows *extend-basis-superset*: $B \subseteq \text{extend-basis } B$
and *independent-extend-basis*: *independent (extend-basis B)*
and *span-extend-basis[simp]*: $\text{span } (\text{extend-basis } B) = \text{UNIV}$

proof –

define p **where** $p \ B' \equiv B \subseteq B' \wedge \text{independent } B' \wedge \text{span } B' = \text{UNIV}$ **for** B'
obtain B' **where** $p \ B'$
using *maximal-independent-subset-extend[OF subset-UNIV B]*
by (*metis top.extremum-uniqueI p-def*)
then have p (*extend-basis B*)
unfolding *extend-basis-def p-def[symmetric]* **by** (*rule someI*)
then show $B \subseteq \text{extend-basis } B$ *independent (extend-basis B)* $\text{span } (\text{extend-basis } B) = \text{UNIV}$
by (*auto simp: p-def*)
qed

lemma *in-span-delete:*

assumes a : $a \in \text{span } S$ **and** na : $a \notin \text{span } (S - \{b\})$
shows $b \in \text{span } (\text{insert } a \ (S - \{b\}))$
by (*metis Diff-empty Diff-insert0 a in-span-insert insert-Diff na*)

lemma *span-redundant*: $x \in \text{span } S \implies \text{span } (\text{insert } x \ S) = \text{span } S$
unfolding *span-def* **by** (*rule hull-redundant*)

lemma *span-trans*: $x \in \text{span } S \implies y \in \text{span } (\text{insert } x \ S) \implies y \in \text{span } S$

by (*simp only: span-redundant*)

lemma *span-insert-0*[*simp*]: $\text{span} (\text{insert } 0 S) = \text{span } S$
 by (*metis span-zero span-redundant*)

lemma *span-delete-0* [*simp*]: $\text{span}(S - \{0\}) = \text{span } S$

proof

show $\text{span} (S - \{0\}) \subseteq \text{span } S$

by (*blast intro!: span-mono*)

next

have $\text{span } S \subseteq \text{span}(\text{insert } 0 (S - \{0\}))$

by (*blast intro!: span-mono*)

also have $\dots \subseteq \text{span}(S - \{0\})$

using *span-insert-0* by *blast*

finally show $\text{span } S \subseteq \text{span} (S - \{0\})$.

qed

lemma *span-image-scale*:

assumes *finite S* and *nz*: $\bigwedge x. x \in S \implies c x \neq 0$

shows $\text{span} ((\lambda x. c x *s x) ` S) = \text{span } S$

using *assms*

proof (*induction S arbitrary: c*)

case (*empty c*) show ?case by *simp*

next

case (*insert x F c*)

show ?case

proof (*intro set-eqI iffI*)

fix *y*

assume $y \in \text{span} ((\lambda x. c x *s x) ` \text{insert } x F)$

then show $y \in \text{span} (\text{insert } x F)$

using *insert* by (*force simp: span-breakdown-eq*)

next

fix *y*

assume $y \in \text{span} (\text{insert } x F)$

then show $y \in \text{span} ((\lambda x. c x *s x) ` \text{insert } x F)$

using *insert*

apply (*clarsimp simp: span-breakdown-eq*)

apply (*rule-tac x=k / c x in exI*)

by *simp*

qed

qed

lemma *exchange-lemma*:

assumes *f*: *finite T*

and *i*: *independent S*

and *sp*: $S \subseteq \text{span } T$

shows $\exists t'. \text{card } t' = \text{card } T \wedge \text{finite } t' \wedge S \subseteq t' \wedge t' \subseteq S \cup T \wedge S \subseteq \text{span } t'$

using *f i sp*

proof (*induct card (T - S) arbitrary: S T rule: less-induct*)

case *less*
note $ft = \langle \text{finite } T \rangle$ **and** $S = \langle \text{independent } S \rangle$ **and** $sp = \langle S \subseteq \text{span } T \rangle$
let $?P = \lambda t'. \text{card } t' = \text{card } T \wedge \text{finite } t' \wedge S \subseteq t' \wedge t' \subseteq S \cup T \wedge S \subseteq \text{span } t'$
show *?case*
proof (*cases* $S \subseteq T \vee T \subseteq S$)
 case *True*
 then show *?thesis*
 proof
 assume $S \subseteq T$ **then show** *?thesis*
 by (*metis* ft *Un-commute* sp *sup-ge1*)
 next
 assume $T \subseteq S$ **then show** *?thesis*
 by (*metis* *Un-absorb* sp *spanning-subset-independent*[*OF* - S sp] ft)
 qed
next
 case *False*
 then have $st: \neg S \subseteq T \neg T \subseteq S$
 by *auto*
 from $st(2)$ **obtain** b **where** $b: b \in T \ b \notin S$
 by *blast*
 from b **have** $T - \{b\} - S \subset T - S$
 by *blast*
 then have *cardlt*: $\text{card } (T - \{b\} - S) < \text{card } (T - S)$
 using ft **by** (*auto* *intro*: *psubset-card-mono*)
 from b ft **have** *ct0*: $\text{card } T \neq 0$
 by *auto*
 show *?thesis*
 proof (*cases* $S \subseteq \text{span } (T - \{b\})$)
 case *True*
 from ft **have** *ftb*: $\text{finite } (T - \{b\})$
 by *auto*
 from *less(1)*[*OF* *cardlt* *ftb* S *True*]
 obtain U **where** $U: \text{card } U = \text{card } (T - \{b\}) \ S \subseteq U \ U \subseteq S \cup (T - \{b\})$
 $S \subseteq \text{span } U$
 and *fu*: $\text{finite } U$ **by** *blast*
 let $?w = \text{insert } b \ U$
 have *th0*: $S \subseteq \text{insert } b \ U$
 using U **by** *blast*
 have *th1*: $\text{insert } b \ U \subseteq S \cup T$
 using $U \ b$ **by** *blast*
 have *bu*: $b \notin U$
 using $b \ U$ **by** *blast*
 from $U(1)$ $ft \ b$ **have** $\text{card } U = (\text{card } T - 1)$
 by *auto*
 then have *th2*: $\text{card } (\text{insert } b \ U) = \text{card } T$
 using *card-insert-disjoint*[*OF* *fu* *bu*] *ct0* **by** *auto*
 from $U(4)$ **have** $S \subseteq \text{span } U$.
 also have $\dots \subseteq \text{span } (\text{insert } b \ U)$
 by (*rule* *span-mono*) *blast*


```

finally have th3:  $S \subseteq \text{span} (\text{insert } b \ U)$  .
from th0 th1 th2 th3 fu have th:  $?P \ ?w$ 
  by blast
from th show ?thesis by blast
next
case False
then obtain a where a:  $a \in S \ a \notin \text{span} (T - \{b\})$ 
  by blast
have ab:  $a \neq b$ 
  using a b by blast
have at:  $a \notin T$ 
  using a ab span-base[of a T - {b}] by auto
have mlt:  $\text{card} ((\text{insert } a (T - \{b\})) - S) < \text{card} (T - S)$ 
  using cardlt ft a b by auto
have ft':  $\text{finite} (\text{insert } a (T - \{b\}))$ 
  using ft by auto
have sp':  $S \subseteq \text{span} (\text{insert } a (T - \{b\}))$ 
proof
  fix x
  assume xs:  $x \in S$ 
  have T:  $T \subseteq \text{insert } b (\text{insert } a (T - \{b\}))$ 
    using b by auto
  have bs:  $b \in \text{span} (\text{insert } a (T - \{b\}))$ 
    by (rule in-span-delete) (use a sp in auto)
  from xs sp have  $x \in \text{span } T$ 
    by blast
  with span-mono[OF T] have x:  $x \in \text{span} (\text{insert } b (\text{insert } a (T - \{b\})))$  ..
  from span-trans[OF bs x] show  $x \in \text{span} (\text{insert } a (T - \{b\}))$  .
qed
from less(1)[OF mlt ft' S sp'] obtain U where U:
   $\text{card } U = \text{card} (\text{insert } a (T - \{b\}))$ 
   $\text{finite } U \ S \subseteq U \ U \subseteq S \cup \text{insert } a (T - \{b\})$ 
   $S \subseteq \text{span } U$  by blast
from U a b ft at ct0 have ?P U
  by auto
then show ?thesis by blast
qed
qed
qed

```

lemma independent-span-bound:

```

assumes f:  $\text{finite } T$ 
  and i:  $\text{independent } S$ 
  and sp:  $S \subseteq \text{span } T$ 
shows  $\text{finite } S \wedge \text{card } S \leq \text{card } T$ 
by (metis exchange-lemma[OF f i sp] finite-subset card-mono)

```

lemma independent-explicit-finite-subsets:

```

 $\text{independent } A \longleftrightarrow (\forall S \subseteq A. \text{finite } S \longrightarrow (\forall u. (\sum_{v \in S. u \ v \ *s \ v}) = 0 \longrightarrow$ 

```

$(\forall v \in S. u v = 0))$

unfolding *dependent-explicit* [of A] **by** (*simp add: disj-not2*)

lemma *independent-if-scalars-zero*:

assumes *fin-A*: *finite A*

and *sum*: $\bigwedge f x. (\sum x \in A. f x * s x) = 0 \implies x \in A \implies f x = 0$

shows *independent A*

proof (*unfold independent-explicit-finite-subsets, clarify*)

fix $S v$ **and** $u :: 'b \Rightarrow 'a$

assume $S: S \subseteq A$ **and** $v: v \in S$

let $?g = \lambda x. \text{if } x \in S \text{ then } u x \text{ else } 0$

have $(\sum v \in A. ?g v * s v) = (\sum v \in S. u v * s v)$

using S *fin-A* **by** (*auto intro!: sum.mono-neutral-cong-right*)

also assume $(\sum v \in S. u v * s v) = 0$

finally have $?g v = 0$ **using** $v S$ *sum* **by** *force*

thus $u v = 0$ **unfolding** *if-P[OF v]* .

qed

lemma *bij-if-span-eq-span-bases*:

assumes B : *independent B* **and** C : *independent C*

and *eq*: $\text{span } B = \text{span } C$

shows $\exists f. \text{bij-betw } f B C$

proof *cases*

assume *finite B* \vee *finite C*

then have *finite B* \wedge *finite C* \wedge *card C* = *card B*

using *independent-span-bound*[of $B C$] *independent-span-bound*[of $C B$] *assms*
span-superset[of B] *span-superset*[of C]

by *auto*

then show *?thesis*

by (*auto intro!: finite-same-card-bij*)

next

assume \neg (*finite B* \vee *finite C*)

then have *infinite B* *infinite C* **by** *auto*

{ **fix** $B C$ **assume** B : *independent B* **and** C : *independent C* **and** *infinite B*
infinite C **and** *eq*: $\text{span } B = \text{span } C$

let $?R = \text{representation } B$ **and** $?R' = \text{representation } C$ **let** $?U = \lambda c. \{v. ?R c v \neq 0\}$

have *in-span-C* [*simp, intro*]: $\langle b \in B \implies b \in \text{span } C \rangle$ **for** b **unfolding**
eq[symmetric] **by** (*rule span-base*)

have *in-span-B* [*simp, intro*]: $\langle c \in C \implies c \in \text{span } B \rangle$ **for** c **unfolding** *eq* **by**
(*rule span-base*)

have $\langle B \subseteq (\bigcup c \in C. ?U c) \rangle$

proof

fix b **assume** $\langle b \in B \rangle$

have $\langle b \in \text{span } C \rangle$

using $\langle b \in B \rangle$ **unfolding** *eq[symmetric]* **by** (*rule span-base*)

have $\langle (\sum v \mid ?R' b v \neq 0. \sum w \mid ?R v w \neq 0. (?R' b v * ?R v w) * s w) =$
 $(\sum v \mid ?R' b v \neq 0. ?R' b v * s (\sum w \mid ?R v w \neq 0. ?R v w * s w)) \rangle$

by (*simp add: scale-sum-right*)

also have $\langle \dots = (\sum v \mid ?R' b v \neq 0. ?R' b v *s v) \rangle$
by (*auto simp: sum-nonzero-representation-eq B eq span-base representation-ne-zero*)
also have $\langle \dots = b \rangle$
by (*rule sum-nonzero-representation-eq[OF C $\langle b \in \text{span } C \rangle$]*)
finally have $?R b b = ?R (\sum v \mid ?R' b v \neq 0. \sum w \mid ?R v w \neq 0. (?R' b v * ?R v w) *s w) b$
by *simp*
also have $\dots = (\sum i \in \{v. ?R' b v \neq 0\}. ?R (\sum w \mid ?R i w \neq 0. (?R' b i * ?R i w) *s w) b)$
by (*subst representation-sum[OF B] (auto intro: span-sum span-scale span-base representation-ne-zero)*)
also have $\dots = (\sum i \in \{v. ?R' b v \neq 0\}. \sum j \in \{w. ?R i w \neq 0\}. ?R ((?R' b i * ?R i j) *s j) b)$
by (*subst representation-sum[OF B] (auto simp add: span-sum span-scale span-base representation-ne-zero)*)
also have $\langle \dots = (\sum v \mid ?R' b v \neq 0. \sum w \mid ?R v w \neq 0. ?R' b v * ?R v w * ?R w b) \rangle$
using $B \langle b \in B \rangle$ **by** (*simp add: representation-scale[OF B] span-base representation-ne-zero*)
finally have $(\sum v \mid ?R' b v \neq 0. \sum w \mid ?R v w \neq 0. ?R' b v * ?R v w * ?R w b) \neq 0$
using *representation-basis[OF B $\langle b \in B \rangle$]* **by** *auto*
then obtain $v w$ **where** $bv: ?R' b v \neq 0$ **and** $vw: ?R v w \neq 0$ **and** $?R' b v * ?R v w * ?R w b \neq 0$
by (*blast elim: sum.not-neutral-contains-not-neutral*)
with *representation-basis[OF B, of w] vw[THEN representation-ne-zero]*
have $\langle ?R' b v \neq 0 \rangle \langle ?R v b \neq 0 \rangle$ **by** (*auto split: if-splits*)
then show $\langle b \in (\bigcup c \in C. ?U c) \rangle$
by (*auto dest: representation-ne-zero*)
qed
then have $B\text{-eq: } \langle B = (\bigcup c \in C. ?U c) \rangle$
by (*auto intro: span-base representation-ne-zero eq*)
have $\text{ordLeq3 } (\text{card-of } B) (\text{card-of } C)$
proof (*subst B-eq, rule card-of-UNION-ordLeq-infinite[OF $\langle \text{infinite } C \rangle$]*)
show $\text{ordLeq3 } (\text{card-of } C) (\text{card-of } C)$
by (*intro ordLeq-refl card-of-Card-order*)
show $\forall c \in C. \text{ordLeq3 } (\text{card-of } \{v. ?R c v \neq 0\}) (\text{card-of } C)$
by (*intro ballI ordLeq3-finite-infinite $\langle \text{infinite } C \rangle$ finite-representation*)
**qed }
from *this*[of $B C$] *this*[of $C B$] $B C \text{ eq } \langle \text{infinite } C \rangle \langle \text{infinite } B \rangle$
show *?thesis* **by** (*auto simp add: ordIso-iff-ordLeq card-of-ordIso*)
qed**

definition $\text{dim} :: 'b \text{ set} \Rightarrow \text{nat}$

where $\text{dim } V = (\text{if } \exists b. \text{independent } b \wedge \text{span } b = \text{span } V \text{ then } \text{card } (\text{SOME } b. \text{independent } b \wedge \text{span } b = \text{span } V) \text{ else } 0)$

lemma *dim-eq-card*:

assumes BV : $\text{span } B = \text{span } V$ **and** B : *independent* B
shows $\text{dim } V = \text{card } B$
proof –
define p **where** $p\ b \equiv \text{independent } b \wedge \text{span } b = \text{span } V$ **for** b
have p (*SOME* B . $p\ B$)
using *assms* **by** (*intro someI*[of $p\ B$]) (*auto simp: p-def*)
then have $\exists f$. *bij-betw* $f\ B$ (*SOME* B . $p\ B$)
by (*subst (asm) p-def, intro bij-if-span-eq-span-bases*[OF B]) (*simp-all add:*
 BV)
then have $\text{card } B = \text{card} (\text{SOME } B. p\ B)$
by (*auto intro: bij-betw-same-card*)
then show *?thesis*
using $BV\ B$
by (*auto simp add: dim-def p-def*)
qed

lemma *basis-card-eq-dim*: $B \subseteq V \implies V \subseteq \text{span } B \implies \text{independent } B \implies \text{card } B = \text{dim } V$
using *dim-eq-card*[of $B\ V$] *span-mono*[of $B\ V$] *span-minimal*[OF - *subspace-span*,
of $V\ B$] **by** *auto*

lemma *basis-exists*:
obtains B **where** $B \subseteq V$ *independent* B $V \subseteq \text{span } B$ $\text{card } B = \text{dim } V$
by (*meson basis-card-eq-dim empty-subsetI independent-empty maximal-independent-subset-extend*)

lemma *dim-eq-card-independent*: $\text{independent } B \implies \text{dim } B = \text{card } B$
by (*rule dim-eq-card*[OF *refl*])

lemma *dim-span*[*simp*]: $\text{dim} (\text{span } S) = \text{dim } S$
by (*auto simp add: dim-def span-span*)

lemma *dim-span-eq-card-independent*: $\text{independent } B \implies \text{dim} (\text{span } B) = \text{card } B$
by (*simp add: dim-eq-card*)

lemma *dim-le-card*: **assumes** $V \subseteq \text{span } W$ *finite* W **shows** $\text{dim } V \leq \text{card } W$
proof –

obtain A **where** *independent* A $A \subseteq V$ $V \subseteq \text{span } A$
using *maximal-independent-subset*[of V] **by** *force*
with *assms independent-span-bound*[of $W\ A$] *basis-card-eq-dim*[of $A\ V$]
show *?thesis* **by** *auto*
qed

lemma *span-eq-dim*: $\text{span } S = \text{span } T \implies \text{dim } S = \text{dim } T$
by (*metis dim-span*)

corollary *dim-le-card'*:
finite $s \implies \text{dim } s \leq \text{card } s$
by (*metis basis-exists card-mono*)

lemma *span-card-ge-dim*:

$B \subseteq V \implies V \subseteq \text{span } B \implies \text{finite } B \implies \text{dim } V \leq \text{card } B$
by (*simp add: dim-le-card*)

lemma *dim-unique*:

$B \subseteq V \implies V \subseteq \text{span } B \implies \text{independent } B \implies \text{card } B = n \implies \text{dim } V = n$
by (*metis basis-card-eq-dim*)

lemma *subspace-sums*: $\llbracket \text{subspace } S; \text{subspace } T \rrbracket \implies \text{subspace } \{x + y \mid x \in S \wedge y \in T\}$

apply (*simp add: subspace-def*)

apply (*intro conjI impI allI; clarsimp simp: algebra-simps*)

using *add.left-neutral apply blast*

apply (*metis add.assoc*)

using *scale-right-distrib by blast*

end

lemma *linear-iff*: $\text{linear } s1 \ s2 \ f \longleftrightarrow$

$(\text{vector-space } s1 \wedge \text{vector-space } s2 \wedge (\forall x \ y. f (x + y) = f x + f y) \wedge (\forall c \ x. f (s1 \ c \ x) = s2 \ c (f x)))$

unfolding *linear-def module-hom-iff vector-space-def module-def by auto*

context begin

qualified lemma *linear-compose*: $\text{linear } s1 \ s2 \ f \implies \text{linear } s2 \ s3 \ g \implies \text{linear } s1 \ s3 \ (g \circ f)$

unfolding *module-hom-iff-linear[symmetric]*

by (*rule module-hom-compose*)

end

locale *vector-space-pair* = *vs1: vector-space s1 + vs2: vector-space s2*

for $s1 :: 'a::\text{field} \Rightarrow 'b::\text{ab-group-add} \Rightarrow 'b$ (**infixr** **a 75*)

and $s2 :: 'a::\text{field} \Rightarrow 'c::\text{ab-group-add} \Rightarrow 'c$ (**infixr** **b 75*)

begin

context *fixes f assumes linear s1 s2 f begin*

interpretation *linear s1 s2 f by fact*

lemmas— from locale *module-hom*

linear-0 = zero

and *linear-add = add*

and *linear-scale = scale*

and *linear-neg = neg*

and *linear-diff = diff*

and *linear-sum = sum*

and *linear-inj-on-iff-eq-0 = inj-on-iff-eq-0*

and *linear-inj-iff-eq-0 = inj-iff-eq-0*

and *linear-subspace-image = subspace-image*

and *linear-subspace-vimage = subspace-vimage*

and *linear-subspace-kernel = subspace-kernel*

```

and linear-span-image = span-image
and linear-dependent-inj-imageD = dependent-inj-imageD
and linear-eq-0-on-span = eq-0-on-span
and linear-independent-injective-image = independent-injective-image
and linear-inj-on-span-independent-image = inj-on-span-independent-image
and linear-inj-on-span-iff-independent-image = inj-on-span-iff-independent-image
and linear-subspace-linear-preimage = subspace-linear-preimage
and linear-spans-image = spans-image
and linear-spanning-surjective-image = spanning-surjective-image
end

```

```

sublocale module-pair
  rewrites module-hom = linear
  by unfold-locales (fact module-hom-eq-linear)

```

```

lemmas— from locale module-pair
  linear-eq-on-span = module-hom-eq-on-span
and linear-compose-scale-right = module-hom-scale
and linear-compose-add = module-hom-add
and linear-zero = module-hom-zero
and linear-compose-sub = module-hom-sub
and linear-compose-neg = module-hom-neg
and linear-compose-scale = module-hom-compose-scale

```

```

lemma linear-indep-image-lemma:
  assumes lf: linear s1 s2 f
    and fB: finite B
    and ifB: vs2.independent (f ‘ B)
    and fi: inj-on f B
    and xsB:  $x \in vs1.span\ B$ 
    and fx:  $f\ x = 0$ 
  shows  $x = 0$ 
  using fB ifB fi xsB fx
proof (induction B arbitrary: x rule: finite-induct)
  case empty
  then show ?case by auto
next
  case (insert a b x)
  have th0:  $f\ ‘\ b \subseteq f\ ‘\ (insert\ a\ b)$ 
    by (simp add: subset-insertI)
  have ifb: vs2.independent (f ‘ b)
    using vs2.independent-mono insert.prem(1) th0 by blast
  have fib: inj-on f b
    using insert.prem(2) by blast
  from vs1.span-breakdown[of a insert a b, simplified, OF insert.prem(3)]
  obtain k where  $x - k * a \in vs1.span\ (b - \{a\})$ 
    by blast
  have  $f\ (x - k * a) \in vs2.span\ (f\ ‘\ b)$ 
    unfolding linear-span-image[OF lf]

```

```

    using insert.hyps(2) k by auto
  then have f x - k * b f a ∈ vs2.span (f ' b)
    by (simp add: linear-diff linear-scale lf)
  then have th: -k * b f a ∈ vs2.span (f ' b)
    using insert.prem(4) by simp
  have xsb: x ∈ vs1.span b
  proof (cases k = 0)
    case True
      with k have x ∈ vs1.span (b - {a}) by simp
      then show ?thesis using vs1.span-mono[of b - {a} b]
        by blast
    next
      case False
      from inj-on-image-set-diff[OF insert.prem(2), of insert a b {a}, symmetric]
      have f ' insert a b - f ' {a} = f ' (insert a b - {a}) by blast
      then have f a ∉ vs2.span (f ' b)
        using vs2.dependent-def insert.hyps(2) insert.prem(1) by fastforce
      moreover have f a ∈ vs2.span (f ' b)
        using False vs2.span-scale[OF th, of - 1 / k] by auto
      ultimately have False
        by blast
      then show ?thesis by blast
  qed
  show x = 0
    using ifb fib xsb insert.IH insert.prem(4) by blast
  qed

```

lemma *linear-eq-on*:

```

  assumes l: linear s1 s2 f linear s1 s2 g
  assumes x: x ∈ vs1.span B and eq:  $\bigwedge b. b \in B \implies f b = g b$ 
  shows f x = g x
  proof -
    interpret d: linear s1 s2  $\lambda x. f x - g x$ 
    using l by (intro linear-compose-sub) (auto simp: module-hom-iff-linear)
    have f x - g x = 0
      by (rule d.eq-0-on-span[OF - x]) (auto simp: eq)
    then show ?thesis by auto
  qed

```

definition *construct* :: 'b set \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c)

```

  where construct B g v = ( $\sum b \mid vs1.representation (vs1.extend-basis B) v b \neq 0.$ 
    vs1.representation (vs1.extend-basis B) v b * b (if b ∈ B then g b else 0))

```

lemma *construct-cong*: ($\bigwedge b. b \in B \implies f b = g b$) \implies *construct* B f = *construct* B g

unfolding *construct-def* by (rule ext, auto intro!: sum.cong)

lemma *linear-construct*:

assumes $B[simp]$: $vs1.independent\ B$
shows $linear\ s1\ s2\ (construct\ B\ f)$
unfolding $module-hom-iff-linear\ linear-iff$
proof $safe$
have $eB[simp]$: $vs1.independent\ (vs1.extend-basis\ B)$
using $vs1.independent-extend-basis[OF\ B]$.
let $?R = vs1.representation\ (vs1.extend-basis\ B)$
fix $c\ x\ y$
have $construct\ B\ f\ (x + y) =$
 $(\sum\ b \in \{b. ?R\ x\ b \neq 0\} \cup \{b. ?R\ y\ b \neq 0\}. ?R\ (x + y)\ b * b\ (if\ b \in B\ then\ f\ b$
 $else\ 0))$
by $(auto\ intro!$: $sum.mono-neutral-cong-left\ simp$: $vs1.finite-representation\ vs1.representation-add$
 $construct-def)$
also have $\dots = construct\ B\ f\ x + construct\ B\ f\ y$
by $(auto\ simp$: $construct-def\ vs1.representation-add\ vs2.scale-left-distrib\ sum.distrib$
 $intro!$: $arg-cong2[where\ f=(+)]\ sum.mono-neutral-cong-right\ vs1.finite-representation)$
finally show $construct\ B\ f\ (x + y) = construct\ B\ f\ x + construct\ B\ f\ y$.

show $construct\ B\ f\ (c * a\ x) = c * b\ construct\ B\ f\ x$
by $(auto\ simp\ del$: $vs2.scale-scale\ intro!$: $sum.mono-neutral-cong-left\ vs1.finite-representation$
 $simp\ add$: $construct-def\ vs2.scale-scale[symmetric]\ vs1.representation-scale$
 $vs2.scale-sum-right)$
qed $intro-locales$

lemma $construct-basis$:

assumes $B[simp]$: $vs1.independent\ B$ **and** b : $b \in B$
shows $construct\ B\ f\ b = f\ b$
proof $-$
have $*$: $vs1.representation\ (vs1.extend-basis\ B)\ b = (\lambda v. if\ v = b\ then\ 1\ else\ 0)$
using $vs1.extend-basis-superset[OF\ B]\ b$
by $(intro\ vs1.representation-basis\ vs1.independent-extend-basis)\ auto$
then have $\{v. vs1.representation\ (vs1.extend-basis\ B)\ b\ v \neq 0\} = \{b\}$
by $auto$
then show $?thesis$
unfolding $construct-def$ **by** $(simp\ add$: $*\ b)$
qed

lemma $construct-outside$:

assumes B : $vs1.independent\ B$ **and** v : $v \in vs1.span\ (vs1.extend-basis\ B - B)$
shows $construct\ B\ f\ v = 0$
unfolding $construct-def$
proof $(clarsimp\ intro!$: $sum.neutral\ simp\ del$: $vs2.scale-eq-0-iff)$
fix b **assume** $b \in B$
then have $vs1.representation\ (vs1.extend-basis\ B - B)\ v\ b = 0$
using $vs1.representation-ne-zero[of\ vs1.extend-basis\ B - B\ v\ b]$ **by** $auto$
moreover have $vs1.representation\ (vs1.extend-basis\ B)\ v = vs1.representation$
 $(vs1.extend-basis\ B - B)\ v$
using $vs1.representation-extend[OF\ vs1.independent-extend-basis[OF\ B]\ v]$ **by**
 $auto$

ultimately show $vs1.representation (vs1.extend-basis B) v b * b f b = 0$
 by *simp*
qed

lemma *construct-add*:

assumes $B[simp]$: $vs1.independent B$
shows $construct B (\lambda x. f x + g x) v = construct B f v + construct B g v$
proof (rule *linear-eq-on*)
show $v \in vs1.span (vs1.extend-basis B)$ **by** *simp*
show $b \in vs1.extend-basis B \implies construct B (\lambda x. f x + g x) b = construct B f b + construct B g b$ **for** b
using $construct-outside[OF B vs1.span-base, of b]$ **by** (cases $b \in B$) (auto *simp*: *construct-basis*)
qed (intro *linear-compose-add linear-construct B*)+

lemma *construct-scale*:

assumes $B[simp]$: $vs1.independent B$
shows $construct B (\lambda x. c * b f x) v = c * b construct B f v$
proof (rule *linear-eq-on*)
show $v \in vs1.span (vs1.extend-basis B)$ **by** *simp*
show $b \in vs1.extend-basis B \implies construct B (\lambda x. c * b f x) b = c * b construct B f b$ **for** b
using $construct-outside[OF B vs1.span-base, of b]$ **by** (cases $b \in B$) (auto *simp*: *construct-basis*)
qed (intro *linear-construct module-hom-scale B*)+

lemma *construct-in-span*:

assumes $B[simp]$: $vs1.independent B$
shows $construct B f v \in vs2.span (f ' B)$
proof –
interpret c : *linear s1 s2 construct B f* **by** (rule *linear-construct*) *fact*
let $?R = vs1.representation B$
have $v \in vs1.span ((vs1.extend-basis B - B) \cup B)$
by (auto *simp*: *Un-absorb2 vs1.extend-basis-superset*)
then obtain $x y$ **where** $v = x + y$ $x \in vs1.span (vs1.extend-basis B - B)$ $y \in vs1.span B$
unfolding *vs1.span-Un* **by** *auto*
moreover have $construct B f (\sum b \mid ?R y b \neq 0. ?R y b * a b) \in vs2.span (f ' B)$
by (auto *simp* *add*: *c.sum c.scale construct-basis vs1.representation-ne-zero intro!*: *vs2.span-sum vs2.span-scale intro*: *vs2.span-base*)
ultimately show $construct B f v \in vs2.span (f ' B)$
by (auto *simp* *add*: *c.add construct-outside vs1.sum-nonzero-representation-eq*)
qed

lemma *linear-compose-sum*:

assumes lS : $\forall a \in S. linear s1 s2 (f a)$
shows *linear s1 s2* $(\lambda x. sum (\lambda a. f a x) S)$
proof (cases *finite S*)

```

case True
then show ?thesis
  using lS by induct (simp-all add: linear-zero linear-compose-add)
next
case False
then show ?thesis
  by (simp add: linear-zero)
qed

```

lemma *in-span-in-range-construct*:

$x \in \text{range } (\text{construct } B f)$ **if** $i: vs1.\text{independent } B$ **and** $x \in vs2.\text{span } (f \text{ ' } B)$

proof –

```

interpret linear (*a) (*b) construct B f
  using i by (rule linear-construct)
obtain bb :: ('b  $\Rightarrow$  'c)  $\Rightarrow$  ('b  $\Rightarrow$  'c)  $\Rightarrow$  'b set  $\Rightarrow$  'b where
   $\forall x0\ x1\ x2. (\exists v4. v4 \in x2 \wedge x1\ v4 \neq x0\ v4) = (bb\ x0\ x1\ x2 \in x2 \wedge x1\ (bb\ x0\ x1\ x2) \neq x0\ (bb\ x0\ x1\ x2))$ 
  by moura
then have f2:  $\forall B\ Ba\ f\ fa. (B \neq Ba \vee bb\ fa\ f\ Ba \in Ba \wedge f\ (bb\ fa\ f\ Ba) \neq fa\ (bb\ fa\ f\ Ba)) \vee f \text{ ' } B = fa \text{ ' } Ba$ 
  by (meson image-cong)
have  $vs1.\text{span } B \subseteq vs1.\text{span } (vs1.\text{extend-basis } B)$ 
  by (simp add: vs1.extend-basis-superset[OF i] vs1.span-mono)
then show  $x \in \text{range } (\text{construct } B f)$ 
  using f2 x by (metis (no-types) construct-basis[OF i, of - f]
    vs1.span-extend-basis[OF i] subsetD span-image spans-image)
qed

```

lemma *range-construct-eq-span*:

$\text{range } (\text{construct } B f) = vs2.\text{span } (f \text{ ' } B)$

if $vs1.\text{independent } B$

by (*auto simp: that construct-in-span in-span-in-range-construct*)

lemma *linear-independent-extend-subspace*:

– legacy: use *construct* instead

assumes $vs1.\text{independent } B$

shows $\exists g. \text{linear } s1\ s2\ g \wedge (\forall x \in B. g\ x = f\ x) \wedge \text{range } g = vs2.\text{span } (f \text{ ' } B)$

by (*rule exI[where x=construct B f]*)

(*auto simp: linear-construct assms construct-basis range-construct-eq-span*)

lemma *linear-independent-extend*:

$vs1.\text{independent } B \Longrightarrow \exists g. \text{linear } s1\ s2\ g \wedge (\forall x \in B. g\ x = f\ x)$

using *linear-independent-extend-subspace[of B f]* **by** *auto*

lemma *linear-exists-left-inverse-on*:

assumes *lf*: *linear s1 s2 f*

assumes $V: vs1.\text{subspace } V$ **and** $f: \text{inj-on } f\ V$

shows $\exists g. g \text{ ' } UNIV \subseteq V \wedge \text{linear } s2\ s1\ g \wedge (\forall v \in V. g\ (f\ v) = v)$

proof –

```

interpret linear s1 s2 f by fact
obtain B where V-eq:  $V = vs1.span\ B$  and B:  $vs1.independent\ B$ 
using  $vs1.maximal-independent-subset[of\ V]$   $vs1.span-minimal[OF - \langle vs1.subspace\ V \rangle]$ 
  by (metis antisym-conv)
have f: inj-on f (vs1.span B)
  using f unfolding V-eq .
show ?thesis
proof (intro exI ballI conjI)
  interpret p: vector-space-pair s2 s1 by unfold-locales
  have fB:  $vs2.independent\ (f\ ' B)$ 
    using independent-injective-image[OF B f] .
  let ?g = p.construct (f ' B) (the-inv-into B f)
  show linear (*b) (*a) ?g
    by (rule p.linear-construct[OF fB])
  have ?g b  $\in vs1.span\ (the-inv-into\ B\ f\ ' f\ ' B)$  for b
    by (intro p.construct-in-span fB)
  moreover have the-inv-into B f ' f ' B = B
    by (auto simp: image-comp comp-def the-inv-into-f-f inj-on-subset[OF f
vs1.span-superset])
    cong: image-cong)
  ultimately show ?g ' UNIV  $\subseteq V$ 
    by (auto simp: V-eq)
  have (?g o f) v = id v if v  $\in vs1.span\ B$  for v
  proof (rule vector-space-pair.linear-eq-on[where x=v])
    show vector-space-pair (*a) (*a) by unfold-locales
    show linear (*a) (*a) (?g o f)
  proof (rule Vector-Spaces.linear-compose[of - (*b)])
    show linear (*a) (*b) f
      by unfold-locales
  qed fact
  show linear (*a) (*a) id by (rule vs1.linear-id)
  show v  $\in vs1.span\ B$  by fact
  show b  $\in B \implies (p.construct\ (f\ ' B)\ (the-inv-into\ B\ f)\ o\ f)\ b = id\ b$  for b
    by (simp add: p.construct-basis fB the-inv-into-f-f inj-on-subset[OF f
vs1.span-superset])
  qed
  then show v  $\in V \implies ?g\ (f\ v) = v$  for v by (auto simp: comp-def id-def V-eq)
  qed
qed

```

lemma linear-exists-right-inverse-on:

```

assumes lf: linear s1 s2 f
assumes vs1.subspace V
shows  $\exists g. g\ ' UNIV \subseteq V \wedge linear\ s2\ s1\ g \wedge (\forall v \in f\ ' V. f\ (g\ v) = v)$ 
proof -
  obtain B where V-eq:  $V = vs1.span\ B$  and B:  $vs1.independent\ B$ 
  using  $vs1.maximal-independent-subset[of\ V]$   $vs1.span-minimal[OF - \langle vs1.subspace\ V \rangle]$ 

```

by (*metis antisym-conv*)
obtain C **where** C : $vs2.independent\ C$ **and** $fB-C$: $f\ 'B \subseteq vs2.span\ C\ C \subseteq f\ 'B$
using $vs2.maximal-independent-subset[of\ f\ 'B]$ **by** *metis*
then have $\forall v \in C. \exists b \in B. v = f\ b$ **by** *auto*
then obtain g **where** g : $\bigwedge v. v \in C \implies g\ v \in B \wedge v. v \in C \implies f\ (g\ v) = v$
by *metis*
show *?thesis*
proof (*intro exI ballI conjI*)
interpret p : *vector-space-pair s2 s1* **by** *unfold-locales*
let $?g = p.construct\ C\ g$
show *linear (*b) (*a) ?g*
by (*rule p.linear-construct[OF C]*)
have $?g\ v \in vs1.span\ (g\ 'C)$ **for** v
by (*rule p.construct-in-span[OF C]*)
also have $\dots \subseteq V$ **unfolding** *V-eq* **using** g **by** (*intro vs1.span-mono*) *auto*
finally show $?g\ 'UNIV \subseteq V$ **by** *auto*
have $(f \circ ?g)\ v = id\ v$ **if** v : $v \in f\ 'V$ **for** v
proof (*rule vector-space-pair.linear-eq-on[where x=v]*)
show *vector-space-pair (*b) (*b)* **by** *unfold-locales*
show *linear (*b) (*b) (f \circ ?g)*
by (*rule Vector-Spaces.linear-compose[of - (*a)]*) *fact+*
show *linear (*b) (*b) id* **by** (*rule vs2.linear-id*)
have $vs2.span\ (f\ 'B) = vs2.span\ C$
using $fB-C\ vs2.span-mono[of\ C\ f\ 'B]\ vs2.span-minimal[of\ f\ 'B\ vs2.span\ C]$
by *auto*
then show $v \in vs2.span\ C$
using v *linear-span-image[OF lf, of B]* **by** (*simp add: V-eq*)
show $(f \circ p.construct\ C\ g)\ b = id\ b$ **if** b : $b \in C$ **for** b
by (*auto simp: p.construct-basis\ g\ C\ b*)
qed
then show $v \in f\ 'V \implies f\ (?g\ v) = v$ **for** v **by** (*auto simp: comp-def id-def*)
qed
qed

lemma *linear-inj-on-left-inverse*:

assumes lf : *linear s1 s2 f*
assumes fi : *inj-on f (vs1.span S)*
shows $\exists g. range\ g \subseteq vs1.span\ S \wedge linear\ s2\ s1\ g \wedge (\forall x \in vs1.span\ S. g\ (f\ x) = x)$
using *linear-exists-left-inverse-on[OF lf vs1.subspace-span fi]*
by (*auto simp: linear-iff-module-hom*)

lemma *linear-injective-left-inverse*: $linear\ s1\ s2\ f \implies inj\ f \implies \exists g. linear\ s2\ s1\ g \wedge g \circ f = id$

using *linear-inj-on-left-inverse[of f UNIV]*
by *force*

lemma *linear-surj-right-inverse*:

assumes lf : *linear* $s1$ $s2$ f
assumes sf : $vs2.span\ T \subseteq f'vs1.span\ S$
shows $\exists g. range\ g \subseteq vs1.span\ S \wedge linear\ s2\ s1\ g \wedge (\forall x \in vs2.span\ T. f\ (g\ x) = x)$
using *linear-exists-right-inverse-on*[*OF* $lf\ vs1.subspace-span,$ *of* S] sf
by (*force simp: linear-iff-module-hom*)

lemma *linear-surjective-right-inverse*: $linear\ s1\ s2\ f \implies surj\ f \implies \exists g. linear\ s2\ s1\ g \wedge f \circ g = id$
using *linear-surj-right-inverse*[*of* $f\ UNIV\ UNIV$]
by (*auto simp: fun-eq-iff*)

lemma *finite-basis-to-basis-subspace-isomorphism*:

assumes s : $vs1.subspace\ S$
and t : $vs2.subspace\ T$
and d : $vs1.dim\ S = vs2.dim\ T$
and fB : *finite* B
and B : $B \subseteq S\ vs1.independent\ B\ S \subseteq vs1.span\ B\ card\ B = vs1.dim\ S$
and fC : *finite* C
and C : $C \subseteq T\ vs2.independent\ C\ T \subseteq vs2.span\ C\ card\ C = vs2.dim\ T$
shows $\exists f. linear\ s1\ s2\ f \wedge f' B = C \wedge f' S = T \wedge inj-on\ f\ S$

proof –

from $B(4)\ C(4)\ card-le-inj$ [*of* $B\ C$] d **obtain** f **where**
 f : $f' B \subseteq C\ inj-on\ f\ B$ **using** $\langle finite\ B \rangle\ \langle finite\ C \rangle$ **by** *auto*
from *linear-independent-extend*[*OF* $B(2)$] **obtain** g **where**
 g : $linear\ s1\ s2\ g\ \forall x \in B. g\ x = f\ x$ **by** *blast*
interpret g : $linear\ s1\ s2\ g$ **by** *fact*
from *inj-on-iff-eq-card*[*OF* $fB,$ *of* f] $f(2)$
have $card\ (f' B) = card\ B$ **by** *simp*
with $B(4)\ C(4)$ **have** ceq : $card\ (f' B) = card\ C$ **using** d
by *simp*
have $g' B = f' B$ **using** $g(2)$
by (*auto simp add: image-iff*)
also **have** $\dots = C$ **using** *card-subset-eq*[*OF* $fC\ f(1)\ ceq$].
finally **have** gBC : $g' B = C$.
have gi : $inj-on\ g\ B$ **using** $f(2)\ g(2)$
by (*auto simp add: inj-on-def*)
note $g0 = linear-indep-image-lemma$ [*OF* $g(1)\ fB,$ *unfolded* $gBC,$ *OF* $C(2)\ gi$]
{
fix $x\ y$
assume x : $x \in S$ **and** y : $y \in S$ **and** gxy : $g\ x = g\ y$
from $B(3)\ x\ y$ **have** x' : $x \in vs1.span\ B$ **and** y' : $y \in vs1.span\ B$
by *blast+*
from gxy **have** $th0$: $g\ (x - y) = 0$
by (*simp add: g.diff*)
have $th1$: $x - y \in vs1.span\ B$ **using** $x'\ y'$
by (*metis vs1.span-diff*)
have $x = y$ **using** $g0$ [*OF* $th1\ th0$] **by** *simp*
}

```

then have  $g \text{ ` } S$ : inj-on  $g \ S$  unfolding inj-on-def by blast
from  $vs1.\text{span-subspace}[OF \ B(1,3) \ s]$ 
have  $g \text{ ` } S = vs2.\text{span} \ (g \text{ ` } B)$ 
  by (simp add: g.span-image)
also have  $\dots = vs2.\text{span} \ C$ 
  unfolding  $gBC \ ..$ 
also have  $\dots = T$ 
  using  $vs2.\text{span-subspace}[OF \ C(1,3) \ t]$  .
finally have  $gS$ :  $g \text{ ` } S = T$  .
from  $g(1) \ gS \ g \text{ ` } S \ gBC$  show ?thesis
  by blast
qed

```

end

```

locale finite-dimensional-vector-space = vector-space +
  fixes  $Basis :: 'b \ set$ 
  assumes finite-Basis: finite Basis
  and independent-Basis: independent Basis
  and span-Basis: span Basis = UNIV
begin

```

```

definition dimension = card Basis

```

```

lemma finiteI-independent: independent B  $\implies$  finite B
  using independent-span-bound[OF finite-Basis, of B] by (auto simp: span-Basis)

```

```

lemma dim-empty [simp]: dim  $\{\}$  = 0
  by (rule dim-unique[OF order-refl]) (auto simp: dependent-def)

```

```

lemma dim-insert:

```

```

  dim (insert x S) = (if x  $\in$  span S then dim S else dim S + 1)

```

```

proof –

```

```

  show ?thesis

```

```

  proof (cases x  $\in$  span S)

```

```

    case True then show ?thesis

```

```

      by (metis dim-span span-redundant)

```

```

  next

```

```

    case False

```

```

    obtain  $B$  where  $B \subseteq \text{span } S$  independent B  $\text{span } S \subseteq \text{span } B$   $\text{card } B = \text{dim}$ 
      (span S)

```

```

    using basis-exists [of span S] by blast

```

```

    have  $\text{dim} \ (\text{span} \ (\text{insert } x \ S)) = \text{Suc} \ (\text{dim } S)$ 

```

```

    proof (rule dim-unique)

```

```

      show  $\text{insert } x \ B \subseteq \text{span} \ (\text{insert } x \ S)$ 

```

```

        by (meson B(1) insertI1 insert-subset order-trans span-base span-mono
subset-insertI)

```

```

      show  $\text{span} \ (\text{insert } x \ S) \subseteq \text{span} \ (\text{insert } x \ B)$ 

```

```

    by (metis ‹ $B \subseteq \text{span } S$ › ‹ $\text{span } S \subseteq \text{span } B$ › span-breakdown-eq span-subspace
subsetI subspace-span)
    show independent (insert x B)
    by (metis B(1-3) independent-insert span-subspace subspace-span False)
    show card (insert x B) = Suc (dim S)
    using B False finiteI-independent by force
  qed
  then show ?thesis
    by (metis False Suc-eq-plus1 dim-span)
  qed
qed

```

```

lemma dim-singleton [simp]: dim{x} = (if x = 0 then 0 else 1)
  by (simp add: dim-insert)

```

proposition choose-subspace-of-subspace:

```

  assumes n ≤ dim S
  obtains T where subspace T T ⊆ span S dim T = n
  proof -
    have ∃ T. subspace T ∧ T ⊆ span S ∧ dim T = n
    using assms
    proof (induction n)
      case 0 then show ?case by (auto intro!: exI[where x={0}] span-zero)
    next
      case (Suc n)
      then obtain T where subspace T T ⊆ span S dim T = n
        by force
      then show ?case
      proof (cases span S ⊆ span T)
        case True
        have span T ⊆ span S
          by (simp add: ‹ $T \subseteq \text{span } S$ › span-minimal)
        then have dim S = dim T
          by (rule span-eq-dim [OF subset-antisym [OF True]])
        then show ?thesis
          using Suc.premis ‹dim T = n› by linarith
      next
        case False
        then obtain y where y: y ∈ S y ∉ T
          by (meson span-mono subsetI)
        then have span (insert y T) ⊆ span S
          by (metis (no-types) ‹ $T \subseteq \text{span } S$ › subsetD insert-subset span-superset
span-mono span-span)
        with ‹dim T = n› ‹subspace T› y show ?thesis
          apply (rule-tac x=span(insert y T) in exI)
          using span-eq-iff by (fastforce simp: dim-insert)
      qed
    qed
  with that show ?thesis by blast

```

qed

lemma *basis-subspace-exists*:

assumes *subspace S*

obtains *B* **where** *finite B B ⊆ S independent B span B = S card B = dim S*

by (*metis assms span-subspace basis-exists finiteI-independent*)

lemma *dim-mono*: **assumes** $V \subseteq \text{span } W$ **shows** $\text{dim } V \leq \text{dim } W$

proof –

obtain *B* **where** *independent B B ⊆ W W ⊆ span B*

using *maximal-independent-subset[of W]* **by** *force*

with *dim-le-card[of V B] assms independent-span-bound[of Basis B] basis-card-eq-dim[of B W]*

span-mono[of B W] span-minimal[OF - subspace-span, of W B]

show *?thesis*

by (*auto simp: finite-Basis span-Basis*)

qed

lemma *dim-subset*: $S \subseteq T \implies \text{dim } S \leq \text{dim } T$

using *dim-mono[of S T]* **by** (*auto intro: span-base*)

lemma *dim-eq-0 [simp]*:

$\text{dim } S = 0 \iff S \subseteq \{0\}$

by (*metis basis-exists card-eq-0-iff dim-span finiteI-independent span-empty subset-empty subset-singletonD*)

lemma *dim-UNIV[simp]*: $\text{dim UNIV} = \text{card Basis}$

using *dim-eq-card[of Basis UNIV]* **by** (*simp add: independent-Basis span-Basis*)

lemma *independent-card-le-dim*: **assumes** $B \subseteq V$ **and** *independent B* **shows** $\text{card } B \leq \text{dim } V$

by (*subst dim-eq-card[symmetric, OF refl ‹independent B›] (rule dim-subset[OF ‹B ⊆ V›])*)

lemma *dim-subset-UNIV*: $\text{dim } S \leq \text{dimension}$

by (*metis dim-subset subset-UNIV dim-UNIV dimension-def*)

lemma *card-ge-dim-independent*:

assumes *BV: B ⊆ V*

and *iB: independent B*

and *dVB: dim V ≤ card B*

shows $V \subseteq \text{span } B$

proof

fix *a*

assume *aV: a ∈ V*

{

assume *aB: a ∉ span B*

then have *iaB: independent (insert a B)*

using *iB aV BV* **by** (*simp add: independent-insert*)

}


```

from  $aV BV$  have  $th0: insert\ a\ B \subseteq V$ 
  by blast
from  $aB$  have  $a \notin B$ 
  by (auto simp add: span-base)
with independent-card-le-dim[OF th0 iaB] dVB finiteI-independent[OF iB]
have False by auto
}
then show  $a \in span\ B$  by blast
qed

```

lemma *card-le-dim-spanning*:

```

assumes  $BV: B \subseteq V$ 
  and  $VB: V \subseteq span\ B$ 
  and  $fB: finite\ B$ 
  and  $dVB: dim\ V \geq card\ B$ 
shows independent\ B
proof –
{
  fix  $a$ 
  assume  $a: a \in B\ a \in span\ (B - \{a\})$ 
  from  $a\ fB$  have  $c0: card\ B \neq 0$ 
    by auto
  from  $a\ fB$  have  $cb: card\ (B - \{a\}) = card\ B - 1$ 
    by auto
  {
    fix  $x$ 
    assume  $x: x \in V$ 
    from  $a$  have  $eq: insert\ a\ (B - \{a\}) = B$ 
      by blast
    from  $x\ VB$  have  $x': x \in span\ B$ 
      by blast
    from span-trans[OF a(2), unfolded eq, OF x']
    have  $x \in span\ (B - \{a\})$  .
  }
  then have  $th1: V \subseteq span\ (B - \{a\})$ 
    by blast
  have  $th2: finite\ (B - \{a\})$ 
    using  $fB$  by auto
  from dim-le-card[OF th1 th2]
  have  $c: dim\ V \leq card\ (B - \{a\})$  .
  from  $c\ c0\ dVB\ cb$  have False by simp
}
then show ?thesis
  unfolding dependent-def by blast
qed

```

lemma *card-eq-dim*: $B \subseteq V \implies card\ B = dim\ V \implies finite\ B \implies independent\ B$
 $\iff V \subseteq span\ B$

by (*metis order-eq-iff card-le-dim-spanning card-ge-dim-independent*)

lemma *subspace-dim-equal*:

assumes *subspace S*
and *subspace T*
and $S \subseteq T$
and $\dim S \geq \dim T$
shows $S = T$

proof –

obtain B **where** $B \leq S$ *independent* $B \wedge S \subseteq \text{span } B$ $\text{card } B = \dim S$
using *basis-exists[of S]* **by** *metis*
then have $\text{span } B \subseteq S$
using *span-mono[of B S]* *span-eq-iff[of S]* *assms* **by** *metis*
then have $\text{span } B = S$
using B **by** *auto*
have $\dim S = \dim T$
using *assms* *dim-subset[of S T]* **by** *auto*
then have $T \subseteq \text{span } B$
using *card-eq-dim[of B T]* B *finiteI-independent* *assms* **by** *auto*
then show *?thesis*
using *assms* $\langle \text{span } B = S \rangle$ **by** *auto*

qed

corollary *dim-eq-span*:

shows $\llbracket S \subseteq T; \dim T \leq \dim S \rrbracket \implies \text{span } S = \text{span } T$
by (*simp add: span-mono subspace-dim-equal*)

lemma *dim-psubset*:

$\text{span } S \subset \text{span } T \implies \dim S < \dim T$
by (*metis (no-types, opaque-lifting) dim-span less-le not-le subspace-dim-equal subspace-span*)

lemma *dim-eq-full*:

shows $\dim S = \text{dimension} \iff \text{span } S = \text{UNIV}$
by (*metis dim-eq-span dim-subset-UNIV span-Basis span-span subset-UNIV dim-UNIV dim-span dimension-def*)

lemma *indep-card-eq-dim-span*:

assumes *independent B*
shows $\text{finite } B \wedge \text{card } B = \dim (\text{span } B)$
using *dim-span-eq-card-independent[OF assms]* *finiteI-independent[OF assms]* **by** *auto*

More general size bound lemmas.

lemma *independent-bound-general*:

$\text{independent } S \implies \text{finite } S \wedge \text{card } S \leq \dim S$
by (*simp add: dim-eq-card-independent finiteI-independent*)

lemma *independent-explicit*:

shows $\text{independent } B \iff \text{finite } B \wedge (\forall c. (\sum v \in B. c v * s v) = 0 \implies (\forall v \in B.$

$c v = 0$)

using *independent-bound-general*
by (*fastforce simp: dependent-finite*)

proposition *dim-sums-Int:*

assumes *subspace S subspace T*

shows $\dim \{x + y \mid x \in S \wedge y \in T\} + \dim(S \cap T) = \dim S + \dim T$ (**is**
 $\dim ?ST + - = -$)

proof –

obtain B **where** $B \subseteq S \cap T$ $S \cap T \subseteq \text{span } B$

and $\text{ind} B$: *independent B*

and $\text{card} B$: $\text{card } B = \dim(S \cap T)$

using *basis-exists by metis*

then obtain $C D$ **where** $B \subseteq C$ $C \subseteq S$ *independent C S* $C \subseteq \text{span } C$

and $B \subseteq D$ $D \subseteq T$ *independent D T* $D \subseteq \text{span } D$

using *maximal-independent-subset-extend*

by (*metis Int-subset-iff* $\langle B \subseteq S \cap T \rangle \text{ind} B$)

then have *finite B finite C finite D*

by (*simp-all add: finiteI-independent indB independent-bound-general*)

have $\text{B}eq$: $B = C \cap D$

proof (*rule spanning-subset-independent [symmetric]*)

show *independent (C ∩ D)*

by (*meson* $\langle \text{independent } C \rangle$ *independent-mono inf.cobounded1*)

qed (*use B* $\langle C \subseteq S \rangle$ $\langle D \subseteq T \rangle$ $\langle B \subseteq C \rangle$ $\langle B \subseteq D \rangle$ **in** *auto*)

then have $\text{D}eq$: $D = B \cup (D - C)$

by *blast*

have CUD : $C \cup D \subseteq ?ST$

proof (*simp, intro conjI*)

show $C \subseteq ?ST$

using *span-zero span-minimal [OF -* $\langle \text{subspace } T \rangle$ $\langle C \subseteq S \rangle$ **by** *force*

show $D \subseteq ?ST$

using *span-zero span-minimal [OF -* $\langle \text{subspace } S \rangle$ $\langle D \subseteq T \rangle$ **by** *force*

qed

have $a v = 0$ **if** 0 : $(\sum v \in C. a v * s v) + (\sum v \in D - C. a v * s v) = 0$

and v : $v \in C \cup (D - C)$ **for** $a v$

proof –

have CsS : $\bigwedge x. x \in C \implies a x * s x \in S$

using $\langle C \subseteq S \rangle$ $\langle \text{subspace } S \rangle$ *subspace-scale* **by** *auto*

have eq : $(\sum v \in D - C. a v * s v) = - (\sum v \in C. a v * s v)$

using *that add-eq-0-iff* **by** *blast*

have $(\sum v \in D - C. a v * s v) \in S$

by (*simp add: eq CsS* $\langle \text{subspace } S \rangle$ *subspace-neg subspace-sum*)

moreover have $(\sum v \in D - C. a v * s v) \in T$

apply (*rule subspace-sum [OF* $\langle \text{subspace } T \rangle$])

by (*meson DiffD1* $\langle D \subseteq T \rangle$ $\langle \text{subspace } T \rangle$ *subset-eq subspace-def*)

ultimately have $(\sum v \in D - C. a v * s v) \in \text{span } B$

using B **by** *blast*

then obtain e **where** e : $(\sum v \in B. e v * s v) = (\sum v \in D - C. a v * s v)$

using *span-finite [OF* $\langle \text{finite } B \rangle$ **by** *force*

```

have  $\bigwedge c v. \llbracket (\sum v \in C. c v * s v) = 0; v \in C \rrbracket \implies c v = 0$ 
  using  $\langle \text{finite } C \rangle \langle \text{independent } C \rangle \text{ independent } D$  by blast
define cc where  $cc x = (\text{if } x \in B \text{ then } a x + e x \text{ else } a x)$  for x
have [simp]:  $C \cap B = B \ D \cap B = B \ C \cap - B = C - D \ B \cap (D - C) = \{\}$ 
  using  $\langle B \subseteq C \rangle \langle B \subseteq D \rangle \text{ Beq}$  by blast+
have f2:  $(\sum v \in C \cap D. e v * s v) = (\sum v \in D - C. a v * s v)$ 
  using  $\text{Beq } e$  by presburger
have f3:  $(\sum v \in C \cup D. a v * s v) = (\sum v \in C - D. a v * s v) + (\sum v \in D - C. a v * s v) + (\sum v \in C \cap D. a v * s v)$ 
  using  $\langle \text{finite } C \rangle \langle \text{finite } D \rangle \text{ sum.union-diff2}$  by blast
have f4:  $(\sum v \in C \cup (D - C). a v * s v) = (\sum v \in C. a v * s v) + (\sum v \in D - C. a v * s v)$ 
  by (meson  $\text{Diff-disjoint } \langle \text{finite } C \rangle \langle \text{finite } D \rangle \text{ finite-Diff sum.union-disjoint}$ )
have  $(\sum v \in C. cc v * s v) = 0$ 
  using 0 f2 f3 f4
apply (simp add: cc-def Beq  $\langle \text{finite } C \rangle \text{ sum.If-cases algebra-simps sum.distrib if-distrib if-distribR}$ )
apply (simp add: add.commute add.left-commute diff-eq)
done
then have  $\bigwedge v. v \in C \implies cc v = 0$ 
  using independent-explicit  $\langle \text{independent } C \rangle \langle \text{finite } C \rangle$  by blast
then have C0:  $\bigwedge v. v \in C - B \implies a v = 0$ 
  by (simp add: cc-def Beq) meson
then have [simp]:  $(\sum x \in C - B. a x * s x) = 0$ 
  by simp
have  $(\sum x \in C. a x * s x) = (\sum x \in B. a x * s x)$ 
proof -
  have  $C - D = C - B$ 
  using Beq by blast
  then show ?thesis
  using Beq  $\langle \sum x \in C - B. a x * s x = 0 \rangle f3 f4$  by auto
qed
with 0 have Dcc0:  $(\sum v \in D. a v * s v) = 0$ 
  by (subst Deq) (simp add:  $\langle \text{finite } B \rangle \langle \text{finite } D \rangle \text{ sum-Un}$ )
then have D0:  $\bigwedge v. v \in D \implies a v = 0$ 
  using independent-explicit  $\langle \text{independent } D \rangle \langle \text{finite } D \rangle$  by blast
show ?thesis
  using v C0 D0 Beq by blast
qed
then have independent  $(C \cup (D - C))$ 
  unfolding independent-explicit
  using independent-explicit
  by (simp add: independent-explicit  $\langle \text{finite } C \rangle \langle \text{finite } D \rangle \text{ sum-Un del: Un-Diff-cancel}$ )
then have indCUD: independent  $(C \cup D)$  by simp
have  $\dim (S \cap T) = \text{card } B$ 
  by (rule dim-unique [OF B indB refl])
moreover have  $\dim S = \text{card } C$ 
  by (metis  $\langle C \subseteq S \rangle \langle \text{independent } C \rangle \langle S \subseteq \text{span } C \rangle \text{ basis-card-eq-dim}$ )
moreover have  $\dim T = \text{card } D$ 

```

by (*metis* $\langle D \subseteq T \rangle \langle \text{independent } D \rangle \langle T \subseteq \text{span } D \rangle$ *basis-card-eq-dim*)
 moreover have $\dim ?ST = \text{card}(C \cup D)$
 proof –
 have *: $\bigwedge x y. \llbracket x \in S; y \in T \rrbracket \implies x + y \in \text{span } (C \cup D)$
 by (*meson* $\langle S \subseteq \text{span } C \rangle \langle T \subseteq \text{span } D \rangle$ *span-add span-mono subsetCE*
sup.cobounded1 sup.cobounded2)
 show *?thesis*
 by (*auto intro: * dim-unique [OF CUD - indCUD refl]*)
 qed
 ultimately show *?thesis*
 using $\langle B = C \cap D \rangle$ [*symmetric*]
 by (*simp add: \langle independent C \rangle \langle independent D \rangle card-Un-Int finiteI-independent*)
 qed

lemma *dependent-biggerset-general*:
(finite S \implies card S > dim S) \implies dependent S
 using *independent-bound-general[of S]* by (*metis linorder-not-le*)

lemma *subset-le-dim*:
S \subseteq span T \implies dim S \leq dim T
 by (*metis dim-span dim-subset*)

lemma *linear-inj-imp-surj*:
 assumes *lf: linear scale scale f*
 and *fi: inj f*
 shows *surj f*
 proof –
 interpret *lf: linear scale scale f* by *fact*
 from *basis-exists[of UNIV]* obtain *B*
 where *B: B \subseteq UNIV independent B UNIV \subseteq span B card B = dim UNIV*
 by *blast*
 from *B(4)* have *d: dim UNIV = card B*
 by *simp*
 have *UNIV \subseteq span (f ‘ B)*
 proof (*rule card-ge-dim-independent*)
 show *independent (f ‘ B)*
 by (*simp add: B(2) fi lf.independent-inj-image*)
 have *card (f ‘ B) = dim UNIV*
 by (*metis B(1) card-image d fi inj-on-subset*)
 then show *dim UNIV \leq card (f ‘ B)*
 by *simp*
 qed *blast*
 then show *?thesis*
 unfolding *lf.span-image surj-def*
 using *B(3)* by *blast*
 qed

end

```

locale finite-dimensional-vector-space-pair-1 =
  vs1: finite-dimensional-vector-space s1 B1 + vs2: vector-space s2
  for s1 :: 'a::field  $\Rightarrow$  'b::ab-group-add  $\Rightarrow$  'b (infixr *a 75)
  and B1 :: 'b set
  and s2 :: 'a::field  $\Rightarrow$  'c::ab-group-add  $\Rightarrow$  'c (infixr *b 75)
begin

sublocale vector-space-pair s1 s2 by unfold-locales

lemma dim-image-eq:
  assumes lf: linear s1 s2 f
    and fi: inj-on f (vs1.span S)
  shows vs2.dim (f ' S) = vs1.dim S
proof -
  interpret lf: linear by fact
  obtain B where B:  $B \subseteq S$  vs1.independent B  $S \subseteq$  vs1.span B  $\text{card } B =$  vs1.dim S
  using vs1.basis-exists[of S] by auto
  then have vs1.span S = vs1.span B
  using vs1.span-mono[of B S] vs1.span-mono[of S vs1.span B] vs1.span-span[of B] by auto
  moreover have  $\text{card } (f ' B) = \text{card } B$ 
  using assms card-image[of f B] subset-inj-on[of f vs1.span S B] B vs1.span-superset by auto
  moreover have  $(f ' B) \subseteq (f ' S)$ 
  using B by auto
  ultimately show ?thesis
  by (metis B(2) B(4) fi lf.dependent-inj-imageD lf.span-image vs2.dim-eq-card-independent vs2.dim-span)
qed

lemma dim-image-le:
  assumes lf: linear s1 s2 f
  shows vs2.dim (f ' S)  $\leq$  vs1.dim (S)
proof -
  from vs1.basis-exists[of S] obtain B where
    B:  $B \subseteq S$  vs1.independent B  $S \subseteq$  vs1.span B  $\text{card } B =$  vs1.dim S by blast
  from B have fB: finite B  $\text{card } B =$  vs1.dim S
  using vs1.independent-bound-general by blast+
  have vs2.dim (f ' S)  $\leq$   $\text{card } (f ' B)$ 
  apply (rule vs2.span-card-ge-dim)
  using lf B fB
  apply (auto simp add: module-hom.span-image module-hom.spans-image subset-image-iff
    linear-iff-module-hom)
  done
  also have ...  $\leq$  vs1.dim S
  using card-image-le[OF fB(1)] fB by simp
  finally show ?thesis .

```

qed

end

locale *finite-dimensional-vector-space-pair* =
vs1: finite-dimensional-vector-space s1 B1 + vs2: finite-dimensional-vector-space
s2 B2
for *s1 :: 'a::field \Rightarrow 'b::ab-group-add \Rightarrow 'b* (**infixr** *a 75)
and *B1 :: 'b set*
and *s2 :: 'a::field \Rightarrow 'c::ab-group-add \Rightarrow 'c* (**infixr** *b 75)
and *B2 :: 'c set*
begin

sublocale *finite-dimensional-vector-space-pair-1* ..

lemma *linear-surjective-imp-injective*:

assumes *lf: linear s1 s2 f and sf: surj f and eq: vs2.dim UNIV = vs1.dim UNIV*

shows *inj f*

proof –

interpret *linear s1 s2 f by fact*

have * : *card (f ‘ B1) \leq vs2.dim UNIV*

using *vs1.finite-Basis vs1.dim-eq-card[of B1 UNIV] sf*

by (*auto simp: vs1.span-Basis vs1.independent-Basis eq*

simp del: vs2.dim-UNIV

intro!: card-image-le)

have *indep-fB: vs2.independent (f ‘ B1)*

using *vs1.finite-Basis vs1.dim-eq-card[of B1 UNIV] sf **

by (*intro vs2.card-le-dim-spanning[of f ‘ B1 UNIV]*) (*auto simp: span-image vs1.span-Basis*)

have *vs2.dim UNIV \leq card (f ‘ B1)*

unfolding *eq sf[symmetric] vs2.dim-span-eq-card-independent[symmetric, OF indep-fB]*

vs2.dim-span

by (*intro vs2.dim-mono*) (*auto simp: span-image vs1.span-Basis*)

with * **have** *card (f ‘ B1) = vs2.dim UNIV by auto*

also have *... = card B1*

unfolding *eq vs1.dim-UNIV ..*

finally have *inj-on f B1*

by (*subst inj-on-iff-eq-card[OF vs1.finite-Basis]*)

then show *inj f*

using *inj-on-span-iff-independent-image[OF indep-fB] vs1.span-Basis by auto*

qed

lemma *linear-injective-imp-surjective*:

assumes *lf: linear s1 s2 f and sf: inj f and eq: vs2.dim UNIV = vs1.dim UNIV*

shows *surj f*

proof –

interpret *linear s1 s2 f by fact*

```

have *: False if  $b \notin vs2.span (f \text{ ` } B1)$  for  $b$ 
proof –
  have *:  $vs2.independent (f \text{ ` } B1)$ 
  using  $vs1.independent-Basis$  by (intro independent-injective-image inj-on-subset[OF sf]) auto
  have **:  $vs2.independent (insert\ b\ (f \text{ ` } B1))$ 
  using  $b \text{ ` }$  by (rule  $vs2.independent-insertI$ )

  have  $b \notin f \text{ ` } B1$  using  $vs2.span-base[of\ b\ f \text{ ` } B1]$   $b$  by auto
  then have  $Suc (card\ B1) = card (insert\ b\ (f \text{ ` } B1))$ 
    using  $sf[THEN\ inj-on-subset, of\ B1]$  by (subst card.insert-remove) (auto intro: vs1.finite-Basis simp: card-image)
  also have  $\dots = vs2.dim (insert\ b\ (f \text{ ` } B1))$ 
    using  $vs2.dim-eq-card-independent[OF\ **]$  by simp
  also have  $vs2.dim (insert\ b\ (f \text{ ` } B1)) \leq vs2.dim\ B2$ 
    by (rule  $vs2.dim-mono$ ) (auto simp: vs2.span-Basis)
  also have  $\dots = card\ B1$ 
    using  $vs1.dim-span[of\ B1]$   $vs2.dim-span[of\ B2]$  unfolding  $vs1.span-Basis\ vs2.span-Basis\ eq$ 
     $vs1.dim-eq-card-independent[OF\ vs1.independent-Basis]$  by simp
  finally show False by simp
qed
have  $f \text{ ` } UNIV = f \text{ ` } vs1.span\ B1$  unfolding  $vs1.span-Basis \dots$ 
also have  $\dots = vs2.span (f \text{ ` } B1)$  unfolding span-image  $\dots$ 
also have  $\dots = UNIV$  using * by blast
finally show ?thesis .
qed

```

lemma *linear-injective-isomorphism:*

```

assumes  $lf: linear\ s1\ s2\ f$ 
  and  $fi: inj\ f$ 
  and  $dims: vs2.dim\ UNIV = vs1.dim\ UNIV$ 
shows  $\exists f'. linear\ s2\ s1\ f' \wedge (\forall x. f' (f\ x) = x) \wedge (\forall x. f (f'\ x) = x)$ 
unfolding isomorphism-expand[symmetric]
using  $linear-injective-imp-surjective[OF\ lf\ fi\ dims]$ 
using  $fi$  left-right-inverse-eq lf linear-injective-left-inverse linear-surjective-right-inverse
by blast

```

lemma *linear-surjective-isomorphism:*

```

assumes  $lf: linear\ s1\ s2\ f$ 
  and  $sf: surj\ f$ 
  and  $dims: vs2.dim\ UNIV = vs1.dim\ UNIV$ 
shows  $\exists f'. linear\ s2\ s1\ f' \wedge (\forall x. f' (f\ x) = x) \wedge (\forall x. f (f'\ x) = x)$ 
using  $linear-surjective-imp-injective[OF\ lf\ sf\ dims]$   $sf$ 
   $linear-exists-right-inverse-on[OF\ lf\ vs1.subspace-UNIV]$ 
   $linear-exists-left-inverse-on[OF\ lf\ vs1.subspace-UNIV]$ 
   $dims\ lf$  linear-injective-isomorphism by auto

```

lemma *basis-to-basis-subspace-isomorphism:*


```

assumes s: vs1.subspace S
  and t: vs2.subspace T
  and d: vs1.dim S = vs2.dim T
  and B:  $B \subseteq S$  vs1.independent B  $S \subseteq vs1.span B$  card B = vs1.dim S
  and C:  $C \subseteq T$  vs2.independent C  $T \subseteq vs2.span C$  card C = vs2.dim T
shows  $\exists f. linear\ s1\ s2\ f \wedge f' B = C \wedge f' S = T \wedge inj\text{-on}\ f\ S$ 
proof –
  from B have fB: finite B
    by (simp add: vs1.finiteI-independent)
  from C have fC: finite C
    by (simp add: vs2.finiteI-independent)
  from finite-basis-to-basis-subspace-isomorphism[OF s t d fB B fC C] show ?thesis
  .
qed

end

```

```

context finite-dimensional-vector-space begin

```

```

lemma linear-surj-imp-inj:

```

```

  assumes lf: linear scale scale f and sf: surj f
  shows inj f

```

```

proof –

```

```

  interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales

```

```

  let ?U = UNIV :: 'b set

```

```

  from basis-exists[of ?U] obtain B

```

```

    where B:  $B \subseteq ?U$  independent B  $?U \subseteq span B$  and d: card B = dim ?U
    by blast

```

```

  {

```

```

    fix x

```

```

    assume x:  $x \in span B$  and fx:  $f x = 0$ 

```

```

    from B(2) have fB: finite B

```

```

      using finiteI-independent by auto

```

```

    have Uspan:  $UNIV \subseteq span (f' B)$ 

```

```

      by (simp add: B(3) lf linear-spanning-surjective-image sf)

```

```

    have fBi: independent (f' B)

```

```

    proof (rule card-le-dim-spanning)

```

```

      show  $card (f' B) \leq dim ?U$ 

```

```

        using card-image-le d fB by fastforce

```

```

    qed (use fB Uspan in auto)

```

```

    have th0:  $dim ?U \leq card (f' B)$ 

```

```

      by (rule span-card-ge-dim) (use Uspan fB in auto)

```

```

    moreover have  $card (f' B) \leq card B$ 

```

```

      by (rule card-image-le, rule fB)

```

```

    ultimately have th1:  $card B = card (f' B)$ 

```

```

      unfolding d by arith

```

```

    have fiB: inj-on f B

```

```

      by (simp add: eq-card-imp-inj-on fB th1)

```

```

from linear-indep-image-lemma[OF lf fB fBi fiB x] fx
have  $x = 0$  by blast
}
then show ?thesis
  unfolding linear-inj-iff-eq-0[OF lf] using B(3) by blast
qed

```

```

lemma linear-inverse-left:
  assumes lf: linear scale scale f
    and lf': linear scale scale f'
  shows  $f \circ f' = id \longleftrightarrow f' \circ f = id$ 
proof –
  {
    fix ff':: 'b  $\Rightarrow$  'b
    assume lf: linear scale scale f linear scale scale f'
    assume f: f  $\circ$  f' = id
    from f have sf: surj f
      by (auto simp add: o-def id-def surj-def) metis
    interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales
    from linear-surjective-isomorphism[OF lf(1) sf] lf f
    have  $f' \circ f = id$ 
      unfolding fun-eq-iff o-def id-def by metis
  }
then show ?thesis
  using lf lf' by metis
qed

```

```

lemma left-inverse-linear:
  assumes lf: linear scale scale f
    and gf: g  $\circ$  f = id
  shows linear scale scale g
proof –
  from gf have fi: inj f
    by (auto simp add: inj-on-def o-def id-def fun-eq-iff) metis
  interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales
  from linear-injective-isomorphism[OF lf fi]
  obtain h :: 'b  $\Rightarrow$  'b where linear scale scale h and  $h: \forall x. h (f x) = x \ \forall x. f (h x) = x$ 
    by blast
  have  $h = g$ 
    by (metis gf h isomorphism-expand left-right-inverse-eq)
  with  $\langle$ linear scale scale h $\rangle$  show ?thesis by blast
qed

```

```

lemma inj-linear-imp-inv-linear:
  assumes linear scale scale f inj f shows linear scale scale (inv f)
  using assms inj-iff left-inverse-linear by blast

```

lemma *right-inverse-linear*:

assumes *lf*: linear scale scale *f*

and *gf*: $f \circ g = id$

shows linear scale scale *g*

proof –

from *gf* **have** *fi*: surj *f*

by (*auto simp add: surj-def o-def id-def*) *metis*

interpret *finite-dimensional-vector-space-pair* scale *Basis* scale *Basis* **by** *unfold-locales*

from *linear-surjective-isomorphism*[*OF lf fi*]

obtain *h*:: 'b \Rightarrow 'b **where** *h*: linear scale scale $h \forall x. h (f x) = x \forall x. f (h x) = x$

by *blast*

then have $h = g$

by (*metis gf isomorphism-expand left-right-inverse-eq*)

with $h(1)$ **show** *?thesis* **by** *blast*

qed

end

context *finite-dimensional-vector-space-pair* **begin**

lemma *subspace-isomorphism*:

assumes *s*: *vs1.subspace* *S*

and *t*: *vs2.subspace* *T*

and *d*: *vs1.dim* *S* = *vs2.dim* *T*

shows $\exists f. \text{linear } s1 \ s2 \ f \wedge f ' S = T \wedge \text{inj-on } f \ S$

proof –

from *vs1.basis-exists*[*of S*] *vs1.finiteI-independent*

obtain *B* **where** *B*: $B \subseteq S$ *vs1.independent* *B* $S \subseteq \text{vs1.span } B$ $\text{card } B = \text{vs1.dim } S$ **and** *fB*: *finite* *B*

by *metis*

from *vs2.basis-exists*[*of T*] *vs2.finiteI-independent*

obtain *C* **where** *C*: $C \subseteq T$ *vs2.independent* *C* $T \subseteq \text{vs2.span } C$ $\text{card } C = \text{vs2.dim } T$ **and** *fC*: *finite* *C*

by *metis*

from *B(4)* *C(4)* *card-le-inj*[*of B C*] *d*

obtain *f* **where** *f*: $f ' B \subseteq C$ *inj-on* *f B* **using** $\langle \text{finite } B \rangle \langle \text{finite } C \rangle$

by *auto*

from *linear-independent-extend*[*OF B(2)*]

obtain *g* **where** *g*: linear *s1 s2 g* $\forall x \in B. g x = f x$

by *blast*

interpret *g*: linear *s1 s2 g* **by** *fact*

from *inj-on-iff-eq-card*[*OF fB, of f*] *f(2)* **have** $\text{card } (f ' B) = \text{card } B$

by *simp*

with *B(4)* *C(4)* **have** *ceq*: $\text{card } (f ' B) = \text{card } C$

using *d* **by** *simp*

have $g ' B = f ' B$

using *g(2)* **by** (*auto simp add: image-iff*)

```

also have ... = C using card-subset-eq[OF fC f(1) ceq] .
finally have gBC: g ' B = C .
have gi: inj-on g B
  using f(2) g(2) by (auto simp add: inj-on-def)
note g0 = linear-indep-image-lemma[OF g(1) fB, unfolded gBC, OF C(2) gi]
{
  fix x y
  assume x: x ∈ S and y: y ∈ S and gxy: g x = g y
  from B(3) x y have x': x ∈ vs1.span B and y': y ∈ vs1.span B
    by blast+
  from gxy have th0: g (x - y) = 0
    by (simp add: linear-diff g)
  have th1: x - y ∈ vs1.span B
    using x' y' by (metis vs1.span-diff)
  have x = y
    using g0[OF th1 th0] by simp
}
then have giS: inj-on g S
  unfolding inj-on-def by blast
from vs1.span-subspace[OF B(1,3) s] have g ' S = vs2.span (g ' B)
  by (simp add: module-hom.span-image[OF g(1)][unfolded linear-iff-module-hom])
also have ... = vs2.span C unfolding gBC ..
also have ... = T using vs2.span-subspace[OF C(1,3) t] .
finally have gS: g ' S = T .
from g(1) gS giS show ?thesis
  by blast
qed

end

hide-const (open) linear

end

```

108 Vector Spaces and Algebras over the Reals

```

theory Real-Vector-Spaces
imports Real Topological-Spaces Vector-Spaces
begin

```

108.1 Real vector spaces

```

class scaleR =
  fixes scaleR :: real ⇒ 'a ⇒ 'a (infixr *_R 75)
begin

abbreviation divideR :: 'a ⇒ real ⇒ 'a (infixl '/_R 70)
  where x /_R r ≡ inverse r *_R x

```

end

class *real-vector* = *scaleR* + *ab-group-add* +
assumes *scaleR-add-right*: $a *_R (x + y) = a *_R x + a *_R y$
and *scaleR-add-left*: $(a + b) *_R x = a *_R x + b *_R x$
and *scaleR-scaleR*: $a *_R b *_R x = (a * b) *_R x$
and *scaleR-one*: $1 *_R x = x$

class *real-algebra* = *real-vector* + *ring* +
assumes *mult-scaleR-left* [*simp*]: $a *_R x * y = a *_R (x * y)$
and *mult-scaleR-right* [*simp*]: $x * a *_R y = a *_R (x * y)$

class *real-algebra-1* = *real-algebra* + *ring-1*

class *real-div-algebra* = *real-algebra-1* + *division-ring*

class *real-field* = *real-div-algebra* + *field*

instantiation *real* :: *real-field*
begin

definition *real-scaleR-def* [*simp*]: *scaleR* $a x = a * x$

instance
by *standard* (*simp-all add: algebra-simps*)

end

locale *linear* = *Vector-Spaces.linear scaleR::=>=>'a::real-vector scaleR::=>=>'b::real-vector*
begin

lemmas *scaleR* = *scale*

end

global-interpretation *real-vector?*: *vector-space scaleR* :: $real \Rightarrow 'a \Rightarrow 'a :: real-vector$

rewrites *Vector-Spaces.linear* $(*_R) (*_R) = linear$

and *Vector-Spaces.linear* $(*) (*_R) = linear$

defines *dependent-raw-def*: *dependent* = *real-vector.dependent*

and *representation-raw-def*: *representation* = *real-vector.representation*

and *subspace-raw-def*: *subspace* = *real-vector.subspace*

and *span-raw-def*: *span* = *real-vector.span*

and *extend-basis-raw-def*: *extend-basis* = *real-vector.extend-basis*

and *dim-raw-def*: *dim* = *real-vector.dim*

proof *unfold-locales*

show *Vector-Spaces.linear* $(*_R) (*_R) = linear$ *Vector-Spaces.linear* $(*) (*_R) = linear$

by (*force simp: linear-def real-scaleR-def[abs-def]*)+

qed (*use scaleR-add-right scaleR-add-left scaleR-scaleR scaleR-one in auto*)

hide-const (open)— locale constants

real-vector.dependent
real-vector.independent
real-vector.representation
real-vector.subspace
real-vector.span
real-vector.extend-basis
real-vector.dim

abbreviation *independent* $x \equiv \neg$ *dependent* x

global-interpretation *real-vector?*: *vector-space-pair* *scaleR*:: \Rightarrow \Rightarrow '*a*::*real-vector*

scaleR:: \Rightarrow \Rightarrow '*b*::*real-vector*

rewrites *Vector-Spaces.linear* $(*_R) (*_R) = \text{linear}$

and *Vector-Spaces.linear* $(*) (*_R) = \text{linear}$

defines *construct-raw-def*: *construct* = *real-vector.construct*

proof *unfold-locales*

show *Vector-Spaces.linear* $(*) (*_R) = \text{linear}$

unfolding *linear-def* *real-scaleR-def* **by** *auto*

qed (*auto simp: linear-def*)

hide-const (open)— locale constants

real-vector.construct

lemma *linear-compose*: *linear* $f \implies$ *linear* $g \implies$ *linear* $(g \circ f)$

unfolding *linear-def* **by** (*rule* *Vector-Spaces.linear-compose*)

Recover original theorem names

lemmas *scaleR-left-commute* = *real-vector.scale-left-commute*

lemmas *scaleR-zero-left* = *real-vector.scale-zero-left*

lemmas *scaleR-minus-left* = *real-vector.scale-minus-left*

lemmas *scaleR-diff-left* = *real-vector.scale-left-diff-distrib*

lemmas *scaleR-sum-left* = *real-vector.scale-sum-left*

lemmas *scaleR-zero-right* = *real-vector.scale-zero-right*

lemmas *scaleR-minus-right* = *real-vector.scale-minus-right*

lemmas *scaleR-diff-right* = *real-vector.scale-right-diff-distrib*

lemmas *scaleR-sum-right* = *real-vector.scale-sum-right*

lemmas *scaleR-eq-0-iff* = *real-vector.scale-eq-0-iff*

lemmas *scaleR-left-imp-eq* = *real-vector.scale-left-imp-eq*

lemmas *scaleR-right-imp-eq* = *real-vector.scale-right-imp-eq*

lemmas *scaleR-cancel-left* = *real-vector.scale-cancel-left*

lemmas *scaleR-cancel-right* = *real-vector.scale-cancel-right*

lemma [*field-simps*]:

$c \neq 0 \implies a = b /_R c \iff c *_R a = b$

$c \neq 0 \implies b /_R c = a \iff b = c *_R a$

$c \neq 0 \implies a + b /_R c = (c *_R a + b) /_R c$

$c \neq 0 \implies a /_R c + b = (a + c *_R b) /_R c$

$$c \neq 0 \implies a - b /_R c = (c *_R a - b) /_R c$$

$$c \neq 0 \implies a /_R c - b = (a - c *_R b) /_R c$$

$$c \neq 0 \implies -(a /_R c) + b = (-a + c *_R b) /_R c$$

$$c \neq 0 \implies -(a /_R c) - b = (-a - c *_R b) /_R c$$

for $a\ b :: 'a :: \text{real-vector}$
by (*auto simp add: scaleR-add-right scaleR-add-left scaleR-diff-right scaleR-diff-left*)

Legacy names

lemmas *scaleR-left-distrib = scaleR-add-left*

lemmas *scaleR-right-distrib = scaleR-add-right*

lemmas *scaleR-left-diff-distrib = scaleR-diff-left*

lemmas *scaleR-right-diff-distrib = scaleR-diff-right*

lemmas *linear-injective-0 = linear-inj-iff-eq-0*

and *linear-injective-on-subspace-0 = linear-inj-on-iff-eq-0*

and *linear-cmul = linear-scale*

and *linear-scaleR = linear-scale-self*

and *subspace-mul = subspace-scale*

and *span-linear-image = linear-span-image*

and *span-0 = span-zero*

and *span-mul = span-scale*

and *injective-scaleR = injective-scale*

lemma *scaleR-minus1-left [simp]: scaleR (-1) x = - x*

for $x :: 'a :: \text{real-vector}$

using *scaleR-minus-left [of 1 x]* **by** *simp*

lemma *scaleR-2:*

fixes $x :: 'a :: \text{real-vector}$

shows *scaleR 2 x = x + x*

unfolding *one-add-one [symmetric]* *scaleR-left-distrib* **by** *simp*

lemma *scaleR-half-double [simp]:*

fixes $a :: 'a :: \text{real-vector}$

shows $(1 / 2) *_R (a + a) = a$

proof –

have $\bigwedge r. r *_R (a + a) = (r * 2) *_R a$

by (*metis scaleR-2 scaleR-scaleR*)

then show *?thesis*

by *simp*

qed

lemma *shift-zero-ident [simp]:*

fixes $f :: 'a \Rightarrow 'b :: \text{real-vector}$

shows $(+)0 \circ f = f$

by *force*

lemma *linear-scale-real:*

fixes $r :: \text{real}$ **shows** *linear f $\implies f (r * b) = r * f b$*

using *linear-scale* by *fastforce*

interpretation *scaleR-left: additive* ($\lambda a. \text{scaleR } a \ x :: 'a::\text{real-vector}$)
by *standard* (rule *scaleR-left-distrib*)

interpretation *scaleR-right: additive* ($\lambda x. \text{scaleR } a \ x :: 'a::\text{real-vector}$)
by *standard* (rule *scaleR-right-distrib*)

lemma *nonzero-inverse-scaleR-distrib*:
 $a \neq 0 \implies x \neq 0 \implies \text{inverse } (\text{scaleR } a \ x) = \text{scaleR } (\text{inverse } a) (\text{inverse } x)$
for $x :: 'a::\text{real-div-algebra}$
by (rule *inverse-unique*) *simp*

lemma *inverse-scaleR-distrib*: $\text{inverse } (\text{scaleR } a \ x) = \text{scaleR } (\text{inverse } a) (\text{inverse } x)$
for $x :: 'a::\{\text{real-div-algebra}, \text{division-ring}\}$
by (metis *inverse-zero nonzero-inverse-scaleR-distrib scale-eq-0-iff*)

lemmas *sum-constant-scaleR* = *real-vector.sum-constant-scale*— legacy name

named-theorems *vector-add-divide-simps* to simplify sums of scaled vectors

lemma [*vector-add-divide-simps*]:
 $v + (b / z) *_R w = (\text{if } z = 0 \text{ then } v \text{ else } (z *_R v + b *_R w) /_R z)$
 $a *_R v + (b / z) *_R w = (\text{if } z = 0 \text{ then } a *_R v \text{ else } ((a *_R z) *_R v + b *_R w) /_R z)$
 $(a / z) *_R v + w = (\text{if } z = 0 \text{ then } w \text{ else } (a *_R v + z *_R w) /_R z)$
 $(a / z) *_R v + b *_R w = (\text{if } z = 0 \text{ then } b *_R w \text{ else } (a *_R v + (b *_R z) *_R w) /_R z)$
 $v - (b / z) *_R w = (\text{if } z = 0 \text{ then } v \text{ else } (z *_R v - b *_R w) /_R z)$
 $a *_R v - (b / z) *_R w = (\text{if } z = 0 \text{ then } a *_R v \text{ else } ((a *_R z) *_R v - b *_R w) /_R z)$
 $(a / z) *_R v - w = (\text{if } z = 0 \text{ then } -w \text{ else } (a *_R v - z *_R w) /_R z)$
 $(a / z) *_R v - b *_R w = (\text{if } z = 0 \text{ then } -b *_R w \text{ else } (a *_R v - (b *_R z) *_R w) /_R z)$
for $v :: 'a :: \text{real-vector}$
by (*simp-all add: divide-inverse-commute scaleR-add-right scaleR-diff-right*)

lemma *eq-vector-fraction-iff* [*vector-add-divide-simps*]:
fixes $x :: 'a :: \text{real-vector}$
shows $(x = (u / v) *_R a) \longleftrightarrow (\text{if } v=0 \text{ then } x = 0 \text{ else } v *_R x = u *_R a)$
by *auto* (metis (*no-types*) *divide-eq-1-iff divide-inverse-commute scaleR-one scaleR-scaleR*)

lemma *vector-fraction-eq-iff* [*vector-add-divide-simps*]:
fixes $x :: 'a :: \text{real-vector}$
shows $((u / v) *_R a = x) \longleftrightarrow (\text{if } v=0 \text{ then } x = 0 \text{ else } u *_R a = v *_R x)$
by (metis *eq-vector-fraction-iff*)

lemma *real-vector-affinity-eq*:
fixes $x :: 'a :: \text{real-vector}$
assumes $m0: m \neq 0$
shows $m *_R x + c = y \longleftrightarrow x = \text{inverse } m *_R y - (\text{inverse } m *_R c)$
(is ?lhs \longleftrightarrow ?rhs)
proof
assume *?lhs*
then have $m *_R x = y - c$ **by** (*simp add: field-simps*)
then have $\text{inverse } m *_R (m *_R x) = \text{inverse } m *_R (y - c)$ **by** *simp*
then show $x = \text{inverse } m *_R y - (\text{inverse } m *_R c)$
using $m0$
by (*simp add: scaleR-diff-right*)
next
assume *?rhs*
with $m0$ **show** $m *_R x + c = y$
by (*simp add: scaleR-diff-right*)
qed

lemma *real-vector-eq-affinity*: $m \neq 0 \implies y = m *_R x + c \longleftrightarrow \text{inverse } m *_R y - (\text{inverse } m *_R c) = x$
for $x :: 'a :: \text{real-vector}$
using *real-vector-affinity-eq* [**where** $m=m$ **and** $x=x$ **and** $y=y$ **and** $c=c$]
by *metis*

lemma *scaleR-eq-iff* [*simp*]: $b + u *_R a = a + u *_R b \longleftrightarrow a = b \vee u = 1$
for $a :: 'a :: \text{real-vector}$
proof (*cases u = 1*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
have $a = b$ **if** $b + u *_R a = a + u *_R b$
proof –
from *that* **have** $(u - 1) *_R a = (u - 1) *_R b$
by (*simp add: algebra-simps*)
with *False* **show** *?thesis*
by *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *scaleR-collapse* [*simp*]: $(1 - u) *_R a + u *_R a = a$
for $a :: 'a :: \text{real-vector}$
by (*simp add: algebra-simps*)

108.2 Embedding of the Reals into any *real-algebra-1*: *of-real*

definition *of-real* :: $\text{real} \Rightarrow 'a :: \text{real-algebra-1}$
where *of-real* $r = \text{scaleR } r \ 1$

lemma *scaleR-conv-of-real*: $\text{scaleR } r \ x = \text{of-real } r * x$
by (*simp add: of-real-def*)

lemma *of-real-0* [*simp*]: $\text{of-real } 0 = 0$
by (*simp add: of-real-def*)

lemma *of-real-1* [*simp*]: $\text{of-real } 1 = 1$
by (*simp add: of-real-def*)

lemma *of-real-add* [*simp*]: $\text{of-real } (x + y) = \text{of-real } x + \text{of-real } y$
by (*simp add: of-real-def scaleR-left-distrib*)

lemma *of-real-minus* [*simp*]: $\text{of-real } (-x) = - \text{of-real } x$
by (*simp add: of-real-def*)

lemma *of-real-diff* [*simp*]: $\text{of-real } (x - y) = \text{of-real } x - \text{of-real } y$
by (*simp add: of-real-def scaleR-left-diff-distrib*)

lemma *of-real-mult* [*simp*]: $\text{of-real } (x * y) = \text{of-real } x * \text{of-real } y$
by (*simp add: of-real-def*)

lemma *of-real-sum*[*simp*]: $\text{of-real } (\text{sum } f \ s) = (\sum x \in s. \text{of-real } (f \ x))$
by (*induct s rule: infinite-finite-induct*) *auto*

lemma *of-real-prod*[*simp*]: $\text{of-real } (\text{prod } f \ s) = (\prod x \in s. \text{of-real } (f \ x))$
by (*induct s rule: infinite-finite-induct*) *auto*

lemma *nonzero-of-real-inverse*:
 $x \neq 0 \implies \text{of-real } (\text{inverse } x) = \text{inverse } (\text{of-real } x :: 'a::\text{real-div-algebra})$
by (*simp add: of-real-def nonzero-inverse-scaleR-distrib*)

lemma *of-real-inverse* [*simp*]:
 $\text{of-real } (\text{inverse } x) = \text{inverse } (\text{of-real } x :: 'a::\{\text{real-div-algebra}, \text{division-ring}\})$
by (*simp add: of-real-def inverse-scaleR-distrib*)

lemma *nonzero-of-real-divide*:
 $y \neq 0 \implies \text{of-real } (x / y) = (\text{of-real } x / \text{of-real } y :: 'a::\text{real-field})$
by (*simp add: divide-inverse nonzero-of-real-inverse*)

lemma *of-real-divide* [*simp*]:
 $\text{of-real } (x / y) = (\text{of-real } x / \text{of-real } y :: 'a::\text{real-div-algebra})$
by (*simp add: divide-inverse*)

lemma *of-real-power* [*simp*]:
 $\text{of-real } (x \wedge n) = (\text{of-real } x :: 'a::\{\text{real-algebra-1}\}) \wedge n$
by (*induct n simp-all*)

lemma *of-real-power-int* [*simp*]:

$of\text{-}real (power\text{-}int\ x\ n) = power\text{-}int (of\text{-}real\ x :: 'a :: \{real\text{-}div\text{-}algebra, division\text{-}ring\})$
 n

by (*auto simp: power-int-def*)

lemma *of-real-eq-iff* [*simp*]: $of\text{-}real\ x = of\text{-}real\ y \longleftrightarrow x = y$

by (*simp add: of-real-def*)

lemma *inj-of-real*: $inj\ of\text{-}real$

by (*auto intro: injI*)

lemmas *of-real-eq-0-iff* [*simp*] = *of-real-eq-iff* [*of - 0, simplified*]

lemmas *of-real-eq-1-iff* [*simp*] = *of-real-eq-iff* [*of - 1, simplified*]

lemma *minus-of-real-eq-of-real-iff* [*simp*]: $-of\text{-}real\ x = of\text{-}real\ y \longleftrightarrow -x = y$

using *of-real-eq-iff*[*of -x y*] **by** (*simp only: of-real-minus*)

lemma *of-real-eq-minus-of-real-iff* [*simp*]: $of\text{-}real\ x = -of\text{-}real\ y \longleftrightarrow x = -y$

using *of-real-eq-iff*[*of x -y*] **by** (*simp only: of-real-minus*)

lemma *of-real-eq-id* [*simp*]: $of\text{-}real = (id :: real \Rightarrow real)$

by (*rule ext*) (*simp add: of-real-def*)

Collapse nested embeddings.

lemma *of-real-of-nat-eq* [*simp*]: $of\text{-}real (of\text{-}nat\ n) = of\text{-}nat\ n$

by (*induct n*) *auto*

lemma *of-real-of-int-eq* [*simp*]: $of\text{-}real (of\text{-}int\ z) = of\text{-}int\ z$

by (*cases z rule: int-diff-cases*) *simp*

lemma *of-real-numeral* [*simp*]: $of\text{-}real (numeral\ w) = numeral\ w$

using *of-real-of-int-eq* [*of numeral w*] **by** *simp*

lemma *of-real-neg-numeral* [*simp*]: $of\text{-}real (- numeral\ w) = - numeral\ w$

using *of-real-of-int-eq* [*of - numeral w*] **by** *simp*

lemma *numeral-power-int-eq-of-real-cancel-iff* [*simp*]:

$power\text{-}int (numeral\ x)\ n = (of\text{-}real\ y :: 'a :: \{real\text{-}div\text{-}algebra, division\text{-}ring\}) \longleftrightarrow$

$power\text{-}int (numeral\ x)\ n = y$

proof –

have $power\text{-}int (numeral\ x)\ n = (of\text{-}real (power\text{-}int (numeral\ x)\ n) :: 'a)$

by *simp*

also have $\dots = of\text{-}real\ y \longleftrightarrow power\text{-}int (numeral\ x)\ n = y$

by (*subst of-real-eq-iff*) *auto*

finally show *?thesis* .

qed

lemma *of-real-eq-numeral-power-int-cancel-iff* [*simp*]:

$(of\text{-}real\ y :: 'a :: \{real\text{-}div\text{-}algebra, division\text{-}ring\}) = power\text{-}int (numeral\ x)\ n \longleftrightarrow$

$y = power\text{-}int (numeral\ x)\ n$

by (subst (1 2) eq-commute) simp

lemma *of-real-eq-of-real-power-int-cancel-iff* [simp]:
 $\text{power-int (of-real } b :: 'a :: \{\text{real-div-algebra, division-ring}\}) w = \text{of-real } x \longleftrightarrow$
 $\text{power-int } b w = x$
 by (metis *of-real-power-int of-real-eq-iff*)

lemma *of-real-in-Ints-iff* [simp]: $\text{of-real } x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$

proof safe

fix x **assume** $(\text{of-real } x :: 'a) \in \mathbb{Z}$
then obtain n **where** $(\text{of-real } x :: 'a) = \text{of-int } n$
 by (auto simp: *Ints-def*)
also have $\text{of-int } n = \text{of-real (real-of-int } n)$
 by simp
finally have $x = \text{real-of-int } n$
 by (subst (asm) *of-real-eq-iff*)
thus $x \in \mathbb{Z}$
 by auto
qed (auto simp: *Ints-def*)

lemma *Ints-of-real* [intro]: $x \in \mathbb{Z} \implies \text{of-real } x \in \mathbb{Z}$
 by simp

Every real algebra has characteristic zero.

instance *real-algebra-1* < *ring-char-0*

proof

from *inj-of-real inj-of-nat* **have** $\text{inj (of-real } \circ \text{ of-nat)}$
 by (rule *inj-compose*)
then show $\text{inj (of-nat } :: \text{nat} \Rightarrow 'a)$
 by (simp add: *comp-def*)

qed

lemma *fraction-scaleR-times* [simp]:

fixes $a :: 'a :: \text{real-algebra-1}$
shows $(\text{numeral } u / \text{numeral } v) *_{\mathbb{R}} (\text{numeral } w * a) = (\text{numeral } u * \text{numeral } w$
 $/ \text{numeral } v) *_{\mathbb{R}} a$
 by (metis (no-types, lifting) *of-real-numeral scaleR-conv-of-real scaleR-scaleR times-divide-eq-left*)

lemma *inverse-scaleR-times* [simp]:

fixes $a :: 'a :: \text{real-algebra-1}$
shows $(1 / \text{numeral } v) *_{\mathbb{R}} (\text{numeral } w * a) = (\text{numeral } w / \text{numeral } v) *_{\mathbb{R}} a$
 by (metis *divide-inverse-commute inverse-eq-divide of-real-numeral scaleR-conv-of-real scaleR-scaleR*)

lemma *scaleR-times* [simp]:

fixes $a :: 'a :: \text{real-algebra-1}$
shows $(\text{numeral } u) *_{\mathbb{R}} (\text{numeral } w * a) = (\text{numeral } u * \text{numeral } w) *_{\mathbb{R}} a$
 by (simp add: *scaleR-conv-of-real*)

instance *real-field* < *field-char-0* ..

108.3 The Set of Real Numbers

definition *Reals* :: 'a::real-algebra-1 set (\mathbb{R})
where \mathbb{R} = *range of-real*

lemma *Reals-of-real* [*simp*]: *of-real* $r \in \mathbb{R}$
by (*simp add: Reals-def*)

lemma *Reals-of-int* [*simp*]: *of-int* $z \in \mathbb{R}$
by (*subst of-real-of-int-eq [symmetric], rule Reals-of-real*)

lemma *Reals-of-nat* [*simp*]: *of-nat* $n \in \mathbb{R}$
by (*subst of-real-of-nat-eq [symmetric], rule Reals-of-real*)

lemma *Reals-numeral* [*simp*]: *numeral* $w \in \mathbb{R}$
by (*subst of-real-numeral [symmetric], rule Reals-of-real*)

lemma *Reals-0* [*simp*]: $0 \in \mathbb{R}$ **and** *Reals-1* [*simp*]: $1 \in \mathbb{R}$
by (*simp-all add: Reals-def*)

lemma *Reals-add* [*simp*]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a + b \in \mathbb{R}$
by (*metis (no-types, opaque-lifting) Reals-def Reals-of-real imageE of-real-add*)

lemma *Reals-minus* [*simp*]: $a \in \mathbb{R} \implies - a \in \mathbb{R}$
by (*auto simp: Reals-def*)

lemma *Reals-minus-iff* [*simp*]: $- a \in \mathbb{R} \iff a \in \mathbb{R}$
using *Reals-minus* **by** *fastforce*

lemma *Reals-diff* [*simp*]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a - b \in \mathbb{R}$
by (*metis Reals-add Reals-minus-iff add-uminus-conv-diff*)

lemma *Reals-mult* [*simp*]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a * b \in \mathbb{R}$
by (*metis (no-types, lifting) Reals-def Reals-of-real imageE of-real-mult*)

lemma *nonzero-Reals-inverse*: $a \in \mathbb{R} \implies a \neq 0 \implies \text{inverse } a \in \mathbb{R}$
for a :: 'a::real-div-algebra
by (*metis Reals-def Reals-of-real imageE of-real-inverse*)

lemma *Reals-inverse*: $a \in \mathbb{R} \implies \text{inverse } a \in \mathbb{R}$
for a :: 'a::{real-div-algebra, division-ring}
using *nonzero-Reals-inverse* **by** *fastforce*

lemma *Reals-inverse-iff* [*simp*]: $\text{inverse } x \in \mathbb{R} \iff x \in \mathbb{R}$
for x :: 'a::{real-div-algebra, division-ring}
by (*metis Reals-inverse inverse-inverse-eq*)

lemma *nonzero-Reals-divide*: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies b \neq 0 \implies a / b \in \mathbb{R}$
for $a\ b :: 'a::\text{real-field}$
by (*simp add: divide-inverse*)

lemma *Reals-divide* [*simp*]: $a \in \mathbb{R} \implies b \in \mathbb{R} \implies a / b \in \mathbb{R}$
for $a\ b :: 'a::\{\text{real-field,field}\}$
using *nonzero-Reals-divide* **by** *fastforce*

lemma *Reals-power* [*simp*]: $a \in \mathbb{R} \implies a \wedge n \in \mathbb{R}$
for $a :: 'a::\text{real-algebra-1}$
by (*metis Reals-def Reals-of-real imageE of-real-power*)

lemma *Reals-cases* [*cases set: Reals*]:
assumes $q \in \mathbb{R}$
obtains (*of-real*) r **where** $q = \text{of-real } r$
unfolding *Reals-def*
proof –
from $\langle q \in \mathbb{R} \rangle$ **have** $q \in \text{range of-real}$ **unfolding** *Reals-def* .
then obtain r **where** $q = \text{of-real } r$..
then show *thesis* ..
qed

lemma *sum-in-Reals* [*intro,simp*]: $(\bigwedge i. i \in s \implies f\ i \in \mathbb{R}) \implies \text{sum } f\ s \in \mathbb{R}$
proof (*induct s rule: infinite-finite-induct*)
case *infinite*
then show ?*case* **by** (*metis Reals-0 sum.infinite*)
qed *simp-all*

lemma *prod-in-Reals* [*intro,simp*]: $(\bigwedge i. i \in s \implies f\ i \in \mathbb{R}) \implies \text{prod } f\ s \in \mathbb{R}$
proof (*induct s rule: infinite-finite-induct*)
case *infinite*
then show ?*case* **by** (*metis Reals-1 prod.infinite*)
qed *simp-all*

lemma *Reals-induct* [*case-names of-real, induct set: Reals*]:
 $q \in \mathbb{R} \implies (\bigwedge r. P (\text{of-real } r)) \implies P\ q$
by (*rule Reals-cases*) *auto*

108.4 Ordered real vector spaces

class *ordered-real-vector* = *real-vector* + *ordered-ab-group-add* +
assumes *scaleR-left-mono*: $x \leq y \implies 0 \leq a \implies a *_{\mathbb{R}} x \leq a *_{\mathbb{R}} y$
and *scaleR-right-mono*: $a \leq b \implies 0 \leq x \implies a *_{\mathbb{R}} x \leq b *_{\mathbb{R}} x$
begin

lemma *scaleR-mono*:
 $a \leq b \implies x \leq y \implies 0 \leq b \implies 0 \leq x \implies a *_{\mathbb{R}} x \leq b *_{\mathbb{R}} y$
by (*meson order-trans scaleR-left-mono scaleR-right-mono*)

lemma *scaleR-mono'*:

$a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \implies a *_R c \leq b *_R d$
by (*rule scaleR-mono*) (*auto intro: order.trans*)

lemma *pos-le-divideR-eq* [*field-simps*]:

$a \leq b /_R c \iff c *_R a \leq b$ (**is** $?P \iff ?Q$) **if** $0 < c$

proof

assume $?P$

with *scaleR-left-mono* **that have** $c *_R a \leq c *_R (b /_R c)$

by *simp*

with that show $?Q$

by (*simp add: scaleR-one scaleR-scaleR inverse-eq-divide*)

next

assume $?Q$

with *scaleR-left-mono* **that have** $c *_R a /_R c \leq b /_R c$

by *simp*

with that show $?P$

by (*simp add: scaleR-one scaleR-scaleR inverse-eq-divide*)

qed

lemma *pos-less-divideR-eq* [*field-simps*]:

$a < b /_R c \iff c *_R a < b$ **if** $c > 0$

using that *pos-le-divideR-eq* [*of c a b*]

by (*auto simp add: le-less scaleR-scaleR scaleR-one*)

lemma *pos-divideR-le-eq* [*field-simps*]:

$b /_R c \leq a \iff b \leq c *_R a$ **if** $c > 0$

using that *pos-le-divideR-eq* [*of inverse c b a*] **by** *simp*

lemma *pos-divideR-less-eq* [*field-simps*]:

$b /_R c < a \iff b < c *_R a$ **if** $c > 0$

using that *pos-less-divideR-eq* [*of inverse c b a*] **by** *simp*

lemma *pos-le-minus-divideR-eq* [*field-simps*]:

$a \leq -(b /_R c) \iff c *_R a \leq -b$ **if** $c > 0$

using that by (*metis add-minus-cancel diff-0 left-minus minus-minus neg-le-iff-le scaleR-add-right uminus-add-conv-diff pos-le-divideR-eq*)

lemma *pos-less-minus-divideR-eq* [*field-simps*]:

$a < -(b /_R c) \iff c *_R a < -b$ **if** $c > 0$

using that by (*metis le-less less-le-not-le pos-divideR-le-eq*

pos-divideR-less-eq pos-le-minus-divideR-eq)

lemma *pos-minus-divideR-le-eq* [*field-simps*]:

$-(b /_R c) \leq a \iff -b \leq c *_R a$ **if** $c > 0$

using that by (*metis pos-divideR-le-eq pos-le-minus-divideR-eq that inverse-positive-iff-positive le-imp-neg-le minus-minus*)

lemma *pos-minus-divideR-less-eq* [*field-simps*]:

– $(b /_R c) < a \longleftrightarrow -b < c *_R a$ **if** $c > 0$
using that by (*simp add: less-le-not-le pos-le-minus-divideR-eq pos-minus-divideR-le-eq*)

lemma *scaleR-image-atLeastAtMost*: $c > 0 \implies \text{scaleR } c \text{ ‘ } \{x..y\} = \{c *_R x..c *_R y\}$

apply (*auto intro!: scaleR-left-mono simp: image-iff Bex-def*)
using *pos-divideR-le-eq [of c] pos-le-divideR-eq [of c]*
apply (*meson local.order-eq-iff*)
done

end

lemma *neg-le-divideR-eq [field-simps]*:
 $a \leq b /_R c \longleftrightarrow b \leq c *_R a$ **(is** $?P \longleftrightarrow ?Q$) **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
using that *pos-le-divideR-eq [of - c a - b]* **by** *simp*

lemma *neg-less-divideR-eq [field-simps]*:
 $a < b /_R c \longleftrightarrow b < c *_R a$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
using that *neg-le-divideR-eq [of c a b]* **by** (*auto simp add: le-less*)

lemma *neg-divideR-le-eq [field-simps]*:
 $b /_R c \leq a \longleftrightarrow c *_R a \leq b$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
using that *pos-divideR-le-eq [of - c - b a]* **by** *simp*

lemma *neg-divideR-less-eq [field-simps]*:
 $b /_R c < a \longleftrightarrow c *_R a < b$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
using that *neg-divideR-le-eq [of c b a]* **by** (*auto simp add: le-less*)

lemma *neg-le-minus-divideR-eq [field-simps]*:
 $a \leq - (b /_R c) \longleftrightarrow -b \leq c *_R a$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$
using that *pos-le-minus-divideR-eq [of - c a - b]* **by** (*simp add: minus-le-iff*)

lemma *neg-less-minus-divideR-eq [field-simps]*:
 $a < - (b /_R c) \longleftrightarrow -b < c *_R a$ **if** $c < 0$
for $a \ b :: 'a :: \text{ordered-real-vector}$

proof –
have $*$: $-b = c *_R a \longleftrightarrow b = - (c *_R a)$
by (*metis add.inverse-inverse*)
from *that neg-le-minus-divideR-eq [of c a b]*
show $?thesis$ **by** (*auto simp add: le-less **)

qed

lemma *neg-minus-divideR-le-eq [field-simps]*:

– $(b /_R c) \leq a \iff c *_R a \leq -b$ if $c < 0$
for $a\ b :: 'a :: \text{ordered-real-vector}$
using that *pos-minus-divideR-le-eq* [of $-c - b\ a$] **by** (*simp add: le-minus-iff*)

lemma *neg-minus-divideR-less-eq* [*field-simps*]:

– $(b /_R c) < a \iff c *_R a < -b$ if $c < 0$
for $a\ b :: 'a :: \text{ordered-real-vector}$
using that **by** (*simp add: less-le-not-le neg-le-minus-divideR-eq neg-minus-divideR-le-eq*)

lemma [*field-split-simps*]:

$a = b /_R c \iff (\text{if } c = 0 \text{ then } a = 0 \text{ else } c *_R a = b)$
 $b /_R c = a \iff (\text{if } c = 0 \text{ then } a = 0 \text{ else } b = c *_R a)$
 $a + b /_R c = (\text{if } c = 0 \text{ then } a \text{ else } (c *_R a + b) /_R c)$
 $a /_R c + b = (\text{if } c = 0 \text{ then } b \text{ else } (a + c *_R b) /_R c)$
 $a - b /_R c = (\text{if } c = 0 \text{ then } a \text{ else } (c *_R a - b) /_R c)$
 $a /_R c - b = (\text{if } c = 0 \text{ then } -b \text{ else } (a - c *_R b) /_R c)$
 $-(a /_R c) + b = (\text{if } c = 0 \text{ then } b \text{ else } (-a + c *_R b) /_R c)$
 $-(a /_R c) - b = (\text{if } c = 0 \text{ then } -b \text{ else } (-a - c *_R b) /_R c)$
for $a\ b :: 'a :: \text{real-vector}$
by (*auto simp add: field-simps*)

lemma [*field-split-simps*]:

$0 < c \implies a \leq b /_R c \iff (\text{if } c > 0 \text{ then } c *_R a \leq b \text{ else if } c < 0 \text{ then } b \leq c *_R a \text{ else } a \leq 0)$
 $0 < c \implies a < b /_R c \iff (\text{if } c > 0 \text{ then } c *_R a < b \text{ else if } c < 0 \text{ then } b < c *_R a \text{ else } a < 0)$
 $0 < c \implies b /_R c \leq a \iff (\text{if } c > 0 \text{ then } b \leq c *_R a \text{ else if } c < 0 \text{ then } c *_R a \leq b \text{ else } a \geq 0)$
 $0 < c \implies b /_R c < a \iff (\text{if } c > 0 \text{ then } b < c *_R a \text{ else if } c < 0 \text{ then } c *_R a < b \text{ else } a > 0)$
 $0 < c \implies a \leq -(b /_R c) \iff (\text{if } c > 0 \text{ then } c *_R a \leq -b \text{ else if } c < 0 \text{ then } -b \leq c *_R a \text{ else } a \leq 0)$
 $0 < c \implies a < -(b /_R c) \iff (\text{if } c > 0 \text{ then } c *_R a < -b \text{ else if } c < 0 \text{ then } -b < c *_R a \text{ else } a < 0)$
 $0 < c \implies -(b /_R c) \leq a \iff (\text{if } c > 0 \text{ then } -b \leq c *_R a \text{ else if } c < 0 \text{ then } c *_R a \leq -b \text{ else } a \geq 0)$
 $0 < c \implies -(b /_R c) < a \iff (\text{if } c > 0 \text{ then } -b < c *_R a \text{ else if } c < 0 \text{ then } c *_R a < -b \text{ else } a > 0)$
for $a\ b :: 'a :: \text{ordered-real-vector}$
by (*clarsimp intro!: field-simps*)+

lemma *scaleR-nonneg-nonneg*: $0 \leq a \implies 0 \leq x \implies 0 \leq a *_R x$

for $x :: 'a :: \text{ordered-real-vector}$
using *scaleR-left-mono* [of $0\ x\ a$] **by** *simp*

lemma *scaleR-nonneg-nonpos*: $0 \leq a \implies x \leq 0 \implies a *_R x \leq 0$

for $x :: 'a :: \text{ordered-real-vector}$
using *scaleR-left-mono* [of $x\ 0\ a$] **by** *simp*

lemma *scaleR-nonpos-nonneg*: $a \leq 0 \implies 0 \leq x \implies a *_{\mathbb{R}} x \leq 0$
for $x :: 'a::\text{ordered-real-vector}$
using *scaleR-right-mono* [of a 0 x] **by** *simp*

lemma *split-scaleR-neg-le*: $(0 \leq a \wedge x \leq 0) \vee (a \leq 0 \wedge 0 \leq x) \implies a *_{\mathbb{R}} x \leq 0$
for $x :: 'a::\text{ordered-real-vector}$
by (*auto simp: scaleR-nonneg-nonpos scaleR-nonpos-nonneg*)

lemma *le-add-iff1*: $a *_{\mathbb{R}} e + c \leq b *_{\mathbb{R}} e + d \iff (a - b) *_{\mathbb{R}} e + c \leq d$
for $c \ d \ e :: 'a::\text{ordered-real-vector}$
by (*simp add: algebra-simps*)

lemma *le-add-iff2*: $a *_{\mathbb{R}} e + c \leq b *_{\mathbb{R}} e + d \iff c \leq (b - a) *_{\mathbb{R}} e + d$
for $c \ d \ e :: 'a::\text{ordered-real-vector}$
by (*simp add: algebra-simps*)

lemma *scaleR-left-mono-neg*: $b \leq a \implies c \leq 0 \implies c *_{\mathbb{R}} a \leq c *_{\mathbb{R}} b$
for $a \ b :: 'a::\text{ordered-real-vector}$
by (*drule scaleR-left-mono* [of $-$ $-$ c], *simp-all*)

lemma *scaleR-right-mono-neg*: $b \leq a \implies c \leq 0 \implies a *_{\mathbb{R}} c \leq b *_{\mathbb{R}} c$
for $c :: 'a::\text{ordered-real-vector}$
by (*drule scaleR-right-mono* [of $-$ $-$ c], *simp-all*)

lemma *scaleR-nonpos-nonpos*: $a \leq 0 \implies b \leq 0 \implies 0 \leq a *_{\mathbb{R}} b$
for $b :: 'a::\text{ordered-real-vector}$
using *scaleR-right-mono-neg* [of a 0 b] **by** *simp*

lemma *split-scaleR-pos-le*: $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \implies 0 \leq a *_{\mathbb{R}} b$
for $b :: 'a::\text{ordered-real-vector}$
by (*auto simp: scaleR-nonneg-nonneg scaleR-nonpos-nonpos*)

lemma *zero-le-scaleR-iff*:
fixes $b :: 'a::\text{ordered-real-vector}$
shows $0 \leq a *_{\mathbb{R}} b \iff 0 < a \wedge 0 \leq b \vee a < 0 \wedge b \leq 0 \vee a = 0$
 (*is ?lhs = ?rhs*)
proof (*cases a = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof
assume *?lhs*
from $\langle a \neq 0 \rangle$ **consider** $a > 0 \mid a < 0$ **by** *arith*
then show *?rhs*
proof *cases*
case *1*
with $\langle ?lhs \rangle$ **have** *inverse a *_R 0 ≤ inverse a *_R (a *_R b)*

```

    by (intro scaleR-mono) auto
  with 1 show ?thesis
    by simp
next
  case 2
  with ⟨?lhs⟩ have  $- \text{inverse } a *_{\mathbb{R}} 0 \leq - \text{inverse } a *_{\mathbb{R}} (a *_{\mathbb{R}} b)$ 
    by (intro scaleR-mono) auto
  with 2 show ?thesis
    by simp
qed
next
  assume ?rhs
  then show ?lhs
    by (auto simp: not-le ⟨a ≠ 0⟩ intro!: split-scaleR-pos-le)
qed
qed

lemma scaleR-le-0-iff:  $a *_{\mathbb{R}} b \leq 0 \iff 0 < a \wedge b \leq 0 \vee a < 0 \wedge 0 \leq b \vee a = 0$ 
  for b :: 'a::ordered-real-vector
  by (insert zero-le-scaleR-iff [of -a b]) force

lemma scaleR-le-cancel-left:  $c *_{\mathbb{R}} a \leq c *_{\mathbb{R}} b \iff (0 < c \implies a \leq b) \wedge (c < 0 \implies b \leq a)$ 
  for b :: 'a::ordered-real-vector
  by (auto simp: neq-iff scaleR-left-mono scaleR-left-mono-neg
    dest: scaleR-left-mono[where a=inverse c] scaleR-left-mono-neg[where c=inverse c])

lemma scaleR-le-cancel-left-pos:  $0 < c \implies c *_{\mathbb{R}} a \leq c *_{\mathbb{R}} b \iff a \leq b$ 
  for b :: 'a::ordered-real-vector
  by (auto simp: scaleR-le-cancel-left)

lemma scaleR-le-cancel-left-neg:  $c < 0 \implies c *_{\mathbb{R}} a \leq c *_{\mathbb{R}} b \iff b \leq a$ 
  for b :: 'a::ordered-real-vector
  by (auto simp: scaleR-le-cancel-left)

lemma scaleR-left-le-one-le:  $0 \leq x \implies a \leq 1 \implies a *_{\mathbb{R}} x \leq x$ 
  for x :: 'a::ordered-real-vector and a :: real
  using scaleR-right-mono[of a 1 x] by simp

```

108.5 Real normed vector spaces

```

class dist =
  fixes dist :: 'a ⇒ 'a ⇒ real

class norm =
  fixes norm :: 'a ⇒ real

class sgn-div-norm = scaleR + norm + sgn +

```

```

assumes sgn-div-norm:  $\text{sgn } x = x /_R \text{ norm } x$ 

class dist-norm = dist + norm + minus +
assumes dist-norm:  $\text{dist } x \ y = \text{norm } (x - y)$ 

class uniformity-dist = dist + uniformity +
assumes uniformity-dist:  $\text{uniformity} = (\text{INF } e \in \{0 <..\}. \text{ principal } \{(x, y). \text{ dist } x \ y < e\})$ 
begin

lemma eventually-uniformity-metric:
  eventually P uniformity  $\longleftrightarrow (\exists e > 0. \forall x \ y. \text{ dist } x \ y < e \longrightarrow P \ (x, y))$ 
unfolding uniformity-dist
by (subst eventually-INF-base)
    (auto simp: eventually-principal subset-eq intro: beXI[of - min - -])

end

class real-normed-vector = real-vector + sgn-div-norm + dist-norm + uniformity-dist + open-uniformity +
assumes norm-eq-zero [simp]:  $\text{norm } x = 0 \longleftrightarrow x = 0$ 
and norm-triangle-ineq:  $\text{norm } (x + y) \leq \text{norm } x + \text{norm } y$ 
and norm-scaleR [simp]:  $\text{norm } (\text{scaleR } a \ x) = |a| * \text{norm } x$ 
begin

lemma norm-ge-zero [simp]:  $0 \leq \text{norm } x$ 
proof –
  have  $0 = \text{norm } (x + -1 *_R x)$ 
    using scaleR-add-left[of  $1 -1 \ x$ ] norm-scaleR[of  $0 \ x$ ] by (simp add: scaleR-one)
    also have  $\dots \leq \text{norm } x + \text{norm } (-1 *_R x)$  by (rule norm-triangle-ineq)
    finally show ?thesis by simp
qed

lemma bdd-below-norm-image: bdd-below (norm ‘ A)
by (meson bdd-belowI2 norm-ge-zero)

end

class real-normed-algebra = real-algebra + real-normed-vector +
assumes norm-mult-ineq:  $\text{norm } (x * y) \leq \text{norm } x * \text{norm } y$ 

class real-normed-algebra-1 = real-algebra-1 + real-normed-algebra +
assumes norm-one [simp]:  $\text{norm } 1 = 1$ 

lemma (in real-normed-algebra-1) scaleR-power [simp]:  $(\text{scaleR } x \ y) \wedge^n = \text{scaleR } (x \wedge^n) (y \wedge^n)$ 
by (induct n) (simp-all add: scaleR-one scaleR-scaleR mult-ac)

class real-normed-div-algebra = real-div-algebra + real-normed-vector +

```

```

assumes norm-mult: norm (x * y) = norm x * norm y

class real-normed-field = real-field + real-normed-div-algebra

instance real-normed-div-algebra < real-normed-algebra-1
proof
  show norm (x * y) ≤ norm x * norm y for x y :: 'a
    by (simp add: norm-mult)
next
  have norm (1 * 1::'a) = norm (1::'a) * norm (1::'a)
    by (rule norm-mult)
  then show norm (1::'a) = 1 by simp
qed

context real-normed-vector begin

lemma norm-zero [simp]: norm (0::'a) = 0
  by simp

lemma zero-less-norm-iff [simp]: norm x > 0 ↔ x ≠ 0
  by (simp add: order-less-le)

lemma norm-not-less-zero [simp]: ¬ norm x < 0
  by (simp add: linorder-not-less)

lemma norm-le-zero-iff [simp]: norm x ≤ 0 ↔ x = 0
  by (simp add: order-le-less)

lemma norm-minus-cancel [simp]: norm (- x) = norm x
proof -
  have - 1 *R x = - (1 *R x)
    unfolding add-eq-0-iff2[symmetric] scaleR-add-left[symmetric]
    using norm-eq-zero
    by fastforce
  then have norm (- x) = norm (scaleR (- 1) x)
    by (simp only: scaleR-one)
  also have ... = |- 1| * norm x
    by (rule norm-scaleR)
  finally show ?thesis by simp
qed

lemma norm-minus-commute: norm (a - b) = norm (b - a)
proof -
  have norm (- (b - a)) = norm (b - a)
    by (rule norm-minus-cancel)
  then show ?thesis by simp
qed

lemma dist-add-cancel [simp]: dist (a + b) (a + c) = dist b c

```

by (simp add: dist-norm)

lemma *dist-add-cancel2* [simp]: $\text{dist } (b + a) (c + a) = \text{dist } b c$
by (simp add: dist-norm)

lemma *norm-uminus-minus*: $\text{norm } (-x - y) = \text{norm } (x + y)$
by (subst (2) norm-minus-cancel[symmetric], subst minus-add-distrib) simp

lemma *norm-triangle-ineq2*: $\text{norm } a - \text{norm } b \leq \text{norm } (a - b)$

proof –

have $\text{norm } (a - b + b) \leq \text{norm } (a - b) + \text{norm } b$

by (rule norm-triangle-ineq)

then show ?thesis by simp

qed

lemma *norm-triangle-ineq3*: $|\text{norm } a - \text{norm } b| \leq \text{norm } (a - b)$

proof –

have $\text{norm } a - \text{norm } b \leq \text{norm } (a - b)$

by (simp add: norm-triangle-ineq2)

moreover have $\text{norm } b - \text{norm } a \leq \text{norm } (a - b)$

by (metis norm-minus-commute norm-triangle-ineq2)

ultimately show ?thesis

by (simp add: abs-le-iff)

qed

lemma *norm-triangle-ineq4*: $\text{norm } (a - b) \leq \text{norm } a + \text{norm } b$

proof –

have $\text{norm } (a + -b) \leq \text{norm } a + \text{norm } (-b)$

by (rule norm-triangle-ineq)

then show ?thesis by simp

qed

lemma *norm-triangle-le-diff*: $\text{norm } x + \text{norm } y \leq e \implies \text{norm } (x - y) \leq e$

by (meson norm-triangle-ineq4 order-trans)

lemma *norm-diff-ineq*: $\text{norm } a - \text{norm } b \leq \text{norm } (a + b)$

proof –

have $\text{norm } a - \text{norm } (-b) \leq \text{norm } (a - -b)$

by (rule norm-triangle-ineq2)

then show ?thesis by simp

qed

lemma *norm-triangle-sub*: $\text{norm } x \leq \text{norm } y + \text{norm } (x - y)$

using norm-triangle-ineq[of y $x - y$] by (simp add: field-simps)

lemma *norm-triangle-le*: $\text{norm } x + \text{norm } y \leq e \implies \text{norm } (x + y) \leq e$

by (rule norm-triangle-ineq [THEN order-trans])

lemma *norm-triangle-lt*: $\text{norm } x + \text{norm } y < e \implies \text{norm } (x + y) < e$

by (rule norm-triangle-ineq [THEN le-less-trans])

lemma norm-add-leD: $\text{norm } (a + b) \leq c \implies \text{norm } b \leq \text{norm } a + c$

by (metis ab-semigroup-add-class.add-commute add-commute diff-le-eq norm-diff-ineq order-trans)

lemma norm-diff-triangle-ineq: $\text{norm } ((a + b) - (c + d)) \leq \text{norm } (a - c) + \text{norm } (b - d)$

proof –

have $\text{norm } ((a + b) - (c + d)) = \text{norm } ((a - c) + (b - d))$

by (simp add: algebra-simps)

also have $\dots \leq \text{norm } (a - c) + \text{norm } (b - d)$

by (rule norm-triangle-ineq)

finally show ?thesis .

qed

lemma norm-diff-triangle-le: $\text{norm } (x - z) \leq e1 + e2$

if $\text{norm } (x - y) \leq e1$ $\text{norm } (y - z) \leq e2$

proof –

have $\text{norm } (x - (y + z - y)) \leq \text{norm } (x - y) + \text{norm } (y - z)$

using norm-diff-triangle-ineq that diff-diff-eq2 by presburger

with that show ?thesis by simp

qed

lemma norm-diff-triangle-less: $\text{norm } (x - z) < e1 + e2$

if $\text{norm } (x - y) < e1$ $\text{norm } (y - z) < e2$

proof –

have $\text{norm } (x - z) \leq \text{norm } (x - y) + \text{norm } (y - z)$

by (metis norm-diff-triangle-ineq add-diff-cancel-left' diff-diff-eq2)

with that show ?thesis by auto

qed

lemma norm-triangle-mono:

$\text{norm } a \leq r \implies \text{norm } b \leq s \implies \text{norm } (a + b) \leq r + s$

by (metis (mono-tags) add-mono-thms-linordered-semiring(1) norm-triangle-ineq order.trans)

lemma norm-sum: $\text{norm } (\text{sum } f A) \leq (\sum i \in A. \text{norm } (f i))$

for $f::'b \Rightarrow 'a$

by (induct A rule: infinite-finite-induct) (auto intro: norm-triangle-mono)

lemma sum-norm-le: $\text{norm } (\text{sum } f S) \leq \text{sum } g S$

if $\bigwedge x. x \in S \implies \text{norm } (f x) \leq g x$

for $f::'b \Rightarrow 'a$

by (rule order-trans [OF norm-sum sum-mono]) (simp add: that)

lemma abs-norm-cancel [simp]: $|\text{norm } a| = \text{norm } a$

by (rule abs-of-nonneg [OF norm-ge-zero])

lemma *sum-norm-bound*:

$norm (sum f S) \leq of\text{-nat} (card S) * K$

if $\bigwedge x. x \in S \implies norm (f x) \leq K$

for $f :: 'b \Rightarrow 'a$

using *sum-norm-le*[*OF that*] *sum-constant*[*symmetric*]

by *simp*

lemma *norm-add-less*: $norm x < r \implies norm y < s \implies norm (x + y) < r + s$
by (*rule order-le-less-trans* [*OF norm-triangle-ineq add-strict-mono*])

end

lemma *dist-scaleR* [*simp*]: $dist (x *_R a) (y *_R a) = |x - y| * norm a$
for $a :: 'a::real\text{-normed-vector}$
by (*metis dist-norm norm-scaleR scaleR-left.diff*)

lemma *norm-mult-less*: $norm x < r \implies norm y < s \implies norm (x * y) < r * s$
for $x y :: 'a::real\text{-normed-algebra}$
by (*rule order-le-less-trans* [*OF norm-mult-ineq*]) (*simp add: mult-strict-mono'*)

lemma *norm-of-real* [*simp*]: $norm (of\text{-real} r :: 'a::real\text{-normed-algebra-1}) = |r|$
by (*simp add: of-real-def*)

lemma *norm-numeral* [*simp*]: $norm (numeral w :: 'a::real\text{-normed-algebra-1}) = numeral w$
by (*subst of-real-numeral* [*symmetric*], *subst norm-of-real*, *simp*)

lemma *norm-neg-numeral* [*simp*]: $norm (- numeral w :: 'a::real\text{-normed-algebra-1}) = numeral w$
by (*subst of-real-neg-numeral* [*symmetric*], *subst norm-of-real*, *simp*)

lemma *norm-of-real-add1* [*simp*]: $norm (of\text{-real} x + 1 :: 'a :: real\text{-normed-div-algebra}) = |x + 1|$
by (*metis norm-of-real of-real-1 of-real-add*)

lemma *norm-of-real-addn* [*simp*]:
 $norm (of\text{-real} x + numeral b :: 'a :: real\text{-normed-div-algebra}) = |x + numeral b|$
by (*metis norm-of-real of-real-add of-real-numeral*)

lemma *norm-of-int* [*simp*]: $norm (of\text{-int} z :: 'a::real\text{-normed-algebra-1}) = |of\text{-int} z|$
by (*subst of-real-of-int-eq* [*symmetric*], *rule norm-of-real*)

lemma *norm-of-nat* [*simp*]: $norm (of\text{-nat} n :: 'a::real\text{-normed-algebra-1}) = of\text{-nat} n$
by (*metis abs-of-nat norm-of-real of-real-of-nat-eq*)

lemma *nonzero-norm-inverse*: $a \neq 0 \implies norm (inverse a) = inverse (norm a)$
for $a :: 'a::real\text{-normed-div-algebra}$
by (*metis inverse-unique norm-mult norm-one right-inverse*)

lemma *norm-inverse*: $\text{norm} (\text{inverse } a) = \text{inverse} (\text{norm } a)$
for $a :: 'a::\{\text{real-normed-div-algebra}, \text{division-ring}\}$
by (*metis inverse-zero nonzero-norm-inverse norm-zero*)

lemma *nonzero-norm-divide*: $b \neq 0 \implies \text{norm} (a / b) = \text{norm } a / \text{norm } b$
for $a b :: 'a::\text{real-normed-field}$
by (*simp add: divide-inverse norm-mult nonzero-norm-inverse*)

lemma *norm-divide*: $\text{norm} (a / b) = \text{norm } a / \text{norm } b$
for $a b :: 'a::\{\text{real-normed-field}, \text{field}\}$
by (*simp add: divide-inverse norm-mult norm-inverse*)

lemma *dist-divide-right*: $\text{dist} (a/c) (b/c) = \text{dist } a b / \text{norm } c$ **for** $c :: 'a :: \text{real-normed-field}$
by (*metis diff-divide-distrib dist-norm norm-divide*)

lemma *norm-inverse-le-norm*:
fixes $x :: 'a::\text{real-normed-div-algebra}$
shows $r \leq \text{norm } x \implies 0 < r \implies \text{norm} (\text{inverse } x) \leq \text{inverse } r$
by (*simp add: le-imp-inverse-le norm-inverse*)

lemma *norm-power-ineq*: $\text{norm} (x \wedge n) \leq \text{norm } x \wedge n$
for $x :: 'a::\text{real-normed-algebra-1}$
proof (*induct n*)
case 0
show $\text{norm} (x \wedge 0) \leq \text{norm } x \wedge 0$ **by** *simp*
next
case (*Suc n*)
have $\text{norm} (x * x \wedge n) \leq \text{norm } x * \text{norm} (x \wedge n)$
by (*rule norm-mult-ineq*)
also from *Suc* **have** $\dots \leq \text{norm } x * \text{norm } x \wedge n$
using *norm-ge-zero* **by** (*rule mult-left-mono*)
finally show $\text{norm} (x \wedge \text{Suc } n) \leq \text{norm } x \wedge \text{Suc } n$
by *simp*
qed

lemma *norm-power*: $\text{norm} (x \wedge n) = \text{norm } x \wedge n$
for $x :: 'a::\text{real-normed-div-algebra}$
by (*induct n*) (*simp-all add: norm-mult*)

lemma *norm-power-int*: $\text{norm} (\text{power-int } x n) = \text{power-int} (\text{norm } x) n$
for $x :: 'a::\text{real-normed-div-algebra}$
by (*cases n rule: int-cases4*) (*auto simp: norm-power power-int-minus norm-inverse*)

lemma *power-eq-imp-eq-norm*:
fixes $w :: 'a::\text{real-normed-div-algebra}$
assumes *eq*: $w \wedge n = z \wedge n$ **and** $n > 0$
shows $\text{norm } w = \text{norm } z$
proof –

```

have norm w ^ n = norm z ^ n
  by (metis (no-types) eq norm-power)
then show ?thesis
  using assms by (force intro: power-eq-imp-eq-base)
qed

```

```

lemma power-eq-1-iff:
  fixes w :: 'a::real-normed-div-algebra
  shows w ^ n = 1  $\implies$  norm w = 1  $\vee$  n = 0
  by (metis norm-one power-0-left power-eq-0-iff power-eq-imp-eq-norm power-one)

```

```

lemma norm-mult-numeral1 [simp]: norm (numeral w * a) = numeral w * norm
a
  for a b :: 'a::{real-normed-field,field}
  by (simp add: norm-mult)

```

```

lemma norm-mult-numeral2 [simp]: norm (a * numeral w) = norm a * numeral
w
  for a b :: 'a::{real-normed-field,field}
  by (simp add: norm-mult)

```

```

lemma norm-divide-numeral [simp]: norm (a / numeral w) = norm a / numeral
w
  for a b :: 'a::{real-normed-field,field}
  by (simp add: norm-divide)

```

```

lemma norm-of-real-diff [simp]:
  norm (of-real b - of-real a :: 'a::real-normed-algebra-1)  $\leq$  |b - a|
  by (metis norm-of-real of-real-diff order-refl)

```

Despite a superficial resemblance, *norm-eq-1* is not relevant.

```

lemma square-norm-one:
  fixes x :: 'a::real-normed-div-algebra
  assumes x2 = 1
  shows norm x = 1
  by (metis assms norm-minus-cancel norm-one power2-eq-1-iff)

```

```

lemma norm-less-p1: norm x < norm (of-real (norm x) + 1 :: 'a)
  for x :: 'a::real-normed-algebra-1
proof -
  have norm x < norm (of-real (norm x + 1) :: 'a)
    by (simp add: of-real-def)
  then show ?thesis
    by simp
qed

```

```

lemma prod-norm: prod ( $\lambda$ x. norm (f x)) A = norm (prod f A)
  for f :: 'a  $\Rightarrow$  'b::{comm-semiring-1,real-normed-div-algebra}
  by (induct A rule: infinite-finite-induct) (auto simp: norm-mult)

```

lemma *norm-prod-le*:
 $\text{norm} (\text{prod } f A) \leq (\prod a \in A. \text{norm} (f a :: 'a :: \{\text{real-normed-algebra-1}, \text{comm-monoid-mult}\}))$
proof (*induct A rule: infinite-finite-induct*)
 case *empty*
 then show ?case by *simp*
next
 case (*insert a A*)
 then have $\text{norm} (\text{prod } f (\text{insert } a A)) \leq \text{norm} (f a) * \text{norm} (\text{prod } f A)$
 by (*simp add: norm-mult-ineq*)
 also have $\text{norm} (\text{prod } f A) \leq (\prod a \in A. \text{norm} (f a))$
 by (*rule insert*)
 finally show ?case
 by (*simp add: insert mult-left-mono*)
next
 case *infinite*
 then show ?case by *simp*
qed

lemma *norm-prod-diff*:
 fixes $z w :: 'i \Rightarrow 'a :: \{\text{real-normed-algebra-1}, \text{comm-monoid-mult}\}$
 shows $(\bigwedge i. i \in I \implies \text{norm} (z i) \leq 1) \implies (\bigwedge i. i \in I \implies \text{norm} (w i) \leq 1) \implies$
 $\text{norm} ((\prod i \in I. z i) - (\prod i \in I. w i)) \leq (\sum i \in I. \text{norm} (z i - w i))$
proof (*induction I rule: infinite-finite-induct*)
 case *empty*
 then show ?case by *simp*
next
 case (*insert i I*)
 note *insert.hyps[simp]*

 have $\text{norm} ((\prod i \in \text{insert } i I. z i) - (\prod i \in \text{insert } i I. w i)) =$
 $\text{norm} ((\prod i \in I. z i) * (z i - w i) + ((\prod i \in I. z i) - (\prod i \in I. w i)) * w i)$
 (is - = $\text{norm} (?t1 + ?t2)$)
 by (*auto simp: field-simps*)
 also have $\dots \leq \text{norm} ?t1 + \text{norm} ?t2$
 by (*rule norm-triangle-ineq*)
 also have $\text{norm} ?t1 \leq \text{norm} (\prod i \in I. z i) * \text{norm} (z i - w i)$
 by (*rule norm-mult-ineq*)
 also have $\dots \leq (\prod i \in I. \text{norm} (z i)) * \text{norm} (z i - w i)$
 by (*rule mult-right-mono*) (*auto intro: norm-prod-le*)
 also have $(\prod i \in I. \text{norm} (z i)) \leq (\prod i \in I. 1)$
 by (*intro prod-mono*) (*auto intro!: insert*)
 also have $\text{norm} ?t2 \leq \text{norm} ((\prod i \in I. z i) - (\prod i \in I. w i)) * \text{norm} (w i)$
 by (*rule norm-mult-ineq*)
 also have $\text{norm} (w i) \leq 1$
 by (*auto intro: insert*)
 also have $\text{norm} ((\prod i \in I. z i) - (\prod i \in I. w i)) \leq (\sum i \in I. \text{norm} (z i - w i))$
 using *insert by auto*
 finally show ?case

```

  by (auto simp: ac-simps mult-right-mono mult-left-mono)
next
  case infinite
  then show ?case by simp
qed

```

```

lemma norm-power-diff:
  fixes z w :: 'a::{real-normed-algebra-1, comm-monoid-mult}
  assumes norm z ≤ 1 norm w ≤ 1
  shows norm (zm - wm) ≤ m * norm (z - w)
proof -
  have norm (zm - wm) = norm ((∏ i < m. z) - (∏ i < m. w))
  by simp
  also have ... ≤ (∑ i < m. norm (z - w))
  by (intro norm-prod-diff) (auto simp: assms)
  also have ... = m * norm (z - w)
  by simp
  finally show ?thesis .
qed

```

108.6 Metric spaces

```

class metric-space = uniformity-dist + open-uniformity +
  assumes dist-eq-0-iff [simp]: dist x y = 0 ↔ x = y
  and dist-triangle2: dist x y ≤ dist x z + dist y z
begin

```

```

lemma dist-self [simp]: dist x x = 0
  by simp

```

```

lemma zero-le-dist [simp]: 0 ≤ dist x y
  using dist-triangle2 [of x x y] by simp

```

```

lemma zero-less-dist-iff: 0 < dist x y ↔ x ≠ y
  by (simp add: less-le)

```

```

lemma dist-not-less-zero [simp]: ¬ dist x y < 0
  by (simp add: not-less)

```

```

lemma dist-le-zero-iff [simp]: dist x y ≤ 0 ↔ x = y
  by (simp add: le-less)

```

```

lemma dist-commute: dist x y = dist y x

```

```

proof (rule order-antisym)
  show dist x y ≤ dist y x
    using dist-triangle2 [of x y x] by simp
  show dist y x ≤ dist x y
    using dist-triangle2 [of y x y] by simp
qed

```

lemma *dist-commute-lessI*: $\text{dist } y \ x < e \implies \text{dist } x \ y < e$
by (*simp add: dist-commute*)

lemma *dist-triangle*: $\text{dist } x \ z \leq \text{dist } x \ y + \text{dist } y \ z$
using *dist-triangle2* [*of x z y*] **by** (*simp add: dist-commute*)

lemma *dist-triangle3*: $\text{dist } x \ y \leq \text{dist } a \ x + \text{dist } a \ y$
using *dist-triangle2* [*of x y a*] **by** (*simp add: dist-commute*)

lemma *abs-dist-diff-le*: $|\text{dist } a \ b - \text{dist } b \ c| \leq \text{dist } a \ c$
using *dist-triangle3*[*of b c a*] *dist-triangle2*[*of a b c*] **by** *simp*

lemma *dist-pos-lt*: $x \neq y \implies 0 < \text{dist } x \ y$
by (*simp add: zero-less-dist-iff*)

lemma *dist-nz*: $x \neq y \longleftrightarrow 0 < \text{dist } x \ y$
by (*simp add: zero-less-dist-iff*)

declare *dist-nz* [*symmetric, simp*]

lemma *dist-triangle-le*: $\text{dist } x \ z + \text{dist } y \ z \leq e \implies \text{dist } x \ y \leq e$
by (*rule order-trans* [*OF dist-triangle2*])

lemma *dist-triangle-lt*: $\text{dist } x \ z + \text{dist } y \ z < e \implies \text{dist } x \ y < e$
by (*rule le-less-trans* [*OF dist-triangle2*])

lemma *dist-triangle-less-add*: $\text{dist } x_1 \ y < e_1 \implies \text{dist } x_2 \ y < e_2 \implies \text{dist } x_1 \ x_2 < e_1 + e_2$
by (*rule dist-triangle-lt* [**where** *z=y*]) *simp*

lemma *dist-triangle-half-l*: $\text{dist } x_1 \ y < e / 2 \implies \text{dist } x_2 \ y < e / 2 \implies \text{dist } x_1 \ x_2 < e$
by (*rule dist-triangle-lt* [**where** *z=y*]) *simp*

lemma *dist-triangle-half-r*: $\text{dist } y \ x_1 < e / 2 \implies \text{dist } y \ x_2 < e / 2 \implies \text{dist } x_1 \ x_2 < e$
by (*rule dist-triangle-half-l*) (*simp-all add: dist-commute*)

lemma *dist-triangle-third*:
assumes $\text{dist } x_1 \ x_2 < e/3$ $\text{dist } x_2 \ x_3 < e/3$ $\text{dist } x_3 \ x_4 < e/3$
shows $\text{dist } x_1 \ x_4 < e$
proof –
have $\text{dist } x_1 \ x_3 < e/3 + e/3$
by (*metis assms(1) assms(2) dist-commute dist-triangle-less-add*)
then have $\text{dist } x_1 \ x_4 < (e/3 + e/3) + e/3$
by (*metis assms(3) dist-commute dist-triangle-less-add*)
then show *?thesis*
by *simp*

qed

subclass *uniform-space*

proof

fix $E\ x$

assume *eventually E uniformity*

then obtain e where $E: 0 < e \wedge x\ y. \text{dist } x\ y < e \implies E\ (x, y)$

by (*auto simp: eventually-uniformity-metric*)

then show $E\ (x, x) \forall_F\ (x, y)$ in *uniformity*. $E\ (y, x)$

by (*auto simp: eventually-uniformity-metric dist-commute*)

show $\exists D. \text{eventually } D \text{ uniformity} \wedge (\forall x\ y\ z. D\ (x, y) \longrightarrow D\ (y, z) \longrightarrow E\ (x, z))$

using $E\ \text{dist-triangle-half-l}$ [where $e=e$]

unfolding *eventually-uniformity-metric*

by (*intro exI[of - $\lambda(x, y). \text{dist } x\ y < e / 2$] exI[of - $e/2$] conjI*)
(*auto simp: dist-commute*)

qed

lemma *open-dist*: $\text{open } S \longleftrightarrow (\forall x \in S. \exists e > 0. \forall y. \text{dist } y\ x < e \longrightarrow y \in S)$

by (*simp add: dist-commute open-uniformity eventually-uniformity-metric*)

lemma *open-ball*: $\text{open } \{y. \text{dist } x\ y < d\}$

unfolding *open-dist*

proof (*intro ballI*)

fix y

assume $*$: $y \in \{y. \text{dist } x\ y < d\}$

then show $\exists e > 0. \forall z. \text{dist } z\ y < e \longrightarrow z \in \{y. \text{dist } x\ y < d\}$

by (*auto intro!: exI[of - $d - \text{dist } x\ y$] simp: field-simps dist-triangle-lt*)

qed

subclass *first-countable-topology*

proof

fix x

show $\exists A::\text{nat} \Rightarrow 'a \text{ set. } (\forall i. x \in A\ i \wedge \text{open } (A\ i)) \wedge (\forall S. \text{open } S \wedge x \in S \longrightarrow (\exists i. A\ i \subseteq S))$

proof (*safe intro!: exI[of - $\lambda n. \{y. \text{dist } x\ y < \text{inverse } (Suc\ n)\}$]*)

fix S

assume $\text{open } S\ x \in S$

then obtain e where $e: 0 < e$ and $\{y. \text{dist } x\ y < e\} \subseteq S$

by (*auto simp: open-dist subset-eq dist-commute*)

moreover

from e obtain i where $\text{inverse } (Suc\ i) < e$

by (*auto dest!: reals-Archimedean*)

then have $\{y. \text{dist } x\ y < \text{inverse } (Suc\ i)\} \subseteq \{y. \text{dist } x\ y < e\}$

by *auto*

ultimately show $\exists i. \{y. \text{dist } x\ y < \text{inverse } (Suc\ i)\} \subseteq S$

by *blast*

qed (*auto intro: open-ball*)

qed

end

instance *metric-space* \subseteq *t2-space*

proof

```

fix x y :: 'a::metric-space
assume xy: x  $\neq$  y
let ?U = {y'. dist x y' < dist x y / 2}
let ?V = {x'. dist y x' < dist x y / 2}
have *: d x z  $\leq$  d x y + d y z  $\implies$  d y z = d z y  $\implies$   $\neg$  (d x y * 2 < d x z  $\wedge$  d z
y * 2 < d x z)
for d :: 'a  $\Rightarrow$  'a  $\Rightarrow$  real and x y z :: 'a
by arith
have open ?U  $\wedge$  open ?V  $\wedge$  x  $\in$  ?U  $\wedge$  y  $\in$  ?V  $\wedge$  ?U  $\cap$  ?V = {}
using dist-pos-lt[OF xy] *[of dist, OF dist-triangle dist-commute]
using open-ball[of - dist x y / 2] by auto
then show  $\exists$  U V. open U  $\wedge$  open V  $\wedge$  x  $\in$  U  $\wedge$  y  $\in$  V  $\wedge$  U  $\cap$  V = {}
by blast
qed

```

Every normed vector space is a metric space.

instance *real-normed-vector* $<$ *metric-space*

proof

```

fix x y z :: 'a
show dist x y = 0  $\longleftrightarrow$  x = y
by (simp add: dist-norm)
show dist x y  $\leq$  dist x z + dist y z
using norm-triangle-ineq4 [of x - z y - z] by (simp add: dist-norm)
qed

```

108.7 Class instances for real numbers

instantiation *real* :: *real-normed-field*

begin

definition *dist-real-def*: dist x y = |x - y|

definition *uniformity-real-def* [code del]:

(*uniformity* :: (real \times real) filter) = (INF e \in {0 <..}. principal {(x, y). dist x y < e})

definition *open-real-def* [code del]:

open (U :: real set) \longleftrightarrow (\forall x \in U. eventually (λ (x', y). x' = x \longrightarrow y \in U) uniformity)

definition *real-norm-def* [simp]: norm r = |r|

instance

by intro-classes (auto simp: abs-mult open-real-def dist-real-def sgn-real-def uni-

formity-real-def)

end

declare *uniformity-Abort*[**where** 'a=real, code]

lemma *dist-of-real* [*simp*]: *dist* (*of-real* $x :: 'a$) (*of-real* y) = *dist* x y
for $a :: 'a::\text{real-normed-div-algebra}$
by (*metis dist-norm norm-of-real of-real-diff real-norm-def*)

declare [[*code abort: open :: real set \Rightarrow bool*]]

instance *real* :: *linorder-topology*

proof

show (*open* :: *real set* \Rightarrow *bool*) = *generate-topology* (*range lessThan* \cup *range greaterThan*)

proof (*rule ext, safe*)

fix $S :: \text{real set}$

assume *open* S

then obtain f **where** $\forall x \in S. 0 < f\ x \wedge (\forall y. \text{dist } y\ x < f\ x \longrightarrow y \in S)$

unfolding *open-dist bchoice-iff* ..

then have $*$: $(\bigcup x \in S. \{x - f\ x <..\} \cap \{.. < x + f\ x\}) = S$ (**is** $?S = S$)

by (*fastforce simp: dist-real-def*)

moreover have *generate-topology* (*range lessThan* \cup *range greaterThan*) $?S$

by (*force intro: generate-topology.Basis generate-topology-Union generate-topology.Int*)

ultimately show *generate-topology* (*range lessThan* \cup *range greaterThan*) S

by *simp*

next

fix $S :: \text{real set}$

assume *generate-topology* (*range lessThan* \cup *range greaterThan*) S

moreover have $\bigwedge a::\text{real}. \text{open } \{.. < a\}$

unfolding *open-dist dist-real-def*

proof *clarify*

fix $x\ a :: \text{real}$

assume $x < a$

then have $0 < a - x \wedge (\forall y. |y - x| < a - x \longrightarrow y \in \{.. < a\})$ **by** *auto*

then show $\exists e > 0. \forall y. |y - x| < e \longrightarrow y \in \{.. < a\}$..

qed

moreover have $\bigwedge a::\text{real}. \text{open } \{a <..\}$

unfolding *open-dist dist-real-def*

proof *clarify*

fix $x\ a :: \text{real}$

assume $a < x$

then have $0 < x - a \wedge (\forall y. |y - x| < x - a \longrightarrow y \in \{a <..\})$ **by** *auto*

then show $\exists e > 0. \forall y. |y - x| < e \longrightarrow y \in \{a <..\}$..

qed

ultimately show *open* S

by *induct auto*

qed

qed

instance real :: linear-continuum-topology ..

lemmas open-real-greaterThan = open-greaterThan[where 'a=real]
 lemmas open-real-lessThan = open-lessThan[where 'a=real]
 lemmas open-real-greaterThanLessThan = open-greaterThanLessThan[where 'a=real]
 lemmas closed-real-atMost = closed-atMost[where 'a=real]
 lemmas closed-real-atLeast = closed-atLeast[where 'a=real]
 lemmas closed-real-atLeastAtMost = closed-atLeastAtMost[where 'a=real]

instance real :: ordered-real-vector

by standard (auto intro: mult-left-mono mult-right-mono)

108.8 Extra type constraints

Only allow *open* in class *topological-space*.

setup <Sign.add-const-constraint
 (const-name <open>, SOME typ <'a::topological-space set \Rightarrow bool>)>

Only allow *uniformity* in class *uniform-space*.

setup <Sign.add-const-constraint
 (const-name <uniformity>, SOME typ <('a::uniformity \times 'a) filter>)>

Only allow *dist* in class *metric-space*.

setup <Sign.add-const-constraint
 (const-name <dist>, SOME typ <'a::metric-space \Rightarrow 'a \Rightarrow real>)>

Only allow *norm* in class *real-normed-vector*.

setup <Sign.add-const-constraint
 (const-name <norm>, SOME typ <'a::real-normed-vector \Rightarrow real>)>

108.9 Sign function

lemma norm-sgn: norm (sgn x) = (if x = 0 then 0 else 1)
 for x :: 'a::real-normed-vector
 by (simp add: sgn-div-norm)

lemma sgn-zero [simp]: sgn (0::'a::real-normed-vector) = 0
 by (simp add: sgn-div-norm)

lemma sgn-zero-iff: sgn x = 0 \longleftrightarrow x = 0
 for x :: 'a::real-normed-vector
 by (simp add: sgn-div-norm)

lemma sgn-minus: sgn (- x) = - sgn x
 for x :: 'a::real-normed-vector
 by (simp add: sgn-div-norm)

lemma *sgn-scaleR*: $\text{sgn} (\text{scaleR } r \ x) = \text{scaleR} (\text{sgn } r) (\text{sgn } x)$
for $x :: 'a::\text{real-normed-vector}$
by (*simp add: sgn-div-norm ac-simps*)

lemma *sgn-one* [*simp*]: $\text{sgn} (1 :: 'a::\text{real-normed-algebra-1}) = 1$
by (*simp add: sgn-div-norm*)

lemma *sgn-of-real*: $\text{sgn} (\text{of-real } r :: 'a::\text{real-normed-algebra-1}) = \text{of-real} (\text{sgn } r)$
unfolding *of-real-def* **by** (*simp only: sgn-scaleR sgn-one*)

lemma *sgn-mult*: $\text{sgn} (x * y) = \text{sgn } x * \text{sgn } y$
for $x \ y :: 'a::\text{real-normed-div-algebra}$
by (*simp add: sgn-div-norm norm-mult*)

hide-fact (**open**) *sgn-mult*

lemma *real-sgn-eq*: $\text{sgn } x = x / |x|$
for $x :: \text{real}$
by (*simp add: sgn-div-norm divide-inverse*)

lemma *zero-le-sgn-iff* [*simp*]: $0 \leq \text{sgn } x \longleftrightarrow 0 \leq x$
for $x :: \text{real}$
by (*cases 0::real x rule: linorder-cases*) *simp-all*

lemma *sgn-le-0-iff* [*simp*]: $\text{sgn } x \leq 0 \longleftrightarrow x \leq 0$
for $x :: \text{real}$
by (*cases 0::real x rule: linorder-cases*) *simp-all*

lemma *norm-conv-dist*: $\text{norm } x = \text{dist } x \ 0$
unfolding *dist-norm* **by** *simp*

declare *norm-conv-dist* [*symmetric, simp*]

lemma *dist-0-norm* [*simp*]: $\text{dist } 0 \ x = \text{norm } x$
for $x :: 'a::\text{real-normed-vector}$
by (*simp add: dist-norm*)

lemma *dist-diff* [*simp*]: $\text{dist } a \ (a - b) = \text{norm } b \ \text{dist } (a - b) \ a = \text{norm } b$
by (*simp-all add: dist-norm*)

lemma *dist-of-int*: $\text{dist} (\text{of-int } m) (\text{of-int } n :: 'a :: \text{real-normed-algebra-1}) = \text{of-int } |m - n|$

proof –

have $\text{dist} (\text{of-int } m) (\text{of-int } n :: 'a) = \text{dist} (\text{of-int } m :: 'a) (\text{of-int } m - (\text{of-int } (m - n)))$

by *simp*

also have $\dots = \text{of-int } |m - n|$ **by** (*subst dist-diff, subst norm-of-int*) *simp*

finally show *?thesis* .

qed

lemma *dist-of-nat*:

dist (of-nat m) (of-nat n :: 'a :: real-normed-algebra-1) = of-int |int m - int n|
by (*subst (1 2) of-int-of-nat-eq [symmetric]*) (*rule dist-of-int*)

108.10 Bounded Linear and Bilinear Operators

lemma *linearI*: *linear f*

if $\bigwedge b1\ b2. f (b1 + b2) = f\ b1 + f\ b2$

$\bigwedge r\ b. f (r *_R b) = r *_R f\ b$

using *that*

by *unfold-locales (auto simp: algebra-simps)*

lemma *linear-iff*:

linear f $\longleftrightarrow (\forall x\ y. f (x + y) = f\ x + f\ y) \wedge (\forall c\ x. f (c *_R x) = c *_R f\ x)$

(**is** *linear f* \longleftrightarrow *?rhs*)

proof

assume *linear f*

then interpret *f: linear f* .

show *?rhs* **by** (*simp add: f.add f.scale*)

next

assume *?rhs*

then show *linear f* **by** (*intro linearI*) *auto*

qed

lemmas *linear-scaleR-left = linear-scale-left*

lemmas *linear-imp-scaleR = linear-imp-scale*

corollary *real-linearD*:

fixes *f :: real \Rightarrow real*

assumes *linear f* **obtains** *c* **where** *f = (*) c*

by (*rule linear-imp-scaleR [OF assms]*) (*force simp: scaleR-conv-of-real*)

lemma *linear-times-of-real*: *linear ($\lambda x. a * \text{of-real } x$)*

by (*auto intro!: linearI simp: distrib-left*)

(*metis mult-scaleR-right scaleR-conv-of-real*)

locale *bounded-linear = linear f* **for** *f :: 'a::real-normed-vector \Rightarrow 'b::real-normed-vector*

+

assumes *bounded: $\exists K. \forall x. \text{norm } (f\ x) \leq \text{norm } x * K$*

begin

lemma *pos-bounded: $\exists K > 0. \forall x. \text{norm } (f\ x) \leq \text{norm } x * K$*

proof –

obtain *K* **where** *K: $\bigwedge x. \text{norm } (f\ x) \leq \text{norm } x * K$*

using *bounded* **by** *blast*

show *?thesis*

proof (*intro exI impI conjI allI*)

```

show  $0 < \max 1 K$ 
  by (rule order-less-le-trans [OF zero-less-one max.cobounded1])
next
  fix  $x$ 
  have  $\text{norm } (f x) \leq \text{norm } x * K$  using  $K$  .
  also have  $\dots \leq \text{norm } x * \max 1 K$ 
    by (rule mult-left-mono [OF max.cobounded2 norm-ge-zero])
  finally show  $\text{norm } (f x) \leq \text{norm } x * \max 1 K$  .
qed
qed

```

```

lemma nonneg-bounded:  $\exists K \geq 0. \forall x. \text{norm } (f x) \leq \text{norm } x * K$ 
  using pos-bounded by (auto intro: order-less-imp-le)

```

```

lemma linear: linear  $f$ 
  by (fact local.linear-axioms)

```

end

```

lemma bounded-linear-intro:
  assumes  $\bigwedge x y. f (x + y) = f x + f y$ 
    and  $\bigwedge r x. f (\text{scaleR } r x) = \text{scaleR } r (f x)$ 
    and  $\bigwedge x. \text{norm } (f x) \leq \text{norm } x * K$ 
  shows bounded-linear  $f$ 
  by standard (blast intro: assms)+

```

```

locale bounded-bilinear =
  fixes  $\text{prod} :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector} \Rightarrow 'c::\text{real-normed-vector}$ 
    (infixl ** 70)
  assumes add-left:  $\text{prod } (a + a') b = \text{prod } a b + \text{prod } a' b$ 
    and add-right:  $\text{prod } a (b + b') = \text{prod } a b + \text{prod } a b'$ 
    and scaleR-left:  $\text{prod } (\text{scaleR } r a) b = \text{scaleR } r (\text{prod } a b)$ 
    and scaleR-right:  $\text{prod } a (\text{scaleR } r b) = \text{scaleR } r (\text{prod } a b)$ 
    and bounded:  $\exists K. \forall a b. \text{norm } (\text{prod } a b) \leq \text{norm } a * \text{norm } b * K$ 
begin

```

```

lemma pos-bounded:  $\exists K > 0. \forall a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$ 
proof –
  obtain  $K$  where  $\bigwedge a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$ 
    using bounded by blast
  then have  $\text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * (\max 1 K)$  for  $a b$ 
    by (rule order.trans) (simp add: mult-left-mono)
  then show ?thesis
    by force
qed

```

```

lemma nonneg-bounded:  $\exists K \geq 0. \forall a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$ 
  using pos-bounded by (auto intro: order-less-imp-le)

```

lemma *additive-right*: $\text{additive } (\lambda b. \text{prod } a \ b)$
by (*rule additive.intro*, *rule add-right*)

lemma *additive-left*: $\text{additive } (\lambda a. \text{prod } a \ b)$
by (*rule additive.intro*, *rule add-left*)

lemma *zero-left*: $\text{prod } 0 \ b = 0$
by (*rule additive.zero* [*OF additive-left*])

lemma *zero-right*: $\text{prod } a \ 0 = 0$
by (*rule additive.zero* [*OF additive-right*])

lemma *minus-left*: $\text{prod } (- \ a) \ b = - \ \text{prod } a \ b$
by (*rule additive.minus* [*OF additive-left*])

lemma *minus-right*: $\text{prod } a \ (- \ b) = - \ \text{prod } a \ b$
by (*rule additive.minus* [*OF additive-right*])

lemma *diff-left*: $\text{prod } (a - a') \ b = \text{prod } a \ b - \text{prod } a' \ b$
by (*rule additive.diff* [*OF additive-left*])

lemma *diff-right*: $\text{prod } a \ (b - b') = \text{prod } a \ b - \text{prod } a \ b'$
by (*rule additive.diff* [*OF additive-right*])

lemma *sum-left*: $\text{prod } (\text{sum } g \ S) \ x = \text{sum } ((\lambda i. \text{prod } (g \ i) \ x)) \ S$
by (*rule additive.sum* [*OF additive-left*])

lemma *sum-right*: $\text{prod } x \ (\text{sum } g \ S) = \text{sum } ((\lambda i. (\text{prod } x \ (g \ i)))) \ S$
by (*rule additive.sum* [*OF additive-right*])

lemma *bounded-linear-left*: $\text{bounded-linear } (\lambda a. a \ ** \ b)$

proof –

obtain K **where** $\bigwedge a \ b. \text{norm } (a \ ** \ b) \leq \text{norm } a \ * \ \text{norm } b \ * \ K$

using *pos-bounded* **by** *blast*

then show *?thesis*

by (*rule-tac* $K = \text{norm } b \ * \ K$ **in** *bounded-linear-intro*) (*auto simp: algebra-simps scaleR-left add-left*)

qed

lemma *bounded-linear-right*: $\text{bounded-linear } (\lambda b. a \ ** \ b)$

proof –

obtain K **where** $\bigwedge a \ b. \text{norm } (a \ ** \ b) \leq \text{norm } a \ * \ \text{norm } b \ * \ K$

using *pos-bounded* **by** *blast*

then show *?thesis*

by (*rule-tac* $K = \text{norm } a \ * \ K$ **in** *bounded-linear-intro*) (*auto simp: algebra-simps scaleR-right add-right*)

qed

lemma *prod-diff-prod*: $(x ** y - a ** b) = (x - a) ** (y - b) + (x - a) ** b + a ** (y - b)$

by (*simp add: diff-left diff-right*)

lemma *flip: bounded-bilinear* $(\lambda x y. y ** x)$

proof

show $\exists K. \forall a b. \text{norm } (b ** a) \leq \text{norm } a * \text{norm } b * K$

by (*metis bounded mult.commute*)

qed (*simp-all add: add-right add-left scaleR-right scaleR-left*)

lemma *comp1*:

assumes *bounded-linear g*

shows *bounded-bilinear* $(\lambda x. (**)) (g x)$

proof *unfold-locales*

interpret *g: bounded-linear g* **by** *fact*

show $\bigwedge a a' b. g (a + a') ** b = g a ** b + g a' ** b$

$\bigwedge a b b'. g a ** (b + b') = g a ** b + g a ** b'$

$\bigwedge r a b. g (r *_R a) ** b = r *_R (g a ** b)$

$\bigwedge a r b. g a ** (r *_R b) = r *_R (g a ** b)$

by (*auto simp: g.add add-left add-right g.scaleR scaleR-left scaleR-right*)

from *g.nonneg-bounded nonneg-bounded* **obtain** *K L*

where *nn: 0 ≤ K 0 ≤ L*

and *K: $\bigwedge x. \text{norm } (g x) \leq \text{norm } x * K$*

and *L: $\bigwedge a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * L$*

by *auto*

have $\text{norm } (g a ** b) \leq \text{norm } a * K * \text{norm } b * L$ **for** *a b*

by (*auto intro!: order-trans[OF K] order-trans[OF L] mult-mono simp: nn*)

then show $\exists K. \forall a b. \text{norm } (g a ** b) \leq \text{norm } a * \text{norm } b * K$

by (*auto intro!: exI[where x=K * L] simp: ac-simps*)

qed

lemma *comp: bounded-linear f \implies bounded-linear g \implies bounded-bilinear* $(\lambda x y. f x ** g y)$

by (*rule bounded-bilinear.flip[OF bounded-bilinear.comp1[OF bounded-bilinear.flip[OF comp1]]]]*)

end

lemma *bounded-linear-ident[simp]*: *bounded-linear* $(\lambda x. x)$

by *standard (auto intro!: exI[of - 1])*

lemma *bounded-linear-zero[simp]*: *bounded-linear* $(\lambda x. 0)$

by *standard (auto intro!: exI[of - 1])*

lemma *bounded-linear-add*:

assumes *bounded-linear f*

and *bounded-linear g*

shows *bounded-linear* $(\lambda x. f x + g x)$

proof –

```

interpret f: bounded-linear f by fact
interpret g: bounded-linear g by fact
show ?thesis
proof
  from f.bounded obtain Kf where Kf: norm (f x) ≤ norm x * Kf for x
    by blast
  from g.bounded obtain Kg where Kg: norm (g x) ≤ norm x * Kg for x
    by blast
  show ∃ K. ∀ x. norm (f x + g x) ≤ norm x * K
    using add-mono[OF Kf Kg]
    by (intro exI[of - Kf + Kg]) (auto simp: field-simps intro: norm-triangle-ineq
order-trans)
  qed (simp-all add: f.add g.add f.scaleR g.scaleR scaleR-right-distrib)
qed

```

```

lemma bounded-linear-minus:
  assumes bounded-linear f
  shows bounded-linear (λx. - f x)
proof -
  interpret f: bounded-linear f by fact
  show ?thesis
    by unfold-locales (simp-all add: f.add f.scaleR f.bounded)
qed

```

```

lemma bounded-linear-sub: bounded-linear f ⇒ bounded-linear g ⇒ bounded-linear
(λx. f x - g x)
  using bounded-linear-add[of f λx. - g x] bounded-linear-minus[of g]
  by (auto simp: algebra-simps)

```

```

lemma bounded-linear-sum:
  fixes f :: 'i ⇒ 'a::real-normed-vector ⇒ 'b::real-normed-vector
  shows (∧ i. i ∈ I ⇒ bounded-linear (f i)) ⇒ bounded-linear (λx. ∑ i∈I. f i
x)
  by (induct I rule: infinite-finite-induct) (auto intro!: bounded-linear-add)

```

```

lemma bounded-linear-compose:
  assumes bounded-linear f
  and bounded-linear g
  shows bounded-linear (λx. f (g x))
proof -
  interpret f: bounded-linear f by fact
  interpret g: bounded-linear g by fact
  show ?thesis
proof unfold-locales
  show f (g (x + y)) = f (g x) + f (g y) for x y
    by (simp only: f.add g.add)
  show f (g (scaleR r x)) = scaleR r (f (g x)) for r x
    by (simp only: f.scaleR g.scaleR)
  from f.pos-bounded obtain Kf where f: ∧ x. norm (f x) ≤ norm x * Kf and

```

Kf: $0 < Kf$
 by *blast*
from g .*pos-bounded* **obtain** Kg **where** $g: \bigwedge x. \text{norm } (g\ x) \leq \text{norm } x * Kg$
 by *blast*
show $\exists K. \forall x. \text{norm } (f\ (g\ x)) \leq \text{norm } x * K$
proof (*intro exI allI*)
 fix x
have $\text{norm } (f\ (g\ x)) \leq \text{norm } (g\ x) * Kf$
 using f .
also have $\dots \leq (\text{norm } x * Kg) * Kf$
 using $g\ Kf$ [*THEN order-less-imp-le*] **by** (*rule mult-right-mono*)
also have $(\text{norm } x * Kg) * Kf = \text{norm } x * (Kg * Kf)$
 by (*rule mult.assoc*)
finally show $\text{norm } (f\ (g\ x)) \leq \text{norm } x * (Kg * Kf)$.
 qed
 qed
 qed

lemma *bounded-bilinear-mult*: *bounded-bilinear* $((*) :: 'a \Rightarrow 'a \Rightarrow 'a :: \text{real-normed-algebra})$
proof (*rule bounded-bilinear.intro*)
show $\exists K. \forall a\ b :: 'a. \text{norm } (a * b) \leq \text{norm } a * \text{norm } b * K$
by (*rule-tac x=1 in exI*) (*simp add: norm-mult-ineq*)
 qed (*auto simp: algebra-simps*)

lemma *bounded-linear-mult-left*: *bounded-linear* $(\lambda x :: 'a :: \text{real-normed-algebra}. x * y)$
 using *bounded-bilinear-mult*
by (*rule bounded-bilinear.bounded-linear-left*)

lemma *bounded-linear-mult-right*: *bounded-linear* $(\lambda y :: 'a :: \text{real-normed-algebra}. x * y)$
 using *bounded-bilinear-mult*
by (*rule bounded-bilinear.bounded-linear-right*)

lemmas *bounded-linear-mult-const* =
bounded-linear-mult-left [*THEN bounded-linear-compose*]

lemmas *bounded-linear-const-mult* =
bounded-linear-mult-right [*THEN bounded-linear-compose*]

lemma *bounded-linear-divide*: *bounded-linear* $(\lambda x. x / y)$
for $y :: 'a :: \text{real-normed-field}$
unfolding *divide-inverse* **by** (*rule bounded-linear-mult-left*)

lemma *bounded-bilinear-scaleR*: *bounded-bilinear* *scaleR*
proof (*rule bounded-bilinear.intro*)
show $\exists K. \forall a\ b. \text{norm } (a *_R b) \leq \text{norm } a * \text{norm } b * K$
 using *less-eq-real-def* **by** *auto*
 qed (*auto simp: algebra-simps*)


```

lemma bounded-linear-scaleR-left: bounded-linear ( $\lambda r. \text{scaleR } r \ x$ )
  using bounded-bilinear-scaleR
  by (rule bounded-bilinear.bounded-linear-left)

lemma bounded-linear-scaleR-right: bounded-linear ( $\lambda x. \text{scaleR } r \ x$ )
  using bounded-bilinear-scaleR
  by (rule bounded-bilinear.bounded-linear-right)

lemmas bounded-linear-scaleR-const =
  bounded-linear-scaleR-left[THEN bounded-linear-compose]

lemmas bounded-linear-const-scaleR =
  bounded-linear-scaleR-right[THEN bounded-linear-compose]

lemma bounded-linear-of-real: bounded-linear ( $\lambda r. \text{of-real } r$ )
  unfolding of-real-def by (rule bounded-linear-scaleR-left)

lemma real-bounded-linear: bounded-linear  $f \longleftrightarrow (\exists c::\text{real}. f = (\lambda x. x * c))$ 
  for  $f :: \text{real} \Rightarrow \text{real}$ 
proof –
  {
    fix  $x$ 
    assume bounded-linear  $f$ 
    then interpret bounded-linear  $f$  .
    from scaleR[of x 1] have  $f \ x = x * f \ 1$ 
      by simp
  }
  then show ?thesis
    by (auto intro: exI[of - f 1] bounded-linear-mult-left)
qed

instance real-normed-algebra-1  $\subseteq$  perfect-space
proof
  fix  $x::'a$ 
  have  $\bigwedge e. 0 < e \implies \exists y. \text{norm } (y - x) < e \wedge y \neq x$ 
    by (rule-tac  $x = x + \text{of-real } (e/2)$  in exI) auto
  then show  $\neg \text{open } \{x\}$ 
    by (clarsimp simp: open-dist dist-norm)
qed

```

108.11 Filters and Limits on Metric Space

```

lemma (in metric-space) nhds-metric: nhds  $x = (\text{INF } e \in \{0 < ..\}. \text{principal } \{y. \text{dist } y \ x < e\})$ 
  unfolding nhds-def
proof (safe intro!: INF-eq)
  fix  $S$ 
  assume open  $S \ x \in S$ 

```

then obtain e **where** $\{y. \text{dist } y \ x < e\} \subseteq S \ 0 < e$
by *(auto simp: open-dist subset-eq)*
then show $\exists e \in \{0 < ..\}. \text{principal } \{y. \text{dist } y \ x < e\} \leq \text{principal } S$
by *auto*
qed *(auto intro!: exI[of - {y. dist x y < e}] for e] open-ball simp: dist-commute)*

lemma *tendsto-iff-uniformity:*

— More general analogus of *tendsto-iff* below. Applies to all uniform spaces, not just metric ones.

fixes $l :: \langle 'b :: \text{uniform-space} \rangle$
shows $\langle (f \longrightarrow l) \ F \longleftrightarrow (\forall E. \text{eventually } E \ \text{uniformity} \longrightarrow (\forall_F \ x \ \text{in } F. E \ (f \ x, l))) \rangle$

proof *(intro iffI allI impI)*

fix $E :: \langle ('b \times 'b) \Rightarrow \text{bool} \rangle$

assume $\langle (f \longrightarrow l) \ F \rangle$ **and** $\langle \text{eventually } E \ \text{uniformity} \rangle$

from $\langle \text{eventually } E \ \text{uniformity} \rangle$

have $\langle \text{eventually } (\lambda(x, y). E \ (y, x)) \ \text{uniformity} \rangle$

by *(simp add: uniformity-sym)*

then have $\langle \forall_F \ (y, x) \ \text{in } \text{uniformity}. y = l \longrightarrow E \ (x, y) \rangle$

using *eventually-mono* **by** *fastforce*

with $\langle (f \longrightarrow l) \ F \rangle$ **have** $\langle \text{eventually } (\lambda x. E \ (x, l)) \ (\text{filtermap } f \ F) \rangle$

by *(simp add: filterlim-def le-filter-def eventually-nhds-uniformity)*

then show $\langle \forall_F \ x \ \text{in } F. E \ (f \ x, l) \rangle$

by *(simp add: eventually-filtermap)*

next

assume $\text{asm}: \langle \forall E. \text{eventually } E \ \text{uniformity} \longrightarrow (\forall_F \ x \ \text{in } F. E \ (f \ x, l)) \rangle$

have $\langle \text{eventually } P \ (\text{filtermap } f \ F) \rangle$ **if** $\langle \forall_F \ (x, y) \ \text{in } \text{uniformity}. x = l \longrightarrow P \ y \rangle$

for P

proof —

from *that* **have** $\langle \forall_F \ (y, x) \ \text{in } \text{uniformity}. x = l \longrightarrow P \ y \rangle$

using *uniformity-sym* **where** $E = \langle \lambda(x, y). x = l \longrightarrow P \ y \rangle$ **by** *auto*

with asm **have** $\langle \forall_F \ x \ \text{in } F. P \ (f \ x) \rangle$

by *auto*

then show *?thesis*

by *(auto simp: eventually-filtermap)*

qed

then show $\langle (f \longrightarrow l) \ F \rangle$

by *(simp add: filterlim-def le-filter-def eventually-nhds-uniformity)*

qed

lemma *(in metric-space) tendsto-iff:* $(f \longrightarrow l) \ F \longleftrightarrow (\forall e > 0. \text{eventually } (\lambda x. \text{dist } (f \ x) \ l < e) \ F)$

unfolding *nhds-metric filterlim-INF filterlim-principal* **by** *auto*

lemma *tendsto-dist-iff:*

$((f \longrightarrow l) \ F) \longleftrightarrow (((\lambda x. \text{dist } (f \ x) \ l) \longrightarrow 0) \ F)$

unfolding *tendsto-iff* **by** *simp*

lemma (in *metric-space*) *tendstoI* [intro?]:

$(\bigwedge e. 0 < e \implies \text{eventually } (\lambda x. \text{dist } (f x) l < e) F) \implies (f \longrightarrow l) F$

by (*auto simp: tendsto-iff*)

lemma (in *metric-space*) *tendstoD*: $(f \longrightarrow l) F \implies 0 < e \implies \text{eventually } (\lambda x. \text{dist } (f x) l < e) F$

by (*auto simp: tendsto-iff*)

lemma (in *metric-space*) *eventually-nhds-metric*:

$\text{eventually } P \text{ (nhds } a) \longleftrightarrow (\exists d > 0. \forall x. \text{dist } x a < d \longrightarrow P x)$

unfolding *nhds-metric*

by (*subst eventually-INF-base*)

(*auto simp: eventually-principal Bex-def subset-eq intro: exI[of - min a b for a b]*)

lemma *eventually-at*: $\text{eventually } P \text{ (at } a \text{ within } S) \longleftrightarrow (\exists d > 0. \forall x \in S. x \neq a \wedge \text{dist } x a < d \longrightarrow P x)$

for $a :: 'a :: \text{metric-space}$

by (*auto simp: eventually-at-filter eventually-nhds-metric*)

lemma *frequently-at*: $\text{frequently } P \text{ (at } a \text{ within } S) \longleftrightarrow (\forall d > 0. \exists x \in S. x \neq a \wedge \text{dist } x a < d \wedge P x)$

for $a :: 'a :: \text{metric-space}$

unfolding *frequently-def eventually-at* **by** *auto*

lemma *eventually-at-le*: $\text{eventually } P \text{ (at } a \text{ within } S) \longleftrightarrow (\exists d > 0. \forall x \in S. x \neq a \wedge \text{dist } x a \leq d \longrightarrow P x)$

for $a :: 'a :: \text{metric-space}$

unfolding *eventually-at-filter eventually-nhds-metric*

apply *safe*

apply (*rule-tac x=d / 2 in exI, auto*)

done

lemma *eventually-at-left-real*: $a > (b :: \text{real}) \implies \text{eventually } (\lambda x. x \in \{b < .. < a\}) \text{ (at-left } a)$

by (*subst eventually-at, rule exI[of - a - b] (force simp: dist-real-def)*)

lemma *eventually-at-right-real*: $a < (b :: \text{real}) \implies \text{eventually } (\lambda x. x \in \{a < .. < b\}) \text{ (at-right } a)$

by (*subst eventually-at, rule exI[of - b - a] (force simp: dist-real-def)*)

lemma *metric-tendsto-imp-tendsto*:

fixes $a :: 'a :: \text{metric-space}$

and $b :: 'b :: \text{metric-space}$

assumes $f: (f \longrightarrow a) F$

and $le: \text{eventually } (\lambda x. \text{dist } (g x) b \leq \text{dist } (f x) a) F$

shows $(g \longrightarrow b) F$

proof (*rule tendstoI*)

fix $e :: \text{real}$

```

assume  $0 < e$ 
with  $f$  have eventually  $(\lambda x. \text{dist } (f x) a < e)$   $F$  by (rule tendstoD)
with  $le$  show eventually  $(\lambda x. \text{dist } (g x) b < e)$   $F$ 
  using le-less-trans by (rule eventually-elim2)
qed

```

```

lemma filterlim-real-sequentially: LIM x sequentially. real x := at-top
proof (clarsimp simp: filterlim-at-top)
  fix  $Z$ 
  show  $\forall_F x \text{ in } \text{sequentially}. Z \leq \text{real } x$ 
    by (meson eventually-sequentiallyI nat-ceiling-le-eq)
qed

```

```

lemma filterlim-nat-sequentially: filterlim nat sequentially at-top
proof –
  have  $\forall_F x \text{ in } \text{at-top}. Z \leq \text{nat } x$  for  $Z$ 
    by (auto intro!: eventually-at-top-linorderI[where c=int Z])
  then show ?thesis
    unfolding filterlim-at-top ..
qed

```

```

lemma filterlim-floor-sequentially: filterlim floor at-top at-top
proof –
  have  $\forall_F x \text{ in } \text{at-top}. Z \leq \lfloor x \rfloor$  for  $Z$ 
    by (auto simp: le-floor-iff intro!: eventually-at-top-linorderI[where c=of-int Z])
  then show ?thesis
    unfolding filterlim-at-top ..
qed

```

```

lemma filterlim-sequentially-iff-filterlim-real:
  filterlim f sequentially F  $\longleftrightarrow$  filterlim  $(\lambda x. \text{real } (f x)) \text{ at-top } F$  (is ?lhs = ?rhs)
proof
  assume ?lhs then show ?rhs
    using filterlim-compose filterlim-real-sequentially by blast
next
  assume  $R: ?rhs$ 
  show ?lhs
  proof –
    have filterlim  $(\lambda x. \text{nat } (\text{floor } (\text{real } (f x)))) \text{ sequentially } F$ 
      by (intro filterlim-compose[OF filterlim-nat-sequentially]
        filterlim-compose[OF filterlim-floor-sequentially] R)
    then show ?thesis by simp
  qed
qed

```

108.11.1 Limits of Sequences

```

lemma lim-sequentially:  $X \longrightarrow L \longleftrightarrow (\forall r > 0. \exists no. \forall n \geq no. \text{dist } (X n) L < r)$ 

```

for $L :: 'a::\text{metric-space}$
unfolding *tendsto-iff eventually-sequentially* ..

lemmas *LIMSEQ-def = lim-sequentially*

lemma *LIMSEQ-iff-nz*: $X \longrightarrow L \longleftrightarrow (\forall r > 0. \exists no > 0. \forall n \geq no. \text{dist } (X \ n) \ L < r)$

for $L :: 'a::\text{metric-space}$
unfolding *lim-sequentially* **by** (*metis Suc-leD zero-less-Suc*)

lemma *metric-LIMSEQ-I*: $(\bigwedge r. 0 < r \implies \exists no. \forall n \geq no. \text{dist } (X \ n) \ L < r) \implies X \longrightarrow L$

for $L :: 'a::\text{metric-space}$
by (*simp add: lim-sequentially*)

lemma *metric-LIMSEQ-D*: $X \longrightarrow L \implies 0 < r \implies \exists no. \forall n \geq no. \text{dist } (X \ n) \ L < r$

for $L :: 'a::\text{metric-space}$
by (*simp add: lim-sequentially*)

lemma *LIMSEQ-norm-0*:

assumes $\bigwedge n::\text{nat}. \text{norm } (f \ n) < 1 / \text{real } (Suc \ n)$
shows $f \longrightarrow 0$

proof (*rule metric-LIMSEQ-I*)

fix $\varepsilon :: \text{real}$

assume $\varepsilon > 0$

then obtain $N::\text{nat}$ **where** $\varepsilon > \text{inverse } N \ N > 0$

by (*metis neq0-conv real-arch-inverse*)

then have $\text{norm } (f \ n) < \varepsilon$ **if** $n \geq N$ **for** n

proof –

have $1 / (Suc \ n) \leq 1 / N$

using $\langle 0 < N \rangle$ *inverse-of-nat-le le-SucI* **that** **by** *blast*

also have $\dots < \varepsilon$

by (*metis (no-types) <inverse (real N) < ε> inverse-eq-divide*)

finally show *?thesis*

by (*meson assms less-eq-real-def not-le order-trans*)

qed

then show $\exists no. \forall n \geq no. \text{dist } (f \ n) \ 0 < \varepsilon$

by *auto*

qed

108.11.2 Limits of Functions

lemma *LIM-def*: $f \ -a \rightarrow L \longleftrightarrow (\forall r > 0. \exists s > 0. \forall x. x \neq a \wedge \text{dist } x \ a < s \longrightarrow \text{dist } (f \ x) \ L < r)$

for $a :: 'a::\text{metric-space}$ **and** $L :: 'b::\text{metric-space}$

unfolding *tendsto-iff eventually-at* **by** *simp*

lemma *metric-LIM-I*:

$(\bigwedge r. 0 < r \implies \exists s > 0. \forall x. x \neq a \wedge \text{dist } x \ a < s \longrightarrow \text{dist } (f \ x) \ L < r) \implies f \ -a \rightarrow L$
for $a :: 'a::\text{metric-space}$ **and** $L :: 'b::\text{metric-space}$
by (*simp add: LIM-def*)

lemma *metric-LIM-D*: $f \ -a \rightarrow L \implies 0 < r \implies \exists s > 0. \forall x. x \neq a \wedge \text{dist } x \ a < s \longrightarrow \text{dist } (f \ x) \ L < r$
for $a :: 'a::\text{metric-space}$ **and** $L :: 'b::\text{metric-space}$
by (*simp add: LIM-def*)

lemma *metric-LIM-imp-LIM*:
fixes $l :: 'a::\text{metric-space}$
and $m :: 'b::\text{metric-space}$
assumes $f: f \ -a \rightarrow l$
and $le: \bigwedge x. x \neq a \implies \text{dist } (g \ x) \ m \leq \text{dist } (f \ x) \ l$
shows $g \ -a \rightarrow m$
by (*rule metric-tendsto-imp-tendsto [OF f]*) (*auto simp: eventually-at-topological le*)

lemma *metric-LIM-equal2*:
fixes $a :: 'a::\text{metric-space}$
assumes $g \ -a \rightarrow l \ 0 < R$
and $\bigwedge x. x \neq a \implies \text{dist } x \ a < R \implies f \ x = g \ x$
shows $f \ -a \rightarrow l$
proof –
have $\bigwedge S. \llbracket \text{open } S; l \in S; \forall_F x \text{ in } at \ a. g \ x \in S \rrbracket \implies \forall_F x \text{ in } at \ a. f \ x \in S$
apply (*simp add: eventually-at*)
by (*metis assms(2) assms(3) dual-order.strict-trans linorder-neqE-linordered-idom*)
then show *?thesis*
using *assms* **by** (*simp add: tendsto-def*)
qed

lemma *metric-LIM-compose2*:
fixes $a :: 'a::\text{metric-space}$
assumes $f: f \ -a \rightarrow b$
and $g: g \ -b \rightarrow c$
and $inj: \exists d > 0. \forall x. x \neq a \wedge \text{dist } x \ a < d \longrightarrow f \ x \neq b$
shows $(\lambda x. g \ (f \ x)) \ -a \rightarrow c$
using *inj* **by** (*intro tendsto-compose-eventually[OF g f]*) (*auto simp: eventually-at*)

lemma *metric-isCont-LIM-compose2*:
fixes $f :: 'a :: \text{metric-space} \Rightarrow -$
assumes f [*unfolded isCont-def*]: *isCont* $f \ a$
and $g: g \ -f \ a \rightarrow l$
and $inj: \exists d > 0. \forall x. x \neq a \wedge \text{dist } x \ a < d \longrightarrow f \ x \neq f \ a$
shows $(\lambda x. g \ (f \ x)) \ -a \rightarrow l$
by (*rule metric-LIM-compose2 [OF f g inj]*)

108.12 Complete metric spaces**108.13 Cauchy sequences**

lemma (in *metric-space*) *Cauchy-def*: $\text{Cauchy } X = (\forall e > 0. \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X \ m) \ (X \ n) < e)$

proof –

have *: *eventually* P (*INF* M . *principal* $\{(X \ m, X \ n) \mid n \ m. m \geq M \wedge n \geq M\}$)
 \longleftrightarrow

$(\exists M. \forall m \geq M. \forall n \geq M. P \ (X \ m, X \ n))$ **for** P

apply (*subst eventually-INF-base*)

subgoal by *simp*

subgoal for $a \ b$

by (*intro bexI[of - max a b]*) (*auto simp: eventually-principal subset-eq*)

subgoal by (*auto simp: eventually-principal, blast*)

done

have $\text{Cauchy } X \longleftrightarrow (\text{INF } M. \text{principal } \{(X \ m, X \ n) \mid n \ m. m \geq M \wedge n \geq M\})$
 \leq *uniformity*

unfolding *Cauchy-uniform-iff le-filter-def* * ..

also have ... = $(\forall e > 0. \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X \ m) \ (X \ n) < e)$

unfolding *uniformity-dist le-INF-iff* **by** (*auto simp: * le-principal*)

finally show *?thesis* .

qed

lemma (in *metric-space*) *Cauchy-altdef*: $\text{Cauchy } f \longleftrightarrow (\forall e > 0. \exists M. \forall m \geq M. \forall n > m. \text{dist } (f \ m) \ (f \ n) < e)$

(is *?lhs* \longleftrightarrow *?rhs*)

proof

assume *?rhs*

show *?lhs*

unfolding *Cauchy-def*

proof (*intro allI impI*)

fix $e :: \text{real}$ **assume** $e > 0$

with $\langle ?rhs \rangle$ **obtain** M **where** $M: m \geq M \implies n > m \implies \text{dist } (f \ m) \ (f \ n) < e$
for $m \ n$

by *blast*

have $\text{dist } (f \ m) \ (f \ n) < e$ **if** $m \geq M \ n \geq M$ **for** $m \ n$

using $M[\text{of } m \ n] \ M[\text{of } n \ m] \ e$ **that** **by** (*cases m n rule: linorder-cases*) (*auto simp: dist-commute*)

then show $\exists M. \forall m \geq M. \forall n \geq M. \text{dist } (f \ m) \ (f \ n) < e$

by *blast*

qed

next

assume *?lhs*

show *?rhs*

proof (*intro allI impI*)

fix $e :: \text{real}$

assume $e > 0$

with $\langle \text{Cauchy } f \rangle$ **obtain** M **where** $\bigwedge m \ n. m \geq M \implies n \geq M \implies \text{dist } (f \ m) \ (f \ n) < e$

unfolding *Cauchy-def* **by** *blast*
then show $\exists M. \forall m \geq M. \forall n > m. \text{dist } (f \ m) \ (f \ n) < e$
by (*intro exI[of - M]*) *force*
qed
qed

lemma (*in metric-space*) *Cauchy-altdef2*: $\text{Cauchy } s \longleftrightarrow (\forall e > 0. \exists N :: \text{nat}. \forall n \geq N. \text{dist}(s \ n)(s \ N) < e)$ (*is ?lhs = ?rhs*)

proof

assume *Cauchy s*
then show *?rhs* **by** (*force simp: Cauchy-def*)
next
assume *?rhs*
{
fix *e::real*
assume $e > 0$
with $\langle ?rhs \rangle$ **obtain** *N* **where** $N: \forall n \geq N. \text{dist } (s \ n) \ (s \ N) < e/2$
by (*erule-tac x=e/2 in allE*) *auto*
{
fix *n m*
assume $nm: N \leq m \wedge N \leq n$
then have $\text{dist } (s \ m) \ (s \ n) < e$ **using** *N*
using *dist-triangle-half-l[of s m s N e s n]*
by *blast*
}
then have $\exists N. \forall m \ n. N \leq m \wedge N \leq n \longrightarrow \text{dist } (s \ m) \ (s \ n) < e$
by *blast*
}
then have *?lhs*
unfolding *Cauchy-def* **by** *blast*
then show *?lhs*
by *blast*
qed

lemma (*in metric-space*) *metric-CauchyI*:

$(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X \ m) \ (X \ n) < e) \implies \text{Cauchy } X$
by (*simp add: Cauchy-def*)

lemma (*in metric-space*) *CauchyI'*:

$(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n > m. \text{dist } (X \ m) \ (X \ n) < e) \implies \text{Cauchy } X$
unfolding *Cauchy-altdef* **by** *blast*

lemma (*in metric-space*) *metric-CauchyD*:

$\text{Cauchy } X \implies 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X \ m) \ (X \ n) < e$
by (*simp add: Cauchy-def*)

lemma (*in metric-space*) *metric-Cauchy-iff2*:

$\text{Cauchy } X = (\forall j. (\exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X \ m) \ (X \ n) < \text{inverse}(\text{real } (\text{Suc } j))))$

apply (*auto simp add: Cauchy-def*)
by (*metis less-trans of-nat-Suc reals-Archimedean*)

lemma *Cauchy-iff2*: $\text{Cauchy } X \longleftrightarrow (\forall j. (\exists M. \forall m \geq M. \forall n \geq M. |X\ m - X\ n| < \text{inverse } (\text{Suc } j))))$
by (*simp only: metric-Cauchy-iff2 dist-real-def*)

lemma *lim-1-over-n [tendsto-intros]*: $((\lambda n. 1 / \text{of-nat } n) \longrightarrow (0 :: 'a :: \text{real-normed-field}))$
sequentially

proof (*subst lim-sequentially, intro allI impI exI*)
fix $e :: \text{real}$ **and** n
assume $e > 0$
have $\text{inverse } e < \text{of-nat } (\text{nat } \lceil \text{inverse } e + 1 \rceil)$ **by** *linarith*
also assume $n \geq \text{nat } \lceil \text{inverse } e + 1 \rceil$
finally show $\text{dist } (1 / \text{of-nat } n :: 'a) 0 < e$
using e **by** (*simp add: field-split-simps norm-divide*)

qed

lemma (*in metric-space*) *complete-def*:

shows $\text{complete } S = (\forall f. (\forall n. f\ n \in S) \wedge \text{Cauchy } f \longrightarrow (\exists l \in S. f \longrightarrow l))$
unfolding *complete-uniform*

proof *safe*

fix $f :: \text{nat} \Rightarrow 'a$
assume $f: \forall n. f\ n \in S$ *Cauchy* f
and $*$: $\forall F \leq \text{principal } S. F \neq \text{bot} \longrightarrow \text{cauchy-filter } F \longrightarrow (\exists x \in S. F \leq \text{nhds } x)$
then show $\exists l \in S. f \longrightarrow l$
unfolding *filterlim-def* **using** f
by (*intro *[rule-format]*)
(auto simp: filtermap-sequentially-ne-bot le-principal eventually-filtermap

Cauchy-uniform)

next

fix $F :: 'a$ *filter*
assume $F \leq \text{principal } S$ $F \neq \text{bot}$ *cauchy-filter* F
assume *seq*: $\forall f. (\forall n. f\ n \in S) \wedge \text{Cauchy } f \longrightarrow (\exists l \in S. f \longrightarrow l)$

from $\langle F \leq \text{principal } S \rangle \langle \text{cauchy-filter } F \rangle$

have *FF-le*: $F \times_F F \leq \text{uniformity-on } S$

by (*simp add: cauchy-filter-def principal-prod-principal[symmetric] prod-filter-mono*)

let $?P = \lambda P e. \text{eventually } P\ F \wedge (\forall x. P\ x \longrightarrow x \in S) \wedge (\forall x\ y. P\ x \longrightarrow P\ y \longrightarrow \text{dist } x\ y < e)$

have $P: \exists P. ?P\ P\ \varepsilon$ **if** $0 < \varepsilon$ **for** $\varepsilon :: \text{real}$

proof –

from *that* **have** *eventually* $(\lambda(x, y). x \in S \wedge y \in S \wedge \text{dist } x\ y < \varepsilon)$ (*uniformity-on* S)

by (*auto simp: eventually-inf-principal eventually-uniformity-metric*)

from *filter-leD[OF FF-le this]* **show** *?thesis*

by (*auto simp: eventually-prod-same*)

qed

```

have  $\exists P. \forall n. ?P (P n) (1 / \text{Suc } n) \wedge P (\text{Suc } n) \leq P n$ 
proof (rule dependent-nat-choice)
  show  $\exists P. ?P P (1 / \text{Suc } 0)$ 
    using  $P[\text{of } 1]$  by auto
next
  fix  $P n$  assume  $?P P (1/\text{Suc } n)$ 
  moreover obtain  $Q$  where  $?P Q (1 / \text{Suc } (\text{Suc } n))$ 
    using  $P[\text{of } 1/\text{Suc } (\text{Suc } n)]$  by auto
  ultimately show  $\exists Q. ?P Q (1 / \text{Suc } (\text{Suc } n)) \wedge Q \leq P$ 
    by (intro exI[ $\text{of } \lambda x. P x \wedge Q x$ ] (auto simp: eventually-conj-iff))
qed
then obtain  $P$  where  $P: \text{eventually } (P n) F P n x \implies x \in S$ 
   $P n x \implies P n y \implies \text{dist } x y < 1 / \text{Suc } n P (\text{Suc } n) \leq P n$ 
  for  $n x y$ 
  by metis
have antimono  $P$ 
  using  $P(4)$  by (rule decseq-SucI)

obtain  $X$  where  $X: P n (X n)$  for  $n$ 
  using  $P(1)[\text{THEN eventually-happens}[OF \langle F \neq \text{bot} \rangle]]$  by metis
have Cauchy  $X$ 
  unfolding metric-Cauchy-iff2 inverse-eq-divide
proof (intro exI allI impI)
  fix  $j m n :: \text{nat}$ 
  assume  $j \leq m j \leq n$ 
  with  $\langle \text{antimono } P \rangle X$  have  $P j (X m) P j (X n)$ 
    by (auto simp: antimono-def)
  then show  $\text{dist } (X m) (X n) < 1 / \text{Suc } j$ 
    by (rule  $P$ )
qed
moreover have  $\forall n. X n \in S$ 
  using  $P(2) X$  by auto
ultimately obtain  $x$  where  $X \longrightarrow x x \in S$ 
  using seq by blast

show  $\exists x \in S. F \leq \text{nhds } x$ 
proof (rule bxI)
  have eventually  $(\lambda y. \text{dist } y x < e) F$  if  $0 < e$  for  $e :: \text{real}$ 
  proof –
    from that have  $(\lambda n. 1 / \text{Suc } n :: \text{real}) \longrightarrow 0 \wedge 0 < e / 2$ 
      by (subst filterlim-sequentially-Suc) (auto intro!: lim-1-over-n)
    then have  $\forall_F n$  in sequentially. dist  $(X n) x < e / 2 \wedge 1 / \text{Suc } n < e / 2$ 
      using  $\langle X \longrightarrow x \rangle$ 
      unfolding tendsto-iff order-tendsto-iff [where 'a=real] eventually-conj-iff
      by blast
    then obtain  $n$  where  $\text{dist } x (X n) < e / 2 \wedge 1 / \text{Suc } n < e / 2$ 
      by (auto simp: eventually-sequentially dist-commute)
    show thesis
  
```

```

    using ‹eventually (P n) F›
  proof eventually-elim
    case (elim y)
    then have  $\text{dist } y (X n) < 1 / \text{Suc } n$ 
      by (intro X P)
    also have  $\dots < e / 2$  by fact
    finally show  $\text{dist } y x < e$ 
      by (rule dist-triangle-half-l) fact
  qed
  qed
  then show  $F \leq \text{nhds } x$ 
    unfolding nhds-metric le-INF-iff le-principal by auto
  qed fact
  qed

```

apparently unused

```

lemma (in metric-space) totally-bounded-metric:
  totally-bounded S  $\longleftrightarrow (\forall e > 0. \exists k. \text{finite } k \wedge S \subseteq (\bigcup x \in k. \{y. \text{dist } x y < e\}))$ 
  unfolding totally-bounded-def eventually-uniformity-metric imp-ex
  apply (subst all-comm)
  apply (intro arg-cong[where f=All] ext, safe)
  subgoal for e
    apply (erule allE[of -  $\lambda(x, y). \text{dist } x y < e$ ])
    apply auto
  done
  subgoal for e P k
    apply (intro exI[of - k])
    apply (force simp: subset-eq)
  done
done

```

```

setup ‹Sign.add-const-constraint (const-name ‹dist›, SOME typ ‹'a::dist  $\Rightarrow$  'a  $\Rightarrow$  real›)›

```

```

lemma cauchy-filter-metric:
  fixes F :: 'a::{uniformity-dist, uniform-space} filter
  shows cauchy-filter F  $\longleftrightarrow (\forall e. e > 0 \longrightarrow (\exists P. \text{eventually } P F \wedge (\forall x y. P x \wedge P y \longrightarrow \text{dist } x y < e)))$ 
  proof (unfold cauchy-filter-def le-filter-def, auto)
    assume assm: ‹ $\forall e > 0. \exists P. \text{eventually } P F \wedge (\forall x y. P x \wedge P y \longrightarrow \text{dist } x y < e)$ ›
    then show ‹eventually P uniformity  $\implies$  eventually P (F  $\times_F$  F)› for P
      apply (auto simp: eventually-uniformity-metric)
      using eventually-prod-same by blast
  next
  fix e :: real
  assume ‹ $e > 0$ ›

```

```

assume asm:  $\langle \forall P. \text{eventually } P \text{ uniformity} \longrightarrow \text{eventually } P (F \times_F F) \rangle$ 

define P where  $\langle P \equiv \lambda(x,y :: 'a). \text{dist } x \ y < e \rangle$ 
with asm  $\langle e > 0 \rangle$  have  $\langle \text{eventually } P (F \times_F F) \rangle$ 
  by (metis case-prod-conv eventually-uniformity-metric)
then
show  $\langle \exists P. \text{eventually } P \ F \wedge (\forall x \ y. P \ x \wedge P \ y \longrightarrow \text{dist } x \ y < e) \rangle$ 
  by (auto simp add: eventually-prod-same P-def)
qed

lemma cauchy-filter-metric-filtermap:
  fixes f ::  $'a \Rightarrow 'b :: \{\text{uniformity-dist, uniform-space}\}$ 
  shows cauchy-filter (filtermap f F)  $\longleftrightarrow$   $(\forall e. e > 0 \longrightarrow (\exists P. \text{eventually } P \ F \wedge (\forall x \ y. P \ x \wedge P \ y \longrightarrow \text{dist } (f \ x) (f \ y) < e)))$ 
proof (subst cauchy-filter-metric, intro iffI allI impI)
  assume  $\langle \forall e > 0. \exists P. \text{eventually } P (filtermap \ f \ F) \wedge (\forall x \ y. P \ x \wedge P \ y \longrightarrow \text{dist } x \ y < e) \rangle$ 
  then show  $\langle e > 0 \implies \exists P. \text{eventually } P \ F \wedge (\forall x \ y. P \ x \wedge P \ y \longrightarrow \text{dist } (f \ x) (f \ y) < e) \rangle$  for e
    unfolding eventually-filtermap by blast
next
  assume asm:  $\langle \forall e > 0. \exists P. \text{eventually } P \ F \wedge (\forall x \ y. P \ x \wedge P \ y \longrightarrow \text{dist } (f \ x) (f \ y) < e) \rangle$ 
  fix e::real assume  $\langle e > 0 \rangle$ 
  then obtain P where  $\langle \text{eventually } P \ F \rangle$  and PPe:  $\langle P \ x \wedge P \ y \longrightarrow \text{dist } (f \ x) (f \ y) < e \rangle$  for x y
    using asm by blast

  show  $\langle \exists P. \text{eventually } P (filtermap \ f \ F) \wedge (\forall x \ y. P \ x \wedge P \ y \longrightarrow \text{dist } x \ y < e) \rangle$ 
    apply (rule exI[of -  $\langle \lambda x. \exists y. P \ y \wedge x = f \ y \rangle$ ])
    using PPe  $\langle \text{eventually } P \ F \rangle$  apply (auto simp: eventually-filtermap)
    by (smt (verit, ccfv-SIG) eventually-elim2)
qed

setup  $\langle \text{Sign.add-const-constraint } (\text{const-name } \langle \text{dist} \rangle, \text{SOME } \text{typ } \langle 'a :: \text{metric-space} \Rightarrow 'a \Rightarrow \text{real} \rangle) \rangle$ 

```

108.13.1 Cauchy Sequences are Convergent

```

class complete-space = metric-space +
  assumes Cauchy-convergent: Cauchy X  $\implies$  convergent X

```

```

lemma Cauchy-convergent-iff: Cauchy X  $\longleftrightarrow$  convergent X
  for X :: nat  $\Rightarrow$   $'a :: \text{complete-space}$ 
  by (blast intro: Cauchy-convergent convergent-Cauchy)

```

To prove that a Cauchy sequence converges, it suffices to show that a subsequence converges.

```

lemma Cauchy-converges-subseq:
  fixes  $u::nat \Rightarrow 'a::metric-space$ 
  assumes Cauchy  $u$ 
    strict-mono  $r$ 
     $(u \circ r) \longrightarrow l$ 
  shows  $u \longrightarrow l$ 
proof -
  have  $*$ : eventually  $(\lambda n. dist (u n) l < e)$  sequentially if  $e > 0$  for  $e$ 
  proof -
    have  $e/2 > 0$  using that by auto
    then obtain  $N1$  where  $N1: \bigwedge m n. m \geq N1 \implies n \geq N1 \implies dist (u m) (u n) < e/2$ 
    using  $\langle Cauchy\ u \rangle$  unfolding Cauchy-def by blast
    obtain  $N2$  where  $N2: \bigwedge n. n \geq N2 \implies dist ((u \circ r) n) l < e / 2$ 
    using order-tendstoD(2)[OF iffD1[OF tendsto-dist-iff  $\langle (u \circ r) \longrightarrow l \rangle \langle e/2 > 0 \rangle]$ 
    unfolding eventually-sequentially by auto
    have  $dist (u n) l < e$  if  $n \geq \max N1 N2$  for  $n$ 
    proof -
      have  $dist (u n) l \leq dist (u n) ((u \circ r) n) + dist ((u \circ r) n) l$ 
      by (rule dist-triangle)
      also have  $\dots < e/2 + e/2$ 
      proof (intro add-strict-mono)
        show  $dist (u n) ((u \circ r) n) < e / 2$ 
        using  $N1[of\ n\ r\ n]$   $N2[of\ n]$  that unfolding comp-def
        by (meson assms(2) le-trans max.bounded-iff strict-mono-imp-increasing)
        show  $dist ((u \circ r) n) l < e / 2$ 
        using  $N2$  that by auto
      qed
    finally show ?thesis by simp
    qed
  then show ?thesis unfolding eventually-sequentially by blast
qed
have  $(\lambda n. dist (u n) l) \longrightarrow 0$ 
by (simp add: less-le-trans * order-tendstoI)
then show ?thesis using tendsto-dist-iff by auto
qed

```

108.14 The set of real numbers is a complete metric space

Proof that Cauchy sequences converge based on the one from <http://pirate.shu.edu/~wachsmut/ira/numseq/proofs/cauconv.html>

If sequence X is Cauchy, then its limit is the lub of $\{r. \exists N. \forall n \geq N. r < X n\}$

```

lemma increasing-LIMSEQ:
  fixes  $f :: nat \Rightarrow real$ 
  assumes inc:  $\bigwedge n. f n \leq f (Suc n)$ 
  and bdd:  $\bigwedge n. f n \leq l$ 

```

and $en: \bigwedge e. 0 < e \implies \exists n. l \leq f n + e$
shows $f \longrightarrow l$
proof (rule increasing-tendsto)
fix x
assume $x < l$
with $dense[of\ 0\ l - x]$ **obtain** e **where** $0 < e \wedge e < l - x$
by *auto*
from $en[OF\ \langle 0 < e \rangle]$ **obtain** n **where** $l - e \leq f n$
by (auto simp: field-simps)
with $\langle e < l - x \rangle \langle 0 < e \rangle$ **have** $x < f n$
by *simp*
with $incseq-SucI[of\ f, OF\ inc]$ **show** eventually $(\lambda n. x < f n)$ sequentially
by (auto simp: eventually-sequentially incseq-def intro: less-le-trans)
qed (use bdd in auto)

lemma *real-Cauchy-convergent*:

fixes $X :: nat \Rightarrow real$
assumes $X: Cauchy\ X$
shows *convergent* X
proof –
define $S :: real\ set$ **where** $S = \{x. \exists N. \forall n \geq N. x < X n\}$
then **have** $mem-S: \bigwedge N x. \forall n \geq N. x < X n \implies x \in S$
by *auto*

have *bound-isUb*: $y \leq x$ **if** $N: \forall n \geq N. X n < x$ **and** $y \in S$ **for** N **and** $x\ y :: real$

proof –
from *that* **have** $\exists M. \forall n \geq M. y < X n$
by (simp add: S-def)
then **obtain** M **where** $\forall n \geq M. y < X n ..$
then **have** $y < X (max\ M\ N)$ **by** *simp*
also **have** $\dots < x$ **using** N **by** *simp*
finally **show** *?thesis* **by** (rule order-less-imp-le)
qed

obtain N **where** $\forall m \geq N. \forall n \geq N. dist\ (X\ m)\ (X\ n) < 1$
using $X[THEN\ metric-CauchyD, OF\ zero-less-one]$ **by** *auto*
then **have** $N: \forall n \geq N. dist\ (X\ n)\ (X\ N) < 1$ **by** *simp*
have [simp]: $S \neq \{\}$
proof (intro exI ex-in-conv[THEN iffD1])
from N **have** $\forall n \geq N. X\ N - 1 < X\ n$
by (simp add: abs-diff-less-iff dist-real-def)
then **show** $X\ N - 1 \in S$ **by** (rule mem-S)
qed
have [simp]: *bdd-above* S
proof
from N **have** $\forall n \geq N. X\ n < X\ N + 1$
by (simp add: abs-diff-less-iff dist-real-def)
then **show** $\bigwedge s. s \in S \implies s \leq X\ N + 1$
by (rule bound-isUb)

```

qed
have  $X \longrightarrow \text{Sup } S$ 
proof (rule metric-LIMSEQ-I)
  fix  $r :: \text{real}$ 
  assume  $0 < r$ 
  then have  $r: 0 < r/2$  by simp
  obtain  $N$  where  $\forall n \geq N. \forall m \geq N. \text{dist } (X\ n) (X\ m) < r/2$ 
    using metric-CauchyD [OF  $X\ r$ ] by auto
  then have  $\forall n \geq N. \text{dist } (X\ n) (X\ N) < r/2$  by simp
  then have  $N: \forall n \geq N. X\ N - r/2 < X\ n \wedge X\ n < X\ N + r/2$ 
    by (simp only: dist-real-def abs-diff-less-iff)

  from  $N$  have  $\forall n \geq N. X\ N - r/2 < X\ n$  by blast
  then have  $X\ N - r/2 \in S$  by (rule mem-S)
  then have  $1: X\ N - r/2 \leq \text{Sup } S$  by (simp add: cSup-upper)

  from  $N$  have  $\forall n \geq N. X\ n < X\ N + r/2$  by blast
  from bound-isUb[OF this]
  have  $2: \text{Sup } S \leq X\ N + r/2$ 
    by (intro cSup-least) simp-all

  show  $\exists N. \forall n \geq N. \text{dist } (X\ n) (\text{Sup } S) < r$ 
  proof (intro exI allI impI)
    fix  $n$ 
    assume  $n: N \leq n$ 
    from  $N\ n$  have  $X\ n < X\ N + r/2$  and  $X\ N - r/2 < X\ n$ 
      by simp-all
    then show  $\text{dist } (X\ n) (\text{Sup } S) < r$  using 1 2
      by (simp add: abs-diff-less-iff dist-real-def)
  qed
qed
then show ?thesis by (auto simp: convergent-def)
qed

instance real :: complete-space
  by intro-classes (rule real-Cauchy-convergent)

class banach = real-normed-vector + complete-space

instance real :: banach ..

lemma tendsto-at-topI-sequentially:
  fixes  $f :: \text{real} \Rightarrow 'b::\text{first-countable-topology}$ 
  assumes *:  $\bigwedge X. \text{filterlim } X \text{ at-top sequentially} \implies (\lambda n. f (X\ n)) \longrightarrow y$ 
  shows  $(f \longrightarrow y) \text{ at-top}$ 
proof -
  obtain  $A$  where  $A: \text{decseq } A \text{ open } (A\ n) \ y \in A\ n \ \text{nhds } y = (\text{INF } n. \text{principal } (A\ n))$ 
    for  $n$ 
    by (rule nhds-countable[of  $y$ ]) (rule that)

```

```

have  $\forall m. \exists k. \forall x \geq k. f x \in A m$ 
proof (rule ccontr)
  assume  $\neg (\forall m. \exists k. \forall x \geq k. f x \in A m)$ 
  then obtain  $m$  where  $\bigwedge k. \exists x \geq k. f x \notin A m$ 
  by auto
  then have  $\exists X. \forall n. (f (X n) \notin A m) \wedge \max n (X n) + 1 \leq X (Suc n)$ 
  by (intro dependent-nat-choice) (auto simp del: max.bounded-iff)
  then obtain  $X$  where  $X: \bigwedge n. f (X n) \notin A m \wedge n. \max n (X n) + 1 \leq X$ 
  (Suc n)
  by auto
  have  $1 \leq n \implies \text{real } n \leq X n$  for  $n$ 
  using  $X[of\ n - 1]$  by auto
  then have filterlim  $X$  at-top sequentially
  by (force intro!: filterlim-at-top-mono[OF filterlim-real-sequentially]
    simp: eventually-sequentially)
  from topological-tendstoD[OF *[OF this] A(2, 3), of  $m$ ]  $X(1)$  show False
  by auto
qed
then obtain  $k$  where  $k m \leq x \implies f x \in A m$  for  $m x$ 
by metis
then show ?thesis
  unfolding at-top-def A by (intro filterlim-base[where  $i=k$ ]) auto
qed

```

lemma tendsto-at-topI-sequentially-real:

```

fixes  $f :: \text{real} \Rightarrow \text{real}$ 
assumes mono: mono  $f$ 
  and limseq:  $(\lambda n. f (\text{real } n)) \longrightarrow y$ 
shows  $(f \longrightarrow y)$  at-top
proof (rule tendstoI)
  fix  $e :: \text{real}$ 
  assume  $0 < e$ 
  with limseq obtain  $N :: \text{nat}$  where  $N: N \leq n \implies |f (\text{real } n) - y| < e$  for  $n$ 
  by (auto simp: lim-sequentially-dist-real-def)
  have  $le: f x \leq y$  for  $x :: \text{real}$ 
  proof -
    obtain  $n$  where  $x \leq \text{real-of-nat } n$ 
    using real-arch-simple[of  $x$ ] ..
    note monoD[OF mono this]
    also have  $f (\text{real-of-nat } n) \leq y$ 
    by (rule LIMSEQ-le-const[OF limseq]) (auto intro!: exI[of -] monoD[OF
mono])
  finally show ?thesis .
qed
have eventually  $(\lambda x. \text{real } N \leq x)$  at-top
by (rule eventually-ge-at-top)
then show eventually  $(\lambda x. \text{dist } (f x) y < e)$  at-top
proof eventually-elim

```



```

    case (elim x)
    with N[of N] le have y - f (real N) < e by auto
    moreover note monoD[OF mono elim]
    ultimately show dist (f x) y < e
      using le[of x] by (auto simp: dist-real-def field-simps)
  qed
qed
end

```

109 Limits on Real Vector Spaces

```

theory Limits
  imports Real-Vector-Spaces
begin

```

```

lemma range-mult [simp]:
  fixes a::real shows range ((* a) = (if a=0 then {0} else UNIV)
  by (simp add: surj-def) (meson dvdE dvd-field-iff)

```

109.1 Filter going to infinity norm

```

definition at-infinity :: 'a::real-normed-vector filter
  where at-infinity = (INF r. principal {x. r ≤ norm x})

```

```

lemma eventually-at-infinity: eventually P at-infinity ↔ (∃ b. ∀ x. b ≤ norm x
→ P x)
  unfolding at-infinity-def
  by (subst eventually-INF-base)
  (auto simp: subset-eq eventually-principal intro!: exI[of - max a b for a b])

```

```

corollary eventually-at-infinity-pos:
  eventually p at-infinity ↔ (∃ b. 0 < b ∧ (∀ x. norm x ≥ b → p x))
  unfolding eventually-at-infinity
  by (meson le-less-trans norm-ge-zero not-le zero-less-one)

```

```

lemma at-infinity-eq-at-top-bot: (at-infinity :: real filter) = sup at-top at-bot

```

proof –

```

  have 1: [∀ n ≥ u. A n; ∀ n ≤ v. A n]
    ⇒ ∃ b. ∀ x. b ≤ |x| → A x for A and u v::real
    by (rule-tac x=max (- v) u in exI) (auto simp: abs-real-def)
  have 2: ∀ x. u ≤ |x| → A x ⇒ ∃ N. ∀ n ≥ N. A n for A and u::real
    by (meson abs-less-iff le-cases less-le-not-le)
  have 3: ∀ x. u ≤ |x| → A x ⇒ ∃ N. ∀ n ≤ N. A n for A and u::real
    by (metis (full-types) abs-ge-self abs-minus-cancel le-minus-iff order-trans)
  show ?thesis
    by (auto simp: filter-eq-iff eventually-sup eventually-at-infinity
      eventually-at-top-linorder eventually-at-bot-linorder intro: 1 2 3)
qed

```

lemma *at-top-le-at-infinity*: $at\text{-}top \leq (at\text{-}infinity :: real\ filter)$
unfolding *at-infinity-eq-at-top-bot* **by** *simp*

lemma *at-bot-le-at-infinity*: $at\text{-}bot \leq (at\text{-}infinity :: real\ filter)$
unfolding *at-infinity-eq-at-top-bot* **by** *simp*

lemma *filterlim-at-top-imp-at-infinity*: $filterlim\ f\ at\text{-}top\ F \implies filterlim\ f\ at\text{-}infinity\ F$
for $f :: - \Rightarrow real$
by (*rule filterlim-mono*[*OF - at-top-le-at-infinity order-refl*])

lemma *filterlim-real-at-infinity-sequentially*: $filterlim\ real\ at\text{-}infinity\ sequentially$
by (*simp add: filterlim-at-top-imp-at-infinity filterlim-real-sequentially*)

lemma *lim-infinity-imp-sequentially*: $(f \longrightarrow l)\ at\text{-}infinity \implies ((\lambda n. f(n)) \longrightarrow l)\ sequentially$
by (*simp add: filterlim-at-top-imp-at-infinity filterlim-compose filterlim-real-sequentially*)

109.1.1 Boundedness

definition *Bfun* :: $('a \Rightarrow 'b::metric\ space) \Rightarrow 'a\ filter \Rightarrow bool$
where *Bfun-metric-def*: $Bfun\ f\ F = (\exists y. \exists K > 0. eventually\ (\lambda x. dist\ (f\ x)\ y \leq K)\ F)$

abbreviation *Bseq* :: $(nat \Rightarrow 'a::metric\ space) \Rightarrow bool$
where $Bseq\ X \equiv Bfun\ X\ sequentially$

lemma *Bseq-conv-Bfun*: $Bseq\ X \longleftrightarrow Bfun\ X\ sequentially ..$

lemma *Bseq-ignore-initial-segment*: $Bseq\ X \implies Bseq\ (\lambda n. X\ (n + k))$
unfolding *Bfun-metric-def* **by** (*subst eventually-sequentially-seg*)

lemma *Bseq-offset*: $Bseq\ (\lambda n. X\ (n + k)) \implies Bseq\ X$
unfolding *Bfun-metric-def* **by** (*subst (asm) eventually-sequentially-seg*)

lemma *Bfun-def*: $Bfun\ f\ F \longleftrightarrow (\exists K > 0. eventually\ (\lambda x. norm\ (f\ x) \leq K)\ F)$
unfolding *Bfun-metric-def norm-conv-dist*

proof *safe*

fix $y\ K$

assume $K: 0 < K$ **and** $*$: $eventually\ (\lambda x. dist\ (f\ x)\ y \leq K)\ F$

moreover have $eventually\ (\lambda x. dist\ (f\ x)\ 0 \leq dist\ (f\ x)\ y + dist\ 0\ y)\ F$

by (*intro always-eventually*) (*metis dist-commute dist-triangle*)

with $*$ **have** $eventually\ (\lambda x. dist\ (f\ x)\ 0 \leq K + dist\ 0\ y)\ F$

by *eventually-elim auto*

with $\langle 0 < K \rangle$ **show** $\exists K > 0. eventually\ (\lambda x. dist\ (f\ x)\ 0 \leq K)\ F$

by (*intro exI*[*of - K + dist 0 y*] *add-pos-nonneg conjI zero-le-dist*) *auto*

qed (*force simp del: norm-conv-dist [symmetric]*)

lemma *BfunI*:
assumes K : eventually $(\lambda x. \text{norm } (f x) \leq K)$ F
shows $Bfun\ f\ F$
unfolding *Bfun-def*
proof (*intro exI conjI allI*)
show $0 < \max K\ 1$ **by** *simp*
show eventually $(\lambda x. \text{norm } (f x) \leq \max K\ 1)$ F
using K **by** (*rule eventually-mono*) *simp*
qed

lemma *BfunE*:
assumes $Bfun\ f\ F$
obtains B **where** $0 < B$ **and** eventually $(\lambda x. \text{norm } (f x) \leq B)$ F
using *assms* **unfolding** *Bfun-def* **by** *blast*

lemma *Cauchy-Bseq*:
assumes *Cauchy* X **shows** $Bseq\ X$
proof –
have $\exists y\ K. 0 < K \wedge (\exists N. \forall n \geq N. \text{dist } (X\ n)\ y \leq K)$
if $\bigwedge m\ n. \llbracket m \geq M; n \geq M \rrbracket \implies \text{dist } (X\ m)\ (X\ n) < 1$ **for** M
by (*meson order.order-iff-strict that zero-less-one*)
with *assms* **show** *?thesis*
by (*force simp: Cauchy-def Bfun-metric-def eventually-sequentially*)
qed

109.1.2 Bounded Sequences

lemma *BseqI'*: $(\bigwedge n. \text{norm } (X\ n) \leq K) \implies Bseq\ X$
by (*intro BfunI*) (*auto simp: eventually-sequentially*)

lemma *Bseq-def*: $Bseq\ X \longleftrightarrow (\exists K > 0. \forall n. \text{norm } (X\ n) \leq K)$
unfolding *Bfun-def* *eventually-sequentially*
proof *safe*
fix $N\ K$
assume $0 < K \forall n \geq N. \text{norm } (X\ n) \leq K$
then show $\exists K > 0. \forall n. \text{norm } (X\ n) \leq K$
by (*intro exI[of - max (Max (norm ‘ X ‘ {..N})) K] max.strict-coboundedI2*)
(auto intro!: imageI not-less[where 'a=nat, THEN iffD1] Max-ge simp: le-max-iff-disj)
qed *auto*

lemma *BseqE*: $Bseq\ X \implies (\bigwedge K. 0 < K \implies \forall n. \text{norm } (X\ n) \leq K \implies Q) \implies Q$
unfolding *Bseq-def* **by** *auto*

lemma *BseqD*: $Bseq\ X \implies \exists K. 0 < K \wedge (\forall n. \text{norm } (X\ n) \leq K)$
by (*simp add: Bseq-def*)

lemma *BseqI*: $0 < K \implies \forall n. \text{norm } (X\ n) \leq K \implies Bseq\ X$

by (auto simp: Bseq-def)

lemma *Bseq-bdd-above*: $Bseq\ X \implies bdd-above\ (range\ X)$
 for $X :: nat \Rightarrow real$
proof (elim BseqE, intro bdd-aboveI2)
 fix $K\ n$
 assume $0 < K \ \forall n. norm\ (X\ n) \leq K$
 then show $X\ n \leq K$
 by (auto elim!: allE[of - n])
qed

lemma *Bseq-bdd-above'*: $Bseq\ X \implies bdd-above\ (range\ (\lambda n. norm\ (X\ n)))$
 for $X :: nat \Rightarrow 'a :: real-normed-vector$
proof (elim BseqE, intro bdd-aboveI2)
 fix $K\ n$
 assume $0 < K \ \forall n. norm\ (X\ n) \leq K$
 then show $norm\ (X\ n) \leq K$
 by (auto elim!: allE[of - n])
qed

lemma *Bseq-bdd-below*: $Bseq\ X \implies bdd-below\ (range\ X)$
 for $X :: nat \Rightarrow real$
proof (elim BseqE, intro bdd-belowI2)
 fix $K\ n$
 assume $0 < K \ \forall n. norm\ (X\ n) \leq K$
 then show $-K \leq X\ n$
 by (auto elim!: allE[of - n])
qed

lemma *Bseq-eventually-mono*:
 assumes eventually $(\lambda n. norm\ (f\ n) \leq norm\ (g\ n))$ sequentially Bseq g
 shows Bseq f
proof –
 from *assms*(2) obtain K where $0 < K$ and eventually $(\lambda n. norm\ (g\ n) \leq K)$
 sequentially
 unfolding *Bfun-def* by fast
 with *assms*(1) have eventually $(\lambda n. norm\ (f\ n) \leq K)$ sequentially
 by (fast elim: eventually-elim2 order-trans)
 with $\langle 0 < K \rangle$ show Bseq f
 unfolding *Bfun-def* by fast
qed

lemma *lemma-NBseq-def*: $(\exists K > 0. \forall n. norm\ (X\ n) \leq K) \longleftrightarrow (\exists N. \forall n. norm\ (X\ n) \leq real(Suc\ N))$
proof safe
 fix $K :: real$
 from *reals-Archimedean2* obtain $n :: nat$ where $K < real\ n ..$
 then have $K \leq real\ (Suc\ n)$ by auto
 moreover assume $\forall m. norm\ (X\ m) \leq K$

ultimately have $\forall n. \text{norm } (X\ n) \leq \text{real } (\text{Suc } n)$
by (*blast intro: order-trans*)
then show $\exists N. \forall n. \text{norm } (X\ n) \leq \text{real } (\text{Suc } N) \dots$
next
show $\bigwedge N. \forall n. \text{norm } (X\ n) \leq \text{real } (\text{Suc } N) \implies \exists K > 0. \forall n. \text{norm } (X\ n) \leq K$
using *of-nat-0-less-iff* **by** *blast*
qed

Alternative definition for *Bseq*.

lemma *Bseq-iff*: $Bseq\ X \longleftrightarrow (\exists N. \forall n. \text{norm } (X\ n) \leq \text{real } (\text{Suc } N))$
by (*simp add: Bseq-def*) (*simp add: lemma-NBseq-def*)

lemma *lemma-NBseq-def2*: $(\exists K > 0. \forall n. \text{norm } (X\ n) \leq K) = (\exists N. \forall n. \text{norm } (X\ n) < \text{real } (\text{Suc } N))$

proof –

have $*$: $\bigwedge N. \forall n. \text{norm } (X\ n) \leq 1 + \text{real } N \implies$
 $\exists N. \forall n. \text{norm } (X\ n) < 1 + \text{real } N$
by (*metis add.commute le-less-trans less-add-one of-nat-Suc*)
then show *?thesis*
unfolding *lemma-NBseq-def*
by (*metis less-le-not-le not-less-iff-gr-or-eq of-nat-Suc*)
qed

Yet another definition for *Bseq*.

lemma *Bseq-iff1a*: $Bseq\ X \longleftrightarrow (\exists N. \forall n. \text{norm } (X\ n) < \text{real } (\text{Suc } N))$
by (*simp add: Bseq-def lemma-NBseq-def2*)

109.1.3 A Few More Equivalence Theorems for Boundedness

Alternative formulation for boundedness.

lemma *Bseq-iff2*: $Bseq\ X \longleftrightarrow (\exists k > 0. \exists x. \forall n. \text{norm } (X\ n + -\ x) \leq k)$
by (*metis BseqE BseqI' add.commute add-cancel-right-left add-uminus-conv-diff norm-add-leD norm-minus-cancel norm-minus-commute*)

Alternative formulation for boundedness.

lemma *Bseq-iff3*: $Bseq\ X \longleftrightarrow (\exists k > 0. \exists N. \forall n. \text{norm } (X\ n + -\ X\ N) \leq k)$
(is *?P* \longleftrightarrow *?Q***)**

proof

assume *?P*
then obtain K **where** $*$: $0 < K$ **and** $**$: $\bigwedge n. \text{norm } (X\ n) \leq K$
by (*auto simp: Bseq-def*)
from $*$ **have** $0 < K + \text{norm } (X\ 0)$ **by** (*rule order-less-le-trans*) *simp*
from $**$ **have** $\forall n. \text{norm } (X\ n - X\ 0) \leq K + \text{norm } (X\ 0)$
by (*auto intro: order-trans norm-triangle-ineq4*)
then have $\forall n. \text{norm } (X\ n + -\ X\ 0) \leq K + \text{norm } (X\ 0)$
by *simp*
with $\langle 0 < K + \text{norm } (X\ 0) \rangle$ **show** *?Q* **by** *blast*

```

next
  assume ?Q
  then show ?P by (auto simp: Bseq-iff2)
qed

```

109.1.4 Upper Bounds and Lubs of Bounded Sequences

```

lemma Bseq-minus-iff: Bseq ( $\lambda n. - (X n) :: 'a::real-normed-vector$ )  $\longleftrightarrow$  Bseq X
  by (simp add: Bseq-def)

```

```

lemma Bseq-add:
  fixes f :: nat  $\Rightarrow$  'a::real-normed-vector
  assumes Bseq f
  shows Bseq ( $\lambda x. f x + c$ )
proof -
  from assms obtain K where K:  $\bigwedge x. \text{norm } (f x) \leq K$ 
  unfolding Bseq-def by blast
  {
    fix x :: nat
    have  $\text{norm } (f x + c) \leq \text{norm } (f x) + \text{norm } c$  by (rule norm-triangle-ineq)
    also have  $\text{norm } (f x) \leq K$  by (rule K)
    finally have  $\text{norm } (f x + c) \leq K + \text{norm } c$  by simp
  }
  then show ?thesis by (rule BseqI')
qed

```

```

lemma Bseq-add-iff: Bseq ( $\lambda x. f x + c$ )  $\longleftrightarrow$  Bseq f
  for f :: nat  $\Rightarrow$  'a::real-normed-vector
  using Bseq-add[of f c] Bseq-add[of  $\lambda x. f x + c - c$ ] by auto

```

```

lemma Bseq-mult:
  fixes f g :: nat  $\Rightarrow$  'a::real-normed-field
  assumes Bseq f and Bseq g
  shows Bseq ( $\lambda x. f x * g x$ )
proof -
  from assms obtain K1 K2 where K:  $\text{norm } (f x) \leq K1$   $K1 > 0$   $\text{norm } (g x) \leq K2$   $K2 > 0$ 
  for x
  unfolding Bseq-def by blast
  then have  $\text{norm } (f x * g x) \leq K1 * K2$  for x
  by (auto simp: norm-mult intro!: mult-mono)
  then show ?thesis by (rule BseqI')
qed

```

```

lemma Bfun-const [simp]: Bfun ( $\lambda -. c$ ) F
  unfolding Bfun-metric-def by (auto intro!: exI[of - c] exI[of - 1::real])

```

```

lemma Bseq-cmult-iff:
  fixes c :: 'a::real-normed-field

```

```

assumes  $c \neq 0$ 
shows  $Bseq (\lambda x. c * f x) \longleftrightarrow Bseq f$ 
proof
  assume  $Bseq (\lambda x. c * f x)$ 
  with  $Bfun-const$  have  $Bseq (\lambda x. inverse\ c * (c * f x))$ 
    by (rule  $Bseq-mult$ )
  with  $\langle c \neq 0 \rangle$  show  $Bseq f$ 
    by (simp add: field-split-simps)
qed (intro  $Bseq-mult\ Bfun-const$ )

lemma  $Bseq-subseq$ :  $Bseq f \implies Bseq (\lambda x. f (g x))$ 
  for  $f :: nat \Rightarrow 'a::real-normed-vector$ 
  unfolding  $Bseq-def$  by auto

lemma  $Bseq-Suc-iff$ :  $Bseq (\lambda n. f (Suc\ n)) \longleftrightarrow Bseq f$ 
  for  $f :: nat \Rightarrow 'a::real-normed-vector$ 
  using  $Bseq-offset[of\ f\ 1]$  by (auto intro:  $Bseq-subseq$ )

lemma  $increasing-Bseq-subseq-iff$ :
  assumes  $\bigwedge x\ y. x \leq y \implies norm (f\ x :: 'a::real-normed-vector) \leq norm (f\ y)$ 
  strict-mono  $g$ 
  shows  $Bseq (\lambda x. f (g\ x)) \longleftrightarrow Bseq f$ 
proof
  assume  $Bseq (\lambda x. f (g\ x))$ 
  then obtain  $K$  where  $K: \bigwedge x. norm (f (g\ x)) \leq K$ 
    unfolding  $Bseq-def$  by auto
  {
    fix  $x :: nat$ 
    from  $filterlim-subseq[OF\ assms(2)]$  obtain  $y$  where  $g\ y \geq x$ 
      by (auto simp: filterlim-at-top eventually-at-top-linorder)
    then have  $norm (f\ x) \leq norm (f (g\ y))$ 
      using  $assms(1)$  by blast
    also have  $norm (f (g\ y)) \leq K$  by (rule  $K$ )
    finally have  $norm (f\ x) \leq K$  .
  }
  then show  $Bseq f$  by (rule  $BseqI'$ )
qed (use  $Bseq-subseq[of\ f\ g]$  in simp-all)

lemma  $nonneg-incseq-Bseq-subseq-iff$ :
  fixes  $f :: nat \Rightarrow real$ 
  and  $g :: nat \Rightarrow nat$ 
  assumes  $\bigwedge x. f\ x \geq 0$  incseq  $f$  strict-mono  $g$ 
  shows  $Bseq (\lambda x. f (g\ x)) \longleftrightarrow Bseq f$ 
  using  $assms$  by (intro  $increasing-Bseq-subseq-iff$ ) (auto simp:  $incseq-def$ )

lemma  $Bseq-eq-bounded$ :  $range\ f \subseteq \{a..b\} \implies Bseq f$ 
  for  $a\ b :: real$ 
proof (rule  $BseqI'$  [where  $K = \max (norm\ a) (norm\ b)$ ])
  fix  $n$  assume  $range\ f \subseteq \{a..b\}$ 

```

```

then have  $f\ n \in \{a..b\}$ 
  by blast
then show  $\text{norm } (f\ n) \leq \max (\text{norm } a) (\text{norm } b)$ 
  by auto
qed

```

```

lemma incseq-bounded:  $\text{incseq } X \implies \forall i. X\ i \leq B \implies B\text{seq } X$ 
for  $B :: \text{real}$ 
by (intro Bseq-eg-bounded[of X X 0 B]) (auto simp: incseq-def)

```

```

lemma decseq-bounded:  $\text{decseq } X \implies \forall i. B \leq X\ i \implies B\text{seq } X$ 
for  $B :: \text{real}$ 
by (intro Bseq-eg-bounded[of X B X 0]) (auto simp: decseq-def)

```

109.1.5 Polynomial function extremal theorem, from HOL Light

```

lemma polyfun-extremal-lemma:

```

```

  fixes  $c :: \text{nat} \Rightarrow 'a :: \text{real-normed-div-algebra}$ 
  assumes  $0 < e$ 
  shows  $\exists M. \forall z. M \leq \text{norm}(z) \longrightarrow \text{norm } (\sum_{i \leq n}. c(i) * z^{\wedge}i) \leq e * \text{norm}(z)$ 
   $^{\wedge}(\text{Suc } n)$ 

```

```

proof (induct n)

```

```

  case 0 with assms

```

```

    show ?case

```

```

      apply (rule-tac x=norm (c 0) / e in exI)

```

```

      apply (auto simp: field-simps)

```

```

      done

```

```

next

```

```

  case (Suc n)

```

```

  obtain  $M$  where  $M: \bigwedge z. M \leq \text{norm } z \implies \text{norm } (\sum_{i \leq n}. c\ i * z^{\wedge}i) \leq e * \text{norm } z$ 
   $^{\wedge} \text{Suc } n$ 

```

```

    using Suc assms by blast

```

```

    show ?case

```

```

    proof (rule exI [where  $x = \max M (1 + \text{norm}(c(\text{Suc } n)) / e)$ ], clarsimp simp
  del: power-Suc)

```

```

      fix  $z :: 'a$ 

```

```

      assume  $z1: M \leq \text{norm } z$  and  $1 + \text{norm } (c (\text{Suc } n)) / e \leq \text{norm } z$ 

```

```

      then have  $z2: e + \text{norm } (c (\text{Suc } n)) \leq e * \text{norm } z$ 

```

```

        using assms by (simp add: field-simps)

```

```

      have  $\text{norm } (\sum_{i \leq n}. c\ i * z^{\wedge}i) \leq e * \text{norm } z^{\wedge} \text{Suc } n$ 

```

```

        using  $M$  [OF z1] by simp

```

```

      then have  $\text{norm } (\sum_{i \leq n}. c\ i * z^{\wedge}i) + \text{norm } (c (\text{Suc } n) * z^{\wedge} \text{Suc } n) \leq e * \text{norm } z^{\wedge} \text{Suc } n + \text{norm } (c (\text{Suc } n) * z^{\wedge} \text{Suc } n)$ 

```

```

        by simp

```

```

      then have  $\text{norm } ((\sum_{i \leq n}. c\ i * z^{\wedge}i) + c (\text{Suc } n) * z^{\wedge} \text{Suc } n) \leq e * \text{norm } z^{\wedge} \text{Suc } n + \text{norm } (c (\text{Suc } n) * z^{\wedge} \text{Suc } n)$ 

```

```

        by (blast intro: norm-triangle-le elim:)

```

```

      also have  $\dots \leq (e + \text{norm } (c (\text{Suc } n))) * \text{norm } z^{\wedge} \text{Suc } n$ 

```

```

        by (simp add: norm-power norm-mult algebra-simps)

```


also have $\dots \leq (e * \text{norm } z) * \text{norm } z \wedge \text{Suc } n$
by (*metis z2 mult.commute mult-left-mono norm-ge-zero norm-power*)
finally show $\text{norm } ((\sum_{i \leq n}. c \ i * z \wedge i) + c \ (\text{Suc } n) * z \wedge \text{Suc } n) \leq e * \text{norm } z \wedge \text{Suc } (\text{Suc } n)$
by simp
qed
qed

lemma polyfun-extremal:

fixes $c :: \text{nat} \Rightarrow 'a::\text{real-normed-div-algebra}$
assumes $k: c \ k \neq 0 \ 1 \leq k$ **and** $kn: k \leq n$
shows eventually $(\lambda z. \text{norm } (\sum_{i \leq n}. c(i) * z \wedge i) \geq B)$ *at-infinity*
using kn
proof (*induction n*)
case 0
then show *?case*
using k **by simp**
next
case ($\text{Suc } m$)
show *?case*
proof (*cases c (Suc m) = 0*)
case True
then show *?thesis using Suc k*
by auto (*metis antisym-conv less-eq-Suc-le not-le*)
next
case False
then obtain M where M:
 $\bigwedge z. M \leq \text{norm } z \implies \text{norm } (\sum_{i \leq m}. c \ i * z \wedge i) \leq \text{norm } (c \ (\text{Suc } m)) / 2$
 $* \text{norm } z \wedge \text{Suc } m$
using *polyfun-extremal-lemma [of norm(c (Suc m)) / 2 c m] Suc*
by auto
have $\exists b. \forall z. b \leq \text{norm } z \implies B \leq \text{norm } (\sum_{i \leq \text{Suc } m}. c \ i * z \wedge i)$
proof (*rule exI [where x=max M (max 1 (|B| / (norm(c (Suc m)) / 2)))]*),
clarsimp simp del: power-Suc
fix $z::'a$
assume $z1: M \leq \text{norm } z \ 1 \leq \text{norm } z$
and $|B| * 2 / \text{norm } (c \ (\text{Suc } m)) \leq \text{norm } z$
then have $z2: |B| \leq \text{norm } (c \ (\text{Suc } m)) * \text{norm } z / 2$
using *False by (simp add: field-simps)*
have $nz: \text{norm } z \leq \text{norm } z \wedge \text{Suc } m$
by (*metis <1 ≤ norm z> One-nat-def less-eq-Suc-le power-increasing power-one-right zero-less-Suc*)
have $*$: $\bigwedge y \ x. \text{norm } (c \ (\text{Suc } m)) * \text{norm } z / 2 \leq \text{norm } y - \text{norm } x \implies B \leq \text{norm } (x + y)$
by (*metis abs-le-iff add.commute norm-diff-ineq order-trans z2*)
have $\text{norm } z * \text{norm } (c \ (\text{Suc } m)) + 2 * \text{norm } (\sum_{i \leq m}. c \ i * z \wedge i) \leq \text{norm } (c \ (\text{Suc } m)) * \text{norm } z + \text{norm } (c \ (\text{Suc } m)) * \text{norm } z \wedge \text{Suc } m$
using M [*of z*] $\text{Suc } z1$ **by auto**
also have $\dots \leq 2 * (\text{norm } (c \ (\text{Suc } m)) * \text{norm } z \wedge \text{Suc } m)$

```

    using nz by (simp add: mult-mono del: power-Suc)
  finally show  $B \leq \text{norm } ((\sum_{i \leq m}. c\ i * z^{\wedge} i) + c\ (\text{Suc } m) * z^{\wedge} \text{Suc } m)$ 
    using Suc.IH
  apply (auto simp: eventually-at-infinity)
  apply (rule *)
  apply (simp add: field-simps norm-mult norm-power)
  done
qed
then show ?thesis
  by (simp add: eventually-at-infinity)
qed
qed

```

109.2 Convergence to Zero

definition $Zfun :: ('a \Rightarrow 'b::\text{real-normed-vector}) \Rightarrow 'a\ \text{filter} \Rightarrow \text{bool}$
 where $Zfun\ f\ F = (\forall r > 0. \text{eventually } (\lambda x. \text{norm } (f\ x) < r)\ F)$

lemma $ZfunI: (\bigwedge r. 0 < r \Longrightarrow \text{eventually } (\lambda x. \text{norm } (f\ x) < r)\ F) \Longrightarrow Zfun\ f\ F$
 by (simp add: Zfun-def)

lemma $ZfunD: Zfun\ f\ F \Longrightarrow 0 < r \Longrightarrow \text{eventually } (\lambda x. \text{norm } (f\ x) < r)\ F$
 by (simp add: Zfun-def)

lemma $Zfun-ssubst: \text{eventually } (\lambda x. f\ x = g\ x)\ F \Longrightarrow Zfun\ g\ F \Longrightarrow Zfun\ f\ F$
 unfolding Zfun-def by (auto elim!: eventually-rew-mp)

lemma $Zfun-zero: Zfun\ (\lambda x. 0)\ F$
 unfolding Zfun-def by simp

lemma $Zfun-norm-iff: Zfun\ (\lambda x. \text{norm } (f\ x))\ F = Zfun\ (\lambda x. f\ x)\ F$
 unfolding Zfun-def by simp

lemma $Zfun-imp-Zfun:$
 assumes $f: Zfun\ f\ F$
 and $g: \text{eventually } (\lambda x. \text{norm } (g\ x) \leq \text{norm } (f\ x) * K)\ F$
 shows $Zfun\ (\lambda x. g\ x)\ F$
proof (cases $0 < K$)
 case $K: \text{True}$
 show ?thesis
proof (rule ZfunI)
 fix $r :: \text{real}$
 assume $0 < r$
 then have $0 < r / K$ using K by simp
 then have $\text{eventually } (\lambda x. \text{norm } (f\ x) < r / K)\ F$
 using ZfunD [OF f] by blast
 with g show $\text{eventually } (\lambda x. \text{norm } (g\ x) < r)\ F$
proof eventually-elim
 case (elim x)

```

    then have  $\text{norm } (f x) * K < r$ 
      by (simp add: pos-less-divide-eq K)
    then show ?case
      by (simp add: order-le-less-trans [OF elim(1)])
  qed
qed
next
case False
then have  $K: K \leq 0$  by (simp only: not-less)
show ?thesis
proof (rule ZfunI)
  fix  $r :: \text{real}$ 
  assume  $0 < r$ 
  from  $g$  show eventually  $(\lambda x. \text{norm } (g x) < r)$   $F$ 
  proof eventually-elim
    case (elim  $x$ )
    also have  $\text{norm } (f x) * K \leq \text{norm } (f x) * 0$ 
      using  $K$  norm-ge-zero by (rule mult-left-mono)
    finally show ?case
      using  $\langle 0 < r \rangle$  by simp
  qed
qed
qed
qed

lemma Zfun-le:  $Zfun\ g\ F \implies \forall x. \text{norm } (f x) \leq \text{norm } (g x) \implies Zfun\ f\ F$ 
  by (erule Zfun-imp-Zfun [where  $K = 1$ ]) simp

lemma Zfun-add:
  assumes  $f: Zfun\ f\ F$ 
  and  $g: Zfun\ g\ F$ 
  shows  $Zfun\ (\lambda x. f\ x + g\ x)\ F$ 
proof (rule ZfunI)
  fix  $r :: \text{real}$ 
  assume  $0 < r$ 
  then have  $r: 0 < r / 2$  by simp
  have eventually  $(\lambda x. \text{norm } (f x) < r/2)$   $F$ 
    using  $f\ r$  by (rule ZfunD)
  moreover
  have eventually  $(\lambda x. \text{norm } (g x) < r/2)$   $F$ 
    using  $g\ r$  by (rule ZfunD)
  ultimately
  show eventually  $(\lambda x. \text{norm } (f x + g x) < r)$   $F$ 
  proof eventually-elim
    case (elim  $x$ )
    have  $\text{norm } (f x + g x) \leq \text{norm } (f x) + \text{norm } (g x)$ 
      by (rule norm-triangle-ineq)
    also have  $\dots < r/2 + r/2$ 
      using elim by (rule add-strict-mono)
    finally show ?case
  end
end

```

by *simp*
 qed
 qed

lemma *Zfun-minus*: $Zfun\ f\ F \implies Zfun\ (\lambda x. -\ f\ x)\ F$
 unfolding *Zfun-def* by *simp*

lemma *Zfun-diff*: $Zfun\ f\ F \implies Zfun\ g\ F \implies Zfun\ (\lambda x. f\ x -\ g\ x)\ F$
 using *Zfun-add* [of *f F* $\lambda x. -\ g\ x$] by (*simp add: Zfun-minus*)

lemma (in *bounded-linear*) *Zfun*:

assumes $g: Zfun\ g\ F$
 shows $Zfun\ (\lambda x. f\ (g\ x))\ F$

proof –

obtain K where $norm\ (f\ x) \leq norm\ x * K$ for x
 using *bounded* by *blast*
 then have eventually $(\lambda x. norm\ (f\ (g\ x)) \leq norm\ (g\ x) * K)$ F
 by *simp*
 with g show *?thesis*
 by (*rule Zfun-imp-Zfun*)

qed

lemma (in *bounded-bilinear*) *Zfun*:

assumes $f: Zfun\ f\ F$
 and $g: Zfun\ g\ F$
 shows $Zfun\ (\lambda x. f\ x **\ g\ x)\ F$

proof (*rule ZfunI*)

fix $r :: real$
 assume $r: 0 < r$
 obtain K where $K: 0 < K$
 and *norm-le*: $norm\ (x **\ y) \leq norm\ x * norm\ y * K$ for $x\ y$
 using *pos-bounded* by *blast*
 from K have $K': 0 < inverse\ K$
 by (*rule positive-imp-inverse-positive*)
 have eventually $(\lambda x. norm\ (f\ x) < r)$ F
 using $f\ r$ by (*rule ZfunD*)

moreover

have eventually $(\lambda x. norm\ (g\ x) < inverse\ K)$ F
 using $g\ K'$ by (*rule ZfunD*)

ultimately

show eventually $(\lambda x. norm\ (f\ x **\ g\ x) < r)$ F

proof *eventually-elim*

case (*elim x*)

have $norm\ (f\ x **\ g\ x) \leq norm\ (f\ x) * norm\ (g\ x) * K$
 by (*rule norm-le*)

also have $norm\ (f\ x) * norm\ (g\ x) * K < r * inverse\ K * K$

by (*intro mult-strict-right-mono mult-strict-mono' norm-ge-zero elim K*)

also from K have $r * inverse\ K * K = r$

by *simp*

finally show ?case .
qed
qed

lemma (in bounded-bilinear) Zfun-left: Zfun f F \implies Zfun ($\lambda x. f x ** a$) F
by (rule bounded-linear-left [THEN bounded-linear.Zfun])

lemma (in bounded-bilinear) Zfun-right: Zfun f F \implies Zfun ($\lambda x. a ** f x$) F
by (rule bounded-linear-right [THEN bounded-linear.Zfun])

lemmas Zfun-mult = bounded-bilinear.Zfun [OF bounded-bilinear-mult]
lemmas Zfun-mult-right = bounded-bilinear.Zfun-right [OF bounded-bilinear-mult]
lemmas Zfun-mult-left = bounded-bilinear.Zfun-left [OF bounded-bilinear-mult]

lemma tendsto-Zfun-iff: ($f \longrightarrow a$) F = Zfun ($\lambda x. f x - a$) F
by (simp only: tendsto-iff Zfun-def dist-norm)

lemma tendsto-0-le:
($f \longrightarrow 0$) F \implies eventually ($\lambda x. \text{norm } (g x) \leq \text{norm } (f x) * K$) F \implies ($g \longrightarrow 0$) F
by (simp add: Zfun-imp-Zfun tendsto-Zfun-iff)

109.2.1 Distance and norms

lemma tendsto-dist [tendsto-intros]:
fixes l m :: 'a::metric-space
assumes f: ($f \longrightarrow l$) F
and g: ($g \longrightarrow m$) F
shows (($\lambda x. \text{dist } (f x) (g x)$) $\longrightarrow \text{dist } l m$) F
proof (rule tendstoI)
fix e :: real
assume 0 < e
then have e2: 0 < e/2 by simp
from tendstoD [OF f e2] tendstoD [OF g e2]
show eventually ($\lambda x. \text{dist } (f x) (g x) < e$) F
proof (eventually-elim)
case (elim x)
then show dist (dist (f x) (g x)) (dist l m) < e
unfolding dist-real-def
using dist-triangle2 [of f x g x l]
and dist-triangle2 [of g x l m]
and dist-triangle3 [of l m f x]
and dist-triangle [of f x m g x]
by arith
qed
qed

lemma continuous-dist[continuous-intros]:
fixes f g :: - \Rightarrow 'a :: metric-space

shows $\text{continuous } F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. \text{dist } (f x) (g x))$

unfolding continuous-def **by** (rule tendsto-dist)

lemma $\text{continuous-on-dist}$ [continuous-intros]:

fixes $f g :: - \Rightarrow 'a :: \text{metric-space}$

shows $\text{continuous-on } s f \implies \text{continuous-on } s g \implies \text{continuous-on } s (\lambda x. \text{dist } (f x) (g x))$

unfolding continuous-on-def **by** (auto intro: tendsto-dist)

lemma $\text{continuous-at-dist}$: $\text{isCont } (\text{dist } a) b$

using $\text{continuous-on-dist}$ [OF $\text{continuous-on-const}$ continuous-on-id] $\text{continuous-on-eq-continuous-within}$ **by** blast

lemma tendsto-norm [tendsto-intros]: $(f \longrightarrow a) F \implies ((\lambda x. \text{norm } (f x)) \longrightarrow \text{norm } a) F$

unfolding norm-conv-dist **by** (intro tendsto-intros)

lemma continuous-norm [continuous-intros]: $\text{continuous } F f \implies \text{continuous } F (\lambda x. \text{norm } (f x))$

unfolding continuous-def **by** (rule tendsto-norm)

lemma $\text{continuous-on-norm}$ [continuous-intros]:

$\text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. \text{norm } (f x))$

unfolding continuous-on-def **by** (auto intro: tendsto-norm)

lemma $\text{continuous-on-norm-id}$ [continuous-intros]: $\text{continuous-on } S \text{ norm}$

by (intro continuous-on-id $\text{continuous-on-norm}$)

lemma tendsto-norm-zero : $(f \longrightarrow 0) F \implies ((\lambda x. \text{norm } (f x)) \longrightarrow 0) F$

by (drule tendsto-norm) simp

lemma $\text{tendsto-norm-zero-cancel}$: $((\lambda x. \text{norm } (f x)) \longrightarrow 0) F \implies (f \longrightarrow 0) F$

unfolding $\text{tendsto-iff dist-norm}$ **by** simp

lemma $\text{tendsto-norm-zero-iff}$: $((\lambda x. \text{norm } (f x)) \longrightarrow 0) F \longleftrightarrow (f \longrightarrow 0) F$

unfolding $\text{tendsto-iff dist-norm}$ **by** simp

lemma tendsto-rabs [tendsto-intros]: $(f \longrightarrow l) F \implies ((\lambda x. |f x|) \longrightarrow |l|) F$

for $l :: \text{real}$

by (fold real-norm-def) (rule tendsto-norm)

lemma continuous-rabs [continuous-intros]:

$\text{continuous } F f \implies \text{continuous } F (\lambda x. |f x| :: \text{real})$

unfolding real-norm-def [symmetric] **by** (rule continuous-norm)

lemma $\text{continuous-on-rabs}$ [continuous-intros]:

$\text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. |f x| :: \text{real})$

unfolding *real-norm-def*[*symmetric*] **by** (*rule continuous-on-norm*)

lemma *tendsto-rabs-zero*: $(f \longrightarrow (0::real)) F \implies ((\lambda x. |f x|) \longrightarrow 0) F$
by (*fold real-norm-def*) (*rule tendsto-norm-zero*)

lemma *tendsto-rabs-zero-cancel*: $((\lambda x. |f x|) \longrightarrow (0::real)) F \implies (f \longrightarrow 0) F$
by (*fold real-norm-def*) (*rule tendsto-norm-zero-cancel*)

lemma *tendsto-rabs-zero-iff*: $((\lambda x. |f x|) \longrightarrow (0::real)) F \iff (f \longrightarrow 0) F$
by (*fold real-norm-def*) (*rule tendsto-norm-zero-iff*)

109.3 Topological Monoid

class *topological-monoid-add* = *topological-space* + *monoid-add* +
assumes *tendsto-add-Pair*: $LIM x (nhds a \times_F nhds b). fst x + snd x \Rightarrow nhds (a + b)$

class *topological-comm-monoid-add* = *topological-monoid-add* + *comm-monoid-add*

lemma *tendsto-add* [*tendsto-intros*]:
fixes $a b :: 'a::topological-monoid-add$
shows $(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. f x + g x) \longrightarrow a + b) F$
using *filterlim-compose*[*OF tendsto-add-Pair*, of $\lambda x. (f x, g x) a b F$]
by (*simp add: nhds-prod*[*symmetric*] *tendsto-Pair*)

lemma *continuous-add* [*continuous-intros*]:
fixes $f g :: - \Rightarrow 'b::topological-monoid-add$
shows $continuous F f \implies continuous F g \implies continuous F (\lambda x. f x + g x)$
unfolding *continuous-def* **by** (*rule tendsto-add*)

lemma *continuous-on-add* [*continuous-intros*]:
fixes $f g :: - \Rightarrow 'b::topological-monoid-add$
shows $continuous-on s f \implies continuous-on s g \implies continuous-on s (\lambda x. f x + g x)$
unfolding *continuous-on-def* **by** (*auto intro: tendsto-add*)

lemma *tendsto-add-zero*:
fixes $f g :: - \Rightarrow 'b::topological-monoid-add$
shows $(f \longrightarrow 0) F \implies (g \longrightarrow 0) F \implies ((\lambda x. f x + g x) \longrightarrow 0) F$
by (*drule* (1) *tendsto-add*) *simp*

lemma *tendsto-sum* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c::topological-comm-monoid-add$
shows $(\bigwedge i. i \in I \implies (f i \longrightarrow a i) F) \implies ((\lambda x. \sum_{i \in I} f i x) \longrightarrow (\sum_{i \in I} a i)) F$
by (*induct I rule: infinite-finite-induct*) (*simp-all add: tendsto-add*)

lemma *tendsto-null-sum*:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c::topological-comm-monoid-add$

assumes $\bigwedge i. i \in I \implies ((\lambda x. f x i) \longrightarrow 0) F$
shows $((\lambda i. \text{sum } (f i) I) \longrightarrow 0) F$
using *tendsto-sum* [of $I \lambda x y. f y x \lambda x. 0$] *assms* **by** *simp*

lemma *continuous-sum* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b::t2\text{-space} \Rightarrow 'c::\text{topological-comm-monoid-add}$
shows $(\bigwedge i. i \in I \implies \text{continuous } F (f i)) \implies \text{continuous } F (\lambda x. \sum_{i \in I}. f i x)$
unfolding *continuous-def* **by** (rule *tendsto-sum*)

lemma *continuous-on-sum* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b::\text{topological-space} \Rightarrow 'c::\text{topological-comm-monoid-add}$
shows $(\bigwedge i. i \in I \implies \text{continuous-on } S (f i)) \implies \text{continuous-on } S (\lambda x. \sum_{i \in I}. f i x)$
unfolding *continuous-on-def* **by** (auto *intro: tendsto-sum*)

instance *nat* :: *topological-comm-monoid-add*
by *standard*
(*simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal*)

instance *int* :: *topological-comm-monoid-add*
by *standard*
(*simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal*)

109.3.1 Topological group

class *topological-group-add* = *topological-monoid-add* + *group-add* +
assumes *tendsto-uminus-nhds*: $(\text{uminus} \longrightarrow - a) (\text{nhds } a)$
begin

lemma *tendsto-minus* [*tendsto-intros*]: $(f \longrightarrow a) F \implies ((\lambda x. - f x) \longrightarrow - a) F$
by (rule *filterlim-compose[OF tendsto-uminus-nhds]*)

end

class *topological-ab-group-add* = *topological-group-add* + *ab-group-add*

instance *topological-ab-group-add* < *topological-comm-monoid-add* ..

lemma *continuous-minus* [*continuous-intros*]: $\text{continuous } F f \implies \text{continuous } F (\lambda x. - f x)$
for $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{topological-group-add}$
unfolding *continuous-def* **by** (rule *tendsto-minus*)

lemma *continuous-on-minus* [*continuous-intros*]: $\text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. - f x)$
for $f :: - \Rightarrow 'b::\text{topological-group-add}$
unfolding *continuous-on-def* **by** (auto *intro: tendsto-minus*)

lemma *tendsto-minus-cancel*: $((\lambda x. - f x) \longrightarrow - a) F \implies (f \longrightarrow a) F$
for $a :: 'a::\text{topological-group-add}$
by (*drule tendsto-minus simp*)

lemma *tendsto-minus-cancel-left*:
 $(f \longrightarrow - (y::\text{topological-group-add})) F \longleftrightarrow ((\lambda x. - f x) \longrightarrow y) F$
using *tendsto-minus-cancel[of f - y F] tendsto-minus[of f - y F]*
by *auto*

lemma *tendsto-diff [tendsto-intros]*:
fixes $a b :: 'a::\text{topological-group-add}$
shows $(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. f x - g x) \longrightarrow a - b) F$
using *tendsto-add [of f a F \lambda x. - g x - b]* **by** (*simp add: tendsto-minus*)

lemma *continuous-diff [continuous-intros]*:
fixes $f g :: 'a::t2\text{-space} \Rightarrow 'b::\text{topological-group-add}$
shows *continuous* $F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. f x - g x)$
unfolding *continuous-def* **by** (*rule tendsto-diff*)

lemma *continuous-on-diff [continuous-intros]*:
fixes $f g :: - \Rightarrow 'b::\text{topological-group-add}$
shows *continuous-on* $s f \implies \text{continuous-on } s g \implies \text{continuous-on } s (\lambda x. f x - g x)$
unfolding *continuous-on-def* **by** (*auto intro: tendsto-diff*)

lemma *continuous-on-op-minus*: *continuous-on* $(s::'a::\text{topological-group-add set})$
 $((-) x)$
by (*rule continuous-intros | simp*)**+**

instance *real-normed-vector* < *topological-ab-group-add*

proof

fix $a b :: 'a$
show $((\lambda x. fst x + snd x) \longrightarrow a + b) (nhds a \times_F nhds b)$
unfolding *tendsto-Zfun-iff add-diff-add*
using *tendsto-fst[OF filterlim-ident, of (a,b)] tendsto-snd[OF filterlim-ident, of (a,b)]*
by (*intro Zfun-add*)
(auto simp: tendsto-Zfun-iff[symmetric] nhds-prod[symmetric] intro!: tendsto-fst)
show $(uminus \longrightarrow - a) (nhds a)$
unfolding *tendsto-Zfun-iff minus-diff-minus*
using *filterlim-ident[of nhds a]*
by (*intro Zfun-minus*) (*simp add: tendsto-Zfun-iff*)

qed

lemmas *real-tendsto-sandwich* = *tendsto-sandwich*[**where** $'a=\text{real}$]

109.3.2 Linear operators and multiplication

lemma *linear-times* [*simp*]: *linear* $(\lambda x. c * x)$
for $c :: 'a::\text{real-algebra}$
by (*auto simp: linearI distrib-left*)

lemma (**in** *bounded-linear*) *tendsto*: $(g \longrightarrow a) F \Longrightarrow ((\lambda x. f (g x)) \longrightarrow f a) F$
by (*simp only: tendsto-Zfun-iff diff [symmetric] Zfun*)

lemma (**in** *bounded-linear*) *continuous*: *continuous* $F g \Longrightarrow \text{continuous } F (\lambda x. f (g x))$
using *tendsto[of g - F]* **by** (*auto simp: continuous-def*)

lemma (**in** *bounded-linear*) *continuous-on*: *continuous-on* $s g \Longrightarrow \text{continuous-on } s (\lambda x. f (g x))$
using *tendsto[of g]* **by** (*auto simp: continuous-on-def*)

lemma (**in** *bounded-linear*) *tendsto-zero*: $(g \longrightarrow 0) F \Longrightarrow ((\lambda x. f (g x)) \longrightarrow 0) F$
by (*drule tendsto*) (*simp only: zero*)

lemma (**in** *bounded-bilinear*) *tendsto*:
 $(f \longrightarrow a) F \Longrightarrow (g \longrightarrow b) F \Longrightarrow ((\lambda x. f x ** g x) \longrightarrow a ** b) F$
by (*simp only: tendsto-Zfun-iff prod-diff-prod Zfun-add Zfun Zfun-left Zfun-right*)

lemma (**in** *bounded-bilinear*) *continuous*:
continuous $F f \Longrightarrow \text{continuous } F g \Longrightarrow \text{continuous } F (\lambda x. f x ** g x)$
using *tendsto[of f - F g]* **by** (*auto simp: continuous-def*)

lemma (**in** *bounded-bilinear*) *continuous-on*:
continuous-on $s f \Longrightarrow \text{continuous-on } s g \Longrightarrow \text{continuous-on } s (\lambda x. f x ** g x)$
using *tendsto[of f - - g]* **by** (*auto simp: continuous-on-def*)

lemma (**in** *bounded-bilinear*) *tendsto-zero*:
assumes $f: (f \longrightarrow 0) F$
and $g: (g \longrightarrow 0) F$
shows $((\lambda x. f x ** g x) \longrightarrow 0) F$
using *tendsto [OF f g]* **by** (*simp add: zero-left*)

lemma (**in** *bounded-bilinear*) *tendsto-left-zero*:
 $(f \longrightarrow 0) F \Longrightarrow ((\lambda x. f x ** c) \longrightarrow 0) F$
by (*rule bounded-linear.tendsto-zero [OF bounded-linear-left]*)

lemma (**in** *bounded-bilinear*) *tendsto-right-zero*:
 $(f \longrightarrow 0) F \Longrightarrow ((\lambda x. c ** f x) \longrightarrow 0) F$
by (*rule bounded-linear.tendsto-zero [OF bounded-linear-right]*)

lemmas *tendsto-of-real* [*tendsto-intros*] =
bounded-linear.tendsto [OF bounded-linear-of-real]

lemmas *tendsto-scaleR* [*tendsto-intros*] =
bounded-bilinear.tendsto [*OF bounded-bilinear-scaleR*]

Analogous type class for multiplication

class *topological-semigroup-mult* = *topological-space* + *semigroup-mult* +
assumes *tendsto-mult-Pair*: *LIM* x (*nhds* $a \times_F$ *nhds* b). *fst* $x * \text{snd } x$:> *nhds*
 $(a * b)$

instance *real-normed-algebra* < *topological-semigroup-mult*

proof

fix $a b :: 'a$
show $((\lambda x. \text{fst } x * \text{snd } x) \longrightarrow a * b)$ (*nhds* $a \times_F$ *nhds* b)
unfolding *nhds-prod[symmetric]*
using *tendsto-fst*[*OF filterlim-ident, of (a,b)*] *tendsto-snd*[*OF filterlim-ident, of*
 (a,b)]
by (*simp add: bounded-bilinear.tendsto* [*OF bounded-bilinear-mult*])
qed

lemma *tendsto-mult* [*tendsto-intros*]:

fixes $a b :: 'a::\text{topological-semigroup-mult}$
shows $(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. f x * g x) \longrightarrow a * b) F$
using *filterlim-compose*[*OF tendsto-mult-Pair, of $\lambda x. (f x, g x) a b F$*]
by (*simp add: nhds-prod[symmetric] tendsto-Pair*)

lemma *tendsto-mult-left*: $(f \longrightarrow l) F \implies ((\lambda x. c * (f x)) \longrightarrow c * l) F$
for $c :: 'a::\text{topological-semigroup-mult}$
by (*rule tendsto-mult* [*OF tendsto-const*])

lemma *tendsto-mult-right*: $(f \longrightarrow l) F \implies ((\lambda x. (f x) * c) \longrightarrow l * c) F$
for $c :: 'a::\text{topological-semigroup-mult}$
by (*rule tendsto-mult* [*OF - tendsto-const*])

lemma *tendsto-mult-left-iff* [*simp*]:

$c \neq 0 \implies \text{tendsto}(\lambda x. c * f x) (c * l) F \iff \text{tendsto } f l F$ **for** $c :: 'a::\{\text{topological-semigroup-mult,field}\}$
by (*auto simp: tendsto-mult-left dest: tendsto-mult-left* [**where** $c = 1/c$])

lemma *tendsto-mult-right-iff* [*simp*]:

$c \neq 0 \implies \text{tendsto}(\lambda x. f x * c) (l * c) F \iff \text{tendsto } f l F$ **for** $c :: 'a::\{\text{topological-semigroup-mult,field}\}$
by (*auto simp: tendsto-mult-right dest: tendsto-mult-left* [**where** $c = 1/c$])

lemma *tendsto-zero-mult-left-iff* [*simp*]:

fixes $c::'a::\{\text{topological-semigroup-mult,field}\}$ **assumes** $c \neq 0$ **shows** $(\lambda n. c * a n) \longrightarrow 0 \iff a \longrightarrow 0$
using *assms tendsto-mult-left tendsto-mult-left-iff* **by** *fastforce*

lemma *tendsto-zero-mult-right-iff* [*simp*]:

fixes $c::'a::\{\text{topological-semigroup-mult,field}\}$ **assumes** $c \neq 0$ **shows** $(\lambda n. a n * c) \longrightarrow 0 \iff a \longrightarrow 0$
using *assms tendsto-mult-right tendsto-mult-right-iff* **by** *fastforce*

lemma *tendsto-zero-divide-iff* [*simp*]:
fixes $c :: 'a :: \{\text{topological-semigroup-mult, field}\}$ **assumes** $c \neq 0$ **shows** $(\lambda n. a / c) \longrightarrow 0 \iff a \longrightarrow 0$
using *tendsto-zero-mult-right-iff* [*of 1/c a*] *assms* **by** (*simp add: field-simps*)

lemma *lim-const-over-n* [*tendsto-intros*]:
fixes $a :: 'a :: \text{real-normed-field}$
shows $(\lambda n. a / \text{of-nat } n) \longrightarrow 0$
using *tendsto-mult* [*OF tendsto-const*] [*of a*] *lim-1-over-n*] **by** *simp*

lemmas *continuous-of-real* [*continuous-intros*] =
bounded-linear.continuous [*OF bounded-linear-of-real*]

lemmas *continuous-scaleR* [*continuous-intros*] =
bounded-bilinear.continuous [*OF bounded-bilinear-scaleR*]

lemmas *continuous-mult* [*continuous-intros*] =
bounded-bilinear.continuous [*OF bounded-bilinear-mult*]

lemmas *continuous-on-of-real* [*continuous-intros*] =
bounded-linear.continuous-on [*OF bounded-linear-of-real*]

lemmas *continuous-on-scaleR* [*continuous-intros*] =
bounded-bilinear.continuous-on [*OF bounded-bilinear-scaleR*]

lemmas *continuous-on-mult* [*continuous-intros*] =
bounded-bilinear.continuous-on [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-zero* =
bounded-bilinear.tendsto-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-left-zero* =
bounded-bilinear.tendsto-left-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-right-zero* =
bounded-bilinear.tendsto-right-zero [*OF bounded-bilinear-mult*]

lemma *continuous-mult-left*:
fixes $c :: 'a :: \text{real-normed-algebra}$
shows *continuous F f* \implies *continuous F* $(\lambda x. c * f x)$
by (*rule continuous-mult* [*OF continuous-const*])

lemma *continuous-mult-right*:
fixes $c :: 'a :: \text{real-normed-algebra}$
shows *continuous F f* \implies *continuous F* $(\lambda x. f x * c)$
by (*rule continuous-mult* [*OF - continuous-const*])

lemma *continuous-on-mult-left*:

fixes $c :: 'a :: \text{real-normed-algebra}$

shows $\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. c * f x)$

by (rule *continuous-on-mult* [*OF continuous-on-const*])

lemma *continuous-on-mult-right*:

fixes $c :: 'a :: \text{real-normed-algebra}$

shows $\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. f x * c)$

by (rule *continuous-on-mult* [*OF - continuous-on-const*])

lemma *continuous-on-mult-const* [*simp*]:

fixes $c :: 'a :: \text{real-normed-algebra}$

shows $\text{continuous-on } s \ ((*) \ c)$

by (*intro continuous-on-mult-left continuous-on-id*)

lemma *tendsto-divide-zero*:

fixes $c :: 'a :: \text{real-normed-field}$

shows $(f \longrightarrow 0) \ F \implies ((\lambda x. f x / c) \longrightarrow 0) \ F$

by (*cases c=0*) (*simp-all add: divide-inverse tendsto-mult-left-zero*)

lemma *tendsto-power* [*tendsto-intros*]: $(f \longrightarrow a) \ F \implies ((\lambda x. f x ^ n) \longrightarrow a ^ n) \ F$

for $f :: 'a \Rightarrow 'b :: \{\text{power, real-normed-algebra}\}$

by (*induct n*) (*simp-all add: tendsto-mult*)

lemma *tendsto-null-power*: $\llbracket (f \longrightarrow 0) \ F; 0 < n \rrbracket \implies ((\lambda x. f x ^ n) \longrightarrow 0) \ F$

for $f :: 'a \Rightarrow 'b :: \{\text{power, real-normed-algebra-1}\}$

using *tendsto-power* [*of f 0 F n*] **by** (*simp add: power-0-left*)

lemma *continuous-power* [*continuous-intros*]: $\text{continuous } F \ f \implies \text{continuous } F \ (\lambda x. (f x) ^ n)$

for $f :: 'a :: \text{t2-space} \Rightarrow 'b :: \{\text{power, real-normed-algebra}\}$

unfolding *continuous-def* **by** (rule *tendsto-power*)

lemma *continuous-on-power* [*continuous-intros*]:

fixes $f :: - \Rightarrow 'b :: \{\text{power, real-normed-algebra}\}$

shows $\text{continuous-on } s \ f \implies \text{continuous-on } s \ (\lambda x. (f x) ^ n)$

unfolding *continuous-on-def* **by** (*auto intro: tendsto-power*)

lemma *tendsto-prod* [*tendsto-intros*]:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c :: \{\text{real-normed-algebra, comm-ring-1}\}$

shows $(\bigwedge i. i \in S \implies (f i \longrightarrow L i) \ F) \implies ((\lambda x. \prod_{i \in S} f i x) \longrightarrow (\prod_{i \in S} L i)) \ F$

by (*induct S rule: infinite-finite-induct*) (*simp-all add: tendsto-mult*)

lemma *continuous-prod* [*continuous-intros*]:

fixes $f :: 'a \Rightarrow 'b :: \text{t2-space} \Rightarrow 'c :: \{\text{real-normed-algebra, comm-ring-1}\}$

shows $(\bigwedge i. i \in S \implies \text{continuous } F \ (f i)) \implies \text{continuous } F \ (\lambda x. \prod_{i \in S} f i x)$

unfolding *continuous-def* **by** (rule *tendsto-prod*)

lemma *continuous-on-prod* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow - \Rightarrow 'c::\{\text{real-normed-algebra, comm-ring-1}\}$
shows $(\bigwedge i. i \in S \implies \text{continuous-on } s (f i)) \implies \text{continuous-on } s (\lambda x. \prod_{i \in S}. f i x)$
unfolding *continuous-on-def* **by** (*auto intro: tendsto-prod*)

lemma *tendsto-of-real-iff*:
 $((\lambda x. \text{of-real } (f x) :: 'a::\text{real-normed-div-algebra}) \longrightarrow \text{of-real } c) F \longleftrightarrow (f \longrightarrow c) F$
unfolding *tendsto-iff* **by** *simp*

lemma *tendsto-add-const-iff*:
 $((\lambda x. c + f x :: 'a::\text{topological-group-add}) \longrightarrow c + d) F \longleftrightarrow (f \longrightarrow d) F$
using *tendsto-add*[*OF tendsto-const*[*of c*], *of f d*]
and *tendsto-add*[*OF tendsto-const*[*of -c*], *of* $\lambda x. c + f x$ *c + d*] **by** *auto*

class *topological-monoid-mult* = *topological-semigroup-mult* + *monoid-mult*
class *topological-comm-monoid-mult* = *topological-monoid-mult* + *comm-monoid-mult*

lemma *tendsto-power-strong* [*tendsto-intros*]:
fixes $f :: - \Rightarrow 'b :: \text{topological-monoid-mult}$
assumes $(f \longrightarrow a) F (g \longrightarrow b) F$
shows $((\lambda x. f x \hat{=} g x) \longrightarrow a \hat{=} b) F$
proof –
have $((\lambda x. f x \hat{=} b) \longrightarrow a \hat{=} b) F$
by (*induction b*) (*auto intro: tendsto-intros assms*)
also from *assms*(2) **have** *eventually* $(\lambda x. g x = b) F$
by (*simp add: nhds-discrete filterlim-principal*)
hence *eventually* $(\lambda x. f x \hat{=} b = f x \hat{=} g x) F$
by *eventually-elim simp*
hence $((\lambda x. f x \hat{=} b) \longrightarrow a \hat{=} b) F \longleftrightarrow ((\lambda x. f x \hat{=} g x) \longrightarrow a \hat{=} b) F$
by (*intro filterlim-cong refl*)
finally show *?thesis* .
qed

lemma *continuous-mult'* [*continuous-intros*]:
fixes $f g :: - \Rightarrow 'b::\text{topological-semigroup-mult}$
shows *continuous* $F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. f x * g x)$
unfolding *continuous-def* **by** (*rule tendsto-mult*)

lemma *continuous-power'* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'b::\text{topological-monoid-mult}$
shows *continuous* $F f \implies \text{continuous } F g \implies \text{continuous } F (\lambda x. f x \hat{=} g x)$
unfolding *continuous-def* **by** (*rule tendsto-power-strong*) *auto*

lemma *continuous-on-mult'* [*continuous-intros*]:
fixes $f g :: - \Rightarrow 'b::\text{topological-semigroup-mult}$

shows $\text{continuous-on } A \ f \implies \text{continuous-on } A \ g \implies \text{continuous-on } A \ (\lambda x. f \ x \ * \ g \ x)$

unfolding continuous-on-def **by** $(\text{auto intro: tendsto-mult})$

lemma $\text{continuous-on-power}'$ [continuous-intros]:

fixes $f :: - \Rightarrow 'b::\text{topological-monoid-mult}$

shows $\text{continuous-on } A \ f \implies \text{continuous-on } A \ g \implies \text{continuous-on } A \ (\lambda x. f \ x \ \wedge \ g \ x)$

unfolding continuous-on-def **by** $(\text{auto intro: tendsto-power-strong})$

lemma tendsto-mult-one :

fixes $f \ g :: - \Rightarrow 'b::\text{topological-monoid-mult}$

shows $(f \longrightarrow 1) \ F \implies (g \longrightarrow 1) \ F \implies ((\lambda x. f \ x \ * \ g \ x) \longrightarrow 1) \ F$

by $(\text{drule } (1) \ \text{tendsto-mult}) \ \text{simp}$

lemma $\text{tendsto-prod}'$ [tendsto-intros]:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c::\text{topological-comm-monoid-mult}$

shows $(\bigwedge i. i \in I \implies (f \ i \longrightarrow a \ i) \ F) \implies ((\lambda x. \prod_{i \in I}. f \ i \ x) \longrightarrow (\prod_{i \in I}. a \ i)) \ F$

by $(\text{induct } I \ \text{rule: infinite-finite-induct}) \ (\text{simp-all add: tendsto-mult})$

lemma $\text{tendsto-one-prod}'$:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c::\text{topological-comm-monoid-mult}$

assumes $\bigwedge i. i \in I \implies ((\lambda x. f \ x \ i) \longrightarrow 1) \ F$

shows $((\lambda i. \text{prod } (f \ i) \ I) \longrightarrow 1) \ F$

using $\text{tendsto-prod}'$ [$\text{of } I \ \lambda x \ y. f \ y \ x \ \lambda x. 1$] **assms** **by** simp

lemma LIMSEQ-prod-0 :

fixes $f :: \text{nat} \Rightarrow 'a::\{\text{semidom}, \text{topological-space}\}$

assumes $f \ i = 0$

shows $(\lambda n. \text{prod } f \ \{..n\}) \longrightarrow 0$

proof $(\text{subst } \text{tendsto-cong})$

show $\forall_F \ n \ \text{in } \text{sequentially. } \text{prod } f \ \{..n\} = 0$

using $\text{assms eventually-at-top-linorder}$ **by** auto

qed auto

lemma $\text{LIMSEQ-prod-nonneg}$:

fixes $f :: \text{nat} \Rightarrow 'a::\{\text{linordered-semidom}, \text{linorder-topology}\}$

assumes $0: \bigwedge n. 0 \leq f \ n$ **and** $a: (\lambda n. \text{prod } f \ \{..n\}) \longrightarrow a$

shows $a \geq 0$

by $(\text{simp add: } 0 \ \text{prod-nonneg LIMSEQ-le-const } [OF \ a])$

lemma $\text{continuous-prod}'$ [continuous-intros]:

fixes $f :: 'a \Rightarrow 'b::\text{t2-space} \Rightarrow 'c::\text{topological-comm-monoid-mult}$

shows $(\bigwedge i. i \in I \implies \text{continuous } F \ (f \ i)) \implies \text{continuous } F \ (\lambda x. \prod_{i \in I}. f \ i \ x)$

unfolding continuous-def **by** $(\text{rule } \text{tendsto-prod}')$

lemma $\text{continuous-on-prod}'$ [continuous-intros]:

fixes $f :: 'a \Rightarrow 'b::\text{topological-space} \Rightarrow 'c::\text{topological-comm-monoid-mult}$

```

shows ( $\bigwedge i. i \in I \implies \text{continuous-on } S (f i) \implies \text{continuous-on } S (\lambda x. \prod_{i \in I}. f i x)$ )
unfolding continuous-on-def by (auto intro: tendsto-prod')

instance nat :: topological-comm-monoid-mult
by standard
(simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

instance int :: topological-comm-monoid-mult
by standard
(simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

class comm-real-normed-algebra-1 = real-normed-algebra-1 + comm-monoid-mult

context real-normed-field
begin

subclass comm-real-normed-algebra-1
proof
from norm-mult[of 1 :: 'a 1] show norm 1 = 1 by simp
qed (simp-all add: norm-mult)

end

```

109.3.3 Inverse and division

```

lemma (in bounded-bilinear) Zfun-prod-Bfun:
assumes f: Zfun f F
and g: Bfun g F
shows Zfun ( $\lambda x. f x ** g x$ ) F
proof –
obtain K where K: 0 ≤ K
and norm-le:  $\bigwedge x y. \text{norm } (x ** y) \leq \text{norm } x * \text{norm } y * K$ 
using nonneg-bounded by blast
obtain B where B: 0 < B
and norm-g: eventually ( $\lambda x. \text{norm } (g x) \leq B$ ) F
using g by (rule BfunE)
have eventually ( $\lambda x. \text{norm } (f x ** g x) \leq \text{norm } (f x) * (B * K)$ ) F
using norm-g proof eventually-elim
case (elim x)
have norm (f x ** g x) ≤ norm (f x) * norm (g x) * K
by (rule norm-le)
also have  $\dots \leq \text{norm } (f x) * B * K$ 
by (intro mult-mono' order-refl norm-g norm-ge-zero mult-nonneg-nonneg K
elim)
also have  $\dots = \text{norm } (f x) * (B * K)$ 
by (rule mult.assoc)
finally show norm (f x ** g x) ≤ norm (f x) * (B * K) .
qed

```


with f **show** *?thesis*
by (rule *Zfun-imp-Zfun*)
qed

lemma (in *bounded-bilinear*) *Bfun-prod-Zfun*:
assumes f : *Bfun* f F
and g : *Zfun* g F
shows *Zfun* $(\lambda x. f\ x\ **\ g\ x)$ F
using *flip* $g\ f$ **by** (rule *bounded-bilinear.Zfun-prod-Bfun*)

lemma *Bfun-inverse*:
fixes a :: '*a*::*real-normed-div-algebra*
assumes f : $(f \longrightarrow a)$ F
assumes a : $a \neq 0$
shows *Bfun* $(\lambda x. \text{inverse } (f\ x))$ F
proof –
from a **have** $0 < \text{norm } a$ **by** *simp*
then have $\exists r > 0. r < \text{norm } a$ **by** (rule *dense*)
then obtain r **where** $r1$: $0 < r$ **and** $r2$: $r < \text{norm } a$
by *blast*
have *eventually* $(\lambda x. \text{dist } (f\ x)\ a < r)$ F
using *tendstoD* [*OF* $r1$] **by** *blast*
then have *eventually* $(\lambda x. \text{norm } (\text{inverse } (f\ x)) \leq \text{inverse } (\text{norm } a - r))$ F
proof *eventually-elim*
case (*elim* x)
then have 1 : $\text{norm } (f\ x - a) < r$
by (*simp* *add: dist-norm*)
then have 2 : $f\ x \neq 0$ **using** $r2$ **by** *auto*
then have $\text{norm } (\text{inverse } (f\ x)) = \text{inverse } (\text{norm } (f\ x))$
by (rule *nonzero-norm-inverse*)
also have $\dots \leq \text{inverse } (\text{norm } a - r)$
proof (rule *le-imp-inverse-le*)
show $0 < \text{norm } a - r$
using $r2$ **by** *simp*
have $\text{norm } a - \text{norm } (f\ x) \leq \text{norm } (a - f\ x)$
by (rule *norm-triangle-ineq2*)
also have $\dots = \text{norm } (f\ x - a)$
by (rule *norm-minus-commute*)
also have $\dots < r$ **using** 1 .
finally show $\text{norm } a - r \leq \text{norm } (f\ x)$
by *simp*
qed
finally show $\text{norm } (\text{inverse } (f\ x)) \leq \text{inverse } (\text{norm } a - r)$.
qed
then show *?thesis* **by** (rule *BfunI*)
qed

lemma *tendsto-inverse* [*tendsto-intros*]:
fixes a :: '*a*::*real-normed-div-algebra*

assumes $f: (f \longrightarrow a) F$
and $a: a \neq 0$
shows $((\lambda x. \text{inverse } (f x)) \longrightarrow \text{inverse } a) F$
proof –
from a **have** $0 < \text{norm } a$ **by** *simp*
with f **have** *eventually* $(\lambda x. \text{dist } (f x) a < \text{norm } a) F$
by *(rule tendstoD)*
then **have** *eventually* $(\lambda x. f x \neq 0) F$
unfolding *dist-norm* **by** *(auto elim!: eventually-mono)*
with a **have** *eventually* $(\lambda x. \text{inverse } (f x) - \text{inverse } a =$
 $- (\text{inverse } (f x) * (f x - a) * \text{inverse } a)) F$
by *(auto elim!: eventually-mono simp: inverse-diff-inverse)*
moreover **have** *Zfun* $(\lambda x. - (\text{inverse } (f x) * (f x - a) * \text{inverse } a)) F$
by *(intro Zfun-minus Zfun-mult-left*
bounded-bilinear.Bfun-prod-Zfun [OF bounded-bilinear-mult]
Bfun-inverse [OF f a] f [unfolded tendsto-Zfun-iff])
ultimately show *?thesis*
unfolding *tendsto-Zfun-iff* **by** *(rule Zfun-ssubst)*
qed

lemma *continuous-inverse*:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-div-algebra}$
assumes *continuous* $F f$
and $f (\text{Lim } F (\lambda x. x)) \neq 0$
shows *continuous* $F (\lambda x. \text{inverse } (f x))$
using *assms* **unfolding** *continuous-def* **by** *(rule tendsto-inverse)*

lemma *continuous-at-within-inverse*[*continuous-intros*]:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-div-algebra}$
assumes *continuous* $(\text{at } a \text{ within } s) f$
and $f a \neq 0$
shows *continuous* $(\text{at } a \text{ within } s) (\lambda x. \text{inverse } (f x))$
using *assms* **unfolding** *continuous-within* **by** *(rule tendsto-inverse)*

lemma *continuous-on-inverse*[*continuous-intros*]:
fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-div-algebra}$
assumes *continuous-on* $s f$
and $\forall x \in s. f x \neq 0$
shows *continuous-on* $s (\lambda x. \text{inverse } (f x))$
using *assms* **unfolding** *continuous-on-def* **by** *(blast intro: tendsto-inverse)*

lemma *tendsto-divide* [*tendsto-intros*]:
fixes $a b :: 'a::\text{real-normed-field}$
shows $(f \longrightarrow a) F \Longrightarrow (g \longrightarrow b) F \Longrightarrow b \neq 0 \Longrightarrow ((\lambda x. f x / g x) \longrightarrow a / b) F$
by *(simp add: tendsto-mult tendsto-inverse divide-inverse)*

lemma *continuous-divide*:
fixes $f g :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-field}$

assumes *continuous F f*
and *continuous F g*
and $g (\text{Lim } F (\lambda x. x)) \neq 0$
shows *continuous F* $(\lambda x. (f x) / (g x))$
using *assms unfolding continuous-def* **by** (rule *tendsto-divide*)

lemma *continuous-at-within-divide*[*continuous-intros*]:
fixes $f g :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-field}$
assumes *continuous (at a within s) f* *continuous (at a within s) g*
and $g a \neq 0$
shows *continuous (at a within s)* $(\lambda x. (f x) / (g x))$
using *assms unfolding continuous-within* **by** (rule *tendsto-divide*)

lemma *isCont-divide*[*continuous-intros, simp*]:
fixes $f g :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-field}$
assumes *isCont f a* *isCont g a* $g a \neq 0$
shows *isCont* $(\lambda x. (f x) / g x) a$
using *assms unfolding continuous-at* **by** (rule *tendsto-divide*)

lemma *continuous-on-divide*[*continuous-intros*]:
fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-field}$
assumes *continuous-on s f* *continuous-on s g*
and $\forall x \in s. g x \neq 0$
shows *continuous-on s* $(\lambda x. (f x) / (g x))$
using *assms unfolding continuous-on-def* **by** (blast *intro: tendsto-divide*)

lemma *tendsto-power-int* [*tendsto-intros*]:
fixes $a :: 'a::\text{real-normed-div-algebra}$
assumes $f: (f \longrightarrow a) F$
and $a: a \neq 0$
shows $(\lambda x. \text{power-int } (f x) n) \longrightarrow \text{power-int } a n) F$
using *assms by (cases n rule: int-cases₄) (auto intro!: tendsto-intros simp: power-int-minus)*

lemma *continuous-power-int*:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-div-algebra}$
assumes *continuous F f*
and $f (\text{Lim } F (\lambda x. x)) \neq 0$
shows *continuous F* $(\lambda x. \text{power-int } (f x) n)$
using *assms unfolding continuous-def* **by** (rule *tendsto-power-int*)

lemma *continuous-at-within-power-int*[*continuous-intros*]:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-div-algebra}$
assumes *continuous (at a within s) f*
and $f a \neq 0$
shows *continuous (at a within s)* $(\lambda x. \text{power-int } (f x) n)$
using *assms unfolding continuous-within* **by** (rule *tendsto-power-int*)

lemma *continuous-on-power-int* [*continuous-intros*]:

fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-div-algebra}$
assumes *continuous-on s f* **and** $\forall x \in s. f\ x \neq 0$
shows *continuous-on s* $(\lambda x. \text{power-int } (f\ x)\ n)$
using *assms unfolding continuous-on-def* **by** *(blast intro: tendsto-power-int)*

lemma *tendsto-power-int'* [*tendsto-intros*]:
fixes $a :: 'a::\text{real-normed-div-algebra}$
assumes $f: (f \longrightarrow a)\ F$
and $a \neq 0 \vee n \geq 0$
shows $((\lambda x. \text{power-int } (f\ x)\ n) \longrightarrow \text{power-int } a\ n)\ F$
using *assms* **by** *(cases n rule: int-cases4) (auto intro!: tendsto-intros simp: power-int-minus)*

lemma *tendsto-sgn* [*tendsto-intros*]: $(f \longrightarrow l)\ F \Longrightarrow l \neq 0 \Longrightarrow ((\lambda x. \text{sgn } (f\ x)) \longrightarrow \text{sgn } l)\ F$
for $l :: 'a::\text{real-normed-vector}$
unfolding *sgn-div-norm* **by** *(simp add: tendsto-intros)*

lemma *continuous-sgn*:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *continuous F f*
and $f\ (\text{Lim } F\ (\lambda x. x)) \neq 0$
shows *continuous F* $(\lambda x. \text{sgn } (f\ x))$
using *assms unfolding continuous-def* **by** *(rule tendsto-sgn)*

lemma *continuous-at-within-sgn*[*continuous-intros*]:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *continuous (at a within s) f*
and $f\ a \neq 0$
shows *continuous (at a within s)* $(\lambda x. \text{sgn } (f\ x))$
using *assms unfolding continuous-within* **by** *(rule tendsto-sgn)*

lemma *isCont-sgn*[*continuous-intros*]:
fixes $f :: 'a::t2\text{-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *isCont f a*
and $f\ a \neq 0$
shows *isCont* $(\lambda x. \text{sgn } (f\ x))\ a$
using *assms unfolding continuous-at* **by** *(rule tendsto-sgn)*

lemma *continuous-on-sgn*[*continuous-intros*]:
fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$
assumes *continuous-on s f*
and $\forall x \in s. f\ x \neq 0$
shows *continuous-on s* $(\lambda x. \text{sgn } (f\ x))$
using *assms unfolding continuous-on-def* **by** *(blast intro: tendsto-sgn)*

lemma *filterlim-at-infinity*:
fixes $f :: - \Rightarrow 'a::\text{real-normed-vector}$
assumes $0 \leq c$

shows $(LIM\ x\ F.\ f\ x\ :\>\ at\text{-infinity}) \longleftrightarrow (\forall r>c.\ eventually\ (\lambda x.\ r \leq norm\ (f\ x))\ F)$
unfolding *filterlim-iff eventually-at-infinity*
proof *safe*
fix $P :: 'a \Rightarrow bool$
fix b
assume $*$: $\forall r>c.\ eventually\ (\lambda x.\ r \leq norm\ (f\ x))\ F$
assume P : $\forall x.\ b \leq norm\ x \longrightarrow P\ x$
have $max\ b\ (c + 1) > c$ **by** *auto*
with $*$ **have** $eventually\ (\lambda x.\ max\ b\ (c + 1) \leq norm\ (f\ x))\ F$
by *auto*
then show $eventually\ (\lambda x.\ P\ (f\ x))\ F$
proof *eventually-elim*
case $(elim\ x)$
with P **show** $P\ (f\ x)$ **by** *auto*
qed
qed *force*

lemma *filterlim-at-infinity-imp-norm-at-top*:
fixes F
assumes *filterlim f at-infinity F*
shows *filterlim $(\lambda x.\ norm\ (f\ x))$ at-top F*
proof –
{
fix $r :: real$
have $\forall_F\ x\ in\ F.\ r \leq norm\ (f\ x)$ **using** *filterlim-at-infinity[of 0 f F] assms*
by $(cases\ r > 0)$
 $(auto\ simp:\ not\ less\ intro:\ always\ eventually\ order.trans[OF\ -\ norm\ ge\ zero])$
}
thus *?thesis* **by** $(auto\ simp:\ filterlim\ at\ top)$
qed

lemma *filterlim-norm-at-top-imp-at-infinity*:
fixes F
assumes *filterlim $(\lambda x.\ norm\ (f\ x))$ at-top F*
shows *filterlim f at-infinity F*
using *filterlim-at-infinity[of 0 f F] assms* **by** $(auto\ simp:\ filterlim\ at\ top)$

lemma *filterlim-norm-at-top: filterlim norm at-top at-infinity*
by $(rule\ filterlim\ at\ infinity\ imp\ norm\ at\ top)\ (rule\ filterlim\ ident)$

lemma *filterlim-at-infinity-conv-norm-at-top*:
 $filterlim\ f\ at\ infinity\ G \longleftrightarrow filterlim\ (\lambda x.\ norm\ (f\ x))\ at\ top\ G$
by $(auto\ simp:\ filterlim\ at\ infinity[OF\ order.refl]\ filterlim\ at\ top\ gt[of\ -\ -\ 0])$

lemma *eventually-not-equal-at-infinity*:
 $eventually\ (\lambda x.\ x \neq (a :: 'a :: \{real\}\ normed\ vector))\ at\ infinity$
proof –
from *filterlim-norm-at-top[where 'a = 'a]*

have $\forall_F x$ in *at-infinity*. *norm a < norm (x::'a)* **by** (*auto simp: filterlim-at-top-dense*)
thus *?thesis* **by** *eventually-elim auto*
qed

lemma *filterlim-int-of-nat-at-topD*:

fixes *F*

assumes *filterlim* ($\lambda x. f$ (*int* *x*)) *F at-top*

shows *filterlim* *f F at-top*

proof –

have *filterlim* ($\lambda x. f$ (*int* (*nat* *x*))) *F at-top*

by (*rule filterlim-compose*[*OF assms filterlim-nat-sequentially*])

also have *?this* \longleftrightarrow *filterlim* *f F at-top*

by (*intro filterlim-cong refl eventually-mono* [*OF eventually-ge-at-top*[*of 0::int*]])

auto

finally show *?thesis* .

qed

lemma *filterlim-int-sequentially* [*tendsto-intros*]:

filterlim *int at-top sequentially*

unfolding *filterlim-at-top*

proof

fix *C :: int*

show *eventually* ($\lambda n. \text{int } n \geq C$) *at-top*

using *eventually-ge-at-top*[*of nat* [*C*]] **by** *eventually-elim linarith*

qed

lemma *filterlim-real-of-int-at-top* [*tendsto-intros*]:

filterlim *real-of-int at-top at-top*

unfolding *filterlim-at-top*

proof

fix *C :: real*

show *eventually* ($\lambda n. \text{real-of-int } n \geq C$) *at-top*

using *eventually-ge-at-top*[*of* [*C*]] **by** *eventually-elim linarith*

qed

lemma *filterlim-abs-real*: *filterlim* (*abs::real* \Rightarrow *real*) *at-top at-top*

proof (*subst filterlim-cong*[*OF refl refl*])

from *eventually-ge-at-top*[*of 0::real*] **show** *eventually* ($\lambda x::\text{real}. |x| = x$) *at-top*

by *eventually-elim simp*

qed (*simp-all add: filterlim-ident*)

lemma *filterlim-of-real-at-infinity* [*tendsto-intros*]:

filterlim (*of-real* $:: \text{real} \Rightarrow 'a :: \text{real-normed-algebra-1}$) *at-infinity at-top*

by (*intro filterlim-norm-at-top-imp-at-infinity*) (*auto simp: filterlim-abs-real*)

lemma *not-tendsto-and-filterlim-at-infinity*:

fixes *c :: 'a::real-normed-vector*

assumes *F* $\neq \text{bot}$

and (*f* \longrightarrow *c*) *F*

```

  and filterlim f at-infinity F
shows False
proof -
  from tendstoD[OF assms(2), of 1/2]
  have eventually ( $\lambda x. \text{dist } (f x) c < 1/2$ ) F
  by simp
  moreover
  from filterlim-at-infinity[of norm c f F] assms(3)
  have eventually ( $\lambda x. \text{norm } (f x) \geq \text{norm } c + 1$ ) F by simp
  ultimately have eventually ( $\lambda x. \text{False}$ ) F
  proof eventually-elim
    fix x
    assume A:  $\text{dist } (f x) c < 1/2$ 
    assume  $\text{norm } (f x) \geq \text{norm } c + 1$ 
    also have  $\text{norm } (f x) = \text{dist } (f x) 0$  by simp
    also have  $\dots \leq \text{dist } (f x) c + \text{dist } c 0$  by (rule dist-triangle)
    finally show False using A by simp
  qed
  with assms show False by simp
qed

```

```

lemma filterlim-at-infinity-imp-not-convergent:
  assumes filterlim f at-infinity sequentially
  shows  $\neg$  convergent f
  by (rule notI, rule not-tendsto-and-filterlim-at-infinity[OF - - assms])
  (simp-all add: convergent-LIMSEQ-iff)

```

```

lemma filterlim-at-infinity-imp-eventually-ne:
  assumes filterlim f at-infinity F
  shows eventually ( $\lambda z. f z \neq c$ ) F
proof -
  have  $\text{norm } c + 1 > 0$ 
  by (intro add-nonneg-pos) simp-all
  with filterlim-at-infinity[OF order.refl, of f F] assms
  have eventually ( $\lambda z. \text{norm } (f z) \geq \text{norm } c + 1$ ) F
  by blast
  then show ?thesis
  by eventually-elim auto
qed

```

```

lemma tendsto-of-nat [tendsto-intros]:
  filterlim (of-nat :: nat  $\Rightarrow$  'a::real-normed-algebra-1) at-infinity sequentially
proof (subst filterlim-at-infinity[OF order.refl], intro allI impI)
  fix r :: real
  assume r:  $r > 0$ 
  define n where  $n = \text{nat } \lceil r \rceil$ 
  from r have n:  $\forall m \geq n. \text{of-nat } m \geq r$ 
  unfolding n-def by linarith
  from eventually-ge-at-top[of n] show eventually ( $\lambda m. \text{norm } (\text{of-nat } m :: 'a) \geq$ 

```

r) *sequentially*
 by *eventually-elim* (use *n* in *simp-all*)
 qed

109.4 Relate *at*, *at-left* and *at-right*

This lemmas are useful for conversion between *at x* to *at-left x* and *at-right x* and also *at-right (0::'a)*.

lemmas *filterlim-split-at-real* = *filterlim-split-at*[**where** 'a=real]

lemma *filtermap-nhds-shift*: *filtermap* ($\lambda x. x - d$) (*nhds* *a*) = *nhds* (*a* - *d*)
for *a d* :: 'a::real-normed-vector
by (*rule filtermap-fun-inverse*[**where** *g*= $\lambda x. x + d$])
 (*auto intro!*: *tendsto-eq-intros filterlim-ident*)

lemma *filtermap-nhds-minus*: *filtermap* ($\lambda x. - x$) (*nhds* *a*) = *nhds* (*- a*)
for *a* :: 'a::real-normed-vector
by (*rule filtermap-fun-inverse*[**where** *g*=*uminus*])
 (*auto intro!*: *tendsto-eq-intros filterlim-ident*)

lemma *filtermap-at-shift*: *filtermap* ($\lambda x. x - d$) (*at* *a*) = *at* (*a* - *d*)
for *a d* :: 'a::real-normed-vector
by (*simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift*[*symmetric*])

lemma *filtermap-at-right-shift*: *filtermap* ($\lambda x. x - d$) (*at-right* *a*) = *at-right* (*a* - *d*)
for *a d* :: real
by (*simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift*[*symmetric*])

lemma *filterlim-shift*:
fixes *d* :: 'a::real-normed-vector
assumes *filterlim f F* (*at* *a*)
shows *filterlim* (*f* \circ (+) *d*) *F* (*at* (*a* - *d*))
unfolding *filterlim-iff*
proof (*intro strip*)
fix *P*
assume *eventually P F*
then have $\forall_F x$ in *filtermap* ($\lambda y. y - d$) (*at* *a*). *P* (*f* (*d* + *x*))
using *assms* **by** (*force simp add: filterlim-iff eventually-filtermap*)
then show ($\forall_F x$ in *at* (*a* - *d*). *P* ((*f* \circ (+) *d*) *x*))
by (*force simp add: filtermap-at-shift*)
 qed

lemma *filterlim-shift-iff*:
fixes *d* :: 'a::real-normed-vector
shows *filterlim* (*f* \circ (+) *d*) *F* (*at* (*a* - *d*)) = *filterlim f F* (*at* *a*) (is ?lhs = ?rhs)
proof
assume *L: ?lhs* **show** ?rhs

using *filterlim-shift* [*OF L*, *of -d*] **by** (*simp add: filterlim-iff*)
qed (*metis filterlim-shift*)

lemma *at-right-to-0*: $\text{at-right } a = \text{filtermap } (\lambda x. x + a) (\text{at-right } 0)$
for $a :: \text{real}$
using *filtermap-at-right-shift*[*of -a 0*] **by** *simp*

lemma *filterlim-at-right-to-0*:
 $\text{filterlim } f F (\text{at-right } a) \longleftrightarrow \text{filterlim } (\lambda x. f (x + a)) F (\text{at-right } 0)$
for $a :: \text{real}$
unfolding *filterlim-def filtermap-filtermap at-right-to-0*[*of a*] ..

lemma *eventually-at-right-to-0*:
 $\text{eventually } P (\text{at-right } a) \longleftrightarrow \text{eventually } (\lambda x. P (x + a)) (\text{at-right } 0)$
for $a :: \text{real}$
unfolding *at-right-to-0*[*of a*] **by** (*simp add: eventually-filtermap*)

lemma *at-to-0*: $\text{at } a = \text{filtermap } (\lambda x. x + a) (\text{at } 0)$
for $a :: 'a::\text{real-normed-vector}$
using *filtermap-at-shift*[*of -a 0*] **by** *simp*

lemma *filterlim-at-to-0*:
 $\text{filterlim } f F (\text{at } a) \longleftrightarrow \text{filterlim } (\lambda x. f (x + a)) F (\text{at } 0)$
for $a :: 'a::\text{real-normed-vector}$
unfolding *filterlim-def filtermap-filtermap at-to-0*[*of a*] ..

lemma *eventually-at-to-0*:
 $\text{eventually } P (\text{at } a) \longleftrightarrow \text{eventually } (\lambda x. P (x + a)) (\text{at } 0)$
for $a :: 'a::\text{real-normed-vector}$
unfolding *at-to-0*[*of a*] **by** (*simp add: eventually-filtermap*)

lemma *filtermap-at-minus*: $\text{filtermap } (\lambda x. - x) (\text{at } a) = \text{at } (- a)$
for $a :: 'a::\text{real-normed-vector}$
by (*simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus*[*symmetric*])

lemma *at-left-minus*: $\text{at-left } a = \text{filtermap } (\lambda x. - x) (\text{at-right } (- a))$
for $a :: \text{real}$
by (*simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus*[*symmetric*])

lemma *at-right-minus*: $\text{at-right } a = \text{filtermap } (\lambda x. - x) (\text{at-left } (- a))$
for $a :: \text{real}$
by (*simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus*[*symmetric*])

lemma *filterlim-at-left-to-right*:
 $\text{filterlim } f F (\text{at-left } a) \longleftrightarrow \text{filterlim } (\lambda x. f (- x)) F (\text{at-right } (- a))$
for $a :: \text{real}$
unfolding *filterlim-def filtermap-filtermap at-left-minus*[*of a*] ..

lemma *eventually-at-left-to-right*:
eventually P (*at-left* a) \longleftrightarrow *eventually* $(\lambda x. P (- x))$ (*at-right* $(-a)$)
for $a :: \text{real}$
unfolding *at-left-minus*[*of a*] **by** (*simp add: eventually-filtermap*)

lemma *filterlim-uminus-at-top-at-bot*: $LIM\ x\ \text{at-bot. } - x :: \text{real} :> \text{at-top}$
unfolding *filterlim-at-top eventually-at-bot-dense*
by (*metis leI minus-less-iff order-less-asm*)

lemma *filterlim-uminus-at-bot-at-top*: $LIM\ x\ \text{at-top. } - x :: \text{real} :> \text{at-bot}$
unfolding *filterlim-at-bot eventually-at-top-dense*
by (*metis leI less-minus-iff order-less-asm*)

lemma *at-bot-mirror* :
shows (*at-bot::('a::{ordered-ab-group-add,linorder} filter)*) = *filtermap* *uminus*
at-top
proof (*rule filtermap-fun-inverse[symmetric]*)
show *filterlim* *uminus* *at-top* (*at-bot::'a filter*)
using *eventually-at-bot-linorder filterlim-at-top le-minus-iff* **by** *force*
show *filterlim* *uminus* (*at-bot::'a filter*) *at-top*
by (*simp add: filterlim-at-bot minus-le-iff*)
qed *auto*

lemma *at-top-mirror* :
shows (*at-top::('a::{ordered-ab-group-add,linorder} filter)*) = *filtermap* *uminus*
at-bot
apply (*subst at-bot-mirror*)
by (*auto simp: filtermap-filtermap*)

lemma *filterlim-at-top-mirror*: $(LIM\ x\ \text{at-top. } f\ x :> F) \longleftrightarrow (LIM\ x\ \text{at-bot. } f\ (-x::\text{real}) :> F)$
unfolding *filterlim-def at-top-mirror filtermap-filtermap* ..

lemma *filterlim-at-bot-mirror*: $(LIM\ x\ \text{at-bot. } f\ x :> F) \longleftrightarrow (LIM\ x\ \text{at-top. } f\ (-x::\text{real}) :> F)$
unfolding *filterlim-def at-bot-mirror filtermap-filtermap* ..

lemma *filterlim-uminus-at-top*: $(LIM\ x\ F. f\ x :> \text{at-top}) \longleftrightarrow (LIM\ x\ F. - (f\ x) :: \text{real} :> \text{at-bot})$
using *filterlim-compose[OF filterlim-uminus-at-bot-at-top, of f F]*
and *filterlim-compose[OF filterlim-uminus-at-top-at-bot, of $\lambda x. - f\ x\ F$]*
by *auto*

lemma *tendsto-at-botI-sequentially*:
fixes $f :: \text{real} \Rightarrow 'b::\text{first-countable-topology}$
assumes $*$: $\bigwedge X. \text{filterlim } X\ \text{at-bot sequentially} \implies (\lambda n. f\ (X\ n)) \longrightarrow y$
shows $(f \longrightarrow y)$ *at-bot*
unfolding *filterlim-at-bot-mirror*
proof (*rule tendsto-at-topI-sequentially*)

```

fix X :: nat  $\Rightarrow$  real assume filterlim X at-top sequentially
thus ( $\lambda n. f (-X n)$ )  $\longrightarrow$  y by (intro *) (auto simp: filterlim-uminus-at-top)
qed

```

```

lemma filterlim-at-infinity-imp-filterlim-at-top:
assumes filterlim (f :: 'a  $\Rightarrow$  real) at-infinity F
assumes eventually ( $\lambda x. f x > 0$ ) F
shows filterlim f at-top F
proof -
from assms(2) have *: eventually ( $\lambda x. \text{norm } (f x) = f x$ ) F by eventually-elim
simp
from assms(1) show ?thesis unfolding filterlim-at-infinity-conv-norm-at-top
by (subst (asm) filterlim-cong[OF refl refl *])
qed

```

```

lemma filterlim-at-infinity-imp-filterlim-at-bot:
assumes filterlim (f :: 'a  $\Rightarrow$  real) at-infinity F
assumes eventually ( $\lambda x. f x < 0$ ) F
shows filterlim f at-bot F
proof -
from assms(2) have *: eventually ( $\lambda x. \text{norm } (f x) = -f x$ ) F by eventually-elim
simp
from assms(1) have filterlim ( $\lambda x. -f x$ ) at-top F
unfolding filterlim-at-infinity-conv-norm-at-top
by (subst (asm) filterlim-cong[OF refl refl *])
thus ?thesis by (simp add: filterlim-uminus-at-top)
qed

```

```

lemma filterlim-uminus-at-bot: (LIM x F. f x :> at-bot)  $\longleftrightarrow$  (LIM x F. - (f x) ::
real :> at-top)
unfolding filterlim-uminus-at-top by simp

```

```

lemma filterlim-inverse-at-top-right: LIM x at-right (0::real). inverse x :> at-top
unfolding filterlim-at-top-gt[where c=0] eventually-at-filter
proof safe
fix Z :: real
assume [arith]: 0 < Z
then have eventually ( $\lambda x. x < \text{inverse } Z$ ) (nhds 0)
by (auto simp: eventually-nhds-metric dist-real-def intro!: exI[of - |inverse Z|])
then show eventually ( $\lambda x. x \neq 0 \longrightarrow x \in \{0<..\} \longrightarrow Z \leq \text{inverse } x$ ) (nhds 0)
by (auto elim!: eventually-mono simp: inverse-eq-divide field-simps)
qed

```

```

lemma tendsto-inverse-0:
fixes x :: -  $\Rightarrow$  'a::real-normed-div-algebra
shows (inverse  $\longrightarrow$  (0::'a)) at-infinity
unfolding tendsto-Zfun-iff diff-0-right Zfun-def eventually-at-infinity
proof safe
fix r :: real

```

```

assume  $0 < r$ 
show  $\exists b. \forall x. b \leq \text{norm } x \longrightarrow \text{norm } (\text{inverse } x :: 'a) < r$ 
proof (intro exI[of - inverse (r / 2)] allI impI)
  fix  $x :: 'a$ 
  from  $\langle 0 < r \rangle$  have  $0 < \text{inverse } (r / 2)$  by simp
  also assume  $*: \text{inverse } (r / 2) \leq \text{norm } x$ 
  finally show  $\text{norm } (\text{inverse } x) < r$ 
    using  $* \langle 0 < r \rangle$ 
    by (subst nonzero-norm-inverse) (simp-all add: inverse-eq-divide field-simps)
qed
qed

```

lemma *tendsto-add-filterlim-at-infinity*:

```

fixes  $c :: 'b::\text{real-normed-vector}$ 
  and  $F :: 'a \text{ filter}$ 
assumes  $(f \longrightarrow c) F$ 
  and filterlim g at-infinity F
shows filterlim  $(\lambda x. f x + g x)$  at-infinity F
proof (subst filterlim-at-infinity[OF order-refl], safe)
  fix  $r :: \text{real}$ 
  assume  $r: r > 0$ 
  from assms(1) have  $(\lambda x. \text{norm } (f x)) \longrightarrow \text{norm } c) F$ 
    by (rule tendsto-norm)
  then have eventually  $(\lambda x. \text{norm } (f x) < \text{norm } c + 1) F$ 
    by (rule order-tendstoD) simp-all
  moreover from  $r$  have  $r + \text{norm } c + 1 > 0$ 
    by (intro add-pos-nonneg) simp-all
  with assms(2) have eventually  $(\lambda x. \text{norm } (g x) \geq r + \text{norm } c + 1) F$ 
    unfolding filterlim-at-infinity[OF order-refl]
    by (elim allE[of - r + norm c + 1]) simp-all
  ultimately show eventually  $(\lambda x. \text{norm } (f x + g x) \geq r) F$ 
proof eventually-elim
  fix  $x :: 'a$ 
  assume  $A: \text{norm } (f x) < \text{norm } c + 1$  and  $B: r + \text{norm } c + 1 \leq \text{norm } (g x)$ 
  from  $A B$  have  $r \leq \text{norm } (g x) - \text{norm } (f x)$ 
    by simp
  also have  $\text{norm } (g x) - \text{norm } (f x) \leq \text{norm } (g x + f x)$ 
    by (rule norm-diff-ineq)
  finally show  $r \leq \text{norm } (f x + g x)$ 
    by (simp add: add-ac)
qed
qed

```

lemma *tendsto-add-filterlim-at-infinity'*:

```

fixes  $c :: 'b::\text{real-normed-vector}$ 
  and  $F :: 'a \text{ filter}$ 
assumes filterlim f at-infinity F
  and  $(g \longrightarrow c) F$ 
shows filterlim  $(\lambda x. f x + g x)$  at-infinity F

```

by (subst add.commute) (rule tendsto-add-filterlim-at-infinity assms)+

lemma filterlim-inverse-at-right-top: $LIM\ x\ at\ top.\ inverse\ x\ :>\ at\ right\ (0::real)$
unfolding filterlim-at
by (auto simp: eventually-at-top-dense)
 (metis tendsto-inverse-0 filterlim-mono at-top-le-at-infinity order-refl)

lemma filterlim-inverse-at-top:
 $(f \longrightarrow (0 :: real))\ F \implies eventually\ (\lambda x.\ 0 < f\ x)\ F \implies LIM\ x\ F.\ inverse\ (f\ x)\ :>\ at\ top$
by (intro filterlim-compose[OF filterlim-inverse-at-top-right])
 (simp add: filterlim-def eventually-filtermap eventually-mono at-within-def le-principal)

lemma filterlim-inverse-at-bot-neg:
 $LIM\ x\ (at\ left\ (0::real)).\ inverse\ x\ :>\ at\ bot$
by (simp add: filterlim-inverse-at-top-right filterlim-uminus-at-bot filterlim-at-left-to-right)

lemma filterlim-inverse-at-bot:
 $(f \longrightarrow (0 :: real))\ F \implies eventually\ (\lambda x.\ f\ x < 0)\ F \implies LIM\ x\ F.\ inverse\ (f\ x)\ :>\ at\ bot$
unfolding filterlim-uminus-at-bot inverse-minus-eq[symmetric]
by (rule filterlim-inverse-at-top) (simp-all add: tendsto-minus-cancel-left[symmetric])

lemma at-right-to-top: $(at\ right\ (0::real)) = filtermap\ inverse\ at\ top$
by (intro filtermap-fun-inverse[symmetric, where g=inverse])
 (auto intro: filterlim-inverse-at-top-right filterlim-inverse-at-right-top)

lemma eventually-at-right-to-top:
 $eventually\ P\ (at\ right\ (0::real)) \iff eventually\ (\lambda x.\ P\ (inverse\ x))\ at\ top$
unfolding at-right-to-top eventually-filtermap ..

lemma filterlim-at-right-to-top:
 $filterlim\ f\ F\ (at\ right\ (0::real)) \iff (LIM\ x\ at\ top.\ f\ (inverse\ x)\ :>\ F)$
unfolding filterlim-def at-right-to-top filtermap-filtermap ..

lemma at-top-to-right: $at\ top = filtermap\ inverse\ (at\ right\ (0::real))$
unfolding at-right-to-top filtermap-filtermap inverse-inverse-eq filtermap-ident ..

lemma eventually-at-top-to-right:
 $eventually\ P\ at\ top \iff eventually\ (\lambda x.\ P\ (inverse\ x))\ (at\ right\ (0::real))$
unfolding at-top-to-right eventually-filtermap ..

lemma filterlim-at-top-to-right:
 $filterlim\ f\ F\ at\ top \iff (LIM\ x\ (at\ right\ (0::real)).\ f\ (inverse\ x)\ :>\ F)$
unfolding filterlim-def at-top-to-right filtermap-filtermap ..

lemma filterlim-inverse-at-infinity:
fixes $x :: - \Rightarrow 'a::\{real\text{-normed-div-algebra},\ division\text{-ring}\}$

```

shows filterlim inverse at-infinity (at (0::'a))
unfolding filterlim-at-infinity[OF order-refl]
proof safe
  fix r :: real
  assume 0 < r
  then show eventually ( $\lambda x::'a. r \leq \text{norm} (\text{inverse } x)$ ) (at 0)
    unfolding eventually-at norm-inverse
    by (intro exI[of - inverse r])
      (auto simp: norm-conv-dist[symmetric] field-simps inverse-eq-divide)
qed

lemma filterlim-inverse-at-iff:
  fixes g :: 'a  $\Rightarrow$  'b::{real-normed-div-algebra, division-ring}
  shows (LIM x F. inverse (g x) :> at 0)  $\longleftrightarrow$  (LIM x F. g x :> at-infinity)
  unfolding filterlim-def filtermap-filtermap[symmetric]
proof
  assume filtermap g F  $\leq$  at-infinity
  then have filtermap inverse (filtermap g F)  $\leq$  filtermap inverse at-infinity
    by (rule filtermap-mono)
  also have ...  $\leq$  at 0
    using tendsto-inverse-0[where 'a='b]
    by (auto intro!: exI[of - 1]
      simp: le-principal eventually-filtermap filterlim-def at-within-def eventually-at-infinity)
  finally show filtermap inverse (filtermap g F)  $\leq$  at 0 .
next
  assume filtermap inverse (filtermap g F)  $\leq$  at 0
  then have filtermap inverse (filtermap inverse (filtermap g F))  $\leq$  filtermap inverse at 0
    by (rule filtermap-mono)
  with filterlim-inverse-at-infinity show filtermap g F  $\leq$  at-infinity
    by (auto intro: order-trans simp: filterlim-def filtermap-filtermap)
qed

lemma tendsto-mult-filterlim-at-infinity:
  fixes c :: 'a::real-normed-field
  assumes (f  $\longrightarrow$  c) F c  $\neq$  0
  assumes filterlim g at-infinity F
  shows filterlim ( $\lambda x. f x * g x$ ) at-infinity F
proof -
  have (( $\lambda x. \text{inverse} (f x) * \text{inverse} (g x)$ )  $\longrightarrow$  inverse c * 0) F
    by (intro tendsto-mult tendsto-inverse assms filterlim-compose[OF tendsto-inverse-0])
  then have filterlim ( $\lambda x. \text{inverse} (f x) * \text{inverse} (g x)$ ) (at (inverse c * 0)) F
    unfolding filterlim-at
    using assms
    by (auto intro: filterlim-at-infinity-imp-eventually-ne tendsto-imp-eventually-ne eventually-conj)
  then show ?thesis
    by (subst filterlim-inverse-at-iff[symmetric]) simp-all

```

qed

lemma *filterlim-power-int-neg-at-infinity*:

fixes $f :: - \Rightarrow 'a::\{\text{real-normed-div-algebra, division-ring}\}$

assumes $n < 0$ **and** $\text{lim}: (f \longrightarrow 0) F$ **and** $\text{ev}: \text{eventually } (\lambda x. f x \neq 0) F$

shows $\text{filterlim } (\lambda x. f x \text{ powi } n) \text{ at-infinity } F$

proof –

have $\text{lim}' : ((\lambda x. f x \wedge \text{nat } (-n)) \longrightarrow 0) F$

by (rule *tendsto-eq-intros* *lim*) + (use $\langle n < 0 \rangle$ **in** *auto*)

have $\text{ev}' : \text{eventually } (\lambda x. f x \wedge \text{nat } (-n) \neq 0) F$

using *ev* **by** *eventually-elim* (use $\langle n < 0 \rangle$ **in** *auto*)

have $\text{filterlim } (\lambda x. \text{inverse } (f x \wedge \text{nat } (-n))) \text{ at-infinity } F$

by (*intro filterlim-compose*[*OF filterlim-inverse-at-infinity*])

(use lim' ev' **in** $\langle \text{auto simp: filterlim-at} \rangle$)

thus *?thesis*

using $\langle n < 0 \rangle$ **by** (*simp add: power-int-def power-inverse*)

qed

lemma *tendsto-inverse-0-at-top*: $\text{LIM } x F. f x := \text{at-top} \Longrightarrow ((\lambda x. \text{inverse } (f x) :: \text{real}) \longrightarrow 0) F$

by (*metis filterlim-at filterlim-mono*[*OF - at-top-le-at-infinity order-refl*] *filterlim-inverse-at-iff*)

lemma *filterlim-inverse-at-top-iff*:

$\text{eventually } (\lambda x. 0 < f x) F \Longrightarrow (\text{LIM } x F. \text{inverse } (f x) := \text{at-top}) \longleftrightarrow (f \longrightarrow (0 :: \text{real})) F$

by (*auto dest: tendsto-inverse-0-at-top filterlim-inverse-at-top*)

lemma *filterlim-at-top-iff-inverse-0*:

$\text{eventually } (\lambda x. 0 < f x) F \Longrightarrow (\text{LIM } x F. f x := \text{at-top}) \longleftrightarrow ((\text{inverse} \circ f) \longrightarrow (0 :: \text{real})) F$

using *filterlim-inverse-at-top-iff* [*of inverse* \circ *f*] **by** *auto*

lemma *real-tendsto-divide-at-top*:

fixes $c::\text{real}$

assumes $(f \longrightarrow c) F$

assumes $\text{filterlim } g \text{ at-top } F$

shows $((\lambda x. f x / g x) \longrightarrow 0) F$

by (*auto simp: divide-inverse-commute*

intro!: *tendsto-mult*[*THEN tendsto-eq-rhs*] *tendsto-inverse-0-at-top assms*)

lemma *mult-nat-left-at-top*: $c > 0 \Longrightarrow \text{filterlim } (\lambda x. c * x) \text{ at-top sequentially}$

for $c :: \text{nat}$

by (rule *filterlim-subseq*) (*auto simp: strict-mono-def*)

lemma *mult-nat-right-at-top*: $c > 0 \Longrightarrow \text{filterlim } (\lambda x. x * c) \text{ at-top sequentially}$

for $c :: \text{nat}$

by (rule *filterlim-subseq*) (*auto simp: strict-mono-def*)

lemma *filterlim-times-pos*:
 $LIM\ x\ F1.\ c * f\ x\ :>\ at_right\ l$
if $filterlim\ f\ (at_right\ p)\ F1\ 0 < c\ l = c * p$
for $c :: 'a :: \{linordered-field, linorder-topology\}$
unfolding *filterlim-iff*
proof *safe*
fix P
assume $\forall_F\ x\ in\ at_right\ l.\ P\ x$
then obtain d **where** $c * p < d \wedge y > c * p \implies y < d \implies P\ y$
unfolding $\langle l = - \rangle\ eventually-at-right-field$
by *auto*
then have $\forall_F\ a\ in\ at_right\ p.\ P\ (c * a)$
by (*auto simp: eventually-at-right-field* $\langle 0 < c \rangle\ field-simps\ intro!$: *exI*[**where** $x=d/c$])
from *that*(1)[*unfolded filterlim-iff, rule-format, OF this*]
show $\forall_F\ x\ in\ F1.\ P\ (c * f\ x)$.
qed

lemma *filtermap-nhds-times*: $c \neq 0 \implies filtermap\ (times\ c)\ (nhds\ a) = nhds\ (c * a)$
for $a\ c :: 'a :: real-normed-field$
by (*rule filtermap-fun-inverse*[**where** $g=\lambda x.\ inverse\ c * x$])
(auto intro!: tendsto-eq-intros filterlim-ident)

lemma *filtermap-times-pos-at-right*:
fixes $c :: 'a :: \{linordered-field, linorder-topology\}$
assumes $c > 0$
shows $filtermap\ (times\ c)\ (at_right\ p) = at_right\ (c * p)$
using *assms*
by (*intro filtermap-fun-inverse*[**where** $g=\lambda x.\ inverse\ c * x$])
(auto intro!: filterlim-ident filterlim-times-pos)

lemma *at-to-infinity*: $(at\ (0 :: 'a :: \{real-normed-field, field\})) = filtermap\ inverse\ at_infinity$
proof (*rule antisym*)
have $(inverse \longrightarrow (0 :: 'a))\ at_infinity$
by (*fact tendsto-inverse-0*)
then show $filtermap\ inverse\ at_infinity \leq at\ (0 :: 'a)$
using *filterlim-def filterlim-ident filterlim-inverse-at-iff* **by** *fastforce*
next
have $filtermap\ inverse\ (filtermap\ inverse\ (at\ (0 :: 'a))) \leq filtermap\ inverse\ at_infinity$
using *filterlim-inverse-at-infinity* **unfolding** *filterlim-def*
by (*rule filtermap-mono*)
then show $at\ (0 :: 'a) \leq filtermap\ inverse\ at_infinity$
by (*simp add: filtermap-ident filtermap-filtermap*)
qed

lemma *lim-at-infinity-0*:
fixes $l :: 'a :: \{real-normed-field, field\}$
shows $(f \longrightarrow l)\ at_infinity \iff ((f \circ inverse) \longrightarrow l)\ (at\ (0 :: 'a))$

by (simp add: tendsto-compose-filtermap at-to-infinity filtermap-filtermap)

lemma *lim-zero-infinity*:

fixes $l :: 'a::\{\text{real-normed-field}, \text{field}\}$

shows $((\lambda x. f(1/x)) \longrightarrow l) \text{ (at } (0::'a)) \implies (f \longrightarrow l) \text{ at-infinity}$

by (simp add: inverse-eq-divide lim-at-infinity-0 comp-def)

We only show rules for multiplication and addition when the functions are either against a real value or against infinity. Further rules are easy to derive by using *filterlim ?f at-top ?F = (LIM x ?F. - ?f x :> at-bot)*.

lemma *filterlim-tendsto-pos-mult-at-top*:

assumes $f: (f \longrightarrow c) F$

and $c: 0 < c$

and $g: LIM x F. g x :> at-top$

shows $LIM x F. (f x * g x :: real) :> at-top$

unfolding *filterlim-at-top-gt*[where $c=0$]

proof *safe*

fix $Z :: real$

assume $0 < Z$

from $f \langle 0 < c \rangle$ have eventually $(\lambda x. c / 2 < f x) F$

by (auto dest!: *tendstoD*[where $e=c/2$] *elim!*: *eventually-mono*
simp: dist-real-def abs-real-def split: if-split-asm)

moreover from g have eventually $(\lambda x. (Z / c * 2) \leq g x) F$

unfolding *filterlim-at-top* by *auto*

ultimately show eventually $(\lambda x. Z \leq f x * g x) F$

proof *eventually-elim*

case (elim x)

with $\langle 0 < Z \rangle \langle 0 < c \rangle$ have $c / 2 * (Z / c * 2) \leq f x * g x$

by (*intro mult-mono*) (*auto simp: zero-le-divide-iff*)

with $\langle 0 < c \rangle$ show $Z \leq f x * g x$

by *simp*

qed

qed

lemma *filterlim-at-top-mult-at-top*:

assumes $f: LIM x F. f x :> at-top$

and $g: LIM x F. g x :> at-top$

shows $LIM x F. (f x * g x :: real) :> at-top$

unfolding *filterlim-at-top-gt*[where $c=0$]

proof *safe*

fix $Z :: real$

assume $0 < Z$

from f have eventually $(\lambda x. 1 \leq f x) F$

unfolding *filterlim-at-top* by *auto*

moreover from g have eventually $(\lambda x. Z \leq g x) F$

unfolding *filterlim-at-top* by *auto*

ultimately show eventually $(\lambda x. Z \leq f x * g x) F$

proof *eventually-elim*

case (elim x)

```

with ⟨0 < Z⟩ have 1 * Z ≤ f x * g x
  by (intro mult-mono) (auto simp: zero-le-divide-iff)
then show Z ≤ f x * g x
  by simp
qed
qed

```

```

lemma filterlim-at-top-mult-tendsto-pos:
  assumes f: (f ⟶ c) F
    and c: 0 < c
    and g: LIM x F. g x :> at-top
  shows LIM x F. (g x * f x :: real) :> at-top
  by (auto simp: mult.commute intro!: filterlim-tendsto-pos-mult-at-top f c g)

```

```

lemma filterlim-tendsto-pos-mult-at-bot:
  fixes c :: real
  assumes (f ⟶ c) F 0 < c filterlim g at-bot F
  shows LIM x F. f x * g x :> at-bot
  using filterlim-tendsto-pos-mult-at-top[OF assms(1,2), of λx. - g x] assms(3)
  unfolding filterlim-uminus-at-bot by simp

```

```

lemma filterlim-tendsto-neg-mult-at-bot:
  fixes c :: real
  assumes c: (f ⟶ c) F c < 0 and g: filterlim g at-top F
  shows LIM x F. f x * g x :> at-bot
  using c filterlim-tendsto-pos-mult-at-top[of λx. - f x - c F, OF - - g]
  unfolding filterlim-uminus-at-bot tendsto-minus-cancel-left by simp

```

```

lemma filterlim-cmult-at-bot-at-top:
  assumes filterlim (h :: - ⇒ real) at-top F c ≠ 0 G = (if c > 0 then at-top else at-bot)
  shows filterlim (λx. c * h x) G F
  using assms filterlim-tendsto-pos-mult-at-top[OF tendsto-const[of c], of h F]
    filterlim-tendsto-neg-mult-at-bot[OF tendsto-const[of c], of h F] by simp

```

```

lemma filterlim-pow-at-top:
  fixes f :: 'a ⇒ real
  assumes 0 < n
    and f: LIM x F. f x :> at-top
  shows LIM x F. (f x) ^ n :: real :> at-top
  using ⟨0 < n⟩
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n) with f show ?case
  by (cases n = 0) (auto intro!: filterlim-at-top-mult-at-top)
qed

```

lemma *filterlim-pow-at-bot-even*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $0 < n \Rightarrow \text{LIM } x F. f x :> \text{at-bot} \Rightarrow \text{even } n \Rightarrow \text{LIM } x F. (f x)^{\wedge} n :> \text{at-top}$

using *filterlim-pow-at-top*[of $n \lambda x. - f x F$] **by** (*simp add: filterlim-uminus-at-top*)

lemma *filterlim-pow-at-bot-odd*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $0 < n \Rightarrow \text{LIM } x F. f x :> \text{at-bot} \Rightarrow \text{odd } n \Rightarrow \text{LIM } x F. (f x)^{\wedge} n :> \text{at-bot}$

using *filterlim-pow-at-top*[of $n \lambda x. - f x F$] **by** (*simp add: filterlim-uminus-at-bot*)

lemma *filterlim-power-at-infinity* [*tendsto-intros*]:

fixes F **and** $f :: 'a \Rightarrow 'b :: \text{real-normed-div-algebra}$

assumes *filterlim f at-infinity* $F n > 0$

shows *filterlim* $(\lambda x. f x^{\wedge} n)$ *at-infinity* F

by (*rule filterlim-norm-at-top-imp-at-infinity*)

(*auto simp: norm-power intro!: filterlim-pow-at-top assms*

intro: filterlim-at-infinity-imp-norm-at-top)

lemma *filterlim-tendsto-add-at-top*:

assumes $f: (f \longrightarrow c) F$

and $g: \text{LIM } x F. g x :> \text{at-top}$

shows $\text{LIM } x F. (f x + g x :: \text{real}) :> \text{at-top}$

unfolding *filterlim-at-top-gt*[**where** $c=0$]

proof *safe*

fix $Z :: \text{real}$

assume $0 < Z$

from f **have** *eventually* $(\lambda x. c - 1 < f x) F$

by (*auto dest!: tendstoD*[**where** $e=1$] *elim!: eventually-mono simp: dist-real-def*)

moreover from g **have** *eventually* $(\lambda x. Z - (c - 1) \leq g x) F$

unfolding *filterlim-at-top* **by** *auto*

ultimately show *eventually* $(\lambda x. Z \leq f x + g x) F$

by *eventually-elim simp*

qed

lemma *LIM-at-top-divide*:

fixes $f g :: 'a \Rightarrow \text{real}$

assumes $f: (f \longrightarrow a) F$ $0 < a$

and $g: (g \longrightarrow 0) F$ *eventually* $(\lambda x. 0 < g x) F$

shows $\text{LIM } x F. f x / g x :> \text{at-top}$

unfolding *divide-inverse*

by (*rule filterlim-tendsto-pos-mult-at-top*[*OF* f]) (*rule filterlim-inverse-at-top*[*OF* g])

lemma *filterlim-at-top-add-at-top*:

assumes $f: \text{LIM } x F. f x :> \text{at-top}$

and $g: \text{LIM } x F. g x :> \text{at-top}$

```

shows LIM x F. (f x + g x :: real) :> at-top
unfolding filterlim-at-top-gt[where c=0]
proof safe
  fix Z :: real
  assume 0 < Z
  from f have eventually (λx. 0 ≤ f x) F
    unfolding filterlim-at-top by auto
  moreover from g have eventually (λx. Z ≤ g x) F
    unfolding filterlim-at-top by auto
  ultimately show eventually (λx. Z ≤ f x + g x) F
    by eventually-elim simp
qed

lemma tendsto-divide-0:
  fixes f :: - ⇒ 'a::{real-normed-div-algebra, division-ring}
  assumes f: (f ⟶ c) F
    and g: LIM x F. g x :> at-infinity
  shows ((λx. f x / g x) ⟶ 0) F
  using tendsto-mult[OF f filterlim-compose[OF tendsto-inverse-0 g]]
  by (simp add: divide-inverse)

lemma linear-plus-1-le-power:
  fixes x :: real
  assumes x: 0 ≤ x
  shows real n * x + 1 ≤ (x + 1) ^ n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from x have real (Suc n) * x + 1 ≤ (x + 1) * (real n * x + 1)
    by (simp add: field-simps)
  also have ... ≤ (x + 1) ^ Suc n
    using Suc x by (simp add: mult-left-mono)
  finally show ?case .
qed

lemma filterlim-realpow-sequentially-gt1:
  fixes x :: 'a :: real-normed-div-algebra
  assumes x[arith]: 1 < norm x
  shows LIM n sequentially. x ^ n :> at-infinity
proof (intro filterlim-at-infinity[THEN iffD2] allI impI)
  fix y :: real
  assume 0 < y
  obtain N :: nat where y < real N * (norm x - 1)
    by (meson diff-gt-0-iff-gt reals-Archimedean3 x)
  also have ... ≤ real N * (norm x - 1) + 1
    by simp
  also have ... ≤ (norm x - 1 + 1) ^ N

```

by (rule linear-plus-1-le-power) simp
 also have $\dots = \text{norm } x \wedge N$
 by simp
 finally have $\forall n \geq N. y \leq \text{norm } x \wedge n$
 by (metis order-less-le-trans power-increasing order-less-imp-le x)
 then show eventually $(\lambda n. y \leq \text{norm } (x \wedge n))$ sequentially
 unfolding eventually-sequentially
 by (auto simp: norm-power)
 qed simp

lemma filterlim-divide-at-infinity:

fixes $f g :: 'a \Rightarrow 'a :: \text{real-normed-field}$
 assumes filterlim f (nhds c) F filterlim g (at 0) F $c \neq 0$
 shows filterlim $(\lambda x. f x / g x)$ at-infinity F

proof –

have filterlim $(\lambda x. f x * \text{inverse } (g x))$ at-infinity F
 by (intro tendsto-mult-filterlim-at-infinity [OF assms(1,3)]
 filterlim-compose [OF filterlim-inverse-at-infinity assms(2)])
 thus ?thesis by (simp add: field-simps)

qed

109.5 Floor and Ceiling

lemma eventually-floor-less:

fixes $f :: 'a \Rightarrow 'b :: \{\text{order-topology, floor-ceiling}\}$
 assumes $f: (f \longrightarrow l) F$
 and $l: l \notin \mathbb{Z}$
 shows $\forall_F x \text{ in } F. \text{of-int } (\text{floor } l) < f x$
 by (intro order-tendstoD [OF f]) (metis Ints-of-int antisym-conv2 floor-correct l)

lemma eventually-less-ceiling:

fixes $f :: 'a \Rightarrow 'b :: \{\text{order-topology, floor-ceiling}\}$
 assumes $f: (f \longrightarrow l) F$
 and $l: l \notin \mathbb{Z}$
 shows $\forall_F x \text{ in } F. f x < \text{of-int } (\text{ceiling } l)$
 by (intro order-tendstoD [OF f]) (metis Ints-of-int l le-of-int-ceiling less-le)

lemma eventually-floor-eq:

fixes $f :: 'a \Rightarrow 'b :: \{\text{order-topology, floor-ceiling}\}$
 assumes $f: (f \longrightarrow l) F$
 and $l: l \notin \mathbb{Z}$
 shows $\forall_F x \text{ in } F. \text{floor } (f x) = \text{floor } l$
 using eventually-floor-less [OF assms] eventually-less-ceiling [OF assms]
 by eventually-elim (meson floor-less-iff less-ceiling-iff not-less-iff-gr-or-eq)

lemma eventually-ceiling-eq:

fixes $f :: 'a \Rightarrow 'b :: \{\text{order-topology, floor-ceiling}\}$
 assumes $f: (f \longrightarrow l) F$

and $l: l \notin \mathbb{Z}$
shows $\forall_F x \text{ in } F. \text{ ceiling } (f x) = \text{ ceiling } l$
using *eventually-floor-less*[*OF assms*] *eventually-less-ceiling*[*OF assms*]
by *eventually-elim* (*meson floor-less-iff less-ceiling-iff not-less-iff-gr-or-eq*)

lemma *tendsto-of-int-floor*:
fixes $f::'a \Rightarrow 'b::\{\text{order-topology, floor-ceiling}\}$
assumes $(f \longrightarrow l) F$
and $l \notin \mathbb{Z}$
shows $((\lambda x. \text{ of-int } (\text{ floor } (f x)) :: 'c::\{\text{ring-1, topological-space}\}) \longrightarrow \text{ of-int } (\text{ floor } l)) F$
using *eventually-floor-eq*[*OF assms*]
by (*simp add: eventually-mono topological-tendstoI*)

lemma *tendsto-of-int-ceiling*:
fixes $f::'a \Rightarrow 'b::\{\text{order-topology, floor-ceiling}\}$
assumes $(f \longrightarrow l) F$
and $l \notin \mathbb{Z}$
shows $((\lambda x. \text{ of-int } (\text{ ceiling } (f x)) :: 'c::\{\text{ring-1, topological-space}\}) \longrightarrow \text{ of-int } (\text{ ceiling } l)) F$
using *eventually-ceiling-eq*[*OF assms*]
by (*simp add: eventually-mono topological-tendstoI*)

lemma *continuous-on-of-int-floor*:
continuous-on (*UNIV* - $\mathbb{Z}::'a::\{\text{order-topology, floor-ceiling}\}$ set)
 $(\lambda x. \text{ of-int } (\text{ floor } x) :: 'b::\{\text{ring-1, topological-space}\})$
unfolding *continuous-on-def*
by (*auto intro!: tendsto-of-int-floor*)

lemma *continuous-on-of-int-ceiling*:
continuous-on (*UNIV* - $\mathbb{Z}::'a::\{\text{order-topology, floor-ceiling}\}$ set)
 $(\lambda x. \text{ of-int } (\text{ ceiling } x) :: 'b::\{\text{ring-1, topological-space}\})$
unfolding *continuous-on-def*
by (*auto intro!: tendsto-of-int-ceiling*)

109.6 Limits of Sequences

lemma [*trans*]: $X = Y \Longrightarrow Y \longrightarrow z \Longrightarrow X \longrightarrow z$
by *simp*

lemma *LIMSEQ-iff*:
fixes $L :: 'a::\text{real-normed-vector}$
shows $(X \longrightarrow L) = (\forall r > 0. \exists \text{ no. } \forall n \geq \text{ no. } \text{ norm } (X n - L) < r)$
unfolding *lim-sequentially dist-norm ..*

lemma *LIMSEQ-I*: $(\bigwedge r. 0 < r \Longrightarrow \exists \text{ no. } \forall n \geq \text{ no. } \text{ norm } (X n - L) < r) \Longrightarrow X \longrightarrow L$
for $L :: 'a::\text{real-normed-vector}$
by (*simp add: LIMSEQ-iff*)

lemma *LIMSEQ-D*: $X \longrightarrow L \implies 0 < r \implies \exists no. \forall n \geq no. \text{norm } (X\ n - L) < r$
for $L :: 'a::\text{real-normed-vector}$
by (*simp add: LIMSEQ-iff*)

lemma *LIMSEQ-linear*: $X \longrightarrow x \implies l > 0 \implies (\lambda n. X\ (n * l)) \longrightarrow x$
unfolding *tendsto-def eventually-sequentially*
by (*metis div-le-dividend div-mult-self1-is-m le-trans mult.commute*)

Transformation of limit.

lemma *Lim-transform*: $(g \longrightarrow a)\ F \implies ((\lambda x. f\ x - g\ x) \longrightarrow 0)\ F \implies (f \longrightarrow a)\ F$
for $a\ b :: 'a::\text{real-normed-vector}$
using *tendsto-add [of g a F $\lambda x. f\ x - g\ x\ 0$]* **by** *simp*

lemma *Lim-transform2*: $(f \longrightarrow a)\ F \implies ((\lambda x. f\ x - g\ x) \longrightarrow 0)\ F \implies (g \longrightarrow a)\ F$
for $a\ b :: 'a::\text{real-normed-vector}$
by (*erule Lim-transform*) (*simp add: tendsto-minus-cancel*)

proposition *Lim-transform-eq*: $((\lambda x. f\ x - g\ x) \longrightarrow 0)\ F \implies (f \longrightarrow a)\ F \iff (g \longrightarrow a)\ F$
for $a :: 'a::\text{real-normed-vector}$
using *Lim-transform Lim-transform2* **by** *blast*

lemma *Lim-transform-eventually*:
 $\llbracket (f \longrightarrow l)\ F; \text{eventually } (\lambda x. f\ x = g\ x)\ F \rrbracket \implies (g \longrightarrow l)\ F$
using *eventually-elim2* **by** (*fastforce simp add: tendsto-def*)

lemma *Lim-transform-within*:
assumes $(f \longrightarrow l)$ (*at x within S*)
and $0 < d$
and $\bigwedge x'. x' \in S \implies 0 < \text{dist } x'\ x \implies \text{dist } x'\ x < d \implies f\ x' = g\ x'$
shows $(g \longrightarrow l)$ (*at x within S*)
proof (*rule Lim-transform-eventually*)
show *eventually* $(\lambda x. f\ x = g\ x)$ (*at x within S*)
using *assms* **by** (*auto simp: eventually-at*)
show $(f \longrightarrow l)$ (*at x within S*)
by *fact*
qed

lemma *filterlim-transform-within*:
assumes *filterlim* $g\ G$ (*at x within S*)
assumes $G \leq F$ $0 < d$ $(\bigwedge x'. x' \in S \implies 0 < \text{dist } x'\ x \implies \text{dist } x'\ x < d \implies f\ x' = g\ x')$
shows *filterlim* $f\ F$ (*at x within S*)
using *assms*
apply (*elim filterlim-mono-eventually*)

unfolding *eventually-at* **by** *auto*

Common case assuming being away from some crucial point like 0.

lemma *Lim-transform-away-within*:

fixes $a\ b :: 'a::t1\text{-space}$

assumes $a \neq b$

and $\forall x \in S. x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$

and $(f \longrightarrow l)$ (at a within S)

shows $(g \longrightarrow l)$ (at a within S)

proof (rule *Lim-transform-eventually*)

show $(f \longrightarrow l)$ (at a within S)

by *fact*

show *eventually* $(\lambda x. f\ x = g\ x)$ (at a within S)

unfolding *eventually-at-topological*

by (rule *exI* [**where** $x = - \{b\}$]) (*simp* add: *open-Compl* *assms*)

qed

lemma *Lim-transform-away-at*:

fixes $a\ b :: 'a::t1\text{-space}$

assumes $ab: a \neq b$

and $fg: \forall x. x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$

and $fl: (f \longrightarrow l)$ (at a)

shows $(g \longrightarrow l)$ (at a)

using *Lim-transform-away-within*[*OF* ab , of *UNIV* $f\ g\ l$] $fg\ fl$ **by** *simp*

Alternatively, within an open set.

lemma *Lim-transform-within-open*:

assumes $(f \longrightarrow l)$ (at a within T)

and *open* s **and** $a \in s$

and $\bigwedge x. x \in s \implies x \neq a \implies f\ x = g\ x$

shows $(g \longrightarrow l)$ (at a within T)

proof (rule *Lim-transform-eventually*)

show *eventually* $(\lambda x. f\ x = g\ x)$ (at a within T)

unfolding *eventually-at-topological*

using *assms* **by** *auto*

show $(f \longrightarrow l)$ (at a within T) **by** *fact*

qed

A congruence rule allowing us to transform limits assuming not at point.

lemma *Lim-cong-within*:

assumes $a = b$

and $x = y$

and $S = T$

and $\bigwedge x. x \neq b \implies x \in T \implies f\ x = g\ x$

shows $(f \longrightarrow x)$ (at a within S) \longleftrightarrow $(g \longrightarrow y)$ (at b within T)

unfolding *tendsto-def* *eventually-at-topological*

using *assms* **by** *simp*

An unbounded sequence's inverse tends to 0.

lemma *LIMSEQ-inverse-zero*:

assumes $\bigwedge r::\text{real}. \exists N. \forall n \geq N. r < X\ n$

shows $(\lambda n. \text{inverse}(X\ n)) \longrightarrow 0$

apply (rule *filterlim-compose*[*OF tendsto-inverse-0*])

by (*metis assms eventually-at-top-linorderI filterlim-at-top-dense filterlim-at-top-imp-at-infinity*)

The sequence $(1::'a) / n$ tends to 0 as n tends to infinity.

lemma *LIMSEQ-inverse-real-of-nat*: $(\lambda n. \text{inverse}(\text{real}(\text{Suc}\ n))) \longrightarrow 0$

by (*metis filterlim-compose tendsto-inverse-0 filterlim-mono order-refl filterlim-Suc filterlim-compose*[*OF filterlim-real-sequentially*] *at-top-le-at-infinity*)

The sequence $r + (1::'a) / n$ tends to r as n tends to infinity is now easily proved.

lemma *LIMSEQ-inverse-real-of-nat-add*: $(\lambda n. r + \text{inverse}(\text{real}(\text{Suc}\ n))) \longrightarrow r$

using *tendsto-add* [*OF tendsto-const LIMSEQ-inverse-real-of-nat*] **by** *auto*

lemma *LIMSEQ-inverse-real-of-nat-add-minus*: $(\lambda n. r + -\text{inverse}(\text{real}(\text{Suc}\ n))) \longrightarrow r$

using *tendsto-add* [*OF tendsto-const tendsto-minus* [*OF LIMSEQ-inverse-real-of-nat*]] **by** *auto*

lemma *LIMSEQ-inverse-real-of-nat-add-minus-mult*: $(\lambda n. r * (1 + -\text{inverse}(\text{real}(\text{Suc}\ n)))) \longrightarrow r$

using *tendsto-mult* [*OF tendsto-const LIMSEQ-inverse-real-of-nat-add-minus* [*of 1*]] **by** *auto*

lemma *lim-inverse-n*: $((\lambda n. \text{inverse}(\text{of-nat}\ n)) \longrightarrow (0::'a::\text{real-normed-field}))$ *sequentially*

using *lim-1-over-n* **by** (*simp add: inverse-eq-divide*)

lemma *lim-inverse-n'*: $((\lambda n. 1 / n) \longrightarrow 0)$ *sequentially*

using *lim-inverse-n*

by (*simp add: inverse-eq-divide*)

lemma *LIMSEQ-Suc-n-over-n*: $(\lambda n. \text{of-nat}(\text{Suc}\ n) / \text{of-nat}\ n :: 'a :: \text{real-normed-field}) \longrightarrow 1$

proof (rule *Lim-transform-eventually*)

show *eventually* $(\lambda n. 1 + \text{inverse}(\text{of-nat}\ n :: 'a) = \text{of-nat}(\text{Suc}\ n) / \text{of-nat}\ n)$ *sequentially*

using *eventually-gt-at-top*[*of 0::nat*]

by *eventually-elim* (*simp add: field-simps*)

have $(\lambda n. 1 + \text{inverse}(\text{of-nat}\ n :: 'a) \longrightarrow 1 + 0)$

by (*intro tendsto-add tendsto-const lim-inverse-n*)

then show $(\lambda n. 1 + \text{inverse}(\text{of-nat}\ n :: 'a) \longrightarrow 1)$

by *simp*

qed

lemma *LIMSEQ-n-over-Suc-n*: $(\lambda n. \text{of-nat } n / \text{of-nat } (\text{Suc } n) :: 'a :: \text{real-normed-field}) \longrightarrow 1$

proof (rule *Lim-transform-eventually*)

show *eventually* $(\lambda n. \text{inverse } (\text{of-nat } (\text{Suc } n) / \text{of-nat } n :: 'a) = \text{of-nat } n / \text{of-nat } (\text{Suc } n))$ *sequentially*

using *eventually-gt-at-top*[*of 0::nat*]

by *eventually-elim* (*simp add: field-simps del: of-nat-Suc*)

have $(\lambda n. \text{inverse } (\text{of-nat } (\text{Suc } n) / \text{of-nat } n :: 'a)) \longrightarrow \text{inverse } 1$

by (*intro tendsto-inverse LIMSEQ-Suc-n-over-n simp-all*)

then show $(\lambda n. \text{inverse } (\text{of-nat } (\text{Suc } n) / \text{of-nat } n :: 'a)) \longrightarrow 1$

by *simp*

qed

109.7 Convergence on sequences

lemma *convergent-cong*:

assumes *eventually* $(\lambda x. f x = g x)$ *sequentially*

shows *convergent* $f \longleftrightarrow$ *convergent* g

unfolding *convergent-def*

by (*subst filterlim-cong[OF refl refl assms]*) (rule *refl*)

lemma *convergent-Suc-iff*: *convergent* $(\lambda n. f (\text{Suc } n)) \longleftrightarrow$ *convergent* f

by (*auto simp: convergent-def filterlim-sequentially-Suc*)

lemma *convergent-ignore-initial-segment*: *convergent* $(\lambda n. f (n + m)) =$ *convergent* f

proof (*induct m arbitrary: f*)

case 0

then show *?case* **by** *simp*

next

case $(\text{Suc } m)$

have *convergent* $(\lambda n. f (n + \text{Suc } m)) \longleftrightarrow$ *convergent* $(\lambda n. f (\text{Suc } n + m))$

by *simp*

also have $\dots \longleftrightarrow$ *convergent* $(\lambda n. f (n + m))$

by (*rule convergent-Suc-iff*)

also have $\dots \longleftrightarrow$ *convergent* f

by (*rule Suc*)

finally show *?case* .

qed

lemma *convergent-add*:

fixes $X Y :: \text{nat} \Rightarrow 'a :: \text{topological-monoid-add}$

assumes *convergent* $(\lambda n. X n)$

and *convergent* $(\lambda n. Y n)$

shows *convergent* $(\lambda n. X n + Y n)$

using *assms* **unfolding** *convergent-def* **by** (*blast intro: tendsto-add*)

lemma *convergent-sum*:

fixes $X :: 'a \Rightarrow \text{nat} \Rightarrow 'b :: \text{topological-comm-monoid-add}$

shows $(\bigwedge i. i \in A \implies \text{convergent } (\lambda n. X i n)) \implies \text{convergent } (\lambda n. \sum_{i \in A}. X i n)$

by (*induct A rule: infinite-finite-induct*) (*simp-all add: convergent-const convergent-add*)

lemma (*in bounded-linear*) *convergent*:

assumes *convergent* $(\lambda n. X n)$

shows *convergent* $(\lambda n. f (X n))$

using *assms unfolding convergent-def* **by** (*blast intro: tendsto*)

lemma (*in bounded-bilinear*) *convergent*:

assumes *convergent* $(\lambda n. X n)$

and *convergent* $(\lambda n. Y n)$

shows *convergent* $(\lambda n. X n ** Y n)$

using *assms unfolding convergent-def* **by** (*blast intro: tendsto*)

lemma *convergent-minus-iff*:

fixes $X :: \text{nat} \Rightarrow 'a::\text{topological-group-add}$

shows *convergent* $X \longleftrightarrow \text{convergent } (\lambda n. - X n)$

unfolding *convergent-def* **by** (*force dest: tendsto-minus*)

lemma *convergent-diff*:

fixes $X Y :: \text{nat} \Rightarrow 'a::\text{topological-group-add}$

assumes *convergent* $(\lambda n. X n)$

assumes *convergent* $(\lambda n. Y n)$

shows *convergent* $(\lambda n. X n - Y n)$

using *assms unfolding convergent-def* **by** (*blast intro: tendsto-diff*)

lemma *convergent-norm*:

assumes *convergent* f

shows *convergent* $(\lambda n. \text{norm } (f n))$

proof –

from *assms* **have** $f \longrightarrow \text{lim } f$

by (*simp add: convergent-LIMSEQ-iff*)

then **have** $(\lambda n. \text{norm } (f n)) \longrightarrow \text{norm } (\text{lim } f)$

by (*rule tendsto-norm*)

then **show** *?thesis*

by (*auto simp: convergent-def*)

qed

lemma *convergent-of-real*:

convergent $f \implies \text{convergent } (\lambda n. \text{of-real } (f n)) :: 'a::\text{real-normed-algebra-1}$

unfolding *convergent-def* **by** (*blast intro!: tendsto-of-real*)

lemma *convergent-add-const-iff*:

convergent $(\lambda n. c + f n :: 'a::\text{topological-ab-group-add}) \longleftrightarrow \text{convergent } f$

proof

assume *convergent* $(\lambda n. c + f n)$

from *convergent-diff*[*OF this convergent-const*[*of c*]] **show** *convergent* f

```

  by simp
next
  assume convergent f
  from convergent-add[OF convergent-const[of c] this] show convergent ( $\lambda n. c + f n$ )
  by simp
qed

```

lemma *convergent-add-const-right-iff*:
 $convergent (\lambda n. f n + c :: 'a::topological-ab-group-add) \longleftrightarrow convergent f$
using *convergent-add-const-iff*[of c f] **by** (*simp add: add-ac*)

lemma *convergent-diff-const-right-iff*:
 $convergent (\lambda n. f n - c :: 'a::topological-ab-group-add) \longleftrightarrow convergent f$
using *convergent-add-const-right-iff*[of f -c] **by** (*simp add: add-ac*)

lemma *convergent-mult*:
fixes $X Y :: nat \Rightarrow 'a::topological-semigroup-mult$
assumes $convergent (\lambda n. X n)$
and $convergent (\lambda n. Y n)$
shows $convergent (\lambda n. X n * Y n)$
using *assms unfolding convergent-def* **by** (*blast intro: tendsto-mult*)

lemma *convergent-mult-const-iff*:
assumes $c \neq 0$
shows $convergent (\lambda n. c * f n :: 'a::\{field,topological-semigroup-mult\}) \longleftrightarrow convergent f$
proof
assume $convergent (\lambda n. c * f n)$
from *assms convergent-mult*[OF this *convergent-const*[of inverse c]]
show $convergent f$ **by** (*simp add: field-simps*)
next
assume $convergent f$
from *convergent-mult*[OF *convergent-const*[of c] this] **show** $convergent (\lambda n. c * f n)$
by *simp*
qed

lemma *convergent-mult-const-right-iff*:
fixes $c :: 'a::\{field,topological-semigroup-mult\}$
assumes $c \neq 0$
shows $convergent (\lambda n. f n * c) \longleftrightarrow convergent f$
using *convergent-mult-const-iff*[OF *assms, of f*] **by** (*simp add: mult-ac*)

lemma *convergent-imp-Bseq*: $convergent f \implies Bseq f$
by (*simp add: Cauchy-Bseq convergent-Cauchy*)

A monotone sequence converges to its least upper bound.

lemma *LIMSEQ-incseq-SUP*:

fixes $X :: \text{nat} \Rightarrow 'a::\{\text{conditionally-complete-linorder, linorder-topology}\}$
assumes $u: \text{bdd-above} (\text{range } X)$
and $X: \text{incseq } X$
shows $X \longrightarrow (\text{SUP } i. X i)$
by (*rule order-tendstoI*)
(auto simp: eventually-sequentially u less-cSUP-iff
intro: X[THEN incseqD] less-le-trans cSUP-lessD[OF u])

lemma *LIMSEQ-decseq-INF:*

fixes $X :: \text{nat} \Rightarrow 'a::\{\text{conditionally-complete-linorder, linorder-topology}\}$
assumes $u: \text{bdd-below} (\text{range } X)$
and $X: \text{decseq } X$
shows $X \longrightarrow (\text{INF } i. X i)$
by (*rule order-tendstoI*)
(auto simp: eventually-sequentially u cINF-less-iff
intro: X[THEN decseqD] le-less-trans less-cINF-D[OF u])

Main monotonicity theorem.

lemma *Bseq-monoseq-convergent:* $Bseq\ X \Longrightarrow monoseq\ X \Longrightarrow convergent\ X$

for $X :: \text{nat} \Rightarrow \text{real}$

by (*auto simp: monoseq-iff convergent-def intro: LIMSEQ-decseq-INF LIMSEQ-incseq-SUP*
dest: Bseq-bdd-above Bseq-bdd-below)

lemma *Bseq-mono-convergent:* $Bseq\ X \Longrightarrow (\forall m\ n. m \leq n \longrightarrow X\ m \leq X\ n) \Longrightarrow convergent\ X$

for $X :: \text{nat} \Rightarrow \text{real}$

by (*auto intro!: Bseq-monoseq-convergent incseq-imp-monoseq simp: incseq-def*)

lemma *monoseq-imp-convergent-iff-Bseq:* $monoseq\ f \Longrightarrow convergent\ f \longleftrightarrow Bseq\ f$

for $f :: \text{nat} \Rightarrow \text{real}$

using *Bseq-monoseq-convergent[of f] convergent-imp-Bseq[of f]* **by** *blast*

lemma *Bseq-monoseq-convergent'-inc:*

fixes $f :: \text{nat} \Rightarrow \text{real}$

shows $Bseq\ (\lambda n. f\ (n + M)) \Longrightarrow (\bigwedge m\ n. M \leq m \Longrightarrow m \leq n \Longrightarrow f\ m \leq f\ n) \Longrightarrow convergent\ f$

by (*subst convergent-ignore-initial-segment [symmetric, of - M]*)
(auto intro!: Bseq-monoseq-convergent simp: monoseq-def)

lemma *Bseq-monoseq-convergent'-dec:*

fixes $f :: \text{nat} \Rightarrow \text{real}$

shows $Bseq\ (\lambda n. f\ (n + M)) \Longrightarrow (\bigwedge m\ n. M \leq m \Longrightarrow m \leq n \Longrightarrow f\ m \geq f\ n) \Longrightarrow convergent\ f$

by (*subst convergent-ignore-initial-segment [symmetric, of - M]*)
(auto intro!: Bseq-monoseq-convergent simp: monoseq-def)

lemma *Cauchy-iff:* $Cauchy\ X \longleftrightarrow (\forall e>0. \exists M. \forall m \geq M. \forall n \geq M. norm\ (X\ m - X\ n) < e)$

for $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$

unfolding *Cauchy-def dist-norm ..*

lemma *CauchyI*: $(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e) \implies \text{Cauchy } X$
for $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
by (*simp add: Cauchy-iff*)

lemma *CauchyD*: $\text{Cauchy } X \implies 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e$
for $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
by (*simp add: Cauchy-iff*)

lemma *incseq-convergent*:
fixes $X :: \text{nat} \Rightarrow \text{real}$
assumes *incseq X*
and $\forall i. X\ i \leq B$
obtains L **where** $X \longrightarrow L \wedge \forall i. X\ i \leq L$

proof *atomize-elim*

from *incseq-bounded*[*OF assms*] $\langle \text{incseq } X \rangle$ *Bseq-monoseq-convergent*[*of X*]
obtain L **where** $X \longrightarrow L$
by (*auto simp: convergent-def monoseq-def incseq-def*)
with $\langle \text{incseq } X \rangle$ **show** $\exists L. X \longrightarrow L \wedge (\forall i. X\ i \leq L)$
by (*auto intro!: exI[*of* - *L*] incseq-le*)

qed

lemma *decseq-convergent*:
fixes $X :: \text{nat} \Rightarrow \text{real}$
assumes *decseq X*
and $\forall i. B \leq X\ i$
obtains L **where** $X \longrightarrow L \wedge \forall i. L \leq X\ i$

proof *atomize-elim*

from *decseq-bounded*[*OF assms*] $\langle \text{decseq } X \rangle$ *Bseq-monoseq-convergent*[*of X*]
obtain L **where** $X \longrightarrow L$
by (*auto simp: convergent-def monoseq-def decseq-def*)
with $\langle \text{decseq } X \rangle$ **show** $\exists L. X \longrightarrow L \wedge (\forall i. L \leq X\ i)$
by (*auto intro!: exI[*of* - *L*] decseq-ge*)

qed

lemma *monoseq-convergent*:
fixes $X :: \text{nat} \Rightarrow \text{real}$
assumes X : *monoseq X* **and** B : $\bigwedge i. |X\ i| \leq B$
obtains L **where** $X \longrightarrow L$
using X **unfolding** *monoseq-iff*

proof

assume *incseq X*

show *thesis*

using *abs-le-D1* [*OF B*] *incseq-convergent* [*OF* $\langle \text{incseq } X \rangle$] **that** **by** *meson*

next

assume *decseq X*

```

show thesis
  using decseq-convergent [OF <decseq X>] that
  by (metis B abs-le-iff add.inverse-inverse neg-le-iff-le)
qed

```

109.8 More about *filterlim* (thanks to Wenda Li)

```

lemma filterlim-at-infinity-times:
  fixes f :: 'a  $\Rightarrow$  'b::real-normed-field
  assumes filterlim f at-infinity F filterlim g at-infinity F
  shows filterlim ( $\lambda x. f x * g x$ ) at-infinity F
proof –
  have ( $\lambda x. inverse (f x) * inverse (g x)$ )  $\longrightarrow 0 * 0$  F
  by (intro tendsto-mult tendsto-inverse assms filterlim-compose[OF tendsto-inverse-0])
  then have filterlim ( $\lambda x. inverse (f x) * inverse (g x)$ ) (at 0) F
  unfolding filterlim-at using assms
  by (auto intro: filterlim-at-infinity-imp-eventually-ne tendsto-imp-eventually-ne
eventually-conj)
  then show ?thesis
  by (subst filterlim-inverse-at-iff[symmetric]) simp-all
qed

```

```

lemma filterlim-at-top-at-bot[elim]:
  fixes f::'a  $\Rightarrow$  'b::{unbounded-dense-linorder} and F::'a filter
  assumes top:filterlim f at-top F and bot: filterlim f at-bot F and F $\neq$ bot
  shows False
proof –
  obtain c::'b where True by auto
  have  $\forall_F x \text{ in } F. c < f x$ 
  using top unfolding filterlim-at-top-dense by auto
  moreover have  $\forall_F x \text{ in } F. f x < c$ 
  using bot unfolding filterlim-at-bot-dense by auto
  ultimately have  $\forall_F x \text{ in } F. c < f x \wedge f x < c$ 
  using eventually-conj by auto
  then have  $\forall_F x \text{ in } F. False$  by (auto elim:eventually-mono)
  then show False using <F $\neq$ bot> by auto
qed

```

```

lemma filterlim-at-top-nhds[elim]:
  fixes f::'a  $\Rightarrow$  'b::{unbounded-dense-linorder,order-topology} and F::'a filter
  assumes top:filterlim f at-top F and tendsto: (f  $\longrightarrow$  c) F and F $\neq$ bot
  shows False
proof –
  obtain c'::'b where c'>c using gt-ex by blast
  have  $\forall_F x \text{ in } F. c' < f x$ 
  using top unfolding filterlim-at-top-dense by auto
  moreover have  $\forall_F x \text{ in } F. f x < c'$ 
  using order-tendstoD[OF tendsto,of c'] <c'>c by auto
  ultimately have  $\forall_F x \text{ in } F. c' < f x \wedge f x < c'$ 

```

using *eventually-conj* by *auto*
 then have $\forall_F x \text{ in } F. \text{ False}$ by (*auto elim:eventually-mono*)
 then show *False* using $\langle F \neq \text{bot} \rangle$ by *auto*
 qed

lemma *filterlim-at-bot-nhds*[*elim*]:
 fixes $f :: 'a \Rightarrow 'b :: \{\text{unbounded-dense-linorder}, \text{order-topology}\}$ and $F :: 'a \text{ filter}$
 assumes *top:filterlim f at-bot F* and *tendsto: (f \longrightarrow c) F* and $F \neq \text{bot}$
 shows *False*

proof –
 obtain $c' :: 'b$ where $c' < c$ using *lt-ex* by *blast*
 have $\forall_F x \text{ in } F. c' > f x$
 using *top unfolding filterlim-at-bot-dense* by *auto*
 moreover have $\forall_F x \text{ in } F. f x > c'$
 using *order-tendstoD[OF tendsto, of c'] $\langle c' < c \rangle$* by *auto*
 ultimately have $\forall_F x \text{ in } F. c' < f x \wedge f x < c'$
 using *eventually-conj* by *auto*
 then have $\forall_F x \text{ in } F. \text{ False}$ by (*auto elim:eventually-mono*)
 then show *False* using $\langle F \neq \text{bot} \rangle$ by *auto*
 qed

lemma *eventually-times-inverse-1*:
 fixes $f :: 'a \Rightarrow 'b :: \{\text{field}, \text{t2-space}\}$
 assumes $(f \longrightarrow c) F$ $c \neq 0$
 shows $\forall_F x \text{ in } F. \text{inverse}(f x) * f x = 1$
 by (*smt (verit) assms eventually-mono mult.commute right-inverse tendsto-imp-eventually-ne*)

lemma *filterlim-at-infinity-divide-iff*:
 fixes $f :: 'a \Rightarrow 'b :: \text{real-normed-field}$
 assumes $(f \longrightarrow c) F$ $c \neq 0$
 shows $(\text{LIM } x \text{ F. } f x / g x \text{ :> at-infinity}) \longleftrightarrow (\text{LIM } x \text{ F. } g x \text{ :> at } 0)$

proof
 assume $\text{LIM } x \text{ F. } f x / g x \text{ :> at-infinity}$
 then have $\text{LIM } x \text{ F. inverse}(f x) * (f x / g x) \text{ :> at-infinity}$
 using *assms tendsto-inverse tendsto-mult-filterlim-at-infinity* by *fastforce*
 then have $\text{LIM } x \text{ F. inverse}(g x) \text{ :> at-infinity}$
 apply (*elim filterlim-mono-eventually*)
 using *eventually-times-inverse-1[OF assms]*
 by (*auto elim:eventually-mono simp add:field-simps*)
 then show *filterlim g (at 0) F* using *filterlim-inverse-at-iff[symmetric]* by *force*

next
 assume *filterlim g (at 0) F*
 then have *filterlim (λx. inverse(g x)) at-infinity F*
 using *filterlim-compose filterlim-inverse-at-infinity* by *blast*
 then have $\text{LIM } x \text{ F. } f x * \text{inverse}(g x) \text{ :> at-infinity}$
 using *tendsto-mult-filterlim-at-infinity[OF assms, of λx. inverse(g x)]*
 by *simp*
 then show $\text{LIM } x \text{ F. } f x / g x \text{ :> at-infinity}$ by (*simp add: divide-inverse*)

qed

lemma *filterlim-tendsto-pos-mult-at-top-iff*:

fixes $f :: 'a \Rightarrow \text{real}$

assumes $(f \longrightarrow c) F$ **and** $0 < c$

shows $(\text{LIM } x F. (f x * g x) :> \text{at-top}) \longleftrightarrow (\text{LIM } x F. g x :> \text{at-top})$

proof

assume *filterlim g at-top F*

then show $\text{LIM } x F. f x * g x :> \text{at-top}$

using *filterlim-tendsto-pos-mult-at-top[OF assms]* **by auto**

next

assume *asm: LIM x F. f x * g x :> at-top*

have $((\lambda x. \text{inverse } (f x)) \longrightarrow \text{inverse } c) F$

using *tendsto-inverse[OF assms(1)]* $\langle 0 < c \rangle$ **by auto**

moreover have $\text{inverse } c > 0$ **using** *assms(2)* **by auto**

ultimately have $\text{LIM } x F. \text{inverse } (f x) * (f x * g x) :> \text{at-top}$

using *filterlim-tendsto-pos-mult-at-top[OF - - asm, of $\lambda x. \text{inverse } (f x)$ inverse c]* **by auto**

then show $\text{LIM } x F. g x :> \text{at-top}$

apply (*elim filterlim-mono-eventually*)

apply *simp-all[2]*

using *eventually-times-inverse-1[OF assms(1)]* $\langle c > 0 \rangle$ *eventually-mono* **by fast-force**

qed

lemma *filterlim-tendsto-pos-mult-at-bot-iff*:

fixes $c :: \text{real}$

assumes $(f \longrightarrow c) F$ $0 < c$

shows $(\text{LIM } x F. f x * g x :> \text{at-bot}) \longleftrightarrow \text{filterlim } g \text{ at-bot } F$

using *filterlim-tendsto-pos-mult-at-top-iff[OF assms(1,2), of $\lambda x. - g x$]*

unfolding *filterlim-uminus-at-bot* **by simp**

lemma *filterlim-tendsto-neg-mult-at-top-iff*:

fixes $f :: 'a \Rightarrow \text{real}$

assumes $(f \longrightarrow c) F$ **and** $c < 0$

shows $(\text{LIM } x F. (f x * g x) :> \text{at-top}) \longleftrightarrow (\text{LIM } x F. g x :> \text{at-bot})$

proof –

have $(\text{LIM } x F. f x * g x :> \text{at-top}) = (\text{LIM } x F. - g x :> \text{at-top})$

apply (*rule filterlim-tendsto-pos-mult-at-top-iff[of $\lambda x. - f x - c F \lambda x. - g x$, simplified]*)

using *assms* **by** (*auto intro: tendsto-intros*)

also have $\dots = (\text{LIM } x F. g x :> \text{at-bot})$

using *filterlim-uminus-at-bot[symmetric]* **by auto**

finally show *?thesis* .

qed

lemma *filterlim-tendsto-neg-mult-at-bot-iff*:

fixes $c :: \text{real}$

assumes $(f \longrightarrow c) F$ $0 > c$

shows $(\text{LIM } x F. f x * g x \text{ :> } \text{at-bot}) \iff \text{filterlim } g \text{ at-top } F$
using *filterlim-tendsto-neg-mult-at-top-iff* [*OF* *assms*(1,2), of $\lambda x. - g x$]
unfolding *filterlim-uminus-at-top* **by** *simp*

109.9 Power Sequences

lemma *Bseq-realpow*: $0 \leq x \implies x \leq 1 \implies \text{Bseq } (\lambda n. x \wedge n)$
for $x :: \text{real}$
by (*metis decseq-bounded decseq-def power-decreasing zero-le-power*)

lemma *monoseq-realpow*: $0 \leq x \implies x \leq 1 \implies \text{monoseq } (\lambda n. x \wedge n)$
for $x :: \text{real}$
using *monoseq-def power-decreasing* **by** *blast*

lemma *convergent-realpow*: $0 \leq x \implies x \leq 1 \implies \text{convergent } (\lambda n. x \wedge n)$
for $x :: \text{real}$
by (*blast intro!*: *Bseq-monoseq-convergent Bseq-realpow monoseq-realpow*)

lemma *LIMSEQ-inverse-realpow-zero*: $1 < x \implies (\lambda n. \text{inverse } (x \wedge n)) \longrightarrow 0$
for $x :: \text{real}$
by (*rule filterlim-compose* [*OF* *tendsto-inverse-0 filterlim-realpow-sequentially-gt1*])
simp

lemma *LIMSEQ-realpow-zero*:
fixes $x :: \text{real}$
assumes $0 \leq x \ x < 1$
shows $(\lambda n. x \wedge n) \longrightarrow 0$
proof (*cases* $x = 0$)
case *False*
with $\langle 0 \leq x \rangle$ **have** $1 < \text{inverse } x$
using $\langle x < 1 \rangle$ **by** (*simp add: one-less-inverse*)
then **have** $(\lambda n. \text{inverse } (\text{inverse } x \wedge n)) \longrightarrow 0$
by (*rule LIMSEQ-inverse-realpow-zero*)
then **show** *?thesis* **by** (*simp add: power-inverse*)
next
case *True*
show *?thesis*
by (*rule LIMSEQ-imp-Suc*) (*simp add: True*)
qed

lemma *LIMSEQ-power-zero* [*tendsto-intros*]: $\text{norm } x < 1 \implies (\lambda n. x \wedge n) \longrightarrow 0$
for $x :: 'a::\text{real-normed-algebra-1}$
apply (*drule LIMSEQ-realpow-zero* [*OF* *norm-ge-zero*])
by (*simp add: Zfun-le norm-power-ineq tendsto-Zfun-iff*)

lemma *LIMSEQ-divide-realpow-zero*: $1 < x \implies (\lambda n. a / (x \wedge n) :: \text{real}) \longrightarrow 0$
by (*rule tendsto-divide-0* [*OF* *tendsto-const filterlim-realpow-sequentially-gt1*])
simp

lemma

tendsto-power-zero:

fixes $x::'a::\text{real-normed-algebra-1}$

assumes *filterlim f at-top F*

assumes $\text{norm } x < 1$

shows $((\lambda y. x \hat{^} (f y)) \longrightarrow 0) F$

proof (*rule tendstoI*)

fix $e::\text{real}$ **assume** $0 < e$

from *tendstoD[OF LIMSEQ-power-zero[OF $\langle \text{norm } x < 1 \rangle \langle 0 < e \rangle$]*

have $\forall_F xa \text{ in sequentially. norm } (x \hat{^} xa) < e$

by *simp*

then obtain N **where** $N: \text{norm } (x \hat{^} n) < e$ **if** $n \geq N$ **for** n

by (*auto simp: eventually-sequentially*)

have $\forall_F i \text{ in } F. f i \geq N$

using $\langle \text{filterlim } f \text{ sequentially } F \rangle$

by (*simp add: filterlim-at-top*)

then show $\forall_F i \text{ in } F. \text{dist } (x \hat{^} f i) 0 < e$

by *eventually-elim (auto simp: N)*

qed

Limit of c^n for $|c| < (1::'a)$.

lemma *LIMSEQ-abs-realpow-zero*: $|c| < 1 \implies (\lambda n. |c| \hat{^} n :: \text{real}) \longrightarrow 0$

by (*rule LIMSEQ-realpow-zero [OF abs-ge-zero]*)

lemma *LIMSEQ-abs-realpow-zero2*: $|c| < 1 \implies (\lambda n. c \hat{^} n :: \text{real}) \longrightarrow 0$

by (*rule LIMSEQ-power-zero simp*)

109.10 Limits of Functions

lemma *LIM-eq*: $f -a \rightarrow L = (\forall r > 0. \exists s > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < s \longrightarrow \text{norm } (f x - L) < r)$

for $a :: 'a::\text{real-normed-vector}$ **and** $L :: 'b::\text{real-normed-vector}$

by (*simp add: LIM-def dist-norm*)

lemma *LIM-I*:

$(\bigwedge r. 0 < r \implies \exists s > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < s \longrightarrow \text{norm } (f x - L) < r) \implies f -a \rightarrow L$

for $a :: 'a::\text{real-normed-vector}$ **and** $L :: 'b::\text{real-normed-vector}$

by (*simp add: LIM-eq*)

lemma *LIM-D*: $f -a \rightarrow L \implies 0 < r \implies \exists s > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < s \longrightarrow \text{norm } (f x - L) < r$

for $a :: 'a::\text{real-normed-vector}$ **and** $L :: 'b::\text{real-normed-vector}$

by (*simp add: LIM-eq*)

lemma *LIM-offset*: $f -a \rightarrow L \implies (\lambda x. f (x + k)) -(a - k) \rightarrow L$

for $a :: 'a::\text{real-normed-vector}$

by (*simp add: filtermap-at-shift[symmetric, of a k] filterlim-def filtermap-filtermap*)

lemma *LIM-offset-zero*: $f -a \rightarrow L \implies (\lambda h. f (a + h)) -0 \rightarrow L$
for $a :: 'a::\text{real-normed-vector}$
by (*drule* *LIM-offset* [**where** $k = a$]) (*simp add: add.commute*)

lemma *LIM-offset-zero-cancel*: $(\lambda h. f (a + h)) -0 \rightarrow L \implies f -a \rightarrow L$
for $a :: 'a::\text{real-normed-vector}$
by (*drule* *LIM-offset* [**where** $k = - a$]) *simp*

lemma *LIM-offset-zero-iff*: *NO-MATCH* $0 a \implies f -a \rightarrow L \longleftrightarrow (\lambda h. f (a + h)) -0 \rightarrow L$
for $f :: 'a :: \text{real-normed-vector} \Rightarrow -$
using *LIM-offset-zero-cancel*[*of f a L*] *LIM-offset-zero*[*of f L a*] **by** *auto*

lemma *tendsto-offset-zero-iff*:
fixes $f :: 'a :: \text{real-normed-vector} \Rightarrow -$
assumes *NO-MATCH* $0 a a \in S$ *open S*
shows $(f \longrightarrow L)$ (*at a within S*) $\longleftrightarrow ((\lambda h. f (a + h)) \longrightarrow L)$ (*at 0*)
using *assms* **by** (*simp add: tendsto-within-open-NO-MATCH LIM-offset-zero-iff*)

lemma *LIM-zero*: $(f \longrightarrow l) F \implies ((\lambda x. f x - l) \longrightarrow 0) F$
for $f :: 'a \Rightarrow 'b::\text{real-normed-vector}$
unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *LIM-zero-cancel*:
fixes $f :: 'a \Rightarrow 'b::\text{real-normed-vector}$
shows $((\lambda x. f x - l) \longrightarrow 0) F \implies (f \longrightarrow l) F$
unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *LIM-zero-iff*: $((\lambda x. f x - l) \longrightarrow 0) F = (f \longrightarrow l) F$
for $f :: 'a \Rightarrow 'b::\text{real-normed-vector}$
unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *LIM-imp-LIM*:
fixes $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$
fixes $g :: 'a::\text{topological-space} \Rightarrow 'c::\text{real-normed-vector}$
assumes $f: f -a \rightarrow l$
and $le: \bigwedge x. x \neq a \implies \text{norm} (g x - m) \leq \text{norm} (f x - l)$
shows $g -a \rightarrow m$
by (*rule* *metric-LIM-imp-LIM* [*OF f*]) (*simp add: dist-norm le*)

lemma *LIM-equal2*:
fixes $f g :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$
assumes $0 < R$
and $\bigwedge x. x \neq a \implies \text{norm} (x - a) < R \implies f x = g x$
shows $g -a \rightarrow l \implies f -a \rightarrow l$
by (*rule* *metric-LIM-equal2* [*OF - assms*]) (*simp-all add: dist-norm*)

lemma *LIM-compose2*:

fixes $a :: 'a::\text{real-normed-vector}$
assumes $f: f -a \rightarrow b$
and $g: g -b \rightarrow c$
and inj: $\exists d>0. \forall x. x \neq a \wedge \text{norm } (x - a) < d \longrightarrow f x \neq b$
shows $(\lambda x. g (f x)) -a \rightarrow c$
by $(\text{rule metric-LIM-compose2 } [OF f g inj \text{ [folded dist-norm]})]$

lemma *real-LIM-sandwich-zero*:

fixes $f g :: 'a::\text{topological-space} \Rightarrow \text{real}$
assumes $f: f -a \rightarrow 0$
and 1: $\bigwedge x. x \neq a \Longrightarrow 0 \leq g x$
and 2: $\bigwedge x. x \neq a \Longrightarrow g x \leq f x$
shows $g -a \rightarrow 0$
proof $(\text{rule LIM-imp-LIM } [OF f])$
fix x
assume $x: x \neq a$
with 1 have $\text{norm } (g x - 0) = g x$ **by** *simp*
also have $g x \leq f x$ **by** $(\text{rule } 2 \text{ [OF } x])$
also have $f x \leq |f x|$ **by** $(\text{rule abs-ge-self})$
also have $|f x| = \text{norm } (f x - 0)$ **by** *simp*
finally show $\text{norm } (g x - 0) \leq \text{norm } (f x - 0)$.
qed

109.11 Continuity

lemma *LIM-isCont-iff*: $(f -a \rightarrow f a) = ((\lambda h. f (a + h)) -0 \rightarrow f a)$
for $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$
by $(\text{rule iffI } [OF \text{ LIM-offset-zero LIM-offset-zero-cancel}])$

lemma *isCont-iff*: $\text{isCont } f x = (\lambda h. f (x + h)) -0 \rightarrow f x$
for $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$
by $(\text{simp add: isCont-def LIM-isCont-iff})$

lemma *isCont-LIM-compose2*:

fixes $a :: 'a::\text{real-normed-vector}$
assumes $f \text{ [unfolded isCont-def]: isCont } f a$
and $g: g -f a \rightarrow l$
and inj: $\exists d>0. \forall x. x \neq a \wedge \text{norm } (x - a) < d \longrightarrow f x \neq f a$
shows $(\lambda x. g (f x)) -a \rightarrow l$
by $(\text{rule LIM-compose2 } [OF f g inj])$

lemma *isCont-norm [simp]*: $\text{isCont } f a \Longrightarrow \text{isCont } (\lambda x. \text{norm } (f x)) a$
for $f :: 'a::\text{t2-space} \Rightarrow 'b::\text{real-normed-vector}$
by $(\text{fact continuous-norm})$

lemma *isCont-rabs [simp]*: $\text{isCont } f a \Longrightarrow \text{isCont } (\lambda x. |f x|) a$
for $f :: 'a::\text{t2-space} \Rightarrow \text{real}$
by $(\text{fact continuous-rabs})$

lemma *isCont-add* [*simp*]: $isCont\ f\ a \implies isCont\ g\ a \implies isCont\ (\lambda x. f\ x + g\ x)\ a$
for $f :: 'a::t2-space \Rightarrow 'b::topological-monoid-add$
by (*fact continuous-add*)

lemma *isCont-minus* [*simp*]: $isCont\ f\ a \implies isCont\ (\lambda x. -\ f\ x)\ a$
for $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
by (*fact continuous-minus*)

lemma *isCont-diff* [*simp*]: $isCont\ f\ a \implies isCont\ g\ a \implies isCont\ (\lambda x. f\ x - g\ x)\ a$
for $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
by (*fact continuous-diff*)

lemma *isCont-mult* [*simp*]: $isCont\ f\ a \implies isCont\ g\ a \implies isCont\ (\lambda x. f\ x * g\ x)\ a$
for $f\ g :: 'a::t2-space \Rightarrow 'b::real-normed-algebra$
by (*fact continuous-mult*)

lemma (**in** *bounded-linear*) *isCont*: $isCont\ g\ a \implies isCont\ (\lambda x. f\ (g\ x))\ a$
by (*fact continuous*)

lemma (**in** *bounded-bilinear*) *isCont*: $isCont\ f\ a \implies isCont\ g\ a \implies isCont\ (\lambda x. f\ x ** g\ x)\ a$
by (*fact continuous*)

lemmas *isCont-scaleR* [*simp*] =
bounded-bilinear.isCont [OF bounded-bilinear-scaleR]

lemmas *isCont-of-real* [*simp*] =
bounded-linear.isCont [OF bounded-linear-of-real]

lemma *isCont-power* [*simp*]: $isCont\ f\ a \implies isCont\ (\lambda x. f\ x \wedge n)\ a$
for $f :: 'a::t2-space \Rightarrow 'b::\{power,real-normed-algebra\}$
by (*fact continuous-power*)

lemma *isCont-sum* [*simp*]: $\forall i \in A. isCont\ (f\ i)\ a \implies isCont\ (\lambda x. \sum_{i \in A} f\ i\ x)\ a$
for $f :: 'a \Rightarrow 'b::t2-space \Rightarrow 'c::topological-comm-monoid-add$
by (*auto intro: continuous-sum*)

109.12 Uniform Continuity

lemma *uniformly-continuous-on-def*:

fixes $f :: 'a::metric-space \Rightarrow 'b::metric-space$

shows *uniformly-continuous-on* $s\ f \longleftrightarrow$

$(\forall e > 0. \exists d > 0. \forall x \in s. \forall x' \in s. dist\ x'\ x < d \longrightarrow dist\ (f\ x')\ (f\ x) < e)$

unfolding *uniformly-continuous-on-uniformity*

uniformity-dist filterlim-INF filterlim-principal eventually-inf-principal

by (*force simp: Ball-def uniformity-dist[symmetric] eventually-uniformity-metric*)

abbreviation *isUCont* :: $['a::metric-space \Rightarrow 'b::metric-space] \Rightarrow bool$

where $isUCont\ f \equiv uniformly-continuous-on\ UNIV\ f$

lemma *isUCont-def*: $isUCont\ f \iff (\forall r>0. \exists s>0. \forall x\ y. dist\ x\ y < s \implies dist\ (f\ x)\ (f\ y) < r)$
by (*auto simp: uniformly-continuous-on-def dist-commute*)

lemma *isUCont-isCont*: $isUCont\ f \implies isCont\ f\ x$
by (*drule uniformly-continuous-imp-continuous*) (*simp add: continuous-on-eq-continuous-at*)

lemma *uniformly-continuous-on-Cauchy*:
fixes $f :: 'a::metric-space \Rightarrow 'b::metric-space$
assumes *uniformly-continuous-on S f Cauchy X* $\bigwedge n. X\ n \in S$
shows *Cauchy* $(\lambda n. f\ (X\ n))$
using *assms*
unfolding *uniformly-continuous-on-def* **by** (*meson Cauchy-def*)

lemma *isUCont-Cauchy*: $isUCont\ f \implies Cauchy\ X \implies Cauchy\ (\lambda n. f\ (X\ n))$
by (*rule uniformly-continuous-on-Cauchy[where S=UNIV and f=f]*) *simp-all*

lemma (*in bounded-linear*) *isUCont*: $isUCont\ f$
unfolding *isUCont-def dist-norm*
proof (*intro allI impI*)
fix $r :: real$
assume $r: 0 < r$
obtain K **where** $K: 0 < K$ **and** *norm-le*: $norm\ (f\ x) \leq norm\ x * K$ **for** x
using *pos-bounded* **by** *blast*
show $\exists s>0. \forall x\ y. norm\ (x - y) < s \implies norm\ (f\ x - f\ y) < r$
proof (*rule exI, safe*)
from $r\ K$ **show** $0 < r / K$ **by** *simp*
next
fix $x\ y :: 'a$
assume $xy: norm\ (x - y) < r / K$
have $norm\ (f\ x - f\ y) = norm\ (f\ (x - y))$ **by** (*simp only: diff*)
also have $\dots \leq norm\ (x - y) * K$ **by** (*rule norm-le*)
also from $K\ xy$ **have** $\dots < r$ **by** (*simp only: pos-less-divide-eq*)
finally show $norm\ (f\ x - f\ y) < r$.
qed
qed

lemma (*in bounded-linear*) *Cauchy*: $Cauchy\ X \implies Cauchy\ (\lambda n. f\ (X\ n))$
by (*rule isUCont [THEN isUCont-Cauchy]*)

lemma *LIM-less-bound*:
fixes $f :: real \Rightarrow real$
assumes *ev*: $b < x \forall x' \in \{ b < .. < x \}. 0 \leq f\ x'$ **and** *isCont f x*
shows $0 \leq f\ x$
proof (*rule tendsto-lowerbound*)
show $(f \longrightarrow f\ x)$ (*at-left x*)
using $\langle isCont\ f\ x \rangle$ **by** (*simp add: filterlim-at-split isCont-def*)
show *eventually* $(\lambda x. 0 \leq f\ x)$ (*at-left x*)

using *ev* by (*auto simp: eventually-at dist-real-def intro!: exI[of - x - b]*)
 qed *simp*

109.13 Nested Intervals and Bisection – Needed for Compactness

lemma *nested-sequence-unique*:

assumes $\forall n. f\ n \leq f\ (Suc\ n) \ \forall n. g\ (Suc\ n) \leq g\ n \ \forall n. f\ n \leq g\ n \ (\lambda n. f\ n - g\ n) \longrightarrow 0$

shows $\exists l::real. ((\forall n. f\ n \leq l) \wedge f \longrightarrow l) \wedge ((\forall n. l \leq g\ n) \wedge g \longrightarrow l)$

proof –

have *incseq* *f* **unfolding** *incseq-Suc-iff* **by** *fact*

have *decseq* *g* **unfolding** *decseq-Suc-iff* **by** *fact*

have $f\ 0 \leq g\ 0$ **for** *n*

proof –

from $\langle decseq\ g \rangle$ have $g\ n \leq g\ 0$

by (*rule decseqD*) *simp*

with $\langle \forall n. f\ n \leq g\ n \rangle$ [THEN *spec*, of *n*] **show** *?thesis*

by *auto*

qed

then obtain *u* where $f \longrightarrow u \ \forall i. f\ i \leq u$

using *incseq-convergent*[OF $\langle incseq\ f \rangle$] **by** *auto*

moreover have $f\ 0 \leq g\ n$ **for** *n*

proof –

from $\langle incseq\ f \rangle$ have $f\ 0 \leq f\ n$ **by** (*rule incseqD*) *simp*

with $\langle \forall n. f\ n \leq g\ n \rangle$ [THEN *spec*, of *n*] **show** *?thesis*

by *simp*

qed

then obtain *l* where $g \longrightarrow l \ \forall i. l \leq g\ i$

using *decseq-convergent*[OF $\langle decseq\ g \rangle$] **by** *auto*

moreover note *LIMSEQ-unique*[OF *assms*(4) *tendsto-diff*[OF $\langle f \longrightarrow u \rangle \langle g \longrightarrow l \rangle$]]

ultimately **show** *?thesis* **by** *auto*

qed

lemma *Bolzano*[*consumes 1, case-names trans local*]:

fixes *P* :: *real* \Rightarrow *real* \Rightarrow *bool*

assumes [*arith*]: $a \leq b$

and *trans*: $\bigwedge a\ b\ c. P\ a\ b \Longrightarrow P\ b\ c \Longrightarrow a \leq b \Longrightarrow b \leq c \Longrightarrow P\ a\ c$

and *local*: $\bigwedge x. a \leq x \Longrightarrow x \leq b \Longrightarrow \exists d > 0. \forall a\ b. a \leq x \wedge x \leq b \wedge b - a < d \longrightarrow P\ a\ b$

shows $P\ a\ b$

proof –

define *bisect* **where** *bisect* $\equiv \lambda(x,y). \text{if } P\ x\ ((x+y) / 2) \text{ then } ((x+y)/2, y) \text{ else } (x, (x+y)/2)$

define *l* *u* **where** $l\ n \equiv \text{fst } ((bisect \sim n)(a,b))$ and $u\ n \equiv \text{snd } ((bisect \sim n)(a,b))$

for *n*

have [*simp*]: $l\ 0 = a \ \bigwedge n. l\ (Suc\ n) = (\text{if } P\ (l\ n)\ ((l\ n + u\ n) / 2) \text{ then } (l\ n + u\ n) / 2 \text{ else } l\ n)$

and $u[simp]: u\ 0 = b \wedge n. u\ (Suc\ n) = (if\ P\ (l\ n)\ ((l\ n + u\ n) / 2)\ then\ u\ n\ else\ (l\ n + u\ n) / 2)$

by (*simp-all add: l-def u-def bisect-def split: prod.split*)

have $[simp]: l\ n \leq u\ n$ **for** n **by** (*induct n auto*)

have $\exists x. ((\forall n. l\ n \leq x) \wedge l \longrightarrow x) \wedge ((\forall n. x \leq u\ n) \wedge u \longrightarrow x)$

proof (*safe intro!: nested-sequence-unique*)

show $l\ n \leq l\ (Suc\ n) \wedge u\ (Suc\ n) \leq u\ n$ **for** n

by (*induct n auto*)

next

have $l\ n - u\ n = (a - b) / 2^n$ **for** n

by (*induct n (auto simp: field-simps)*)

then show $(\lambda n. l\ n - u\ n) \longrightarrow 0$

by (*simp add: LIMSEQ-divide-realpow-zero*)

qed fact

then obtain x **where** $x: \bigwedge n. l\ n \leq x \wedge \bigwedge n. x \leq u\ n$ **and** $l \longrightarrow x \wedge u \longrightarrow x$

by *auto*

obtain d **where** $0 < d$ **and** $d: a \leq x \implies x \leq b \implies b - a < d \implies P\ a\ b$ **for** $a\ b$

using $\langle l\ 0 \leq x \rangle \langle x \leq u\ 0 \rangle$ *local[of x]* **by** *auto*

show $P\ a\ b$

proof (*rule ccontr*)

assume $\neg P\ a\ b$

have $\neg P\ (l\ n)\ (u\ n)$ **for** n

proof (*induct n*)

case 0

then show *?case*

by (*simp add: $\langle \neg P\ a\ b \rangle$*)

next

case $(Suc\ n)$

with *trans[of l n (l n + u n) / 2 u n]* **show** *?case*

by *auto*

qed

moreover

{

have *eventually* $(\lambda n. x - d / 2 < l\ n)$ *sequentially*

using $\langle 0 < d \rangle \langle l \longrightarrow x \rangle$ **by** (*intro order-tendstoD[of - x]*) *auto*

moreover have *eventually* $(\lambda n. u\ n < x + d / 2)$ *sequentially*

using $\langle 0 < d \rangle \langle u \longrightarrow x \rangle$ **by** (*intro order-tendstoD[of - x]*) *auto*

ultimately have *eventually* $(\lambda n. P\ (l\ n)\ (u\ n))$ *sequentially*

proof *eventually-elim*

case (*elim n*)

from *add-strict-mono[OF this]* **have** $u\ n - l\ n < d$ **by** *simp*

with x **show** $P\ (l\ n)\ (u\ n)$ **by** (*rule d*)

qed

}

ultimately show *False* **by** *simp*

qed
qed

lemma *compact-Icc*[simp, intro]: compact {a .. b::real}
proof (cases a ≤ b, rule compactI)
 fix C
 assume C: a ≤ b ∀ t ∈ C. open t {a..b} ⊆ ∪ C
 define T where T = {a .. b}
 from C(1,3) show ∃ C' ⊆ C. finite C' ∧ {a..b} ⊆ ∪ C'
 proof (induct rule: Bolzano)
 case (trans a b c)
 then have *: {a..c} = {a..b} ∪ {b..c}
 by auto
 with trans obtain C1 C2
 where C1 ⊆ C finite C1 {a..b} ⊆ ∪ C1 C2 ⊆ C finite C2 {b..c} ⊆ ∪ C2
 by auto
 with trans show ?case
 unfolding * by (intro exI[of - C1 ∪ C2]) auto
 next
 case (local x)
 with C have x ∈ ∪ C by auto
 with C(2) obtain c where x ∈ c open c c ∈ C
 by auto
 then obtain e where 0 < e {x - e <..< x + e} ⊆ c
 by (auto simp: open-dist dist-real-def subset-eq Ball-def abs-less-iff)
 with ⟨c ∈ C⟩ show ?case
 by (safe intro!: exI[of - e/2] exI[of - {c}]) auto
 qed
 qed simp

lemma *continuous-image-closed-interval*:
 fixes a b and f :: real ⇒ real
 defines S ≡ {a..b}
 assumes a ≤ b and f: continuous-on S f
 shows ∃ c d. f'S = {c..d} ∧ c ≤ d
 proof -
 have S: compact S S ≠ {}
 using ⟨a ≤ b⟩ by (auto simp: S-def)
 obtain c where c ∈ S ∀ d ∈ S. f d ≤ f c
 using continuous-attains-sup[OF S f] by auto
 moreover obtain d where d ∈ S ∀ c ∈ S. f d ≤ f c
 using continuous-attains-inf[OF S f] by auto
 moreover have connected (f'S)
 using connected-continuous-image[OF f] connected-Icc by (auto simp: S-def)
 ultimately have f'S = {f d .. f c} ∧ f d ≤ f c
 by (auto simp: connected-iff-interval)
 then show ?thesis
 by auto

qed

lemma *open-Collect-positive*:

fixes $f :: 'a::\text{topological-space} \Rightarrow \text{real}$

assumes f : *continuous-on s f*

shows $\exists A. \text{open } A \wedge A \cap s = \{x \in s. 0 < f x\}$

using *continuous-on-open-invariant*[*THEN iffD1, OF f, rule-format, of {0 <..}*]

by (*auto simp: Int-def field-simps*)

lemma *open-Collect-less-Int*:

fixes $f g :: 'a::\text{topological-space} \Rightarrow \text{real}$

assumes f : *continuous-on s f*

and g : *continuous-on s g*

shows $\exists A. \text{open } A \wedge A \cap s = \{x \in s. f x < g x\}$

using *open-Collect-positive*[*OF continuous-on-diff* [*OF g f*]] **by** (*simp add: field-simps*)

109.14 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

lemma *isCont-eq-Ub*:

fixes $f :: \text{real} \Rightarrow 'a::\text{linorder-topology}$

shows $a \leq b \implies \forall x::\text{real}. a \leq x \wedge x \leq b \longrightarrow \text{isCont } f x \implies$

$\exists M. (\forall x. a \leq x \wedge x \leq b \longrightarrow f x \leq M) \wedge (\exists x. a \leq x \wedge x \leq b \wedge f x = M)$

using *continuous-attains-sup*[*of {a..b} f*]

by (*auto simp: continuous-at-imp-continuous-on Ball-def Bex-def*)

lemma *isCont-eq-Lb*:

fixes $f :: \text{real} \Rightarrow 'a::\text{linorder-topology}$

shows $a \leq b \implies \forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } f x \implies$

$\exists M. (\forall x. a \leq x \wedge x \leq b \longrightarrow M \leq f x) \wedge (\exists x. a \leq x \wedge x \leq b \wedge f x = M)$

using *continuous-attains-inf*[*of {a..b} f*]

by (*auto simp: continuous-at-imp-continuous-on Ball-def Bex-def*)

lemma *isCont-bounded*:

fixes $f :: \text{real} \Rightarrow 'a::\text{linorder-topology}$

shows $a \leq b \implies \forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } f x \implies \exists M. \forall x. a \leq x \wedge x \leq b \longrightarrow f x \leq M$

using *isCont-eq-Ub*[*of a b f*] **by** *auto*

lemma *isCont-has-Ub*:

fixes $f :: \text{real} \Rightarrow 'a::\text{linorder-topology}$

shows $a \leq b \implies \forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } f x \implies$

$\exists M. (\forall x. a \leq x \wedge x \leq b \longrightarrow f x \leq M) \wedge (\forall N. N < M \longrightarrow (\exists x. a \leq x \wedge x \leq b \wedge N < f x))$

using *isCont-eq-Ub*[*of a b f*] **by** *auto*

lemma *isCont-Lb-Ub*:

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $a \leq b \forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } f x$

```

shows  $\exists L M. (\forall x. a \leq x \wedge x \leq b \longrightarrow L \leq f x \wedge f x \leq M) \wedge$ 
   $(\forall y. L \leq y \wedge y \leq M \longrightarrow (\exists x. a \leq x \wedge x \leq b \wedge (f x = y)))$ 
proof –
  obtain  $M$  where  $M: a \leq M M \leq b \forall x. a \leq x \wedge x \leq b \longrightarrow f x \leq f M$ 
    using isCont-eq-Ub[OF assms] by auto
  obtain  $L$  where  $L: a \leq L L \leq b \forall x. a \leq x \wedge x \leq b \longrightarrow f L \leq f x$ 
    using isCont-eq-Lb[OF assms] by auto
  have  $(\forall x. a \leq x \wedge x \leq b \longrightarrow f L \leq f x \wedge f x \leq f M)$ 
    using  $M L$  by simp
  moreover
  have  $(\forall y. f L \leq y \wedge y \leq f M \longrightarrow (\exists x \geq a. x \leq b \wedge f x = y))$ 
  proof (cases  $L \leq M$ )
    case True then show ?thesis
      using IVT[of f L - M] M L assms by (metis order.trans)
    next
      case False then show ?thesis
        using IVT2[of f L - M]
        by (metis L(2) M(1) assms(2) le-cases order.trans)
  qed
  ultimately show ?thesis
    by blast
qed

```

Continuity of inverse function.

lemma *isCont-inverse-function*:

```

fixes  $f g :: real \Rightarrow real$ 
assumes  $d: 0 < d$ 
  and inj:  $\bigwedge z. |z-x| \leq d \implies g (f z) = z$ 
  and cont:  $\bigwedge z. |z-x| \leq d \implies isCont f z$ 
shows isCont  $g (f x)$ 

```

proof –

```

let  $?A = f (x - d)$ 
let  $?B = f (x + d)$ 
let  $?D = \{x - d..x + d\}$ 

```

have f : *continuous-on* $?D$ f

using *cont* **by** (*intro continuous-at-imp-continuous-on ballI*) *auto*

then have g : *continuous-on* $(f^{-1} ?D)$ g

using *inj* **by** (*intro continuous-on-inv*) *auto*

from $d f$ **have** $\{min ?A ?B <..< max ?A ?B\} \subseteq f^{-1} ?D$

by (*intro connected-contains-Ioo connected-continuous-image*) (*auto split: split-min split-max*)

with g **have** *continuous-on* $\{min ?A ?B <..< max ?A ?B\} g$

by (*rule continuous-on-subset*)

moreover

have $(?A < f x \wedge f x < ?B) \vee (?B < f x \wedge f x < ?A)$

using d *inj* **by** (*intro continuous-inj-imp-mono[OF - - f] inj-on-imageI2[of g, OF inj-onI]*) *auto*

then have $f x \in \{\min ?A ?B <..
by *auto*
ultimately show *?thesis*
by (*simp add: continuous-on-eq-continuous-at*)
qed$

lemma *isCont-inverse-function2*:

fixes $f g :: \text{real} \Rightarrow \text{real}$

shows

$\llbracket a < x; x < b; \llbracket a \leq z; z \leq b \rrbracket \Longrightarrow g (f z) = z;$

$\llbracket a \leq z; z \leq b \rrbracket \Longrightarrow g (f z) = z;$

$\llbracket a \leq z; z \leq b \rrbracket \Longrightarrow \text{isCont } f z \rrbracket \Longrightarrow \text{isCont } g (f x)$

apply (*rule isCont-inverse-function* [**where** $f=f$ **and** $d=\min (x - a) (b - x)$])

apply (*simp-all add: abs-le-iff*)

done

Bartle/Sherbert: Introduction to Real Analysis, Theorem 4.2.9, p. 110.

lemma *LIM-fun-gt-zero*: $f -c \rightarrow l \Longrightarrow 0 < l \Longrightarrow \exists r. 0 < r \wedge (\forall x. x \neq c \wedge |c - x| < r \longrightarrow 0 < f x)$

for $f :: \text{real} \Rightarrow \text{real}$

by (*force simp: dest: LIM-D*)

lemma *LIM-fun-less-zero*: $f -c \rightarrow l \Longrightarrow l < 0 \Longrightarrow \exists r. 0 < r \wedge (\forall x. x \neq c \wedge |c - x| < r \longrightarrow f x < 0)$

for $f :: \text{real} \Rightarrow \text{real}$

by (*drule LIM-D* [**where** $r=-l$]) *force+*

lemma *LIM-fun-not-zero*: $f -c \rightarrow l \Longrightarrow l \neq 0 \Longrightarrow \exists r. 0 < r \wedge (\forall x. x \neq c \wedge |c - x| < r \longrightarrow f x \neq 0)$

for $f :: \text{real} \Rightarrow \text{real}$

using *LIM-fun-gt-zero*[*of f l c*] *LIM-fun-less-zero*[*of f l c*] **by** (*auto simp: neq-iff*)

lemma *Lim-topological*:

$(f \longrightarrow l) \text{ net} \longleftrightarrow$

trivial-limit net $\vee (\forall S. \text{open } S \longrightarrow l \in S \longrightarrow \text{eventually } (\lambda x. f x \in S) \text{ net})$

unfolding *tendsto-def trivial-limit-eq* **by** *auto*

lemma *eventually-within-Un*:

eventually P (*at* x *within* $(s \cup t)$) \longleftrightarrow

eventually P (*at* x *within* s) \wedge *eventually* P (*at* x *within* t)

unfolding *eventually-at-filter*

by (*auto elim!: eventually-rev-mp*)

lemma *Lim-within-Un*:

$(f \longrightarrow l) \text{ (at } x \text{ within } (s \cup t)) \longleftrightarrow$

$(f \longrightarrow l) \text{ (at } x \text{ within } s) \wedge (f \longrightarrow l) \text{ (at } x \text{ within } t)$

unfolding *tendsto-def*

by (*auto simp: eventually-within-Un*)

end

theory *Inequalities*

imports *Real-Vector-Spaces*

begin

lemma *Chebyshev-sum-upper*:

fixes $a\ b :: \text{nat} \Rightarrow 'a :: \text{linordered-idom}$

assumes $\bigwedge i\ j. i \leq j \implies j < n \implies a\ i \leq a\ j$

assumes $\bigwedge i\ j. i \leq j \implies j < n \implies b\ i \geq b\ j$

shows $\text{of-nat } n * (\sum_{k=0..<n}. a\ k * b\ k) \leq (\sum_{k=0..<n}. a\ k) * (\sum_{k=0..<n}. b\ k)$

proof –

let $?S = (\sum_{j=0..<n}. (\sum_{k=0..<n}. (a\ j - a\ k) * (b\ j - b\ k)))$

have $2 * (\text{of-nat } n * (\sum_{j=0..<n}. (a\ j * b\ j)) - (\sum_{j=0..<n}. b\ j) * (\sum_{k=0..<n}. a\ k)) = ?S$

by (*simp only: one-add-one[symmetric] algebra-simps*)

(*simp add: algebra-simps sum-subtractf sum.distrib sum.swap[of $\lambda i\ j. a\ i * b\ j$] sum-distrib-left*)

also

{ fix $i\ j :: \text{nat}$ **assume** $i < n\ j < n$

hence $a\ i - a\ j \leq 0 \wedge b\ i - b\ j \geq 0 \vee a\ i - a\ j \geq 0 \wedge b\ i - b\ j \leq 0$

using *assms* **by** (*cases $i \leq j$*) (*auto simp: algebra-simps*)

} then have $?S \leq 0$

by (*auto intro!: sum-nonpos simp: mult-le-0-iff*)

finally show $?thesis$ **by** (*simp add: algebra-simps*)

qed

lemma *Chebyshev-sum-upper-nat*:

fixes $a\ b :: \text{nat} \Rightarrow \text{nat}$

shows $(\bigwedge i\ j. \llbracket i \leq j; j < n \rrbracket \implies a\ i \leq a\ j) \implies$

$(\bigwedge i\ j. \llbracket i \leq j; j < n \rrbracket \implies b\ i \geq b\ j) \implies$

$n * (\sum_{i=0..<n}. a\ i * b\ i) \leq (\sum_{i=0..<n}. a\ i) * (\sum_{i=0..<n}. b\ i)$

using *Chebyshev-sum-upper* [**where** $'a = \text{real}$, *of $n\ a\ b$*]

by (*simp del: of-nat-mult of-nat-sum add: of-nat-mult[symmetric] of-nat-sum[symmetric]*)

end

110 Infinite Series

theory *Series*

imports *Limits Inequalities*

begin

110.1 Definition of infinite summability

definition *sums* :: $(\text{nat} \Rightarrow 'a :: \{\text{topological-space, comm-monoid-add}\}) \Rightarrow 'a \Rightarrow \text{bool}$
(*infixr sums 80*)

where $f \text{ sums } s \iff (\lambda n. \sum i < n. f i) \longrightarrow s$

definition $\text{summable} :: (\text{nat} \Rightarrow 'a :: \{\text{topological-space, comm-monoid-add}\}) \Rightarrow \text{bool}$
where $\text{summable } f \iff (\exists s. f \text{ sums } s)$

definition $\text{suminf} :: (\text{nat} \Rightarrow 'a :: \{\text{topological-space, comm-monoid-add}\}) \Rightarrow 'a$
(binder $\sum 10)$
where $\text{suminf } f = (\text{THE } s. f \text{ sums } s)$

Variants of the definition

lemma $\text{sums-def}' : f \text{ sums } s \iff (\lambda n. \sum i = 0..n. f i) \longrightarrow s$
unfolding sums-def
apply ($\text{subst filterlim-sequentially-Suc [symmetric]}$)
apply ($\text{simp only: lessThan-Suc-atMost atLeast0AtMost}$)
done

lemma $\text{sums-def-le} : f \text{ sums } s \iff (\lambda n. \sum i \leq n. f i) \longrightarrow s$
by ($\text{simp add: sums-def}' \text{ atMost-atLeast0}$)

lemma $\text{bounded-imp-summable}$:

fixes $a :: \text{nat} \Rightarrow 'a :: \{\text{conditionally-complete-linorder, linorder-topology, linordered-comm-semiring-strict}\}$
assumes $0 : \bigwedge n. a n \geq 0$ **and** $\text{bounded} : \bigwedge n. (\sum k \leq n. a k) \leq B$
shows $\text{summable } a$

proof –

have $\text{bdd-above } (\text{range } (\lambda n. \sum k \leq n. a k))$
by ($\text{meson bdd-aboveI2 bounded}$)
moreover have $\text{incseq } (\lambda n. \sum k \leq n. a k)$
by ($\text{simp add: mono-def 0 sum-mono2}$)
ultimately obtain s **where** $(\lambda n. \sum k \leq n. a k) \longrightarrow s$
using LIMSEQ-incseq-SUP **by** blast
then show $?thesis$
by ($\text{auto simp: sums-def-le summable-def}$)

qed

110.2 Infinite summability on topological monoids

lemma $\text{sums-subst}[trans] : f = g \implies g \text{ sums } z \implies f \text{ sums } z$
by simp

lemma $\text{sums-cong} : (\bigwedge n. f n = g n) \implies f \text{ sums } c \iff g \text{ sums } c$
by presburger

lemma $\text{sums-summable} : f \text{ sums } l \implies \text{summable } f$
by ($\text{simp add: sums-def summable-def, blast}$)

lemma $\text{summable-iff-convergent} : \text{summable } f \iff \text{convergent } (\lambda n. \sum i < n. f i)$
by ($\text{simp add: summable-def sums-def convergent-def}$)

lemma $\text{summable-iff-convergent}' : \text{summable } f \iff \text{convergent } (\lambda n. \text{sum } f \{..n\})$

by (simp add: convergent-def summable-def sums-def-le)

lemma *suminf-eq-lim*: $\text{suminf } f = \lim (\lambda n. \sum_{i < n}. f i)$
 by (simp add: suminf-def sums-def lim-def)

lemma *sums-zero*[simp, intro]: $(\lambda n. 0)$ sums 0
 unfolding sums-def by simp

lemma *summable-zero*[simp, intro]: summable $(\lambda n. 0)$
 by (rule sums-zero [THEN sums-summable])

lemma *sums-group*: f sums $s \implies 0 < k \implies (\lambda n. \text{sum } f \{n * k ..< n * k + k\})$
sums s
 apply (simp only: sums-def sum.nat-group tendsto-def eventually-sequentially)
 apply (erule all-forward imp-forward exE| assumption)+
 by (metis le-square mult.commute mult.left-neutral mult-le-cancel2 mult-le-mono)

lemma *suminf-cong*: $(\bigwedge n. f n = g n) \implies \text{suminf } f = \text{suminf } g$
 by presburger

lemma *summable-cong*:

fixes $f g :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector}$
 assumes eventually $(\lambda x. f x = g x)$ sequentially
 shows summable $f = \text{summable } g$

proof –

from assms obtain N where $N: \forall n \geq N. f n = g n$

by (auto simp: eventually-at-top-linorder)

define C where $C = (\sum_{k < N}. f k - g k)$

from eventually-ge-at-top[of N]

have eventually $(\lambda n. \text{sum } f \{..<n\} = C + \text{sum } g \{..<n\})$ sequentially

proof eventually-elim

case (elim n)

then have $\{..<n\} = \{..<N\} \cup \{N..<n\}$

by auto

also have $\text{sum } f \dots = \text{sum } f \{..<N\} + \text{sum } f \{N..<n\}$

by (intro sum.union-disjoint) auto

also from N have $\text{sum } f \{N..<n\} = \text{sum } g \{N..<n\}$

by (intro sum.cong) simp-all

also have $\text{sum } f \{..<N\} + \text{sum } g \{N..<n\} = C + (\text{sum } g \{..<N\} + \text{sum } g \{N..<n\})$

unfolding C -def by (simp add: algebra-simps sum-subtractf)

also have $\text{sum } g \{..<N\} + \text{sum } g \{N..<n\} = \text{sum } g (\{..<N\} \cup \{N..<n\})$

by (intro sum.union-disjoint [symmetric]) auto

also from elim have $\{..<N\} \cup \{N..<n\} = \{..<n\}$

by auto

finally show $\text{sum } f \{..<n\} = C + \text{sum } g \{..<n\}$.

qed

from convergent-cong[OF this] show ?thesis

by (simp add: summable-iff-convergent convergent-add-const-iff)

qed

lemma *sums-finite*:

assumes [*simp*]: *finite N*
and *f*: $\bigwedge n. n \notin N \implies f\ n = 0$
shows *f sums* ($\sum_{n \in N}. f\ n$)

proof –

have *eq*: *sum f* {..*n* + *Suc (Max N)*} = *sum f N* **for** *n*
by (*rule sum.mono-neutral-right*) (*auto simp: add-increasing less-Suc-eq-le f*)
show ?*thesis*
unfolding *sums-def*
by (*rule LIMSEQ-offset[of - Suc (Max N)]*)
(*simp add: eq atLeast0LessThan del: add-Suc-right*)

qed

corollary *sums-0*: $(\bigwedge n. f\ n = 0) \implies (f\ \text{sums}\ 0)$

by (*metis (no-types) finite.emptyI sum.empty sums-finite*)

lemma *summable-finite*: *finite N* $\implies (\bigwedge n. n \notin N \implies f\ n = 0) \implies \text{summable } f$

by (*rule sums-summable*) (*rule sums-finite*)

lemma *sums-If-finite-set*: *finite A* $\implies (\lambda r. \text{if } r \in A \text{ then } f\ r \text{ else } 0) \text{ sums } (\sum_{r \in A}. f\ r)$

using *sums-finite*[*of A* ($\lambda r. \text{if } r \in A \text{ then } f\ r \text{ else } 0$)] **by** *simp*

lemma *summable-If-finite-set*[*simp, intro*]: *finite A* $\implies \text{summable } (\lambda r. \text{if } r \in A \text{ then } f\ r \text{ else } 0)$

by (*rule sums-summable*) (*rule sums-If-finite-set*)

lemma *sums-If-finite*: *finite* {*r. P r*} $\implies (\lambda r. \text{if } P\ r \text{ then } f\ r \text{ else } 0) \text{ sums } (\sum_{r \mid P\ r}. f\ r)$

using *sums-If-finite-set*[*of* {*r. P r*}] **by** *simp*

lemma *summable-If-finite*[*simp, intro*]: *finite* {*r. P r*} $\implies \text{summable } (\lambda r. \text{if } P\ r \text{ then } f\ r \text{ else } 0)$

by (*rule sums-summable*) (*rule sums-If-finite*)

lemma *sums-single*: $(\lambda r. \text{if } r = i \text{ then } f\ r \text{ else } 0) \text{ sums } f\ i$

using *sums-If-finite*[*of* $\lambda r. r = i$] **by** *simp*

lemma *summable-single*[*simp, intro*]: *summable* $(\lambda r. \text{if } r = i \text{ then } f\ r \text{ else } 0)$

by (*rule sums-summable*) (*rule sums-single*)

context

fixes *f* :: *nat* \Rightarrow 'a::*{t2-space, comm-monoid-add}*

begin

lemma *summable-sums*[*intro*]: *summable f* $\implies f \text{ sums } (\text{suminf } f)$

by (*simp add: summable-def sums-def suminf-def*)

(metis convergent-LIMSEQ-iff convergent-def lim-def)

lemma *summable-LIMSEQ*: $\text{summable } f \implies (\lambda n. \sum i < n. f i) \longrightarrow \text{suminf } f$
by (rule *summable-sums* [unfolding *sums-def*])

lemma *summable-LIMSEQ'*: $\text{summable } f \implies (\lambda n. \sum i \leq n. f i) \longrightarrow \text{suminf } f$
using *sums-def-le* **by** *blast*

lemma *sums-unique*: $f \text{ sums } s \implies s = \text{suminf } f$
by (metis *limI suminf-eq-lim sums-def*)

lemma *sums-iff*: $f \text{ sums } x \longleftrightarrow \text{summable } f \wedge \text{suminf } f = x$
by (metis *summable-sums sums-summable sums-unique*)

lemma *summable-sums-iff*: $\text{summable } f \longleftrightarrow f \text{ sums } \text{suminf } f$
by (auto simp: *sums-iff summable-sums*)

lemma *sums-unique2*: $f \text{ sums } a \implies f \text{ sums } b \implies a = b$
for $a b :: 'a$
by (simp add: *sums-iff*)

lemma *sums-Uniq*: $\exists \leq_1 a. f \text{ sums } a$
for $a b :: 'a$
by (simp add: *sums-unique2 Uniq-def*)

lemma *suminf-finite*:
assumes N : *finite* N
and f : $\bigwedge n. n \notin N \implies f n = 0$
shows $\text{suminf } f = (\sum n \in N. f n)$
using *sums-finite*[OF *assms*, THEN *sums-unique*] **by** *simp*

end

lemma *suminf-zero*[*simp*]: $\text{suminf } (\lambda n. 0 :: 'a :: \{t2\text{-space, comm-monoid-add}\}) = 0$
by (rule *sums-zero* [THEN *sums-unique, symmetric*])

110.3 Infinite summability on ordered, topological monoids

lemma *sums-le*: $(\bigwedge n. f n \leq g n) \implies f \text{ sums } s \implies g \text{ sums } t \implies s \leq t$
for $f g :: \text{nat} \Rightarrow 'a :: \{\text{ordered-comm-monoid-add, linorder-topology}\}$
by (rule *LIMSEQ-le*) (auto intro: *sum-mono simp: sums-def*)

context

fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{ordered-comm-monoid-add, linorder-topology}\}$
begin

lemma *suminf-le*: $(\bigwedge n. f n \leq g n) \implies \text{summable } f \implies \text{summable } g \implies \text{suminf } f \leq \text{suminf } g$
using *sums-le* **by** *blast*

lemma *sum-le-suminf*:

shows $\text{summable } f \implies \text{finite } I \implies (\bigwedge n. n \in - I \implies 0 \leq f n) \implies \text{sum } f I \leq \text{suminf } f$

by (*rule sums-le[OF - sums-If-finite-set summable-sums]*) *auto*

lemma *suminf-nonneg*: $\text{summable } f \implies (\bigwedge n. 0 \leq f n) \implies 0 \leq \text{suminf } f$

using *sum-le-suminf* **by** *force*

lemma *suminf-le-const*: $\text{summable } f \implies (\bigwedge n. \text{sum } f \{..<n\} \leq x) \implies \text{suminf } f \leq x$

by (*metis LIMSEQ-le-const2 summable-LIMSEQ*)

lemma *suminf-eq-zero-iff*:

assumes *summable f* **and** *pos*: $\bigwedge n. 0 \leq f n$

shows $\text{suminf } f = 0 \iff (\forall n. f n = 0)$

proof

assume *L*: $\text{suminf } f = 0$

then have *f*: $(\lambda n. \sum i < n. f i) \longrightarrow 0$

using *summable-LIMSEQ[of f]* **assms** **by** *simp*

then have $\bigwedge i. (\sum n \in \{i\}. f n) \leq 0$

by (*metis L <summable f> order-refl pos sum.infinite sum-le-suminf*)

with *pos* **show** $\forall n. f n = 0$

by (*simp add: order.antisym*)

qed (*metis suminf-zero fun-eq-iff*)

lemma *suminf-pos-iff*: $\text{summable } f \implies (\bigwedge n. 0 \leq f n) \implies 0 < \text{suminf } f \iff (\exists i. 0 < f i)$

using *sum-le-suminf[of {}]* *suminf-eq-zero-iff* **by** (*simp add: less-le*)

lemma *suminf-pos2*:

assumes *summable f* $\bigwedge n. 0 \leq f n$ $0 < f i$

shows $0 < \text{suminf } f$

proof –

have $0 < (\sum n < \text{Suc } i. f n)$

using *assms* **by** (*intro sum-pos2[where i=i]*) *auto*

also have $\dots \leq \text{suminf } f$

using *assms* **by** (*intro sum-le-suminf*) *auto*

finally show *?thesis* .

qed

lemma *suminf-pos*: $\text{summable } f \implies (\bigwedge n. 0 < f n) \implies 0 < \text{suminf } f$

by (*intro suminf-pos2[where i=0]*) (*auto intro: less-imp-le*)

end

context

fixes *f* :: *nat* \Rightarrow '*a*::{*ordered-cancel-comm-monoid-add,linorder-topology*}

begin

lemma *sum-less-suminf2*:
 $summable\ f \implies (\bigwedge m. m \geq n \implies 0 \leq f\ m) \implies n \leq i \implies 0 < f\ i \implies sum\ f\ \{..<n\} < suminf\ f$
using *sum-le-suminf*[of $f\ \{..< Suc\ i\}$]
and *add-strict-increasing*[of $f\ i\ sum\ f\ \{..<n\}\ sum\ f\ \{..<i\}$]
and *sum-mono2*[of $\{..<i\}\ \{..<n\}\ f$]
by (*auto simp: less-imp-le ac-simps*)

lemma *sum-less-suminf*: $summable\ f \implies (\bigwedge m. m \geq n \implies 0 < f\ m) \implies sum\ f\ \{..<n\} < suminf\ f$
using *sum-less-suminf2*[of $n\ n$] **by** (*simp add: less-imp-le*)

end

lemma *summableI-nonneg-bounded*:
fixes $f :: nat \Rightarrow 'a::\{ordered-comm-monoid-add,linorder-topology,conditionally-complete-linorder\}$
assumes *pos*[*simp*]: $\bigwedge n. 0 \leq f\ n$
and *le*: $\bigwedge n. (\sum i < n. f\ i) \leq x$
shows *summable* f
unfolding *summable-def sums-def* [*abs-def*]
proof (*rule exI LIMSEQ-incseq-SUP*) +
show *bdd-above* (*range* ($\lambda n. sum\ f\ \{..<n\}$))
using *le* **by** (*auto simp: bdd-above-def*)
show *incseq* ($\lambda n. sum\ f\ \{..<n\}$)
by (*auto simp: mono-def intro!: sum-mono2*)
qed

lemma *summableI*[*intro, simp*]: *summable* f
for $f :: nat \Rightarrow 'a::\{canonically-ordered-monoid-add,linorder-topology,complete-linorder\}$
by (*intro summableI-nonneg-bounded*[**where** $x=top$] *zero-le top-greatest*)

lemma *suminf-eq-SUP-real*:
assumes $X: summable\ X \bigwedge i. 0 \leq X\ i$ **shows** $suminf\ X = (SUP\ i. \sum n < i. X\ n::real)$
by (*intro LIMSEQ-unique*[*OF summable-LIMSEQ*] $X\ LIMSEQ-incseq-SUP$)
(*auto intro!: bdd-aboveI2*[**where** $M = \sum i. X\ i$] *sum-le-suminf X monoI sum-mono2*)

110.4 Infinite summability on topological monoids

context

fixes $f\ g :: nat \Rightarrow 'a::\{t2-space,topological-comm-monoid-add\}$
begin

lemma *sums-Suc*:

assumes ($\lambda n. f\ (Suc\ n)$) *sums* l
shows $f\ sums\ (l + f\ 0)$

proof –

have ($\lambda n. (\sum i < n. f\ (Suc\ i)) + f\ 0$) $\longrightarrow l + f\ 0$

using *assms* **by** (*auto intro!*: *tendsto-add simp: sums-def*)
moreover have $(\sum i < n. f (Suc i)) + f 0 = (\sum i < Suc n. f i)$ **for** *n*
unfolding *lessThan-Suc-eq-insert-0*
by (*simp add: ac-simps sum.atLeast1-atMost-eq image-Suc-lessThan*)
ultimately show *?thesis*
by (*auto simp: sums-def simp del: sum.lessThan-Suc intro: filterlim-sequentially-Suc[THEN iffD1]*)
qed

lemma *sums-add*: $f \text{ sums } a \implies g \text{ sums } b \implies (\lambda n. f n + g n) \text{ sums } (a + b)$
unfolding *sums-def* **by** (*simp add: sum.distrib tendsto-add*)

lemma *summable-add*: $\text{summable } f \implies \text{summable } g \implies \text{summable } (\lambda n. f n + g n)$
unfolding *summable-def* **by** (*auto intro: sums-add*)

lemma *suminf-add*: $\text{summable } f \implies \text{summable } g \implies \text{suminf } f + \text{suminf } g = (\sum n. f n + g n)$
by (*intro sums-unique sums-add summable-sums*)

end

context

fixes *f* :: 'i \Rightarrow nat \Rightarrow 'a::*{t2-space, topological-comm-monoid-add}*
and *I* :: 'i *set*

begin

lemma *sums-sum*: $(\bigwedge i. i \in I \implies (f i) \text{ sums } (x i)) \implies (\lambda n. \sum i \in I. f i n) \text{ sums } (\sum i \in I. x i)$
by (*induct I rule: infinite-finite-induct*) (*auto intro!: sums-add*)

lemma *suminf-sum*: $(\bigwedge i. i \in I \implies \text{summable } (f i)) \implies (\sum n. \sum i \in I. f i n) = (\sum i \in I. \sum n. f i n)$
using *sums-unique[OF sums-sum, OF summable-sums]* **by** *simp*

lemma *summable-sum*: $(\bigwedge i. i \in I \implies \text{summable } (f i)) \implies \text{summable } (\lambda n. \sum i \in I. f i n)$
using *sums-summable[OF sums-sum[OF summable-sums]]* .

end

lemma *sums-If-finite-set'*:

fixes *f g* :: nat \Rightarrow 'a::*{t2-space, topological-ab-group-add}*
assumes *g sums S* **and** *finite A* **and** $S' = S + (\sum n \in A. f n - g n)$
shows $(\lambda n. \text{if } n \in A \text{ then } f n \text{ else } g n) \text{ sums } S'$

proof –

have $(\lambda n. g n + (\text{if } n \in A \text{ then } f n - g n \text{ else } 0)) \text{ sums } (S + (\sum n \in A. f n - g n))$
by (*intro sums-add assms sums-If-finite-set'*)

also have $(\lambda n. g\ n + (\text{if } n \in A \text{ then } f\ n - g\ n \text{ else } 0)) = (\lambda n. \text{if } n \in A \text{ then } f\ n \text{ else } g\ n)$
by (*simp add: fun-eq-iff*)
finally show *?thesis using assms by simp*
qed

110.5 Infinite summability on real normed vector spaces

context

fixes $f :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
begin

lemma *sums-Suc-iff*: $(\lambda n. f\ (\text{Suc } n)) \text{ sums } s \iff f \text{ sums } (s + f\ 0)$

proof –

have $f \text{ sums } (s + f\ 0) \iff (\lambda i. \sum j < \text{Suc } i. f\ j) \longrightarrow s + f\ 0$

by (*subst filterlim-sequentially-Suc (simp add: sums-def)*)

also have $\dots \iff (\lambda i. (\sum j < i. f\ (\text{Suc } j)) + f\ 0) \longrightarrow s + f\ 0$

by (*simp add: ac-simps lessThan-Suc-eq-insert-0 image-Suc-lessThan sum.atLeast1-atMost-eq*)

also have $\dots \iff (\lambda n. f\ (\text{Suc } n)) \text{ sums } s$

proof

assume $(\lambda i. (\sum j < i. f\ (\text{Suc } j)) + f\ 0) \longrightarrow s + f\ 0$

with *tendsto-add[OF this tendsto-const, of - f 0]* **show** $(\lambda i. f\ (\text{Suc } i)) \text{ sums } s$

by (*simp add: sums-def*)

qed (*auto intro: tendsto-add simp: sums-def*)

finally show *?thesis ..*

qed

lemma *summable-Suc-iff*: $\text{summable } (\lambda n. f\ (\text{Suc } n)) = \text{summable } f$

proof

assume *summable f*

then have $f \text{ sums } \text{suminf } f$

by (*rule summable-sums*)

then have $(\lambda n. f\ (\text{Suc } n)) \text{ sums } (\text{suminf } f - f\ 0)$

by (*simp add: sums-Suc-iff*)

then show $\text{summable } (\lambda n. f\ (\text{Suc } n))$

unfolding *summable-def* **by** *blast*

qed (*auto simp: sums-Suc-iff summable-def*)

lemma *sums-Suc-imp*: $f\ 0 = 0 \implies (\lambda n. f\ (\text{Suc } n)) \text{ sums } s \implies (\lambda n. f\ n) \text{ sums } s$
using *sums-Suc-iff* **by** *simp*

end

context

fixes $f :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$
begin

lemma *sums-diff*: $f \text{ sums } a \implies g \text{ sums } b \implies (\lambda n. f\ n - g\ n) \text{ sums } (a - b)$

unfolding *sums-def* **by** (*simp add: sum-subtractf tendsto-diff*)

lemma *summable-diff*: $\text{summable } f \implies \text{summable } g \implies \text{summable } (\lambda n. f\ n - g\ n)$

unfolding *summable-def* **by** (*auto intro: sums-diff*)

lemma *suminf-diff*: $\text{summable } f \implies \text{summable } g \implies \text{suminf } f - \text{suminf } g = (\sum n. f\ n - g\ n)$

by (*intro sums-unique sums-diff summable-sums*)

lemma *sums-minus*: $f\ \text{sums } a \implies (\lambda n. - f\ n)\ \text{sums } (- a)$

unfolding *sums-def* **by** (*simp add: sum-negf tendsto-minus*)

lemma *summable-minus*: $\text{summable } f \implies \text{summable } (\lambda n. - f\ n)$

unfolding *summable-def* **by** (*auto intro: sums-minus*)

lemma *suminf-minus*: $\text{summable } f \implies (\sum n. - f\ n) = - (\sum n. f\ n)$

by (*intro sums-unique [symmetric] sums-minus summable-sums*)

lemma *sums-iff-shift*: $(\lambda i. f\ (i + n))\ \text{sums } s \longleftrightarrow f\ \text{sums } (s + (\sum i < n. f\ i))$

proof (*induct n arbitrary: s*)

case 0

then show *?case* **by** *simp*

next

case (*Suc n*)

then have $(\lambda i. f\ (\text{Suc } i + n))\ \text{sums } s \longleftrightarrow (\lambda i. f\ (i + n))\ \text{sums } (s + f\ n)$

by (*subst sums-Suc-iff*) *simp*

with *Suc* **show** *?case*

by (*simp add: ac-simps*)

qed

corollary *sums-iff-shift'*: $(\lambda i. f\ (i + n))\ \text{sums } (s - (\sum i < n. f\ i)) \longleftrightarrow f\ \text{sums } s$

by (*simp add: sums-iff-shift*)

lemma *sums-zero-iff-shift*:

assumes $\bigwedge i. i < n \implies f\ i = 0$

shows $(\lambda i. f\ (i+n))\ \text{sums } s \longleftrightarrow (\lambda i. f\ i)\ \text{sums } s$

by (*simp add: asms sums-iff-shift*)

lemma *summable-iff-shift* [*simp*]: $\text{summable } (\lambda n. f\ (n + k)) \longleftrightarrow \text{summable } f$

by (*metis diff-add-cancel summable-def sums-iff-shift [abs-def]*)

lemma *sums-split-initial-segment*: $f\ \text{sums } s \implies (\lambda i. f\ (i + n))\ \text{sums } (s - (\sum i < n. f\ i))$

by (*simp add: sums-iff-shift*)

lemma *summable-ignore-initial-segment*: $\text{summable } f \implies \text{summable } (\lambda n. f\ (n + k))$

by (*simp add: summable-iff-shift*)

lemma *suminf-minus-initial-segment*: $\text{summable } f \implies (\sum n. f (n + k)) = (\sum n. f n) - (\sum i < k. f i)$

by (*rule sums-unique[symmetric]*) (*auto simp: sums-iff-shift*)

lemma *suminf-split-initial-segment*: $\text{summable } f \implies \text{suminf } f = (\sum n. f(n + k)) + (\sum i < k. f i)$

by (*auto simp add: suminf-minus-initial-segment*)

lemma *suminf-split-head*: $\text{summable } f \implies (\sum n. f (Suc n)) = \text{suminf } f - f 0$
using *suminf-split-initial-segment[of 1]* **by** *simp*

lemma *suminf-exist-split*:

fixes $r :: \text{real}$

assumes $0 < r$ **and** *summable* f

shows $\exists N. \forall n \geq N. \text{norm } (\sum i. f (i + n)) < r$

proof –

from *LIMSEQ-D[OF summable-LIMSEQ[OF ‹summable f› ‹0 < r›]*

obtain $N :: \text{nat}$ **where** $\forall n \geq N. \text{norm } (\text{sum } f \{..<n\} - \text{suminf } f) < r$

by *auto*

then show *?thesis*

by (*auto simp: norm-minus-commute suminf-minus-initial-segment[OF ‹summable f›]*)

qed

lemma *summable-LIMSEQ-zero*:

assumes *summable* f **shows** $f \longrightarrow 0$

proof –

have *Cauchy* $(\lambda n. \text{sum } f \{..<n\})$

using *LIMSEQ-imp-Cauchy assms summable-LIMSEQ* **by** *blast*

then show *?thesis*

unfolding *Cauchy-iff LIMSEQ-iff*

by (*metis add.commute add-diff-cancel-right' diff-zero le-SucI sum.lessThan-Suc*)

qed

lemma *summable-imp-convergent*: $\text{summable } f \implies \text{convergent } f$

by (*force dest!: summable-LIMSEQ-zero simp: convergent-def*)

lemma *summable-imp-Bseq*: $\text{summable } f \implies \text{Bseq } f$

by (*simp add: convergent-imp-Bseq summable-imp-convergent*)

end

lemma *summable-minus-iff*: $\text{summable } (\lambda n. - f n) \iff \text{summable } f$

for $f :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector}$

by (*auto dest: summable-minus*)

lemma (**in** *bounded-linear*) *sums*: $(\lambda n. X n) \text{ sums } a \implies (\lambda n. f (X n)) \text{ sums } (f a)$

unfolding *sums-def* **by** (*drule tendsto*) (*simp only: sum*)

lemma (in *bounded-linear*) *summable*: $\text{summable } (\lambda n. X n) \implies \text{summable } (\lambda n. f (X n))$

unfolding *summable-def* **by** (*auto intro: sums*)

lemma (in *bounded-linear*) *suminf*: $\text{summable } (\lambda n. X n) \implies f (\sum n. X n) = (\sum n. f (X n))$

by (*intro sums-unique sums summable-sums*)

lemmas *sums-of-real* = *bounded-linear.sums* [*OF bounded-linear-of-real*]

lemmas *summable-of-real* = *bounded-linear.summable* [*OF bounded-linear-of-real*]

lemmas *suminf-of-real* = *bounded-linear.suminf* [*OF bounded-linear-of-real*]

lemmas *sums-scaleR-left* = *bounded-linear.sums*[*OF bounded-linear-scaleR-left*]

lemmas *summable-scaleR-left* = *bounded-linear.summable*[*OF bounded-linear-scaleR-left*]

lemmas *suminf-scaleR-left* = *bounded-linear.suminf*[*OF bounded-linear-scaleR-left*]

lemmas *sums-scaleR-right* = *bounded-linear.sums*[*OF bounded-linear-scaleR-right*]

lemmas *summable-scaleR-right* = *bounded-linear.summable*[*OF bounded-linear-scaleR-right*]

lemmas *suminf-scaleR-right* = *bounded-linear.suminf*[*OF bounded-linear-scaleR-right*]

lemma *summable-const-iff*: $\text{summable } (\lambda-. c) \iff c = 0$

for $c :: 'a::\text{real-normed-vector}$

proof –

have $\neg \text{summable } (\lambda-. c)$ **if** $c \neq 0$

proof –

from *that* **have** *filterlim* $(\lambda n. \text{of_nat } n * \text{norm } c)$ *at-top sequentially*

by (*subst mult.commute*)

(*auto intro!: filterlim-tendsto-pos-mult-at-top filterlim-real-sequentially*)

then **have** $\neg \text{convergent } (\lambda n. \text{norm } (\sum k < n. c))$

by (*intro filterlim-at-infinity-imp-not-convergent filterlim-at-top-imp-at-infinity*)

(*simp-all add: sum-constant-scaleR*)

then **show** *?thesis*

unfolding *summable-iff-convergent* **using** *convergent-norm* **by** *blast*

qed

then **show** *?thesis* **by** *auto*

qed

110.6 Infinite summability on real normed algebras

context

fixes $f :: \text{nat} \Rightarrow 'a::\text{real-normed-algebra}$

begin

lemma *sums-mult*: $f \text{ sums } a \implies (\lambda n. c * f n) \text{ sums } (c * a)$

by (*rule bounded-linear.sums* [*OF bounded-linear-mult-right*])

lemma *summable-mult*: $\text{summable } f \implies \text{summable } (\lambda n. c * f n)$

by (*rule bounded-linear.summable* [*OF bounded-linear-mult-right*])

lemma *suminf-mult*: $\text{summable } f \implies \text{suminf } (\lambda n. c * f n) = c * \text{suminf } f$
by (rule *bounded-linear.suminf* [*OF bounded-linear-mult-right, symmetric*])

lemma *sums-mult2*: $f \text{ sums } a \implies (\lambda n. f n * c) \text{ sums } (a * c)$
by (rule *bounded-linear.sums* [*OF bounded-linear-mult-left*])

lemma *summable-mult2*: $\text{summable } f \implies \text{summable } (\lambda n. f n * c)$
by (rule *bounded-linear.summable* [*OF bounded-linear-mult-left*])

lemma *suminf-mult2*: $\text{summable } f \implies \text{suminf } f * c = (\sum n. f n * c)$
by (rule *bounded-linear.suminf* [*OF bounded-linear-mult-left*])

end

lemma *sums-mult-iff*:
fixes $f :: \text{nat} \Rightarrow 'a::\{\text{real-normed-algebra, field}\}$
assumes $c \neq 0$
shows $(\lambda n. c * f n) \text{ sums } (c * d) \longleftrightarrow f \text{ sums } d$
using *sums-mult*[*of f d c*] *sums-mult*[*of $\lambda n. c * f n$ c * d inverse c*]
by (force *simp: field-simps assms*)

lemma *sums-mult2-iff*:
fixes $f :: \text{nat} \Rightarrow 'a::\{\text{real-normed-algebra, field}\}$
assumes $c \neq 0$
shows $(\lambda n. f n * c) \text{ sums } (d * c) \longleftrightarrow f \text{ sums } d$
using *sums-mult-iff*[*OF assms, of f d*] **by** (*simp add: mult.commute*)

lemma *sums-of-real-iff*:
 $(\lambda n. \text{of-real } (f n) :: 'a::\{\text{real-normed-div-algebra}\}) \text{ sums of-real } c \longleftrightarrow f \text{ sums } c$
by (*simp add: sums-def of-real-sum[symmetric] tendsto-of-real-iff del: of-real-sum*)

110.7 Infinite summability on real normed fields

context

fixes $c :: 'a::\{\text{real-normed-field}\}$

begin

lemma *sums-divide*: $f \text{ sums } a \implies (\lambda n. f n / c) \text{ sums } (a / c)$
by (rule *bounded-linear.sums* [*OF bounded-linear-divide*])

lemma *summable-divide*: $\text{summable } f \implies \text{summable } (\lambda n. f n / c)$
by (rule *bounded-linear.summable* [*OF bounded-linear-divide*])

lemma *suminf-divide*: $\text{summable } f \implies \text{suminf } (\lambda n. f n / c) = \text{suminf } f / c$
by (rule *bounded-linear.suminf* [*OF bounded-linear-divide, symmetric*])

lemma *summable-inverse-divide*: $\text{summable } (\text{inverse} \circ f) \implies \text{summable } (\lambda n. c / f n)$
by (*auto dest: summable-mult [of - c] simp: field-simps*)

lemma *sums-mult-D*: $(\lambda n. c * f n) \text{ sums } a \implies c \neq 0 \implies f \text{ sums } (a/c)$
using *sums-mult-iff* **by** *fastforce*

lemma *summable-mult-D*: $\text{summable } (\lambda n. c * f n) \implies c \neq 0 \implies \text{summable } f$
by (*auto dest: summable-divide*)

Sum of a geometric progression.

lemma *geometric-sums*:

assumes *norm c < 1*

shows $(\lambda n. c^n) \text{ sums } (1 / (1 - c))$

proof –

have *neq-0: c - 1 ≠ 0*

using *assms* **by** *auto*

then have $(\lambda n. c^n / (c - 1) - 1 / (c - 1)) \longrightarrow 0 / (c - 1) - 1 / (c - 1)$

by (*intro tendsto-intros assms*)

then have $(\lambda n. (c^n - 1) / (c - 1)) \longrightarrow 1 / (1 - c)$

by (*simp add: nonzero-minus-divide-right [OF neq-0] diff-divide-distrib*)

with *neq-0* **show** $(\lambda n. c^n) \text{ sums } (1 / (1 - c))$

by (*simp add: sums-def geometric-sum*)

qed

lemma *summable-geometric*: $\text{norm } c < 1 \implies \text{summable } (\lambda n. c^n)$

by (*rule geometric-sums [THEN sums-summable]*)

lemma *suminf-geometric*: $\text{norm } c < 1 \implies \text{suminf } (\lambda n. c^n) = 1 / (1 - c)$

by (*rule sums-unique[symmetric]*) (*rule geometric-sums*)

lemma *summable-geometric-iff* [*simp*]: $\text{summable } (\lambda n. c^n) \iff \text{norm } c < 1$

proof

assume *summable* $(\lambda n. c^n :: 'a :: \text{real-normed-field})$

then have $(\lambda n. \text{norm } c^n) \longrightarrow 0$

by (*simp add: norm-power [symmetric] tendsto-norm-zero-iff summable-LIMSEQ-zero*)

from *order-tendstoD(2)[OF this zero-less-one]* **obtain** *n* **where** $\text{norm } c^n < 1$

by (*auto simp: eventually-at-top-linorder*)

then show $\text{norm } c < 1$ **using** *one-le-power[of norm c n]*

by (*cases norm c ≥ 1*) (*linarith, simp*)

qed (*rule summable-geometric*)

end

Biconditional versions for constant factors

context

fixes $c :: 'a :: \text{real-normed-field}$

begin

lemma *summable-cmult-iff* [*simp*]: $\text{summable } (\lambda n. c * f n) \iff c=0 \vee \text{summable } f$

proof –
have $\llbracket \text{summable } (\lambda n. c * f n); c \neq 0 \rrbracket \implies \text{summable } f$
using *summable-mult-D* **by** *blast*
then show *?thesis*
by (*auto simp: summable-mult*)
qed

lemma *summable-divide-iff* [*simp*]: $\text{summable } (\lambda n. f n / c) \longleftrightarrow c \neq 0 \vee \text{summable } f$

proof –
have $\llbracket \text{summable } (\lambda n. f n / c); c \neq 0 \rrbracket \implies \text{summable } f$
by (*auto dest: summable-divide* [**where** $c = 1/c$])
then show *?thesis*
by (*auto simp: summable-divide*)
qed

end

lemma *power-half-series*: $(\lambda n. (1/2::\text{real})^{\text{Suc } n}) \text{ sums } 1$

proof –
have $2: (\lambda n. (1/2::\text{real})^n) \text{ sums } 2$
using *geometric-sums* [*of* $1/2::\text{real}$] **by** *auto*
have $(\lambda n. (1/2::\text{real})^{\text{Suc } n}) = (\lambda n. (1/2)^n / 2)$
by (*simp add: mult.commute*)
then show *?thesis*
using *sums-divide* [*OF* 2 , *of* 2] **by** *simp*
qed

110.8 Telescoping

lemma *telescope-sums*:

fixes $c :: 'a::\text{real-normed-vector}$
assumes $f \longrightarrow c$
shows $(\lambda n. f (\text{Suc } n) - f n) \text{ sums } (c - f 0)$
unfolding *sums-def*

proof (*subst filterlim-sequentially-Suc* [*symmetric*])

have $(\lambda n. \sum k < \text{Suc } n. f (\text{Suc } k) - f k) = (\lambda n. f (\text{Suc } n) - f 0)$
by (*simp add: lessThan-Suc-atMost atLeast0AtMost* [*symmetric*] *sum-Suc-diff*)
also have $\dots \longrightarrow c - f 0$

by (*intro tendsto-diff LIMSEQ-Suc* [*OF* *assms*] *tendsto-const*)

finally show $(\lambda n. \sum n < \text{Suc } n. f (\text{Suc } n) - f n) \longrightarrow c - f 0$.

qed

lemma *telescope-sums'*:

fixes $c :: 'a::\text{real-normed-vector}$
assumes $f \longrightarrow c$
shows $(\lambda n. f n - f (\text{Suc } n)) \text{ sums } (f 0 - c)$
using *sums-minus* [*OF* *telescope-sums* [*OF* *assms*]] **by** (*simp add: algebra-simps*)

lemma *telescope-summable*:
fixes $c :: 'a::real-normed-vector$
assumes $f \longrightarrow c$
shows *summable* $(\lambda n. f (Suc\ n) - f\ n)$
using *telescope-sums*[*OF assms*] **by** (*simp add: sums-iff*)

lemma *telescope-summable'*:
fixes $c :: 'a::real-normed-vector$
assumes $f \longrightarrow c$
shows *summable* $(\lambda n. f\ n - f (Suc\ n))$
using *summable-minus*[*OF telescope-summable*[*OF assms*]] **by** (*simp add: algebra-simps*)

110.9 Infinite summability on Banach spaces

Cauchy-type criterion for convergence of series (c.f. Harrison).

lemma *summable-Cauchy*: *summable* $f \longleftrightarrow (\forall e > 0. \exists N. \forall m \geq N. \forall n. \text{norm} (\text{sum } f \{m..<n\}) < e)$ (**is** - = *?rhs*)
for $f :: nat \Rightarrow 'a::banach$

proof

assume $f: \text{summable } f$
show *?rhs*
proof *clarify*
fix $e :: real$
assume $0 < e$
then obtain M **where** $M: \bigwedge m\ n. \llbracket m \geq M; n \geq M \rrbracket \implies \text{norm} (\text{sum } f \{..<m\}) - \text{sum } f \{..<n\}) < e$
using f **by** (*force simp add: summable-iff-convergent Cauchy-convergent-iff [symmetric] Cauchy-iff*)
have $\text{norm} (\text{sum } f \{m..<n\}) < e$ **if** $m \geq M$ **for** $m\ n$
proof (*cases m n rule: linorder-class.le-cases*)
assume $m \leq n$
then show *?thesis*
by (*metis (mono-tags, opaque-lifting) M atLeast0LessThan order-trans sum-diff-nat-ivl that zero-le*)
next
assume $n \leq m$
then show *?thesis*
by (*simp add: ‹0 < e›*)
qed
then show $\exists N. \forall m \geq N. \forall n. \text{norm} (\text{sum } f \{m..<n\}) < e$
by *blast*
qed
next
assume $r: ?rhs$
then show *summable* f
unfolding *summable-iff-convergent Cauchy-convergent-iff [symmetric] Cauchy-iff*
proof *clarify*
fix $e :: real$

```

assume  $0 < e$ 
with  $r$  obtain  $N$  where  $N: \bigwedge m n. m \geq N \implies \text{norm} (\text{sum } f \{m..<n\}) < e$ 
  by blast
have  $\text{norm} (\text{sum } f \{..<m\} - \text{sum } f \{..<n\}) < e$  if  $m \geq N$   $n \geq N$  for  $m$   $n$ 
proof (cases  $m$   $n$  rule: linorder-class.le-cases)
  assume  $m \leq n$ 
  then show ?thesis
    by (metis  $N$  finite-lessThan lessThan-minus-lessThan lessThan-subset-iff
norm-minus-commute sum-diff  $\langle m \geq N \rangle$ )
  next
    assume  $n \leq m$ 
    then show ?thesis
      by (metis  $N$  finite-lessThan lessThan-minus-lessThan lessThan-subset-iff
sum-diff  $\langle n \geq N \rangle$ )
  qed
then show  $\exists M. \forall m \geq M. \forall n \geq M. \text{norm} (\text{sum } f \{..<m\} - \text{sum } f \{..<n\}) < e$ 
  by blast
qed
qed

```

lemma *summable-Cauchy'*:

```

fixes  $f :: \text{nat} \Rightarrow 'a :: \text{banach}$ 
assumes  $ev: \text{eventually} (\lambda m. \forall n \geq m. \text{norm} (\text{sum } f \{m..<n\}) \leq g\ m)$  sequentially
assumes  $g0: g \longrightarrow 0$ 
shows summable  $f$ 
proof (subst summable-Cauchy, intro allI impI, goal-cases)
  case (1  $e$ )
  then have  $\forall_F x$  in sequentially.  $g\ x < e$ 
    using  $g0$  order-tendstoD(2) by blast
  with  $ev$  have eventually  $(\lambda m. \forall n. \text{norm} (\text{sum } f \{m..<n\}) < e)$  at-top
  proof eventually-elim
    case (elim  $m$ )
    show ?case
    proof
      fix  $n$ 
      from elim show  $\text{norm} (\text{sum } f \{m..<n\}) < e$ 
        by (cases  $n \geq m$ ) auto
    qed
  qed
  thus ?case by (auto simp: eventually-at-top-linorder)
qed

```

context

```

fixes  $f :: \text{nat} \Rightarrow 'a :: \text{banach}$ 
begin

```

Absolute convergence implies normal convergence.

```

lemma summable-norm-cancel: summable  $(\lambda n. \text{norm} (f\ n)) \implies \text{summable } f$ 
unfolding summable-Cauchy

```

apply (*erule all-forward imp-forward ex-forward | assumption*)+
apply (*fastforce simp add: order-le-less-trans [OF norm-sum] order-le-less-trans [OF abs-ge-self]*)
done

lemma *summable-norm*: $\text{summable } (\lambda n. \text{norm } (f n)) \implies \text{norm } (\text{suminf } f) \leq (\sum n. \text{norm } (f n))$

by (*auto intro: LIMSEQ-le tendsto-norm summable-norm-cancel summable-LIMSEQ norm-sum*)

Comparison tests.

lemma *summable-comparison-test*:

assumes $fg: \exists N. \forall n \geq N. \text{norm } (f n) \leq g n$ **and** $g: \text{summable } g$
shows *summable* f

proof –

obtain N **where** $N: \bigwedge n. n \geq N \implies \text{norm } (f n) \leq g n$

using *assms* **by** *blast*

show *?thesis*

proof (*clarsimp simp add: summable-Cauchy*)

fix $e :: \text{real}$

assume $0 < e$

then obtain Ng **where** $Ng: \bigwedge m n. m \geq Ng \implies \text{norm } (\text{sum } g \{m..<n\}) < e$

using g **by** (*fastforce simp: summable-Cauchy*)

with N **have** $\text{norm } (\text{sum } f \{m..<n\}) < e$ **if** $m \geq \max N Ng$ **for** $m n$

proof –

have $\text{norm } (\text{sum } f \{m..<n\}) \leq \text{sum } g \{m..<n\}$

using N **that** **by** (*force intro: sum-norm-le*)

also have $\dots \leq \text{norm } (\text{sum } g \{m..<n\})$

by *simp*

also have $\dots < e$

using Ng **that** **by** *auto*

finally show *?thesis* .

qed

then show $\exists N. \forall m \geq N. \forall n. \text{norm } (\text{sum } f \{m..<n\}) < e$

by *blast*

qed

qed

lemma *summable-comparison-test-ev*:

eventually $(\lambda n. \text{norm } (f n) \leq g n)$ *sequentially* $\implies \text{summable } g \implies \text{summable } f$

by (*rule summable-comparison-test*) (*auto simp: eventually-at-top-linorder*)

A better argument order.

lemma *summable-comparison-test'*: $\text{summable } g \implies (\bigwedge n. n \geq N \implies \text{norm } (f n) \leq g n) \implies \text{summable } f$

by (*rule summable-comparison-test*) *auto*

110.10 The Ratio Test

lemma *summable-ratio-test*:

assumes $c < 1 \wedge n. n \geq N \implies \text{norm } (f \text{ (Suc } n)) \leq c * \text{norm } (f \text{ } n)$

shows *summable* f

proof (*cases* $0 < c$)

case *True*

show *summable* f

proof (*rule* *summable-comparison-test*)

show $\exists N'. \forall n \geq N'. \text{norm } (f \text{ } n) \leq (\text{norm } (f \text{ } N) / (c \wedge N)) * c \wedge n$

proof (*intro* *exI* *allI* *impI*)

fix n

assume $N \leq n$

then show $\text{norm } (f \text{ } n) \leq (\text{norm } (f \text{ } N) / (c \wedge N)) * c \wedge n$

proof (*induct* *rule: inc-induct*)

case *base*

with *True* **show** *?case* **by** *simp*

next

case (*step* m)

have $\text{norm } (f \text{ (Suc } m)) / c \wedge \text{Suc } m * c \wedge n \leq \text{norm } (f \text{ } m) / c \wedge m * c \wedge n$

using $\langle 0 < c \rangle \langle c < 1 \rangle \text{assms}(2)[OF \langle N \leq m \rangle]$ **by** (*simp* *add: field-simps*)

with *step* **show** *?case* **by** *simp*

qed

qed

show *summable* $(\lambda n. \text{norm } (f \text{ } N) / c \wedge N * c \wedge n)$

using $\langle 0 < c \rangle \langle c < 1 \rangle$ **by** (*intro* *summable-mult* *summable-geometric*) *simp*

qed

next

case *False*

have $f \text{ (Suc } n) = 0$ **if** $n \geq N$ **for** n

proof –

from *that* **have** $\text{norm } (f \text{ (Suc } n)) \leq c * \text{norm } (f \text{ } n)$

by (*rule* *assms*(2))

also have $\dots \leq 0$

using *False* **by** (*simp* *add: not-less mult-nonpos-nonneg*)

finally show *?thesis*

by *auto*

qed

then show *summable* f

by (*intro* *sums-summable*[*OF* *sums-finite*, *of* $\{.. \text{Suc } N\}$]) (*auto* *simp: not-le Suc-less-eq2*)

qed

end

Application to convergence of the log function

lemma *norm-summable-ln-series*:

fixes $z :: 'a :: \{\text{real-normed-field, banach}\}$

assumes $\text{norm } z < 1$

shows *summable* $(\lambda n. \text{norm } (z \wedge n / \text{of-nat } n))$


```

proof (rule summable-comparison-test)
  show summable ( $\lambda n. \text{norm } (z \wedge n)$ )
    using assms unfolding norm-power by (intro summable-geometric) auto
  have  $\text{norm } z \wedge n / \text{real } n \leq \text{norm } z \wedge n$  for  $n$ 
  proof (cases n = 0)
    case False
    hence  $\text{norm } z \wedge n * 1 \leq \text{norm } z \wedge n * \text{real } n$ 
    by (intro mult-left-mono) auto
    thus ?thesis
    using False by (simp add: field-simps)
  qed auto
  thus  $\exists N. \forall n \geq N. \text{norm } (\text{norm } (z \wedge n / \text{of-nat } n)) \leq \text{norm } (z \wedge n)$ 
    by (intro exI[of - 0]) (auto simp: norm-power norm-divide)
qed

```

Relations among convergence and absolute convergence for power series.

lemma *Abel-lemma:*

```

fixes  $a :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$ 
assumes  $r: 0 \leq r$ 
  and  $r0: r < r0$ 
  and  $M: \bigwedge n. \text{norm } (a \ n) * r0 \wedge n \leq M$ 
shows summable ( $\lambda n. \text{norm } (a \ n) * r \wedge n$ )
proof (rule summable-comparison-test')
  show summable ( $\lambda n. M * (r / r0) \wedge n$ )
    using assms by (auto simp add: summable-mult summable-geometric)
  show  $\text{norm } (\text{norm } (a \ n) * r \wedge n) \leq M * (r / r0) \wedge n$  for  $n$ 
    using  $r \ r0 \ M$  [of n] dual-order.order-iff-strict
    by (fastforce simp add: abs-mult field-simps)
qed

```

Summability of geometric series for real algebras.

lemma *complete-algebra-summable-geometric:*

```

fixes  $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$ 
assumes  $\text{norm } x < 1$ 
shows summable ( $\lambda n. x \wedge n$ )
proof (rule summable-comparison-test)
  show  $\exists N. \forall n \geq N. \text{norm } (x \wedge n) \leq \text{norm } x \wedge n$ 
    by (simp add: norm-power-ineq)
  from assms show summable ( $\lambda n. \text{norm } x \wedge n$ )
    by (simp add: summable-geometric)
qed

```

110.11 Cauchy Product Formula

Proof based on Analysis WebNotes: Chapter 07, Class 41 <http://www.math.unl.edu/~webnotes/classes/class41/prp77.htm>

lemma *Cauchy-product-sums:*

```

fixes  $a \ b :: \text{nat} \Rightarrow 'a::\{\text{real-normed-algebra}, \text{banach}\}$ 

```

```

assumes  $a$ : summable ( $\lambda k. \text{norm } (a\ k)$ )
and  $b$ : summable ( $\lambda k. \text{norm } (b\ k)$ )
shows ( $\lambda k. \sum_{i \leq k}. a\ i * b\ (k - i)$ ) sums ( $(\sum k. a\ k) * (\sum k. b\ k)$ )
proof -
let ?S1 =  $\lambda n::\text{nat}. \{..<n\} \times \{..<n\}$ 
let ?S2 =  $\lambda n::\text{nat}. \{(i,j). i + j < n\}$ 
have S1-mono:  $\bigwedge m\ n. m \leq n \implies ?S1\ m \subseteq ?S1\ n$  by auto
have S2-le-S1:  $\bigwedge n. ?S2\ n \subseteq ?S1\ n$  by auto
have S1-le-S2:  $\bigwedge n. ?S1\ (n\ \text{div}\ 2) \subseteq ?S2\ n$  by auto
have finite-S1:  $\bigwedge n. \text{finite } (?S1\ n)$  by simp
with S2-le-S1 have finite-S2:  $\bigwedge n. \text{finite } (?S2\ n)$  by (rule finite-subset)

let ?g =  $\lambda(i,j). a\ i * b\ j$ 
let ?f =  $\lambda(i,j). \text{norm } (a\ i) * \text{norm } (b\ j)$ 
have f-nonneg:  $\bigwedge x. 0 \leq ?f\ x$  by auto
then have norm-sum-f:  $\bigwedge A. \text{norm } (\text{sum } ?f\ A) = \text{sum } ?f\ A$ 
unfolding real-norm-def
by (simp only: abs-of-nonneg sum-nonneg [rule-format])

have ( $\lambda n. (\sum_{k < n}. a\ k) * (\sum_{k < n}. b\ k)$ )  $\longrightarrow$  ( $\sum k. a\ k$ ) * ( $\sum k. b\ k$ )
by (intro tendsto-mult summable-LIMSEQ summable-norm-cancel [OF a] summable-norm-cancel
[OF b])
then have 1: ( $\lambda n. \text{sum } ?g\ (?S1\ n)$ )  $\longrightarrow$  ( $\sum k. a\ k$ ) * ( $\sum k. b\ k$ )
by (simp only: sum-product sum.Sigma [rule-format] finite-lessThan)

have ( $\lambda n. (\sum_{k < n}. \text{norm } (a\ k)) * (\sum_{k < n}. \text{norm } (b\ k))$ )  $\longrightarrow$  ( $\sum k. \text{norm } (a\ k)$ ) * ( $\sum k. \text{norm } (b\ k)$ )
using a b by (intro tendsto-mult summable-LIMSEQ)
then have ( $\lambda n. \text{sum } ?f\ (?S1\ n)$ )  $\longrightarrow$  ( $\sum k. \text{norm } (a\ k)$ ) * ( $\sum k. \text{norm } (b\ k)$ )
by (simp only: sum-product sum.Sigma [rule-format] finite-lessThan)
then have convergent ( $\lambda n. \text{sum } ?f\ (?S1\ n)$ )
by (rule convergentI)
then have Cauchy: Cauchy ( $\lambda n. \text{sum } ?f\ (?S1\ n)$ )
by (rule convergent-Cauchy)
have Zfun ( $\lambda n. \text{sum } ?f\ (?S1\ n - ?S2\ n)$ ) sequentially
proof (rule ZfunI, simp only: eventually-sequentially norm-sum-f)
fix r :: real
assume r:  $0 < r$ 
from CauchyD [OF Cauchy r] obtain N
where  $\forall m \geq N. \forall n \geq N. \text{norm } (\text{sum } ?f\ (?S1\ m) - \text{sum } ?f\ (?S1\ n)) < r$  ..
then have  $\bigwedge m\ n. N \leq n \implies n \leq m \implies \text{norm } (\text{sum } ?f\ (?S1\ m - ?S1\ n)) <$ 
 $r$ 
by (simp only: sum-diff finite-S1 S1-mono)
then have N:  $\bigwedge m\ n. N \leq n \implies n \leq m \implies \text{sum } ?f\ (?S1\ m - ?S1\ n) < r$ 
by (simp only: norm-sum-f)
show  $\exists N. \forall n \geq N. \text{sum } ?f\ (?S1\ n - ?S2\ n) < r$ 
proof (intro exI allI impI)
fix n
assume  $2 * N \leq n$ 

```

then have $n: N \leq n \text{ div } 2$ **by** *simp*
have $\text{sum } ?f (?S1\ n - ?S2\ n) \leq \text{sum } ?f (?S1\ n - ?S1\ (n \text{ div } 2))$
by (*intro sum-mono2 finite-Diff finite-S1 f-nonneg Diff-mono subset-refl S1-le-S2*)
also have $\dots < r$
using $n \text{ div-le-dividend}$ **by** (*rule N*)
finally show $\text{sum } ?f (?S1\ n - ?S2\ n) < r$.
qed
qed
then have $Zfun (\lambda n. \text{sum } ?g (?S1\ n - ?S2\ n))$ *sequentially*
apply (*rule Zfun-le [rule-format]*)
apply (*simp only: norm-sum-f*)
apply (*rule order-trans [OF norm-sum sum-mono]*)
apply (*auto simp add: norm-mult-ineq*)
done
then have $2: (\lambda n. \text{sum } ?g (?S1\ n) - \text{sum } ?g (?S2\ n)) \longrightarrow 0$
unfolding *tendsto-Zfun-iff diff-0-right*
by (*simp only: sum-diff finite-S1 S2-le-S1*)
with 1 **have** $(\lambda n. \text{sum } ?g (?S2\ n)) \longrightarrow (\sum k. a\ k) * (\sum k. b\ k)$
by (*rule Lim-transform2*)
then show *?thesis*
by (*simp only: sums-def sum.triangle-reindex*)
qed

lemma *Cauchy-product:*

fixes $a\ b :: \text{nat} \Rightarrow 'a::\{\text{real-normed-algebra,banach}\}$
assumes *summable* $(\lambda k. \text{norm } (a\ k))$
and *summable* $(\lambda k. \text{norm } (b\ k))$
shows $(\sum k. a\ k) * (\sum k. b\ k) = (\sum k. \sum i \leq k. a\ i * b\ (k - i))$
using *assms* **by** (*rule Cauchy-product-sums [THEN sums-unique]*)

lemma *summable-Cauchy-product:*

fixes $a\ b :: \text{nat} \Rightarrow 'a::\{\text{real-normed-algebra,banach}\}$
assumes *summable* $(\lambda k. \text{norm } (a\ k))$
and *summable* $(\lambda k. \text{norm } (b\ k))$
shows *summable* $(\lambda k. \sum i \leq k. a\ i * b\ (k - i))$
using *Cauchy-product-sums [OF assms]* **by** (*simp add: sums-iff*)

110.12 Series on reals

lemma *summable-norm-comparison-test:*

$\exists N. \forall n \geq N. \text{norm } (f\ n) \leq g\ n \implies \text{summable } g \implies \text{summable } (\lambda n. \text{norm } (f\ n))$
by (*rule summable-comparison-test*) *auto*

lemma *summable-rabs-comparison-test:* $\exists N. \forall n \geq N. |f\ n| \leq g\ n \implies \text{summable } g \implies \text{summable } (\lambda n. |f\ n|)$

for $f :: \text{nat} \Rightarrow \text{real}$

by (*rule summable-comparison-test*) *auto*

lemma *summable-rabs-cancel*: $\text{summable } (\lambda n. |f n|) \implies \text{summable } f$
for $f :: \text{nat} \Rightarrow \text{real}$
by (rule *summable-norm-cancel*) *simp*

lemma *summable-rabs*: $\text{summable } (\lambda n. |f n|) \implies |\text{suminf } f| \leq (\sum n. |f n|)$
for $f :: \text{nat} \Rightarrow \text{real}$
by (fold *real-norm-def*) (rule *summable-norm*)

lemma *norm-suminf-le*:
assumes $\bigwedge n. \text{norm } (f n :: 'a :: \text{banach}) \leq g n$ *summable* g
shows $\text{norm } (\text{suminf } f) \leq \text{suminf } g$
proof –
have $*$: *summable* $(\lambda n. \text{norm } (f n))$
using *assms summable-norm-comparison-test* **by** *blast*
hence $\text{norm } (\text{suminf } f) \leq (\sum n. \text{norm } (f n))$ **by** (*intro summable-norm*) *auto*
also have $\dots \leq \text{suminf } g$ **by** (*intro suminf-le * assms allI*)
finally show *?thesis* .
qed

lemma *summable-zero-power* [*simp*]: $\text{summable } (\lambda n. 0 \wedge n :: 'a :: \{\text{comm-ring-1}, \text{topological-space}\})$
proof –
have $(\lambda n. 0 \wedge n :: 'a) = (\lambda n. \text{if } n = 0 \text{ then } 0 \wedge 0 \text{ else } 0)$
by (*intro ext*) (*simp add: zero-power*)
moreover have *summable* \dots **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *summable-zero-power'* [*simp*]: $\text{summable } (\lambda n. f n * 0 \wedge n :: 'a :: \{\text{ring-1}, \text{topological-space}\})$
proof –
have $(\lambda n. f n * 0 \wedge n :: 'a) = (\lambda n. \text{if } n = 0 \text{ then } f 0 * 0 \wedge 0 \text{ else } 0)$
by (*intro ext*) (*simp add: zero-power*)
moreover have *summable* \dots **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *summable-power-series*:
fixes $z :: \text{real}$
assumes *le-1*: $\bigwedge i. f i \leq 1$
and *nonneg*: $\bigwedge i. 0 \leq f i$
and $z: 0 \leq z < 1$
shows *summable* $(\lambda i. f i * z \wedge i)$
proof (rule *summable-comparison-test*[*OF - summable-geometric*])
show $\text{norm } z < 1$
using z **by** (*auto simp: less-imp-le*)
show $\bigwedge n. \exists N. \forall na \geq N. \text{norm } (f na * z \wedge na) \leq z \wedge na$
using z
by (*auto intro!*: *exI*[*of - 0*] *mult-left-le-one-le simp: abs-mult nonneg power-abs less-imp-le le-1*)
qed

lemma *summable-0-powser*: *summable* ($\lambda n. f n * 0 \wedge n :: 'a::\text{real-normed-div-algebra}$)
by *simp*

lemma *summable-powser-split-head*:

summable ($\lambda n. f (Suc n) * z \wedge n :: 'a::\text{real-normed-div-algebra}$) = *summable* ($\lambda n. f n * z \wedge n$)

proof –

have *summable* ($\lambda n. f (Suc n) * z \wedge n$) \longleftrightarrow *summable* ($\lambda n. f (Suc n) * z \wedge Suc n$)

(**is** *?lhs* \longleftrightarrow *?rhs*)

proof

show *?rhs* **if** *?lhs*

using *summable-mult2*[*OF that, of z*]

by (*simp add: power-commutes algebra-simps*)

show *?lhs* **if** *?rhs*

using *summable-mult2*[*OF that, of inverse z*]

by (*cases z \neq 0, subst (asm) power-Suc2*) (*simp-all add: algebra-simps*)

qed

also have ... \longleftrightarrow *summable* ($\lambda n. f n * z \wedge n$) **by** (*rule summable-Suc-iff*)

finally show *?thesis* .

qed

lemma *summable-powser-ignore-initial-segment*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{real-normed-div-algebra}$

shows *summable* ($\lambda n. f (n + m) * z \wedge n$) \longleftrightarrow *summable* ($\lambda n. f n * z \wedge n$)

proof (*induction m*)

case (*Suc m*)

have *summable* ($\lambda n. f (n + Suc m) * z \wedge n$) = *summable* ($\lambda n. f (Suc n + m) * z \wedge n$)

by *simp*

also have ... = *summable* ($\lambda n. f (n + m) * z \wedge n$)

by (*rule summable-powser-split-head*)

also have ... = *summable* ($\lambda n. f n * z \wedge n$)

by (*rule Suc.IH*)

finally show *?case* .

qed *simp-all*

lemma *powser-split-head*:

fixes $f :: \text{nat} \Rightarrow 'a::\{\text{real-normed-div-algebra}, \text{banach}\}$

assumes *summable* ($\lambda n. f n * z \wedge n$)

shows *suminf* ($\lambda n. f n * z \wedge n$) = $f 0 + \text{suminf}$ ($\lambda n. f (Suc n) * z \wedge n$) * z

and *suminf* ($\lambda n. f (Suc n) * z \wedge n$) * z = *suminf* ($\lambda n. f n * z \wedge n$) – $f 0$

and *summable* ($\lambda n. f (Suc n) * z \wedge n$)

proof –

from *assms* **show** *summable* ($\lambda n. f (Suc n) * z \wedge n$)

by (*subst summable-powser-split-head*)

from *suminf-mult2*[*OF this, of z*]

have ($\sum n. f (Suc n) * z \wedge n$) * z = ($\sum n. f (Suc n) * z \wedge Suc n$)

by (*simp add: power-commutes algebra-simps*)
 also from *assms* have $\dots = \text{suminf } (\lambda n. f n * z ^ n) - f 0$
 by (*subst suminf-split-head*) *simp-all*
 finally show $\text{suminf } (\lambda n. f n * z ^ n) = f 0 + \text{suminf } (\lambda n. f (Suc n) * z ^ n)$
 $* z$
 by *simp*
 then show $\text{suminf } (\lambda n. f (Suc n) * z ^ n) * z = \text{suminf } (\lambda n. f n * z ^ n) - f 0$
 by *simp*
 qed

lemma *summable-partial-sum-bound*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{banach}$
 and $e :: \text{real}$
 assumes *summable*: *summable* f
 and $e: e > 0$
 obtains N where $\bigwedge m n. m \geq N \implies \text{norm } (\sum_{k=m..n} f k) < e$
 proof –
 from *summable* have *Cauchy* $(\lambda n. \sum_{k<n} f k)$
 by (*simp add: Cauchy-convergent-iff summable-iff-convergent*)
 from *CauchyD* [*OF this e*] obtain N
 where $N: \bigwedge m n. m \geq N \implies n \geq N \implies \text{norm } ((\sum_{k<m} f k) - (\sum_{k<n} f k)) < e$
 by *blast*
 have $\text{norm } (\sum_{k=m..n} f k) < e$ if $m: m \geq N$ for $m n$
 proof (cases $n \geq m$)
 case *True*
 with m have $\text{norm } ((\sum_{k<Suc\ n} f k) - (\sum_{k<m} f k)) < e$
 by (*intro N*) *simp-all*
 also from *True* have $(\sum_{k<Suc\ n} f k) - (\sum_{k<m} f k) = (\sum_{k=m..n} f k)$
 by (*subst sum-diff [symmetric]*) (*simp-all add: sum.last-plus*)
 finally show *?thesis* .
 next
 case *False*
 with e show *?thesis* by *simp-all*
 qed
 then show *?thesis* by (*rule that*)
 qed

lemma *power-sums-if*:

$(\lambda n. (\text{if } n = m \text{ then } (1 :: 'a :: \{\text{ring-1, topological-space}\}) \text{ else } 0) * z ^ n) \text{ sums } z ^ m$
 proof –
 have $(\lambda n. (\text{if } n = m \text{ then } 1 \text{ else } 0) * z ^ n) = (\lambda n. \text{if } n = m \text{ then } z ^ n \text{ else } 0)$
 by (*intro ext*) *auto*
 then show *?thesis*
 by (*simp add: sums-single*)
 qed

lemma

fixes $f :: \text{nat} \Rightarrow \text{real}$

assumes *summable f*
and *inj g*
and *pos: $\bigwedge x. 0 \leq f x$*
shows *summable-reindex: summable (f o g)*
and *suminf-reindex-mono: suminf (f o g) \leq suminf f*
and *suminf-reindex: ($\bigwedge x. x \notin \text{range } g \implies f x = 0$) \implies suminf (f o g) = suminf f*
proof –
from $\langle \text{inj } g \rangle$ **have** [*simp*]: $\bigwedge A. \text{inj-on } g A$
by (*rule subset-inj-on*) *simp*

have *smaller: $\forall n. (\sum i < n. (f \circ g) i) \leq \text{suminf } f$*
proof
fix *n*
have $\forall n' \in (g \text{ ‘ } \{..<n\}$). $n' < \text{Suc } (\text{Max } (g \text{ ‘ } \{..<n\}))$
by (*metis Max-ge finite-imageI finite-lessThan not-le not-less-eq*)
then obtain *m* **where** $n: \bigwedge n'. n' < n \implies g n' < m$
by *blast*

have $(\sum i < n. f (g i)) = \text{sum } f (g \text{ ‘ } \{..<n\})$
by (*simp add: sum.reindex*)
also have $\dots \leq (\sum i < m. f i)$
by (*rule sum-mono2*) (*auto simp add: pos n[rule-format]*)
also have $\dots \leq \text{suminf } f$
using $\langle \text{summable } f \rangle$
by (*rule sum-le-suminf*) (*simp-all add: pos*)
finally show $(\sum i < n. (f \circ g) i) \leq \text{suminf } f$
by *simp*
qed

have *incseq* $(\lambda n. \sum i < n. (f \circ g) i)$
by (*rule incseq-SucI*) (*auto simp add: pos*)
then obtain *L* **where** $L: (\lambda n. \sum i < n. (f \circ g) i) \longrightarrow L$
using *smaller* **by**(*rule incseq-convergent*)
then have $(f \circ g)$ *sums* *L*
by (*simp add: sums-def*)
then show *summable* $(f \circ g)$
by (*auto simp add: sums-iff*)

then have $(\lambda n. \sum i < n. (f \circ g) i) \longrightarrow \text{suminf } (f \circ g)$
by (*rule summable-LIMSEQ*)
then show *le: suminf* $(f \circ g) \leq \text{suminf } f$
by(*rule LIMSEQ-le-const2*)(*blast intro: smaller[rule-format]*)

assume $f: \bigwedge x. x \notin \text{range } g \implies f x = 0$

from $\langle \text{summable } f \rangle$ **have** *suminf* $f \leq \text{suminf } (f \circ g)$
proof (*rule suminf-le-const*)
fix *n*

have $\forall n' \in (g - \{..<n\}). n' < \text{Suc} (\text{Max} (g - \{..<n\}))$
by (*auto intro: Max-ge simp add: finite-vimageI less-Suc-eq-le*)
then obtain m **where** $n: \bigwedge n'. g n' < n \implies n' < m$
by *blast*
have $(\sum i < n. f i) = (\sum i \in \{..<n\} \cap \text{range } g. f i)$
using f **by** (*auto intro: sum.mono-neutral-cong-right*)
also have $\dots = (\sum i \in g - \{..<n\}. (f \circ g) i)$
by (*rule sum.reindex-cong[where l=g](auto)*)
also have $\dots \leq (\sum i < m. (f \circ g) i)$
by (*rule sum-mono2(auto simp add: pos n)*)
also have $\dots \leq \text{suminf } (f \circ g)$
using $\langle \text{summable } (f \circ g) \rangle$ **by** (*rule sum-le-suminf (simp-all add: pos)*)
finally show $\text{sum } f \{..<n\} \leq \text{suminf } (f \circ g)$.
qed
with le **show** $\text{suminf } (f \circ g) = \text{suminf } f$
by (*rule antisym*)
qed

lemma *sums-mono-reindex:*

assumes *subseq: strict-mono g*
and *zero: $\bigwedge n. n \notin \text{range } g \implies f n = 0$*
shows $(\lambda n. f (g n)) \text{ sums } c \longleftrightarrow f \text{ sums } c$
unfolding *sums-def*

proof

assume $lim: (\lambda n. \sum k < n. f k) \longrightarrow c$
have $(\lambda n. \sum k < n. f (g k)) = (\lambda n. \sum k < g n. f k)$

proof

fix $n :: \text{nat}$
from *subseq* **have** $(\sum k < n. f (g k)) = (\sum k \in g \{..<n\}. f k)$
by (*subst sum.reindex (auto intro: strict-mono-imp-inj-on)*)
also from *subseq* **have** $\dots = (\sum k < g n. f k)$
by (*intro sum.mono-neutral-left ballI zero*)
(auto simp: strict-mono-less strict-mono-less-eq)
finally show $(\sum k < n. f (g k)) = (\sum k < g n. f k)$.

qed

also from *LIMSEQ-subseq-LIMSEQ[OF lim subseq]* **have** $\dots \longrightarrow c$
by (*simp only: o-def*)

finally show $(\lambda n. \sum k < n. f (g k)) \longrightarrow c$.

next

assume $lim: (\lambda n. \sum k < n. f (g k)) \longrightarrow c$
define $g\text{-inv } n = (\text{LEAST } m. g m \geq n)$ **for** n
from *filterlim-subseq[OF subseq]* **have** $g\text{-inv-ex: } \exists m. g m \geq n$ **for** n
by (*auto simp: filterlim-at-top eventually-at-top-linorder*)
then have $g\text{-inv: } g (g\text{-inv } n) \geq n$ **for** n
unfolding $g\text{-inv-def}$ **by** (*rule LeastI-ex*)
have $g\text{-inv-least: } m \geq g\text{-inv } n$ **if** $g m \geq n$ **for** m n
using *that* **unfolding** $g\text{-inv-def}$ **by** (*rule Least-le*)
have $g\text{-inv-least': } g m < n$ **if** $m < g\text{-inv } n$ **for** m n
using *that* $g\text{-inv-least[of } n \ m]$ **by** *linarith*


```

have  $(\lambda n. \sum k < n. f k) = (\lambda n. \sum k < g\text{-inv } n. f (g k))$ 
proof
  fix  $n :: nat$ 
  {
    fix  $k$ 
    assume  $k: k \in \{..<n\} - g'\{..<g\text{-inv } n\}$ 
    have  $k \notin \text{range } g$ 
    proof (rule notI, elim imageE)
      fix  $l$ 
      assume  $l: k = g l$ 
      have  $g l < g (g\text{-inv } n)$ 
        by (rule less-le-trans[OF - g-inv]) (use k l in simp-all)
      with subseq have  $l < g\text{-inv } n$ 
        by (simp add: strict-mono-less)
      with k l show False
        by simp
    qed
    then have  $f k = 0$ 
      by (rule zero)
  }
  with g-inv-least' g-inv have  $(\sum k < n. f k) = (\sum k \in g'\{..<g\text{-inv } n\}. f k)$ 
    by (intro sum.mono-neutral-right) auto
  also from subseq have  $\dots = (\sum k < g\text{-inv } n. f (g k))$ 
    using strict-mono-imp-inj-on by (subst sum.reindex) simp-all
  finally show  $(\sum k < n. f k) = (\sum k < g\text{-inv } n. f (g k))$  .
qed
also {
  fix  $K n :: nat$ 
  assume  $g K \leq n$ 
  also have  $n \leq g (g\text{-inv } n)$ 
    by (rule g-inv)
  finally have  $K \leq g\text{-inv } n$ 
    using subseq by (simp add: strict-mono-less-eq)
}
then have filterlim g-inv at-top sequentially
  by (auto simp: filterlim-at-top eventually-at-top-linorder)
with lim have  $(\lambda n. \sum k < g\text{-inv } n. f (g k)) \longrightarrow c$ 
  by (rule filterlim-compose)
finally show  $(\lambda n. \sum k < n. f k) \longrightarrow c$  .
qed

```

lemma *summable-mono-reindex*:

assumes *subseq: strict-mono g*

and *zero: $\bigwedge n. n \notin \text{range } g \implies f n = 0$*

shows *summable $(\lambda n. f (g n)) \iff$ summable f*

using *sums-mono-reindex[of g f, OF assms]* by (simp add: summable-def)

lemma *suminf-mono-reindex*:

fixes $f :: nat \Rightarrow 'a :: \{t2\text{-space, comm-monoid-add}\}$

```

assumes strict-mono  $g \wedge n. n \notin \text{range } g \implies f\ n = 0$ 
shows  $\text{suminf } (\lambda n. f\ (g\ n)) = \text{suminf } f$ 
proof (cases summable f)
  case True
    with sums-mono-reindex [of g f, OF assms]
      and summable-mono-reindex [of g f, OF assms]
    show ?thesis
      by (simp add: sums-iff)
  next
    case False
    then have  $\neg(\exists c. f\ \text{sums } c)$ 
      unfolding summable-def by blast
    then have  $\text{suminf } f = \text{The } (\lambda-. \text{False})$ 
      by (simp add: suminf-def)
    moreover from False have  $\neg \text{summable } (\lambda n. f\ (g\ n))$ 
      using summable-mono-reindex[of g f, OF assms] by simp
    then have  $\neg(\exists c. (\lambda n. f\ (g\ n))\ \text{sums } c)$ 
      unfolding summable-def by blast
    then have  $\text{suminf } (\lambda n. f\ (g\ n)) = \text{The } (\lambda-. \text{False})$ 
      by (simp add: suminf-def)
    ultimately show ?thesis by simp
qed

lemma summable-bounded-partials:
  fixes  $f :: \text{nat} \Rightarrow 'a :: \{\text{real-normed-vector, complete-space}\}$ 
  assumes bound: eventually  $(\lambda x0. \forall a \geq x0. \forall b > a. \text{norm } (\text{sum } f\ \{a <.. b\}) \leq g\ a)$ 
  sequentially
  assumes  $g: g \longrightarrow 0$ 
  shows summable f unfolding summable-iff-convergent'
proof (intro Cauchy-convergent CauchyI', goal-cases)
  case  $(1\ \varepsilon)$ 
  with  $g$  have eventually  $(\lambda x. |g\ x| < \varepsilon)$  sequentially
    by (auto simp: tendsto-iff)
  from eventually-conj[OF this bound] obtain  $x0$  where  $x0$ :
     $\bigwedge x. x \geq x0 \implies |g\ x| < \varepsilon \wedge a\ b. x0 \leq a \implies a < b \implies \text{norm } (\text{sum } f\ \{a <.. b\})$ 
     $\leq g\ a$ 
  unfolding eventually-at-top-linorder by auto
show ?case
proof (intro exI[of - x0] allI impI)
  fix  $m\ n$  assume  $mn: x0 \leq m\ m < n$ 
  have  $\text{dist } (\text{sum } f\ \{..m\})\ (\text{sum } f\ \{..n\}) = \text{norm } (\text{sum } f\ \{..n\} - \text{sum } f\ \{..m\})$ 
    by (simp add: dist-norm norm-minus-commute)
  also have  $\text{sum } f\ \{..n\} - \text{sum } f\ \{..m\} = \text{sum } f\ (\{..n\} - \{..m\})$ 
    using  $mn$  by (intro Groups-Big.sum-diff [symmetric]) auto
  also have  $\{..n\} - \{..m\} = \{m <.. n\}$  using  $mn$  by auto
  also have  $\text{norm } (\text{sum } f\ \{m <.. n\}) \leq g\ m$  using  $mn$  by (intro x0) auto
  also have  $\dots \leq |g\ m|$  by simp
  also have  $\dots < \varepsilon$  using  $mn$  by (intro x0) auto
  finally show  $\text{dist } (\text{sum } f\ \{..m\})\ (\text{sum } f\ \{..n\}) < \varepsilon$  .

```

qed
 qed
 end

111 Differentiation

theory Deriv
 imports Limits
 begin

111.1 Frechet derivative

definition *has-derivative* :: ('a::real-normed-vector \Rightarrow 'b::real-normed-vector) \Rightarrow
 ('a \Rightarrow 'b) \Rightarrow 'a filter \Rightarrow bool (**infix** (*has'-derivative*) 50)
where (*f has-derivative f'*) $F \iff$
 bounded-linear $f' \wedge$
 $((\lambda y. ((f y - f (Lim F (\lambda x. x))) - f' (y - Lim F (\lambda x. x))) /_R norm (y - Lim$
 $F (\lambda x. x))) \longrightarrow 0) F$

Usually the filter F is *at x within s*. (*f has-derivative D*) (*at x within s*) means: D is the derivative of function f at point x within the set s . Where s is used to express left or right sided derivatives. In most cases s is either a variable or *UNIV*.

These are the only cases we'll care about, probably.

lemma *has-derivative-within*: (*f has-derivative f'*) (*at x within s*) \iff
 bounded-linear $f' \wedge ((\lambda y. (1 / norm(y - x)) *_R (f y - (f x + f' (y - x))))$
 $\longrightarrow 0)$ (*at x within s*)
unfolding *has-derivative-def tendsto-iff*
by (*subst eventually-Lim-ident-at*) (*auto simp add: field-simps*)

lemma *has-derivative-eq-rhs*: (*f has-derivative f'*) $F \implies f' = g' \implies$ (*f has-derivative*
 $g')$ F
by *simp*

definition *has-field-derivative* :: ('a::real-normed-field \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a filter \Rightarrow
 bool
 (**infix** (*has'-field'-derivative*) 50)
where (*f has-field-derivative D*) $F \iff$ (*f has-derivative* (*) D) F

lemma *DERIV-cong*: (*f has-field-derivative X*) $F \implies X = Y \implies$ (*f has-field-derivative*
 Y) F
by *simp*

definition *has-vector-derivative* :: (real \Rightarrow 'b::real-normed-vector) \Rightarrow 'b \Rightarrow real
 filter \Rightarrow bool
 (**infix** *has'-vector'-derivative* 50)
where (*f has-vector-derivative f'*) *net* \iff (*f has-derivative* ($\lambda x. x *_R f'$) *net*)

```

lemma has-vector-derivative-eq-rhs:
  (f has-vector-derivative X)  $F \implies X = Y \implies$  (f has-vector-derivative Y)  $F$ 
  by simp

named-theorems derivative-intros structural introduction rules for derivatives
setup <
  let
    val eq-thms = @{thms has-derivative-eq-rhs DERIV-cong has-vector-derivative-eq-rhs}
    fun eq-rule thm = get-first (try (fn eq-thm => eq-thm OF [thm])) eq-thms
  in
    Global-Theory.add-thms-dynamic
      (binding <derivative-eq-intros>,
        fn context =>
          Named-Theorems.get (Context.proof-of context) named-theorems <derivative-intros>
            |> map-filter eq-rule)
  end
  >

```

The following syntax is only used as a legacy syntax.

```

abbreviation (input)
  FDERIV :: ('a::real-normed-vector  $\Rightarrow$  'b::real-normed-vector)  $\Rightarrow$  'a  $\Rightarrow$  ('a  $\Rightarrow$  'b)
   $\Rightarrow$  bool
  ((FDERIV (-)/ (-)/  $\Rightarrow$  (-)) [1000, 1000, 60] 60)
  where FDERIV f x  $\Rightarrow$  f'  $\equiv$  (f has-derivative f') (at x)

```

```

lemma has-derivative-bounded-linear: (f has-derivative f')  $F \implies$  bounded-linear f'
  by (simp add: has-derivative-def)

```

```

lemma has-derivative-linear: (f has-derivative f')  $F \implies$  linear f'
  using bounded-linear.linear[OF has-derivative-bounded-linear] .

```

```

lemma has-derivative-ident[derivative-intros, simp]: (( $\lambda x. x$ ) has-derivative ( $\lambda x. x$ ))  $F$ 
  by (simp add: has-derivative-def)

```

```

lemma has-derivative-id [derivative-intros, simp]: (id has-derivative id)  $F$ 
  by (metis eq-id-iff has-derivative-ident)

```

```

lemma shift-has-derivative-id: ((+) d has-derivative ( $\lambda x. x$ ))  $F$ 
  using has-derivative-def by fastforce

```

```

lemma has-derivative-const[derivative-intros, simp]: (( $\lambda x. c$ ) has-derivative ( $\lambda x. 0$ ))  $F$ 
  by (simp add: has-derivative-def)

```

```

lemma (in bounded-linear) bounded-linear: bounded-linear f ..

```

```

lemma (in bounded-linear) has-derivative:

```

$(g \text{ has-derivative } g') F \implies ((\lambda x. f (g x)) \text{ has-derivative } (\lambda x. f (g' x))) F$
unfolding *has-derivative-def*
by (*auto simp add: bounded-linear-compose [OF bounded-linear] scaleR diff dest: tendsto*)

lemmas *has-derivative-scaleR-right [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-scaleR-right]*

lemmas *has-derivative-scaleR-left [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-scaleR-left]*

lemmas *has-derivative-mult-right [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-mult-right]*

lemmas *has-derivative-mult-left [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-mult-left]*

lemmas *has-derivative-of-real[derivative-intros, simp] = bounded-linear.has-derivative[OF bounded-linear-of-real]*

lemma *has-derivative-add[simp, derivative-intros]:*
assumes *f: (f has-derivative f') F*
and *g: (g has-derivative g') F*
shows $((\lambda x. f x + g x) \text{ has-derivative } (\lambda x. f' x + g' x)) F$
unfolding *has-derivative-def*
proof *safe*
let $?x = \text{Lim } F (\lambda x. x)$
let $?D = \lambda f f' y. ((f y - f ?x) - f' (y - ?x)) /_R \text{norm } (y - ?x)$
have $((\lambda x. ?D f f' x + ?D g g' x) \longrightarrow (0 + 0)) F$
using *f g by (intro tendsto-add) (auto simp: has-derivative-def)*
then show $(?D (\lambda x. f x + g x) (\lambda x. f' x + g' x) \longrightarrow 0) F$
by (*simp add: field-simps scaleR-add-right scaleR-diff-right*)
qed (*blast intro: bounded-linear-add f g has-derivative-bounded-linear*)

lemma *has-derivative-sum[simp, derivative-intros]:*
 $(\bigwedge i. i \in I \implies (f i \text{ has-derivative } f' i) F) \implies$
 $((\lambda x. \sum_{i \in I}. f i x) \text{ has-derivative } (\lambda x. \sum_{i \in I}. f' i x)) F$
by (*induct I rule: infinite-finite-induct*) *simp-all*

lemma *has-derivative-minus[simp, derivative-intros]:*
 $(f \text{ has-derivative } f') F \implies ((\lambda x. - f x) \text{ has-derivative } (\lambda x. - f' x)) F$
using *has-derivative-scaleR-right[of f f' F -1] by simp*

lemma *has-derivative-diff[simp, derivative-intros]:*
 $(f \text{ has-derivative } f') F \implies (g \text{ has-derivative } g') F \implies$
 $((\lambda x. f x - g x) \text{ has-derivative } (\lambda x. f' x - g' x)) F$
by (*simp only: diff-conv-add-uminus has-derivative-add has-derivative-minus*)

lemma *has-derivative-at-within:*

$(f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \longleftrightarrow$
 $(\text{bounded-linear } f' \wedge ((\lambda y. ((f y - f x) - f' (y - x)) /_R \text{norm } (y - x)) \longrightarrow$
 $0) \text{ (at } x \text{ within } s))$

proof (cases at x within $s = \text{bot}$)

case *True*

then show *?thesis*

by (*metis (no-types, lifting) has-derivative-within tendsto-bot*)

next

case *False*

then show *?thesis*

by (*simp add: Lim-ident-at has-derivative-def*)

qed

lemma *has-derivative-iff-norm*:

$(f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \longleftrightarrow$
 $\text{bounded-linear } f' \wedge ((\lambda y. \text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x))$
 $\longrightarrow 0) \text{ (at } x \text{ within } s)$

using *tendsto-norm-zero-iff[of - at x within s , where 'b='b, symmetric]*

by (*simp add: has-derivative-at-within divide-inverse ac-simps*)

lemma *has-derivative-at*:

$(f \text{ has-derivative } D) \text{ (at } x) \longleftrightarrow$

$(\text{bounded-linear } D \wedge (\lambda h. \text{norm } (f (x + h) - f x - D h) / \text{norm } h) -0 \rightarrow 0)$

by (*simp add: has-derivative-iff-norm LIM-offset-zero-iff*)

lemma *field-has-derivative-at*:

fixes $x :: 'a::\text{real-normed-field}$

shows $(f \text{ has-derivative } (*) D) \text{ (at } x) \longleftrightarrow (\lambda h. (f (x + h) - f x) / h) -0 \rightarrow D$
 (**is** *?lhs = ?rhs*)

proof –

have *?lhs* = $(\lambda h. \text{norm } (f (x + h) - f x - D * h) / \text{norm } h) -0 \rightarrow 0$

by (*simp add: bounded-linear-mult-right has-derivative-at*)

also have $\dots = (\lambda y. \text{norm } ((f (x + y) - f x - D * y) / y)) -0 \rightarrow 0$

by (*simp cong: LIM-cong flip: nonzero-norm-divide*)

also have $\dots = (\lambda y. \text{norm } ((f (x + y) - f x) / y - D / y * y)) -0 \rightarrow 0$

by (*simp only: diff-divide-distrib times-divide-eq-left [symmetric]*)

also have $\dots = \text{?rhs}$

by (*simp add: tendsto-norm-zero-iff LIM-zero-iff cong: LIM-cong*)

finally show *?thesis* .

qed

lemma *has-derivative-iff-Ex*:

$(f \text{ has-derivative } f') \text{ (at } x) \longleftrightarrow$

$\text{bounded-linear } f' \wedge (\exists e. (\forall h. f (x+h) = f x + f' h + e h) \wedge ((\lambda h. \text{norm } (e h)$
 $/ \text{norm } h) \longrightarrow 0) \text{ (at } 0))$

unfolding *has-derivative-at by force*

lemma *has-derivative-at-within-iff-Ex*:

assumes $x \in S$ *open* S

shows $(f \text{ has-derivative } f') \text{ (at } x \text{ within } S) \longleftrightarrow$
 $\text{bounded-linear } f' \wedge (\exists e. (\forall h. x+h \in S \longrightarrow f(x+h) = f x + f' h + e h) \wedge$
 $((\lambda h. \text{norm } (e h) / \text{norm } h) \longrightarrow 0) \text{ (at } 0))$
(is ?lhs = ?rhs)

proof safe

show *bounded-linear* f'

if $(f \text{ has-derivative } f') \text{ (at } x \text{ within } S)$

using *has-derivative-bounded-linear* **that by blast**

show $\exists e. (\forall h. x + h \in S \longrightarrow f(x+h) = f x + f' h + e h) \wedge (\lambda h. \text{norm } (e h)$
 $/ \text{norm } h) - 0 \rightarrow 0$

if $(f \text{ has-derivative } f') \text{ (at } x \text{ within } S)$

by (*metis (full-types) assms that has-derivative-iff-Ex at-within-open*)

show $(f \text{ has-derivative } f') \text{ (at } x \text{ within } S)$

if *bounded-linear* f'

and *eq [rule-format]*: $\forall h. x + h \in S \longrightarrow f(x+h) = f x + f' h + e h$

and $0: (\lambda h. \text{norm } (e (h::'a)::'b) / \text{norm } h) - 0 \rightarrow 0$

for e

proof –

have $1: f y - f x = f' (y-x) + e (y-x)$ **if** $y \in S$ **for** y

using *eq [of y-x] that by simp*

have $2: ((\lambda y. \text{norm } (e (y-x)) / \text{norm } (y-x)) \longrightarrow 0) \text{ (at } x \text{ within } S)$

by (*simp add: 0 assms tendsto-offset-zero-iff*)

have $((\lambda y. \text{norm } (f y - f x - f' (y-x)) / \text{norm } (y-x)) \longrightarrow 0) \text{ (at } x$
 $\text{within } S)$

by (*simp add: Lim-cong-within 1 2*)

then show *?thesis*

by (*simp add: has-derivative-iff-norm ‹bounded-linear f'›*)

qed

qed

lemma *has-derivativeI*:

bounded-linear $f' \implies$

$((\lambda y. ((f y - f x) - f' (y-x)) /_R \text{norm } (y-x)) \longrightarrow 0) \text{ (at } x \text{ within } s) \implies$
 $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$

by (*simp add: has-derivative-at-within*)

lemma *has-derivativeI-sandwich*:

assumes $e: 0 < e$

and *bounded: bounded-linear* f'

and *sandwich*: $(\bigwedge y. y \in s \implies y \neq x \implies \text{dist } y x < e \implies$
 $\text{norm } ((f y - f x) - f' (y-x)) / \text{norm } (y-x) \leq H y)$

and $(H \longrightarrow 0) \text{ (at } x \text{ within } s)$

shows $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$

unfolding *has-derivative-iff-norm*

proof safe

show $((\lambda y. \text{norm } (f y - f x - f' (y-x)) / \text{norm } (y-x)) \longrightarrow 0) \text{ (at } x \text{ within } s)$

proof (*rule tendsto-sandwich[where f= $\lambda x. 0$]*)

show $(H \longrightarrow 0) \text{ (at } x \text{ within } s)$ **by fact**

show *eventually* $(\lambda n. \text{norm } (f n - f x - f' (n - x)) / \text{norm } (n - x) \leq H n)$
(at x within s)
unfolding *eventually-at using e sandwich by auto*
qed *(auto simp: le-divide-eq)*
qed fact

lemma *has-derivative-subset*:
 $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \implies t \subseteq s \implies (f \text{ has-derivative } f') \text{ (at } x \text{ within } t)$
by *(auto simp add: has-derivative-iff-norm intro: tendsto-within-subset)*

lemma *has-derivative-within-singleton-iff*:
 $(f \text{ has-derivative } g) \text{ (at } x \text{ within } \{x\}) \iff \text{bounded-linear } g$
by *(auto intro!: has-derivativeI-sandwich[where e=1] has-derivative-bounded-linear)*

111.1.1 Limit transformation for derivatives

lemma *has-derivative-transform-within*:
assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
and $0 < d$
and $x \in s$
and $\bigwedge x'. \llbracket x' \in s; \text{dist } x' x < d \rrbracket \implies f x' = g x'$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
using *assms*
unfolding *has-derivative-within*
by *(force simp add: intro: Lim-transform-within)*

lemma *has-derivative-transform-within-open*:
assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } t)$
and *open s*
and $x \in s$
and $\bigwedge x. x \in s \implies f x = g x$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } t)$
using *assms* **unfolding** *has-derivative-within*
by *(force simp add: intro: Lim-transform-within-open)*

lemma *has-derivative-transform*:
assumes $x \in s \bigwedge x. x \in s \implies g x = f x$
assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
using *assms*
by *(intro has-derivative-transform-within[OF - zero-less-one, where g=g]) auto*

lemma *has-derivative-transform-eventually*:
assumes $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
 $(\forall_F x' \text{ in at } x \text{ within } s. f x' = g x')$
assumes $f x = g x \ x \in s$
shows $(g \text{ has-derivative } f') \text{ (at } x \text{ within } s)$
using *assms*

proof –

from *assms(2,3)* **obtain** d **where** $d > 0 \wedge x'. x' \in s \implies \text{dist } x' x < d \implies f x' = g x'$

by (*force simp: eventually-at*)

from *has-derivative-transform-within[OF assms(1) this(1) assms(4) this(2)]*

show *?thesis* .

qed

lemma *has-field-derivative-transform-within*:

assumes (*f has-field-derivative f'*) (*at a within S*)

and $0 < d$

and $a \in S$

and $\bigwedge x. \llbracket x \in S; \text{dist } x a < d \rrbracket \implies f x = g x$

shows (*g has-field-derivative f'*) (*at a within S*)

using *assms unfolding has-field-derivative-def*

by (*metis has-derivative-transform-within*)

lemma *has-field-derivative-transform-within-open*:

assumes (*f has-field-derivative f'*) (*at a*)

and *open S a ∈ S*

and $\bigwedge x. x \in S \implies f x = g x$

shows (*g has-field-derivative f'*) (*at a*)

using *assms unfolding has-field-derivative-def*

by (*metis has-derivative-transform-within-open*)

111.2 Continuity

lemma *has-derivative-continuous*:

assumes *f: (f has-derivative f') (at x within s)*

shows *continuous (at x within s) f*

proof –

from *f* **interpret** F : *bounded-linear f'*

by (*rule has-derivative-bounded-linear*)

note $F.\text{tendsto}[\text{tendsto-intros}]$

let $?L = \lambda f. (f \longrightarrow 0)$ (*at x within s*)

have $?L (\lambda y. \text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x))$

using *f unfolding has-derivative-iff-norm* **by** *blast*

then have $?L (\lambda y. \text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x) * \text{norm } (y - x))$ (*is ?m*)

by (*rule tendsto-mult-zero (auto intro!: tendsto-eq-intros)*)

also have $?m \longleftrightarrow ?L (\lambda y. \text{norm } ((f y - f x) - f' (y - x)))$

by (*intro filterlim-cong (simp-all add: eventually-at-filter)*)

finally have $?L (\lambda y. (f y - f x) - f' (y - x))$

by (*rule tendsto-norm-zero-cancel*)

then have $?L (\lambda y. ((f y - f x) - f' (y - x)) + f' (y - x))$

by (*rule tendsto-eq-intros (auto intro!: tendsto-eq-intros simp: F.zero)*)

then have $?L (\lambda y. f y - f x)$

by *simp*

from *tendsto-add[OF this tendsto-const, of f x]* **show** *?thesis*

by (*simp add: continuous-within*)
qed

111.3 Composition

lemma *tendsto-at-iff-tendsto-nhds-within*:

$f x = y \implies (f \longrightarrow y) \text{ (at } x \text{ within } s) \longleftrightarrow (f \longrightarrow y) \text{ (inf (nhds } x) \text{ (principal } s))$

unfolding *tendsto-def eventually-inf-principal eventually-at-filter*
by (*intro ext all-cong imp-cong (auto elim!: eventually-mono)*)

lemma *has-derivative-in-compose*:

assumes f : (*f has-derivative f'*) (at x within s)

and g : (*g has-derivative g'*) (at $(f x)$ within $(f's)$)

shows $((\lambda x. g (f x)) \text{ has-derivative } (\lambda x. g' (f' x))) \text{ (at } x \text{ within } s)$

proof –

from f **interpret** F : *bounded-linear f'*

by (*rule has-derivative-bounded-linear*)

from g **interpret** G : *bounded-linear g'*

by (*rule has-derivative-bounded-linear*)

from F *bounded* **obtain** kF **where** kF : $\bigwedge x. \text{norm } (f' x) \leq \text{norm } x * kF$

by *fast*

from G *bounded* **obtain** kG **where** kG : $\bigwedge x. \text{norm } (g' x) \leq \text{norm } x * kG$

by *fast*

note $G.tendsto$ [*tendsto-intros*]

let $?L = \lambda f. (f \longrightarrow 0) \text{ (at } x \text{ within } s)$

let $?D = \lambda f f' x y. (f y - f x) - f' (y - x)$

let $?N = \lambda f f' x y. \text{norm } (?D f f' x y) / \text{norm } (y - x)$

let $?gf = \lambda x. g (f x)$ **and** $?gf' = \lambda x. g' (f' x)$

define Nf **where** $Nf = ?N f f' x$

define Ng **where** [*abs-def*]: $Ng y = ?N g g' (f x) (f y)$ **for** y

show *?thesis*

proof (*rule has-derivativeI-sandwich[of 1]*)

show *bounded-linear* $(\lambda x. g' (f' x))$

using $f g$ **by** (*blast intro: bounded-linear-compose has-derivative-bounded-linear*)

next

fix $y :: 'a$

assume *neq*: $y \neq x$

have $?N ?gf ?gf' x y = \text{norm } (g' (?D f f' x y) + ?D g g' (f x) (f y)) / \text{norm } (y - x)$

by (*simp add: G.diff G.add field-simps*)

also have $\dots \leq \text{norm } (g' (?D f f' x y)) / \text{norm } (y - x) + Ng y * (\text{norm } (f y - f x) / \text{norm } (y - x))$

by (*simp add: add-divide-distrib[symmetric] divide-right-mono norm-triangle-ineq G.zero Ng-def*)

also have $\dots \leq Nf y * kG + Ng y * (Nf y + kF)$

proof (*intro add-mono mult-left-mono*)

```

have norm (f y - f x) = norm (?D f f' x y + f' (y - x))
  by simp
also have ... ≤ norm (?D f f' x y) + norm (f' (y - x))
  by (rule norm-triangle-ineq)
also have ... ≤ norm (?D f f' x y) + norm (y - x) * kF
  using kF by (intro add-mono) simp
finally show norm (f y - f x) / norm (y - x) ≤ Nf y + kF
  by (simp add: neq Nf-def field-simps)
qed (use kG in ⟨simp-all add: Ng-def Nf-def neq zero-le-divide-iff field-simps⟩)
finally show ?N ?gf ?gf' x y ≤ Nf y * kG + Ng y * (Nf y + kF) .
next
have [tendsto-intros]: ?L Nf
  using f unfolding has-derivative-iff-norm Nf-def ..
from f have (f ⟶ f x) (at x within s)
  by (blast intro: has-derivative-continuous continuous-within[THEN iffD1])
then have f': LIM x at x within s. f x :> inf (nhds (f x)) (principal (f's))
  unfolding filterlim-def
  by (simp add: eventually-filtermap eventually-at-filter le-principal)

have ((?N g g' (f x)) ⟶ 0) (at (f x) within f's)
  using g unfolding has-derivative-iff-norm ..
then have g': ((?N g g' (f x)) ⟶ 0) (inf (nhds (f x)) (principal (f's)))
  by (rule tendsto-at-iff-tendsto-nhds-within[THEN iffD1, rotated]) simp

have [tendsto-intros]: ?L Ng
  unfolding Ng-def by (rule filterlim-compose[OF g' f'])
show ((λy. Nf y * kG + Ng y * (Nf y + kF)) ⟶ 0) (at x within s)
  by (intro tendsto-eq-intros) auto
qed simp
qed

lemma has-derivative-compose:
  (f has-derivative f') (at x within s) ⟹ (g has-derivative g') (at (f x)) ⟹
  ((λx. g (f x)) has-derivative (λx. g' (f' x))) (at x within s)
  by (blast intro: has-derivative-in-compose has-derivative-subset)

lemma has-derivative-in-compose2:
  assumes ∧x. x ∈ t ⟹ (g has-derivative g' x) (at x within t)
  assumes f' s ⊆ t x ∈ s
  assumes (f has-derivative f') (at x within s)
  shows ((λx. g (f x)) has-derivative (λy. g' (f x) (f' y))) (at x within s)
  using assms
  by (auto intro: has-derivative-subset intro!: has-derivative-in-compose[of f f' x s
  g])

lemma (in bounded-bilinear) FDERIV:
  assumes f: (f has-derivative f') (at x within s) and g: (g has-derivative g') (at x
  within s)
  shows ((λx. f x ** g x) has-derivative (λh. f x ** g' h + f' h ** g x)) (at x

```

within s)

proof –

from *bounded-linear.bounded* [*OF has-derivative-bounded-linear* [*OF f*]]
obtain KF **where** *norm-F*: $\bigwedge x. \text{norm } (f' x) \leq \text{norm } x * KF$ **by** *fast*

from *pos-bounded* **obtain** K

where K : $0 < K$ **and** *norm-prod*: $\bigwedge a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$

by *fast*

let $?D = \lambda f f' y. f y - f x - f' (y - x)$

let $?N = \lambda f f' y. \text{norm } (?D f f' y) / \text{norm } (y - x)$

define Ng **where** $Ng = ?N g g'$

define Nf **where** $Nf = ?N f f'$

let $?fun1 = \lambda y. \text{norm } (f y ** g y - f x ** g x - (f x ** g' (y - x) + f' (y - x) ** g x)) / \text{norm } (y - x)$

let $?fun2 = \lambda y. \text{norm } (f x) * Ng y * K + Nf y * \text{norm } (g y) * K + KF * \text{norm } (g y - g x) * K$

let $?F = \text{at } x \text{ within } s$

show *?thesis*

proof (*rule has-derivativeI-sandwich*[*of 1*])

show *bounded-linear* ($\lambda h. f x ** g' h + f' h ** g x$)

by (*intro bounded-linear-add*

bounded-linear-compose [*OF bounded-linear-right*] *bounded-linear-compose*

[*OF bounded-linear-left*]

has-derivative-bounded-linear [*OF g*] *has-derivative-bounded-linear* [*OF f*])

next

from g **have** ($g \longrightarrow g x$) $?F$

by (*intro continuous-within*[*THEN iffD1*] *has-derivative-continuous*)

moreover from $f g$ **have** ($Nf \longrightarrow 0$) $?F$ ($Ng \longrightarrow 0$) $?F$

by (*simp-all add: has-derivative-iff-norm Ng-def Nf-def*)

ultimately have ($?fun2 \longrightarrow \text{norm } (f x) * 0 * K + 0 * \text{norm } (g x) * K + KF * \text{norm } (0::'b) * K$) $?F$

by (*intro tendsto-intros*) (*simp-all add: LIM-zero-iff*)

then show ($?fun2 \longrightarrow 0$) $?F$

by *simp*

next

fix $y :: 'd$

assume $y \neq x$

have $?fun1 y =$

$\text{norm } (f x ** ?D g g' y + ?D f f' y ** g y + f' (y - x) ** (g y - g x)) / \text{norm } (y - x)$

by (*simp add: diff-left diff-right add-left add-right field-simps*)

also have $\dots \leq (\text{norm } (f x) * \text{norm } (?D g g' y) * K + \text{norm } (?D f f' y) * \text{norm } (g y) * K +$

$\text{norm } (y - x) * KF * \text{norm } (g y - g x) * K) / \text{norm } (y - x)$

by (*intro divide-right-mono mult-mono'*

order-trans [*OF norm-triangle-ineq add-mono*])

```

      order-trans [OF norm-prod mult-right-mono]
      mult-nonneg-nonneg order-refl norm-ge-zero norm-F
      K [THEN order-less-imp-le])
  also have ... = ?fun2 y
    by (simp add: add-divide-distrib Ng-def Nf-def)
  finally show ?fun1 y ≤ ?fun2 y .
qed simp
qed

lemmas has-derivative-mult[simp, derivative-intros] = bounded-bilinear.FDERIV[OF
bounded-bilinear-mult]
lemmas has-derivative-scaleR[simp, derivative-intros] = bounded-bilinear.FDERIV[OF
bounded-bilinear-scaleR]

lemma has-derivative-prod[simp, derivative-intros]:
  fixes f :: 'i ⇒ 'a::real-normed-vector ⇒ 'b::real-normed-field
  shows (∧i. i ∈ I ⇒ (f i has-derivative f' i) (at x within S)) ⇒
    ((λx. ∏i∈I. f i x) has-derivative (λy. ∑i∈I. f' i y * (∏j∈I - {i}. f j x)))
  (at x within S)
proof (induct I rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case (insert i I)
  let ?P = λy. f i x * (∑i∈I. f' i y * (∏j∈I - {i}. f j x)) + (f' i y) * (∏i∈I. f
i x)
  have ((λx. f i x * (∏i∈I. f i x)) has-derivative ?P) (at x within S)
    using insert by (intro has-derivative-mult) auto
  also have ?P = (λy. ∑i'∈insert i I. f' i' y * (∏j∈insert i I - {i'}. f j x))
    using insert(1,2)
  by (auto simp add: sum-distrib-left insert-Diff-if intro!: ext sum.cong)
  finally show ?case
    using insert by simp
qed

lemma has-derivative-power[simp, derivative-intros]:
  fixes f :: 'a :: real-normed-vector ⇒ 'b :: real-normed-field
  assumes f: (f has-derivative f') (at x within S)
  shows ((λx. f x^n) has-derivative (λy. of-nat n * f' y * f x^(n - 1))) (at x within
S)
  using has-derivative-prod[OF f, of {..< n}] by (simp add: prod-constant ac-simps)

lemma has-derivative-inverse':
  fixes x :: 'a::real-normed-div-algebra
  assumes x: x ≠ 0
  shows (inverse has-derivative (λh. - (inverse x * h * inverse x))) (at x within

```

S)
 (is (- has-derivative ?f) -)
proof (rule has-derivativeI-sandwich)
 show bounded-linear $(\lambda h. - (inverse\ x * h * inverse\ x))$
 by (simp add: bounded-linear-minus bounded-linear-mult-const bounded-linear-mult-right)
 show $0 < norm\ x$ using x by simp
 have $(inverse \longrightarrow inverse\ x)$ (at x within S)
 using tendsto-inverse tendsto-ident-at x by auto
 then show $((\lambda y. norm\ (inverse\ y - inverse\ x) * norm\ (inverse\ x)) \longrightarrow 0)$ (at x within S)
 by (simp add: LIM-zero-iff tendsto-mult-left-zero tendsto-norm-zero)
next
 fix $y :: 'a$
 assume $h: y \neq x$ dist $y\ x < norm\ x$
 then have $y \neq 0$ by auto
 have $norm\ (inverse\ y - inverse\ x - ?f\ (y - x)) / norm\ (y - x)$
 $= norm\ (- (inverse\ y * (y - x) * inverse\ x - inverse\ x * (y - x) * inverse\ x)) /$
 $norm\ (y - x)$
 by (simp add: $\langle y \neq 0 \rangle$ inverse-diff-inverse x)
 also have $\dots = norm\ ((inverse\ y - inverse\ x) * (y - x) * inverse\ x) / norm\ (y - x)$
 by (simp add: left-diff-distrib norm-minus-commute)
 also have $\dots \leq norm\ (inverse\ y - inverse\ x) * norm\ (y - x) * norm\ (inverse\ x) / norm\ (y - x)$
 by (simp add: norm-mult)
 also have $\dots = norm\ (inverse\ y - inverse\ x) * norm\ (inverse\ x)$
 by simp
 finally show $norm\ (inverse\ y - inverse\ x - ?f\ (y - x)) / norm\ (y - x) \leq norm\ (inverse\ y - inverse\ x) * norm\ (inverse\ x)$.
qed

lemma has-derivative-inverse[simp, derivative-intros]:

fixes $f :: - \Rightarrow 'a::real-normed-div-algebra$
 assumes $x: f\ x \neq 0$
 and $f: (f\ has-derivative\ f')$ (at x within S)
 shows $((\lambda x. inverse\ (f\ x))\ has-derivative\ (\lambda h. - (inverse\ (f\ x) * f'\ h * inverse\ (f\ x))))$
 (at x within S)
 using has-derivative-compose[OF $f\ has-derivative-inverse'$, OF x].

lemma has-derivative-divide[simp, derivative-intros]:

fixes $f :: - \Rightarrow 'a::real-normed-div-algebra$
 assumes $f: (f\ has-derivative\ f')$ (at x within S)
 and $g: (g\ has-derivative\ g')$ (at x within S)
 assumes $x: g\ x \neq 0$
 shows $((\lambda x. f\ x / g\ x)\ has-derivative\ (\lambda h. - f\ x * (inverse\ (g\ x) * g'\ h * inverse\ (g\ x)) + f'\ h / g\ x))$ (at x within S)

using *has-derivative-mult*[*OF f has-derivative-inverse*[*OF x g*]]
by (*simp add: field-simps*)

lemma *has-derivative-power-int'*:

fixes $x :: 'a::\text{real-normed-field}$

assumes $x: x \neq 0$

shows $((\lambda x. \text{power-int } x \ n) \text{ has-derivative } (\lambda y. y * (\text{of-int } n * \text{power-int } x \ (n - 1)))) \text{ (at } x \text{ within } S)$

proof (*cases n rule: int-cases4*)

case (*nonneg n*)

thus *?thesis using x*

by (*cases n = 0*) (*auto intro!: derivative-eq-intros simp: field-simps power-int-diff fun-eq-iff*

simp flip: power-Suc)

next

case (*neg n*)

thus *?thesis using x*

by (*auto intro!: derivative-eq-intros simp: field-simps power-int-diff power-int-minus simp flip: power-Suc power-Suc2 power-add*)

qed

lemma *has-derivative-power-int*[*simp, derivative-intros*]:

fixes $f :: - \Rightarrow 'a::\text{real-normed-field}$

assumes $x: f \ x \neq 0$

and $f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } S)$

shows $((\lambda x. \text{power-int } (f \ x) \ n) \text{ has-derivative } (\lambda h. f' \ h * (\text{of-int } n * \text{power-int } (f \ x) \ (n - 1)))) \text{ (at } x \text{ within } S)$

using *has-derivative-compose*[*OF f has-derivative-power-int', OF x*].

Conventional form requires mult-AC laws. Types real and complex only.

lemma *has-derivative-divide'*[*derivative-intros*]:

fixes $f :: - \Rightarrow 'a::\text{real-normed-field}$

assumes $f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } S)$

and $g: (g \text{ has-derivative } g') \text{ (at } x \text{ within } S)$

and $x: g \ x \neq 0$

shows $((\lambda x. f \ x / g \ x) \text{ has-derivative } (\lambda h. (f' \ h * g \ x - f \ x * g' \ h) / (g \ x * g \ x))) \text{ (at } x \text{ within } S)$

proof –

have $f' \ h / g \ x - f \ x * (\text{inverse } (g \ x) * g' \ h * \text{inverse } (g \ x)) =$
 $(f' \ h * g \ x - f \ x * g' \ h) / (g \ x * g \ x)$ **for** h

by (*simp add: field-simps x*)

then show *?thesis*

using *has-derivative-divide* [*OF f g*] x

by *simp*

qed

111.4 Uniqueness

This can not generally shown for (*has-derivative*), as we need to approach the point from all directions. There is a proof in *Analysis* for *euclidean-space*.

lemma *has-derivative-at2*: $(f \text{ has-derivative } f') (at\ x) \longleftrightarrow$
 $\text{bounded-linear } f' \wedge ((\lambda y. (1 / (\text{norm}(y - x))) *_{\mathbb{R}} (f\ y - (f\ x + f' (y - x))))$
 $\longrightarrow 0) (at\ x)$
using *has-derivative-within* [*of f f' x UNIV*]
by *simp*

lemma *has-derivative-zero-unique*:

assumes $((\lambda x. 0) \text{ has-derivative } F) (at\ x)$
shows $F = (\lambda h. 0)$

proof –

interpret F : *bounded-linear F*

using *assms* **by** (*rule has-derivative-bounded-linear*)

let $?r = \lambda h. \text{norm } (F\ h) / \text{norm } h$

have $*$: $?r - 0 \rightarrow 0$

using *assms* **unfolding** *has-derivative-at* **by** *simp*

show $F = (\lambda h. 0)$

proof

show $F\ h = 0$ **for** h

proof (*rule ccontr*)

assume $*$: $\neg ?thesis$

then have h : $h \neq 0$

by (*auto simp add: F.zero*)

with $*$ **have** $0 < ?r\ h$

by *simp*

from *LIM-D* [*OF * this*] **obtain** S

where S : $0 < S$ **and** r : $\bigwedge x. x \neq 0 \implies \text{norm } x < S \implies ?r\ x < ?r\ h$

by *auto*

from *dense* [*OF S*] **obtain** t **where** t : $0 < t \wedge t < S$..

let $?x = \text{scaleR } (t / \text{norm } h) h$

have $?x \neq 0$ **and** $\text{norm } ?x < S$

using $t\ h$ **by** *simp-all*

then have $?r\ ?x < ?r\ h$

by (*rule r*)

then show *False*

using $t\ h$ **by** (*simp add: F.scaleR*)

qed

qed

qed

lemma *has-derivative-unique*:

assumes $(f \text{ has-derivative } F) (at\ x)$

and $(f \text{ has-derivative } F') (at\ x)$

shows $F = F'$

proof –

have $((\lambda x. 0) \text{ has-derivative } (\lambda h. F\ h - F'\ h)) (at\ x)$

using *has-derivative-diff* [*OF assms*] **by** *simp*
then have $(\lambda h. F h - F' h) = (\lambda h. 0)$
by (*rule has-derivative-zero-unique*)
then show $F = F'$
unfolding *fun-eq-iff right-minus-eq* .
qed

lemma *has-derivative-Uniq*: $\exists \leq_1 F. (f \text{ has-derivative } F) (at\ x)$
by (*simp add: Uniq-def has-derivative-unique*)

111.5 Differentiability predicate

definition *differentiable* :: $('a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector}) \Rightarrow 'a$
filter $\Rightarrow \text{bool}$

(**infix** *differentiable* 50)

where *f differentiable* $F \longleftrightarrow (\exists D. (f \text{ has-derivative } D) F)$

lemma *differentiable-subset*:

f differentiable (at x within s) $\implies t \subseteq s \implies f \text{ differentiable (at x within t)}$

unfolding *differentiable-def* **by** (*blast intro: has-derivative-subset*)

lemmas *differentiable-within-subset* = *differentiable-subset*

lemma *differentiable-ident* [*simp, derivative-intros*]: $(\lambda x. x)$ *differentiable* F

unfolding *differentiable-def* **by** (*blast intro: has-derivative-ident*)

lemma *differentiable-const* [*simp, derivative-intros*]: $(\lambda z. a)$ *differentiable* F

unfolding *differentiable-def* **by** (*blast intro: has-derivative-const*)

lemma *differentiable-in-compose*:

f differentiable (at (g x) within (g's)) $\implies g \text{ differentiable (at x within s) $\implies$$

$(\lambda x. f (g x)) \text{ differentiable (at x within s)}$

unfolding *differentiable-def* **by** (*blast intro: has-derivative-in-compose*)

lemma *differentiable-compose*:

f differentiable (at (g x)) $\implies g \text{ differentiable (at x within s) $\implies$$

$(\lambda x. f (g x)) \text{ differentiable (at x within s)}$

by (*blast intro: differentiable-in-compose differentiable-subset*)

lemma *differentiable-add* [*simp, derivative-intros*]:

f differentiable $F \implies g \text{ differentiable } F \implies (\lambda x. f x + g x)$ *differentiable* F

unfolding *differentiable-def* **by** (*blast intro: has-derivative-add*)

lemma *differentiable-sum* [*simp, derivative-intros*]:

assumes *finite* $s \forall a \in s. (f a)$ *differentiable* *net*

shows $(\lambda x. \text{sum } (\lambda a. f a x) s)$ *differentiable* *net*

proof –

from *bchoice* [*OF assms*](2) [*unfolded differentiable-def*]

show *?thesis*

by (auto intro!: has-derivative-sum simp: differentiable-def)
qed

lemma *differentiable-minus* [simp, derivative-intros]:
 f differentiable $F \implies (\lambda x. - f x)$ differentiable F
unfolding *differentiable-def* **by** (blast intro: has-derivative-minus)

lemma *differentiable-diff* [simp, derivative-intros]:
 f differentiable $F \implies g$ differentiable $F \implies (\lambda x. f x - g x)$ differentiable F
unfolding *differentiable-def* **by** (blast intro: has-derivative-diff)

lemma *differentiable-mult* [simp, derivative-intros]:
fixes $f g :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-algebra}$
shows f differentiable (at x within s) $\implies g$ differentiable (at x within s) \implies
 $(\lambda x. f x * g x)$ differentiable (at x within s)
unfolding *differentiable-def* **by** (blast intro: has-derivative-mult)

lemma *differentiable-cmult-left-iff* [simp]:
fixes $c :: 'a::\text{real-normed-field}$
shows $(\lambda t. c * q t)$ differentiable at $t \iff c = 0 \vee (\lambda t. q t)$ differentiable at t
(is ?lhs = ?rhs)

proof

assume L : ?lhs
{ assume $c \neq 0$
 then have q differentiable at t
 using *differentiable-mult* [OF *differentiable-const* L , of concl: $1/c$] **by** auto
} **then show** ?rhs
 by auto

qed auto

lemma *differentiable-cmult-right-iff* [simp]:
fixes $c :: 'a::\text{real-normed-field}$
shows $(\lambda t. q t * c)$ differentiable at $t \iff c = 0 \vee (\lambda t. q t)$ differentiable at t
(is ?lhs = ?rhs)
by (*simp add: mult.commute flip: differentiable-cmult-left-iff*)

lemma *differentiable-inverse* [simp, derivative-intros]:
fixes $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-field}$
shows f differentiable (at x within s) $\implies f x \neq 0 \implies$
 $(\lambda x. \text{inverse } (f x))$ differentiable (at x within s)
unfolding *differentiable-def* **by** (blast intro: has-derivative-inverse)

lemma *differentiable-divide* [simp, derivative-intros]:
fixes $f g :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-field}$
shows f differentiable (at x within s) $\implies g$ differentiable (at x within s) \implies
 $g x \neq 0 \implies (\lambda x. f x / g x)$ differentiable (at x within s)
unfolding *divide-inverse* **by** *simp*

lemma *differentiable-power* [simp, derivative-intros]:

fixes $f g :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-field}$
shows f differentiable (at x within s) $\implies (\lambda x. f x \hat{=} n)$ differentiable (at x within s)

unfolding differentiable-def **by** (blast intro: has-derivative-power)

lemma differentiable-power-int [simp, derivative-intros]:

fixes $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-field}$

shows f differentiable (at x within s) $\implies f x \neq 0 \implies$
 $(\lambda x. \text{power-int } (f x) n)$ differentiable (at x within s)

unfolding differentiable-def **by** (blast intro: has-derivative-power-int)

lemma differentiable-scaleR [simp, derivative-intros]:

f differentiable (at x within s) $\implies g$ differentiable (at x within s) \implies
 $(\lambda x. f x *_{\mathbb{R}} g x)$ differentiable (at x within s)

unfolding differentiable-def **by** (blast intro: has-derivative-scaleR)

lemma has-derivative-imp-has-field-derivative:

$(f \text{ has-derivative } D) F \implies (\bigwedge x. x * D' = D x) \implies (f \text{ has-field-derivative } D') F$

unfolding has-field-derivative-def

by (rule has-derivative-eq-rhs[of $f D$]) (simp-all add: fun-eq-iff mult.commute)

lemma has-field-derivative-imp-has-derivative:

$(f \text{ has-field-derivative } D) F \implies (f \text{ has-derivative } (*) D) F$

by (simp add: has-field-derivative-def)

lemma DERIV-subset:

$(f \text{ has-field-derivative } f') \text{ (at } x \text{ within } s) \implies t \subseteq s \implies$
 $(f \text{ has-field-derivative } f') \text{ (at } x \text{ within } t)$

by (simp add: has-field-derivative-def has-derivative-subset)

lemma has-field-derivative-at-within:

$(f \text{ has-field-derivative } f') \text{ (at } x) \implies (f \text{ has-field-derivative } f') \text{ (at } x \text{ within } s)$

using DERIV-subset **by** blast

abbreviation (input)

$DERIV :: ('a::\text{real-normed-field} \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$

$((DERIV (-)/ (-)/ :> (-)) [1000, 1000, 60] 60)$

where $DERIV f x :> D \equiv (f \text{ has-field-derivative } D) \text{ (at } x)$

abbreviation has-real-derivative :: $(\text{real} \Rightarrow \text{real}) \Rightarrow \text{real} \Rightarrow \text{real filter} \Rightarrow \text{bool}$

$(\text{infix } (\text{has}'\text{-real}'\text{-derivative}) 50)$

where $(f \text{ has-real-derivative } D) F \equiv (f \text{ has-field-derivative } D) F$

lemma real-differentiable-def:

f differentiable at x within $s \iff (\exists D. (f \text{ has-real-derivative } D) \text{ (at } x \text{ within } s))$

proof safe

assume f differentiable at x within s

then obtain f' **where** $*$: $(f \text{ has-derivative } f') \text{ (at } x \text{ within } s)$

unfolding differentiable-def **by** auto

then obtain c **where** $f' = ((*) c)$
by (*metis real-bounded-linear has-derivative-bounded-linear mult.commute fun-eq-iff*)
with $*$ **show** $\exists D. (f \text{ has-real-derivative } D) \text{ (at } x \text{ within } s)$
unfolding *has-field-derivative-def* **by** *auto*
qed (*auto simp: differentiable-def has-field-derivative-def*)

lemma *real-differentiableE* [*elim?*]:
assumes $f: f \text{ differentiable (at } x \text{ within } s)$
obtains df **where** ($f \text{ has-real-derivative } df$) (at x within s)
using *assms* **by** (*auto simp: real-differentiable-def*)

lemma *has-field-derivative-iff*:
 $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } S) \longleftrightarrow$
 $((\lambda y. (f y - f x) / (y - x)) \longrightarrow D) \text{ (at } x \text{ within } S)$
proof –
have $((\lambda y. \text{norm } (f y - f x - D * (y - x)) / \text{norm } (y - x)) \longrightarrow 0) \text{ (at } x$
*within } S)
 $= ((\lambda y. (f y - f x) / (y - x) - D) \longrightarrow 0) \text{ (at } x \text{ within } S)$
by (*smt (verit, best) Lim-cong-within divide-diff-eq-iff norm-divide right-minus-eq*
tendsto-norm-zero-iff)
then show *?thesis*
by (*simp add: has-field-derivative-def has-derivative-iff-norm bounded-linear-mult-right*
LIM-zero-iff)
qed*

lemma *DERIV-def*: $DERIV f x \text{ :> } D \longleftrightarrow (\lambda h. (f (x + h) - f x) / h) - 0 \rightarrow D$
unfolding *field-has-derivative-at has-field-derivative-def has-field-derivative-iff* ..

due to Christian Pardillo Laursen, replacing a proper epsilon-delta horror

lemma *field-derivative-lim-unique*:
assumes $f: (f \text{ has-field-derivative } df) \text{ (at } z)$
and $s: s \longrightarrow 0 \wedge n. s n \neq 0$
and $a: (\lambda n. (f (z + s n) - f z) / s n) \longrightarrow a$
shows $df = a$
proof –
have $((\lambda k. (f (z + k) - f z) / k) \longrightarrow df) \text{ (at } 0)$
using f **by** (*simp add: DERIV-def*)
with s **have** $((\lambda n. (f (z + s n) - f z) / s n) \longrightarrow df)$
by (*simp flip: LIMSEQ-SEQ-conv*)
then show *?thesis*
using a **by** (*rule LIMSEQ-unique*)
qed

lemma *mult-commute-abs*: $(\lambda x. x * c) = ((*) c)$
for $c :: 'a::\text{ab-semigroup-mult}$
by (*simp add: fun-eq-iff mult.commute*)

lemma *DERIV-compose-FDERIV*:
fixes $f::\text{real}\Rightarrow\text{real}$

assumes *DERIV* $f (g x) :> f'$
assumes (*g has-derivative g'*) (at x within s)
shows $((\lambda x. f (g x)) \text{ has-derivative } (\lambda x. g' x * f'))$ (at x within s)
using *assms has-derivative-compose*[of $g g' x s f (*) f'$]
by (*auto simp: has-field-derivative-def ac-simps*)

111.6 Vector derivative

It's for real derivatives only, and not obviously generalisable to field derivatives

lemma *has-real-derivative-iff-has-vector-derivative*:

$(f \text{ has-real-derivative } y) F \longleftrightarrow (f \text{ has-vector-derivative } y) F$

unfolding *has-vector-derivative-def has-field-derivative-def real-scaleR-def mult-commute-abs*
 ..

lemma *has-field-derivative-subset*:

$(f \text{ has-field-derivative } y) \text{ (at } x \text{ within } s) \implies t \subseteq s \implies$

$(f \text{ has-field-derivative } y) \text{ (at } x \text{ within } t)$

by (*fact DERIV-subset*)

lemma *has-vector-derivative-const*[*simp, derivative-intros*]: $((\lambda x. c) \text{ has-vector-derivative } 0)$ *net*

by (*auto simp: has-vector-derivative-def*)

lemma *has-vector-derivative-id*[*simp, derivative-intros*]: $((\lambda x. x) \text{ has-vector-derivative } 1)$ *net*

by (*auto simp: has-vector-derivative-def*)

lemma *has-vector-derivative-minus*[*derivative-intros*]:

$(f \text{ has-vector-derivative } f') \text{ net} \implies ((\lambda x. - f x) \text{ has-vector-derivative } (- f')) \text{ net}$

by (*auto simp: has-vector-derivative-def*)

lemma *has-vector-derivative-add*[*derivative-intros*]:

$(f \text{ has-vector-derivative } f') \text{ net} \implies (g \text{ has-vector-derivative } g') \text{ net} \implies$

$((\lambda x. f x + g x) \text{ has-vector-derivative } (f' + g')) \text{ net}$

by (*auto simp: has-vector-derivative-def scaleR-right-distrib*)

lemma *has-vector-derivative-sum*[*derivative-intros*]:

$(\bigwedge i. i \in I \implies (f i \text{ has-vector-derivative } f' i) \text{ net}) \implies$

$((\lambda x. \sum i \in I. f i x) \text{ has-vector-derivative } (\sum i \in I. f' i)) \text{ net}$

by (*auto simp: has-vector-derivative-def fun-eq-iff scaleR-sum-right intro!: derivative-eq-intros*)

lemma *has-vector-derivative-diff*[*derivative-intros*]:

$(f \text{ has-vector-derivative } f') \text{ net} \implies (g \text{ has-vector-derivative } g') \text{ net} \implies$

$((\lambda x. f x - g x) \text{ has-vector-derivative } (f' - g')) \text{ net}$

by (*auto simp: has-vector-derivative-def scaleR-diff-right*)

lemma *has-vector-derivative-add-const*:

$((\lambda t. g t + z) \text{ has-vector-derivative } f') \text{ net} = ((\lambda t. g t) \text{ has-vector-derivative } f')$
net
apply (*intro iffI*)
apply (*force dest: has-vector-derivative-diff [where $g = \lambda t. z$, OF - has-vector-derivative-const]*)
apply (*force dest: has-vector-derivative-add [OF - has-vector-derivative-const]*)
done

lemma *has-vector-derivative-diff-const:*

$((\lambda t. g t - z) \text{ has-vector-derivative } f') \text{ net} = ((\lambda t. g t) \text{ has-vector-derivative } f')$
net
using *has-vector-derivative-add-const [where $z = -z$]*
by *simp*

lemma (*in bounded-linear*) *has-vector-derivative:*

assumes (*$g \text{ has-vector-derivative } g'$*) *F*
shows $((\lambda x. f (g x)) \text{ has-vector-derivative } f g')$ *F*
using *has-derivative[OF assms[unfolded has-vector-derivative-def]]*
by (*simp add: has-vector-derivative-def scaleR*)

lemma (*in bounded-bilinear*) *has-vector-derivative:*

assumes (*$f \text{ has-vector-derivative } f'$*) (*at x within s*)
and (*$g \text{ has-vector-derivative } g'$*) (*at x within s*)
shows $((\lambda x. f x ** g x) \text{ has-vector-derivative } (f x ** g' + f' ** g x))$ (*at x within s*)
using *FDERIV[OF assms(1-2)[unfolded has-vector-derivative-def]]*
by (*simp add: has-vector-derivative-def scaleR-right scaleR-left scaleR-right-distrib*)

lemma *has-vector-derivative-scaleR[derivative-intros]:*

$(f \text{ has-field-derivative } f') \text{ (at } x \text{ within } s) \implies (g \text{ has-vector-derivative } g') \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x *_R g x) \text{ has-vector-derivative } (f x *_R g' + f' *_R g x)) \text{ (at } x \text{ within } s)$
unfolding *has-real-derivative-iff-has-vector-derivative*
by (*rule bounded-bilinear.has-vector-derivative[OF bounded-bilinear-scaleR]*)

lemma *has-vector-derivative-mult[derivative-intros]:*

$(f \text{ has-vector-derivative } f') \text{ (at } x \text{ within } s) \implies (g \text{ has-vector-derivative } g') \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x * g x) \text{ has-vector-derivative } (f x * g' + f' * g x)) \text{ (at } x \text{ within } s)$
for *$f g :: \text{real} \Rightarrow 'a :: \text{real-normed-algebra}$*
by (*rule bounded-bilinear.has-vector-derivative[OF bounded-bilinear-mult]*)

lemma *has-vector-derivative-of-real[derivative-intros]:*

$(f \text{ has-field-derivative } D) F \implies ((\lambda x. \text{of-real } (f x)) \text{ has-vector-derivative } (\text{of-real } D)) F$
by (*rule bounded-linear.has-vector-derivative[OF bounded-linear-of-real]*)
(simp add: has-real-derivative-iff-has-vector-derivative)

lemma *has-vector-derivative-real-field:*

$(f \text{ has-field-derivative } f') \text{ (at } (\text{of-real } a)) \implies ((\lambda x. f (\text{of-real } x)) \text{ has-vector-derivative } f')$

f') (at a within s)
using *has-derivative-compose*[*of of-real of-real a - f (*) f'*]
by (*simp add: scaleR-conv-of-real ac-simps has-vector-derivative-def has-field-derivative-def*)

lemma *has-vector-derivative-continuous*:
 $(f \text{ has-vector-derivative } D) \text{ (at } x \text{ within } s) \implies \text{continuous (at } x \text{ within } s) f$
by (*auto intro: has-derivative-continuous simp: has-vector-derivative-def*)

lemma *continuous-on-vector-derivative*:
 $(\bigwedge x. x \in S \implies (f \text{ has-vector-derivative } f' x) \text{ (at } x \text{ within } S)) \implies \text{continuous-on } S f$
by (*auto simp: continuous-on-eq-continuous-within intro!: has-vector-derivative-continuous*)

lemma *has-vector-derivative-mult-right*[*derivative-intros*]:
fixes $a :: 'a::\text{real-normed-algebra}$
shows $(f \text{ has-vector-derivative } x) F \implies ((\lambda x. a * f x) \text{ has-vector-derivative } (a * x)) F$
by (*rule bounded-linear.has-vector-derivative[OF bounded-linear-mult-right]*)

lemma *has-vector-derivative-mult-left*[*derivative-intros*]:
fixes $a :: 'a::\text{real-normed-algebra}$
shows $(f \text{ has-vector-derivative } x) F \implies ((\lambda x. f x * a) \text{ has-vector-derivative } (x * a)) F$
by (*rule bounded-linear.has-vector-derivative[OF bounded-linear-mult-left]*)

lemma *has-vector-derivative-divide*[*derivative-intros*]:
fixes $a :: 'a::\text{real-normed-field}$
shows $(f \text{ has-vector-derivative } x) F \implies ((\lambda x. f x / a) \text{ has-vector-derivative } (x / a)) F$
using *has-vector-derivative-mult-left* [*of f x F inverse a*]
by (*simp add: field-class.field-divide-inverse*)

111.7 Derivatives

lemma *DERIV-D*: $DERIV f x :> D \implies (\lambda h. (f (x + h) - f x) / h) - 0 \rightarrow D$
by (*simp add: DERIV-def*)

lemma *has-field-derivativeD*:
 $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } S) \implies$
 $((\lambda y. (f y - f x) / (y - x)) \longrightarrow D) \text{ (at } x \text{ within } S)$
by (*simp add: has-field-derivative-iff*)

lemma *DERIV-const* [*simp, derivative-intros*]: $((\lambda x. k) \text{ has-field-derivative } 0) F$
by (*rule has-derivative-imp-has-field-derivative[OF has-derivative-const]*) *auto*

lemma *DERIV-ident* [*simp, derivative-intros*]: $((\lambda x. x) \text{ has-field-derivative } 1) F$
by (*rule has-derivative-imp-has-field-derivative[OF has-derivative-ident]*) *auto*

lemma *field-differentiable-add*[*derivative-intros*]:

$(f \text{ has-field-derivative } f') F \implies (g \text{ has-field-derivative } g') F \implies$
 $((\lambda z. f z + g z) \text{ has-field-derivative } f' + g') F$
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-add])
 (auto simp: has-field-derivative-def field-simps mult-commute-abs)

corollary *DERIV-add*:

$(f \text{ has-field-derivative } D) (\text{at } x \text{ within } s) \implies (g \text{ has-field-derivative } E) (\text{at } x \text{ within } s) \implies$
 $((\lambda x. f x + g x) \text{ has-field-derivative } D + E) (\text{at } x \text{ within } s)$
by (rule field-differentiable-add)

lemma *field-differentiable-minus*[*derivative-intros*]:

$(f \text{ has-field-derivative } f') F \implies ((\lambda z. - (f z)) \text{ has-field-derivative } -f') F$
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-minus])
 (auto simp: has-field-derivative-def field-simps mult-commute-abs)

corollary *DERIV-minus*:

$(f \text{ has-field-derivative } D) (\text{at } x \text{ within } s) \implies$
 $((\lambda x. - f x) \text{ has-field-derivative } -D) (\text{at } x \text{ within } s)$
by (rule field-differentiable-minus)

lemma *field-differentiable-diff*[*derivative-intros*]:

$(f \text{ has-field-derivative } f') F \implies$
 $(g \text{ has-field-derivative } g') F \implies ((\lambda z. f z - g z) \text{ has-field-derivative } f' - g') F$
by (simp only: diff-conv-add-uminus field-differentiable-add field-differentiable-minus)

corollary *DERIV-diff*:

$(f \text{ has-field-derivative } D) (\text{at } x \text{ within } s) \implies$
 $(g \text{ has-field-derivative } E) (\text{at } x \text{ within } s) \implies$
 $((\lambda x. f x - g x) \text{ has-field-derivative } D - E) (\text{at } x \text{ within } s)$
by (rule field-differentiable-diff)

lemma *DERIV-continuous*: $(f \text{ has-field-derivative } D) (\text{at } x \text{ within } s) \implies \text{continuous} (\text{at } x \text{ within } s) f$

by (drule has-derivative-continuous[OF has-field-derivative-imp-has-derivative])
simp

corollary *DERIV-isCont*: $\text{DERIV } f x \text{ :> } D \implies \text{isCont } f x$

by (rule *DERIV-continuous*)

lemma *DERIV-atLeastAtMost-imp-continuous-on*:

assumes $\bigwedge x. [a \leq x; x \leq b] \implies \exists y. \text{DERIV } f x \text{ :> } y$

shows *continuous-on* $\{a..b\} f$

by (*meson* *DERIV-isCont* *assms* *atLeastAtMost-iff* *continuous-at-imp-continuous-at-within* *continuous-on-eq-continuous-within*)

lemma *DERIV-continuous-on*:

$(\bigwedge x. x \in s \implies (f \text{ has-field-derivative } (D x)) (\text{at } x \text{ within } s)) \implies \text{continuous-on } s f$

unfolding *continuous-on-eq-continuous-within*
by (*intro continuous-at-imp-continuous-on ballI DERIV-continuous*)

lemma *DERIV-mult'*:
 $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies (g \text{ has-field-derivative } E) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x * g x) \text{ has-field-derivative } f x * E + D * g x) \text{ (at } x \text{ within } s)$
by (*rule has-derivative-imp-has-field-derivative[OF has-derivative-mult]*)
(auto simp: field-simps mult-commute-abs dest: has-field-derivative-imp-has-derivative)

lemma *DERIV-mult[derivative-intros]*:
 $(f \text{ has-field-derivative } Da) \text{ (at } x \text{ within } s) \implies (g \text{ has-field-derivative } Db) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x * g x) \text{ has-field-derivative } Da * g x + Db * f x) \text{ (at } x \text{ within } s)$
by (*rule has-derivative-imp-has-field-derivative[OF has-derivative-mult]*)
(auto simp: field-simps dest: has-field-derivative-imp-has-derivative)

Derivative of linear multiplication

lemma *DERIV-cmult*:
 $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. c * f x) \text{ has-field-derivative } c * D) \text{ (at } x \text{ within } s)$
by (*drule DERIV-mult' [OF DERIV-const]*) *simp*

lemma *DERIV-cmult-right*:
 $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x * c) \text{ has-field-derivative } D * c) \text{ (at } x \text{ within } s)$
using *DERIV-cmult* **by** (*auto simp add: ac-simps*)

lemma *DERIV-cmult-Id [simp]*: $((*) c \text{ has-field-derivative } c) \text{ (at } x \text{ within } s)$
using *DERIV-ident [THEN DERIV-cmult, where c = c and x = x]* **by** *simp*

lemma *DERIV-cdivide*:
 $(f \text{ has-field-derivative } D) \text{ (at } x \text{ within } s) \implies$
 $((\lambda x. f x / c) \text{ has-field-derivative } D / c) \text{ (at } x \text{ within } s)$
using *DERIV-cmult-right[of f D x s 1 / c]* **by** *simp*

lemma *DERIV-unique*: $DERIV f x :> D \implies DERIV f x :> E \implies D = E$
unfolding *DERIV-def* **by** (*rule LIM-unique*)

lemma *DERIV-Uniq*: $\exists_{\leq 1} D. DERIV f x :> D$
by (*simp add: DERIV-unique Uniq-def*)

lemma *DERIV-sum[derivative-intros]*:
 $(\bigwedge n. n \in S \implies ((\lambda x. f x n) \text{ has-field-derivative } (f' x n)) F) \implies$
 $((\lambda x. \text{sum } (f x) S) \text{ has-field-derivative } \text{sum } (f' x) S) F$
by (*rule has-derivative-imp-has-field-derivative [OF has-derivative-sum]*)
(auto simp: sum-distrib-left mult-commute-abs dest: has-field-derivative-imp-has-derivative)

lemma *DERIV-inverse'[derivative-intros]*:

assumes (f has-field-derivative D) (at x within s)
and $f x \neq 0$
shows $((\lambda x. \text{inverse } (f x)) \text{ has-field-derivative } - (\text{inverse } (f x) * D * \text{inverse } (f x)))$
(at x within s)
proof –
have (f has-derivative $(\lambda x. x * D)$) = (f has-derivative $(*) D$)
by (*rule arg-cong [of $\lambda x. x * D$] (simp add: fun-eq-iff)*)
with *assms* **have** (f has-derivative $(\lambda x. x * D)$) (at x within s)
by (*auto dest!: has-field-derivative-imp-has-derivative*)
then show *?thesis* **using** $\langle f x \neq 0 \rangle$
by (*auto intro: has-derivative-imp-has-field-derivative has-derivative-inverse*)
qed

Power of -1

lemma *DERIV-inverse*:

$x \neq 0 \implies ((\lambda x. \text{inverse}(x)) \text{ has-field-derivative } - (\text{inverse } x \wedge \text{Suc } (\text{Suc } 0)))$ (at x within s)
by (*drule DERIV-inverse' [OF DERIV-ident] simp*)

Derivative of inverse

lemma *DERIV-inverse-fun*:

$(f \text{ has-field-derivative } d)$ (at x within s) $\implies f x \neq 0 \implies$
 $((\lambda x. \text{inverse } (f x)) \text{ has-field-derivative } - (d * \text{inverse}(f x \wedge \text{Suc } (\text{Suc } 0))))$
(at x within s)
by (*drule (1) DERIV-inverse'*) (*simp add: ac-simps nonzero-inverse-mult-distrib*)

Derivative of quotient

lemma *DERIV-divide[derivative-intros]*:

$(f \text{ has-field-derivative } D)$ (at x within s) \implies
 $(g \text{ has-field-derivative } E)$ (at x within s) $\implies g x \neq 0 \implies$
 $((\lambda x. f x / g x) \text{ has-field-derivative } (D * g x - f x * E) / (g x * g x))$ (at x within s)
by (*rule has-derivative-imp-has-field-derivative[OF has-derivative-divide]*)
(*auto dest: has-field-derivative-imp-has-derivative simp: field-simps*)

lemma *DERIV-quotient*:

$(f \text{ has-field-derivative } d)$ (at x within s) \implies
 $(g \text{ has-field-derivative } e)$ (at x within s) $\implies g x \neq 0 \implies$
 $((\lambda y. f y / g y) \text{ has-field-derivative } (d * g x - (e * f x)) / (g x \wedge \text{Suc } (\text{Suc } 0)))$
(at x within s)
by (*drule (2) DERIV-divide*) (*simp add: mult.commute*)

lemma *DERIV-power-Suc*:

$(f \text{ has-field-derivative } D)$ (at x within s) \implies
 $((\lambda x. f x \wedge \text{Suc } n) \text{ has-field-derivative } (1 + \text{of-nat } n) * (D * f x \wedge n))$ (at x within s)
by (*rule has-derivative-imp-has-field-derivative[OF has-derivative-power]*)
(*auto simp: has-field-derivative-def*)

lemma *DERIV-power*[*derivative-intros*]:
 (*f* has-field-derivative *D*) (at *x* within *s*) \implies
 $((\lambda x. f x \wedge^n) \text{ has-field-derivative of-nat } n * (D * f x \wedge^{(n - \text{Suc } 0)}))$ (at *x* within *s*)
by (*rule* has-derivative-imp-has-field-derivative[*OF* has-derivative-power])
 (*auto simp*: has-field-derivative-def)

lemma *DERIV-pow*: $((\lambda x. x \wedge^n) \text{ has-field-derivative real } n * (x \wedge^{(n - \text{Suc } 0)}))$
 (at *x* within *s*)
using *DERIV-power* [*OF* *DERIV-ident*] **by** *simp*

lemma *DERIV-power-int* [*derivative-intros*]:
assumes [*derivative-intros*]: (*f* has-field-derivative *d*) (at *x* within *s*) **and** [*simp*]:
f *x* $\neq 0$

shows $((\lambda x. \text{power-int } (f x) n) \text{ has-field-derivative of-int } n * \text{power-int } (f x) (n - 1) * d)$ (at *x* within *s*)

proof (*cases* *n* *rule*: *int-cases4*)

case (*nonneg* *n*)

thus ?*thesis*

by (*cases* *n* = 0)

(*auto intro!*: *derivative-eq-intros simp: field-simps power-int-diff simp flip: power-Suc power-Suc2 power-add*)

next

case (*neg* *n*)

thus ?*thesis*

by (*auto intro!*: *derivative-eq-intros simp: field-simps power-int-diff power-int-minus simp flip: power-Suc power-Suc2 power-add*)

qed

lemma *DERIV-chain'*: (*f* has-field-derivative *D*) (at *x* within *s*) \implies *DERIV* *g* (*f* *x*) $:\>$ *E* \implies
 $((\lambda x. g (f x)) \text{ has-field-derivative } E * D)$ (at *x* within *s*)
using *has-derivative-compose*[*of* *f* (*) *D* *x* *s* *g* (*) *E*]
by (*simp only*: *has-field-derivative-def mult-commute-abs ac-simps*)

corollary *DERIV-chain2*: *DERIV* *f* (*g* *x*) $:\>$ *Da* \implies (*g* has-field-derivative *Db*)
 (at *x* within *s*) \implies
 $((\lambda x. f (g x)) \text{ has-field-derivative } Da * Db)$ (at *x* within *s*)
by (*rule* *DERIV-chain'*)

Standard version

lemma *DERIV-chain*:

DERIV *f* (*g* *x*) $:\>$ *Da* \implies (*g* has-field-derivative *Db*) (at *x* within *s*) \implies

(*f* \circ *g* has-field-derivative *Da* * *Db*) (at *x* within *s*)

by (*drule* (1) *DERIV-chain'*, *simp add*: *o-def mult.commute*)

lemma *DERIV-image-chain*:

(*f* has-field-derivative *Da*) (at (*g* *x*) within (*g* ' *s*)) \implies

(*g has-field-derivative Db*) (*at x within s*) \implies
 (*f o g has-field-derivative Da * Db*) (*at x within s*)
using *has-derivative-in-compose* [*of g (*) Db x s f (*) Da*]
by (*simp add: has-field-derivative-def o-def mult-commute-abs ac-simps*)

lemma *DERIV-chain-s*:

assumes ($\bigwedge x. x \in s \implies \text{DERIV } g \ x \ :> \ g'(x)$)
and *DERIV f x :> f'*
and $f \ x \in s$
shows *DERIV* ($\lambda x. g(f \ x)$) $x \ :> \ f' * g'(f \ x)$
by (*metis (full-types) DERIV-chain' mult.commute assms*)

lemma *DERIV-chain3*:

assumes ($\bigwedge x. \text{DERIV } g \ x \ :> \ g'(x)$)
and *DERIV f x :> f'*
shows *DERIV* ($\lambda x. g(f \ x)$) $x \ :> \ f' * g'(f \ x)$
by (*metis UNIV-I DERIV-chain-s [of UNIV] assms*)

Alternative definition for differentiability

lemma *DERIV-LIM-iff*:

fixes $f :: 'a :: \{\text{real-normed-vector, inverse}\} \Rightarrow 'a$
shows ($(\lambda h. (f \ (a + h) - f \ a) / h) \ -0 \rightarrow D$) = ($(\lambda x. (f \ x - f \ a) / (x - a)) \ -a \rightarrow D$) (**is** *?lhs = ?rhs*)

proof

assume *?lhs*
then have ($\lambda x. (f \ (a + (x + - \ a)) - f \ a) / (x + - \ a)$) $-0 \ - \ - \ a \rightarrow D$
by (*rule LIM-offset*)
then show *?rhs*
by *simp*

next

assume *?rhs*
then have ($\lambda x. (f \ (x+a) - f \ a) / ((x+a) - a)$) $-a \ - \ a \rightarrow D$
by (*rule LIM-offset*)
then show *?lhs*
by (*simp add: add.commute*)

qed

lemma *has-field-derivative-cong-ev*:

assumes $x = y$
and $*$: *eventually* ($\lambda x. x \in S \longrightarrow f \ x = g \ x$) (*nhds x*)
and $u = v \ S = t \ x \in S$
shows (*f has-field-derivative u*) (*at x within S*) = (*g has-field-derivative v*) (*at y within t*)
unfolding *has-field-derivative-iff*
proof (*rule filterlim-cong*)
from *assms* **have** $f \ y = g \ y$
by (*auto simp: eventually-nhds*)
with $*$ **show** $\forall_F \ z \ \text{in } \text{at } x \ \text{within } S. (f \ z - f \ x) / (z - x) = (g \ z - g \ y) / (z -$

y)
unfolding *eventually-at-filter*
by *eventually-elim (auto simp: assms ⟨f y = g y⟩)*
qed (*simp-all add: assms*)

lemma *has-field-derivative-cong-eventually*:
assumes *eventually* $(\lambda x. f x = g x)$ *(at x within S)* $f x = g x$
shows $(f \text{ has-field-derivative } u)$ *(at x within S)* $= (g \text{ has-field-derivative } u)$ *(at x within S)*
unfolding *has-field-derivative-iff*
proof (*rule tendsto-cong*)
show $\forall_F y \text{ in at } x \text{ within } S. (f y - f x) / (y - x) = (g y - g x) / (y - x)$
using *assms by (auto elim: eventually-mono)*
qed

lemma *DERIV-cong-ev*:
 $x = y \implies \text{eventually } (\lambda x. f x = g x)$ *(nhds x) \implies u = v \implies*
 $\text{DERIV } f x \text{ :> } u \longleftrightarrow \text{DERIV } g y \text{ :> } v$
by (*rule has-field-derivative-cong-ev*) *simp-all*

lemma *DERIV-mirror*: $(\text{DERIV } f (- x) \text{ :> } y) \longleftrightarrow (\text{DERIV } (\lambda x. f (- x)) x \text{ :> } - y)$
for $f :: \text{real} \Rightarrow \text{real}$ **and** $x y :: \text{real}$
by (*simp add: DERIV-def filterlim-at-split filterlim-at-left-to-right tendsto-minus-cancel-left field-simps conj-commute*)

lemma *DERIV-shift*:
 $(f \text{ has-field-derivative } y)$ *(at (x + z)) = ((\lambda x. f (x + z)) has-field-derivative y)*
(at x)
by (*simp add: DERIV-def field-simps*)

lemma *DERIV-at-within-shift-lemma*:
assumes $(f \text{ has-field-derivative } y)$ *(at (z+x) within (+) z ‘ S)*
shows $(f \circ (+)z \text{ has-field-derivative } y)$ *(at x within S)*
proof –
have $((+)z \text{ has-field-derivative } 1)$ *(at x within S)*
by (*rule derivative-eq-intros | simp*) $+$
with *assms DERIV-image-chain show ?thesis*
by (*metis mult.right-neutral*)
qed

lemma *DERIV-at-within-shift*:
 $(f \text{ has-field-derivative } y)$ *(at (z+x) within (+) z ‘ S) \longleftrightarrow*
 $((\lambda x. f (z+x)) \text{ has-field-derivative } y)$ *(at x within S)* **(is ?lhs = ?rhs)**
proof
assume *?lhs then show ?rhs*
using *DERIV-at-within-shift-lemma unfolding o-def by blast*
next
have [*simp*]: $(\lambda x. x - z) ‘ (+) z ‘ S = S$

by force
 assume $R: ?rhs$
 have $(f \circ (+) z \circ (+) (- z) \text{ has-field-derivative } y) \text{ (at } (z + x) \text{ within } (+) z \text{ ' } S)$
 by (rule *DERIV-at-within-shift-lemma*) (use R in $\langle \text{simp add: o-def} \rangle$)
 then show $?lhs$
 by (simp add: o-def)
 qed

lemma *floor-has-real-derivative*:
 fixes $f :: \text{real} \Rightarrow 'a::\{\text{floor-ceiling, order-topology}\}$
 assumes $\text{isCont } f \ x$
 and $f \ x \notin \mathbb{Z}$
 shows $((\lambda x. \text{floor } (f \ x)) \text{ has-real-derivative } 0) \text{ (at } x)$
proof (subst *DERIV-cong-ev*[*OF refl - refl*])
 show $((\lambda x. \text{floor } (f \ x)) \text{ has-real-derivative } 0) \text{ (at } x)$
 by simp
 have $\forall_F y \text{ in at } x. \lfloor f \ y \rfloor = \lfloor f \ x \rfloor$
 by (rule *eventually-floor-eq*[*OF assms*[*unfolded continuous-at*]])
 then show $\forall_F y \text{ in nhds } x. \text{real-of-int } \lfloor f \ y \rfloor = \text{real-of-int } \lfloor f \ x \rfloor$
 unfolding *eventually-at-filter*
 by *eventually-elim auto*
 qed

lemmas *has-derivative-floor*[*derivative-intros*] =
floor-has-real-derivative[*THEN DERIV-compose-FDERIV*]

lemma *continuous-floor*:
 fixes $x::\text{real}$
 shows $x \notin \mathbb{Z} \implies \text{continuous (at } x) (\text{real-of-int} \circ \text{floor})$
 using *floor-has-real-derivative* [where $f=id$]
 by (auto simp: o-def has-field-derivative-def intro: has-derivative-continuous)

lemma *continuous-frac*:
 fixes $x::\text{real}$
 assumes $x \notin \mathbb{Z}$
 shows $\text{continuous (at } x) \text{frac}$
proof –
 have $\text{isCont } (\lambda x. \text{real-of-int } \lfloor x \rfloor) \ x$
 using *continuous-floor* [*OF assms*] by (simp add: o-def)
 then have $*: \text{continuous (at } x) (\lambda x. x - \text{real-of-int } \lfloor x \rfloor)$
 by (intro *continuous-intros*)
 moreover have $\forall_F x \text{ in nhds } x. \text{frac } x = x - \text{real-of-int } \lfloor x \rfloor$
 by (simp add: *frac-def*)
 ultimately show $?thesis$
 by (simp add: *LIM-imp-LIM frac-def isCont-def*)
 qed

Caratheodory formulation of derivative at a point

lemma *CARAT-DERIV*:

$(\text{DERIV } f x \text{ :> } l) \longleftrightarrow (\exists g. (\forall z. f z - f x = g z * (z - x)) \wedge \text{isCont } g x \wedge g x = l)$
 (is ?lhs = ?rhs)
proof
 assume ?lhs
 show $\exists g. (\forall z. f z - f x = g z * (z - x)) \wedge \text{isCont } g x \wedge g x = l$
proof (intro exI conjI)
 let ?g = $(\lambda z. \text{if } z = x \text{ then } l \text{ else } (f z - f x) / (z - x))$
 show $\forall z. f z - f x = ?g z * (z - x)$
 by simp
 show $\text{isCont } ?g x$
 using <?lhs> by (simp add: isCont-iff DERIV-def cong: LIM-equal [rule-format])
 show ?g x = l
 by simp
 qed
next
 assume ?rhs
 then show ?lhs
 by (auto simp add: isCont-iff DERIV-def cong: LIM-cong)
 qed

111.8 Local extrema

If $(0::'a) < f' x$ then x is Locally Strictly Increasing At The Right.

lemma *has-real-derivative-pos-inc-right*:

fixes $f :: \text{real} \Rightarrow \text{real}$
 assumes *der*: $(f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S)$
 and $l: 0 < l$
 shows $\exists d > 0. \forall h > 0. x + h \in S \longrightarrow h < d \longrightarrow f x < f (x + h)$
 using *assms*
proof –
 from *der* [THEN *has-field-derivativeD*, THEN *tendstoD*, OF l , unfolded *eventually-at*]
 obtain s where $s: 0 < s$
 and *all*: $\bigwedge xa. xa \in S \Longrightarrow xa \neq x \wedge \text{dist } xa x < s \longrightarrow |(f xa - f x) / (xa - x) - l| < l$
 by (auto simp: *dist-real-def*)
 then show *thesis*
proof (intro exI conjI strip)
 show $0 < s$ by (rule s)
next
 fix $h :: \text{real}$
 assume $0 < h \wedge h < s \wedge x + h \in S$
 with *all* [of $x + h$] show $f x < f (x + h)$
proof (simp add: *abs-if dist-real-def pos-less-divide-eq split: if-split-asm*)
 assume $\neg (f (x + h) - f x) / h < l$ and $h: 0 < h$
 with l have $0 < (f (x + h) - f x) / h$
 by *arith*
 then show $f x < f (x + h)$

by (simp add: pos-less-divide-eq h)
 qed
 qed
 qed

lemma DERIV-pos-inc-right:

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $\text{der}: \text{DERIV } f \ x \ :> \ l$

and $l: 0 < l$

shows $\exists d > 0. \forall h > 0. h < d \longrightarrow f \ x < f \ (x + h)$

using has-real-derivative-pos-inc-right[OF assms]

by auto

lemma has-real-derivative-neg-dec-left:

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $\text{der}: (f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S)$

and $l < 0$

shows $\exists d > 0. \forall h > 0. x - h \in S \longrightarrow h < d \longrightarrow f \ x < f \ (x - h)$

proof -

from $\langle l < 0 \rangle$ have $l: -l > 0$

by simp

from der [THEN has-field-derivativeD, THEN tendstoD, OF l, unfolded eventually-at]

obtain s where $s: 0 < s$

and $\text{all}: \bigwedge xa. xa \in S \implies xa \neq x \wedge \text{dist } xa \ x < s \longrightarrow |(f \ xa - f \ x) / (xa - x) - l| < -l$

by (auto simp: dist-real-def)

then show ?thesis

proof (intro exI conjI strip)

show $0 < s$ by (rule s)

next

fix $h :: \text{real}$

assume $0 < h \ h < s \ x - h \in S$

with all [of $x - h$] show $f \ x < f \ (x - h)$

proof (simp add: abs-if pos-less-divide-eq dist-real-def split: if-split-asm)

assume $-(f \ (x - h) - f \ x) / h < l$ and $h: 0 < h$

with l have $0 < (f \ (x - h) - f \ x) / h$

by arith

then show $f \ x < f \ (x - h)$

by (simp add: pos-less-divide-eq h)

qed

qed

qed

lemma DERIV-neg-dec-left:

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $\text{der}: \text{DERIV } f \ x \ :> \ l$

and $l: l < 0$

shows $\exists d > 0. \forall h > 0. h < d \longrightarrow f \ x < f \ (x - h)$

using *has-real-derivative-neg-dec-left* [*OF assms*]
by *auto*

lemma *has-real-derivative-pos-inc-left*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $(f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S) \Longrightarrow 0 < l \Longrightarrow$

$\exists d > 0. \forall h > 0. x - h \in S \longrightarrow h < d \longrightarrow f(x - h) < f x$

by (*rule has-real-derivative-neg-dec-left* [*of* $\lambda x. - f x - l x S$, *simplified*])
(auto simp add: DERIV-minus)

lemma *DERIV-pos-inc-left*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $DERIV f x :> l \Longrightarrow 0 < l \Longrightarrow \exists d > 0. \forall h > 0. h < d \longrightarrow f(x - h) < f x$

using *has-real-derivative-pos-inc-left*
by *blast*

lemma *has-real-derivative-neg-dec-right*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $(f \text{ has-real-derivative } l) \text{ (at } x \text{ within } S) \Longrightarrow l < 0 \Longrightarrow$

$\exists d > 0. \forall h > 0. x + h \in S \longrightarrow h < d \longrightarrow f x > f(x + h)$

by (*rule has-real-derivative-pos-inc-right* [*of* $\lambda x. - f x - l x S$, *simplified*])
(auto simp add: DERIV-minus)

lemma *DERIV-neg-dec-right*:

fixes $f :: \text{real} \Rightarrow \text{real}$

shows $DERIV f x :> l \Longrightarrow l < 0 \Longrightarrow \exists d > 0. \forall h > 0. h < d \longrightarrow f x > f(x + h)$

using *has-real-derivative-neg-dec-right* **by** *blast*

lemma *DERIV-local-max*:

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes *der*: $DERIV f x :> l$

and $d: 0 < d$

and *le*: $\forall y. |x - y| < d \longrightarrow f y \leq f x$

shows $l = 0$

proof (*cases rule: linorder-cases* [*of* $l\ 0$])

case *equal*

then show *?thesis* .

next

case *less*

from *DERIV-neg-dec-left* [*OF der less*]

obtain d' **where** $d': 0 < d'$ **and** *lt*: $\forall h > 0. h < d' \longrightarrow f x < f(x - h)$

by *blast*

obtain e **where** $0 < e \wedge e < d \wedge e < d'$

using *field-lbound-gt-zero* [*OF d d'*] ..

with *lt le* [*THEN spec* [**where** $x = x - e$]] **show** *?thesis*

by (*auto simp add: abs-if*)

next

```

case greater
from DERIV-pos-inc-right [OF der greater]
obtain d' where d':  $0 < d'$  and lt:  $\forall h > 0. h < d' \longrightarrow f\ x < f\ (x + h)$ 
by blast
obtain e where  $0 < e \wedge e < d \wedge e < d'$ 
using field-lbound-gt-zero [OF d d'] ..
with lt le [THEN spec [where  $x=x + e$ ]] show ?thesis
by (auto simp add: abs-if)
qed

```

Similar theorem for a local minimum

```

lemma DERIV-local-min:
fixes f :: real  $\Rightarrow$  real
shows DERIV f x :> l  $\Longrightarrow 0 < d \Longrightarrow \forall y. |x - y| < d \longrightarrow f\ x \leq f\ y \Longrightarrow l = 0$ 
by (drule DERIV-minus [THEN DERIV-local-max]) auto

```

In particular, if a function is locally flat

```

lemma DERIV-local-const:
fixes f :: real  $\Rightarrow$  real
shows DERIV f x :> l  $\Longrightarrow 0 < d \Longrightarrow \forall y. |x - y| < d \longrightarrow f\ x = f\ y \Longrightarrow l = 0$ 
by (auto dest!: DERIV-local-max)

```

111.9 Rolle’s Theorem

Lemma about introducing open ball in open interval

```

lemma lemma-interval-lt:
fixes a b x :: real
assumes  $a < x < b$ 
shows  $\exists d. 0 < d \wedge (\forall y. |x - y| < d \longrightarrow a < y \wedge y < b)$ 
using linorder-linear [of x - a b - x]
proof
assume  $x - a \leq b - x$ 
with assms show ?thesis
by (rule-tac x = x - a in exI) auto
next
assume  $b - x \leq x - a$ 
with assms show ?thesis
by (rule-tac x = b - x in exI) auto
qed

```

```

lemma lemma-interval:  $a < x \Longrightarrow x < b \Longrightarrow \exists d. 0 < d \wedge (\forall y. |x - y| < d \longrightarrow a \leq y \wedge y \leq b)$ 
for a b x :: real
by (force dest: lemma-interval-lt)

```

Rolle’s Theorem. If f is defined and continuous on the closed interval $[a, b]$ and differentiable on the open interval (a, b) , and $f\ a = f\ b$, then there exists $x_0 \in (a, b)$ such that $f'\ x_0 = (0::'a)$

theorem *Rolle-deriv*:

fixes $f :: \text{real} \Rightarrow \text{real}$

assumes $a < b$

and $\text{fab}: f a = f b$

and $\text{contf}: \text{continuous-on } \{a..b\} f$

and $\text{derf}: \bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow (f \text{ has-derivative } f' x) \text{ (at } x)$

shows $\exists z. a < z \wedge z < b \wedge f' z = (\lambda v. 0)$

proof –

have $le: a \leq b$

using $\langle a < b \rangle$ **by** *simp*

have $(a + b) / 2 \in \{a..b\}$

using $\text{assms}(1)$ **by** *auto*

then have $*$: $\{a..b\} \neq \{\}$

by *auto*

obtain x **where** $x\text{-max}: \forall z. a \leq z \wedge z \leq b \longrightarrow f z \leq f x$ **and** $a \leq x \wedge x \leq b$

using $\text{continuous-attains-sup}[\text{OF compact-Icc} * \text{contf}]$

by (*meson atLeastAtMost-iff*)

obtain x' **where** $x'\text{-min}: \forall z. a \leq z \wedge z \leq b \longrightarrow f x' \leq f z$ **and** $a \leq x' \wedge x' \leq b$

using $\text{continuous-attains-inf}[\text{OF compact-Icc} * \text{contf}]$ **by** (*meson atLeastAtMost-iff*)

consider $a < x \wedge x < b \mid x = a \vee x = b$

using $\langle a \leq x \rangle \langle x \leq b \rangle$ **by** *arith*

then show *?thesis*

proof *cases*

case 1

– f attains its maximum within the interval

then obtain l **where** $\text{der}: \text{DERIV } f x :=> l$

using $\text{derf differentiable-def real-differentiable-def}$ **by** *blast*

obtain d **where** $d: 0 < d$ **and** $\text{bound}: \forall y. |x - y| < d \longrightarrow a \leq y \wedge y \leq b$

using $\text{lemma-interval} [\text{OF } 1]$ **by** *blast*

then have $\text{bound}' : \forall y. |x - y| < d \longrightarrow f y \leq f x$

using $x\text{-max}$ **by** *blast*

– the derivative at a local maximum is zero

have $l = 0$

by (*rule DERIV-local-max [OF der d bound']*)

with 1 $\text{der derf} [\text{of } x]$ **show** *?thesis*

by (*metis has-derivative-unique has-field-derivative-def mult-zero-left*)

next

case 2

then have $\text{fx}: f b = f x$ **by** (*auto simp add: fab*)

consider $a < x' \wedge x' < b \mid x' = a \vee x' = b$

using $\langle a \leq x' \rangle \langle x' \leq b \rangle$ **by** *arith*

then show *?thesis*

proof *cases*

case 1

– f attains its minimum within the interval

then obtain l **where** $\text{der}: \text{DERIV } f x' :=> l$

using $\text{derf differentiable-def real-differentiable-def}$ **by** *blast*

from $\text{lemma-interval} [\text{OF } 1]$

obtain d **where** $d: 0 < d$ **and** $bound: \forall y. |x' - y| < d \longrightarrow a \leq y \wedge y \leq b$
by *blast*
then have $bound': \forall y. |x' - y| < d \longrightarrow f x' \leq f y$
using *x'-min by blast*
have $l = 0$ **by** (*rule DERIV-local-min [OF der d bound']*)
— the derivative at a local minimum is zero
then show *?thesis using 1 der derf [of x']*
by (*metis has-derivative-unique has-field-derivative-def mult-zero-left*)
next
case 2
— f is constant throughout the interval
then have $fx': f b = f x'$ **by** (*auto simp: fab*)
from *dense [OF <a < b>]* **obtain** r **where** $r: a < r \wedge r < b$ **by** *blast*
obtain d **where** $d: 0 < d$ **and** $bound: \forall y. |r - y| < d \longrightarrow a \leq y \wedge y \leq b$
using *lemma-interval [OF r]* **by** *blast*
have $eq\text{-}fb: f z = f b$ **if** $a \leq z$ **and** $z \leq b$ **for** z
proof (*rule order-antisym*)
show $f z \leq f b$ **by** (*simp add: fx x-max that*)
show $f b \leq f z$ **by** (*simp add: fx' x'-min that*)
qed
have $bound': \forall y. |r - y| < d \longrightarrow f r = f y$
proof (*intro strip*)
fix $y :: real$
assume $lt: |r - y| < d$
then have $f y = f b$ **by** (*simp add: eq-fb bound*)
then show $f r = f y$ **by** (*simp add: eq-fb r order-less-imp-le*)
qed
obtain l **where** $der: DERIV f r := l$
using *derf differentiable-def r(1) r(2) real-differentiable-def* **by** *blast*
have $l = 0$
by (*rule DERIV-local-const [OF der d bound']*)
— the derivative of a constant function is zero
with r *der derf [of r]* **show** *?thesis*
by (*metis has-derivative-unique has-field-derivative-def mult-zero-left*)
qed
qed
qed

corollary *Rolle:*

fixes $a b :: real$
assumes $ab: a < b$ $f a = f b$ *continuous-on {a..b} f*
and *dif [rule-format]: $\bigwedge x. [a < x; x < b] \implies f$ differentiable (at x)*
shows $\exists z. a < z \wedge z < b \wedge DERIV f z := 0$
proof —
obtain f' **where** $f': \bigwedge x. [a < x; x < b] \implies (f \text{ has-derivative } f' x)$ (at x)
using *dif unfolding differentiable-def* **by** *metis*
then have $\exists z. a < z \wedge z < b \wedge f' z = (\lambda v. 0)$
by (*metis Rolle-deriv [OF ab]*)
then show *?thesis*

using f' *has-derivative-imp-has-field-derivative* by *fastforce*
qed

111.10 Mean Value Theorem

theorem *mvt*:

fixes $f :: real \Rightarrow real$
assumes $a < b$
and *contf*: *continuous-on* $\{a..b\}$ f
and *derf*: $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow (f \text{ has-derivative } f' x) (at x)$
obtains ξ **where** $a < \xi < b$ $f b - f a = (f' \xi) (b - a)$
proof –
have $\exists \xi. a < \xi \wedge \xi < b \wedge (\lambda y. f' \xi y - (f b - f a) / (b - a) * y) = (\lambda v. 0)$
proof (*intro Rolle-deriv[OF <a < b>]*)
fix x
assume $x: a < x < b$
show $((\lambda x. f x - (f b - f a) / (b - a) * x)$
has-derivative $(\lambda y. f' x y - (f b - f a) / (b - a) * y)) (at x)$
by (*intro derivative-intros derf[OF x]*)
qed (*use assms in <auto intro!> continuous-intros simp: field-simps*)
then show *?thesis*
by (*smt (verit, ccfv-SIG) pos-le-divide-eq pos-less-divide-eq that*)
qed

theorem *MVT*:

fixes $a b :: real$
assumes *lt*: $a < b$
and *contf*: *continuous-on* $\{a..b\}$ f
and *dif*: $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow f \text{ differentiable } (at x)$
shows $\exists l z. a < z \wedge z < b \wedge DERIV f z := l \wedge f b - f a = (b - a) * l$
proof –
obtain $f' :: real \Rightarrow real \Rightarrow real$
where *derf*: $\bigwedge x. a < x \Longrightarrow x < b \Longrightarrow (f \text{ has-derivative } f' x) (at x)$
using *dif unfolding differentiable-def* by *metis*
then obtain z **where** $a < z < b$ $f b - f a = (f' z) (b - a)$
using *mvt [OF lt contf]* by *blast*
then show *?thesis*
by (*simp add: ac-simps*)
(metis derf dif has-derivative-unique has-field-derivative-imp-has-derivative real-differentiable-def)
qed

corollary *MVT2*:

assumes $a < b$ **and** *der*: $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow DERIV f x := f' x$
shows $\exists z :: real. a < z \wedge z < b \wedge (f b - f a = (b - a) * f' z)$
proof –
have $\exists l z. a < z \wedge$
 $z < b \wedge$
 $(f \text{ has-real-derivative } l) (at z) \wedge$

$f b - f a = (b - a) * l$
proof (rule MVT [OF $\langle a < b \rangle$])
show *continuous-on* $\{a..b\}$ f
by (meson DERIV-continuous atLeastAtMost-iff continuous-at-imp-continuous-on
der)
show $\bigwedge x. \llbracket a < x; x < b \rrbracket \implies f$ *differentiable* (at x)
using *assms* **by** (force dest: order-less-imp-le simp add: real-differentiable-def)
qed
with *assms* **show** ?thesis
by (blast dest: DERIV-unique order-less-imp-le)
qed

lemma *pos-deriv-imp-strict-mono*:

assumes $\bigwedge x. (f$ *has-real-derivative* $f' x)$ (at x)
assumes $\bigwedge x. f' x > 0$
shows *strict-mono* f
proof (rule *strict-monoI*)
fix $x y :: \text{real}$ **assume** $xy: x < y$
from *assms* **and** xy **have** $\exists z > x. z < y \wedge f y - f x = (y - x) * f' z$
by (*intro* MVT2) (*auto* dest: *connectedD-interval*)
then **obtain** z **where** $z: z > x \wedge z < y \wedge f y - f x = (y - x) * f' z$ **by** *blast*
note $\langle f y - f x = (y - x) * f' z \rangle$
also **have** $(y - x) * f' z > 0$ **using** xy *assms* **by** (*intro* *mult-pos-pos*) *auto*
finally **show** $f x < f y$ **by** *simp*
qed

proposition *deriv-nonneg-imp-mono*:

assumes *deriv*: $\bigwedge x. x \in \{a..b\} \implies (g$ *has-real-derivative* $g' x)$ (at x)
assumes *nonneg*: $\bigwedge x. x \in \{a..b\} \implies g' x \geq 0$
assumes $ab: a \leq b$
shows $g a \leq g b$
proof (*cases* $a < b$)
assume $a < b$
from *deriv* **have** $\bigwedge x. \llbracket x \geq a; x \leq b \rrbracket \implies (g$ *has-real-derivative* $g' x)$ (at x) **by**
simp
with MVT2[OF $\langle a < b \rangle$] **and** *deriv*
obtain ξ **where** ξ -*ab*: $\xi > a \wedge \xi < b$ **and** g -*ab*: $g b - g a = (b - a) * g' \xi$ **by**
blast
from ξ -*ab* ab *nonneg* **have** $(b - a) * g' \xi \geq 0$ **by** *simp*
with g -*ab* **show** ?thesis **by** *simp*
qed (*insert* $ab, simp$)

111.10.1 A function is constant if its derivative is 0 over an interval.

lemma *DERIV-isconst-end*:

fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $a < b$ **and** *contf*: *continuous-on* $\{a..b\}$ f
and 0 : $\bigwedge x. \llbracket a < x; x < b \rrbracket \implies \text{DERIV } f x := 0$

shows $f b = f a$
using *MVT* [OF $\langle a < b \rangle$] 0 *DERIV-unique contf real-differentiable-def*
by (*fastforce simp: algebra-simps*)

lemma *DERIV-isconst2*:

fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $a < b$ **and** *contf*: *continuous-on* $\{a..b\}$ f **and** *derf*: $\bigwedge x. \llbracket a < x; x < b \rrbracket \implies \text{DERIV } f x :=> 0$
and $a \leq x \leq b$
shows $f x = f a$
proof (*cases a < x*)
case *True*
have $*$: *continuous-on* $\{a..x\}$ f
using $\langle x \leq b \rangle$ *contf continuous-on-subset* **by** *fastforce*
show *?thesis*
by (*rule DERIV-isconst-end* [OF *True **]) (*use* $\langle x \leq b \rangle$ *derf in auto*)
qed (*use* $\langle a \leq x \rangle$ **in auto**)

lemma *DERIV-isconst3*:

fixes $a b x y :: \text{real}$
assumes $a < b$
and $x \in \{a <..< < b\}$
and $y \in \{a <..< < b\}$
and *derivable*: $\bigwedge x. x \in \{a <..< < b\} \implies \text{DERIV } f x :=> 0$
shows $f x = f y$
proof (*cases x = y*)
case *False*
let $?a = \min x y$
let $?b = \max x y$
have $*$: $\text{DERIV } f z :=> 0$ **if** $?a \leq z \leq ?b$ **for** z
proof –
have $a < z$ **and** $z < b$
using *that* $\langle x \in \{a <..< < b\} \rangle$ **and** $\langle y \in \{a <..< < b\} \rangle$ **by** *auto*
then **have** $z \in \{a <..< < b\}$ **by** *auto*
then **show** $\text{DERIV } f z :=> 0$ **by** (*rule derivable*)
qed
have *isCont*: *continuous-on* $\{?a..?b\}$ f
by (*meson * DERIV-continuous-on atLeastAtMost-iff has-field-derivative-at-within*)
have *DERIV*: $\bigwedge z. \llbracket ?a < z; z < ?b \rrbracket \implies \text{DERIV } f z :=> 0$
using $*$ **by** *auto*
have $?a < ?b$ **using** $\langle x \neq y \rangle$ **by** *auto*
from *DERIV-isconst2*[OF *this isCont DERIV, of x*] **and** *DERIV-isconst2*[OF *this isCont DERIV, of y*]
show *?thesis* **by** *auto*
qed *auto*

lemma *DERIV-isconst-all*:

fixes $f :: \text{real} \Rightarrow \text{real}$
shows $\forall x. \text{DERIV } f x :=> 0 \implies f x = f y$

```

apply (rule linorder-cases [of x y])
apply (metis DERIV-continuous DERIV-isconst-end continuous-at-imp-continuous-on)+
done

```

lemma *DERIV-const-ratio-const*:

```

fixes f :: real ⇒ real
assumes a ≠ b and df:  $\bigwedge x. \text{DERIV } f \ x \ :> \ k$ 
shows f b - f a = (b - a) * k
proof (cases a b rule: linorder-cases)
  case less
    show ?thesis
      using MVT [OF less] df
      by (metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on
real-differentiable-def)
  next
    case greater
      have f a - f b = (a - b) * k
        using MVT [OF greater] df
        by (metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on
real-differentiable-def)
      then show ?thesis
        by (simp add: algebra-simps)
qed auto

```

lemma *DERIV-const-ratio-const2*:

```

fixes f :: real ⇒ real
assumes a ≠ b and df:  $\bigwedge x. \text{DERIV } f \ x \ :> \ k$ 
shows (f b - f a) / (b - a) = k
using DERIV-const-ratio-const [OF assms] ⟨a ≠ b⟩ by auto

```

lemma *real-average-minus-first* [simp]: $(a + b) / 2 - a = (b - a) / 2$

```

for a b :: real
by simp

```

lemma *real-average-minus-second* [simp]: $(b + a) / 2 - a = (b - a) / 2$

```

for a b :: real
by simp

```

Gallileo's "trick": average velocity = av. of end velocities.

lemma *DERIV-const-average*:

```

fixes v :: real ⇒ real
and a b :: real
assumes neq: a ≠ b
and der:  $\bigwedge x. \text{DERIV } v \ x \ :> \ k$ 
shows v ((a + b) / 2) = (v a + v b) / 2
proof (cases rule: linorder-cases [of a b])
  case equal
    with neq show ?thesis by simp
next

```



```

case less
have (v b - v a) / (b - a) = k
  by (rule DERIV-const-ratio-const2 [OF neq der])
then have (b - a) * ((v b - v a) / (b - a)) = (b - a) * k
  by simp
moreover have (v ((a + b) / 2) - v a) / ((a + b) / 2 - a) = k
  by (rule DERIV-const-ratio-const2 [OF - der]) (simp add: neq)
ultimately show ?thesis
  using neq by force
next
case greater
have (v b - v a) / (b - a) = k
  by (rule DERIV-const-ratio-const2 [OF neq der])
then have (b - a) * ((v b - v a) / (b - a)) = (b - a) * k
  by simp
moreover have (v ((b + a) / 2) - v a) / ((b + a) / 2 - a) = k
  by (rule DERIV-const-ratio-const2 [OF - der]) (simp add: neq)
ultimately show ?thesis
  using neq by (force simp add: add.commute)
qed

```

111.10.2 A function with positive derivative is increasing

A simple proof using the MVT, by Jeremy Avigad. And variants.

lemma *DERIV-pos-imp-increasing-open*:

```

fixes a b :: real
  and f :: real ⇒ real
assumes a < b
  and  $\bigwedge x. a < x \implies x < b \implies (\exists y. \text{DERIV } f \ x \ :> y \wedge y > 0)$ 
  and con: continuous-on {a..b} f
shows f a < f b
proof (rule ccontr)
  assume f:  $\neg$  ?thesis
  have  $\exists l z. a < z \wedge z < b \wedge \text{DERIV } f \ z \ :> l \wedge f b - f a = (b - a) * l$ 
    by (rule MVT) (use assms real-differentiable-def in ⟨force+⟩)
  then obtain l z where z: a < z z < b DERIV f z :> l and f b - f a = (b - a)
  * l
    by auto
  with assms f have  $\neg l > 0$ 
    by (metis linorder-not-le mult-le-0-iff diff-le-0-iff-le)
  with assms z show False
    by (metis DERIV-unique)
qed

```

lemma *DERIV-pos-imp-increasing*:

```

fixes a b :: real and f :: real ⇒ real
assumes a < b
  and der:  $\bigwedge x. [a \leq x; x \leq b] \implies \exists y. \text{DERIV } f \ x \ :> y \wedge y > 0$ 
shows f a < f b

```

by (metis less-le-not-le DERIV-atLeastAtMost-imp-continuous-on DERIV-pos-imp-increasing-open [OF ‹a < b›] der)

lemma DERIV-nonneg-imp-nondecreasing:

fixes a b :: real
 and f :: real \Rightarrow real
 assumes a \leq b
 and $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \exists y. \text{DERIV } f x \text{ :> } y \wedge y \geq 0$
 shows f a \leq f b
proof (rule ccontr, cases a = b)
 assume \neg ?thesis and a = b
 then show False by auto
next
 assume *: \neg ?thesis
 assume a \neq b
 with ‹a \leq b› have a < b
 by linarith
 moreover have continuous-on {a..b} f
 by (meson DERIV-isCont assms(2) atLeastAtMost-iff continuous-at-imp-continuous-on)
 ultimately have $\exists l z. a < z \wedge z < b \wedge \text{DERIV } f z \text{ :> } l \wedge f b - f a = (b - a)$
 * l
 using assms MVT [OF ‹a < b›, of f] real-differentiable-def less-eq-real-def by
 blast
 then obtain l z where lz: a < z z < b DERIV f z :> l and **: f b - f a = (b - a) * l
 by auto
 with * have a < b f b < f a by auto
 with ** have $\neg l \geq 0$ by (auto simp add: not-le algebra-simps)
 (metis * add-le-cancel-right assms(1) less-eq-real-def mult-right-mono add-left-mono linear order-refl)
 with assms lz show False
 by (metis DERIV-unique order-less-imp-le)
qed

lemma DERIV-neg-imp-decreasing-open:

fixes a b :: real
 and f :: real \Rightarrow real
 assumes a < b
 and $\bigwedge x. a < x \Longrightarrow x < b \Longrightarrow \exists y. \text{DERIV } f x \text{ :> } y \wedge y < 0$
 and con: continuous-on {a..b} f
 shows f a > f b
proof -
 have $(\lambda x. -f x) a < (\lambda x. -f x) b$
proof (rule DERIV-pos-imp-increasing-open [of a b])
 show $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow \exists y. ((\lambda x. -f x) \text{ has-real-derivative } y) (at x) \wedge 0 < y$
 using assms
 by simp (metis field-differentiable-minus neg-0-less-iff-less)
 show continuous-on {a..b} $(\lambda x. -f x)$

using *con continuous-on-minus* by *blast*
 qed (use *assms in auto*)
 then show *?thesis*
 by *simp*
 qed

lemma *DERIV-neg-imp-decreasing*:

fixes $a b :: \text{real}$ and $f :: \text{real} \Rightarrow \text{real}$
 assumes $a < b$
 and *der*: $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \exists y. \text{DERIV } f x :> y \wedge y < 0$
 shows $f a > f b$
 by (*metis less-le-not-le DERIV-atLeastAtMost-imp-continuous-on DERIV-neg-imp-decreasing-open*
 [OF $\langle a < b \rangle$] *der*)

lemma *DERIV-nonpos-imp-nonincreasing*:

fixes $a b :: \text{real}$
 and $f :: \text{real} \Rightarrow \text{real}$
 assumes $a \leq b$
 and $\bigwedge x. \llbracket a \leq x; x \leq b \rrbracket \Longrightarrow \exists y. \text{DERIV } f x :> y \wedge y \leq 0$
 shows $f a \geq f b$
 proof –
 have $(\lambda x. -f x) a \leq (\lambda x. -f x) b$
 using *DERIV-nonneg-imp-nondecreasing* [of $a b \lambda x. -f x$] *assms DERIV-minus*
 by *fastforce*
 then show *?thesis*
 by *simp*
 qed

lemma *DERIV-pos-imp-increasing-at-bot*:

fixes $f :: \text{real} \Rightarrow \text{real}$
 assumes $\bigwedge x. x \leq b \Longrightarrow (\exists y. \text{DERIV } f x :> y \wedge y > 0)$
 and *lim*: $(f \longrightarrow \text{flim})$ *at-bot*
 shows $\text{flim} < f b$
 proof –
 have $\exists N. \forall n \leq N. f n \leq f (b - 1)$
 by (*rule-tac x=b - 2 in exI*) (*force intro: order.strict-implies-order DERIV-pos-imp-increasing assms*)
 then have $\text{flim} \leq f (b - 1)$
 by (*auto simp: eventually-at-bot-linorder tendsto-upperbound* [OF *lim*])
 also have $\dots < f b$
 by (*force intro: DERIV-pos-imp-increasing* [where $f=f$] *assms*)
 finally show *?thesis* .
 qed

lemma *DERIV-neg-imp-decreasing-at-top*:

fixes $f :: \text{real} \Rightarrow \text{real}$
 assumes *der*: $\bigwedge x. x \geq b \Longrightarrow \exists y. \text{DERIV } f x :> y \wedge y < 0$
 and *lim*: $(f \longrightarrow \text{flim})$ *at-top*
 shows $\text{flim} < f b$

```

apply (rule DERIV-pos-imp-increasing-at-bot [where  $f = \lambda i. f (-i)$  and  $b = -b$ , simplified])
apply (metis DERIV-mirror der le-minus-iff neg-0-less-iff-less)
apply (metis filterlim-at-top-mirror lim)
done

```

Derivative of inverse function

lemma *DERIV-inverse-function*:

```

fixes  $f\ g :: real \Rightarrow real$ 
assumes  $der: DERIV\ f\ (g\ x) :> D$ 
and  $neq: D \neq 0$ 
and  $x: a < x\ x < b$ 
and  $inj: \bigwedge y. \llbracket a < y; y < b \rrbracket \implies f\ (g\ y) = y$ 
and  $cont: isCont\ g\ x$ 
shows  $DERIV\ g\ x :> inverse\ D$ 
unfolding has-field-derivative-iff
proof (rule LIM-equal2)
show  $0 < min\ (x - a)\ (b - x)$ 
using  $x$  by arith
next
fix  $y$ 
assume  $norm\ (y - x) < min\ (x - a)\ (b - x)$ 
then have  $a < y$  and  $y < b$ 
by (simp-all add: abs-less-iff)
then show  $(g\ y - g\ x) / (y - x) = inverse\ ((f\ (g\ y) - x) / (g\ y - g\ x))$ 
by (simp add: inj)
next
have  $(\lambda z. (f\ z - f\ (g\ x)) / (z - g\ x)) -g\ x \rightarrow D$ 
by (rule der [unfolded has-field-derivative-iff])
then have  $1: (\lambda z. (f\ z - x) / (z - g\ x)) -g\ x \rightarrow D$ 
using  $inj\ x$  by simp
have  $2: \exists d > 0. \forall y. y \neq x \wedge norm\ (y - x) < d \longrightarrow g\ y \neq g\ x$ 
proof (rule exI, safe)
show  $0 < min\ (x - a)\ (b - x)$ 
using  $x$  by simp
next
fix  $y$ 
assume  $norm\ (y - x) < min\ (x - a)\ (b - x)$ 
then have  $y: a < y\ y < b$ 
by (simp-all add: abs-less-iff)
assume  $g\ y = g\ x$ 
then have  $f\ (g\ y) = f\ (g\ x)$  by simp
then have  $y = x$  using  $inj\ y\ x$  by simp
also assume  $y \neq x$ 
finally show False by simp
qed
have  $(\lambda y. (f\ (g\ y) - x) / (g\ y - g\ x)) -x \rightarrow D$ 
using  $cont\ 1\ 2$  by (rule isCont-LIM-compose2)
then show  $(\lambda y. inverse\ ((f\ (g\ y) - x) / (g\ y - g\ x))) -x \rightarrow inverse\ D$ 

```

using *neq* by (*rule tendsto-inverse*)
qed

111.11 Generalized Mean Value Theorem

theorem *GMVT*:

fixes *a b* :: *real*
 assumes *alb*: $a < b$
 and *fc*: $\forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } f \ x$
 and *fd*: $\forall x. a < x \wedge x < b \longrightarrow f \text{ differentiable } (at \ x)$
 and *gc*: $\forall x. a \leq x \wedge x \leq b \longrightarrow \text{isCont } g \ x$
 and *gd*: $\forall x. a < x \wedge x < b \longrightarrow g \text{ differentiable } (at \ x)$
 shows $\exists g'c \ f'c \ c.$
 $DERIV \ g \ c \ :> \ g'c \ \wedge \ DERIV \ f \ c \ :> \ f'c \ \wedge \ a < c \ \wedge \ c < b \ \wedge \ (f \ b - f \ a) * g'c =$
 $(g \ b - g \ a) * f'c$
proof –
 let *?h* = $\lambda x. (f \ b - f \ a) * g \ x - (g \ b - g \ a) * f \ x$
 have $\exists l \ z. a < z \ \wedge \ z < b \ \wedge \ DERIV \ ?h \ z \ :> \ l \ \wedge \ ?h \ b - ?h \ a = (b - a) * l$
proof (*rule MVT*)
 from *assms* show $a < b$ by *simp*
 show *continuous-on* $\{a..b\}$ *?h*
 by (*simp add: continuous-at-imp-continuous-on fc gc*)
 show $\bigwedge x. \llbracket a < x; x < b \rrbracket \Longrightarrow ?h \text{ differentiable } (at \ x)$
 using *fd gd* by *simp*
 qed
 then obtain *l* where $l: \exists z. a < z \ \wedge \ z < b \ \wedge \ DERIV \ ?h \ z \ :> \ l \ \wedge \ ?h \ b - ?h \ a$
 $= (b - a) * l ..$
 then obtain *c* where $c: a < c \ \wedge \ c < b \ \wedge \ DERIV \ ?h \ c \ :> \ l \ \wedge \ ?h \ b - ?h \ a =$
 $(b - a) * l ..$

from *c* have *cint*: $a < c \ \wedge \ c < b$ by *auto*
 then obtain *g'c* where *g'c*: $DERIV \ g \ c \ :> \ g'c$
 using *gd real-differentiable-def* by *blast*
 from *c* have $a < c \ \wedge \ c < b$ by *auto*
 then obtain *f'c* where *f'c*: $DERIV \ f \ c \ :> \ f'c$
 using *fd real-differentiable-def* by *blast*

 from *c* have $DERIV \ ?h \ c \ :> \ l$ by *auto*
 moreover have $DERIV \ ?h \ c \ :> \ g'c * (f \ b - f \ a) - f'c * (g \ b - g \ a)$
 using *g'c f'c* by (*auto intro!: derivative-eq-intros*)
 ultimately have *leq*: $l = g'c * (f \ b - f \ a) - f'c * (g \ b - g \ a)$ by (*rule*
DERIV-unique)

have $?h \ b - ?h \ a = (b - a) * (g'c * (f \ b - f \ a) - f'c * (g \ b - g \ a))$
proof –
 from *c* have $?h \ b - ?h \ a = (b - a) * l$ by *auto*
 also from *leq* have $... = (b - a) * (g'c * (f \ b - f \ a) - f'c * (g \ b - g \ a))$ by
simp
 finally show *?thesis* by *simp*

qed
moreover have $?h\ b - ?h\ a = 0$
proof –
have $?h\ b - ?h\ a =$
 $((f\ b)*(g\ b) - (f\ a)*(g\ b) - (g\ b)*(f\ b) + (g\ a)*(f\ b)) -$
 $((f\ b)*(g\ a) - (f\ a)*(g\ a) - (g\ b)*(f\ a) + (g\ a)*(f\ a))$
by (*simp add: algebra-simps*)
then show $?thesis$ **by** *auto*
qed
ultimately have $(b - a) * (g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a)) = 0$ **by** *auto*
with *alb* **have** $g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a) = 0$ **by** *simp*
then have $g'c * (f\ b - f\ a) = f'c * (g\ b - g\ a)$ **by** *simp*
then have $(f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$ **by** (*simp add: ac-simps*)
with $g'c\ f'c\ cint$ **show** $?thesis$ **by** *auto*
qed

lemma *GMVT'*:

fixes $f\ g :: real \Rightarrow real$
assumes $a < b$
and *isCont-f*: $\bigwedge z. a \leq z \Longrightarrow z \leq b \Longrightarrow isCont\ f\ z$
and *isCont-g*: $\bigwedge z. a \leq z \Longrightarrow z \leq b \Longrightarrow isCont\ g\ z$
and *DERIV-g*: $\bigwedge z. a < z \Longrightarrow z < b \Longrightarrow DERIV\ g\ z :> (g'\ z)$
and *DERIV-f*: $\bigwedge z. a < z \Longrightarrow z < b \Longrightarrow DERIV\ f\ z :> (f'\ z)$
shows $\exists c. a < c \wedge c < b \wedge (f\ b - f\ a) * g'\ c = (g\ b - g\ a) * f'\ c$
proof –
have $\exists g'c\ f'c\ c. DERIV\ g\ c :> g'c \wedge DERIV\ f\ c :> f'c \wedge$
 $a < c \wedge c < b \wedge (f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$
using *assms* **by** (*intro GMVT*) (*force simp: real-differentiable-def*)+
then obtain c **where** $a < c < b \wedge (f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$
using *DERIV-f* *DERIV-g* **by** (*force dest: DERIV-unique*)
then show $?thesis$
by *auto*
qed

111.12 L'Hopitals rule

lemma *isCont-If-ge*:

fixes $a :: 'a :: linorder-topology$
assumes *continuous* (*at-left* a) g **and** $f: (f \longrightarrow g\ a)$ (*at-right* a)
shows *isCont* ($\lambda x. if\ x \leq a\ then\ g\ x\ else\ f\ x$) a (**is** *isCont* $?gf\ a$)
proof –
have $g: (g \longrightarrow g\ a)$ (*at-left* a)
using *assms* *continuous-within* **by** *blast*
show $?thesis$
unfolding *isCont-def* *continuous-within*
proof (*intro filterlim-split-at; simp*)
show $(?gf \longrightarrow g\ a)$ (*at-left* a)
by (*subst filterlim-cong[OF refl refl, where g=g]*) (*simp-all add: eventually-at-filter less-le* g)

show $(?gf \longrightarrow g a) (at\text{-}right\ a)$
by $(subst\ filterlim\text{-}cong[OF\ refl\ refl, \mathbf{where}\ g=f]) (simp\text{-}all\ add: \textit{eventually-at-filter\ less-le}\ f)$
qed
qed

lemma *lhospital-right-0*:

fixes $f0\ g0 :: real \Rightarrow real$
assumes $f0: (f0 \longrightarrow 0) (at\text{-}right\ 0)$
and $g0: (g0 \longrightarrow 0) (at\text{-}right\ 0)$
and *ev*:
 $eventually\ (\lambda x. g0\ x \neq 0) (at\text{-}right\ 0)$
 $eventually\ (\lambda x. g'\ x \neq 0) (at\text{-}right\ 0)$
 $eventually\ (\lambda x. DERIV\ f0\ x :> f'\ x) (at\text{-}right\ 0)$
 $eventually\ (\lambda x. DERIV\ g0\ x :> g'\ x) (at\text{-}right\ 0)$
and *lim*: $filterlim\ (\lambda x. (f'\ x / g'\ x))\ F (at\text{-}right\ 0)$
shows $filterlim\ (\lambda x. f0\ x / g0\ x)\ F (at\text{-}right\ 0)$

proof –

define f **where** $[abs\text{-}def]: f\ x = (if\ x \leq 0\ then\ 0\ else\ f0\ x)$ **for** x
then have $f\ 0 = 0$ **by** *simp*

define g **where** $[abs\text{-}def]: g\ x = (if\ x \leq 0\ then\ 0\ else\ g0\ x)$ **for** x
then have $g\ 0 = 0$ **by** *simp*

have $eventually\ (\lambda x. g0\ x \neq 0 \wedge g'\ x \neq 0 \wedge$
 $DERIV\ f0\ x :> (f'\ x) \wedge DERIV\ g0\ x :> (g'\ x)) (at\text{-}right\ 0)$
using *ev* **by** *eventually-elim auto*
then obtain a **where** $[arith]: 0 < a$
and $g0\text{-}neq\text{-}0: \bigwedge x. 0 < x \Longrightarrow x < a \Longrightarrow g0\ x \neq 0$
and $g'\text{-}neq\text{-}0: \bigwedge x. 0 < x \Longrightarrow x < a \Longrightarrow g'\ x \neq 0$
and $f0: \bigwedge x. 0 < x \Longrightarrow x < a \Longrightarrow DERIV\ f0\ x :> (f'\ x)$
and $g0: \bigwedge x. 0 < x \Longrightarrow x < a \Longrightarrow DERIV\ g0\ x :> (g'\ x)$
unfolding *eventually-at* **by** $(auto\ simp: dist\text{-}real\text{-}def)$

have $g\text{-}neq\text{-}0: \bigwedge x. 0 < x \Longrightarrow x < a \Longrightarrow g\ x \neq 0$
using $g0\text{-}neq\text{-}0$ **by** $(simp\ add: g\text{-}def)$

have $f: DERIV\ f\ x :> (f'\ x)$ **if** $x: 0 < x < a$ **for** x
using *that*
by $(intro\ DERIV\text{-}cong\text{-}ev[THEN\ iffD1, OF\ \text{-}\ \text{-}\ f0[OF\ x]])$
 $(auto\ simp: f\text{-}def\ eventually\text{-}nhds\text{-}metric\ dist\text{-}real\text{-}def\ intro!: exI[of\ \text{-}\ x])$

have $g: DERIV\ g\ x :> (g'\ x)$ **if** $x: 0 < x < a$ **for** x
using *that*
by $(intro\ DERIV\text{-}cong\text{-}ev[THEN\ iffD1, OF\ \text{-}\ \text{-}\ g0[OF\ x]])$
 $(auto\ simp: g\text{-}def\ eventually\text{-}nhds\text{-}metric\ dist\text{-}real\text{-}def\ intro!: exI[of\ \text{-}\ x])$

have *isCont* $f\ 0$
unfolding *f-def* **by** $(intro\ isCont\text{-}If\text{-}ge\ f0\ continuous\text{-}const)$

have $isCont\ g\ 0$
unfolding $g\text{-def}$ **by** (*intro isCont-If-ge g-0 continuous-const*)

have $\exists \zeta. \forall x \in \{0 <..< a\}. 0 < \zeta\ x \wedge \zeta\ x < x \wedge f\ x / g\ x = f'\ (\zeta\ x) / g'\ (\zeta\ x)$
proof (*rule bchoice, rule ballI*)

fix x
assume $x \in \{0 <..< a\}$
then have $x[arith]: 0 < x\ x < a$ **by** *auto*
with $g'\text{-neq-0}\ g\text{-neq-0}\ \langle g\ 0 = 0 \rangle$ **have** $g': \bigwedge x. 0 < x \implies x < a \implies 0 \neq g'\ x$
 $g\ 0 \neq g\ x$
by *auto*
have $\bigwedge x. 0 \leq x \implies x < a \implies isCont\ f\ x$
using $\langle isCont\ f\ 0 \rangle\ f$ **by** (*auto intro: DERIV-isCont simp: le-less*)
moreover have $\bigwedge x. 0 \leq x \implies x < a \implies isCont\ g\ x$
using $\langle isCont\ g\ 0 \rangle\ g$ **by** (*auto intro: DERIV-isCont simp: le-less*)
ultimately have $\exists c. 0 < c \wedge c < x \wedge (f\ x - f\ 0) * g'\ c = (g\ x - g\ 0) * f'\ c$
using $f\ g\ \langle x < a \rangle$ **by** (*intro GMVT'*) *auto*
then obtain c **where** $*$: $0 < c\ c < x\ (f\ x - f\ 0) * g'\ c = (g\ x - g\ 0) * f'\ c$
by *blast*
moreover
from $*\ g'(1)[of\ c]\ g'(2)$ **have** $(f\ x - f\ 0) / (g\ x - g\ 0) = f'\ c / g'\ c$
by (*simp add: field-simps*)
ultimately show $\exists y. 0 < y \wedge y < x \wedge f\ x / g\ x = f'\ y / g'\ y$
using $\langle f\ 0 = 0 \rangle\ \langle g\ 0 = 0 \rangle$ **by** (*auto intro!: exI[of - c]*)

qed
then obtain ζ **where** $\forall x \in \{0 <..< a\}. 0 < \zeta\ x \wedge \zeta\ x < x \wedge f\ x / g\ x = f'\ (\zeta\ x) / g'\ (\zeta\ x)$..
then have ζ : *eventually* $(\lambda x. 0 < \zeta\ x \wedge \zeta\ x < x \wedge f\ x / g\ x = f'\ (\zeta\ x) / g'\ (\zeta\ x))$ (*at-right 0*)
unfolding *eventually-at* **by** (*intro exI[of - a]*) (*auto simp: dist-real-def*)
moreover
from ζ **have** *eventually* $(\lambda x. norm\ (\zeta\ x) \leq x)$ (*at-right 0*)
by *eventually-elim auto*
then have $((\lambda x. norm\ (\zeta\ x)) \longrightarrow 0)$ (*at-right 0*)
by (*rule-tac real-tendsto-sandwich[where f= $\lambda x. 0$ and h= $\lambda x. x$] auto*)
then have $(\zeta \longrightarrow 0)$ (*at-right 0*)
by (*rule tendsto-norm-zero-cancel*)
with ζ **have** *filterlim* ζ (*at-right 0*) (*at-right 0*)
by (*auto elim!: eventually-mono simp: filterlim-at*)
from *this lim* **have** *filterlim* $(\lambda t. f'\ (\zeta\ t) / g'\ (\zeta\ t))\ F$ (*at-right 0*)
by (*rule-tac filterlim-compose[of - - - ζ]*)
ultimately have *filterlim* $(\lambda t. f\ t / g\ t)\ F$ (*at-right 0*) (**is** $?P$)
by (*rule-tac filterlim-cong[THEN iffD1, OF refl refl]*)
(auto elim: eventually-mono)
also have $?P \longleftrightarrow ?thesis$
by (*rule filterlim-cong*) (*auto simp: f-def g-def eventually-at-filter*)
finally show $?thesis$.

qed

lemma *lhospital-right*:

$(f \longrightarrow 0) (at\text{-}right\ x) \implies (g \longrightarrow 0) (at\text{-}right\ x) \implies$
 $eventually\ (\lambda x. g\ x \neq 0) (at\text{-}right\ x) \implies$
 $eventually\ (\lambda x. g'\ x \neq 0) (at\text{-}right\ x) \implies$
 $eventually\ (\lambda x. DERIV\ f\ x\ :\>\ f'\ x) (at\text{-}right\ x) \implies$
 $eventually\ (\lambda x. DERIV\ g\ x\ :\>\ g'\ x) (at\text{-}right\ x) \implies$
 $filterlim\ (\lambda x. (f'\ x / g'\ x))\ F\ (at\text{-}right\ x) \implies$
 $filterlim\ (\lambda x. f\ x / g\ x)\ F\ (at\text{-}right\ x)$
for $x :: real$
unfolding *eventually-at-right-to-0*[of - x] *filterlim-at-right-to-0*[of - - x] *DERIV-shift*
by (*rule lhospital-right-0*)

lemma *lhospital-left*:

$(f \longrightarrow 0) (at\text{-}left\ x) \implies (g \longrightarrow 0) (at\text{-}left\ x) \implies$
 $eventually\ (\lambda x. g\ x \neq 0) (at\text{-}left\ x) \implies$
 $eventually\ (\lambda x. g'\ x \neq 0) (at\text{-}left\ x) \implies$
 $eventually\ (\lambda x. DERIV\ f\ x\ :\>\ f'\ x) (at\text{-}left\ x) \implies$
 $eventually\ (\lambda x. DERIV\ g\ x\ :\>\ g'\ x) (at\text{-}left\ x) \implies$
 $filterlim\ (\lambda x. (f'\ x / g'\ x))\ F\ (at\text{-}left\ x) \implies$
 $filterlim\ (\lambda x. f\ x / g\ x)\ F\ (at\text{-}left\ x)$
for $x :: real$
unfolding *eventually-at-left-to-right* *filterlim-at-left-to-right* *DERIV-mirror*
by (*rule lhospital-right*[**where** $f' = \lambda x. - f' (- x)$]) (*auto simp: DERIV-mirror*)

lemma *lhospital*:

$(f \longrightarrow 0) (at\ x) \implies (g \longrightarrow 0) (at\ x) \implies$
 $eventually\ (\lambda x. g\ x \neq 0) (at\ x) \implies$
 $eventually\ (\lambda x. g'\ x \neq 0) (at\ x) \implies$
 $eventually\ (\lambda x. DERIV\ f\ x\ :\>\ f'\ x) (at\ x) \implies$
 $eventually\ (\lambda x. DERIV\ g\ x\ :\>\ g'\ x) (at\ x) \implies$
 $filterlim\ (\lambda x. (f'\ x / g'\ x))\ F\ (at\ x) \implies$
 $filterlim\ (\lambda x. f\ x / g\ x)\ F\ (at\ x)$
for $x :: real$
unfolding *eventually-at-split* *filterlim-at-split*
by (*auto intro!*: *lhospital-right*[of $f\ x\ g\ g'\ f'$] *lhospital-left*[of $f\ x\ g\ g'\ f'$])

lemma *lhospital-right-0-at-top*:

fixes $f\ g :: real \Rightarrow real$
assumes $g\ 0$: *LIM* x *at-right* 0 . $g\ x\ :\>\ at\text{-}top$
and *ev*:
 $eventually\ (\lambda x. g'\ x \neq 0) (at\text{-}right\ 0)$
 $eventually\ (\lambda x. DERIV\ f\ x\ :\>\ f'\ x) (at\text{-}right\ 0)$
 $eventually\ (\lambda x. DERIV\ g\ x\ :\>\ g'\ x) (at\text{-}right\ 0)$
and *lim*: $((\lambda x. (f'\ x / g'\ x)) \longrightarrow x) (at\text{-}right\ 0)$
shows $((\lambda x. f\ x / g\ x) \longrightarrow x) (at\text{-}right\ 0)$
unfolding *tendsto-iff*

proof *safe*
fix $e :: \text{real}$
assume $0 < e$
with $\text{lim}[\text{unfolded tendsto-iff}, \text{rule-format}, \text{of } e / 4]$
have *eventually* $(\lambda t. \text{dist } (f' t / g' t) x < e / 4)$ *(at-right 0)*
by *simp*
from *eventually-conj*[*OF* *eventually-conj*[*OF* *ev*(1) *ev*(2)] *eventually-conj*[*OF* *ev*(3) *this*]]
obtain a **where** [*arith*]: $0 < a$
and $g'\text{-neq-0}$: $\bigwedge x. 0 < x \implies x < a \implies g' x \neq 0$
and $f0$: $\bigwedge x. 0 < x \implies x \leq a \implies \text{DERIV } f x :> (f' x)$
and $g0$: $\bigwedge x. 0 < x \implies x \leq a \implies \text{DERIV } g x :> (g' x)$
and Df : $\bigwedge t. 0 < t \implies t < a \implies \text{dist } (f' t / g' t) x < e / 4$
unfolding *eventually-at-le* **by** (*auto simp: dist-real-def*)

from Df **have** *eventually* $(\lambda t. t < a)$ *(at-right 0)* *eventually* $(\lambda t :: \text{real}. 0 < t)$ *(at-right 0)*
unfolding *eventually-at* **by** (*auto intro!: exI*[*of* - *a*] *simp: dist-real-def*)

moreover
have *eventually* $(\lambda t. 0 < g t)$ *(at-right 0)* *eventually* $(\lambda t. g a < g t)$ *(at-right 0)*
using $g\text{-0}$ **by** (*auto elim: eventually-mono simp: filterlim-at-top-dense*)

moreover
have inv-g : $((\lambda x. \text{inverse } (g x)) \longrightarrow 0)$ *(at-right 0)*
using *tendsto-inverse-0 filterlim-mono*[*OF* $g\text{-0}$ *at-top-le-at-infinity order-refl*]
by (*rule filterlim-compose*)
then **have** $((\lambda x. \text{norm } (1 - g a * \text{inverse } (g x))) \longrightarrow \text{norm } (1 - g a * 0))$ *(at-right 0)*
by (*intro tendsto-intros*)
then **have** $((\lambda x. \text{norm } (1 - g a / g x)) \longrightarrow 1)$ *(at-right 0)*
by (*simp add: inverse-eq-divide*)
from *this*[*unfolded tendsto-iff, rule-format, of 1*]
have *eventually* $(\lambda x. \text{norm } (1 - g a / g x) < 2)$ *(at-right 0)*
by (*auto elim!: eventually-mono simp: dist-real-def*)

moreover
from inv-g **have** $((\lambda t. \text{norm } ((f a - x * g a) * \text{inverse } (g t))) \longrightarrow \text{norm } ((f a - x * g a) * 0))$ *(at-right 0)*
by (*intro tendsto-intros*)
then **have** $((\lambda t. \text{norm } (f a - x * g a) / \text{norm } (g t)) \longrightarrow 0)$ *(at-right 0)*
by (*simp add: inverse-eq-divide*)
from *this*[*unfolded tendsto-iff, rule-format, of e / 2*] $\langle 0 < e \rangle$
have *eventually* $(\lambda t. \text{norm } (f a - x * g a) / \text{norm } (g t) < e / 2)$ *(at-right 0)*
by (*auto simp: dist-real-def*)

ultimately show *eventually* $(\lambda t. \text{dist } (f t / g t) x < e)$ *(at-right 0)*
proof *eventually-elim*

fix t **assume** $t[\text{arith}]$: $0 < t \wedge t < a \wedge g a < g t \wedge 0 < g t$
assume ineq : $\text{norm } (1 - g a / g t) < 2 \text{ norm } (f a - x * g a) / \text{norm } (g t) < e / 2$
have $\exists y. t < y \wedge y < a \wedge (g a - g t) * f' y = (f a - f t) * g' y$
using $f0\ g0\ t(1,2)$ **by** (*intro GMVT'*) (*force intro!: DERIV-isCont*)
then obtain y **where** $[\text{arith}]$: $t < y \wedge y < a$
and $D\text{-eq0}$: $(g a - g t) * f' y = (f a - f t) * g' y$
by *blast*
from $D\text{-eq0}$ **have** $D\text{-eq}$: $(f t - f a) / (g t - g a) = f' y / g' y$
using $\langle g a < g t \rangle\ g'\text{-neg-0}[\text{of } y]$ **by** (*auto simp add: field-simps*)

have $*$: $f t / g t - x = ((f t - f a) / (g t - g a) - x) * (1 - g a / g t) + (f a - x * g a) / g t$
by (*simp add: field-simps*)
have $\text{norm } (f t / g t - x) \leq \text{norm } (((f t - f a) / (g t - g a) - x) * (1 - g a / g t)) + \text{norm } ((f a - x * g a) / g t)$
unfolding $*$ **by** (*rule norm-triangle-ineq*)
also have $\dots = \text{dist } (f' y / g' y) x * \text{norm } (1 - g a / g t) + \text{norm } (f a - x * g a) / \text{norm } (g t)$
by (*simp add: abs-mult D-eq dist-real-def*)
also have $\dots < (e / 4) * 2 + e / 2$
using $\text{ineq } Df[\text{of } y] \langle 0 < e \rangle$ **by** (*intro add-le-less-mono mult-mono*) *auto*
finally show $\text{dist } (f t / g t) x < e$
by (*simp add: dist-real-def*)
qed
qed

lemma *lhopital-right-at-top*:

$LIM\ x\ \text{at-right}\ x. (g :: \text{real} \Rightarrow \text{real})\ x\ :\>\ \text{at-top} \Longrightarrow$
 $\text{eventually } (\lambda x. g' x \neq 0)\ (\text{at-right}\ x) \Longrightarrow$
 $\text{eventually } (\lambda x. DERIV\ f\ x\ :\>\ f' x)\ (\text{at-right}\ x) \Longrightarrow$
 $\text{eventually } (\lambda x. DERIV\ g\ x\ :\>\ g' x)\ (\text{at-right}\ x) \Longrightarrow$
 $((\lambda x. (f' x / g' x)) \longrightarrow y)\ (\text{at-right}\ x) \Longrightarrow$
 $((\lambda x. f x / g x) \longrightarrow y)\ (\text{at-right}\ x)$
unfolding *eventually-at-right-to-0*[*of - x*] *filterlim-at-right-to-0*[*of - - x*] *DERIV-shift*
by (*rule lhopital-right-0-at-top*)

lemma *lhopital-left-at-top*:

$LIM\ x\ \text{at-left}\ x. g\ x\ :\>\ \text{at-top} \Longrightarrow$
 $\text{eventually } (\lambda x. g' x \neq 0)\ (\text{at-left}\ x) \Longrightarrow$
 $\text{eventually } (\lambda x. DERIV\ f\ x\ :\>\ f' x)\ (\text{at-left}\ x) \Longrightarrow$
 $\text{eventually } (\lambda x. DERIV\ g\ x\ :\>\ g' x)\ (\text{at-left}\ x) \Longrightarrow$
 $((\lambda x. (f' x / g' x)) \longrightarrow y)\ (\text{at-left}\ x) \Longrightarrow$
 $((\lambda x. f x / g x) \longrightarrow y)\ (\text{at-left}\ x)$
for $x :: \text{real}$
unfolding *eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror*

by (rule lhospital-right-at-top[where $f' = \lambda x. - f'(-x)$]) (auto simp: DERIV-mirror)

lemma lhospital-at-top:

LIM x at x . ($g :: \text{real} \Rightarrow \text{real}$) $x \text{ :> at-top} \Longrightarrow$
 eventually $(\lambda x. g' x \neq 0)$ (at x) \Longrightarrow
 eventually $(\lambda x. \text{DERIV } f x \text{ :> } f' x)$ (at x) \Longrightarrow
 eventually $(\lambda x. \text{DERIV } g x \text{ :> } g' x)$ (at x) \Longrightarrow
 $((\lambda x. (f' x / g' x)) \longrightarrow y)$ (at x) \Longrightarrow
 $((\lambda x. f x / g x) \longrightarrow y)$ (at x)

unfolding eventually-at-split filterlim-at-split

by (auto intro!: lhospital-right-at-top[of $g x g' f f'$] lhospital-left-at-top[of $g x g' f f'$])

lemma lhospital-at-top-at-top:

fixes $f g :: \text{real} \Rightarrow \text{real}$

assumes $g\text{-}0$: LIM x at-top. $g x \text{ :> at-top}$

and g' : eventually $(\lambda x. g' x \neq 0)$ at-top

and Df : eventually $(\lambda x. \text{DERIV } f x \text{ :> } f' x)$ at-top

and Dg : eventually $(\lambda x. \text{DERIV } g x \text{ :> } g' x)$ at-top

and lim : $((\lambda x. (f' x / g' x)) \longrightarrow x)$ at-top

shows $((\lambda x. f x / g x) \longrightarrow x)$ at-top

unfolding filterlim-at-top-to-right

proof (rule lhospital-right-0-at-top)

let $?F = \lambda x. f$ (inverse x)

let $?G = \lambda x. g$ (inverse x)

let $?R = \text{at-right } (0 :: \text{real})$

let $?D = \lambda f' x. f' (inverse x) * - (inverse x \wedge \text{Suc } (\text{Suc } 0))$

show LIM x $?R$. $?G x \text{ :> at-top}$

using $g\text{-}0$ **unfolding** filterlim-at-top-to-right .

show eventually $(\lambda x. \text{DERIV } ?G x \text{ :> } ?D g' x)$ $?R$

unfolding eventually-at-right-to-top

using Dg eventually-ge-at-top[where $c=1$]

by eventually-elim (rule derivative-eq-intros DERIV-chain'[where $f = \text{inverse}$])

| simp)+

show eventually $(\lambda x. \text{DERIV } ?F x \text{ :> } ?D f' x)$ $?R$

unfolding eventually-at-right-to-top

using Df eventually-ge-at-top[where $c=1$]

by eventually-elim (rule derivative-eq-intros DERIV-chain'[where $f = \text{inverse}$])

| simp)+

show eventually $(\lambda x. ?D g' x \neq 0)$ $?R$

unfolding eventually-at-right-to-top

using g' eventually-ge-at-top[where $c=1$]

by eventually-elim auto

show $((\lambda x. ?D f' x / ?D g' x) \longrightarrow x)$ $?R$

unfolding filterlim-at-right-to-top

apply (intro filterlim-cong[THEN iffD2, OF refl refl - lim])

using eventually-ge-at-top[where $c=1$]

by eventually-elim simp

qed

lemma *lhospital-right-at-top-at-top*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $f\text{-}0$: $LIM\ x\ \text{at-right}\ a.\ f\ x\ \text{:>}\ \text{at-top}$

assumes $g\text{-}0$: $LIM\ x\ \text{at-right}\ a.\ g\ x\ \text{:>}\ \text{at-top}$

and ev :

eventually $(\lambda x.\ DERIV\ f\ x\ \text{:>}\ f'\ x)\ (\text{at-right}\ a)$

eventually $(\lambda x.\ DERIV\ g\ x\ \text{:>}\ g'\ x)\ (\text{at-right}\ a)$

and lim : *filterlim* $(\lambda x.\ (f'\ x / g'\ x))\ \text{at-top}\ (\text{at-right}\ a)$

shows *filterlim* $(\lambda x.\ f\ x / g\ x)\ \text{at-top}\ (\text{at-right}\ a)$

proof –

from lim **have** pos : *eventually* $(\lambda x.\ f'\ x / g'\ x > 0)\ (\text{at-right}\ a)$

unfolding *filterlim-at-top-dense* **by** *blast*

have $((\lambda x.\ g\ x / f\ x) \longrightarrow 0)\ (\text{at-right}\ a)$

proof (*rule* *lhospital-right-at-top*)

from pos **show** *eventually* $(\lambda x.\ f'\ x \neq 0)\ (\text{at-right}\ a)$ **by** *eventually-elim auto*

from *tendsto-inverse-0-at-top[OF* lim *]*

show $((\lambda x.\ g'\ x / f'\ x) \longrightarrow 0)\ (\text{at-right}\ a)$ **by** *simp*

qed *fact+*

moreover **from** $f\text{-}0\ g\text{-}0$

have *eventually* $(\lambda x.\ f\ x > 0)\ (\text{at-right}\ a)$ *eventually* $(\lambda x.\ g\ x > 0)\ (\text{at-right}\ a)$

unfolding *filterlim-at-top-dense* **by** *blast+*

hence *eventually* $(\lambda x.\ g\ x / f\ x > 0)\ (\text{at-right}\ a)$ **by** *eventually-elim simp*

ultimately **have** *filterlim* $(\lambda x.\ \text{inverse}\ (g\ x / f\ x))\ \text{at-top}\ (\text{at-right}\ a)$

by (*rule* *filterlim-inverse-at-top*)

thus *?thesis* **by** *simp*

qed

lemma *lhospital-right-at-top-at-bot*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes $f\text{-}0$: $LIM\ x\ \text{at-right}\ a.\ f\ x\ \text{:>}\ \text{at-top}$

assumes $g\text{-}0$: $LIM\ x\ \text{at-right}\ a.\ g\ x\ \text{:>}\ \text{at-bot}$

and ev :

eventually $(\lambda x.\ DERIV\ f\ x\ \text{:>}\ f'\ x)\ (\text{at-right}\ a)$

eventually $(\lambda x.\ DERIV\ g\ x\ \text{:>}\ g'\ x)\ (\text{at-right}\ a)$

and lim : *filterlim* $(\lambda x.\ (f'\ x / g'\ x))\ \text{at-bot}\ (\text{at-right}\ a)$

shows *filterlim* $(\lambda x.\ f\ x / g\ x)\ \text{at-bot}\ (\text{at-right}\ a)$

proof –

from $ev(2)$ **have** ev' : *eventually* $(\lambda x.\ DERIV\ (\lambda x.\ -g\ x)\ x\ \text{:>}\ -g'\ x)\ (\text{at-right}\ a)$

by *eventually-elim* (*auto intro: derivative-intros*)

have *filterlim* $(\lambda x.\ f\ x / (-g\ x))\ \text{at-top}\ (\text{at-right}\ a)$

by (*rule* *lhospital-right-at-top-at-top*[**where** $f' = f'$ **and** $g' = \lambda x.\ -g'\ x$])

(*insert* $assms\ ev'$, *auto simp: filterlim-uminus-at-bot*)

hence *filterlim* $(\lambda x.\ -(f\ x / g\ x))\ \text{at-top}\ (\text{at-right}\ a)$ **by** *simp*

thus *?thesis* **by** (*simp add: filterlim-uminus-at-bot*)

qed

lemma *lhospital-left-at-top-at-top*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$
assumes $f\text{-}0$: $LIM\ x\ \text{at-left}\ a.\ f\ x\ :>\ \text{at-top}$
assumes $g\text{-}0$: $LIM\ x\ \text{at-left}\ a.\ g\ x\ :>\ \text{at-top}$
and ev :
 eventually $(\lambda x.\ DERIV\ f\ x\ :>\ f'\ x)\ (\text{at-left}\ a)$
 eventually $(\lambda x.\ DERIV\ g\ x\ :>\ g'\ x)\ (\text{at-left}\ a)$
and lim : *filterlim* $(\lambda x.\ (f'\ x / g'\ x))\ \text{at-top}\ (\text{at-left}\ a)$
shows *filterlim* $(\lambda x.\ f\ x / g\ x)\ \text{at-top}\ (\text{at-left}\ a)$
by (*insert assms, unfold eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror,*
 rule lhopital-right-at-top-at-top[where $f' = \lambda x.\ -\ f'\ (-\ x)$])
 (*insert assms, auto simp: DERIV-mirror*)

lemma *lhopital-left-at-top-at-bot*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$
assumes $f\text{-}0$: $LIM\ x\ \text{at-left}\ a.\ f\ x\ :>\ \text{at-top}$
assumes $g\text{-}0$: $LIM\ x\ \text{at-left}\ a.\ g\ x\ :>\ \text{at-bot}$
and ev :
 eventually $(\lambda x.\ DERIV\ f\ x\ :>\ f'\ x)\ (\text{at-left}\ a)$
 eventually $(\lambda x.\ DERIV\ g\ x\ :>\ g'\ x)\ (\text{at-left}\ a)$
and lim : *filterlim* $(\lambda x.\ (f'\ x / g'\ x))\ \text{at-bot}\ (\text{at-left}\ a)$
shows *filterlim* $(\lambda x.\ f\ x / g\ x)\ \text{at-bot}\ (\text{at-left}\ a)$
by (*insert assms, unfold eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror,*
 rule lhopital-right-at-top-at-bot[where $f' = \lambda x.\ -\ f'\ (-\ x)$])
 (*insert assms, auto simp: DERIV-mirror*)

lemma *lhopital-at-top-at-top*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$
assumes $f\text{-}0$: $LIM\ x\ \text{at}\ a.\ f\ x\ :>\ \text{at-top}$
assumes $g\text{-}0$: $LIM\ x\ \text{at}\ a.\ g\ x\ :>\ \text{at-top}$
and ev :
 eventually $(\lambda x.\ DERIV\ f\ x\ :>\ f'\ x)\ (\text{at}\ a)$
 eventually $(\lambda x.\ DERIV\ g\ x\ :>\ g'\ x)\ (\text{at}\ a)$
and lim : *filterlim* $(\lambda x.\ (f'\ x / g'\ x))\ \text{at-top}\ (\text{at}\ a)$
shows *filterlim* $(\lambda x.\ f\ x / g\ x)\ \text{at-top}\ (\text{at}\ a)$
using *assms unfolding eventually-at-split filterlim-at-split*
by (*auto intro!*: *lhopital-right-at-top-at-top[of $f\ a\ g\ f'\ g'$]*
 lhopital-left-at-top-at-top[of $f\ a\ g\ f'\ g'$])

lemma *lhopital-at-top-at-bot*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$
assumes $f\text{-}0$: $LIM\ x\ \text{at}\ a.\ f\ x\ :>\ \text{at-top}$
assumes $g\text{-}0$: $LIM\ x\ \text{at}\ a.\ g\ x\ :>\ \text{at-bot}$
and ev :
 eventually $(\lambda x.\ DERIV\ f\ x\ :>\ f'\ x)\ (\text{at}\ a)$
 eventually $(\lambda x.\ DERIV\ g\ x\ :>\ g'\ x)\ (\text{at}\ a)$
and lim : *filterlim* $(\lambda x.\ (f'\ x / g'\ x))\ \text{at-bot}\ (\text{at}\ a)$
shows *filterlim* $(\lambda x.\ f\ x / g\ x)\ \text{at-bot}\ (\text{at}\ a)$

```

using assms unfolding eventually-at-split filterlim-at-split
by (auto intro!: lhospital-right-at-top-at-bot[of f a g f' g']
      lhospital-left-at-top-at-bot[of f a g f' g'])

```

end

112 Nth Roots of Real Numbers

```

theory NthRoot
  imports Deriv
begin

```

112.1 Existence of Nth Root

Existence follows from the Intermediate Value Theorem

lemma *realpow-pos-nth*:

fixes $a :: \text{real}$

assumes $n: 0 < n$

and $a: 0 < a$

shows $\exists r > 0. r^n = a$

proof –

have $\exists r \geq 0. r \leq (\max 1 a) \wedge r^n = a$

proof (*rule IVT*)

show $0^n \leq a$

using $n a$ **by** (*simp add: power-0-left*)

show $0 \leq \max 1 a$

by *simp*

from n **have** $n1: 1 \leq n$

by *simp*

have $a \leq \max 1 a^1$

by *simp*

also have $\max 1 a^1 \leq \max 1 a^n$

using $n1$ **by** (*rule power-increasing*) *simp*

finally show $a \leq \max 1 a^n$.

show $\forall r. 0 \leq r \wedge r \leq \max 1 a \longrightarrow \text{isCont } (\lambda x. x^n) r$

by *simp*

qed

then obtain r **where** $r: 0 \leq r \wedge r^n = a$

by *fast*

with $n a$ **have** $r \neq 0$

by (*auto simp add: power-0-left*)

with r **have** $0 < r \wedge r^n = a$

by *simp*

then show *?thesis ..*

qed

lemma *realpow-pos-nth2*: $(0 :: \text{real}) < a \implies \exists r > 0. r^{\text{Suc } n} = a$

by (blast intro: realpow-pos-nth)

Uniqueness of nth positive root.

lemma *realpow-pos-nth-unique*: $0 < n \implies 0 < a \implies \exists!r. 0 < r \wedge r^n = a$ for
 $a :: real$

by (auto intro!: realpow-pos-nth simp: power-eq-iff-eq-base)

112.2 Nth Root

We define roots of negative reals such that $root\ n\ (-x) = -\ root\ n\ x$. This allows us to omit side conditions from many theorems.

lemma *inj-sgn-power*:

assumes $0 < n$

shows *inj* $(\lambda y. sgn\ y * |y|^n :: real)$

(is *inj* ?f)

proof (rule *injI*)

have $x: (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b) \implies a \neq b$ for $a\ b :: real$

by auto

fix $x\ y$

assume ?f $x = ?f\ y$

with *power-eq-iff-eq-base*[of $n\ |x|\ |y|$] $\langle 0 < n \rangle$ show $x = y$

by (cases rule: *linorder-cases*[of $0\ x$, case-product *linorder-cases*[of $0\ y$]])
 (simp-all add: x)

qed

lemma *sgn-power-injE*:

$sgn\ a * |a|^n = x \implies x = sgn\ b * |b|^n \implies 0 < n \implies a = b$

for $a\ b :: real$

using *inj-sgn-power*[THEN *injD*, of $n\ a\ b$] by *simp*

definition *root* :: $nat \Rightarrow real \Rightarrow real$

where $root\ n\ x = (if\ n = 0\ then\ 0\ else\ the-inv\ (\lambda y. sgn\ y * |y|^n)\ x)$

lemma *root-0* [*simp*]: $root\ 0\ x = 0$

by (*simp* add: *root-def*)

lemma *root-sgn-power*: $0 < n \implies root\ n\ (sgn\ y * |y|^n) = y$

using *the-inv-f-f*[OF *inj-sgn-power*] by (*simp* add: *root-def*)

lemma *sgn-power-root*:

assumes $0 < n$

shows $sgn\ (root\ n\ x) * |(root\ n\ x)|^n = x$

(is ?f $(root\ n\ x) = x$)

proof (cases $x = 0$)

case *True*

with *assms* *root-sgn-power*[of $n\ 0$] show ?thesis

by *simp*

next


```

case False
with realpow-pos-nth[OF  $\langle 0 < n \rangle$ , of  $|x|$ ]
obtain  $r$  where  $0 < r$   $r \wedge n = |x|$ 
  by auto
with  $\langle x \neq 0 \rangle$  have  $S: x \in \text{range } ?f$ 
  by (intro image-eqI[of - - sgn  $x * r$ ])
    (auto simp: abs-mult sgn-mult power-mult-distrib abs-sgn-eq mult-sgn-abs)
from  $\langle 0 < n \rangle$  f-the-inv-into-f[OF inj-sgn-power[OF  $\langle 0 < n \rangle$ ] this] show ?thesis
  by (simp add: root-def)
qed

```

```

lemma split-root:  $P (\text{root } n \ x) \longleftrightarrow (n = 0 \longrightarrow P \ 0) \wedge (0 < n \longrightarrow (\forall y. \text{sgn } y * |y| \wedge n = x \longrightarrow P \ y))$ 
proof (cases  $n = 0$ )
  case True
    then show ?thesis by simp
  next
    case False
      then show ?thesis
        by simp (metis root-sgn-power sgn-power-root)
qed

```

```

lemma real-root-zero [simp]:  $\text{root } n \ 0 = 0$ 
  by (simp split: split-root add: sgn-zero-iff)

```

```

lemma real-root-minus:  $\text{root } n \ (- \ x) = - \ \text{root } n \ x$ 
  by (clarsimp split: split-root elim!: sgn-power-injE simp: sgn-minus)

```

```

lemma real-root-less-mono:  $0 < n \implies x < y \implies \text{root } n \ x < \text{root } n \ y$ 
proof (clarsimp split: split-root)
  have  $*$ :  $0 < b \implies a < 0 \implies \neg a > b$  for  $a \ b :: \text{real}$ 
    by auto
  fix  $a \ b :: \text{real}$ 
  assume  $0 < n$   $\text{sgn } a * |a| \wedge n < \text{sgn } b * |b| \wedge n$ 
  then show  $a < b$ 
    using power-less-imp-less-base[of  $a \ n \ b$ ]
      power-less-imp-less-base[of -  $b \ n - a$ ]
    by (simp add: sgn-real-def * [of  $a \wedge n - ((- \ b) \wedge n)$ ]  

      split: if-split-asm)
qed

```

```

lemma real-root-gt-zero:  $0 < n \implies 0 < x \implies 0 < \text{root } n \ x$ 
  using real-root-less-mono[of  $n \ 0 \ x$ ] by simp

```

```

lemma real-root-ge-zero:  $0 \leq x \implies 0 \leq \text{root } n \ x$ 
  using real-root-gt-zero[of  $n \ x$ ]
  by (cases  $n = 0$ ) (auto simp add: le-less)

```

```

lemma real-root-pow-pos:  $0 < n \implies 0 < x \implies \text{root } n \ x \wedge n = x$ 

```

using *sgn-power-root*[of n x] *real-root-gt-zero*[of n x] **by** *simp*

lemma *real-root-pow-pos2* [*simp*]: $0 < n \implies 0 \leq x \implies \text{root } n \ x \ ^n = x$
by (*auto simp add: order-le-less real-root-pow-pos*)

lemma *sgn-root*: $0 < n \implies \text{sgn} (\text{root } n \ x) = \text{sgn } x$
by (*auto split: split-root simp: sgn-real-def*)

lemma *odd-real-root-pow*: $\text{odd } n \implies \text{root } n \ x \ ^n = x$
using *sgn-power-root*[of n x]
by (*simp add: odd-pos sgn-real-def split: if-split-asm*)

lemma *real-root-power-cancel*: $0 < n \implies 0 \leq x \implies \text{root } n \ (x \ ^n) = x$
using *root-sgn-power*[of n x] **by** (*auto simp add: le-less power-0-left*)

lemma *odd-real-root-power-cancel*: $\text{odd } n \implies \text{root } n \ (x \ ^n) = x$
using *root-sgn-power*[of n x]
by (*simp add: odd-pos sgn-real-def power-0-left split: if-split-asm*)

lemma *real-root-pos-unique*: $0 < n \implies 0 \leq y \implies y \ ^n = x \implies \text{root } n \ x = y$
using *root-sgn-power*[of n y] **by** (*auto simp add: le-less power-0-left*)

lemma *odd-real-root-unique*: $\text{odd } n \implies y \ ^n = x \implies \text{root } n \ x = y$
by (*erule subst, rule odd-real-root-power-cancel*)

lemma *real-root-one* [*simp*]: $0 < n \implies \text{root } n \ 1 = 1$
by (*simp add: real-root-pos-unique*)

Root function is strictly monotonic, hence injective.

lemma *real-root-le-mono*: $0 < n \implies x \leq y \implies \text{root } n \ x \leq \text{root } n \ y$
by (*auto simp add: order-le-less real-root-less-mono*)

lemma *real-root-less-iff* [*simp*]: $0 < n \implies \text{root } n \ x < \text{root } n \ y \longleftrightarrow x < y$
by (*cases $x < y$*) (*simp-all add: real-root-less-mono linorder-not-less real-root-le-mono*)

lemma *real-root-le-iff* [*simp*]: $0 < n \implies \text{root } n \ x \leq \text{root } n \ y \longleftrightarrow x \leq y$
by (*cases $x \leq y$*) (*simp-all add: real-root-le-mono linorder-not-le real-root-less-mono*)

lemma *real-root-eq-iff* [*simp*]: $0 < n \implies \text{root } n \ x = \text{root } n \ y \longleftrightarrow x = y$
by (*simp add: order-eq-iff*)

lemmas *real-root-gt-0-iff* [*simp*] = *real-root-less-iff* [**where** $x=0$, *simplified*]

lemmas *real-root-lt-0-iff* [*simp*] = *real-root-less-iff* [**where** $y=0$, *simplified*]

lemmas *real-root-ge-0-iff* [*simp*] = *real-root-le-iff* [**where** $x=0$, *simplified*]

lemmas *real-root-le-0-iff* [*simp*] = *real-root-le-iff* [**where** $y=0$, *simplified*]

lemmas *real-root-eq-0-iff* [*simp*] = *real-root-eq-iff* [**where** $y=0$, *simplified*]

lemma *real-root-gt-1-iff* [*simp*]: $0 < n \implies 1 < \text{root } n \ y \longleftrightarrow 1 < y$
using *real-root-less-iff* [**where** $x=1$] **by** *simp*

lemma *real-root-lt-1-iff* [*simp*]: $0 < n \implies \text{root } n \ x < 1 \longleftrightarrow x < 1$
using *real-root-less-iff* [**where** $y=1$] **by** *simp*

lemma *real-root-ge-1-iff* [*simp*]: $0 < n \implies 1 \leq \text{root } n \ y \longleftrightarrow 1 \leq y$
using *real-root-le-iff* [**where** $x=1$] **by** *simp*

lemma *real-root-le-1-iff* [*simp*]: $0 < n \implies \text{root } n \ x \leq 1 \longleftrightarrow x \leq 1$
using *real-root-le-iff* [**where** $y=1$] **by** *simp*

lemma *real-root-eq-1-iff* [*simp*]: $0 < n \implies \text{root } n \ x = 1 \longleftrightarrow x = 1$
using *real-root-eq-iff* [**where** $y=1$] **by** *simp*

Roots of multiplication and division.

lemma *real-root-mult*: $\text{root } n \ (x * y) = \text{root } n \ x * \text{root } n \ y$
by (*auto split: split-root elim!: sgn-power-injE*
simp: sgn-mult abs-mult power-mult-distrib)

lemma *real-root-inverse*: $\text{root } n \ (\text{inverse } x) = \text{inverse } (\text{root } n \ x)$
by (*auto split: split-root elim!: sgn-power-injE*
simp: power-inverse)

lemma *real-root-divide*: $\text{root } n \ (x / y) = \text{root } n \ x / \text{root } n \ y$
by (*simp add: divide-inverse real-root-mult real-root-inverse*)

lemma *real-root-abs*: $0 < n \implies \text{root } n \ |x| = |\text{root } n \ x|$
by (*simp add: abs-if real-root-minus*)

lemma *root-abs-power*: $n > 0 \implies \text{abs } (\text{root } n \ (y \hat{\ } n)) = \text{abs } y$
using *root- sgn-power* [*of* n]
by (*metis abs-ge-zero power-abs real-root-abs real-root-power-cancel*)

lemma *real-root-power*: $0 < n \implies \text{root } n \ (x \hat{\ } k) = \text{root } n \ x \hat{\ } k$
by (*induct k*) (*simp-all add: real-root-mult*)

Roots of roots.

lemma *real-root-Suc-0* [*simp*]: $\text{root } (\text{Suc } 0) \ x = x$
by (*simp add: odd-real-root-unique*)

lemma *real-root-mult-exp*: $\text{root } (m * n) \ x = \text{root } m \ (\text{root } n \ x)$
by (*auto split: split-root elim!: sgn-power-injE*
simp: sgn-zero-iff sgn-mult power-mult[symmetric]
abs-mult power-mult-distrib abs- sgn-eq)

lemma *real-root-commute*: $\text{root } m \ (\text{root } n \ x) = \text{root } n \ (\text{root } m \ x)$
by (*simp add: real-root-mult-exp [symmetric] mult.commute*)

Monotonicity in first argument.

lemma *real-root-strict-decreasing*:

assumes $0 < n \ n < N \ 1 < x$
shows $\text{root } N \ x < \text{root } n \ x$
proof –
from *assms* **have** $\text{root } n \ (\text{root } N \ x) \wedge n < \text{root } N \ (\text{root } n \ x) \wedge N$
by (*simp add: real-root-commute power-strict-increasing del: real-root-pow-pos2*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *real-root-strict-increasing*:
assumes $0 < n \ n < N \ 0 < x \ x < 1$
shows $\text{root } n \ x < \text{root } N \ x$
proof –
from *assms* **have** $\text{root } N \ (\text{root } n \ x) \wedge N < \text{root } n \ (\text{root } N \ x) \wedge n$
by (*simp add: real-root-commute power-strict-decreasing del: real-root-pow-pos2*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *real-root-decreasing*: $0 < n \implies n \leq N \implies 1 \leq x \implies \text{root } N \ x \leq \text{root } n \ x$
by (*auto simp add: order-le-less real-root-strict-decreasing*)

lemma *real-root-increasing*: $0 < n \implies n \leq N \implies 0 \leq x \implies x \leq 1 \implies \text{root } n \ x \leq \text{root } N \ x$
by (*auto simp add: order-le-less real-root-strict-increasing*)

Continuity and derivatives.

lemma *isCont-real-root*: *isCont* (*root* *n*) *x*
proof (*cases* $n > 0$)
case *True*
let $?f = \lambda y::\text{real}. \text{sgn } y * |y| \wedge n$
have *continuous-on* ($\{0..\} \cup \{.. 0\}$) ($\lambda x. \text{if } 0 < x \text{ then } x \wedge n \text{ else } -((-x) \wedge n)$)
 $:: \text{real}$
using *True* **by** (*intro continuous-on-If continuous-intros*) *auto*
then **have** *continuous-on UNIV ?f*
by (*rule continuous-on-cong[THEN iffD1, rotated 2]*) (*auto simp: not-less le-less True*)
then **have** [*simp*]: *isCont* $?f \ x$ **for** *x*
by (*simp add: continuous-on-eq-continuous-at*)
have *isCont* (*root* *n*) ($?f \ (\text{root } n \ x)$)
by (*rule isCont-inverse-function [where f=?f and d=1]*) (*auto simp: root-sgn-power True*)
then **show** *?thesis*
by (*simp add: sgn-power-root True*)
next
case *False*
then **show** *?thesis*
by (*simp add: root-def[abs-def]*)
qed

lemma *tendsto-real-root* [*tendsto-intros*]:

$(f \longrightarrow x) F \implies ((\lambda x. \text{root } n (f x)) \longrightarrow \text{root } n x) F$
using *isCont-tendsto-compose*[*OF isCont-real-root, of f x F*].

lemma *continuous-real-root* [*continuous-intros*]:
continuous F f \implies *continuous F* $(\lambda x. \text{root } n (f x))$
unfolding *continuous-def* **by** (*rule tendsto-real-root*)

lemma *continuous-on-real-root* [*continuous-intros*]:
continuous-on s f \implies *continuous-on s* $(\lambda x. \text{root } n (f x))$
unfolding *continuous-on-def* **by** (*auto intro: tendsto-real-root*)

lemma *DERIV-real-root*:
assumes *n*: $0 < n$
and *x*: $0 < x$
shows *DERIV* $(\text{root } n) x \text{:> inverse (real } n * \text{root } n x \wedge (n - \text{Suc } 0))$
proof (*rule DERIV-inverse-function*)
show $0 < x$
using *x*.
show $x < x + 1$
by *simp*
show *DERIV* $(\lambda x. x \wedge n) (\text{root } n x) \text{:> real } n * \text{root } n x \wedge (n - \text{Suc } 0)$
by (*rule DERIV-pow*)
show $\text{real } n * \text{root } n x \wedge (n - \text{Suc } 0) \neq 0$
using *n x* **by** *simp*
show *isCont* $(\text{root } n) x$
by (*rule isCont-real-root*)
qed (*use n in auto*)

lemma *DERIV-odd-real-root*:
assumes *n*: *odd n*
and *x*: $x \neq 0$
shows *DERIV* $(\text{root } n) x \text{:> inverse (real } n * \text{root } n x \wedge (n - \text{Suc } 0))$
proof (*rule DERIV-inverse-function*)
show $x - 1 < x x < x + 1$
by *auto*
show *DERIV* $(\lambda x. x \wedge n) (\text{root } n x) \text{:> real } n * \text{root } n x \wedge (n - \text{Suc } 0)$
by (*rule DERIV-pow*)
show $\text{real } n * \text{root } n x \wedge (n - \text{Suc } 0) \neq 0$
using *odd-pos* [*OF n*] *x* **by** *simp*
show *isCont* $(\text{root } n) x$
by (*rule isCont-real-root*)
qed (*use n odd-real-root-pow in auto*)

lemma *DERIV-even-real-root*:
assumes *n*: $0 < n$
and *even n*
and *x*: $x < 0$
shows *DERIV* $(\text{root } n) x \text{:> inverse (- real } n * \text{root } n x \wedge (n - \text{Suc } 0))$
proof (*rule DERIV-inverse-function*)

```

show  $x - 1 < x$ 
  by simp
show  $x < 0$ 
  using  $x$  .
show  $-(\text{root } n \ y \ ^n) = y$  if  $x - 1 < y$  and  $y < 0$  for  $y$ 
proof -
  have  $\text{root } n \ (-y) \ ^n = -y$ 
    using that  $\langle 0 < n \rangle$  by simp
  with real-root-minus and  $\langle \text{even } n \rangle$ 
  show  $-(\text{root } n \ y \ ^n) = y$  by simp
qed
show DERIV  $(\lambda x. -(x \ ^n)) (\text{root } n \ x) :> - \text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)}$ 
  by (auto intro! : derivative-eq-intros)
show  $-\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)} \neq 0$ 
  using  $n \ x$  by simp
show isCont  $(\text{root } n) \ x$ 
  by (rule isCont-real-root)
qed

```

lemma *DERIV-real-root-generic*:

```

assumes  $0 < n$ 
  and  $x \neq 0$ 
  and  $\text{even } n \implies 0 < x \implies D = \text{inverse } (\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)})$ 
  and  $\text{even } n \implies x < 0 \implies D = - \text{inverse } (\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)})$ 
  and  $\text{odd } n \implies D = \text{inverse } (\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)})$ 
shows DERIV  $(\text{root } n) \ x :> D$ 
using assms
by (cases even n, cases 0 < x)
  (auto intro : DERIV-real-root[THEN DERIV-cong]
    DERIV-odd-real-root[THEN DERIV-cong]
    DERIV-even-real-root[THEN DERIV-cong])

```

lemma *power-tendsto-0-iff* [*simp*]:

```

fixes  $f :: 'a \Rightarrow \text{real}$ 
assumes  $n > 0$ 
shows  $((\lambda x. f \ x \ ^n) \longrightarrow 0) \ F \longleftrightarrow (f \longrightarrow 0) \ F$ 
proof -
  have  $((\lambda x. |\text{root } n \ (f \ x \ ^n)|) \longrightarrow 0) \ F \implies (f \longrightarrow 0) \ F$ 
    by (auto simp : assms root-abs-power tendsto-rabs-zero-iff)
  then have  $((\lambda x. f \ x \ ^n) \longrightarrow 0) \ F \implies (f \longrightarrow 0) \ F$ 
    by (metis tendsto-real-root abs-0 real-root-zero tendsto-rabs)
  with assms show ?thesis
    by (auto simp : tendsto-null-power)
qed

```

112.3 Square Root

definition *sqrt* :: $\text{real} \Rightarrow \text{real}$

where $\text{sqrt} = \text{root } 2$

lemma *pos2*: $0 < (2::nat)$
by *simp*

lemma *real-sqrt-unique*: $y^2 = x \implies 0 \leq y \implies \text{sqrt } x = y$
unfolding *sqrt-def* **by** (rule *real-root-pos-unique* [*OF pos2*])

lemma *real-sqrt-abs* [*simp*]: $\text{sqrt } (x^2) = |x|$
by (metis *power2-abs abs-ge-zero real-sqrt-unique*)

lemma *real-sqrt-pow2* [*simp*]: $0 \leq x \implies (\text{sqrt } x)^2 = x$
unfolding *sqrt-def* **by** (rule *real-root-pow-pos2* [*OF pos2*])

lemma *real-sqrt-pow2-iff* [*simp*]: $(\text{sqrt } x)^2 = x \iff 0 \leq x$
by (metis *real-sqrt-pow2 zero-le-power2*)

lemma *real-sqrt-zero* [*simp*]: $\text{sqrt } 0 = 0$
unfolding *sqrt-def* **by** (rule *real-root-zero*)

lemma *real-sqrt-one* [*simp*]: $\text{sqrt } 1 = 1$
unfolding *sqrt-def* **by** (rule *real-root-one* [*OF pos2*])

lemma *real-sqrt-four* [*simp*]: $\text{sqrt } 4 = 2$
using *real-sqrt-abs*[*of 2*] **by** *simp*

lemma *real-sqrt-minus*: $\text{sqrt } (-x) = -\text{sqrt } x$
unfolding *sqrt-def* **by** (rule *real-root-minus*)

lemma *real-sqrt-mult*: $\text{sqrt } (x * y) = \text{sqrt } x * \text{sqrt } y$
unfolding *sqrt-def* **by** (rule *real-root-mult*)

lemma *real-sqrt-mult-self*[*simp*]: $\text{sqrt } a * \text{sqrt } a = |a|$
using *real-sqrt-abs*[*of a*] **unfolding** *power2-eq-square* *real-sqrt-mult* .

lemma *real-sqrt-inverse*: $\text{sqrt } (\text{inverse } x) = \text{inverse } (\text{sqrt } x)$
unfolding *sqrt-def* **by** (rule *real-root-inverse*)

lemma *real-sqrt-divide*: $\text{sqrt } (x / y) = \text{sqrt } x / \text{sqrt } y$
unfolding *sqrt-def* **by** (rule *real-root-divide*)

lemma *real-sqrt-power*: $\text{sqrt } (x ^ k) = \text{sqrt } x ^ k$
unfolding *sqrt-def* **by** (rule *real-root-power* [*OF pos2*])

lemma *real-sqrt-gt-zero*: $0 < x \implies 0 < \text{sqrt } x$
unfolding *sqrt-def* **by** (rule *real-root-gt-zero* [*OF pos2*])

lemma *real-sqrt-ge-zero*: $0 \leq x \implies 0 \leq \text{sqrt } x$
unfolding *sqrt-def* **by** (rule *real-root-ge-zero*)

lemma *real-sqrt-less-mono*: $x < y \implies \text{sqrt } x < \text{sqrt } y$
unfolding *sqrt-def* **by** (rule *real-root-less-mono* [*OF pos2*])

lemma *real-sqrt-le-mono*: $x \leq y \implies \text{sqrt } x \leq \text{sqrt } y$
unfolding *sqrt-def* **by** (rule *real-root-le-mono* [*OF pos2*])

lemma *real-sqrt-less-iff* [*simp*]: $\text{sqrt } x < \text{sqrt } y \longleftrightarrow x < y$
unfolding *sqrt-def* **by** (rule *real-root-less-iff* [*OF pos2*])

lemma *real-sqrt-le-iff* [*simp*]: $\text{sqrt } x \leq \text{sqrt } y \longleftrightarrow x \leq y$
unfolding *sqrt-def* **by** (rule *real-root-le-iff* [*OF pos2*])

lemma *real-sqrt-eq-iff* [*simp*]: $\text{sqrt } x = \text{sqrt } y \longleftrightarrow x = y$
unfolding *sqrt-def* **by** (rule *real-root-eq-iff* [*OF pos2*])

lemma *real-less-lsqrt*: $0 \leq y \implies x < y^2 \implies \text{sqrt } x < y$
using *real-sqrt-less-iff*[*of x y²*] **by** *simp*

lemma *real-le-lsqrt*: $0 \leq y \implies x \leq y^2 \implies \text{sqrt } x \leq y$
using *real-sqrt-le-iff*[*of x y²*] **by** *simp*

lemma *real-le-rsqrt*: $x^2 \leq y \implies x \leq \text{sqrt } y$
using *real-sqrt-le-mono*[*of x² y*] **by** *simp*

lemma *real-less-rsqrt*: $x^2 < y \implies x < \text{sqrt } y$
using *real-sqrt-less-mono*[*of x² y*] **by** *simp*

lemma *real-sqrt-power-even*:
assumes *even n* $x \geq 0$
shows $\text{sqrt } x \wedge n = x \wedge (n \text{ div } 2)$
proof –
from *assms* **obtain** *k* **where** $n = 2 * k$ **by** (*auto elim!: evenE*)
with *assms* **show** *?thesis* **by** (*simp add: power-mult*)
qed

lemma *sqrt-le-D*: $\text{sqrt } x \leq y \implies x \leq y^2$
by (*meson not-le real-less-rsqrt*)

lemma *sqrt-ge-absD*: $|x| \leq \text{sqrt } y \implies x^2 \leq y$
using *real-sqrt-le-iff*[*of x²*] **by** *simp*

lemma *sqrt-even-pow2*:
assumes *n: even n*
shows $\text{sqrt } (2 \wedge n) = 2 \wedge (n \text{ div } 2)$
proof –
from *n* **obtain** *m* **where** $n = 2 * m$..
from *m* **have** $\text{sqrt } (2 \wedge n) = \text{sqrt } ((2 \wedge m)^2)$
by (*simp only: power-mult[symmetric] mult.commute*)
then show *?thesis*

using m by *simp*
qed

lemmas *real-sqrt-gt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** $x=0$, *unfolded real-sqrt-zero*]
 lemmas *real-sqrt-lt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** $y=0$, *unfolded real-sqrt-zero*]
 lemmas *real-sqrt-ge-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** $x=0$, *unfolded real-sqrt-zero*]
 lemmas *real-sqrt-le-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** $y=0$, *unfolded real-sqrt-zero*]
 lemmas *real-sqrt-eq-0-iff* [*simp*] = *real-sqrt-eq-iff* [**where** $y=0$, *unfolded real-sqrt-zero*]

lemmas *real-sqrt-gt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** $x=1$, *unfolded real-sqrt-one*]
 lemmas *real-sqrt-lt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** $y=1$, *unfolded real-sqrt-one*]
 lemmas *real-sqrt-ge-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** $x=1$, *unfolded real-sqrt-one*]
 lemmas *real-sqrt-le-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** $y=1$, *unfolded real-sqrt-one*]
 lemmas *real-sqrt-eq-1-iff* [*simp*] = *real-sqrt-eq-iff* [**where** $y=1$, *unfolded real-sqrt-one*]

lemma *sqrt-add-le-add-sqrt*:
 assumes $0 \leq x \leq y$
 shows $\text{sqrt}(x + y) \leq \text{sqrt } x + \text{sqrt } y$
 by (rule *power2-le-imp-le*) (*simp-all add: power2-sum assms*)

lemma *isCont-real-sqrt*: *isCont sqrt x*
 unfolding *sqrt-def* by (rule *isCont-real-root*)

lemma *tendsto-real-sqrt* [*tendsto-intros*]:
 $(f \longrightarrow x) F \implies ((\lambda x. \text{sqrt}(f x)) \longrightarrow \text{sqrt } x) F$
 unfolding *sqrt-def* by (rule *tendsto-real-root*)

lemma *continuous-real-sqrt* [*continuous-intros*]:
 $\text{continuous } F f \implies \text{continuous } F (\lambda x. \text{sqrt}(f x))$
 unfolding *sqrt-def* by (rule *continuous-real-root*)

lemma *continuous-on-real-sqrt* [*continuous-intros*]:
 $\text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. \text{sqrt}(f x))$
 unfolding *sqrt-def* by (rule *continuous-on-real-root*)

lemma *DERIV-real-sqrt-generic*:
 assumes $x \neq 0$
 and $x > 0 \implies D = \text{inverse}(\text{sqrt } x) / 2$
 and $x < 0 \implies D = - \text{inverse}(\text{sqrt } x) / 2$
 shows *DERIV sqrt x :=> D*
 using *assms* unfolding *sqrt-def*
 by (*auto intro!*: *DERIV-real-root-generic*)

lemma *DERIV-real-sqrt*: $0 < x \implies \text{DERIV sqrt } x :=> \text{inverse}(\text{sqrt } x) / 2$
 using *DERIV-real-sqrt-generic* by *simp*

declare
DERIV-real-sqrt-generic[*THEN DERIV-chain2, derivative-intros*]
DERIV-real-root-generic[*THEN DERIV-chain2, derivative-intros*]

lemmas *has-derivative-real-sqrt*[*derivative-intros*] = *DERIV-real-sqrt*[*THEN DERIV-compose-FDERIV*]

lemma *not-real-square-gt-zero* [*simp*]: $\neg 0 < x * x \longleftrightarrow x = 0$
for $x :: \text{real}$
apply *auto*
using *linorder-less-linear* [**where** $x = x$ **and** $y = 0$]
apply (*simp add: zero-less-mult-iff*)
done

lemma *real-sqrt-abs2* [*simp*]: $\text{sqrt}(x * x) = |x|$
apply (*subst power2-eq-square [symmetric]*)
apply (*rule real-sqrt-abs*)
done

lemma *real-inv-sqrt-pow2*: $0 < x \implies (\text{inverse}(\text{sqrt } x))^2 = \text{inverse } x$
by (*simp add: power-inverse*)

lemma *real-sqrt-eq-zero-cancel*: $0 \leq x \implies \text{sqrt } x = 0 \implies x = 0$
by *simp*

lemma *real-sqrt-ge-one*: $1 \leq x \implies 1 \leq \text{sqrt } x$
by *simp*

lemma *sqrt-divide-self-eq*:
assumes *nneg*: $0 \leq x$
shows $\text{sqrt } x / x = \text{inverse}(\text{sqrt } x)$
proof (*cases x = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have *pos*: $0 < x$
using *nneg* **by** *arith*
show *?thesis*
proof (*rule right-inverse-eq [THEN iffD1, symmetric]*)
show $\text{sqrt } x / x \neq 0$
by (*simp add: divide-inverse nneg False*)
show $\text{inverse}(\text{sqrt } x) / (\text{sqrt } x / x) = 1$
by (*simp add: divide-inverse mult.assoc [symmetric]*
power2-eq-square [symmetric] real-inv-sqrt-pow2 pos False)
qed
qed

lemma *real-div-sqrt*: $0 \leq x \implies x / \text{sqrt } x = \text{sqrt } x$
by (*cases x = 0*) (*simp-all add: sqrt-divide-self-eq [of x] field-simps*)

lemma *real-divide-square-eq* [*simp*]: $(r * a) / (r * r) = a / r$

for $a r :: \text{real}$
by (*cases* $r = 0$) (*simp-all add: divide-inverse ac-simps*)

lemma *lemma-real-divide-sqrt-less*: $0 < u \implies u / \text{sqrt } 2 < u$
by (*simp add: divide-less-eq*)

lemma *four-x-squared*: $4 * x^2 = (2 * x)^2$
for $x :: \text{real}$
by (*simp add: power2-eq-square*)

lemma *sqrt-at-top*: *LIM* x *at-top*. *sqrt* $x :: \text{real} :>$ *at-top*
by (*rule filterlim-at-top-at-top*[**where** $Q = \lambda x. \text{True}$ **and** $P = \lambda x. 0 < x$ **and** $g = \text{power2}$])
(auto intro: eventually-gt-at-top)

112.4 Square Root of Sum of Squares

lemma *sum-squares-bound*: $2 * x * y \leq x^2 + y^2$
for $x y :: 'a::\text{linordered-field}$
proof –
have $(x - y)^2 = x * x - 2 * x * y + y * y$
by *algebra*
then have $0 \leq x^2 - 2 * x * y + y^2$
by (*metis sum-power2-ge-zero zero-le-double-add-iff-zero-le-single-add power2-eq-square*)
then show *?thesis*
by *arith*
qed

lemma *arith-geo-mean*:
fixes $u :: 'a::\text{linordered-field}$
assumes $u^2 = x * y$ $x \geq 0$ $y \geq 0$
shows $u \leq (x + y) / 2$
apply (*rule power2-le-imp-le*)
using *sum-squares-bound assms*
apply (*auto simp: zero-le-mult-iff*)
apply (*auto simp: algebra-simps power2-eq-square*)
done

lemma *arith-geo-mean-sqrt*:
fixes $x :: \text{real}$
assumes $x \geq 0$ $y \geq 0$
shows $\text{sqrt } (x * y) \leq (x + y) / 2$
apply (*rule arith-geo-mean*)
using *assms*
apply (*auto simp: zero-le-mult-iff*)
done

lemma *real-sqrt-sum-squares-mult-ge-zero* [*simp*]: $0 \leq \text{sqrt } ((x^2 + y^2) * (x^2 + y^2))$

by (metis real-sqrt-ge-0-iff split-mult-pos-le sum-power2-ge-zero)

lemma real-sqrt-sum-squares-mult-squared-eq [simp]:
 $(\text{sqrt} ((x^2 + y^2) * (xa^2 + ya^2)))^2 = (x^2 + y^2) * (xa^2 + ya^2)$
 by (simp add: zero-le-mult-iff)

lemma real-sqrt-sum-squares-eq-cancel: $\text{sqrt} (x^2 + y^2) = x \implies y = 0$
 by (drule arg-cong [where f = $\lambda x. x^2$]) simp

lemma real-sqrt-sum-squares-eq-cancel2: $\text{sqrt} (x^2 + y^2) = y \implies x = 0$
 by (drule arg-cong [where f = $\lambda x. x^2$]) simp

lemma real-sqrt-sum-squares-ge1 [simp]: $x \leq \text{sqrt} (x^2 + y^2)$
 by (rule power2-le-imp-le) simp-all

lemma real-sqrt-sum-squares-ge2 [simp]: $y \leq \text{sqrt} (x^2 + y^2)$
 by (rule power2-le-imp-le) simp-all

lemma real-sqrt-ge-abs1 [simp]: $|x| \leq \text{sqrt} (x^2 + y^2)$
 by (rule power2-le-imp-le) simp-all

lemma real-sqrt-ge-abs2 [simp]: $|y| \leq \text{sqrt} (x^2 + y^2)$
 by (rule power2-le-imp-le) simp-all

lemma le-real-sqrt-sumsq [simp]: $x \leq \text{sqrt} (x * x + y * y)$
 by (simp add: power2-eq-square [symmetric])

lemma sqrt-sum-squares-le-sum:
 $\llbracket 0 \leq x; 0 \leq y \rrbracket \implies \text{sqrt} (x^2 + y^2) \leq x + y$
 by (rule power2-le-imp-le) (simp-all add: power2-sum)

lemma L2-set-mult-ineq-lemma:
 fixes a b c d :: real
 shows $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
 proof –
 have $0 \leq (a * d - b * c)^2$ by simp
 also have $\dots = a^2 * d^2 + b^2 * c^2 - 2 * (a * d) * (b * c)$
 by (simp only: power2-diff power-mult-distrib)
 also have $\dots = a^2 * d^2 + b^2 * c^2 - 2 * (a * c) * (b * d)$
 by simp
 finally show $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
 by simp
 qed

lemma sqrt-sum-squares-le-sum-abs: $\text{sqrt} (x^2 + y^2) \leq |x| + |y|$
 by (rule power2-le-imp-le) (simp-all add: power2-sum)

lemma real-sqrt-sum-squares-triangle-ineq:
 $\text{sqrt} ((a + c)^2 + (b + d)^2) \leq \text{sqrt} (a^2 + b^2) + \text{sqrt} (c^2 + d^2)$

proof –
have $(a * c + b * d) \leq (\text{sqrt } (a^2 + b^2) * \text{sqrt } (c^2 + d^2))$
by (*rule power2-le-imp-le*) (*simp-all add: power2-sum power-mult-distrib ring-distrib L2-set-mult-ineq-lemma add commute*)
then have $(a + c)^2 + (b + d)^2 \leq (\text{sqrt } (a^2 + b^2) + \text{sqrt } (c^2 + d^2))^2$
by (*simp add: power2-sum*)
then show *?thesis*
by (*auto intro: power2-le-imp-le*)
qed

lemma *real-sqrt-sum-squares-less*: $|x| < u / \text{sqrt } 2 \implies |y| < u / \text{sqrt } 2 \implies \text{sqrt } (x^2 + y^2) < u$
apply (*rule power2-less-imp-less*)
apply *simp*
apply (*drule power-strict-mono [OF - abs-ge-zero pos2]*)
apply (*drule power-strict-mono [OF - abs-ge-zero pos2]*)
apply (*simp add: power-divide*)
apply (*drule order-le-less-trans [OF abs-ge-zero]*)
apply (*simp add: zero-less-divide-iff*)
done

lemma *sqrt2-less-2*: $\text{sqrt } 2 < (2::\text{real})$
by (*metis not-less not-less-iff-gr-or-eq numeral-less-iff real-sqrt-four real-sqrt-le-iff semiring-norm(75) semiring-norm(78) semiring-norm(85)*)

lemma *sqrt-sum-squares-half-less*:
 $x < u/2 \implies y < u/2 \implies 0 \leq x \implies 0 \leq y \implies \text{sqrt } (x^2 + y^2) < u$
apply (*rule real-sqrt-sum-squares-less*)
apply (*auto simp add: abs-if field-simps*)
apply (*rule le-less-trans [where y = x*2]*)
using *less-eq-real-def sqrt2-less-2* **apply** *force*
apply *assumption*
apply (*rule le-less-trans [where y = y*2]*)
using *less-eq-real-def sqrt2-less-2 mult-le-cancel-left*
apply *auto*
done

lemma *LIMSEQ-root*: $(\lambda n. \text{root } n n) \longrightarrow 1$

proof –
define *x* **where** $x\ n = \text{root } n\ n - 1$ **for** *n*
have $x \longrightarrow \text{sqrt } 0$
proof (*rule tendsto-sandwich[OF - - tendsto-const]*)
show $(\lambda x. \text{sqrt } (2 / x)) \longrightarrow \text{sqrt } 0$
by (*intro tendsto-intros tendsto-divide-0[OF tendsto-const] filterlim-mono[OF filterlim-real-sequentially]*)
(simp-all add: at-infinity-eq-at-top-bot)
have $x\ n \leq \text{sqrt } (2 / \text{real } n)$ **if** $2 < n$ **for** $n :: \text{nat}$
proof –
have $1 + (\text{real } (n - 1) * n) / 2 * (x\ n)^2 = 1 + \text{of-nat } (n \text{ choose } 2) * (x\ n)^2$

by (auto simp add: choose-two field-char-0-class.of-nat-div mod-eq-0-iff-dvd)
 also have ... $\leq (\sum k \in \{0, 2\}. \text{of-nat } (n \text{ choose } k) * x \ n^k)$
 by (simp add: x-def)
 also have ... $\leq (\sum k \leq n. \text{of-nat } (n \text{ choose } k) * x \ n^k)$
 using $\langle 2 < n \rangle$
 by (intro sum-mono2) (auto intro!: mult-nonneg-nonneg zero-le-power simp:
 x-def le-diff-eq)
 also have ... $= (x \ n + 1) \ ^n$
 by (simp add: binomial-ring)
 also have ... $= n$
 using $\langle 2 < n \rangle$ by (simp add: x-def)
 finally have $\text{real } (n - 1) * (\text{real } n / 2 * (x \ n)^2) \leq \text{real } (n - 1) * 1$
 by simp
 then have $(x \ n)^2 \leq 2 / \text{real } n$
 using $\langle 2 < n \rangle$ unfolding mult-le-cancel-left by (simp add: field-simps)
 from real-sqrt-le-mono[OF this] show ?thesis
 by simp
 qed
 then show eventually $(\lambda n. x \ n \leq \text{sqrt } (2 / \text{real } n))$ sequentially
 by (auto intro!: exI[of - 3] simp: eventually-sequentially)
 show eventually $(\lambda n. \text{sqrt } 0 \leq x \ n)$ sequentially
 by (auto intro!: exI[of - 1] simp: eventually-sequentially le-diff-eq x-def)
 qed
 from tendsto-add[OF this tendsto-const[of 1]] show ?thesis
 by (simp add: x-def)
 qed

lemma LIMSEQ-root-const:

assumes $0 < c$

shows $(\lambda n. \text{root } n \ c) \longrightarrow 1$

proof -

have ge-1: $(\lambda n. \text{root } n \ c) \longrightarrow 1$ if $1 \leq c$ for $c :: \text{real}$

proof -

define x where $x \ n = \text{root } n \ c - 1$ for n

have $x \longrightarrow 0$

proof (rule tendsto-sandwich[OF - - tendsto-const])

show $(\lambda n. c / n) \longrightarrow 0$

by (intro tendsto-divide-0[OF tendsto-const] filterlim-mono[OF filterlim-real-sequentially])
 (simp-all add: at-infinity-eq-at-top-bot)

have $x \ n \leq c / n$ if $1 < n$ for $n :: \text{nat}$

proof -

have $1 + x \ n * n = 1 + \text{of-nat } (n \text{ choose } 1) * x \ n^1$

by (simp add: choose-one)

also have ... $\leq (\sum k \in \{0, 1\}. \text{of-nat } (n \text{ choose } k) * x \ n^k)$

by (simp add: x-def)

also have ... $\leq (\sum k \leq n. \text{of-nat } (n \text{ choose } k) * x \ n^k)$

using $\langle 1 < n \rangle \ \langle 1 \leq c \rangle$

by (intro sum-mono2)

(auto intro!: mult-nonneg-nonneg zero-le-power simp: x-def le-diff-eq)

```

    also have ... = (x n + 1) ^ n
      by (simp add: binomial-ring)
    also have ... = c
      using <1 < n> <1 ≤ c> by (simp add: x-def)
    finally show ?thesis
      using <1 ≤ c> <1 < n> by (simp add: field-simps)
  qed
  then show eventually (λn. x n ≤ c / n) sequentially
    by (auto intro!: exI[of - 3] simp: eventually-sequentially)
  show eventually (λn. 0 ≤ x n) sequentially
    using <1 ≤ c>
    by (auto intro!: exI[of - 1] simp: eventually-sequentially le-diff-eq x-def)
  qed
  from tendsto-add[OF this tendsto-const[of 1]] show ?thesis
    by (simp add: x-def)
  qed
  show ?thesis
  proof (cases 1 ≤ c)
    case True
      with ge-1 show ?thesis by blast
    next
      case False
        with <0 < c> have 1 ≤ 1 / c
          by simp
        then have (λn. 1 / root n (1 / c)) → 1 / 1
          by (intro tendsto-divide tendsto-const ge-1 <1 ≤ 1 / c> one-neq-zero)
        then show ?thesis
          by (rule filterlim-cong[THEN iffD1, rotated 3])
            (auto intro!: exI[of - 1] simp: eventually-sequentially real-root-divide)
  qed
  qed

```

Legacy theorem names:

```

lemmas real-root-pos2 = real-root-power-cancel
lemmas real-root-pos-pos = real-root-gt-zero [THEN order-less-imp-le]
lemmas real-root-pos-pos-le = real-root-ge-zero
lemmas real-sqrt-eq-zero-cancel-iff = real-sqrt-eq-0-iff

```

end

113 Power Series, Transcendental Functions etc.

```

theory Transcendental
imports Series Deriv NthRoot
begin

```

A theorem about the factorial function on the reals.

```

lemma square-fact-le-2-fact: fact n * fact n ≤ (fact (2 * n)) :: real

```

```

proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have (fact (Suc n)) * (fact (Suc n)) = of-nat (Suc n) * of-nat (Suc n) * (fact n
* fact n :: real)
  by (simp add: field-simps)
  also have ... ≤ of-nat (Suc n) * of-nat (Suc n) * fact (2 * n)
  by (rule mult-left-mono [OF Suc]) simp
  also have ... ≤ of-nat (Suc (Suc (2 * n))) * of-nat (Suc (2 * n)) * fact (2 *
n)
  by (rule mult-right-mono)+ (auto simp: field-simps)
  also have ... = fact (2 * Suc n) by (simp add: field-simps)
  finally show ?case .
qed

```

```

lemma fact-in-Reals: fact n ∈ ℝ
by (induction n) auto

```

```

lemma of-real-fact [simp]: of-real (fact n) = fact n
by (metis of-nat-fact of-real-of-nat-eq)

```

```

lemma pochhammer-of-real: pochhammer (of-real x) n = of-real (pochhammer x
n)
by (simp add: pochhammer-prod)

```

```

lemma norm-fact [simp]: norm (fact n :: 'a::real-normed-algebra-1) = fact n
proof –
  have (fact n :: 'a) = of-real (fact n)
  by simp
  also have norm ... = fact n
  by (subst norm-of-real) simp
  finally show ?thesis .
qed

```

```

lemma root-test-convergence:
  fixes f :: nat ⇒ 'a::banach
  assumes f: (λn. root n (norm (f n))) —————> x — could be weakened to lim sup
  and x < 1
  shows summable f
proof –
  have 0 ≤ x
  by (rule LIMSEQ-le[OF tendsto-const f]) (auto intro!: exI[of - 1])
  from ⟨x < 1⟩ obtain z where z: x < z < 1
  by (metis dense)
  from f ⟨x < z⟩ have eventually (λn. root n (norm (f n)) < z) sequentially
  by (rule order-tendstoD)
  then have eventually (λn. norm (f n) ≤ z^n) sequentially

```


using *eventually-ge-at-top*
proof *eventually-elim*
fix n
assume *less: root n (norm $(f\ n)) < z$ and $n: 1 \leq n$*
from *power-strict-mono[OF less, of n] n show $\text{norm } (f\ n) \leq z \wedge n$*
by *simp*
qed
then show *summable f*
unfolding *eventually-sequentially*
using $z < 0 \leq x$ **by** *(auto intro!: summable-comparison-test[OF - summable-geometric])*
qed

113.1 More facts about binomial coefficients

These facts could have been proven before, but having real numbers makes the proofs a lot easier.

lemma *central-binomial-odd:*

odd $n \implies n \text{ choose } (\text{Suc } (n \text{ div } 2)) = n \text{ choose } (n \text{ div } 2)$

proof –

assume *odd n*

hence *$\text{Suc } (n \text{ div } 2) \leq n$ by presburger*

hence *$n \text{ choose } (\text{Suc } (n \text{ div } 2)) = n \text{ choose } (n - \text{Suc } (n \text{ div } 2))$*

by *(rule binomial-symmetric)*

also from *$\langle \text{odd } n \rangle$ have $n - \text{Suc } (n \text{ div } 2) = n \text{ div } 2$ by presburger*

finally show *?thesis .*

qed

lemma *binomial-less-binomial-Suc:*

assumes *$k: k < n \text{ div } 2$*

shows *$n \text{ choose } k < n \text{ choose } (\text{Suc } k)$*

proof –

from *k have $k': k \leq n \text{ Suc } k \leq n$ by simp-all*

from *k' have $\text{real } (n \text{ choose } k) = \text{fact } n / (\text{fact } k * \text{fact } (n - k))$*

by *(simp add: binomial-fact)*

also from *k' have $n - k = \text{Suc } (n - \text{Suc } k)$ by simp*

also from *k' have $\text{fact } \dots = (\text{real } n - \text{real } k) * \text{fact } (n - \text{Suc } k)$*

by *(subst fact-Suc) (simp-all add: of-nat-diff)*

also from *k have $\text{fact } k = \text{fact } (\text{Suc } k) / (\text{real } k + 1)$ by (simp add: field-simps)*

also have *$\text{fact } n / (\text{fact } (\text{Suc } k) / (\text{real } k + 1) * ((\text{real } n - \text{real } k) * \text{fact } (n - \text{Suc } k))) =$*

*$(n \text{ choose } (\text{Suc } k)) * ((\text{real } k + 1) / (\text{real } n - \text{real } k))$*

using *k by (simp add: field-split-simps binomial-fact)*

also from *assms have $(\text{real } k + 1) / (\text{real } n - \text{real } k) < 1$ by simp*

finally show *?thesis using k by (simp add: mult-less-cancel-left)*

qed

lemma *binomial-strict-mono:*

assumes *$k < k' \ 2*k' \leq n$*

shows *$n \text{ choose } k < n \text{ choose } k'$*

```

proof –
  from assms have  $k \leq k' - 1$  by simp
  thus ?thesis
  proof (induction rule: inc-induct)
    case base
    with assms binomial-less-binomial-Suc[of  $k' - 1$   $n$ ]
    show ?case by simp
  next
  case (step  $k$ )
  from step.prems step.hyps assms have  $n$  choose  $k < n$  choose (Suc  $k$ )
    by (intro binomial-less-binomial-Suc) simp-all
  also have  $\dots < n$  choose  $k'$  by (rule step.IH)
  finally show ?case .
qed
qed

```

```

lemma binomial-mono:
  assumes  $k \leq k'$   $2 * k' \leq n$ 
  shows  $n$  choose  $k \leq n$  choose  $k'$ 
  using assms binomial-strict-mono[of  $k$   $k'$   $n$ ] by (cases  $k = k'$ ) simp-all

```

```

lemma binomial-strict-antimono:
  assumes  $k < k'$   $2 * k \geq n$   $k' \leq n$ 
  shows  $n$  choose  $k > n$  choose  $k'$ 
proof –
  from assms have  $n$  choose  $(n - k) > n$  choose  $(n - k')$ 
    by (intro binomial-strict-mono) (simp-all add: algebra-simps)
  with assms show ?thesis by (simp add: binomial-symmetric [symmetric])
qed

```

```

lemma binomial-antimono:
  assumes  $k \leq k'$   $k \geq n \text{ div } 2$   $k' \leq n$ 
  shows  $n$  choose  $k \geq n$  choose  $k'$ 
proof (cases  $k = k'$ )
  case False
  note not-eq = False
  show ?thesis
  proof (cases  $k = n \text{ div } 2 \wedge \text{odd } n$ )
    case False
    with assms( $2$ ) have  $2 * k \geq n$  by presburger
    with not-eq assms binomial-strict-antimono[of  $k$   $k'$   $n$ ]
    show ?thesis by simp
  next
  case True
  have  $n$  choose  $k' \leq n$  choose (Suc ( $n \text{ div } 2$ ))
  proof (cases  $k' = \text{Suc } (n \text{ div } 2)$ )
    case False
    with assms True not-eq have  $\text{Suc } (n \text{ div } 2) < k'$  by simp
    with assms binomial-strict-antimono[of  $\text{Suc } (n \text{ div } 2)$   $k'$   $n$ ] True

```

```

    show ?thesis by auto
  qed simp-all
  also from True have ... = n choose k by (simp add: central-binomial-odd)
  finally show ?thesis .
qed
qed simp-all

```

lemma *binomial-maximum*: n choose $k \leq n$ choose $(n \text{ div } 2)$

```

proof -
  have  $k \leq n \text{ div } 2 \iff 2*k \leq n$  by linarith
  consider  $2*k \leq n \mid 2*k \geq n \mid k \leq n \mid k > n$  by linarith
  thus ?thesis
proof cases
  case 1
  thus ?thesis by (intro binomial-mono) linarith+
next
  case 2
  thus ?thesis by (intro binomial-antimono) simp-all
qed (simp-all add: binomial-eq-0)
qed

```

lemma *binomial-maximum'*: $(2*n)$ choose $k \leq (2*n)$ choose n
 using *binomial-maximum*[of $2*n$] by simp

lemma *central-binomial-lower-bound*:

```

  assumes  $n > 0$ 
  shows  $4^n / (2*\text{real } n) \leq \text{real } ((2*n) \text{ choose } n)$ 
proof -
  from binomial[of 1 1  $2*n$ ]
  have  $4^n = (\sum_{k \leq 2*n}. (2*n) \text{ choose } k)$ 
  by (simp add: power-mult power2-eq-square One-nat-def [symmetric] del: One-nat-def)
  also have  $\{..2*n\} = \{0<.. $2*n\} \cup \{0,2*n\}$  by auto
  also have  $(\sum_{k \in \dots} (2*n) \text{ choose } k) =$ 
     $(\sum_{k \in \{0<.. $2*n\}}. (2*n) \text{ choose } k) + (\sum_{k \in \{0,2*n\}}. (2*n) \text{ choose } k)$ 
  by (subst sum.union-disjoint) auto
  also have  $(\sum_{k \in \{0,2*n\}}. (2*n) \text{ choose } k) \leq (\sum_{k \leq 1}. (n \text{ choose } k)^2)$ 
  by (cases n) simp-all
  also from assms have ...  $\leq (\sum_{k \leq n}. (n \text{ choose } k)^2)$ 
  by (intro sum-mono2) auto
  also have ... =  $(2*n) \text{ choose } n$  by (rule choose-square-sum)
  also have  $(\sum_{k \in \{0<.. $2*n\}}. (2*n) \text{ choose } k) \leq (\sum_{k \in \{0<.. $2*n\}}. (2*n) \text{ choose } n)$ 
  by (intro sum-mono binomial-maximum')
  also have ... =  $\text{card } \{0<.. $2*n\} * ((2*n) \text{ choose } n)$  by simp
  also have  $\text{card } \{0<.. $2*n\} \leq 2*n - 1$  by (cases n) simp-all
  also have  $(2 * n - 1) * (2 * n \text{ choose } n) + (2 * n \text{ choose } n) = ((2*n) \text{ choose } n) * (2*n)$ 
  using assms by (simp add: algebra-simps)
  finally have  $4^n \leq (2 * n \text{ choose } n) * (2 * n)$  by simp-all$$$$$$ 
```

hence $\text{real } (4 \wedge n) \leq \text{real } ((2 * n \text{ choose } n) * (2 * n))$
by *(subst of-nat-le-iff)*
with *assms* **show** *?thesis* **by** *(simp add: field-simps)*
qed

113.2 Properties of Power Series

lemma *powser-zero [simp]*: $(\sum n. f n * 0 \wedge n) = f 0$
for $f :: \text{nat} \Rightarrow 'a::\text{real-normed-algebra-1}$
proof –
have $(\sum n < 1. f n * 0 \wedge n) = (\sum n. f n * 0 \wedge n)$
by *(subst suminf-finite[where N={0}]) (auto simp: power-0-left)*
then **show** *?thesis* **by** *simp*
qed

lemma *powser-sums-zero*: $(\lambda n. a n * 0 \wedge n)$ *sums* $a 0$
for $a :: \text{nat} \Rightarrow 'a::\text{real-normed-div-algebra}$
using *sums-finite [of {0}]* $\lambda n. a n * 0 \wedge n$
by *simp*

lemma *powser-sums-zero-iff [simp]*: $(\lambda n. a n * 0 \wedge n)$ *sums* $x \iff a 0 = x$
for $a :: \text{nat} \Rightarrow 'a::\text{real-normed-div-algebra}$
using *powser-sums-zero sums-unique2* **by** *blast*

Power series has a circle or radius of convergence: if it sums for x , then it sums absolutely for z with $|z| < |x|$.

lemma *powser-insidea*:
fixes $x z :: 'a::\text{real-normed-div-algebra}$
assumes *1: summable* $(\lambda n. f n * x \wedge n)$
and *2: norm* $z < \text{norm } x$
shows *summable* $(\lambda n. \text{norm } (f n * z \wedge n))$
proof –
from *2* **have** $x \neq 0$ **by** *clarsimp*
from *1* **have** $(\lambda n. f n * x \wedge n) \longrightarrow 0$
by *(rule summable-LIMSEQ-zero)*
then **have** *convergent* $(\lambda n. f n * x \wedge n)$
by *(rule convergentI)*
then **have** *Cauchy* $(\lambda n. f n * x \wedge n)$
by *(rule convergent-Cauchy)*
then **have** *Bseq* $(\lambda n. f n * x \wedge n)$
by *(rule Cauchy-Bseq)*
then **obtain** K **where** *3: 0 < K* **and** *4: $\forall n. \text{norm } (f n * x \wedge n) \leq K$*
by *(auto simp: Bseq-def)*
have $\exists N. \forall n \geq N. \text{norm } (\text{norm } (f n * z \wedge n)) \leq K * \text{norm } (z \wedge n) * \text{inverse } (\text{norm } (x \wedge n))$
proof *(intro exI allI impI)*
fix $n :: \text{nat}$
assume $0 \leq n$
have $\text{norm } (\text{norm } (f n * z \wedge n)) * \text{norm } (x \wedge n) =$

$\text{norm } (f\ n * x^{\wedge}n) * \text{norm } (z^{\wedge}n)$
 by (simp add: norm-mult abs-mult)
 also have $\dots \leq K * \text{norm } (z^{\wedge}n)$
 by (simp only: mult-right-mono \neq norm-ge-zero)
 also have $\dots = K * \text{norm } (z^{\wedge}n) * (\text{inverse } (\text{norm } (x^{\wedge}n)) * \text{norm } (x^{\wedge}n))$
 by (simp add: x-neq-0)
 also have $\dots = K * \text{norm } (z^{\wedge}n) * \text{inverse } (\text{norm } (x^{\wedge}n)) * \text{norm } (x^{\wedge}n)$
 by (simp only: mult.assoc)
 finally show $\text{norm } (\text{norm } (f\ n * z^{\wedge}n)) \leq K * \text{norm } (z^{\wedge}n) * \text{inverse } (\text{norm } (x^{\wedge}n))$
 by (simp add: mult-le-cancel-right x-neq-0)
 qed
 moreover have summable $(\lambda n. K * \text{norm } (z^{\wedge}n) * \text{inverse } (\text{norm } (x^{\wedge}n)))$
 proof –
 from 2 have $\text{norm } (z * \text{inverse } x) < 1$
 using x-neq-0
 by (simp add: norm-mult nonzero-norm-inverse divide-inverse [where 'a=real, symmetric])
 then have summable $(\lambda n. \text{norm } (z * \text{inverse } x)^{\wedge}n)$
 by (rule summable-geometric)
 then have summable $(\lambda n. K * \text{norm } (z * \text{inverse } x)^{\wedge}n)$
 by (rule summable-mult)
 then show summable $(\lambda n. K * \text{norm } (z^{\wedge}n) * \text{inverse } (\text{norm } (x^{\wedge}n)))$
 using x-neq-0
 by (simp add: norm-mult nonzero-norm-inverse power-mult-distrib power-inverse norm-power mult.assoc)
 qed
 ultimately show summable $(\lambda n. \text{norm } (f\ n * z^{\wedge}n))$
 by (rule summable-comparison-test)
 qed

lemma powser-inside:

fixes $f :: \text{nat} \Rightarrow 'a::\{\text{real-normed-div-algebra}, \text{banach}\}$

shows

$\text{summable } (\lambda n. f\ n * (x^{\wedge}n)) \implies \text{norm } z < \text{norm } x \implies$
 $\text{summable } (\lambda n. f\ n * (z^{\wedge}n))$

by (rule powser-insidea [THEN summable-norm-cancel])

lemma powser-times-n-limit-0:

fixes $x :: 'a::\{\text{real-normed-div-algebra}, \text{banach}\}$

assumes $\text{norm } x < 1$

shows $(\lambda n. \text{of-nat } n * x^{\wedge}n) \longrightarrow 0$

proof –

have $\text{norm } x / (1 - \text{norm } x) \geq 0$

using assms by (auto simp: field-split-simps)

moreover obtain N where $N: \text{norm } x / (1 - \text{norm } x) < \text{of-int } N$

using ex-le-of-int by (meson ex-less-of-int)

ultimately have $N0: N > 0$

by auto

```

then have *: real-of-int (N + 1) * norm x / real-of-int N < 1
  using N assms by (auto simp: field-simps)
have **: real-of-int N * (norm x * (real-of-nat (Suc n) * norm (x ^ n))) ≤
  real-of-nat n * (norm x * ((1 + N) * norm (x ^ n))) if N ≤ int n for n :: nat
proof –
  from that have real-of-int N * real-of-nat (Suc n) ≤ real-of-nat n * real-of-int
  (1 + N)
  by (simp add: algebra-simps)
  then have (real-of-int N * real-of-nat (Suc n)) * (norm x * norm (x ^ n)) ≤
  (real-of-nat n * (1 + N)) * (norm x * norm (x ^ n))
  using N0 mult-mono by fastforce
  then show ?thesis
  by (simp add: algebra-simps)
qed
show ?thesis using *
  by (rule summable-LIMSEQ-zero [OF summable-ratio-test, where N1=nat N])
  (simp add: N0 norm-mult field-simps ** del: of-nat-Suc of-int-add)
qed

```

corollary *lim-n-over-pown*:

```

fixes x :: 'a::{real-normed-field,banach}
shows 1 < norm x ⇒ ((λn. of-nat n / x ^ n) → 0) sequentially
using power-times-n-limit-0 [of inverse x]
by (simp add: norm-divide field-split-simps)

```

lemma *sum-split-even-odd*:

```

fixes f :: nat ⇒ real
shows (∑ i<2 * n. if even i then f i else g i) = (∑ i<n. f (2 * i)) + (∑ i<n.
g (2 * i + 1))
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have (∑ i<2 * Suc n. if even i then f i else g i) =
  (∑ i<n. f (2 * i)) + (∑ i<n. g (2 * i + 1)) + (f (2 * n) + g (2 * n + 1))
  using Suc.hyps unfolding One-nat-def by auto
  also have ... = (∑ i<Suc n. f (2 * i)) + (∑ i<Suc n. g (2 * i + 1))
  by auto
  finally show ?case .
qed

```

lemma *sums-if'*:

```

fixes g :: nat ⇒ real
assumes g sums x
shows (λ n. if even n then 0 else g ((n - 1) div 2)) sums x
unfolding sums-def
proof (rule LIMSEQ-I)
  fix r :: real

```

```

assume  $0 < r$ 
from  $\langle g \text{ sums } x \rangle$  [unfolded sums-def, THEN LIMSEQ-D, OF this]
obtain no where no-eq:  $\bigwedge n. n \geq \text{no} \implies (\text{norm } (\text{sum } g \{..<n\} - x) < r)$ 
by blast

let  $?SUM = \lambda m. \sum_{i < m. \text{if even } i \text{ then } 0 \text{ else } g ((i - 1) \text{ div } 2)}$ 
have  $(\text{norm } (?SUM \text{ } m - x) < r)$  if  $m \geq 2 * \text{no}$  for  $m$ 
proof –
  from that have  $m \text{ div } 2 \geq \text{no}$  by auto
  have sum-eq:  $?SUM (2 * (m \text{ div } 2)) = \text{sum } g \{..< m \text{ div } 2\}$ 
    using sum-split-even-odd by auto
  then have  $(\text{norm } (?SUM (2 * (m \text{ div } 2)) - x) < r)$ 
    using no-eq unfolding sum-eq using  $\langle m \text{ div } 2 \geq \text{no} \rangle$  by auto
  moreover
  have  $?SUM (2 * (m \text{ div } 2)) = ?SUM m$ 
  proof (cases even m)
    case True
      then show ?thesis
        by (auto simp: even-two-times-div-two)
    next
      case False
        then have eq:  $\text{Suc } (2 * (m \text{ div } 2)) = m$  by simp
        then have even  $(2 * (m \text{ div } 2))$  using  $\langle \text{odd } m \rangle$  by auto
        have  $?SUM m = ?SUM (\text{Suc } (2 * (m \text{ div } 2)))$  unfolding eq ..
        also have  $\dots = ?SUM (2 * (m \text{ div } 2))$  using  $\langle \text{even } (2 * (m \text{ div } 2)) \rangle$  by
auto
        finally show ?thesis by auto
      qed
    ultimately show ?thesis by auto
  qed
then show  $\exists \text{no}. \forall m \geq \text{no}. \text{norm } (?SUM m - x) < r$ 
by blast
qed

lemma sums-if:
  fixes  $g :: \text{nat} \Rightarrow \text{real}$ 
  assumes  $g \text{ sums } x$  and  $f \text{ sums } y$ 
  shows  $(\lambda n. \text{if even } n \text{ then } f (n \text{ div } 2) \text{ else } g ((n - 1) \text{ div } 2)) \text{ sums } (x + y)$ 
proof –
  let  $?s = \lambda n. \text{if even } n \text{ then } 0 \text{ else } f ((n - 1) \text{ div } 2)$ 
  have if-sum:  $(\text{if } B \text{ then } (0 :: \text{real}) \text{ else } E) + (\text{if } B \text{ then } T \text{ else } 0) = (\text{if } B \text{ then } T \text{ else } E)$ 
  for  $B \ T \ E$ 
  by (cases B) auto
  have g-sums:  $(\lambda n. \text{if even } n \text{ then } 0 \text{ else } g ((n - 1) \text{ div } 2)) \text{ sums } x$ 
    using sums-if' [OF  $\langle g \text{ sums } x \rangle$ ].
  have if-eq:  $\bigwedge B \ T \ E. (\text{if } \neg B \text{ then } T \text{ else } E) = (\text{if } B \text{ then } E \text{ else } T)$ 
    by auto
  have  $?s \text{ sums } y$  using sums-if' [OF  $\langle f \text{ sums } y \rangle$ ].

```

```

from this[unfolding sums-def, THEN LIMSEQ-Suc]
have ( $\lambda n.$  if even  $n$  then  $f (n \text{ div } 2)$  else  $0$ ) sums  $y$ 
by (simp add: lessThan-Suc-eq-insert-0 sum.atLeast1-atMost-eq image-Suc-lessThan
      if-eq sums-def cong del: if-weak-cong)
from sums-add[OF  $g$ -sums this] show ?thesis
by (simp only: if-sum)
qed

```

113.3 Alternating series test / Leibniz formula

lemma sums-alternating-upper-lower:

```

fixes  $a :: \text{nat} \Rightarrow \text{real}$ 
assumes mono:  $\bigwedge n. a (Suc\ n) \leq a\ n$ 
and a-pos:  $\bigwedge n. 0 \leq a\ n$ 
and  $a \longrightarrow 0$ 
shows  $\exists l. ((\forall n. (\sum_{i < 2*n} (-1)^{i*a\ i} \leq l) \wedge (\lambda n. \sum_{i < 2*n} (-1)^{i*a\ i} \longrightarrow l) \wedge$ 
 $(\forall n. l \leq (\sum_{i < 2*n + 1} (-1)^{i*a\ i})) \wedge (\lambda n. \sum_{i < 2*n + 1} (-1)^{i*a\ i} \longrightarrow l)$ 
 $(\text{is } \exists l. ((\forall n. ?f\ n \leq l) \wedge -) \wedge ((\forall n. l \leq ?g\ n) \wedge -)))$ 
proof (rule nested-sequence-unique)
have fg-diff:  $\bigwedge n. ?f\ n - ?g\ n = - a (2 * n)$  by auto

```

show $\forall n. ?f\ n \leq ?f (Suc\ n)$

proof

```

show  $?f\ n \leq ?f (Suc\ n)$  for  $n$ 
using mono[of  $2*n$ ] by auto

```

qed

show $\forall n. ?g (Suc\ n) \leq ?g\ n$

proof

```

show  $?g (Suc\ n) \leq ?g\ n$  for  $n$ 
using mono[of  $Suc (2*n)$ ] by auto

```

qed

show $\forall n. ?f\ n \leq ?g\ n$

proof

```

show  $?f\ n \leq ?g\ n$  for  $n$ 
using fg-diff a-pos by auto

```

qed

show $(\lambda n. ?f\ n - ?g\ n) \longrightarrow 0$

unfolding fg-diff

proof (rule LIMSEQ-I)

fix $r :: \text{real}$

assume $0 < r$

with $\langle a \longrightarrow 0 \rangle$ [THEN LIMSEQ-D] **obtain** N **where** $\bigwedge n. n \geq N \implies$
 $\text{norm } (a\ n - 0) < r$

by auto

then have $\forall n \geq N. \text{norm } (- a (2 * n) - 0) < r$

by auto

then show $\exists N. \forall n \geq N. \text{norm } (- a (2 * n) - 0) < r$

by *auto*
 qed
 qed

lemma *summable-Leibniz'*:

fixes $a :: \text{nat} \Rightarrow \text{real}$

assumes *a-zero*: $a \longrightarrow 0$

and *a-pos*: $\bigwedge n. 0 \leq a\ n$

and *a-monotone*: $\bigwedge n. a\ (\text{Suc}\ n) \leq a\ n$

shows *summable*: *summable* $(\lambda n. (-1)^{\widehat{n}} * a\ n)$

and $\bigwedge n. (\sum i < 2*n. (-1)^{\widehat{i*a}\ i}) \leq (\sum i. (-1)^{\widehat{i*a}\ i})$

and $(\lambda n. \sum i < 2*n. (-1)^{\widehat{i*a}\ i}) \longrightarrow (\sum i. (-1)^{\widehat{i*a}\ i})$

and $\bigwedge n. (\sum i. (-1)^{\widehat{i*a}\ i}) \leq (\sum i < 2*n+1. (-1)^{\widehat{i*a}\ i})$

and $(\lambda n. \sum i < 2*n+1. (-1)^{\widehat{i*a}\ i}) \longrightarrow (\sum i. (-1)^{\widehat{i*a}\ i})$

proof –

let $?S = \lambda n. (-1)^{\widehat{n}} * a\ n$

let $?P = \lambda n. \sum i < n. ?S\ i$

let $?f = \lambda n. ?P\ (2 * n)$

let $?g = \lambda n. ?P\ (2 * n + 1)$

obtain $l :: \text{real}$

where *below-l*: $\forall n. ?f\ n \leq l$

and $?f \longrightarrow l$

and *above-l*: $\forall n. l \leq ?g\ n$

and $?g \longrightarrow l$

using *sums-alternating-upper-lower*[*OF a-monotone a-pos a-zero*] **by** *blast*

let $?Sa = \lambda m. \sum n < m. ?S\ n$

have $?Sa \longrightarrow l$

proof (*rule LIMSEQ-I*)

fix $r :: \text{real}$

assume $0 < r$

with $\langle ?f \longrightarrow l \rangle$ [*THEN LIMSEQ-D*]

obtain *f-no* **where** $f: \bigwedge n. n \geq f\text{-no} \implies \text{norm}\ (?f\ n - l) < r$

by *auto*

from $\langle 0 < r \rangle$ $\langle ?g \longrightarrow l \rangle$ [*THEN LIMSEQ-D*]

obtain *g-no* **where** $g: \bigwedge n. n \geq g\text{-no} \implies \text{norm}\ (?g\ n - l) < r$

by *auto*

have $\text{norm}\ (?Sa\ n - l) < r$ **if** $n \geq (\max\ (2 * f\text{-no})\ (2 * g\text{-no}))$ **for** n

proof –

from *that* **have** $n \geq 2 * f\text{-no}$ **and** $n \geq 2 * g\text{-no}$ **by** *auto*

show *?thesis*

proof (*cases even n*)

case *True*

then **have** *n-eq*: $2 * (n\ \text{div}\ 2) = n$

by (*simp add: even-two-times-div-two*)

with $\langle n \geq 2 * f\text{-no} \rangle$ **have** $n\ \text{div}\ 2 \geq f\text{-no}$

by *auto*

from f [*OF this*] **show** *?thesis*

unfolding *n-eq atLeastLessThanSuc-atLeastAtMost* .

```

next
  case False
  then have even  $(n - 1)$  by simp
  then have n-eq:  $2 * ((n - 1) \text{ div } 2) = n - 1$ 
    by (simp add: even-two-times-div-two)
  then have range-eq:  $n - 1 + 1 = n$ 
    using odd-pos[OF False] by auto
  from n-eq  $\langle n \geq 2 * g\text{-no} \rangle$  have  $(n - 1) \text{ div } 2 \geq g\text{-no}$ 
    by auto
  from g[OF this] show ?thesis
    by (simp only: n-eq range-eq)
qed
qed
then show  $\exists no. \forall n \geq no. \text{norm } (?S a \ n - l) < r$  by blast
qed
then have sums-l:  $(\lambda i. (-1)^i * a \ i) \text{ sums } l$ 
  by (simp only: sums-def)
then show summable ?S
  by (auto simp: summable-def)

have  $l = \text{suminf } ?S$  by (rule sums-unique[OF sums-l])

fix n
show suminf ?S  $\leq ?g \ n$ 
  unfolding sums-unique[OF sums-l, symmetric] using above-l by auto
show ?f  $n \leq \text{suminf } ?S$ 
  unfolding sums-unique[OF sums-l, symmetric] using below-l by auto
show ?g  $\longrightarrow \text{suminf } ?S$ 
  using  $\langle ?g \longrightarrow l \ \langle l = \text{suminf } ?S \rangle$  by auto
show ?f  $\longrightarrow \text{suminf } ?S$ 
  using  $\langle ?f \longrightarrow l \ \langle l = \text{suminf } ?S \rangle$  by auto
qed

theorem summable-Leibniz:
  fixes a :: nat  $\Rightarrow$  real
  assumes a-zero:  $a \longrightarrow 0$ 
    and monoseq a
  shows summable  $(\lambda n. (-1)^n * a \ n)$  (is ?summable)
    and  $0 < a \ 0 \longrightarrow$ 
       $(\forall n. (\sum i. (-1)^i * a \ i) \in \{ \sum i < 2*n. (-1)^i * a \ i .. \sum i < 2*n+1. (-1)^i$ 
* a i}) (is ?pos)
    and  $a \ 0 < 0 \longrightarrow$ 
       $(\forall n. (\sum i. (-1)^i * a \ i) \in \{ \sum i < 2*n+1. (-1)^i * a \ i .. \sum i < 2*n. (-1)^i$ 
* a i}) (is ?neg)
    and  $(\lambda n. \sum i < 2*n. (-1)^i * a \ i) \longrightarrow (\sum i. (-1)^i * a \ i)$  (is ?f)
    and  $(\lambda n. \sum i < 2*n+1. (-1)^i * a \ i) \longrightarrow (\sum i. (-1)^i * a \ i)$  (is ?g)
proof -
  have ?summable  $\wedge$  ?pos  $\wedge$  ?neg  $\wedge$  ?f  $\wedge$  ?g
  proof (cases  $(\forall n. 0 \leq a \ n) \wedge (\forall m. \forall n \geq m. a \ n \leq a \ m)$ )

```

```

case True
then have ord:  $\bigwedge n m. m \leq n \implies a\ n \leq a\ m$ 
  and ge0:  $\bigwedge n. 0 \leq a\ n$ 
  by auto
have mono:  $a\ (\text{Suc } n) \leq a\ n$  for  $n$ 
  using ord[where  $n = \text{Suc } n$  and  $m = n$ ] by auto
note leibniz = summable-Leibniz'[OF  $\langle a \longrightarrow 0 \rangle$  ge0]
from leibniz[OF mono]
show ?thesis using  $\langle 0 \leq a\ 0 \rangle$  by auto
next
let  $?a = \lambda n. - a\ n$ 
case False
with monoseq-le[OF  $\langle \text{monoseq } a \rangle \langle a \longrightarrow 0 \rangle$ ]
have  $(\forall n. a\ n \leq 0) \wedge (\forall m. \forall n \geq m. a\ m \leq a\ n)$  by auto
then have ord:  $\bigwedge n m. m \leq n \implies ?a\ n \leq ?a\ m$  and ge0:  $\bigwedge n. 0 \leq ?a\ n$ 
  by auto
have monotone:  $?a\ (\text{Suc } n) \leq ?a\ n$  for  $n$ 
  using ord[where  $n = \text{Suc } n$  and  $m = n$ ] by auto
note leibniz =
  summable-Leibniz'[OF - ge0, of  $\lambda x. x$ ,
  OF tendsto-minus[OF  $\langle a \longrightarrow 0 \rangle$ , unfolded minus-zero] monotone]
have summable  $(\lambda n. (-1)^\wedge n * ?a\ n)$ 
  using leibniz(1) by auto
then obtain  $l$  where  $(\lambda n. (-1)^\wedge n * ?a\ n)$  sums  $l$ 
  unfolding summable-def by auto
from this[THEN sums-minus] have  $(\lambda n. (-1)^\wedge n * a\ n)$  sums  $-l$ 
  by auto
then have ?summable by (auto simp: summable-def)
moreover
have  $|- a - - b| = |a - b|$  for  $a\ b :: \text{real}$ 
  unfolding minus-diff-minus by auto

from suminf-minus[OF leibniz(1), unfolded mult-minus-right minus-minus]
have move-minus:  $(\sum n. - ((-1)^\wedge n * a\ n)) = - (\sum n. (-1)^\wedge n * a\ n)$ 
  by auto

have ?pos using  $\langle 0 \leq ?a\ 0 \rangle$  by auto
moreover have ?neg
  using leibniz(2,4)
  unfolding mult-minus-right sum-negf move-minus neg-le-iff-le
  by auto
moreover have ?f and ?g
  using leibniz(3,5)[unfolded mult-minus-right sum-negf move-minus, THEN
tendsto-minus-cancel]
  by auto
  ultimately show ?thesis by auto
qed
then show ?summable and ?pos and ?neg and ?f and ?g
  by safe

```

qed

113.4 Term-by-Term Differentiability of Power Series

definition $\text{diffs} :: (\text{nat} \Rightarrow 'a::\text{ring-1}) \Rightarrow \text{nat} \Rightarrow 'a$
where $\text{diffs } c = (\lambda n. \text{of-nat } (\text{Suc } n) * c (\text{Suc } n))$

Lemma about distributing negation over it.

lemma diffs-minus : $\text{diffs } (\lambda n. - c n) = (\lambda n. - \text{diffs } c n)$
by ($\text{simp add: diffs-def}$)

lemma diffs-equiv :

fixes $x :: 'a::\{\text{real-normed-vector}, \text{ring-1}\}$
shows $\text{summable } (\lambda n. \text{diffs } c n * x^n) \implies$
 $(\lambda n. \text{of-nat } n * c n * x^{(n - \text{Suc } 0)}) \text{ sums } (\sum n. \text{diffs } c n * x^n)$
unfolding diffs-def
by ($\text{simp add: summable-sums sums-Suc-imp}$)

lemma lemma-termdiff1 :

fixes $z :: 'a :: \{\text{monoid-mult}, \text{comm-ring}\}$
shows $(\sum p < m. (((z + h) ^ (m - p)) * (z ^ p)) - (z ^ m)) =$
 $(\sum p < m. (z ^ p) * (((z + h) ^ (m - p)) - (z ^ (m - p))))$
by ($\text{auto simp: algebra-simps power-add [symmetric]}$)

lemma $\text{sumr-diff-mult-const2}$: $\text{sum } f \{..<n\} - \text{of-nat } n * r = (\sum i < n. f i - r)$
for $r :: 'a::\text{ring-1}$
by ($\text{simp add: sum-subtractf}$)

lemma lemma-termdiff2 :

fixes $h :: 'a::\text{field}$
assumes $h: h \neq 0$
shows $((z + h) ^ n - z ^ n) / h - \text{of-nat } n * z ^ (n - \text{Suc } 0) =$
 $h * (\sum p < n - \text{Suc } 0. \sum q < n - \text{Suc } 0 - p. (z + h) ^ q * z ^ (n - 2 -$
 $q))$
(is ?lhs = ?rhs)

proof ($\text{cases } n$)

case ($\text{Suc } m$)

have $0: \bigwedge x k. (\sum n < \text{Suc } k. h * (z ^ x * (z ^ (k - n) * (h + z) ^ n))) =$
 $(\sum j < \text{Suc } k. h * ((h + z) ^ j * z ^ (x + k - j)))$

by ($\text{auto simp add: power-add [symmetric] mult.commute intro: sum.cong}$)

have $*$: $(\sum i < m. z ^ i * ((z + h) ^ (m - i) - z ^ (m - i))) =$

$(\sum i < m. \sum j < m - i. h * ((z + h) ^ j * z ^ (m - \text{Suc } j)))$

by ($\text{force simp add: less-iff-Suc-add sum-distrib-left diff-power-eq-sum ac-simps}$)

0

$\text{simp del: sum.lessThan-Suc power-Suc intro: sum.cong}$

have $h * ?lhs = (z + h) ^ n - z ^ n - h * \text{of-nat } n * z ^ (n - \text{Suc } 0)$

by ($\text{simp add: right-diff-distrib diff-divide-distrib h mult.assoc [symmetric]}$)

also have $\dots = h * ((\sum p < \text{Suc } m. (z + h) ^ p * z ^ (m - p)) - \text{of-nat } (\text{Suc } m) * z ^ m)$

by (simp add: Suc diff-power-eq-sum h right-diff-distrib [symmetric] mult.assoc
 del: power-Suc sum.lessThan-Suc of-nat-Suc)
 also have ... = $h * ((\sum p < \text{Suc } m. (z + h) ^{(m-p)} * z ^p) - \text{of-nat } (\text{Suc } m) * z ^m)$
 by (subst sum.nat-diff-reindex[symmetric]) simp
 also have ... = $h * (\sum i < \text{Suc } m. (z + h) ^{(m-i)} * z ^i - z ^m)$
 by (simp add: sum-subtractf)
 also have ... = $h * ?rhs$
 by (simp add: lemma-termdiff1 sum-distrib-left Suc *)
 finally have $h * ?lhs = h * ?rhs$.
 then show ?thesis
 by (simp add: h)
 qed auto

lemma real-sum-nat-ivl-bounded2:
 fixes $K :: 'a::\text{linordered-semidom}$
 assumes $f: \bigwedge p::\text{nat}. p < n \implies f p \leq K$ and $K: 0 \leq K$
 shows $\text{sum } f \{..<n-k\} \leq \text{of-nat } n * K$
proof –
 have $\text{sum } f \{..<n-k\} \leq (\sum i < n - k. K)$
 by (rule sum-mono [OF f]) auto
 also have ... $\leq \text{of-nat } n * K$
 by (auto simp: mult-right-mono K)
 finally show ?thesis .
 qed

lemma lemma-termdiff3:
 fixes $h z :: 'a::\text{real-normed-field}$
 assumes 1: $h \neq 0$
 and 2: $\text{norm } z \leq K$
 and 3: $\text{norm } (z + h) \leq K$
 shows $\text{norm } (((z + h) ^n - z ^n) / h - \text{of-nat } n * z ^{(n - \text{Suc } 0)}) \leq$
 $\text{of-nat } n * \text{of-nat } (n - \text{Suc } 0) * K ^{(n - 2)} * \text{norm } h$
proof –
 have $\text{norm } (((z + h) ^n - z ^n) / h - \text{of-nat } n * z ^{(n - \text{Suc } 0)}) =$
 $\text{norm } (\sum p < n - \text{Suc } 0. \sum q < n - \text{Suc } 0 - p. (z + h) ^q * z ^{(n - 2 - q)})$
 $* \text{norm } h$
 by (metis (lifting, no-types) lemma-termdiff2 [OF 1] mult.commute norm-mult)
 also have ... $\leq \text{of-nat } n * (\text{of-nat } (n - \text{Suc } 0) * K ^{(n - 2)}) * \text{norm } h$
proof (rule mult-right-mono [OF - norm-ge-zero])
 from norm-ge-zero 2 have $K: 0 \leq K$
 by (rule order-trans)
 have le-Kn: $\text{norm } ((z + h) ^i * z ^j) \leq K ^n$ if $i + j = n$ for $i j n$
proof –
 have $\text{norm } (z + h) ^i * \text{norm } z ^j \leq K ^i * K ^j$
 by (intro mult-mono power-mono 2 3 norm-ge-zero zero-le-power K)
 also have ... = $K ^n$
 by (metis power-add that)

finally show *?thesis*
by (*simp add: norm-mult norm-power*)
qed
then have $\bigwedge p q.$
 $\llbracket p < n; q < n - \text{Suc } 0 \rrbracket \implies \text{norm } ((z + h) \wedge^q * z \wedge^{(n - 2 - q)}) \leq K \wedge^{(n - 2)}$
by (*simp del: subst-all*)
then
show $\text{norm } (\sum p < n - \text{Suc } 0. \sum q < n - \text{Suc } 0 - p. (z + h) \wedge^q * z \wedge^{(n - 2 - q)}) \leq$
 $\text{of-nat } n * (\text{of-nat } (n - \text{Suc } 0) * K \wedge^{(n - 2)})$
by (*intro order-trans [OF norm-sum]*
real-sum-nat-ivl-bounded2 mult-nonneg-nonneg of-nat-0-le-iff zero-le-power
K)
qed
also have $\dots = \text{of-nat } n * \text{of-nat } (n - \text{Suc } 0) * K \wedge^{(n - 2)} * \text{norm } h$
by (*simp only: mult.assoc*)
finally show *?thesis* .
qed

lemma *lemma-termdiff4:*

fixes $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector}$
and $k :: \text{real}$
assumes $k: 0 < k$
and $le: \bigwedge h. h \neq 0 \implies \text{norm } h < k \implies \text{norm } (f h) \leq K * \text{norm } h$
shows $f - 0 \rightarrow 0$
proof (*rule tendsto-norm-zero-cancel*)
show $(\lambda h. \text{norm } (f h)) - 0 \rightarrow 0$
proof (*rule real-tendsto-sandwich*)
show *eventually* $(\lambda h. 0 \leq \text{norm } (f h))$ (*at 0*)
by *simp*
show *eventually* $(\lambda h. \text{norm } (f h) \leq K * \text{norm } h)$ (*at 0*)
using k **by** (*auto simp: eventually-at dist-norm le*)
show $(\lambda h. 0) - (0::'a) \rightarrow (0::\text{real})$
by (*rule tendsto-const*)
have $(\lambda h. K * \text{norm } h) - (0::'a) \rightarrow K * \text{norm } (0::'a)$
by (*intro tendsto-intros*)
then show $(\lambda h. K * \text{norm } h) - (0::'a) \rightarrow 0$
by *simp*
qed
qed

lemma *lemma-termdiff5:*

fixes $g :: 'a::\text{real-normed-vector} \Rightarrow \text{nat} \Rightarrow 'b::\text{banach}$
and $k :: \text{real}$
assumes $k: 0 < k$
and $f: \text{summable } f$
and $le: \bigwedge h n. h \neq 0 \implies \text{norm } h < k \implies \text{norm } (g h n) \leq f n * \text{norm } h$
shows $(\lambda h. \text{suminf } (g h)) - 0 \rightarrow 0$

proof (rule lemma-termdiff4 [OF k])
fix $h :: 'a$
assume $h \neq 0$ **and** $\text{norm } h < k$
then have $1: \forall n. \text{norm } (g \ h \ n) \leq f \ n * \text{norm } h$
 by (simp add: le)
then have $\exists N. \forall n \geq N. \text{norm } (\text{norm } (g \ h \ n)) \leq f \ n * \text{norm } h$
 by simp
moreover from f **have** $2: \text{summable } (\lambda n. f \ n * \text{norm } h)$
 by (rule summable-mult2)
ultimately have $3: \text{summable } (\lambda n. \text{norm } (g \ h \ n))$
 by (rule summable-comparison-test)
then have $\text{norm } (\text{suminf } (g \ h)) \leq (\sum n. \text{norm } (g \ h \ n))$
 by (rule summable-norm)
also from $1 \ 3 \ 2$ **have** $(\sum n. \text{norm } (g \ h \ n)) \leq (\sum n. f \ n * \text{norm } h)$
 by (simp add: suminf-le)
also from f **have** $(\sum n. f \ n * \text{norm } h) = \text{suminf } f * \text{norm } h$
 by (rule suminf-mult2 [symmetric])
finally show $\text{norm } (\text{suminf } (g \ h)) \leq \text{suminf } f * \text{norm } h .$
qed

lemma termdiffs-aux:

fixes $x :: 'a :: \{\text{real-normed-field, banach}\}$
assumes $1: \text{summable } (\lambda n. \text{diffs } (\text{diffs } c) \ n * K \ ^n)$
 and $2: \text{norm } x < \text{norm } K$
shows $(\lambda h. \sum n. c \ n * ((x + h) \ ^n - x \ ^n) / h - \text{of-nat } n * x \ ^{(n - \text{Suc } 0)})$
 $-0 \rightarrow 0$
proof –
 from dense [OF 2] **obtain** r **where** $r1: \text{norm } x < r$ **and** $r2: r < \text{norm } K$
 by fast
 from norm-ge-zero $r1$ **have** $r: 0 < r$
 by (rule order-le-less-trans)
 then have $r\text{-neq-0}: r \neq 0$ **by** simp
 show ?thesis
 proof (rule lemma-termdiff5)
 show $0 < r - \text{norm } x$
 using $r1$ **by** simp
 from $r \ r2$ **have** $\text{norm } (\text{of-real } r :: 'a) < \text{norm } K$
 by simp
 with 1 **have** $\text{summable } (\lambda n. \text{norm } (\text{diffs } (\text{diffs } c) \ n * (\text{of-real } r \ ^n)))$
 by (rule powser-insidea)
 then have $\text{summable } (\lambda n. \text{diffs } (\text{diffs } (\lambda n. \text{norm } (c \ n))) \ n * r \ ^n)$
 using r **by** (simp add: diff-def norm-mult norm-power del: of-nat-Suc)
 then have $\text{summable } (\lambda n. \text{of-nat } n * \text{diffs } (\lambda n. \text{norm } (c \ n)) \ n * r \ ^{(n - \text{Suc } 0}))$
 by (rule diff-equiv [THEN sums-summable])
 also have $(\lambda n. \text{of-nat } n * \text{diffs } (\lambda n. \text{norm } (c \ n)) \ n * r \ ^{(n - \text{Suc } 0)}) =$

```

      (λn. diffs (λm. of-nat (m - Suc 0) * norm (c m) * inverse r) n * (r
^ n))
    by (simp add: diffs-def r-neq-0 fun-eq-iff split: nat-diff-split)
  finally have summable
    (λn. of-nat n * (of-nat (n - Suc 0) * norm (c n) * inverse r) * r ^ (n - Suc
0))
  by (rule diffs-equiv [THEN sums-summable])
  also have
    (λn. of-nat n * (of-nat (n - Suc 0) * norm (c n) * inverse r) * r ^ (n - Suc
0)) =
    (λn. norm (c n) * of-nat n * of-nat (n - Suc 0) * r ^ (n - 2))
  by (rule ext) (simp add: r-neq-0 split: nat-diff-split)
  finally show summable (λn. norm (c n) * of-nat n * of-nat (n - Suc 0) * r
^ (n - 2)) .
next
  fix h :: 'a and n
  assume h: h ≠ 0
  assume norm h < r - norm x
  then have norm x + norm h < r by simp
  with norm-triangle-ineq
  have xh: norm (x + h) < r
    by (rule order-le-less-trans)
  have norm (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0))
  ≤ real n * (real (n - Suc 0) * (r ^ (n - 2) * norm h))
  by (metis (mono-tags, lifting) h mult.assoc lemma-termdiff3 less-eq-real-def
r1 xh)
  then show norm (c n * (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc
0))) ≤
  norm (c n) * of-nat n * of-nat (n - Suc 0) * r ^ (n - 2) * norm h
  by (simp only: norm-mult mult.assoc mult-left-mono [OF - norm-ge-zero])
qed
qed

```

lemma *termdiffs*:

```

  fixes K x :: 'a::{real-normed-field,banach}
  assumes 1: summable (λn. c n * K ^ n)
    and 2: summable (λn. (diffs c) n * K ^ n)
    and 3: summable (λn. (diffs (diffs c)) n * K ^ n)
    and 4: norm x < norm K
  shows DERIV (λx. ∑ n. c n * x ^ n) x := (∑ n. (diffs c) n * x ^ n)
  unfolding DERIV-def
  proof (rule LIM-zero-cancel)
  show (λh. (suminf (λn. c n * (x + h) ^ n) - suminf (λn. c n * x ^ n)) / h
- suminf (λn. diffs c n * x ^ n)) - 0 → 0
  proof (rule LIM-equal2)
  show 0 < norm K - norm x
  using 4 by (simp add: less-diff-eq)
  next
  fix h :: 'a

```



```

assume  $\text{norm } (h - 0) < \text{norm } K - \text{norm } x$ 
then have  $\text{norm } x + \text{norm } h < \text{norm } K$  by simp
then have  $5: \text{norm } (x + h) < \text{norm } K$ 
  by (rule norm-triangle-ineq [THEN order-le-less-trans])
have summable  $(\lambda n. c\ n * x^{\wedge} n)$ 
  and summable  $(\lambda n. c\ n * (x + h)^{\wedge} n)$ 
  and summable  $(\lambda n. \text{diffs } c\ n * x^{\wedge} n)$ 
  using  $1\ 2\ 4\ 5$  by (auto elim: powser-inside)
then have  $((\sum n. c\ n * (x + h)^{\wedge} n) - (\sum n. c\ n * x^{\wedge} n)) / h - (\sum n. \text{diffs } c\ n * x^{\wedge} n) =$ 
   $(\sum n. (c\ n * (x + h)^{\wedge} n - c\ n * x^{\wedge} n) / h - \text{of-nat } n * c\ n * x^{\wedge} (n - \text{Suc } 0))$ 
  by (intro sums-unique sums-diff sums-divide diffs-equiv summable-sums)
then show  $((\sum n. c\ n * (x + h)^{\wedge} n) - (\sum n. c\ n * x^{\wedge} n)) / h - (\sum n. \text{diffs } c\ n * x^{\wedge} n) =$ 
   $(\sum n. c\ n * ((x + h)^{\wedge} n - x^{\wedge} n) / h - \text{of-nat } n * x^{\wedge} (n - \text{Suc } 0))$ 
  by (simp add: algebra-simps)
next
  show  $(\lambda h. \sum n. c\ n * ((x + h)^{\wedge} n - x^{\wedge} n) / h - \text{of-nat } n * x^{\wedge} (n - \text{Suc } 0)) - 0 \rightarrow 0$ 
  by (rule termdiffs-aux [OF 3 4])
qed
qed

```

113.5 The Derivative of a Power Series Has the Same Radius of Convergence

```

lemma termdiff-converges:
  fixes  $x :: 'a :: \{\text{real-normed-field, banach}\}$ 
  assumes  $K: \text{norm } x < K$ 
  and  $sm: \bigwedge x. \text{norm } x < K \implies \text{summable}(\lambda n. c\ n * x^{\wedge} n)$ 
  shows summable  $(\lambda n. \text{diffs } c\ n * x^{\wedge} n)$ 
proof (cases  $x = 0$ )
  case True
  then show ?thesis
    using powser-sums-zero sums-summable by auto
  next
  case False
  then have  $K > 0$ 
    using K less-trans zero-less-norm-iff by blast
  then obtain  $r :: \text{real}$  where  $r: \text{norm } x < \text{norm } r \ \text{norm } r < K \ r > 0$ 
    using K False
  by (auto simp: field-simps abs-less-iff add-pos-pos intro: that [of (norm x + K) / 2])
  have to0:  $(\lambda n. \text{of-nat } n * (x / \text{of-real } r)^{\wedge} n) \longrightarrow 0$ 
    using  $r$  by (simp add: norm-divide powser-times-n-limit-0 [of x / of-real r])
  obtain  $N$  where  $N: \bigwedge n. n \geq N \implies \text{real-of-nat } n * \text{norm } x^{\wedge} n < r^{\wedge} n$ 
    using  $r$  LIMSEQ-D [OF to0, of 1]
  by (auto simp: norm-divide norm-mult norm-power field-simps)

```

```

have summable ( $\lambda n. (of\text{-}nat\ n * c\ n) * x \wedge n$ )
proof (rule summable-comparison-test')
  show summable ( $\lambda n. norm\ (c\ n * of\text{-}real\ r \wedge n)$ )
    apply (rule powser-insidea [OF sm [of of-real ((r+K)/2)]])
    using N r norm-of-real [of r + K, where 'a = 'a] by auto
  show  $\bigwedge n. N \leq n \implies norm\ (of\text{-}nat\ n * c\ n * x \wedge n) \leq norm\ (c\ n * of\text{-}real\ r \wedge n)$ 
    using N r by (fastforce simp add: norm-mult norm-power less-eq-real-def)
qed
then have summable ( $\lambda n. (of\text{-}nat\ (Suc\ n) * c(Suc\ n)) * x \wedge Suc\ n$ )
  using summable-iff-shift [of  $\lambda n. of\text{-}nat\ n * c\ n * x \wedge n\ 1$ ]
  by simp
then have summable ( $\lambda n. (of\text{-}nat\ (Suc\ n) * c(Suc\ n)) * x \wedge n$ )
  using False summable-mult2 [of  $\lambda n. (of\text{-}nat\ (Suc\ n) * c(Suc\ n)) * x \wedge n$ ] * x
  inverse x]
  by (simp add: mult.assoc) (auto simp: ac-simps)
then show ?thesis
  by (simp add: diffs-def)
qed

```

lemma *termdiff-converges-all*:

```

fixes x :: 'a::{real-normed-field,banach}
assumes  $\bigwedge x. summable\ (\lambda n. c\ n * x \wedge n)$ 
shows summable ( $\lambda n. diffs\ c\ n * x \wedge n$ )
by (rule termdiff-converges [where  $K = 1 + norm\ x$ ]) (use assms in auto)

```

lemma *termdiffs-strong*:

```

fixes K x :: 'a::{real-normed-field,banach}
assumes sm: summable ( $\lambda n. c\ n * K \wedge n$ )
  and K: norm x < norm K
shows DERIV ( $\lambda x. \sum n. c\ n * x \wedge n$ ) x := ( $\sum n. diffs\ c\ n * x \wedge n$ )
proof –
  have norm K + norm x < norm K + norm K
    using K by force
  then have K2: norm ((of-real (norm K) + of-real (norm x)) / 2 :: 'a) < norm K
    by (auto simp: norm-triangle-lt norm-divide field-simps)
  then have [simp]: norm ((of-real (norm K) + of-real (norm x)) :: 'a) < norm K * 2
    by simp
  have summable ( $\lambda n. c\ n * (of\text{-}real\ (norm\ x + norm\ K) / 2) \wedge n$ )
    by (metis K2 summable-norm-cancel [OF powser-insidea [OF sm]] add commute of-real-add)
  moreover have  $\bigwedge x. norm\ x < norm\ K \implies summable\ (\lambda n. diffs\ c\ n * x \wedge n)$ 
    by (blast intro: sm termdiff-converges powser-inside)
  moreover have  $\bigwedge x. norm\ x < norm\ K \implies summable\ (\lambda n. diffs(diffs\ c)\ n * x \wedge n)$ 
    by (blast intro: sm termdiff-converges powser-inside)
  ultimately show ?thesis

```

by (rule termdiffs [where $K = \text{of-real } (\text{norm } x + \text{norm } K) / 2$])
 (use K in $\langle \text{auto simp: field-simps simp flip: of-real-add} \rangle$)

qed

lemma termdiffs-strong-converges-everywhere:

fixes $K x :: 'a :: \{\text{real-normed-field, banach}\}$
 assumes $\bigwedge y. \text{summable } (\lambda n. c n * y ^ n)$
 shows $((\lambda x. \sum n. c n * x ^ n) \text{ has-field-derivative } (\sum n. \text{diffs } c n * x ^ n)) \text{ (at } x)$
 using termdiffs-strong[OF assms[of of-real (norm x + 1)], of x]
 by (force simp del: of-real-add)

lemma termdiffs-strong':

fixes $z :: 'a :: \{\text{real-normed-field, banach}\}$
 assumes $\bigwedge z. \text{norm } z < K \implies \text{summable } (\lambda n. c n * z ^ n)$
 assumes $\text{norm } z < K$
 shows $((\lambda z. \sum n. c n * z ^ n) \text{ has-field-derivative } (\sum n. \text{diffs } c n * z ^ n)) \text{ (at } z)$
 proof (rule termdiffs-strong)
 define $L :: \text{real}$ where $L = (\text{norm } z + K) / 2$
 have $0 \leq \text{norm } z$ by simp
 also note $\langle \text{norm } z < K \rangle$
 finally have $K: K \geq 0$ by simp
 from assms K have $L: L \geq 0$ $\text{norm } z < L$ $L < K$ by (simp-all add: L-def)
 from L show $\text{norm } z < \text{norm } (\text{of-real } L :: 'a)$ by simp
 from L show $\text{summable } (\lambda n. c n * \text{of-real } L ^ n)$ by (intro assms(1)) simp-all

qed

lemma termdiffs-sums-strong:

fixes $z :: 'a :: \{\text{banach, real-normed-field}\}$
 assumes $\text{sums: } \bigwedge z. \text{norm } z < K \implies (\lambda n. c n * z ^ n) \text{ sums } f z$
 assumes $\text{deriv: } (f \text{ has-field-derivative } f') \text{ (at } z)$
 assumes $\text{norm: } \text{norm } z < K$
 shows $(\lambda n. \text{diffs } c n * z ^ n) \text{ sums } f'$

proof –

have $\text{summable: } \text{summable } (\lambda n. \text{diffs } c n * z ^ n)$
 by (intro termdiff-converges[OF norm] sums-summable[OF sums])
 from norm have $\text{eventually } (\lambda z. z \in \text{norm} - \{.. < K\}) \text{ (nhds } z)$
 by (intro eventually-nhds-in-open open-vimage)
 (simp-all add: continuous-on-norm)
 hence $\text{eq: eventually } (\lambda z. (\sum n. c n * z ^ n) = f z) \text{ (nhds } z)$
 by eventually-elim (insert sums, simp add: sums-iff)

have $((\lambda z. \sum n. c n * z ^ n) \text{ has-field-derivative } (\sum n. \text{diffs } c n * z ^ n)) \text{ (at } z)$

by (intro termdiffs-strong'[OF - norm] sums-summable[OF sums])

hence $(f \text{ has-field-derivative } (\sum n. \text{diffs } c n * z ^ n)) \text{ (at } z)$

by (subst (asm) DERIV-cong-ev[OF refl eq refl])

from this and deriv have $(\sum n. \text{diffs } c n * z ^ n) = f'$ by (rule DERIV-unique)

with summable show ?thesis by (simp add: sums-iff)

qed

lemma *isCont-powser*:

fixes $K x :: 'a::\{\text{real-normed-field}, \text{banach}\}$

assumes *summable* $(\lambda n. c n * K ^ n)$

assumes $\text{norm } x < \text{norm } K$

shows *isCont* $(\lambda x. \sum n. c n * x ^ n) x$

using *termdiffs-strong*[*OF assms*] **by** (*blast intro!*: *DERIV-isCont*)

lemmas *isCont-powser'* = *isCont-o2*[*OF - isCont-powser*]

lemma *isCont-powser-converges-everywhere*:

fixes $K x :: 'a::\{\text{real-normed-field}, \text{banach}\}$

assumes $\bigwedge y. \text{summable } (\lambda n. c n * y ^ n)$

shows *isCont* $(\lambda x. \sum n. c n * x ^ n) x$

using *termdiffs-strong*[*OF assms*[*of of-real (norm x + 1)*], *of x*]

by (*force intro!*: *DERIV-isCont simp del: of-real-add*)

lemma *powser-limit-0*:

fixes $a :: \text{nat} \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$

assumes $s: 0 < s$

and *sm*: $\bigwedge x. \text{norm } x < s \implies (\lambda n. a n * x ^ n) \text{ sums } (f x)$

shows $(f \longrightarrow a 0) \text{ (at } 0)$

proof –

have $\text{norm } (\text{of-real } s / 2 :: 'a) < s$

using *s* **by** (*auto simp: norm-divide*)

then have *summable* $(\lambda n. a n * (\text{of-real } s / 2) ^ n)$

by (*rule sums-summable* [*OF sm*])

then have $((\lambda x. \sum n. a n * x ^ n) \text{ has-field-derivative } (\sum n. \text{diffs } a n * 0 ^ n))$
(*at 0*)

by (*rule termdiffs-strong*) (*use s in <auto simp: norm-divide>*)

then have *isCont* $(\lambda x. \sum n. a n * x ^ n) 0$

by (*blast intro: DERIV-continuous*)

then have $((\lambda x. \sum n. a n * x ^ n) \longrightarrow a 0) \text{ (at } 0)$

by (*simp add: continuous-within*)

moreover have $(\lambda x. f x - (\sum n. a n * x ^ n)) -0 \rightarrow 0$

apply (*clarsimp simp: LIM-eq*)

apply (*rule-tac x=s in exI*)

using *s sm sums-unique* **by** *fastforce*

ultimately show *?thesis*

by (*rule Lim-transform*)

qed

lemma *powser-limit-0-strong*:

fixes $a :: \text{nat} \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$

assumes $s: 0 < s$

and *sm*: $\bigwedge x. x \neq 0 \implies \text{norm } x < s \implies (\lambda n. a n * x ^ n) \text{ sums } (f x)$

shows $(f \longrightarrow a 0) \text{ (at } 0)$

proof –

have $*$: $((\lambda x. \text{if } x = 0 \text{ then } a 0 \text{ else } f x) \longrightarrow a 0) \text{ (at } 0)$

by (*rule powser-limit-0* [*OF s*]) (*auto simp: powser-sums-zero sm*)

show *?thesis*
 using * by (auto cong: *Lim-cong-within*)
 qed

113.6 Derivability of power series

lemma *DERIV-series'*:

fixes $f :: \text{real} \Rightarrow \text{nat} \Rightarrow \text{real}$
 assumes *DERIV-f*: $\bigwedge n. \text{DERIV } (\lambda x. f x n) x0 :> (f' x0 n)$
 and *allf-summable*: $\bigwedge x. x \in \{a <..< < b\} \Longrightarrow \text{summable } (f x)$
 and *x0-in-I*: $x0 \in \{a <..< < b\}$
 and *summable (f' x0)*
 and *summable L*
 and *L-def*: $\bigwedge n x y. x \in \{a <..< < b\} \Longrightarrow y \in \{a <..< < b\} \Longrightarrow |f x n - f y n| \leq$
 $L n * |x - y|$
 shows *DERIV* $(\lambda x. \text{suminf } (f x)) x0 :> (\text{suminf } (f' x0))$
 unfolding *DERIV-def*
 proof (rule *LIM-I*)
 fix $r :: \text{real}$
 assume $0 < r$ then have $0 < r/3$ by auto

obtain *N-L* where *N-L*: $\bigwedge n. N-L \leq n \Longrightarrow |\sum i. L (i + n)| < r/3$
 using *suminf-exist-split*[*OF* $\langle 0 < r/3 \rangle \langle \text{summable } L \rangle$] by auto

obtain *N-f'* where *N-f'*: $\bigwedge n. N-f' \leq n \Longrightarrow |\sum i. f' x0 (i + n)| < r/3$
 using *suminf-exist-split*[*OF* $\langle 0 < r/3 \rangle \langle \text{summable } (f' x0) \rangle$] by auto

let $?N = \text{Suc } (\max N-L N-f')$
 have $|\sum i. f' x0 (i + ?N)| < r/3$ (is *?f'-part* $< r/3$)
 and *L-estimate*: $|\sum i. L (i + ?N)| < r/3$
 using *N-L*[*of ?N*] and *N-f'* [*of ?N*] by auto

let $?diff = \lambda i x. (f (x0 + x) i - f x0 i) / x$

let $?r = r / (3 * \text{real } ?N)$
 from $\langle 0 < r \rangle$ have $0 < ?r$ by simp

let $?s = \lambda n. \text{SOME } s. 0 < s \wedge (\forall x. x \neq 0 \wedge |x| < s \longrightarrow |?diff n x - f' x0 n| < ?r)$

define *S'* where $S' = \text{Min } (?s ' \{..< ?N\})$

have $0 < S'$

unfolding *S'-def*

proof (rule *iffD2*[*OF Min-gr-iff*])

show $\forall x \in (?s ' \{..< ?N\}). 0 < x$

proof

fix x

assume $x \in ?s ' \{..< ?N\}$

then obtain n where $x = ?s n$ and $n \in \{..< ?N\}$

using *image-iff*[*THEN iffD1*] **by** *blast*
from *DERIV-D*[*OF DERIV-f*[**where** $n=n$], *THEN LIM-D*, *OF* $\langle 0 < ?r \rangle$,
unfolded real-norm-def]
obtain s **where** *s-bound*: $0 < s \wedge (\forall x. x \neq 0 \wedge |x| < s \longrightarrow |\text{?diff } n \ x - f' \ x0 \ n| < ?r)$
by *auto*
have $0 < ?s \ n$
by (*rule someI2*[**where** $a=s$]) (*auto simp: s-bound simp del: of-nat-Suc*)
then show $0 < x$ **by** (*simp only: $\langle x = ?s \ n \rangle$*)
qed
qed *auto*

define S **where** $S = \min (\min (x0 - a) (b - x0)) \ S'$
then have $0 < S$ **and** *S-a*: $S \leq x0 - a$ **and** *S-b*: $S \leq b - x0$
and $S \leq S'$ **using** *x0-in-I* **and** $\langle 0 < S' \rangle$
by *auto*

have $|(\text{suminf } (f (x0 + x)) - \text{suminf } (f x0)) / x - \text{suminf } (f' x0)| < r$
if $x \neq 0$ **and** $|x| < S$ **for** x
proof –
from *that* **have** *x-in-I*: $x0 + x \in \{a <..< b\}$
using *S-a S-b* **by** *auto*

note *diff-smbL* = *summable-diff*[*OF allf-summable*[*OF x-in-I*] *allf-summable*[*OF x0-in-I*]]
note *div-smbL* = *summable-divide*[*OF diff-smbL*]
note *all-smbL* = *summable-diff*[*OF div-smbL* $\langle \text{summable } (f' x0) \rangle$]
note *ign* = *summable-ignore-initial-segment*[**where** $k=?N$]
note *diff-shft-smbL* = *summable-diff*[*OF ign*[*OF allf-summable*[*OF x-in-I*]]
ign[*OF allf-summable*[*OF x0-in-I*]]]
note *div-shft-smbL* = *summable-divide*[*OF diff-shft-smbL*]
note *all-shft-smbL* = *summable-diff*[*OF div-smbL ign*[*OF* $\langle \text{summable } (f' x0) \rangle$]]

have $1: |(\text{?diff } (n + ?N) \ x)| \leq L (n + ?N)$ **for** n

proof –
have $|\text{?diff } (n + ?N) \ x| \leq L (n + ?N) * |(x0 + x) - x0| / |x|$
using *divide-right-mono*[*OF L-def*[*OF x-in-I x0-in-I*] *abs-ge-zero*]
by (*simp only: abs-divide*)
with $\langle x \neq 0 \rangle$ **show** *?thesis* **by** *auto*

qed
note $2 = \text{summable-rabs-comparison-test}$ [*OF* - *ign*[*OF* $\langle \text{summable } L \rangle$]]

from 1 **have** $|\sum i. \text{?diff } (i + ?N) \ x| \leq (\sum i. L (i + ?N))$

by (*metis* (*lifting*) *abs-idempotent*

order-trans[*OF summable-rabs*[*OF* 2] *suminf-le*[*OF* - 2 *ign*[*OF* $\langle \text{summable } L \rangle$]]])

then have $|\sum i. \text{?diff } (i + ?N) \ x| \leq r / 3$ (**is** *?L-part* $\leq r/3$)

using *L-estimate* **by** *auto*

have $|\sum n < ?N. \text{?diff } n \ x - f' \ x0 \ n| \leq (\sum n < ?N. |\text{?diff } n \ x - f' \ x0 \ n|) ..$

also have $\dots < (\sum n < ?N. ?r)$
proof (*rule sum-strict-mono*)
fix n
assume $n \in \{.. < ?N\}$
have $|x| < S$ **using** $\langle |x| < S \rangle$.
also have $S \leq S'$ **using** $\langle S \leq S' \rangle$.
also have $S' \leq ?s n$
unfolding S' -*def*
proof (*rule Min-le-iff[THEN iffD2]*)
have $?s n \in (?s ' \{.. < ?N\}) \wedge ?s n \leq ?s n$
using $\langle n \in \{.. < ?N\} \rangle$ **by** *auto*
then show $\exists a \in (?s ' \{.. < ?N\}). a \leq ?s n$
by *blast*
qed *auto*
finally have $|x| < ?s n$.

from *DERIV-D[OF DERIV-f[where n=n], THEN LIM-D, OF $\langle 0 < ?r \rangle$,
unfolded real-norm-def diff-0-right, unfolded some-eq-ex[symmetric], THEN
conjunct2]*
have $\forall x. x \neq 0 \wedge |x| < ?s n \longrightarrow |?diff n x - f' x0 n| < ?r$.
with $\langle x \neq 0 \rangle$ **and** $\langle |x| < ?s n \rangle$ **show** $|?diff n x - f' x0 n| < ?r$
by *blast*
qed *auto*
also have $\dots = \text{of-nat } (\text{card } \{.. < ?N\}) * ?r$
by (*rule sum-constant*)
also have $\dots = \text{real } ?N * ?r$
by *simp*
also have $\dots = r/3$
by (*auto simp del: of-nat-Suc*)
finally have $|\sum n < ?N. ?diff n x - f' x0 n| < r / 3$ (**is** $?diff\text{-part} < r / 3$) .

from *suminf-diff[OF allf-summable[OF x-in-I] allf-summable[OF x0-in-I]]*
have $|(suminf (f (x0 + x)) - (suminf (f x0))) / x - suminf (f' x0)| =$
 $|\sum n. ?diff n x - f' x0 n|$
unfolding *suminf-diff[OF div-smb1 $\langle summable (f' x0) \rangle$, symmetric]*
using *suminf-divide[OF diff-smb1, symmetric]* **by** *auto*
also have $\dots \leq ?diff\text{-part} + |(\sum n. ?diff (n + ?N) x) - (\sum n. f' x0 (n +$
 $?N))|$
unfolding *suminf-split-initial-segment[OF all-smb1, where k=?N]*
unfolding *suminf-diff[OF div-shft-smb1 ign[OF $\langle summable (f' x0) \rangle$]]*
apply (*simp only: add commute*)
using *abs-triangle-ineq* **by** *blast*
also have $\dots \leq ?diff\text{-part} + ?L\text{-part} + ?f'\text{-part}$
using *abs-triangle-ineq4* **by** *auto*
also have $\dots < r / 3 + r/3 + r/3$
using $\langle ?diff\text{-part} < r/3 \rangle$ $\langle ?L\text{-part} \leq r/3 \rangle$ **and** $\langle ?f'\text{-part} < r/3 \rangle$
by (*rule add-strict-mono [OF add-less-le-mono]*)
finally show *?thesis*
by *auto*

qed
then show $\exists s > 0. \forall x. x \neq 0 \wedge \text{norm } (x - 0) < s \longrightarrow$
 $\text{norm } (((\sum n. f (x0 + x) n) - (\sum n. f x0 n)) / x - (\sum n. f' x0 n)) < r$
using $\langle 0 < S \rangle$ **by auto**
qed

lemma *DERIV-power-series'*:
fixes $f :: \text{nat} \Rightarrow \text{real}$
assumes *converges*: $\bigwedge x. x \in \{-R <..< R\} \implies \text{summable } (\lambda n. f n * \text{real } (\text{Suc } n) * x^{\wedge} n)$
and *x0-in-I*: $x0 \in \{-R <..< R\}$
and $0 < R$
shows *DERIV* $(\lambda x. (\sum n. f n * x^{\wedge}(\text{Suc } n))) x0 :> (\sum n. f n * \text{real } (\text{Suc } n) * x0^{\wedge} n)$
(is *DERIV* $(\lambda x. \text{suminf } (?f x)) x0 :> \text{suminf } (?f' x0)$ *)*
proof –
have *for-subinterval*: *DERIV* $(\lambda x. \text{suminf } (?f x)) x0 :> \text{suminf } (?f' x0)$
if $0 < R'$ **and** $R' < R$ **and** $-R' < x0$ **and** $x0 < R'$ **for** R'
proof –
from *that* **have** $x0 \in \{-R' <..< R'\}$ **and** $R' \in \{-R <..< R\}$ **and** $x0 \in \{-R <..< R\}$
by auto
show *?thesis*
proof (*rule* *DERIV-series'*)
show *summable* $(\lambda n. |f n * \text{real } (\text{Suc } n) * R'^{\wedge} n|)$
proof –
have $(R' + R) / 2 < R$ **and** $0 < (R' + R) / 2$
using $\langle 0 < R' \rangle \langle 0 < R \rangle \langle R' < R \rangle$ **by** (*auto simp: field-simps*)
then have *in-Rball*: $(R' + R) / 2 \in \{-R <..< R\}$
using $\langle R' < R \rangle$ **by auto**
have *norm* $R' < \text{norm } ((R' + R) / 2)$
using $\langle 0 < R' \rangle \langle 0 < R \rangle \langle R' < R \rangle$ **by** (*auto simp: field-simps*)
from *powser-insidea*[*OF* *converges*[*OF* *in-Rball*] *this*] **show** *?thesis*
by auto
qed

next
fix $n x y$
assume $x \in \{-R' <..< R'\}$ **and** $y \in \{-R' <..< R'\}$
show $|\text{?f } x n - \text{?f } y n| \leq |f n * \text{real } (\text{Suc } n) * R'^{\wedge} n| * |x - y|$
proof –
have $|f n * x^{\wedge}(\text{Suc } n) - f n * y^{\wedge}(\text{Suc } n)| =$
 $(|f n| * |x - y|) * |\sum p < \text{Suc } n. x^{\wedge} p * y^{\wedge}(n - p)|$
unfolding *right-diff-distrib*[*symmetric*] *diff-power-eq-sum abs-mult*
by auto
also have $\dots \leq (|f n| * |x - y|) * (|\text{real } (\text{Suc } n)| * |R'^{\wedge} n|)$
proof (*rule* *mult-left-mono*)
have $|\sum p < \text{Suc } n. x^{\wedge} p * y^{\wedge}(n - p)| \leq (\sum p < \text{Suc } n. |x^{\wedge} p * y^{\wedge}(n - p)|)$
by (*rule* *sum-abs*)


```

also have ...  $\leq (\sum p < \text{Suc } n. R' \wedge n)$ 
proof (rule sum-mono)
  fix p
  assume  $p \in \{.. < \text{Suc } n\}$ 
  then have  $p \leq n$  by auto
  have  $|x \wedge n| \leq R' \wedge n$  if  $x \in \{-R' < .. < R'\}$  for n and  $x :: \text{real}$ 
  proof -
    from that have  $|x| \leq R'$  by auto
    then show ?thesis
      unfolding power-abs by (rule power-mono) auto
    qed
  from mult-mono[OF this[OF  $\langle x \in \{-R' < .. < R'\} \rangle$ , of p] this[OF  $\langle y \in \{-R' < .. < R'\} \rangle$ , of n-p]]
  and  $\langle 0 < R' \rangle$ 
  have  $|x \wedge p * y \wedge (n - p)| \leq R' \wedge p * R' \wedge (n - p)$ 
  unfolding abs-mult by auto
  then show  $|x \wedge p * y \wedge (n - p)| \leq R' \wedge n$ 
  unfolding power-add[symmetric] using  $\langle p \leq n \rangle$  by auto
  qed
  also have ... = real (Suc n) *  $R' \wedge n$ 
  unfolding sum-constant card-atLeastLessThan by auto
  finally show  $|\sum p < \text{Suc } n. x \wedge p * y \wedge (n - p)| \leq |\text{real } (\text{Suc } n)| * |R' \wedge n|$ 
  unfolding abs-of-nonneg[OF zero-le-power[OF less-imp-le[OF  $\langle 0 < R' \rangle$ ]]]
  by linarith
  show  $0 \leq |f n| * |x - y|$ 
  unfolding abs-mult[symmetric] by auto
  qed
  also have ... =  $|f n| * \text{real } (\text{Suc } n) * R' \wedge n * |x - y|$ 
  unfolding abs-mult mult.assoc[symmetric] by algebra
  finally show ?thesis .
  qed
next
  show DERIV  $(\lambda x. ?f x n) x0 :> ?f' x0 n$  for n
  by (auto intro!: derivative-eq-intros simp del: power-Suc)
next
  fix x
  assume  $x \in \{-R' < .. < R'\}$ 
  then have  $R' \in \{-R < .. < R\}$  and norm x < norm R'
  using assms  $\langle R' < R \rangle$  by auto
  have summable  $(\lambda n. f n * x \wedge n)$ 
  proof (rule summable-comparison-test, intro exI allI impI)
    fix n
    have le:  $|f n| * 1 \leq |f n| * \text{real } (\text{Suc } n)$ 
    by (rule mult-left-mono) auto
    show norm  $(f n * x \wedge n) \leq \text{norm } (f n * \text{real } (\text{Suc } n) * x \wedge n)$ 
    unfolding real-norm-def abs-mult
    using le mult-right-mono by fastforce
  qed (rule powser-insidea[OF converges[OF  $\langle R' \in \{-R < .. < R\} \rangle$ ]  $\langle \text{norm } x < \text{norm } R' \rangle$ ])

```

```

from this[THEN summable-mult2[where  $c=x$ ], simplified mult.assoc, simplified mult commute]
  show summable (?f x) by auto
next
  show summable (?f' x0)
    using converges[OF  $\langle x0 \in \{-R <..< R\} \rangle$ ] .
  show  $x0 \in \{-R' <..< R'\}$ 
    using  $\langle x0 \in \{-R' <..< R'\} \rangle$  .
  qed
qed
let ?R = (R + |x0|) / 2
have |x0| < ?R
  using assms by (auto simp: field-simps)
then have - ?R < x0
proof (cases  $x0 < 0$ )
  case True
    then have - x0 < ?R
      using  $\langle |x0| < ?R \rangle$  by auto
    then show ?thesis
      unfolding neg-less-iff-less[symmetric, of - x0] by auto
  next
    case False
      have - ?R < 0 using assms by auto
      also have ...  $\leq x0$  using False by auto
      finally show ?thesis .
  qed
then have  $0 < ?R$  ?R < R - ?R < x0 and  $x0 < ?R$ 
  using assms by (auto simp: field-simps)
from for-subinterval[OF this] show ?thesis .
qed

```

```

lemma geometric-deriv-sums:
  fixes z :: 'a :: {real-normed-field,banach}
  assumes norm z < 1
  shows ( $\lambda n.$  of-nat (Suc n) *  $z^n$ ) sums (1 / (1 - z)^2)
proof -
  have ( $\lambda n.$  diffs ( $\lambda n.$  1) n *  $z^n$ ) sums (1 / (1 - z)^2)
  proof (rule termdiffs-sums-strong)
    fix z :: 'a assume norm z < 1
    thus ( $\lambda n.$  1 *  $z^n$ ) sums (1 / (1 - z)) by (simp add: geometric-sums)
  qed (insert assms, auto intro!: derivative-eq-intros simp: power2-eq-square)
  thus ?thesis unfolding diffs-def by simp
qed

```

```

lemma isCont-pochhammer [continuous-intros]: isCont ( $\lambda z.$  pochhammer z n) z
  for z :: 'a::real-normed-field
  by (induct n) (auto simp: pochhammer-rec')

```

```

lemma continuous-on-pochhammer [continuous-intros]: continuous-on A ( $\lambda z.$  pochham-

```

```

mer z n)
  for A :: 'a::real-normed-field set
  by (intro continuous-at-imp-continuous-on ballI isCont-pochhammer)

lemmas continuous-on-pochhammer' [continuous-intros] =
  continuous-on-compose2[OF continuous-on-pochhammer - subset-UNIV]

```

113.7 Exponential Function

```

definition exp :: 'a ⇒ 'a::{real-normed-algebra-1,banach}
  where exp = (λx. ∑ n. x^n /_R fact n)

```

lemma *summable-exp-generic*:

```

fixes x :: 'a::{real-normed-algebra-1,banach}
defines S-def: S ≡ λn. x^n /_R fact n
shows summable S

```

proof –

```

have S-Suc: ∧n. S (Suc n) = (x * S n) /_R (Suc n)
  unfolding S-def by (simp del: mult-Suc)

```

```

obtain r :: real where r0: 0 < r and r1: r < 1
  using dense [OF zero-less-one] by fast

```

```

obtain N :: nat where N: norm x < real N * r
  using ex-less-of-nat-mult r0 by auto

```

```

from r1 show ?thesis

```

```

proof (rule summable-ratio-test [rule-format])

```

```

  fix n :: nat

```

```

  assume n: N ≤ n

```

```

  have norm x ≤ real N * r

```

```

    using N by (rule order-less-imp-le)

```

```

  also have real N * r ≤ real (Suc n) * r

```

```

    using r0 n by (simp add: mult-right-mono)

```

```

  finally have norm x * norm (S n) ≤ real (Suc n) * r * norm (S n)

```

```

    using norm-ge-zero by (rule mult-right-mono)

```

```

  then have norm (x * S n) ≤ real (Suc n) * r * norm (S n)

```

```

    by (rule order-trans [OF norm-mult-ineq])

```

```

  then have norm (x * S n) / real (Suc n) ≤ r * norm (S n)

```

```

    by (simp add: pos-divide-le-eq ac-simps)

```

```

  then show norm (S (Suc n)) ≤ r * norm (S n)

```

```

    by (simp add: S-Suc inverse-eq-divide)

```

qed

qed

lemma *summable-norm-exp*: summable (λn. norm (x^n /_R fact n))

```

  for x :: 'a::{real-normed-algebra-1,banach}

```

proof (rule summable-norm-comparison-test [OF exI, rule-format])

```

  show summable (λn. norm x^n /_R fact n)

```

```

    by (rule summable-exp-generic)

```

```

  show norm (x^n /_R fact n) ≤ norm x^n /_R fact n for n

```

```

    by (simp add: norm-power-ineq)

```

qed

lemma *summable-exp*: *summable* ($\lambda n.$ *inverse* (*fact* n) * $x^{\wedge} n$)
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
using *summable-exp-generic* [**where** $x=x$]
by (*simp add: scaleR-conv-of-real nonzero-of-real-inverse*)

lemma *exp-converges*: ($\lambda n.$ $x^{\wedge} n /_R$ *fact* n) *sums* *exp* x
unfolding *exp-def* **by** (*rule summable-exp-generic* [*THEN summable-sums*])

lemma *exp-fdiffs*:
diffs ($\lambda n.$ *inverse* (*fact* n)) = ($\lambda n.$ *inverse* (*fact* $n :: 'a::\{\text{real-normed-field}, \text{banach}\}$))
by (*simp add: diffs-def mult-ac nonzero-inverse-mult-distrib nonzero-of-real-inverse del: mult-Suc of-nat-Suc*)

lemma *diffs-of-real*: *diffs* ($\lambda n.$ *of-real* (f n)) = ($\lambda n.$ *of-real* (*diffs* f n))
by (*simp add: diffs-def*)

lemma *DERIV-exp* [*simp*]: *DERIV* *exp* x $:$ $>$ *exp* x

unfolding *exp-def scaleR-conv-of-real*

proof (*rule DERIV-cong*)

have *sinv*: *summable* ($\lambda n.$ *of-real* (*inverse* (*fact* n)) * $x^{\wedge} n$) **for** $x::'a$

by (*rule exp-converges* [*THEN sums-summable, unfolded scaleR-conv-of-real*])

note $xx = \text{exp-converges}$ [*THEN sums-summable, unfolded scaleR-conv-of-real*]

show ($(\lambda x.$ $\sum n.$ *of-real* (*inverse* (*fact* n)) * $x^{\wedge} n$) *has-field-derivative*

$(\sum n.$ *diffs* ($\lambda n.$ *of-real* (*inverse* (*fact* n))) $n * x^{\wedge} n$) (at x)

by (*rule termdiffs* [**where** $K=\text{of-real } (1 + \text{norm } x)$]) (*simp-all only: diffs-of-real exp-fdiffs sinv norm-of-real*)

show ($\sum n.$ *diffs* ($\lambda n.$ *of-real* (*inverse* (*fact* n))) $n * x^{\wedge} n$) = ($\sum n.$ *of-real* (*inverse* (*fact* n)) * $x^{\wedge} n$)

by (*simp add: diffs-of-real exp-fdiffs*)

qed

declare *DERIV-exp*[*THEN DERIV-chain2, derivative-intros*]

and *DERIV-exp*[*THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros*]

lemmas *has-derivative-exp*[*derivative-intros*] = *DERIV-exp*[*THEN DERIV-compose-FDERIV*]

lemma *norm-exp*: *norm* (*exp* x) \leq *exp* (*norm* x)

proof –

from *summable-norm*[*OF summable-norm-exp, of x*]

have *norm* (*exp* x) \leq ($\sum n.$ *inverse* (*fact* n) * *norm* ($x^{\wedge} n$))

by (*simp add: exp-def*)

also have $\dots \leq \text{exp } (\text{norm } x)$

using *summable-exp-generic*[*of norm x*] *summable-norm-exp*[*of x*]

by (*auto simp: exp-def intro!: suminf-le norm-power-ineq*)

finally show *?thesis* .

qed

lemma *isCont-exp*: *isCont exp x*
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
by (rule *DERIV-exp* [*THEN DERIV-isCont*])

lemma *isCont-exp'* [*simp*]: *isCont f a \implies isCont ($\lambda x. \text{exp } (f x)$) a*
for $f :: - \implies 'a::\{\text{real-normed-field}, \text{banach}\}$
by (rule *isCont-o2* [*OF - isCont-exp*])

lemma *tendsto-exp* [*tendsto-intros*]: *(f \longrightarrow a) F \implies (($\lambda x. \text{exp } (f x)$) \longrightarrow exp a) F*
for $f :: - \implies 'a::\{\text{real-normed-field}, \text{banach}\}$
by (rule *isCont-tendsto-compose* [*OF isCont-exp*])

lemma *continuous-exp* [*continuous-intros*]: *continuous F f \implies continuous F ($\lambda x. \text{exp } (f x)$)*
for $f :: - \implies 'a::\{\text{real-normed-field}, \text{banach}\}$
unfolding *continuous-def* **by** (rule *tendsto-exp*)

lemma *continuous-on-exp* [*continuous-intros*]: *continuous-on s f \implies continuous-on s ($\lambda x. \text{exp } (f x)$)*
for $f :: - \implies 'a::\{\text{real-normed-field}, \text{banach}\}$
unfolding *continuous-on-def* **by** (auto intro: *tendsto-exp*)

113.7.1 Properties of the Exponential Function

lemma *exp-zero* [*simp*]: *exp 0 = 1*
unfolding *exp-def* **by** (*simp add: scaleR-conv-of-real*)

lemma *exp-series-add-commuting*:
fixes $x y :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
defines *S-def*: $S \equiv \lambda x n. x^n /_R \text{fact } n$
assumes *comm*: $x * y = y * x$
shows $S (x + y) n = (\sum_{i \leq n}. S x i * S y (n - i))$
proof (*induct n*)
case 0
show ?case
unfolding *S-def* **by** *simp*
next
case (*Suc n*)
have *S-Suc*: $\bigwedge x n. S x (\text{Suc } n) = (x * S x n) /_R \text{real } (\text{Suc } n)$
unfolding *S-def* **by** (*simp del: mult-Suc*)
then have *times-S*: $\bigwedge x n. x * S x n = \text{real } (\text{Suc } n) *_R S x (\text{Suc } n)$
by *simp*
have *S-comm*: $\bigwedge n. S x n * y = y * S x n$
by (*simp add: power-commuting-commutes comm S-def*)
have $\text{real } (\text{Suc } n) *_R S (x + y) (\text{Suc } n) = (x + y) * (\sum_{i \leq n}. S x i * S y (n - i))$

by (*metis Suc.hyps times-S*)
 also have ... = $x * (\sum_{i \leq n}. S x i * S y (n - i)) + y * (\sum_{i \leq n}. S x i * S y (n - i))$
 by (*rule distrib-right*)
 also have ... = $(\sum_{i \leq n}. x * S x i * S y (n - i)) + (\sum_{i \leq n}. S x i * y * S y (n - i))$
 by (*simp add: sum-distrib-left ac-simps S-comm*)
 also have ... = $(\sum_{i \leq n}. x * S x i * S y (n - i)) + (\sum_{i \leq n}. S x i * (y * S y (n - i)))$
 by (*simp add: ac-simps*)
 also have ... = $(\sum_{i \leq n}. \text{real } (Suc i) *_R (S x (Suc i) * S y (n - i))) + (\sum_{i \leq n}. \text{real } (Suc n - i) *_R (S x i * S y (Suc n - i)))$
 by (*simp add: times-S Suc-diff-le*)
 also have $(\sum_{i \leq n}. \text{real } (Suc i) *_R (S x (Suc i) * S y (n - i))) = (\sum_{i \leq Suc n}. \text{real } i *_R (S x i * S y (Suc n - i)))$
 by (*subst sum.atMost-Suc-shift*) *simp*
 also have $(\sum_{i \leq n}. \text{real } (Suc n - i) *_R (S x i * S y (Suc n - i))) = (\sum_{i \leq Suc n}. \text{real } (Suc n - i) *_R (S x i * S y (Suc n - i)))$
 by *simp*
 also have $(\sum_{i \leq Suc n}. \text{real } i *_R (S x i * S y (Suc n - i))) + (\sum_{i \leq Suc n}. \text{real } (Suc n - i) *_R (S x i * S y (Suc n - i))) = (\sum_{i \leq Suc n}. \text{real } (Suc n) *_R (S x i * S y (Suc n - i)))$
 by (*simp flip: sum.distrib scaleR-add-left of-nat-add*)
 also have ... = $\text{real } (Suc n) *_R (\sum_{i \leq Suc n}. S x i * S y (Suc n - i))$
 by (*simp only: scaleR-right.sum*)
 finally show $S (x + y) (Suc n) = (\sum_{i \leq Suc n}. S x i * S y (Suc n - i))$
 by (*simp del: sum.cl-ivl-Suc*)
 qed

lemma *exp-add-commuting*: $x * y = y * x \implies \text{exp } (x + y) = \text{exp } x * \text{exp } y$
 by (*simp only: exp-def Cauchy-product summable-norm-exp exp-series-add-commuting*)

lemma *exp-times-arg-commute*: $\text{exp } A * A = A * \text{exp } A$
 by (*simp add: exp-def suminf-mult[symmetric] summable-exp-generic power-commutes suminf-mult2*)

lemma *exp-add*: $\text{exp } (x + y) = \text{exp } x * \text{exp } y$
 for $x y :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 by (*rule exp-add-commuting*) (*simp add: ac-simps*)

lemma *exp-double*: $\text{exp}(2 * z) = \text{exp } z ^ 2$
 by (*simp add: exp-add-commuting mult-2 power2-eq-square*)

lemmas *mult-exp-exp = exp-add* [*symmetric*]

lemma *exp-of-real*: $\text{exp } (\text{of-real } x) = \text{of-real } (\text{exp } x)$
 unfolding *exp-def*
 apply (*subst suminf-of-real [OF summable-exp-generic]*)
 apply (*simp add: scaleR-conv-of-real*)

done

lemmas *of-real-exp* = *exp-of-real*[*symmetric*]

corollary *exp-in-Reals* [*simp*]: $z \in \mathbb{R} \implies \exp z \in \mathbb{R}$
by (*metis Reals-cases Reals-of-real exp-of-real*)

lemma *exp-not-eq-zero* [*simp*]: $\exp x \neq 0$

proof

have $\exp x * \exp (-x) = 1$
by (*simp add: exp-add-commuting[symmetric]*)
also assume $\exp x = 0$
finally show *False* **by** *simp*

qed

lemma *exp-minus-inverse*: $\exp x * \exp (-x) = 1$
by (*simp add: exp-add-commuting[symmetric]*)

lemma *exp-minus*: $\exp (-x) = \text{inverse} (\exp x)$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
by (*intro inverse-unique [symmetric] exp-minus-inverse*)

lemma *exp-diff*: $\exp (x - y) = \exp x / \exp y$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
using *exp-add [of x - y]* **by** (*simp add: exp-minus divide-inverse*)

lemma *exp-of-nat-mult*: $\exp (\text{of-nat } n * x) = \exp x ^ n$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
by (*induct n*) (*auto simp: distrib-left exp-add mult.commute*)

corollary *exp-of-nat2-mult*: $\exp (x * \text{of-nat } n) = \exp x ^ n$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
by (*metis exp-of-nat-mult mult-of-nat-commute*)

lemma *exp-sum*: $\text{finite } I \implies \exp (\text{sum } f I) = \text{prod } (\lambda x. \exp (f x)) I$
by (*induct I rule: finite-induct*) (*auto simp: exp-add-commuting mult.commute*)

lemma *exp-divide-power-eq*:

fixes $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes $n > 0$
shows $\exp (x / \text{of-nat } n) ^ n = \exp x$
using *assms*

proof (*induction n arbitrary: x*)

case (*Suc n*)

show *?case*

proof (*cases n = 0*)

case *True*

then show *?thesis* **by** *simp*

next

```

case False
have [simp]:  $1 + (\text{of-nat } n * \text{of-nat } n + \text{of-nat } n * 2) \neq (0::'a)$ 
  using of-nat-eq-iff [of  $1 + n * n + n * 2$   $0$ ]
  by simp
from False have [simp]:  $x * \text{of-nat } n / (1 + \text{of-nat } n) / \text{of-nat } n = x / (1 +$ 
of-nat } n)
  by simp
have [simp]:  $x / (1 + \text{of-nat } n) + x * \text{of-nat } n / (1 + \text{of-nat } n) = x$ 
  using of-nat-neq-0
  by (auto simp add: field-split-simps)
show ?thesis
  using Suc.IH [of  $x * \text{of-nat } n / (1 + \text{of-nat } n)$ ] False
  by (simp add: exp-add [symmetric])
qed
qed simp

```

```

lemma exp-power-int:
  fixes  $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$ 
  shows  $\text{exp } x \text{ powi } n = \text{exp } (\text{of-int } n * x)$ 
proof (cases  $n \geq 0$ )
  case True
  have  $\text{exp } x \text{ powi } n = \text{exp } x \wedge \text{nat } n$ 
    using True by (simp add: power-int-def)
  thus ?thesis
    using True by (subst (asm) exp-of-nat-mult [symmetric]) auto
next
  case False
  have  $\text{exp } x \text{ powi } n = \text{inverse } (\text{exp } x \wedge \text{nat } (-n))$ 
    using False by (simp add: power-int-def field-simps)
  also have  $\text{exp } x \wedge \text{nat } (-n) = \text{exp } (\text{of-nat } (\text{nat } (-n)) * x)$ 
    using False by (subst exp-of-nat-mult) auto
  also have  $\text{inverse } \dots = \text{exp } (-(\text{of-nat } (\text{nat } (-n)) * x))$ 
    by (subst exp-minus) (auto simp: field-simps)
  also have  $-(\text{of-nat } (\text{nat } (-n)) * x) = \text{of-int } n * x$ 
    using False by simp
  finally show ?thesis .
qed

```

113.7.2 Properties of the Exponential Function on Reals

Comparisons of $\text{exp } x$ with zero.

Proof: because every exponential can be seen as a square.

```

lemma exp-ge-zero [simp]:  $0 \leq \text{exp } x$ 
  for  $x :: \text{real}$ 
proof –
  have  $0 \leq \text{exp } (x/2) * \text{exp } (x/2)$ 
    by simp
  then show ?thesis

```


by (*simp add: exp-add [symmetric]*)
qed

lemma *exp-gt-zero* [*simp*]: $0 < \exp x$
 for $x :: \text{real}$
 by (*simp add: order-less-le*)

lemma *not-exp-less-zero* [*simp*]: $\neg \exp x < 0$
 for $x :: \text{real}$
 by (*simp add: not-less*)

lemma *not-exp-le-zero* [*simp*]: $\neg \exp x \leq 0$
 for $x :: \text{real}$
 by (*simp add: not-le*)

lemma *abs-exp-cancel* [*simp*]: $|\exp x| = \exp x$
 for $x :: \text{real}$
 by *simp*

Strict monotonicity of exponential.

lemma *exp-ge-add-one-self-aux*:
 fixes $x :: \text{real}$
 assumes $0 \leq x$
 shows $1 + x \leq \exp x$
 using *order-le-imp-less-or-eq [OF assms]*
proof
 assume $0 < x$
 have $1 + x \leq (\sum n < 2. \text{inverse} (\text{fact } n) * x^n)$
 by (*auto simp: numeral-2-eq-2*)
 also have $\dots \leq (\sum n. \text{inverse} (\text{fact } n) * x^n)$
 using $\langle 0 < x \rangle$ by (*auto simp add: zero-le-mult-iff intro: sum-le-suminf [OF summable-exp]*)
 finally show $1 + x \leq \exp x$
 by (*simp add: exp-def*)
qed *auto*

lemma *exp-gt-one*: $0 < x \implies 1 < \exp x$
 for $x :: \text{real}$
proof –
 assume $x: 0 < x$
 then have $1 < 1 + x$ by *simp*
 also from x have $1 + x \leq \exp x$
 by (*simp add: exp-ge-add-one-self-aux*)
 finally show *?thesis* .
qed

lemma *exp-less-mono*:
 fixes $x y :: \text{real}$
 assumes $x < y$

shows $\exp x < \exp y$

proof –

from $\langle x < y \rangle$ **have** $0 < y - x$ **by** *simp*

then have $1 < \exp (y - x)$ **by** (*rule exp-gt-one*)

then have $1 < \exp y / \exp x$ **by** (*simp only: exp-diff*)

then show $\exp x < \exp y$ **by** *simp*

qed

lemma *exp-less-cancel*: $\exp x < \exp y \implies x < y$

for $x y :: \text{real}$

unfolding *linorder-not-le* [*symmetric*]

by (*auto simp: order-le-less exp-less-mono*)

lemma *exp-less-cancel-iff* [*iff*]: $\exp x < \exp y \longleftrightarrow x < y$

for $x y :: \text{real}$

by (*auto intro: exp-less-mono exp-less-cancel*)

lemma *exp-le-cancel-iff* [*iff*]: $\exp x \leq \exp y \longleftrightarrow x \leq y$

for $x y :: \text{real}$

by (*auto simp: linorder-not-less [symmetric]*)

lemma *exp-inj-iff* [*iff*]: $\exp x = \exp y \longleftrightarrow x = y$

for $x y :: \text{real}$

by (*simp add: order-eq-iff*)

Comparisons of $\exp x$ with one.

lemma *one-less-exp-iff* [*simp*]: $1 < \exp x \longleftrightarrow 0 < x$

for $x :: \text{real}$

using *exp-less-cancel-iff* [**where** $x = 0$ **and** $y = x$] **by** *simp*

lemma *exp-less-one-iff* [*simp*]: $\exp x < 1 \longleftrightarrow x < 0$

for $x :: \text{real}$

using *exp-less-cancel-iff* [**where** $x = x$ **and** $y = 0$] **by** *simp*

lemma *one-le-exp-iff* [*simp*]: $1 \leq \exp x \longleftrightarrow 0 \leq x$

for $x :: \text{real}$

using *exp-le-cancel-iff* [**where** $x = 0$ **and** $y = x$] **by** *simp*

lemma *exp-le-one-iff* [*simp*]: $\exp x \leq 1 \longleftrightarrow x \leq 0$

for $x :: \text{real}$

using *exp-le-cancel-iff* [**where** $x = x$ **and** $y = 0$] **by** *simp*

lemma *exp-eq-one-iff* [*simp*]: $\exp x = 1 \longleftrightarrow x = 0$

for $x :: \text{real}$

using *exp-inj-iff* [**where** $x = x$ **and** $y = 0$] **by** *simp*

lemma *lemma-exp-total*: $1 \leq y \implies \exists x. 0 \leq x \wedge x \leq y - 1 \wedge \exp x = y$

for $y :: \text{real}$

proof (*rule IVT*)

```

assume  $1 \leq y$ 
then have  $0 \leq y - 1$  by simp
then have  $1 + (y - 1) \leq \exp (y - 1)$ 
  by (rule exp-ge-add-one-self-aux)
then show  $y \leq \exp (y - 1)$  by simp
qed (simp-all add: le-diff-eq)

```

```

lemma exp-total:  $0 < y \implies \exists x. \exp x = y$ 
  for  $y :: \text{real}$ 
proof (rule linorder-le-cases [of 1 y])
  assume  $1 \leq y$ 
  then show  $\exists x. \exp x = y$ 
    by (fast dest: lemma-exp-total)
next
  assume  $0 < y$  and  $y < 1$ 
  then have  $1 \leq \text{inverse } y$ 
    by (simp add: one-le-inverse-iff)
  then obtain  $x$  where  $\exp x = \text{inverse } y$ 
    by (fast dest: lemma-exp-total)
  then have  $\exp (-x) = y$ 
    by (simp add: exp-minus)
  then show  $\exists x. \exp x = y$  ..
qed

```

113.8 Natural Logarithm

```

class ln = real-normed-algebra-1 + banach +
  fixes  $\ln :: 'a \Rightarrow 'a$ 
  assumes ln-one [simp]:  $\ln 1 = 0$ 

```

```

definition powr ::  $'a \Rightarrow 'a \Rightarrow 'a::\ln$  (infixr powr 80)
  — exponentiation via ln and exp
  where  $x \text{ powr } a \equiv \text{if } x = 0 \text{ then } 0 \text{ else } \exp (a * \ln x)$ 

```

```

lemma powr-0 [simp]:  $0 \text{ powr } z = 0$ 
  by (simp add: powr-def)

```

```

instantiation real :: ln
begin

```

```

definition ln-real ::  $\text{real} \Rightarrow \text{real}$ 
  where  $\text{ln-real } x = (\text{THE } u. \exp u = x)$ 

```

```

instance
  by intro-classes (simp add: ln-real-def)

```

```

end

```

lemma *powr-eq-0-iff* [simp]: $w \text{ powr } z = 0 \longleftrightarrow w = 0$
by (*simp add: powr-def*)

lemma *ln-exp* [simp]: $\ln (\text{exp } x) = x$
for $x :: \text{real}$
by (*simp add: ln-real-def*)

lemma *exp-ln* [simp]: $0 < x \implies \text{exp } (\ln x) = x$
for $x :: \text{real}$
by (*auto dest: exp-total*)

lemma *exp-ln-iff* [simp]: $\text{exp } (\ln x) = x \longleftrightarrow 0 < x$
for $x :: \text{real}$
by (*metis exp-gt-zero exp-ln*)

lemma *ln-unique*: $\text{exp } y = x \implies \ln x = y$
for $x :: \text{real}$
by (*erule subst*) (*rule ln-exp*)

lemma *ln-mult*: $0 < x \implies 0 < y \implies \ln (x * y) = \ln x + \ln y$
for $x :: \text{real}$
by (*rule ln-unique*) (*simp add: exp-add*)

lemma *ln-prod*: $\text{finite } I \implies (\bigwedge i. i \in I \implies f i > 0) \implies \ln (\text{prod } f I) = \text{sum } (\lambda x. \ln(f x)) I$
for $f :: 'a \Rightarrow \text{real}$
by (*induct I rule: finite-induct*) (*auto simp: ln-mult prod-pos*)

lemma *ln-inverse*: $0 < x \implies \ln (\text{inverse } x) = - \ln x$
for $x :: \text{real}$
by (*rule ln-unique*) (*simp add: exp-minus*)

lemma *ln-div*: $0 < x \implies 0 < y \implies \ln (x / y) = \ln x - \ln y$
for $x :: \text{real}$
by (*rule ln-unique*) (*simp add: exp-diff*)

lemma *ln-realpow*: $0 < x \implies \ln (x^{\widehat{n}}) = \text{real } n * \ln x$
by (*rule ln-unique*) (*simp add: exp-of-nat-mult*)

lemma *ln-less-cancel-iff* [simp]: $0 < x \implies 0 < y \implies \ln x < \ln y \longleftrightarrow x < y$
for $x :: \text{real}$
by (*subst exp-less-cancel-iff [symmetric]*) *simp*

lemma *ln-le-cancel-iff* [simp]: $0 < x \implies 0 < y \implies \ln x \leq \ln y \longleftrightarrow x \leq y$
for $x :: \text{real}$
by (*simp add: linorder-not-less [symmetric]*)

lemma *ln-inj-iff* [simp]: $0 < x \implies 0 < y \implies \ln x = \ln y \longleftrightarrow x = y$
for $x :: \text{real}$

by (simp add: order-eq-iff)

lemma *ln-add-one-self-le-self*: $0 \leq x \implies \ln(1 + x) \leq x$
 for $x :: \text{real}$
 by (rule exp-le-cancel-iff [THEN iffD1]) (simp add: exp-ge-add-one-self-aux)

lemma *ln-less-self* [simp]: $0 < x \implies \ln x < x$
 for $x :: \text{real}$
 by (rule order-less-le-trans [where $y = \ln(1 + x)$]) (simp-all add: ln-add-one-self-le-self)

lemma *ln-ge-iff*: $\bigwedge x :: \text{real}. 0 < x \implies y \leq \ln x \longleftrightarrow \exp y \leq x$
 using exp-le-cancel-iff exp-total by force

lemma *ln-ge-zero* [simp]: $1 \leq x \implies 0 \leq \ln x$
 for $x :: \text{real}$
 using ln-le-cancel-iff [of 1 x] by simp

lemma *ln-ge-zero-imp-ge-one*: $0 \leq \ln x \implies 0 < x \implies 1 \leq x$
 for $x :: \text{real}$
 using ln-le-cancel-iff [of 1 x] by simp

lemma *ln-ge-zero-iff* [simp]: $0 < x \implies 0 \leq \ln x \longleftrightarrow 1 \leq x$
 for $x :: \text{real}$
 using ln-le-cancel-iff [of 1 x] by simp

lemma *ln-less-zero-iff* [simp]: $0 < x \implies \ln x < 0 \longleftrightarrow x < 1$
 for $x :: \text{real}$
 using ln-less-cancel-iff [of x 1] by simp

lemma *ln-le-zero-iff* [simp]: $0 < x \implies \ln x \leq 0 \longleftrightarrow x \leq 1$
 for $x :: \text{real}$
 by (metis less-numeral-extra(1) ln-le-cancel-iff ln-one)

lemma *ln-gt-zero*: $1 < x \implies 0 < \ln x$
 for $x :: \text{real}$
 using ln-less-cancel-iff [of 1 x] by simp

lemma *ln-gt-zero-imp-gt-one*: $0 < \ln x \implies 0 < x \implies 1 < x$
 for $x :: \text{real}$
 using ln-less-cancel-iff [of 1 x] by simp

lemma *ln-gt-zero-iff* [simp]: $0 < x \implies 0 < \ln x \longleftrightarrow 1 < x$
 for $x :: \text{real}$
 using ln-less-cancel-iff [of 1 x] by simp

lemma *ln-eq-zero-iff* [simp]: $0 < x \implies \ln x = 0 \longleftrightarrow x = 1$
 for $x :: \text{real}$
 using ln-inj-iff [of x 1] by simp

lemma *ln-less-zero*: $0 < x \implies x < 1 \implies \ln x < 0$
for $x :: \text{real}$
by *simp*

lemma *ln-neg-is-const*: $x \leq 0 \implies \ln x = (\text{THE } x. \text{False})$
for $x :: \text{real}$
by (*auto simp: ln-real-def intro!: arg-cong*[**where** $f = \text{The}$])

lemma *powr-eq-one-iff* [*simp*]:
 $a \text{ powr } x = 1 \iff x = 0 \text{ if } a > 1$ **for** $a :: \text{real}$
using *that* **by** (*auto simp: powr-def split: if-splits*)

lemma *isCont-ln*:
fixes $x :: \text{real}$
assumes $x \neq 0$
shows *isCont* $\ln x$
proof (*cases* $0 < x$)
case *True*
then have *isCont* $\ln (\exp (\ln x))$
by (*intro isCont-inverse-function*[**where** $d = |x|$ **and** $f = \exp$]) *auto*
with *True* **show** *?thesis*
by *simp*
next
case *False*
with $\langle x \neq 0 \rangle$ **show** *isCont* $\ln x$
unfolding *isCont-def*
by (*subst filterlim-cong*[*OF* - *refl*, *of* - *nhds* ($\ln 0$) - $\lambda\cdot. \ln 0$])
(*auto simp: ln-neg-is-const not-less eventually-at dist-real-def*
intro!: exI[*of* - $|x|$])
qed

lemma *tendsto-ln* [*tendsto-intros*]: $(f \longrightarrow a) F \implies a \neq 0 \implies ((\lambda x. \ln (f x)) \longrightarrow \ln a) F$
for $a :: \text{real}$
by (*rule isCont-tendsto-compose* [*OF isCont-ln*])

lemma *continuous-ln*:
 $\text{continuous } F f \implies f (\text{Lim } F (\lambda x. x)) \neq 0 \implies \text{continuous } F (\lambda x. \ln (f x :: \text{real}))$
unfolding *continuous-def* **by** (*rule tendsto-ln*)

lemma *isCont-ln'* [*continuous-intros*]:
 $\text{continuous } (\text{at } x) f \implies f x \neq 0 \implies \text{continuous } (\text{at } x) (\lambda x. \ln (f x :: \text{real}))$
unfolding *continuous-at* **by** (*rule tendsto-ln*)

lemma *continuous-within-ln* [*continuous-intros*]:
 $\text{continuous } (\text{at } x \text{ within } s) f \implies f x \neq 0 \implies \text{continuous } (\text{at } x \text{ within } s) (\lambda x. \ln (f x :: \text{real}))$
unfolding *continuous-within* **by** (*rule tendsto-ln*)

lemma *continuous-on-ln* [*continuous-intros*]:
continuous-on s f $\implies (\forall x \in s. f x \neq 0) \implies \text{continuous-on } s (\lambda x. \ln (f x :: \text{real}))$
unfolding *continuous-on-def* **by** (*auto intro: tendsto-ln*)

lemma *DERIV-ln*: $0 < x \implies \text{DERIV } \ln x \text{ :> inverse } x$
for $x :: \text{real}$
by (*rule DERIV-inverse-function* [**where** $f = \text{exp}$ **and** $a = 0$ **and** $b = x + 1$])
(*auto intro: DERIV-cong* [*OF DERIV-exp exp-ln*] *isCont-ln*)

lemma *DERIV-ln-divide*: $0 < x \implies \text{DERIV } \ln x \text{ :> } 1 / x$
for $x :: \text{real}$
by (*rule DERIV-ln*[*THEN DERIV-cong*]) (*simp-all add: divide-inverse*)

declare *DERIV-ln-divide*[*THEN DERIV-chain2, derivative-intros*]
and *DERIV-ln-divide*[*THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros*]

lemmas *has-derivative-ln*[*derivative-intros*] = *DERIV-ln*[*THEN DERIV-compose-FDERIV*]

lemma *ln-series*:
assumes $0 < x$ **and** $x < 2$
shows $\ln x = (\sum n. (-1)^{\wedge} n * (1 / \text{real } (n + 1)) * (x - 1)^{\wedge} (\text{Suc } n))$
(*is ln x = suminf (?f (x - 1))*)
proof –
let $?f' = \lambda x n. (-1)^{\wedge} n * (x - 1)^{\wedge} n$

have $\ln x - \text{suminf } (?f (x - 1)) = \ln 1 - \text{suminf } (?f (1 - 1))$
proof (*rule DERIV-isconst3* [**where** $x = x$])
fix $x :: \text{real}$
assume $x \in \{0 <..< 2\}$
then have $0 < x$ **and** $x < 2$ **by** *auto*
have $\text{norm } (1 - x) < 1$
using $\langle 0 < x \rangle$ **and** $\langle x < 2 \rangle$ **by** *auto*
have $1 / x = 1 / (1 - (1 - x))$ **by** *auto*
also have $\dots = (\sum n. (1 - x)^{\wedge} n)$
using *geometric-sums*[*OF* $\langle \text{norm } (1 - x) < 1 \rangle$] **by** (*rule sums-unique*)
also have $\dots = \text{suminf } (?f' x)$
unfolding *power-mult-distrib*[*symmetric*]
by (*rule arg-cong*[**where** $f = \text{suminf}$], *rule arg-cong*[**where** $f = (\wedge)$], *auto*)
finally have *DERIV ln x :> suminf (?f' x)*
using *DERIV-ln*[*OF* $\langle 0 < x \rangle$] **unfolding** *divide-inverse* **by** *auto*
moreover
have *repos*: $\bigwedge h x :: \text{real}. h - 1 + x = h + x - 1$ **by** *auto*
have *DERIV* $(\lambda x. \text{suminf } (?f x)) (x - 1) \text{ :>}$
 $(\sum n. (-1)^{\wedge} n * (1 / \text{real } (n + 1)) * \text{real } (\text{Suc } n) * (x - 1)^{\wedge} n)$
proof (*rule DERIV-power-series*)
show $x - 1 \in \{-1 <..< 1\}$ **and** $(0 :: \text{real}) < 1$
using $\langle 0 < x \rangle$ $\langle x < 2 \rangle$ **by** *auto*
next

```

fix x :: real
assume x ∈ {− 1 <..< 1}
then show summable (λn. (− 1) ^ n * (1 / real (n + 1)) * real (Suc n) *
x ^ n)
  by (simp add: abs-if flip: power-mult-distrib)
qed
then have DERIV (λx. suminf (?f x)) (x − 1) :> suminf (?f' x)
  unfolding One-nat-def by auto
then have DERIV (λx. suminf (?f (x − 1))) x :> suminf (?f' x)
  unfolding DERIV-def repos .
ultimately have DERIV (λx. ln x − suminf (?f (x − 1))) x :> suminf (?f'
x) − suminf (?f' x)
  by (rule DERIV-diff)
then show DERIV (λx. ln x − suminf (?f (x − 1))) x :> 0 by auto
qed (auto simp: assms)
then show ?thesis by auto
qed

```

lemma exp-first-terms:

```

fixes x :: 'a::{real-normed-algebra-1,banach}
shows exp x = (∑ n<k. inverse(fact n) *R (x ^ n)) + (∑ n. inverse(fact (n +
k)) *R (x ^ (n + k)))
proof −
  have exp x = suminf (λn. inverse(fact n) *R (x ^ n))
    by (simp add: exp-def)
  also from summable-exp-generic have ... = (∑ n. inverse(fact(n+k)) *R (x ^
(n + k))) +
    (∑ n::nat<k. inverse(fact n) *R (x ^ n)) (is - = - + ?a)
    by (rule suminf-split-initial-segment)
  finally show ?thesis by simp
qed

```

lemma exp-first-term: exp x = 1 + (∑ n. inverse (fact (Suc n)) *_R (x ^ Suc n))
for x :: 'a::{real-normed-algebra-1,banach}
using exp-first-terms[of x 1] **by** simp

lemma exp-first-two-terms: exp x = 1 + x + (∑ n. inverse (fact (n + 2)) *_R (x
^ (n + 2)))
for x :: 'a::{real-normed-algebra-1,banach}
using exp-first-terms[of x 2] **by** (simp add: eval-nat-numeral)

lemma exp-bound:

```

fixes x :: real
assumes a: 0 ≤ x
  and b: x ≤ 1
shows exp x ≤ 1 + x + x2
proof −
  have suminf (λn. inverse(fact (n+2)) * (x ^ (n + 2))) ≤ x2
proof −

```



```

have ( $\lambda n. x^2 / 2 * (1/2) ^ n$ ) sums ( $x^2 / 2 * (1 / (1 - 1/2))$ )
  by (intro sums-mult geometric-sums) simp
then have sumsx: ( $\lambda n. x^2 / 2 * (1/2) ^ n$ ) sums  $x^2$ 
  by simp
have suminf ( $\lambda n. \text{inverse}(\text{fact } (n+2)) * (x ^ (n + 2))$ )  $\leq$  suminf ( $\lambda n. (x^2/2)$ 
*  $((1/2) ^ n)$ )
proof (intro suminf-le allI)
  show  $\text{inverse}(\text{fact } (n + 2)) * x ^ (n + 2) \leq (x^2/2) * ((1/2) ^ n)$  for  $n :: \text{nat}$ 
proof –
  have  $(2::\text{nat}) * 2 ^ n \leq \text{fact } (n + 2)$ 
    by (induct n) simp-all
  then have  $\text{real } ((2::\text{nat}) * 2 ^ n) \leq \text{real-of-nat } (\text{fact } (n + 2))$ 
    by (simp only: of-nat-le-iff)
  then have  $((2::\text{real}) * 2 ^ n) \leq \text{fact } (n + 2)$ 
    unfolding of-nat-fact by simp
  then have  $\text{inverse}(\text{fact } (n + 2)) \leq \text{inverse}((2::\text{real}) * 2 ^ n)$ 
    by (rule le-imp-inverse-le) simp
  then have  $\text{inverse}(\text{fact } (n + 2)) \leq 1/(2::\text{real}) * (1/2) ^ n$ 
    by (simp add: power-inverse [symmetric])
  then have  $\text{inverse}(\text{fact } (n + 2)) * (x ^ n * x^2) \leq 1/2 * (1/2) ^ n * (1 * x^2)$ 
    by (rule mult-mono) (rule mult-mono, simp-all add: power-le-one a b)
  then show ?thesis
    unfolding power-add by (simp add: ac-simps del: fact-Suc)
qed
show summable ( $\lambda n. \text{inverse}(\text{fact } (n + 2)) * x ^ (n + 2)$ )
  by (rule summable-exp [THEN summable-ignore-initial-segment])
show summable ( $\lambda n. x^2 / 2 * (1/2) ^ n$ )
  by (rule sums-summable [OF sumsx])
qed
also have  $\dots = x^2$ 
  by (rule sums-unique [THEN sym]) (rule sumsx)
finally show ?thesis .
qed
then show ?thesis
  unfolding exp-first-two-terms by auto
qed

corollary exp-half-le2:  $\text{exp}(1/2) \leq (2::\text{real})$ 
  using exp-bound [of 1/2]
  by (simp add: field-simps)

corollary exp-le:  $\text{exp } 1 \leq (3::\text{real})$ 
  using exp-bound [of 1]
  by (simp add: field-simps)

lemma exp-bound-half:  $\text{norm } z \leq 1/2 \implies \text{norm } (\text{exp } z) \leq 2$ 
  by (blast intro: order-trans intro!: exp-half-le2 norm-exp)

lemma exp-bound-lemma:

```

assumes $\text{norm } z \leq 1/2$
shows $\text{norm } (\exp z) \leq 1 + 2 * \text{norm } z$
proof –
have $*$: $(\text{norm } z)^2 \leq \text{norm } z * 1$
unfolding *power2-eq-square*
by (*rule mult-left-mono*) (*use assms in auto*)
have $\text{norm } (\exp z) \leq \exp (\text{norm } z)$
by (*rule norm-exp*)
also have $\dots \leq 1 + (\text{norm } z) + (\text{norm } z)^2$
using *assms exp-bound by auto*
also have $\dots \leq 1 + 2 * \text{norm } z$
using $*$ **by** *auto*
finally show *?thesis* .
qed

lemma *real-exp-bound-lemma*: $0 \leq x \implies x \leq 1/2 \implies \exp x \leq 1 + 2 * x$
for $x :: \text{real}$
using *exp-bound-lemma [of x] by simp*

lemma *ln-one-minus-pos-upper-bound*:
fixes $x :: \text{real}$
assumes a : $0 \leq x$ **and** b : $x < 1$
shows $\ln (1 - x) \leq -x$
proof –
have $(1 - x) * (1 + x + x^2) = 1 - x^3$
by (*simp add: algebra-simps power2-eq-square power3-eq-cube*)
also have $\dots \leq 1$
by (*auto simp: a*)
finally have $(1 - x) * (1 + x + x^2) \leq 1$.
moreover have c : $0 < 1 + x + x^2$
by (*simp add: add-pos-nonneg a*)
ultimately have $1 - x \leq 1 / (1 + x + x^2)$
by (*elim mult-imp-le-div-pos*)
also have $\dots \leq 1 / \exp x$
by (*metis a abs-one b exp-bound exp-gt-zero frac-le less-eq-real-def real-sqrt-abs real-sqrt-pow2-iff real-sqrt-power*)
also have $\dots = \exp (-x)$
by (*auto simp: exp-minus divide-inverse*)
finally have $1 - x \leq \exp (-x)$.
also have $1 - x = \exp (\ln (1 - x))$
by (*metis b diff-0 exp-ln-iff less-iff-diff-less-0 minus-diff-eq*)
finally have $\exp (\ln (1 - x)) \leq \exp (-x)$.
then show *?thesis*
by (*auto simp only: exp-le-cancel-iff*)
qed

lemma *exp-ge-add-one-self [simp]*: $1 + x \leq \exp x$
for $x :: \text{real}$
proof (*cases* $0 \leq x \vee x \leq -1$)

```

case True
then show ?thesis
  by (meson exp-ge-add-one-self-aux exp-ge-zero order.trans real-add-le-0-iff)
next
case False
then have ln1: ln (1 + x) ≤ x
  using ln-one-minus-pos-upper-bound [of -x] by simp
have 1 + x = exp (ln (1 + x))
  using False by auto
also have ... ≤ exp x
  by (simp add: ln1)
finally show ?thesis .
qed

```

```

lemma ln-one-plus-pos-lower-bound:
  fixes x :: real
  assumes a: 0 ≤ x and b: x ≤ 1
  shows x - x2 ≤ ln (1 + x)
proof -
  have exp (x - x2) = exp x / exp (x2)
    by (rule exp-diff)
  also have ... ≤ (1 + x + x2) / exp (x2)
    by (metis a b divide-right-mono exp-bound exp-ge-zero)
  also have ... ≤ (1 + x + x2) / (1 + x2)
    by (simp add: a divide-left-mono add-pos-nonneg)
  also from a have ... ≤ 1 + x
    by (simp add: field-simps add-strict-increasing zero-le-mult-iff)
  finally have exp (x - x2) ≤ 1 + x .
  also have ... = exp (ln (1 + x))
  proof -
    from a have 0 < 1 + x by auto
    then show ?thesis
      by (auto simp only: exp-ln-iff [THEN sym])
  qed
  finally have exp (x - x2) ≤ exp (ln (1 + x)) .
  then show ?thesis
    by (metis exp-le-cancel-iff)
qed

```

```

lemma ln-one-minus-pos-lower-bound:
  fixes x :: real
  assumes a: 0 ≤ x and b: x ≤ 1/2
  shows - x - 2 * x2 ≤ ln (1 - x)
proof -
  from b have c: x < 1 by auto
  then have ln (1 - x) = - ln (1 + x / (1 - x))
    by (auto simp: ln-inverse [symmetric] field-simps intro: arg-cong [where f=ln])
  also have - (x / (1 - x)) ≤ ...
  proof -

```

```

    have  $\ln (1 + x / (1 - x)) \leq x / (1 - x)$ 
      using  $a\ c$  by (intro ln-add-one-self-le-self) auto
    then show ?thesis
      by auto
  qed
  also have  $-(x / (1 - x)) = -x / (1 - x)$ 
    by auto
  finally have  $d: -x / (1 - x) \leq \ln (1 - x)$  .
  have  $0 < 1 - x$  using  $a\ b$  by simp
  then have  $e: -x - 2 * x^2 \leq -x / (1 - x)$ 
    using mult-right-le-one-le[of  $x * x\ 2 * x$ ]  $a\ b$ 
    by (simp add: field-simps power2-eq-square)
  from  $e\ d$  show  $-x - 2 * x^2 \leq \ln (1 - x)$ 
    by (rule order-trans)
  qed

```

lemma *ln-add-one-self-le-self2*:

```

  fixes  $x :: real$ 
  shows  $-1 < x \implies \ln (1 + x) \leq x$ 
    by (metis diff-gt-0-iff-gt diff-minus-eq-add exp-ge-add-one-self exp-le-cancel-iff
    exp-ln minus-less-iff)

```

lemma *abs-ln-one-plus-x-minus-x-bound-nonneg*:

```

  fixes  $x :: real$ 
  assumes  $x: 0 \leq x$  and  $x1: x \leq 1$ 
  shows  $|\ln (1 + x) - x| \leq x^2$ 
  proof -
    from  $x$  have  $\ln (1 + x) \leq x$ 
      by (rule ln-add-one-self-le-self)
    then have  $\ln (1 + x) - x \leq 0$ 
      by simp
    then have  $|\ln(1 + x) - x| = -(\ln(1 + x) - x)$ 
      by (rule abs-of-nonpos)
    also have  $\dots = x - \ln (1 + x)$ 
      by simp
    also have  $\dots \leq x^2$ 
    proof -
      from  $x\ x1$  have  $x - x^2 \leq \ln (1 + x)$ 
        by (intro ln-one-plus-pos-lower-bound)
      then show ?thesis
        by simp
    qed
  qed
  finally show ?thesis .
  qed

```

lemma *abs-ln-one-plus-x-minus-x-bound-nonpos*:

```

  fixes  $x :: real$ 
  assumes  $a: -(1/2) \leq x$  and  $b: x \leq 0$ 
  shows  $|\ln (1 + x) - x| \leq 2 * x^2$ 

```

proof –

have *: $-(-x) - 2 * (-x)^2 \leq \ln(1 - (-x))$

by (*metis a b diff-zero ln-one-minus-pos-lower-bound minus-diff-eq neg-le-iff-le*)

have $|\ln(1 + x) - x| = x - \ln(1 - (-x))$

using *a ln-add-one-self-le-self2 [of x]* **by** (*simp add: abs-if*)

also have $\dots \leq 2 * x^2$

using * **by** (*simp add: algebra-simps*)

finally show *?thesis* .

qed

lemma *abs-ln-one-plus-x-minus-x-bound*:

fixes *x :: real*

assumes $|x| \leq 1/2$

shows $|\ln(1 + x) - x| \leq 2 * x^2$

proof (*cases 0 ≤ x*)

case *True*

then show *?thesis*

using *abs-ln-one-plus-x-minus-x-bound-nonneg assms* **by** *fastforce*

next

case *False*

then show *?thesis*

using *abs-ln-one-plus-x-minus-x-bound-nonpos assms* **by** *auto*

qed

lemma *ln-x-over-x-mono*:

fixes *x :: real*

assumes *x: exp 1 ≤ x x ≤ y*

shows $\ln y / y \leq \ln x / x$

proof –

note *x*

moreover have $0 < \exp(1::real)$ **by** *simp*

ultimately have *a: 0 < x* **and** *b: 0 < y*

by (*fast intro: less-le-trans order-trans*)**+**

have $x * \ln y - x * \ln x = x * (\ln y - \ln x)$

by (*simp add: algebra-simps*)

also have $\dots = x * \ln(y / x)$

by (*simp only: ln-div a b*)

also have $y / x = (x + (y - x)) / x$

by *simp*

also have $\dots = 1 + (y - x) / x$

using *x a* **by** (*simp add: field-simps*)

also have $x * \ln(1 + (y - x) / x) \leq x * ((y - x) / x)$

using *x a*

by (*intro mult-left-mono ln-add-one-self-le-self*) *simp-all*

also have $\dots = y - x$

using *a* **by** *simp*

also have $\dots = (y - x) * \ln(\exp 1)$ **by** *simp*

also have $\dots \leq (y - x) * \ln x$

using a *exp-total of-nat-1* $x(1)$ by (*fastforce intro: mult-left-mono*)
 also have $\dots = y * \ln x - x * \ln x$
 by (*rule left-diff-distrib*)
 finally have $x * \ln y \leq y * \ln x$
 by *arith*
 then have $\ln y \leq (y * \ln x) / x$
 using a by (*simp add: field-simps*)
 also have $\dots = y * (\ln x / x)$ by *simp*
 finally show *?thesis*
 using b by (*simp add: field-simps*)
 qed

lemma *ln-le-minus-one*: $0 < x \implies \ln x \leq x - 1$
 for $x :: \text{real}$
 using *exp-ge-add-one-self*[*of ln x*] by *simp*

corollary *ln-diff-le*: $0 < x \implies 0 < y \implies \ln x - \ln y \leq (x - y) / y$
 for $x :: \text{real}$
 by (*simp add: ln-div [symmetric] diff-divide-distrib ln-le-minus-one*)

lemma *ln-eq-minus-one*:
 fixes $x :: \text{real}$
 assumes $0 < x$ $\ln x = x - 1$
 shows $x = 1$

proof –
 let $?l = \lambda y. \ln y - y + 1$
 have $D: \bigwedge x :: \text{real}. 0 < x \implies \text{DERIV } ?l x :> (1 / x - 1)$
 by (*auto intro!: derivative-eq-intros*)

show *?thesis*

proof (*cases rule: linorder-cases*)

assume $x < 1$

from *dense*[*OF* $\langle x < 1 \rangle$] obtain a where $x < a < 1$ by *blast*

from $\langle x < a \rangle$ have $?l x < ?l a$

proof (*rule DERIV-pos-imp-increasing*)

fix y

assume $x \leq y \leq a$

with $\langle 0 < x \rangle \langle a < 1 \rangle$ have $0 < 1 / y - 1$ $0 < y$

by (*auto simp: field-simps*)

with D show $\exists z. \text{DERIV } ?l y :> z \wedge 0 < z$ by *blast*

qed

also have $\dots \leq 0$

using *ln-le-minus-one* $\langle 0 < x \rangle \langle x < a \rangle$ by (*auto simp: field-simps*)

finally show $x = 1$ using *assms* by *auto*

next

assume $1 < x$

from *dense*[*OF this*] obtain a where $1 < a < x$ by *blast*

from $\langle a < x \rangle$ have $?l x < ?l a$

proof (*rule DERIV-neg-imp-decreasing*)

```

fix  $y$ 
assume  $a \leq y \leq x$ 
with  $\langle 1 < a \rangle$  have  $1 / y - 1 < 0 \ 0 < y$ 
  by (auto simp: field-simps)
with  $D$  show  $\exists z. \text{DERIV } ?l \ y :> z \wedge z < 0$ 
  by blast
qed
also have  $\dots \leq 0$ 
  using ln-le-minus-one  $\langle 1 < a \rangle$  by (auto simp: field-simps)
finally show  $x = 1$  using assms by auto
next
  assume  $x = 1$ 
  then show ?thesis by simp
qed
qed

lemma ln-x-over-x-tendsto-0:  $((\lambda x::\text{real}. \ln x / x) \longrightarrow 0)$  at-top
proof (rule lhospital-at-top-at-top[where  $f' = \text{inverse}$  and  $g' = \lambda \cdot 1$ ])
  from eventually-gt-at-top[of  $0::\text{real}$ ]
  show  $\forall_F x$  in at-top.  $(\ln \text{ has-real-derivative } \text{inverse } x)$  (at x)
  by eventually-elim (auto intro!: derivative-eq-intros simp: field-simps)
qed (use tendsto-inverse-0 in
   $\langle \text{auto simp: filterlim-ident dest!: tendsto-mono}[OF \text{at-top-le-at-infinity}] \rangle$ )

lemma exp-ge-one-plus-x-over-n-power-n:
  assumes  $x \geq - \text{real } n \ n > 0$ 
  shows  $(1 + x / \text{of-nat } n) ^ n \leq \exp x$ 
proof (cases  $x = - \text{of-nat } n$ )
  case False
    from assms False have  $(1 + x / \text{of-nat } n) ^ n = \exp (\text{of-nat } n * \ln (1 + x / \text{of-nat } n))$ 
    by (subst exp-of-nat-mult, subst exp-ln) (simp-all add: field-simps)
    also from assms False have  $\ln (1 + x / \text{real } n) \leq x / \text{real } n$ 
    by (intro ln-add-one-self-le-self2) (simp-all add: field-simps)
    with assms have  $\exp (\text{of-nat } n * \ln (1 + x / \text{of-nat } n)) \leq \exp x$ 
    by (simp add: field-simps)
    finally show ?thesis .
  next
    case True
    then show ?thesis by (simp add: zero-power)
qed

lemma exp-ge-one-minus-x-over-n-power-n:
  assumes  $x \leq \text{real } n \ n > 0$ 
  shows  $(1 - x / \text{of-nat } n) ^ n \leq \exp (-x)$ 
  using exp-ge-one-plus-x-over-n-power-n[of  $n \ -x$ ] assms by simp

lemma exp-at-bot:  $(\exp \longrightarrow (0::\text{real}))$  at-bot
  unfolding tendsto-Zfun-iff

```

proof (rule ZfunI, simp add: eventually-at-bot-dense)

fix $r :: \text{real}$

assume $0 < r$

have $\exp x < r$ **if** $x < \ln r$ **for** x

by (metis $\langle 0 < r \rangle$ exp-less-mono exp-ln that)

then show $\exists k. \forall n < k. \exp n < r$ **by** auto

qed

lemma exp-at-top: LIM x at-top. $\exp x :: \text{real} :> \text{at-top}$

by (rule filterlim-at-top-at-top[**where** $Q = \lambda x. \text{True}$ **and** $P = \lambda x. 0 < x$ **and** $g = \ln$])
(auto intro: eventually-gt-at-top)

lemma lim-exp-minus-1: $((\lambda z :: 'a. (\exp(z) - 1) / z) \longrightarrow 1) (\text{at } 0)$

for $x :: 'a :: \{\text{real-normed-field, banach}\}$

proof –

have $((\lambda z :: 'a. \exp(z) - 1)$ has-field-derivative 1) (at 0)

by (intro derivative-eq-intros | simp)+

then show ?thesis

by (simp add: Deriv.has-field-derivative-iff)

qed

lemma ln-at-0: LIM x at-right 0. $\ln (x :: \text{real}) :> \text{at-bot}$

by (rule filterlim-at-bot-at-right[**where** $Q = \lambda x. 0 < x$ **and** $P = \lambda x. \text{True}$ **and** $g = \exp$])

(auto simp: eventually-at-filter)

lemma ln-at-top: LIM x at-top. $\ln (x :: \text{real}) :> \text{at-top}$

by (rule filterlim-at-top-at-top[**where** $Q = \lambda x. 0 < x$ **and** $P = \lambda x. \text{True}$ **and** $g = \exp$])

(auto intro: eventually-gt-at-top)

lemma filtermap-ln-at-top: filtermap $(\ln :: \text{real} \Rightarrow \text{real})$ at-top = at-top

by (intro filtermap-fun-inverse[of exp] exp-at-top ln-at-top) auto

lemma filtermap-exp-at-top: filtermap $(\exp :: \text{real} \Rightarrow \text{real})$ at-top = at-top

by (intro filtermap-fun-inverse[of ln] exp-at-top ln-at-top)

(auto simp: eventually-at-top-dense)

lemma filtermap-ln-at-right: filtermap \ln (at-right $(0 :: \text{real})$) = at-bot

by (auto intro!: filtermap-fun-inverse[**where** $g = \lambda x. \exp x$] ln-at-0

simp: filterlim-at exp-at-bot)

lemma tendsto-power-div-exp-0: $((\lambda x. x \wedge k / \exp x) \longrightarrow (0 :: \text{real}))$ at-top

proof (induct k)

case 0

show $((\lambda x. x \wedge 0 / \exp x) \longrightarrow (0 :: \text{real}))$ at-top

by (simp add: inverse-eq-divide[symmetric])

(metis filterlim-compose[OF tendsto-inverse-0] exp-at-top filterlim-mono
at-top-le-at-infinity order-refl)


```

next
  case (Suc k)
  show ?case
  proof (rule lhospital-at-top-at-top)
    show eventually ( $\lambda x. \text{DERIV } (\lambda x. x \wedge \text{Suc } k) x :> (\text{real } (\text{Suc } k) * x \wedge k)$ ) at-top
      by eventually-elim (intro derivative-eq-intros, auto)
    show eventually ( $\lambda x. \text{DERIV } \exp x :> \exp x$ ) at-top
      by eventually-elim auto
    show eventually ( $\lambda x. \exp x \neq 0$ ) at-top
      by auto
    from tendsto-mult[OF tendsto-const Suc, of real (Suc k)]
    show (( $\lambda x. \text{real } (\text{Suc } k) * x \wedge k / \exp x \longrightarrow 0$ ) at-top)
      by simp
  qed (rule exp-at-top)
qed

```

113.8.1 A couple of simple bounds

```

lemma exp-plus-inverse-exp:
  fixes x::real
  shows  $2 \leq \exp x + \text{inverse } (\exp x)$ 
  proof -
    have  $2 \leq \exp x + \exp (-x)$ 
      using exp-ge-add-one-self [of x] exp-ge-add-one-self [of -x]
      by linarith
    then show ?thesis
      by (simp add: exp-minus)
  qed

```

```

lemma real-le-x-sinh:
  fixes x::real
  assumes  $0 \leq x$ 
  shows  $x \leq (\exp x - \text{inverse}(\exp x)) / 2$ 
  proof -
    have *:  $\exp a - \text{inverse}(\exp a) - 2*a \leq \exp b - \text{inverse}(\exp b) - 2*b$  if  $a \leq b$ 
    for a b::real
      using exp-plus-inverse-exp
      by (fastforce intro: derivative-eq-intros DERIV-nonneg-imp-nondecreasing [OF that])
    show ?thesis
      using*[OF assms] by simp
  qed

```

```

lemma real-le-abs-sinh:
  fixes x::real
  shows  $\text{abs } x \leq \text{abs}((\exp x - \text{inverse}(\exp x)) / 2)$ 
  proof (cases  $0 \leq x$ )
    case True
    show ?thesis

```

```

  using real-le-x-sinh [OF True] True by (simp add: abs-if)
next
case False
have  $-x \leq (\exp(-x) - \text{inverse}(\exp(-x))) / 2$ 
  by (meson False linear neg-le-0-iff-le real-le-x-sinh)
also have  $\dots \leq |(\exp x - \text{inverse}(\exp x)) / 2|$ 
  by (metis (no-types, opaque-lifting) abs-divide abs-le-iff abs-minus-cancel
    add.inverse-inverse exp-minus minus-diff-eq order-refl)
finally show ?thesis
  using False by linarith
qed

```

113.9 The general logarithm

definition *log* :: *real* \Rightarrow *real* \Rightarrow *real*

— logarithm of *x* to base *a*

where $\text{log } a \ x = \ln x / \ln a$

lemma *tendsto-log* [*tendsto-intros*]:

$(f \longrightarrow a) \ F \Longrightarrow (g \longrightarrow b) \ F \Longrightarrow 0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < b \Longrightarrow$

$((\lambda x. \text{log } (f \ x) \ (g \ x)) \longrightarrow \text{log } a \ b) \ F$

unfolding *log-def* by (*intro tendsto-intros*) *auto*

lemma *continuous-log*:

assumes *continuous F f*

and *continuous F g*

and $0 < f \ (\text{Lim } F \ (\lambda x. \ x))$

and $f \ (\text{Lim } F \ (\lambda x. \ x)) \neq 1$

and $0 < g \ (\text{Lim } F \ (\lambda x. \ x))$

shows *continuous F* $(\lambda x. \ \text{log } (f \ x) \ (g \ x))$

using *assms* **unfolding** *continuous-def* by (*rule tendsto-log*)

lemma *continuous-at-within-log*[*continuous-intros*]:

assumes *continuous (at a within s) f*

and *continuous (at a within s) g*

and $0 < f \ a$

and $f \ a \neq 1$

and $0 < g \ a$

shows *continuous (at a within s)* $(\lambda x. \ \text{log } (f \ x) \ (g \ x))$

using *assms* **unfolding** *continuous-within* by (*rule tendsto-log*)

lemma *isCont-log*[*continuous-intros*, *simp*]:

assumes *isCont f a isCont g a* $0 < f \ a \ f \ a \neq 1 \ 0 < g \ a$

shows *isCont* $(\lambda x. \ \text{log } (f \ x) \ (g \ x)) \ a$

using *assms* **unfolding** *continuous-at* by (*rule tendsto-log*)

lemma *continuous-on-log*[*continuous-intros*]:

assumes *continuous-on s f continuous-on s g*

and $\forall x \in s. \ 0 < f \ x \ \forall x \in s. \ f \ x \neq 1 \ \forall x \in s. \ 0 < g \ x$

shows *continuous-on s* ($\lambda x. \log (f x) (g x)$)
using *assms unfolding continuous-on-def* **by** (*fast intro: tendsto-log*)

lemma *powr-one-eq-one* [*simp*]: $1 \text{ powr } a = 1$
by (*simp add: powr-def*)

lemma *powr-zero-eq-one* [*simp*]: $x \text{ powr } 0 = (\text{if } x = 0 \text{ then } 0 \text{ else } 1)$
by (*simp add: powr-def*)

lemma *powr-one-gt-zero-iff* [*simp*]: $x \text{ powr } 1 = x \longleftrightarrow 0 \leq x$
for $x :: \text{real}$
by (*auto simp: powr-def*)
declare *powr-one-gt-zero-iff* [*THEN iffD2, simp*]

lemma *powr-diff*:
fixes $w :: 'a :: \{\text{ln, real-normed-field}\}$ **shows** $w \text{ powr } (z1 - z2) = w \text{ powr } z1 / w \text{ powr } z2$
by (*simp add: powr-def algebra-simps exp-diff*)

lemma *powr-mult*: $0 \leq x \implies 0 \leq y \implies (x * y) \text{ powr } a = (x \text{ powr } a) * (y \text{ powr } a)$
for $a x y :: \text{real}$
by (*simp add: powr-def exp-add [symmetric] ln-mult distrib-left*)

lemma *powr-ge-pzero* [*simp*]: $0 \leq x \text{ powr } y$
for $x y :: \text{real}$
by (*simp add: powr-def*)

lemma *powr-non-neg*[*simp*]: $\neg a \text{ powr } x < 0$ **for** $a x :: \text{real}$
using *powr-ge-pzero*[*of a x*] **by** *arith*

lemma *inverse-powr*: $\bigwedge y :: \text{real}. 0 \leq y \implies \text{inverse } y \text{ powr } a = \text{inverse } (y \text{ powr } a)$
by (*simp add: exp-minus ln-inverse powr-def*)

lemma *powr-divide*: $\llbracket 0 \leq x; 0 \leq y \rrbracket \implies (x / y) \text{ powr } a = (x \text{ powr } a) / (y \text{ powr } a)$
for $a b x :: \text{real}$
by (*simp add: divide-inverse powr-mult inverse-powr*)

lemma *powr-add*: $x \text{ powr } (a + b) = (x \text{ powr } a) * (x \text{ powr } b)$
for $a b x :: 'a :: \{\text{ln, real-normed-field}\}$
by (*simp add: powr-def exp-add [symmetric] distrib-right*)

lemma *powr-mult-base*: $0 \leq x \implies x * x \text{ powr } y = x \text{ powr } (1 + y)$
for $x :: \text{real}$
by (*auto simp: powr-add*)

lemma *powr-powr*: $(x \text{ powr } a) \text{ powr } b = x \text{ powr } (a * b)$
for $a b x :: \text{real}$
by (*simp add: powr-def*)

lemma *powr-powr-swap*: $(x \text{ powr } a) \text{ powr } b = (x \text{ powr } b) \text{ powr } a$
for $a \ b \ x :: \text{real}$
by (*simp add: powr-powr mult.commute*)

lemma *powr-minus*: $x \text{ powr } (- a) = \text{inverse } (x \text{ powr } a)$
for $a \ x :: 'a::\{\text{ln,real-normed-field}\}$
by (*simp add: powr-def exp-minus [symmetric]*)

lemma *powr-minus-divide*: $x \text{ powr } (- a) = 1/(x \text{ powr } a)$
for $a \ x :: 'a::\{\text{ln,real-normed-field}\}$
by (*simp add: divide-inverse powr-minus*)

lemma *powr-sum*: $x \neq 0 \implies \text{finite } A \implies x \text{ powr } \text{sum } f \ A = (\prod_{y \in A} x \text{ powr } f \ y)$
by (*simp add: powr-def exp-sum sum-distrib-right*)

lemma *divide-powr-uminus*: $a / b \text{ powr } c = a * b \text{ powr } (- c)$
for $a \ b \ c :: \text{real}$
by (*simp add: powr-minus-divide*)

lemma *powr-less-mono*: $a < b \implies 1 < x \implies x \text{ powr } a < x \text{ powr } b$
for $a \ b \ x :: \text{real}$
by (*simp add: powr-def*)

lemma *powr-less-cancel*: $x \text{ powr } a < x \text{ powr } b \implies 1 < x \implies a < b$
for $a \ b \ x :: \text{real}$
by (*simp add: powr-def*)

lemma *powr-less-cancel-iff* [*simp*]: $1 < x \implies x \text{ powr } a < x \text{ powr } b \iff a < b$
for $a \ b \ x :: \text{real}$
by (*blast intro: powr-less-cancel powr-less-mono*)

lemma *powr-le-cancel-iff* [*simp*]: $1 < x \implies x \text{ powr } a \leq x \text{ powr } b \iff a \leq b$
for $a \ b \ x :: \text{real}$
by (*simp add: linorder-not-less [symmetric]*)

lemma *powr-realpow*: $0 < x \implies x \text{ powr } (\text{real } n) = x^{\wedge} n$
by (*induction n*) (*simp-all add: ac-simps powr-add*)

lemma *powr-realpow'*: $(z :: \text{real}) \geq 0 \implies n \neq 0 \implies z \text{ powr } \text{of-nat } n = z^{\wedge} n$
by (*cases z = 0*) (*auto simp: powr-realpow*)

lemma *powr-real-of-int'*:
assumes $x \geq 0 \ x \neq 0 \vee n > 0$
shows $x \text{ powr } \text{real-of-int } n = \text{power-int } x \ n$
by (*metis assms exp-ln-iff exp-power-int nless-le power-int-eq-0-iff powr-def*)

lemma *log-ln*: $\ln x = \log (\exp(1)) \ x$
by (*simp add: log-def*)

lemma *DERIV-log*:

assumes $x > 0$

shows *DERIV* $(\lambda y. \log b y) x :> 1 / (\ln b * x)$

proof –

define *lb* **where** $lb = 1 / \ln b$

moreover have *DERIV* $(\lambda y. lb * \ln y) x :> lb / x$

using $\langle x > 0 \rangle$ **by** (*auto intro!*: *derivative-eq-intros*)

ultimately show *?thesis*

by (*simp add*: *log-def*)

qed

lemmas *DERIV-log*[*THEN DERIV-chain2*, *derivative-intros*]

and *DERIV-log*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemma *powr-log-cancel* [*simp*]: $0 < a \implies a \neq 1 \implies 0 < x \implies a \text{ powr } (\log a x) = x$

by (*simp add*: *powr-def log-def*)

lemma *log-powr-cancel* [*simp*]: $0 < a \implies a \neq 1 \implies \log a (a \text{ powr } y) = y$

by (*simp add*: *log-def powr-def*)

lemma *log-mult*:

$0 < a \implies a \neq 1 \implies 0 < x \implies 0 < y \implies$

$\log a (x * y) = \log a x + \log a y$

by (*simp add*: *log-def ln-mult divide-inverse distrib-right*)

lemma *log-eq-div-ln-mult-log*:

$0 < a \implies a \neq 1 \implies 0 < b \implies b \neq 1 \implies 0 < x \implies$

$\log a x = (\ln b / \ln a) * \log b x$

by (*simp add*: *log-def divide-inverse*)

Base 10 logarithms

lemma *log-base-10-eq1*: $0 < x \implies \log 10 x = (\ln (\exp 1) / \ln 10) * \ln x$

by (*simp add*: *log-def*)

lemma *log-base-10-eq2*: $0 < x \implies \log 10 x = (\log 10 (\exp 1)) * \ln x$

by (*simp add*: *log-def*)

lemma *log-one* [*simp*]: $\log a 1 = 0$

by (*simp add*: *log-def*)

lemma *log-eq-one* [*simp*]: $0 < a \implies a \neq 1 \implies \log a a = 1$

by (*simp add*: *log-def*)

lemma *log-inverse*: $0 < a \implies a \neq 1 \implies 0 < x \implies \log a (\text{inverse } x) = - \log a x$

using *ln-inverse log-def* **by** *auto*

lemma *log-divide*: $0 < a \implies a \neq 1 \implies 0 < x \implies 0 < y \implies \log a (x/y) = \log a x - \log a y$

by (*simp add: log-mult divide-inverse log-inverse*)

lemma *powr-gt-zero* [*simp*]: $0 < x \text{ powr } a \longleftrightarrow x \neq 0$

for $a x :: \text{real}$

by (*simp add: powr-def*)

lemma *powr-nonneg-iff* [*simp*]: $a \text{ powr } x \leq 0 \longleftrightarrow a = 0$

for $a x :: \text{real}$

by (*meson not-less powr-gt-zero*)

lemma *log-add-eq-powr*: $0 < b \implies b \neq 1 \implies 0 < x \implies \log b x + y = \log b (x * b \text{ powr } y)$

and *add-log-eq-powr*: $0 < b \implies b \neq 1 \implies 0 < x \implies y + \log b x = \log b (b \text{ powr } y * x)$

and *log-minus-eq-powr*: $0 < b \implies b \neq 1 \implies 0 < x \implies \log b x - y = \log b (x * b \text{ powr } -y)$

and *minus-log-eq-powr*: $0 < b \implies b \neq 1 \implies 0 < x \implies y - \log b x = \log b (b \text{ powr } y / x)$

by (*simp-all add: log-mult log-divide*)

lemma *log-less-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies 0 < y \implies \log a x < \log a y \longleftrightarrow x < y$

using *powr-less-cancel-iff* [*of a*] *powr-log-cancel* [*of a x*] *powr-log-cancel* [*of a y*]

by (*metis less-eq-real-def less-trans not-le zero-less-one*)

lemma *log-inj*:

assumes $1 < b$

shows *inj-on* ($\log b$) $\{0 < ..\}$

proof (*rule inj-onI, simp*)

fix $x y$

assume *pos*: $0 < x$ $0 < y$ **and** $*$: $\log b x = \log b y$

show $x = y$

proof (*cases rule: linorder-cases*)

assume $x = y$

then show *?thesis* **by** *simp*

next

assume $x < y$

then have $\log b x < \log b y$

using *log-less-cancel-iff* [*OF* $\langle 1 < b \rangle$] *pos* **by** *simp*

then show *?thesis* **using** $*$ **by** *simp*

next

assume $y < x$

then have $\log b y < \log b x$

using *log-less-cancel-iff* [*OF* $\langle 1 < b \rangle$] *pos* **by** *simp*

then show *?thesis* **using** $*$ **by** *simp*

qed

qed

lemma *log-le-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies 0 < y \implies \log a x \leq \log a y \longleftrightarrow x \leq y$

by (*simp add: linorder-not-less* [*symmetric*])

lemma *zero-less-log-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies 0 < \log a x \longleftrightarrow 1 < x$

using *log-less-cancel-iff* [*of a 1 x*] **by** *simp*

lemma *zero-le-log-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies 0 \leq \log a x \longleftrightarrow 1 \leq x$

using *log-le-cancel-iff* [*of a 1 x*] **by** *simp*

lemma *log-less-zero-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies \log a x < 0 \longleftrightarrow x < 1$

using *log-less-cancel-iff* [*of a x 1*] **by** *simp*

lemma *log-le-zero-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies \log a x \leq 0 \longleftrightarrow x \leq 1$

using *log-le-cancel-iff* [*of a x 1*] **by** *simp*

lemma *one-less-log-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies 1 < \log a x \longleftrightarrow a < x$

using *log-less-cancel-iff* [*of a a x*] **by** *simp*

lemma *one-le-log-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies 1 \leq \log a x \longleftrightarrow a \leq x$

using *log-le-cancel-iff* [*of a a x*] **by** *simp*

lemma *log-less-one-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies \log a x < 1 \longleftrightarrow x < a$

using *log-less-cancel-iff* [*of a x a*] **by** *simp*

lemma *log-le-one-cancel-iff* [*simp*]: $1 < a \implies 0 < x \implies \log a x \leq 1 \longleftrightarrow x \leq a$

using *log-le-cancel-iff* [*of a x a*] **by** *simp*

lemma *le-log-iff*:

fixes $b x y :: \text{real}$

assumes $1 < b x > 0$

shows $y \leq \log b x \longleftrightarrow b \text{ powr } y \leq x$

using *assms*

by (*metis less-irrefl less-trans powr-le-cancel-iff powr-log-cancel zero-less-one*)

lemma *less-log-iff*:

assumes $1 < b x > 0$

shows $y < \log b x \longleftrightarrow b \text{ powr } y < x$

by (*metis assms dual-order.strict-trans less-irrefl powr-less-cancel-iff powr-log-cancel zero-less-one*)

lemma

assumes $1 < b x > 0$

shows *log-less-iff*: $\log b x < y \longleftrightarrow x < b \text{ powr } y$

and *log-le-iff*: $\log b x \leq y \longleftrightarrow x \leq b \text{ powr } y$

using *le-log-iff* [*OF assms, of y*] *less-log-iff* [*OF assms, of y*]

by *auto*

lemmas *powr-le-iff* = *le-log-iff*[*symmetric*]
and *powr-less-iff* = *less-log-iff*[*symmetric*]
and *less-powr-iff* = *log-less-iff*[*symmetric*]
and *le-powr-iff* = *log-le-iff*[*symmetric*]

lemma *le-log-of-power*:

assumes $b^n \leq m$ $1 < b$

shows $n \leq \log b m$

proof –

from *assms* **have** $0 < m$ **by** (*metis less-trans zero-less-power less-le-trans zero-less-one*)

thus *?thesis* **using** *assms* **by** (*simp add: le-log-iff powr-realpow*)

qed

lemma *le-log2-of-power*: $2^n \leq m \implies n \leq \log 2 m$ **for** $m n :: \text{nat}$

using *le-log-of-power*[*of 2*] **by** *simp*

lemma *log-of-power-le*: $\llbracket m \leq b^n; b > 1; m > 0 \rrbracket \implies \log b (\text{real } m) \leq n$

by (*simp add: log-le-iff powr-realpow*)

lemma *log2-of-power-le*: $\llbracket m \leq 2^n; m > 0 \rrbracket \implies \log 2 m \leq n$ **for** $m n :: \text{nat}$

using *log-of-power-le*[*of - 2*] **by** *simp*

lemma *log-of-power-less*: $\llbracket m < b^n; b > 1; m > 0 \rrbracket \implies \log b (\text{real } m) < n$

by (*simp add: log-less-iff powr-realpow*)

lemma *log2-of-power-less*: $\llbracket m < 2^n; m > 0 \rrbracket \implies \log 2 m < n$ **for** $m n :: \text{nat}$

using *log-of-power-less*[*of - 2*] **by** *simp*

lemma *less-log-of-power*:

assumes $b^n < m$ $1 < b$

shows $n < \log b m$

proof –

have $0 < m$ **by** (*metis assms less-trans zero-less-power zero-less-one*)

thus *?thesis* **using** *assms* **by** (*simp add: less-log-iff powr-realpow*)

qed

lemma *less-log2-of-power*: $2^n < m \implies n < \log 2 m$ **for** $m n :: \text{nat}$

using *less-log-of-power*[*of 2*] **by** *simp*

lemma *gr-one-powr*[*simp*]:

fixes $x y :: \text{real}$ **shows** $\llbracket x > 1; y > 0 \rrbracket \implies 1 < x^y$

by(*simp add: less-powr-iff*)

lemma *log-pow-cancel* [*simp*]:

$a > 0 \implies a \neq 1 \implies \log a (a^b) = b$

by (*simp add: ln-realpow log-def*)

lemma *floor-log-eq-powr-iff*: $x > 0 \implies b > 1 \implies \lfloor \log b x \rfloor = k \iff b \text{ powr } k \leq x \wedge x < b \text{ powr } (k + 1)$

by (*auto simp: floor-eq-iff powr-le-iff less-powr-iff*)

lemma *floor-log-nat-eq-powr-iff*:

fixes $b n k :: \text{nat}$

shows $\llbracket b \geq 2; k > 0 \rrbracket \implies \text{floor } (\log b (\text{real } k)) = n \iff b^{\wedge}n \leq k \wedge k < b^{\wedge}(n+1)$

by (*auto simp: floor-log-eq-powr-iff powr-add powr-realpow*

of-nat-power[symmetric] of-nat-mult[symmetric] ac-simps

simp del: of-nat-power of-nat-mult)

lemma *floor-log-nat-eq-if*:

fixes $b n k :: \text{nat}$

assumes $b^{\wedge}n \leq k < b^{\wedge}(n+1) \ b \geq 2$

shows $\text{floor } (\log b (\text{real } k)) = n$

proof –

have $k \geq 1$

using *assms linorder-le-less-linear* **by** *force*

with *assms show ?thesis*

by(*simp add: floor-log-nat-eq-powr-iff*)

qed

lemma *ceiling-log-eq-powr-iff*:

$\llbracket x > 0; b > 1 \rrbracket \implies \lceil \log b x \rceil = \text{int } k + 1 \iff b \text{ powr } k < x \wedge x \leq b \text{ powr } (k + 1)$

by (*auto simp: ceiling-eq-iff powr-less-iff le-powr-iff*)

lemma *ceiling-log-nat-eq-powr-iff*:

fixes $b n k :: \text{nat}$

shows $\llbracket b \geq 2; k > 0 \rrbracket \implies \text{ceiling } (\log b (\text{real } k)) = \text{int } n + 1 \iff (b^{\wedge}n < k \wedge k \leq b^{\wedge}(n+1))$

using *ceiling-log-eq-powr-iff*

by (*auto simp: powr-add powr-realpow of-nat-power[symmetric] of-nat-mult[symmetric] ac-simps*

simp del: of-nat-power of-nat-mult)

lemma *ceiling-log-nat-eq-if*:

fixes $b n k :: \text{nat}$

assumes $b^{\wedge}n < k \leq b^{\wedge}(n+1) \ b \geq 2$

shows $\lceil \log (\text{real } b) (\text{real } k) \rceil = \text{int } n + 1$

using *assms ceiling-log-nat-eq-powr-iff* **by** *force*

lemma *floor-log2-div2*:

fixes $n :: \text{nat}$

assumes $n \geq 2$

shows $\lfloor \log 2 (\text{real } n) \rfloor = \lfloor \log 2 (n \text{ div } 2) \rfloor + 1$

proof *cases*

assume $n=2$ **thus** *?thesis* **by** *simp*

next
let $?m = n \text{ div } 2$
assume $n \neq 2$
hence $1 \leq ?m$ **using** *assms* **by** *arith*
then obtain i **where** $i: 2^i \leq ?m \text{ } ?m < 2^{i+1}$
using *ex-power-ivl1* [of $2 ?m$] **by** *auto*
have $2^{i+1} \leq 2 * ?m$ **using** $i(1)$ **by** *simp*
also have $2 * ?m \leq n$ **by** *arith*
finally have $*$: $2^{i+1} \leq \dots$.
have $n < 2^{i+1+1}$ **using** $i(2)$ **by** *simp*
from *floor-log-nat-eq-if* [*OF * this*] *floor-log-nat-eq-if* [*OF i*]
show *?thesis* **by** *simp*
qed

lemma *ceiling-log2-div2*:
assumes $n \geq 2$
shows $\text{ceiling}(\log 2 \text{ (real } n)) = \text{ceiling}(\log 2 ((n-1) \text{ div } 2 + 1)) + 1$
proof *cases*
assume $n=2$ **thus** *?thesis* **by** *simp*
next
let $?m = (n-1) \text{ div } 2 + 1$
assume $n \neq 2$
hence $2 \leq ?m$ **using** *assms* **by** *arith*
then obtain i **where** $i: 2^i < ?m \text{ } ?m \leq 2^{i+1}$
using *ex-power-ivl2* [of $2 ?m$] **by** *auto*
have $n \leq 2 * ?m$ **by** *arith*
also have $2 * ?m \leq 2^{i+1+1}$ **using** $i(2)$ **by** *simp*
finally have $*$: $n \leq \dots$.
have $2^{i+1} < n$ **using** $i(1)$ **by** (*auto simp: less-Suc-eq-0-disj*)
from *ceiling-log-nat-eq-if* [*OF this **] *ceiling-log-nat-eq-if* [*OF i*]
show *?thesis* **by** *simp*
qed

lemma *powr-real-of-int*:
 $x > 0 \implies x \text{ powr real-of-int } n = (\text{if } n \geq 0 \text{ then } x^{\text{nat } n} \text{ else inverse } (x^{\text{nat } (-n)}))$
using *powr-realpow* [of $x \text{ nat } n$] *powr-realpow* [of $x \text{ nat } (-n)$]
by (*auto simp: field-simps powr-minus*)

lemma *powr-numeral* [*simp*]: $0 \leq x \implies x \text{ powr (numeral } n \text{ :: real)} = x^{\text{(numeral } n)}$
by (*metis less-le power-zero-numeral powr-0 of-nat-numeral powr-realpow*)

lemma *powr-int*:
assumes $x > 0$
shows $x \text{ powr } i = (\text{if } i \geq 0 \text{ then } x^{\text{nat } i} \text{ else } 1 / x^{\text{nat } (-i)})$
by (*simp add: assms inverse-eq-divide powr-real-of-int*)

lemma *power-of-nat-log-ge*: $b > 1 \implies b^{\text{nat } \lceil \log b \rceil} \geq x$

by (smt (verit) less-log-of-power of-nat-ceiling)

lemma *power-of-nat-log-le*:

assumes $b > 1$ $x \geq 1$

shows $b^{\lceil \log b x \rceil} \leq x$

proof –

have $\lceil \log b x \rceil \geq 0$

using *assms* by *auto*

then show *?thesis*

by (smt (verit) *assms le-log-iff of-int-floor-le powr-int*)

qed

definition *powr-real* :: $real \Rightarrow real \Rightarrow real$

where [*code-abbrev*, *simp*]: *powr-real* = *Transcendental.powr*

lemma *compute-powr-real* [*code*]:

powr-real b i =

(if $b \leq 0$ then *Code.abort* (*STR "powr-real with nonpositive base"*) (λ -. *powr-real* b i))

else if $\lfloor i \rfloor = i$ then (if $0 \leq i$ then $b^{\lceil i \rceil}$ else $1 / b^{\lceil -i \rceil}$)

else *Code.abort* (*STR "powr-real with non-integer exponent"*) (λ -. *powr-real* b i))

for b i :: *real*

by (*auto simp: powr-int*)

lemma *powr-one*: $0 \leq x \Longrightarrow x \text{ powr } 1 = x$

for x :: *real*

using *powr-realpow* [*of x 1*] by *simp*

lemma *powr-neg-one*: $0 < x \Longrightarrow x \text{ powr } - 1 = 1 / x$

for x :: *real*

using *powr-int* [*of x - 1*] by *simp*

lemma *powr-neg-numeral*: $0 < x \Longrightarrow x \text{ powr } - \text{numeral } n = 1 / x^{\text{numeral } n}$

for x :: *real*

using *powr-int* [*of x - numeral n*] by *simp*

lemma *root-powr-inverse*: $0 < n \Longrightarrow 0 < x \Longrightarrow \text{root } n \ x = x \text{ powr } (1/n)$

by (*rule real-root-pos-unique*) (*auto simp: powr-realpow[symmetric] powr-powr*)

lemma *ln-powr*: $x \neq 0 \Longrightarrow \ln (x \text{ powr } y) = y * \ln x$

for x :: *real*

by (*simp add: powr-def*)

lemma *ln-root*: $n > 0 \Longrightarrow b > 0 \Longrightarrow \ln (\text{root } n \ b) = \ln b / n$

by (*simp add: root-powr-inverse ln-powr*)

lemma *ln-sqrt*: $0 < x \Longrightarrow \ln (\text{sqrt } x) = \ln x / 2$

by (*simp add: ln-powr ln-powr[symmetric] mult.commute*)

lemma *log-root*: $n > 0 \implies a > 0 \implies \log b (\text{root } n \ a) = \log b \ a / n$
by (*simp add: log-def ln-root*)

lemma *log-powr*: $x \neq 0 \implies \log b (x \ \text{powr } y) = y * \log b \ x$
by (*simp add: log-def ln-powr*)

lemma *log-nat-power*: $0 < x \implies \log b (x^{\wedge} n) = \text{real } n * \log b \ x$
by (*simp add: log-powr powr-realpow [symmetric]*)

lemma *log-of-power-eq*:
assumes $m = b^{\wedge} n \ b > 1$
shows $n = \log b (\text{real } m)$
proof –
have $n = \log b (b^{\wedge} n)$ **using** *assms(2)* **by** (*simp add: log-nat-power*)
also have $\dots = \log b \ m$ **using** *assms* **by** *simp*
finally show *?thesis* .
qed

lemma *log2-of-power-eq*: $m = 2^{\wedge} n \implies n = \log 2 \ m$ **for** $m \ n :: \text{nat}$
using *log-of-power-eq[of - 2]* **by** *simp*

lemma *log-base-change*: $0 < a \implies a \neq 1 \implies \log b \ x = \log a \ x / \log a \ b$
by (*simp add: log-def*)

lemma *log-base-pow*: $0 < a \implies \log (a^{\wedge} n) \ x = \log a \ x / n$
by (*simp add: log-def ln-realpow*)

lemma *log-base-powr*: $a \neq 0 \implies \log (a \ \text{powr } b) \ x = \log a \ x / b$
by (*simp add: log-def ln-powr*)

lemma *log-base-root*: $n > 0 \implies b > 0 \implies \log (\text{root } n \ b) \ x = n * (\log b \ x)$
by (*simp add: log-def ln-root*)

lemma *ln-bound*: $0 < x \implies \ln x \leq x$ **for** $x :: \text{real}$
using *ln-le-minus-one* **by** *force*

lemma *powr-mono*:
fixes $x :: \text{real}$
assumes $a \leq b$ **and** $1 \leq x$ **shows** $x \ \text{powr } a \leq x \ \text{powr } b$
using *assms less-eq-real-def* **by** *auto*

lemma *ge-one-powr-ge-zero*: $1 \leq x \implies 0 \leq a \implies 1 \leq x \ \text{powr } a$
for $x :: \text{real}$
using *powr-mono* **by** *fastforce*

lemma *powr-less-mono2*: $0 < a \implies 0 \leq x \implies x < y \implies x \ \text{powr } a < y \ \text{powr } a$
for $x :: \text{real}$

by (simp add: powr-def)

lemma *powr-less-mono2-neg*: $a < 0 \implies 0 < x \implies x < y \implies y \text{ powr } a < x \text{ powr } a$

for $x :: \text{real}$
by (simp add: powr-def)

lemma *powr-mono2*: $x \text{ powr } a \leq y \text{ powr } a$ if $0 \leq a$ $0 \leq x$ $x \leq y$

for $x :: \text{real}$
using *less-eq-real-def powr-less-mono2* that by auto

lemma *powr-le1*: $0 \leq a \implies 0 \leq x \implies x \leq 1 \implies x \text{ powr } a \leq 1$

for $x :: \text{real}$
using *powr-mono2* by fastforce

lemma *powr-mono2'*:

fixes $a x y :: \text{real}$
assumes $a \leq 0$ $x > 0$ $x \leq y$
shows $x \text{ powr } a \geq y \text{ powr } a$

proof –

from *assms* have $x \text{ powr } - a \leq y \text{ powr } - a$
by (intro *powr-mono2*) *simp-all*
with *assms* show ?thesis
by (auto simp: *powr-minus field-simps*)

qed

lemma *powr-mono-both*:

fixes $x :: \text{real}$
assumes $0 \leq a$ $a \leq b$ $1 \leq x$ $x \leq y$
shows $x \text{ powr } a \leq y \text{ powr } b$
by (meson *assms order.trans powr-mono powr-mono2 zero-le-one*)

lemma *powr-inj*: $0 < a \implies a \neq 1 \implies a \text{ powr } x = a \text{ powr } y \iff x = y$

for $x :: \text{real}$
unfolding *powr-def exp-inj-iff* by *simp*

lemma *powr-half-sqrt*: $0 \leq x \implies x \text{ powr } (1/2) = \text{sqrt } x$

by (simp add: *powr-def root-powr-inverse sqrt-def*)

lemma *square-powr-half* [*simp*]:

fixes $x :: \text{real}$ shows $x^2 \text{ powr } (1/2) = |x|$
by (simp add: *powr-half-sqrt*)

lemma *ln-powr-bound*: $1 \leq x \implies 0 < a \implies \ln x \leq (x \text{ powr } a) / a$

for $x :: \text{real}$
by (metis *exp-gt-zero linear ln-eq-zero-iff ln-exp ln-less-self ln-powr mult commute mult-imp-le-div-pos not-less powr-gt-zero*)

lemma *ln-powr-bound2*:

```

fixes  $x :: \text{real}$ 
assumes  $1 < x$  and  $0 < a$ 
shows  $(\ln x) \text{ powr } a \leq (a \text{ powr } a) * x$ 
proof –
  from assms have  $\ln x \leq (x \text{ powr } (1 / a)) / (1 / a)$ 
    by (metis less-eq-real-def ln-powr-bound zero-less-divide-1-iff)
  also have  $\dots = a * (x \text{ powr } (1 / a))$ 
    by simp
  finally have  $(\ln x) \text{ powr } a \leq (a * (x \text{ powr } (1 / a))) \text{ powr } a$ 
    by (metis assms less-imp-le ln-gt-zero powr-mono2)
  also have  $\dots = (a \text{ powr } a) * ((x \text{ powr } (1 / a)) \text{ powr } a)$ 
    using assms powr-mult by auto
  also have  $(x \text{ powr } (1 / a)) \text{ powr } a = x \text{ powr } ((1 / a) * a)$ 
    by (rule powr-powr)
  also have  $\dots = x$  using assms
    by auto
  finally show ?thesis .
qed

```

```

lemma tendsto-powr:
  fixes  $a b :: \text{real}$ 
  assumes  $f: (f \longrightarrow a) F$ 
    and  $g: (g \longrightarrow b) F$ 
    and  $a \neq 0$ 
  shows  $((\lambda x. f x \text{ powr } g x) \longrightarrow a \text{ powr } b) F$ 
  unfolding powr-def
proof (rule filterlim-If)
  from  $f$  show  $((\lambda x. 0) \longrightarrow (if\ a = 0\ then\ 0\ else\ exp\ (b * \ln\ a))) (inf\ F\ (principal\ \{x.\ f\ x = 0\}))$ 
    by simp (auto simp: filterlim-iff eventually-inf-principal elim: eventually-mono dest: t1-space-nhds)
  from  $f\ g\ a$  show  $((\lambda x. exp\ (g\ x * \ln\ (f\ x))) \longrightarrow (if\ a = 0\ then\ 0\ else\ exp\ (b * \ln\ a))) (inf\ F\ (principal\ \{x.\ f\ x \neq 0\}))$ 
    by (auto intro!: tendsto-intros intro: tendsto-mono inf-le1)
qed

```

```

lemma tendsto-powr'[tendsto-intros]:
  fixes  $a :: \text{real}$ 
  assumes  $f: (f \longrightarrow a) F$ 
    and  $g: (g \longrightarrow b) F$ 
    and  $a \neq 0 \vee (b > 0 \wedge eventually\ (\lambda x. f\ x \geq 0) F)$ 
  shows  $((\lambda x. f x \text{ powr } g x) \longrightarrow a \text{ powr } b) F$ 
proof –
  from  $a$  consider  $a \neq 0 \mid a = 0\ b > 0\ eventually\ (\lambda x. f\ x \geq 0) F$ 
    by auto
  then show ?thesis
proof cases
  case 1

```

```

  with f g show ?thesis by (rule tendsto-powr)
next
case 2
have (( $\lambda x. \text{if } f x = 0 \text{ then } 0 \text{ else } \exp (g x * \ln (f x))$ )  $\longrightarrow 0$ ) F
proof (intro filterlim-I)
  have filterlim f (principal {0<..}) (inf F (principal {z. f z  $\neq$  0}))
  using  $\langle \text{eventually } (\lambda x. f x \geq 0) F \rangle$ 
  by (auto simp: filterlim-iff eventually-inf-principal
      eventually-principal elim: eventually-mono)
  moreover have filterlim f (nhds a) (inf F (principal {z. f z  $\neq$  0}))
  by (rule tendsto-mono[OF - f]) simp-all
  ultimately have f: filterlim f (at-right 0) (inf F (principal {x. f x  $\neq$  0}))
  by (simp add: at-within-def filterlim-inf  $\langle a = 0 \rangle$ )
  have g: (g  $\longrightarrow b$ ) (inf F (principal {z. f z  $\neq$  0}))
  by (rule tendsto-mono[OF - g]) simp-all
  show (( $\lambda x. \exp (g x * \ln (f x))$ )  $\longrightarrow 0$ ) (inf F (principal {x. f x  $\neq$  0}))
  by (rule filterlim-compose[OF exp-at-bot] filterlim-tendsto-pos-mult-at-bot
      filterlim-compose[OF ln-at-0] f g  $\langle b > 0 \rangle$ )
qed simp-all
with  $\langle a = 0 \rangle$  show ?thesis
  by (simp add: powr-def)
qed
qed

```

```

lemma continuous-powr:
  assumes continuous F f
  and continuous F g
  and f (Lim F ( $\lambda x. x$ ))  $\neq 0$ 
  shows continuous F ( $\lambda x. (f x) \text{ powr } (g x :: \text{real})$ )
  using assms unfolding continuous-def by (rule tendsto-powr)

```

```

lemma continuous-at-within-powr[continuous-intros]:
  fixes f g :: -  $\Rightarrow$  real
  assumes continuous (at a within s) f
  and continuous (at a within s) g
  and f a  $\neq 0$ 
  shows continuous (at a within s) ( $\lambda x. (f x) \text{ powr } (g x)$ )
  using assms unfolding continuous-within by (rule tendsto-powr)

```

```

lemma isCont-powr[continuous-intros, simp]:
  fixes f g :: -  $\Rightarrow$  real
  assumes isCont f a isCont g a f a  $\neq 0$ 
  shows isCont ( $\lambda x. (f x) \text{ powr } g x$ ) a
  using assms unfolding continuous-at by (rule tendsto-powr)

```

```

lemma continuous-on-powr[continuous-intros]:
  fixes f g :: -  $\Rightarrow$  real
  assumes continuous-on s f continuous-on s g and  $\forall x \in s. f x \neq 0$ 
  shows continuous-on s ( $\lambda x. (f x) \text{ powr } (g x)$ )

```

using *assms* **unfolding** *continuous-on-def* **by** (*fast intro: tendsto-powr*)

lemma *tendsto-powr2*:

fixes $a :: \text{real}$
assumes $f: (f \longrightarrow a) F$
and $g: (g \longrightarrow b) F$
and $\forall_F x \text{ in } F. 0 \leq f x$
and $b: 0 < b$
shows $((\lambda x. f x \text{ powr } g x) \longrightarrow a \text{ powr } b) F$
using *tendsto-powr'*[*of f a F g b*] *assms* **by** *auto*

lemma *has-derivative-powr*[*derivative-intros*]:

assumes $g[\text{derivative-intros}]: (g \text{ has-derivative } g') \text{ (at } x \text{ within } X)$
and $f[\text{derivative-intros}]: (f \text{ has-derivative } f') \text{ (at } x \text{ within } X)$
assumes $pos: 0 < g x$ **and** $x \in X$
shows $((\lambda x. g x \text{ powr } f x :: \text{real}) \text{ has-derivative } (\lambda h. (g x \text{ powr } f x) * (f' h * \ln (g x) + g' h * f x / g x))) \text{ (at } x \text{ within } X)$
proof –
have $\forall_F x \text{ in at } x \text{ within } X. g x > 0$
by (*rule order-tendstoD*[*OF - pos*])
(*rule has-derivative-continuous*[*OF g, unfolded continuous-within*])
then obtain d **where** $d > 0$ **and** $pos': \bigwedge x'. x' \in X \implies \text{dist } x' x < d \implies 0 < g x'$
using *pos* **unfolding** *eventually-at* **by** *force*
have $((\lambda x. \text{exp } (f x * \ln (g x))) \text{ has-derivative } (\lambda h. (g x \text{ powr } f x) * (f' h * \ln (g x) + g' h * f x / g x))) \text{ (at } x \text{ within } X)$
using *pos*
by (*auto intro!*: *derivative-eq-intros simp: field-split-simps powr-def*)
then show *?thesis*
by (*rule has-derivative-transform-within*[*OF - <d > 0> <x ∈ X>*]) (*auto simp: powr-def dest: pos'*)
qed

lemma *DERIV-powr*:

fixes $r :: \text{real}$
assumes $g: \text{DERIV } g x := m$
and $pos: g x > 0$
and $f: \text{DERIV } f x := r$
shows $\text{DERIV } (\lambda x. g x \text{ powr } f x) x := (g x \text{ powr } f x) * (r * \ln (g x) + m * f x / g x)$
using *assms*
by (*auto intro!*: *derivative-eq-intros ext simp: has-field-derivative-def algebra-simps*)

lemma *DERIV-fun-powr*:

fixes $r :: \text{real}$
assumes $g: \text{DERIV } g x := m$
and $pos: g x > 0$
shows $\text{DERIV } (\lambda x. (g x) \text{ powr } r) x := r * (g x) \text{ powr } (r - \text{of-nat } 1) * m$
using *DERIV-powr*[*OF g pos DERIV-const, of r*] *pos*

by (simp add: powr-diff field-simps)

lemma *has-real-derivative-powr*:

assumes $z > 0$

shows $((\lambda z. z \text{ powr } r) \text{ has-real-derivative } r * z \text{ powr } (r - 1)) \text{ (at } z)$

proof (subst DERIV-cong-ev[OF refl - refl])

from *assms* have eventually $(\lambda z. z \neq 0)$ (nhds z)

by (intro t1-space-nhds) auto

then show eventually $(\lambda z. z \text{ powr } r = \exp (r * \ln z))$ (nhds z)

unfolding *powr-def* by eventually-elim simp

from *assms* show $((\lambda z. \exp (r * \ln z)) \text{ has-real-derivative } r * z \text{ powr } (r - 1))$
(at z)

by (auto intro!: derivative-eq-intros simp: powr-def field-simps exp-diff)

qed

declare *has-real-derivative-powr*[THEN DERIV-chain2, derivative-intros]

lemma *tendsto-zero-powrI*:

assumes $(f \longrightarrow (0::\text{real})) F (g \longrightarrow b) F \forall_F x \text{ in } F. 0 \leq f x < b$

shows $((\lambda x. f x \text{ powr } g x) \longrightarrow 0) F$

using *tendsto-powr2*[OF *assms*] by simp

lemma *continuous-on-powr'*:

fixes $f g :: - \Rightarrow \text{real}$

assumes *continuous-on* s *f* *continuous-on* s g

and $\forall x \in s. f x \geq 0 \wedge (f x = 0 \longrightarrow g x > 0)$

shows *continuous-on* s $(\lambda x. (f x) \text{ powr } (g x))$

unfolding *continuous-on-def*

proof

fix x

assume $x: x \in s$

from *assms* x show $((\lambda x. f x \text{ powr } g x) \longrightarrow f x \text{ powr } g x)$ (at x within s)

proof (cases $f x = 0$)

case *True*

from *assms*(3) have eventually $(\lambda x. f x \geq 0)$ (at x within s)

by (auto simp: at-within-def eventually-inf-principal)

with *True* x *assms* show ?thesis

by (auto intro!: tendsto-zero-powrI[of $f - g$ x] simp: *continuous-on-def*)

next

case *False*

with *assms* x show ?thesis

by (auto intro!: tendsto-powr' simp: *continuous-on-def*)

qed

qed

lemma *tendsto-neg-powr*:

assumes $s < 0$

and $f: \text{LIM } x F. f x :> \text{at-top}$

shows $((\lambda x. f x \text{ powr } s) \longrightarrow (0::\text{real})) F$

proof –

have $((\lambda x. \exp (s * \ln (f x))) \longrightarrow (0::real)) F$ (**is** ?X)
by (auto intro!: filterlim-compose[OF exp-at-bot] filterlim-compose[OF ln-at-top]
 filterlim-tendsto-neg-mult-at-bot assms)
also have ?X $\longleftrightarrow ((\lambda x. f x \text{ powr } s) \longrightarrow (0::real)) F$
using f filterlim-at-top-dense[of f F]
by (intro filterlim-cong[OF refl refl]) (auto simp: neq-iff powr-def elim: eventually-mono)
finally show ?thesis .
qed

lemma tendsto-exp-limit-at-right: $((\lambda y. (1 + x * y) \text{ powr } (1 / y)) \longrightarrow \exp x)$
 (at-right 0)

for $x :: real$
proof (cases $x = 0$)
case True
then show ?thesis **by** simp
next
case False
have $((\lambda y. \ln (1 + x * y)::real) \text{ has-real-derivative } 1 * x)$ (at 0)
by (auto intro!: derivative-eq-intros)
then have $((\lambda y. \ln (1 + x * y) / y) \longrightarrow x)$ (at 0)
by (auto simp: has-field-derivative-def field-has-derivative-at)
then have *: $((\lambda y. \exp (\ln (1 + x * y) / y)) \longrightarrow \exp x)$ (at 0)
by (rule tendsto-intros)
then show ?thesis
proof (rule filterlim-mono-eventually)
show eventually $(\lambda xa. \exp (\ln (1 + x * xa) / xa) = (1 + x * xa) \text{ powr } (1 / xa))$
 (at-right 0)
unfolding eventually-at-right[OF zero-less-one]
using False
by (intro exI[of - 1 / |x|]) (auto simp: field-simps powr-def abs-if add-nonneg-eq-0-iff)
qed (simp-all add: at-eq-sup-left-right)
qed

lemma tendsto-exp-limit-at-top: $((\lambda y. (1 + x / y) \text{ powr } y) \longrightarrow \exp x)$ at-top
for $x :: real$
by (simp add: filterlim-at-top-to-right inverse-eq-divide tendsto-exp-limit-at-right)

lemma tendsto-exp-limit-sequentially: $(\lambda n. (1 + x / n) \wedge n) \longrightarrow \exp x$
for $x :: real$

proof (rule filterlim-mono-eventually)
from reals-Archimedean2 [of |x|] **obtain** $n :: nat$ **where** *: $real\ n > |x|$..
then have eventually $(\lambda n :: nat. 0 < 1 + x / real\ n)$ at-top
by (intro eventually-sequentiallyI [of n]) (auto simp: field-split-simps)
then show eventually $(\lambda n. (1 + x / n) \text{ powr } n = (1 + x / n) \wedge n)$ at-top
by (rule eventually-mono) (erule powr-realpow)
show $(\lambda n. (1 + x / real\ n) \text{ powr } real\ n) \longrightarrow \exp x$
by (rule filterlim-compose [OF tendsto-exp-limit-at-top filterlim-real-sequentially])

qed auto

113.10 Sine and Cosine

definition *sin-coeff* :: nat \Rightarrow real

where *sin-coeff* = (λn . if even n then 0 else $(-1)^{(n - \text{Suc } 0) \text{ div } 2} / (\text{fact } n)$)

definition *cos-coeff* :: nat \Rightarrow real

where *cos-coeff* = (λn . if even n then $(-1)^{(n \text{ div } 2)} / (\text{fact } n)$ else 0)

definition *sin* :: 'a \Rightarrow 'a::{real-normed-algebra-1,banach}

where *sin* = (λx . $\sum n$. *sin-coeff* n *_R $x^{\wedge} n$)

definition *cos* :: 'a \Rightarrow 'a::{real-normed-algebra-1,banach}

where *cos* = (λx . $\sum n$. *cos-coeff* n *_R $x^{\wedge} n$)

lemma *sin-coeff-0* [*simp*]: *sin-coeff* 0 = 0

unfolding *sin-coeff-def* by *simp*

lemma *cos-coeff-0* [*simp*]: *cos-coeff* 0 = 1

unfolding *cos-coeff-def* by *simp*

lemma *sin-coeff-Suc*: *sin-coeff* (Suc n) = *cos-coeff* n / real (Suc n)

unfolding *cos-coeff-def* *sin-coeff-def*

by (*simp* del: *mult-Suc*)

lemma *cos-coeff-Suc*: *cos-coeff* (Suc n) = - *sin-coeff* n / real (Suc n)

unfolding *cos-coeff-def* *sin-coeff-def*

by (*simp* del: *mult-Suc*) (*auto elim: oddE*)

lemma *summable-norm-sin*: summable (λn . norm (*sin-coeff* n *_R $x^{\wedge} n$))

for x :: 'a::{real-normed-algebra-1,banach}

proof (*rule summable-comparison-test* [*OF* - *summable-norm-exp*])

show $\exists N$. $\forall n \geq N$. norm (norm (*sin-coeff* n *_R $x^{\wedge} n$)) \leq norm ($x^{\wedge} n$ /_R fact n)

unfolding *sin-coeff-def*

by (*auto simp: divide-inverse abs-mult power-abs* [*symmetric*] *zero-le-mult-iff*)

qed

lemma *summable-norm-cos*: summable (λn . norm (*cos-coeff* n *_R $x^{\wedge} n$))

for x :: 'a::{real-normed-algebra-1,banach}

proof (*rule summable-comparison-test* [*OF* - *summable-norm-exp*])

show $\exists N$. $\forall n \geq N$. norm (norm (*cos-coeff* n *_R $x^{\wedge} n$)) \leq norm ($x^{\wedge} n$ /_R fact n)

unfolding *cos-coeff-def*

by (*auto simp: divide-inverse abs-mult power-abs* [*symmetric*] *zero-le-mult-iff*)

qed

lemma *sin-converges*: $(\lambda n. \text{sin-coeff } n *_{\mathbb{R}} x^{\wedge} n) \text{ sums } \text{sin } x$
unfolding *sin-def*
by (*metis (full-types) summable-norm-cancel summable-norm-sin summable-sums*)

lemma *cos-converges*: $(\lambda n. \text{cos-coeff } n *_{\mathbb{R}} x^{\wedge} n) \text{ sums } \text{cos } x$
unfolding *cos-def*
by (*metis (full-types) summable-norm-cancel summable-norm-cos summable-sums*)

lemma *sin-of-real*: $\text{sin } (\text{of-real } x) = \text{of-real } (\text{sin } x)$
for $x :: \text{real}$

proof –

have $(\lambda n. \text{of-real } (\text{sin-coeff } n *_{\mathbb{R}} x^{\wedge} n)) = (\lambda n. \text{sin-coeff } n *_{\mathbb{R}} (\text{of-real } x)^{\wedge} n)$

proof

show $\text{of-real } (\text{sin-coeff } n *_{\mathbb{R}} x^{\wedge} n) = \text{sin-coeff } n *_{\mathbb{R}} \text{of-real } x^{\wedge} n$ **for** n

by (*simp add: scaleR-conv-of-real*)

qed

also have $\dots \text{ sums } (\text{sin } (\text{of-real } x))$

by (*rule sin-converges*)

finally have $(\lambda n. \text{of-real } (\text{sin-coeff } n *_{\mathbb{R}} x^{\wedge} n)) \text{ sums } (\text{sin } (\text{of-real } x))$.

then show *?thesis*

using *sums-unique2 sums-of-real [OF sin-converges]* **by** *blast*

qed

corollary *sin-in-Reals [simp]*: $z \in \mathbb{R} \implies \text{sin } z \in \mathbb{R}$
by (*metis Reals-cases Reals-of-real sin-of-real*)

lemma *cos-of-real*: $\text{cos } (\text{of-real } x) = \text{of-real } (\text{cos } x)$
for $x :: \text{real}$

proof –

have $(\lambda n. \text{of-real } (\text{cos-coeff } n *_{\mathbb{R}} x^{\wedge} n)) = (\lambda n. \text{cos-coeff } n *_{\mathbb{R}} (\text{of-real } x)^{\wedge} n)$

proof

show $\text{of-real } (\text{cos-coeff } n *_{\mathbb{R}} x^{\wedge} n) = \text{cos-coeff } n *_{\mathbb{R}} \text{of-real } x^{\wedge} n$ **for** n

by (*simp add: scaleR-conv-of-real*)

qed

also have $\dots \text{ sums } (\text{cos } (\text{of-real } x))$

by (*rule cos-converges*)

finally have $(\lambda n. \text{of-real } (\text{cos-coeff } n *_{\mathbb{R}} x^{\wedge} n)) \text{ sums } (\text{cos } (\text{of-real } x))$.

then show *?thesis*

using *sums-unique2 sums-of-real [OF cos-converges]*

by *blast*

qed

corollary *cos-in-Reals [simp]*: $z \in \mathbb{R} \implies \text{cos } z \in \mathbb{R}$
by (*metis Reals-cases Reals-of-real cos-of-real*)

lemma *diffs-sin-coeff*: $\text{diffs } \text{sin-coeff} = \text{cos-coeff}$
by (*simp add: diffs-def sin-coeff-Suc del: of-nat-Suc*)

lemma *diffs-cos-coeff*: $\text{diffs cos-coeff} = (\lambda n. - \text{sin-coeff } n)$
by (*simp add: diffs-def cos-coeff-Suc del: of-nat-Suc*)

lemma *sin-int-times-real*: $\text{sin (of-int } m * \text{of-real } x) = \text{of-real (sin (of-int } m * x))$
by (*metis sin-of-real of-real-mult of-real-of-int-eq*)

lemma *cos-int-times-real*: $\text{cos (of-int } m * \text{of-real } x) = \text{of-real (cos (of-int } m * x))$
by (*metis cos-of-real of-real-mult of-real-of-int-eq*)

Now at last we can get the derivatives of exp, sin and cos.

lemma *DERIV-sin* [*simp*]: $\text{DERIV sin } x \text{ :> } \text{cos } x$
for $x :: 'a :: \{\text{real-normed-field, banach}\}$
unfolding *sin-def cos-def scaleR-conv-of-real*
apply (*rule DERIV-cong*)
apply (*rule termdiffs [where K=of-real (norm x) + 1 :: 'a]*)
apply (*simp-all add: norm-less-p1 diffs-of-real diffs-sin-coeff diffs-cos-coeff*
summable-minus-iff scaleR-conv-of-real [symmetric]
summable-norm-sin [THEN summable-norm-cancel]
summable-norm-cos [THEN summable-norm-cancel])
done

declare *DERIV-sin*[*THEN DERIV-chain2, derivative-intros*]
and *DERIV-sin*[*THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros*]

lemmas *has-derivative-sin*[*derivative-intros*] = *DERIV-sin*[*THEN DERIV-compose-FDERIV*]

lemma *DERIV-cos* [*simp*]: $\text{DERIV cos } x \text{ :> } - \text{sin } x$
for $x :: 'a :: \{\text{real-normed-field, banach}\}$
unfolding *sin-def cos-def scaleR-conv-of-real*
apply (*rule DERIV-cong*)
apply (*rule termdiffs [where K=of-real (norm x) + 1 :: 'a]*)
apply (*simp-all add: norm-less-p1 diffs-of-real diffs-minus suminf-minus*
diffs-sin-coeff diffs-cos-coeff
summable-minus-iff scaleR-conv-of-real [symmetric]
summable-norm-sin [THEN summable-norm-cancel]
summable-norm-cos [THEN summable-norm-cancel])
done

declare *DERIV-cos*[*THEN DERIV-chain2, derivative-intros*]
and *DERIV-cos*[*THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros*]

lemmas *has-derivative-cos*[*derivative-intros*] = *DERIV-cos*[*THEN DERIV-compose-FDERIV*]

lemma *isCont-sin*: $\text{isCont sin } x$
for $x :: 'a :: \{\text{real-normed-field, banach}\}$
by (*rule DERIV-sin [THEN DERIV-isCont]*)

lemma *continuous-on-sin-real*: *continuous-on* {*a..b*} *sin* **for** *a::real*
using *continuous-at-imp-continuous-on isCont-sin* **by** *blast*

lemma *isCont-cos*: *isCont* *cos* *x*
for *x :: 'a::*{*real-normed-field,banach*}
by (*rule DERIV-cos [THEN DERIV-isCont]*)

lemma *continuous-on-cos-real*: *continuous-on* {*a..b*} *cos* **for** *a::real*
using *continuous-at-imp-continuous-on isCont-cos* **by** *blast*

context

fixes *f :: 'a::t2-space* \Rightarrow *'b::*{*real-normed-field,banach*}
begin

lemma *isCont-sin'* [*simp*]: *isCont* *f* *a* \Longrightarrow *isCont* ($\lambda x. \sin (f x)$) *a*
by (*rule isCont-o2 [OF - isCont-sin]*)

lemma *isCont-cos'* [*simp*]: *isCont* *f* *a* \Longrightarrow *isCont* ($\lambda x. \cos (f x)$) *a*
by (*rule isCont-o2 [OF - isCont-cos]*)

lemma *tendsto-sin* [*tendsto-intros*]: (*f* \longrightarrow *a*) *F* \Longrightarrow ($\lambda x. \sin (f x)$) \longrightarrow *sin*
a) *F*
by (*rule isCont-tendsto-compose [OF isCont-sin]*)

lemma *tendsto-cos* [*tendsto-intros*]: (*f* \longrightarrow *a*) *F* \Longrightarrow ($\lambda x. \cos (f x)$) \longrightarrow *cos*
a) *F*
by (*rule isCont-tendsto-compose [OF isCont-cos]*)

lemma *continuous-sin* [*continuous-intros*]: *continuous* *F* *f* \Longrightarrow *continuous* *F* ($\lambda x. \sin (f x)$)
unfolding *continuous-def* **by** (*rule tendsto-sin*)

lemma *continuous-on-sin* [*continuous-intros*]: *continuous-on* *s* *f* \Longrightarrow *continuous-on*
s ($\lambda x. \sin (f x)$)
unfolding *continuous-on-def* **by** (*auto intro: tendsto-sin*)

lemma *continuous-cos* [*continuous-intros*]: *continuous* *F* *f* \Longrightarrow *continuous* *F* ($\lambda x. \cos (f x)$)
unfolding *continuous-def* **by** (*rule tendsto-cos*)

lemma *continuous-on-cos* [*continuous-intros*]: *continuous-on* *s* *f* \Longrightarrow *continuous-on*
s ($\lambda x. \cos (f x)$)
unfolding *continuous-on-def* **by** (*auto intro: tendsto-cos*)

end

lemma *continuous-within-sin*: *continuous* (*at* *z* *within* *s*) *sin*
for *z :: 'a::*{*real-normed-field,banach*}

by (simp add: continuous-within tendsto-sin)

lemma continuous-within-cos: continuous (at z within s) cos
 for z :: 'a::{real-normed-field,banach}
 by (simp add: continuous-within tendsto-cos)

113.11 Properties of Sine and Cosine

lemma sin-zero [simp]: sin 0 = 0
 by (simp add: sin-def sin-coeff-def scaleR-conv-of-real)

lemma cos-zero [simp]: cos 0 = 1
 by (simp add: cos-def cos-coeff-def scaleR-conv-of-real)

lemma DERIV-fun-sin: DERIV g x :> m \implies DERIV ($\lambda x.$ sin (g x)) x :> cos (g x) * m
 by (fact derivative-intros)

lemma DERIV-fun-cos: DERIV g x :> m \implies DERIV ($\lambda x.$ cos(g x)) x :> - sin (g x) * m
 by (fact derivative-intros)

113.12 Deriving the Addition Formulas

The product of two cosine series.

lemma cos-x-cos-y:
 fixes x :: 'a::{real-normed-field,banach}
 shows
 ($\lambda p.$ $\sum_{n \leq p}.$
 if even p \wedge even n
 then $((-1)^{\wedge(p \text{ div } 2)} * (p \text{ choose } n) / (fact p)) *_{\mathbb{R}} (x^{\wedge n}) * y^{\wedge(p-n)}$ else
 0)
 sums (cos x * cos y)

proof -

have (cos-coeff n * cos-coeff (p - n)) *_R (x[^]n * y[^](p - n)) =
 (if even p \wedge even n then $((-1)^{\wedge(p \text{ div } 2)} * (p \text{ choose } n) / (fact p)) *_{\mathbb{R}} (x^{\wedge n})$
 * y[^](p - n)
 else 0)

if n \leq p **for** n p :: nat

proof -

from that **have** *: even n \implies even p \implies

$(-1)^{\wedge(n \text{ div } 2)} * (-1)^{\wedge((p - n) \text{ div } 2)} = (-1 :: \text{real})^{\wedge(p \text{ div } 2)}$

by (metis div-add power-add le-add-diff-inverse odd-add)

with that **show** ?thesis

by (auto simp: algebra-simps cos-coeff-def binomial-fact)

qed

then have ($\lambda p.$ $\sum_{n \leq p}.$ if even p \wedge even n
 then $((-1)^{\wedge(p \text{ div } 2)} * (p \text{ choose } n) / (fact p)) *_{\mathbb{R}} (x^{\wedge n}) * y^{\wedge(p-n)}$
 else 0) =

```

      (λp. ∑ n≤p. (cos-coeff n * cos-coeff (p - n)) *R (x^n * y^(p-n)))
    by simp
  also have ... = (λp. ∑ n≤p. (cos-coeff n *R x^n) * (cos-coeff (p - n) *R
y^(p-n)))
    by (simp add: algebra-simps)
  also have ... sums (cos x * cos y)
    using summable-norm-cos
    by (auto simp: cos-def scaleR-conv-of-real intro!: Cauchy-product-sums)
  finally show ?thesis .
qed

```

The product of two sine series.

lemma *sin-x-sin-y*:

fixes $x :: 'a :: \{real-normed-field, banach\}$

shows

```

(λp. ∑ n≤p.
  if even p ∧ odd n
  then -((-1) ^(p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p-n)
  else 0)
sums (sin x * sin y)

```

proof –

```

have (sin-coeff n * sin-coeff (p - n)) *R (x^n * y^(p-n)) =
  (if even p ∧ odd n
  then -((-1) ^(p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p-n)
  else 0)

```

if $n \leq p$ **for** $n p :: nat$

proof –

```

have (-1) ^((n - Suc 0) div 2) * (-1) ^((p - Suc n) div 2) = -((-1 ::
real) ^(p div 2))

```

if np : odd n even p

proof –

have $p > 0$

using $\langle n \leq p \rangle$ *neq0-conv that(1)* **by** *blast*

then have §: $(-1 :: real) ^{^(p div 2 - Suc 0)} = -((-1) ^{^(p div 2)})$

using $\langle even p \rangle$ **by** *(auto simp add: dvd-def power-eq-if)*

```

from  $\langle n \leq p \rangle$   $np$  have *:  $n - Suc 0 + (p - Suc n) = p - Suc (Suc 0)$   $Suc$ 
 $(Suc 0) \leq p$ 

```

by *arith+*

have $(p - Suc (Suc 0)) div 2 = p div 2 - Suc 0$

by *simp*

with $\langle n \leq p \rangle$ np § * **show** ?thesis

by *(simp add: flip: div-add power-add)*

qed

then show ?thesis

using $\langle n \leq p \rangle$ **by** *(auto simp: algebra-simps sin-coeff-def binomial-fact)*

qed

then have $(\lambda p. \sum n \leq p. \text{if even } p \wedge \text{odd } n$

```

  then -((-1) ^(p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p-n)

```

$else 0) =$

$(\lambda p. \sum n \leq p. (\text{sin-coeff } n * \text{sin-coeff } (p - n)) *_R (x^{\wedge} n * y^{\wedge}(p-n)))$
by simp
also have ... = $(\lambda p. \sum n \leq p. (\text{sin-coeff } n *_R x^{\wedge} n) * (\text{sin-coeff } (p - n) *_R y^{\wedge}(p-n)))$
by (simp add: algebra-simps)
also have ... *sums (sin x * sin y)*
using summable-norm-sin
by (auto simp: sin-def scaleR-conv-of-real intro!: Cauchy-product-sums)
finally show ?thesis .
qed

lemma *sums-cos-x-plus-y:*

fixes $x :: 'a :: \{\text{real-normed-field, banach}\}$

shows

$(\lambda p. \sum n \leq p.$
if even p
then $((-1)^{\wedge}(p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_R (x^{\wedge} n) * y^{\wedge}(p-n)$
else 0)
sums cos (x + y)

proof –

have

$(\sum n \leq p.$
if even p then $((-1)^{\wedge}(p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_R (x^{\wedge} n) * y^{\wedge}(p-n)$
*else 0) = cos-coeff p *_R ((x + y)^{\wedge} p)*

for $p :: \text{nat}$

proof –

have

$(\sum n \leq p. \text{if even } p \text{ then } ((-1)^{\wedge}(p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_R (x^{\wedge} n)$
 $* y^{\wedge}(p-n) \text{ else } 0) =$
 $(\text{if even } p \text{ then } \sum n \leq p. ((-1)^{\wedge}(p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_R (x^{\wedge} n)$
 $* y^{\wedge}(p-n) \text{ else } 0)$

by simp

also have ... =

$(\text{if even } p$
then of-real $((-1)^{\wedge}(p \text{ div } 2) / (\text{fact } p)) * (\sum n \leq p. (p \text{ choose } n) *_R (x^{\wedge} n)$
 $* y^{\wedge}(p-n))$
else 0)

by (auto simp: sum-distrib-left field-simps scaleR-conv-of-real nonzero-of-real-divide)

also have ... = $\text{cos-coeff } p *_R ((x + y)^{\wedge} p)$

by (simp add: cos-coeff-def binomial-ring [of x y] scaleR-conv-of-real atLeast0AtMost)

finally show ?thesis .

qed

then have

$(\lambda p. \sum n \leq p.$
if even p
then $((-1)^{\wedge}(p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_R (x^{\wedge} n) * y^{\wedge}(p-n)$
else 0) = $(\lambda p. \text{cos-coeff } p *_R ((x+y)^{\wedge} p))$

by simp

also have ... $\text{sums cos } (x + y)$
 by (rule cos-converges)
 finally show ?thesis .
 qed

theorem cos-add:

fixes $x :: 'a::\{\text{real-normed-field,banach}\}$
 shows $\text{cos } (x + y) = \text{cos } x * \text{cos } y - \text{sin } x * \text{sin } y$
proof –
 have
 (if even $p \wedge$ even n
 then $((-1) \wedge (p \text{ div } 2) * \text{int } (p \text{ choose } n) / (\text{fact } p)) *_{\mathbb{R}} (x \wedge n) * y \wedge (p-n)$
 else 0) –
 (if even $p \wedge$ odd n
 then $-((-1) \wedge (p \text{ div } 2) * \text{int } (p \text{ choose } n) / (\text{fact } p)) *_{\mathbb{R}} (x \wedge n) * y \wedge (p-n)$
 else 0) =
 (if even p then $((-1) \wedge (p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_{\mathbb{R}} (x \wedge n) * y \wedge (p-n)$
 else 0)
 if $n \leq p$ for $n p :: \text{nat}$
 by simp
 then have
 ($\lambda p. \sum_{n \leq p}. (\text{if even } p \text{ then } ((-1) \wedge (p \text{ div } 2) * (p \text{ choose } n) / (\text{fact } p)) *_{\mathbb{R}} (x \wedge n) * y \wedge (p-n) \text{ else } 0)$
 sums $(\text{cos } x * \text{cos } y - \text{sin } x * \text{sin } y)$
 using sums-diff [OF cos-x-cos-y [of x y] sin-x-sin-y [of x y]]
 by (simp add: sum-subtractf [symmetric])
 then show ?thesis
 by (blast intro: sums-cos-x-plus-y sums-unique2)
 qed

lemma sin-minus-converges: $(\lambda n. -(\text{sin-coeff } n *_{\mathbb{R}} (-x) \wedge n)) \text{ sums sin } x$

proof –
 have [simp]: $\bigwedge n. -(\text{sin-coeff } n *_{\mathbb{R}} (-x) \wedge n) = (\text{sin-coeff } n *_{\mathbb{R}} x \wedge n)$
 by (auto simp: sin-coeff-def elim!: oddE)
 show ?thesis
 by (simp add: sin-def summable-norm-sin [THEN summable-norm-cancel, THEN summable-sums])
 qed

lemma sin-minus [simp]: $\text{sin } (-x) = -\text{sin } x$

for $x :: 'a::\{\text{real-normed-algebra-1,banach}\}$
 using sin-minus-converges [of x]
 by (auto simp: sin-def summable-norm-sin [THEN summable-norm-cancel] suminf-minus sums-iff equation-minus-iff)

lemma cos-minus-converges: $(\lambda n. (\text{cos-coeff } n *_{\mathbb{R}} (-x) \wedge n)) \text{ sums cos } x$

proof –
 have [simp]: $\bigwedge n. (\text{cos-coeff } n *_{\mathbb{R}} (-x) \wedge n) = (\text{cos-coeff } n *_{\mathbb{R}} x \wedge n)$
 by (auto simp: Transcendental.cos-coeff-def elim!: evenE)

show *?thesis*
by (*simp add: cos-def summable-norm-cos [THEN summable-norm-cancel, THEN summable-sums]*)
qed

lemma *cos-minus [simp]: $\cos(-x) = \cos x$*
for $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$
using *cos-minus-converges [of x] by (metis cos-def sums-unique)*

lemma *cos-abs-real [simp]: $\cos |x :: \text{real}| = \cos x$*
by (*simp add: abs-iff*)

lemma *sin-cos-squared-add [simp]: $(\sin x)^2 + (\cos x)^2 = 1$*
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
using *cos-add [of x -x]*
by (*simp add: power2-eq-square algebra-simps*)

lemma *sin-cos-squared-add2 [simp]: $(\cos x)^2 + (\sin x)^2 = 1$*
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
by (*subst add.commute, rule sin-cos-squared-add*)

lemma *sin-cos-squared-add3 [simp]: $\cos x * \cos x + \sin x * \sin x = 1$*
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
using *sin-cos-squared-add2 [unfolded power2-eq-square]* .

lemma *sin-squared-eq: $(\sin x)^2 = 1 - (\cos x)^2$*
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
unfolding *eq-diff-eq by (rule sin-cos-squared-add)*

lemma *cos-squared-eq: $(\cos x)^2 = 1 - (\sin x)^2$*
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
unfolding *eq-diff-eq by (rule sin-cos-squared-add2)*

lemma *abs-sin-le-one [simp]: $|\sin x| \leq 1$*
for $x :: \text{real}$
by (*rule power2-le-imp-le (simp-all add: sin-squared-eq)*)

lemma *sin-ge-minus-one [simp]: $-1 \leq \sin x$*
for $x :: \text{real}$
using *abs-sin-le-one [of x] by (simp add: abs-le-iff)*

lemma *sin-le-one [simp]: $\sin x \leq 1$*
for $x :: \text{real}$
using *abs-sin-le-one [of x] by (simp add: abs-le-iff)*

lemma *abs-cos-le-one [simp]: $|\cos x| \leq 1$*
for $x :: \text{real}$
by (*rule power2-le-imp-le (simp-all add: cos-squared-eq)*)

lemma *cos-ge-minus-one* [simp]: $-1 \leq \cos x$
for $x :: \text{real}$
using *abs-cos-le-one* [of x] **by** (*simp add: abs-le-iff*)

lemma *cos-le-one* [simp]: $\cos x \leq 1$
for $x :: \text{real}$
using *abs-cos-le-one* [of x] **by** (*simp add: abs-le-iff*)

lemma *cos-diff*: $\cos (x - y) = \cos x * \cos y + \sin x * \sin y$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
using *cos-add* [of $x - y$] **by** *simp*

lemma *cos-double*: $\cos(2*x) = (\cos x)^2 - (\sin x)^2$
for $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$
using *cos-add* [where $x=x$ and $y=x$] **by** (*simp add: power2-eq-square*)

lemma *sin-cos-le1*: $|\sin x * \sin y + \cos x * \cos y| \leq 1$
for $x :: \text{real}$
using *cos-diff* [of $x y$] **by** (*metis abs-cos-le-one add.commute*)

lemma *DERIV-fun-pow*: $\text{DERIV } g \ x \ :> \ m \implies \text{DERIV } (\lambda x. (g \ x) \ ^n) \ x \ :> \ \text{real } n * (g \ x) \ ^{(n-1)} * m$
by (*auto intro!: derivative-eq-intros simp*)

lemma *DERIV-fun-exp*: $\text{DERIV } g \ x \ :> \ m \implies \text{DERIV } (\lambda x. \exp (g \ x)) \ x \ :> \ \exp (g \ x) * m$
by (*auto intro!: derivative-intros*)

113.13 The Constant Pi

definition *pi* :: *real*
where $pi = 2 * (\text{THE } x. 0 \leq x \wedge x \leq 2 \wedge \cos x = 0)$

Show that there’s a least positive x with $\cos x = (0::'a)$; hence define pi.

lemma *sin-paired*: $(\lambda n. (-1) \ ^n / (\text{fact } (2 * n + 1)) * x \ ^{(2 * n + 1)}) \ \text{sums } \sin x$

for $x :: \text{real}$
proof –
have $(\lambda n. \sum k = n*2..<n*2 + 2. \sin\text{-coeff } k * x \ ^k) \ \text{sums } \sin x$
by (*rule sums-group*) (*use sin-converges* [of x , *unfolded scaleR-conv-of-real*] **in** *auto*)
then show *?thesis*
by (*simp add: sin-coeff-def ac-simps*)
qed

lemma *sin-gt-zero-02*:
fixes $x :: \text{real}$
assumes $0 < x$ **and** $x < 2$
shows $0 < \sin x$

proof –

let $?f = \lambda n :: \text{nat}. \sum k = n * 2 .. < n * 2 + 2. (-1) ^ k / (\text{fact } (2 * k + 1)) * x ^ (2 * k + 1)$

have $pos: \forall n. 0 < ?f n$

proof

fix $n :: \text{nat}$

let $?k2 = \text{real } (\text{Suc } (\text{Suc } (4 * n)))$

let $?k3 = \text{real } (\text{Suc } (\text{Suc } (\text{Suc } (4 * n))))$

have $x * x < ?k2 * ?k3$

using $assms$ by $(\text{intro } \text{mult-strict-mono}', \text{simp-all})$

then have $x * x * x * x ^ (n * 4) < ?k2 * ?k3 * x * x ^ (n * 4)$

by $(\text{intro } \text{mult-strict-right-mono } \text{zero-less-power } \langle 0 < x \rangle)$

then show $0 < ?f n$

by $(\text{simp add: ac-simps divide-less-eq})$

qed

have $sums: ?f \text{ sums } \sin x$

by $(\text{rule } \text{sin-paired } [\text{THEN } \text{sums-group}]) \text{ simp}$

show $0 < \sin x$

unfolding $sums-unique$ $[OF \text{ sums}]$ using $sums-summable$ $[OF \text{ sums}]$ pos by $(\text{simp add: suminf-pos})$

qed

lemma $\text{cos-double-less-one}: 0 < x \implies x < 2 \implies \cos (2 * x) < 1$

for $x :: \text{real}$

using sin-gt-zero-02 $[\text{where } x = x]$ by $(\text{auto simp: cos-squared-eq cos-double})$

lemma $\text{cos-paired}: (\lambda n. (-1) ^ n / (\text{fact } (2 * n)) * x ^ (2 * n)) \text{ sums } \cos x$

for $x :: \text{real}$

proof –

have $(\lambda n. \sum k = n * 2 .. < n * 2 + 2. \text{cos-coeff } k * x ^ k) \text{ sums } \cos x$

by $(\text{rule } \text{sums-group})$ $(\text{use } \text{cos-converges } [\text{of } x, \text{unfolded } \text{scaleR-conv-of-real}] \text{ in } \text{auto})$

then show $?thesis$

by $(\text{simp add: cos-coeff-def ac-simps})$

qed

lemma sum-pos-lt-pair :

fixes $f :: \text{nat} \Rightarrow \text{real}$

assumes $f: \text{summable } f$ and $fplus: \bigwedge d. 0 < f (k + (\text{Suc } (\text{Suc } 0) * d)) + f (k + ((\text{Suc } (\text{Suc } 0) * d) + 1))$

shows $\text{sum } f \{..<k\} < \text{suminf } f$

proof –

have $(\lambda n. \sum n = n * \text{Suc } (\text{Suc } 0) .. < n * \text{Suc } (\text{Suc } 0) + \text{Suc } (\text{Suc } 0). f (n + k))$

$\text{sums } (\sum n. f (n + k))$

proof $(\text{rule } \text{sums-group})$

show $(\lambda n. f (n + k)) \text{ sums } (\sum n. f (n + k))$

by $(\text{simp add: f-summable-iff-shift summable-sums})$

qed auto

with $fplus$ have $0 < (\sum n. f (n + k))$

```

apply (simp add: add commute)
apply (metis (no-types, lifting) suminf-pos summable-def sums-unique)
done
then show ?thesis
  by (simp add: f suminf-minus-initial-segment)
qed

lemma cos-two-less-zero [simp]:  $\cos 2 < (0::\text{real})$ 
proof –
  note fact-Suc [simp del]
  from sums-minus [OF cos-paired]
  have *:  $(\lambda n. - ((-1)^n * 2^{(2 * n)} / \text{fact } (2 * n))) \text{ sums} - \cos (2::\text{real})$ 
    by simp
  then have sm: summable  $(\lambda n. - ((-1::\text{real})^n * 2^{(2 * n)} / (\text{fact } (2 * n))))$ 
    by (rule sums-summable)
  have  $0 < (\sum n < \text{Suc } (\text{Suc } (\text{Suc } 0)). - ((-1::\text{real})^n * 2^{(2 * n)} / (\text{fact } (2 * n))))$ 
    by (simp add: fact-num-eq-if power-eq-if)
  moreover have  $(\sum n < \text{Suc } (\text{Suc } (\text{Suc } 0)). - ((-1::\text{real})^n * 2^{(2 * n)} / (\text{fact } (2 * n)))) <$ 
     $(\sum n. - ((-1)^n * 2^{(2 * n)} / (\text{fact } (2 * n))))$ 
    proof –
    {
      fix d
      let ?six4d = Suc (Suc (Suc (Suc (Suc (Suc (Suc (4 * d)))))))
      have  $(4::\text{real}) * (\text{fact } (?six4d)) < (\text{Suc } (\text{Suc } (?six4d)) * \text{fact } (\text{Suc } (?six4d)))$ 
        unfolding of-nat-mult by (rule mult-strict-mono) (simp-all add: fact-less-mono)
      then have  $(4::\text{real}) * (\text{fact } (?six4d)) < (\text{fact } (\text{Suc } (\text{Suc } (?six4d))))$ 
        by (simp only: fact-Suc [of Suc (?six4d)] of-nat-mult of-nat-fact)
      then have  $(4::\text{real}) * \text{inverse } (\text{fact } (\text{Suc } (\text{Suc } (?six4d)))) < \text{inverse } (\text{fact } (?six4d))$ 
        by (simp add: inverse-eq-divide less-divide-eq)
    }
  then show ?thesis
    by (force intro!: sum-pos-lt-pair [OF sm] simp add: divide-inverse algebra-simps)
  qed
  ultimately have  $0 < (\sum n. - ((-1::\text{real})^n * 2^{(2 * n)} / (\text{fact } (2 * n))))$ 
    by (rule order-less-trans)
  moreover from * have  $-\cos 2 = (\sum n. - ((-1::\text{real})^n * 2^{(2 * n)} / (\text{fact } (2 * n))))$ 
    by (rule sums-unique)
  ultimately have  $(0::\text{real}) < -\cos 2$  by simp
  then show ?thesis by simp
qed

```

lemmas *cos-two-neq-zero* [*simp*] = *cos-two-less-zero* [*THEN less-imp-neq*]

lemmas *cos-two-le-zero* [*simp*] = *cos-two-less-zero* [*THEN order-less-imp-le*]

```

lemma cos-is-zero:  $\exists!x::\text{real}. 0 \leq x \wedge x \leq 2 \wedge \cos x = 0$ 
proof (rule ex-ex1I)
  show  $\exists x::\text{real}. 0 \leq x \wedge x \leq 2 \wedge \cos x = 0$ 
    by (rule IVT2) simp-all
next
  fix a b :: real
  assume ab:  $0 \leq a \wedge a \leq 2 \wedge \cos a = 0 \wedge 0 \leq b \wedge b \leq 2 \wedge \cos b = 0$ 
  have cosd:  $\bigwedge x::\text{real}. \cos \text{ differentiable } (at\ x)$ 
    unfolding real-differentiable-def by (auto intro: DERIV-cos)
  show  $a = b$ 
  proof (cases a b rule: linorder-cases)
    case less
    then obtain z where  $a < z < b$  (cos has-real-derivative 0) (at z)
      using Rolle by (metis cosd continuous-on-cos-real ab)
    then have  $\sin z = 0$ 
      using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
    then show ?thesis
      by (metis <a < z> <z < b> ab order-less-le-trans less-le sin-gt-zero-02)
    next
    case greater
    then obtain z where  $b < z < a$  (cos has-real-derivative 0) (at z)
      using Rolle by (metis cosd continuous-on-cos-real ab)
    then have  $\sin z = 0$ 
      using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
    then show ?thesis
      by (metis <b < z> <z < a> ab order-less-le-trans less-le sin-gt-zero-02)
  qed auto
qed

lemma pi-half:  $\pi/2 = (THE\ x. 0 \leq x \wedge x \leq 2 \wedge \cos x = 0)$ 
  by (simp add: pi-def)

lemma cos-pi-half [simp]:  $\cos (\pi/2) = 0$ 
  by (simp add: pi-half cos-is-zero [THEN theI'])

lemma cos-of-real-pi-half [simp]:  $\cos ((\text{of-real } \pi/2) :: 'a) = 0$ 
  if SORT-CONSTRAINT('a::{real-field,banach,real-normed-algebra-1})
  by (metis cos-pi-half cos-of-real eq-numeral-simps(4))
    nonzero-of-real-divide of-real-0 of-real-numeral)

lemma pi-half-gt-zero [simp]:  $0 < \pi/2$ 
proof –
  have  $0 \leq \pi/2$ 
    by (simp add: pi-half cos-is-zero [THEN theI'])
  then show ?thesis
    by (metis cos-pi-half cos-zero less-eq-real-def one-neq-zero)
qed

```

lemmas *pi-half-neq-zero* [simp] = *pi-half-gt-zero* [THEN *less-imp-neq, symmetric*]
lemmas *pi-half-ge-zero* [simp] = *pi-half-gt-zero* [THEN *order-less-imp-le*]

lemma *pi-half-less-two* [simp]: $\pi/2 < 2$

proof –

have $\pi/2 \leq 2$

by (simp add: *pi-half cos-is-zero* [THEN *theI'*])

then show *?thesis*

by (metis *cos-pi-half cos-two-neq-zero le-less*)

qed

lemmas *pi-half-neq-two* [simp] = *pi-half-less-two* [THEN *less-imp-neq*]

lemmas *pi-half-le-two* [simp] = *pi-half-less-two* [THEN *order-less-imp-le*]

lemma *pi-gt-zero* [simp]: $0 < \pi$

using *pi-half-gt-zero* **by** *simp*

lemma *pi-ge-zero* [simp]: $0 \leq \pi$

by (rule *pi-gt-zero* [THEN *order-less-imp-le*])

lemma *pi-neq-zero* [simp]: $\pi \neq 0$

by (rule *pi-gt-zero* [THEN *less-imp-neq, symmetric*])

lemma *pi-not-less-zero* [simp]: $\neg \pi < 0$

by (simp add: *linorder-not-less*)

lemma *minus-pi-half-less-zero*: $-(\pi/2) < 0$

by *simp*

lemma *m2pi-less-pi*: $-(2*\pi) < \pi$

by *simp*

lemma *sin-pi-half* [simp]: $\sin(\pi/2) = 1$

using *sin-cos-squared-add2* [where $x = \pi/2$]

using *sin-gt-zero-02* [OF *pi-half-gt-zero pi-half-less-two*]

by (simp add: *power2-eq-1-iff*)

lemma *sin-of-real-pi-half* [simp]: $\sin((\text{of-real } \pi/2) :: 'a) = 1$

if *SORT-CONSTRAINT*('a::{*real-field, banach, real-normed-algebra-1*})

using *sin-pi-half*

by (metis *sin-pi-half eq-numeral-simps(4) nonzero-of-real-divide of-real-1 of-real-numeral sin-of-real*)

lemma *sin-cos-eq*: $\sin x = \cos(\text{of-real } \pi/2 - x)$

for $x :: 'a :: \{\text{real-normed-field, banach}\}$

by (simp add: *cos-diff*)

lemma *minus-sin-cos-eq*: $-\sin x = \cos(x + \text{of-real } \pi/2)$

for $x :: 'a :: \{\text{real-normed-field, banach}\}$

by (*simp add: cos-add nonzero-of-real-divide*)

lemma *cos-sin-eq*: $\cos x = \sin (\text{of-real } \pi/2 - x)$
 for $x :: 'a::\{\text{real-normed-field,banach}\}$
 using *sin-cos-eq* [*of of-real pi/2 - x*] by *simp*

lemma *sin-add*: $\sin (x + y) = \sin x * \cos y + \cos x * \sin y$
 for $x :: 'a::\{\text{real-normed-field,banach}\}$
 using *cos-add* [*of of-real pi/2 - x - y*]
 by (*simp add: cos-sin-eq*) (*simp add: sin-cos-eq*)

lemma *sin-diff*: $\sin (x - y) = \sin x * \cos y - \cos x * \sin y$
 for $x :: 'a::\{\text{real-normed-field,banach}\}$
 using *sin-add* [*of x - y*] by *simp*

lemma *sin-double*: $\sin(2 * x) = 2 * \sin x * \cos x$
 for $x :: 'a::\{\text{real-normed-field,banach}\}$
 using *sin-add* [**where** $x=x$ **and** $y=x$] by *simp*

lemma *cos-of-real-pi* [*simp*]: $\cos (\text{of-real } \pi) = -1$
 using *cos-add* [**where** $x = \pi/2$ **and** $y = \pi/2$]
 by (*simp add: cos-of-real*)

lemma *sin-of-real-pi* [*simp*]: $\sin (\text{of-real } \pi) = 0$
 using *sin-add* [**where** $x = \pi/2$ **and** $y = \pi/2$]
 by (*simp add: sin-of-real*)

lemma *cos-pi* [*simp*]: $\cos \pi = -1$
 using *cos-add* [**where** $x = \pi/2$ **and** $y = \pi/2$] by *simp*

lemma *sin-pi* [*simp*]: $\sin \pi = 0$
 using *sin-add* [**where** $x = \pi/2$ **and** $y = \pi/2$] by *simp*

lemma *sin-periodic-pi* [*simp*]: $\sin (x + \pi) = - \sin x$
 by (*simp add: sin-add*)

lemma *sin-periodic-pi2* [*simp*]: $\sin (\pi + x) = - \sin x$
 by (*simp add: sin-add*)

lemma *cos-periodic-pi* [*simp*]: $\cos (x + \pi) = - \cos x$
 by (*simp add: cos-add*)

lemma *cos-periodic-pi2* [*simp*]: $\cos (\pi + x) = - \cos x$
 by (*simp add: cos-add*)

lemma *sin-periodic* [*simp*]: $\sin (x + 2 * \pi) = \sin x$
 by (*simp add: sin-add sin-double cos-double*)

lemma *cos-periodic* [*simp*]: $\cos (x + 2 * \pi) = \cos x$

by (simp add: cos-add sin-double cos-double)

lemma *cos-npi* [simp]: $\cos (\text{real } n * \pi) = (-1) ^ n$
by (induct n) (auto simp: distrib-right)

lemma *cos-npi2* [simp]: $\cos (\pi * \text{real } n) = (-1) ^ n$
by (metis cos-npi mult.commute)

lemma *sin-npi* [simp]: $\sin (\text{real } n * \pi) = 0$
for $n :: \text{nat}$
by (induct n) (auto simp: distrib-right)

lemma *sin-npi2* [simp]: $\sin (\pi * \text{real } n) = 0$
for $n :: \text{nat}$
by (simp add: mult.commute [of pi])

lemma *cos-two-pi* [simp]: $\cos (2 * \pi) = 1$
by (simp add: cos-double)

lemma *sin-two-pi* [simp]: $\sin (2 * \pi) = 0$
by (simp add: sin-double)

context

fixes $w :: 'a::\{\text{real-normed-field}, \text{banach}\}$

begin

lemma *sin-times-sin*: $\sin w * \sin z = (\cos (w - z) - \cos (w + z)) / 2$
by (simp add: cos-diff cos-add)

lemma *sin-times-cos*: $\sin w * \cos z = (\sin (w + z) + \sin (w - z)) / 2$
by (simp add: sin-diff sin-add)

lemma *cos-times-sin*: $\cos w * \sin z = (\sin (w + z) - \sin (w - z)) / 2$
by (simp add: sin-diff sin-add)

lemma *cos-times-cos*: $\cos w * \cos z = (\cos (w - z) + \cos (w + z)) / 2$
by (simp add: cos-diff cos-add)

lemma *cos-double-cos*: $\cos (2 * w) = 2 * \cos w ^ 2 - 1$
by (simp add: cos-double sin-squared-eq)

lemma *cos-double-sin*: $\cos (2 * w) = 1 - 2 * \sin w ^ 2$
by (simp add: cos-double sin-squared-eq)

end

lemma *sin-plus-sin*: $\sin w + \sin z = 2 * \sin ((w + z) / 2) * \cos ((w - z) / 2)$
for $w :: 'a::\{\text{real-normed-field}, \text{banach}\}$

```

apply (simp add: mult.assoc sin-times-cos)
apply (simp add: field-simps)
done

```

```

lemma sin-diff-sin:  $\sin w - \sin z = 2 * \sin ((w - z) / 2) * \cos ((w + z) / 2)$ 
for  $w :: 'a::\{\text{real-normed-field}, \text{banach}\}$ 
apply (simp add: mult.assoc sin-times-cos)
apply (simp add: field-simps)
done

```

```

lemma cos-plus-cos:  $\cos w + \cos z = 2 * \cos ((w + z) / 2) * \cos ((w - z) / 2)$ 
for  $w :: 'a::\{\text{real-normed-field}, \text{banach}, \text{field}\}$ 
apply (simp add: mult.assoc cos-times-cos)
apply (simp add: field-simps)
done

```

```

lemma cos-diff-cos:  $\cos w - \cos z = 2 * \sin ((w + z) / 2) * \sin ((z - w) / 2)$ 
for  $w :: 'a::\{\text{real-normed-field}, \text{banach}, \text{field}\}$ 
apply (simp add: mult.assoc sin-times-sin)
apply (simp add: field-simps)
done

```

```

lemma sin-pi-minus [simp]:  $\sin (\pi - x) = \sin x$ 
by (metis sin-minus sin-periodic-pi minus-minus uminus-add-conv-diff)

```

```

lemma cos-pi-minus [simp]:  $\cos (\pi - x) = - (\cos x)$ 
by (metis cos-minus cos-periodic-pi uminus-add-conv-diff)

```

```

lemma sin-minus-pi [simp]:  $\sin (x - \pi) = - (\sin x)$ 
by (simp add: sin-diff)

```

```

lemma cos-minus-pi [simp]:  $\cos (x - \pi) = - (\cos x)$ 
by (simp add: cos-diff)

```

```

lemma sin-2pi-minus [simp]:  $\sin (2 * \pi - x) = - (\sin x)$ 
by (metis sin-periodic-pi2 add-diff-eq mult-2 sin-pi-minus)

```

```

lemma cos-2pi-minus [simp]:  $\cos (2 * \pi - x) = \cos x$ 
by (metis (no-types, opaque-lifting) cos-add cos-minus cos-two-pi sin-minus sin-two-pi
diff-0-right minus-diff-eq mult-1 mult-zero-left uminus-add-conv-diff)

```

```

lemma sin-gt-zero2:  $0 < x \implies x < \pi/2 \implies 0 < \sin x$ 
by (metis sin-gt-zero-02 order-less-trans pi-half-less-two)

```

```

lemma sin-less-zero:
  assumes  $-\pi/2 < x$  and  $x < 0$ 
  shows  $\sin x < 0$ 
proof -
  have  $0 < \sin (-x)$ 

```

using *assms* by (*simp only: sin-gt-zero2*)
 then show *?thesis* by *simp*
 qed

lemma *pi-less-4*: $\pi < 4$
 using *pi-half-less-two* by *auto*

lemma *cos-gt-zero*: $0 < x \implies x < \pi/2 \implies 0 < \cos x$
 by (*simp add: cos-sin-eq sin-gt-zero2*)

lemma *cos-gt-zero-pi*: $-(\pi/2) < x \implies x < \pi/2 \implies 0 < \cos x$
 using *cos-gt-zero* [*of x*] *cos-gt-zero* [*of -x*]
 by (*cases rule: linorder-cases* [*of x 0*]) *auto*

lemma *cos-ge-zero*: $-(\pi/2) \leq x \implies x \leq \pi/2 \implies 0 \leq \cos x$
 by (*auto simp: order-le-less cos-gt-zero-pi*)
 (*metis cos-pi-half eq-divide-eq eq-numeral-simps(4)*)

lemma *sin-gt-zero*: $0 < x \implies x < \pi \implies 0 < \sin x$
 by (*simp add: sin-cos-eq cos-gt-zero-pi*)

lemma *sin-lt-zero*: $\pi < x \implies x < 2 * \pi \implies \sin x < 0$
 using *sin-gt-zero* [*of x - pi*]
 by (*simp add: sin-diff*)

lemma *pi-ge-two*: $2 \leq \pi$

proof (*rule ccontr*)

assume $\neg ?thesis$

then have $\pi < 2$ by *auto*

have $\exists y > \pi. y < 2 \wedge y < 2 * \pi$

proof (*cases* $2 < 2 * \pi$)

case *True*

with *dense*[*OF* $\langle \pi < 2 \rangle$] show *?thesis* by *auto*

next

case *False*

have $\pi < 2 * \pi$ by *auto*

from *dense*[*OF this*] and *False* show *?thesis* by *auto*

qed

then obtain *y* where $\pi < y$ and $y < 2$ and $y < 2 * \pi$

by *blast*

then have $0 < \sin y$

using *sin-gt-zero-02* by *auto*

moreover have $\sin y < 0$

using *sin-gt-zero*[*of y - pi*] $\langle \pi < y \rangle$ and $\langle y < 2 * \pi \rangle$ *sin-periodic-pi*[*of y - pi*]

by *auto*

ultimately show *False* by *auto*

qed

lemma *sin-ge-zero*: $0 \leq x \implies x \leq \pi \implies 0 \leq \sin x$
by (*auto simp: order-le-less sin-gt-zero*)

lemma *sin-le-zero*: $\pi \leq x \implies x < 2 * \pi \implies \sin x \leq 0$
using *sin-ge-zero* [*of x - pi*] **by** (*simp add: sin-diff*)

lemma *sin-pi-divide-n-ge-0* [*simp*]:
assumes $n \neq 0$
shows $0 \leq \sin (\pi / \text{real } n)$
by (*rule sin-ge-zero*) (*use assms in <simp-all add: field-split-simps>*)

lemma *sin-pi-divide-n-gt-0*:
assumes $2 \leq n$
shows $0 < \sin (\pi / \text{real } n)$
by (*rule sin-gt-zero*) (*use assms in <simp-all add: field-split-simps>*)

Proof resembles that of *cos-is-zero* but with π for the upper bound

lemma *cos-total*:
assumes $y: -1 \leq y \leq 1$
shows $\exists!x. 0 \leq x \wedge x \leq \pi \wedge \cos x = y$
proof (*rule ex-ex1I*)
show $\exists x::\text{real}. 0 \leq x \wedge x \leq \pi \wedge \cos x = y$
by (*rule IVT2*) (*simp-all add: y*)
next
fix $a \ b :: \text{real}$
assume $ab: 0 \leq a \wedge a \leq \pi \wedge \cos a = y \ 0 \leq b \wedge b \leq \pi \wedge \cos b = y$
have *cosd*: $\bigwedge x::\text{real}. \cos$ differentiable (at x)
unfolding *real-differentiable-def* **by** (*auto intro: DERIV-cos*)
show $a = b$
proof (*cases a b rule: linorder-cases*)
case less
then obtain z **where** $a < z < b$ (*cos has-real-derivative 0*) (at z)
using *Rolle* **by** (*metis cosd continuous-on-cos-real ab*)
then have $\sin z = 0$
using *DERIV-cos DERIV-unique neg-equal-0-iff-equal* **by** *blast*
then show *?thesis*
by (*metis <a < z> <z < b> ab order-less-le-trans less-le sin-gt-zero*)
next
case greater
then obtain z **where** $b < z < a$ (*cos has-real-derivative 0*) (at z)
using *Rolle* **by** (*metis cosd continuous-on-cos-real ab*)
then have $\sin z = 0$
using *DERIV-cos DERIV-unique neg-equal-0-iff-equal* **by** *blast*
then show *?thesis*
by (*metis <b < z> <z < a> ab order-less-le-trans less-le sin-gt-zero*)
qed *auto*
qed

lemma *sin-total*:

assumes $y: -1 \leq y \wedge y \leq 1$
shows $\exists!x. -(\pi/2) \leq x \wedge x \leq \pi/2 \wedge \sin x = y$
proof –
from *cos-total* [*OF* y]
obtain x **where** $x: 0 \leq x \wedge x \leq \pi \wedge \cos x = y$
and *uniq*: $\bigwedge x'. 0 \leq x' \implies x' \leq \pi \implies \cos x' = y \implies x' = x$
by *blast*
show *?thesis*
unfolding *sin-cos-eq*
proof (*rule ex1I* [**where** $a = \pi/2 - x$])
show $-(\pi/2) \leq z \wedge z \leq \pi/2 \wedge \cos(\text{of-real } \pi/2 - z) = y \implies$
 $z = \pi/2 - x$ **for** z
using *uniq* [*of* $\pi/2 - z$] **by** *auto*
qed (*use* x **in** *auto*)
qed

lemma *cos-zero-lemma*:

assumes $0 \leq x \wedge \cos x = 0$
shows $\exists n. \text{odd } n \wedge x = \text{of-nat } n * (\pi/2)$
proof –
have $xle: x < (1 + \text{real-of-int } \lfloor x/\pi \rfloor) * \pi$
using *floor-correct* [*of* x/π]
by (*simp add: add.commute divide-less-eq*)
obtain n **where** $\text{real } n * \pi \leq x < \text{real } (\text{Suc } n) * \pi$
proof
show $\text{real } (\text{nat } \lfloor x / \pi \rfloor) * \pi \leq x$
using *assms floor-divide-lower* [*of* π x] **by** *auto*
show $x < \text{real } (\text{Suc } (\text{nat } \lfloor x / \pi \rfloor)) * \pi$
using *assms floor-divide-upper* [*of* π x] **by** (*simp add: xle*)
qed
then have $x: 0 \leq x - n * \pi \wedge (x - n * \pi) \leq \pi \wedge \cos(x - n * \pi) = 0$
by (*auto simp: algebra-simps cos-diff assms*)
then have $\exists!x. 0 \leq x \wedge x \leq \pi \wedge \cos x = 0$
by (*auto simp: intro!: cos-total*)
then obtain ϑ **where** $\vartheta: 0 \leq \vartheta \wedge \vartheta \leq \pi \wedge \cos \vartheta = 0$
and *uniq*: $\bigwedge \varphi. 0 \leq \varphi \implies \varphi \leq \pi \implies \cos \varphi = 0 \implies \varphi = \vartheta$
by *blast*
then have $x - \text{real } n * \pi = \vartheta$
using x **by** *blast*
moreover have $\pi/2 = \vartheta$
using *pi-half-ge-zero uniq* **by** *fastforce*
ultimately show *?thesis*
by (*rule-tac* $x = \text{Suc } (2 * n)$ **in** *exI*) (*simp add: algebra-simps*)
qed

lemma *sin-zero-lemma*:

assumes $0 \leq x \wedge \sin x = 0$
shows $\exists n::\text{nat. even } n \wedge x = \text{real } n * (\pi/2)$
proof –

obtain n **where** $odd\ n$ **and** $n: x + pi/2 = of\text{-nat}\ n * (pi/2)$ $n > 0$
using $cos\text{-zero}\text{-lemma}$ [of $x + pi/2$] $assms$ **by** ($auto\ simp\ add: cos\text{-add}$)
then have $x = real\ (n - 1) * (pi/2)$
by ($simp\ add: algebra\text{-simps}\ of\text{-nat}\text{-diff}$)
then show $?thesis$
by ($simp\ add: \langle odd\ n \rangle$)
qed

lemma $cos\text{-zero}\text{-iff}$:
 $cos\ x = 0 \longleftrightarrow ((\exists n. odd\ n \wedge x = real\ n * (pi/2)) \vee (\exists n. odd\ n \wedge x = - (real\ n * (pi/2))))$
(is $?lhs = ?rhs$ **)**
proof –
have $*$: $cos\ (real\ n * pi/2) = 0$ **if** $odd\ n$ **for** $n :: nat$
proof –
from $that$ **obtain** m **where** $n = 2 * m + 1 ..$
then show $?thesis$
by ($simp\ add: field\text{-simps}$) ($simp\ add: cos\text{-add}\ add\text{-divide}\text{-distrib}$)
qed
show $?thesis$
proof
show $?rhs$ **if** $?lhs$
using $that\ cos\text{-zero}\text{-lemma}$ [of x] $cos\text{-zero}\text{-lemma}$ [of $-x$] **by** $force$
show $?lhs$ **if** $?rhs$
using $that$ **by** ($auto\ dest: * simp\ del: eq\text{-divide}\text{-eq}\text{-numeral}1$)
qed
qed

lemma $sin\text{-zero}\text{-iff}$:
 $sin\ x = 0 \longleftrightarrow ((\exists n. even\ n \wedge x = real\ n * (pi/2)) \vee (\exists n. even\ n \wedge x = - (real\ n * (pi/2))))$
(is $?lhs = ?rhs$ **)**
proof
show $?rhs$ **if** $?lhs$
using $that\ sin\text{-zero}\text{-lemma}$ [of x] $sin\text{-zero}\text{-lemma}$ [of $-x$] **by** $force$
show $?lhs$ **if** $?rhs$
using $that$ **by** ($auto\ elim: evenE$)
qed

lemma $sin\text{-zero}\text{-pi}\text{-iff}$:
fixes $x::real$
assumes $|x| < pi$
shows $sin\ x = 0 \longleftrightarrow x = 0$
proof
show $x = 0$ **if** $sin\ x = 0$
using $that\ assms$ **by** ($auto\ simp: sin\text{-zero}\text{-iff}$)
qed $auto$

lemma $cos\text{-zero}\text{-iff}\text{-int}$: $cos\ x = 0 \longleftrightarrow (\exists i. odd\ i \wedge x = of\text{-int}\ i * (pi/2))$

proof –
have 1: $\bigwedge n. \text{ odd } n \implies \exists i. \text{ odd } i \wedge \text{ real } n = \text{real-of-int } i$
by (*metis even-of-nat-iff of-int-of-nat-eq*)
have 2: $\bigwedge n. \text{ odd } n \implies \exists i. \text{ odd } i \wedge -(\text{real } n * \text{pi}) = \text{real-of-int } i * \text{pi}$
by (*metis even-minus even-of-nat-iff mult.commute mult-minus-right of-int-minus of-int-of-nat-eq*)
have 3: $\llbracket \text{ odd } i; \forall n. \text{ even } n \vee \text{ real-of-int } i \neq -(\text{real } n) \rrbracket$
 $\implies \exists n. \text{ odd } n \wedge \text{ real-of-int } i = \text{real } n$ **for** i
by (*cases i rule: int-cases2*) *auto*
show *?thesis*
by (*force simp: cos-zero-iff intro!: 1 2 3*)
qed

lemma *sin-zero-iff-int*: $\sin x = 0 \longleftrightarrow (\exists i. \text{ even } i \wedge x = \text{of-int } i * (\text{pi}/2))$ (**is** *?lhs* = *?rhs*)

proof *safe*
assume *?lhs*
then consider (*plus*) n **where** $\text{ even } n \wedge x = \text{real } n * (\text{pi}/2)$ | (*minus*) n **where** $\text{ even } n \wedge x = -(\text{real } n * (\text{pi}/2))$
using *sin-zero-iff* **by** *auto*
then show $\exists n. \text{ even } n \wedge x = \text{of-int } n * (\text{pi}/2)$
proof *cases*
case *plus*
then show *?rhs*
by (*metis even-of-nat-iff of-int-of-nat-eq*)
next
case *minus*
then show *?thesis*
by (*rule-tac x=- (int n) in exI*) *simp*
qed
next
fix $i :: \text{int}$
assume $\text{ even } i$
then show $\sin (\text{of-int } i * (\text{pi}/2)) = 0$
by (*cases i rule: int-cases2, simp-all add: sin-zero-iff*)
qed

lemma *sin-zero-iff-int2*: $\sin x = 0 \longleftrightarrow (\exists i :: \text{int}. x = \text{of-int } i * \text{pi})$

proof –
have $\sin x = 0 \longleftrightarrow (\exists i. \text{ even } i \wedge x = \text{real-of-int } i * (\text{pi}/2))$
by (*auto simp: sin-zero-iff-int*)
also have $\dots = (\exists j. x = \text{real-of-int } (2*j) * (\text{pi}/2))$
using *dvd-triv-left* **by** *blast*
also have $\dots = (\exists i :: \text{int}. x = \text{of-int } i * \text{pi})$
by *auto*
finally show *?thesis* .
qed

lemma *cos-zero-iff-int2*:


```

fixes  $x::\text{real}$ 
shows  $\cos x = 0 \iff (\exists n::\text{int}. x = n * \text{pi} + \text{pi}/2)$ 
using sin-zero-iff-int2[of  $x-\text{pi}/2$ ] unfolding sin-cos-eq
by (auto simp add: algebra-simps)

lemma sin-npi-int [simp]:  $\sin (\text{pi} * \text{of-int } n) = 0$ 
by (simp add: sin-zero-iff-int2)

lemma cos-monotone-0-pi:
assumes  $0 \leq y$  and  $y < x$  and  $x \leq \text{pi}$ 
shows  $\cos x < \cos y$ 
proof –
  have  $-(x - y) < 0$  using assms by auto
  from MVT2[OF  $\langle y < x \rangle$  DERIV-cos]
  obtain  $z$  where  $y < z$  and  $z < x$  and cos-diff:  $\cos x - \cos y = (x - y) * -$ 
   $\sin z$ 
  by auto
  then have  $0 < z$  and  $z < \text{pi}$ 
  using assms by auto
  then have  $0 < \sin z$ 
  using sin-gt-zero by auto
  then have  $\cos x - \cos y < 0$ 
  unfolding cos-diff minus-mult-commute[symmetric]
  using  $\langle -(x - y) < 0 \rangle$  by (rule mult-pos-neg2)
  then show ?thesis by auto
qed

lemma cos-monotone-0-pi-le:
assumes  $0 \leq y$  and  $y \leq x$  and  $x \leq \text{pi}$ 
shows  $\cos x \leq \cos y$ 
proof (cases  $y < x$ )
  case True
  show ?thesis
  using cos-monotone-0-pi[OF  $\langle 0 \leq y \rangle$  True  $\langle x \leq \text{pi} \rangle$ ] by auto
next
  case False
  then have  $y = x$  using  $\langle y \leq x \rangle$  by auto
  then show ?thesis by auto
qed

lemma cos-monotone-minus-pi-0:
assumes  $-\text{pi} \leq y$  and  $y < x$  and  $x \leq 0$ 
shows  $\cos y < \cos x$ 
proof –
  have  $0 \leq -x$  and  $-x < -y$  and  $-y \leq \text{pi}$ 
  using assms by auto
  from cos-monotone-0-pi[OF this] show ?thesis
  unfolding cos-minus .
qed

```

lemma *cos-monotone-minus-pi-0'*:

assumes $- \pi \leq y$ **and** $y \leq x$ **and** $x \leq 0$

shows $\cos y \leq \cos x$

proof (*cases* $y < x$)

case *True*

show *?thesis* **using** *cos-monotone-minus-pi-0* [*OF* $\langle -\pi \leq y \rangle$ *True* $\langle x \leq 0 \rangle$]

by *auto*

next

case *False*

then have $y = x$ **using** $\langle y \leq x \rangle$ **by** *auto*

then show *?thesis* **by** *auto*

qed

lemma *sin-monotone-2pi*:

assumes $-(\pi/2) \leq y$ **and** $y < x$ **and** $x \leq \pi/2$

shows $\sin y < \sin x$

unfolding *sin-cos-eq*

using *assms* **by** (*auto intro: cos-monotone-0-pi*)

lemma *sin-monotone-2pi-le*:

assumes $-(\pi/2) \leq y$ **and** $y \leq x$ **and** $x \leq \pi/2$

shows $\sin y \leq \sin x$

by (*metis assms le-less sin-monotone-2pi*)

lemma *sin-x-le-x*:

fixes $x :: \text{real}$

assumes $x \geq 0$

shows $\sin x \leq x$

proof –

let $?f = \lambda x. x - \sin x$

have $\bigwedge u. [0 \leq u; u \leq x] \implies \exists y. (?f \text{ has-real-derivative } 1 - \cos u) (at\ u)$

by (*auto intro!: derivative-eq-intros simp: field-simps*)

then have $?f\ x \geq ?f\ 0$

by (*metis cos-le-one diff-ge-0-iff-ge DERIV-nonneg-imp-nondecreasing* [*OF assms*])

then show $\sin x \leq x$ **by** *simp*

qed

lemma *sin-x-ge-neg-x*:

fixes $x :: \text{real}$

assumes $x: x \geq 0$

shows $\sin x \geq -x$

proof –

let $?f = \lambda x. x + \sin x$

have $\S: \bigwedge u. [0 \leq u; u \leq x] \implies \exists y. (?f \text{ has-real-derivative } 1 + \cos u) (at\ u)$

by (*auto intro!: derivative-eq-intros simp: field-simps*)

have $?f\ x \geq ?f\ 0$

by (*rule DERIV-nonneg-imp-nondecreasing* [*OF assms*]) (*use* \S *real-0-le-add-iff*)

in force)
then show $\sin x \geq -x$ **by** *simp*
qed

lemma *abs-sin-x-le-abs-x*: $|\sin x| \leq |x|$
for $x :: \text{real}$
using *sin-x-ge-neg-x* [of x] *sin-x-le-x* [of x] *sin-x-ge-neg-x* [of $-x$] *sin-x-le-x* [of $-x$]
by (*auto simp: abs-real-def*)

113.14 More Corollaries about Sine and Cosine

lemma *sin-cos-npi* [*simp*]: $\sin (\text{real } (\text{Suc } (2 * n)) * \text{pi}/2) = (-1) ^ n$
proof –
have $\sin ((\text{real } n + 1/2) * \text{pi}) = \cos (\text{real } n * \text{pi})$
by (*auto simp: algebra-simps sin-add*)
then show *?thesis*
by (*simp add: distrib-right add-divide-distrib add.commute mult.commute* [of pi])
qed

lemma *cos-2npi* [*simp*]: $\cos (2 * \text{real } n * \text{pi}) = 1$
for $n :: \text{nat}$
by (*cases even n*) (*simp-all add: cos-double mult.assoc*)

lemma *cos-3over2-pi* [*simp*]: $\cos (3/2 * \text{pi}) = 0$
proof –
have $\cos (3/2 * \text{pi}) = \cos (\text{pi} + \text{pi}/2)$
by *simp*
also have $\dots = 0$
by (*subst cos-add, simp*)
finally show *?thesis* .
qed

lemma *sin-2npi* [*simp*]: $\sin (2 * \text{real } n * \text{pi}) = 0$
for $n :: \text{nat}$
by (*auto simp: mult.assoc sin-double*)

lemma *sin-3over2-pi* [*simp*]: $\sin (3/2 * \text{pi}) = -1$
proof –
have $\sin (3/2 * \text{pi}) = \sin (\text{pi} + \text{pi}/2)$
by *simp*
also have $\dots = -1$
by (*subst sin-add, simp*)
finally show *?thesis* .
qed

lemma *cos-pi-eq-zero* [*simp*]: $\cos (\text{pi} * \text{real } (\text{Suc } (2 * m)) / 2) = 0$
by (*simp only: cos-add sin-add of-nat-Suc distrib-right distrib-left add-divide-distrib,*

auto)

lemma *DERIV-cos-add* [*simp*]: *DERIV* ($\lambda x. \cos (x + k)$) *xa* :> $-\sin (xa + k)$
by (*auto intro!*: *derivative-eq-intros*)

lemma *sin-zero-norm-cos-one*:
fixes *x* :: 'a::{*real-normed-field*,*banach*}
assumes *sin x = 0*
shows *norm (cos x) = 1*
using *sin-cos-squared-add* [*of x, unfolded assms*]
by (*simp add: square-norm-one*)

lemma *sin-zero-abs-cos-one*: *sin x = 0* $\implies |\cos x| = (1::\text{real})$
using *sin-zero-norm-cos-one* **by** *fastforce*

lemma *cos-one-sin-zero*:
fixes *x* :: 'a::{*real-normed-field*,*banach*}
assumes *cos x = 1*
shows *sin x = 0*
using *sin-cos-squared-add* [*of x, unfolded assms*]
by *simp*

lemma *sin-times-pi-eq-0*: *sin (x * pi) = 0* $\longleftrightarrow x \in \mathbb{Z}$
by (*simp add: sin-zero-iff-int2*) (*metis Ints-cases Ints-of-int*)

lemma *cos-one-2pi*: *cos x = 1* $\longleftrightarrow (\exists n::\text{nat}. x = n * 2 * \text{pi}) \vee (\exists n::\text{nat}. x = -$
 $(n * 2 * \text{pi}))$
(is ?lhs = ?rhs)

proof

assume *?lhs*
then have *sin x = 0*
by (*simp add: cos-one-sin-zero*)
then show *?rhs*
proof (*simp only: sin-zero-iff, elim exE disjE conjE*)
fix *n* :: *nat*
assume *n: even n x = real n * (pi/2)*
then obtain *m* **where** *m: n = 2 * m*
using *dvdE* **by** *blast*
then have *me: even m* **using** $\langle ?lhs \rangle$ *n*
by (*auto simp: field-simps*) (*metis one-neq-neg-one power-minus-odd power-one*)
show *?rhs*
using *m me n*
by (*auto simp: field-simps elim!: evenE*)

next

fix *n* :: *nat*
assume *n: even n x = - (real n * (pi/2))*
then obtain *m* **where** *m: n = 2 * m*
using *dvdE* **by** *blast*
then have *me: even m* **using** $\langle ?lhs \rangle$ *n*

```

  by (auto simp: field-simps) (metis one-neq-neg-one power-minus-odd power-one)
  show ?rhs
    using m me n
    by (auto simp: field-simps elim!: evenE)
qed
next
  assume ?rhs
  then show  $\cos x = 1$ 
    by (metis cos-2npi cos-minus mult.assoc mult.left-commute)
qed

```

```

lemma cos-one-2pi-int:  $\cos x = 1 \iff (\exists n::\text{int}. x = n * 2 * \pi)$  (is ?lhs = ?rhs)
proof
  assume  $\cos x = 1$ 
  then show ?rhs
    by (metis cos-one-2pi mult.commute mult-minus-right of-int-minus of-int-of-nat-eq)
next
  assume ?rhs
  then show  $\cos x = 1$ 
    by (clarsimp simp add: cos-one-2pi) (metis mult-minus-right of-int-of-nat)
qed

```

```

lemma cos-npi-int [simp]:
  fixes  $n::\text{int}$  shows  $\cos(\pi * \text{of-int } n) = (\text{if even } n \text{ then } 1 \text{ else } -1)$ 
  by (auto simp: algebra-simps cos-one-2pi-int elim!: oddE evenE)

```

```

lemma sin-cos-sqrt:  $0 \leq \sin x \implies \sin x = \text{sqrt}(1 - (\cos(x) ^ 2))$ 
  using sin-squared-eq real-sqrt-unique by fastforce

```

```

lemma sin-eq-0-pi:  $-\pi < x \implies x < \pi \implies \sin x = 0 \implies x = 0$ 
  by (metis sin-gt-zero sin-minus minus-less-iff neg-0-less-iff-less not-less-iff-gr-or-eq)

```

```

lemma cos-treble-cos:  $\cos(3 * x) = 4 * \cos x ^ 3 - 3 * \cos x$ 
  for  $x :: 'a::\{\text{real-normed-field}, \text{banach}\}$ 
proof -
  have *:  $(\sin x * (\sin x * 3)) = 3 - (\cos x * (\cos x * 3))$ 
    by (simp add: mult.assoc [symmetric] sin-squared-eq [unfolded power2-eq-square])
  have  $\cos(3 * x) = \cos(2*x + x)$ 
    by simp
  also have  $\dots = 4 * \cos x ^ 3 - 3 * \cos x$ 
    unfolding cos-add cos-double sin-double
    by (simp add: * field-simps power2-eq-square power3-eq-cube)
  finally show ?thesis .
qed

```

```

lemma cos-45:  $\cos(\pi/4) = \text{sqrt } 2 / 2$ 
proof -
  let ?c =  $\cos(\pi/4)$ 
  let ?s =  $\sin(\pi/4)$ 

```

```

have nonneg:  $0 \leq ?c$ 
  by (simp add: cos-ge-zero)
have  $0 = \cos (\pi/4 + \pi/4)$ 
  by simp
also have  $\cos (\pi/4 + \pi/4) = ?c^2 - ?s^2$ 
  by (simp only: cos-add power2-eq-square)
also have  $\dots = 2 * ?c^2 - 1$ 
  by (simp add: sin-squared-eq)
finally have  $?c^2 = (\text{sqrt } 2 / 2)^2$ 
  by (simp add: power-divide)
then show ?thesis
  using nonneg by (rule power2-eq-imp-eq) simp
qed

```

```

lemma cos-30:  $\cos (\pi/6) = \text{sqrt } 3/2$ 
proof –
  let  $?c = \cos (\pi/6)$ 
  let  $?s = \sin (\pi/6)$ 
  have pos-c:  $0 < ?c$ 
    by (rule cos-gt-zero) simp-all
  have  $0 = \cos (\pi/6 + \pi/6 + \pi/6)$ 
    by simp
  also have  $\dots = (?c * ?c - ?s * ?s) * ?c - (?s * ?c + ?c * ?s) * ?s$ 
    by (simp only: cos-add sin-add)
  also have  $\dots = ?c * (?c^2 - 3 * ?s^2)$ 
    by (simp add: algebra-simps power2-eq-square)
  finally have  $?c^2 = (\text{sqrt } 3/2)^2$ 
    using pos-c by (simp add: sin-squared-eq power-divide)
  then show ?thesis
    using pos-c [THEN order-less-imp-le]
    by (rule power2-eq-imp-eq) simp
qed

```

```

lemma sin-45:  $\sin (\pi/4) = \text{sqrt } 2 / 2$ 
  by (simp add: sin-cos-eq cos-45)

```

```

lemma sin-60:  $\sin (\pi/3) = \text{sqrt } 3/2$ 
  by (simp add: sin-cos-eq cos-30)

```

```

lemma cos-60:  $\cos (\pi/3) = 1/2$ 
proof –
  have  $0 \leq \cos (\pi/3)$ 
    by (rule cos-ge-zero) (use pi-half-ge-zero in <linarith+>)
  then show ?thesis
    by (simp add: cos-squared-eq sin-60 power-divide power2-eq-imp-eq)
qed

```

```

lemma sin-30:  $\sin (\pi/6) = 1/2$ 
  by (simp add: sin-cos-eq cos-60)

```

lemma *cos-120*: $\cos (2 * \pi / 3) = -1 / 2$
and *sin-120*: $\sin (2 * \pi / 3) = \text{sqrt } 3 / 2$
using *sin-double*[of $\pi / 3$] *cos-double*[of $\pi / 3$]
by (*simp-all add: power2-eq-square sin-60 cos-60*)

lemma *cos-120'*: $\cos (\pi * 2 / 3) = -1 / 2$
using *cos-120* **by** (*subst mult.commute*)

lemma *sin-120'*: $\sin (\pi * 2 / 3) = \text{sqrt } 3 / 2$
using *sin-120* **by** (*subst mult.commute*)

lemma *cos-integer-2pi*: $n \in \mathbf{Z} \implies \cos(2 * \pi * n) = 1$
by (*metis Ints-cases cos-one-2pi-int mult.assoc mult.commute*)

lemma *sin-integer-2pi*: $n \in \mathbf{Z} \implies \sin(2 * \pi * n) = 0$
by (*metis sin-two-pi Ints-mult mult.assoc mult.commute sin-times-pi-eq-0*)

lemma *cos-int-2pin* [*simp*]: $\cos ((2 * \pi) * \text{of-int } n) = 1$
by (*simp add: cos-one-2pi-int*)

lemma *sin-int-2pin* [*simp*]: $\sin ((2 * \pi) * \text{of-int } n) = 0$
by (*metis Ints-of-int sin-integer-2pi*)

lemma *sincos-principal-value*: $\exists y. (-\pi < y \wedge y \leq \pi) \wedge (\sin y = \sin x \wedge \cos y = \cos x)$
proof –
define *y* **where** $y \equiv \pi - (2 * \pi) * \text{frac } ((\pi - x) / (2 * \pi))$
have $-\pi < y \wedge y \leq \pi$
by (*auto simp: field-simps frac-lt-1 y-def*)
moreover
have $\sin y = \sin x \wedge \cos y = \cos x$
unfolding *y-def*
apply (*simp-all add: frac-def divide-simps sin-add cos-add*)
by (*metis sin-int-2pin cos-int-2pin diff-zero add.right-neutral mult.commute mult.left-neutral mult-zero-left*)
ultimately
show *?thesis* **by** *metis*
qed

113.15 Tangent

definition *tan* :: $'a \Rightarrow 'a::\{\text{real-normed-field,banach}\}$
where $\text{tan} = (\lambda x. \sin x / \cos x)$

lemma *tan-of-real*: $\text{of-real } (\text{tan } x) = (\text{tan } (\text{of-real } x)) :: 'a::\{\text{real-normed-field,banach}\}$
by (*simp add: tan-def sin-of-real cos-of-real*)

lemma *tan-in-Reals* [*simp*]: $z \in \mathbf{R} \implies \text{tan } z \in \mathbf{R}$

for $z :: 'a::\{\text{real-normed-field},\text{banach}\}$
by (*simp add: tan-def*)

lemma *tan-zero* [*simp*]: $\tan 0 = 0$
by (*simp add: tan-def*)

lemma *tan-pi* [*simp*]: $\tan \pi = 0$
by (*simp add: tan-def*)

lemma *tan-npi* [*simp*]: $\tan (\text{real } n * \pi) = 0$
for $n :: \text{nat}$
by (*simp add: tan-def*)

lemma *tan-pi-half* [*simp*]: $\tan (\pi / 2) = 0$
by (*simp add: tan-def*)

lemma *tan-minus* [*simp*]: $\tan (-x) = -\tan x$
by (*simp add: tan-def*)

lemma *tan-periodic* [*simp*]: $\tan (x + 2 * \pi) = \tan x$
by (*simp add: tan-def*)

lemma *lemma-tan-add1*: $\cos x \neq 0 \implies \cos y \neq 0 \implies 1 - \tan x * \tan y = \cos (x + y) / (\cos x * \cos y)$
by (*simp add: tan-def cos-add field-simps*)

lemma *add-tan-eq*: $\cos x \neq 0 \implies \cos y \neq 0 \implies \tan x + \tan y = \sin(x + y) / (\cos x * \cos y)$
for $x :: 'a::\{\text{real-normed-field},\text{banach}\}$
by (*simp add: tan-def sin-add field-simps*)

lemma *tan-eq-0-cos-sin*: $\tan x = 0 \iff \cos x = 0 \vee \sin x = 0$
by (*auto simp: tan-def*)

Note: half of these zeros would normally be regarded as undefined cases.

lemma *tan-eq-0-Ex*:
assumes $\tan x = 0$
obtains $k::\text{int}$ **where** $x = (k/2) * \pi$
using *assms*
by (*metis cos-zero-iff-int mult.commute sin-zero-iff-int tan-eq-0-cos-sin times-divide-eq-left*)

lemma *tan-add*:
 $\cos x \neq 0 \implies \cos y \neq 0 \implies \cos (x + y) \neq 0 \implies \tan (x + y) = (\tan x + \tan y) / (1 - \tan x * \tan y)$
for $x :: 'a::\{\text{real-normed-field},\text{banach}\}$
by (*simp add: add-tan-eq lemma-tan-add1 field-simps*) (*simp add: tan-def*)

lemma *tan-double*: $\cos x \neq 0 \implies \cos (2 * x) \neq 0 \implies \tan (2 * x) = (2 * \tan x)$


```

/ (1 - (tan x)2)
  for x :: 'a::{real-normed-field,banach}
  using tan-add [of x x] by (simp add: power2-eq-square)

lemma tan-gt-zero: 0 < x  $\implies$  x < pi/2  $\implies$  0 < tan x
  by (simp add: tan-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi)

lemma tan-less-zero:
  assumes - pi/2 < x and x < 0
  shows tan x < 0
proof -
  have 0 < tan (- x)
    using assms by (simp only: tan-gt-zero)
  then show ?thesis by simp
qed

lemma tan-half: tan x = sin (2 * x) / (cos (2 * x) + 1)
  for x :: 'a::{real-normed-field,banach,field}
  unfolding tan-def sin-double cos-double sin-squared-eq
  by (simp add: power2-eq-square)

lemma tan-30: tan (pi/6) = 1 / sqrt 3
  unfolding tan-def by (simp add: sin-30 cos-30)

lemma tan-45: tan (pi/4) = 1
  unfolding tan-def by (simp add: sin-45 cos-45)

lemma tan-60: tan (pi/3) = sqrt 3
  unfolding tan-def by (simp add: sin-60 cos-60)

lemma DERIV-tan [simp]: cos x  $\neq$  0  $\implies$  DERIV tan x :=> inverse ((cos x)2)
  for x :: 'a::{real-normed-field,banach}
  unfolding tan-def
  by (auto intro!: derivative-eq-intros, simp add: divide-inverse power2-eq-square)

declare DERIV-tan[THEN DERIV-chain2, derivative-intros]
  and DERIV-tan[THEN DERIV-chain2, unfolded has-field-derivative-def, deriva-
  tive-intros]

lemmas has-derivative-tan[derivative-intros] = DERIV-tan[THEN DERIV-compose-FDERIV]

lemma isCont-tan: cos x  $\neq$  0  $\implies$  isCont tan x
  for x :: 'a::{real-normed-field,banach}
  by (rule DERIV-tan [THEN DERIV-isCont])

lemma isCont-tan' [simp,continuous-intros]:
  fixes a :: 'a::{real-normed-field,banach} and f :: 'a  $\Rightarrow$  'a
  shows isCont f a  $\implies$  cos (f a)  $\neq$  0  $\implies$  isCont ( $\lambda x$ . tan (f x)) a
  by (rule isCont-o2 [OF - isCont-tan])

```

lemma *tendsto-tan* [*tendsto-intros*]:

fixes $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$

shows $(f \longrightarrow a) F \Longrightarrow \cos a \neq 0 \Longrightarrow ((\lambda x. \tan (f x)) \longrightarrow \tan a) F$

by (*rule isCont-tendsto-compose* [*OF isCont-tan*])

lemma *continuous-tan*:

fixes $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$

shows $\text{continuous } F f \Longrightarrow \cos (f (\text{Lim } F (\lambda x. x))) \neq 0 \Longrightarrow \text{continuous } F (\lambda x. \tan (f x))$

unfolding *continuous-def* **by** (*rule tendsto-tan*)

lemma *continuous-on-tan* [*continuous-intros*]:

fixes $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$

shows $\text{continuous-on } s f \Longrightarrow (\forall x \in s. \cos (f x) \neq 0) \Longrightarrow \text{continuous-on } s (\lambda x. \tan (f x))$

unfolding *continuous-on-def* **by** (*auto intro: tendsto-tan*)

lemma *continuous-within-tan* [*continuous-intros*]:

fixes $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$

shows $\text{continuous (at } x \text{ within } s) f \Longrightarrow$

$\cos (f x) \neq 0 \Longrightarrow \text{continuous (at } x \text{ within } s) (\lambda x. \tan (f x))$

unfolding *continuous-within* **by** (*rule tendsto-tan*)

lemma *LIM-cos-div-sin*: $(\lambda x. \cos(x)/\sin(x)) -\pi/2 \rightarrow 0$

by (*rule tendsto-cong-limit*, (*rule tendsto-intros*)+, *simp-all*)

lemma *lemma-tan-total*:

assumes $0 < y$ **shows** $\exists x. 0 < x \wedge x < \pi/2 \wedge y < \tan x$

proof –

obtain s **where** $0 < s$

and $s: \bigwedge x. [x \neq \pi/2; \text{norm } (x - \pi/2) < s] \Longrightarrow \text{norm } (\cos x / \sin x - 0) < \text{inverse } y$

using *LIM-D* [*OF LIM-cos-div-sin*, *of inverse y*] **that** *assms* **by** *force*

obtain e **where** $e: 0 < e \wedge e < s \wedge e < \pi/2$

using $\langle 0 < s \rangle$ *field-lbound-gt-zero pi-half-gt-zero* **by** *blast*

show *?thesis*

proof (*intro exI conjI*)

have $0 < \sin e \wedge 0 < \cos e$

using e **by** (*auto intro: cos-gt-zero sin-gt-zero2 simp: mult.commute*)

then

show $y < \tan (\pi/2 - e)$

using s [*of pi/2 - e*] e *assms*

by (*simp add: tan-def sin-diff cos-diff*) (*simp add: field-simps split: if-split-asm*)

qed (*use e in auto*)

qed

lemma *tan-total-pos*:

assumes $0 \leq y$ **shows** $\exists x. 0 \leq x \wedge x < \pi/2 \wedge \tan x = y$

```

proof (cases y = 0)
  case True
  then show ?thesis
    using pi-half-gt-zero tan-zero by blast
next
  case False
  with assms have y > 0
    by linarith
  obtain x where x: 0 < x x < pi/2 y < tan x
    using lemma-tan-total ‹0 < y› by blast
  have ∃ u ≥ 0. u ≤ x ∧ tan u = y
  proof (intro IVT allI impI)
    show isCont tan u if 0 ≤ u ∧ u ≤ x for u
    proof –
      have cos u ≠ 0
        using antisym-conv2 cos-gt-zero that x(2) by fastforce
      with assms show ?thesis
        by (auto intro!: DERIV-tan [THEN DERIV-isCont])
    qed
  qed (use assms x in auto)
  then show ?thesis
    using x(2) by auto
qed

lemma lemma-tan-total1: ∃ x. -(pi/2) < x ∧ x < (pi/2) ∧ tan x = y
proof (cases 0::real y rule: le-cases)
  case le
  then show ?thesis
    by (meson less-le-trans minus-pi-half-less-zero tan-total-pos)
next
  case ge
  with tan-total-pos [of -y] obtain x where 0 ≤ x x < pi/2 tan x = - y
    by force
  then show ?thesis
    by (rule-tac x=-x in exI) auto
qed

proposition tan-total: ∃! x. -(pi/2) < x ∧ x < (pi/2) ∧ tan x = y
proof –
  have u = v if u: -(pi/2) < u u < pi/2 and v: -(pi/2) < v v < pi/2
    and eq: tan u = tan v for u v
  proof (cases u v rule: linorder-cases)
  case less
  have ∧x. u ≤ x ∧ x ≤ v → isCont tan x
    by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-le-trans u(1)
v(2))
  then have continuous-on {u..v} tan
    by (simp add: continuous-at-imp-continuous-on)
  moreover have ∧x. u < x ∧ x < v ⇒ tan differentiable (at x)

```

by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3)
 order.strict-trans u(1) v(2))
 ultimately obtain z where $u < z < v$ DERIV tan z $:> 0$
 by (metis less Rolle eq)
 moreover have $\cos z \neq 0$
 by (metis (no-types) $\langle u < z \rangle \langle z < v \rangle$ cos-gt-zero-pi less-le-trans linorder-not-less
 not-less-iff-gr-or-eq u(1) v(2))
 ultimately show ?thesis
 using DERIV-unique [OF - DERIV-tan] by fastforce
 next
 case greater
 have $\bigwedge x. v \leq x \wedge x \leq u \implies \text{isCont tan } x$
 by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-le-trans u(2)
 v(1))
 then have continuous-on $\{v..u\}$ tan
 by (simp add: continuous-at-imp-continuous-on)
 moreover have $\bigwedge x. v < x \wedge x < u \implies \text{tan differentiable (at } x)$
 by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3)
 order.strict-trans u(2) v(1))
 ultimately obtain z where $v < z < u$ DERIV tan z $:> 0$
 by (metis greater Rolle eq)
 moreover have $\cos z \neq 0$
 by (metis $\langle v < z \rangle \langle z < u \rangle$ cos-gt-zero-pi less-eq-real-def less-le-trans or-
 der-less-irrefl u(2) v(1))
 ultimately show ?thesis
 using DERIV-unique [OF - DERIV-tan] by fastforce
 qed auto
 then have $\exists! x. -(pi/2) < x \wedge x < pi/2 \wedge \text{tan } x = y$
 if $x: -(pi/2) < x < pi/2$ tan x = y for x
 using that by auto
 then show ?thesis
 using lemma-tan-total1 [where y = y]
 by auto
 qed

lemma tan-monotone:

assumes $-(pi/2) < y$ and $y < x$ and $x < pi/2$
 shows tan y < tan x

proof -

have DERIV tan x' $:>$ inverse $((\cos x')^2)$ if $y \leq x' \leq x$ for x'

proof -

have $-(pi/2) < x'$ and $x' < pi/2$

using that assms by auto

with cos-gt-zero-pi have $\cos x' \neq 0$ by force

then show DERIV tan x' $:>$ inverse $((\cos x')^2)$

by (rule DERIV-tan)

qed

from MVT2[OF $\langle y < x \rangle$ this]

obtain z where $y < z$ and $z < x$

and *tan-diff*: $\tan x - \tan y = (x - y) * \text{inverse } ((\cos z)^2)$ **by** *auto*
then have $-(\pi/2) < z$ **and** $z < \pi/2$
using *assms* **by** *auto*
then have $0 < \cos z$
using *cos-gt-zero-pi* **by** *auto*
then have *inv-pos*: $0 < \text{inverse } ((\cos z)^2)$
by *auto*
have $0 < x - y$ **using** $\langle y < x \rangle$ **by** *auto*
with *inv-pos* **have** $0 < \tan x - \tan y$
unfolding *tan-diff* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *tan-monotone'*:
assumes $-(\pi/2) < y$
and $y < \pi/2$
and $-(\pi/2) < x$
and $x < \pi/2$
shows $y < x \longleftrightarrow \tan y < \tan x$
proof
assume $y < x$
then show $\tan y < \tan x$
using *tan-monotone* **and** $\langle -(\pi/2) < y \rangle$ **and** $\langle x < \pi/2 \rangle$ **by** *auto*
next
assume $\tan y < \tan x$
show $y < x$
proof (*rule ccontr*)
assume $\neg ?thesis$
then have $x \leq y$ **by** *auto*
then have $\tan x \leq \tan y$
proof (*cases* $x = y$)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have $x < y$ **using** $\langle x \leq y \rangle$ **by** *auto*
from *tan-monotone*[*OF* $\langle -(\pi/2) < x \rangle$ *this* $\langle y < \pi/2 \rangle$] **show** *?thesis*
by *auto*
qed
then show *False*
using $\langle \tan y < \tan x \rangle$ **by** *auto*
qed
qed

lemma *tan-inverse*: $1 / (\tan y) = \tan (\pi/2 - y)$
unfolding *tan-def* *sin-cos-eq*[*of* y] *cos-sin-eq*[*of* y] **by** *auto*

lemma *tan-periodic-pi[simp]*: $\tan (x + \pi) = \tan x$
by (*simp* *add*: *tan-def*)

```

lemma tan-periodic-nat[simp]:  $\tan (x + \text{real } n * \pi) = \tan x$ 
proof (induct n arbitrary: x)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have split-pi-off:  $x + \text{real } (\text{Suc } n) * \pi = (x + \text{real } n * \pi) + \pi$ 
  unfolding Suc-eq-plus1 of-nat-add distrib-right by auto
  show ?case
  unfolding split-pi-off using Suc by auto
qed

lemma tan-periodic-int[simp]:  $\tan (x + \text{of-int } i * \pi) = \tan x$ 
proof (cases 0 ≤ i)
  case False
  then have i-nat:  $\text{of-int } i = - \text{of-int } (\text{nat } (- i))$  by auto
  then show ?thesis
  by (smt (verit, best) mult-minus-left of-int-of-nat-eq tan-periodic-nat)
qed (use zero-le-imp-eq-int in fastforce)

lemma tan-periodic-n[simp]:  $\tan (x + \text{numeral } n * \pi) = \tan x$ 
  using tan-periodic-int[of - numeral n] by simp

lemma tan-minus-45 [simp]:  $\tan (-(\pi/4)) = -1$ 
  unfolding tan-def by (simp add: sin-45 cos-45)

lemma tan-diff:
   $\cos x \neq 0 \implies \cos y \neq 0 \implies \cos (x - y) \neq 0 \implies \tan (x - y) = (\tan x - \tan y) / (1 + \tan x * \tan y)$ 
  for  $x :: 'a :: \{\text{real-normed-field, banach}\}$ 
  using tan-add [of x - y] by simp

lemma tan-pos-pi2-le:  $0 \leq x \implies x < \pi/2 \implies 0 \leq \tan x$ 
  using less-eq-real-def tan-gt-zero by auto

lemma cos-tan:  $|x| < \pi/2 \implies \cos x = 1 / \text{sqrt } (1 + \tan x ^ 2)$ 
  using cos-gt-zero-pi [of x]
  by (simp add: field-split-simps tan-def real-sqrt-divide abs-if split: if-split-asm)

lemma cos-tan-half:  $\cos x \neq 0 \implies \cos (2*x) = (1 - (\tan x)^2) / (1 + (\tan x)^2)$ 
  unfolding cos-double tan-def by (auto simp add:field-simps)

lemma sin-tan:  $|x| < \pi/2 \implies \sin x = \tan x / \text{sqrt } (1 + \tan x ^ 2)$ 
  using cos-gt-zero [of x] cos-gt-zero [of -x]
  by (force simp: field-split-simps tan-def real-sqrt-divide abs-if split: if-split-asm)

lemma sin-tan-half:  $\sin (2*x) = 2 * \tan x / (1 + (\tan x)^2)$ 

```

unfolding *sin-double tan-def*
by (*cases cos x=0*) (*auto simp add:field-simps power2-eq-square*)

lemma *tan-mono-le*: $-(\pi/2) < x \implies x \leq y \implies y < \pi/2 \implies \tan x \leq \tan y$
using *less-eq-real-def tan-monotone* **by** *auto*

lemma *tan-mono-lt-eq*:
 $-(\pi/2) < x \implies x < \pi/2 \implies -(\pi/2) < y \implies y < \pi/2 \implies \tan x < \tan y$
 $\longleftrightarrow x < y$
using *tan-monotone'* **by** *blast*

lemma *tan-mono-le-eq*:
 $-(\pi/2) < x \implies x < \pi/2 \implies -(\pi/2) < y \implies y < \pi/2 \implies \tan x \leq \tan y$
 $\longleftrightarrow x \leq y$
by (*meson tan-mono-le not-le tan-monotone*)

lemma *tan-bound-pi2*: $|x| < \pi/4 \implies |\tan x| < 1$
using *tan-45 tan-monotone [of x pi/4] tan-monotone [of -x pi/4]*
by (*auto simp: abs-if split: if-split-asm*)

lemma *tan-cot*: $\tan(\pi/2 - x) = \text{inverse}(\tan x)$
by (*simp add: tan-def sin-diff cos-diff*)

113.16 Cotangent

definition *cot* :: 'a \Rightarrow 'a::{*real-normed-field,banach*}
where *cot* = ($\lambda x. \cos x / \sin x$)

lemma *cot-of-real*: $\text{of-real}(\cot x) = (\cot(\text{of-real } x))$:: 'a::{*real-normed-field,banach*}
by (*simp add: cot-def sin-of-real cos-of-real*)

lemma *cot-in-Reals* [*simp*]: $z \in \mathbf{R} \implies \cot z \in \mathbf{R}$
for z :: 'a::{*real-normed-field,banach*}
by (*simp add: cot-def*)

lemma *cot-zero* [*simp*]: $\cot 0 = 0$
by (*simp add: cot-def*)

lemma *cot-pi* [*simp*]: $\cot \pi = 0$
by (*simp add: cot-def*)

lemma *cot-npi* [*simp*]: $\cot(\text{real } n * \pi) = 0$
for n :: *nat*
by (*simp add: cot-def*)

lemma *cot-minus* [*simp*]: $\cot(-x) = -\cot x$
by (*simp add: cot-def*)

lemma *cot-periodic* [*simp*]: $\cot(x + 2 * \pi) = \cot x$

by (simp add: cot-def)

lemma cot-altdef: $\cot x = \text{inverse } (\tan x)$
by (simp add: cot-def tan-def)

lemma tan-altdef: $\tan x = \text{inverse } (\cot x)$
by (simp add: cot-def tan-def)

lemma tan-cot': $\tan (\pi/2 - x) = \cot x$
by (simp add: tan-cot cot-altdef)

lemma cot-gt-zero: $0 < x \implies x < \pi/2 \implies 0 < \cot x$
by (simp add: cot-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi)

lemma cot-less-zero:
assumes lb: $-\pi/2 < x$ and $x < 0$
shows $\cot x < 0$
by (smt (verit) assms cot-gt-zero cot-minus divide-minus-left)

lemma DERIV-cot [simp]: $\sin x \neq 0 \implies \text{DERIV } \cot x :> -\text{inverse } ((\sin x)^2)$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
unfolding cot-def using cos-squared-eq[of x]
by (auto intro!: derivative-eq-intros) (simp add: divide-inverse power2-eq-square)

lemma isCont-cot: $\sin x \neq 0 \implies \text{isCont } \cot x$
for $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
by (rule DERIV-cot [THEN DERIV-isCont])

lemma isCont-cot' [simp, continuous-intros]:
 $\text{isCont } f a \implies \sin (f a) \neq 0 \implies \text{isCont } (\lambda x. \cot (f x)) a$
for $a :: 'a :: \{\text{real-normed-field}, \text{banach}\}$ and $f :: 'a \Rightarrow 'a$
by (rule isCont-o2 [OF isCont-cot])

lemma tendsto-cot [tendsto-intros]: $(f \longrightarrow a) F \implies \sin a \neq 0 \implies ((\lambda x. \cot (f x)) \longrightarrow \cot a) F$
for $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
by (rule isCont-tendsto-compose [OF isCont-cot])

lemma continuous-cot:
 $\text{continuous } F f \implies \sin (f (\text{Lim } F (\lambda x. x))) \neq 0 \implies \text{continuous } F (\lambda x. \cot (f x))$
for $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
unfolding continuous-def by (rule tendsto-cot)

lemma continuous-on-cot [continuous-intros]:
fixes $f :: 'a \Rightarrow 'a :: \{\text{real-normed-field}, \text{banach}\}$
shows $\text{continuous-on } s f \implies (\forall x \in s. \sin (f x) \neq 0) \implies \text{continuous-on } s (\lambda x. \cot (f x))$
unfolding continuous-on-def by (auto intro: tendsto-cot)

lemma *continuous-within-cot* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows *continuous* (at x within s) $f \Longrightarrow \sin (f x) \neq 0 \Longrightarrow \text{continuous}$ (at x within s) ($\lambda x. \text{cot} (f x)$)
unfolding *continuous-within* **by** (rule *tendsto-cot*)

113.17 Inverse Trigonometric Functions

definition *arcsin* :: *real* \Rightarrow *real*
where $\text{arcsin } y = (\text{THE } x. -(pi/2) \leq x \wedge x \leq pi/2 \wedge \sin x = y)$

definition *arccos* :: *real* \Rightarrow *real*
where $\text{arccos } y = (\text{THE } x. 0 \leq x \wedge x \leq pi \wedge \cos x = y)$

definition *arctan* :: *real* \Rightarrow *real*
where $\text{arctan } y = (\text{THE } x. -(pi/2) < x \wedge x < pi/2 \wedge \tan x = y)$

lemma *arcsin*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y \wedge \text{arcsin } y \leq pi/2$
 $\wedge \sin (\text{arcsin } y) = y$
unfolding *arcsin-def* **by** (rule *theI'* [*OF sin-total*])

lemma *arcsin-pi*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y \wedge \text{arcsin } y \leq pi$
 $\wedge \sin (\text{arcsin } y) = y$
by (*drule* (1) *arcsin*) (*force intro: order-trans*)

lemma *sin-arcsin* [*simp*]: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow \sin (\text{arcsin } y) = y$
by (*blast dest: arcsin*)

lemma *arcsin-bounded*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y \wedge \text{arcsin } y \leq pi/2$
by (*blast dest: arcsin*)

lemma *arcsin-lbound*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin } y$
by (*blast dest: arcsin*)

lemma *arcsin-ubound*: $-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow \text{arcsin } y \leq pi/2$
by (*blast dest: arcsin*)

lemma *arcsin-lt-bounded*:
assumes $-1 < y < 1$
shows $-(pi/2) < \text{arcsin } y \wedge \text{arcsin } y < pi/2$
proof –
have $\text{arcsin } y \neq pi/2$
by (*metis arcsin assms not-less not-less-iff-gr-or-eq sin-pi-half*)
moreover have $\text{arcsin } y \neq -(pi/2)$
by (*metis arcsin assms minus-divide-left not-less not-less-iff-gr-or-eq sin-minus sin-pi-half*)
ultimately show *?thesis*
using *arcsin-bounded* [*of y*] *assms* **by** *auto*

qed

lemma *arcsin-sin*: $-(\pi/2) \leq x \implies x \leq \pi/2 \implies \arcsin (\sin x) = x$
unfolding *arcsin-def*
using *the1-equality [OF sin-total]* **by** *simp*

lemma *arcsin-unique*:
assumes $-\pi/2 \leq x$ **and** $x \leq \pi/2$ **and** $\sin x = y$ **shows** $\arcsin y = x$
using *arcsin-sin[of x]* **assms** **by** *force*

lemma *arcsin-0* [*simp*]: $\arcsin 0 = 0$
using *arcsin-sin [of 0]* **by** *simp*

lemma *arcsin-1* [*simp*]: $\arcsin 1 = \pi/2$
using *arcsin-sin [of pi/2]* **by** *simp*

lemma *arcsin-minus-1* [*simp*]: $\arcsin (-1) = -(\pi/2)$
using *arcsin-sin [of - pi/2]* **by** *simp*

lemma *arcsin-minus*: $-1 \leq x \implies x \leq 1 \implies \arcsin (-x) = -\arcsin x$
by (*metis (no-types, opaque-lifting) arcsin arcsin-sin minus-minus neg-le-iff-le sin-minus*)

lemma *arcsin-one-half* [*simp*]: $\arcsin (1/2) = \pi / 6$
and *arcsin-minus-one-half* [*simp*]: $\arcsin (-(1/2)) = -\pi / 6$
by (*intro arcsin-unique; simp add: sin-30 field-simps*)**+**

lemma *arcsin-one-over-sqrt-2*: $\arcsin (1 / \sqrt{2}) = \pi / 4$
by (*rule arcsin-unique*) (*auto simp: sin-45 field-simps*)

lemma *arcsin-eq-iff*: $|x| \leq 1 \implies |y| \leq 1 \implies \arcsin x = \arcsin y \iff x = y$
by (*metis abs-le-iff arcsin minus-le-iff*)

lemma *cos-arcsin-nonzero*: $-1 < x \implies x < 1 \implies \cos (\arcsin x) \neq 0$
using *arcsin-lt-bounded cos-gt-zero-pi* **by** *force*

lemma *arccos*: $-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y \wedge \arccos y \leq \pi \wedge \cos (\arccos y) = y$
unfolding *arccos-def* **by** (*rule theI' [OF cos-total]*)

lemma *cos-arccos* [*simp*]: $-1 \leq y \implies y \leq 1 \implies \cos (\arccos y) = y$
by (*blast dest: arccos*)

lemma *arccos-bounded*: $-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y \wedge \arccos y \leq \pi$
by (*blast dest: arccos*)

lemma *arccos-lbound*: $-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y$
by (*blast dest: arccos*)

lemma *arccos-ubound*: $-1 \leq y \implies y \leq 1 \implies \arccos y \leq \pi$
 by (*blast dest: arccos*)

lemma *arccos-lt-bounded*:

assumes $-1 < y < 1$

shows $0 < \arccos y \wedge \arccos y < \pi$

proof –

have $\arccos y \neq 0$

by (*metis (no-types) arccos assms(1) assms(2) cos-zero less-eq-real-def less-irrefl*)

moreover have $\arccos y \neq -\pi$

by (*metis arccos assms(1) assms(2) cos-minus cos-pi not-less not-less-iff-gr-or-eq*)

ultimately show *?thesis*

using *arccos-bounded [of y] assms*

by (*metis arccos cos-pi not-less not-less-iff-gr-or-eq*)

qed

lemma *arccos-cos*: $0 \leq x \implies x \leq \pi \implies \arccos (\cos x) = x$

by (*auto simp: arccos-def intro!: the1-equality cos-total*)

lemma *arccos-cos2*: $x \leq 0 \implies -\pi \leq x \implies \arccos (\cos x) = -x$

by (*auto simp: arccos-def intro!: the1-equality cos-total*)

lemma *arccos-unique*:

assumes $0 \leq x$ **and** $x \leq \pi$ **and** $\cos x = y$ **shows** $\arccos y = x$

using *arccos-cos assms* **by** *blast*

lemma *cos-arcsin*:

assumes $-1 \leq x \leq 1$

shows $\cos (\arcsin x) = \sqrt{1 - x^2}$

proof (*rule power2-eq-imp-eq*)

show $(\cos (\arcsin x))^2 = (\sqrt{1 - x^2})^2$

by (*simp add: square-le-1 assms cos-squared-eq*)

show $0 \leq \cos (\arcsin x)$

using *arcsin assms cos-ge-zero* **by** *blast*

show $0 \leq \sqrt{1 - x^2}$

by (*simp add: square-le-1 assms*)

qed

lemma *sin-arccos*:

assumes $-1 \leq x \leq 1$

shows $\sin (\arccos x) = \sqrt{1 - x^2}$

proof (*rule power2-eq-imp-eq*)

show $(\sin (\arccos x))^2 = (\sqrt{1 - x^2})^2$

by (*simp add: square-le-1 assms sin-squared-eq*)

show $0 \leq \sin (\arccos x)$

by (*simp add: arccos-bounded assms sin-ge-zero*)

show $0 \leq \sqrt{1 - x^2}$

by (*simp add: square-le-1 assms*)

qed

lemma *arccos-0* [*simp*]: $\arccos 0 = \pi/2$
using *arccos-cos pi-half-ge-zero* **by** *fastforce*

lemma *arccos-1* [*simp*]: $\arccos 1 = 0$
using *arccos-cos* **by** *force*

lemma *arccos-minus-1* [*simp*]: $\arccos (-1) = \pi$
by (*metis arccos-cos cos-pi order-refl pi-ge-zero*)

lemma *arccos-minus*: $-1 \leq x \implies x \leq 1 \implies \arccos (-x) = \pi - \arccos x$
by (*smt (verit, ccfv-threshold) arccos arccos-cos cos-minus cos-minus-pi*)

lemma *arccos-one-half* [*simp*]: $\arccos (1/2) = \pi / 3$
and *arccos-minus-one-half* [*simp*]: $\arccos (-(1/2)) = 2 * \pi / 3$
by (*intro arccos-unique; simp add: cos-60 cos-120*)**+**

lemma *arccos-one-over-sqrt-2*: $\arccos (1 / \sqrt{2}) = \pi / 4$
by (*rule arccos-unique*) (*auto simp: cos-45 field-simps*)

corollary *arccos-minus-abs*:
assumes $|x| \leq 1$
shows $\arccos (-x) = \pi - \arccos x$
using *assms* **by** (*simp add: arccos-minus*)

lemma *sin-arccos-nonzero*: $-1 < x \implies x < 1 \implies \sin (\arccos x) \neq 0$
using *arccos-lt-bounded sin-gt-zero* **by** *force*

lemma *arctan*: $-(\pi/2) < \arctan y \wedge \arctan y < \pi/2 \wedge \tan (\arctan y) = y$
unfolding *arctan-def* **by** (*rule theI' [OF tan-total]*)

lemma *tan-arctan*: $\tan (\arctan y) = y$
by (*simp add: arctan*)

lemma *arctan-bounded*: $-(\pi/2) < \arctan y \wedge \arctan y < \pi/2$
by (*auto simp only: arctan*)

lemma *arctan-lbound*: $-(\pi/2) < \arctan y$
by (*simp add: arctan*)

lemma *arctan-ubound*: $\arctan y < \pi/2$
by (*auto simp only: arctan*)

lemma *arctan-unique*:
assumes $-(\pi/2) < x$
and $x < \pi/2$
and $\tan x = y$
shows $\arctan y = x$
using *assms arctan [of y] tan-total [of y]* **by** (*fast elim: ex1E*)

```

lemma arctan-tan:  $-(\pi/2) < x \implies x < \pi/2 \implies \arctan (\tan x) = x$ 
  by (rule arctan-unique) simp-all

lemma arctan-zero-zero [simp]:  $\arctan 0 = 0$ 
  by (rule arctan-unique) simp-all

lemma arctan-minus:  $\arctan (-x) = -\arctan x$ 
  using arctan [of x] by (auto simp: arctan-unique)

lemma cos-arctan-not-zero [simp]:  $\cos (\arctan x) \neq 0$ 
  by (intro less-imp-neq [symmetric] cos-gt-zero-pi arctan-lbound arctan-ubound)

lemma tan-eq-arctan-Ex:
  shows  $\tan x = y \iff (\exists k::\text{int}. x = \arctan y + k*\pi \vee (x = \pi/2 + k*\pi \wedge y=0))$ 
  proof
    assume lhs:  $\tan x = y$ 
    obtain  $k::\text{int}$  where  $k: -\pi/2 < x - k*\pi \leq \pi/2$ 
    proof
      define  $k$  where  $k \equiv \text{ceiling} (x/\pi - 1/2)$ 
      show  $-\pi/2 < x - \text{real-of-int } k * \pi$ 
        using ceiling-divide-lower [of  $\pi*2 (x * 2 - \pi)$ ] by (auto simp: k-def field-simps)
      show  $x - k*\pi \leq \pi/2$ 
        using ceiling-divide-upper [of  $\pi*2 (x * 2 - \pi)$ ] by (auto simp: k-def field-simps)
    qed
    have  $x = \arctan y + \text{of-int } k * \pi$  when  $x \neq \pi/2 + k*\pi$ 
    proof -
      have  $\tan (x - k * \pi) = y$  using lhs tan-periodic-int[of - -k] by auto
      then have  $\arctan y = x - \text{real-of-int } k * \pi$ 
        by (smt (verit) arctan-tan lhs divide-minus-left k mult-minus-left of-int-minus tan-periodic-int that)
      then show ?thesis by auto
    qed
    then show  $\exists k. x = \arctan y + \text{of-int } k * \pi \vee (x = \pi/2 + k*\pi \wedge y=0)$ 
      using lhs k by force
  qed (auto simp: arctan)

lemma arctan-tan-eq-abs-pi:
  assumes  $\cos \vartheta \neq 0$ 
  obtains  $k$  where  $\arctan (\tan \vartheta) = \vartheta - \text{of-int } k * \pi$ 
  by (metis add.commute assms cos-zero-iff-int2 eq-diff-eq tan-eq-arctan-Ex)

lemma tan-eq:
  assumes  $\tan x = \tan y \wedge \tan x \neq 0$ 
  obtains  $k::\text{int}$  where  $x = y + k * \pi$ 
  proof -

```

obtain $k0$ **where** $k0: x = \arctan (\tan y) + \text{real-of-int } k0 * \pi$
using *assms tan-eq-arctan-Ex*[of $x \tan y$] **by** *auto*
obtain $k1$ **where** $k1: \arctan (\tan y) = y - \text{of-int } k1 * \pi$
using *arctan-tan-eq-abs-pi* *assms tan-eq-0-cos-sin* **by** *auto*
have $x = y + (k0 - k1) * \pi$
using $k0$ $k1$ **by** (*auto simp: algebra-simps*)
with that show *?thesis*
by *blast*
qed

lemma *cos-arctan*: $\cos (\arctan x) = 1 / \text{sqrt } (1 + x^2)$
proof (*rule power2-eq-imp-eq*)
have $0 < 1 + x^2$ **by** (*simp add: add-pos-nonneg*)
show $0 \leq 1 / \text{sqrt } (1 + x^2)$ **by** *simp*
show $0 \leq \cos (\arctan x)$
by (*intro less-imp-le cos-gt-zero-pi arctan-lbound arctan-ubound*)
have $(\cos (\arctan x))^2 * (1 + (\tan (\arctan x))^2) = 1$
unfolding *tan-def* **by** (*simp add: distrib-left power-divide*)
then show $(\cos (\arctan x))^2 = (1 / \text{sqrt } (1 + x^2))^2$
using $\langle 0 < 1 + x^2 \rangle$ **by** (*simp add: arctan power-divide eq-divide-eq*)
qed

lemma *sin-arctan*: $\sin (\arctan x) = x / \text{sqrt } (1 + x^2)$
using *add-pos-nonneg* [*OF zero-less-one zero-le-power2* [of x]]
using *tan-arctan* [of x] **unfolding** *tan-def cos-arctan*
by (*simp add: eq-divide-eq*)

lemma *tan-sec*: $\cos x \neq 0 \implies 1 + (\tan x)^2 = (\text{inverse } (\cos x))^2$
for $x :: 'a :: \{\text{real-normed-field, banach, field}\}$
by (*simp add: add-divide-eq-iff inverse-eq-divide power2-eq-square tan-def*)

lemma *arctan-less-iff*: $\arctan x < \arctan y \iff x < y$
by (*metis tan-monotone' arctan-lbound arctan-ubound tan-arctan*)

lemma *arctan-le-iff*: $\arctan x \leq \arctan y \iff x \leq y$
by (*simp only: not-less [symmetric] arctan-less-iff*)

lemma *arctan-eq-iff*: $\arctan x = \arctan y \iff x = y$
by (*simp only: eq-iff [where 'a=real] arctan-le-iff*)

lemma *zero-less-arctan-iff* [*simp*]: $0 < \arctan x \iff 0 < x$
using *arctan-less-iff* [of 0 x] **by** *simp*

lemma *arctan-less-zero-iff* [*simp*]: $\arctan x < 0 \iff x < 0$
using *arctan-less-iff* [of x 0] **by** *simp*

lemma *zero-le-arctan-iff* [*simp*]: $0 \leq \arctan x \iff 0 \leq x$
using *arctan-le-iff* [of 0 x] **by** *simp*

lemma *arctan-le-zero-iff* [*simp*]: $\text{arctan } x \leq 0 \longleftrightarrow x \leq 0$
using *arctan-le-iff* [*of x 0*] **by** *simp*

lemma *arctan-eq-zero-iff* [*simp*]: $\text{arctan } x = 0 \longleftrightarrow x = 0$
using *arctan-eq-iff* [*of x 0*] **by** *simp*

lemma *continuous-on-arcsin'*: *continuous-on* $\{-1 .. 1\}$ *arcsin*

proof –

have *continuous-on* (*sin* ‘ $\{- \text{pi}/2 .. \text{pi}/2\}$) *arcsin*
by (*rule continuous-on-inv*) (*auto intro: continuous-intros simp: arcsin-sin*)

also have *sin* ‘ $\{- \text{pi}/2 .. \text{pi}/2\} = \{-1 .. 1\}$

proof safe

fix $x :: \text{real}$

assume $x \in \{-1..1\}$

then show $x \in \text{sin} \text{ ‘ } \{- \text{pi}/2.. \text{pi}/2\}$

using *arcsin-lbound arcsin-ubound*

by (*intro image-eqI*[**where** $x = \text{arcsin } x$]) *auto*

qed *simp*

finally show *?thesis* .

qed

lemma *continuous-on-arcsin* [*continuous-intros*]:

continuous-on s f $\implies (\forall x \in s. -1 \leq f x \wedge f x \leq 1) \implies \text{continuous-on } s (\lambda x. \text{arcsin } (f x))$

using *continuous-on-compose*[*of s f, OF - continuous-on-subset*[*OF continuous-on-arcsin'*]]

by (*auto simp: comp-def subset-eq*)

lemma *isCont-arcsin*: $-1 < x \implies x < 1 \implies \text{isCont } \text{arcsin } x$

using *continuous-on-arcsin'*[*THEN continuous-on-subset, of* $\{-1 <.. < 1\}$]

by (*auto simp: continuous-on-eq-continuous-at subset-eq*)

lemma *continuous-on-arccos'*: *continuous-on* $\{-1 .. 1\}$ *arccos*

proof –

have *continuous-on* (*cos* ‘ $\{0 .. \text{pi}\}$) *arccos*

by (*rule continuous-on-inv*) (*auto intro: continuous-intros simp: arccos-cos*)

also have *cos* ‘ $\{0 .. \text{pi}\} = \{-1 .. 1\}$

proof safe

fix $x :: \text{real}$

assume $x \in \{-1..1\}$

then show $x \in \text{cos} \text{ ‘ } \{0.. \text{pi}\}$

using *arccos-lbound arccos-ubound*

by (*intro image-eqI*[**where** $x = \text{arccos } x$]) *auto*

qed *simp*

finally show *?thesis* .

qed

lemma *continuous-on-arccos* [*continuous-intros*]:

continuous-on s f $\implies (\forall x \in s. -1 \leq f x \wedge f x \leq 1) \implies \text{continuous-on } s (\lambda x.$

arccos ($f x$)

using *continuous-on-compose*[*of s f*, *OF - continuous-on-subset*[*OF continuous-on-arccos*]]
by (*auto simp: comp-def subset-eq*)

lemma *isCont-arccos*: $-1 < x \implies x < 1 \implies \text{isCont } \text{arccos } x$

using *continuous-on-arccos*["*THEN continuous-on-subset, of* { $-1 < .. < 1$ }]
by (*auto simp: continuous-on-eq-continuous-at subset-eq*)

lemma *isCont-arctan*: $\text{isCont } \text{arctan } x$

proof –

obtain u **where** u : $-(\pi/2) < u < \text{arctan } x$
by (*meson arctan arctan-less-iff linordered-field-no-lb*)
obtain v **where** v : $\text{arctan } x < v < \pi/2$
by (*meson arctan-less-iff arctan-ubound linordered-field-no-ub*)
have $\text{isCont } \text{arctan } (\tan (\text{arctan } x))$
proof (*rule isCont-inverse-function2* [*of u arctan x v*])
show $\bigwedge z. \llbracket u \leq z; z \leq v \rrbracket \implies \text{arctan } (\tan z) = z$
using *arctan-unique* $u(1) v(2)$ **by** *auto*
then show $\bigwedge z. \llbracket u \leq z; z \leq v \rrbracket \implies \text{isCont } \tan z$
by (*metis arctan cos-gt-zero-pi isCont-tan less-irrefl*)
qed (*use u v in auto*)
then show *?thesis*
by (*simp add: arctan*)

qed

lemma *tendsto-arctan* [*tendsto-intros*]: $(f \longrightarrow x) F \implies ((\lambda x. \text{arctan } (f x)) \longrightarrow \text{arctan } x) F$

by (*rule isCont-tendsto-compose* [*OF isCont-arctan*])

lemma *continuous-arctan* [*continuous-intros*]: $\text{continuous } F f \implies \text{continuous } F (\lambda x. \text{arctan } (f x))$

unfolding *continuous-def* **by** (*rule tendsto-arctan*)

lemma *continuous-on-arctan* [*continuous-intros*]:

$\text{continuous-on } s f \implies \text{continuous-on } s (\lambda x. \text{arctan } (f x))$

unfolding *continuous-on-def* **by** (*auto intro: tendsto-arctan*)

lemma *DERIV-arcsin*:

assumes $-1 < x < 1$

shows $\text{DERIV } \text{arcsin } x :> \text{inverse } (\text{sqrt } (1 - x^2))$

proof (*rule DERIV-inverse-function*)

show (*sin has-real-derivative sqrt (1 - x^2)*) (*at (arcsin x)*)

by (*rule derivative-eq-intros* | *use assms cos-arcsin in force*)**+**

show $\text{sqrt } (1 - x^2) \neq 0$

using *abs-square-eq-1 assms* **by** *force*

qed (*use assms isCont-arcsin in auto*)

lemma *DERIV-arccos*:

assumes $-1 < x < 1$
shows $DERIV \arccos x := inverse (- \sqrt{1 - x^2})$
proof (rule $DERIV$ -inverse-function)
show $(\cos \text{ has-real-derivative } - \sqrt{1 - x^2}) (at (\arccos x))$
by (rule $derivative$ -eq-intros | use *assms sin-arccos* **in force**)
show $-\sqrt{1 - x^2} \neq 0$
using *abs-square-eq-1 assms* **by force**
qed (use *assms isCont-arccos* **in auto**)

lemma $DERIV$ -arctan: $DERIV \arctan x := inverse (1 + x^2)$
proof (rule $DERIV$ -inverse-function)
have $inverse ((\cos (\arctan x))^2) = 1 + x^2$
by (*metis arctan cos-arctan-not-zero power-inverse tan-sec*)
then show $(\tan \text{ has-real-derivative } 1 + x^2) (at (\arctan x))$
by (*auto intro!: derivative-eq-intros*)
show $\bigwedge y. \llbracket x - 1 < y; y < x + 1 \rrbracket \implies \tan (\arctan y) = y$
using *tan-arctan* **by blast**
show $1 + x^2 \neq 0$
by (*metis power-one sum-power2-eq-zero-iff zero-neq-one*)
qed (use *isCont-arctan* **in auto**)

declare

$DERIV$ -arcsin[*THEN DERIV-chain2, derivative-intros*]
 $DERIV$ -arcsin[*THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros*]
 $DERIV$ -arccos[*THEN DERIV-chain2, derivative-intros*]
 $DERIV$ -arccos[*THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros*]
 $DERIV$ -arctan[*THEN DERIV-chain2, derivative-intros*]
 $DERIV$ -arctan[*THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros*]

lemmas has -derivative-arctan[*derivative-intros*] = $DERIV$ -arctan[*THEN DERIV-compose-FDERIV*]
and has -derivative-arccos[*derivative-intros*] = $DERIV$ -arccos[*THEN DERIV-compose-FDERIV*]
and has -derivative-arcsin[*derivative-intros*] = $DERIV$ -arcsin[*THEN DERIV-compose-FDERIV*]

lemma $filterlim$ -tan-at-right: $filterlim \tan \text{ at-bot } (at\text{-right } (- (\pi/2)))$
by (rule $filterlim$ -at-bot-at-right[**where** $Q = \lambda x. -\pi/2 < x \wedge x < \pi/2$ **and** $P = \lambda x. True$ **and** $g = \arctan$])
 (*auto simp: arctan le-less eventually-at dist-real-def simp del: less-divide-eq-numeral1 intro!: tan-monotone exI[of - $\pi/2$]*)

lemma $filterlim$ -tan-at-left: $filterlim \tan \text{ at-top } (at\text{-left } (\pi/2))$
by (rule $filterlim$ -at-top-at-left[**where** $Q = \lambda x. -\pi/2 < x \wedge x < \pi/2$ **and** $P = \lambda x. True$ **and** $g = \arctan$])
 (*auto simp: arctan le-less eventually-at dist-real-def simp del: less-divide-eq-numeral1 intro!: tan-monotone exI[of - $\pi/2$]*)

lemma $tendsto$ -arctan-at-top: $(\arctan \longrightarrow (\pi/2)) \text{ at-top}$

proof (*rule tendstoI*)
fix $e :: \text{real}$
assume $0 < e$
define y **where** $y = \text{pi}/2 - \min (\text{pi}/2) e$
then have $y: 0 \leq y \ y < \text{pi}/2 \ \text{pi}/2 \leq e + y$
using $\langle 0 < e \rangle$ **by** *auto*
show *eventually* $(\lambda x. \text{dist} (\arctan x) (\text{pi}/2) < e)$ *at-top*
proof (*intro eventually-at-top-dense[THEN iffD2] exI allI impI*)
fix x
assume $\tan y < x$
then have $\arctan (\tan y) < \arctan x$
by (*simp add: arctan-less-iff*)
with y **have** $y < \arctan x$
by (*subst (asm) arctan-tan) simp-all*
with *arctan-ubound[of x, arith]* $y \ \langle 0 < e \rangle$
show $\text{dist} (\arctan x) (\text{pi}/2) < e$
by (*simp add: dist-real-def*)
qed
qed

lemma *tendsto-arctan-at-bot: (arctan \longrightarrow $-\ (\text{pi}/2)$) at-bot*
unfolding *filterlim-at-bot-mirror arctan-minus*
by (*intro tendsto-minus tendsto-arctan-at-top*)

113.18 Prove Totality of the Trigonometric Functions

lemma *cos-arccos-abs: $|y| \leq 1 \implies \cos (\arccos y) = y$*
by (*simp add: abs-le-iff*)

lemma *sin-arccos-abs: $|y| \leq 1 \implies \sin (\arccos y) = \text{sqrt} (1 - y^2)$*
by (*simp add: sin-arccos abs-le-iff*)

lemma *sin-mono-less-eq:*
 $-\ (\text{pi}/2) \leq x \implies x \leq \text{pi}/2 \implies -\ (\text{pi}/2) \leq y \implies y \leq \text{pi}/2 \implies \sin x < \sin y$
 $\longleftrightarrow x < y$
by (*metis not-less-iff-gr-or-eq sin-monotone-2pi*)

lemma *sin-mono-le-eq:*
 $-\ (\text{pi}/2) \leq x \implies x \leq \text{pi}/2 \implies -\ (\text{pi}/2) \leq y \implies y \leq \text{pi}/2 \implies \sin x \leq \sin y$
 $\longleftrightarrow x \leq y$
by (*meson leD le-less-linear sin-monotone-2pi sin-monotone-2pi-le*)

lemma *sin-inj-pi:*
 $-\ (\text{pi}/2) \leq x \implies x \leq \text{pi}/2 \implies -\ (\text{pi}/2) \leq y \implies y \leq \text{pi}/2 \implies \sin x = \sin y$
 $\implies x = y$
by (*metis arcsin-sin*)

lemma *arcsin-le-iff:*
assumes $x \geq -1 \ x \leq 1 \ y \geq -\text{pi}/2 \ y \leq \text{pi}/2$

shows $\arcsin x \leq y \longleftrightarrow x \leq \sin y$
proof –
have $\arcsin x \leq y \longleftrightarrow \sin (\arcsin x) \leq \sin y$
using *arcsin-bounded*[of x] *assms* **by** (*subst sin-mono-le-eq*) *auto*
also from *assms* **have** $\sin (\arcsin x) = x$ **by** *simp*
finally show *?thesis* .
qed

lemma *le-arcsin-iff*:
assumes $x \geq -1$ $x \leq 1$ $y \geq -\pi/2$ $y \leq \pi/2$
shows $\arcsin x \geq y \longleftrightarrow x \geq \sin y$
proof –
have $\arcsin x \geq y \longleftrightarrow \sin (\arcsin x) \geq \sin y$
using *arcsin-bounded*[of x] *assms* **by** (*subst sin-mono-le-eq*) *auto*
also from *assms* **have** $\sin (\arcsin x) = x$ **by** *simp*
finally show *?thesis* .
qed

lemma *cos-mono-less-eq*: $0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x < \cos y \longleftrightarrow y < x$
by (*meson cos-monotone-0-pi cos-monotone-0-pi-le leD le-less-linear*)

lemma *cos-mono-le-eq*: $0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x \leq \cos y \longleftrightarrow y \leq x$
by (*metis arccos-cos cos-monotone-0-pi-le eq-iff linear*)

lemma *cos-inj-pi*: $0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x = \cos y \implies x = y$
by (*metis arccos-cos*)

lemma *arccos-le-pi2*: $\llbracket 0 \leq y; y \leq 1 \rrbracket \implies \arccos y \leq \pi/2$
by (*metis (mono-tags) arccos-0 arccos cos-le-one cos-monotone-0-pi-le cos-pi cos-pi-half pi-half-ge-zero antisym-conv less-eq-neg-nonpos linear minus-minus order.trans order-refl*)

lemma *sincos-total-pi-half*:
assumes $0 \leq x$ $0 \leq y$ $x^2 + y^2 = 1$
shows $\exists t. 0 \leq t \wedge t \leq \pi/2 \wedge x = \cos t \wedge y = \sin t$
proof –
have $x1: x \leq 1$
using *assms* **by** (*metis le-add-same-cancel1 power2-le-imp-le power-one zero-le-power2*)
with *assms* **have** $*$: $0 \leq \arccos x$ $\cos (\arccos x) = x$
by (*auto simp: arccos*)
from *assms* **have** $y = \sqrt{1 - x^2}$
by (*metis abs-of-nonneg add commute add-diff-cancel real-sqrt-abs*)
with $x1$ $*$ *assms* *arccos-le-pi2* [of x] **show** *?thesis*
by (*rule-tac x=arccos x in exI*) (*auto simp: sin-arccos*)
qed

lemma *sincos-total-pi*:
assumes $0 \leq y$ $x^2 + y^2 = 1$
shows $\exists t. 0 \leq t \wedge t \leq \pi \wedge x = \cos t \wedge y = \sin t$
proof (*cases rule: le-cases [of 0 x]*)
case *le*
from *sincos-total-pi-half [OF le]* **show** *?thesis*
by (*metis pi-ge-two pi-half-le-two add.commute add-le-cancel-left add-mono assms*)
next
case *ge*
then have $0 \leq -x$
by *simp*
then obtain *t* **where** $t \geq 0 \ t \leq \pi/2 \ -x = \cos t \ y = \sin t$
using *sincos-total-pi-half assms*
by *auto (metis <0 ≤ - x> power2-minus)*
show *?thesis*
by (*rule exI [where x = pi - t]*) (*use t in auto*)
qed

lemma *sincos-total-2pi-le*:
assumes $x^2 + y^2 = 1$
shows $\exists t. 0 \leq t \wedge t \leq 2 * \pi \wedge x = \cos t \wedge y = \sin t$
proof (*cases rule: le-cases [of 0 y]*)
case *le*
from *sincos-total-pi [OF le]* **show** *?thesis*
by (*metis assms le-add-same-cancel1 mult.commute mult-2-right order.trans*)
next
case *ge*
then have $0 \leq -y$
by *simp*
then obtain *t* **where** $t \geq 0 \ t \leq \pi \ x = \cos t \ -y = \sin t$
using *sincos-total-pi assms*
by *auto (metis <0 ≤ - y> power2-minus)*
show *?thesis*
by (*rule exI [where x = 2 * pi - t]*) (*use t in auto*)
qed

lemma *sincos-total-2pi*:
assumes $x^2 + y^2 = 1$
obtains *t* **where** $0 \leq t \ t < 2*\pi \ x = \cos t \ y = \sin t$
proof –
from *sincos-total-2pi-le [OF assms]*
obtain *t* **where** $0 \leq t \ t \leq 2*\pi \ x = \cos t \ y = \sin t$
by *blast*
show *?thesis*
by (*cases t = 2 * pi*) (*use t that in <force+>*)
qed

lemma *arcsin-less-mono*: $|x| \leq 1 \implies |y| \leq 1 \implies \arcsin x < \arcsin y \iff x < y$

by (rule trans [OF sin-mono-less-eq [symmetric]]) (use arcsin-ubound arcsin-lbound in auto)

lemma arcsin-le-mono: $|x| \leq 1 \implies |y| \leq 1 \implies \arcsin x \leq \arcsin y \longleftrightarrow x \leq y$
using arcsin-less-mono not-le **by** blast

lemma arcsin-less-arcsin: $-1 \leq x \implies x < y \implies y \leq 1 \implies \arcsin x < \arcsin y$
using arcsin-less-mono **by** auto

lemma arcsin-le-arcsin: $-1 \leq x \implies x \leq y \implies y \leq 1 \implies \arcsin x \leq \arcsin y$
using arcsin-le-mono **by** auto

lemma arcsin-nonneg: $x \in \{0..1\} \implies \arcsin x \geq 0$
using arcsin-le-arcsin[of 0 x] **by** simp

lemma arccos-less-mono: $|x| \leq 1 \implies |y| \leq 1 \implies \arccos x < \arccos y \longleftrightarrow y < x$
by (rule trans [OF cos-mono-less-eq [symmetric]]) (use arccos-ubound arccos-lbound in auto)

lemma arccos-le-mono: $|x| \leq 1 \implies |y| \leq 1 \implies \arccos x \leq \arccos y \longleftrightarrow y \leq x$
using arccos-less-mono [of y x] **by** (simp add: not-le [symmetric])

lemma arccos-less-arccos: $-1 \leq x \implies x < y \implies y \leq 1 \implies \arccos y < \arccos x$
using arccos-less-mono **by** auto

lemma arccos-le-arccos: $-1 \leq x \implies x \leq y \implies y \leq 1 \implies \arccos y \leq \arccos x$
using arccos-le-mono **by** auto

lemma arccos-eq-iff: $|x| \leq 1 \wedge |y| \leq 1 \implies \arccos x = \arccos y \longleftrightarrow x = y$
using cos-arccos-abs **by** fastforce

lemma arccos-cos-eq-abs:

assumes $|\vartheta| \leq \pi$

shows $\arccos (\cos \vartheta) = |\vartheta|$

unfolding arccos-def

proof (intro the-equality conjI; clarify?)

show $\cos |\vartheta| = \cos \vartheta$

by (simp add: abs-real-def)

show $x = |\vartheta|$ **if** $\cos x = \cos \vartheta$ $0 \leq x \leq \pi$ **for** x

by (simp add: ‹cos | ϑ | = cos ϑ › assms cos-inj-pi that)

qed (use assms in auto)

lemma arccos-cos-eq-abs-2pi:

obtains k **where** $\arccos (\cos \vartheta) = |\vartheta - \text{of-int } k * (2 * \pi)|$

proof –

define k **where** $k \equiv \lfloor (\vartheta + \pi) / (2 * \pi) \rfloor$

have $\text{lepi: } |\vartheta - \text{of-int } k * (2 * \pi)| \leq \pi$

using floor-divide-lower [of 2*pi $\vartheta + \pi$] floor-divide-upper [of 2*pi $\vartheta + \pi$]

```

  by (auto simp: k-def abs-if algebra-simps)
  have arccos (cos  $\vartheta$ ) = arccos (cos ( $\vartheta - \text{of-int } k * (2 * \text{pi})$ ))
  using cos-int-2pin sin-int-2pin by (simp add: cos-diff mult.commute)
  also have ... =  $|\vartheta - \text{of-int } k * (2 * \text{pi})|$ 
  using arccos-cos-eq-abs lepi by blast
  finally show ?thesis
  using that by metis
qed

```

lemma *arccos-arctan*:

```

  assumes  $-1 < x \ x < 1$ 
  shows arccos  $x = \text{pi}/2 - \text{arctan}(x / \text{sqrt}(1 - x^2))$ 
proof -
  have arctan( $x / \text{sqrt}(1 - x^2)$ ) - ( $\text{pi}/2 - \text{arccos } x$ ) = 0
  proof (rule sin-eq-0-pi)
    show  $-\text{pi} < \text{arctan}(x / \text{sqrt}(1 - x^2)) - (\text{pi}/2 - \text{arccos } x)$ 
    using arctan-lbound [of  $x / \text{sqrt}(1 - x^2)$ ] arccos-bounded [of  $x$ ] assms
    by (simp add: algebra-simps)
  next
    show  $\text{arctan}(x / \text{sqrt}(1 - x^2)) - (\text{pi}/2 - \text{arccos } x) < \text{pi}$ 
    using arctan-ubound [of  $x / \text{sqrt}(1 - x^2)$ ] arccos-bounded [of  $x$ ] assms
    by (simp add: algebra-simps)
  next
    show  $\sin(\text{arctan}(x / \text{sqrt}(1 - x^2)) - (\text{pi}/2 - \text{arccos } x)) = 0$ 
    using assms
    by (simp add: algebra-simps sin-diff cos-add sin-arccos sin-arctan cos-arctan
      power2-eq-square square-eq-1-iff)
  qed
  then show ?thesis
  by simp
qed

```

lemma *arcsin-plus-arccos*:

```

  assumes  $-1 \leq x \ x \leq 1$ 
  shows arcsin  $x + \text{arccos } x = \text{pi}/2$ 
proof -
  have arcsin  $x = \text{pi}/2 - \text{arccos } x$ 
  apply (rule sin-inj-pi)
  using assms arcsin [OF assms] arccos [OF assms]
  by (auto simp: algebra-simps sin-diff)
  then show ?thesis
  by (simp add: algebra-simps)
qed

```

lemma *arcsin-arccos-eq*: $-1 \leq x \implies x \leq 1 \implies \text{arcsin } x = \text{pi}/2 - \text{arccos } x$
 using *arcsin-plus-arccos* by force

lemma *arccos-arcsin-eq*: $-1 \leq x \implies x \leq 1 \implies \text{arccos } x = \text{pi}/2 - \text{arcsin } x$
 using *arcsin-plus-arccos* by force

lemma *arcsin-arctan*: $-1 < x \implies x < 1 \implies \arcsin x = \arctan(x / \text{sqrt}(1 - x^2))$
by (*simp add: arccos-arctan arcsin-arccos-eq*)

lemma *arcsin-arccos-sqrt-pos*: $0 \leq x \implies x \leq 1 \implies \arcsin x = \arccos(\text{sqrt}(1 - x^2))$
by (*smt (verit, del-insts) arccos-cos arcsin-0 arcsin-le-arcsin arcsin-pi cos-arcsin*)

lemma *arcsin-arccos-sqrt-neg*: $-1 \leq x \implies x \leq 0 \implies \arcsin x = -\arccos(\text{sqrt}(1 - x^2))$
using *arcsin-arccos-sqrt-pos* [*of -x*]
by (*simp add: arcsin-minus*)

lemma *arccos-arcsin-sqrt-pos*: $0 \leq x \implies x \leq 1 \implies \arccos x = \arcsin(\text{sqrt}(1 - x^2))$
by (*smt (verit, del-insts) arccos-lbound arccos-le-pi2 arcsin-sin sin-arccos*)

lemma *arccos-arcsin-sqrt-neg*: $-1 \leq x \implies x \leq 0 \implies \arccos x = \pi - \arcsin(\text{sqrt}(1 - x^2))$
using *arccos-arcsin-sqrt-pos* [*of -x*]
by (*simp add: arccos-minus*)

lemma *cos-limit-1*:
assumes $(\lambda j. \cos (\vartheta j)) \longrightarrow 1$
shows $\exists k. (\lambda j. \vartheta j - \text{of-int } (k j) * (2 * \pi)) \longrightarrow 0$
proof –
have $\forall_F j$ *in sequentially*. $\cos (\vartheta j) \in \{-1..1\}$
by *auto*
then have $(\lambda j. \arccos (\cos (\vartheta j))) \longrightarrow \arccos 1$
using *continuous-on-tendsto-compose* [*OF continuous-on-arccos' assms*] **by** *auto*
moreover have $\bigwedge j. \exists k. \arccos (\cos (\vartheta j)) = |\vartheta j - \text{of-int } k * (2 * \pi)|$
using *arccos-cos-eq-abs-2pi* **by** *metis*
then have $\exists k. \forall j. \arccos (\cos (\vartheta j)) = |\vartheta j - \text{of-int } (k j) * (2 * \pi)|$
by *metis*
ultimately have $\exists k. (\lambda j. |\vartheta j - \text{of-int } (k j) * (2 * \pi)|) \longrightarrow 0$
by *auto*
then show *?thesis*
by (*simp add: tendsto-rabs-zero-iff*)
qed

lemma *cos-diff-limit-1*:
assumes $(\lambda j. \cos (\vartheta j - \Theta)) \longrightarrow 1$
obtains k **where** $(\lambda j. \vartheta j - \text{of-int } (k j) * (2 * \pi)) \longrightarrow \Theta$
proof –
obtain k **where** $(\lambda j. (\vartheta j - \Theta) - \text{of-int } (k j) * (2 * \pi)) \longrightarrow 0$
using *cos-limit-1* [*OF assms*] **by** *auto*
then have $(\lambda j. \Theta + ((\vartheta j - \Theta) - \text{of-int } (k j) * (2 * \pi))) \longrightarrow \Theta + 0$
by (*rule tendsto-add* [*OF tendsto-const*])
with that show *?thesis*

by *auto*
qed

113.19 Machin’s formula

lemma *arctan-one*: $\arctan 1 = \pi/4$
by (*rule arctan-unique*) (*simp-all add: tan-45 m2pi-less-pi*)

lemma *tan-total-pi4*:
assumes $|x| < 1$
shows $\exists z. -(\pi/4) < z \wedge z < \pi/4 \wedge \tan z = x$

proof
show $-(\pi/4) < \arctan x \wedge \arctan x < \pi/4 \wedge \tan(\arctan x) = x$
unfolding *arctan-one* [*symmetric*] *arctan-minus* [*symmetric*]
unfolding *arctan-less-iff*
using *assms* by (*auto simp: arctan*)

qed

lemma *arctan-add*:
assumes $|x| \leq 1 \ |y| < 1$
shows $\arctan x + \arctan y = \arctan((x + y) / (1 - x * y))$
proof (*rule arctan-unique* [*symmetric*])
have $-(\pi/4) \leq \arctan x - (\pi/4) < \arctan y$
unfolding *arctan-one* [*symmetric*] *arctan-minus* [*symmetric*]
unfolding *arctan-le-iff* *arctan-less-iff*
using *assms* by *auto*
from *add-le-less-mono* [*OF this*] show 1: $-(\pi/2) < \arctan x + \arctan y$
by *simp*
have $\arctan x \leq \pi/4 \ \arctan y < \pi/4$
unfolding *arctan-one* [*symmetric*]
unfolding *arctan-le-iff* *arctan-less-iff*
using *assms* by *auto*
from *add-le-less-mono* [*OF this*] show 2: $\arctan x + \arctan y < \pi/2$
by *simp*
show $\tan(\arctan x + \arctan y) = (x + y) / (1 - x * y)$
using *cos-gt-zero-pi* [*OF 1 2*] by (*simp add: arctan tan-add*)

qed

lemma *arctan-double*: $|x| < 1 \implies 2 * \arctan x = \arctan((2 * x) / (1 - x^2))$
by (*metis arctan-add linear mult-2 not-less power2-eq-square*)

theorem *machin*: $\pi/4 = 4 * \arctan(1 / 5) - \arctan(1/239)$

proof –
have $|1 / 5| < (1 :: \text{real})$
by *auto*
from *arctan-add*[*OF less-imp-le*[*OF this*] *this*] have $2 * \arctan(1 / 5) = \arctan(5 / 12)$
by *auto*
moreover


```

have |5 / 12| < (1 :: real)
  by auto
from arctan-add[OF less-imp-le[OF this] this] have 2 * arctan (5 / 12) = arctan
(120 / 119)
  by auto
moreover
have |1| ≤ (1::real) and |1/239| < (1::real)
  by auto
from arctan-add[OF this] have arctan 1 + arctan (1/239) = arctan (120 /
119)
  by auto
ultimately have arctan 1 + arctan (1/239) = 4 * arctan (1 / 5)
  by auto
then show ?thesis
  unfolding arctan-one by algebra
qed

```

lemma *machin-Euler*: $5 * \arctan (1 / 7) + 2 * \arctan (3 / 79) = \pi/4$

proof –

```

have 17: |1 / 7| < (1 :: real) by auto
with arctan-double have 2 * arctan (1 / 7) = arctan (7 / 24)
  by simp (simp add: field-simps)
moreover
have |7 / 24| < (1 :: real) by auto
with arctan-double have 2 * arctan (7 / 24) = arctan (336 / 527)
  by simp (simp add: field-simps)
moreover
have |336 / 527| < (1 :: real) by auto
from arctan-add[OF less-imp-le[OF 17] this]
have arctan(1/7) + arctan (336 / 527) = arctan (2879 / 3353)
  by auto
ultimately have I: 5 * arctan (1 / 7) = arctan (2879 / 3353) by auto
have 379: |3 / 79| < (1 :: real) by auto
with arctan-double have II: 2 * arctan (3 / 79) = arctan (237 / 3116)
  by simp (simp add: field-simps)
have *: |2879 / 3353| < (1 :: real) by auto
have |237 / 3116| < (1 :: real) by auto
from arctan-add[OF less-imp-le[OF *] this] have arctan (2879/3353) + arctan
(237/3116) = pi/4
  by (simp add: arctan-one)
with I II show ?thesis by auto
qed

```

113.20 Introducing the inverse tangent power series

lemma *monoseq-arctan-series*:

fixes $x :: \text{real}$

assumes $|x| \leq 1$

shows *monoseq* ($\lambda n. 1 / \text{real} (n * 2 + 1) * x^{(n * 2 + 1)}$)

```

    (is monoseq ?a)
  proof (cases x = 0)
    case True
    then show ?thesis by (auto simp: monoseq-def)
  next
    case False
    have norm x ≤ 1 and x ≤ 1 and -1 ≤ x
      using assms by auto
    show monoseq ?a
  proof -
    have mono: 1 / real (Suc (Suc n * 2)) * x ^ Suc (Suc n * 2) ≤
      1 / real (Suc (n * 2)) * x ^ Suc (n * 2)
      if 0 ≤ x and x ≤ 1 for n and x :: real
    proof (rule mult-mono)
      show 1 / real (Suc (Suc n * 2)) ≤ 1 / real (Suc (n * 2))
        by (rule frac-le) simp-all
      show 0 ≤ 1 / real (Suc (n * 2))
        by auto
      show x ^ Suc (Suc n * 2) ≤ x ^ Suc (n * 2)
        by (rule power-decreasing) (simp-all add: ‹0 ≤ x› ‹x ≤ 1›)
      show 0 ≤ x ^ Suc (Suc n * 2)
        by (rule zero-le-power) (simp add: ‹0 ≤ x›)
    qed
  show ?thesis
  proof (cases 0 ≤ x)
    case True
    from mono[OF this ‹x ≤ 1›, THEN allI]
    show ?thesis
      unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI2)
  next
    case False
    then have 0 ≤ -x and -x ≤ 1
      using ‹-1 ≤ x› by auto
    from mono[OF this]
    have 1 / real (Suc (Suc n * 2)) * x ^ Suc (Suc n * 2) ≥
      1 / real (Suc (n * 2)) * x ^ Suc (n * 2) for n
      using ‹0 ≤ -x› by auto
    then show ?thesis
      unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI1[OF allI])
  qed
qed
qed
qed

```

lemma *zeroseq-arctan-series*:

```

  fixes x :: real
  assumes |x| ≤ 1
  shows (λn. 1 / real (n * 2 + 1) * x ^ (n * 2 + 1)) → 0
    (is ?a → 0)
  proof (cases x = 0)

```

```

case True
then show ?thesis by simp
next
case False
have norm x ≤ 1 and x ≤ 1 and -1 ≤ x
  using assms by auto
show ?a → 0
proof (cases |x| < 1)
case True
then have norm x < 1 by auto
  from tendsto-mult[OF LIMSEQ-inverse-real-of-nat LIMSEQ-power-zero[OF
⟨norm x < 1⟩, THEN LIMSEQ-Suc]]
  have (λn. 1 / real (n + 1) * x ^ (n + 1)) → 0
    unfolding inverse-eq-divide Suc-eq-plus1 by simp
  then show ?thesis
    using pos2 by (rule LIMSEQ-linear)
next
case False
then have x = -1 ∨ x = 1
  using ⟨|x| ≤ 1⟩ by auto
then have n-eq: ∧ n. x ^ (n * 2 + 1) = x
  unfolding One-nat-def by auto
  from tendsto-mult[OF LIMSEQ-inverse-real-of-nat[THEN LIMSEQ-linear, OF
pos2, unfolded inverse-eq-divide] tendsto-const[of x]]
  show ?thesis
  unfolding n-eq Suc-eq-plus1 by auto
qed
qed

```

lemma *summable-arctan-series*:

```

fixes n :: nat
assumes |x| ≤ 1
shows summable (λ k. (-1) ^ k * (1 / real (k*2+1) * x ^ (k*2+1)))
  (is summable (?c x))
by (rule summable-Leibniz(1),
  rule zeroseq-arctan-series[OF assms],
  rule monoseq-arctan-series[OF assms])

```

lemma *DERIV-arctan-series*:

```

assumes |x| < 1
shows DERIV (λ x'. ∑ k. (-1) ^ k * (1 / real (k * 2 + 1) * x' ^ (k * 2 + 1)))
x :=>
  (∑ k. (-1) ^ k * x ^ (k * 2))
  (is DERIV ?arctan - :=> ?Int)
proof -
let ?f = λ n. if even n then (-1) ^ (n div 2) * 1 / real (Suc n) else 0

have n-even: even n ⇒ 2 * (n div 2) = n for n :: nat
  by presburger

```

```

then have if-eq:  $?f\ n * \text{real}\ (\text{Suc}\ n) * x'^{\wedge}n =$ 
   $(\text{if even } n \text{ then } (-1)^{\wedge}(n \text{ div } 2) * x'^{\wedge}(2 * (n \text{ div } 2)) \text{ else } 0)$ 
for  $n\ x'$ 
by auto

have summable-Integral: summable  $(\lambda\ n.\ (-1)^{\wedge}n * x^{\wedge}(2 * n))$  if  $|x| < 1$  for
 $x :: \text{real}$ 
proof –
  from that have  $x^2 < 1$ 
    by (simp add: abs-square-less-1)
  have summable  $(\lambda\ n.\ (-1)^{\wedge}n * (x^2)^{\wedge}n)$ 
    by (rule summable-Leibniz(1))
   $(\text{auto intro!: LIMSEQ-realpow-zero monoseq-realpow } \langle x^2 < 1 \rangle \text{ order-less-imp-le[OF } \langle x^2 < 1 \rangle])$ 
  then show ?thesis
    by (simp only: power-mult)
qed

have sums-even:  $(\text{sums})\ f = (\text{sums})\ (\lambda\ n.\ \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0)$ 
for  $f :: \text{nat} \Rightarrow \text{real}$ 
proof –
  have  $f\ \text{sums}\ x = (\lambda\ n.\ \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0)\ \text{sums}\ x$  for  $x :: \text{real}$ 
proof
  assume  $f\ \text{sums}\ x$ 
  from sums-if[OF sums-zero this] show  $(\lambda n.\ \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0)\ \text{sums}\ x$ 
    by auto
  next
  assume  $(\lambda\ n.\ \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0)\ \text{sums}\ x$ 
  from LIMSEQ-linear[OF this[simplified sums-def] pos2, simplified sum-split-even-odd[simplified mult.commute]]
  show  $f\ \text{sums}\ x$ 
  unfolding sums-def by auto
qed
then show ?thesis ..
qed

have Int-eq:  $(\sum\ n.\ ?f\ n * \text{real}\ (\text{Suc}\ n) * x^{\wedge}n) = ?\text{Int}$ 
unfolding if-eq mult.commute[of - 2]
  suminf-def sums-even[of  $\lambda\ n.\ (-1)^{\wedge}n * x^{\wedge}(2 * n)$ , symmetric]
by auto

have arctan-eq:  $(\sum\ n.\ ?f\ n * x^{\wedge}(\text{Suc}\ n)) = ?\text{arctan}\ x$  for  $x$ 
proof –
  have if-eq':  $\bigwedge n.\ (\text{if even } n \text{ then } (-1)^{\wedge}(n \text{ div } 2) * 1 / \text{real}\ (\text{Suc}\ n) \text{ else } 0) * x^{\wedge}\text{Suc}\ n =$ 
 $(\text{if even } n \text{ then } (-1)^{\wedge}(n \text{ div } 2) * (1 / \text{real}\ (\text{Suc}\ (2 * (n \text{ div } 2)))) * x^{\wedge}\text{Suc}\ (2 * (n \text{ div } 2))) \text{ else } 0)$ 
    using n-even by auto

```

```

have idx-eq:  $\bigwedge n. n * 2 + 1 = \text{Suc } (2 * n)$ 
by auto
then show ?thesis
  unfolding if-eq' idx-eq suminf-def
    sums-even[of  $\lambda n. (-1) ^ n * (1 / \text{real } (\text{Suc } (2 * n))) * x ^ \text{Suc } (2 * n)$ ],
symmetric]
  by auto
qed

```

```

have DERIV ( $\lambda x. \sum n. ?f n * x ^ (\text{Suc } n)$ ) x :> ( $\sum n. ?f n * \text{real } (\text{Suc } n) * x ^ n$ )
proof (rule DERIV-power-series')
  show  $x \in \{-1 <..< 1\}$ 
  using  $\langle |x| < 1 \rangle$  by auto
  show summable ( $\lambda n. ?f n * \text{real } (\text{Suc } n) * x ^ n$ )
  if x'-bounds:  $x' \in \{-1 <..< 1\}$  for  $x' :: \text{real}$ 
  proof –
    from that have  $|x'| < 1$  by auto
    then show ?thesis
      using that sums-summable sums-if [OF sums-0 [of  $\lambda x. 0$ ] summable-sums
[OF summable-Integral]]
      by (auto simp add: if-distrib [of  $\lambda x. x * y$  for  $y$ ] cong: if-cong)
    qed
  qed auto
then show ?thesis
  by (simp only: Int-eq arctan-eq)
qed

```

lemma *arctan-series*:

```

assumes  $|x| \leq 1$ 
shows  $\text{arctan } x = (\sum k. (-1) ^ k * (1 / \text{real } (k * 2 + 1)) * x ^ (k * 2 + 1))$ 
  (is - = suminf ( $\lambda n. ?c x n$ ))
proof –
  let  $?c' = \lambda x n. (-1) ^ n * x ^ (n * 2)$ 

```

```

have DERIV-arctan-suminf: DERIV ( $\lambda x. \text{suminf } (?c x)$ ) x :> (suminf ( $?c' x$ ))
  if  $0 < r$  and  $r < 1$  and  $|x| < r$  for  $r x :: \text{real}$ 

```

```

proof (rule DERIV-arctan-series)
  from that show  $|x| < 1$ 
  using  $\langle r < 1 \rangle$  and  $\langle |x| < r \rangle$  by auto
qed

```

```

{
  fix  $x :: \text{real}$ 
  assume  $|x| \leq 1$ 
  note summable-Leibniz[OF zeroseq-arctan-series[OF this] monoseq-arctan-series[OF this]]
} note arctan-series-borders = this

```

```

have when-less-one: arctan x = ( $\sum k. ?c x k$ ) if  $|x| < 1$  for  $x :: \text{real}$ 
proof –
  obtain r where  $|x| < r$  and  $r < 1$ 
  using dense[OF  $\langle |x| < 1 \rangle$ ] by blast
  then have  $0 < r$  and  $-r < x$  and  $x < r$  by auto

  have suminf-eq-arctan-bounded: suminf (?c x) – arctan x = suminf (?c a) –
  arctan a
  if  $-r < a$  and  $b < r$  and  $a < b$  and  $a \leq x$  and  $x \leq b$  for  $x a b$ 
  proof –
    from that have  $|x| < r$  by auto
    show suminf (?c x) – arctan x = suminf (?c a) – arctan a
    proof (rule DERIV-isconst2[of a b])
      show  $a < b$  and  $a \leq x$  and  $x \leq b$ 
      using  $\langle a < b \rangle \langle a \leq x \rangle \langle x \leq b \rangle$  by auto
      have  $\forall x. -r < x \wedge x < r \longrightarrow \text{DERIV } (\lambda x. \text{suminf } (?c x) - \text{arctan } x) x$ 
       $:=> 0$ 
      proof (rule allI, rule impI)
        fix x
        assume  $-r < x \wedge x < r$ 
        then have  $|x| < r$  by auto
        with  $\langle r < 1 \rangle$  have  $|x| < 1$  by auto
        have  $|-(x^2)| < 1$  using abs-square-less-1  $\langle |x| < 1 \rangle$  by auto
        then have  $(\lambda n. (- (x^2)) ^ n \text{ sums } (1 / (1 - (- (x^2))))$ 
          unfolding real-norm-def[symmetric] by (rule geometric-sums)
        then have  $(?c' x) \text{ sums } (1 / (1 - (- (x^2))))$ 
          unfolding power-mult-distrib[symmetric] power-mult mult.commute[of -
2] by auto
        then have suminf-c'-eq-geom: inverse (1 + x2) = suminf (?c' x)
          using sums-unique unfolding inverse-eq-divide by auto
        have DERIV ( $\lambda x. \text{suminf } (?c x) x :=> (\text{inverse } (1 + x^2))$ )
          unfolding suminf-c'-eq-geom
          by (rule DERIV-arctan-suminf[OF  $\langle 0 < r \rangle \langle r < 1 \rangle \langle |x| < r \rangle$ ])
        from DERIV-diff [OF this DERIV-arctan] show DERIV ( $\lambda x. \text{suminf } (?c$ 
x) – arctan x) x  $:=> 0$ 
          by auto
        qed
        then have DERIV-in-rball:  $\forall y. a \leq y \wedge y \leq b \longrightarrow \text{DERIV } (\lambda x. \text{suminf }$ 
(?c x) – arctan x) y  $:=> 0$ 
          using  $\langle -r < a \rangle \langle b < r \rangle$  by auto
        then show  $\bigwedge y. \llbracket a < y; y < b \rrbracket \implies \text{DERIV } (\lambda x. \text{suminf } (?c x) - \text{arctan}$ 
x) y  $:=> 0$ 
          using  $\langle |x| < r \rangle$  by auto
        show continuous-on {a..b} ( $\lambda x. \text{suminf } (?c x) - \text{arctan } x$ )
          using DERIV-in-rball DERIV-atLeastAtMost-imp-continuous-on by blast
        qed
      qed
    have suminf-arctan-zero: suminf (?c 0) – arctan 0 = 0

```

```

unfolding Suc-eq-plus1 [symmetric] power-Suc2 mult-zero-right arctan-zero-zero
suminf-zero
  by auto

have suminf (?c x) - arctan x = 0
proof (cases x = 0)
  case True
    then show ?thesis
      using suminf-arctan-zero by auto
  next
    case False
      then have  $0 < |x|$  and  $-|x| < |x|$ 
        by auto
      have suminf (?c (-|x|)) - arctan (-|x|) = suminf (?c 0) - arctan 0
        by (rule suminf-eq-arctan-bounded[where x1=0 and a1=-|x| and b1=|x|,
symmetric])
          (simp-all only: <|x| < r> <-|x| < |x|> neg-less-iff-less)
      moreover
        have suminf (?c x) - arctan x = suminf (?c (-|x|)) - arctan (-|x|)
          by (rule suminf-eq-arctan-bounded[where x1=x and a1=-|x| and b1=|x|])
            (simp-all only: <|x| < r> <-|x| < |x|> neg-less-iff-less)
        ultimately show ?thesis
          using suminf-arctan-zero by auto
      qed
    then show ?thesis by auto
  qed

show arctan x = suminf ( $\lambda n. ?c\ x\ n$ )
proof (cases  $|x| < 1$ )
  case True
    then show ?thesis by (rule when-less-one)
  next
    case False
      then have  $|x| = 1$  using  $<|x| \leq 1>$  by auto
      let ?a =  $\lambda x\ n. |1 / \text{real } (n * 2 + 1) * x^{(n * 2 + 1)}|$ 
      let ?diff =  $\lambda x\ n. |\text{arctan } x - (\sum i < n. ?c\ x\ i)|$ 
      have ?diff 1 n ≤ ?a 1 n for n :: nat
      proof -
        have  $0 < (1 :: \text{real})$  by auto
        moreover
          have ?diff x n ≤ ?a x n if  $0 < x$  and  $x < 1$  for x :: real
          proof -
            from that have  $|x| \leq 1$  and  $|x| < 1$ 
              by auto
            from  $<0 < x>$  have  $0 < 1 / \text{real } (0 * 2 + (1::\text{nat})) * x^{(0 * 2 + 1)}$ 
              by auto
            note bounds = mp[OF arctan-series-borders(2)[OF <|x| ≤ 1>] this, unfolded
when-less-one[OF <|x| < 1>, symmetric], THEN spec]
            have  $0 < 1 / \text{real } (n * 2 + 1) * x^{(n * 2 + 1)}$ 

```

by (rule mult-pos-pos) (simp-all only: zero-less-power[OF $0 < x$], auto)
 then have a-pos: $?a \ x \ n = 1 / \text{real } (n*2+1) * x^{(n*2+1)}$
 by (rule abs-of-pos)
 show ?thesis
 proof (cases even n)
 case True
 then have sgn-pos: $(-1)^{\hat{n}} = (1::\text{real})$ by auto
 from $\langle \text{even } n \rangle$ obtain m where $n = 2 * m ..$
 then have $2 * m = n ..$
 from bounds[of m, unfolded this atLeastAtMost-iff]
 have $|\arctan x - (\sum_{i < n. (?c \ x \ i))| \leq (\sum_{i < n + 1. (?c \ x \ i)} - (\sum_{i < n. (?c \ x \ i)})$
 (?c x i)
 by auto
 also have $\dots = ?c \ x \ n$ by auto
 also have $\dots = ?a \ x \ n$ unfolding sgn-pos a-pos by auto
 finally show ?thesis .
 next
 case False
 then have sgn-neg: $(-1)^{\hat{n}} = (-1::\text{real})$ by auto
 from $\langle \text{odd } n \rangle$ obtain m where $n = 2 * m + 1 ..$
 then have m-def: $2 * m + 1 = n ..$
 then have m-plus: $2 * (m + 1) = n + 1$ by auto
 from bounds[of m + 1, unfolded this atLeastAtMost-iff, THEN conjunct1]
 bounds[of m, unfolded m-def atLeastAtMost-iff, THEN conjunct2]
 have $|\arctan x - (\sum_{i < n. (?c \ x \ i))| \leq (\sum_{i < n. (?c \ x \ i)} - (\sum_{i < n+1. (?c \ x \ i)})$
 (?c x i) by auto
 also have $\dots = - ?c \ x \ n$ by auto
 also have $\dots = ?a \ x \ n$ unfolding sgn-neg a-pos by auto
 finally show ?thesis .
 qed
 qed
 hence $\forall x \in \{ 0 < .. < 1 \}. 0 \leq ?a \ x \ n - ?diff \ x \ n$ by auto
 moreover have isCont $(\lambda x. ?a \ x \ n - ?diff \ x \ n) \ x$ for x
 unfolding diff-conv-add-uminus divide-inverse
 by (auto intro!: isCont-add isCont-rabs continuous-ident isCont-minus
 isCont-arctan
 continuous-at-within-inverse isCont-mult isCont-power continuous-const
 isCont-sum
 simp del: add-uminus-conv-diff)
 ultimately have $0 \leq ?a \ 1 \ n - ?diff \ 1 \ n$
 by (rule LIM-less-bound)
 then show ?thesis by auto
 qed
 have $?a \ 1 \ \longrightarrow 0$
 unfolding tendsto-rabs-zero-iff power-one divide-inverse One-nat-def
 by (auto intro!: tendsto-mult LIMSEQ-linear LIMSEQ-inverse-real-of-nat simp
 del: of-nat-Suc)
 have $?diff \ 1 \ \longrightarrow 0$
 proof (rule LIMSEQ-I)


```

fix r :: real
assume 0 < r
obtain N :: nat where N-I: N ≤ n ⇒ ?a 1 n < r for n
  using LIMSEQ-D[OF ‹?a 1 ⟶ 0› ‹0 < r›] by auto
have norm (?diff 1 n - 0) < r if N ≤ n for n
  using ‹?diff 1 n ≤ ?a 1 n› N-I[OF that] by auto
then show ∃ N. ∀ n ≥ N. norm (?diff 1 n - 0) < r by blast
qed
from this [unfolded tendsto-rabs-zero-iff, THEN tendsto-add [OF - tendsto-const],
of - arctan 1, THEN tendsto-minus]
have (?c 1) sums (arctan 1) unfolding sums-def by auto
then have arctan 1 = (∑ i. ?c 1 i) by (rule sums-unique)

show ?thesis
proof (cases x = 1)
  case True
    then show ?thesis by (simp add: ‹arctan 1 = (∑ i. ?c 1 i)›)
  next
    case False
      then have x = -1 using ‹|x| = 1› by auto

      have - (pi/2) < 0 using pi-gt-zero by auto
      have - (2 * pi) < 0 using pi-gt-zero by auto

      have c-minus-minus: ?c (- 1) i = - ?c 1 i for i by auto

      have arctan (- 1) = arctan (tan (-(pi/4)))
        unfolding tan-45 tan-minus ..
      also have ... = - (pi/4)
        by (rule arctan-tan) (auto simp: order-less-trans[OF ‹- (pi/2) < 0›
pi-gt-zero])
      also have ... = - (arctan (tan (pi/4)))
        unfolding neg-equal-iff-equal
      by (rule arctan-tan[symmetric]) (auto simp: order-less-trans[OF ‹- (2 * pi)
< 0› pi-gt-zero])
      also have ... = - (arctan 1)
        unfolding tan-45 ..
      also have ... = - (∑ i. ?c 1 i)
        using ‹arctan 1 = (∑ i. ?c 1 i)› by auto
      also have ... = (∑ i. ?c (- 1) i)
        using suminf-minus[OF sums-summable[OF ‹(?c 1) sums (arctan 1)›]]
        unfolding c-minus-minus by auto
      finally show ?thesis using ‹x = -1› by auto
    qed
  qed
qed

lemma arctan-half: arctan x = 2 * arctan (x / (1 + sqrt(1 + x2)))
for x :: real

```

proof –

obtain y **where** $low: -(\pi/2) < y$ **and** $high: y < \pi/2$ **and** $y\text{-eq}: \tan y = x$
using tan-total **by** blast
then have $low2: -(\pi/2) < y / 2$ **and** $high2: y / 2 < \pi/2$
by auto

have $0 < \cos y$ **by** $(\text{rule } \text{cos-gt-zero-pi}[\text{OF } low \text{ high}])$
then have $\cos y \neq 0$ **and** $\text{cos-sqrt}: \text{sqrt}((\cos y)^2) = \cos y$
by auto

have $1 + (\tan y)^2 = 1 + (\sin y)^2 / (\cos y)^2$
unfolding tan-def power-divide **..**
also have $\dots = (\cos y)^2 / (\cos y)^2 + (\sin y)^2 / (\cos y)^2$
using $\langle \cos y \neq 0 \rangle$ **by** auto
also have $\dots = 1 / (\cos y)^2$
unfolding $\text{add-divide-distrib}[\text{symmetric}]$ $\text{sin-cos-squared-add2}$ **..**
finally have $1 + (\tan y)^2 = 1 / (\cos y)^2$.

have $\sin y / (\cos y + 1) = \tan y / ((\cos y + 1) / \cos y)$
unfolding tan-def **using** $\langle \cos y \neq 0 \rangle$ **by** $(\text{simp } \text{add: field-simps})$
also have $\dots = \tan y / (1 + 1 / \cos y)$
using $\langle \cos y \neq 0 \rangle$ **unfolding** $\text{add-divide-distrib}$ **by** auto
also have $\dots = \tan y / (1 + 1 / \text{sqrt}((\cos y)^2))$
unfolding cos-sqrt **..**
also have $\dots = \tan y / (1 + \text{sqrt}(1 / (\cos y)^2))$
unfolding real-sqrt-divide **by** auto
finally have $\text{eq}: \sin y / (\cos y + 1) = \tan y / (1 + \text{sqrt}(1 + (\tan y)^2))$
unfolding $\langle 1 + (\tan y)^2 = 1 / (\cos y)^2 \rangle$.

have $\arctan x = y$
using arctan-tan low $high$ $y\text{-eq}$ **by** auto
also have $\dots = 2 * (\arctan(\tan(y/2)))$
using $\text{arctan-tan}[\text{OF } low2 \text{ high2}]$ **by** auto
also have $\dots = 2 * (\arctan(\sin y / (\cos y + 1)))$
unfolding tan-half **by** auto
finally show $?thesis$
unfolding $\text{eq } \langle \tan y = x \rangle$.

qed

lemma $\text{arctan-monotone}: x < y \implies \arctan x < \arctan y$
by $(\text{simp } \text{only: arctan-less-iff})$

lemma $\text{arctan-monotone}': x \leq y \implies \arctan x \leq \arctan y$
by $(\text{simp } \text{only: arctan-le-iff})$

lemma arctan-inverse :
assumes $x \neq 0$
shows $\arctan(1 / x) = \text{sgn } x * \pi/2 - \arctan x$
proof $(\text{rule } \text{arctan-unique})$

```

have §:  $x > 0 \implies \arctan x < \pi$ 
  using arctan-bounded [of x] by linarith
show  $-(\pi/2) < \operatorname{sgn} x * \pi/2 - \arctan x$ 
  using assms by (auto simp: sgn-real-def arctan algebra-simps §)
show  $\operatorname{sgn} x * \pi/2 - \arctan x < \pi/2$ 
  using arctan-bounded [of - x] assms
  by (auto simp: algebra-simps sgn-real-def arctan-minus)
show  $\tan (\operatorname{sgn} x * \pi/2 - \arctan x) = 1 / x$ 
  unfolding tan-inverse [of arctan x, unfolded tan-arctan] sgn-real-def
  by (simp add: tan-def cos-arctan sin-arctan sin-diff cos-diff)
qed

```

```

theorem pi-series:  $\pi/4 = (\sum k. (-1)^k * 1 / \operatorname{real} (k * 2 + 1))$ 
(is - = ?SUM)
proof -
  have  $\pi/4 = \arctan 1$ 
    using arctan-one by auto
  also have  $\dots = ?SUM$ 
    using arctan-series[of 1] by auto
  finally show ?thesis by auto
qed

```

113.21 Existence of Polar Coordinates

```

lemma cos-x-y-le-one:  $|x / \operatorname{sqrt} (x^2 + y^2)| \leq 1$ 
  by (rule power2-le-imp-le [OF - zero-le-one])
  (simp add: power-divide divide-le-eq not-sum-power2-lt-zero)

```

```

lemma polar-Ex:  $\exists r::\operatorname{real}. \exists a. x = r * \cos a \wedge y = r * \sin a$ 

```

```

proof -
  have polar-ex1:  $\exists r a. x = r * \cos a \wedge y = r * \sin a$  if  $0 < y$  for  $y$ 
  proof -
    have  $x = \operatorname{sqrt} (x^2 + y^2) * \cos (\operatorname{arccos} (x / \operatorname{sqrt} (x^2 + y^2)))$ 
      by (simp add: cos-arccos-abs [OF cos-x-y-le-one])
    moreover have  $y = \operatorname{sqrt} (x^2 + y^2) * \sin (\operatorname{arccos} (x / \operatorname{sqrt} (x^2 + y^2)))$ 
      using that
      by (simp add: sin-arccos-abs [OF cos-x-y-le-one] power-divide right-diff-distrib
flip: real-sqrt-mult)
    ultimately show ?thesis
      by blast
  qed
  show ?thesis
  proof (cases  $0::\operatorname{real} y$  rule: linorder-cases)
    case less
      then show ?thesis
        by (rule polar-ex1)
    next
      case equal
      then show ?thesis

```

```

    by (force simp: intro!: cos-zero sin-zero)
  next
    case greater
    with polar-ex1 [where y=-y] show ?thesis
      by auto (metis cos-minus minus-minus minus-mult-right sin-minus)
  qed
qed

```

113.22 Basics about polynomial functions: products, extremal behaviour and root counts

```

lemma pairs-le-eq-Sigma:  $\{(i, j). i + j \leq m\} = \text{Sigma } (\text{atMost } m) (\lambda r. \text{atMost } (m - r))$ 
  for  $m :: \text{nat}$ 
  by auto

```

```

lemma sum-up-index-split:  $(\sum_{k \leq m + n} f k) = (\sum_{k \leq m} f k) + (\sum_{k = \text{Suc } m..m + n} f k)$ 
  by (metis atLeast0AtMost Suc-eq-plus1 le0 sum.ub-add-nat)

```

```

lemma Sigma-interval-disjoint:  $(\text{SIGMA } i:A. \{..v\ i\}) \cap (\text{SIGMA } i:A. \{v\ i<..w\}) = \{\}$ 
  for  $w :: 'a::\text{order}$ 
  by auto

```

```

lemma product-atMost-eq-Un:  $A \times \{..m\} = (\text{SIGMA } i:A. \{..m - i\}) \cup (\text{SIGMA } i:A. \{m - i<..m\})$ 
  for  $m :: \text{nat}$ 
  by auto

```

```

lemma polynomial-product:
  fixes  $x :: 'a::\text{idom}$ 
  assumes  $m: \bigwedge i. i > m \implies a\ i = 0$ 
  and  $n: \bigwedge j. j > n \implies b\ j = 0$ 
  shows  $(\sum_{i \leq m}. (a\ i) * x^i) * (\sum_{j \leq n}. (b\ j) * x^j) = (\sum_{r \leq m + n}. (\sum_{k \leq r}. (a\ k) * (b\ (r - k)))) * x^r$ 
proof -
  have  $\bigwedge i\ j. \llbracket m + n - i < j; a\ i \neq 0 \rrbracket \implies b\ j = 0$ 
    by (meson le-add-diff leI le-less-trans m n)
  then have  $\S: (\sum_{(i,j) \in (\text{SIGMA } i:\{..m+n\}. \{m+n - i<..m+n\})}. a\ i * x^i * (b\ j * x^j)) = 0$ 
    by (clarsimp simp add: sum-Un Sigma-interval-disjoint intro!: sum.neutral)
  have  $(\sum_{i \leq m}. (a\ i) * x^i) * (\sum_{j \leq n}. (b\ j) * x^j) = (\sum_{i \leq m}. \sum_{j \leq n}. (a\ i * x^i * (b\ j * x^j)))$ 
    by (rule sum-product)
  also have  $\dots = (\sum_{i \leq m + n}. \sum_{j \leq n + m}. a\ i * x^i * (b\ j * x^j))$ 
    using assms by (auto simp: sum-up-index-split)
  also have  $\dots = (\sum_{r \leq m + n}. \sum_{j \leq m + n - r}. a\ r * x^r * (b\ j * x^j))$ 
    by (simp add: add-ac sum.Sigma product-atMost-eq-Un sum-Un Sigma-interval-disjoint)

```

§)

also have $\dots = (\sum (i,j) \in \{(i,j). i+j \leq m+n\}. (a \ i * x \wedge i) * (b \ j * x \wedge j))$
by (*auto simp: pairs-le-eq-Sigma sum.Sigma*)
also have $\dots = (\sum k \leq m + n. \sum i \leq k. a \ i * x \wedge i * (b \ (k - i) * x \wedge (k - i)))$
by (*rule sum.triangle-reindex-eq*)
also have $\dots = (\sum r \leq m + n. (\sum k \leq r. (a \ k) * (b \ (r - k))) * x \wedge r)$
by (*auto simp: algebra-simps sum-distrib-left simp flip: power-add intro!: sum.cong*)
finally show *?thesis* .
qed

lemma *polynomial-product-nat*:

fixes $x :: nat$
assumes $m: \bigwedge i. i > m \implies a \ i = 0$
and $n: \bigwedge j. j > n \implies b \ j = 0$
shows $(\sum i \leq m. (a \ i) * x \wedge i) * (\sum j \leq n. (b \ j) * x \wedge j) =$
 $(\sum r \leq m + n. (\sum k \leq r. (a \ k) * (b \ (r - k)))) * x \wedge r$
using *polynomial-product* [*of m a n b x*] *assms*
by (*simp only: of-nat-mult* [*symmetric*] *of-nat-power* [*symmetric*] *of-nat-eq-iff Int.int-sum* [*symmetric*])

lemma *polyfun-diff*:

fixes $x :: 'a::idom$
assumes $1 \leq n$
shows $(\sum i \leq n. a \ i * x \wedge i) - (\sum i \leq n. a \ i * y \wedge i) =$
 $(x - y) * (\sum j < n. (\sum i = Suc \ j..n. a \ i * y \wedge (i - j - 1)) * x \wedge j)$
proof –
have $h: \text{bij-betw } (\lambda(i,j). (j,i)) ((SIGMA \ i : \text{atMost } n. \text{lessThan } i)) (SIGMA \ j :$
 $\text{lessThan } n. \{Suc \ j..n\})$
by (*auto simp: bij-betw-def inj-on-def*)
have $(\sum i \leq n. a \ i * x \wedge i) - (\sum i \leq n. a \ i * y \wedge i) = (\sum i \leq n. a \ i * (x \wedge i - y \wedge i))$
by (*simp add: right-diff-distrib sum-subtractf*)
also have $\dots = (\sum i \leq n. a \ i * (x - y) * (\sum j < i. y \wedge (i - Suc \ j) * x \wedge j))$
by (*simp add: power-diff-sumr2 mult.assoc*)
also have $\dots = (\sum i \leq n. \sum j < i. a \ i * (x - y) * (y \wedge (i - Suc \ j) * x \wedge j))$
by (*simp add: sum-distrib-left*)
also have $\dots = (\sum (i,j) \in (SIGMA \ i : \text{atMost } n. \text{lessThan } i). a \ i * (x - y) *$
 $(y \wedge (i - Suc \ j) * x \wedge j))$
by (*simp add: sum.Sigma*)
also have $\dots = (\sum (j,i) \in (SIGMA \ j : \text{lessThan } n. \{Suc \ j..n\}). a \ i * (x - y) *$
 $(y \wedge (i - Suc \ j) * x \wedge j))$
by (*auto simp: sum.reindex-bij-betw* [*OF h, symmetric*] *intro: sum.cong-simp*)
also have $\dots = (\sum j < n. \sum i = Suc \ j..n. a \ i * (x - y) * (y \wedge (i - Suc \ j) * x \wedge j))$
by (*simp add: sum.Sigma*)
also have $\dots = (x - y) * (\sum j < n. (\sum i = Suc \ j..n. a \ i * y \wedge (i - j - 1)) * x \wedge j)$
by (*simp add: sum-distrib-left mult-ac*)
finally show *?thesis* .
qed

lemma *polyfun-diff-alt*:

fixes $x :: 'a::idom$
assumes $1 \leq n$
shows $(\sum_{i \leq n}. a \ i * x \widehat{i}) - (\sum_{i \leq n}. a \ i * y \widehat{i}) =$
 $(x - y) * ((\sum_{j < n}. \sum_{k < n-j}. a(j+k+1) * y \widehat{k} * x \widehat{j}))$
proof –
have $(\sum_{i = Suc \ j..n}. a \ i * y \widehat{(i-j-1)}) = (\sum_{k < n-j}. a(j+k+1) * y \widehat{k})$
if $j < n$ **for** $j :: nat$
proof –
have $\bigwedge k. k < n - j \implies k \in (\lambda i. i - Suc \ j) \ ` \ {Suc \ j..n}$
by $(rule-tac \ x=k + Suc \ j \ in \ image-eqI, \ auto)$
then have $h: \text{bij-betw } (\lambda i. i - (j + 1)) \ \{Suc \ j..n\} \ (\text{lessThan } (n-j))$
by $(auto \ simp: \ \text{bij-betw-def} \ inj-on-def)$
then show $?thesis$
by $(auto \ simp: \ \text{sum.reindex-bij-betw} \ [OF \ h, \ symmetric] \ intro: \ \text{sum.cong-simp})$
qed
then show $?thesis$
by $(simp \ add: \ \text{polyfun-diff} \ [OF \ assms] \ \text{sum-distrib-right})$
qed

lemma *polyfun-linear-factor*:

fixes $a :: 'a::idom$
shows $\exists b. \forall z. (\sum_{i \leq n}. c(i) * z \widehat{i}) = (z - a) * (\sum_{i < n}. b(i) * z \widehat{i}) + (\sum_{i \leq n}. c(i) * a \widehat{i})$
proof $(cases \ n = 0)$
case *True* **then show** $?thesis$
by *simp*
next
case *False*
have $(\exists b. \forall z. (\sum_{i \leq n}. c \ i * z \widehat{i}) = (z - a) * (\sum_{i < n}. b \ i * z \widehat{i}) + (\sum_{i \leq n}. c \ i * a \widehat{i})) \longleftrightarrow$
 $(\exists b. \forall z. (\sum_{i \leq n}. c \ i * z \widehat{i}) - (\sum_{i \leq n}. c \ i * a \widehat{i}) = (z - a) * (\sum_{i < n}. b \ i * z \widehat{i}))$
by $(simp \ add: \ \text{algebra-simps})$
also have $\dots \longleftrightarrow$
 $(\exists b. \forall z. (z - a) * (\sum_{j < n}. (\sum_{i = Suc \ j..n}. c \ i * a \widehat{(i - Suc \ j)}) * z \widehat{j}) =$
 $(z - a) * (\sum_{i < n}. b \ i * z \widehat{i}))$
using *False* **by** $(simp \ add: \ \text{polyfun-diff})$
also have $\dots = True$ **by** *auto*
finally show $?thesis$
by *simp*
qed

lemma *polyfun-linear-factor-root*:

fixes $a :: 'a::idom$
assumes $(\sum_{i \leq n}. c(i) * a \widehat{i}) = 0$
obtains b **where** $\bigwedge z. (\sum_{i \leq n}. c \ i * z \widehat{i}) = (z - a) * (\sum_{i < n}. b \ i * z \widehat{i})$
using *polyfun-linear-factor* $[of \ c \ n \ a]$ **assms** **by** *auto*

lemma *isCont-polynom*: *isCont* $(\lambda w. \sum_{i \leq n}. c\ i * w^{\wedge}i)$ *a*
for $c :: \text{nat} \Rightarrow 'a::\text{real-normed-div-algebra}$
by *simp*

lemma *zero-polynom-imp-zero-coeffs*:

fixes $c :: \text{nat} \Rightarrow 'a::\{\text{ab-semigroup-mult}, \text{real-normed-div-algebra}\}$
assumes $\bigwedge w. (\sum_{i \leq n}. c\ i * w^{\wedge}i) = 0 \quad k \leq n$
shows $c\ k = 0$

using *assms*

proof (*induction n arbitrary: c k*)

case 0

then show *?case*

by *simp*

next

case (*Suc n c k*)

have [*simp*]: $c\ 0 = 0$ **using** *Suc.prem1* [*of 0*]

by *simp*

have $(\sum_{i \leq \text{Suc } n}. c\ i * w^{\wedge}i) = w * (\sum_{i \leq n}. c\ (\text{Suc } i) * w^{\wedge}i)$ **for** *w*

proof –

have $(\sum_{i \leq \text{Suc } n}. c\ i * w^{\wedge}i) = (\sum_{i \leq n}. c\ (\text{Suc } i) * w^{\wedge} \text{Suc } i)$

unfolding *Set-Interval.sum.atMost-Suc-shift*

by *simp*

also have $\dots = w * (\sum_{i \leq n}. c\ (\text{Suc } i) * w^{\wedge}i)$

by (*simp add: sum-distrib-left ac-simps*)

finally show *?thesis* .

qed

then have $w: \bigwedge w. w \neq 0 \implies (\sum_{i \leq n}. c\ (\text{Suc } i) * w^{\wedge}i) = 0$

using *Suc* **by** *auto*

then have $(\lambda h. \sum_{i \leq n}. c\ (\text{Suc } i) * h^{\wedge}i) - 0 \rightarrow 0$

by (*simp cong: LIM-cong*) — the case $w = 0$ by continuity

then have $(\sum_{i \leq n}. c\ (\text{Suc } i) * 0^{\wedge}i) = 0$

using *isCont-polynom* [*of 0* $\lambda i. c\ (\text{Suc } i)\ n$] *LIM-unique*

by (*force simp: Limits.isCont-iff*)

then have $\bigwedge w. (\sum_{i \leq n}. c\ (\text{Suc } i) * w^{\wedge}i) = 0$

using *w* **by** *metis*

then have $\bigwedge i. i \leq n \implies c\ (\text{Suc } i) = 0$

using *Suc.IH* [*of* $\lambda i. c\ (\text{Suc } i)$] **by** *blast*

then show *?case* **using** $\langle k \leq \text{Suc } n \rangle$

by (*cases k*) *auto*

qed

lemma *polyfun-rootbound*:

fixes $c :: \text{nat} \Rightarrow 'a::\{\text{idom}, \text{real-normed-div-algebra}\}$

assumes $c\ k \neq 0 \quad k \leq n$

shows *finite* $\{z. (\sum_{i \leq n}. c(i) * z^{\wedge}i) = 0\} \wedge \text{card } \{z. (\sum_{i \leq n}. c(i) * z^{\wedge}i) = 0\}$
 $\leq n$

using *assms*

proof (*induction n arbitrary: c k*)

```

case 0
then show ?case
  by simp
next
case (Suc m c k)
let ?succase = ?case
show ?case
proof (cases {z. ( $\sum_{i \leq \text{Suc } m}. c(i) * z^{\widehat{i}} = 0$ ) = {}})
  case True
  then show ?succase
    by simp
  next
  case False
  then obtain z0 where z0: ( $\sum_{i \leq \text{Suc } m}. c(i) * z0^{\widehat{i}} = 0$ )
    by blast
  then obtain b where b:  $\bigwedge w. (\sum_{i \leq \text{Suc } m}. c i * w^{\widehat{i}}) = (w - z0) * (\sum_{i \leq m}. b i * w^{\widehat{i}})$ 
    using polyfun-linear-factor-root [OF z0, unfolded lessThan-Suc-atMost]
    by blast
  then have eq: {z. ( $\sum_{i \leq \text{Suc } m}. c i * z^{\widehat{i}} = 0$ )} = insert z0 {z. ( $\sum_{i \leq m}. b i * z^{\widehat{i}} = 0$ )}
    by auto
  have  $\neg (\forall k \leq m. b k = 0)$ 
  proof
    assume [simp]:  $\forall k \leq m. b k = 0$ 
    then have  $\bigwedge w. (\sum_{i \leq m}. b i * w^{\widehat{i}}) = 0$ 
      by simp
    then have  $\bigwedge w. (\sum_{i \leq \text{Suc } m}. c i * w^{\widehat{i}}) = 0$ 
      using b by simp
    then have  $\bigwedge k. k \leq \text{Suc } m \implies c k = 0$ 
      using zero-polynom-imp-zero-coeffs by blast
    then show False using Suc.prem by blast
  qed
  then obtain k' where bk':  $b k' \neq 0 \wedge k' \leq m$ 
    by blast
  show ?succase
    using Suc.IH [of b k'] bk'
    by (simp add: eq card-insert-if del: sum.atMost-Suc)
  qed
qed

```

lemma

```

fixes c :: nat  $\Rightarrow$  'a::{idom,real-normed-div-algebra}
assumes c k  $\neq 0 \wedge k \leq n$ 
shows polyfun-roots-finite: finite {z. ( $\sum_{i \leq n}. c(i) * z^{\widehat{i}} = 0$ )}
  and polyfun-roots-card: card {z. ( $\sum_{i \leq n}. c(i) * z^{\widehat{i}} = 0$ )}  $\leq n$ 
using polyfun-rootbound assms by auto

```

lemma polyfun-finite-roots:

fixes $c :: \text{nat} \Rightarrow 'a::\{\text{idom,real-normed-div-algebra}\}$
shows $\text{finite } \{x. (\sum_{i \leq n}. c\ i * x^{\widehat{i}}) = 0\} \longleftrightarrow (\exists i \leq n. c\ i \neq 0)$
 (is ?lhs = ?rhs)
proof
 assume ?lhs
moreover have $\neg \text{finite } \{x. (\sum_{i \leq n}. c\ i * x^{\widehat{i}}) = 0\}$ **if** $\forall i \leq n. c\ i = 0$
proof –
 from that **have** $\bigwedge x. (\sum_{i \leq n}. c\ i * x^{\widehat{i}}) = 0$
 by simp
 then **show** ?thesis
 using ex-new-if-finite [OF infinite-UNIV-char-0 [where 'a='a]]
 by auto
qed
ultimately show ?rhs **by** metis
next
 assume ?rhs
with polyfun-rootbound **show** ?lhs **by** blast
qed

lemma polyfun-eq-0: $(\forall x. (\sum_{i \leq n}. c\ i * x^{\widehat{i}}) = 0) \longleftrightarrow (\forall i \leq n. c\ i = 0)$
for $c :: \text{nat} \Rightarrow 'a::\{\text{idom,real-normed-div-algebra}\}$

using zero-polynom-imp-zero-coeffs **by** auto

lemma polyfun-eq-coeffs: $(\forall x. (\sum_{i \leq n}. c\ i * x^{\widehat{i}}) = (\sum_{i \leq n}. d\ i * x^{\widehat{i}})) \longleftrightarrow$
 $(\forall i \leq n. c\ i = d\ i)$

for $c :: \text{nat} \Rightarrow 'a::\{\text{idom,real-normed-div-algebra}\}$

proof –
have $(\forall x. (\sum_{i \leq n}. c\ i * x^{\widehat{i}}) = (\sum_{i \leq n}. d\ i * x^{\widehat{i}})) \longleftrightarrow (\forall x. (\sum_{i \leq n}. (c\ i - d\ i) * x^{\widehat{i}}) = 0)$
by (simp add: left-diff-distrib Groups-Big.sum-subtractf)
also have $\dots \longleftrightarrow (\forall i \leq n. c\ i - d\ i = 0)$
by (rule polyfun-eq-0)
finally show ?thesis
by simp
qed

lemma polyfun-eq-const:

fixes $c :: \text{nat} \Rightarrow 'a::\{\text{idom,real-normed-div-algebra}\}$

shows $(\forall x. (\sum_{i \leq n}. c\ i * x^{\widehat{i}}) = k) \longleftrightarrow c\ 0 = k \wedge (\forall i \in \{1..n\}. c\ i = 0)$
 (is ?lhs = ?rhs)

proof –
have $*$: $\forall x. (\sum_{i \leq n}. (\text{if } i=0 \text{ then } k \text{ else } 0) * x^{\widehat{i}}) = k$
by (induct n) auto
show ?thesis
proof
 assume ?lhs
with $*$ **have** $(\forall i \leq n. c\ i = (\text{if } i=0 \text{ then } k \text{ else } 0))$
by (simp add: polyfun-eq-coeffs [symmetric])

```

  then show ?rhs by simp
next
  assume ?rhs
  then show ?lhs by (induct n) auto
qed
qed

```

lemma *root-polyfun*:

```

  fixes z :: 'a::idom
  assumes 1 ≤ n
  shows z^n = a ↔ (∑ i≤n. (if i = 0 then -a else if i=n then 1 else 0) * z^i)
= 0
  using assms by (cases n) (simp-all add: sum.atLeast-Suc-atMost atLeast0AtMost
[symmetric])

```

lemma

```

  assumes SORT-CONSTRAINT('a::{idom,real-normed-div-algebra})
  and 1 ≤ n
  shows finite-roots-unity: finite {z::'a. z^n = 1}
  and card-roots-unity: card {z::'a. z^n = 1} ≤ n
  using polyfun-rootbound [of λi. if i = 0 then -1 else if i=n then 1 else 0 n n]
assms(2)
  by (auto simp: root-polyfun [OF assms(2)])

```

113.23 Hyperbolic functions

definition *sinh* :: 'a :: {*banach, real-normed-algebra-1*} ⇒ 'a **where**
 $\sinh x = (\exp x - \exp (-x)) /_R 2$

definition *cosh* :: 'a :: {*banach, real-normed-algebra-1*} ⇒ 'a **where**
 $\cosh x = (\exp x + \exp (-x)) /_R 2$

definition *tanh* :: 'a :: {*banach, real-normed-field*} ⇒ 'a **where**
 $\tanh x = \sinh x / \cosh x$

definition *arsinh* :: 'a :: {*banach, real-normed-algebra-1, ln*} ⇒ 'a **where**
 $\operatorname{arsinh} x = \ln (x + (x^2 + 1) \operatorname{powr} \text{of-real} (1/2))$

definition *arcosh* :: 'a :: {*banach, real-normed-algebra-1, ln*} ⇒ 'a **where**
 $\operatorname{arcosh} x = \ln (x + (x^2 - 1) \operatorname{powr} \text{of-real} (1/2))$

definition *artanh* :: 'a :: {*banach, real-normed-field, ln*} ⇒ 'a **where**
 $\operatorname{artanh} x = \ln ((1 + x) / (1 - x)) / 2$

lemma *arsinh-0* [*simp*]: $\operatorname{arsinh} 0 = 0$
 by (*simp* add: *arsinh-def*)

lemma *arcosh-1* [*simp*]: $\operatorname{arcosh} 1 = 0$
 by (*simp* add: *arcosh-def*)

lemma *artanh-0* [*simp*]: *artanh* 0 = 0
 by (*simp* add: *artanh-def*)

lemma *tanh-altdef*:
 $\tanh x = (\exp x - \exp (-x)) / (\exp x + \exp (-x))$
proof –
 have $\tanh x = (2 *_{\mathbb{R}} \sinh x) / (2 *_{\mathbb{R}} \cosh x)$
 by (*simp* add: *tanh-def scaleR-conv-of-real*)
 also have $2 *_{\mathbb{R}} \sinh x = \exp x - \exp (-x)$
 by (*simp* add: *sinh-def*)
 also have $2 *_{\mathbb{R}} \cosh x = \exp x + \exp (-x)$
 by (*simp* add: *cosh-def*)
 finally show ?thesis .
qed

lemma *tanh-real-altdef*: $\tanh (x::\text{real}) = (1 - \exp (-2 * x)) / (1 + \exp (-2 * x))$
proof –
 have [*simp*]: $\exp (2 * x) = \exp x * \exp x$
 by (*subst* *exp-add* [*symmetric*]; *simp*)
 have $\tanh x = (2 * \exp (-x) * \sinh x) / (2 * \exp (-x) * \cosh x)$
 by (*simp* add: *tanh-def*)
 also have $2 * \exp (-x) * \sinh x = 1 - \exp (-2*x)$
 by (*simp* add: *exp-minus field-simps sinh-def*)
 also have $2 * \exp (-x) * \cosh x = 1 + \exp (-2*x)$
 by (*simp* add: *exp-minus field-simps cosh-def*)
 finally show ?thesis .
qed

lemma *sinh-converges*: $(\lambda n. \text{if even } n \text{ then } 0 \text{ else } x^n /_{\mathbb{R}} \text{fact } n)$ sums *sinh* *x*
proof –
 have $(\lambda n. (x^n /_{\mathbb{R}} \text{fact } n - (-x)^n /_{\mathbb{R}} \text{fact } n) /_{\mathbb{R}} 2)$ sums *sinh* *x*
 unfolding *sinh-def* by (*intro* *sums-scaleR-right* *sums-diff* *exp-converges*)
 also have $(\lambda n. (x^n /_{\mathbb{R}} \text{fact } n - (-x)^n /_{\mathbb{R}} \text{fact } n) /_{\mathbb{R}} 2) =$
 $(\lambda n. \text{if even } n \text{ then } 0 \text{ else } x^n /_{\mathbb{R}} \text{fact } n)$ by *auto*
 finally show ?thesis .
qed

lemma *cosh-converges*: $(\lambda n. \text{if even } n \text{ then } x^n /_{\mathbb{R}} \text{fact } n \text{ else } 0)$ sums *cosh* *x*
proof –
 have $(\lambda n. (x^n /_{\mathbb{R}} \text{fact } n + (-x)^n /_{\mathbb{R}} \text{fact } n) /_{\mathbb{R}} 2)$ sums *cosh* *x*
 unfolding *cosh-def* by (*intro* *sums-scaleR-right* *sums-add* *exp-converges*)
 also have $(\lambda n. (x^n /_{\mathbb{R}} \text{fact } n + (-x)^n /_{\mathbb{R}} \text{fact } n) /_{\mathbb{R}} 2) =$
 $(\lambda n. \text{if even } n \text{ then } x^n /_{\mathbb{R}} \text{fact } n \text{ else } 0)$ by *auto*
 finally show ?thesis .
qed

lemma *sinh-0* [*simp*]: $\sinh 0 = 0$
 by (*simp add: sinh-def*)

lemma *cosh-0* [*simp*]: $\cosh 0 = 1$
proof –

have $\cosh 0 = (1/2) *_{\mathbb{R}} (1 + 1)$ **by** (*simp add: cosh-def*)
 also have $\dots = 1$ **by** (*rule scaleR-half-double*)
 finally show *?thesis* .

qed

lemma *tanh-0* [*simp*]: $\tanh 0 = 0$
 by (*simp add: tanh-def*)

lemma *sinh-minus* [*simp*]: $\sinh (-x) = -\sinh x$
 by (*simp add: sinh-def algebra-simps*)

lemma *cosh-minus* [*simp*]: $\cosh (-x) = \cosh x$
 by (*simp add: cosh-def algebra-simps*)

lemma *tanh-minus* [*simp*]: $\tanh (-x) = -\tanh x$
 by (*simp add: tanh-def*)

lemma *sinh-ln-real*: $x > 0 \implies \sinh (\ln x :: \text{real}) = (x - \text{inverse } x) / 2$
 by (*simp add: sinh-def exp-minus*)

lemma *cosh-ln-real*: $x > 0 \implies \cosh (\ln x :: \text{real}) = (x + \text{inverse } x) / 2$
 by (*simp add: cosh-def exp-minus*)

lemma *tanh-ln-real*:

$\tanh (\ln x :: \text{real}) = (x^2 - 1) / (x^2 + 1)$ **if** $x > 0$

proof –

from that have $(x * 2 - \text{inverse } x * 2) * (x^2 + 1) =$
 $(x^2 - 1) * (2 * x + 2 * \text{inverse } x)$

by (*simp add: field-simps power2-eq-square*)

moreover have $x^2 + 1 > 0$

using that by (*simp add: ac-simps add-pos-nonneg*)

moreover have $2 * x + 2 * \text{inverse } x > 0$

using that by (*simp add: add-pos-pos*)

ultimately have $(x * 2 - \text{inverse } x * 2) /$

$(2 * x + 2 * \text{inverse } x) =$

$(x^2 - 1) / (x^2 + 1)$

by (*simp add: frac-eq-eq*)

with that show *?thesis*

by (*simp add: tanh-def sinh-ln-real cosh-ln-real*)

qed

lemma *has-field-derivative-scaleR-right* [*derivative-intros*]:

$(f \text{ has-field-derivative } D) F \implies ((\lambda x. c *_{\mathbb{R}} f x) \text{ has-field-derivative } (c *_{\mathbb{R}} D)) F$

unfolding *has-field-derivative-def*

using *has-derivative-scaleR-right*[of $f \lambda x. D * x F c$]
by (*simp add: mult-scaleR-left [symmetric] del: mult-scaleR-left*)

lemma *has-field-derivative-sinh* [*THEN DERIV-chain2, derivative-intros*]:
 (*sinh has-field-derivative cosh x*) (at ($x :: 'a :: \{\text{banach, real-normed-field}\}$))
unfolding *sinh-def cosh-def* **by** (*auto intro!: derivative-eq-intros*)

lemma *has-field-derivative-cosh* [*THEN DERIV-chain2, derivative-intros*]:
 (*cosh has-field-derivative sinh x*) (at ($x :: 'a :: \{\text{banach, real-normed-field}\}$))
unfolding *sinh-def cosh-def* **by** (*auto intro!: derivative-eq-intros*)

lemma *has-field-derivative-tanh* [*THEN DERIV-chain2, derivative-intros*]:
 (*cosh x \neq 0 \implies (tanh has-field-derivative $1 - \tanh x^2$)*)
 (at ($x :: 'a :: \{\text{banach, real-normed-field}\}$))
unfolding *tanh-def* **by** (*auto intro!: derivative-eq-intros simp: power2-eq-square field-split-simps*)

lemma *has-derivative-sinh* [*derivative-intros*]:
fixes $g :: 'a \Rightarrow ('a :: \{\text{banach, real-normed-field}\})$
assumes (g *has-derivative* ($\lambda x. Db * x$)) (at x within s)
shows ($\lambda x. \sinh (g x)$ *has-derivative* ($\lambda y. (\cosh (g x) * Db) * y$)) (at x within s)
proof –
have ($\lambda x. -g x$ *has-derivative* ($\lambda y. -(Db * y)$)) (at x within s)
using *assms* **by** (*intro derivative-intros*)
also have ($\lambda y. -(Db * y) = (\lambda x. (-Db) * x$) **by** (*simp add: fun-eq-iff*)
finally have ($\lambda x. \sinh (g x)$ *has-derivative*
 ($\lambda y. (\exp (g x) * Db * y - \exp (-g x) * (-Db) * y) /_R 2$)) (at x within s)
unfolding *sinh-def* **by** (*intro derivative-intros assms*)
also have ($\lambda y. (\exp (g x) * Db * y - \exp (-g x) * (-Db) * y) /_R 2 = (\lambda y.$
 ($\cosh (g x) * Db) * y$)
by (*simp add: fun-eq-iff cosh-def algebra-simps*)
finally show *?thesis* .
qed

lemma *has-derivative-cosh* [*derivative-intros*]:
fixes $g :: 'a \Rightarrow ('a :: \{\text{banach, real-normed-field}\})$
assumes (g *has-derivative* ($\lambda y. Db * y$)) (at x within s)
shows ($\lambda x. \cosh (g x)$ *has-derivative* ($\lambda y. (\sinh (g x) * Db) * y$)) (at x within s)
proof –
have ($\lambda x. -g x$ *has-derivative* ($\lambda y. -(Db * y)$)) (at x within s)
using *assms* **by** (*intro derivative-intros*)
also have ($\lambda y. -(Db * y) = (\lambda y. (-Db) * y$) **by** (*simp add: fun-eq-iff*)
finally have ($\lambda x. \cosh (g x)$ *has-derivative*
 ($\lambda y. (\exp (g x) * Db * y + \exp (-g x) * (-Db) * y) /_R 2$)) (at x within s)
unfolding *cosh-def* **by** (*intro derivative-intros assms*)
also have ($\lambda y. (\exp (g x) * Db * y + \exp (-g x) * (-Db) * y) /_R 2 = (\lambda y.$
 ($\sinh (g x) * Db) * y$)

by (*simp add: fun-eq-iff sinh-def algebra-simps*)
 finally show ?thesis .
 qed

lemma *sinh-plus-cosh*: $\sinh x + \cosh x = \exp x$

proof –
 have $\sinh x + \cosh x = (1/2) *_{\mathbb{R}} (\exp x + \exp x)$
 by (*simp add: sinh-def cosh-def algebra-simps*)
 also have $\dots = \exp x$ by (*rule scaleR-half-double*)
 finally show ?thesis .
 qed

lemma *cosh-plus-sinh*: $\cosh x + \sinh x = \exp x$

by (*subst add.commute*) (*rule sinh-plus-cosh*)

lemma *cosh-minus-sinh*: $\cosh x - \sinh x = \exp (-x)$

proof –
 have $\cosh x - \sinh x = (1/2) *_{\mathbb{R}} (\exp (-x) + \exp (-x))$
 by (*simp add: sinh-def cosh-def algebra-simps*)
 also have $\dots = \exp (-x)$ by (*rule scaleR-half-double*)
 finally show ?thesis .
 qed

lemma *sinh-minus-cosh*: $\sinh x - \cosh x = -\exp (-x)$

using *cosh-minus-sinh*[of x] by (*simp add: algebra-simps*)

context

fixes $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$

begin

lemma *sinh-zero-iff*: $\sinh x = 0 \iff \exp x \in \{1, -1\}$

by (*auto simp: sinh-def field-simps exp-minus power2-eq-square square-eq-1-iff*)

lemma *cosh-zero-iff*: $\cosh x = 0 \iff \exp x^2 = -1$

by (*auto simp: cosh-def exp-minus field-simps power2-eq-square eq-neg-iff-add-eq-0*)

lemma *cosh-square-eq*: $\cosh x^2 = \sinh x^2 + 1$

by (*simp add: cosh-def sinh-def algebra-simps power2-eq-square exp-add [symmetric] scaleR-conv-of-real*)

lemma *sinh-square-eq*: $\sinh x^2 = \cosh x^2 - 1$

by (*simp add: cosh-square-eq*)

lemma *hyperbolic-pythagoras*: $\cosh x^2 - \sinh x^2 = 1$

by (*simp add: cosh-square-eq*)

lemma *sinh-add*: $\sinh (x + y) = \sinh x * \cosh y + \cosh x * \sinh y$

by (*simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric]*)

lemma *sinh-diff*: $\sinh (x - y) = \sinh x * \cosh y - \cosh x * \sinh y$
by (*simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric]*)

lemma *cosh-add*: $\cosh (x + y) = \cosh x * \cosh y + \sinh x * \sinh y$
by (*simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric]*)

lemma *cosh-diff*: $\cosh (x - y) = \cosh x * \cosh y - \sinh x * \sinh y$
by (*simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric]*)

lemma *tanh-add*:

$$\tanh (x + y) = (\tanh x + \tanh y) / (1 + \tanh x * \tanh y)$$

if $\cosh x \neq 0 \ \cosh y \neq 0$

proof –

have $(\sinh x * \cosh y + \cosh x * \sinh y) * (1 + \sinh x * \sinh y / (\cosh x * \cosh y)) =$

$(\cosh x * \cosh y + \sinh x * \sinh y) * ((\sinh x * \cosh y + \sinh y * \cosh x) / (\cosh y * \cosh x))$

using that by (*simp add: field-split-simps*)

also have $(\sinh x * \cosh y + \sinh y * \cosh x) / (\cosh y * \cosh x) = \sinh x / \cosh x + \sinh y / \cosh y$

using that by (*simp add: field-split-simps*)

finally have $(\sinh x * \cosh y + \cosh x * \sinh y) * (1 + \sinh x * \sinh y / (\cosh x * \cosh y)) =$

$(\sinh x / \cosh x + \sinh y / \cosh y) * (\cosh x * \cosh y + \sinh x * \sinh y)$

by simp

then show *?thesis*

using that by (*auto simp add: tanh-def sinh-add cosh-add eq-divide-eq*)

(*simp-all add: field-split-simps*)

qed

lemma *sinh-double*: $\sinh (2 * x) = 2 * \sinh x * \cosh x$
using *sinh-add[of x]* **by** *simp*

lemma *cosh-double*: $\cosh (2 * x) = \cosh x ^ 2 + \sinh x ^ 2$
using *cosh-add[of x]* **by** (*simp add: power2-eq-square*)

end

lemma *sinh-field-def*: $\sinh z = (\exp z - \exp (-z)) / (2 :: 'a :: \{\text{banach, real-normed-field}\})$
by (*simp add: sinh-def scaleR-conv-of-real*)

lemma *cosh-field-def*: $\cosh z = (\exp z + \exp (-z)) / (2 :: 'a :: \{\text{banach, real-normed-field}\})$
by (*simp add: cosh-def scaleR-conv-of-real*)

113.23.1 More specific properties of the real functions

lemma *plus-inverse-ge-2*:
fixes $x :: \text{real}$

assumes $x > 0$
shows $x + \text{inverse } x \geq 2$
proof –
have $0 \leq (x - 1)^2$ **by** *simp*
also have $\dots = x^2 - 2*x + 1$ **by** (*simp add: power2-eq-square algebra-simps*)
finally show *?thesis* **using** *assms* **by** (*simp add: field-simps power2-eq-square*)
qed

lemma *sinh-real-nonneg-iff* [*simp*]: $\text{sinh } (x :: \text{real}) \geq 0 \iff x \geq 0$
by (*simp add: sinh-def*)

lemma *sinh-real-pos-iff* [*simp*]: $\text{sinh } (x :: \text{real}) > 0 \iff x > 0$
by (*simp add: sinh-def*)

lemma *sinh-real-nonpos-iff* [*simp*]: $\text{sinh } (x :: \text{real}) \leq 0 \iff x \leq 0$
by (*simp add: sinh-def*)

lemma *sinh-real-neg-iff* [*simp*]: $\text{sinh } (x :: \text{real}) < 0 \iff x < 0$
by (*simp add: sinh-def*)

lemma *cosh-real-ge-1*: $\text{cosh } (x :: \text{real}) \geq 1$
using *plus-inverse-ge-2*[*of exp x*] **by** (*simp add: cosh-def exp-minus*)

lemma *cosh-real-pos* [*simp*]: $\text{cosh } (x :: \text{real}) > 0$
using *cosh-real-ge-1*[*of x*] **by** *simp*

lemma *cosh-real-nonneg*[*simp*]: $\text{cosh } (x :: \text{real}) \geq 0$
using *cosh-real-ge-1*[*of x*] **by** *simp*

lemma *cosh-real-nonzero* [*simp*]: $\text{cosh } (x :: \text{real}) \neq 0$
using *cosh-real-ge-1*[*of x*] **by** *simp*

lemma *arsinh-real-def*: $\text{arsinh } (x :: \text{real}) = \ln (x + \text{sqrt } (x^2 + 1))$
by (*simp add: arsinh-def powr-half-sqrt*)

lemma *arcosh-real-def*: $x \geq 1 \implies \text{arcosh } (x :: \text{real}) = \ln (x + \text{sqrt } (x^2 - 1))$
by (*simp add: arcosh-def powr-half-sqrt*)

lemma *arsinh-real-aux*: $0 < x + \text{sqrt } (x^2 + 1) :: \text{real}$

proof (*cases x < 0*)
case *True*
have $(-x)^2 = x^2$ **by** *simp*
also have $x^2 < x^2 + 1$ **by** *simp*
finally have $\text{sqrt } ((-x)^2) < \text{sqrt } (x^2 + 1)$
by (*rule real-sqrt-less-mono*)
thus *?thesis* **using** *True* **by** *simp*
qed (*auto simp: add-nonneg-pos*)

lemma *arsinh-minus-real* [*simp*]: $\text{arsinh } (-x :: \text{real}) = -\text{arsinh } x$

proof –

have $\operatorname{arsinh}(-x) = \ln(\operatorname{sqrt}(x^2 + 1) - x)$
by (*simp add: arsinh-real-def*)
also have $\operatorname{sqrt}(x^2 + 1) - x = \operatorname{inverse}(\operatorname{sqrt}(x^2 + 1) + x)$
using *arsinh-real-aux*[of x] **by** (*simp add: field-split-simps algebra-simps power2-eq-square*)
also have $\ln \dots = -\operatorname{arsinh} x$
using *arsinh-real-aux*[of x] **by** (*simp add: arsinh-real-def ln-inverse*)
finally show *?thesis* .

qed

lemma *artanh-minus-real* [*simp*]:

assumes $\operatorname{abs} x < 1$
shows $\operatorname{artanh}(-x::\operatorname{real}) = -\operatorname{artanh} x$
using *assms* **by** (*simp add: artanh-def ln-div field-simps*)

lemma *sinh-less-cosh-real*: $\operatorname{sinh}(x :: \operatorname{real}) < \operatorname{cosh} x$

by (*simp add: sinh-def cosh-def*)

lemma *sinh-le-cosh-real*: $\operatorname{sinh}(x :: \operatorname{real}) \leq \operatorname{cosh} x$

by (*simp add: sinh-def cosh-def*)

lemma *tanh-real-lt-1*: $\operatorname{tanh}(x :: \operatorname{real}) < 1$

by (*simp add: tanh-def sinh-less-cosh-real*)

lemma *tanh-real-gt-neg1*: $\operatorname{tanh}(x :: \operatorname{real}) > -1$

proof –

have $-\operatorname{cosh} x < \operatorname{sinh} x$ **by** (*simp add: sinh-def cosh-def field-split-simps*)
thus *?thesis* **by** (*simp add: tanh-def field-simps*)

qed

lemma *tanh-real-bounds*: $\operatorname{tanh}(x :: \operatorname{real}) \in \{-1 < .. < 1\}$

using *tanh-real-lt-1 tanh-real-gt-neg1* **by** *simp*

context

fixes $x :: \operatorname{real}$

begin

lemma *arsinh-sinh-real*: $\operatorname{arsinh}(\operatorname{sinh} x) = x$

by (*simp add: arsinh-real-def powr-def sinh-square-eq sinh-plus-cosh*)

lemma *arcosh-cosh-real*: $x \geq 0 \implies \operatorname{arcosh}(\operatorname{cosh} x) = x$

by (*simp add: arcosh-real-def powr-def cosh-square-eq cosh-real-ge-1 cosh-plus-sinh*)

lemma *artanh-tanh-real*: $\operatorname{artanh}(\operatorname{tanh} x) = x$

proof –

have $\operatorname{artanh}(\operatorname{tanh} x) = \ln(\operatorname{cosh} x * (\operatorname{cosh} x + \operatorname{sinh} x) / (\operatorname{cosh} x * (\operatorname{cosh} x - \operatorname{sinh} x))) / 2$

by (*simp add: artanh-def tanh-def field-split-simps*)

also have $\operatorname{cosh} x * (\operatorname{cosh} x + \operatorname{sinh} x) / (\operatorname{cosh} x * (\operatorname{cosh} x - \operatorname{sinh} x)) =$

$(\cosh x + \sinh x) / (\cosh x - \sinh x)$ **by** *simp*
also have $\dots = (\exp x)^2$
by (*simp add: cosh-plus-sinh cosh-minus-sinh exp-minus field-simps power2-eq-square*)
also have $\ln ((\exp x)^2) / 2 = x$ **by** (*simp add: ln-realpow*)
finally show *?thesis* .
qed

lemma *sinh-real-zero-iff* [*simp*]: $\sinh x = 0 \longleftrightarrow x = 0$
by (*metis arsinh-0 arsinh-sinh-real sinh-0*)

lemma *cosh-real-one-iff* [*simp*]: $\cosh x = 1 \longleftrightarrow x = 0$
by (*smt (verit, best) Transcendental.arcosh-cosh-real cosh-0 cosh-minus*)

lemma *tanh-real-nonneg-iff* [*simp*]: $\tanh x \geq 0 \longleftrightarrow x \geq 0$
by (*simp add: tanh-def field-simps*)

lemma *tanh-real-pos-iff* [*simp*]: $\tanh x > 0 \longleftrightarrow x > 0$
by (*simp add: tanh-def field-simps*)

lemma *tanh-real-nonpos-iff* [*simp*]: $\tanh x \leq 0 \longleftrightarrow x \leq 0$
by (*simp add: tanh-def field-simps*)

lemma *tanh-real-neg-iff* [*simp*]: $\tanh x < 0 \longleftrightarrow x < 0$
by (*simp add: tanh-def field-simps*)

lemma *tanh-real-zero-iff* [*simp*]: $\tanh x = 0 \longleftrightarrow x = 0$
by (*simp add: tanh-def field-simps*)

end

lemma *sinh-real-strict-mono*: *strict-mono* (*sinh* :: *real* \Rightarrow *real*)
by (*rule pos-deriv-imp-strict-mono derivative-intros*) + *auto*

lemma *cosh-real-strict-mono*:
assumes $0 \leq x$ **and** $x < (y::\text{real})$
shows $\cosh x < \cosh y$

proof –

from *assms* **have** $\exists z > x. z < y \wedge \cosh y - \cosh x = (y - x) * \sinh z$

by (*intro MVT2*) (*auto dest: connectedD-interval intro!: derivative-eq-intros*)

then obtain z **where** $z: z > x \ z < y \ \cosh y - \cosh x = (y - x) * \sinh z$ **by**

blast

note $\langle \cosh y - \cosh x = (y - x) * \sinh z \rangle$

also from $\langle z > x \rangle$ **and** *assms* **have** $(y - x) * \sinh z > 0$ **by** (*intro mult-pos-pos*)

auto

finally show $\cosh x < \cosh y$ **by** *simp*

qed

lemma *tanh-real-strict-mono*: *strict-mono* (*tanh* :: *real* \Rightarrow *real*)

proof –

have *: $\tanh x^2 < 1$ **for** $x :: \text{real}$
using *tanh-real-bounds*[of x] **by** (*simp add: abs-square-less-1 abs-if*)
show ?thesis
by (*rule pos-deriv-imp-strict-mono*) (*insert *, auto intro!: derivative-intros*)
qed

lemma *sinh-real-abs* [*simp*]: $\sinh (\text{abs } x :: \text{real}) = \text{abs } (\sinh x)$
by (*simp add: abs-if*)

lemma *cosh-real-abs* [*simp*]: $\cosh (\text{abs } x :: \text{real}) = \cosh x$
by (*simp add: abs-if*)

lemma *tanh-real-abs* [*simp*]: $\tanh (\text{abs } x :: \text{real}) = \text{abs } (\tanh x)$
by (*auto simp: abs-if*)

lemma *sinh-real-eq-iff* [*simp*]: $\sinh x = \sinh y \longleftrightarrow x = (y :: \text{real})$
using *sinh-real-strict-mono* **by** (*simp add: strict-mono-eq*)

lemma *tanh-real-eq-iff* [*simp*]: $\tanh x = \tanh y \longleftrightarrow x = (y :: \text{real})$
using *tanh-real-strict-mono* **by** (*simp add: strict-mono-eq*)

lemma *cosh-real-eq-iff* [*simp*]: $\cosh x = \cosh y \longleftrightarrow \text{abs } x = \text{abs } (y :: \text{real})$

proof –

have $\cosh x = \cosh y \longleftrightarrow x = y$ **if** $x \geq 0 \ y \geq 0$ **for** $x \ y :: \text{real}$
using *cosh-real-strict-mono*[of $x \ y$] *cosh-real-strict-mono*[of $y \ x$] *that*
by (*cases x y rule: linorder-cases*) *auto*
from *this*[of $\text{abs } x \ \text{abs } y$] **show** ?thesis **by** *simp*

qed

lemma *sinh-real-le-iff* [*simp*]: $\sinh x \leq \sinh y \longleftrightarrow x \leq (y :: \text{real})$
using *sinh-real-strict-mono* **by** (*simp add: strict-mono-less-eq*)

lemma *cosh-real-nonneg-le-iff*: $x \geq 0 \implies y \geq 0 \implies \cosh x \leq \cosh y \longleftrightarrow x \leq (y :: \text{real})$
using *cosh-real-strict-mono*[of $x \ y$] *cosh-real-strict-mono*[of $y \ x$]
by (*cases x y rule: linorder-cases*) *auto*

lemma *cosh-real-nonpos-le-iff*: $x \leq 0 \implies y \leq 0 \implies \cosh x \leq \cosh y \longleftrightarrow x \geq (y :: \text{real})$
using *cosh-real-nonneg-le-iff*[of $-x \ -y$] **by** *simp*

lemma *tanh-real-le-iff* [*simp*]: $\tanh x \leq \tanh y \longleftrightarrow x \leq (y :: \text{real})$
using *tanh-real-strict-mono* **by** (*simp add: strict-mono-less-eq*)

lemma *sinh-real-less-iff* [*simp*]: $\sinh x < \sinh y \longleftrightarrow x < (y :: \text{real})$
using *sinh-real-strict-mono* **by** (*simp add: strict-mono-less*)

lemma *cosh-real-nonneg-less-iff*: $x \geq 0 \implies y \geq 0 \implies \cosh x < \cosh y \longleftrightarrow x < (y :: \text{real})$

(*y::real*)
using *cosh-real-strict-mono*[of *x y*] *cosh-real-strict-mono*[of *y x*]
by (*cases x y rule: linorder-cases*) *auto*

lemma *cosh-real-nonpos-less-iff*: $x \leq 0 \implies y \leq 0 \implies \cosh x < \cosh y \iff x >$
(*y::real*)
using *cosh-real-nonneg-less-iff*[of $-x -y$] **by** *simp*

lemma *tanh-real-less-iff* [*simp*]: $\tanh x < \tanh y \iff x <$ (*y::real*)
using *tanh-real-strict-mono* **by** (*simp add: strict-mono-less*)

113.23.2 Limits

lemma *sinh-real-at-top*: *filterlim* (*sinh* :: *real* \Rightarrow *real*) *at-top at-top*

proof –

have *: (($\lambda x. - \exp (-x)$) \longrightarrow ($-0::real$)) *at-top*
by (*intro tendsto-minus filterlim-compose*[*OF exp-at-bot*] *filterlim-uminus-at-bot-at-top*)
have *filterlim* ($\lambda x. (1/2) * (-\exp (-x) + \exp x) :: real$) *at-top at-top*
by (*rule filterlim-tendsto-pos-mult-at-top*[*OF - -*
filterlim-tendsto-add-at-top[*OF **]] *tendsto-const*) + (*auto simp:*
exp-at-top)

also have ($\lambda x. (1/2) * (-\exp (-x) + \exp x) :: real = \sinh$)
by (*simp add: fun-eq-iff sinh-def*)
finally show *?thesis* .

qed

lemma *sinh-real-at-bot*: *filterlim* (*sinh* :: *real* \Rightarrow *real*) *at-bot at-bot*

proof –

have *filterlim* ($\lambda x. -\sinh x :: real$) *at-bot at-top*
by (*simp add: filterlim-uminus-at-top* [*symmetric*] *sinh-real-at-top*)
also have ($\lambda x. -\sinh x :: real = (\lambda x. \sinh (-x))$) **by** *simp*
finally show *?thesis* **by** (*subst filterlim-at-bot-mirror*)

qed

lemma *cosh-real-at-top*: *filterlim* (*cosh* :: *real* \Rightarrow *real*) *at-top at-top*

proof –

have *: (($\lambda x. \exp (-x)$) \longrightarrow ($0::real$)) *at-top*
by (*intro filterlim-compose*[*OF exp-at-bot*] *filterlim-uminus-at-bot-at-top*)
have *filterlim* ($\lambda x. (1/2) * (\exp (-x) + \exp x) :: real$) *at-top at-top*
by (*rule filterlim-tendsto-pos-mult-at-top*[*OF - -*
filterlim-tendsto-add-at-top[*OF **]] *tendsto-const*) + (*auto simp:*
exp-at-top)

also have ($\lambda x. (1/2) * (\exp (-x) + \exp x) :: real = \cosh$)
by (*simp add: fun-eq-iff cosh-def*)
finally show *?thesis* .

qed

lemma *cosh-real-at-bot*: *filterlim* (*cosh* :: *real* \Rightarrow *real*) *at-top at-bot*

proof –

have *filterlim* ($\lambda x. \cosh (-x) :: \text{real}$) *at-top at-top*
by (*simp add: cosh-real-at-top*)
thus *?thesis* **by** (*subst filterlim-at-bot-mirror*)
qed

lemma *tanh-real-at-top*: ($\tanh \longrightarrow (1 :: \text{real})$) *at-top*
proof –
have ($(\lambda x :: \text{real}. (1 - \exp (- 2 * x)) / (1 + \exp (- 2 * x))) \longrightarrow (1 - 0) / (1 + 0)$) *at-top*
by (*intro tendsto-intros filterlim-compose*[*OF exp-at-bot*]
filterlim-tendsto-neg-mult-at-bot[*OF tendsto-const*] *filterlim-ident*) *auto*
also have ($\lambda x :: \text{real}. (1 - \exp (- 2 * x)) / (1 + \exp (- 2 * x)) = \tanh$)
by (*rule ext*) (*simp add: tanh-real-altdef*)
finally show *?thesis* **by** *simp*
qed

lemma *tanh-real-at-bot*: ($\tanh \longrightarrow (-1 :: \text{real})$) *at-bot*
proof –
have ($(\lambda x :: \text{real}. -\tanh x \longrightarrow -1)$) *at-top*
by (*intro tendsto-minus tanh-real-at-top*)
also have ($\lambda x. -\tanh x :: \text{real} = (\lambda x. \tanh (-x))$) **by** *simp*
finally show *?thesis* **by** (*subst filterlim-at-bot-mirror*)
qed

113.23.3 Properties of the inverse hyperbolic functions

lemma *isCont-sinh*: *isCont sinh* ($x :: 'a :: \{\text{real-normed-field, banach}\}$)
unfolding *sinh-def* [*abs-def*] **by** (*auto intro!: continuous-intros*)

lemma *isCont-cosh*: *isCont cosh* ($x :: 'a :: \{\text{real-normed-field, banach}\}$)
unfolding *cosh-def* [*abs-def*] **by** (*auto intro!: continuous-intros*)

lemma *isCont-tanh*: $\cosh x \neq 0 \implies \text{isCont tanh}$ ($x :: 'a :: \{\text{real-normed-field, banach}\}$)
unfolding *tanh-def* [*abs-def*]
by (*auto intro!: continuous-intros isCont-divide isCont-sinh isCont-cosh*)

lemma *continuous-on-sinh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a :: \{\text{real-normed-field, banach}\}$
assumes *continuous-on A f*
shows *continuous-on A* ($\lambda x. \sinh (f x)$)
unfolding *sinh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-on-cosh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a :: \{\text{real-normed-field, banach}\}$
assumes *continuous-on A f*
shows *continuous-on A* ($\lambda x. \cosh (f x)$)
unfolding *cosh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-sinh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* $F f$
shows *continuous* $F (\lambda x. \sinh (f x))$
unfolding *sinh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-cosh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* $F f$
shows *continuous* $F (\lambda x. \cosh (f x))$
unfolding *cosh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-on-tanh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous-on* $A f \wedge x. x \in A \implies \cosh (f x) \neq 0$
shows *continuous-on* $A (\lambda x. \tanh (f x))$
unfolding *tanh-def* **using** *assms* **by** (*intro continuous-intros*) *auto*

lemma *continuous-at-within-tanh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* (*at* x *within* A) $f \cosh (f x) \neq 0$
shows *continuous* (*at* x *within* A) $(\lambda x. \tanh (f x))$
unfolding *tanh-def* **using** *assms* **by** (*intro continuous-intros continuous-divide*)
auto

lemma *continuous-tanh* [*continuous-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
assumes *continuous* $F f \cosh (f (\text{Lim } F (\lambda x. x))) \neq 0$
shows *continuous* $F (\lambda x. \tanh (f x))$
unfolding *tanh-def* **using** *assms* **by** (*intro continuous-intros continuous-divide*)
auto

lemma *tendsto-sinh* [*tendsto-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $(f \longrightarrow a) F \implies ((\lambda x. \sinh (f x)) \longrightarrow \sinh a) F$
by (*rule isCont-tendsto-compose* [*OF isCont-sinh*])

lemma *tendsto-cosh* [*tendsto-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $(f \longrightarrow a) F \implies ((\lambda x. \cosh (f x)) \longrightarrow \cosh a) F$
by (*rule isCont-tendsto-compose* [*OF isCont-cosh*])

lemma *tendsto-tanh* [*tendsto-intros*]:
fixes $f :: - \Rightarrow 'a::\{\text{real-normed-field}, \text{banach}\}$
shows $(f \longrightarrow a) F \implies \cosh a \neq 0 \implies ((\lambda x. \tanh (f x)) \longrightarrow \tanh a) F$
by (*rule isCont-tendsto-compose* [*OF isCont-tanh*])

lemma *arsinh-real-has-field-derivative* [*derivative-intros*]:

fixes $x :: \text{real}$
shows $(\text{arsinh has-field-derivative } (1 / (\text{sqrt } (x^2 + 1))))$ (at x within A)
proof –
have $\text{pos}: 1 + x^2 > 0$ **by** $(\text{intro add-pos-nonneg})$ *auto*
from pos arsinh-real-aux [of x] **show** $?thesis$ **unfolding** arsinh-def [abs-def]
by $(\text{auto intro!}: \text{derivative-eq-intros simp: powr-minus powr-half-sqrt field-split-simps})$
qed

lemma $\text{arcosh-real-has-field-derivative}$ [derivative-intros]:
fixes $x :: \text{real}$
assumes $x > 1$
shows $(\text{arcosh has-field-derivative } (1 / (\text{sqrt } (x^2 - 1))))$ (at x within A)
proof –
from assms **have** $x + \text{sqrt } (x^2 - 1) > 0$ **by** $(\text{simp add: add-pos-pos})$
thus $?thesis$ **using** assms **unfolding** arcosh-def [abs-def]
by $(\text{auto intro!}: \text{derivative-eq-intros}$
 $\text{simp: powr-minus powr-half-sqrt field-split-simps power2-eq-1-iff})$
qed

lemma $\text{artanh-real-has-field-derivative}$ [derivative-intros]:
 $(\text{artanh has-field-derivative } (1 / (1 - x^2)))$ (at x within A) **if**
 $|x| < 1$ **for** $x :: \text{real}$
proof –
from that **have** $-1 < x < 1$ **by** linarith+
hence $(\text{artanh has-field-derivative } (4 - 4 * x) / ((1 + x) * (1 - x) * (1 - x) * 4))$
 $(\text{at } x \text{ within } A)$ **unfolding** artanh-def [abs-def]
by $(\text{auto intro!}: \text{derivative-eq-intros simp: powr-minus powr-half-sqrt})$
also have $(4 - 4 * x) / ((1 + x) * (1 - x) * (1 - x) * 4) = 1 / ((1 + x) * (1 - x))$
using $\langle -1 < x \rangle \langle x < 1 \rangle$ **by** $(\text{simp add: frac-eq-eq})$
also have $(1 + x) * (1 - x) = 1 - x^2$
by $(\text{simp add: algebra-simps power2-eq-square})$
finally show $?thesis$.
qed

lemma $\text{continuous-on-arsinh}$ [continuous-intros]: $\text{continuous-on } A$ ($\text{arsinh} :: \text{real} \Rightarrow \text{real}$)
by $(\text{rule DERIV-continuous-on derivative-intros})+$

lemma $\text{continuous-on-arcosh}$ [continuous-intros]:
assumes $A \subseteq \{1..\}$
shows $\text{continuous-on } A$ ($\text{arcosh} :: \text{real} \Rightarrow \text{real}$)
proof –
have $\text{pos}: x + \text{sqrt } (x^2 - 1) > 0$ **if** $x \geq 1$ **for** x
using that **by** $(\text{intro add-pos-nonneg})$ *auto*
show $?thesis$
unfolding arcosh-def [abs-def]
by $(\text{intro continuous-on-subset } [OF - \text{assms}] \text{continuous-on-ln continuous-on-add})$

continuous-on-id continuous-on-powr')
 (auto dest: pos simp: powr-half-sqrt intro!: continuous-intros)
 qed

lemma *continuous-on-artanh* [*continuous-intros*]:
 assumes $A \subseteq \{-1 < .. < 1\}$
 shows *continuous-on* A (*artanh* :: *real* \Rightarrow *real*)
 unfolding *artanh-def* [*abs-def*]
 by (*intro continuous-on-subset* [*OF - assms*]) (*auto intro!: continuous-intros*)

lemma *continuous-on-arsinh'* [*continuous-intros*]:
 fixes $f :: \text{real} \Rightarrow \text{real}$
 assumes *continuous-on* A f
 shows *continuous-on* A ($\lambda x. \text{arsinh } (f x)$)
 by (*rule continuous-on-compose2*[*OF continuous-on-arsinh assms*]) *auto*

lemma *continuous-on-arcosh'* [*continuous-intros*]:
 fixes $f :: \text{real} \Rightarrow \text{real}$
 assumes *continuous-on* A f $\wedge x. x \in A \Rightarrow f x \geq 1$
 shows *continuous-on* A ($\lambda x. \text{arcosh } (f x)$)
 by (*rule continuous-on-compose2*[*OF continuous-on-arcosh assms(1) order.refl*])
 (*use assms(2) in auto*)

lemma *continuous-on-artanh'* [*continuous-intros*]:
 fixes $f :: \text{real} \Rightarrow \text{real}$
 assumes *continuous-on* A f $\wedge x. x \in A \Rightarrow f x \in \{-1 < .. < 1\}$
 shows *continuous-on* A ($\lambda x. \text{artanh } (f x)$)
 by (*rule continuous-on-compose2*[*OF continuous-on-artanh assms(1) order.refl*])
 (*use assms(2) in auto*)

lemma *isCont-arsinh* [*continuous-intros*]: *isCont* *arsinh* ($x :: \text{real}$)
 using *continuous-on-arsinh[of UNIV]* by (*auto simp: continuous-on-eq-continuous-at*)

lemma *isCont-arcosh* [*continuous-intros*]:
 assumes $x > 1$
 shows *isCont* *arcosh* ($x :: \text{real}$)
proof –
 have *continuous-on* $\{1 :: \text{real} < ..\}$ *arcosh*
 by (*rule continuous-on-arcosh*) *auto*
 with *assms* **show** ?thesis by (*auto simp: continuous-on-eq-continuous-at*)
 qed

lemma *isCont-artanh* [*continuous-intros*]:
 assumes $x > -1$ $x < 1$
 shows *isCont* *artanh* ($x :: \text{real}$)
proof –
 have *continuous-on* $\{-1 < .. < (1 :: \text{real})\}$ *artanh*
 by (*rule continuous-on-artanh*) *auto*
 with *assms* **show** ?thesis by (*auto simp: continuous-on-eq-continuous-at*)

qed

lemma *tendsto-arsinh* [*tendsto-intros*]: $(f \longrightarrow a) F \implies ((\lambda x. \text{arsinh } (f x)) \longrightarrow \text{arsinh } a) F$
for $f :: - \Rightarrow \text{real}$
by (*rule isCont-tendsto-compose* [*OF isCont-arsinh*])

lemma *tendsto-arcosh-strong* [*tendsto-intros*]:
fixes $f :: - \Rightarrow \text{real}$
assumes $(f \longrightarrow a) F$ $a \geq 1$ *eventually* $(\lambda x. f x \geq 1) F$
shows $((\lambda x. \text{arcosh } (f x)) \longrightarrow \text{arcosh } a) F$
by (*rule continuous-on-tendsto-compose*[*OF continuous-on-arcosh*[*OF order.refl*]])
(use assms in auto)

lemma *tendsto-arcosh*:
fixes $f :: - \Rightarrow \text{real}$
assumes $(f \longrightarrow a) F$ $a > 1$
shows $((\lambda x. \text{arcosh } (f x)) \longrightarrow \text{arcosh } a) F$
by (*rule isCont-tendsto-compose* [*OF isCont-arcosh*]) *(use assms in auto)*

lemma *tendsto-arcosh-at-left-1*: $(\text{arcosh} \longrightarrow 0)$ *(at-right (1::real))*
proof –
have $(\text{arcosh} \longrightarrow \text{arcosh } 1)$ *(at-right (1::real))*
by (*rule tendsto-arcosh-strong*) *(auto simp: eventually-at intro!: exI[of - 1])*
thus *?thesis by simp*
 qed

lemma *tendsto-artanh* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow \text{real}$
assumes $(f \longrightarrow a) F$ $a > -1$ $a < 1$
shows $((\lambda x. \text{artanh } (f x)) \longrightarrow \text{artanh } a) F$
by (*rule isCont-tendsto-compose* [*OF isCont-artanh*]) *(use assms in auto)*

lemma *continuous-arsinh* [*continuous-intros*]:
 $\text{continuous } F f \implies \text{continuous } F (\lambda x. \text{arsinh } (f x :: \text{real}))$
unfolding *continuous-def* **by** (*rule tendsto-arsinh*)

lemma *continuous-arcosh-strong* [*continuous-intros*]:
assumes $\text{continuous } F f$ *eventually* $(\lambda x. f x \geq 1) F$
shows $\text{continuous } F (\lambda x. \text{arcosh } (f x :: \text{real}))$
proof (*cases F = bot*)
case *False*
show *?thesis*
unfolding *continuous-def*
proof (*intro tendsto-arcosh-strong*)
show $1 \leq f$ $(\text{Lim } F (\lambda x. x))$
using *assms False* **unfolding** *continuous-def* **by** (*rule tendsto-lowerbound*)
 qed *(insert assms, auto simp: continuous-def)*

qed *auto*

lemma *continuous-arcosh*:

continuous F f \implies f (Lim F ($\lambda x. x$)) > 1 \implies continuous F ($\lambda x. \text{arcosh } (f x :: \text{real})$)

unfolding *continuous-def* **by** (*rule tendsto-arcosh*) *auto*

lemma *continuous-artanh* [*continuous-intros*]:

continuous F f \implies f (Lim F ($\lambda x. x$)) $\in \{-1 < .. < 1\}$ \implies continuous F ($\lambda x. \text{artanh } (f x :: \text{real})$)

unfolding *continuous-def* **by** (*rule tendsto-artanh*) *auto*

lemma *arsinh-real-at-top*:

filterlim (arsinh :: real \Rightarrow real) at-top at-top

proof (*subst filterlim-cong*[*OF refl refl*])

show *filterlim ($\lambda x. \ln (x + \text{sqrt } (1 + x^2))$) at-top at-top*

by (*intro filterlim-compose*[*OF ln-at-top filterlim-at-top-add-at-top*] *filterlim-ident filterlim-compose*[*OF sqrt-at-top*] *filterlim-tendsto-add-at-top*[*OF tendsto-const*])

filterlim-pow-at-top) *auto*

qed (*auto intro!*: *eventually-mono*[*OF eventually-ge-at-top*[*of 1*]] *simp*: *arsinh-real-def add-ac*)

lemma *arsinh-real-at-bot*:

filterlim (arsinh :: real \Rightarrow real) at-bot at-bot

proof –

have *filterlim ($\lambda x::\text{real}. -\text{arsinh } x$) at-bot at-top*

by (*subst filterlim-uminus-at-top* [*symmetric*]) (*rule arsinh-real-at-top*)

also have ($\lambda x::\text{real}. -\text{arsinh } x = (\lambda x. \text{arsinh } (-x))$) **by** *simp*

finally show *?thesis*

by (*subst filterlim-at-bot-mirror*)

qed

lemma *arcosh-real-at-top*:

filterlim (arcosh :: real \Rightarrow real) at-top at-top

proof (*subst filterlim-cong*[*OF refl refl*])

show *filterlim ($\lambda x. \ln (x + \text{sqrt } (-1 + x^2))$) at-top at-top*

by (*intro filterlim-compose*[*OF ln-at-top filterlim-at-top-add-at-top*] *filterlim-ident filterlim-compose*[*OF sqrt-at-top*] *filterlim-tendsto-add-at-top*[*OF tendsto-const*])

filterlim-pow-at-top) *auto*

qed (*auto intro!*: *eventually-mono*[*OF eventually-ge-at-top*[*of 1*]] *simp*: *arcosh-real-def*)

lemma *artanh-real-at-left-1*:

filterlim (artanh :: real \Rightarrow real) at-top (at-left 1)

proof –

have **: filterlim ($\lambda x::\text{real}. (1 + x) / (1 - x)$) at-top (at-left 1)*

by (*rule LIM-at-top-divide*)

(*auto intro!*: *tendsto-eq-intros eventually-mono*[*OF eventually-at-left-real*[*of*

```

0]])
  have filterlim ( $\lambda x::real. (1/2) * \ln ((1 + x) / (1 - x))$ ) at-top (at-left 1)
    by (intro filterlim-tendsto-pos-mult-at-top[OF tendsto-const] *
        filterlim-compose[OF ln-at-top]) auto
  also have ( $\lambda x::real. (1/2) * \ln ((1 + x) / (1 - x))$ ) = artanh
    by (simp add: artanh-def [abs-def])
  finally show ?thesis .
qed

```

lemma *artanh-real-at-right-1*:

```

  filterlim (artanh :: real  $\Rightarrow$  real) at-bot (at-right (-1))
proof -
  have ?thesis  $\longleftrightarrow$  filterlim ( $\lambda x::real. -\text{artanh } x$ ) at-top (at-right (-1))
    by (simp add: filterlim-uminus-at-bot)
  also have ...  $\longleftrightarrow$  filterlim ( $\lambda x::real. \text{artanh } (-x)$ ) at-top (at-right (-1))
    by (intro filterlim-cong refl eventually-mono[OF eventually-at-right-real[of -1
1]]) auto
  also have ...  $\longleftrightarrow$  filterlim (artanh :: real  $\Rightarrow$  real) at-top (at-left 1)
    by (simp add: filterlim-at-left-to-right)
  also have ... by (rule artanh-real-at-left-1)
  finally show ?thesis .
qed

```

113.24 Simprocs for root and power literals

lemma *numeral-powr-numeral-real* [simp]:

```

  numeral m powr numeral n = (numeral m ^ numeral n :: real)
  by (simp add: powr-numeral)

```

context

begin

private lemma *sqrt-numeral-simproc-aux*:

```

  assumes  $m * m \equiv n$ 
  shows sqrt (numeral n :: real)  $\equiv$  numeral m

```

proof -

```

  have numeral n  $\equiv$  numeral m * (numeral m :: real) by (simp add: assms
[symmetric])

```

```

  moreover have sqrt ...  $\equiv$  numeral m by (subst real-sqrt-abs2) simp

```

```

  ultimately show sqrt (numeral n :: real)  $\equiv$  numeral m by simp

```

qed

private lemma *root-numeral-simproc-aux*:

```

  assumes Num.pow m n  $\equiv$  x
  shows root (numeral n) (numeral x :: real)  $\equiv$  numeral m
  by (subst assms [symmetric], subst numeral-pow, subst real-root-pos2) simp-all

```

private lemma *powr-numeral-simproc-aux*:

```

  assumes Num.pow y n = x

```

```

shows numeral x powr (m / numeral n :: real)  $\equiv$  numeral y powr m
by (subst assms [symmetric], subst numeral-pow, subst powr-numeral [symmetric])
    (simp, subst powr-powr, simp-all)

```

```

private lemma numeral-powr-inverse-eq:
  numeral x powr (inverse (numeral n)) = numeral x powr (1 / numeral n :: real)
by simp

```

ML \langle

```
signature ROOT-NUMERAL-SIMPROC = sig
```

```

val sqrt : int option -> int -> int option
val sqrt' : int option -> int -> int option
val nth-root : int option -> int -> int -> int option
val nth-root' : int option -> int -> int -> int option
val sqrt-simproc : Proof.context -> cterm -> thm option
val root-simproc : int * int -> Proof.context -> cterm -> thm option
val powr-simproc : int * int -> Proof.context -> cterm -> thm option

```

```
end
```

```
structure Root-Numeral-Simproc : ROOT-NUMERAL-SIMPROC = struct
```

```

fun iterate NONE p f x =
  let
    fun go x = if p x then x else go (f x)
  in
    SOME (go x)
  end
| iterate (SOME threshold) p f x =
  let
    fun go (threshold, x) =
      if p x then SOME x else if threshold = 0 then NONE else go (threshold -
1, f x)
  in
    go (threshold, x)
  end

```

```

fun nth-root - 1 x = SOME x
| nth-root - - 0 = SOME 0
| nth-root - - 1 = SOME 1
| nth-root threshold n x =
  let
    fun newton-step y = ((n - 1) * y + x div Integer.pow (n - 1) y) div n
    fun is-root y = Integer.pow n y <= x andalso x < Integer.pow n (y + 1)
  in

```

```

    if x < n then
      SOME 1
    else if x < Integer.pow n 2 then
      SOME 1
    else
      let
        val y = Real.floor (Math.pow (Real.fromInt x, Real.fromInt 1 / Real.fromInt
n))
      in
        if is-root y then
          SOME y
        else
          iterate threshold is-root newton-step ((x + n - 1) div n)
        end
      end
    end
  end

```

```

fun nth-root' - 1 x = SOME x
  | nth-root' - 0 = SOME 0
  | nth-root' - 1 = SOME 1
  | nth-root' threshold n x = if x < n then NONE else if x < Integer.pow n 2 then
NONE else
  case nth-root threshold n x of
    NONE => NONE
  | SOME y => if Integer.pow n y = x then SOME y else NONE

```

```

fun sqrt - 0 = SOME 0
  | sqrt - 1 = SOME 1
  | sqrt threshold n =
    let
      fun aux (a, b) = if n >= b * b then aux (b, b * b) else (a, b)
      val (lower-root, lower-n) = aux (1, 2)
      fun newton-step x = (x + n div x) div 2
      fun is-sqrt r = r*r <= n andalso n < (r+1)*(r+1)
      val y = Real.floor (Math.sqrt (Real.fromInt n))
    in
      if is-sqrt y then
        SOME y
      else
        Option.mapPartial (iterate threshold is-sqrt newton-step o (fn x => x *
lower-root))
          (sqrt threshold (n div lower-n))
    end
  end

```

```

fun sqrt' threshold x =
  case sqrt threshold x of
    NONE => NONE
  | SOME y => if y * y = x then SOME y else NONE

```

```

fun sqrt-simproc ctxt ct =

```

```

let
  val n = ct |> Thm.term-of |> dest-comb |> snd |> dest-comb |> snd |>
HOLogic.dest-numeral
in
  case sqrt' (SOME 10000) n of
    NONE => NONE
  | SOME m =>
      SOME (Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt o HO-
Logic.mk-numeral) [m, n])
        @{thm sqrt-numeral-simproc-aux})
end
handle TERM - => NONE

```

```

fun root-simproc (threshold1, threshold2) ctxt ct =
let
  val [n, x] =
    ct |> Thm.term-of |> strip-comb |> snd |> map (dest-comb #> snd #>
HOLogic.dest-numeral)
in
  if n > threshold1 orelse x > threshold2 then NONE else
    case nth-root' (SOME 100) n x of
      NONE => NONE
    | SOME m =>
        SOME (Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt o HO-
Logic.mk-numeral) [m, n, x])
          @{thm root-numeral-simproc-aux})
end
handle TERM - => NONE
  | Match => NONE

```

```

fun powr-simproc (threshold1, threshold2) ctxt ct =
let
  val eq-thm = Conv.try-conv (Conv.rewr-conv @{thm numeral-powr-inverse-eq})
ct
  val ct = Thm.dest-equals-rhs (Thm.cprop-of eq-thm)
  val (-, [x, t]) = strip-comb (Thm.term-of ct)
  val (-, [m, n]) = strip-comb t
  val [x, n] = map (dest-comb #> snd #> HOLogic.dest-numeral) [x, n]
in
  if n > threshold1 orelse x > threshold2 then NONE else
    case nth-root' (SOME 100) n x of
      NONE => NONE
    | SOME y =>
        let
          val [y, n, x] = map HOLogic.mk-numeral [y, n, x]
          val thm = Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt) [y, n,
x, m])
        in
          @{thm powr-numeral-simproc-aux}
        end
end

```

```

        SOME (@{thm transitive} OF [eq-thm, thm])
      end
    end
    handle TERM - => NONE
      | Match => NONE
    end
  ›

end

simproc-setup sqrt-numeral (sqrt (numeral n)) =
  ⟨K Root-Numeral-Simproc.sqrt-simproc⟩

simproc-setup root-numeral (root (numeral n) (numeral x)) =
  ⟨K (Root-Numeral-Simproc.root-simproc (200, Integer.pow 200 2))⟩

simproc-setup powr-divide-numeral
  (numeral x powr (m / numeral n :: real) | numeral x powr (inverse (numeral n)
  :: real)) =
  ⟨K (Root-Numeral-Simproc.powr-simproc (200, Integer.pow 200 2))⟩

lemma root 100 1267650600228229401496703205376 = 2
  by simp

lemma sqrt 196 = 14
  by simp

lemma 256 powr (7 / 4 :: real) = 16384
  by simp

lemma 27 powr (inverse 3) = (3::real)
  by simp

end

```

114 Complex Numbers: Rectangular and Polar Representations

```

theory Complex
imports Transcendental Real-Vector-Spaces
begin

```

We use the **codatatype** command to define the type of complex numbers. This allows us to use **primcorec** to define complex functions by defining their real and imaginary result separately.

```

codatatype complex = Complex (Re: real) (Im: real)

```

lemma *complex-surj*: $\text{Complex } (\text{Re } z) (\text{Im } z) = z$
by (*rule complex.collapse*)

lemma *complex-eqI* [*intro?*]: $\text{Re } x = \text{Re } y \implies \text{Im } x = \text{Im } y \implies x = y$
by (*rule complex.expand*) *simp*

lemma *complex-eq-iff*: $x = y \iff \text{Re } x = \text{Re } y \wedge \text{Im } x = \text{Im } y$
by (*auto intro: complex.expand*)

114.1 Addition and Subtraction

instantiation *complex* :: *ab-group-add*
begin

primcorec *zero-complex*
where
 $\text{Re } 0 = 0$
 $|\text{Im } 0 = 0$

primcorec *plus-complex*
where
 $\text{Re } (x + y) = \text{Re } x + \text{Re } y$
 $|\text{Im } (x + y) = \text{Im } x + \text{Im } y$

primcorec *uminus-complex*
where
 $\text{Re } (-x) = -\text{Re } x$
 $|\text{Im } (-x) = -\text{Im } x$

primcorec *minus-complex*
where
 $\text{Re } (x - y) = \text{Re } x - \text{Re } y$
 $|\text{Im } (x - y) = \text{Im } x - \text{Im } y$

instance
by *standard* (*simp-all add: complex-eq-iff*)

end

114.2 Multiplication and Division

instantiation *complex* :: *field*
begin

primcorec *one-complex*
where
 $\text{Re } 1 = 1$
 $|\text{Im } 1 = 0$

primcorec *times-complex*

where

$$\begin{aligned} \text{Re } (x * y) &= \text{Re } x * \text{Re } y - \text{Im } x * \text{Im } y \\ \text{Im } (x * y) &= \text{Re } x * \text{Im } y + \text{Im } x * \text{Re } y \end{aligned}$$

primcorec *inverse-complex*

where

$$\begin{aligned} \text{Re } (\text{inverse } x) &= \text{Re } x / ((\text{Re } x)^2 + (\text{Im } x)^2) \\ \text{Im } (\text{inverse } x) &= - \text{Im } x / ((\text{Re } x)^2 + (\text{Im } x)^2) \end{aligned}$$

definition $x \text{ div } y = x * \text{inverse } y$ for $x y :: \text{complex}$

instance

by *standard*

(*simp-all add: complex-eq-iff divide-complex-def*
distrib-left distrib-right right-diff-distrib left-diff-distrib
power2-eq-square add-divide-distrib [symmetric])

end

lemma *Re-divide*: $\text{Re } (x / y) = (\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2)$

by (*simp add: divide-complex-def add-divide-distrib*)

lemma *Im-divide*: $\text{Im } (x / y) = (\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2)$

by (*simp add: divide-complex-def diff-divide-distrib*)

lemma *Complex-divide*:

$$(x / y) = \text{Complex } ((\text{Re } x * \text{Re } y + \text{Im } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2)) \\ ((\text{Im } x * \text{Re } y - \text{Re } x * \text{Im } y) / ((\text{Re } y)^2 + (\text{Im } y)^2))$$

by (*metis Im-divide Re-divide complex-surj*)

lemma *Re-power2*: $\text{Re } (x \wedge 2) = (\text{Re } x)^2 - (\text{Im } x)^2$

by (*simp add: power2-eq-square*)

lemma *Im-power2*: $\text{Im } (x \wedge 2) = 2 * \text{Re } x * \text{Im } x$

by (*simp add: power2-eq-square*)

lemma *Re-power-real [simp]*: $\text{Im } x = 0 \implies \text{Re } (x \wedge n) = \text{Re } x \wedge n$

by (*induct n simp-all*)

lemma *Im-power-real [simp]*: $\text{Im } x = 0 \implies \text{Im } (x \wedge n) = 0$

by (*induct n simp-all*)

114.3 Scalar Multiplication

instantiation *complex :: real-field*

begin

```

primcorec scaleR-complex
  where
    Re (scaleR r x) = r * Re x
    | Im (scaleR r x) = r * Im x

instance
proof
  fix a b :: real and x y :: complex
  show scaleR a (x + y) = scaleR a x + scaleR a y
    by (simp add: complex-eq-iff distrib-left)
  show scaleR (a + b) x = scaleR a x + scaleR b x
    by (simp add: complex-eq-iff distrib-right)
  show scaleR a (scaleR b x) = scaleR (a * b) x
    by (simp add: complex-eq-iff mult.assoc)
  show scaleR 1 x = x
    by (simp add: complex-eq-iff)
  show scaleR a x * y = scaleR a (x * y)
    by (simp add: complex-eq-iff algebra-simps)
  show x * scaleR a y = scaleR a (x * y)
    by (simp add: complex-eq-iff algebra-simps)
qed

end

```

114.4 Numerals, Arithmetic, and Embedding from R

```

declare [[coercion of-real :: real ⇒ complex]]
declare [[coercion of-rat :: rat ⇒ complex]]
declare [[coercion of-int :: int ⇒ complex]]
declare [[coercion of-nat :: nat ⇒ complex]]

```

```

abbreviation complex-of-nat::nat ⇒ complex
  where complex-of-nat ≡ of-nat

```

```

abbreviation complex-of-int::int ⇒ complex
  where complex-of-int ≡ of-int

```

```

abbreviation complex-of-rat::rat ⇒ complex
  where complex-of-rat ≡ of-rat

```

```

abbreviation complex-of-real :: real ⇒ complex
  where complex-of-real ≡ of-real

```

```

lemma complex-Re-of-nat [simp]: Re (of-nat n) = of-nat n
  by (induct n) simp-all

```

```

lemma complex-Im-of-nat [simp]: Im (of-nat n) = 0
  by (induct n) simp-all

```

lemma *complex-Re-of-int* [*simp*]: $\text{Re} (\text{of-int } z) = \text{of-int } z$
by (*cases z rule: int-diff-cases*) *simp*

lemma *complex-Im-of-int* [*simp*]: $\text{Im} (\text{of-int } z) = 0$
by (*cases z rule: int-diff-cases*) *simp*

lemma *complex-Re-numeral* [*simp*]: $\text{Re} (\text{numeral } v) = \text{numeral } v$
using *complex-Re-of-int* [*of numeral v*] **by** *simp*

lemma *complex-Im-numeral* [*simp*]: $\text{Im} (\text{numeral } v) = 0$
using *complex-Im-of-int* [*of numeral v*] **by** *simp*

lemma *Re-complex-of-real* [*simp*]: $\text{Re} (\text{complex-of-real } z) = z$
by (*simp add: of-real-def*)

lemma *Im-complex-of-real* [*simp*]: $\text{Im} (\text{complex-of-real } z) = 0$
by (*simp add: of-real-def*)

lemma *Re-divide-numeral* [*simp*]: $\text{Re} (z / \text{numeral } w) = \text{Re } z / \text{numeral } w$
by (*simp add: Re-divide sqr-conv-mult*)

lemma *Im-divide-numeral* [*simp*]: $\text{Im} (z / \text{numeral } w) = \text{Im } z / \text{numeral } w$
by (*simp add: Im-divide sqr-conv-mult*)

lemma *Re-divide-of-nat* [*simp*]: $\text{Re} (z / \text{of-nat } n) = \text{Re } z / \text{of-nat } n$
by (*cases n*) (*simp-all add: Re-divide field-split-simps power2-eq-square del: of-nat-Suc*)

lemma *Im-divide-of-nat* [*simp*]: $\text{Im} (z / \text{of-nat } n) = \text{Im } z / \text{of-nat } n$
by (*cases n*) (*simp-all add: Im-divide field-split-simps power2-eq-square del: of-nat-Suc*)

lemma *Re-inverse* [*simp*]: $r \in \mathbb{R} \implies \text{Re} (\text{inverse } r) = \text{inverse} (\text{Re } r)$
by (*metis Re-complex-of-real Reals-cases of-real-inverse*)

lemma *Im-inverse* [*simp*]: $r \in \mathbb{R} \implies \text{Im} (\text{inverse } r) = 0$
by (*metis Im-complex-of-real Reals-cases of-real-inverse*)

lemma *of-real-Re* [*simp*]: $z \in \mathbb{R} \implies \text{of-real} (\text{Re } z) = z$
by (*auto simp: Reals-def*)

lemma *complex-Re-fact* [*simp*]: $\text{Re} (\text{fact } n) = \text{fact } n$
proof –
have (*fact n :: complex*) = *of-real* (*fact n*)
by *simp*
also have $\text{Re } \dots = \text{fact } n$
by (*subst Re-complex-of-real*) *simp-all*
finally show *?thesis* .
qed

lemma *surj-Re: surj Re*
by (*metis Re-complex-of-real surj-def*)

lemma *surj-Im: surj Im*
by (*metis complex.sel(2) surj-def*)

lemma *complex-Im-fact [simp]: Im (fact n) = 0*
by (*metis complex-Im-of-nat of-nat-fact*)

lemma *Re-prod-Reals: ($\bigwedge x. x \in A \implies f x \in \mathbb{R}$) \implies Re (prod f A) = prod ($\lambda x.$ Re (f x)) A*
proof (*induction A rule: infinite-finite-induct*)
case (*insert x A*)
hence *Re (prod f (insert x A)) = Re (f x) * Re (prod f A) - Im (f x) * Im (prod f A)*
by *simp*
also from *insert.prem*s **have** *f x \in \mathbb{R}* **by** *simp*
hence *Im (f x) = 0* **by** (*auto elim!: Reals-cases*)
also have *Re (prod f A) = ($\prod x \in A. Re (f x)$)*
by (*intro insert.IH insert.prem*s) *auto*
finally show *?case* **using** *insert.hyps* **by** *simp*
qed *auto*

114.5 The Complex Number i

primcorec *imaginary-unit :: complex* (*i*)
where
 $Re\ i = 0$
 $Im\ i = 1$

lemma *Complex-eq: Complex a b = a + i * b*
by (*simp add: complex-eq-iff*)

lemma *complex-eq: a = Re a + i * Im a*
by (*simp add: complex-eq-iff*)

lemma *fun-complex-eq: f = ($\lambda x. Re (f x) + i * Im (f x)$)*
by (*simp add: fun-eq-iff complex-eq*)

lemma *i-squared [simp]: i * i = -1*
by (*simp add: complex-eq-iff*)

lemma *power2-i [simp]: i² = -1*
by (*simp add: power2-eq-square*)

lemma *inverse-i [simp]: inverse i = - i*
by (*rule inverse-unique*) *simp*

lemma *divide-i [simp]: x / i = - i * x*

by (*simp add: divide-complex-def*)

lemma *complex-i-mult-minus* [*simp*]: $i * (i * x) = - x$
 by (*simp add: mult.assoc [symmetric]*)

lemma *complex-i-not-zero* [*simp*]: $i \neq 0$
 by (*simp add: complex-eq-iff*)

lemma *complex-i-not-one* [*simp*]: $i \neq 1$
 by (*simp add: complex-eq-iff*)

lemma *complex-i-not-numeral* [*simp*]: $i \neq \text{numeral } w$
 by (*simp add: complex-eq-iff*)

lemma *complex-i-not-neg-numeral* [*simp*]: $i \neq - \text{numeral } w$
 by (*simp add: complex-eq-iff*)

lemma *complex-split-polar*: $\exists r a. z = \text{complex-of-real } r * (\cos a + i * \sin a)$
 by (*simp add: complex-eq-iff polar-Ex*)

lemma *i-even-power* [*simp*]: $i ^ (n * 2) = (-1) ^ n$
 by (*metis mult.commute power2-i power-mult*)

lemma *i-even-power'* [*simp*]: $\text{even } n \implies i ^ n = (-1) ^ (n \text{ div } 2)$
 by (*metis dvd-mult-div-cancel power2-i power-mult*)

lemma *Re-i-times* [*simp*]: $\text{Re } (i * z) = - \text{Im } z$
 by *simp*

lemma *Im-i-times* [*simp*]: $\text{Im } (i * z) = \text{Re } z$
 by *simp*

lemma *i-times-eq-iff*: $i * w = z \longleftrightarrow w = - (i * z)$
 by *auto*

lemma *divide-numeral-i* [*simp*]: $z / (\text{numeral } n * i) = - (i * z) / \text{numeral } n$
 by (*metis divide-divide-eq-left divide-i mult commute mult-minus-right*)

lemma *imaginary-eq-real-iff* [*simp*]:
 assumes $y \in \text{Reals } x \in \text{Reals}$
 shows $i * y = x \longleftrightarrow x=0 \wedge y=0$
 by (*metis Im-complex-of-real Im-i-times assms mult-zero-right of-real-0 of-real-Re*)

lemma *real-eq-imaginary-iff* [*simp*]:
 assumes $y \in \text{Reals } x \in \text{Reals}$
 shows $x = i * y \longleftrightarrow x=0 \wedge y=0$
 using *assms imaginary-eq-real-iff* by *fastforce*

114.6 Vector Norm

instantiation *complex* :: *real-normed-field*
begin

definition *norm* $z = \text{sqrt } ((\text{Re } z)^2 + (\text{Im } z)^2)$

abbreviation *cmod* :: *complex* \Rightarrow *real*
where *cmod* \equiv *norm*

definition *complex-sgn-def*: *sgn* $x = x /_R \text{ cmod } x$

definition *dist-complex-def*: *dist* $x \ y = \text{cmod } (x - y)$

definition *uniformity-complex-def* [*code del*]:
(uniformity :: (*complex* \times *complex*) *filter*) = (*INF* $e \in \{0 <.. \}$. *principal* $\{(x, y). \text{ dist } x \ y < e\}$)

definition *open-complex-def* [*code del*]:
open ($U :: \text{complex set}$) $\longleftrightarrow (\forall x \in U. \text{ eventually } (\lambda(x', y). x' = x \longrightarrow y \in U) \text{ uniformity})$

instance

proof

fix $r :: \text{real}$ **and** $x \ y :: \text{complex}$ **and** $S :: \text{complex set}$

show $(\text{norm } x = 0) = (x = 0)$

by (*simp add: norm-complex-def complex-eq-iff*)

show $\text{norm } (x + y) \leq \text{norm } x + \text{norm } y$

by (*simp add: norm-complex-def complex-eq-iff real-sqrt-sum-squares-triangle-ineq*)

show $\text{norm } (\text{scaleR } r \ x) = |r| * \text{norm } x$

by (*simp add: norm-complex-def complex-eq-iff power-mult-distrib distrib-left [symmetric] real-sqrt-mult*)

show $\text{norm } (x * y) = \text{norm } x * \text{norm } y$

by (*simp add: norm-complex-def complex-eq-iff real-sqrt-mult [symmetric] power2-eq-square algebra-simps*)

qed (*rule complex-sgn-def dist-complex-def open-complex-def uniformity-complex-def*)+

end

declare *uniformity-Abort*[**where** $'a = \text{complex}$, *code*]

lemma *norm-ii* [*simp*]: $\text{norm } i = 1$

by (*simp add: norm-complex-def*)

lemma *cmod-unit-one*: $\text{cmod } (\cos a + i * \sin a) = 1$

by (*simp add: norm-complex-def*)

lemma *cmod-complex-polar*: $\text{cmod } (r * (\cos a + i * \sin a)) = |r|$

by (*simp add: norm-mult cmod-unit-one*)

lemma *complex-Re-le-cmod*: $Re\ x \leq cmod\ x$
unfolding *norm-complex-def* **by** (*rule real-sqrt-sum-squares-ge1*)

lemma *complex-mod-minus-le-complex-mod*: $- cmod\ x \leq cmod\ x$
by (*rule order-trans [OF - norm-ge-zero]*) *simp*

lemma *complex-mod-triangle-ineq2*: $cmod\ (b + a) - cmod\ b \leq cmod\ a$
by (*rule ord-le-eq-trans [OF norm-triangle-ineq2]*) *simp*

lemma *abs-Re-le-cmod*: $|Re\ x| \leq cmod\ x$
by (*simp add: norm-complex-def*)

lemma *abs-Im-le-cmod*: $|Im\ x| \leq cmod\ x$
by (*simp add: norm-complex-def*)

lemma *cmod-le*: $cmod\ z \leq |Re\ z| + |Im\ z|$
using *norm-complex-def sqrt-sum-squares-le-sum-abs* **by** *presburger*

lemma *cmod-eq-Re*: $Im\ z = 0 \implies cmod\ z = |Re\ z|$
by (*simp add: norm-complex-def*)

lemma *cmod-eq-Im*: $Re\ z = 0 \implies cmod\ z = |Im\ z|$
by (*simp add: norm-complex-def*)

lemma *cmod-power2*: $(cmod\ z)^2 = (Re\ z)^2 + (Im\ z)^2$
by (*simp add: norm-complex-def*)

lemma *cmod-plus-Re-le-0-iff*: $cmod\ z + Re\ z \leq 0 \iff Re\ z = - cmod\ z$
using *abs-Re-le-cmod[of z]* **by** *auto*

lemma *cmod-Re-le-iff*: $Im\ x = Im\ y \implies cmod\ x \leq cmod\ y \iff |Re\ x| \leq |Re\ y|$
by (*metis add.commute add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff*)

lemma *cmod-Im-le-iff*: $Re\ x = Re\ y \implies cmod\ x \leq cmod\ y \iff |Im\ x| \leq |Im\ y|$
by (*metis add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff*)

lemma *Im-eq-0*: $|Re\ z| = cmod\ z \implies Im\ z = 0$
by (*subst (asm) power-eq-iff-eq-base[symmetric, where n=2]*) (*auto simp add: norm-complex-def*)

lemma *abs-sqrt-wlog*: $(\bigwedge x. x \geq 0 \implies P\ x\ (x^2)) \implies P\ |x|\ (x^2)$
for $x::'a::linordered-idom$
by (*metis abs-ge-zero power2-abs*)

lemma *complex-abs-le-norm*: $|Re\ z| + |Im\ z| \leq \sqrt{2} * norm\ z$
unfolding *norm-complex-def*
apply (*rule abs-sqrt-wlog [where x=Re z]*)
apply (*rule abs-sqrt-wlog [where x=Im z]*)

```

apply (rule power2-le-imp-le)
apply (simp-all add: power2-sum add.commute sum-squares-bound real-sqrt-mult
[symmetric])
done

```

lemma *complex-unit-circle*: $z \neq 0 \implies (\operatorname{Re} z / \operatorname{cmod} z)^2 + (\operatorname{Im} z / \operatorname{cmod} z)^2 = 1$
by (simp add: norm-complex-def complex-eq-iff power2-eq-square add-divide-distrib [symmetric])

Properties of complex signum.

lemma *sgn-eq*: $\operatorname{sgn} z = z / \operatorname{complex-of-real} (\operatorname{cmod} z)$
by (simp add: sgn-div-norm divide-inverse scaleR-conv-of-real mult.commute)

lemma *Re-sgn* [simp]: $\operatorname{Re}(\operatorname{sgn} z) = \operatorname{Re}(z) / \operatorname{cmod} z$
by (simp add: complex-sgn-def divide-inverse)

lemma *Im-sgn* [simp]: $\operatorname{Im}(\operatorname{sgn} z) = \operatorname{Im}(z) / \operatorname{cmod} z$
by (simp add: complex-sgn-def divide-inverse)

114.7 Absolute value

instantiation *complex* :: field-abs-sgn
begin

definition *abs-complex* :: $\operatorname{complex} \Rightarrow \operatorname{complex}$
where *abs-complex* = *of-real* \circ *norm*

instance

proof **qed** (auto simp add: abs-complex-def complex-sgn-def norm-divide norm-mult scaleR-conv-of-real field-simps)
end

114.8 Completeness of the Complexes

lemma *bounded-linear-Re*: *bounded-linear Re*
by (rule bounded-linear-intro [where $K=1$]) (simp-all add: norm-complex-def)

lemma *bounded-linear-Im*: *bounded-linear Im*
by (rule bounded-linear-intro [where $K=1$]) (simp-all add: norm-complex-def)

lemmas *Cauchy-Re* = *bounded-linear.Cauchy* [OF *bounded-linear-Re*]

lemmas *Cauchy-Im* = *bounded-linear.Cauchy* [OF *bounded-linear-Im*]

lemmas *tendsto-Re* [tendsto-intros] = *bounded-linear.tendsto* [OF *bounded-linear-Re*]

lemmas *tendsto-Im* [tendsto-intros] = *bounded-linear.tendsto* [OF *bounded-linear-Im*]

lemmas *isCont-Re* [simp] = *bounded-linear.isCont* [OF *bounded-linear-Re*]

lemmas *isCont-Im* [simp] = *bounded-linear.isCont* [OF *bounded-linear-Im*]

lemmas *continuous-Re* [simp] = *bounded-linear.continuous* [OF *bounded-linear-Re*]

lemmas *continuous-Im* [simp] = *bounded-linear.continuous* [OF *bounded-linear-Im*]

lemmas *continuous-on-Re* [continuous-intros] = *bounded-linear.continuous-on*[OF *bounded-linear-Re*]

lemmas *continuous-on-Im* [*continuous-intros*] = *bounded-linear.continuous-on*[*OF bounded-linear-Im*]

lemmas *has-derivative-Re* [*derivative-intros*] = *bounded-linear.has-derivative*[*OF bounded-linear-Re*]

lemmas *has-derivative-Im* [*derivative-intros*] = *bounded-linear.has-derivative*[*OF bounded-linear-Im*]

lemmas *sums-Re* = *bounded-linear.sums* [*OF bounded-linear-Re*]

lemmas *sums-Im* = *bounded-linear.sums* [*OF bounded-linear-Im*]

lemmas *Re-suminf* = *bounded-linear.suminf*[*OF bounded-linear-Re*]

lemmas *Im-suminf* = *bounded-linear.suminf*[*OF bounded-linear-Im*]

lemma *tendsto-Complex* [*tendsto-intros*]:

$(f \longrightarrow a) F \implies (g \longrightarrow b) F \implies ((\lambda x. \text{Complex } (f x) (g x)) \longrightarrow \text{Complex } a b) F$

unfolding *Complex-eq* **by** (*auto intro!*: *tendsto-intros*)

lemma *tendsto-complex-iff*:

$(f \longrightarrow x) F \iff (((\lambda x. \text{Re } (f x)) \longrightarrow \text{Re } x) F \wedge ((\lambda x. \text{Im } (f x)) \longrightarrow \text{Im } x) F)$

proof *safe*

assume $((\lambda x. \text{Re } (f x)) \longrightarrow \text{Re } x) F ((\lambda x. \text{Im } (f x)) \longrightarrow \text{Im } x) F$

from *tendsto-Complex*[*OF this*] **show** $(f \longrightarrow x) F$

unfolding *complex.collapse* .

qed (*auto intro*: *tendsto-intros*)

lemma *continuous-complex-iff*:

$\text{continuous } F f \iff \text{continuous } F (\lambda x. \text{Re } (f x)) \wedge \text{continuous } F (\lambda x. \text{Im } (f x))$

by (*simp only*: *continuous-def tendsto-complex-iff*)

lemma *continuous-on-of-real-o-iff* [*simp*]:

$\text{continuous-on } S (\lambda x. \text{complex-of-real } (g x)) = \text{continuous-on } S g$

using *continuous-on-Re continuous-on-of-real* **by** *fastforce*

lemma *continuous-on-of-real-id* [*simp*]:

$\text{continuous-on } S (\text{of-real} :: \text{real} \Rightarrow 'a::\text{real-normed-algebra-1})$

by (*rule continuous-on-of-real* [*OF continuous-on-id*])

lemma *has-vector-derivative-complex-iff*: $(f \text{ has-vector-derivative } x) F \iff$

$((\lambda x. \text{Re } (f x)) \text{ has-field-derivative } (\text{Re } x)) F \wedge$

$((\lambda x. \text{Im } (f x)) \text{ has-field-derivative } (\text{Im } x)) F$

by (*simp add*: *has-vector-derivative-def has-field-derivative-def has-derivative-def tendsto-complex-iff algebra-simps bounded-linear-scaleR-left bounded-linear-mult-right*)

lemma *has-field-derivative-Re*[*derivative-intros*]:

$(f \text{ has-vector-derivative } D) F \implies ((\lambda x. \text{Re } (f x)) \text{ has-field-derivative } (\text{Re } D)) F$

unfolding *has-vector-derivative-complex-iff* **by** *safe*

lemma *has-field-derivative-Im*[*derivative-intros*]:

$(f \text{ has-vector-derivative } D) F \implies ((\lambda x. \text{Im } (f x)) \text{ has-field-derivative } (\text{Im } D)) F$

unfolding *has-vector-derivative-complex-iff* **by** *safe*

instance *complex* :: *banach*

proof

fix *X* :: *nat* \Rightarrow *complex*

assume *X*: *Cauchy X*

then have $(\lambda n. \text{Complex } (\text{Re } (X\ n)) (\text{Im } (X\ n))) \longrightarrow$

$\text{Complex } (\text{lim } (\lambda n. \text{Re } (X\ n))) (\text{lim } (\lambda n. \text{Im } (X\ n)))$

by (*intro tendsto-Complex convergent-LIMSEQ-iff* [THEN *iffD1*]

Cauchy-convergent-iff [THEN *iffD1*] *Cauchy-Re Cauchy-Im*)

then show *convergent X*

unfolding *complex.collapse* **by** (*rule convergentI*)

qed

declare *DERIV-power* [where '*a=complex, unfolded of-nat-def*[*symmetric*], *derivative-intros*]

114.9 Complex Conjugation

primcorec *cnj* :: *complex* \Rightarrow *complex*

where

$\text{Re } (\text{cnj } z) = \text{Re } z$

| $\text{Im } (\text{cnj } z) = - \text{Im } z$

lemma *complex-cnj-cancel-iff* [*simp*]: $\text{cnj } x = \text{cnj } y \longleftrightarrow x = y$

by (*simp add: complex-eq-iff*)

lemma *complex-cnj-cnj* [*simp*]: $\text{cnj } (\text{cnj } z) = z$

by (*simp add: complex-eq-iff*)

lemma *in-image-cnj-iff*: $z \in \text{cnj } ` A \longleftrightarrow \text{cnj } z \in A$

by (*metis complex-cnj-cnj image-iff*)

lemma *image-cnj-conv-vimage-cnj*: $\text{cnj } ` A = \text{cnj } - ` A$

using *in-image-cnj-iff* **by** *blast*

lemma *complex-cnj-zero* [*simp*]: $\text{cnj } 0 = 0$

by (*simp add: complex-eq-iff*)

lemma *complex-cnj-zero-iff* [*iff*]: $\text{cnj } z = 0 \longleftrightarrow z = 0$

by (*simp add: complex-eq-iff*)

lemma *complex-cnj-one-iff* [*simp*]: $\text{cnj } z = 1 \longleftrightarrow z = 1$

by (*simp add: complex-eq-iff*)

lemma *complex-cnj-add* [*simp*]: $\text{cnj } (x + y) = \text{cnj } x + \text{cnj } y$

by (*simp add: complex-eq-iff*)

lemma *cnj-sum* [*simp*]: $\text{cnj } (\text{sum } f\ s) = (\sum_{x \in s. \text{cnj } (f\ x)})$

by (induct s rule: infinite-finite-induct) auto

lemma *complex-cnj-diff* [simp]: $\text{cnj } (x - y) = \text{cnj } x - \text{cnj } y$
by (simp add: complex-eq-iff)

lemma *complex-cnj-minus* [simp]: $\text{cnj } (- x) = - \text{cnj } x$
by (simp add: complex-eq-iff)

lemma *complex-cnj-one* [simp]: $\text{cnj } 1 = 1$
by (simp add: complex-eq-iff)

lemma *complex-cnj-mult* [simp]: $\text{cnj } (x * y) = \text{cnj } x * \text{cnj } y$
by (simp add: complex-eq-iff)

lemma *cnj-prod* [simp]: $\text{cnj } (\text{prod } f s) = (\prod x \in s. \text{cnj } (f x))$
by (induct s rule: infinite-finite-induct) auto

lemma *complex-cnj-inverse* [simp]: $\text{cnj } (\text{inverse } x) = \text{inverse } (\text{cnj } x)$
by (simp add: complex-eq-iff)

lemma *complex-cnj-divide* [simp]: $\text{cnj } (x / y) = \text{cnj } x / \text{cnj } y$
by (simp add: divide-complex-def)

lemma *complex-cnj-power* [simp]: $\text{cnj } (x ^ n) = \text{cnj } x ^ n$
by (induct n) simp-all

lemma *complex-cnj-of-nat* [simp]: $\text{cnj } (\text{of-nat } n) = \text{of-nat } n$
by (simp add: complex-eq-iff)

lemma *complex-cnj-of-int* [simp]: $\text{cnj } (\text{of-int } z) = \text{of-int } z$
by (simp add: complex-eq-iff)

lemma *complex-cnj-numeral* [simp]: $\text{cnj } (\text{numeral } w) = \text{numeral } w$
by (simp add: complex-eq-iff)

lemma *complex-cnj-neg-numeral* [simp]: $\text{cnj } (- \text{numeral } w) = - \text{numeral } w$
by (simp add: complex-eq-iff)

lemma *complex-cnj-scaleR* [simp]: $\text{cnj } (\text{scaleR } r x) = \text{scaleR } r (\text{cnj } x)$
by (simp add: complex-eq-iff)

lemma *complex-mod-cnj* [simp]: $\text{cmod } (\text{cnj } z) = \text{cmod } z$
by (simp add: norm-complex-def)

lemma *complex-cnj-complex-of-real* [simp]: $\text{cnj } (\text{of-real } x) = \text{of-real } x$
by (simp add: complex-eq-iff)

lemma *complex-cnj-i* [simp]: $\text{cnj } i = - i$
by (simp add: complex-eq-iff)

lemma *complex-add-cnj*: $z + \text{cnj } z = \text{complex-of-real } (2 * \text{Re } z)$
by (*simp add: complex-eq-iff*)

lemma *complex-diff-cnj*: $z - \text{cnj } z = \text{complex-of-real } (2 * \text{Im } z) * i$
by (*simp add: complex-eq-iff*)

lemma *Ints-cnj* [*intro*]: $x \in \mathbf{Z} \implies \text{cnj } x \in \mathbf{Z}$
by (*auto elim!: Ints-cases*)

lemma *cnj-in-Ints-iff* [*simp*]: $\text{cnj } x \in \mathbf{Z} \longleftrightarrow x \in \mathbf{Z}$
using *Ints-cnj*[*of x*] *Ints-cnj*[*of cnj x*] **by** *auto*

lemma *complex-mult-cnj*: $z * \text{cnj } z = \text{complex-of-real } ((\text{Re } z)^2 + (\text{Im } z)^2)$
by (*simp add: complex-eq-iff power2-eq-square*)

lemma *cnj-add-mult-eq-Re*: $z * \text{cnj } w + \text{cnj } z * w = 2 * \text{Re } (z * \text{cnj } w)$
by (*rule complex-eqI*) *auto*

lemma *complex-mod-mult-cnj*: $\text{cmod } (z * \text{cnj } z) = (\text{cmod } z)^2$
by (*simp add: norm-mult power2-eq-square*)

lemma *complex-mod-sqrt-Re-mult-cnj*: $\text{cmod } z = \text{sqrt } (\text{Re } (z * \text{cnj } z))$
by (*simp add: norm-complex-def power2-eq-square*)

lemma *complex-In-mult-cnj-zero* [*simp*]: $\text{Im } (z * \text{cnj } z) = 0$
by *simp*

lemma *complex-cnj-fact* [*simp*]: $\text{cnj } (\text{fact } n) = \text{fact } n$
by (*subst of-nat-fact* [*symmetric*], *subst complex-cnj-of-nat*) *simp*

lemma *complex-cnj-pochhammer* [*simp*]: $\text{cnj } (\text{pochhammer } z \ n) = \text{pochhammer } (\text{cnj } z) \ n$
by (*induct n arbitrary: z*) (*simp-all add: pochhammer-rec*)

lemma *bounded-linear-cnj*: *bounded-linear cnj*
using *complex-cnj-add complex-cnj-scaleR* **by** (*rule bounded-linear-intro* [**where** $K=1$]) *simp*

lemma *linear-cnj*: *linear cnj*
using *bounded-linear.linear*[*OF bounded-linear-cnj*].

lemmas *tendsto-cnj* [*tendsto-intros*] = *bounded-linear.tendsto* [*OF bounded-linear-cnj*]
and *isCont-cnj* [*simp*] = *bounded-linear.isCont* [*OF bounded-linear-cnj*]
and *continuous-cnj* [*simp, continuous-intros*] = *bounded-linear.continuous* [*OF bounded-linear-cnj*]
and *continuous-on-cnj* [*simp, continuous-intros*] = *bounded-linear.continuous-on* [*OF bounded-linear-cnj*]
and *has-derivative-cnj* [*simp, derivative-intros*] = *bounded-linear.has-derivative*

[OF bounded-linear-cn]j

lemma *lim-cn]j*: $((\lambda x. \text{cnj}(f x)) \longrightarrow \text{cnj } l) F \longleftrightarrow (f \longrightarrow l) F$

by (*simp add: tendsto-iff dist-complex-def complex-cn]j-diff [symmetric] del: complex-cn]j-diff*)

lemma *sums-cn]j*: $((\lambda x. \text{cnj}(f x)) \text{ sums } \text{cnj } l) \longleftrightarrow (f \text{ sums } l)$

by (*simp add: sums-def lim-cn]j-cn]j-sum [symmetric] del: cn]j-sum*)

lemma *differentiable-cn]j-iff*:

$(\lambda z. \text{cnj } (f z))$ differentiable at x within $A \longleftrightarrow f$ differentiable at x within A

proof

assume $(\lambda z. \text{cnj } (f z))$ differentiable at x within A

then obtain D **where** $(\lambda z. \text{cnj } (f z))$ has-derivative D (at x within A)

by (*auto simp: differentiable-def*)

from has-derivative-cn]j[OF this] **show** f differentiable at x within A

by (*auto simp: differentiable-def*)

next

assume f differentiable at x within A

then obtain D **where** f has-derivative D (at x within A)

by (*auto simp: differentiable-def*)

from has-derivative-cn]j[OF this] **show** $(\lambda z. \text{cnj } (f z))$ differentiable at x within A

by (*auto simp: differentiable-def*)

qed

lemma *has-vector-derivative-cn]j* [derivative-intros]:

assumes f has-vector-derivative f' (at z within A)

shows $(\lambda z. \text{cnj } (f z))$ has-vector-derivative $\text{cnj } f'$ (at z within A)

using *assms* **by** (*auto simp: has-vector-derivative-complex-iff intro: derivative-intros*)

lemma *has-field-derivative-cn]j-cn]j*:

assumes f has-field-derivative F (at $(\text{cnj } z)$)

shows $(\text{cnj} \circ f \circ \text{cnj})$ has-field-derivative $\text{cnj } F$ (at z)

proof –

have $\text{cnj } -0 \rightarrow \text{cnj } 0$

by (*subst lim-cn]j auto*)

also have $\text{cnj } 0 = 0$

by *simp*

finally have $*$: *filterlim* cnj (at 0) (at 0)

by (*auto simp: filterlim-at eventually-at-filter*)

have $(\lambda h. (f (\text{cnj } z + \text{cnj } h) - f (\text{cnj } z)) / \text{cnj } h) -0 \rightarrow F$

by (*rule filterlim-compose[OF - *]*) (use *assms* **in** \langle *auto simp: DERIV-def* \rangle)

thus *?thesis*

by (*subst (asm) lim-cn]j [symmetric]*) (*simp add: DERIV-def*)

qed

114.10 Basic Lemmas

lemma *complex-of-real-code*[code-unfold]: *of-real* = (λx . *Complex* x 0)
by (*intro ext*, *auto simp: complex-eq-iff*)

lemma *complex-eq-0*: $z=0 \iff (Re\ z)^2 + (Im\ z)^2 = 0$
by (*metis zero-complex.sel complex-eqI sum-power2-eq-zero-iff*)

lemma *complex-neq-0*: $z\neq 0 \iff (Re\ z)^2 + (Im\ z)^2 > 0$
by (*metis complex-eq-0 less-numeral-extra(3) sum-power2-gt-zero-iff*)

lemma *complex-norm-square*: *of-real* ((*norm* z)²) = $z * cnj\ z$
by (*cases z*)
(*auto simp: complex-eq-iff norm-complex-def power2-eq-square[symmetric] of-real-power[symmetric]*
simp del: of-real-power)

lemma *complex-div-cnj*: $a / b = (a * cnj\ b) / (norm\ b)^2$
using *complex-norm-square* **by** *auto*

lemma *Re-complex-div-eq-0*: $Re\ (a / b) = 0 \iff Re\ (a * cnj\ b) = 0$
by (*auto simp add: Re-divide*)

lemma *Im-complex-div-eq-0*: $Im\ (a / b) = 0 \iff Im\ (a * cnj\ b) = 0$
by (*auto simp add: Im-divide*)

lemma *complex-div-gt-0*: $(Re\ (a / b) > 0 \iff Re\ (a * cnj\ b) > 0) \wedge (Im\ (a / b) > 0 \iff Im\ (a * cnj\ b) > 0)$

proof (*cases b = 0*)

case *True*

then show *?thesis* **by** *auto*

next

case *False*

then have $0 < (Re\ b)^2 + (Im\ b)^2$

by (*simp add: complex-eq-iff sum-power2-gt-zero-iff*)

then show *?thesis*

by (*simp add: Re-divide Im-divide zero-less-divide-iff*)

qed

lemma *Re-complex-div-gt-0*: $Re\ (a / b) > 0 \iff Re\ (a * cnj\ b) > 0$
and *Im-complex-div-gt-0*: $Im\ (a / b) > 0 \iff Im\ (a * cnj\ b) > 0$
using *complex-div-gt-0* **by** *auto*

lemma *Re-complex-div-ge-0*: $Re\ (a / b) \geq 0 \iff Re\ (a * cnj\ b) \geq 0$
by (*metis le-less Re-complex-div-eq-0 Re-complex-div-gt-0*)

lemma *Im-complex-div-ge-0*: $Im\ (a / b) \geq 0 \iff Im\ (a * cnj\ b) \geq 0$
by (*metis Im-complex-div-eq-0 Im-complex-div-gt-0 le-less*)

lemma *Re-complex-div-lt-0*: $Re\ (a / b) < 0 \iff Re\ (a * cnj\ b) < 0$
by (*metis less-asym neq-iff Re-complex-div-eq-0 Re-complex-div-gt-0*)

- lemma** *Im-complex-div-lt-0*: $Im (a / b) < 0 \longleftrightarrow Im (a * cnj b) < 0$
by (*metis Im-complex-div-eq-0 Im-complex-div-gt-0 less-asm neq-iff*)
- lemma** *Re-complex-div-le-0*: $Re (a / b) \leq 0 \longleftrightarrow Re (a * cnj b) \leq 0$
by (*metis not-le Re-complex-div-gt-0*)
- lemma** *Im-complex-div-le-0*: $Im (a / b) \leq 0 \longleftrightarrow Im (a * cnj b) \leq 0$
by (*metis Im-complex-div-gt-0 not-le*)
- lemma** *Re-divide-of-real [simp]*: $Re (z / of-real r) = Re z / r$
by (*simp add: Re-divide power2-eq-square*)
- lemma** *Im-divide-of-real [simp]*: $Im (z / of-real r) = Im z / r$
by (*simp add: Im-divide power2-eq-square*)
- lemma** *Re-divide-Reals [simp]*: $r \in \mathbb{R} \implies Re (z / r) = Re z / Re r$
by (*metis Re-divide-of-real of-real-Re*)
- lemma** *Im-divide-Reals [simp]*: $r \in \mathbb{R} \implies Im (z / r) = Im z / Re r$
by (*metis Im-divide-of-real of-real-Re*)
- lemma** *Re-sum[simp]*: $Re (sum f s) = (\sum x \in s. Re (f x))$
by (*induct s rule: infinite-finite-induct*) *auto*
- lemma** *Im-sum[simp]*: $Im (sum f s) = (\sum x \in s. Im (f x))$
by (*induct s rule: infinite-finite-induct*) *auto*
- lemma** *sum-Re-le-cmod*: $(\sum i \in I. Re (z i)) \leq cmod (\sum i \in I. z i)$
by (*metis Re-sum complex-Re-le-cmod*)
- lemma** *sum-Im-le-cmod*: $(\sum i \in I. Im (z i)) \leq cmod (\sum i \in I. z i)$
by (*smt (verit, best) Im-sum abs-Im-le-cmod sum.cong*)
- lemma** *sums-complex-iff*: $f \text{ sums } x \longleftrightarrow ((\lambda x. Re (f x)) \text{ sums } Re x) \wedge ((\lambda x. Im (f x)) \text{ sums } Im x)$
unfolding *sums-def tendsto-complex-iff Im-sum Re-sum ..*
- lemma** *summable-complex-iff*: $\text{summable } f \longleftrightarrow \text{summable } (\lambda x. Re (f x)) \wedge \text{summable } (\lambda x. Im (f x))$
unfolding *summable-def sums-complex-iff[abs-def]* **by** (*metis complex.sel*)
- lemma** *summable-complex-of-real [simp]*: $\text{summable } (\lambda n. \text{complex-of-real } (f n)) \longleftrightarrow \text{summable } f$
unfolding *summable-complex-iff* **by** *simp*
- lemma** *summable-Re*: $\text{summable } f \implies \text{summable } (\lambda x. Re (f x))$
unfolding *summable-complex-iff* **by** *blast*

lemma *summable-Im*: $\text{summable } f \implies \text{summable } (\lambda x. \text{Im } (f x))$
unfolding *summable-complex-iff* **by** *blast*

lemma *complex-is-Nat-iff*: $z \in \mathbf{N} \iff \text{Im } z = 0 \wedge (\exists i. \text{Re } z = \text{of-nat } i)$
by (*auto simp: Nats-def complex-eq-iff*)

lemma *complex-is-Int-iff*: $z \in \mathbf{Z} \iff \text{Im } z = 0 \wedge (\exists i. \text{Re } z = \text{of-int } i)$
by (*auto simp: Ints-def complex-eq-iff*)

lemma *complex-is-Real-iff*: $z \in \mathbf{R} \iff \text{Im } z = 0$
by (*auto simp: Reals-def complex-eq-iff*)

lemma *Reals-cnj-iff*: $z \in \mathbf{R} \iff \text{cnj } z = z$
by (*auto simp: complex-is-Real-iff complex-eq-iff*)

lemma *in-Reals-norm*: $z \in \mathbf{R} \implies \text{norm } z = |\text{Re } z|$
by (*simp add: complex-is-Real-iff norm-complex-def*)

lemma *Re-Reals-divide*: $r \in \mathbf{R} \implies \text{Re } (r / z) = \text{Re } r * \text{Re } z / (\text{norm } z)^2$
by (*simp add: Re-divide complex-is-Real-iff cmod-power2*)

lemma *Im-Reals-divide*: $r \in \mathbf{R} \implies \text{Im } (r / z) = -\text{Re } r * \text{Im } z / (\text{norm } z)^2$
by (*simp add: Im-divide complex-is-Real-iff cmod-power2*)

lemma *series-comparison-complex*:

fixes $f :: \text{nat} \Rightarrow 'a::\text{banach}$

assumes $sg: \text{summable } g$

and $\bigwedge n. g n \in \mathbf{R} \wedge \bigwedge n. \text{Re } (g n) \geq 0$

and $fg: \bigwedge n. n \geq N \implies \text{norm}(f n) \leq \text{norm}(g n)$

shows $\text{summable } f$

proof –

have $g: \bigwedge n. \text{cmod } (g n) = \text{Re } (g n)$

using *assms* **by** (*metis abs-of-nonneg in-Reals-norm*)

show *?thesis*

by (*metis fg g sg summable-comparison-test summable-complex-iff*)

qed

114.11 Polar Form for Complex Numbers

lemma *complex-unimodular-polar*:

assumes $\text{norm } z = 1$

obtains t **where** $0 \leq t < 2 * \text{pi}$ $z = \text{Complex } (\text{cos } t) (\text{sin } t)$

by (*metis cmod-power2 one-power2 complex-surj sincos-total-2pi [of Re z Im z] assms*)

114.11.1 $\cos \theta + i \sin \theta$

primcorec $\text{cis} :: \text{real} \Rightarrow \text{complex}$

where

$\text{Re } (\text{cis } a) = \text{cos } a$

| $Im (cis a) = sin a$

lemma *cis-zero* [*simp*]: $cis 0 = 1$
by (*simp add: complex-eq-iff*)

lemma *norm-cis* [*simp*]: $norm (cis a) = 1$
by (*simp add: norm-complex-def*)

lemma *sgn-cis* [*simp*]: $sgn (cis a) = cis a$
by (*simp add: sgn-div-norm*)

lemma *cis-2pi* [*simp*]: $cis (2 * pi) = 1$
by (*simp add: cis.ctr complex-eq-iff*)

lemma *cis-neq-zero* [*simp*]: $cis a \neq 0$
by (*metis norm-cis norm-zero zero-neq-one*)

lemma *cis-cnj*: $cnj (cis t) = cis (-t)$
by (*simp add: complex-eq-iff*)

lemma *cis-mult*: $cis a * cis b = cis (a + b)$
by (*simp add: complex-eq-iff cos-add sin-add*)

lemma *DeMoivre*: $(cis a) ^ n = cis (real n * a)$
by (*induct n*) (*simp-all add: algebra-simps cis-mult*)

lemma *cis-inverse* [*simp*]: $inverse (cis a) = cis (- a)$
by (*simp add: complex-eq-iff*)

lemma *cis-divide*: $cis a / cis b = cis (a - b)$
by (*simp add: divide-complex-def cis-mult*)

lemma *divide-conv-cnj*: $norm z = 1 \implies x / z = x * cnj z$
by (*metis complex-div-cnj div-by-1 mult-1 of-real-1 power2-eq-square*)

lemma *i-not-in-Reals* [*simp, intro*]: $i \notin \mathbb{R}$
by (*auto simp: complex-is-Real-iff*)

lemma *powr-power-complex*: $z \neq 0 \vee n \neq 0 \implies (z powr u :: complex) ^ n = z$
*powr (of-nat n * u)*
by (*induction n*) (*auto simp: algebra-simps powr-add*)

lemma *cos-n-Re-cis-pow-n*: $cos (real n * a) = Re (cis a ^ n)$
by (*auto simp add: DeMoivre*)

lemma *sin-n-Im-cis-pow-n*: $sin (real n * a) = Im (cis a ^ n)$
by (*auto simp add: DeMoivre*)

lemma *cis-pi* [*simp*]: $cis pi = -1$

by (*simp add: complex-eq-iff*)

lemma *cis-pi-half*[*simp*]: $\text{cis } (\pi / 2) = i$
 by (*simp add: cis.ctr complex-eq-iff*)

lemma *cis-minus-pi-half*[*simp*]: $\text{cis } (-(\pi / 2)) = -i$
 by (*simp add: cis.ctr complex-eq-iff*)

lemma *cis-multiple-2pi*[*simp*]: $n \in \mathbf{Z} \implies \text{cis } (2 * \pi * n) = 1$
 by (*auto elim!: Ints-cases simp: cis.ctr one-complex.ctr*)

lemma *minus-cis*: $-\text{cis } x = \text{cis } (x + \pi)$
 by (*simp flip: cis-mult*)

lemma *minus-cis'*: $-\text{cis } x = \text{cis } (x - \pi)$
 by (*simp flip: cis-divide*)

114.11.2 $r(\cos \theta + i \sin \theta)$

definition *rcis* :: $\text{real} \Rightarrow \text{real} \Rightarrow \text{complex}$
 where *rcis* r $a = \text{complex-of-real } r * \text{cis } a$

lemma *Re-rcis* [*simp*]: $\text{Re}(\text{rcis } r a) = r * \cos a$
 by (*simp add: rcis-def*)

lemma *Im-rcis* [*simp*]: $\text{Im}(\text{rcis } r a) = r * \sin a$
 by (*simp add: rcis-def*)

lemma *rcis-Ex*: $\exists r a. z = \text{rcis } r a$
 by (*simp add: complex-eq-iff polar-Ex*)

lemma *complex-mod-rcis* [*simp*]: $\text{cmod } (\text{rcis } r a) = |r|$
 by (*simp add: rcis-def norm-mult*)

lemma *cis-rcis-eq*: $\text{cis } a = \text{rcis } 1 a$
 by (*simp add: rcis-def*)

lemma *rcis-mult*: $\text{rcis } r1 a * \text{rcis } r2 b = \text{rcis } (r1 * r2) (a + b)$
 by (*simp add: rcis-def cis-mult*)

lemma *rcis-zero-mod* [*simp*]: $\text{rcis } 0 a = 0$
 by (*simp add: rcis-def*)

lemma *rcis-zero-arg* [*simp*]: $\text{rcis } r 0 = \text{complex-of-real } r$
 by (*simp add: rcis-def*)

lemma *rcis-eq-zero-iff* [*simp*]: $\text{rcis } r a = 0 \iff r = 0$
 by (*simp add: rcis-def*)

lemma *DeMoivre2*: $(rcis\ r\ a) \wedge n = rcis\ (r \wedge n)\ (real\ n * a)$
by (*simp add: rcis-def power-mult-distrib DeMoivre*)

lemma *rcis-inverse*: $inverse(rcis\ r\ a) = rcis\ (1 / r)\ (- a)$
by (*simp add: divide-inverse rcis-def*)

lemma *rcis-divide*: $rcis\ r1\ a / rcis\ r2\ b = rcis\ (r1 / r2)\ (a - b)$
by (*simp add: rcis-def cis-divide [symmetric]*)

114.11.3 Complex exponential

lemma *exp-Reals-eq*:
assumes $z \in \mathbb{R}$
shows $exp\ z = of-real\ (exp\ (Re\ z))$
using *assms* **by** (*auto elim!: Reals-cases simp: exp-of-real*)

lemma *cis-conv-exp*: $cis\ b = exp\ (i * b)$

proof –
have $(i * complex-of-real\ b) \wedge n /_R\ fact\ n =$
 $of-real\ (cos-coeff\ n * b \wedge n) + i * of-real\ (sin-coeff\ n * b \wedge n)$
for $n :: nat$
proof –
have $i \wedge n = fact\ n *_R\ (cos-coeff\ n + i * sin-coeff\ n)$
by (*induct n*)
(simp-all add: sin-coeff-Suc cos-coeff-Suc complex-eq-iff Re-divide Im-divide
field-simps
power2-eq-square add-nonneg-eq-0-iff)
then show *?thesis*
by (*simp add: field-simps*)
qed
then show *?thesis*
using *sin-converges [of b] cos-converges [of b]*
by (*auto simp add: Complex-eq cis.ctr exp-def simp del: of-real-mult*
intro!: sums-unique sums-add sums-mult sums-of-real)
qed

lemma *exp-eq-polar*: $exp\ z = exp\ (Re\ z) * cis\ (Im\ z)$
unfolding *cis-conv-exp exp-of-real [symmetric] mult-exp-exp*
by (*cases z*) (*simp add: Complex-eq*)

lemma *Re-exp*: $Re\ (exp\ z) = exp\ (Re\ z) * cos\ (Im\ z)$
unfolding *exp-eq-polar* **by** *simp*

lemma *Im-exp*: $Im\ (exp\ z) = exp\ (Re\ z) * sin\ (Im\ z)$
unfolding *exp-eq-polar* **by** *simp*

lemma *norm-cos-sin [simp]*: $norm\ (Complex\ (cos\ t)\ (sin\ t)) = 1$
by (*simp add: norm-complex-def*)

lemma *norm-exp-eq-Re* [simp]: $\text{norm } (\exp z) = \exp (\text{Re } z)$
 by (simp add: cis.code cmod-complex-polar exp-eq-polar Complex-eq)

lemma *complex-exp-exists*: $\exists a r. z = \text{complex-of-real } r * \exp a$
 using cis-conv-exp rcis-Ex rcis-def by force

lemma *exp-pi-i* [simp]: $\exp (\text{of-real } \pi * i) = -1$
 by (metis cis-conv-exp cis-pi mult.commute)

lemma *exp-pi-i'* [simp]: $\exp (i * \text{of-real } \pi) = -1$
 using cis-conv-exp cis-pi by auto

lemma *exp-two-pi-i* [simp]: $\exp (2 * \text{of-real } \pi * i) = 1$
 by (simp add: exp-eq-polar complex-eq-iff)

lemma *exp-two-pi-i'* [simp]: $\exp (i * (\text{of-real } \pi * 2)) = 1$
 by (metis exp-two-pi-i mult.commute)

lemma *continuous-on-cis* [continuous-intros]:
 continuous-on $A f \implies \text{continuous-on } A (\lambda x. \text{cis } (f x))$
 by (auto simp: cis-conv-exp intro!: continuous-intros)

lemma *tendsto-exp-0-Re-at-bot*: $(\exp \longrightarrow 0) (\text{filtercomap } \text{Re } \text{at-bot})$

proof –

have $(\lambda z. \text{cmod } (\exp z)) \longrightarrow 0) (\text{filtercomap } \text{Re } \text{at-bot})$
 by (auto intro!: filterlim-filtercomapI exp-at-bot)
 thus ?thesis
 using tendsto-norm-zero-iff by blast

qed

lemma *filterlim-exp-at-infinity-Re-at-top*: $\text{filterlim } \exp \text{ at-infinity } (\text{filtercomap } \text{Re } \text{at-top})$

proof –

have $\text{filterlim } (\lambda z. \text{norm } (\exp z)) \text{ at-top } (\text{filtercomap } \text{Re } \text{at-top})$
 by (auto intro!: filterlim-filtercomapI exp-at-top)
 thus ?thesis
 using filterlim-norm-at-top-imp-at-infinity by blast

qed

114.11.4 Complex argument

definition *Arg* :: $\text{complex} \Rightarrow \text{real}$

where $\text{Arg } z = (\text{if } z = 0 \text{ then } 0 \text{ else } (\text{SOME } a. \text{sgn } z = \text{cis } a \wedge -\pi < a \wedge a \leq \pi))$

lemma *Arg-zero*: $\text{Arg } 0 = 0$
 by (simp add: Arg-def)

lemma *cis-Arg-unique*:

assumes $\text{sgn } z = \text{cis } x$ **and** $-pi < x$ **and** $x \leq pi$
shows $\text{Arg } z = x$
proof –
from *assms* **have** $z \neq 0$ **by** *auto*
have $(\text{SOME } a. \text{sgn } z = \text{cis } a \wedge -pi < a \wedge a \leq pi) = x$
proof
fix a
define d **where** $d = a - x$
assume $a: \text{sgn } z = \text{cis } a \wedge -pi < a \wedge a \leq pi$
from a *assms* **have** $-(2*pi) < d \wedge d < 2*pi$
unfolding $d\text{-def}$ **by** *simp*
moreover
from a *assms* **have** $\cos a = \cos x$ **and** $\sin a = \sin x$
by $(\text{simp-all add: complex-eq-iff})$
then **have** $\cos d = 1$
by $(\text{simp add: d-def cos-diff})$
moreover **from** \cos **have** $\sin d = 0$
by $(\text{rule cos-one-sin-zero})$
ultimately **have** $d = 0$
by $(\text{auto simp: sin-zero-iff elim!: evenE dest!: less-2-cases})$
then **show** $a = x$
by (simp add: d-def)
qed $(\text{simp add: assms del: Re-sgn Im-sgn})$
with $\langle z \neq 0 \rangle$ **show** $\text{Arg } z = x$
by $(\text{simp add: Arg-def})$
qed

lemma *Arg-correct*:

assumes $z \neq 0$
shows $\text{sgn } z = \text{cis } (\text{Arg } z) \wedge -pi < \text{Arg } z \wedge \text{Arg } z \leq pi$
proof $(\text{simp add: Arg-def assms, rule someI-ex})$
obtain $r a$ **where** $z = \text{rcis } r a$
using *rcis-Ex* **by** *fast*
with *assms* **have** $r \neq 0$ **by** *auto*
define b **where** $b = (\text{if } 0 < r \text{ then } a \text{ else } a + pi)$
have $b: \text{sgn } z = \text{cis } b$
using $\langle r \neq 0 \rangle$ **by** $(\text{simp add: z b-def rcis-def of-real-def sgn-scaleR sgn-if complex-eq-iff})$
have *cis-2pi-nat*: $\text{cis } (2 * pi * \text{real-of-nat } n) = 1$ **for** n
by $(\text{induct } n) (\text{simp-all add: distrib-left cis-mult [symmetric] complex-eq-iff})$
have *cis-2pi-int*: $\text{cis } (2 * pi * \text{real-of-int } x) = 1$ **for** x
by $(\text{cases } x \text{ rule: int-diff-cases})$
 $(\text{simp add: right-diff-distrib cis-divide [symmetric] cis-2pi-nat})$
define c **where** $c = b - 2 * pi * \text{of-int } [(b - pi) / (2 * pi)]$
have $\text{sgn } z = \text{cis } c$
by $(\text{simp add: b c-def cis-divide [symmetric] cis-2pi-int})$
moreover **have** $-pi < c \wedge c \leq pi$
using *ceiling-correct* $[\text{of } (b - pi) / (2 * pi)]$
by $(\text{simp add: c-def less-divide-eq divide-le-eq algebra-simps del: le-of-int-ceiling})$

ultimately show $\exists a. \text{sgn } z = \text{cis } a \wedge -\pi < a \wedge a \leq \pi$
by *fast*
qed

lemma *Arg-bounded*: $-\pi < \text{Arg } z \wedge \text{Arg } z \leq \pi$
by (*cases* $z = 0$) (*simp-all add: Arg-zero Arg-correct*)

lemma *cis-Arg*: $z \neq 0 \implies \text{cis } (\text{Arg } z) = \text{sgn } z$
by (*simp add: Arg-correct*)

lemma *rcis-cmod-Arg*: $\text{rcis } (\text{cmod } z) (\text{Arg } z) = z$
by (*cases* $z = 0$) (*simp-all add: rcis-def cis-Arg sgn-div-norm of-real-def*)

lemma *rcis-cnj*:
shows $\text{cnj } a = \text{rcis } (\text{cmod } a) (-\text{Arg } a)$
by (*metis cis-cnj complex-cnj-complex-of-real complex-cnj-mult rcis-cmod-Arg rcis-def*)

lemma *cos-Arg-i-mult-zero* [*simp*]: $y \neq 0 \implies \text{Re } y = 0 \implies \cos (\text{Arg } y) = 0$
using *cis-Arg [of y]* **by** (*simp add: complex-eq-iff*)

lemma *Arg-ii* [*simp*]: $\text{Arg } i = \pi/2$
by (*rule cis-Arg-unique; simp add: sgn-eq*)

lemma *Arg-minus-ii* [*simp*]: $\text{Arg } (-i) = -\pi/2$
proof (*rule cis-Arg-unique*)
show $\text{sgn } (-i) = \text{cis } (-\pi/2)$
by (*simp add: sgn-eq*)
show $-\pi/2 \leq \pi$
using *pi-not-less-zero* **by** *linarith*
qed *auto*

lemma *cos-Arg*: $z \neq 0 \implies \cos (\text{Arg } z) = \text{Re } z / \text{norm } z$
by (*metis Re-sgn cis.sel(1) cis-Arg*)

lemma *sin-Arg*: $z \neq 0 \implies \sin (\text{Arg } z) = \text{Im } z / \text{norm } z$
by (*metis Im-sgn cis.sel(2) cis-Arg*)

114.12 Complex n-th roots

lemma *bij-betw-roots-unity*:
assumes $n > 0$
shows $\text{bij-betw } (\lambda k. \text{cis } (2 * \pi * \text{real } k / \text{real } n)) \{..<n\} \{z. z^n = 1\}$
(is bij-betw ?f -)
unfolding *bij-betw-def*
proof (*intro conjI*)
show *inj*: *inj-on ?f {..<n}* **unfolding** *inj-on-def*
proof (*safe, goal-cases*)
case ($1 k l$)

hence $kl: k < n \wedge l < n$ by *simp-all*
 from 1 have $1 = ?f k / ?f l$ by *simp*
 also have $\dots = \text{cis } (2 * \pi * (\text{real } k - \text{real } l) / n)$
 using *assms* by (*simp add: field-simps cis-divide*)
 finally have $\text{cos } (2 * \pi * (\text{real } k - \text{real } l) / n) = 1$
 by (*simp add: complex-eq-iff*)
 then obtain $m :: \text{int}$ where $2 * \pi * (\text{real } k - \text{real } l) / \text{real } n = \text{real-of-int } m$
 $* 2 * \pi$
 by (*subst (asm) cos-one-2pi-int*) *blast*
 hence $\text{real-of-int } (\text{int } k - \text{int } l) = \text{real-of-int } (m * \text{int } n)$
 unfolding of-int-diff of-int-mult using assms
 by (*simp add: nonzero-divide-eq-eq*)
 also note *of-int-eq-iff*
 finally have $*$: $\text{abs } m * n = \text{abs } (\text{int } k - \text{int } l)$ by (*simp add: abs-mult*)
 also have $\dots < \text{int } n$ using *kl* by *linarith*
 finally have $m = 0$ using *assms* by *simp*
 with $*$ show $k = l$ by *simp*
 qed

have *subset*: $?f \{ \dots < n \} \subseteq \{ z. z \wedge n = 1 \}$
 proof *safe*
 fix $k :: \text{nat}$
 have $\text{cis } (2 * \pi * \text{real } k / \text{real } n) \wedge n = \text{cis } (2 * \pi) \wedge k$
 using *assms* by (*simp add: DeMoivre mult-ac*)
 also have $\text{cis } (2 * \pi) = 1$ by (*simp add: complex-eq-iff*)
 finally show $?f k \wedge n = 1$ by *simp*
 qed

have $n = \text{card } \{ \dots < n \}$ by *simp*
 also from *assms* and *subset* have $\dots \leq \text{card } \{ z :: \text{complex}. z \wedge n = 1 \}$
 by (*intro card-inj-on-le [OF inj]*) (*auto simp: finite-roots-unity*)
 finally have *card*: $\text{card } \{ z :: \text{complex}. z \wedge n = 1 \} = n$
 using *assms* by (*intro antisym card-roots-unity*) *auto*

have *card* ($?f \{ \dots < n \}$) = *card* $\{ z :: \text{complex}. z \wedge n = 1 \}$
 using *card inj* by (*subst card-image*) *auto*
 with *subset* and *assms* show $?f \{ \dots < n \} = \{ z :: \text{complex}. z \wedge n = 1 \}$
 by (*intro card-subset-eq finite-roots-unity*) *auto*
 qed

lemma *card-roots-unity-eq*:
 assumes $n > 0$
 shows *card* $\{ z :: \text{complex}. z \wedge n = 1 \} = n$
 using *bij-betw-same-card [OF bij-betw-roots-unity [OF assms]]* by *simp*

lemma *bij-betw-nth-root-unity*:
 fixes $c :: \text{complex}$ and $n :: \text{nat}$
 assumes $c \neq 0$ and $n: n > 0$
 defines $c' \equiv \text{root } n (\text{norm } c) * \text{cis } (\text{Arg } c / n)$

shows *bij-betw* $(\lambda z. c' * z) \{z. z \wedge n = 1\} \{z. z \wedge n = c\}$
proof –
 have $c' \wedge n = \text{of-real } (\text{root } n \text{ (norm } c) \wedge n) * \text{cis } (\text{Arg } c)$
 unfolding *of-real-power* using n by (*simp add: c'-def power-mult-distrib DeMoivre*)
 also from n have $\text{root } n \text{ (norm } c) \wedge n = \text{norm } c$ by *simp*
 also from c have $\text{of-real } \dots * \text{cis } (\text{Arg } c) = c$ by (*simp add: cis-Arg Complex.sgn-eq*)
 finally have [*simp*]: $c' \wedge n = c$.

show *?thesis* unfolding *bij-betw-def inj-on-def*
proof *safe*
 fix $z :: \text{complex}$ assume $z \wedge n = 1$
 hence $(c' * z) \wedge n = c' \wedge n$ by (*simp add: power-mult-distrib*)
 also have $c' \wedge n = \text{of-real } (\text{root } n \text{ (norm } c) \wedge n) * \text{cis } (\text{Arg } c)$
 unfolding *of-real-power* using n by (*simp add: c'-def power-mult-distrib DeMoivre*)
 also from n have $\text{root } n \text{ (norm } c) \wedge n = \text{norm } c$ by *simp*
 also from c have $\dots * \text{cis } (\text{Arg } c) = c$ by (*simp add: cis-Arg Complex.sgn-eq*)
 finally show $(c' * z) \wedge n = c$.
next
 fix z assume $z: c = z \wedge n$
 define z' where $z' = z / c'$
 from c and n have $c' \neq 0$ by (*auto simp: c'-def*)
 with n c have $z = c' * z'$ and $z' \wedge n = 1$
 by (*auto simp: z'-def power-divide z*)
 thus $z \in (\lambda z. c' * z) \{z. z \wedge n = 1\}$ by *blast*
qed (*insert c n, auto simp: c'-def*)
qed

lemma *finite-nth-roots* [*intro*]:
 assumes $n > 0$
 shows *finite* $\{z :: \text{complex}. z \wedge n = c\}$
proof (*cases c = 0*)
 case *True*
 with *assms* have $\{z :: \text{complex}. z \wedge n = c\} = \{0\}$ by *auto*
 thus *?thesis* by *simp*
next
 case *False*
 from *assms* have *finite* $\{z :: \text{complex}. z \wedge n = 1\}$ by (*intro finite-roots-unity simp-all*)
 also have *?this* \longleftrightarrow *?thesis*
 by (*rule bij-betw-finite, rule bij-betw-nth-root-unity*) *fact+*
 finally show *?thesis* .
qed

lemma *card-nth-roots*:
 assumes $c \neq 0$ $n > 0$
 shows *card* $\{z :: \text{complex}. z \wedge n = c\} = n$

proof –

have $\text{card } \{z. z^n = c\} = \text{card } \{z::\text{complex}. z^n = 1\}$
by (*rule sym, rule bij-betw-same-card, rule bij-betw-nth-root-unity*) *fact+*
also have $\dots = n$ **by** (*rule card-roots-unity-eq*) *fact+*
finally show *?thesis* .

qed

lemma *sum-roots-unity*:

assumes $n > 1$

shows $\sum \{z::\text{complex}. z^n = 1\} = 0$

proof –

define ω **where** $\omega = \text{cis } (2 * \text{pi} / \text{real } n)$

have [*simp*]: $\omega \neq 1$

proof

assume $\omega = 1$

with *assms* **obtain** $k :: \text{int}$ **where** $2 * \text{pi} / \text{real } n = 2 * \text{pi} * \text{of-int } k$

by (*auto simp: ω -def complex-eq-iff cos-one-2pi-int*)

with *assms* **have** $\text{real } n * \text{of-int } k = 1$ **by** (*simp add: field-simps*)

also have $\text{real } n * \text{of-int } k = \text{of-int } (\text{int } n * k)$ **by** *simp*

also have $1 = (\text{of-int } 1 :: \text{real})$ **by** *simp*

also note *of-int-eq-iff*

finally show *False* **using** *assms* **by** (*auto simp: zmult-eq-1-iff*)

qed

have $(\sum z \mid z^n = 1. z :: \text{complex}) = (\sum k < n. \text{cis } (2 * \text{pi} * \text{real } k / \text{real } n))$
using *assms* **by** (*intro sum.reindex-bij-betw [symmetric] bij-betw-roots-unity*)

auto

also have $\dots = (\sum k < n. \omega^k)$

by (*intro sum.cong refl*) (*auto simp: ω -def DeMoivre mult-ac*)

also have $\dots = (\omega^n - 1) / (\omega - 1)$

by (*subst geometric-sum*) *auto*

also have $\omega^n - 1 = \text{cis } (2 * \text{pi}) - 1$ **using** *assms* **by** (*auto simp: ω -def DeMoivre*)

also have $\dots = 0$ **by** (*simp add: complex-eq-iff*)

finally show *?thesis* **by** *simp*

qed

lemma *sum-nth-roots*:

assumes $n > 1$

shows $\sum \{z::\text{complex}. z^n = c\} = 0$

proof (*cases c = 0*)

case *True*

with *assms* **have** $\{z::\text{complex}. z^n = c\} = \{0\}$ **by** *auto*

also have $\sum \dots = 0$ **by** *simp*

finally show *?thesis* .

next

case *False*

define c' **where** $c' = \text{root } n (\text{norm } c) * \text{cis } (\text{Arg } c / n)$

from *False* **and** *assms* **have** $(\sum \{z. z^n = c\}) = (\sum z \mid z^n = 1. c' * z)$

by (*subst sum.reindex-bij-betw* [*OF bij-betw-nth-root-unity, symmetric*])
 (*auto simp: sum-distrib-left finite-roots-unity c'-def*)
 also from *assms* have $\dots = 0$
 by (*simp add: sum-distrib-left* [*symmetric*] *sum-roots-unity*)
 finally show *?thesis* .
 qed

114.13 Square root of complex numbers

primcorec *csqrt* :: *complex* \Rightarrow *complex*

where

$Re\ (csqrt\ z) = sqrt\ ((cmod\ z + Re\ z) / 2)$
 $|Im\ (csqrt\ z) = (if\ Im\ z = 0\ then\ 1\ else\ sgn\ (Im\ z)) * sqrt\ ((cmod\ z - Re\ z) / 2)$

lemma *csqrt-of-real-nonneg* [*simp*]: $Im\ x = 0 \implies Re\ x \geq 0 \implies csqrt\ x = sqrt\ (Re\ x)$

by (*simp add: complex-eq-iff norm-complex-def*)

lemma *csqrt-of-real-nonpos* [*simp*]: $Im\ x = 0 \implies Re\ x \leq 0 \implies csqrt\ x = i * sqrt\ |Re\ x|$

by (*simp add: complex-eq-iff norm-complex-def*)

lemma *of-real-sqrt*: $x \geq 0 \implies of\ real\ (sqrt\ x) = csqrt\ (of\ real\ x)$

by (*simp add: complex-eq-iff norm-complex-def*)

lemma *csqrt-0* [*simp*]: $csqrt\ 0 = 0$

by *simp*

lemma *csqrt-1* [*simp*]: $csqrt\ 1 = 1$

by *simp*

lemma *csqrt-ii* [*simp*]: $csqrt\ i = (1 + i) / sqrt\ 2$

by (*simp add: complex-eq-iff Re-divide Im-divide real-sqrt-divide real-div-sqrt*)

lemma *power2-csqrt*[*simp, algebra*]: $(csqrt\ z)^2 = z$

proof (*cases Im z = 0*)

case *True*

then show *?thesis*

using *real-sqrt-pow2*[*of Re z*] *real-sqrt-pow2*[*of - Re z*]

by (*cases 0::real Re z rule: linorder-cases*)

(*simp-all add: complex-eq-iff Re-power2 Im-power2 power2-eq-square cmod-eq-Re*)

next

case *False*

moreover have $cmod\ z * cmod\ z - Re\ z * Re\ z = Im\ z * Im\ z$

by (*simp add: norm-complex-def power2-eq-square*)

moreover have $|Re\ z| \leq cmod\ z$

by (*simp add: norm-complex-def*)

ultimately show *?thesis*

by (*simp add: Re-power2 Im-power2 complex-eq-iff real-sgn-eq
field-simps real-sqrt-mult[symmetric] real-sqrt-divide*)
qed

lemma *csqrt-power-even*:

assumes *even n*

shows $\text{csqrt } z \wedge n = z \wedge (n \text{ div } 2)$

by (*metis assms dvd-mult-div-cancel power2-csqrt power-mult*)

lemma *norm-csqrt* [*simp*]: $\text{norm } (\text{csqrt } z) = \text{sqrt } (\text{norm } z)$

by (*metis abs-of-nonneg norm-ge-zero norm-mult power2-csqrt power2-eq-square
real-sqrt-abs*)

lemma *csqrt-eq-0* [*simp*]: $\text{csqrt } z = 0 \iff z = 0$

by *auto* (*metis power2-csqrt power-eq-0-iff*)

lemma *csqrt-eq-1* [*simp*]: $\text{csqrt } z = 1 \iff z = 1$

by *auto* (*metis power2-csqrt power2-eq-1-iff*)

lemma *csqrt-principal*: $0 < \text{Re } (\text{csqrt } z) \vee \text{Re } (\text{csqrt } z) = 0 \wedge 0 \leq \text{Im } (\text{csqrt } z)$

by (*auto simp add: not-less cmod-plus-Re-le-0-iff Im-eq-0*)

lemma *Re-csqrt*: $0 \leq \text{Re } (\text{csqrt } z)$

by (*metis csqrt-principal le-less*)

lemma *csqrt-square*:

assumes $0 < \text{Re } b \vee (\text{Re } b = 0 \wedge 0 \leq \text{Im } b)$

shows $\text{csqrt } (b \wedge 2) = b$

proof –

have $\text{csqrt } (b \wedge 2) = b \vee \text{csqrt } (b \wedge 2) = -b$

by (*simp add: power2-eq-iff[symmetric]*)

moreover have $\text{csqrt } (b \wedge 2) \neq -b \vee b = 0$

using *csqrt-principal*[of $b \wedge 2$] *assms*

by (*intro disjCI notI*) (*auto simp: complex-eq-iff*)

ultimately show *?thesis*

by *auto*

qed

lemma *csqrt-unique*: $w^2 = z \implies 0 < \text{Re } w \vee \text{Re } w = 0 \wedge 0 \leq \text{Im } w \implies \text{csqrt } z = w$

by (*auto simp: csqrt-square*)

lemma *csqrt-minus* [*simp*]:

assumes $\text{Im } x < 0 \vee (\text{Im } x = 0 \wedge 0 \leq \text{Re } x)$

shows $\text{csqrt } (-x) = i * \text{csqrt } x$

proof –

have $\text{csqrt } ((i * \text{csqrt } x) \wedge 2) = i * \text{csqrt } x$

proof (*rule csqrt-square*)

have $\text{Im } (\text{csqrt } x) \leq 0$

using *assms* **by** (*auto simp add: cmod-eq-Re mult-le-0-iff field-simps complex-Re-le-cmod*)
then show $0 < \text{Re } (i * \text{csqrt } x) \vee \text{Re } (i * \text{csqrt } x) = 0 \wedge 0 \leq \text{Im } (i * \text{csqrt } x)$
by (*auto simp add: Re-csqrt simp del: csqrt.simps*)
qed
also have $(i * \text{csqrt } x)^2 = -x$
by (*simp add: power-mult-distrib*)
finally show *?thesis* .
qed

Legacy theorem names

lemmas *cmod-def = norm-complex-def*

lemma *legacy-Complex-simps:*

shows *Complex-eq-0: Complex a b = 0 \longleftrightarrow a = 0 \wedge b = 0*
and *complex-add: Complex a b + Complex c d = Complex (a + c) (b + d)*
and *complex-minus: $-$ (Complex a b) = Complex $-$ a $-$ b*
and *complex-diff: Complex a b $-$ Complex c d = Complex (a - c) (b - d)*
and *Complex-eq-1: Complex a b = 1 \longleftrightarrow a = 1 \wedge b = 0*
and *Complex-eq-neg-1: Complex a b = - 1 \longleftrightarrow a = - 1 \wedge b = 0*
and *complex-mult: Complex a b * Complex c d = Complex (a * c - b * d) (a * d + b * c)*
and *complex-inverse: inverse (Complex a b) = Complex (a / (a² + b²)) $-$ b / (a² + b²)*
and *Complex-eq-numeral: Complex a b = numeral w \longleftrightarrow a = numeral w \wedge b = 0*
and *Complex-eq-neg-numeral: Complex a b = - numeral w \longleftrightarrow a = - numeral w \wedge b = 0*
and *complex-scaleR: scaleR r (Complex a b) = Complex (r * a) (r * b)*
and *Complex-eq-i: Complex x y = i \longleftrightarrow x = 0 \wedge y = 1*
and *i-mult-Complex: i * Complex a b = Complex $-$ b a*
and *Complex-mult-i: Complex a b * i = Complex $-$ b a*
and *i-complex-of-real: i * complex-of-real r = Complex 0 r*
and *complex-of-real-i: complex-of-real r * i = Complex 0 r*
and *Complex-add-complex-of-real: Complex x y + complex-of-real r = Complex (x+r) y*
and *complex-of-real-add-Complex: complex-of-real r + Complex x y = Complex (r+x) y*
and *Complex-mult-complex-of-real: Complex x y * complex-of-real r = Complex (x*r) (y*r)*
and *complex-of-real-mult-Complex: complex-of-real r * Complex x y = Complex (r*x) (r*y)*
and *complex-eq-cancel-iff2: (Complex x y = complex-of-real xa) = (x = xa \wedge y = 0)*
and *complex-cnj: cnj (Complex a b) = Complex a $-$ b*
and *Complex-sum': sum $(\lambda x. \text{Complex } (f x) 0)$ s = Complex (sum f s) 0*
and *Complex-sum: Complex (sum f s) 0 = sum $(\lambda x. \text{Complex } (f x) 0)$ s*
and *complex-of-real-def: complex-of-real r = Complex r 0*
and *complex-norm: cmod (Complex x y) = sqrt (x² + y²)*

by (*simp-all add: norm-complex-def field-simps complex-eq-iff Re-divide Im-divide*)

lemma *Complex-in-Reals: Complex $x \neq 0 \in \mathbb{R}$*
 by (*metis Reals-of-real complex-of-real-def*)

Express a complex number as a linear combination of two others, not collinear with the origin

lemma *complex-axes:*

assumes $\text{Im } (y/x) \neq 0$

obtains $a \ b$ **where** $z = \text{of-real } a * x + \text{of-real } b * y$

proof –

define dd **where** $dd \equiv \text{Re } y * \text{Im } x - \text{Im } y * \text{Re } x$

define a **where** $a = (\text{Im } z * \text{Re } y - \text{Re } z * \text{Im } y) / dd$

define b **where** $b = (\text{Re } z * \text{Im } x - \text{Im } z * \text{Re } x) / dd$

have $dd \neq 0$

using *assms* **by** (*auto simp: dd-def Im-complex-div-eq-0*)

have $a * \text{Re } x + b * \text{Re } y = \text{Re } z$

using $\langle dd \neq 0 \rangle$

apply (*simp add: a-def b-def field-simps*)

by (*metis dd-def diff-add-cancel distrib-right mult.assoc mult.commute*)

moreover **have** $a * \text{Im } x + b * \text{Im } y = \text{Im } z$

using $\langle dd \neq 0 \rangle$

apply (*simp add: a-def b-def field-simps*)

by (*metis (no-types) dd-def diff-add-cancel distrib-right mult.assoc mult.commute*)

ultimately **have** $z = \text{of-real } a * x + \text{of-real } b * y$

by (*simp add: complex-eqI*)

then show *?thesis* **using** *that* **by** *simp*

qed

end

115 MacLaurin and Taylor Series

theory *MacLaurin*

imports *Transcendental*

begin

115.1 Maclaurin’s Theorem with Lagrange Form of Remainder

This is a very long, messy proof even now that it’s been broken down into lemmas.

lemma *Maclaurin-lemma:*

$0 < h \implies$

$\exists B::\text{real. } f h = (\sum_{m < n. (j \ m / (\text{fact } m)) * (h \hat{=} m)) + (B * ((h \hat{=} n) / (\text{fact } n)))$

by (*rule exI[where $x = (f h - (\sum_{m < n. (j \ m / (\text{fact } m)) * h \hat{=} m)) * (\text{fact } n) / (h \hat{=} n)$]*) *simp*)

lemma *eq-diff-eq'*: $x = y - z \longleftrightarrow y = x + z$
for $x y z :: \text{real}$
by *arith*

lemma *fact-diff-Suc*: $n < \text{Suc } m \implies \text{fact } (\text{Suc } m - n) = (\text{Suc } m - n) * \text{fact } (m - n)$
by (*subst fact-reduce*) *auto*

lemma *Maclaurin-lemma2*:
fixes B
assumes *DERIV*: $\forall m t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{diff } m) t :> \text{diff } (\text{Suc } m) t$
and *INIT*: $n = \text{Suc } k$
defines *difg* \equiv
 $(\lambda m t :: \text{real}. \text{diff } m t -$
 $((\sum p < n - m. \text{diff } (m + p) 0 / \text{fact } p * t ^ p) + B * (t ^ (n - m) / \text{fact } (n - m))))$
(is *difg* $\equiv (\lambda m t. \text{diff } m t - ?\text{difg } m t)$
shows $\forall m t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{difg } m) t :> \text{difg } (\text{Suc } m) t$
proof (*rule allI impI*)
fix $m t$
assume *INIT2*: $m < n \wedge 0 \leq t \wedge t \leq h$
have *DERIV* (*difg* m) $t :> \text{diff } (\text{Suc } m) t -$
 $((\sum x < n - m. \text{real } x * t ^ (x - \text{Suc } 0) * \text{diff } (m + x) 0 / \text{fact } x) +$
 $\text{real } (n - m) * t ^ (n - \text{Suc } m) * B / \text{fact } (n - m))$
by (*auto simp: difg-def intro!: derivative-eq-intros DERIV[rule-format, OF INIT2]*)
moreover
from *INIT2* **have** *intvl*: $\{.. < n - m\} = \text{insert } 0 (\text{Suc } \{.. < n - \text{Suc } m\})$ **and** $0 < n - m$
unfolding *atLeast0LessThan[symmetric]* **by** *auto*
have $(\sum x < n - m. \text{real } x * t ^ (x - \text{Suc } 0) * \text{diff } (m + x) 0 / \text{fact } x) =$
 $(\sum x < n - \text{Suc } m. \text{real } (\text{Suc } x) * t ^ x * \text{diff } (\text{Suc } m + x) 0 / \text{fact } (\text{Suc } x))$
unfolding *intvl* **by** (*subst sum.insert*) (*auto simp: sum.reindex*)
moreover
have *fact-neq-0*: $\bigwedge x. (\text{fact } x) + \text{real } x * (\text{fact } x) \neq 0$
by (*metis add-pos-pos fact-gt-zero less-add-same-cancel1 less-add-same-cancel2 less-numeral-extra(3) mult-less-0-iff of-nat-less-0-iff*)
have $\bigwedge x. (\text{Suc } x) * t ^ x * \text{diff } (\text{Suc } m + x) 0 / \text{fact } (\text{Suc } x) = \text{diff } (\text{Suc } m + x) 0 * t ^ x / \text{fact } x$
by (*rule nonzero-divide-eq-eq[THEN iffD2]*) *auto*
moreover
have $(n - m) * t ^ (n - \text{Suc } m) * B / \text{fact } (n - m) = B * (t ^ (n - \text{Suc } m) / \text{fact } (n - \text{Suc } m))$
using $\langle 0 < n - m \rangle$ **by** (*simp add: field-split-simps fact-reduce*)
ultimately show *DERIV* (*difg* m) $t :> \text{difg } (\text{Suc } m) t$
unfolding *difg-def* **by** (*simp add: mult commute*)
qed

lemma *Maclaurin*:
assumes $h: 0 < h$
and $n: 0 < n$
and *diff-0*: $\text{diff } 0 = f$
and *diff-Suc*: $\forall m t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{diff } m) t \text{ :> diff}$
 $(\text{Suc } m) t$
shows
 $\exists t::\text{real}. 0 < t \wedge t < h \wedge$
 $f h = \text{sum } (\lambda m. (\text{diff } m \ 0 / \text{fact } m) * h \wedge^m) \{..<n\} + (\text{diff } n \ t / \text{fact } n) * h$
 \wedge^n
proof –
from n **obtain** m **where** $m: n = \text{Suc } m$
by $(\text{cases } n)$ $(\text{simp add: } n)$
from m **have** $m < n$ **by** *simp*

obtain B **where** $f\text{-}h$: $f h = (\sum m < n. \text{diff } m \ 0 / \text{fact } m * h \wedge^m) + B * (h \wedge^n / \text{fact } n)$
using *Maclaurin-lemma* $[OF \ h] ..$

define g **where** $[abs\text{-}def]$: $g \ t =$
 $f \ t - (\text{sum } (\lambda m. (\text{diff } m \ 0 / \text{fact } m) * t \wedge^m) \{..<n\} + B * (t \wedge^n / \text{fact } n))$ **for** t
have $g2$: $g \ 0 = 0 \ g \ h = 0$
by $(\text{simp-all add: } m \ f\text{-}h \ g\text{-}def \ \text{lessThan-Suc-eq-insert-0 image-iff diff-0 sum.reindex})$

define difg **where** $[abs\text{-}def]$: $\text{difg } m \ t =$
 $\text{diff } m \ t - (\text{sum } (\lambda p. (\text{diff } (m + p) \ 0 / \text{fact } p) * (t \wedge^p)) \{..<n-m\} +$
 $B * ((t \wedge^{(n - m)}) / \text{fact } (n - m)))$ **for** $m \ t$
have *diffg-0*: $\text{difg } 0 = g$
by $(\text{simp add: difg-def } g\text{-}def \ \text{diff-0})$
have *diffg-Suc*: $\forall m t. m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{difg } m) t \text{ :> difg}$
 $(\text{Suc } m) t$
using *diff-Suc m unfolding difg-def [abs-def]* **by** $(\text{rule } \text{Maclaurin-lemma2})$
have *diffg-eq-0*: $\forall m < n. \text{difg } m \ 0 = 0$
by $(\text{auto simp: difg-def } m \ \text{Suc-diff-le lessThan-Suc-eq-insert-0 image-iff sum.reindex})$
have *isCont-difg*: $\bigwedge m x. m < n \implies 0 \leq x \implies x \leq h \implies \text{isCont } (\text{difg } m) x$
by $(\text{rule } \text{DERIV-isCont } [OF \ \text{difg-Suc } [\text{rule-format}]]) \text{ simp}$
have *differentiable-difg*: $\bigwedge m x. m < n \implies 0 \leq x \implies x \leq h \implies \text{difg } m \ \text{differentiable}$
 $(\text{at } x)$
using *difg-Suc real-differentiable-def* **by** *auto*
have *difg-Suc-eq-0*:
 $\bigwedge m t. m < n \implies 0 \leq t \implies t \leq h \implies \text{DERIV } (\text{difg } m) t \text{ :> } 0 \implies \text{difg } (\text{Suc}$
 $m) t = 0$
by $(\text{rule } \text{DERIV-unique } [OF \ \text{difg-Suc } [\text{rule-format}]]) \text{ simp}$

have $\exists t. 0 < t \wedge t < h \wedge \text{DERIV } (\text{difg } m) t \text{ :> } 0$
using $\langle m < n \rangle$
proof $(\text{induct } m)$
case 0
show $?case$

```

proof (rule Rolle)
  show  $0 < h$  by fact
  show  $\text{difg } 0 \ 0 = \text{difg } 0 \ h$ 
    by (simp add: difg-0 g2)
  show continuous-on  $\{0..h\}$  (difg 0)
    by (simp add: continuous-at-imp-continuous-on isCont-difg n)
qed (simp add: differentiable-difg n)
next
case (Suc m')
then obtain  $t$  where  $0 < t < h$   $\text{DERIV } (\text{difg } m') \ t \text{ :> } 0$ 
  by force
have  $\exists t'. 0 < t' \wedge t' < t \wedge \text{DERIV } (\text{difg } (\text{Suc } m')) \ t' \text{ :> } 0$ 
proof (rule Rolle)
  show  $0 < t$  by fact
  show  $\text{difg } (\text{Suc } m') \ 0 = \text{difg } (\text{Suc } m') \ t$ 
    using  $t \langle \text{Suc } m' < n \rangle$  by (simp add: difg-Suc-eq-0 difg-eq-0)
  have  $\bigwedge x. 0 \leq x \wedge x \leq t \implies \text{isCont } (\text{difg } (\text{Suc } m')) \ x$ 
    using  $\langle t < h \rangle \langle \text{Suc } m' < n \rangle$  by (simp add: isCont-difg)
  then show continuous-on  $\{0..t\}$  (difg (Suc m'))
    by (simp add: continuous-at-imp-continuous-on)
qed (use  $\langle t < h \rangle \langle \text{Suc } m' < n \rangle$  in  $\langle \text{simp add: differentiable-difg} \rangle$ )
with  $\langle t < h \rangle$  show ?case
  by auto
qed
then obtain  $t$  where  $0 < t < h$   $\text{difg } (\text{Suc } m) \ t = 0$ 
  using  $\langle m < n \rangle$  difg-Suc-eq-0 by force
show ?thesis
proof (intro exI conjI)
  show  $0 < t < h$  by fact+
  show  $f \ h = (\sum m < n. \text{diff } m \ 0 / (\text{fact } m) * h \wedge^m) + \text{diff } n \ t / (\text{fact } n) * h \wedge^n$ 
    using  $\langle \text{difg } (\text{Suc } m) \ t = 0 \rangle$  by (simp add: m f-h difg-def)
qed
qed

lemma Maclaurin2:
  fixes  $n :: \text{nat}$ 
  and  $h :: \text{real}$ 
  assumes INIT1:  $0 < h$ 
  and INIT2:  $\text{diff } 0 = f$ 
  and DERIV:  $\forall m \ t. m < n \wedge 0 \leq t \wedge t \leq h \implies \text{DERIV } (\text{diff } m) \ t \text{ :> } \text{diff}$ 
  (Suc m)  $t$ 
  shows  $\exists t. 0 < t \wedge t \leq h \wedge f \ h = (\sum m < n. \text{diff } m \ 0 / (\text{fact } m) * h \wedge^m) + \text{diff}$ 
   $n \ t / \text{fact } n * h \wedge^n$ 
proof (cases n)
  case 0
  with INIT1 INIT2 show ?thesis by fastforce
next
case Suc
  then have  $n > 0$  by simp

```


from *Maclaurin* [*OF INIT1 this INIT2 DERIV*]
show *?thesis* **by** *fastforce*
qed

lemma *Maclaurin-minus*:

fixes $n :: \text{nat}$ **and** $h :: \text{real}$
assumes $h < 0$ $0 < n$ $\text{diff } 0 = f$
and *DERIV*: $\forall m t. m < n \wedge h \leq t \wedge t \leq 0 \longrightarrow \text{DERIV } (\text{diff } m) t :> \text{diff } (\text{Suc } m) t$
shows $\exists t. h < t \wedge t < 0 \wedge f h = (\sum_{m < n}. \text{diff } m 0 / \text{fact } m * h ^ m) + \text{diff } n t / \text{fact } n * h ^ n$
proof –

Transform *ABL'* into *derivative-intros* format.

note *DERIV'* = *DERIV-chain'*[*OF - DERIV*[*rule-format*], *THEN DERIV-cong*]
let *?sum* = $\lambda t.$
 $(\sum_{m < n}. (-1) ^ m * \text{diff } m (-0) / (\text{fact } m) * (-h) ^ m) +$
 $(-1) ^ n * \text{diff } n (-t) / (\text{fact } n) * (-h) ^ n$
from *assms* **have** $\exists t > 0. t < -h \wedge f(-(-h)) = ?sum t$
by (*intro Maclaurin*) (*auto intro!*: *derivative-eq-intros DERIV'*)
then obtain t **where** $0 < t < -h \wedge f(-(-h)) = ?sum t$
by *blast*
moreover have $(-1) ^ n * \text{diff } n (-t) * (-h) ^ n / \text{fact } n = \text{diff } n (-t) * h ^ n / \text{fact } n$
by (*auto simp: power-mult-distrib[symmetric]*)
moreover
have $(\sum_{m < n}. (-1) ^ m * \text{diff } m 0 * (-h) ^ m / \text{fact } m) = (\sum_{m < n}. \text{diff } m 0 * h ^ m / \text{fact } m)$
by (*auto intro: sum.cong simp add: power-mult-distrib[symmetric]*)
ultimately have $h < -t \wedge -t < 0 \wedge$
 $f h = (\sum_{m < n}. \text{diff } m 0 / (\text{fact } m) * h ^ m) + \text{diff } n (-t) / (\text{fact } n) * h ^ n$
by *auto*
then show *?thesis* ..
qed

115.2 More Convenient "Bidirectional" Version.

lemma *Maclaurin-bi-le*:

fixes $n :: \text{nat}$ **and** $x :: \text{real}$
assumes $\text{diff } 0 = f$
and *DERIV*: $\forall m t. m < n \wedge |t| \leq |x| \longrightarrow \text{DERIV } (\text{diff } m) t :> \text{diff } (\text{Suc } m) t$
shows $\exists t. |t| \leq |x| \wedge f x = (\sum_{m < n}. \text{diff } m 0 / (\text{fact } m) * x ^ m) + \text{diff } n t / (\text{fact } n) * x ^ n$
(is $\exists t. - \wedge f x = ?f x t$ **)**
proof (*cases n = 0*)
case *True*
with $\langle \text{diff } 0 = f \rangle$ **show** *?thesis* **by** *force*
next
case *False*

```

show ?thesis
proof (cases rule: linorder-cases)
  assume  $x = 0$ 
  with  $\langle n \neq 0 \rangle \langle \text{diff } 0 = f \rangle$  DERIV have  $|0| \leq |x| \wedge f x = ?f x 0$ 
    by auto
  then show ?thesis ..
next
  assume  $x < 0$ 
  with  $\langle n \neq 0 \rangle$  DERIV have  $\exists t > x. t < 0 \wedge \text{diff } 0 x = ?f x t$ 
    by (intro Maclaurin-minus) auto
  then obtain  $t$  where  $x < t < 0$ 
     $\text{diff } 0 x = (\sum_{m < n}. \text{diff } m 0 / \text{fact } m * x \wedge m) + \text{diff } n t / \text{fact } n * x \wedge n$ 
    by blast
  with  $\langle x < 0 \rangle \langle \text{diff } 0 = f \rangle$  show ?thesis by force
next
  assume  $x > 0$ 
  with  $\langle n \neq 0 \rangle \langle \text{diff } 0 = f \rangle$  DERIV have  $\exists t > 0. t < x \wedge \text{diff } 0 x = ?f x t$ 
    by (intro Maclaurin) auto
  then obtain  $t$  where  $0 < t < x$ 
     $\text{diff } 0 x = (\sum_{m < n}. \text{diff } m 0 / \text{fact } m * x \wedge m) + \text{diff } n t / \text{fact } n * x \wedge n$ 
    by blast
  with  $\langle x > 0 \rangle \langle \text{diff } 0 = f \rangle$  have  $|t| \leq |x| \wedge f x = ?f x t$  by simp
  then show ?thesis ..
qed
qed

```

lemma Maclaurin-all-lt:

```

fixes  $x :: \text{real}$ 
assumes INIT1:  $\text{diff } 0 = f$ 
  and INIT2:  $0 < n$ 
  and INIT3:  $x \neq 0$ 
  and DERIV:  $\forall m x. \text{DERIV } (\text{diff } m) x \text{ :> } \text{diff } (\text{Suc } m) x$ 
shows  $\exists t. 0 < |t| \wedge |t| < |x| \wedge f x =$ 
   $(\sum_{m < n}. (\text{diff } m 0 / \text{fact } m) * x \wedge m) + (\text{diff } n t / \text{fact } n) * x \wedge n$ 
  (is  $\exists t. - \wedge - \wedge f x = ?f x t)$ 
proof (cases rule: linorder-cases)
  assume  $x = 0$ 
  with INIT3 show ?thesis ..
next
  assume  $x < 0$ 
  with assms have  $\exists t > x. t < 0 \wedge f x = ?f x t$ 
    by (intro Maclaurin-minus) auto
  then show ?thesis by force
next
  assume  $x > 0$ 
  with assms have  $\exists t > 0. t < x \wedge f x = ?f x t$ 
    by (intro Maclaurin) auto
  then show ?thesis by force
qed

```

lemma *Maclaurin-zero*: $x = 0 \implies n \neq 0 \implies (\sum_{m < n}. (\text{diff } m \ 0 / \text{fact } m) * x^{\wedge} m) = \text{diff } 0 \ 0$
for $x :: \text{real}$ **and** $n :: \text{nat}$
by *simp*

lemma *Maclaurin-all-le*:
fixes $x :: \text{real}$ **and** $n :: \text{nat}$
assumes *INIT*: $\text{diff } 0 = f$
and *DERIV*: $\forall m \ x. \text{DERIV } (\text{diff } m) \ x :> \text{diff } (\text{Suc } m) \ x$
shows $\exists t. |t| \leq |x| \wedge f \ x = (\sum_{m < n}. (\text{diff } m \ 0 / \text{fact } m) * x^{\wedge} m) + (\text{diff } n \ t / \text{fact } n) * x^{\wedge} n$
(is $\exists t. - \wedge f \ x = ?f \ x \ t)$
proof (*cases* $n = 0$)
case *True*
with *INIT* **show** *?thesis* **by** *force*
next
case *False*
show *?thesis*
using *DERIV* *INIT* *Maclaurin-bi-le* **by** *auto*
qed

lemma *Maclaurin-all-le-objl*:
 $\text{diff } 0 = f \wedge (\forall m \ x. \text{DERIV } (\text{diff } m) \ x :> \text{diff } (\text{Suc } m) \ x) \longrightarrow$
 $(\exists t :: \text{real}. |t| \leq |x| \wedge f \ x = (\sum_{m < n}. (\text{diff } m \ 0 / \text{fact } m) * x^{\wedge} m) + (\text{diff } n \ t / \text{fact } n) * x^{\wedge} n)$
for $x :: \text{real}$ **and** $n :: \text{nat}$
by (*blast intro: Maclaurin-all-le*)

115.3 Version for Exponential Function

lemma *Maclaurin-exp-lt*:
fixes $x :: \text{real}$ **and** $n :: \text{nat}$
shows
 $x \neq 0 \implies n > 0 \implies$
 $(\exists t. 0 < |t| \wedge |t| < |x| \wedge \exp \ x = (\sum_{m < n}. (x^{\wedge} m) / \text{fact } m) + (\exp \ t / \text{fact } n) * x^{\wedge} n)$
using *Maclaurin-all-lt* [**where** $\text{diff} = \lambda n. \exp$ **and** $f = \exp$ **and** $x = x$ **and** $n = n$] **by** *auto*

lemma *Maclaurin-exp-le*:
fixes $x :: \text{real}$ **and** $n :: \text{nat}$
shows $\exists t. |t| \leq |x| \wedge \exp \ x = (\sum_{m < n}. (x^{\wedge} m) / \text{fact } m) + (\exp \ t / \text{fact } n) * x^{\wedge} n$
using *Maclaurin-all-le-objl* [**where** $\text{diff} = \lambda n. \exp$ **and** $f = \exp$ **and** $x = x$ **and** $n = n$] **by** *auto*

corollary *exp-lower-Taylor-quadratic*: $0 \leq x \implies 1 + x + x^2 / 2 \leq \exp \ x$

for $x :: \text{real}$
using *Maclaurin-exp-le* [of x 3] **by** (*auto simp: numeral-3-eq-3 power2-eq-square*)

corollary *ln-2-less-1*: $\ln 2 < (1 :: \text{real})$
by (*smt (verit) ln-eq-minus-one ln-le-minus-one*)

115.4 Version for Sine Function

lemma *mod-exhaust-less-4*: $m \bmod 4 = 0 \vee m \bmod 4 = 1 \vee m \bmod 4 = 2 \vee m \bmod 4 = 3$
for $m :: \text{nat}$
by *auto*

It is unclear why so many variant results are needed.

lemma *sin-expansion-lemma*: $\sin (x + \text{real} (\text{Suc } m) * \text{pi} / 2) = \cos (x + \text{real } m * \text{pi} / 2)$
by (*auto simp: cos-add sin-add add-divide-distrib distrib-right*)

lemma *Maclaurin-sin-expansion2*:

$\exists t. |t| \leq |x| \wedge$

$\sin x = (\sum m < n. \text{sin-coeff } m * x \wedge m) + (\sin (t + 1/2 * \text{real } n * \text{pi}) / \text{fact } n) * x \wedge n$

proof (*cases* $n = 0 \vee x = 0$)

case *False*

let $?diff = \lambda n x. \sin (x + 1/2 * \text{real } n * \text{pi})$

have $\exists t. 0 < |t| \wedge |t| < |x| \wedge \sin x =$

$(\sum m < n. (?diff m 0 / \text{fact } m) * x \wedge m) + (?diff n t / \text{fact } n) * x \wedge n$

proof (*rule* *Maclaurin-all-lt*)

show $\forall m x. ((\lambda t. \sin (t + 1/2 * \text{real } m * \text{pi})) \text{has-real-derivative}$

$\sin (x + 1/2 * \text{real} (\text{Suc } m) * \text{pi})) (at x)$

by (*rule* *allI derivative-eq-intros* | *use* *sin-expansion-lemma in force*)+

qed (*use* *False in auto*)

then show *?thesis*

apply (*rule* *ex-forward, simp*)

apply (*rule* *sum.cong[OF refl]*)

apply (*auto simp: sin-coeff-def sin-zero-iff elim: oddE simp del: of-nat-Suc*)

done

qed *auto*

lemma *Maclaurin-sin-expansion*:

$\exists t. \sin x = (\sum m < n. \text{sin-coeff } m * x \wedge m) + (\sin (t + 1/2 * \text{real } n * \text{pi}) / \text{fact } n) * x \wedge n$

using *Maclaurin-sin-expansion2* [of x n] **by** *blast*

lemma *Maclaurin-sin-expansion3*:

assumes $n > 0 \ x > 0$

shows $\exists t. 0 < t \wedge t < x \wedge$

$\sin x = (\sum m < n. \text{sin-coeff } m * x \wedge m) + (\sin (t + 1/2 * \text{real } n * \text{pi}) / \text{fact } n) * x \wedge n$

proof –
let $?diff = \lambda n x. \sin (x + 1/2 * \text{real } n * \text{pi})$
have $\exists t. 0 < t \wedge t < x \wedge \sin x = (\sum m < n. ?diff m 0 / (\text{fact } m) * x \wedge m) +$
 $?diff n t / \text{fact } n * x \wedge n$
proof (rule Maclaurin)
show $\forall m t. m < n \wedge 0 \leq t \wedge t \leq x \longrightarrow$
 $((\lambda u. \sin (u + 1/2 * \text{real } m * \text{pi})) \text{ has-real-derivative}$
 $\sin (t + 1/2 * \text{real } (\text{Suc } m) * \text{pi})) (\text{at } t)$
apply (simp add: sin-expansion-lemma del: of-nat-Suc)
apply (force intro!: derivative-eq-intros)
done
qed (use assms in auto)
then show ?thesis
apply (rule ex-forward, simp)
apply (rule sum.cong[OF refl])
apply (auto simp: sin-coeff-def sin-zero-iff elim: oddE simp del: of-nat-Suc)
done
qed

lemma Maclaurin-sin-expansion4:

assumes $0 < x$
shows $\exists t. 0 < t \wedge t \leq x \wedge \sin x = (\sum m < n. \text{sin-coeff } m * x \wedge m) + (\sin (t +$
 $1/2 * \text{real } n * \text{pi}) / \text{fact } n) * x \wedge n$
proof –
let $?diff = \lambda n x. \sin (x + 1/2 * \text{real } n * \text{pi})$
have $\exists t. 0 < t \wedge t \leq x \wedge \sin x = (\sum m < n. ?diff m 0 / (\text{fact } m) * x \wedge m) +$
 $?diff n t / \text{fact } n * x \wedge n$
proof (rule Maclaurin2)
show $\forall m t. m < n \wedge 0 \leq t \wedge t \leq x \longrightarrow$
 $((\lambda u. \sin (u + 1/2 * \text{real } m * \text{pi})) \text{ has-real-derivative}$
 $\sin (t + 1/2 * \text{real } (\text{Suc } m) * \text{pi})) (\text{at } t)$
apply (simp add: sin-expansion-lemma del: of-nat-Suc)
apply (force intro!: derivative-eq-intros)
done
qed (use assms in auto)
then show ?thesis
apply (rule ex-forward, simp)
apply (rule sum.cong[OF refl])
apply (auto simp: sin-coeff-def sin-zero-iff elim: oddE simp del: of-nat-Suc)
done
qed

115.5 Maclaurin Expansion for Cosine Function

lemma sumr-cos-zero-one [simp]: $(\sum m < \text{Suc } n. \text{cos-coeff } m * 0 \wedge m) = 1$
by (induct n) auto

lemma cos-expansion-lemma: $\cos (x + \text{real } (\text{Suc } m) * \text{pi} / 2) = - \sin (x + \text{real } m * \text{pi} / 2)$

by (auto simp: cos-add sin-add distrib-right add-divide-distrib)

lemma *Maclaurin-cos-expansion*:

$\exists t::\text{real}. |t| \leq |x| \wedge$

$\cos x = (\sum m < n. \text{cos-coeff } m * x ^ m) + (\cos(t + 1/2 * \text{real } n * \text{pi}) / \text{fact } n) * x ^ n$

proof (cases $n = 0 \vee x = 0$)

case *False*

let $?diff = \lambda n x. \cos(x + 1/2 * \text{real } n * \text{pi})$

have $\exists t. 0 < |t| \wedge |t| < |x| \wedge \cos x =$

$(\sum m < n. (?diff m 0 / \text{fact } m) * x ^ m) + (?diff n t / \text{fact } n) * x ^ n$

proof (rule *Maclaurin-all-lt*)

show $\forall m x. ((\lambda t. \cos(t + 1/2 * \text{real } m * \text{pi})) \text{has-real-derivative}$

$\cos(x + 1/2 * \text{real } (\text{Suc } m) * \text{pi})) (at x)$

using *cos-expansion-lemma*

by (intro *allI derivative-eq-intros* | *simp*)+

qed (use *False in auto*)

then show *?thesis*

apply (rule *ex-forward, simp*)

apply (rule *sum.cong[OF refl]*)

apply (auto simp: *cos-coeff-def cos-zero-iff elim: evenE simp del: of-nat-Suc*)

done

qed *auto*

lemma *Maclaurin-cos-expansion2*:

assumes $x > 0 \ n > 0$

shows $\exists t. 0 < t \wedge t < x \wedge$

$\cos x = (\sum m < n. \text{cos-coeff } m * x ^ m) + (\cos(t + 1/2 * \text{real } n * \text{pi}) / \text{fact } n) * x ^ n$

proof –

let $?diff = \lambda n x. \cos(x + 1/2 * \text{real } n * \text{pi})$

have $\exists t. 0 < t \wedge t < x \wedge \cos x = (\sum m < n. ?diff m 0 / (\text{fact } m) * x ^ m) +$

$?diff n t / \text{fact } n * x ^ n$

proof (rule *Maclaurin*)

show $\forall m t. m < n \wedge 0 \leq t \wedge t \leq x \longrightarrow$

$((\lambda u. \cos(u + 1/2 * \text{real } m * \text{pi})) \text{has-real-derivative}$

$\cos(t + 1/2 * \text{real } (\text{Suc } m) * \text{pi})) (at t)$

by (*simp add: cos-expansion-lemma del: of-nat-Suc*)

qed (use *assms in auto*)

then show *?thesis*

apply (rule *ex-forward, simp*)

apply (rule *sum.cong[OF refl]*)

apply (auto simp: *cos-coeff-def cos-zero-iff elim: evenE*)

done

qed

lemma *Maclaurin-minus-cos-expansion*:

assumes $n > 0 \ x < 0$

shows $\exists t. x < t \wedge t < 0 \wedge$

$$\cos x = (\sum m < n. \text{cos-coeff } m * x \wedge m) + ((\cos (t + 1/2 * \text{real } n * \text{pi}) / \text{fact } n) * x \wedge n)$$

proof –

let $?diff = \lambda n x. \cos (x + 1/2 * \text{real } n * \text{pi})$

have $\exists t. x < t \wedge t < 0 \wedge \cos x = (\sum m < n. ?diff m 0 / (\text{fact } m) * x \wedge m) + ?diff n t / \text{fact } n * x \wedge n$

proof (rule *Maclaurin-minus*)

show $\forall m t. m < n \wedge x \leq t \wedge t \leq 0 \longrightarrow$
 $((\lambda u. \cos (u + 1 / 2 * \text{real } m * \text{pi})) \text{ has-real-derivative}$
 $\cos (t + 1 / 2 * \text{real } (\text{Suc } m) * \text{pi})) (\text{at } t)$

by (*simp add: cos-expansion-lemma del: of-nat-Suc*)

qed (*use assms in auto*)

then show $?thesis$

apply (*rule ex-forward, simp*)

apply (*rule sum.cong[OF refl]*)

apply (*auto simp: cos-coeff-def cos-zero-iff elim: evenE*)

done

qed

lemma *sin-bound-lemma*: $x = y \implies |u| \leq v \implies |(x + u) - y| \leq v$
for $x y u v :: \text{real}$
by *auto*

lemma *Maclaurin-sin-bound*: $|\sin x - (\sum m < n. \text{sin-coeff } m * x \wedge m)| \leq \text{inverse} (\text{fact } n) * |x| \wedge n$

proof –

have *est*: $x \leq 1 \implies 0 \leq y \implies x * y \leq 1 * y$ **for** $x y :: \text{real}$

by (*rule mult-right-mono simp-all*)

let $?diff = \lambda (n::\text{nat}) (x::\text{real}).$
 $\text{if } n \bmod 4 = 0 \text{ then } \sin x$
 $\text{else if } n \bmod 4 = 1 \text{ then } \cos x$
 $\text{else if } n \bmod 4 = 2 \text{ then } - \sin x$
 $\text{else } - \cos x$

have *diff-0*: $?diff 0 = \sin$ **by** *simp*

have *DERIV* ($?diff m$) $x :> ?diff (\text{Suc } m) x$ **for** m **and** x
using *mod-exhaust-less-4 [of m]*

by (*auto simp: mod-Suc intro!: derivative-eq-intros*)

then have *DERIV-diff*: $\forall m x. \text{DERIV } (?diff m) x :> ?diff (\text{Suc } m) x$
by *blast*

from *Maclaurin-all-le [OF diff-0 DERIV-diff]*

obtain t **where** $t1: |t| \leq |x|$
and $t2: \sin x = (\sum m < n. ?diff m 0 / (\text{fact } m) * x \wedge m) + ?diff n t / (\text{fact } n) * x \wedge n$

by *fast*

have *diff-m-0*: $?diff m 0 = (\text{if even } m \text{ then } 0 \text{ else } (- 1) \wedge ((m - \text{Suc } 0) \text{ div } 2))$

```

for  $m$ 
  using mod-exhaust-less-4 [of m]
  by (auto simp: minus-one-power-iff even-even-mod-4-iff [of m] dest: even-mod-4-div-2
odd-mod-4-div-2)
  show ?thesis
    apply (subst t2)
    apply (rule sin-bound-lemma)
    apply (rule sum.cong[OF refl])
    unfolding sin-coeff-def
    apply (subst diff-m-0, simp)
    using est
    apply (auto intro: mult-right-mono [where  $b=1$ , simplified] mult-right-mono
simp: ac-simps divide-inverse power-abs [symmetric] abs-mult)
  done
qed

```

116 Taylor series

We use MacLaurin and the translation of the expansion point c to 0 to prove Taylor’s theorem.

lemma *Taylor-up*:

```

assumes INIT:  $n > 0$  diff 0 = f
  and DERIV:  $\forall m t. m < n \wedge a \leq t \wedge t \leq b \longrightarrow \text{DERIV } (\text{diff } m) t \text{ :> } (\text{diff } (\text{Suc } m) t)$ 
  and INTERV:  $a \leq c < b$ 
shows  $\exists t::\text{real}. c < t \wedge t < b \wedge$ 
   $f b = (\sum_{m < n}. (\text{diff } m c / \text{fact } m) * (b - c)^{\wedge m}) + (\text{diff } n t / \text{fact } n) * (b - c)^{\wedge n}$ 

```

proof –

```

from INTERV have  $0 < b - c$  by arith
moreover from INIT have  $n > 0$   $(\lambda x. \text{diff } m (x + c)) 0 = (\lambda x. f (x + c))$ 
  by auto
moreover
have  $\forall m t. m < n \wedge 0 \leq t \wedge t \leq b - c \longrightarrow \text{DERIV } (\lambda x. \text{diff } m (x + c)) t \text{ :> } \text{diff } (\text{Suc } m) (t + c)$ 
proof (intro strip)
  fix  $m t$ 
  assume  $m < n \wedge 0 \leq t \wedge t \leq b - c$ 
  with DERIV and INTERV have  $\text{DERIV } (\text{diff } m) (t + c) \text{ :> } \text{diff } (\text{Suc } m) (t + c)$ 
  by auto
  moreover from DERIV-ident and DERIV-const have  $\text{DERIV } (\lambda x. x + c) t \text{ :> } 1 + 0$ 
  by (rule DERIV-add)
  ultimately have  $\text{DERIV } (\lambda x. \text{diff } m (x + c)) t \text{ :> } \text{diff } (\text{Suc } m) (t + c) * (1 + 0)$ 
  by (rule DERIV-chain2)
  then show  $\text{DERIV } (\lambda x. \text{diff } m (x + c)) t \text{ :> } \text{diff } (\text{Suc } m) (t + c)$ 

```


by *simp*
qed
 ultimately obtain x where
 $0 < x \wedge x < b - c \wedge$
 $f (b - c + c) =$
 $(\sum m < n. \text{diff } m (0 + c) / \text{fact } m * (b - c) ^ m) + \text{diff } n (x + c) / \text{fact } n$
 $* (b - c) ^ n$
 by (rule *Maclaurin* [THEN *exE*])
 then show ?thesis
 by (smt (verit) *sum.cong*)
qed

lemma *Taylor-down*:

fixes $a :: \text{real}$ and $n :: \text{nat}$
 assumes *INIT*: $n > 0$ $\text{diff } 0 = f$
 and *DERIV*: $(\forall m t. m < n \wedge a \leq t \wedge t \leq b \longrightarrow \text{DERIV } (\text{diff } m) t \text{ :> } \text{diff}$
 $(\text{Suc } m) t)$
 and *INTERV*: $a < c \leq b$
 shows $\exists t. a < t \wedge t < c \wedge$
 $f a = (\sum m < n. (\text{diff } m c / \text{fact } m) * (a - c) ^ m) + (\text{diff } n t / \text{fact } n) * (a -$
 $c) ^ n$
proof –
 from *INTERV* have $a - c < 0$ by *arith*
 moreover from *INIT* have $n > 0$ $(\lambda x. \text{diff } m (x + c)) 0 = (\lambda x. f (x + c))$
 by *auto*
 moreover
 have $\forall m t. m < n \wedge a - c \leq t \wedge t \leq 0 \longrightarrow \text{DERIV } (\lambda x. \text{diff } m (x + c)) t \text{ :>}$
 $\text{diff } (\text{Suc } m) (t + c)$
proof (rule *allI impI*) +
 fix $m t$
 assume $m < n \wedge a - c \leq t \wedge t \leq 0$
 with *DERIV* and *INTERV* have $\text{DERIV } (\text{diff } m) (t + c) \text{ :> } \text{diff } (\text{Suc } m) (t$
 $+ c)$
 by *auto*
 moreover from *DERIV-ident* and *DERIV-const* have $\text{DERIV } (\lambda x. x + c) t$
 $\text{:> } 1 + 0$
 by (rule *DERIV-add*)
 ultimately show $\text{DERIV } (\lambda x. \text{diff } m (x + c)) t \text{ :> } \text{diff } (\text{Suc } m) (t + c)$
 using *DERIV-chain2* *DERIV-shift* by *blast*
qed
 ultimately obtain x where
 $a - c < x \wedge x < 0 \wedge$
 $f (a - c + c) =$
 $(\sum m < n. \text{diff } m (0 + c) / \text{fact } m * (a - c) ^ m) + \text{diff } n (x + c) / \text{fact } n$
 $* (a - c) ^ n$
 by (rule *Maclaurin-minus* [THEN *exE*])
 then have $a < x + c \wedge x + c < c \wedge$
 $f a = (\sum m < n. \text{diff } m c / \text{fact } m * (a - c) ^ m) + \text{diff } n (x + c) / \text{fact } n *$
 $(a - c) ^ n$

by *fastforce*
 then show *?thesis* by *fastforce*
 qed

theorem *Taylor*:

fixes $a :: \text{real}$ and $n :: \text{nat}$
 assumes *INIT*: $n > 0$ *diff* $0 = f$
 and *DERIV*: $\forall m t. m < n \wedge a \leq t \wedge t \leq b \longrightarrow \text{DERIV } (\text{diff } m) t :> \text{diff}$
 (*Suc m*) t
 and *INTERV*: $a \leq c \ c \leq b \ a \leq x \ x \leq b \ x \neq c$
 shows $\exists t.$
 (*if* $x < c$ then $x < t \wedge t < c$ else $c < t \wedge t < x$) \wedge
 $f x = (\sum_{m < n. (\text{diff } m c / \text{fact } m) * (x - c)^{\wedge m}) + (\text{diff } n t / \text{fact } n) * (x -$
 $c)^{\wedge n}$
proof (*cases* $x < c$)
 case *True*
 note *INIT*
 moreover have $\forall m t. m < n \wedge x \leq t \wedge t \leq b \longrightarrow \text{DERIV } (\text{diff } m) t :> \text{diff}$
 (*Suc m*) t
 using *DERIV* and *INTERV* by *fastforce*
 ultimately show *?thesis*
 using *True INTERV Taylor-down* by *simp*
 next
 case *False*
 note *INIT*
 moreover have $\forall m t. m < n \wedge a \leq t \wedge t \leq x \longrightarrow \text{DERIV } (\text{diff } m) t :> \text{diff}$
 (*Suc m*) t
 using *DERIV* and *INTERV* by *fastforce*
 ultimately show *?thesis*
 using *Taylor-up INTERV False* by *simp*
 qed

end

117 Comprehensive Complex Theory

theory *Complex-Main*

imports

Complex

MacLaurin

begin

end

References

- [1] H. Davenport. *The Higher Arithmetic*. Cambridge University Press, 1992.
- [2] M. Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.
- [3] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison–Wesley, Boston, MA, USA, 2nd edition, 1994.
- [4] M. Holz, K. Steffens, and E. Weitz. *Introduction to Cardinal Arithmetic*. Birkhäuser, 1999.
- [5] D. Leijen. Division and modulus for computer scientists. 2001.
- [6] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.