

The Supplemental Isabelle/HOL Library

December 17, 2025

Contents

1	Implementation of Association Lists	21
1.1	<i>update</i> and <i>updates</i>	21
1.2	<i>delete</i>	23
1.3	<i>update-with-aux</i> and <i>delete-aux</i>	24
1.4	<i>restrict</i>	26
1.5	<i>clearjunk</i>	27
1.6	<i>map-ran</i>	28
1.7	<i>merge</i>	29
1.8	<i>compose</i>	30
1.9	<i>map-entry</i>	32
1.10	<i>map-default</i>	32
2	Axiomatic Declaration of Bounded Natural Functors	33
3	Generalized Corecursor Sugar (<i>corec</i> and friends)	33
3.1	Coinduction	34
4	A general “while” combinator	36
4.1	<i>while-option</i>	37
4.2	<i>while</i>	38
4.3	Termination, <i>lfp</i> and <i>gfp</i>	38
4.4	<i>while-Some</i> and <i>while-saturate</i>	40
4.5	Reflexive, transitive closure	41
5	The Bourbaki-Witt tower construction for transfinite iteration	42
5.1	Connect with the while combinator for executability on chain-finite lattices.	45
6	Division with modulus centered towards zero.	47
7	Order on characters	50

8	A generic phantom type	51
9	Cardinality of types	51
9.1	Preliminary lemmas	51
9.2	Cardinalities of types	52
9.3	Classes with at least 1 and 2	53
9.4	A type class for deciding finiteness of types	53
9.5	A type class for computing the cardinality of types	53
9.6	Instantiations for <i>card-UNIV</i>	54
10	Code setup for sets with cardinality type information	57
11	Eliminating pattern matches	60
12	Lazy types in generated code	60
12.1	The type <i>lazy</i>	61
12.2	Implementation	63
13	Test infrastructure for the code generator	63
13.1	YXML encoding for <i>term</i>	63
13.2	Test engine and drivers	65
14	A combinator to build partial equivalence relations from a predicate and an equivalence relation	66
15	Formalisation of chain-complete partial orders, continuity and admissibility	67
15.1	Continuity	69
15.1.1	Theorem collection <i>cont-intro</i>	70
15.2	Admissibility	76
15.3	(=) as order	80
15.4	ccpo for products	81
15.5	Complete lattices as ccpo	85
15.6	Parallel fixpoint induction	88
16	Confluence	92
17	Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums	95
17.1	The datatype universe	95
17.2	Freeness: Distinctness of Constructors	97
17.3	Set Constructions	100

18 Bijections between natural numbers and other types	104
18.1 Type $\text{nat} \times \text{nat}$	105
18.2 Type $\text{nat} + \text{nat}$	106
18.3 Type int	107
18.4 Type nat list	108
18.5 Finite sets of naturals	109
18.5.1 Preliminaries	109
18.5.2 From sets to naturals	109
18.5.3 From naturals to sets	109
18.5.4 Proof of isomorphism	110
19 Encoding (almost) everything into natural numbers	110
19.1 The class of countable types	110
19.2 Conversion functions	111
19.3 Finite types are countable	111
19.4 Automatically proving countability of old-style datatypes	111
19.5 Automatically proving countability of datatypes	112
19.6 More Countable types	112
19.7 The rationals are countably infinite	113
20 Infinite Sets and Related Concepts	114
20.1 The set of natural numbers is infinite	114
20.2 The set of integers is also infinite	115
20.3 Infinitely Many and Almost All	115
20.4 Enumeration of an Infinite Set	118
20.5 Properties of <i>wellorder-class.enumerate</i> on finite sets	120
21 Countable sets	121
21.1 Predicate for countable sets	121
21.2 Enumerate a countable set	122
21.3 Closure properties of countability	125
21.4 Misc lemmas	127
21.5 Uncountable	129
22 Countable Complete Lattices	129
22.0.1 Instances of countable complete lattices	135
23 Type of (at Most) Countable Sets	135
23.1 Cardinal stuff	135
23.2 The type of countable sets	136
23.3 Additional lemmas	142
23.3.1 <i>empty</i>	142
23.3.2 <i>cinsert</i>	142
23.3.3 <i>cimage</i>	143

23.3.4	bounded quantification	143
23.3.5	<i>cUnion</i>	143
23.4	Setup for Lifting/Transfer	143
23.4.1	Relator and predicator properties	143
23.4.2	Transfer rules for the Transfer package	144
23.5	Registration as BNF	145
24	Debugging facilities for code generated towards Isabelle/ML	146
25	Sequence of Properties on Subsequences	147
26	Common discrete functions	149
26.1	Discrete logarithm	149
26.2	Discrete square root	150
27	Pi and Function Sets	153
27.1	Basic Properties of <i>Pi</i>	154
27.2	Composition With a Restricted Domain: <i>compose</i>	156
27.3	Bounded Abstraction: <i>restrict</i>	156
27.4	Bijections Between Sets	157
27.5	Extensionality	158
27.6	Cardinality	159
27.7	Extensional Function Spaces	159
27.7.1	Injective Extensional Function Spaces	162
27.7.2	Misc properties of functions, composition and restriction from HOL Light	162
27.7.3	Cardinality	163
27.8	The pigeonhole principle	163
27.9	Products of sums	164
28	Partitions and Disjoint Sets	164
28.1	Set of Disjoint Sets	164
28.1.1	Family of Disjoint Sets	165
28.2	Construct Disjoint Sequences	167
28.3	Partitions	168
28.4	Constructions of partitions	168
28.5	Finiteness of partitions	169
28.6	Equivalence of partitions and equivalence classes	169
28.7	Refinement of partitions	170
28.8	The coarsest common refinement of a set of partitions	171
29	Type of finite sets defined as a subtype of sets	171
29.1	Definition of the type	172
29.2	Basic operations and type class instantiations	172
29.3	Other operations	175

29.4	Transferred lemmas from Set.thy	177
29.5	Additional lemmas	193
29.5.1	<i>ffUnion</i>	193
29.5.2	<i>fbind</i>	193
29.5.3	<i>fsingleton</i>	193
29.5.4	<i>fempty</i>	193
29.5.5	<i>fset</i>	194
29.5.6	<i>ffilter</i>	194
29.5.7	<i>fset-of-list</i>	194
29.5.8	<i>finsert</i>	195
29.5.9	<i>fimage</i>	195
29.5.10	bounded quantification	195
29.5.11	<i>fcard</i>	196
29.5.12	<i>sorted-list-of-fset</i>	198
29.5.13	<i>ffold</i>	198
29.5.14	(\subset)	199
29.5.15	Group operations	199
29.5.16	Semilattice operations	200
29.6	Choice in fsets	202
29.7	Induction and Cases rules for fsets	202
29.8	Lemmas depending on induction	202
29.9	Setup for Lifting/Transfer	203
29.9.1	Relator and predicator properties	203
29.9.2	Transfer rules for the Transfer package	203
29.10	BNF setup	205
29.11	Size setup	206
29.12	Advanced relator customization	206
29.12.1	Countability	206
29.13	Quickcheck setup	207
29.14	Code Generation Setup	208
30	Type of finite maps defined as a subtype of maps	209
30.1	Auxiliary constants and lemmas over <i>map</i>	209
30.2	Abstract characterisation	210
30.3	Operations	211
30.4	BNF setup	224
30.5	<i>size</i> setup	228
30.6	Additional operations	228
30.7	Additional properties	230
30.8	Lifting/transfer setup	230
30.9	View as datatype	230
30.10	Code setup	231
30.11	Instances	232
30.12	Tests	233

31 Disjoint FSets	233
32 Lists with elements distinct as canonical example for datatype invariants	234
32.1 The type of distinct lists	235
32.2 Executable version obeying invariant	237
32.3 Induction principle and case distinction	238
32.4 Functorial structure	238
32.5 Quickcheck generators	238
32.6 BNF instance	238
33 Type of dual ordered lattices	239
33.1 Pointwise ordering	241
33.2 Binary infimum and supremum	242
33.3 Top and bottom elements	243
33.4 Complement	244
33.5 Complete lattice operations	245
34 Equipollence and Other Relations Connected with Cardinality	246
34.1 Eqpoll	246
34.2 The strict relation	249
34.3 Mapping by an injection	250
34.4 Inserting elements into sets	250
34.5 Binary sums and unions	251
34.6 Binary Cartesian products	251
34.7 General Unions	252
34.8 General Cartesian products (Pi)	253
34.9 Misc other resultd	254
35 Continuity and iterations	258
35.1 Continuity for complete lattices	259
35.1.1 Least fixed points in countable complete lattices	261
36 Extended natural numbers (i.e. with infinity)	262
36.1 Type definition	262
36.2 Constructors and numbers	263
36.3 Addition	265
36.4 Multiplication	265
36.5 Numerals	266
36.6 Subtraction	266
36.7 Ordering	267
36.8 Cancellation simprocs	270
36.9 Well-ordering	271

36.10	Complete Lattice	271
36.11	Traditional theorem names	272
37	Liminf and Limsup on conditionally complete lattices	272
37.0.1	<i>Liminf</i> and <i>Limsup</i>	274
37.1	More Limits	278
38	Extended real number line	279
38.1	Definition and basic properties	281
38.1.1	Addition	283
38.1.2	Linear order on <i>ereal</i>	285
38.1.3	Multiplication	291
38.1.4	Power	297
38.1.5	Subtraction	297
38.1.6	Division	301
38.2	Complete lattice	305
38.3	Extended real intervals	307
38.4	Topological space	309
38.5	Relation to <i>enat</i>	315
38.6	Limits on <i>ereal</i>	316
38.6.1	Convergent sequences	318
38.6.2	Sums	322
38.6.3	Continuity	328
38.6.4	liminf and limsup	330
38.6.5	Tests for code generator	333
39	Indicator Function	333
40	The type of non-negative extended real numbers	336
40.1	Defining the extended non-negative reals	339
40.2	Cancellation simprocs	342
40.3	Order with top	342
40.4	Arithmetic	345
40.5	Coercion from <i>real</i> to <i>ennreal</i>	349
40.6	Coercion from <i>ennreal</i> to <i>real</i>	353
40.7	Coercion from <i>enat</i> to <i>ennreal</i>	354
40.8	Topology on <i>ennreal</i>	355
40.9	Approximation lemmas	362
40.10	<i>ennreal</i> theorems	363
41	Logarithm of Natural Numbers	367
41.1	Preliminaries	367
41.2	Floorlog	367
41.3	369

41.4 Bitlen	370
42 Various algebraic structures combined with a lattice	372
42.1 Positive Part, Negative Part, Absolute Value	373
43 Floating-Point Numbers	377
43.1 Real operations preserving the representation as floating point number	377
43.2 Arithmetic operations on floating point numbers	379
43.3 Quickcheck	381
43.4 Represent floats as unique mantissa and exponent	382
43.5 Compute arithmetic operations	384
43.6 Lemmas for types <i>real</i> , <i>nat</i> , <i>int</i>	385
43.7 Rounding Real Numbers	385
43.8 Rounding Floats	387
43.9 Truncating Real Numbers	388
43.10 Truncating Floats	390
43.11 Approximation of positive rationals	392
43.12 Division	394
43.13 Approximate Addition	394
43.14 Approximate Multiplication	397
43.15 Approximate Power	398
43.16 Lemmas needed by Approximate	400
44 Pointwise instantiation of functions to algebra type classes	404
45 Pointwise instantiation of functions to division	408
45.1 Syntactic with division	408
46 Lexicographic order on functions	409
47 The <i>going-to</i> filter	410
48 Big sum and product over function bodies	412
48.1 Abstract product	413
48.2 Concrete sum	414
48.3 Concrete product	415
49 Infinite Type Class	416
50 Algebraic operations on sets	417
51 Interval Type	423
51.1 Membership	428
51.2 Quickcheck	435

52 Approximate Operations on Intervals of Floating Point Numbers	436
52.1 Intervals with Floating Point Bounds	437
52.2 intros for <i>real-interval</i>	438
52.3 bounds for lists	439
52.4 constants for code generation	442
53 Immutable Arrays with Code Generation	442
53.1 Fundamental operations	442
53.2 Generic code equations	443
53.3 Auxiliary operations for code generation	444
53.4 Code Generation for SML	445
53.5 Code Generation for Haskell	445
54 Definition of Landau symbols	446
54.1 Definition of Landau symbols	447
54.2 Landau symbols and limits	460
54.3 Flatness of real functions	466
54.4 Asymptotic Equivalence	467
55 Values extended by a bottom element	474
55.1 Values extended by a top element	476
55.2 Values extended by a top and a bottom element	478
56 Infinite Streams	480
56.1 prepend list to stream	481
56.2 set of streams with elements in some fixed set	482
56.3 nth, take, drop for streams	483
56.4 unary predicates lifted to streams	485
56.5 recurring stream out of a list	486
56.6 iterated application of a function	487
56.7 stream repeating a single element	487
56.8 stream of natural numbers	488
56.9 flatten a stream of lists	488
56.10 merge a stream of streams	488
56.11 product of two streams	489
56.12 interleave two streams	489
56.13 zip	489
56.14 zip via function	490
57 List prefixes, suffixes, and homeomorphic embedding	491
57.1 Prefix order on lists	491
57.2 Basic properties of prefixes	492
57.3 Prefixes	494

57.4	Longest Common Prefix	495
57.5	Parallel lists	497
57.6	Suffix order on lists	497
57.7	Suffixes	501
57.8	Homeomorphic embedding on lists	503
57.9	Subsequences (special case of homeomorphic embedding) . . .	504
57.10	Appending elements	506
57.11	Relation to standard list operations	506
57.12	Contiguous sublists	507
57.12.1	<i>sublist</i>	507
57.12.2	<i>sublists</i>	509
57.13	Parametricity	509
58	Linear Temporal Logic on Streams	511
59	Preliminaries	511
60	Linear temporal logic	511
61	Weak vs. strong until (contributed by Michael Foster, University of Sheffield)	521
62	Lists as vectors	522
62.1	+ and −	523
62.2	Inner product	524
63	Definitions of Least Upper Bounds and Greatest Lower Bounds	525
63.1	Rules for the Relations $* \leq$ and $\leq *$	525
63.2	Rules about the Operators <i>leastP</i> , <i>ub</i> and <i>lub</i>	526
63.3	Rules about the Operators <i>greatestP</i> , <i>isLb</i> and <i>isGlb</i>	527
64	An abstract view on maps for code generation.	530
64.1	Parametricity transfer rules	530
64.2	Type definition and primitive operations	532
64.3	Functorial structure	533
64.4	Derived operations	533
64.5	Properties	535
64.5.1	<i>entries</i> , <i>ordered-entries</i> , and <i>fold</i>	542
64.6	Code generator setup	545
65	Monad notation for arbitrary types	545
66	Less common functions on lists	547

67 (Finite) Multisets	554
67.1 The type of multisets	554
67.2 Representing multisets	555
67.3 Basic operations	556
67.3.1 Conversion to set and membership	556
67.3.2 Union	558
67.3.3 Difference	559
67.3.4 Min and Max	561
67.3.5 Equality of multisets	561
67.3.6 Pointwise ordering induced by count	563
67.3.7 Intersection and bounded union	566
67.3.8 Additional intersection facts	567
67.3.9 Additional bounded union facts	569
67.4 Replicate and repeat operations	570
67.4.1 Simprocs	571
67.4.2 Conditionally complete lattice	572
67.4.3 Filter (with comprehension syntax)	574
67.4.4 Size	576
67.5 Induction and case splits	578
67.5.1 Strong induction and subset induction for multisets	579
67.6 Least and greatest elements	579
67.7 The fold combinator	580
67.8 Image	581
67.9 Further conversions	584
67.10 More properties of the replicate, repeat, and image operations	589
67.11 Big operators	591
67.12 Multiset as order-ignorant lists	598
67.13 The multiset order	600
67.13.1 Well-foundedness	601
67.13.2 Closure-free presentation	602
67.13.3 Monotonicity	602
67.13.4 The multiset extension is cancellative for multiset union	603
67.13.5 Strict partial-order properties	604
67.13.6 Strict total-order properties	605
67.14 Quasi-executable version of the multiset extension	606
67.14.1 Monotonicity of multiset union	607
67.14.2 Termination proofs with multiset orders	607
67.15 Legacy theorem bindings	608
67.16 Naive implementation using lists	609
67.17 BNF setup	612
67.18 Size setup	614
67.19 Lemmas about Size	614
67.20 The set of multisets of a given size	615

68 More Theorems about the Multiset Order	617
68.1 Alternative Characterizations	617
68.1.1 The Dershowitz–Manna Ordering	617
68.1.2 The Huet–Oppen Ordering	617
68.1.3 Monotonicity	618
68.1.4 Properties of Orders	618
68.1.5 Simplifications	622
68.2 Simprocs	623
68.3 Additional facts and instantiations	623
69 Fixed Length Lists	625
70 Non-negative, non-positive integers and reals	627
70.1 Non-positive integers	627
70.2 Non-negative reals	629
70.3 Non-positive reals	630
71 Numeral Syntax for Types	633
71.1 Numeral Types	633
71.2 <i>num1</i>	633
71.3 Locales for modular arithmetic subtypes	635
71.4 Ring class instances	637
71.5 Order instances	638
71.6 Code setup and type classes for code generation	639
71.7 Syntax	641
71.8 Examples	641
72 ω-words	641
72.1 Type declaration and elementary operations	642
72.2 Subsequence, Prefix, and Suffix	643
72.3 Prepending	645
72.4 The limit set of an ω -word	645
72.5 Index sequences and piecewise definitions	648
73 Combinator syntax for generic, open state monads (single-threaded monads)	650
73.1 Motivation	650
73.2 State transformations and combinators	650
73.3 Monad laws	651
73.4 Do-syntax	651
74 Canonical order on option type	652

75 Futures and parallel lists for code generated towards Isabelle/ML	656
75.1 Futures	656
75.2 Parallel lists	657
76 Input syntax for pattern aliases (or “as-patterns” in Haskell)	657
76.1 Definition	658
76.2 Usage	659
77 Periodic Functions	659
78 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view	662
78.1 Preliminary: auxiliary operations for <i>almost everywhere zero</i> .	662
78.2 Type definition	665
78.3 Additive structure	667
78.4 Multiplicative structure	669
78.5 Single-point mappings	670
78.6 Integral domains	672
78.7 Mapping order	672
78.8 Fundamental mapping notions	673
78.9 Degree	675
78.10 Inductive structure	676
78.11 Quasi-functorial structure	676
78.12 Canonical dense representation of $\text{nat} \Rightarrow_0 'a$	677
78.13 Canonical sparse representation of $'a \Rightarrow_0 'b$	679
78.14 Size estimation	680
78.15 Further mapping operations and properties	681
78.16 Free Abelian Groups Over a Type	681
79 Exponentiation by Squaring	685
80 Preorders with explicit equivalence relation	685
81 Additive group operations on product types	687
81.1 Operations	687
81.2 Class instances	688
82 Roots of real quadratics	689
83 Pretty syntax for Quotient operations	691
84 Quotient infrastructure for the set type	691
84.1 Contravariant set map (vimage) and set relator, rules for the Quotient package	691

85 Quotient infrastructure for the product type	693
85.1 Rules for the Quotient package	693
86 Quotient infrastructure for the option type	695
86.1 Rules for the Quotient package	695
87 Quotient infrastructure for the list type	696
87.1 Rules for the Quotient package	696
88 Quotient infrastructure for the sum type	700
88.1 Rules for the Quotient package	700
89 Quotient types	701
89.1 Equivalence relations and quotient types	701
89.2 Equality on quotients	702
89.3 Picking representing elements	703
90 Ramsey's Theorem	703
90.1 Preliminary definitions	703
90.1.1 The n -element subsets of a set A	704
90.1.2 Further properties, involving equipollence	706
90.1.3 Partition predicates	706
90.2 Finite versions of Ramsey's theorem	707
90.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers	707
90.2.2 Trivial cases	708
90.2.3 Ramsey's theorem with TWO colours and arbitrary exponents (hypergraph version)	708
90.2.4 Full Ramsey's theorem with multiple colours and ar- bitrary exponents	709
90.2.5 Simple graph version	709
90.3 Preliminaries for the infinitary version	709
90.3.1 “Axiom” of Dependent Choice	709
90.3.2 Partition functions	710
90.4 Ramsey's Theorem: Infinitary Version	710
90.5 Disjunctive Well-Foundedness	711
91 Modulo and congruence on the reals	711
92 Generic reflection and reification	716
93 Assigning lengths to types by type classes	717

94 Saturated arithmetic	719
94.1 The type of saturated naturals	719
94.2 Enumeration	723
95 Set Idioms	723
95.1 Idioms for being a suitable union/intersection of something .	724
95.2 The “Relative to” operator	728
96 Signed division: negative results rounded towards zero rather than minus infinity.	731
97 State monad	735
98 Comparators on linear quasi-orders	739
98.1 Basic properties	739
98.2 Fundamental comparator combinators	743
98.3 Direct implementations for linear orders on selected types . .	744
99 Stably sorted lists	745
100 Alternative sorting algorithms	748
100.1 Quicksort	748
100.2 Mergesort	749
100.3 Lexicographic products	750
101A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming	751
102 Time functions for various standard library operations. Also defines <i>itrev</i>.	752
103A table-based implementation of the reflexive transitive closure	754
104 Binary Tree	756
104.1 <i>map-tree</i>	758
104.2 <i>size</i>	758
104.3 <i>set-tree</i>	759
104.4 <i>subtrees</i>	759
104.5 <i>height</i> and <i>min-height</i>	759
104.6 <i>complete</i>	760
104.7 <i>acomplete</i>	760
104.8 <i>wbalanced</i>	761
104.9 <i>ipl</i>	761

104.10	List of entries	761
104.11	Binary Search Tree	762
104.12	Heap	762
104.13	Mirror	762
105	Multiset of Elements of Binary Tree	763
106	Unordered pairs	765
107A	type of finite bit strings	768
107.1	Preliminaries	769
107.2	Fundamentals	769
107.2.1	Type definition	769
107.2.2	Basic arithmetic	769
107.2.3	Basic tool setup	770
107.2.4	Basic code generation setup	771
107.2.5	Basic conversions	772
107.3	Elementary case distinctions	778
107.3.1	Basic ordering	778
107.4	Enumeration	781
107.5	Bit-wise operations	781
107.6	Conversions including casts	786
107.6.1	Generic unsigned conversion	786
107.6.2	Generic signed conversion	788
107.6.3	More	789
107.7	Arithmetic operations	794
107.8	Ordering	796
107.9	Bit-wise operations	798
107.10	More shift operations	800
107.11	Single-bit operations	800
107.12	Rotation	801
107.13	Split and cat operations	802
107.14	More on conversions	803
107.15	Testing bits	806
107.16	Word Arithmetic	809
107.17	Transferring goals from words to ints	813
107.18	Order on fixed-length words	815
107.19	Conditions for the addition (etc) of two words to overflow	816
107.20	Some proof tool support	818
107.21	More on overflows and monotonicity	819
107.22	Arithmetic type class instantiations	823
107.23	Word and nat	823
107.24	Cardinality, finiteness of set of words	827
107.25	Bitwise Operations on Words	827

107.25.1	Shift functions in terms of lists of bools	832
107.25.2	Mask	834
107.25.3	Slices	836
107.25.4	Revcast	837
107.26	Split and cat	838
107.26.1	Split and slice	838
107.27	Rotation	839
107.27.1	Word rotation commutes with bit-wise operations	840
107.28	Maximum machine word	841
107.29	Recursion combinator for words	844
107.30	Some more naive computations rules	845
107.31	Executable intervals	846
107.32	Tool support	846
108	The Field of Integers mod 2	847
109	Pointwise order on product types	851
109.1	Pointwise ordering	851
109.2	Binary infimum and supremum	852
109.3	Top and bottom elements	853
109.4	Complete lattice operations	854
109.5	Complete distributive lattices	855
109.6	Bekic's Theorem	855
110	Finite Lattices	856
110.1	Finite Complete Lattices	856
110.2	Finite Distributive Lattices	858
110.3	Linear Orders	859
110.4	Finite Linear Orders	860
111	Lexicographic order on lists	860
112	Lexicographic order on lists	862
113	Prefix order on lists as order class instance	863
114	Lexicographic order on product types	864
115	Subsequence Ordering	866
115.1	Definitions and basic lemmas	866
116	Records based on BNF/datatype machinery	868
117	Implementation of mappings with Association Lists	869

118	Avoidance of pattern matching on natural numbers	874
118.1	Case analysis	874
118.2	Preprocessors	874
118.3	Candidates which need special treatment	875
119	Implementation of natural numbers as binary numerals	875
119.1	Representation	875
119.2	Basic arithmetic	876
119.3	Conversions	877
120	Code generation of prolog programs	878
121	Setup for Numerals	878
122	Implementation of integer numbers by target-language integers	878
123	Implementation of natural numbers by target-language integers	886
123.1	Implementation for <i>nat</i>	886
124	Implementation of natural and integer numbers by target-language integers	891
125	Preprocessor setup for floats implemented by target language numerals	891
126	Abstract type of association lists with unique keys	892
126.1	Preliminaries	892
126.2	Type (<i>'key, 'value</i>) <i>alist</i>	892
126.3	Primitive operations	893
126.4	Abstract operation properties	893
126.5	Further operations	894
126.5.1	Equality	894
126.5.2	Size	894
126.6	Quickcheck generators	894
127	alist is a BNF	896
128	Multisets partially implemented by association lists	896
129	Implementation of Red-Black Trees	901
129.1	Datatype of RB trees	901
129.2	Tree properties	902
129.2.1	Content of a tree	902
129.2.2	Search tree properties	902

129.2.3 Tree lookup	903
129.2.4 Red-black properties	905
129.3 Insertion	906
129.4 Deletion	909
129.5 Modifying existing entries	915
129.6 Mapping all entries	915
129.7 Folding over entries	916
129.8 Bulkloading a tree	916
129.9 Building a RBT from a sorted list	917
129.10 Union and intersection of sorted associative lists	923
129.1 Code generator setup	940
130 Abstract type of RBT trees	941
130.1 Type definition	942
130.2 Primitive operations	942
130.3 Derived operations	943
130.4 Abstract lookup properties	943
130.5 Quickcheck generators	946
130.6 Hide implementation details	946
131 Implementation of mappings with Red-Black Trees	947
131.1 Data type and invariant	947
131.2 Operations	947
131.3 Invariant preservation	948
131.4 Map Semantics	948
132 Implementation of sets using RBT trees	949
133 Definition of code datatype constructors	949
134 Lemmas	949
134.1 Auxiliary lemmas	949
134.2 fold and filter	949
134.3 foldi and Ball	950
134.4 foldi and Bex	950
134.5 folding over non empty trees and selecting the minimal and maximal element	950
134.5.1 concrete	950
134.5.2 abstract	953
135 Code equations	954
136 Introduction	960
137 Termination	961

138	Partial Functions	961
139	Higher-Order Functions	961
139.1	Limitations	962
140	Predefined Functions	963
141	Locales	963
142	Fine Points	964
143	Common constants	966
144	Pairs	966
145	Filters	966
146	Bounded quantifiers	966
147	Operations on Predicates	966
148	Setup for Numerals	967
149	Arithmetic operations	967
149.1	Arithmetic on naturals and integers	967
149.2	Inductive definitions for ordering on naturals	967
150	Alternative list definitions	968
150.1	Alternative rules for <i>length</i>	968
150.2	Alternative rules for <i>list-all2</i>	968
150.3	Alternative rules for membership in lists	968
151	Setup for String.literal	969
152	Simplification rules for optimisation	969
153	A Prototype of Quickcheck based on the Predicate Com- piler	969
154	TFL: recursive function definitions	969
154.1	Lemmas for TFL	969
154.2	Rule setup	970
155	Program extraction from proofs involving datatypes and in- ductive predicates	971
156	Refute	971

1 Implementation of Association Lists

```
theory AList
  imports Main
begin
```

```
context
begin
```

The operations preserve distinctness of keys and function *clearjunk* distributes over them. Since *clearjunk* enforces distinctness of keys it can be used to establish the invariant, e.g. for inductive proofs.

1.1 *update* and *updates*

```
qualified primrec update :: 'key  $\Rightarrow$  'val  $\Rightarrow$  ('key  $\times$  'val) list  $\Rightarrow$  ('key  $\times$  'val) list
  where
    update k v [] = [(k, v)]
    | update k v (p # ps) = (if fst p = k then (k, v) # ps else p # update k v ps)
```

```
lemma update-conv': map-of (update k v al) = (map-of al)( $k \mapsto v$ )
  <proof>
```

```
corollary update-conv: map-of (update k v al) k' = ((map-of al)( $k \mapsto v$ )) k'
  <proof>
```

```
lemma dom-update: fst ` set (update k v al) = {k}  $\cup$  fst ` set al
  <proof>
```

```
lemma update-keys:
  map fst (update k v al) =
    (if k  $\in$  set (map fst al) then map fst al else map fst al @ [k])
  <proof>
```

```
lemma distinct-update:
  assumes distinct (map fst al)
  shows distinct (map fst (update k v al))
  <proof>
```

```
lemma update-filter:
  a  $\neq$  k  $\implies$  update k v [q  $\leftarrow$  ps. fst q  $\neq$  a] = [q  $\leftarrow$  update k v ps. fst q  $\neq$  a]
  <proof>
```

```
lemma update-triv: map-of al k = Some v  $\implies$  update k v al = al
  <proof>
```

```
lemma update-nonempty [simp]: update k v al  $\neq$  []
  <proof>
```

lemma *update-eqD*: $\text{update } k \ v \ al = \text{update } k \ v' \ al' \implies v = v'$
 ⟨proof⟩

lemma *update-last [simp]*: $\text{update } k \ v \ (\text{update } k \ v' \ al) = \text{update } k \ v \ al$
 ⟨proof⟩

Note that the lists are not necessarily the same: $\text{update } k \ v \ (\text{update } k' \ v' \ []) = [(k', v'), (k, v)]$ and $\text{update } k' \ v' \ (\text{update } k \ v \ []) = [(k, v), (k', v')]$.

lemma *update-swap*:
 $k \neq k' \implies \text{map-of } (\text{update } k \ v \ (\text{update } k' \ v' \ al)) = \text{map-of } (\text{update } k' \ v' \ (\text{update } k \ v \ al))$
 ⟨proof⟩

lemma *update-Some-unfold*:
 $\text{map-of } (\text{update } k \ v \ al) \ x = \text{Some } y \iff$
 $x = k \wedge v = y \vee x \neq k \wedge \text{map-of } al \ x = \text{Some } y$
 ⟨proof⟩

lemma *image-update [simp]*: $x \notin A \implies \text{map-of } (\text{update } x \ y \ al) \ `A = \text{map-of } al \ `A$
 ⟨proof⟩ **definition** *updates* ::
 $'key \ list \Rightarrow 'val \ list \Rightarrow ('key \times 'val) \ list \Rightarrow ('key \times 'val) \ list$
where $\text{updates } ks \ vs = \text{fold } (\text{case-prod } \text{update}) \ (\text{zip } ks \ vs)$

lemma *updates-simps [simp]*:
 $\text{updates } [] \ vs \ ps = ps$
 $\text{updates } ks \ [] \ ps = ps$
 $\text{updates } (k \# ks) \ (v \# vs) \ ps = \text{updates } ks \ vs \ (\text{update } k \ v \ ps)$
 ⟨proof⟩

lemma *updates-key-simp [simp]*:
 $\text{updates } (k \ \# \ ks) \ vs \ ps =$
 $(\text{case } vs \ \text{of } [] \Rightarrow ps \mid v \ \# \ vs \Rightarrow \text{updates } ks \ vs \ (\text{update } k \ v \ ps))$
 ⟨proof⟩

lemma *updates-conv'*: $\text{map-of } (\text{updates } ks \ vs \ al) = (\text{map-of } al)(ks[\mapsto]vs)$
 ⟨proof⟩

lemma *updates-conv*: $\text{map-of } (\text{updates } ks \ vs \ al) \ k = ((\text{map-of } al)(ks[\mapsto]vs)) \ k$
 ⟨proof⟩

lemma *distinct-updates*:
assumes $\text{distinct } (\text{map } \text{fst } al)$
shows $\text{distinct } (\text{map } \text{fst } (\text{updates } ks \ vs \ al))$
 ⟨proof⟩

lemma *updates-append1 [simp]*: $\text{size } ks < \text{size } vs \implies$
 $\text{updates } (ks@[k]) \ vs \ al = \text{update } k \ (vs[\text{size } ks]) \ (\text{updates } ks \ vs \ al)$
 ⟨proof⟩

lemma *updates-list-update-drop* [simp]:
 $\text{size } ks \leq i \implies i < \text{size } vs \implies$
 $\text{updates } ks \text{ (vs[i:=v]) } al = \text{updates } ks \text{ vs } al$
 ⟨proof⟩

lemma *update-updates-conv-if*:
 $\text{map-of } (\text{updates } xs \text{ ys } (\text{update } x \text{ y } al)) =$
 map-of
 $(\text{if } x \in \text{set } (\text{take } (\text{length } ys) \text{ xs})$
 $\text{then } \text{updates } xs \text{ ys } al$
 $\text{else } (\text{update } x \text{ y } (\text{updates } xs \text{ ys } al)))$
 ⟨proof⟩

lemma *updates-twist* [simp]:
 $k \notin \text{set } ks \implies$
 $\text{map-of } (\text{updates } ks \text{ vs } (\text{update } k \text{ v } al)) = \text{map-of } (\text{update } k \text{ v } (\text{updates } ks \text{ vs } al))$
 ⟨proof⟩

lemma *updates-apply-notin* [simp]:
 $k \notin \text{set } ks \implies \text{map-of } (\text{updates } ks \text{ vs } al) \text{ k} = \text{map-of } al \text{ k}$
 ⟨proof⟩

lemma *updates-append-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies \text{updates } (xs @ zs) \text{ ys } al = \text{updates } xs \text{ ys } al$
 ⟨proof⟩

lemma *updates-append2-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies \text{updates } xs \text{ (ys @ zs) } al = \text{updates } xs \text{ ys } al$
 ⟨proof⟩

1.2 delete

qualified definition *delete* :: 'key \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list
where *delete-eq*: *delete* *k* = *filter* ($\lambda(k', -). k \neq k'$)

lemma *delete-simps* [simp]:
 $\text{delete } k \text{ []} = \text{[]}$
 $\text{delete } k \text{ (p \# ps)} = (\text{if } \text{fst } p = k \text{ then } \text{delete } k \text{ ps else } p \# \text{delete } k \text{ ps})$
 ⟨proof⟩

lemma *delete-conv'*: $\text{map-of } (\text{delete } k \text{ al}) = (\text{map-of } al)(k := \text{None})$
 ⟨proof⟩

corollary *delete-conv*: $\text{map-of } (\text{delete } k \text{ al}) \text{ k}' = ((\text{map-of } al)(k := \text{None})) \text{ k}'$
 ⟨proof⟩

lemma *delete-keys*: $\text{map } \text{fst } (\text{delete } k \text{ al}) = \text{removeAll } k \text{ (map } \text{fst } al)$
 ⟨proof⟩

lemma *distinct-delete*:

assumes *distinct* (*map fst al*)
shows *distinct* (*map fst (delete k al)*)
 ⟨*proof*⟩

lemma *delete-id* [*simp*]: $k \notin \text{fst} \text{ ` set } al \implies \text{delete } k \text{ al} = al$
 ⟨*proof*⟩

lemma *delete-idem*: $\text{delete } k (\text{delete } k \text{ al}) = \text{delete } k \text{ al}$
 ⟨*proof*⟩

lemma *map-of-delete* [*simp*]: $k' \neq k \implies \text{map-of } (\text{delete } k \text{ al}) \text{ } k' = \text{map-of } al \text{ } k'$
 ⟨*proof*⟩

lemma *delete-notin-dom*: $k \notin \text{fst} \text{ ` set } (\text{delete } k \text{ al})$
 ⟨*proof*⟩

lemma *dom-delete-subset*: $\text{fst} \text{ ` set } (\text{delete } k \text{ al}) \subseteq \text{fst} \text{ ` set } al$
 ⟨*proof*⟩

lemma *delete-update-same*: $\text{delete } k (\text{update } k \text{ } v \text{ al}) = \text{delete } k \text{ al}$
 ⟨*proof*⟩

lemma *delete-update*: $k \neq l \implies \text{delete } l (\text{update } k \text{ } v \text{ al}) = \text{update } k \text{ } v (\text{delete } l \text{ al})$
 ⟨*proof*⟩

lemma *delete-twist*: $\text{delete } x (\text{delete } y \text{ al}) = \text{delete } y (\text{delete } x \text{ al})$
 ⟨*proof*⟩

lemma *length-delete-le*: $\text{length } (\text{delete } k \text{ al}) \leq \text{length } al$
 ⟨*proof*⟩

1.3 *update-with-aux* and *delete-aux*

qualified primrec *update-with-aux* ::

$'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

where

$\text{update-with-aux } v \text{ } k \text{ } f \text{ } [] = [(k, f v)]$

$| \text{update-with-aux } v \text{ } k \text{ } f \text{ } (p \# ps) =$
 $(\text{if } (\text{fst } p = k) \text{ then } (k, f (\text{snd } p)) \# ps \text{ else } p \# \text{update-with-aux } v \text{ } k \text{ } f \text{ } ps)$

The above *delete* traverses all the list even if it has found the key. This one does not have to keep going because it assumes the invariant that keys are distinct.

qualified fun *delete-aux* :: $'key \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

where

$\text{delete-aux } k \text{ } [] = []$

$| \text{delete-aux } k \text{ } ((k', v) \# xs) = (\text{if } k = k' \text{ then } xs \text{ else } (k', v) \# \text{delete-aux } k \text{ } xs)$

lemma *map-of-update-with-aux'*:

$$\begin{aligned} & \text{map-of } (\text{update-with-aux } v \ k \ f \ ps) \ k' = \\ & ((\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \text{ of } None \Rightarrow f \ v \mid Some \ v \Rightarrow f \ v))) \ k' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *map-of-update-with-aux*:

$$\begin{aligned} & \text{map-of } (\text{update-with-aux } v \ k \ f \ ps) = \\ & (\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \text{ of } None \Rightarrow f \ v \mid Some \ v \Rightarrow f \ v)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dom-update-with-aux*: $\text{fst} \text{ ' set } (\text{update-with-aux } v \ k \ f \ ps) = \{k\} \cup \text{fst} \text{ ' set } ps$

$\langle \text{proof} \rangle$

lemma *distinct-update-with-aux [simp]*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } (\text{update-with-aux } v \ k \ f \ ps)) = \text{distinct } (\text{map } \text{fst } ps) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *set-update-with-aux*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \\ & \text{set } (\text{update-with-aux } v \ k \ f \ xs) = \\ & (\text{set } xs - \{k\} \times UNIV \cup \{(k, f \ (\text{case map-of } xs \ k \text{ of } None \Rightarrow v \mid Some \ v \Rightarrow v))\}) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *set-delete-aux*: $\text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{set } (\text{delete-aux } k \ xs) = \text{set } xs - \{k\} \times UNIV$

$\langle \text{proof} \rangle$

lemma *dom-delete-aux*: $\text{distinct } (\text{map } \text{fst } ps) \Longrightarrow \text{fst} \text{ ' set } (\text{delete-aux } k \ ps) = \text{fst} \text{ ' set } ps - \{k\}$

$\langle \text{proof} \rangle$

lemma *distinct-delete-aux [simp]*: $\text{distinct } (\text{map } \text{fst } ps) \Longrightarrow \text{distinct } (\text{map } \text{fst } (\text{delete-aux } k \ ps))$

$\langle \text{proof} \rangle$

lemma *map-of-delete-aux'*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{map-of } (\text{delete-aux } k \ xs) = (\text{map-of } xs)(k := None) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *map-of-delete-aux*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{map-of } (\text{delete-aux } k \ xs) \ k' = ((\text{map-of } xs)(k := None)) \\ & k' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *delete-aux-eq-Nil-conv*: $\text{delete-aux } k \ ts = [] \longleftrightarrow ts = [] \vee (\exists v. ts = [(k, v)])$

$\langle \text{proof} \rangle$

1.4 restrict

qualified definition $\text{restrict} :: 'key \text{ set} \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$
where restrict-eq : $\text{restrict } A = \text{filter } (\lambda(k, v). k \in A)$

lemma restr-simps [simp]:

$\text{restrict } A [] = []$

$\text{restrict } A (p \# ps) = (\text{if } \text{fst } p \in A \text{ then } p \# \text{restrict } A \text{ ps else } \text{restrict } A \text{ ps})$

$\langle \text{proof} \rangle$

lemma restr-conv' : $\text{map-of } (\text{restrict } A \text{ al}) = ((\text{map-of } \text{al})|' A)$

$\langle \text{proof} \rangle$

corollary restr-conv : $\text{map-of } (\text{restrict } A \text{ al}) \text{ } k = ((\text{map-of } \text{al})|' A) \text{ } k$

$\langle \text{proof} \rangle$

lemma distinct-restr : $\text{distinct } (\text{map } \text{fst } \text{al}) \implies \text{distinct } (\text{map } \text{fst } (\text{restrict } A \text{ al}))$

$\langle \text{proof} \rangle$

lemma restr-empty [simp]:

$\text{restrict } \{\} \text{ al} = []$

$\text{restrict } A [] = []$

$\langle \text{proof} \rangle$

lemma restr-in [simp]: $x \in A \implies \text{map-of } (\text{restrict } A \text{ al}) \text{ } x = \text{map-of } \text{al } x$

$\langle \text{proof} \rangle$

lemma restr-out [simp]: $x \notin A \implies \text{map-of } (\text{restrict } A \text{ al}) \text{ } x = \text{None}$

$\langle \text{proof} \rangle$

lemma dom-restr [simp]: $\text{fst } ' \text{ set } (\text{restrict } A \text{ al}) = \text{fst } ' \text{ set } \text{al} \cap A$

$\langle \text{proof} \rangle$

lemma restr-upd-same [simp]: $\text{restrict } (-\{x\}) (\text{update } x \text{ } y \text{ al}) = \text{restrict } (-\{x\}) \text{ al}$

$\langle \text{proof} \rangle$

lemma restr-restr [simp]: $\text{restrict } A (\text{restrict } B \text{ al}) = \text{restrict } (A \cap B) \text{ al}$

$\langle \text{proof} \rangle$

lemma restr-update [simp]:

$\text{map-of } (\text{restrict } D (\text{update } x \text{ } y \text{ al})) =$

$\text{map-of } ((\text{if } x \in D \text{ then } (\text{update } x \text{ } y (\text{restrict } (D - \{x\}) \text{ al})) \text{ else } \text{restrict } D \text{ al}))$

$\langle \text{proof} \rangle$

lemma restr-delete [simp]:

$\text{delete } x (\text{restrict } D \text{ al}) = (\text{if } x \in D \text{ then } \text{restrict } (D - \{x\}) \text{ al else } \text{restrict } D \text{ al})$

$\langle \text{proof} \rangle$

lemma *update-restr*:

$\text{map-of } (\text{update } x \ y \ (\text{restrict } D \ al)) = \text{map-of } (\text{update } x \ y \ (\text{restrict } (D - \{x\}) \ al))$
 $\langle \text{proof} \rangle$

lemma *update-restr-conv* [simp]:

$x \in D \implies$
 $\text{map-of } (\text{update } x \ y \ (\text{restrict } D \ al)) = \text{map-of } (\text{update } x \ y \ (\text{restrict } (D - \{x\}) \ al))$
 $\langle \text{proof} \rangle$

lemma *restr-updates* [simp]:

$\text{length } xs = \text{length } ys \implies \text{set } xs \subseteq D \implies$
 $\text{map-of } (\text{restrict } D \ (\text{updates } xs \ ys \ al)) =$
 $\text{map-of } (\text{updates } xs \ ys \ (\text{restrict } (D - \text{set } xs) \ al))$
 $\langle \text{proof} \rangle$

lemma *restr-delete-twist*: $(\text{restrict } A \ (\text{delete } a \ ps)) = \text{delete } a \ (\text{restrict } A \ ps)$

$\langle \text{proof} \rangle$

1.5 clearjunk

qualified function *clearjunk* :: $('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

where

$\text{clearjunk } [] = []$
 $| \text{clearjunk } (p \# ps) = p \# \text{clearjunk } (\text{delete } (\text{fst } p) \ ps)$
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

lemma *map-of-clearjunk*: $\text{map-of } (\text{clearjunk } al) = \text{map-of } al$

$\langle \text{proof} \rangle$

lemma *clearjunk-keys-set*: $\text{set } (\text{map } \text{fst } (\text{clearjunk } al)) = \text{set } (\text{map } \text{fst } al)$

$\langle \text{proof} \rangle$

lemma *dom-clearjunk*: $\text{fst } ' \text{set } (\text{clearjunk } al) = \text{fst } ' \text{set } al$

$\langle \text{proof} \rangle$

lemma *distinct-clearjunk* [simp]: $\text{distinct } (\text{map } \text{fst } (\text{clearjunk } al))$

$\langle \text{proof} \rangle$

lemma *ran-clearjunk*: $\text{ran } (\text{map-of } (\text{clearjunk } al)) = \text{ran } (\text{map-of } al)$

$\langle \text{proof} \rangle$

lemma *ran-map-of*: $\text{ran } (\text{map-of } al) = \text{snd } ' \text{set } (\text{clearjunk } al)$

$\langle \text{proof} \rangle$

lemma *graph-map-of*: $\text{Map.graph } (\text{map-of } al) = \text{set } (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-update*: $\text{clearjunk } (\text{update } k \ v \ al) = \text{update } k \ v \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-updates*: $\text{clearjunk } (\text{updates } ks \ vs \ al) = \text{updates } ks \ vs \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-delete*: $\text{clearjunk } (\text{delete } x \ al) = \text{delete } x \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-restrict*: $\text{clearjunk } (\text{restrict } A \ al) = \text{restrict } A \ (\text{clearjunk } al)$
 $\langle \text{proof} \rangle$

lemma *distinct-clearjunk-id* [simp]: $\text{distinct } (\text{map } \text{fst } al) \implies \text{clearjunk } al = al$
 $\langle \text{proof} \rangle$

lemma *clearjunk-idem*: $\text{clearjunk } (\text{clearjunk } al) = \text{clearjunk } al$
 $\langle \text{proof} \rangle$

lemma *length-clearjunk*: $\text{length } (\text{clearjunk } al) \leq \text{length } al$
 $\langle \text{proof} \rangle$

lemma *delete-map*:
assumes $\bigwedge kv. \text{fst } (f \ kv) = \text{fst } kv$
shows $\text{delete } k \ (\text{map } f \ ps) = \text{map } f \ (\text{delete } k \ ps)$
 $\langle \text{proof} \rangle$

lemma *clearjunk-map*:
assumes $\bigwedge kv. \text{fst } (f \ kv) = \text{fst } kv$
shows $\text{clearjunk } (\text{map } f \ ps) = \text{map } f \ (\text{clearjunk } ps)$
 $\langle \text{proof} \rangle$

1.6 map-ran

definition *map-ran* :: $('key \Rightarrow 'val1 \Rightarrow 'val2) \Rightarrow ('key \times 'val1) \text{ list} \Rightarrow ('key \times 'val2) \text{ list}$
where $\text{map-ran } f = \text{map } (\lambda(k, v). (k, f \ k \ v))$

lemma *map-ran-simps* [simp]:
 $\text{map-ran } f \ [] = []$
 $\text{map-ran } f \ ((k, v) \# ps) = (k, f \ k \ v) \# \text{map-ran } f \ ps$
 $\langle \text{proof} \rangle$

lemma *map-ran-Cons-sel*: $\text{map-ran } f \ (p \# ps) = (\text{fst } p, f \ (\text{fst } p) \ (\text{snd } p)) \# \text{map-ran } f \ ps$
 $\langle \text{proof} \rangle$

lemma *length-map-ran[simp]*: $\text{length } (\text{map-ran } f \text{ al}) = \text{length } \text{al}$
 $\langle \text{proof} \rangle$

lemma *map-fst-map-ran[simp]*: $\text{map } \text{fst } (\text{map-ran } f \text{ al}) = \text{map } \text{fst } \text{al}$
 $\langle \text{proof} \rangle$

lemma *dom-map-ran*: $\text{fst } ' \text{ set } (\text{map-ran } f \text{ al}) = \text{fst } ' \text{ set } \text{al}$
 $\langle \text{proof} \rangle$

lemma *map-ran-conv*: $\text{map-of } (\text{map-ran } f \text{ al}) \text{ k} = \text{map-option } (f \text{ k}) (\text{map-of } \text{al } \text{k})$
 $\langle \text{proof} \rangle$

lemma *distinct-map-ran*: $\text{distinct } (\text{map } \text{fst } \text{al}) \implies \text{distinct } (\text{map } \text{fst } (\text{map-ran } f \text{ al}))$
 $\langle \text{proof} \rangle$

lemma *map-ran-filter*: $\text{map-ran } f \text{ } [p \leftarrow \text{ps}. \text{fst } p \neq a] = [p \leftarrow \text{map-ran } f \text{ ps}. \text{fst } p \neq a]$
 $\langle \text{proof} \rangle$

lemma *clearjunk-map-ran*: $\text{clearjunk } (\text{map-ran } f \text{ al}) = \text{map-ran } f (\text{clearjunk } \text{al})$
 $\langle \text{proof} \rangle$

1.7 merge

qualified definition *merge* :: $('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$
where $\text{merge } \text{qs } \text{ps} = \text{foldr } (\lambda(k, v). \text{update } k \text{ v}) \text{ ps } \text{qs}$

lemma *merge-simps [simp]*:
 $\text{merge } \text{qs } [] = \text{qs}$
 $\text{merge } \text{qs } (p \# \text{ps}) = \text{update } (\text{fst } p) (\text{snd } p) (\text{merge } \text{qs } \text{ps})$
 $\langle \text{proof} \rangle$

lemma *merge-updates*: $\text{merge } \text{qs } \text{ps} = \text{updates } (\text{rev } (\text{map } \text{fst } \text{ps})) (\text{rev } (\text{map } \text{snd } \text{ps})) \text{ qs}$
 $\langle \text{proof} \rangle$

lemma *dom-merge*: $\text{fst } ' \text{ set } (\text{merge } \text{xs } \text{ys}) = \text{fst } ' \text{ set } \text{xs} \cup \text{fst } ' \text{ set } \text{ys}$
 $\langle \text{proof} \rangle$

lemma *distinct-merge*: $\text{distinct } (\text{map } \text{fst } \text{xs}) \implies \text{distinct } (\text{map } \text{fst } (\text{merge } \text{xs } \text{ys}))$
 $\langle \text{proof} \rangle$

lemma *clearjunk-merge*: $\text{clearjunk } (\text{merge } \text{xs } \text{ys}) = \text{merge } (\text{clearjunk } \text{xs}) \text{ ys}$
 $\langle \text{proof} \rangle$

lemma *merge-conv'*: $\text{map-of } (\text{merge } \text{xs } \text{ys}) = \text{map-of } \text{xs} ++ \text{map-of } \text{ys}$

$\langle \text{proof} \rangle$

corollary *merge-conv*: $\text{map-of } (\text{merge } xs \ ys) \ k = (\text{map-of } xs \ ++ \ \text{map-of } ys) \ k$
 $\langle \text{proof} \rangle$

lemma *merge-empty*: $\text{map-of } (\text{merge } [] \ ys) = \text{map-of } ys$
 $\langle \text{proof} \rangle$

lemma *merge-assoc* [simp]: $\text{map-of } (\text{merge } m1 \ (\text{merge } m2 \ m3)) = \text{map-of } (\text{merge } (\text{merge } m1 \ m2) \ m3)$
 $\langle \text{proof} \rangle$

lemma *merge-Some-iff*:
 $\text{map-of } (\text{merge } m \ n) \ k = \text{Some } x \longleftrightarrow$
 $\text{map-of } n \ k = \text{Some } x \vee \text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{Some } x$
 $\langle \text{proof} \rangle$

lemmas *merge-SomeD* [dest!] = *merge-Some-iff* [THEN iffD1]

lemma *merge-find-right* [simp]: $\text{map-of } n \ k = \text{Some } v \implies \text{map-of } (\text{merge } m \ n) \ k = \text{Some } v$
 $\langle \text{proof} \rangle$

lemma *merge-None* [iff]: $(\text{map-of } (\text{merge } m \ n) \ k = \text{None}) = (\text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{None})$
 $\langle \text{proof} \rangle$

lemma *merge-upd* [simp]: $\text{map-of } (\text{merge } m \ (\text{update } k \ v \ n)) = \text{map-of } (\text{update } k \ v \ (\text{merge } m \ n))$
 $\langle \text{proof} \rangle$

lemma *merge-updatess* [simp]:
 $\text{map-of } (\text{merge } m \ (\text{updates } xs \ ys \ n)) = \text{map-of } (\text{updates } xs \ ys \ (\text{merge } m \ n))$
 $\langle \text{proof} \rangle$

lemma *merge-append*: $\text{map-of } (xs \ @ \ ys) = \text{map-of } (\text{merge } ys \ xs)$
 $\langle \text{proof} \rangle$

1.8 compose

qualified function *compose* :: $('key \times 'a) \ \text{list} \Rightarrow ('a \times 'b) \ \text{list} \Rightarrow ('key \times 'b) \ \text{list}$
where

$\text{compose } [] \ ys = []$
 $| \text{compose } (x \ \# \ xs) \ ys =$
 $\quad (\text{case } \text{map-of } ys \ (\text{snd } x) \ \text{of}$
 $\quad \quad \text{None} \Rightarrow \text{compose } (\text{delete } (\text{fst } x) \ xs) \ ys$
 $\quad \quad | \text{Some } v \Rightarrow (\text{fst } x, v) \ \# \ \text{compose } xs \ ys)$
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

lemma *compose-first-None* [simp]: $\text{map-of } xs \ k = \text{None} \implies \text{map-of } (\text{compose } xs \ ys) \ k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *compose-conv*: $\text{map-of } (\text{compose } xs \ ys) \ k = (\text{map-of } ys \circ_m \text{map-of } xs) \ k$
 $\langle \text{proof} \rangle$

lemma *compose-conv'*: $\text{map-of } (\text{compose } xs \ ys) = (\text{map-of } ys \circ_m \text{map-of } xs)$
 $\langle \text{proof} \rangle$

lemma *compose-first-Some* [simp]: $\text{map-of } xs \ k = \text{Some } v \implies \text{map-of } (\text{compose } xs \ ys) \ k = \text{map-of } ys \ v$
 $\langle \text{proof} \rangle$

lemma *dom-compose*: $\text{fst } \text{'set } (\text{compose } xs \ ys) \subseteq \text{fst } \text{'set } xs$
 $\langle \text{proof} \rangle$

lemma *distinct-compose*:
assumes *distinct* (map fst xs)
shows *distinct* (map fst (compose xs ys))
 $\langle \text{proof} \rangle$

lemma *compose-delete-twist*: $\text{compose } (\text{delete } k \ xs) \ ys = \text{delete } k \ (\text{compose } xs \ ys)$
 $\langle \text{proof} \rangle$

lemma *compose-clearjunk*: $\text{compose } xs \ (\text{clearjunk } ys) = \text{compose } xs \ ys$
 $\langle \text{proof} \rangle$

lemma *clearjunk-compose*: $\text{clearjunk } (\text{compose } xs \ ys) = \text{compose } (\text{clearjunk } xs) \ ys$
 $\langle \text{proof} \rangle$

lemma *compose-empty* [simp]: $\text{compose } xs \ [] = []$
 $\langle \text{proof} \rangle$

lemma *compose-Some-iff*:
 $(\text{map-of } (\text{compose } xs \ ys) \ k = \text{Some } v) \longleftrightarrow$
 $(\exists k'. \text{map-of } xs \ k = \text{Some } k' \wedge \text{map-of } ys \ k' = \text{Some } v)$
 $\langle \text{proof} \rangle$

lemma *map-comp-None-iff*:
 $\text{map-of } (\text{compose } xs \ ys) \ k = \text{None} \longleftrightarrow$
 $(\text{map-of } xs \ k = \text{None} \vee (\exists k'. \text{map-of } xs \ k = \text{Some } k' \wedge \text{map-of } ys \ k' = \text{None}))$
 $\langle \text{proof} \rangle$

1.9 *map-entry*

qualified fun *map-entry* :: 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list

where

map-entry *k f* [] = []
 | *map-entry* *k f* (*p* # *ps*) =
 (if *fst* *p* = *k* then (*k*, *f* (*snd* *p*)) # *ps* else *p* # *map-entry* *k f* *ps*)

lemma *map-of-map-entry*:

map-of (*map-entry* *k f* *xs*) =
 (*map-of* *xs*)(*k* := case *map-of* *xs* *k* of None \Rightarrow None | Some *v'* \Rightarrow Some (*f* *v'*))
 <proof>

lemma *dom-map-entry*: *fst* ' set (*map-entry* *k f* *xs*) = *fst* ' set *xs*

<proof>

lemma *distinct-map-entry*:

assumes *distinct* (*map* *fst* *xs*)
shows *distinct* (*map* *fst* (*map-entry* *k f* *xs*))
 <proof>

1.10 *map-default*

fun *map-default* :: 'key \Rightarrow 'val \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list

where

map-default *k v f* [] = [(*k*, *v*)]
 | *map-default* *k v f* (*p* # *ps*) =
 (if *fst* *p* = *k* then (*k*, *f* (*snd* *p*)) # *ps* else *p* # *map-default* *k v f* *ps*)

lemma *map-of-map-default*:

map-of (*map-default* *k v f* *xs*) =
 (*map-of* *xs*)(*k* := case *map-of* *xs* *k* of None \Rightarrow Some *v* | Some *v'* \Rightarrow Some (*f* *v'*))
 <proof>

lemma *dom-map-default*: *fst* ' set (*map-default* *k v f* *xs*) = insert *k* (*fst* ' set *xs*)

<proof>

lemma *distinct-map-default*:

assumes *distinct* (*map* *fst* *xs*)
shows *distinct* (*map* *fst* (*map-default* *k v f* *xs*))
 <proof>

end

end

2 Axiomatic Declaration of Bounded Natural Functors

```

theory BNF-Axiomatization
imports Main
keywords
  bnf-axiomatization :: thy-decl
begin

⟨ML⟩

end

```

3 Generalized Corecursor Sugar (corec and friends)

```

theory BNF-Corec
imports Main
keywords
  corec :: thy-defn and
  corecursive :: thy-goal-defn and
  friend-of-corec :: thy-goal-defn and
  coinduction-upto :: thy-decl
begin

lemma obj-distinct-prems:  $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$ 
  ⟨proof⟩

lemma inject-refine:  $g (f x) = x \Longrightarrow g (f y) = y \Longrightarrow f x = f y \longleftrightarrow x = y$ 
  ⟨proof⟩

lemma convol-apply:  $\text{BNF-Def.convol } f g x = (f x, g x)$ 
  ⟨proof⟩

lemma Grp-UNIV-id:  $\text{BNF-Def.Grp UNIV id} = (=)$ 
  ⟨proof⟩

lemma sum-comp-cases:
  assumes  $f \circ \text{Inl} = g \circ \text{Inl}$  and  $f \circ \text{Inr} = g \circ \text{Inr}$ 
  shows  $f = g$ 
  ⟨proof⟩

lemma case-sum-Inl-Inr-L:  $\text{case-sum } (f \circ \text{Inl}) (f \circ \text{Inr}) = f$ 
  ⟨proof⟩

lemma eq-o-InrI:  $\llbracket g \circ \text{Inl} = h; \text{case-sum } h f = g \rrbracket \Longrightarrow f = g \circ \text{Inr}$ 
  ⟨proof⟩

lemma id-bnf-o:  $\text{BNF-Composition.id-bnf} \circ f = f$ 

```

$\langle \text{proof} \rangle$

lemma *o-id-bnf*: $f \circ \text{BNF-Composition.id-bnf} = f$
 $\langle \text{proof} \rangle$

lemma *if-True-False*:

$(\text{if } P \text{ then True else } Q) \longleftrightarrow P \vee Q$
 $(\text{if } P \text{ then False else } Q) \longleftrightarrow \neg P \wedge Q$
 $(\text{if } P \text{ then } Q \text{ else True}) \longleftrightarrow \neg P \vee Q$
 $(\text{if } P \text{ then } Q \text{ else False}) \longleftrightarrow P \wedge Q$
 $\langle \text{proof} \rangle$

lemma *if-distrib-fun*: $(\text{if } c \text{ then } f \text{ else } g) x = (\text{if } c \text{ then } f x \text{ else } g x)$
 $\langle \text{proof} \rangle$

3.1 Coinduction

lemma *eq-comp-compI*: $a \circ b = f \circ x \implies x \circ c = \text{id} \implies f = a \circ (b \circ c)$
 $\langle \text{proof} \rangle$

lemma *self-bounded-weaken-left*: $(a :: 'a :: \text{semilattice-inf}) \leq \inf a b \implies a \leq b$
 $\langle \text{proof} \rangle$

lemma *self-bounded-weaken-right*: $(a :: 'a :: \text{semilattice-inf}) \leq \inf b a \implies a \leq b$
 $\langle \text{proof} \rangle$

lemma *symp-iff*: $\text{symp } R \longleftrightarrow R = R^{-1-1}$
 $\langle \text{proof} \rangle$

lemma *equivp-inf*: $\llbracket \text{equivp } R; \text{equivp } S \rrbracket \implies \text{equivp } (\inf R S)$
 $\langle \text{proof} \rangle$

lemma *vimage2p-rel-prod*:

$(\lambda x y. \text{rel-prod } R S (\text{BNF-Def.convolve } f1 g1 x) (\text{BNF-Def.convolve } f2 g2 y)) =$
 $(\inf (\text{BNF-Def.vimage2p } f1 f2 R) (\text{BNF-Def.vimage2p } g1 g2 S))$
 $\langle \text{proof} \rangle$

lemma *predicate2I-obj*: $(\forall x y. P x y \longrightarrow Q x y) \implies P \leq Q$
 $\langle \text{proof} \rangle$

lemma *predicate2D-obj*: $P \leq Q \implies P x y \longrightarrow Q x y$
 $\langle \text{proof} \rangle$

locale *cong* =

fixes *rel* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool})$
and *eval* :: $'b \Rightarrow 'a$
and *retr* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$
assumes *rel-mono*: $\bigwedge R S. R \leq S \implies \text{rel } R \leq \text{rel } S$
and *equivp-retr*: $\bigwedge R. \text{equivp } R \implies \text{equivp } (\text{retr } R)$

and *retr-eval*: $\bigwedge R \ x \ y. \llbracket (\text{rel-fun } (\text{rel } R) \ R) \ \text{eval } \text{eval}; \text{rel } (\text{inf } R \ (\text{retr } R)) \ x \ y \rrbracket$
 \implies
 $\text{retr } R \ (\text{eval } x) \ (\text{eval } y)$
begin

definition *cong* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{cong } R \equiv \text{equivp } R \wedge (\text{rel-fun } (\text{rel } R) \ R) \ \text{eval } \text{eval}$

lemma *cong-retr*: $\text{cong } R \implies \text{cong } (\text{inf } R \ (\text{retr } R))$
 $\langle \text{proof} \rangle$

lemma *cong-equivp*: $\text{cong } R \implies \text{equivp } R$
 $\langle \text{proof} \rangle$

definition *gen-cong* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**
 $\text{gen-cong } R \ j1 \ j2 \equiv \forall R'. \ R \leq R' \wedge \text{cong } R' \longrightarrow R' \ j1 \ j2$

lemma *gen-cong-reflp*[*intro*, *simp*]: $x = y \implies \text{gen-cong } R \ x \ y$
 $\langle \text{proof} \rangle$

lemma *gen-cong-symp*[*intro*]: $\text{gen-cong } R \ x \ y \implies \text{gen-cong } R \ y \ x$
 $\langle \text{proof} \rangle$

lemma *gen-cong-transp*[*intro*]: $\text{gen-cong } R \ x \ y \implies \text{gen-cong } R \ y \ z \implies \text{gen-cong } R \ x \ z$
 $\langle \text{proof} \rangle$

lemma *equivp-gen-cong*: $\text{equivp } (\text{gen-cong } R)$
 $\langle \text{proof} \rangle$

lemma *leq-gen-cong*: $R \leq \text{gen-cong } R$
 $\langle \text{proof} \rangle$

lemmas *imp-gen-cong*[*intro*] = *predicate2D*[*OF leq-gen-cong*]

lemma *gen-cong-minimal*: $\llbracket R \leq R'; \text{cong } R' \rrbracket \implies \text{gen-cong } R \leq R'$
 $\langle \text{proof} \rangle$

lemma *congdd-base-gen-congdd-base-aux*:
 $\text{rel } (\text{gen-cong } R) \ x \ y \implies R \leq R' \implies \text{cong } R' \implies R' \ (\text{eval } x) \ (\text{eval } y)$
 $\langle \text{proof} \rangle$

lemma *cong-gen-cong*: $\text{cong } (\text{gen-cong } R)$
 $\langle \text{proof} \rangle$

lemma *gen-cong-eval-rel-fun*:
 $(\text{rel-fun } (\text{rel } (\text{gen-cong } R)) \ (\text{gen-cong } R)) \ \text{eval } \text{eval}$
 $\langle \text{proof} \rangle$

lemma *gen-cong-eval*:

$rel\ (gen-cong\ R)\ x\ y \implies gen-cong\ R\ (eval\ x)\ (eval\ y)$
 $\langle proof \rangle$

lemma *gen-cong-idem*: $gen-cong\ (gen-cong\ R) = gen-cong\ R$

$\langle proof \rangle$

lemma *gen-cong-rho*:

$\varrho = eval \circ f \implies rel\ (gen-cong\ R)\ (f\ x)\ (f\ y) \implies gen-cong\ R\ (\varrho\ x)\ (\varrho\ y)$
 $\langle proof \rangle$

lemma *coinduction*:

assumes *coind*: $\forall R. R \leq retr\ R \longrightarrow R \leq (=)$

assumes *cih*: $R \leq retr\ (gen-cong\ R)$

shows $R \leq (=)$

$\langle proof \rangle$

end

lemma *rel-sum-case-sum*:

$rel-fun\ (rel-sum\ R\ S)\ T\ (case-sum\ f1\ g1)\ (case-sum\ f2\ g2) = (rel-fun\ R\ T\ f1\ f2$
 $\wedge\ rel-fun\ S\ T\ g1\ g2)$
 $\langle proof \rangle$

context

fixes *rel eval rel' eval' retr emb*

assumes *base*: $cong\ rel\ eval\ retr$

and *step*: $cong\ rel'\ eval'\ retr$

and *emb*: $eval' \circ emb = eval$

and *emb-transfer*: $rel-fun\ (rel\ R)\ (rel'\ R)\ emb\ emb$

begin

interpretation *base*: $cong\ rel\ eval\ retr\ \langle proof \rangle$

interpretation *step*: $cong\ rel'\ eval'\ retr\ \langle proof \rangle$

lemma *gen-cong-emb*: $base.gen-cong\ R \leq step.gen-cong\ R$

$\langle proof \rangle$

end

named-theorems *friend-of-corec-simps*

$\langle ML \rangle$

end

4 A general “while” combinator

theory *While-Combinator*

imports *Main*

begin

Defining partial functions in HOL is tricky. This theory provides a while-combinator that facilitates the definition of (potentially) partial tail-recursive functions.

The theory provides the function *while-option* $b f s$ that iterates f on s while b is true. If iteration terminates with t , *Some* t is returned, *None* otherwise. Thus termination can be shown by proving that *Some* is always returned (for some subset of inputs).

Convenient variations include *while-Some* (for more efficient code) and *while-saturate* (for saturating a set).

4.1 while-option

definition *while-option* $:: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ option}$ **where**
while-option $b c s = (\text{if } (\exists k. \neg b ((c \rightsquigarrow k) s))$
 then Some $((c \rightsquigarrow (\text{LEAST } k. \neg b ((c \rightsquigarrow k) s))) s)$
 else None)

theorem *while-option-unfold*[code]:

while-option $b c s = (\text{if } b s \text{ then } \text{while-option } b c (c s) \text{ else } \text{Some } s)$
 <proof>

lemma *while-option-stop2*:

while-option $b c s = \text{Some } t \implies \exists k. t = (c \rightsquigarrow k) s \wedge \neg b t$
 <proof>

lemma *while-option-stop*: *while-option* $b c s = \text{Some } t \implies \neg b t$
 <proof>

theorem *while-option-rule*:

assumes *step*: $\bigwedge s. P s \implies b s \implies P (c s)$
and *result*: *while-option* $b c s = \text{Some } t$
and *init*: $P s$
shows $P t$
 <proof>

lemma *funpow-commute*:

$\llbracket \forall k' < k. f (c ((c \rightsquigarrow k') s)) = c' (f ((c \rightsquigarrow k') s)) \rrbracket \implies f ((c \rightsquigarrow k) s) = (c' \rightsquigarrow k) (f s)$
 <proof>

lemma *while-option-commute-invariant*:

assumes *Invariant*: $\bigwedge s. P s \implies b s \implies P (c s)$
assumes *TestCommute*: $\bigwedge s. P s \implies b s = b' (f s)$
assumes *BodyCommute*: $\bigwedge s. P s \implies b s \implies f (c s) = c' (f s)$
assumes *Initial*: $P s$
shows *map-option* $f (\text{while-option } b c s) = \text{while-option } b' c' (f s)$
 <proof>

lemma *while-option-commute*:

assumes $\bigwedge s. b\ s = b'\ (f\ s) \ \bigwedge s. \llbracket b\ s \rrbracket \implies f\ (c\ s) = c'\ (f\ s)$

shows $\text{map-option } f\ (\text{while-option } b\ c\ s) = \text{while-option } b'\ c'\ (f\ s)$

$\langle \text{proof} \rangle$

4.2 while

definition $\text{while} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

where $\text{while } b\ c\ s = \text{the } (\text{while-option } b\ c\ s)$

lemma *while-unfold* [code]:

$\text{while } b\ c\ s = (\text{if } b\ s \text{ then } \text{while } b\ c\ (c\ s) \text{ else } s)$

$\langle \text{proof} \rangle$

lemma *def-while-unfold*:

assumes $\text{fdef}: f == \text{while test do}$

shows $f\ x = (\text{if test } x \text{ then } f(\text{do } x) \text{ else } x)$

$\langle \text{proof} \rangle$

The proof rule for *while*, where P is the invariant.

theorem *while-rule-lemma*:

assumes *invariant*: $\bigwedge s. P\ s \implies b\ s \implies P\ (c\ s)$

and *terminate*: $\bigwedge s. P\ s \implies \neg b\ s \implies Q\ s$

and *wf*: $\text{wf } \{(t, s). P\ s \wedge b\ s \wedge t = c\ s\}$

shows $P\ s \implies Q\ (\text{while } b\ c\ s)$

$\langle \text{proof} \rangle$

theorem *while-rule*:

$\llbracket P\ s;$

$\bigwedge s. \llbracket P\ s; b\ s \rrbracket \implies P\ (c\ s);$

$\bigwedge s. \llbracket P\ s; \neg b\ s \rrbracket \implies Q\ s;$

$\text{wf } r;$

$\bigwedge s. \llbracket P\ s; b\ s \rrbracket \implies (c\ s, s) \in r \rrbracket \implies$

$Q\ (\text{while } b\ c\ s)$

$\langle \text{proof} \rangle$

Combine invariant preservation and variant decrease in one goal:

theorem *while-rule2*:

$\llbracket P\ s;$

$\bigwedge s. \llbracket P\ s; b\ s \rrbracket \implies P\ (c\ s) \wedge (c\ s, s) \in r;$

$\bigwedge s. \llbracket P\ s; \neg b\ s \rrbracket \implies Q\ s;$

$\text{wf } r \rrbracket \implies$

$Q\ (\text{while } b\ c\ s)$

$\langle \text{proof} \rangle$

4.3 Termination, lfp and gfp

theorem *wf-while-option-Some*:

assumes $\text{wf } \{(t, s). (P\ s \wedge b\ s) \wedge t = c\ s\}$

and $\bigwedge s. P\ s \implies b\ s \implies P\ (c\ s)$ **and** $P\ s$

shows $\exists t. \text{while-option } b \ c \ s = \text{Some } t$
 $\langle \text{proof} \rangle$

lemma *wf-rel-while-option-Some*:

assumes *wf*: $wf \ R$

assumes *smaller*: $\bigwedge s. P \ s \wedge b \ s \implies (c \ s, s) \in R$

assumes *inv*: $\bigwedge s. P \ s \wedge b \ s \implies P(c \ s)$

assumes *init*: $P \ s$

shows $\exists t. \text{while-option } b \ c \ s = \text{Some } t$

$\langle \text{proof} \rangle$

theorem *measure-while-option-Some*: **fixes** $f :: 'a \Rightarrow \text{nat}$

shows $(\bigwedge s. P \ s \implies b \ s \implies P(c \ s) \wedge f(c \ s) < f \ s)$

$\implies P \ s \implies \exists t. \text{while-option } b \ c \ s = \text{Some } t$

$\langle \text{proof} \rangle$

Kleene iteration starting from the empty set and assuming some finite bounding set:

lemma *while-option-finite-subset-Some*: **fixes** $C :: 'a \text{ set}$

assumes *mono f* **and** $\bigwedge X. X \subseteq C \implies f \ X \subseteq C$ **and** *finite C*

shows $\exists P. \text{while-option } (\lambda A. f \ A \neq A) \ f \ \{\} = \text{Some } P$

$\langle \text{proof} \rangle$

lemma *lfp-the-while-option*:

assumes *mono f* **and** $\bigwedge X. X \subseteq C \implies f \ X \subseteq C$ **and** *finite C*

shows $\text{lfp } f = \text{the}(\text{while-option } (\lambda A. f \ A \neq A) \ f \ \{\})$

$\langle \text{proof} \rangle$

lemma *lfp-while*:

assumes *mono f* **and** $\bigwedge X. X \subseteq C \implies f \ X \subseteq C$ **and** *finite C*

shows $\text{lfp } f = \text{while } (\lambda A. f \ A \neq A) \ f \ \{\}$

$\langle \text{proof} \rangle$

lemma *wf-finite-less*:

assumes *finite* $(C :: 'a::\text{order set})$

shows $wf \ \{(x, y). \{x, y\} \subseteq C \wedge x < y\}$

$\langle \text{proof} \rangle$

lemma *wf-finite-greater*:

assumes *finite* $(C :: 'a::\text{order set})$

shows $wf \ \{(x, y). \{x, y\} \subseteq C \wedge y < x\}$

$\langle \text{proof} \rangle$

lemma *while-option-finite-increasing-Some*:

fixes $f :: 'a::\text{order} \Rightarrow 'a$

assumes *mono f* **and** *finite* $(UNIV :: 'a \text{ set})$ **and** $s \leq f \ s$

shows $\exists P. \text{while-option } (\lambda A. f \ A \neq A) \ f \ s = \text{Some } P$

$\langle \text{proof} \rangle$

lemma *lfp-the-while-option-lattice*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono* f **and** *finite* ($UNIV :: 'a \text{ set}$)
shows $\text{lfp } f = \text{the } (\text{while-option } (\lambda A. f A \neq A) f \text{ bot})$
 $\langle \text{proof} \rangle$

lemma *lfp-while-lattice*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono* f **and** *finite* ($UNIV :: 'a \text{ set}$)
shows $\text{lfp } f = \text{while } (\lambda A. f A \neq A) f \text{ bot}$
 $\langle \text{proof} \rangle$

lemma *while-option-finite-decreasing-Some*:
fixes $f :: 'a::\text{order} \Rightarrow 'a$
assumes *mono* f **and** *finite* ($UNIV :: 'a \text{ set}$) **and** $f s \leq s$
shows $\exists P. \text{while-option } (\lambda A. f A \neq A) f s = \text{Some } P$
 $\langle \text{proof} \rangle$

lemma *gfp-the-while-option-lattice*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono* f **and** *finite* ($UNIV :: 'a \text{ set}$)
shows $\text{gfp } f = \text{the}(\text{while-option } (\lambda A. f A \neq A) f \text{ top})$
 $\langle \text{proof} \rangle$

lemma *gfp-while-lattice*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono* f **and** *finite* ($UNIV :: 'a \text{ set}$)
shows $\text{gfp } f = \text{while } (\lambda A. f A \neq A) f \text{ top}$
 $\langle \text{proof} \rangle$

4.4 *while-Some* **and** *while-saturate*

A variation intended for efficient code. The problem with *while-option* $b \ c$: the computations of b and c may share subcomputations but they need to be performed twice.

definition *while-Some* $:: ('s \Rightarrow 's \text{ option}) \Rightarrow 's \Rightarrow 's \text{ option}$ **where**
 $\text{while-Some } f = \text{while-option } (\lambda s. f s \neq \text{None}) (\text{the } o \ f)$

lemma *while-Some-rec[code]*:
 $\text{while-Some } f \ x = (\text{case } f \ x \text{ of } \text{None} \Rightarrow \text{Some } x \mid \text{Some } y \Rightarrow \text{while-Some } f \ y)$
 $\langle \text{proof} \rangle$

A frequent special case: saturation of a set.

definition *while-saturate* $:: ('a \text{ set} \Rightarrow 'a \text{ set}) \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set option}$ **where**
 $\text{while-saturate } f = \text{while-option } (\lambda M. \neg f M \subseteq M) (\lambda M. M \cup f M)$

lemma *while-option-cong*: $(\bigwedge s. b \ s \Longrightarrow c \ s = c' \ s) \Longrightarrow \text{while-option } b \ c \ s = \text{while-option } b \ c' \ s$

<proof>

lemma *while-saturate-code*[code]: *while-saturate* f M =
while-Some ($\lambda M. \text{let } M' = f\ M \text{ in if } M' \subseteq M \text{ then None else Some } (M \cup M')$) M
<proof>

Termination:

lemma *while-option-sat-finite-subset-Some*: **fixes** $C :: 'a \text{ set}$
assumes *mono* f **and** $\bigwedge X. X \subseteq C \implies f\ X \subseteq C$ **and** *finite* C **and** $M \subseteq C$
shows $\exists S. \text{while-option } (\lambda M. \neg f\ M \subseteq M) (\lambda M. M \cup f\ M) M = \text{Some } S$
<proof>

corollary *while-saturate-finite-subset-Some*:
assumes *mono* f **and** $\bigwedge X. X \subseteq C \implies f\ X \subseteq C$ **and** *finite* C **and** $M \subseteq C$
shows $\exists S. \text{while-saturate } f\ M = \text{Some } S$
<proof>

Correctness: finds the least saturated/closed set above M

lemma *while-option-sat-prefix*: **assumes** *mono* f
and *while-option* ($\lambda M. \neg f\ M \subseteq M$) ($\lambda M. M \cup f\ M$) $M = \text{Some } S$
and $M \subseteq P$ **and** $f\ P \subseteq P$
shows $S \subseteq P$
<proof>

corollary *while-saturate-prefix*:
 $\llbracket \text{mono } f; \text{while-saturate } f\ M = \text{Some } S; M \subseteq P; f\ P \subseteq P \rrbracket \implies S \subseteq P$
<proof>

4.5 Reflexive, transitive closure

Computing the reflexive, transitive closure by iterating a successor function.
 Stops when an element is found that does not satisfy the test.

More refined (and hence more efficient) versions can be found in ITP 2011
 paper by Nipkow (the theories are in the AFP entry *Flyspeck* by Nipkow)
 and the AFP article *Executable Transitive Closures* by René Thiemann.

context

fixes $p :: 'a \Rightarrow \text{bool}$
and $f :: 'a \Rightarrow 'a \text{ list}$
and $x :: 'a$

begin

qualified fun *rtrancl-while-test* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow \text{bool}$
where *rtrancl-while-test* ($ws, -$) = ($ws \neq [] \wedge p(\text{hd } ws)$)

qualified fun *rtrancl-while-step* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow 'a \text{ list} \times 'a \text{ set}$
where *rtrancl-while-step* (ws, Z) =
 ($\text{let } x = \text{hd } ws; \text{new} = \text{remdups } (\text{filter } (\lambda y. y \notin Z) (f\ x))$
 $\text{in } (\text{new} @ \text{tl } ws, \text{set new } \cup Z)$)

definition *rtrancl-while* :: ('a list * 'a set) option

where *rtrancl-while* = while-option *rtrancl-while-test* *rtrancl-while-step* ([x], {x})

qualified fun *rtrancl-while-invariant* :: 'a list × 'a set ⇒ bool

where *rtrancl-while-invariant* (ws, Z) =

($x \in Z \wedge \text{set } ws \subseteq Z \wedge \text{distinct } ws \wedge \{(x, y). y \in \text{set}(f x)\} \text{ “ } (Z - \text{set } ws) \subseteq Z$
 \wedge
 $Z \subseteq \{(x, y). y \in \text{set}(f x)\}^* \text{ “ } \{x\} \wedge (\forall z \in Z - \text{set } ws. p z)$)

qualified lemma *rtrancl-while-invariant*:

assumes *inv*: *rtrancl-while-invariant* *st* **and** *test*: *rtrancl-while-test* *st*

shows *rtrancl-while-invariant* (*rtrancl-while-step* *st*)

⟨proof⟩

lemma *rtrancl-while-Some*:

assumes *rtrancl-while* = *Some*(ws, Z)

shows if ws = []

then $Z = \{(x, y). y \in \text{set}(f x)\}^* \text{ “ } \{x\} \wedge (\forall z \in Z. p z)$

else $\neg p(\text{hd } ws) \wedge \text{hd } ws \in \{(x, y). y \in \text{set}(f x)\}^* \text{ “ } \{x\}$

⟨proof⟩

lemma *rtrancl-while-finite-Some*:

assumes *finite* ($\{(x, y). y \in \text{set}(f x)\}^* \text{ “ } \{x\}$) (**is finite** ?Cl)

shows $\exists y. \text{rtrancl-while} = \text{Some } y$

⟨proof⟩

end

end

5 The Bourbaki-Witt tower construction for transfinite iteration

theory *Bourbaki-Witt-Fixpoint*

imports *While-Combinator*

begin

lemma *ChainsI* [intro?]:

($\bigwedge a b. \llbracket a \in Y; b \in Y \rrbracket \implies (a, b) \in r \vee (b, a) \in r \implies Y \in \text{Chains } r$
 ⟨proof⟩

lemma *in-Chains-subset*: $\llbracket M \in \text{Chains } r; M' \subseteq M \rrbracket \implies M' \in \text{Chains } r$

⟨proof⟩

lemma *in-ChainsD*: $\llbracket M \in \text{Chains } r; x \in M; y \in M \rrbracket \implies (x, y) \in r \vee (y, x) \in r$

⟨proof⟩

lemma *Chains-FieldD*: $\llbracket M \in \text{Chains } r; x \in M \rrbracket \implies x \in \text{Field } r$
 $\langle \text{proof} \rangle$

lemma *in-Chains-conv-chain*: $M \in \text{Chains } r \iff \text{Complete-Partial-Order.chain}$
 $(\lambda x y. (x, y) \in r) M$
 $\langle \text{proof} \rangle$

lemma *partial-order-on-trans*:
 $\llbracket \text{partial-order-on } A \text{ } r; (x, y) \in r; (y, z) \in r \rrbracket \implies (x, z) \in r$
 $\langle \text{proof} \rangle$

locale *bourbaki-witt-fixpoint* =
 fixes *lub* :: $'a \text{ set} \Rightarrow 'a$
 and *leq* :: $('a \times 'a) \text{ set}$
 and *f* :: $'a \Rightarrow 'a$
 assumes *po*: *Partial-order leq*
 and *lub-least*: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\}; \bigwedge x. x \in M \implies (x, z) \in \text{leq} \rrbracket \implies (\text{lub } M, z) \in \text{leq}$
 and *lub-upper*: $\llbracket M \in \text{Chains } \text{leq}; x \in M \rrbracket \implies (x, \text{lub } M) \in \text{leq}$
 and *lub-in-Field*: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\} \rrbracket \implies \text{lub } M \in \text{Field } \text{leq}$
 and *increasing*: $\bigwedge x. x \in \text{Field } \text{leq} \implies (x, f x) \in \text{leq}$
begin

lemma *leq-trans*: $\llbracket (x, y) \in \text{leq}; (y, z) \in \text{leq} \rrbracket \implies (x, z) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *leq-refl*: $x \in \text{Field } \text{leq} \implies (x, x) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *leq-antisym*: $\llbracket (x, y) \in \text{leq}; (y, x) \in \text{leq} \rrbracket \implies x = y$
 $\langle \text{proof} \rangle$

inductive-set *iterates-above* :: $'a \Rightarrow 'a \text{ set}$
 for *a*
where
 base: $a \in \text{iterates-above } a$
 | step: $x \in \text{iterates-above } a \implies f x \in \text{iterates-above } a$
 | Sup: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\}; \bigwedge x. x \in M \implies x \in \text{iterates-above } a \rrbracket \implies \text{lub } M \in \text{iterates-above } a$

definition *fixp-above* :: $'a \Rightarrow 'a$
where *fixp-above* *a* = (if $a \in \text{Field } \text{leq}$ then $\text{lub } (\text{iterates-above } a)$ else *a*)

lemma *fixp-above-outside*: $a \notin \text{Field } \text{leq} \implies \text{fixp-above } a = a$
 $\langle \text{proof} \rangle$

lemma *fixp-above-inside*: $a \in \text{Field } \text{leq} \implies \text{fixp-above } a = \text{lub } (\text{iterates-above } a)$
 $\langle \text{proof} \rangle$

context

notes *leq-refl* [*intro!*, *simp*]

and *base* [*intro*]

and *step* [*intro*]

and *Sup* [*intro*]

and *leq-trans* [*trans*]

begin

lemma *iterates-above-le-f*: $\llbracket x \in \text{iterates-above } a; a \in \text{Field } \text{leq} \rrbracket \implies (x, f\ x) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *iterates-above-Field*: $\llbracket x \in \text{iterates-above } a; a \in \text{Field } \text{leq} \rrbracket \implies x \in \text{Field } \text{leq}$
 $\langle \text{proof} \rangle$

lemma *iterates-above-ge*:
 assumes $y: y \in \text{iterates-above } a$
 and $a: a \in \text{Field } \text{leq}$
 shows $(a, y) \in \text{leq}$
 $\langle \text{proof} \rangle$

lemma *iterates-above-lub*:
 assumes $M: M \in \text{Chains } \text{leq}$
 and *nempty*: $M \neq \{\}$
 and *upper*: $\bigwedge y. y \in M \implies \exists z \in M. (y, z) \in \text{leq} \wedge z \in \text{iterates-above } a$
 shows $\text{lub } M \in \text{iterates-above } a$
 $\langle \text{proof} \rangle$

lemma *iterates-above-successor*:
 assumes $y: y \in \text{iterates-above } a$
 and $a: a \in \text{Field } \text{leq}$
 shows $y = a \vee y \in \text{iterates-above } (f\ a)$
 $\langle \text{proof} \rangle$

lemma *iterates-above-Sup-aux*:
 assumes $M: M \in \text{Chains } \text{leq}$ $M \neq \{\}$
 and $M': M' \in \text{Chains } \text{leq}$ $M' \neq \{\}$
 and *comp*: $\bigwedge x. x \in M \implies x \in \text{iterates-above } (\text{lub } M') \vee \text{lub } M' \in \text{iterates-above } x$
 shows $(\text{lub } M, \text{lub } M') \in \text{leq} \vee \text{lub } M \in \text{iterates-above } (\text{lub } M')$
 $\langle \text{proof} \rangle$

lemma *iterates-above-triangle*:
 assumes $x: x \in \text{iterates-above } a$
 and $y: y \in \text{iterates-above } a$
 and $a: a \in \text{Field } \text{leq}$
 shows $x \in \text{iterates-above } y \vee y \in \text{iterates-above } x$
 $\langle \text{proof} \rangle$

lemma *chain-iterates-above*:

assumes $a: a \in \text{Field } \text{leq}$

shows $\text{iterates-above } a \in \text{Chains } \text{leq} \text{ (is } ?C \in -)$

$\langle \text{proof} \rangle$

lemma *fixp-iterates-above*: $\text{fixp-above } a \in \text{iterates-above } a$

$\langle \text{proof} \rangle$

lemma *fixp-above-Field*: $a \in \text{Field } \text{leq} \implies \text{fixp-above } a \in \text{Field } \text{leq}$

$\langle \text{proof} \rangle$

lemma *fixp-above-unfold*:

assumes $a: a \in \text{Field } \text{leq}$

shows $\text{fixp-above } a = f (\text{fixp-above } a) \text{ (is } ?a = f ?a)$

$\langle \text{proof} \rangle$

end

lemma *fixp-above-induct* [*case-names adm base step*]:

assumes $\text{adm}: \text{ccpo.admissible } \text{lub } (\lambda x y. (x, y) \in \text{leq}) P$

and $\text{base}: P a$

and $\text{step}: \bigwedge x. P x \implies P (f x)$

shows $P (\text{fixp-above } a)$

$\langle \text{proof} \rangle$

end

5.1 Connect with the while combinator for executability on chain-finite lattices.

context *bourbaki-witt-fixpoint* **begin**

lemma *in-Chains-finite*: — Translation from $\llbracket \text{Complete-Partial-Order.chain } (\leq) ?A; \text{finite } ?A; ?A \neq \{\} \rrbracket \implies \text{Sup } ?A \in ?A$.

assumes $M \in \text{Chains } \text{leq}$

and $M \neq \{\}$

and $\text{finite } M$

shows $\text{lub } M \in M$

$\langle \text{proof} \rangle$

lemma *fun-pow-iterates-above*: $(f \smallfrown k) a \in \text{iterates-above } a$

$\langle \text{proof} \rangle$

lemma *chfin-iterates-above-fun-pow*:

assumes $x \in \text{iterates-above } a$

assumes $\forall M \in \text{Chains } \text{leq}. \text{finite } M$

shows $\exists j. x = (f \smallfrown j) a$

$\langle \text{proof} \rangle$

lemma *Chain-finite-iterates-above-fun-pow-iff*:

assumes $\forall M \in \text{Chains leq. finite } M$

shows $x \in \text{iterates-above } a \longleftrightarrow (\exists j. x = (f \smallfrown j) a)$

<proof>

lemma *fixp-above-Kleene-iter-ex*:

assumes $(\forall M \in \text{Chains leq. finite } M)$

obtains k **where** $\text{fixp-above } a = (f \smallfrown k) a$

<proof>

context **fixes** a **assumes** $a: a \in \text{Field leq}$ **begin**

lemma *funpow-Field-leq*: $(f \smallfrown k) a \in \text{Field leq}$

<proof>

lemma *funpow-prefix*: $j < k \implies ((f \smallfrown j) a, (f \smallfrown k) a) \in \text{leq}$

<proof>

lemma *funpow-suffix*: $(f \smallfrown \text{Suc } k) a = (f \smallfrown k) a \implies ((f \smallfrown (j + k)) a, (f \smallfrown k) a) \in \text{leq}$

<proof>

lemma *funpow-stability*: $(f \smallfrown \text{Suc } k) a = (f \smallfrown k) a \implies ((f \smallfrown j) a, (f \smallfrown k) a) \in \text{leq}$

<proof>

lemma *funpow-in-Chains*: $\{(f \smallfrown k) a \mid k. \text{True}\} \in \text{Chains leq}$

<proof>

lemma *fixp-above-Kleene-iter*:

assumes $\forall M \in \text{Chains leq. finite } M$ — convenient but surely not necessary

assumes $(f \smallfrown \text{Suc } k) a = (f \smallfrown k) a$

shows $\text{fixp-above } a = (f \smallfrown k) a$

<proof>

context **assumes** $\text{chfin}: \forall M \in \text{Chains leq. finite } M$ **begin**

lemma *Chain-finite-wf*: $\text{wf } \{(f \smallfrown (f \smallfrown k) a), (f \smallfrown k) a \mid k. f \smallfrown (f \smallfrown k) a \neq (f \smallfrown k) a\}$

<proof>

lemma *while-option-finite-increasing*: $\exists P. \text{while-option } (\lambda A. f A \neq A) f a = \text{Some } P$

<proof>

lemma *fixp-above-the-while-option*: $\text{fixp-above } a = \text{the } (\text{while-option } (\lambda A. f A \neq A) f a)$

<proof>

lemma *fixp-above-conv-while*: $\text{fixp-above } a = \text{while } (\lambda A. f A \neq A) f a$
 $\langle \text{proof} \rangle$

end

end

end

lemma *bourbaki-witt-fixpoint-complete-latticeI*:
fixes $f :: 'a :: \text{complete-lattice} \Rightarrow 'a$
assumes $\bigwedge x. x \leq f x$
shows *bourbaki-witt-fixpoint* $\text{Sup } \{(x, y). x \leq y\} f$
 $\langle \text{proof} \rangle$

end

6 Division with modulus centered towards zero.

theory *Centered-Division*

imports *Main*

begin

lemma *off-iff-abs-mod-2-eq-one*:
 $\langle \text{odd } l \longleftrightarrow |l| \bmod 2 = 1 \rangle \text{ for } l :: \text{int}$
 $\langle \text{proof} \rangle$

The following specification of division on integers centers the modulus around zero. This is useful e.g. to define division on Gauss numbers. N.b.: This is not mentioned [2].

definition *centered-divide* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ (**infixl** $\langle \text{cdiv} \rangle 70$)
where $\langle k \text{ cdiv } l = \text{sgn } l * ((k + |l| \text{ div } 2) \text{ div } |l|) \rangle$

definition *centered-modulo* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ (**infixl** $\langle \text{cmod} \rangle 70$)
where $\langle k \text{ cmod } l = (k + |l| \text{ div } 2) \bmod |l| - |l| \text{ div } 2 \rangle$

Example: $k \text{ cmod } 5 \in \{-2, -1, 0, 1, 2\}$

lemma *signed-take-bit-eq-cmod*:
 $\langle \text{signed-take-bit } n k = k \text{ cmod } (2 \wedge \text{Suc } n) \rangle$
 $\langle \text{proof} \rangle$

Property $\text{signed-take-bit } n k = k \text{ cmod } 2^{\text{Suc } n}$ is the key to generalize centered division to arbitrary structures satisfying *ring-bit-operations*, but so far it is not clear what practical relevance that would have.

lemma *cdiv-mult-cmod-eq*:
 $\langle k \text{ cdiv } l * l + k \text{ cmod } l = k \rangle$
 $\langle \text{proof} \rangle$

lemma *mult-cdiv-cmod-eq*:
 $\langle l * (k \text{ cdiv } l) + k \text{ cmod } l = k \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-cdiv-mult-eq*:
 $\langle k \text{ cmod } l + k \text{ cdiv } l * l = k \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-mult-cdiv-eq*:
 $\langle k \text{ cmod } l + l * (k \text{ cdiv } l) = k \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-cdiv-mult-eq-cmod*:
 $\langle k - k \text{ cdiv } l * l = k \text{ cmod } l \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-mult-cdiv-eq-cmod*:
 $\langle k - l * (k \text{ cdiv } l) = k \text{ cmod } l \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-cmod-eq-cdiv-mult*:
 $\langle k - k \text{ cmod } l = k \text{ cdiv } l * l \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-cmod-eq-mult-cdiv*:
 $\langle k - k \text{ cmod } l = l * (k \text{ cdiv } l) \rangle$
 $\langle \text{proof} \rangle$

lemma *cdiv-0-eq [simp]*:
 $\langle k \text{ cdiv } 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-0-eq [simp]*:
 $\langle k \text{ cmod } 0 = k \rangle$
 $\langle \text{proof} \rangle$

lemma *cdiv-1-eq [simp]*:
 $\langle k \text{ cdiv } 1 = k \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-1-eq [simp]*:
 $\langle k \text{ cmod } 1 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *zero-cdiv-eq [simp]*:
 $\langle 0 \text{ cdiv } k = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *zero-cmod-eq [simp]*:

$\langle 0 \text{ cmod } k = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *cdiv-minus-eq*:
 $\langle k \text{ cdiv } - l = - (k \text{ cdiv } l) \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-minus-eq [simp]*:
 $\langle k \text{ cmod } - l = k \text{ cmod } l \rangle$
 $\langle \text{proof} \rangle$

lemma *cdiv-abs-eq*:
 $\langle k \text{ cdiv } |l| = \text{sgn } l * (k \text{ cdiv } l) \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-abs-eq [simp]*:
 $\langle k \text{ cmod } |l| = k \text{ cmod } l \rangle$
 $\langle \text{proof} \rangle$

lemma *nonzero-mult-cdiv-cancel-right*:
 $\langle k * l \text{ cdiv } l = k \rangle \text{ if } \langle l \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *cdiv-self-eq [simp]*:
 $\langle k \text{ cdiv } k = 1 \rangle \text{ if } \langle k \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-self-eq [simp]*:
 $\langle k \text{ cmod } k = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-less-divisor*:
 $\langle k \text{ cmod } l < |l| - |l| \text{ div } 2 \rangle \text{ if } \langle l \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *cmod-less-equal-divisor*:
 $\langle k \text{ cmod } l \leq |l| \text{ div } 2 \rangle \text{ if } \langle l \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *divisor-less-equal-cmod'*:
 $\langle |l| \text{ div } 2 - |l| \leq k \text{ cmod } l \rangle \text{ if } \langle l \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *divisor-less-equal-cmod*:
 $\langle -(|l| \text{ div } 2) \leq k \text{ cmod } l \rangle \text{ if } \langle l \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *abs-cmod-less-equal*:
 $\langle |k \text{ cmod } l| \leq |l| \text{ div } 2 \rangle \text{ if } \langle l \neq 0 \rangle$

⟨proof⟩

end

7 Order on characters

theory *Char-ord*

imports *Main*

begin

instantiation *char* :: *linorder*

begin

definition *less-eq-char* :: $\langle \text{char} \Rightarrow \text{char} \Rightarrow \text{bool} \rangle$

where $\langle c1 \leq c2 \iff \text{of-char } c1 \leq (\text{of-char } c2 :: \text{nat}) \rangle$

definition *less-char* :: $\langle \text{char} \Rightarrow \text{char} \Rightarrow \text{bool} \rangle$

where $\langle c1 < c2 \iff \text{of-char } c1 < (\text{of-char } c2 :: \text{nat}) \rangle$

instance

⟨proof⟩

end

lemma *less-eq-char-simp* [*simp*, *code*]:

$\langle \text{Char } b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7 \leq \text{Char } c0\ c1\ c2\ c3\ c4\ c5\ c6\ c7$

$\iff \text{lexordp-eq } [b7, b6, b5, b4, b3, b2, b1, b0] [c7, c6, c5, c4, c3, c2, c1, c0] \rangle$

⟨proof⟩

lemma *less-char-simp* [*simp*, *code*]:

$\langle \text{Char } b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7 < \text{Char } c0\ c1\ c2\ c3\ c4\ c5\ c6\ c7$

$\iff \text{ord-class.lexordp } [b7, b6, b5, b4, b3, b2, b1, b0] [c7, c6, c5, c4, c3, c2, c1, c0] \rangle$

⟨proof⟩

instantiation *char* :: *distrib-lattice*

begin

definition $\langle (\text{inf} :: \text{char} \Rightarrow -) = \text{min} \rangle$

definition $\langle (\text{sup} :: \text{char} \Rightarrow -) = \text{max} \rangle$

instance

⟨proof⟩

end

code-identifier

code-module *Char-ord* \rightarrow

```

    (SML) Str and (OCaml) Str and (Haskell) Str and (Scala) Str
end

```

8 A generic phantom type

```

theory Phantom-Type
imports Main
begin

```

```

datatype ('a, 'b) phantom = phantom (of-phantom: 'b)

```

```

lemma type-definition-phantom': type-definition of-phantom phantom UNIV
<proof>

```

```

lemma phantom-comp-of-phantom [simp]: phantom  $\circ$  of-phantom = id
and of-phantom-comp-phantom [simp]: of-phantom  $\circ$  phantom = id
<proof>

```

```

syntax -Phantom :: type  $\Rightarrow$  logic ( $\langle$ ( $\langle$ indent=1 notation= $\langle$ mixfix Phantom $\rangle$  $\rangle$ Phantom/(1'(-))) $\rangle$ )
syntax-consts -Phantom == phantom

```

```

translations

```

```

    Phantom('t) => CONST phantom :: -  $\Rightarrow$  ('t, -) phantom

```

```

<ML>

```

```

lemma of-phantom-inject [simp]:
    of-phantom x = of-phantom y  $\longleftrightarrow$  x = y
<proof>

```

```

end

```

9 Cardinality of types

```

theory Cardinality
imports Phantom-Type
begin

```

9.1 Preliminary lemmas

```

lemma (in type-definition) univ:
    UNIV = Abs ' A
<proof>

```

```

lemma (in type-definition) card: card (UNIV :: 'b set) = card A
<proof>

```

9.2 Cardinalities of types

syntax *-type-card* :: *type* => *nat* ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix CARD} \rangle \rangle \text{CARD} / (1'(-')) \rangle \rangle$)

syntax-consts *-type-card* == *card*

translations *CARD*('t) => *CONST card* (*CONST UNIV* :: 't *set*)

$\langle ML \rangle$

lemma *card-prod* [*simp*]: *CARD*('a × 'b) = *CARD*('a) * *CARD*('b)
 $\langle proof \rangle$

lemma *card-UNIV-sum*: *CARD*('a + 'b) = (if *CARD*('a) ≠ 0 ∧ *CARD*('b) ≠ 0 then *CARD*('a) + *CARD*('b) else 0)
 $\langle proof \rangle$

lemma *card-sum* [*simp*]: *CARD*('a + 'b) = *CARD*('a::finite) + *CARD*('b::finite)
 $\langle proof \rangle$

lemma *card-UNIV-option*: *CARD*('a option) = (if *CARD*('a) = 0 then 0 else *CARD*('a) + 1)
 $\langle proof \rangle$

lemma *card-option* [*simp*]: *CARD*('a option) = *Suc CARD*('a::finite)
 $\langle proof \rangle$

lemma *card-UNIV-set*: *CARD*('a set) = (if *CARD*('a) = 0 then 0 else $2^{\text{CARD}('a)}$)
 $\langle proof \rangle$

lemma *card-set* [*simp*]: *CARD*('a set) = $2^{\text{CARD}('a::finite)}$
 $\langle proof \rangle$

lemma *card-nat* [*simp*]: *CARD*(*nat*) = 0
 $\langle proof \rangle$

lemma *card-fun*: *CARD*('a ⇒ 'b) = (if *CARD*('a) ≠ 0 ∧ *CARD*('b) ≠ 0 ∨ *CARD*('b) = 1 then *CARD*('b) ^ *CARD*('a) else 0)
 $\langle proof \rangle$

corollary *finite-UNIV-fun*:
 $\text{finite } (UNIV :: ('a \Rightarrow 'b) \text{ set}) \longleftrightarrow$
 $\text{finite } (UNIV :: 'a \text{ set}) \wedge \text{finite } (UNIV :: 'b \text{ set}) \vee \text{CARD}('b) = 1$
 (is ?lhs \longleftrightarrow ?rhs)
 $\langle proof \rangle$

lemma *card-literal*: *CARD*(*String.literal*) = 0
 $\langle proof \rangle$

9.3 Classes with at least 1 and 2

Class `finite` already captures "at least 1"

lemma *zero-less-card-finite* [simp]: $0 < \text{CARD}('a::\text{finite})$
 $\langle \text{proof} \rangle$

lemma *one-le-card-finite* [simp]: $\text{Suc } 0 \leq \text{CARD}('a::\text{finite})$
 $\langle \text{proof} \rangle$

class *CARD-1* =
assumes *CARD-1*: $\text{CARD } ('a) = 1$
begin

subclass *finite*
 $\langle \text{proof} \rangle$

end

Class for cardinality "at least 2"

class *card2* = *finite* +
assumes *two-le-card*: $2 \leq \text{CARD}('a)$

lemma *one-less-card*: $\text{Suc } 0 < \text{CARD}('a::\text{card2})$
 $\langle \text{proof} \rangle$

lemma *one-less-int-card*: $1 < \text{int } \text{CARD}('a::\text{card2})$
 $\langle \text{proof} \rangle$

9.4 A type class for deciding finiteness of types

type-synonym *'a finite-UNIV* = (*'a*, *bool*) *phantom*

class *finite-UNIV* =
fixes *finite-UNIV* :: (*'a*, *bool*) *phantom*
assumes *finite-UNIV*: *finite-UNIV* = *Phantom*('a) (*finite* (*UNIV* :: 'a *set*))

lemma *finite-UNIV-code* [code-unfold]:
 $\text{finite } (\text{UNIV} :: 'a :: \text{finite-UNIV } \text{set})$
 $\longleftrightarrow \text{of-phantom } (\text{finite-UNIV} :: 'a \text{ finite-UNIV})$
 $\langle \text{proof} \rangle$

9.5 A type class for computing the cardinality of types

definition *is-list-UNIV* :: 'a *list* \Rightarrow *bool*
where *is-list-UNIV* *xs* = (let *c* = $\text{CARD}('a)$ in if *c* = 0 then *False* else $\text{size } (\text{remdups } xs) = c$)

lemma *is-list-UNIV-iff*: $\text{is-list-UNIV } xs \longleftrightarrow \text{set } xs = \text{UNIV}$
 $\langle \text{proof} \rangle$

type-synonym 'a card-UNIV = ('a, nat) phantom

class card-UNIV = finite-UNIV +
fixes card-UNIV :: 'a card-UNIV
assumes card-UNIV: card-UNIV = Phantom('a) CARD('a)

9.6 Instantiations for card-UNIV

instantiation nat :: card-UNIV **begin**
definition finite-UNIV = Phantom(nat) False
definition card-UNIV = Phantom(nat) 0
instance <proof>
end

instantiation int :: card-UNIV **begin**
definition finite-UNIV = Phantom(int) False
definition card-UNIV = Phantom(int) 0
instance <proof>
end

instantiation natural :: card-UNIV **begin**
definition finite-UNIV = Phantom(natural) False
definition card-UNIV = Phantom(natural) 0
instance
 <proof>
end

instantiation integer :: card-UNIV **begin**
definition finite-UNIV = Phantom(integer) False
definition card-UNIV = Phantom(integer) 0
instance
 <proof>
end

instantiation list :: (type) card-UNIV **begin**
definition finite-UNIV = Phantom('a list) False
definition card-UNIV = Phantom('a list) 0
instance <proof>
end

instantiation unit :: card-UNIV **begin**
definition finite-UNIV = Phantom(unit) True
definition card-UNIV = Phantom(unit) 1
instance <proof>
end

instantiation bool :: card-UNIV **begin**
definition finite-UNIV = Phantom(bool) True

definition *card-UNIV* = *Phantom*(*bool*) 2
instance $\langle \text{proof} \rangle$
end

instantiation *char* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*char*) *True*
definition *card-UNIV* = *Phantom*(*char*) 256
instance $\langle \text{proof} \rangle$
end

instantiation *prod* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('a \times 'b)
 (*of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*) \wedge *of-phantom* (*finite-UNIV* :: 'b
finite-UNIV))
instance $\langle \text{proof} \rangle$
end

instantiation *prod* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('a \times 'b)
 (*of-phantom* (*card-UNIV* :: 'a *card-UNIV*) * *of-phantom* (*card-UNIV* :: 'b *card-UNIV*))
instance $\langle \text{proof} \rangle$
end

instantiation *sum* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('a + 'b)
 (*of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*) \wedge *of-phantom* (*finite-UNIV* :: 'b
finite-UNIV))
instance
 $\langle \text{proof} \rangle$
end

instantiation *sum* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('a + 'b)
 (*let* *ca* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*);
 cb = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)
 in if *ca* \neq 0 \wedge *cb* \neq 0 *then* *ca* + *cb* *else* 0)
instance $\langle \text{proof} \rangle$
end

instantiation *fun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('a \Rightarrow 'b)
 (*let* *cb* = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)
 in *cb* = 1 \vee *of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*) \wedge *cb* \neq 0)
instance
 $\langle \text{proof} \rangle$
end

instantiation *fun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('a \Rightarrow 'b)

```

    (let ca = of-phantom (card-UNIV :: 'a card-UNIV);
      cb = of-phantom (card-UNIV :: 'b card-UNIV)
      in if ca ≠ 0 ∧ cb ≠ 0 ∨ cb = 1 then cb ^ ca else 0)
instance ⟨proof⟩
end

instantiation option :: (finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a option) (of-phantom (finite-UNIV :: 'a fi-
nite-UNIV))
instance ⟨proof⟩
end

instantiation option :: (card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a option)
  (let c = of-phantom (card-UNIV :: 'a card-UNIV) in if c ≠ 0 then Suc c else 0)
instance ⟨proof⟩
end

instantiation String.literal :: card-UNIV begin
definition finite-UNIV = Phantom(String.literal) False
definition card-UNIV = Phantom(String.literal) 0
instance
  ⟨proof⟩
end

instantiation set :: (finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a set) (of-phantom (finite-UNIV :: 'a fi-
nite-UNIV))
instance ⟨proof⟩
end

instantiation set :: (card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a set)
  (let c = of-phantom (card-UNIV :: 'a card-UNIV) in if c = 0 then 0 else 2 ^ c)
instance ⟨proof⟩
end

lemma UNIV-finite-1: UNIV = set [finite-1.a1]
  ⟨proof⟩

lemma UNIV-finite-2: UNIV = set [finite-2.a1, finite-2.a2]
  ⟨proof⟩

lemma UNIV-finite-3: UNIV = set [finite-3.a1, finite-3.a2, finite-3.a3]
  ⟨proof⟩

lemma UNIV-finite-4: UNIV = set [finite-4.a1, finite-4.a2, finite-4.a3, finite-4.a4]
  ⟨proof⟩

```


lemma *UNIV-finite-5*:

UNIV = set [*finite-5.a*₁, *finite-5.a*₂, *finite-5.a*₃, *finite-5.a*₄, *finite-5.a*₅]
 ⟨*proof*⟩

instantiation *Enum.finite-1* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-1*) 1

instance

⟨*proof*⟩

end

instantiation *Enum.finite-2* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-2*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-2*) 2

instance

⟨*proof*⟩

end

instantiation *Enum.finite-3* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-3*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-3*) 3

instance

⟨*proof*⟩

end

instantiation *Enum.finite-4* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-4*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-4*) 4

instance

⟨*proof*⟩

end

instantiation *Enum.finite-5* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-5*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-5*) 5

instance

⟨*proof*⟩

end

end

10 Code setup for sets with cardinality type information

theory *Code-Cardinality* **imports** *Cardinality* **begin**

Implement *CARD*(*a*) via *card-UNIV-class.card-UNIV* and provide implementations for *finite*, *card*, (\subseteq), and ($=$) if the calling context already

provides *finite-UNIV* and *card-UNIV* instances. If we implemented the latter always via *card-UNIV-class.card-UNIV*, we would require instances of essentially all element types, i.e., a lot of instantiation proofs and – at run time – possibly slow dictionary constructions.

context
begin

qualified definition *card-UNIV'* :: 'a *card-UNIV*
where *card-UNIV'* = *Phantom*('a) *CARD*('a)

lemma *CARD-code* [*code-unfold*]:
 CARD('a) = *of-phantom* (*card-UNIV'* :: 'a *card-UNIV*)
 ⟨*proof*⟩

lemma *card-UNIV'-code* [*code*]:
 card-UNIV' = *card-UNIV*
 ⟨*proof*⟩

end

lemma *card-Compl*:
 finite A \implies *card* (– A) = *card* (*UNIV* :: 'a *set*) – *card* (A :: 'a *set*)
 ⟨*proof*⟩

context fixes *xs* :: 'a :: *finite-UNIV* *list*
begin

qualified definition *finite'* :: 'a *set* \Rightarrow *bool*
where [*simp*, *code-abbrev*]: *finite'* = *finite*

lemma *finite'-code* [*code*]:
 finite' (*set xs*) \longleftrightarrow *True*
 finite' (*List.coiset xs*) \longleftrightarrow *of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*)
 ⟨*proof*⟩

end

context fixes *xs* :: 'a :: *card-UNIV* *list*
begin

qualified definition *card'* :: 'a *set* \Rightarrow *nat*
where [*simp*, *code-abbrev*]: *card'* = *card*

lemma *card'-code* [*code*]:
 card' (*set xs*) = *length* (*remdups xs*)
 card' (*List.coiset xs*) = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*) – *length* (*remdups xs*)
 ⟨*proof*⟩ **definition** *subset'* :: 'a *set* \Rightarrow 'a *set* \Rightarrow *bool*
where [*simp*, *code-abbrev*]: *subset'* = (\subseteq)

lemma *subset'-code* [code]:
 $\text{subset}' A (\text{List.coset } ys) \longleftrightarrow (\forall y \in \text{set } ys. y \notin A)$
 $\text{subset}' (\text{set } ys) B \longleftrightarrow (\forall y \in \text{set } ys. y \in B)$
 $\text{subset}' (\text{List.coset } xs) (\text{set } ys) \longleftrightarrow (\text{let } n = \text{CARD}('a) \text{ in } n > 0 \wedge \text{card}(\text{set } (xs @ ys)) = n)$
 <proof> **definition** *eq-set* :: 'a set \Rightarrow 'a set \Rightarrow bool
 where [simp, code-abbrev]: *eq-set* = (=)

lemma *eq-set-code* [code]:
 fixes *ys*
 defines *rhs* \equiv
 let $n = \text{CARD}('a)$
 in if $n = 0$ then False else
 let $xs' = \text{remdups } xs$; $ys' = \text{remdups } ys$
 in $\text{length } xs' + \text{length } ys' = n \wedge (\forall x \in \text{set } xs'. x \notin \text{set } ys') \wedge (\forall y \in \text{set } ys'. y \notin \text{set } xs')$
 shows *eq-set* (*List.coset* *xs*) (*set* *ys*) \longleftrightarrow *rhs*
 and *eq-set* (*set* *ys*) (*List.coset* *xs*) \longleftrightarrow *rhs*
 and *eq-set* (*set* *xs*) (*set* *ys*) $\longleftrightarrow (\forall x \in \text{set } xs. x \in \text{set } ys) \wedge (\forall y \in \text{set } ys. y \in \text{set } xs)$
 and *eq-set* (*List.coset* *xs*) (*List.coset* *ys*) $\longleftrightarrow (\forall x \in \text{set } xs. x \in \text{set } ys) \wedge (\forall y \in \text{set } ys. y \in \text{set } xs)$
 <proof>
 end

Provide more informative exceptions than Match for non-rewritten cases. If generated code raises one these exceptions, then a code equation calls the mentioned operator for an element type that is not an instance of *card-UNIV* and is therefore not implemented via *card-UNIV-class.card-UNIV*. Constrain the element type with sort *card-UNIV* to change this.

lemma *card-code* [code]:
 $\text{card } (\text{set } xs) = \text{length } (\text{remdups } xs)$
 $\text{card } (\text{List.coset } xs) =$
 $\text{Code.abort } (\text{STR } "card (\text{List.coset } -) \text{ requires type class instance card-UNIV}")$
 $(\lambda -. \text{card } (\text{List.coset } xs))$
 <proof>

lemma *coset-subseteq-set-code* [code]:
 $\text{set } xs \subseteq B = \text{list-all } (\lambda x. x \in B) xs$
 $A \subseteq \text{List.coset } ys = \text{list-all } (\lambda y. y \notin A) ys$
 $\text{List.coset } xs \subseteq \text{set } ys \longleftrightarrow$
 (if $xs = [] \wedge ys = []$ then False
 else $\text{Code.abort } (\text{STR } "subset-eq (\text{List.coset } -) (\text{List.set } -) \text{ requires type class instance card-UNIV}")$
 $(\lambda -. \text{List.coset } xs \subseteq \text{set } ys))$
 <proof>

```

notepad begin — test code setup
  <proof>

end

end

```

11 Eliminating pattern matches

```

theory Case-Converter
  imports Main
begin

```

```

definition missing-pattern-match :: String.literal  $\Rightarrow$  (unit  $\Rightarrow$  'a)  $\Rightarrow$  'a where
  [code del]: missing-pattern-match m f = f ()

```

```

lemma missing-pattern-match-cong [cong]:
  m = m'  $\Rightarrow$  missing-pattern-match m f = missing-pattern-match m' f
  <proof>

```

```

lemma missing-pattern-match-code [code-unfold]:
  missing-pattern-match = Code.abort
  <proof>

```

```

  <ML>

```

```

end

```

12 Lazy types in generated code

```

theory Code-Lazy
imports Case-Converter
keywords
  code-lazy-type
  activate-lazy-type
  deactivate-lazy-type
  activate-lazy-types
  deactivate-lazy-types
  print-lazy-types :: thy-decl
begin

```

This theory and the CodeLazy tool described in [3].

It hooks into Isabelle’s code generator such that the generated code evaluates a user-specified set of type constructors lazily, even in target languages with eager evaluation. The lazy type must be algebraic, i.e., values must be built from constructors and a corresponding case operator decomposes them. Every datatype and codatatype is algebraic and thus eligible for lazification.

12.1 The type *lazy*

```

typedef 'a lazy = UNIV :: 'a set <proof>
setup-lifting type-definition-lazy
lift-definition delay :: (unit ⇒ 'a) ⇒ 'a lazy is λf. f () <proof>
lift-definition force :: 'a lazy ⇒ 'a is λx. x <proof>

code-datatype delay
lemma force-delay [code]: force (delay f) = f () <proof>
lemma delay-force: delay (λ-. force s) = s <proof>

definition termify-lazy2 :: 'a :: typerep lazy ⇒ term
  where termify-lazy2 x =
    Code-Evaluation.App (Code-Evaluation.Const (STR "Code-Lazy.delay") (TYPEREP((unit
    ⇒ 'a) ⇒ 'a lazy)))
    (Code-Evaluation.Const (STR "Pure.dummy-pattern") (TYPEREP((unit ⇒
    'a))))

definition termify-lazy ::
  (String.literal ⇒ 'typerep ⇒ 'term) ⇒
  ('term ⇒ 'term ⇒ 'term) ⇒
  (String.literal ⇒ 'typerep ⇒ 'term ⇒ 'term) ⇒
  'typerep ⇒ ('typerep ⇒ 'typerep ⇒ 'typerep) ⇒ ('typerep ⇒ 'typerep) ⇒
  ('a ⇒ 'term) ⇒ 'typerep ⇒ 'a :: typerep lazy ⇒ 'term ⇒ term
  where termify-lazy - - - - - x = termify-lazy2 x

declare [[code drop: Code-Evaluation.term-of :: - lazy ⇒ -]]

lemma term-of-lazy-code [code]:
  Code-Evaluation.term-of x ≡
    termify-lazy
      Code-Evaluation.Const Code-Evaluation.App Code-Evaluation.Abs
        TYPEREP(unit) (λT U. typerep.Typerep (STR "fun") [T, U]) (λT. type-
        rep.Typerep (STR "Code-Lazy.lazy") [T])
      Code-Evaluation.term-of TYPEREP('a) x (Code-Evaluation.Const (STR ""))
      (TYPEREP(unit))
  for x :: 'a :: {typerep, term-of} lazy
  <proof>

```

The implementations of `- lazy` using language primitives cache forced values.

Term reconstruction for `lazy` looks into the lazy value and reconstructs it to the depth it has been evaluated. This is not done for Haskell as we do not know of any portable way to inspect whether a lazy value has been evaluated to or not.

```

code-printing code-module Lazy ↪ (SML) file ~~/src/HOL/Library/Tools/lazy.ML
  for type-constructor lazy constant delay force termify-lazy
| type-constructor lazy ↪ (SML) - Lazy.lazy
| constant delay ↪ (SML) Lazy.lazy

```

```
| constant force  $\rightarrow$  (SML) Lazy.force
| constant termify-lazy  $\rightarrow$  (SML) Lazy.termify'-lazy
```

code-reserved (*SML*) *Lazy*

code-printing — For code generation within the Isabelle environment, we reuse the thread-safe implementation of lazy from `~/src/Pure/Concurrent/lazy.ML`

```
  code-module Lazy  $\rightarrow$  (Eval)  $\langle \rangle$  for constant undefined
| type-constructor lazy  $\rightarrow$  (Eval) - Lazy.lazy
| constant delay  $\rightarrow$  (Eval) Lazy.lazy
| constant force  $\rightarrow$  (Eval) Lazy.force
| code-module Termify-Lazy  $\rightarrow$  (Eval) file ~/src/HOL/Library/Tools/termify-lazy.ML
  for constant termify-lazy
| constant termify-lazy  $\rightarrow$  (Eval) Termify'-Lazy.termify'-lazy
```

code-reserved (*Eval*) *Termify-Lazy*

code-printing

```
  type-constructor lazy  $\rightarrow$  (OCaml) - Lazy.t
| constant delay  $\rightarrow$  (OCaml) Lazy.from'-fun
| constant force  $\rightarrow$  (OCaml) Lazy.force
| code-module Termify-Lazy  $\rightarrow$  (OCaml) file ~/src/HOL/Library/Tools/termify-lazy.ocaml
  for constant termify-lazy
| constant termify-lazy  $\rightarrow$  (OCaml) Termify'-Lazy.termify'-lazy
```

code-reserved (*OCaml*) *Lazy Termify-Lazy*

code-printing

```
  code-module Lazy  $\rightarrow$  (Haskell) file ~/src/HOL/Library/Tools/lazy.hs
    for type-constructor lazy constant delay force
| type-constructor lazy  $\rightarrow$  (Haskell) Lazy.Lazy -
| constant delay  $\rightarrow$  (Haskell) Lazy.delay
| constant force  $\rightarrow$  (Haskell) Lazy.force
```

code-reserved (*Haskell*) *Lazy*

code-printing

```
  code-module Lazy  $\rightarrow$  (Scala) file ~/src/HOL/Library/Tools/lazy.scala
    for type-constructor lazy constant delay force termify-lazy
| type-constructor lazy  $\rightarrow$  (Scala) Lazy.Lazy[-]
| constant delay  $\rightarrow$  (Scala) Lazy.delay
| constant force  $\rightarrow$  (Scala) Lazy.force
| constant termify-lazy  $\rightarrow$  (Scala) Lazy.termify'-lazy
```

code-reserved (*Scala*) *Lazy*

Make evaluation with the simplifier respect *delays*.

lemma *delay-lazy-cong*: *delay f = delay f* \langle *proof* \rangle
 \langle *ML* \rangle

12.2 Implementation

$\langle ML \rangle$

end

13 Test infrastructure for the code generator

```
theory Code-Test
imports Main
keywords test-code :: diag
begin
```

13.1 YXML encoding for *term*

```
datatype (plugins del: code size quickcheck) yxml-of-term = YXML
```

```
lemma yot-anything: x = (y :: yxml-of-term)
   $\langle proof \rangle$ 
```

```
definition yot-empty :: yxml-of-term where [code del]: yot-empty = YXML
```

```
definition yot-literal :: String.literal  $\Rightarrow$  yxml-of-term
```

```
  where [code del]: yot-literal - = YXML
```

```
definition yot-append :: yxml-of-term  $\Rightarrow$  yxml-of-term  $\Rightarrow$  yxml-of-term
```

```
  where [code del]: yot-append - - = YXML
```

```
definition yot-concat :: yxml-of-term list  $\Rightarrow$  yxml-of-term
```

```
  where [code del]: yot-concat - = YXML
```

Serialise *yxml-of-term* to native string of target language

```
code-printing type-constructor yxml-of-term
```

```
   $\rightarrow$  (SML) string
```

```
  and (OCaml) string
```

```
  and (Haskell) String
```

```
  and (Scala) String
```

```
| constant yot-empty
```

```
   $\rightarrow$  (SML)
```

```
  and (OCaml)
```

```
  and (Haskell)
```

```
  and (Scala)
```

```
| constant yot-literal
```

```
   $\rightarrow$  (SML) -
```

```
  and (OCaml) -
```

```
  and (Haskell) -
```

```
  and (Scala) -
```

```
| constant yot-append
```

```
   $\rightarrow$  (SML) String.concat [(-), (-)]
```

```
  and (OCaml) String.concat [(-); (-)]
```

```
  and (Haskell) infixr 5 ++
```

```
  and (Scala) infixl 5 +
```

```
| constant yot-concat
  ↪ (SML) String.concat
  and (OCaml) String.concat
  and (Haskell) Prelude.concat
  and (Scala) -.mkString()
```

Stripped-down implementations of Isabelle’s XML tree with YXML encoding as defined in `~/src/Pure/PIDE/xml.ML`, `~/src/Pure/PIDE/yxml.ML` sufficient to encode *term* as in `~/src/Pure/term_xml.ML`.

```
datatype (plugins del: code size quickcheck) xml-tree = XML-Tree
```

```
lemma xml-tree-anything: x = (y :: xml-tree)
  <proof>
```

```
context begin
  <ML>
```

```
type-synonym attributes = (String.literal × String.literal) list
type-synonym body = xml-tree list
```

```
definition Elem :: String.literal ⇒ attributes ⇒ xml-tree list ⇒ xml-tree
where [code del]: Elem - - - = XML-Tree
```

```
definition Text :: String.literal ⇒ xml-tree
where [code del]: Text - = XML-Tree
```

```
definition node :: xml-tree list ⇒ xml-tree
where node ts = Elem (STR "'") [] ts
```

```
definition tagged :: String.literal ⇒ String.literal option ⇒ xml-tree list ⇒ xml-tree
where tagged tag x ts = Elem tag (case x of None ⇒ [] | Some x' ⇒ [(STR "'0'", x')]) ts
```

```
definition list where list f xs = map (node ∘ f) xs
```

```
definition X :: yxml-of-term where X = yot-literal (STR 0x05)
definition Y :: yxml-of-term where Y = yot-literal (STR 0x06)
definition XY :: yxml-of-term where XY = yot-append X Y
definition XYX :: yxml-of-term where XYX = yot-append XY X
```

```
end
```

```
code-datatype xml.Elem xml.Text
```

```
definition yxml-string-of-xml-tree :: xml-tree ⇒ yxml-of-term ⇒ yxml-of-term
where [code del]: yxml-string-of-xml-tree - - = YXML
```

```
lemma yxml-string-of-xml-tree-code [code]:
  yxml-string-of-xml-tree (xml.Elem name atts ts) rest =
```



```

yot-append xml.XY (
  yot-append (yot-literal name) (
    foldr (λ(a, x) rest.
      yot-append xml.Y (
        yot-append (yot-literal a) (
          yot-append (yot-literal (STR "'='')) (
            yot-append (yot-literal x) rest)))) atts (
        foldr yxml-string-of-xml-tree ts (
          yot-append xml.XYX rest))))
  yxml-string-of-xml-tree (xml.Text s) rest = yot-append (yot-literal s) rest
⟨proof⟩

```

definition *yxml-string-of-body* :: *xml.body* ⇒ *yxml-of-term*
where *yxml-string-of-body* ts = foldr *yxml-string-of-xml-tree* ts *yot-empty*

Encoding *term* into XML trees as defined in `~/src/Pure/term_xml.ML`.

definition *xml-of-typ* :: *Typerep.typerep* ⇒ *xml.body*
where [code del]: *xml-of-typ* - = [XML-Tree]

definition *xml-of-term* :: *Code-Evaluation.term* ⇒ *xml.body*
where [code del]: *xml-of-term* - = [XML-Tree]

lemma *xml-of-typ-code* [code]:
xml-of-typ (*typerep.TypeRep* t args) = [xml.tagged (STR "'0'") (Some t) (xml.list
xml-of-typ args)]
⟨proof⟩

lemma *xml-of-term-code* [code]:
xml-of-term (*Code-Evaluation.Const* x ty) = [xml.tagged (STR "'0'") (Some x)
(*xml-of-typ* ty)]
xml-of-term (*Code-Evaluation.App* t1 t2) = [xml.tagged (STR "'5'") None [xml.node
(*xml-of-term* t1), xml.node (*xml-of-term* t2)]]
xml-of-term (*Code-Evaluation.Abs* x ty t) = [xml.tagged (STR "'4'") (Some x)
[xml.node (*xml-of-typ* ty), xml.node (*xml-of-term* t)]]
— FIXME: *Code-Evaluation.Free* is used only in *HOL.Quickcheck-Narrowing* to
represent uninstantiated parameters in constructors. Here, we always translate
them to **Free** variables.
xml-of-term (*Code-Evaluation.Free* x ty) = [xml.tagged (STR "'1'") (Some x)
(*xml-of-typ* ty)]
⟨proof⟩

definition *yxml-string-of-term* :: *Code-Evaluation.term* ⇒ *yxml-of-term*
where *yxml-string-of-term* = *yxml-string-of-body* ∘ *xml-of-term*

13.2 Test engine and drivers

⟨ML⟩

end

14 A combinator to build partial equivalence relations from a predicate and an equivalence relation

```
theory Combine-PER
  imports Main
begin
```

```
unbundle lattice-syntax
```

```
definition combine-per :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  where combine-per P R = ( $\lambda x y. P x \wedge P y$ )  $\sqcap$  R
```

```
lemma combine-per-simp [simp]:
  combine-per P R x y  $\longleftrightarrow$  P x  $\wedge$  P y  $\wedge$  x  $\approx$  y for R (infixl  $\langle \approx \rangle$  50)
  <proof>
```

```
lemma combine-per-top [simp]: combine-per  $\top$  R = R
  <proof>
```

```
lemma combine-per-eq [simp]: combine-per P HOL.eq = HOL.eq  $\sqcap$  ( $\lambda x y. P x$ )
  <proof>
```

```
lemma symp-combine-per: symp R  $\Longrightarrow$  symp (combine-per P R)
  <proof>
```

```
lemma transp-combine-per: transp R  $\Longrightarrow$  transp (combine-per P R)
  <proof>
```

```
lemma combine-perI: P x  $\Longrightarrow$  P y  $\Longrightarrow$  x  $\approx$  y  $\Longrightarrow$  combine-per P R x y for R
  (infixl  $\langle \approx \rangle$  50)
  <proof>
```

```
lemma symp-combine-per-symp: symp R  $\Longrightarrow$  symp (combine-per P R)
  <proof>
```

```
lemma transp-combine-per-transp: transp R  $\Longrightarrow$  transp (combine-per P R)
  <proof>
```

```
lemma equivp-combine-per-part-equivp [intro?]:
  fixes R (infixl  $\langle \approx \rangle$  50)
  assumes  $\exists x. P x$  and equivp R
  shows part-equivp (combine-per P R)
  <proof>
```

```
end
```

15 Formalisation of chain-complete partial orders, continuity and admissibility

theory *Complete-Partial-Order2*

imports *Main*

begin

unbundle *lattice-syntax*

lemma *chain-transfer* [*transfer-rule*]:

includes *lifting-syntax*

shows $((A \text{ ===> } A \text{ ===> } (=)) \text{ ===> } \text{rel-set } A \text{ ===> } (=)) \text{ Complete-Partial-Order.chain}$
Complete-Partial-Order.chain
 $\langle \text{proof} \rangle$

lemma *linorder-chain* [*simp, intro!*]:

fixes $Y :: - :: \text{linorder set}$

shows *Complete-Partial-Order.chain* $(\leq) \ Y$

$\langle \text{proof} \rangle$

lemma *fun-lub-apply*: $\bigwedge \text{Sup. fun-lub Sup } Y \ x = \text{Sup } ((\lambda f. f \ x) \text{ ‘ } Y)$

$\langle \text{proof} \rangle$

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* $\{\} = (\lambda -. \text{lub } \{\})$

$\langle \text{proof} \rangle$

lemma *chain-fun-ordD*:

assumes *Complete-Partial-Order.chain* $(\text{fun-ord } le) \ Y$

shows *Complete-Partial-Order.chain* $le \ ((\lambda f. f \ x) \text{ ‘ } Y)$

$\langle \text{proof} \rangle$

lemma *chain-Diff*:

Complete-Partial-Order.chain $\text{ord } A$

$\implies \text{Complete-Partial-Order.chain } \text{ord } (A - B)$

$\langle \text{proof} \rangle$

lemma *chain-rel-prodD1*:

Complete-Partial-Order.chain $(\text{rel-prod } \text{orda } \text{ordb}) \ Y$

$\implies \text{Complete-Partial-Order.chain } \text{orda } (\text{fst} \text{ ‘ } Y)$

$\langle \text{proof} \rangle$

lemma *chain-rel-prodD2*:

Complete-Partial-Order.chain $(\text{rel-prod } \text{orda } \text{ordb}) \ Y$

$\implies \text{Complete-Partial-Order.chain } \text{ordb } (\text{snd} \text{ ‘ } Y)$

$\langle \text{proof} \rangle$

context *ccpo* **begin**

lemma *ccpo-fun*: *class.ccpo* (*fun-lub* *Sup*) (*fun-ord* (\leq)) (*mk-less* (*fun-ord* (\leq)))
 ⟨*proof*⟩

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain* (\leq) *Y* \implies *Sup* *Y* \leq *x*
 $\longleftrightarrow (\forall y \in Y. y \leq x)$
 ⟨*proof*⟩

lemma *Sup-minus-bot*:
assumes *chain*: *Complete-Partial-Order.chain* (\leq) *A*
shows $\sqcup (A - \{\sqcup \{\}\}) = \sqcup A$
 (**is** ?*lhs* = ?*rhs*)
 ⟨*proof*⟩

lemma *mono-lub*:
fixes *le-b* (**infix** \sqsubseteq 60) **and** *Y* *f*
assumes *chain*: *Complete-Partial-Order.chain* (*fun-ord* (\leq)) *Y*
and *mono*: $\bigwedge f. f \in Y \implies \text{monotone } \text{le-b } (\leq) f$
shows *monotone* (\sqsubseteq) (\leq) (*fun-lub* *Sup* *Y*)
 ⟨*proof*⟩

context
fixes *le-b* (**infix** \sqsubseteq 60) **and** *Y* *f*
assumes *chain*: *Complete-Partial-Order.chain* *le-b* *Y*
and *mono1*: $\bigwedge y. y \in Y \implies \text{monotone } \text{le-b } (\leq) (\lambda x. f x y)$
and *mono2*: $\bigwedge x a b. \llbracket x \in Y; a \sqsubseteq b; a \in Y; b \in Y \rrbracket \implies f x a \leq f x b$
begin

lemma *Sup-mono*:
assumes *le*: $x \sqsubseteq y$ **and** *x*: $x \in Y$ **and** *y*: $y \in Y$
shows $\sqcup (f x \text{ ‘ } Y) \leq \sqcup (f y \text{ ‘ } Y)$ (**is** - \leq ?*rhs*)
 ⟨*proof*⟩

lemma *diag-Sup*: $\sqcup ((\lambda x. \sqcup (f x \text{ ‘ } Y)) \text{ ‘ } Y) = \sqcup ((\lambda x. f x x) \text{ ‘ } Y)$ (**is** ?*lhs* = ?*rhs*)
 ⟨*proof*⟩

end

lemma *Sup-image-mono-le*:
fixes *le-b* (**infix** \sqsubseteq 60) **and** *Sup-b* ($\langle \bigvee \rangle$)
assumes *ccpo*: *class.ccpo* *Sup-b* (\sqsubseteq) *lt-b*
assumes *chain*: *Complete-Partial-Order.chain* (\sqsubseteq) *Y*
and *mono*: $\bigwedge x y. \llbracket x \sqsubseteq y; x \in Y \rrbracket \implies f x \leq f y$
shows *Sup* (*f* ‘ *Y*) $\leq f (\bigvee Y)$
 ⟨*proof*⟩

lemma *swap-Sup*:
fixes *le-b* (**infix** \sqsubseteq 60)
assumes *Y*: *Complete-Partial-Order.chain* (\sqsubseteq) *Y*

and Z : *Complete-Partial-Order.chain* (*fun-ord* (\leq)) Z
and mono : $\bigwedge f. f \in Z \implies \text{monotone } (\sqsubseteq) (\leq) f$
shows $\bigsqcup ((\lambda x. \bigsqcup (x \text{ ' } Y)) \text{ ' } Z) = \bigsqcup ((\lambda x. \bigsqcup ((\lambda f. f x) \text{ ' } Z)) \text{ ' } Y)$
(is $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemma *fixp-mono*:
assumes fg : *fun-ord* (\leq) $f g$
and f : *monotone* (\leq) (\leq) f
and g : *monotone* (\leq) (\leq) g
shows *ccpo-class.fixp* $f \leq \text{ccpo-class.fixp } g$
 $\langle \text{proof} \rangle$

context **fixes** *ordb* :: $'b \Rightarrow 'b \Rightarrow \text{bool}$ (**infix** $\langle \sqsubseteq \rangle$ 60) **begin**

lemma *iterates-mono*:
assumes f : $f \in \text{ccpo.iterates } (\text{fun-lub } \text{Sup}) (\text{fun-ord } (\leq)) F$
and mono : $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \implies \text{monotone } (\sqsubseteq) (\leq) (F f)$
shows *monotone* (\sqsubseteq) (\leq) f
 $\langle \text{proof} \rangle$

lemma *fixp-preserves-mono*:
assumes mono : $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. F f x)$
and mono2 : $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \implies \text{monotone } (\sqsubseteq) (\leq) (F f)$
shows *monotone* (\sqsubseteq) (\leq) (*ccpo.fixp* (*fun-lub* *Sup*) (*fun-ord* (\leq)) F)
(is *monotone* - - *?fixp*)
 $\langle \text{proof} \rangle$

end

end

lemma *monotone2monotone*:
assumes 2 : $\bigwedge x. \text{monotone } \text{ordb } \text{ordc } (\lambda y. f x y)$
and t : *monotone* *orda* *ordb* ($\lambda x. t x$)
and 1 : $\bigwedge y. \text{monotone } \text{orda } \text{ordc } (\lambda x. f x y)$
and *trans*: *transp* *ordc*
shows *monotone* *orda* *ordc* ($\lambda x. f x (t x)$)
 $\langle \text{proof} \rangle$

15.1 Continuity

definition *cont* :: $('a \text{ set} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \text{ set} \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$

where

$\text{cont } \text{luba } \text{orda } \text{lubb } \text{ordb } f \longleftrightarrow$
 $(\forall Y. \text{Complete-Partial-Order.chain } \text{orda } Y \longrightarrow Y \neq \{\} \longrightarrow f (\text{luba } Y) = \text{lubb } (f \text{ ' } Y))$

definition $mcont :: ('a \text{ set} \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \text{ set} \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$

where

$mcont \text{ luba } orda \text{ lubb } ordb \text{ } f \longleftrightarrow$
 $\text{monotone } orda \text{ } ordb \text{ } f \wedge \text{cont } luba \text{ } orda \text{ lubb } ordb \text{ } f$

15.1.1 Theorem collection *cont-intro*

named-theorems *cont-intro continuity and admissibility intro rules*
 $\langle ML \rangle$

lemmas [*cont-intro*] =

call-mono
let-mono
if-mono
option.const-mono
tailrec.const-mono
bind-mono

experiment begin

The following proof by simplification diverges if variables are not handled properly.

lemma $(\bigwedge f. \text{monotone } R \text{ } S \text{ } f \Longrightarrow \text{thesis}) \Longrightarrow \text{monotone } R \text{ } S \text{ } g \Longrightarrow \text{thesis}$
 $\langle \text{proof} \rangle$

end

declare *if-mono[simp]*

lemma *monotone-id'* [*cont-intro*]: $\text{monotone } ord \text{ } ord \text{ } (\lambda x. x)$
 $\langle \text{proof} \rangle$

lemma *monotone-applyI*:

$\text{monotone } orda \text{ } ordb \text{ } F \Longrightarrow \text{monotone } (\text{fun-ord } orda) \text{ } ordb \text{ } (\lambda f. F \text{ } (f \text{ } x))$
 $\langle \text{proof} \rangle$

lemma *monotone-if-fun* [*partial-function-mono*]:

$\llbracket \text{monotone } (\text{fun-ord } orda) \text{ } (\text{fun-ord } ordb) \text{ } F; \text{monotone } (\text{fun-ord } orda) \text{ } (\text{fun-ord } ordb) \text{ } G \rrbracket$
 $\Longrightarrow \text{monotone } (\text{fun-ord } orda) \text{ } (\text{fun-ord } ordb) \text{ } (\lambda f \text{ } n. \text{if } c \text{ } n \text{ then } F \text{ } f \text{ } n \text{ else } G \text{ } f \text{ } n)$
 $\langle \text{proof} \rangle$

lemma *monotone-fun-apply-fun* [*partial-function-mono*]:

$\text{monotone } (\text{fun-ord } (\text{fun-ord } ord)) \text{ } (\text{fun-ord } ord) \text{ } (\lambda f \text{ } n. f \text{ } t \text{ } (g \text{ } n))$
 $\langle \text{proof} \rangle$

lemma *monotone-fun-ord-apply*:

$\text{monotone } orda \text{ } (\text{fun-ord } ordb) \text{ } f \longleftrightarrow (\forall x. \text{monotone } orda \text{ } ordb \text{ } (\lambda y. f \text{ } y \text{ } x))$

```

  <proof>

context preorder begin

declare transp-on-le[cont-intro]

lemma monotone-const [simp, cont-intro]: monotone ord ( $\leq$ ) ( $\lambda\cdot$ . c)
  <proof>

end

lemma transp-le [cont-intro, simp]:
  class.preorder ord (mk-less ord)  $\implies$  transp ord
  <proof>

context partial-function-definitions begin

declare const-mono [cont-intro, simp]

lemma transp-le [cont-intro, simp]: transp leq
  <proof>

lemma preorder [cont-intro, simp]: class.preorder leq (mk-less leq)
  <proof>

declare ccpo[cont-intro, simp]

end

lemma contI [intro?]:
  ( $\bigwedge Y$ .  $\llbracket \text{Complete-Partial-Order.chain ord } Y; Y \neq \{\} \rrbracket \implies f \text{ (luba } Y) = \text{lubb}$ 
  ( $f \text{ ' } Y$ ))
   $\implies \text{cont luba ord } \text{lubb ordb } f$ 
  <proof>

lemma contD:
   $\llbracket \text{cont luba ord } \text{lubb ordb } f; \text{Complete-Partial-Order.chain ord } Y; Y \neq \{\} \rrbracket$ 
   $\implies f \text{ (luba } Y) = \text{lubb (f ' } Y)$ 
  <proof>

lemma cont-id [simp, cont-intro]:  $\bigwedge \text{Sup. cont Sup ord Sup ord id}$ 
  <proof>

lemma cont-id' [simp, cont-intro]:  $\bigwedge \text{Sup. cont Sup ord Sup ord } (\lambda x. x)$ 
  <proof>

lemma cont-applyI [cont-intro]:
  assumes cont: cont luba ord lubb ordb g
  shows cont (fun-lub luba) (fun-ord orda) lubb ordb ( $\lambda f. g \text{ (f } x)$ )

```

$\langle \text{proof} \rangle$

lemma *call-cont*: $\text{cont } (\text{fun-lub lub}) (\text{fun-ord ord}) \text{ lub ord } (\lambda f. f t)$
 $\langle \text{proof} \rangle$

lemma *cont-if* [*cont-intro*]:
 $\llbracket \text{cont luba orda lubb ordb } f; \text{cont luba orda lubb ordb } g \rrbracket$
 $\implies \text{cont luba orda lubb ordb } (\lambda x. \text{if } c \text{ then } f x \text{ else } g x)$
 $\langle \text{proof} \rangle$

lemma *mcontI* [*intro?*]:
 $\llbracket \text{monotone orda ordb } f; \text{cont luba orda lubb ordb } f \rrbracket \implies \text{mcont luba orda lubb ordb } f$
 $\langle \text{proof} \rangle$

lemma *mcont-mono*: $\text{mcont luba orda lubb ordb } f \implies \text{monotone orda ordb } f$
 $\langle \text{proof} \rangle$

lemma *mcont-cont* [*simp*]: $\text{mcont luba orda lubb ordb } f \implies \text{cont luba orda lubb ordb } f$
 $\langle \text{proof} \rangle$

lemma *mcont-monoD*:
 $\llbracket \text{mcont luba orda lubb ordb } f; \text{orda } x y \rrbracket \implies \text{ordb } (f x) (f y)$
 $\langle \text{proof} \rangle$

lemma *mcont-contD*:
 $\llbracket \text{mcont luba orda lubb ordb } f; \text{Complete-Partial-Order.chain orda } Y; Y \neq \{\} \rrbracket$
 $\implies f (\text{luba } Y) = \text{lubb } (f ' Y)$
 $\langle \text{proof} \rangle$

lemma *mcont-call* [*cont-intro*, *simp*]:
 $\text{mcont } (\text{fun-lub lub}) (\text{fun-ord ord}) \text{ lub ord } (\lambda f. f t)$
 $\langle \text{proof} \rangle$

lemma *mcont-id'* [*cont-intro*, *simp*]: $\text{mcont lub ord lub ord } (\lambda x. x)$
 $\langle \text{proof} \rangle$

lemma *mcont-applyI*:
 $\text{mcont luba orda lubb ordb } (\lambda x. F x) \implies \text{mcont } (\text{fun-lub luba}) (\text{fun-ord orda}) \text{ lubb ordb } (\lambda f. F (f x))$
 $\langle \text{proof} \rangle$

lemma *mcont-if* [*cont-intro*, *simp*]:
 $\llbracket \text{mcont luba orda lubb ordb } (\lambda x. f x); \text{mcont luba orda lubb ordb } (\lambda x. g x) \rrbracket$
 $\implies \text{mcont luba orda lubb ordb } (\lambda x. \text{if } c \text{ then } f x \text{ else } g x)$
 $\langle \text{proof} \rangle$

lemma *cont-fun-lub-apply*:

cont luba orda (fun-lub lubb) (fun-ord ordb) f \longleftrightarrow $(\forall x. \text{cont luba orda lubb ordb } (\lambda y. f y x))$
 ⟨proof⟩

lemma *mcont-fun-lub-apply*:

mcont luba orda (fun-lub lubb) (fun-ord ordb) f \longleftrightarrow $(\forall x. \text{mcont luba orda lubb ordb } (\lambda y. f y x))$
 ⟨proof⟩

context *ccpo* **begin**

lemma *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup* (\leq) $(\lambda x. c)$
 ⟨proof⟩

lemma *mcont-const* [*cont-intro*, *simp*]:

mcont luba orda Sup (\leq) $(\lambda x. c)$
 ⟨proof⟩

lemma *cont-apply*:

assumes 2: $\bigwedge x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)$
and *t*: *cont luba orda lubb ordb* $(\lambda x. t x)$
and 1: $\bigwedge y. \text{cont luba orda Sup } (\leq) (\lambda x. f x y)$
and *mono*: *monotone orda ordb* $(\lambda x. t x)$
and *mono2*: $\bigwedge x. \text{monotone ordb } (\leq) (\lambda y. f x y)$
and *mono1*: $\bigwedge y. \text{monotone orda } (\leq) (\lambda x. f x y)$
shows *cont luba orda Sup* (\leq) $(\lambda x. f x (t x))$
 ⟨proof⟩

lemma *mcont2mcont'*:

$\llbracket \bigwedge x. \text{mcont lub' ord' Sup } (\leq) (\lambda y. f x y);$
 $\bigwedge y. \text{mcont lub ord Sup } (\leq) (\lambda x. f x y);$
 $\text{mcont lub ord lub' ord' } (\lambda y. t y) \rrbracket$
 $\implies \text{mcont lub ord Sup } (\leq) (\lambda x. f x (t x))$
 ⟨proof⟩

lemma *mcont2mcont*:

$\llbracket \text{mcont lub' ord' Sup } (\leq) (\lambda x. f x); \text{mcont lub ord lub' ord' } (\lambda x. t x) \rrbracket$
 $\implies \text{mcont lub ord Sup } (\leq) (\lambda x. f (t x))$
 ⟨proof⟩

context

fixes *ord* :: 'b \Rightarrow 'b \Rightarrow bool (**infix** \trianglelefteq 60)
and *lub* :: 'b set \Rightarrow 'b ($\triangleleft \bigvee \triangleright$)

begin

lemma *cont-fun-lub-Sup*:

assumes *chainM*: *Complete-Partial-Order.chain* $(\text{fun-ord } (\leq)) M$
and *mcont* [*rule-format*]: $\forall f \in M. \text{mcont lub } (\trianglelefteq) \text{ Sup } (\leq) f$
shows *cont lub* $(\trianglelefteq) \text{ Sup } (\leq) (\text{fun-lub Sup } M)$

$\langle \text{proof} \rangle$

lemma *mcont-fun-lub-Sup*:

[[*Complete-Partial-Order.chain* (*fun-ord* (\leq)) *M*;
 $\forall f \in M. \text{mcont lub ord Sup } (\leq) f$]]
 $\implies \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (\text{fun-lub Sup } M)$

$\langle \text{proof} \rangle$

lemma *iterates-mcont*:

assumes *f*: $f \in \text{ccpo.iterates } (\text{fun-lub Sup}) (\text{fun-ord } (\leq)) F$
and *mono*: $\bigwedge f. \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) f \implies \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (F f)$
shows $\text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) f$

$\langle \text{proof} \rangle$

lemma *fixp-preserves-mcont*:

assumes *mono*: $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. F f x)$
and *mcont*: $\bigwedge f. \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) f \implies \text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (F f)$
shows $\text{mcont lub } (\sqsubseteq) \text{ Sup } (\leq) (\text{ccpo.fixp } (\text{fun-lub Sup}) (\text{fun-ord } (\leq)) F)$
(is mcont - - - ?fixp)

$\langle \text{proof} \rangle$

end

context

fixes *F* :: $'c \Rightarrow 'c$ **and** *U* :: $'c \Rightarrow 'b \Rightarrow 'a$ **and** *C* :: $('b \Rightarrow 'a) \Rightarrow 'c$ **and** *f*
assumes *mono*: $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. U (F (C f)) x)$
and *eq*: $f \equiv C (\text{ccpo.fixp } (\text{fun-lub Sup}) (\text{fun-ord } (\leq)) (\lambda f. U (F (C f))))$
and *inverse*: $\bigwedge f. U (C f) = f$

begin

lemma *fixp-preserves-mono-uc*:

assumes *mono2*: $\bigwedge f. \text{monotone ord } (\leq) (U f) \implies \text{monotone ord } (\leq) (U (F f))$
shows $\text{monotone ord } (\leq) (U f)$

$\langle \text{proof} \rangle$

lemma *fixp-preserves-mcont-uc*:

assumes *mcont*: $\bigwedge f. \text{mcont lubb ordb Sup } (\leq) (U f) \implies \text{mcont lubb ordb Sup } (\leq) (U (F f))$
shows $\text{mcont lubb ordb Sup } (\leq) (U f)$

$\langle \text{proof} \rangle$

end

lemmas *fixp-preserves-mono1* = *fixp-preserves-mono-uc*[of $\lambda x. x - \lambda x. x$, *OF* - - *refl*]

lemmas *fixp-preserves-mono2* =

fixp-preserves-mono-uc[of *case-prod* - *curry*, *unfolded case-prod-curry* *curry-case-prod*, *OF* - - *refl*]

lemmas *fixp-preserves-mono3* =

fixp-preserves-mono-uc[of $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$, unfolded *case-prod-curry curry-case-prod*, *OF - - refl*]

lemmas *fixp-preserves-mono4* =

fixp-preserves-mono-uc[of $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$, unfolded *case-prod-curry curry-case-prod*, *OF - - refl*]

lemmas *fixp-preserves-mcont1* = *fixp-preserves-mcont-uc*[of $\lambda x. x - \lambda x. x$, *OF - - refl*]

lemmas *fixp-preserves-mcont2* =

fixp-preserves-mcont-uc[of *case-prod - curry*, unfolded *case-prod-curry curry-case-prod*, *OF - - refl*]

lemmas *fixp-preserves-mcont3* =

fixp-preserves-mcont-uc[of $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$, unfolded *case-prod-curry curry-case-prod*, *OF - - refl*]

lemmas *fixp-preserves-mcont4* =

fixp-preserves-mcont-uc[of $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$, unfolded *case-prod-curry curry-case-prod*, *OF - - refl*]

end

lemma (in *preorder*) *monotone-if-bot*:

fixes *bot*

assumes *mono*: $\bigwedge x y. \llbracket x \leq y; \neg (x \leq \text{bound}) \rrbracket \implies \text{ord } (f x) (f y)$

and *bot*: $\bigwedge x. \neg x \leq \text{bound} \implies \text{ord } \text{bot } (f x) \text{ ord } \text{bot } \text{bot}$

shows *monotone* (\leq) *ord* ($\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x$)

<proof>

lemma (in *ccpo*) *mcont-if-bot*:

fixes *bot* **and** *lub* ($\langle \bigvee \rangle$) **and** *ord* (**infix** $\langle \sqsubseteq \rangle$ 60)

assumes *ccpo*: *class.ccpo* *lub* (\sqsubseteq) *lt*

and *mono*: $\bigwedge x y. \llbracket x \leq y; \neg x \leq \text{bound} \rrbracket \implies f x \sqsubseteq f y$

and *cont*: $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } (\leq) Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg x \leq \text{bound} \rrbracket \implies f (\bigsqcup Y) = \bigvee (f \restriction Y)$

and *bot*: $\bigwedge x. \neg x \leq \text{bound} \implies \text{bot} \sqsubseteq f x$

shows *mcont* *Sup* (\leq) *lub* (\sqsubseteq) ($\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x$) (**is** *mcont* - - ?g)

<proof>

context *partial-function-definitions* **begin**

lemma *mcont-const* [*cont-intro*, *simp*]:

mcont *luba* *orda* *lub* *leq* ($\lambda x. c$)

<proof>

lemmas [*cont-intro*, *simp*] =

ccpo.cont-const[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *mono2mono*:

assumes *monotone* *ordb* *leq* ($\lambda y. f y$) *monotone* *orda* *ordb* ($\lambda x. t x$)

shows *monotone orda leq* ($\lambda x. f \ (t \ x)$)
 <proof>

lemmas *mcont2mcont'* = *ccpo.mcont2mcont'*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *mcont2mcont* = *ccpo.mcont2mcont*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono1* = *ccpo.fixp-preserves-mono1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono2* = *ccpo.fixp-preserves-mono2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono3* = *ccpo.fixp-preserves-mono3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono4* = *ccpo.fixp-preserves-mono4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont1* = *ccpo.fixp-preserves-mcont1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont2* = *ccpo.fixp-preserves-mcont2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont3* = *ccpo.fixp-preserves-mcont3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont4* = *ccpo.fixp-preserves-mcont4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *monotone-if-bot*:

fixes *bot*

assumes *g*: $\bigwedge x. g \ x = (if \ leq \ x \ bound \ then \ bot \ else \ f \ x)$

and *mono*: $\bigwedge x \ y. \llbracket leq \ x \ y; \neg leq \ x \ bound \rrbracket \implies ord \ (f \ x) \ (f \ y)$

and *bot*: $\bigwedge x. \neg leq \ x \ bound \implies ord \ bot \ (f \ x) \ ord \ bot \ bot$

shows *monotone leq ord g*

<proof>

lemma *mcont-if-bot*:

fixes *bot*

assumes *ccpo*: *class.ccpo lub' ord* (*mk-less ord*)

and *bot*: $\bigwedge x. \neg leq \ x \ bound \implies ord \ bot \ (f \ x)$

and *g*: $\bigwedge x. g \ x = (if \ leq \ x \ bound \ then \ bot \ else \ f \ x)$

and *mono*: $\bigwedge x \ y. \llbracket leq \ x \ y; \neg leq \ x \ bound \rrbracket \implies ord \ (f \ x) \ (f \ y)$

and *cont*: $\bigwedge Y. \llbracket Complete-Partial-Order.chain \ leq \ Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg leq \ x \ bound \rrbracket \implies f \ (lub \ Y) = lub' \ (f \ ' \ Y)$

shows *mcont lub leq lub' ord g*

<proof>

end

15.2 Admissibility

lemma *admissible-subst*:

assumes *adm*: *ccpo.admissible luba orda* ($\lambda x. P \ x$)

and *mcont*: *mcont lubb ordb luba orda f*
shows *ccpo.admissible lubb ordb* ($\lambda x. P (f x)$)
<proof>

lemmas [*simp, cont-intro*] =
admissible-all
admissible-ball
admissible-const
admissible-conj

lemma *admissible-disj'* [*simp, cont-intro*]:
 $\llbracket \text{class.ccpo lub ord (mk-less ord)}; \text{ccpo.admissible lub ord } P; \text{ccpo.admissible lub ord } Q \rrbracket$
 $\implies \text{ccpo.admissible lub ord } (\lambda x. P x \vee Q x)$
<proof>

lemma *admissible-imp'* [*cont-intro*]:
 $\llbracket \text{class.ccpo lub ord (mk-less ord)};$
 $\text{ccpo.admissible lub ord } (\lambda x. \neg P x);$
 $\text{ccpo.admissible lub ord } (\lambda x. Q x) \rrbracket$
 $\implies \text{ccpo.admissible lub ord } (\lambda x. P x \longrightarrow Q x)$
<proof>

lemma *admissible-imp* [*cont-intro*]:
 $(Q \implies \text{ccpo.admissible lub ord } (\lambda x. P x))$
 $\implies \text{ccpo.admissible lub ord } (\lambda x. Q \longrightarrow P x)$
<proof>

lemma *admissible-not-mem'* [*THEN admissible-subst, cont-intro, simp*]:
shows *admissible-not-mem*: *ccpo.admissible Union* (\subseteq) ($\lambda A. x \notin A$)
<proof>

lemma *admissible-eqI*:
assumes *f*: *cont luba orda lub ord* ($\lambda x. f x$)
and *g*: *cont luba orda lub ord* ($\lambda x. g x$)
shows *ccpo.admissible luba orda* ($\lambda x. f x = g x$)
<proof>

corollary *admissible-eq-mcontI* [*cont-intro*]:
 $\llbracket \text{mcont luba orda lub ord } (\lambda x. f x);$
 $\text{mcont luba orda lub ord } (\lambda x. g x) \rrbracket$
 $\implies \text{ccpo.admissible luba orda } (\lambda x. f x = g x)$
<proof>

lemma *admissible-iff* [*cont-intro, simp*]:
 $\llbracket \text{ccpo.admissible lub ord } (\lambda x. P x \longrightarrow Q x); \text{ccpo.admissible lub ord } (\lambda x. Q x \longrightarrow P x) \rrbracket$
 $\implies \text{ccpo.admissible lub ord } (\lambda x. P x \longleftrightarrow Q x)$
<proof>

context *ccpo* **begin**

lemma *admissible-leI*:

assumes *f*: *mcont luba orda Sup* (\leq) ($\lambda x. f\ x$)
and *g*: *mcont luba orda Sup* (\leq) ($\lambda x. g\ x$)
shows *ccpo.admissible luba orda* ($\lambda x. f\ x \leq g\ x$)

$\langle proof \rangle$

end

lemma *admissible-leI*:

fixes *ord* (**infix** $\langle \sqsubseteq \rangle$ 60) **and** *lub* ($\langle \bigvee \rangle$)
assumes *class.ccpo lub* (\sqsubseteq) (*mk-less* (\sqsubseteq))
and *mcont luba orda lub* (\sqsubseteq) ($\lambda x. f\ x$)
and *mcont luba orda lub* (\sqsubseteq) ($\lambda x. g\ x$)
shows *ccpo.admissible luba orda* ($\lambda x. f\ x \sqsubseteq g\ x$)

$\langle proof \rangle$

declare *ccpo-class.admissible-leI* [*cont-intro*]

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible Sup* (\leq) ($\lambda x. \neg (\leq) x\ y$)

$\langle proof \rangle$

end

lemma (**in** *preorder*) *preorder* [*cont-intro*, *simp*]: *class.preorder* (\leq) (*mk-less* (\leq))

$\langle proof \rangle$

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] =

admissible-leI [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]
ccpo.admissible-not-below [*THEN admissible-subst*, *OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]

end

$\langle ML \rangle$

inductive *compact* :: (*'a set* \Rightarrow *'a*) \Rightarrow (*'a* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow *'a* \Rightarrow *bool*

for *lub ord x*

where *compact*:

\llbracket *ccpo.admissible lub ord* ($\lambda y. \neg \text{ord } x\ y$);
ccpo.admissible lub ord ($\lambda y. x \neq y$) \rrbracket
 \Rightarrow *compact lub ord x*

$\langle ML \rangle$

context *ccpo* **begin**

lemma *compactI*:

assumes *ccpo.admissible* *Sup* (\leq) $(\lambda y. \neg x \leq y)$

shows *ccpo.compact* *Sup* (\leq) *x*

$\langle proof \rangle$

lemma *compact-bot*:

assumes $x = \text{Sup } \{\}$

shows *ccpo.compact* *Sup* (\leq) *x*

$\langle proof \rangle$

end

lemma *admissible-compact-neg'* [*THEN* *admissible-subst*, *cont-intro*, *simp*]:

shows *admissible-compact-neg*: *ccpo.compact* *lub* *ord* *k* \implies *ccpo.admissible* *lub* *ord* $(\lambda x. k \neq x)$

$\langle proof \rangle$

lemma *admissible-neg-compact'* [*THEN* *admissible-subst*, *cont-intro*, *simp*]:

shows *admissible-neg-compact*: *ccpo.compact* *lub* *ord* *k* \implies *ccpo.admissible* *lub* *ord* $(\lambda x. x \neq k)$

$\langle proof \rangle$

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] = *ccpo.compact-bot*[*OF* *Partial-Function.ccpo*[*OF* *partial-function-definitions-axioms*]]

end

context *ccpo* **begin**

lemma *fixp-strong-induct*:

assumes [*cont-intro*]: *ccpo.admissible* *Sup* (\leq) *P*

and *mono*: *monotone* (\leq) (\leq) *f*

and *bot*: $P (\sqcup \{\})$

and *step*: $\bigwedge x. \llbracket x \leq \text{ccpo-class.fixp } f; P x \rrbracket \implies P (f x)$

shows $P (\text{ccpo-class.fixp } f)$

$\langle proof \rangle$

end

context *partial-function-definitions* **begin**

lemma *fixp-strong-induct-uc*:

fixes $F :: 'c \Rightarrow 'c$

```

and  $U :: 'c \Rightarrow 'b \Rightarrow 'a$ 
and  $C :: ('b \Rightarrow 'a) \Rightarrow 'c$ 
and  $P :: ('b \Rightarrow 'a) \Rightarrow \text{bool}$ 
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:  $\bigwedge f'. \llbracket P (U f'); \text{le-fun } (U f') (U f) \rrbracket \Longrightarrow P (U (F f'))$ 
shows  $P (U f)$ 
 $\langle \text{proof} \rangle$ 

end

```

15.3 (=) as order

```

definition lub-singleton :: ('a set  $\Rightarrow$  'a)  $\Rightarrow$  bool
where lub-singleton lub  $\longleftrightarrow (\forall a. \text{lub } \{a\} = a)$ 

```

```

definition the-Sup :: 'a set  $\Rightarrow$  'a
where the-Sup A = (THE a. a  $\in$  A)

```

```

lemma lub-singleton-the-Sup [cont-intro, simp]: lub-singleton the-Sup
 $\langle \text{proof} \rangle$ 

```

```

lemma (in ccpo) lub-singleton: lub-singleton Sup
 $\langle \text{proof} \rangle$ 

```

```

lemma (in partial-function-definitions) lub-singleton [cont-intro, simp]: lub-singleton
lub
 $\langle \text{proof} \rangle$ 

```

```

lemma preorder-eq [cont-intro, simp]:
class.preorder (=) (mk-less (=))
 $\langle \text{proof} \rangle$ 

```

```

lemma monotone-eqI [cont-intro]:
assumes class.preorder ord (mk-less ord)
shows monotone (=) ord f
 $\langle \text{proof} \rangle$ 

```

```

lemma cont-eqI [cont-intro]:
fixes f :: 'a  $\Rightarrow$  'b
assumes lub-singleton lub
shows cont the-Sup (=) lub ord f
 $\langle \text{proof} \rangle$ 

```

```

lemma mcont-eqI [cont-intro, simp]:
 $\llbracket \text{class.preorder } \text{ord } (\text{mk-less } \text{ord}); \text{lub-singleton } \text{lub} \rrbracket$ 

```


$\Rightarrow mcont\ the-Sup\ (=)\ lub\ ord\ f$
 $\langle proof \rangle$

15.4 ccpo for products

definition $prod-lub :: ('a\ set \Rightarrow 'a) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('a \times 'b)\ set \Rightarrow 'a \times 'b$
where $prod-lub\ Sup-a\ Sup-b\ Y = (Sup-a\ (fst\ 'Y),\ Sup-b\ (snd\ 'Y))$

lemma $lub-singleton-prod-lub\ [cont-intro,\ simp]:$
 $\llbracket lub-singleton\ luba;\ lub-singleton\ lubb \rrbracket \Rightarrow lub-singleton\ (prod-lub\ luba\ lubb)$
 $\langle proof \rangle$

lemma $prod-lub-empty\ [simp]: prod-lub\ luba\ lubb\ \{\} = (luba\ \{\},\ lubb\ \{\})$
 $\langle proof \rangle$

lemma $preorder-rel-prodI\ [cont-intro,\ simp]:$
assumes $class.preorder\ orda\ (mk-less\ orda)$
and $class.preorder\ ordb\ (mk-less\ ordb)$
shows $class.preorder\ (rel-prod\ orda\ ordb)\ (mk-less\ (rel-prod\ orda\ ordb))$
 $\langle proof \rangle$

lemma $order-rel-prodI:$
assumes $a: class.order\ orda\ (mk-less\ orda)$
and $b: class.order\ ordb\ (mk-less\ ordb)$
shows $class.order\ (rel-prod\ orda\ ordb)\ (mk-less\ (rel-prod\ orda\ ordb))$
 $(is\ class.order\ ?ord\ ?ord')$
 $\langle proof \rangle$

lemma $monotone-rel-prodI:$
assumes $mono2: \bigwedge a. monotone\ ordb\ ordc\ (\lambda b. f\ (a,\ b))$
and $mono1: \bigwedge b. monotone\ orda\ ordc\ (\lambda a. f\ (a,\ b))$
and $a: class.preorder\ orda\ (mk-less\ orda)$
and $b: class.preorder\ ordb\ (mk-less\ ordb)$
and $c: class.preorder\ ordc\ (mk-less\ ordc)$
shows $monotone\ (rel-prod\ orda\ ordb)\ ordc\ f$
 $\langle proof \rangle$

lemma $monotone-rel-prodD1:$
assumes $mono: monotone\ (rel-prod\ orda\ ordb)\ ordc\ f$
and $preorder: class.preorder\ ordb\ (mk-less\ ordb)$
shows $monotone\ orda\ ordc\ (\lambda a. f\ (a,\ b))$
 $\langle proof \rangle$

lemma $monotone-rel-prodD2:$
assumes $mono: monotone\ (rel-prod\ orda\ ordb)\ ordc\ f$
and $preorder: class.preorder\ orda\ (mk-less\ orda)$
shows $monotone\ ordb\ ordc\ (\lambda b. f\ (a,\ b))$
 $\langle proof \rangle$

lemma *monotone-case-prodI*:

[[$\wedge a. \text{monotone } \text{ordb } \text{ordc } (f a); \wedge b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f a b);$
 $\text{class.preorder } \text{orda } (\text{mk-less } \text{orda}); \text{class.preorder } \text{ordb } (\text{mk-less } \text{ordb});$
 $\text{class.preorder } \text{ordc } (\text{mk-less } \text{ordc})$]]
 $\implies \text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } (\text{case-prod } f)$
 $\langle \text{proof} \rangle$

lemma *monotone-case-prodD1*:

assumes *mono*: $\text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } (\text{case-prod } f)$
and *preorder*: $\text{class.preorder } \text{ordb } (\text{mk-less } \text{ordb})$
shows $\text{monotone } \text{orda } \text{ordc } (\lambda a. f a b)$
 $\langle \text{proof} \rangle$

lemma *monotone-case-prodD2*:

assumes *mono*: $\text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } (\text{case-prod } f)$
and *preorder*: $\text{class.preorder } \text{orda } (\text{mk-less } \text{orda})$
shows $\text{monotone } \text{ordb } \text{ordc } (f a)$
 $\langle \text{proof} \rangle$

context

fixes *orda ordb ordc*
assumes *a*: $\text{class.preorder } \text{orda } (\text{mk-less } \text{orda})$
and *b*: $\text{class.preorder } \text{ordb } (\text{mk-less } \text{ordb})$
and *c*: $\text{class.preorder } \text{ordc } (\text{mk-less } \text{ordc})$

begin

lemma *monotone-rel-prod-iff*:

$\text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } f \longleftrightarrow$
 $(\forall a. \text{monotone } \text{ordb } \text{ordc } (\lambda b. f (a, b))) \wedge$
 $(\forall b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f (a, b)))$
 $\langle \text{proof} \rangle$

lemma *monotone-case-prod-iff* [*simp*]:

$\text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } (\text{case-prod } f) \longleftrightarrow$
 $(\forall a. \text{monotone } \text{ordb } \text{ordc } (f a)) \wedge (\forall b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f a b))$
 $\langle \text{proof} \rangle$

end

lemma *monotone-case-prod-apply-iff*:

$\text{monotone } \text{orda } \text{ordb } (\lambda x. (\text{case-prod } f x) y) \longleftrightarrow \text{monotone } \text{orda } \text{ordb } (\text{case-prod } (\lambda a b. f a b y))$
 $\langle \text{proof} \rangle$

lemma *monotone-case-prod-applyD*:

$\text{monotone } \text{orda } \text{ordb } (\lambda x. (\text{case-prod } f x) y)$
 $\implies \text{monotone } \text{orda } \text{ordb } (\text{case-prod } (\lambda a b. f a b y))$
 $\langle \text{proof} \rangle$

lemma *monotone-case-prod-applyI*:

monotone orda ordb (case-prod ($\lambda a b. f a b y$))
 \implies *monotone orda ordb ($\lambda x. (case-prod f x) y$)*
 ⟨proof⟩

lemma *cont-case-prod-apply-iff*:

cont luba orda lubb ordb ($\lambda x. (case-prod f x) y$) \longleftrightarrow cont luba orda lubb ordb
(case-prod ($\lambda a b. f a b y$))
 ⟨proof⟩

lemma *cont-case-prod-applyI*:

cont luba orda lubb ordb (case-prod ($\lambda a b. f a b y$))
 \implies *cont luba orda lubb ordb ($\lambda x. (case-prod f x) y$)*
 ⟨proof⟩

lemma *cont-case-prod-applyD*:

cont luba orda lubb ordb ($\lambda x. (case-prod f x) y$)
 \implies *cont luba orda lubb ordb (case-prod ($\lambda a b. f a b y$))*
 ⟨proof⟩

lemma *mcont-case-prod-apply-iff [simp]*:

mcont luba orda lubb ordb ($\lambda x. (case-prod f x) y$) \longleftrightarrow
mcont luba orda lubb ordb (case-prod ($\lambda a b. f a b y$))
 ⟨proof⟩

lemma *cont-prodD1*:

assumes *cont: cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc f*
and *class.preorder orda (mk-less orda)*
and *luba: lub-singleton luba*
shows *cont lubb ordb lubc ordc ($\lambda y. f (x, y)$)*
 ⟨proof⟩

lemma *cont-prodD2*:

assumes *cont: cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc f*
and *class.preorder ordb (mk-less ordb)*
and *lubb: lub-singleton lubb*
shows *cont luba orda lubc ordc ($\lambda x. f (x, y)$)*
 ⟨proof⟩

lemma *cont-case-prodD1*:

assumes *cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc (case-prod f)*
and *class.preorder orda (mk-less orda)*
and *lub-singleton luba*
shows *cont lubb ordb lubc ordc (f x)*
 ⟨proof⟩

lemma *cont-case-prodD2*:

assumes *cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc (case-prod f)*

and *class.preorder ordb (mk-less ordb)*
and *lub-singleton lubb*
shows *cont luba orda lubc ordc* $(\lambda x. f x y)$
 <proof>

context *ccpo* **begin**

lemma *cont-prodI*:

assumes *mono: monotone (rel-prod orda ordb)* $(\leq) f$
and *cont1: $\bigwedge x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f (x, y))$*
and *cont2: $\bigwedge y. \text{cont luba orda Sup } (\leq) (\lambda x. f (x, y))$*
and *class.preorder orda (mk-less orda)*
and *class.preorder ordb (mk-less ordb)*
shows *cont (prod-lub luba lubb) (rel-prod orda ordb) Sup* $(\leq) f$
 <proof>

lemma *cont-case-prodI*:

assumes *monotone (rel-prod orda ordb)* $(\leq) (\text{case-prod } f)$
and $\bigwedge x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)$
and $\bigwedge y. \text{cont luba orda Sup } (\leq) (\lambda x. f x y)$
and *class.preorder orda (mk-less orda)*
and *class.preorder ordb (mk-less ordb)*
shows *cont (prod-lub luba lubb) (rel-prod orda ordb) Sup* $(\leq) (\text{case-prod } f)$
 <proof>

lemma *cont-case-prod-iff*:

$\llbracket \text{monotone (rel-prod orda ordb)} (\leq) (\text{case-prod } f);$
 $\text{class.preorder orda (mk-less orda); lub-singleton luba};$
 $\text{class.preorder ordb (mk-less ordb); lub-singleton lubb} \rrbracket$
 $\implies \text{cont (prod-lub luba lubb) (rel-prod orda ordb) Sup } (\leq) (\text{case-prod } f) \longleftrightarrow$
 $(\forall x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)) \wedge (\forall y. \text{cont luba orda Sup } (\leq) (\lambda x. f x y))$
 <proof>

end

context *partial-function-definitions* **begin**

lemma *mono2mono2*:

assumes *f: monotone (rel-prod ordb ordc) leq* $(\lambda(x, y). f x y)$
and *t: monotone orda ordb* $(\lambda x. t x)$
and *t': monotone orda ordc* $(\lambda x. t' x)$
shows *monotone orda leq* $(\lambda x. f (t x) (t' x))$
 <proof>

lemma *cont-case-prodI [cont-intro]*:

$\llbracket \text{monotone (rel-prod orda ordb) leq (case-prod } f);$
 $\bigwedge x. \text{cont lubb ordb lub leq } (\lambda y. f x y);$
 $\bigwedge y. \text{cont luba orda lub leq } (\lambda x. f x y);$

```

class.preorder orda (mk-less orda);
class.preorder ordb (mk-less ordb) ]
 $\Rightarrow$  cont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f)
<proof>

```

lemma *cont-case-prod-iff*:

```

[ monotone (rel-prod orda ordb) leq (case-prod f);
  class.preorder orda (mk-less orda); lub-singleton luba;
  class.preorder ordb (mk-less ordb); lub-singleton lubb ]
 $\Rightarrow$  cont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f)  $\longleftrightarrow$ 
  ( $\forall x.$  cont lubb ordb lub leq ( $\lambda y.$  f x y))  $\wedge$  ( $\forall y.$  cont luba orda lub leq ( $\lambda x.$  f x y))
<proof>

```

lemma *mcont-case-prod-iff [simp]*:

```

[ class.preorder orda (mk-less orda); lub-singleton luba;
  class.preorder ordb (mk-less ordb); lub-singleton lubb ]
 $\Rightarrow$  mcont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f)  $\longleftrightarrow$ 
  ( $\forall x.$  mcont lubb ordb lub leq ( $\lambda y.$  f x y))  $\wedge$  ( $\forall y.$  mcont luba orda lub leq ( $\lambda x.$  f x y))
<proof>

```

end

lemma *mono2mono-case-prod [cont-intro]*:

```

assumes  $\bigwedge x y.$  monotone orda ordb ( $\lambda f.$  pair f x y)
shows monotone orda ordb ( $\lambda f.$  case-prod (pair f) x)
<proof>

```

15.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: class.ccpo Sup (\leq) ($<$)
 <proof>

lemma *complete-lattice-ccpo'*: class.ccpo Sup (\leq) (mk-less (\leq))
 <proof>

lemma *complete-lattice-partial-function-definitions*:
 partial-function-definitions (\leq) Sup
 <proof>

lemma *complete-lattice-partial-function-definitions-dual*:
 partial-function-definitions (\geq) Inf
 <proof>

lemmas [cont-intro, simp] =

```

Partial-Function.ccpo[OF complete-lattice-partial-function-definitions]
Partial-Function.ccpo[OF complete-lattice-partial-function-definitions-dual]

```

lemma *mono2mono-inf*:

assumes f : *monotone ord* (\leq) $(\lambda x. f\ x)$
and g : *monotone ord* (\leq) $(\lambda x. g\ x)$
shows *monotone ord* (\leq) $(\lambda x. f\ x \sqcap g\ x)$
 $\langle proof \rangle$

lemma *mcont-const* [*simp*]: *mcont lub ord Sup* (\leq) $(\lambda -. c)$
 $\langle proof \rangle$

lemma *mono2mono-sup*:

assumes f : *monotone ord* (\leq) $(\lambda x. f\ x)$
and g : *monotone ord* (\leq) $(\lambda x. g\ x)$
shows *monotone ord* (\leq) $(\lambda x. f\ x \sqcup g\ x)$
 $\langle proof \rangle$

lemma *Sup-image-sup*:

assumes $Y \neq \{\}$
shows $\bigsqcup ((\bigsqcup) x \text{ ‘ } Y) = x \sqcup \bigsqcup Y$
 $\langle proof \rangle$

lemma *mcont-sup1*: *mcont Sup* (\leq) *Sup* (\leq) $(\lambda y. x \sqcup y)$
 $\langle proof \rangle$

lemma *mcont-sup2*: *mcont Sup* (\leq) *Sup* (\leq) $(\lambda x. x \sqcup y)$
 $\langle proof \rangle$

lemma *mcont2mcont-sup* [*cont-intro*, *simp*]:

\llbracket *mcont lub ord Sup* (\leq) $(\lambda x. f\ x)$;
mcont lub ord Sup (\leq) $(\lambda x. g\ x)$ \rrbracket
 \implies *mcont lub ord Sup* (\leq) $(\lambda x. f\ x \sqcup g\ x)$
 $\langle proof \rangle$

end

lemmas [*cont-intro*] = *admissible-leI*[*OF complete-lattice-ccpo*]

context *complete-distrib-lattice* **begin**

lemma *mcont-inf1*: *mcont Sup* (\leq) *Sup* (\leq) $(\lambda y. x \sqcap y)$
 $\langle proof \rangle$

lemma *mcont-inf2*: *mcont Sup* (\leq) *Sup* (\leq) $(\lambda x. x \sqcap y)$
 $\langle proof \rangle$

lemma *mcont2mcont-inf* [*cont-intro*, *simp*]:

\llbracket *mcont lub ord Sup* (\leq) $(\lambda x. f\ x)$;
mcont lub ord Sup (\leq) $(\lambda x. g\ x)$ \rrbracket
 \implies *mcont lub ord Sup* (\leq) $(\lambda x. f\ x \sqcap g\ x)$

$\langle \text{proof} \rangle$

end

interpretation *lfp: partial-function-definitions* $(\leq) :: - :: \text{complete-lattice} \Rightarrow - \text{Sup}$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

interpretation *gfp: partial-function-definitions* $(\geq) :: - :: \text{complete-lattice} \Rightarrow - \text{Inf}$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma *insert-mono* [*partial-function-mono*]:
 $\text{monotone } (\text{fun-ord } (\subseteq)) (\subseteq) A \implies \text{monotone } (\text{fun-ord } (\subseteq)) (\subseteq) (\lambda y. \text{insert } x (A y))$
 $\langle \text{proof} \rangle$

lemma *mono2mono-insert* [*THEN lfp.mono2mono, cont-intro, simp*]:
shows *monotone-insert*: $\text{monotone } (\subseteq) (\subseteq) (\text{insert } x)$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-insert* [*THEN lfp.mcont2mcont, cont-intro, simp*]:
shows *mcont-insert*: $\text{mcont } \text{Union } (\subseteq) \text{Union } (\subseteq) (\text{insert } x)$
 $\langle \text{proof} \rangle$

lemma *mono2mono-image* [*THEN lfp.mono2mono, cont-intro, simp*]:
shows *monotone-image*: $\text{monotone } (\subseteq) (\subseteq) ((\cdot) f)$
 $\langle \text{proof} \rangle$

lemma *cont-image*: $\text{cont } \text{Union } (\subseteq) \text{Union } (\subseteq) ((\cdot) f)$
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-image* [*THEN lfp.mcont2mcont, cont-intro, simp*]:
shows *mcont-image*: $\text{mcont } \text{Union } (\subseteq) \text{Union } (\subseteq) ((\cdot) f)$
 $\langle \text{proof} \rangle$

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro, simp*]:
 $\text{monotone ord } (\subseteq) f \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup f x)$
 $\langle \text{proof} \rangle$

lemma *cont-Sup*:
assumes *cont lub ord Union* $(\subseteq) f$
shows *cont lub ord Sup* $(\leq) (\lambda x. \bigsqcup f x)$
 $\langle \text{proof} \rangle$

lemma *mcont-Sup*: $mcont\ lub\ ord\ Union\ (\subseteq)\ f \implies mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. \bigsqcup f\ x)$
 <proof>

lemma *monotone-SUP*:
 $\llbracket monotone\ ord\ (\subseteq)\ f; \bigwedge y. monotone\ ord\ (\leq)\ (\lambda x. g\ x\ y) \rrbracket \implies monotone\ ord\ (\leq)\ (\lambda x. \bigsqcup_{y \in f\ x} g\ x\ y)$
 <proof>

lemma *monotone-SUP2*:
 $(\bigwedge y. y \in A \implies monotone\ ord\ (\leq)\ (\lambda x. g\ x\ y)) \implies monotone\ ord\ (\leq)\ (\lambda x. \bigsqcup_{y \in A} g\ x\ y)$
 <proof>

lemma *cont-SUP*:
 assumes $f: mcont\ lub\ ord\ Union\ (\subseteq)\ f$
 and $g: \bigwedge y. mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. g\ x\ y)$
 shows $cont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. \bigsqcup_{y \in f\ x} g\ x\ y)$
 <proof>

lemma *mcont-SUP* [*cont-intro, simp*]:
 $\llbracket mcont\ lub\ ord\ Union\ (\subseteq)\ f; \bigwedge y. mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. g\ x\ y) \rrbracket \implies mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. \bigsqcup_{y \in f\ x} g\ x\ y)$
 <proof>

end

lemma *admissible-Ball* [*cont-intro, simp*]:
 $\llbracket \bigwedge x. cppo.admissible\ lub\ ord\ (\lambda A. P\ A\ x); mcont\ lub\ ord\ Union\ (\subseteq)\ f; class.cppo\ lub\ ord\ (mk-less\ ord) \rrbracket \implies cppo.admissible\ lub\ ord\ (\lambda A. \forall x \in f\ A. P\ A\ x)$
 <proof>

lemma *admissible-Bex* [*THEN admissible-subst, cont-intro, simp*]:
 shows *admissible-Bex*: $cpo.admissible\ Union\ (\subseteq)\ (\lambda A. \exists x \in A. P\ x)$
 <proof>

15.6 Parallel fixpoint induction

context
 fixes $luba :: 'a\ set \Rightarrow 'a$
 and $orda :: 'a \Rightarrow 'a \Rightarrow bool$
 and $lubb :: 'b\ set \Rightarrow 'b$
 and $ordb :: 'b \Rightarrow 'b \Rightarrow bool$
 assumes $a: class.cppo\ luba\ orda\ (mk-less\ orda)$
 and $b: class.cppo\ lubb\ ordb\ (mk-less\ ordb)$
begin

interpretation *a*: *ccpo luba orda mk-less orda* *<proof>*

interpretation *b*: *ccpo lubb ordb mk-less ordb* *<proof>*

lemma *ccpo-rel-prodI*:

class.ccpo (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
(is class.ccpo ?lub ?ord ?ord')
<proof>

interpretation *ab*: *ccpo prod-lub luba lubb rel-prod orda ordb mk-less* (*rel-prod orda ordb*)
<proof>

lemma *monotone-map-prod* [*simp*]:

monotone (*rel-prod orda ordb*) (*rel-prod ordc ordd*) (*map-prod f g*) \longleftrightarrow
monotone orda ordc f \wedge *monotone ordb ordd g*
<proof>

lemma *parallel-fixp-induct*:

assumes *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) ($\lambda x. P$
(fst x) (snd x))
and *f*: *monotone orda orda f*
and *g*: *monotone ordb ordb g*
and *bot*: *P* (*luba {}*) (*lubb {}*)
and *step*: $\bigwedge x y. P\ x\ y \implies P\ (f\ x)\ (g\ y)$
shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
<proof>

end

lemma *parallel-fixp-induct-uc*:

assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: $\bigwedge x. \text{monotone } (\text{fun-ord } orda) \text{ orda } (\lambda f. U1\ (F\ (C1\ f))\ x)$
and *G*: $\bigwedge y. \text{monotone } (\text{fun-ord } ordb) \text{ ordb } (\lambda g. U2\ (G\ (C2\ g))\ y)$
and *eq1*: *f* $\equiv C1\ (\text{ccpo.fixp } (\text{fun-lub luba})\ (\text{fun-ord } orda))\ (\lambda f. U1\ (F\ (C1\ f)))$
and *eq2*: *g* $\equiv C2\ (\text{ccpo.fixp } (\text{fun-lub lubb})\ (\text{fun-ord } ordb))\ (\lambda g. U2\ (G\ (C2\ g)))$
and *inverse*: $\bigwedge f. U1\ (C1\ f) = f$
and *inverse2*: $\bigwedge g. U2\ (C2\ g) = g$
and *adm*: *ccpo.admissible* (*prod-lub* (*fun-lub luba*) (*fun-lub lubb*)) (*rel-prod* (*fun-ord*
orda) (*fun-ord ordb*)) ($\lambda x. P\ (\text{fst } x)\ (\text{snd } x)$)
and *bot*: *P* ($\lambda-. \text{luba } \{\}$) ($\lambda-. \text{lubb } \{\}$)
and *step*: $\bigwedge f g. P\ (U1\ f)\ (U2\ g) \implies P\ (U1\ (F\ f))\ (U2\ (G\ g))$
shows *P* (*U1 f*) (*U2 g*)
<proof>

lemmas *parallel-fixp-induct-1-1* = *parallel-fixp-induct-uc*[

of - - - $\lambda x. x - \lambda x. x\ \lambda x. x - \lambda x. x$,
OF - - - - - refl refl]

lemmas *parallel-fixp-induct-2-2* = *parallel-fixp-induct-uc*[
of - - - *case-prod* - *curry case-prod* - *curry*,
where $P = \lambda f g. P (curry f) (curry g)$,
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - *refl refl*]
for P

lemma *monotone-fst*: *monotone* (*rel-prod* *orda* *ordb*) *orda* *fst*
 ⟨*proof*⟩

lemma *mcont-fst*: *mcont* (*prod-lub* *luba* *lubb*) (*rel-prod* *orda* *ordb*) *luba* *orda* *fst*
 ⟨*proof*⟩

lemma *mcont2mcont-fst* [*cont-intro*, *simp*]:
 $mcont\ lub\ ord\ (prod-lub\ luba\ lubb)\ (rel-prod\ orda\ ordb)\ t$
 $\implies mcont\ lub\ ord\ luba\ orda\ (\lambda x. fst\ (t\ x))$
 ⟨*proof*⟩

lemma *monotone-snd*: *monotone* (*rel-prod* *orda* *ordb*) *ordb* *snd*
 ⟨*proof*⟩

lemma *mcont-snd*: *mcont* (*prod-lub* *luba* *lubb*) (*rel-prod* *orda* *ordb*) *lubb* *ordb* *snd*
 ⟨*proof*⟩

lemma *mcont2mcont-snd* [*cont-intro*, *simp*]:
 $mcont\ lub\ ord\ (prod-lub\ luba\ lubb)\ (rel-prod\ orda\ ordb)\ t$
 $\implies mcont\ lub\ ord\ lubb\ ordb\ (\lambda x. snd\ (t\ x))$
 ⟨*proof*⟩

lemma *monotone-Pair*:
 $\llbracket monotone\ ord\ orda\ f; monotone\ ord\ ordb\ g \rrbracket$
 $\implies monotone\ ord\ (rel-prod\ orda\ ordb)\ (\lambda x. (f\ x, g\ x))$
 ⟨*proof*⟩

lemma *cont-Pair*:
 $\llbracket cont\ lub\ ord\ luba\ orda\ f; cont\ lub\ ord\ lubb\ ordb\ g \rrbracket$
 $\implies cont\ lub\ ord\ (prod-lub\ luba\ lubb)\ (rel-prod\ orda\ ordb)\ (\lambda x. (f\ x, g\ x))$
 ⟨*proof*⟩

lemma *mcont-Pair*:
 $\llbracket mcont\ lub\ ord\ luba\ orda\ f; mcont\ lub\ ord\ lubb\ ordb\ g \rrbracket$
 $\implies mcont\ lub\ ord\ (prod-lub\ luba\ lubb)\ (rel-prod\ orda\ ordb)\ (\lambda x. (f\ x, g\ x))$
 ⟨*proof*⟩

context *partial-function-definitions*
begin

Specialised versions of *mcont-call* for admissibility proofs for parallel
 fixpoint inductions

```

lemmas mcont-call-fst [cont-intro] = mcont-call[THEN mcont2mcont, OF mcont-fst]
lemmas mcont-call-snd [cont-intro] = mcont-call[THEN mcont2mcont, OF mcont-snd]
end

```

```

lemma map-option-mono [partial-function-mono]:
  mono-option B  $\implies$  mono-option ( $\lambda f. \text{map-option } g \ (B \ f)$ )
<proof>

```

```

lemma compact-flat-lub [cont-intro]: ccpo.compact (flat-lub x) (flat-ord x) y
<proof>

```

```

end

```

```

theory Conditional-Parametricity
imports Main
keywords parametric-constant :: thy-decl
begin

```

```

context includes lifting-syntax begin

```

```

qualified definition Rel-match :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  bool where
  Rel-match R x y = R x y

```

```

named-theorems parametricity-preprocess

```

```

lemma bi-unique-Rel-match [parametricity-preprocess]:
  bi-unique A = Rel-match (A  $\implies$  A  $\implies$  (=)) (=) (=)
<proof>

```

```

lemma bi-total-Rel-match [parametricity-preprocess]:
  bi-total A = Rel-match ((A  $\implies$  (=))  $\implies$  (=)) All All
<proof>

```

```

lemma is-equality-Rel: is-equality A  $\implies$  Transfer.Rel A t t
<proof>

```

```

lemma Rel-Rel-match: Transfer.Rel R x y  $\implies$  Rel-match R x y
<proof>

```

```

lemma Rel-match-Rel: Rel-match R x y  $\implies$  Transfer.Rel R x y
<proof>

```

```

lemma Rel-Rel-match-eq: Transfer.Rel R x y = Rel-match R x y
<proof>

```

```

lemma Rel-match-app:
  assumes Rel-match (A  $\implies$  B) f g and Transfer.Rel A x y
  shows Rel-match B (f x) (g y)

```

$\langle \text{proof} \rangle$
end
 $\langle ML \rangle$
end
theory *Confluence* **imports**
 Main
begin

16 Confluence

definition *semiconfluentp* :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ **where**
semiconfluentp $r \longleftrightarrow r^{-1-1} \text{ OO } r^{**} \leq r^{**} \text{ OO } r^{-1-1**}$

definition *confluentp* :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ **where**
confluentp $r \longleftrightarrow r^{-1-1**} \text{ OO } r^{**} \leq r^{**} \text{ OO } r^{-1-1**}$

definition *strong-confluentp* :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ **where**
strong-confluentp $r \longleftrightarrow r^{-1-1} \text{ OO } r \leq r^{**} \text{ OO } (r^{-1-1})^{==}$

lemma *semiconfluentpI* [intro?]:
semiconfluentp r **if** $\bigwedge x y z. \llbracket r x y; r^{**} x z \rrbracket \Longrightarrow \exists u. r^{**} y u \wedge r^{**} z u$
 $\langle \text{proof} \rangle$

lemma *semiconfluentpD*: $\exists u. r^{**} y u \wedge r^{**} z u$ **if** *semiconfluentp* r $r x y r^{**} x z$
 $\langle \text{proof} \rangle$

lemma *confluentpI*:
confluentp r **if** $\bigwedge x y z. \llbracket r^{**} x y; r^{**} x z \rrbracket \Longrightarrow \exists u. r^{**} y u \wedge r^{**} z u$
 $\langle \text{proof} \rangle$

lemma *confluentpD*: $\exists u. r^{**} y u \wedge r^{**} z u$ **if** *confluentp* r $r^{**} x y r^{**} x z$
 $\langle \text{proof} \rangle$

lemma *strong-confluentpI* [intro?]:
strong-confluentp r **if** $\bigwedge x y z. \llbracket r x y; r x z \rrbracket \Longrightarrow \exists u. r^{**} y u \wedge r^{==} z u$
 $\langle \text{proof} \rangle$

lemma *strong-confluentpD*: $\exists u. r^{**} y u \wedge r^{==} z u$ **if** *strong-confluentp* r $r x y r x z$
 $\langle \text{proof} \rangle$

lemma *semiconfluentp-imp-confluentp*: *confluentp* r **if** r : *semiconfluentp* r
 $\langle \text{proof} \rangle$

lemma *confluentp-imp-semiconfluentp*: *semiconfluentp* r **if** *confluentp* r
 $\langle \text{proof} \rangle$

lemma *confluentp-eq-semiconfluentp*: $\text{confluentp } r \longleftrightarrow \text{semiconfluentp } r$
 ⟨proof⟩

lemma *confluentp-conv-strong-confluentp-rtrancp*:
 $\text{confluentp } r \longleftrightarrow \text{strong-confluentp } (r^{**})$
 ⟨proof⟩

lemma *strong-confluentp-into-semiconfluentp*:
 $\text{semiconfluentp } r$ **if** r : $\text{strong-confluentp } r$
 ⟨proof⟩

lemma *strong-confluentp-imp-confluentp*: $\text{confluentp } r$ **if** $\text{strong-confluentp } r$
 ⟨proof⟩

lemma *semiconfluentp-equivclp*: $\text{equivclp } r = r^{**} \text{ OO } r^{-1-1**}$ **if** r : $\text{semiconfluentp } r$
 ⟨proof⟩

end

theory *Confluent-Quotient* **imports**

Confluence

begin

Functors with finite setters preserve wide intersection for any equivalence relation that respects the mapper.

lemma *Inter-finite-subset*:
assumes $\forall A \in \mathcal{A}. \text{finite } A$
shows $\exists \mathcal{B} \subseteq \mathcal{A}. \text{finite } \mathcal{B} \wedge (\bigcap \mathcal{B}) = (\bigcap \mathcal{A})$
 ⟨proof⟩

locale *wide-intersection-finite* =
fixes $E :: 'Fa \Rightarrow 'Fa \Rightarrow \text{bool}$
and $\text{mapFa} :: ('a \Rightarrow 'a) \Rightarrow 'Fa \Rightarrow 'Fa$
and $\text{setFa} :: 'Fa \Rightarrow 'a \text{ set}$
assumes $\text{equiv}: \text{equivp } E$
and $\text{map-E}: E \ x \ y \Longrightarrow E \ (\text{mapFa } f \ x) \ (\text{mapFa } f \ y)$
and $\text{map-id}: \text{mapFa } \text{id} \ x = x$
and $\text{map-cong}: \forall a \in \text{setFa } x. f \ a = g \ a \Longrightarrow \text{mapFa } f \ x = \text{mapFa } g \ x$
and $\text{set-map}: \text{setFa } (\text{mapFa } f \ x) = f \ ` \ \text{setFa } x$
and $\text{finite}: \text{finite } (\text{setFa } x)$
begin

lemma *binary-intersection*:
assumes $E \ y \ z$ **and** $y: \text{setFa } y \subseteq Y$ **and** $z: \text{setFa } z \subseteq Z$ **and** $a: a \in Y \ a \in Z$
shows $\exists x. E \ x \ y \wedge \text{setFa } x \subseteq Y \wedge \text{setFa } x \subseteq Z$
 ⟨proof⟩

lemma *finite-intersection*:

assumes $E: \forall y \in A. E y z$
and $fin: \text{finite } A$
and $sub: \forall y \in A. \text{setFa } y \subseteq Y y \wedge a \in Y y$
shows $\exists x. E x z \wedge (\forall y \in A. \text{setFa } x \subseteq Y y)$
 $\langle \text{proof} \rangle$

lemma *wide-intersection*:

assumes *inter-nonempty*: $\bigcap Ss \neq \{\}$
shows $(\bigcap As \in Ss. \{(x, x'). E x x'\} \text{ “ } \{x. \text{setFa } x \subseteq As\} \subseteq \{(x, x'). E x x'\} \text{ “ } \{x. \text{setFa } x \subseteq \bigcap Ss\} \text{ (is } ?lhs \subseteq ?rhs)$
 $\langle \text{proof} \rangle$

end

Subdistributivity for quotients via confluence

lemma *rtranclp-transp-reflp*: $R^{**} = R$ if *transp* R *reflp* R
 $\langle \text{proof} \rangle$

lemma *rtranclp-equivp*: $R^{**} = R$ if *equivp* R
 $\langle \text{proof} \rangle$

locale *confluent-quotient* =

fixes $Rb :: 'Fb \Rightarrow 'Fb \Rightarrow \text{bool}$
and $Ea :: 'Fa \Rightarrow 'Fa \Rightarrow \text{bool}$
and $Eb :: 'Fb \Rightarrow 'Fb \Rightarrow \text{bool}$
and $Ec :: 'Fc \Rightarrow 'Fc \Rightarrow \text{bool}$
and $Eab :: 'Fab \Rightarrow 'Fab \Rightarrow \text{bool}$
and $Ebc :: 'Fbc \Rightarrow 'Fbc \Rightarrow \text{bool}$
and $\pi\text{-Faba} :: 'Fab \Rightarrow 'Fa$
and $\pi\text{-Fabb} :: 'Fab \Rightarrow 'Fb$
and $\pi\text{-Fbcb} :: 'Fbc \Rightarrow 'Fb$
and $\pi\text{-Fbcc} :: 'Fbc \Rightarrow 'Fc$
and $rel\text{-Fab} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'Fa \Rightarrow 'Fb \Rightarrow \text{bool}$
and $rel\text{-Fbc} :: ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow 'Fb \Rightarrow 'Fc \Rightarrow \text{bool}$
and $rel\text{-Fac} :: ('a \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow 'Fa \Rightarrow 'Fc \Rightarrow \text{bool}$
and $set\text{-Fab} :: 'Fab \Rightarrow ('a \times 'b) \text{ set}$
and $set\text{-Fbc} :: 'Fbc \Rightarrow ('b \times 'c) \text{ set}$
assumes *confluent*: *confluentp* Rb
and *retract1-ab*: $\bigwedge x y. Rb (\pi\text{-Fabb } x) y \implies \exists z. Eab x z \wedge y = \pi\text{-Fabb } z \wedge \text{set-Fab } z \subseteq \text{set-Fab } x$
and *retract1-bc*: $\bigwedge x y. Rb (\pi\text{-Fbcb } x) y \implies \exists z. Ebc x z \wedge y = \pi\text{-Fbcb } z \wedge \text{set-Fbc } z \subseteq \text{set-Fbc } x$
and *generated-b*: $Eb \leq \text{equivclp } Rb$
and *transp-a*: *transp* Ea
and *transp-c*: *transp* Ec
and *equivp-ab*: *equivp* Eab
and *equivp-bc*: *equivp* Ebc
and *in-rel-Fab*: $\bigwedge A x y. rel\text{-Fab } A x y \longleftrightarrow (\exists z. z \in \{x. \text{set-Fab } x \subseteq \{(x, y). A x y\}\} \wedge \pi\text{-Faba } z = x \wedge \pi\text{-Fabb } z = y)$

and *in-rel-Fbc*: $\bigwedge B \ x \ y. \ rel-Fbc \ B \ x \ y \longleftrightarrow (\exists z. \ z \in \{x. \ set-Fbc \ x \subseteq \{(x, y). \ B \ x \ y\}\} \wedge \pi-Fbcb \ z = x \wedge \pi-Fbcc \ z = y)$
and *rel-comp*: $\bigwedge A \ B. \ rel-Fac \ (A \ OO \ B) = rel-Fab \ A \ OO \ rel-Fbc \ B$
and $\pi-Faba-respect$: $rel-fun \ Eab \ Ea \ \pi-Faba \ \pi-Faba$
and $\pi-Fbcc-respect$: $rel-fun \ Ebc \ Ec \ \pi-Fbcc \ \pi-Fbcc$
begin

lemma *retract-ab*: $Rb^{**} \ (\pi-Fabb \ x) \ y \implies \exists z. \ Eab \ x \ z \wedge y = \pi-Fabb \ z \wedge set-Fab \ z \subseteq set-Fab \ x$
 $\langle proof \rangle$

lemma *retract-bc*: $Rb^{**} \ (\pi-Fbcb \ x) \ y \implies \exists z. \ Ebc \ x \ z \wedge y = \pi-Fbcb \ z \wedge set-Fbc \ z \subseteq set-Fbc \ x$
 $\langle proof \rangle$

lemma *subdistributivity*: $rel-Fab \ A \ OO \ Eb \ OO \ rel-Fbc \ B \leq Ea \ OO \ rel-Fac \ (A \ OO \ B) \ OO \ Ec$
 $\langle proof \rangle$

end

end

17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory *Old-Datatype*
imports *Main*
begin

17.1 The datatype universe

definition *Node* = $\{p. \ \exists f \ x \ k. \ p = (f :: nat \Rightarrow 'b + nat, \ x :: 'a + nat) \wedge f \ k = Inr \ 0\}$

typedef $('a, 'b) \ node = Node :: ((nat \Rightarrow 'b + nat) * ('a + nat)) \ set$
morphisms *Rep-Node* *Abs-Node*
 $\langle proof \rangle$

Datatypes will be represented by sets of type *node*

type-synonym $'a \ item = ('a, unit) \ node \ set$
type-synonym $('a, 'b) \ dtree = ('a, 'b) \ node \ set$

definition *Push* :: $[('b + nat), nat \Rightarrow ('b + nat)] \Rightarrow (nat \Rightarrow ('b + nat))$

where *Push* == $(\%b \ h. \ case-nat \ b \ h)$

definition *Push-Node* :: $[('b + nat), ('a, 'b) \ node] \Rightarrow ('a, 'b) \ node$

where $Push\text{-}Node == (\%n\ x.\ Abs\text{-}Node\ (apfst\ (Push\ n)\ (Rep\text{-}Node\ x)))$

definition $Atom :: ('a + nat) \Rightarrow ('a, 'b)\ dtree$

where $Atom == (\%x.\ \{Abs\text{-}Node(\%k.\ Inr\ 0,\ x)\})$

definition $Scons :: [('a, 'b)\ dtree, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$

where $Scons\ M\ N == (Push\text{-}Node\ (Inr\ 1)\ 'M)\ Un\ (Push\text{-}Node\ (Inr\ (Suc\ 1))\ 'N)$

definition $Leaf :: 'a \Rightarrow ('a, 'b)\ dtree$

where $Leaf == Atom \circ Inl$

definition $Numb :: nat \Rightarrow ('a, 'b)\ dtree$

where $Numb == Atom \circ Inr$

definition $In0 :: ('a, 'b)\ dtree \Rightarrow ('a, 'b)\ dtree$

where $In0(M) == Scons\ (Numb\ 0)\ M$

definition $In1 :: ('a, 'b)\ dtree \Rightarrow ('a, 'b)\ dtree$

where $In1(M) == Scons\ (Numb\ 1)\ M$

definition $Lim :: ('b \Rightarrow ('a, 'b)\ dtree) \Rightarrow ('a, 'b)\ dtree$

where $Lim\ f == \bigcup \{z.\ \exists x.\ z = Push\text{-}Node\ (Inl\ x)\ ' (f\ x)\}$

definition $ndepth :: ('a, 'b)\ node \Rightarrow nat$

where $ndepth(n) == (\%(f,x).\ LEAST\ k.\ f\ k = Inr\ 0)\ (Rep\text{-}Node\ n)$

definition $ntrunc :: [nat, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$

where $ntrunc\ k\ N == \{n.\ n \in N \wedge ndepth(n) < k\}$

definition $uprod :: [('a, 'b)\ dtree\ set, ('a, 'b)\ dtree\ set] \Rightarrow ('a, 'b)\ dtree\ set$

where $uprod\ A\ B == UN\ x:A.\ UN\ y:B.\ \{Scons\ x\ y\}$

definition $usum :: [('a, 'b)\ dtree\ set, ('a, 'b)\ dtree\ set] \Rightarrow ('a, 'b)\ dtree\ set$

where $usum\ A\ B == In0'A\ Un\ In1'B$

definition $Split :: [(('a, 'b)\ dtree, ('a, 'b)\ dtree) \Rightarrow 'c, ('a, 'b)\ dtree] \Rightarrow 'c$

where $Split\ c\ M == THE\ u.\ \exists x\ y.\ M = Scons\ x\ y \wedge u = c\ x\ y$

definition $Case :: [(('a, 'b)\ dtree) \Rightarrow 'c, (('a, 'b)\ dtree) \Rightarrow 'c, ('a, 'b)\ dtree] \Rightarrow 'c$

where $Case\ c\ d\ M == THE\ u.\ (\exists x.\ M = In0(x) \wedge u = c(x)) \vee (\exists y.\ M = In1(y) \wedge u = d(y))$

definition $dprod :: [((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}, ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}]$
 $=> ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}$
where $dprod \ r \ s == UN \ (x, x') : r. \ UN \ (y, y') : s. \ \{(Scons \ x \ y, \ Scons \ x' \ y')\}$

definition $dsum :: [((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}, ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}]$
 $=> ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}$
where $dsum \ r \ s == (UN \ (x, x') : r. \ \{(In0(x), In0(x'))\}) \ Un \ (UN \ (y, y') : s. \ \{(In1(y), In1(y'))\})$

lemma $apfst\text{-}convE$:
 $[\ [\ q = apfst \ f \ p; \ \forall x \ y. \ [\ p = (x, y); \ q = (f(x), y) \] \implies R \] \implies R$
 $\langle proof \rangle$

lemma $Push\text{-}inject1$: $Push \ i \ f = Push \ j \ g \implies i=j$
 $\langle proof \rangle$

lemma $Push\text{-}inject2$: $Push \ i \ f = Push \ j \ g \implies f=g$
 $\langle proof \rangle$

lemma $Push\text{-}inject$:
 $[\ Push \ i \ f = Push \ j \ g; \ [\ i=j; \ f=g \] \implies P \] \implies P$
 $\langle proof \rangle$

lemma $Push\text{-}neq\text{-}K0$: $Push \ (Inr \ (Suc \ k)) \ f = (\%z. \ Inr \ 0) \implies P$
 $\langle proof \rangle$

lemmas $Abs\text{-}Node\text{-}inj = Abs\text{-}Node\text{-}inject \ [THEN \ [2] \ rev\text{-}iffD1]$

lemma $Node\text{-}K0\text{-}I$: $(\lambda k. \ Inr \ 0, \ a) \in Node$
 $\langle proof \rangle$

lemma $Node\text{-}Push\text{-}I$: $p \in Node \implies apfst \ (Push \ i) \ p \in Node$
 $\langle proof \rangle$

17.2 Freeness: Distinctness of Constructors

lemma $Scons\text{-}not\text{-}Atom \ [iff]$: $Scons \ M \ N \neq Atom(a)$
 $\langle proof \rangle$

lemmas *Atom-not-Scons* [iff] = *Scons-not-Atom* [THEN not-sym]

lemma *inj-Atom*: *inj*(*Atom*)

⟨*proof*⟩

lemmas *Atom-inject* = *inj-Atom* [THEN *injD*]

lemma *Atom-Atom-eq* [iff]: (*Atom*(*a*)=*Atom*(*b*)) = (*a*=*b*)

⟨*proof*⟩

lemma *inj-Leaf*: *inj*(*Leaf*)

⟨*proof*⟩

lemmas *Leaf-inject* [*dest!*] = *inj-Leaf* [THEN *injD*]

lemma *inj-Numb*: *inj*(*Numb*)

⟨*proof*⟩

lemmas *Numb-inject* [*dest!*] = *inj-Numb* [THEN *injD*]

lemma *Push-Node-inject*:

[| *Push-Node i m = Push-Node j n*; [| *i=j*; *m=n* |] ==> *P* |]
[|] ==> *P*

⟨*proof*⟩

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M* <= *M'*

⟨*proof*⟩

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M' N'* ==> *N* <= *N'*

⟨*proof*⟩

lemma *Scons-inject1*: *Scons M N* = *Scons M' N'* ==> *M*=*M'*

⟨*proof*⟩

lemma *Scons-inject2*: *Scons M N* = *Scons M' N'* ==> *N*=*N'*

⟨*proof*⟩

lemma *Scons-inject*:

[| *Scons M N* = *Scons M' N'*; [| *M*=*M'*; *N*=*N'* |] ==> *P* |] ==> *P*

$\langle \text{proof} \rangle$

lemma *Scons-Scons-eq* [iff]: $(\text{Scons } M \ N = \text{Scons } M' \ N') = (M=M' \wedge N=N')$
 $\langle \text{proof} \rangle$

lemma *Scons-not-Leaf* [iff]: $\text{Scons } M \ N \neq \text{Leaf}(a)$
 $\langle \text{proof} \rangle$

lemmas *Leaf-not-Scons* [iff] = *Scons-not-Leaf* [THEN not-sym]

lemma *Scons-not-Numb* [iff]: $\text{Scons } M \ N \neq \text{Numb}(k)$
 $\langle \text{proof} \rangle$

lemmas *Numb-not-Scons* [iff] = *Scons-not-Numb* [THEN not-sym]

lemma *Leaf-not-Numb* [iff]: $\text{Leaf}(a) \neq \text{Numb}(k)$
 $\langle \text{proof} \rangle$

lemmas *Numb-not-Leaf* [iff] = *Leaf-not-Numb* [THEN not-sym]

lemma *ndepth-K0*: $\text{ndepth } (\text{Abs-Node}(\%k. \text{Inr } 0, x)) = 0$
 $\langle \text{proof} \rangle$

lemma *ndepth-Push-Node-aux*:
 $\text{case-nat } (\text{Inr } (\text{Suc } i)) \ f \ k = \text{Inr } 0 \longrightarrow \text{Suc}(\text{LEAST } x. f \ x = \text{Inr } 0) \leq k$
 $\langle \text{proof} \rangle$

lemma *ndepth-Push-Node*:
 $\text{ndepth } (\text{Push-Node } (\text{Inr } (\text{Suc } i)) \ n) = \text{Suc}(\text{ndepth}(n))$
 $\langle \text{proof} \rangle$

lemma *ntrunc-0* [simp]: $\text{ntrunc } 0 \ M = \{\}$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Atom* [simp]: $\text{ntrunc } (\text{Suc } k) (\text{Atom } a) = \text{Atom}(a)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Leaf* [simp]: $\text{ntrunc } (\text{Suc } k) (\text{Leaf } a) = \text{Leaf}(a)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Numb* [simp]: $\text{ntrunc } (\text{Suc } k) (\text{Numb } i) = \text{Numb}(i)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-Scons* [simp]:
 $\text{ntrunc } (\text{Suc } k) (\text{Scons } M N) = \text{Scons } (\text{ntrunc } k M) (\text{ntrunc } k N)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-one-In0* [simp]: $\text{ntrunc } (\text{Suc } 0) (\text{In0 } M) = \{\}$
 $\langle \text{proof} \rangle$

lemma *ntrunc-In0* [simp]: $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In0 } M) = \text{In0 } (\text{ntrunc } (\text{Suc } k) M)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-one-In1* [simp]: $\text{ntrunc } (\text{Suc } 0) (\text{In1 } M) = \{\}$
 $\langle \text{proof} \rangle$

lemma *ntrunc-In1* [simp]: $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In1 } M) = \text{In1 } (\text{ntrunc } (\text{Suc } k) M)$
 $\langle \text{proof} \rangle$

17.3 Set Constructions

lemma *uprodI* [intro!]: $\llbracket M \in A; N \in B \rrbracket \implies \text{Scons } M N \in \text{uprod } A B$
 $\langle \text{proof} \rangle$

lemma *uprodE* [elim!]:
 $\llbracket c \in \text{uprod } A B;$
 $\quad \bigwedge x y. \llbracket x \in A; y \in B; c = \text{Scons } x y \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *uprodE2*: $\llbracket \text{Scons } M N \in \text{uprod } A B; \llbracket M \in A; N \in B \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *usum-In0I* [*intro*]: $M \in A \implies \text{In0}(M) \in \text{usum } A \ B$
 $\langle \text{proof} \rangle$

lemma *usum-In1I* [*intro*]: $N \in B \implies \text{In1}(N) \in \text{usum } A \ B$
 $\langle \text{proof} \rangle$

lemma *usumE* [*elim!*]:
 $\llbracket u \in \text{usum } A \ B; \wedge x. \llbracket x \in A; u = \text{In0}(x) \rrbracket \implies P; \wedge y. \llbracket y \in B; u = \text{In1}(y) \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *In0-not-In1* [*iff*]: $\text{In0}(M) \neq \text{In1}(N)$
 $\langle \text{proof} \rangle$

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*]

lemma *In0-inject*: $\text{In0}(M) = \text{In0}(N) \implies M = N$
 $\langle \text{proof} \rangle$

lemma *In1-inject*: $\text{In1}(M) = \text{In1}(N) \implies M = N$
 $\langle \text{proof} \rangle$

lemma *In0-eq* [*iff*]: $(\text{In0 } M = \text{In0 } N) = (M = N)$
 $\langle \text{proof} \rangle$

lemma *In1-eq* [*iff*]: $(\text{In1 } M = \text{In1 } N) = (M = N)$
 $\langle \text{proof} \rangle$

lemma *inj-In0*: *inj In0*
 $\langle \text{proof} \rangle$

lemma *inj-In1*: *inj In1*
 $\langle \text{proof} \rangle$

lemma *Lim-inject*: $\text{Lim } f = \text{Lim } g \implies f = g$
 $\langle \text{proof} \rangle$

lemma *ntrunc-subsetI*: $\text{ntrunc } k \ M \leq M$

$\langle \text{proof} \rangle$

lemma *ntrunc-subsetD*: $(!!k. \text{ntrunc } k \ M \leq N) \implies M \leq N$

$\langle \text{proof} \rangle$

lemma *ntrunc-equality*: $(!!k. \text{ntrunc } k \ M = \text{ntrunc } k \ N) \implies M = N$

$\langle \text{proof} \rangle$

lemma *ntrunc-o-equality*:

$[! \text{!!}k. (\text{ntrunc}(k) \circ h1) = (\text{ntrunc}(k) \circ h2)] \implies h1 = h2$

$\langle \text{proof} \rangle$

lemma *uprod-mono*: $[! \ A \leq A'; \ B \leq B'] \implies \text{uprod } A \ B \leq \text{uprod } A' \ B'$

$\langle \text{proof} \rangle$

lemma *usum-mono*: $[! \ A \leq A'; \ B \leq B'] \implies \text{usum } A \ B \leq \text{usum } A' \ B'$

$\langle \text{proof} \rangle$

lemma *Scons-mono*: $[! \ M \leq M'; \ N \leq N'] \implies \text{Scons } M \ N \leq \text{Scons } M' \ N'$

$\langle \text{proof} \rangle$

lemma *In0-mono*: $M \leq N \implies \text{In0}(M) \leq \text{In0}(N)$

$\langle \text{proof} \rangle$

lemma *In1-mono*: $M \leq N \implies \text{In1}(M) \leq \text{In1}(N)$

$\langle \text{proof} \rangle$

lemma *Split [simp]*: $\text{Split } c \ (\text{Scons } M \ N) = c \ M \ N$

$\langle \text{proof} \rangle$

lemma *Case-In0 [simp]*: $\text{Case } c \ d \ (\text{In0 } M) = c(M)$

$\langle \text{proof} \rangle$

lemma *Case-In1 [simp]*: $\text{Case } c \ d \ (\text{In1 } N) = d(N)$

$\langle \text{proof} \rangle$

lemma *ntrunc-UN1*: $ntrunc\ k\ (UN\ x.\ f(x)) = (UN\ x.\ ntrunc\ k\ (f\ x))$
 $\langle proof \rangle$

lemma *Scons-UN1-x*: $Scons\ (UN\ x.\ f\ x)\ M = (UN\ x.\ Scons\ (f\ x)\ M)$
 $\langle proof \rangle$

lemma *Scons-UN1-y*: $Scons\ M\ (UN\ x.\ f\ x) = (UN\ x.\ Scons\ M\ (f\ x))$
 $\langle proof \rangle$

lemma *In0-UN1*: $In0\ (UN\ x.\ f(x)) = (UN\ x.\ In0(f(x)))$
 $\langle proof \rangle$

lemma *In1-UN1*: $In1\ (UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
 $\langle proof \rangle$

lemma *dprodI* [*intro!*]:
 $\llbracket (M, M') \in r; (N, N') \in s \rrbracket \implies (Scons\ M\ N, Scons\ M'\ N') \in dprod\ r\ s$
 $\langle proof \rangle$

lemma *dprodE* [*elim!*]:
 $\llbracket c \in dprod\ r\ s; \bigwedge x\ y\ x'\ y'. \llbracket (x, x') \in r; (y, y') \in s; c = (Scons\ x\ y, Scons\ x'\ y') \rrbracket \implies P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *dsum-In0I* [*intro*]: $(M, M') \in r \implies (In0(M), In0(M')) \in dsum\ r\ s$
 $\langle proof \rangle$

lemma *dsum-In1I* [*intro*]: $(N, N') \in s \implies (In1(N), In1(N')) \in dsum\ r\ s$
 $\langle proof \rangle$

lemma *dsumE* [*elim!*]:
 $\llbracket w \in dsum\ r\ s; \bigwedge x\ x'. \llbracket (x, x') \in r; w = (In0(x), In0(x')) \rrbracket \implies P; \bigwedge y\ y'. \llbracket (y, y') \in s; w = (In1(y), In1(y')) \rrbracket \implies P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *dprod-mono*: $[[\ r \leq r';\ s \leq s'\]] \implies dprod\ r\ s \leq dprod\ r'\ s'$
 $\langle proof \rangle$

lemma *dsum-mono*: $[[\ r \leq r';\ s \leq s'\]] \implies dsum\ r\ s \leq dsum\ r'\ s'$
 $\langle proof \rangle$

lemma *dprod-Sigma*: $(dprod\ (A \times B)\ (C \times D)) \leq (uprod\ A\ C) \times (uprod\ B\ D)$
 $\langle proof \rangle$

lemmas *dprod-subset-Sigma* = *subset-trans* [OF *dprod-mono* *dprod-Sigma*]

lemma *dprod-subset-Sigma2*:
 $(dprod\ (Sigma\ A\ B)\ (Sigma\ C\ D)) \leq Sigma\ (uprod\ A\ C)\ (Split\ (\%x\ y.\ uprod\ (B\ x)\ (D\ y)))$
 $\langle proof \rangle$

lemma *dsum-Sigma*: $(dsum\ (A \times B)\ (C \times D)) \leq (usum\ A\ C) \times (usum\ B\ D)$
 $\langle proof \rangle$

lemmas *dsum-subset-Sigma* = *subset-trans* [OF *dsum-mono* *dsum-Sigma*]

lemma *Domain-dprod* [simp]: $Domain\ (dprod\ r\ s) = uprod\ (Domain\ r)\ (Domain\ s)$
 $\langle proof \rangle$

lemma *Domain-dsum* [simp]: $Domain\ (dsum\ r\ s) = usum\ (Domain\ r)\ (Domain\ s)$
 $\langle proof \rangle$

hides popular names

hide-type (**open**) *node item*

hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

$\langle ML \rangle$

end

18 Bijections between natural numbers and other types

theory *Nat-Bijection*


```

imports Main
begin

```

18.1 Type $\text{nat} \times \text{nat}$

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

```

definition triangle :: nat  $\Rightarrow$  nat
  where triangle n = (n * Suc n) div 2

```

```

lemma triangle-0 [simp]: triangle 0 = 0
  <proof>

```

```

lemma triangle-Suc [simp]: triangle (Suc n) = triangle n + Suc n
  <proof>

```

```

definition prod-encode :: nat  $\times$  nat  $\Rightarrow$  nat
  where prod-encode = ( $\lambda(m, n).$  triangle (m + n) + m)

```

In this auxiliary function, $\text{triangle } k + m$ is an invariant.

```

fun prod-decode-aux :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\times$  nat
  where prod-decode-aux k m =
    (if m  $\leq$  k then (m, k - m) else prod-decode-aux (Suc k) (m - Suc k))

```

```

declare prod-decode-aux.simps [simp del]

```

```

definition prod-decode :: nat  $\Rightarrow$  nat  $\times$  nat
  where prod-decode = prod-decode-aux 0

```

```

lemma prod-encode-prod-decode-aux: prod-encode (prod-decode-aux k m) = triangle
k + m
  <proof>

```

```

lemma prod-decode-inverse [simp]: prod-encode (prod-decode n) = n
  <proof>

```

```

lemma prod-decode-triangle-add: prod-decode (triangle k + m) = prod-decode-aux
k m
  <proof>

```

```

lemma prod-encode-inverse [simp]: prod-decode (prod-encode x) = x
  <proof>

```

```

lemma inj-prod-encode: inj-on prod-encode A
  <proof>

```

```

lemma inj-prod-decode: inj-on prod-decode A
  <proof>

```

lemma *surj-prod-encode*: *surj prod-encode*
 $\langle \text{proof} \rangle$

lemma *surj-prod-decode*: *surj prod-decode*
 $\langle \text{proof} \rangle$

lemma *bij-prod-encode*: *bij prod-encode*
 $\langle \text{proof} \rangle$

lemma *bij-prod-decode*: *bij prod-decode*
 $\langle \text{proof} \rangle$

lemma *prod-encode-eq* [simp]: *prod-encode* $x = \text{prod-encode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *prod-decode-eq* [simp]: *prod-decode* $x = \text{prod-decode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

Ordering properties

lemma *le-prod-encode-1*: $a \leq \text{prod-encode } (a, b)$
 $\langle \text{proof} \rangle$

lemma *le-prod-encode-2*: $b \leq \text{prod-encode } (a, b)$
 $\langle \text{proof} \rangle$

18.2 Type $\text{nat} + \text{nat}$

definition *sum-encode* :: $\text{nat} + \text{nat} \Rightarrow \text{nat}$
where *sum-encode* $x = (\text{case } x \text{ of } \text{Inl } a \Rightarrow 2 * a \mid \text{Inr } b \Rightarrow \text{Suc } (2 * b))$

definition *sum-decode* :: $\text{nat} \Rightarrow \text{nat} + \text{nat}$
where *sum-decode* $n = (\text{if even } n \text{ then } \text{Inl } (n \text{ div } 2) \text{ else } \text{Inr } (n \text{ div } 2))$

lemma *sum-encode-inverse* [simp]: *sum-decode* (*sum-encode* x) = x
 $\langle \text{proof} \rangle$

lemma *sum-decode-inverse* [simp]: *sum-encode* (*sum-decode* n) = n
 $\langle \text{proof} \rangle$

lemma *inj-sum-encode*: *inj-on sum-encode* A
 $\langle \text{proof} \rangle$

lemma *inj-sum-decode*: *inj-on sum-decode* A
 $\langle \text{proof} \rangle$

lemma *surj-sum-encode*: *surj sum-encode*
 $\langle \text{proof} \rangle$

lemma *surj-sum-decode*: *surj sum-decode*
 $\langle \text{proof} \rangle$

lemma *bij-sum-encode*: *bij sum-encode*
 $\langle \text{proof} \rangle$

lemma *bij-sum-decode*: *bij sum-decode*
 $\langle \text{proof} \rangle$

lemma *sum-encode-eq*: *sum-encode* $x = \text{sum-encode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *sum-decode-eq*: *sum-decode* $x = \text{sum-decode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

18.3 Type *int*

definition *int-encode* :: *int* \Rightarrow *nat*
where *int-encode* $i = \text{sum-encode } (if\ 0 \leq i\ \text{then } Inl\ (nat\ i)\ \text{else } Inr\ (nat\ (-\ i - 1)))$

definition *int-decode* :: *nat* \Rightarrow *int*
where *int-decode* $n = (\text{case } \text{sum-decode } n\ \text{of } Inl\ a \Rightarrow int\ a \mid Inr\ b \Rightarrow -\ int\ b - 1)$

lemma *int-encode-inverse* [*simp*]: *int-decode* (*int-encode* x) = x
 $\langle \text{proof} \rangle$

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode* n) = n
 $\langle \text{proof} \rangle$

lemma *inj-int-encode*: *inj-on int-encode* A
 $\langle \text{proof} \rangle$

lemma *inj-int-decode*: *inj-on int-decode* A
 $\langle \text{proof} \rangle$

lemma *surj-int-encode*: *surj int-encode*
 $\langle \text{proof} \rangle$

lemma *surj-int-decode*: *surj int-decode*
 $\langle \text{proof} \rangle$

lemma *bij-int-encode*: *bij int-encode*
 $\langle \text{proof} \rangle$

lemma *bij-int-decode*: *bij int-decode*
 $\langle \text{proof} \rangle$

lemma *int-encode-eq*: *int-encode* $x = \text{int-encode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *int-decode-eq*: $\text{int-decode } x = \text{int-decode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

18.4 Type *nat list*

fun *list-encode* :: $\text{nat list} \Rightarrow \text{nat}$
where
 $\text{list-encode } [] = 0$
 $| \text{list-encode } (x \# xs) = \text{Suc } (\text{prod-encode } (x, \text{list-encode } xs))$

function *list-decode* :: $\text{nat} \Rightarrow \text{nat list}$
where
 $\text{list-decode } 0 = []$
 $| \text{list-decode } (\text{Suc } n) = (\text{case prod-decode } n \text{ of } (x, y) \Rightarrow x \# \text{list-decode } y)$
 $\langle \text{proof} \rangle$

termination *list-decode*
 $\langle \text{proof} \rangle$

lemma *list-encode-inverse* [*simp*]: $\text{list-decode } (\text{list-encode } x) = x$
 $\langle \text{proof} \rangle$

lemma *list-decode-inverse* [*simp*]: $\text{list-encode } (\text{list-decode } n) = n$
 $\langle \text{proof} \rangle$

lemma *inj-list-encode*: *inj-on list-encode A*
 $\langle \text{proof} \rangle$

lemma *inj-list-decode*: *inj-on list-decode A*
 $\langle \text{proof} \rangle$

lemma *surj-list-encode*: *surj list-encode*
 $\langle \text{proof} \rangle$

lemma *surj-list-decode*: *surj list-decode*
 $\langle \text{proof} \rangle$

lemma *bij-list-encode*: *bij list-encode*
 $\langle \text{proof} \rangle$

lemma *bij-list-decode*: *bij list-decode*
 $\langle \text{proof} \rangle$

lemma *list-encode-eq*: $\text{list-encode } x = \text{list-encode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *list-decode-eq*: $\text{list-decode } x = \text{list-decode } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

18.5 Finite sets of naturals

18.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: $\text{finite } (\text{Suc } -' F) \longleftrightarrow \text{finite } F$
 $\langle \text{proof} \rangle$

lemma *vimage-Suc-insert-0*: $\text{Suc } -' \text{insert } 0 A = \text{Suc } -' A$
 $\langle \text{proof} \rangle$

lemma *vimage-Suc-insert-Suc*: $\text{Suc } -' \text{insert } (\text{Suc } n) A = \text{insert } n (\text{Suc } -' A)$
 $\langle \text{proof} \rangle$

lemma *div2-even-ext-nat*:
fixes $x y :: \text{nat}$
assumes $x \text{ div } 2 = y \text{ div } 2$
and $\text{even } x \longleftrightarrow \text{even } y$
shows $x = y$
 $\langle \text{proof} \rangle$

18.5.2 From sets to naturals

definition *set-encode* :: $\text{nat set} \Rightarrow \text{nat}$
where $\text{set-encode} = \text{sum } ((\hat{\ }) 2)$

lemma *set-encode-empty [simp]*: $\text{set-encode } \{\} = 0$
 $\langle \text{proof} \rangle$

lemma *set-encode-inf*: $\neg \text{finite } A \Longrightarrow \text{set-encode } A = 0$
 $\langle \text{proof} \rangle$

lemma *set-encode-insert [simp]*: $\text{finite } A \Longrightarrow n \notin A \Longrightarrow \text{set-encode } (\text{insert } n A) = 2^{\hat{n}} + \text{set-encode } A$
 $\langle \text{proof} \rangle$

lemma *even-set-encode-iff*: $\text{finite } A \Longrightarrow \text{even } (\text{set-encode } A) \longleftrightarrow 0 \notin A$
 $\langle \text{proof} \rangle$

lemma *set-encode-vimage-Suc*: $\text{set-encode } (\text{Suc } -' A) = \text{set-encode } A \text{ div } 2$
 $\langle \text{proof} \rangle$

lemmas $\text{set-encode-div-2} = \text{set-encode-vimage-Suc}$ [symmetric]

18.5.3 From naturals to sets

definition *set-decode* :: $\text{nat} \Rightarrow \text{nat set}$
where $\text{set-decode } x = \{n. \text{odd } (x \text{ div } 2 \hat{\ } n)\}$

lemma *set-decode-0 [simp]*: $0 \in \text{set-decode } x \longleftrightarrow \text{odd } x$
 $\langle \text{proof} \rangle$

lemma *set-decode-Suc* [simp]: $\text{Suc } n \in \text{set-decode } x \longleftrightarrow n \in \text{set-decode } (x \text{ div } 2)$
 ⟨proof⟩

lemma *set-decode-zero* [simp]: $\text{set-decode } 0 = \{\}$
 ⟨proof⟩

lemma *set-decode-div-2*: $\text{set-decode } (x \text{ div } 2) = \text{Suc } -' \text{ set-decode } x$
 ⟨proof⟩

lemma *set-decode-plus-power-2*:
 $n \notin \text{set-decode } z \implies \text{set-decode } (2^n + z) = \text{insert } n (\text{set-decode } z)$
 ⟨proof⟩

lemma *finite-set-decode* [simp]: $\text{finite } (\text{set-decode } n)$
 ⟨proof⟩

18.5.4 Proof of isomorphism

lemma *set-decode-inverse* [simp]: $\text{set-encode } (\text{set-decode } n) = n$
 ⟨proof⟩

lemma *set-encode-inverse* [simp]: $\text{finite } A \implies \text{set-decode } (\text{set-encode } A) = A$
 ⟨proof⟩

lemma *inj-on-set-encode*: $\text{inj-on set-encode } (\text{Collect finite})$
 ⟨proof⟩

lemma *set-encode-eq*: $\text{finite } A \implies \text{finite } B \implies \text{set-encode } A = \text{set-encode } B \longleftrightarrow A = B$
 ⟨proof⟩

lemma *subset-decode-imp-le*:
 assumes $\text{set-decode } m \subseteq \text{set-decode } n$
 shows $m \leq n$
 ⟨proof⟩

end

19 Encoding (almost) everything into natural numbers

theory *Countable*
imports *Old-Datatype HOL.Rat Nat-Bijection*
begin

19.1 The class of countable types

class *countable* =

assumes *ex-inj*: $\exists \text{to-nat} :: 'a \Rightarrow \text{nat}. \text{inj to-nat}$

lemma *countable-classI*:

fixes *f* :: $'a \Rightarrow \text{nat}$

assumes $\bigwedge x y. f\ x = f\ y \implies x = y$

shows *OFCLASS*('a, *countable-class*)

<proof>

19.2 Conversion functions

definition *to-nat* :: $'a::\text{countable} \Rightarrow \text{nat}$ **where**

to-nat = (*SOME f. inj f*)

definition *from-nat* :: $\text{nat} \Rightarrow 'a::\text{countable}$ **where**

from-nat = *inv (to-nat :: 'a \Rightarrow nat)*

lemma *inj-to-nat [simp]*: *inj to-nat*

<proof>

lemma *inj-on-to-nat[simp, intro]*: *inj-on to-nat S*

<proof>

lemma *surj-from-nat [simp]*: *surj from-nat*

<proof>

lemma *to-nat-split [simp]*: *to-nat x = to-nat y \longleftrightarrow x = y*

<proof>

lemma *from-nat-to-nat [simp]*:

from-nat (to-nat x) = x

<proof>

19.3 Finite types are countable

subclass (in *finite*) *countable*

<proof>

19.4 Automatically proving countability of old-style datatypes

context

begin

qualified inductive *finite-item* :: $'a \text{ Old-Datatype.item} \Rightarrow \text{bool}$ **where**

undefined: finite-item undefined

| *In0*: *finite-item x \implies finite-item (Old-Datatype.In0 x)*

| *In1*: *finite-item x \implies finite-item (Old-Datatype.In1 x)*

| *Leaf*: *finite-item (Old-Datatype.Leaf a)*

| *Scons*: $\llbracket \text{finite-item } x; \text{finite-item } y \rrbracket \implies \text{finite-item (Old-Datatype.Scons } x\ y)$

qualified function *nth-item* :: $\text{nat} \Rightarrow ('a::\text{countable}) \text{ Old-Datatype.item}$

where

```

  nth-item 0 = undefined
| nth-item (Suc n) =
  (case sum-decode n of
    Inl i ⇒
      (case sum-decode i of
        Inl j ⇒ Old-Datatype.In0 (nth-item j)
      | Inr j ⇒ Old-Datatype.In1 (nth-item j))
    | Inr i ⇒
      (case sum-decode i of
        Inl j ⇒ Old-Datatype.Leaf (from-nat j)
      | Inr j ⇒
          (case prod-decode j of
            (a, b) ⇒ Old-Datatype.Scons (nth-item a) (nth-item b))))
⟨proof⟩

```

lemma *le-sum-encode-Inl*: $x \leq y \implies x \leq \text{sum-encode } (\text{Inl } y)$
 ⟨proof⟩

lemma *le-sum-encode-Inr*: $x \leq y \implies x \leq \text{sum-encode } (\text{Inr } y)$
 ⟨proof⟩ **termination**
 ⟨proof⟩

lemma *nth-item-covers*: $\text{finite-item } x \implies \exists n. \text{nth-item } n = x$
 ⟨proof⟩

theorem *countable-datatype*:

```

  fixes Rep :: 'b ⇒ ('a::countable) Old-Datatype.item
  fixes Abs :: ('a::countable) Old-Datatype.item ⇒ 'b
  fixes rep-set :: ('a::countable) Old-Datatype.item ⇒ bool
  assumes type: type-definition Rep Abs (Collect rep-set)
  assumes finite-item:  $\bigwedge x. \text{rep-set } x \implies \text{finite-item } x$ 
  shows OFCLASS('b, countable-class)
⟨proof⟩

```

⟨ML⟩

end

19.5 Automatically proving countability of datatypes

⟨ML⟩

19.6 More Countable types

Naturals

instance *nat* :: *countable*
 ⟨proof⟩

Pairs

instance *prod* :: (*countable*, *countable*) *countable*
 ⟨*proof*⟩

Sums

instance *sum* :: (*countable*, *countable*) *countable*
 ⟨*proof*⟩

Integers

instance *int* :: *countable*
 ⟨*proof*⟩

Options

instance *option* :: (*countable*) *countable*
 ⟨*proof*⟩

Lists

instance *list* :: (*countable*) *countable*
 ⟨*proof*⟩

String literals

instance *String.literal* :: *countable*
 ⟨*proof*⟩

Functions

instance *fun* :: (*finite*, *countable*) *countable*
 ⟨*proof*⟩

Typereps

instance *typerep* :: *countable*
 ⟨*proof*⟩

19.7 The rationals are countably infinite

definition *nat-to-rat-surj* :: *nat* \Rightarrow *rat* **where**
nat-to-rat-surj *n* = (let (*a*, *b*) = *prod-decode* *n* in *Fract* (*int-decode* *a*) (*int-decode* *b*))

lemma *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*
 ⟨*proof*⟩

lemma *Rats-eq-range-nat-to-rat-surj*: $\mathbb{Q} = \text{range } \text{nat-to-rat-surj}$
 ⟨*proof*⟩

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
 $\mathbb{Q} = \text{range } (\text{of-rat} \circ \text{nat-to-rat-surj})$
 ⟨*proof*⟩

lemma *surj-of-rat-nat-to-rat-surj*:
 $r \in \mathbf{Q} \implies \exists n. r = \text{of-rat } (\text{nat-to-rat-surj } n)$
 $\langle \text{proof} \rangle$

end

instance *rat :: countable*
 $\langle \text{proof} \rangle$

theorem *rat-denum*: $\exists f :: \text{nat} \Rightarrow \text{rat}. \text{surj } f$
 $\langle \text{proof} \rangle$

end

20 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

20.1 The set of natural numbers is infinite

lemma *infinite-nat-iff-unbounded-le*: $\text{infinite } S \longleftrightarrow (\forall m. \exists n \geq m. n \in S)$
for $S :: \text{nat set}$
 $\langle \text{proof} \rangle$

lemma *infinite-nat-iff-unbounded*: $\text{infinite } S \longleftrightarrow (\forall m. \exists n > m. n \in S)$
for $S :: \text{nat set}$
 $\langle \text{proof} \rangle$

lemma *finite-nat-iff-bounded*: $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{..<k\})$
for $S :: \text{nat set}$
 $\langle \text{proof} \rangle$

lemma *finite-nat-iff-bounded-le*: $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{.. k\})$
for $S :: \text{nat set}$
 $\langle \text{proof} \rangle$

lemma *finite-nat-bounded*: $\text{finite } S \implies \exists k. S \subseteq \{..<k\}$
for $S :: \text{nat set}$
 $\langle \text{proof} \rangle$

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k , there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*: $\forall m > k. \exists n > m. n \in S \implies \text{infinite } (S :: \text{nat set})$
 $\langle \text{proof} \rangle$

lemma *nat-not-finite*: $\text{finite } (\text{UNIV}::\text{nat set}) \implies R$
 $\langle \text{proof} \rangle$

lemma *range-inj-infinite*:
fixes $f :: \text{nat} \Rightarrow 'a$
assumes $\text{inj } f$
shows $\text{infinite } (\text{range } f)$
 $\langle \text{proof} \rangle$

20.2 The set of integers is also infinite

lemma *infinite-int-iff-infinite-nat-abs*: $\text{infinite } S \longleftrightarrow \text{infinite } ((\text{nat} \circ \text{abs}) ' S)$
for $S :: \text{int set}$
 $\langle \text{proof} \rangle$

proposition *infinite-int-iff-unbounded-le*: $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| \geq m \wedge n \in S)$
for $S :: \text{int set}$
 $\langle \text{proof} \rangle$

proposition *infinite-int-iff-unbounded*: $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| > m \wedge n \in S)$
for $S :: \text{int set}$
 $\langle \text{proof} \rangle$

proposition *finite-int-iff-bounded*: $\text{finite } S \longleftrightarrow (\exists k. \text{abs } ' S \subseteq \{..<k\})$
for $S :: \text{int set}$
 $\langle \text{proof} \rangle$

proposition *finite-int-iff-bounded-le*: $\text{finite } S \longleftrightarrow (\exists k. \text{abs } ' S \subseteq \{.. k\})$
for $S :: \text{int set}$
 $\langle \text{proof} \rangle$

lemma *infinite-split*: — courtesy of Michael Schmidt
fixes $S :: 'a \text{ set}$
assumes $\text{infinite } S$
obtains $A B$
where $A \subseteq S \ B \subseteq S \ \text{infinite } A \ \text{infinite } B \ A \cap B = \{\}$
 $\langle \text{proof} \rangle$

20.3 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM [simp]*: $\neg (\text{INFM } x. P x) \longleftrightarrow (\text{MOST } x. \neg P x)$
 $\langle \text{proof} \rangle$

lemma *not-MOST [simp]*: $\neg (\text{MOST } x. P x) \longleftrightarrow (\text{INFM } x. \neg P x)$

$\langle proof \rangle$

lemma *INFM-const* [simp]: $(INFM\ x::'a.\ P) \longleftrightarrow P \wedge infinite\ (UNIV::'a\ set)$
 $\langle proof \rangle$

lemma *MOST-const* [simp]: $(MOST\ x::'a.\ P) \longleftrightarrow P \vee finite\ (UNIV::'a\ set)$
 $\langle proof \rangle$

lemma *INFM-imp-distrib*: $(INFM\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow ((MOST\ x.\ P\ x) \longrightarrow (INFM\ x.\ Q\ x))$
 $\langle proof \rangle$

lemma *MOST-imp-iff*: $MOST\ x.\ P\ x \Longrightarrow (MOST\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow (MOST\ x.\ Q\ x)$
 $\langle proof \rangle$

lemma *INFM-conjI*: $INFM\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow INFM\ x.\ P\ x \wedge Q\ x$
 $\langle proof \rangle$

Properties of quantifiers with injective functions.

lemma *INFM-inj*: $INFM\ x.\ P\ (f\ x) \Longrightarrow inj\ f \Longrightarrow INFM\ x.\ P\ x$
 $\langle proof \rangle$

lemma *MOST-inj*: $MOST\ x.\ P\ x \Longrightarrow inj\ f \Longrightarrow MOST\ x.\ P\ (f\ x)$
 $\langle proof \rangle$

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [simp]:
 $\neg (INFM\ x.\ x = a)$
 $\neg (INFM\ x.\ a = x)$
 $\langle proof \rangle$

lemma *MOST-neq* [simp]:
 $MOST\ x.\ x \neq a$
 $MOST\ x.\ a \neq x$
 $\langle proof \rangle$

lemma *INFM-neq* [simp]:
 $(INFM\ x::'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV::'a\ set)$
 $(INFM\ x::'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV::'a\ set)$
 $\langle proof \rangle$

lemma *MOST-eq* [simp]:
 $(MOST\ x::'a.\ x = a) \longleftrightarrow finite\ (UNIV::'a\ set)$
 $(MOST\ x::'a.\ a = x) \longleftrightarrow finite\ (UNIV::'a\ set)$
 $\langle proof \rangle$

lemma *MOST-eq-imp*:
 $MOST\ x.\ x = a \longrightarrow P\ x$

MOST $x. a = x \longrightarrow P x$
 $\langle \text{proof} \rangle$

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_{\infty} n. P n) \longleftrightarrow (\exists m. \forall n > m. P n)$
for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

lemma *MOST-nat-le*: $(\forall_{\infty} n. P n) \longleftrightarrow (\exists m. \forall n \geq m. P n)$
for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

lemma *INFM-nat*: $(\exists_{\infty} n. P n) \longleftrightarrow (\forall m. \exists n > m. P n)$
for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

lemma *INFM-nat-le*: $(\exists_{\infty} n. P n) \longleftrightarrow (\forall m. \exists n \geq m. P n)$
for $P :: \text{nat} \Rightarrow \text{bool}$
 $\langle \text{proof} \rangle$

lemma *MOST-INFM*: $\text{infinite } (UNIV :: 'a \text{ set}) \Longrightarrow \text{MOST } x :: 'a. P x \Longrightarrow \text{INFM } x :: 'a. P x$
 $\langle \text{proof} \rangle$

lemma *MOST-Suc-iff*: $(\text{MOST } n. P (\text{Suc } n)) \longleftrightarrow (\text{MOST } n. P n)$
 $\langle \text{proof} \rangle$

lemma *MOST-SucI*: $\text{MOST } n. P n \Longrightarrow \text{MOST } n. P (\text{Suc } n)$
and *MOST-SucD*: $\text{MOST } n. P (\text{Suc } n) \Longrightarrow \text{MOST } n. P n$
 $\langle \text{proof} \rangle$

lemma *MOST-ge-nat*: $\text{MOST } n :: \text{nat}. m \leq n$
 $\langle \text{proof} \rangle$

lemma *Inf-many-def*: $\text{Inf-many } P \longleftrightarrow \text{infinite } \{x. P x\}$ $\langle \text{proof} \rangle$

lemma *Alm-all-def*: $\text{Alm-all } P \longleftrightarrow \neg (\text{INFM } x. \neg P x)$ $\langle \text{proof} \rangle$

lemma *INFM-iff-infinite*: $(\text{INFM } x. P x) \longleftrightarrow \text{infinite } \{x. P x\}$ $\langle \text{proof} \rangle$

lemma *MOST-iff-cofinite*: $(\text{MOST } x. P x) \longleftrightarrow \text{finite } \{x. \neg P x\}$ $\langle \text{proof} \rangle$

lemma *INFM-EX*: $(\exists_{\infty} x. P x) \Longrightarrow (\exists x. P x)$ $\langle \text{proof} \rangle$

lemma *ALL-MOST*: $\forall x. P x \Longrightarrow \forall_{\infty} x. P x$ $\langle \text{proof} \rangle$

lemma *INFM-mono*: $\exists_{\infty} x. P x \Longrightarrow (\bigwedge x. P x \Longrightarrow Q x) \Longrightarrow \exists_{\infty} x. Q x$ $\langle \text{proof} \rangle$

lemma *MOST-mono*: $\forall_{\infty} x. P x \Longrightarrow (\bigwedge x. P x \Longrightarrow Q x) \Longrightarrow \forall_{\infty} x. Q x$ $\langle \text{proof} \rangle$

lemma *INFM-disj-distrib*: $(\exists_{\infty} x. P x \vee Q x) \longleftrightarrow (\exists_{\infty} x. P x) \vee (\exists_{\infty} x. Q x)$
 $\langle \text{proof} \rangle$

lemma *MOST-rev-mp*: $\forall_{\infty} x. P x \Longrightarrow \forall_{\infty} x. P x \longrightarrow Q x \Longrightarrow \forall_{\infty} x. Q x$ $\langle \text{proof} \rangle$

lemma *MOST-conj-distrib*: $(\forall_{\infty} x. P x \wedge Q x) \longleftrightarrow (\forall_{\infty} x. P x) \wedge (\forall_{\infty} x. Q x)$
 $\langle \text{proof} \rangle$

lemma *MOST-conjI*: $\text{MOST } x. P x \Longrightarrow \text{MOST } x. Q x \Longrightarrow \text{MOST } x. P x \wedge Q x$
 $\langle \text{proof} \rangle$

lemma *INFM-finite-Bex-distrib*: $\text{finite } A \Longrightarrow (\text{INFM } y. \exists x \in A. P x y) \longleftrightarrow (\exists x \in A.$

INFM $y. P\ x\ y$ $\langle \text{proof} \rangle$

lemma *MOST-finite-Ball-distrib*: $\text{finite } A \implies (\text{MOST } y. \forall x \in A. P\ x\ y) \longleftrightarrow (\forall x \in A. \text{MOST } y. P\ x\ y)$ $\langle \text{proof} \rangle$

lemma *INFM-E*: $\text{INFM } x. P\ x \implies (\bigwedge x. P\ x \implies \text{thesis}) \implies \text{thesis}$ $\langle \text{proof} \rangle$

lemma *MOST-I*: $(\bigwedge x. P\ x) \implies \text{MOST } x. P\ x$ $\langle \text{proof} \rangle$

lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

20.4 Enumeration of an Infinite Set

The set’s element type must be wellordered (e.g. the natural numbers).

Could be generalized to $\text{enumerate}'\ S\ n = (\text{SOME } t. t \in s \wedge \text{finite } \{s \in S. s < t\} \wedge \text{card } \{s \in S. s < t\} = n)$.

primrec (*in wellorder*) *enumerate* :: $'a\ \text{set} \Rightarrow \text{nat} \Rightarrow 'a$

where

enumerate-0: $\text{enumerate } S\ 0 = (\text{LEAST } n. n \in S)$

| *enumerate-Suc*: $\text{enumerate } S\ (\text{Suc } n) = \text{enumerate } (S - \{\text{LEAST } n. n \in S\})\ n$

lemma *enumerate-Suc'*: $\text{enumerate } S\ (\text{Suc } n) = \text{enumerate } (S - \{\text{enumerate } S\ 0\})\ n$
 $\langle \text{proof} \rangle$

lemma *enumerate-in-set*: $\text{infinite } S \implies \text{enumerate } S\ n \in S$
 $\langle \text{proof} \rangle$

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: $\text{infinite } S \implies \text{enumerate } S\ n < \text{enumerate } S\ (\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *enumerate-mono*: $m < n \implies \text{infinite } S \implies \text{enumerate } S\ m < \text{enumerate } S\ n$
 $\langle \text{proof} \rangle$

lemma *enumerate-mono-iff* [*simp*]:
 $\text{infinite } S \implies \text{enumerate } S\ m < \text{enumerate } S\ n \longleftrightarrow m < n$
 $\langle \text{proof} \rangle$

lemma *enumerate-mono-le-iff* [*simp*]:
 $\text{infinite } S \implies \text{enumerate } S\ m \leq \text{enumerate } S\ n \longleftrightarrow m \leq n$
 $\langle \text{proof} \rangle$

lemma *le-enumerate*:

assumes S : $\text{infinite } S$

shows $n \leq \text{enumerate } S\ n$

$\langle \text{proof} \rangle$

lemma *infinite-enumerate*:

assumes fS : $\text{infinite } S$

shows $\exists r :: \text{nat} \Rightarrow \text{nat}. \text{strict-mono } r \wedge (\forall n. r\ n \in S)$
 $\langle \text{proof} \rangle$

lemma *enumerate-Suc''*:
fixes $S :: 'a :: \text{wellorder set}$
assumes *infinite* S
shows $\text{enumerate } S (\text{Suc } n) = (\text{LEAST } s. s \in S \wedge \text{enumerate } S\ n < s)$
 $\langle \text{proof} \rangle$

lemma *enumerate-Ex*:
fixes $S :: \text{nat set}$
assumes $S: \text{infinite } S$
and $s: s \in S$
shows $\exists n. \text{enumerate } S\ n = s$
 $\langle \text{proof} \rangle$

lemma *inj-enumerate*:
fixes $S :: 'a :: \text{wellorder set}$
assumes $S: \text{infinite } S$
shows $\text{inj } (\text{enumerate } S)$
 $\langle \text{proof} \rangle$

To generalise this, we’d need a condition that all initial segments were finite

lemma *bij-enumerate*:
fixes $S :: \text{nat set}$
assumes $S: \text{infinite } S$
shows $\text{bij-betw } (\text{enumerate } S) \text{ UNIV } S$
 $\langle \text{proof} \rangle$

lemma
fixes $S :: \text{nat set}$
assumes $S: \text{infinite } S$
shows $\text{range-enumerate: range } (\text{enumerate } S) = S$
and $\text{strict-mono-enumerate: strict-mono } (\text{enumerate } S)$
 $\langle \text{proof} \rangle$

A pair of weird and wonderful lemmas from HOL Light.

lemma *finite-transitivity-chain*:
assumes *finite* A
and $R: \bigwedge x. \neg R\ x\ x \wedge x\ y\ z. \llbracket R\ x\ y; R\ y\ z \rrbracket \Longrightarrow R\ x\ z$
and $A: \bigwedge x. x \in A \Longrightarrow \exists y. y \in A \wedge R\ x\ y$
shows $A = \{\}$
 $\langle \text{proof} \rangle$

corollary *Union-maximal-sets*:
assumes *finite* \mathcal{F}
shows $\bigcup \{T \in \mathcal{F}. \forall U \in \mathcal{F}. \neg T \subset U\} = \bigcup \mathcal{F}$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

20.5 Properties of *wellorder-class.enumerate* on finite sets

lemma *finite-enumerate-in-set*: $\llbracket \text{finite } S; n < \text{card } S \rrbracket \implies \text{enumerate } S \ n \in S$
 ⟨proof⟩

lemma *finite-enumerate-step*: $\llbracket \text{finite } S; \text{Suc } n < \text{card } S \rrbracket \implies \text{enumerate } S \ n < \text{enumerate } S \ (\text{Suc } n)$
 ⟨proof⟩

lemma *finite-enumerate-mono*: $\llbracket m < n; \text{finite } S; n < \text{card } S \rrbracket \implies \text{enumerate } S \ m < \text{enumerate } S \ n$
 ⟨proof⟩

lemma *finite-enumerate-mono-iff* [simp]:
 $\llbracket \text{finite } S; m < \text{card } S; n < \text{card } S \rrbracket \implies \text{enumerate } S \ m < \text{enumerate } S \ n \longleftrightarrow m < n$
 ⟨proof⟩

lemma *finite-le-enumerate*:
 assumes *finite* *S* *n* < *card* *S*
 shows *n* ≤ *enumerate* *S* *n*
 ⟨proof⟩

lemma *finite-enumerate*:
 assumes *fS*: *finite* *S*
 shows $\exists r :: \text{nat} \Rightarrow \text{nat. strict-mono-on } \{..<\text{card } S\} \ r \wedge (\forall n < \text{card } S. r \ n \in S)$
 ⟨proof⟩

lemma *finite-enumerate-Suc''*:
 fixes *S* :: 'a::wellorder set
 assumes *finite* *S* *Suc* *n* < *card* *S*
 shows *enumerate* *S* (*Suc* *n*) = (LEAST *s*. *s* ∈ *S* ∧ *enumerate* *S* *n* < *s*)
 ⟨proof⟩

lemma *finite-enumerate-initial-segment*:
 fixes *S* :: 'a::wellorder set
 assumes *finite* *S* and *n*: *n* < *card* (*S* ∩ {..*s*})
 shows *enumerate* (*S* ∩ {..*s*) *n* = *enumerate* *S* *n*
 ⟨proof⟩

lemma *finite-enumerate-Ex*:
 fixes *S* :: 'a::wellorder set
 assumes *S*: *finite* *S*
 and *s*: *s* ∈ *S*
 shows $\exists n < \text{card } S. \text{enumerate } S \ n = s$
 ⟨proof⟩

lemma *finite-enum-subset*:
 assumes $\bigwedge i. i < \text{card } X \implies \text{enumerate } X \ i = \text{enumerate } Y \ i$ and *finite* *X* *finite* *Y*
 $\text{card } X \leq \text{card } Y$

shows $X \subseteq Y$
 $\langle \text{proof} \rangle$

lemma *finite-enum-ext*:

assumes $\bigwedge i. i < \text{card } X \implies \text{enumerate } X \ i = \text{enumerate } Y \ i$ **and** *finite* X *finite* Y
 $\text{card } X = \text{card } Y$
shows $X = Y$
 $\langle \text{proof} \rangle$

lemma *finite-bij-enumerate*:

fixes $S :: 'a::\text{wellorder set}$
assumes S : *finite* S
shows *bij-betw* $(\text{enumerate } S) \ \{.. $\text{card } S\}$ S
 $\langle \text{proof} \rangle$$

lemma *ex-bij-betw-strict-mono-card*:

fixes $M :: 'a::\text{wellorder set}$
assumes *finite* M
obtains h **where** *bij-betw* $h \ \{.. $\text{card } M\}$ M **and** *strict-mono-on* $\{.. $\text{card } M\}$ h
 $\langle \text{proof} \rangle$$$

end

21 Countable sets

theory *Countable-Set*

imports *Countable Infinite-Set*

begin

21.1 Predicate for countable sets

definition *countable* $:: 'a \text{ set} \Rightarrow \text{bool}$ **where**

countable $S \longleftrightarrow (\exists f::'a \Rightarrow \text{nat}. \text{inj-on } f \ S)$

lemma *countable-as-injective-image-subset*: *countable* $S \longleftrightarrow (\exists f. \exists K::\text{nat set}. S = f \ ` \ K \wedge \text{inj-on } f \ K)$

$\langle \text{proof} \rangle$

lemma *countableE*:

assumes S : *countable* S **obtains** $f :: 'a \Rightarrow \text{nat}$ **where** *inj-on* $f \ S$
 $\langle \text{proof} \rangle$

lemma *countableI*: *inj-on* $(f::'a \Rightarrow \text{nat}) \ S \implies \text{countable } S$

$\langle \text{proof} \rangle$

lemma *countableI'*: *inj-on* $(f::'a \Rightarrow 'b::\text{countable}) \ S \implies \text{countable } S$

$\langle \text{proof} \rangle$

lemma *countableE-bij*:

assumes S : countable S **obtains** $f :: \text{nat} \Rightarrow 'a$ **and** $C :: \text{nat set}$ **where** $\text{bij-betw } f \ C \ S$
 $\langle \text{proof} \rangle$

lemma *countableI-bij*: $\text{bij-betw } f \ (C :: \text{nat set}) \ S \implies \text{countable } S$
 $\langle \text{proof} \rangle$

lemma *countable-finite*: $\text{finite } S \implies \text{countable } S$
 $\langle \text{proof} \rangle$

lemma *countableI-bij1*: $\text{bij-betw } f \ A \ B \implies \text{countable } A \implies \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countableI-bij2*: $\text{bij-betw } f \ B \ A \implies \text{countable } A \implies \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countable-iff-bij[simp]*: $\text{bij-betw } f \ A \ B \implies \text{countable } A \longleftrightarrow \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countable-subset*: $A \subseteq B \implies \text{countable } B \implies \text{countable } A$
 $\langle \text{proof} \rangle$

lemma *countableI-type[intro, simp]*: $\text{countable } (A :: 'a :: \text{countable set})$
 $\langle \text{proof} \rangle$

21.2 Enumerate a countable set

lemma *countableE-infinite*:
assumes $\text{countable } S$ *infinite* S
obtains $e :: 'a \Rightarrow \text{nat}$ **where** $\text{bij-betw } e \ S \ \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *countable-infiniteE'*:
assumes $\text{countable } A$ *infinite* A
obtains g **where** $\text{bij-betw } g \ (\text{UNIV} :: \text{nat set}) \ A$
 $\langle \text{proof} \rangle$

lemma *countable-enum-cases*:
assumes $\text{countable } S$
obtains $(\text{finite}) \ f :: 'a \Rightarrow \text{nat}$ **where** $\text{finite } S \ \text{bij-betw } f \ S \ \{.. < \text{card } S\}$
 $\quad \mid (\text{infinite}) \ f :: 'a \Rightarrow \text{nat}$ **where** $\text{infinite } S \ \text{bij-betw } f \ S \ \text{UNIV}$
 $\langle \text{proof} \rangle$

definition *to-nat-on* :: $'a \ \text{set} \Rightarrow 'a \Rightarrow \text{nat}$ **where**
 $\text{to-nat-on } S = (\text{SOME } f. \text{ if } \text{finite } S \text{ then } \text{bij-betw } f \ S \ \{.. < \text{card } S\} \text{ else } \text{bij-betw } f \ S \ \text{UNIV})$

definition *from-nat-into* :: $'a \ \text{set} \Rightarrow \text{nat} \Rightarrow 'a$ **where**
 $\text{from-nat-into } S \ n = (\text{if } n \in \text{to-nat-on } S \ 'S \text{ then } \text{inv-into } S \ (\text{to-nat-on } S) \ n \text{ else } \dots)$

SOME $s. s \in S$)

lemma *to-nat-on-finite*: $\text{finite } S \implies \text{bij-betw } (\text{to-nat-on } S) S \{..< \text{card } S\}$
 $\langle \text{proof} \rangle$

lemma *to-nat-on-infinite*: $\text{countable } S \implies \text{infinite } S \implies \text{bij-betw } (\text{to-nat-on } S) S$
 UNIV
 $\langle \text{proof} \rangle$

lemma *bij-betw-from-nat-into-finite*: $\text{finite } S \implies \text{bij-betw } (\text{from-nat-into } S) \{..< \text{card } S\} S$
 $\langle \text{proof} \rangle$

lemma *bij-betw-from-nat-into*: $\text{countable } S \implies \text{infinite } S \implies \text{bij-betw } (\text{from-nat-into } S) \text{ UNIV } S$
 $\langle \text{proof} \rangle$

The sum/product over the enumeration of a finite set equals simply the sum/product over the set

context *comm-monoid-set*
begin

lemma *card-from-nat-into*:
 $F (\lambda i. h (\text{from-nat-into } A i)) \{..< \text{card } A\} = F h A$
 $\langle \text{proof} \rangle$

end

lemma *countable-as-injective-image*:
assumes $\text{countable } A \text{ infinite } A$
obtains $f :: \text{nat} \Rightarrow 'a$ **where** $A = \text{range } f \text{ inj } f$
 $\langle \text{proof} \rangle$

lemma *inj-on-to-nat-on[intro]*: $\text{countable } A \implies \text{inj-on } (\text{to-nat-on } A) A$
 $\langle \text{proof} \rangle$

lemma *to-nat-on-inj[simp]*:
 $\text{countable } A \implies a \in A \implies b \in A \implies \text{to-nat-on } A a = \text{to-nat-on } A b \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *from-nat-into-to-nat-on[simp]*: $\text{countable } A \implies a \in A \implies \text{from-nat-into } A (\text{to-nat-on } A a) = a$
 $\langle \text{proof} \rangle$

lemma *subset-range-from-nat-into*: $\text{countable } A \implies A \subseteq \text{range } (\text{from-nat-into } A)$
 $\langle \text{proof} \rangle$

lemma *from-nat-into*: $A \neq \{\} \implies \text{from-nat-into } A n \in A$
 $\langle \text{proof} \rangle$

lemma *range-from-nat-into-subset*: $A \neq \{\}$ \implies $\text{range } (\text{from-nat-into } A) \subseteq A$
 ⟨proof⟩

lemma *range-from-nat-into[simp]*: $A \neq \{\}$ \implies $\text{countable } A \implies \text{range } (\text{from-nat-into } A) = A$
 ⟨proof⟩

lemma *image-to-nat-on*: $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A \text{ ‘ } A = \text{UNIV}$
 ⟨proof⟩

lemma *to-nat-on-surj*: $\text{countable } A \implies \text{infinite } A \implies \exists a \in A. \text{to-nat-on } A \ a = n$
 ⟨proof⟩

lemma *to-nat-on-from-nat-into[simp]*: $n \in \text{to-nat-on } A \text{ ‘ } A \implies \text{to-nat-on } A \ (\text{from-nat-into } A \ n) = n$
 ⟨proof⟩

lemma *to-nat-on-from-nat-into-infinite[simp]*:
 $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A \ (\text{from-nat-into } A \ n) = n$
 ⟨proof⟩

lemma *from-nat-into-inj*:
 $\text{countable } A \implies m \in \text{to-nat-on } A \text{ ‘ } A \implies n \in \text{to-nat-on } A \text{ ‘ } A \implies$
 $\text{from-nat-into } A \ m = \text{from-nat-into } A \ n \longleftrightarrow m = n$
 ⟨proof⟩

lemma *from-nat-into-inj-infinite[simp]*:
 $\text{countable } A \implies \text{infinite } A \implies \text{from-nat-into } A \ m = \text{from-nat-into } A \ n \longleftrightarrow m$
 $= n$
 ⟨proof⟩

lemma *eq-from-nat-into-iff*:
 $\text{countable } A \implies x \in A \implies i \in \text{to-nat-on } A \text{ ‘ } A \implies x = \text{from-nat-into } A \ i \longleftrightarrow$
 $i = \text{to-nat-on } A \ x$
 ⟨proof⟩

lemma *from-nat-into-surj*: $\text{countable } A \implies a \in A \implies \exists n. \text{from-nat-into } A \ n = a$
 ⟨proof⟩

lemma *from-nat-into-inject[simp]*:
 $A \neq \{\} \implies \text{countable } A \implies B \neq \{\} \implies \text{countable } B \implies \text{from-nat-into } A =$
 $\text{from-nat-into } B \longleftrightarrow A = B$
 ⟨proof⟩

lemma *inj-on-from-nat-into*: $\text{inj-on from-nat-into } (\{A. A \neq \{\} \wedge \text{countable } A\})$
 ⟨proof⟩

21.3 Closure properties of countability

lemma *countable-SIGMA*[intro, simp]:

$\text{countable } I \implies (\bigwedge i. i \in I \implies \text{countable } (A \ i)) \implies \text{countable } (\text{SIGMA } i : I. A \ i)$
 <proof>

lemma *countable-image*[intro, simp]:

assumes *countable* A
shows *countable* $(f' A)$
 <proof>

lemma *countable-image-inj-on*: $\text{countable } (f' A) \implies \text{inj-on } f \ A \implies \text{countable } A$

<proof>

lemma *countable-image-inj-Int-vimage*:

$\llbracket \text{inj-on } f \ S; \text{countable } A \rrbracket \implies \text{countable } (S \cap f^{-1} A)$
 <proof>

lemma *countable-image-inj-gen*:

$\llbracket \text{inj-on } f \ S; \text{countable } A \rrbracket \implies \text{countable } \{x \in S. f \ x \in A\}$
 <proof>

lemma *countable-image-inj-eq*:

$\text{inj-on } f \ S \implies \text{countable}(f' S) \longleftrightarrow \text{countable } S$
 <proof>

lemma *countable-image-inj*:

$\llbracket \text{countable } A; \text{inj } f \rrbracket \implies \text{countable } \{x. f \ x \in A\}$
 <proof>

lemma *countable-UN*[intro, simp]:

fixes $I :: 'i \text{ set}$ **and** $A :: 'i \Rightarrow 'a \text{ set}$
assumes I : *countable* I
assumes A : $\bigwedge i. i \in I \implies \text{countable } (A \ i)$
shows *countable* $(\bigcup_{i \in I}. A \ i)$
 <proof>

lemma *countable-Un*[intro]: $\text{countable } A \implies \text{countable } B \implies \text{countable } (A \cup B)$

<proof>

lemma *countable-Un-iff*[simp]: $\text{countable } (A \cup B) \longleftrightarrow \text{countable } A \wedge \text{countable } B$

<proof>

lemma *countable-Plus*[intro, simp]:

$\text{countable } A \implies \text{countable } B \implies \text{countable } (A <+> B)$
 <proof>

lemma *countable-empty*[intro, simp]: *countable* $\{\}$

<proof>

lemma *countable-insert*[*intro, simp*]: *countable* $A \implies \text{countable } (\text{insert } a \ A)$
 ⟨*proof*⟩

lemma *countable-Int1*[*intro, simp*]: *countable* $A \implies \text{countable } (A \cap B)$
 ⟨*proof*⟩

lemma *countable-Int2*[*intro, simp*]: *countable* $B \implies \text{countable } (A \cap B)$
 ⟨*proof*⟩

lemma *countable-INT*[*intro, simp*]: $i \in I \implies \text{countable } (A \ i) \implies \text{countable } (\bigcap_{i \in I} A \ i)$
 ⟨*proof*⟩

lemma *countable-Diff*[*intro, simp*]: *countable* $A \implies \text{countable } (A - B)$
 ⟨*proof*⟩

lemma *countable-insert-eq* [*simp*]: *countable* $(\text{insert } x \ A) = \text{countable } A$
 ⟨*proof*⟩

lemma *countable-vimage*: $B \subseteq \text{range } f \implies \text{countable } (f^{-1} B) \implies \text{countable } B$
 ⟨*proof*⟩

lemma *surj-countable-vimage*: *surj* $f \implies \text{countable } (f^{-1} B) \implies \text{countable } B$
 ⟨*proof*⟩

lemma *countable-Collect*[*simp*]: *countable* $A \implies \text{countable } \{a \in A. \varphi \ a\}$
 ⟨*proof*⟩

lemma *countable-Image*:
 assumes $\bigwedge y. y \in Y \implies \text{countable } (X^{-1} \{y\})$
 assumes *countable* Y
 shows *countable* $(X^{-1} Y)$
 ⟨*proof*⟩

lemma *countable-relpow*:
 fixes $X :: 'a \ \text{rel}$
 assumes *Image-X*: $\bigwedge Y. \text{countable } Y \implies \text{countable } (X^{-1} Y)$
 assumes *Y: countable* Y
 shows *countable* $((X^{\sim n})^{-1} Y)$
 ⟨*proof*⟩

lemma *countable-funpow*:
 fixes $f :: 'a \ \text{set} \Rightarrow 'a \ \text{set}$
 assumes $\bigwedge A. \text{countable } A \implies \text{countable } (f \ A)$
 and *countable* A
 shows *countable* $((f^{\sim n}) \ A)$
 ⟨*proof*⟩

lemma *countable-rtranc1*:

$(\bigwedge Y. \text{countable } Y \implies \text{countable } (X \text{ “ } Y)) \implies \text{countable } Y \implies \text{countable } (X^* \text{ “ } Y)$
 $\langle \text{proof} \rangle$

lemma *countable-lists*[*intro, simp*]:

assumes A : *countable* A **shows** *countable* (*lists* A)
 $\langle \text{proof} \rangle$

lemma *Collect-finite-eq-lists*: *Collect finite* = *set ‘ lists UNIV*

$\langle \text{proof} \rangle$

lemma *countable-Collect-finite*: *countable* (*Collect* (*finite::’a::countable set* \implies *bool*))

$\langle \text{proof} \rangle$

lemma *countable-int*: *countable* \mathbb{Z}

$\langle \text{proof} \rangle$

lemma *countable-rat*: *countable* \mathbb{Q}

$\langle \text{proof} \rangle$

lemma *Collect-finite-subset-eq-lists*: $\{A. \text{finite } A \wedge A \subseteq T\} = \text{set ‘ lists } T$

$\langle \text{proof} \rangle$

lemma *countable-Collect-finite-subset*:

countable $T \implies \text{countable } \{A. \text{finite } A \wedge A \subseteq T\}$
 $\langle \text{proof} \rangle$

lemma *countable-Fpow*: *countable* $S \implies \text{countable } (\text{Fpow } S)$

$\langle \text{proof} \rangle$

lemma *countable-set-option* [*simp*]: *countable* (*set-option* x)

$\langle \text{proof} \rangle$

21.4 Misc lemmas

lemma *countable-subset-image*:

$\text{countable } B \wedge B \subseteq (f \text{ ‘ } A) \longleftrightarrow (\exists A'. \text{countable } A' \wedge A' \subseteq A \wedge (B = f \text{ ‘ } A'))$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *ex-subset-image-inj*:

$(\exists T. T \subseteq f \text{ ‘ } S \wedge P \ T) \longleftrightarrow (\exists T. T \subseteq S \wedge \text{inj-on } f \ T \wedge P \ (f \text{ ‘ } T))$
 $\langle \text{proof} \rangle$

lemma *all-subset-image-inj*:

$(\forall T. T \subseteq f \text{ ‘ } S \longrightarrow P \ T) \longleftrightarrow (\forall T. T \subseteq S \wedge \text{inj-on } f \ T \longrightarrow P \ (f \text{ ‘ } T))$
 $\langle \text{proof} \rangle$

lemma *ex-countable-subset-image-inj*:

$(\exists T. \text{countable } T \wedge T \subseteq f^{-1} S \wedge P T) \longleftrightarrow$
 $(\exists T. \text{countable } T \wedge T \subseteq S \wedge \text{inj-on } f T \wedge P (f^{-1} T))$
 $\langle \text{proof} \rangle$

lemma *all-countable-subset-image-inj*:

$(\forall T. \text{countable } T \wedge T \subseteq f^{-1} S \longrightarrow P T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \wedge$
 $\text{inj-on } f T \longrightarrow P(f^{-1} T))$
 $\langle \text{proof} \rangle$

lemma *ex-countable-subset-image*:

$(\exists T. \text{countable } T \wedge T \subseteq f^{-1} S \wedge P T) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge P (f^{-1} T))$
 $\langle \text{proof} \rangle$

lemma *all-countable-subset-image*:

$(\forall T. \text{countable } T \wedge T \subseteq f^{-1} S \longrightarrow P T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \longrightarrow$
 $P(f^{-1} T))$
 $\langle \text{proof} \rangle$

lemma *countable-image-eq*:

$\text{countable}(f^{-1} S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f^{-1} S = f^{-1} T)$
 $\langle \text{proof} \rangle$

lemma *countable-image-eq-inj*:

$\text{countable}(f^{-1} S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f^{-1} S = f^{-1} T \wedge \text{inj-on } f T)$
 $\langle \text{proof} \rangle$

lemma *infinite-countable-subset'*:

assumes X : *infinite* X **shows** $\exists C \subseteq X. \text{countable } C \wedge \text{infinite } C$
 $\langle \text{proof} \rangle$

lemma *countable-all*:

assumes S : *countable* S
shows $(\forall s \in S. P s) \longleftrightarrow (\forall n :: \text{nat}. \text{from-nat-into } S n \in S \longrightarrow P (\text{from-nat-into } S n))$
 $\langle \text{proof} \rangle$

lemma *finite-sequence-to-countable-set*:

assumes *countable* X
obtains F **where** $\bigwedge i. F i \subseteq X \wedge i. F i \subseteq F (\text{Suc } i) \wedge i. \text{finite } (F i) (\bigcup i. F i) = X$
 $\langle \text{proof} \rangle$

lemma *transfer-countable[transfer-rule]*:

bi-unique $R \implies \text{rel-fun } (\text{rel-set } R) (=) \text{countable countable}$
 $\langle \text{proof} \rangle$

21.5 Uncountable

abbreviation *uncountable* **where**

uncountable $A \equiv \neg \text{countable } A$

lemma *uncountable-def*: *uncountable* $A \longleftrightarrow A \neq \{\}$ $\wedge \neg (\exists f :: (\text{nat} \Rightarrow 'a). \text{range } f = A)$

<proof>

lemma *uncountable-bij-betw*: *bij-betw* $f A B \Longrightarrow \text{uncountable } B \Longrightarrow \text{uncountable } A$

<proof>

lemma *uncountable-infinite*: *uncountable* $A \Longrightarrow \text{infinite } A$

<proof>

lemma *uncountable-minus-countable*:

uncountable $A \Longrightarrow \text{countable } B \Longrightarrow \text{uncountable } (A - B)$

<proof>

lemma *countable-Diff-eq [simp]*: *countable* $(A - \{x\}) = \text{countable } A$

<proof>

Every infinite set can be covered by a pairwise disjoint family of infinite sets. This version doesn't achieve equality, as it only covers a countable subset

lemma *infinite-infinite-partition*:

assumes *infinite* A

obtains $C :: \text{nat} \Rightarrow 'a \text{ set}$

where *pairwise* $(\lambda i j. \text{disjnt } (C i) (C j)) \text{ UNIV } (\bigcup i. C i) \subseteq A \wedge i. \text{infinite } (C$

$i)$

<proof>

end

22 Countable Complete Lattices

theory *Countable-Complete-Lattices*

imports *Main Countable-Set*

begin

lemma *UNIV-nat-eq*: $\text{UNIV} = \text{insert } 0 (\text{range } \text{Suc})$

<proof>

class *countable-complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +

assumes *ccInf-lower*: *countable* $A \Longrightarrow x \in A \Longrightarrow \text{Inf } A \leq x$

assumes *ccInf-greatest*: *countable* $A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow z \leq x) \Longrightarrow z \leq \text{Inf } A$

assumes *ccSup-upper*: *countable* $A \Longrightarrow x \in A \Longrightarrow x \leq \text{Sup } A$

assumes *ccSup-least*: *countable* $A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow x \leq z) \Longrightarrow \text{Sup } A \leq z$

assumes *ccInf-empty [simp]*: $\text{Inf } \{\} = \text{top}$

assumes *ccSup-empty* [*simp*]: $\text{Sup } \{\} = \text{bot}$
begin

subclass *bounded-lattice*
 $\langle \text{proof} \rangle$

lemma *ccINF-lower*: $\text{countable } A \implies i \in A \implies (\text{INF } i \in A. f i) \leq f i$
 $\langle \text{proof} \rangle$

lemma *ccINF-greatest*: $\text{countable } A \implies (\bigwedge i. i \in A \implies u \leq f i) \implies u \leq (\text{INF } i \in A. f i)$
 $\langle \text{proof} \rangle$

lemma *ccSUP-upper*: $\text{countable } A \implies i \in A \implies f i \leq (\text{SUP } i \in A. f i)$
 $\langle \text{proof} \rangle$

lemma *ccSUP-least*: $\text{countable } A \implies (\bigwedge i. i \in A \implies f i \leq u) \implies (\text{SUP } i \in A. f i) \leq u$
 $\langle \text{proof} \rangle$

lemma *ccInf-lower2*: $\text{countable } A \implies u \in A \implies u \leq v \implies \text{Inf } A \leq v$
 $\langle \text{proof} \rangle$

lemma *ccINF-lower2*: $\text{countable } A \implies i \in A \implies f i \leq u \implies (\text{INF } i \in A. f i) \leq u$
 $\langle \text{proof} \rangle$

lemma *ccSup-upper2*: $\text{countable } A \implies u \in A \implies v \leq u \implies v \leq \text{Sup } A$
 $\langle \text{proof} \rangle$

lemma *ccSUP-upper2*: $\text{countable } A \implies i \in A \implies u \leq f i \implies u \leq (\text{SUP } i \in A. f i)$
 $\langle \text{proof} \rangle$

lemma *le-ccInf-iff*: $\text{countable } A \implies b \leq \text{Inf } A \longleftrightarrow (\forall a \in A. b \leq a)$
 $\langle \text{proof} \rangle$

lemma *le-ccINF-iff*: $\text{countable } A \implies u \leq (\text{INF } i \in A. f i) \longleftrightarrow (\forall i \in A. u \leq f i)$
 $\langle \text{proof} \rangle$

lemma *ccSup-le-iff*: $\text{countable } A \implies \text{Sup } A \leq b \longleftrightarrow (\forall a \in A. a \leq b)$
 $\langle \text{proof} \rangle$

lemma *ccSUP-le-iff*: $\text{countable } A \implies (\text{SUP } i \in A. f i) \leq u \longleftrightarrow (\forall i \in A. f i \leq u)$
 $\langle \text{proof} \rangle$

lemma *ccInf-insert* [*simp*]: $\text{countable } A \implies \text{Inf } (\text{insert } a A) = \text{inf } a (\text{Inf } A)$
 $\langle \text{proof} \rangle$

lemma *ccINF-insert [simp]: countable A \implies (INF x \in insert a A. f x) = inf (f a) (Inf (f ‘ A))*
 ⟨proof⟩

lemma *ccSup-insert [simp]: countable A \implies Sup (insert a A) = sup a (Sup A)*
 ⟨proof⟩

lemma *ccSUP-insert [simp]: countable A \implies (SUP x \in insert a A. f x) = sup (f a) (Sup (f ‘ A))*
 ⟨proof⟩

lemma *ccINF-empty [simp]: (INF x \in {}. f x) = top*
 ⟨proof⟩

lemma *ccSUP-empty [simp]: (SUP x \in {}. f x) = bot*
 ⟨proof⟩

lemma *ccInf-superset-mono: countable A \implies B \subseteq A \implies Inf A \leq Inf B*
 ⟨proof⟩

lemma *ccSup-subset-mono: countable B \implies A \subseteq B \implies Sup A \leq Sup B*
 ⟨proof⟩

lemma *ccInf-mono:*
 assumes [intro]: countable B countable A
 assumes $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$
 shows Inf A \leq Inf B
 ⟨proof⟩

lemma *ccINF-mono:*
 countable A \implies countable B \implies ($\bigwedge m. m \in B \implies \exists n \in A. f n \leq g m$) \implies (INF n \in A. f n) \leq (INF n \in B. g n)
 ⟨proof⟩

lemma *ccSup-mono:*
 assumes [intro]: countable B countable A
 assumes $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$
 shows Sup A \leq Sup B
 ⟨proof⟩

lemma *ccSUP-mono:*
 countable A \implies countable B \implies ($\bigwedge n. n \in A \implies \exists m \in B. f n \leq g m$) \implies (SUP n \in A. f n) \leq (SUP n \in B. g n)
 ⟨proof⟩

lemma *ccINF-superset-mono:*
 countable A \implies B \subseteq A \implies ($\bigwedge x. x \in B \implies f x \leq g x$) \implies (INF x \in A. f x) \leq (INF x \in B. g x)
 ⟨proof⟩

lemma *ccSUP-subset-mono*:

$\text{countable } B \implies A \subseteq B \implies (\bigwedge x. x \in A \implies f\ x \leq g\ x) \implies (\text{SUP } x \in A. f\ x) \leq (\text{SUP } x \in B. g\ x)$
 ⟨proof⟩

lemma *less-eq-ccInf-inter*: $\text{countable } A \implies \text{countable } B \implies \text{sup } (\text{Inf } A) (\text{Inf } B) \leq \text{Inf } (A \cap B)$
 ⟨proof⟩

lemma *ccSup-inter-less-eq*: $\text{countable } A \implies \text{countable } B \implies \text{Sup } (A \cap B) \leq \text{inf } (\text{Sup } A) (\text{Sup } B)$
 ⟨proof⟩

lemma *ccInf-union-distrib*: $\text{countable } A \implies \text{countable } B \implies \text{Inf } (A \cup B) = \text{inf } (\text{Inf } A) (\text{Inf } B)$
 ⟨proof⟩

lemma *ccINF-union*:

$\text{countable } A \implies \text{countable } B \implies (\text{INF } i \in A \cup B. M\ i) = \text{inf } (\text{INF } i \in A. M\ i) (\text{INF } i \in B. M\ i)$
 ⟨proof⟩

lemma *ccSup-union-distrib*: $\text{countable } A \implies \text{countable } B \implies \text{Sup } (A \cup B) = \text{sup } (\text{Sup } A) (\text{Sup } B)$
 ⟨proof⟩

lemma *ccSUP-union*:

$\text{countable } A \implies \text{countable } B \implies (\text{SUP } i \in A \cup B. M\ i) = \text{sup } (\text{SUP } i \in A. M\ i) (\text{SUP } i \in B. M\ i)$
 ⟨proof⟩

lemma *ccINF-inf-distrib*: $\text{countable } A \implies \text{inf } (\text{INF } a \in A. f\ a) (\text{INF } a \in A. g\ a) = (\text{INF } a \in A. \text{inf } (f\ a) (g\ a))$
 ⟨proof⟩

lemma *ccSUP-sup-distrib*: $\text{countable } A \implies \text{sup } (\text{SUP } a \in A. f\ a) (\text{SUP } a \in A. g\ a) = (\text{SUP } a \in A. \text{sup } (f\ a) (g\ a))$
 ⟨proof⟩

lemma *ccINF-const [simp]*: $A \neq \{\} \implies (\text{INF } i \in A. f) = f$
 ⟨proof⟩

lemma *ccSUP-const [simp]*: $A \neq \{\} \implies (\text{SUP } i \in A. f) = f$
 ⟨proof⟩

lemma *ccINF-top [simp]*: $(\text{INF } x \in A. \text{top}) = \text{top}$
 ⟨proof⟩

lemma *ccSUP-bot* [simp]: $(\text{SUP } x \in A. \text{ bot}) = \text{ bot}$
 ⟨proof⟩

lemma *ccINF-commute*: $\text{countable } A \implies \text{countable } B \implies (\text{INF } i \in A. \text{ INF } j \in B. f \ i \ j) = (\text{INF } j \in B. \text{ INF } i \in A. f \ i \ j)$
 ⟨proof⟩

lemma *ccSUP-commute*: $\text{countable } A \implies \text{countable } B \implies (\text{SUP } i \in A. \text{ SUP } j \in B. f \ i \ j) = (\text{SUP } j \in B. \text{ SUP } i \in A. f \ i \ j)$
 ⟨proof⟩

end

context

fixes $a :: 'a :: \{\text{countable-complete-lattice}, \text{linorder}\}$

begin

lemma *less-ccSup-iff*: $\text{countable } S \implies a < \text{Sup } S \longleftrightarrow (\exists x \in S. a < x)$
 ⟨proof⟩

lemma *less-ccSUP-iff*: $\text{countable } A \implies a < (\text{SUP } i \in A. f \ i) \longleftrightarrow (\exists x \in A. a < f \ x)$
 ⟨proof⟩

lemma *ccInf-less-iff*: $\text{countable } S \implies \text{Inf } S < a \longleftrightarrow (\exists x \in S. x < a)$
 ⟨proof⟩

lemma *ccINF-less-iff*: $\text{countable } A \implies (\text{INF } i \in A. f \ i) < a \longleftrightarrow (\exists x \in A. f \ x < a)$
 ⟨proof⟩

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
assumes *sup-ccInf*: $\text{countable } B \implies \text{sup } a \ (\text{Inf } B) = (\text{INF } b \in B. \text{ sup } a \ b)$
assumes *inf-ccSup*: $\text{countable } B \implies \text{inf } a \ (\text{Sup } B) = (\text{SUP } b \in B. \text{ inf } a \ b)$
begin

lemma *sup-ccINF*:
 $\text{countable } B \implies \text{sup } a \ (\text{INF } b \in B. f \ b) = (\text{INF } b \in B. \text{ sup } a \ (f \ b))$
 ⟨proof⟩

lemma *inf-ccSUP*:
 $\text{countable } B \implies \text{inf } a \ (\text{SUP } b \in B. f \ b) = (\text{SUP } b \in B. \text{ inf } a \ (f \ b))$
 ⟨proof⟩

subclass *distrib-lattice*
 ⟨proof⟩

lemma *ccInf-sup*:

countable $B \implies \sup (\text{Inf } B) \ a = (\text{INF } b \in B. \sup b \ a)$
 ⟨proof⟩

lemma *ccSup-inf*:

countable $B \implies \inf (\text{Sup } B) \ a = (\text{SUP } b \in B. \inf b \ a)$
 ⟨proof⟩

lemma *ccINF-sup*:

countable $B \implies \sup (\text{INF } b \in B. f \ b) \ a = (\text{INF } b \in B. \sup (f \ b) \ a)$
 ⟨proof⟩

lemma *ccSUP-inf*:

countable $B \implies \inf (\text{SUP } b \in B. f \ b) \ a = (\text{SUP } b \in B. \inf (f \ b) \ a)$
 ⟨proof⟩

lemma *ccINF-sup-distrib2*:

countable $A \implies \text{countable } B \implies \sup (\text{INF } a \in A. f \ a) (\text{INF } b \in B. g \ b) = (\text{INF } a \in A. \text{INF } b \in B. \sup (f \ a) (g \ b))$
 ⟨proof⟩

lemma *ccSUP-inf-distrib2*:

countable $A \implies \text{countable } B \implies \inf (\text{SUP } a \in A. f \ a) (\text{SUP } b \in B. g \ b) = (\text{SUP } a \in A. \text{SUP } b \in B. \inf (f \ a) (g \ b))$
 ⟨proof⟩

context

fixes $f :: 'a \Rightarrow 'b :: \text{countable-complete-lattice}$

assumes *mono f*

begin

lemma *mono-ccInf*:

countable $A \implies f (\text{Inf } A) \leq (\text{INF } x \in A. f \ x)$
 ⟨proof⟩

lemma *mono-ccSup*:

countable $A \implies (\text{SUP } x \in A. f \ x) \leq f (\text{Sup } A)$
 ⟨proof⟩

lemma *mono-ccINF*:

countable $I \implies f (\text{INF } i \in I. A \ i) \leq (\text{INF } x \in I. f \ (A \ x))$
 ⟨proof⟩

lemma *mono-ccSUP*:

countable $I \implies (\text{SUP } x \in I. f \ (A \ x)) \leq f (\text{SUP } i \in I. A \ i)$
 ⟨proof⟩

end

end

22.0.1 Instances of countable complete lattices

instance *fun* :: (*type*, *countable-complete-lattice*) *countable-complete-lattice*
 ⟨*proof*⟩

subclass (**in** *complete-lattice*) *countable-complete-lattice*
 ⟨*proof*⟩

subclass (**in** *complete-distrib-lattice*) *countable-complete-distrib-lattice*
 ⟨*proof*⟩

end

23 Type of (at Most) Countable Sets

theory *Countable-Set-Type*
imports *Countable-Set*
begin

23.1 Cardinal stuff

context
includes *cardinal-syntax*
begin

lemma *countable-card-of-nat*: *countable A* $\longleftrightarrow |A| \leq_o |UNIV::nat\ set|$
 ⟨*proof*⟩

lemma *countable-card-le-natLeq*: *countable A* $\longleftrightarrow |A| \leq_o natLeq$
 ⟨*proof*⟩

lemma *countable-or-card-of*:
assumes *countable A*
shows (*finite A* $\wedge |A| <_o |UNIV::nat\ set|$) \vee
 (*infinite A* $\wedge |A| =_o |UNIV::nat\ set|$)
 ⟨*proof*⟩

lemma *countable-cases-card-of[elim]*:
assumes *countable A*
obtains (*Fin*) *finite A* $|A| <_o |UNIV::nat\ set|$
 | (*Inf*) *infinite A* $|A| =_o |UNIV::nat\ set|$
 ⟨*proof*⟩

lemma *countable-or*:
countable A $\implies (\exists f::'a \Rightarrow nat. \text{finite } A \wedge \text{inj-on } f\ A) \vee (\exists f::'a \Rightarrow nat. \text{infinite } A$
 $\wedge \text{bij-betw } f\ A\ UNIV)$
 ⟨*proof*⟩

lemma *countable-cases[elim]*:

```

assumes countable A
obtains (Fin) f :: 'a ⇒ nat where finite A inj-on f A
      | (Inf) f :: 'a ⇒ nat where infinite A bij-betw f A UNIV
⟨proof⟩

```

```

lemma countable-ordLeq:
assumes |A| ≤o |B| and countable B
shows countable A
⟨proof⟩

```

```

lemma countable-ordLess:
assumes AB: |A| <o |B| and B: countable B
shows countable A
⟨proof⟩

```

end

23.2 The type of countable sets

```

typedef 'a cset = {A :: 'a set. countable A} morphisms rcset acset
⟨proof⟩

```

setup-lifting type-definition-cset

```

declare
  rcset-inverse[simp]
  acset-inverse[Transfer.transferred, unfolded mem-Collect-eq, simp]
  acset-inject[Transfer.transferred, unfolded mem-Collect-eq, simp]
  rcset[Transfer.transferred, unfolded mem-Collect-eq, simp]

```

```

instantiation cset :: (type) {bounded-lattice-bot, distrib-lattice, minus}
begin

```

```

lift-definition bot-cset :: 'a cset is {} parametric empty-transfer ⟨proof⟩

```

```

lift-definition less-eq-cset :: 'a cset ⇒ 'a cset ⇒ bool
is subset-eq parametric subset-transfer ⟨proof⟩

```

```

definition less-cset :: 'a cset ⇒ 'a cset ⇒ bool
where xs < ys ≡ xs ≤ ys ∧ xs ≠ (ys::'a cset)

```

```

lemma less-cset-transfer[transfer-rule]:
includes lifting-syntax
assumes [transfer-rule]: bi-unique A
shows ((pcr-cset A) ==> (pcr-cset A) ==> (=)) (⊆) (<)
⟨proof⟩

```

```

lift-definition sup-cset :: 'a cset ⇒ 'a cset ⇒ 'a cset
is union parametric union-transfer ⟨proof⟩

```


lift-definition *inf-cset* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset
is *inter parametric inter-transfer* \langle proof \rangle

lift-definition *minus-cset* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset
is *minus parametric Diff-transfer* \langle proof \rangle

instance \langle proof \rangle

end

abbreviation *empty* :: 'a cset **where** *empty* \equiv *bot*

abbreviation *csubset-eq* :: 'a cset \Rightarrow 'a cset \Rightarrow bool **where** *csubset-eq* *xs ys* \equiv *xs* \leq *ys*

abbreviation *csubset* :: 'a cset \Rightarrow 'a cset \Rightarrow bool **where** *csubset* *xs ys* \equiv *xs* $<$ *ys*

abbreviation *cUn* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset **where** *cUn* *xs ys* \equiv *sup xs ys*

abbreviation *cInt* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset **where** *cInt* *xs ys* \equiv *inf xs ys*

abbreviation *cDiff* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset **where** *cDiff* *xs ys* \equiv *minus xs ys*

lift-definition *cin* :: 'a \Rightarrow 'a cset \Rightarrow bool **is** (\in) **parametric member-transfer**
 \langle proof \rangle

lift-definition *cinsert* :: 'a \Rightarrow 'a cset \Rightarrow 'a cset **is** *insert parametric Lifting-Set.insert-transfer*
 \langle proof \rangle

abbreviation *csingle* :: 'a \Rightarrow 'a cset **where** *csingle* *x* \equiv *cinsert x empty*

lift-definition *cimage* :: ('a \Rightarrow 'b) \Rightarrow 'a cset \Rightarrow 'b cset **is** (\cdot) **parametric image-transfer**
 \langle proof \rangle

lift-definition *cBall* :: 'a cset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **is** *Ball parametric Ball-transfer*
 \langle proof \rangle

lift-definition *cBex* :: 'a cset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **is** *Bex parametric Bex-transfer*
 \langle proof \rangle

lift-definition *cUnion* :: 'a cset cset \Rightarrow 'a cset **is** *Union parametric Union-transfer*
 \langle proof \rangle

abbreviation (*input*) *cUNION* :: 'a cset \Rightarrow ('a \Rightarrow 'b cset) \Rightarrow 'b cset
where *cUNION* *A f* \equiv *cUnion (cimage f A)*

lemma *Union-conv-UNION*: $\bigcup A = \bigcup (id \cdot A)$
 \langle proof \rangle

lemmas *cset-eqI* = *set-eqI* [*Transfer.transferred*]

lemmas *cset-eq-iff* [*no-atp*] = *set-eq-iff* [*Transfer.transferred*]

lemmas *cBallI* [*intro!*] = *ballI* [*Transfer.transferred*]

lemmas *cbspec* [*dest?*] = *bspec* [*Transfer.transferred*]

lemmas *cBallE* [*elim*] = *ballE* [*Transfer.transferred*]

lemmas *cBexI* [*intro*] = *bexI* [*Transfer.transferred*]

lemmas *rev-cBexI* [*intro?*] = *rev-bexI* [*Transfer.transferred*]

lemmas *cBexCI* = *bexCI* [*Transfer.transferred*]

lemmas *cBexE* [*elim!*] = *bexE* [*Transfer.transferred*]

```

lemmas cBall-triv[simp] = ball-triv[Transfer.transferred]
lemmas cBex-triv[simp] = bex-triv[Transfer.transferred]
lemmas cBex-triv-one-point1[simp] = bex-triv-one-point1[Transfer.transferred]
lemmas cBex-triv-one-point2[simp] = bex-triv-one-point2[Transfer.transferred]
lemmas cBex-one-point1[simp] = bex-one-point1[Transfer.transferred]
lemmas cBex-one-point2[simp] = bex-one-point2[Transfer.transferred]
lemmas cBall-one-point1[simp] = ball-one-point1[Transfer.transferred]
lemmas cBall-one-point2[simp] = ball-one-point2[Transfer.transferred]
lemmas cBall-conj-distrib = ball-conj-distrib[Transfer.transferred]
lemmas cBex-disj-distrib = bex-disj-distrib[Transfer.transferred]
lemmas cBall-cong = ball-cong[Transfer.transferred]
lemmas cBex-cong = bex-cong[Transfer.transferred]
lemmas csubsetI[intro!] = subsetI[Transfer.transferred]
lemmas csubsetD[elim, intro?] = subsetD[Transfer.transferred]
lemmas rev-csubsetD[no-atp, intro?] = rev-subsetD[Transfer.transferred]
lemmas csubsetCE[no-atp, elim] = subsetCE[Transfer.transferred]
lemmas csubset-eq[no-atp] = subset-eq[Transfer.transferred]
lemmas contra-csubsetD[no-atp] = contra-subsetD[Transfer.transferred]
lemmas csubset-refl = subset-refl[Transfer.transferred]
lemmas csubset-trans = subset-trans[Transfer.transferred]
lemmas cset-rev-mp = rev-subsetD[Transfer.transferred]
lemmas cset-mp = subsetD[Transfer.transferred]
lemmas csubset-not-fsubset-eq[code] = subset-not-subset-eq[Transfer.transferred]
lemmas eq-cmem-trans = eq-mem-trans[Transfer.transferred]
lemmas csubset-antisym[intro!] = subset-antisym[Transfer.transferred]
lemmas cequalityD1 = equalityD1[Transfer.transferred]
lemmas cequalityD2 = equalityD2[Transfer.transferred]
lemmas cequalityE = equalityE[Transfer.transferred]
lemmas cequalityCE[elim] = equalityCE[Transfer.transferred]
lemmas eqcset-imp-iff = eqset-imp-iff[Transfer.transferred]
lemmas eqelem-imp-iff = equelem-imp-iff[Transfer.transferred]
lemmas cempty-iff[simp] = empty-iff[Transfer.transferred]
lemmas cempty-fsubsetI[iff] = empty-subsetI[Transfer.transferred]
lemmas equals-cemptyI = equalsOI[Transfer.transferred]
lemmas equals-cemptyD = equalsOD[Transfer.transferred]
lemmas cBall-cempty[simp] = ball-empty[Transfer.transferred]
lemmas cBex-cempty[simp] = bex-empty[Transfer.transferred]
lemmas cInt-iff[simp] = Int-iff[Transfer.transferred]
lemmas cIntI[intro!] = IntI[Transfer.transferred]
lemmas cIntD1 = IntD1[Transfer.transferred]
lemmas cIntD2 = IntD2[Transfer.transferred]
lemmas cIntE[elim!] = IntE[Transfer.transferred]
lemmas cUn-iff[simp] = Un-iff[Transfer.transferred]
lemmas cUnI1[elim?] = UnI1[Transfer.transferred]
lemmas cUnI2[elim?] = UnI2[Transfer.transferred]
lemmas cUnCI[intro!] = UnCI[Transfer.transferred]
lemmas cuUnE[elim!] = UnE[Transfer.transferred]
lemmas cDiff-iff[simp] = Diff-iff[Transfer.transferred]
lemmas cDiffI[intro!] = DiffI[Transfer.transferred]

```

```

lemmas cDiffD1 = DiffD1[Transfer.transferred]
lemmas cDiffD2 = DiffD2[Transfer.transferred]
lemmas cDiffE[elim!] = DiffE[Transfer.transferred]
lemmas cinsert-iff[simp] = insert-iff[Transfer.transferred]
lemmas cinsertI1 = insertI1[Transfer.transferred]
lemmas cinsertI2 = insertI2[Transfer.transferred]
lemmas cinsertE[elim!] = insertE[Transfer.transferred]
lemmas cinsertCI[intro!] = insertCI[Transfer.transferred]
lemmas csubset-cinsert-iff = subset-insert-iff[Transfer.transferred]
lemmas cinsert-ident = insert-ident[Transfer.transferred]
lemmas csingletonI[intro!,no-atp] = singletonI[Transfer.transferred]
lemmas csingletonD[dest!,no-atp] = singletonD[Transfer.transferred]
lemmas fsingletonE = csingletonD [elim-format]
lemmas csingleton-iff = singleton-iff[Transfer.transferred]
lemmas csingleton-inject[dest!] = singleton-inject[Transfer.transferred]
lemmas csingleton-finsert-inj-eq[iff,no-atp] = singleton-insert-inj-eq[Transfer.transferred]
lemmas csingleton-finsert-inj-eq'[iff,no-atp] = singleton-insert-inj-eq'[Transfer.transferred]
lemmas csubset-csingletonD = subset-singletonD[Transfer.transferred]
lemmas cDiff-single-cinsert = Diff-single-insert[Transfer.transferred]
lemmas cdoubleton-eq-iff = doubleton-eq-iff[Transfer.transferred]
lemmas cUn-csingleton-iff = Un-singleton-iff[Transfer.transferred]
lemmas csingleton-cUn-iff = singleton-Un-iff[Transfer.transferred]
lemmas cimage-eqI[simp, intro] = image-eqI[Transfer.transferred]
lemmas cimageI = imageI[Transfer.transferred]
lemmas rev-cimage-eqI = rev-image-eqI[Transfer.transferred]
lemmas cimageE[elim!] = imageE[Transfer.transferred]
lemmas Compr-cimage-eq = Compr-image-eq[Transfer.transferred]
lemmas cimage-cUn = image-Un[Transfer.transferred]
lemmas cimage-iff = image-iff[Transfer.transferred]
lemmas cimage-csubset-iff[no-atp] = image-subset-iff[Transfer.transferred]
lemmas cimage-csubsetI = image-subsetI[Transfer.transferred]
lemmas cimage-ident[simp] = image-ident[Transfer.transferred]
lemmas if-split-cin1 = if-split-mem1[Transfer.transferred]
lemmas if-split-cin2 = if-split-mem2[Transfer.transferred]
lemmas cpsubsetI[intro!,no-atp] = psubsetI[Transfer.transferred]
lemmas cpsubsetE[elim!,no-atp] = psubsetE[Transfer.transferred]
lemmas cpsubset-finsert-iff = psubset-insert-iff[Transfer.transferred]
lemmas cpsubset-eq = psubset-eq[Transfer.transferred]
lemmas cpsubset-imp-fsubset = psubset-imp-subset[Transfer.transferred]
lemmas cpsubset-trans = psubset-trans[Transfer.transferred]
lemmas cpsubsetD = psubsetD[Transfer.transferred]
lemmas cpsubset-csubset-trans = psubset-subset-trans[Transfer.transferred]
lemmas csubset-cpsubset-trans = subset-psubset-trans[Transfer.transferred]
lemmas cpsubset-imp-ex-fmem = psubset-imp-ex-mem[Transfer.transferred]
lemmas csubset-cinsertI = subset-insertI[Transfer.transferred]
lemmas csubset-cinsertI2 = subset-insertI2[Transfer.transferred]
lemmas csubset-cinsert = subset-insert[Transfer.transferred]
lemmas cUn-upper1 = Un-upper1[Transfer.transferred]
lemmas cUn-upper2 = Un-upper2[Transfer.transferred]

```

```

lemmas cUn-least = Un-least[Transfer.transferred]
lemmas cInt-lower1 = Int-lower1[Transfer.transferred]
lemmas cInt-lower2 = Int-lower2[Transfer.transferred]
lemmas cInt-greatest = Int-greatest[Transfer.transferred]
lemmas cDiff-csubset = Diff-subset[Transfer.transferred]
lemmas cDiff-csubset-conv = Diff-subset-conv[Transfer.transferred]
lemmas csubset-cempty[simp] = subset-empty[Transfer.transferred]
lemmas not-cpsubset-cempty[iff] = not-psubset-empty[Transfer.transferred]
lemmas cinsert-is-cUn = insert-is-Un[Transfer.transferred]
lemmas cinsert-not-cempty[simp] = insert-not-empty[Transfer.transferred]
lemmas cempty-not-cinsert = empty-not-insert[Transfer.transferred]
lemmas cinsert-absorb = insert-absorb[Transfer.transferred]
lemmas cinsert-absorb2[simp] = insert-absorb2[Transfer.transferred]
lemmas cinsert-commute = insert-commute[Transfer.transferred]
lemmas cinsert-csubset[simp] = insert-subset[Transfer.transferred]
lemmas cinsert-cinter-cinsert[simp] = insert-inter-insert[Transfer.transferred]
lemmas cinsert-disjoint[simp,no-atp] = insert-disjoint[Transfer.transferred]
lemmas disjoint-cinsert[simp,no-atp] = disjoint-insert[Transfer.transferred]
lemmas cimage-cempty[simp] = image-empty[Transfer.transferred]
lemmas cimage-cinsert[simp] = image-insert[Transfer.transferred]
lemmas cimage-constant = image-constant[Transfer.transferred]
lemmas cimage-constant-conv = image-constant-conv[Transfer.transferred]
lemmas cimage-cimage = image-image[Transfer.transferred]
lemmas cinsert-cimage[simp] = insert-image[Transfer.transferred]
lemmas cimage-is-cempty[iff] = image-is-empty[Transfer.transferred]
lemmas cempty-is-cimage[iff] = empty-is-image[Transfer.transferred]
lemmas cimage-cong = image-cong[Transfer.transferred]
lemmas cimage-cInt-csubset = image-Int-subset[Transfer.transferred]
lemmas cimage-cDiff-csubset = image-diff-subset[Transfer.transferred]
lemmas cInt-absorb = Int-absorb[Transfer.transferred]
lemmas cInt-left-absorb = Int-left-absorb[Transfer.transferred]
lemmas cInt-commute = Int-commute[Transfer.transferred]
lemmas cInt-left-commute = Int-left-commute[Transfer.transferred]
lemmas cInt-assoc = Int-assoc[Transfer.transferred]
lemmas cInt-ac = Int-ac[Transfer.transferred]
lemmas cInt-absorb1 = Int-absorb1[Transfer.transferred]
lemmas cInt-absorb2 = Int-absorb2[Transfer.transferred]
lemmas cInt-cempty-left = Int-empty-left[Transfer.transferred]
lemmas cInt-cempty-right = Int-empty-right[Transfer.transferred]
lemmas disjoint-iff-cnot-equal = disjoint-iff-not-equal[Transfer.transferred]
lemmas cInt-cUn-distrib = Int-Un-distrib[Transfer.transferred]
lemmas cInt-cUn-distrib2 = Int-Un-distrib2[Transfer.transferred]
lemmas cInt-csubset-iff[no-atp, simp] = Int-subset-iff[Transfer.transferred]
lemmas cUn-absorb = Un-absorb[Transfer.transferred]
lemmas cUn-left-absorb = Un-left-absorb[Transfer.transferred]
lemmas cUn-commute = Un-commute[Transfer.transferred]
lemmas cUn-left-commute = Un-left-commute[Transfer.transferred]
lemmas cUn-assoc = Un-assoc[Transfer.transferred]
lemmas cUn-ac = Un-ac[Transfer.transferred]

```

```

lemmas cUn-absorb1 = Un-absorb1[Transfer.transferred]
lemmas cUn-absorb2 = Un-absorb2[Transfer.transferred]
lemmas cUn-empty-left = Un-empty-left[Transfer.transferred]
lemmas cUn-empty-right = Un-empty-right[Transfer.transferred]
lemmas cUn-cinsert-left[simp] = Un-insert-left[Transfer.transferred]
lemmas cUn-cinsert-right[simp] = Un-insert-right[Transfer.transferred]
lemmas cInt-cinsert-left = Int-insert-left[Transfer.transferred]
lemmas cInt-cinsert-left-if0[simp] = Int-insert-left-if0[Transfer.transferred]
lemmas cInt-cinsert-left-if1[simp] = Int-insert-left-if1[Transfer.transferred]
lemmas cInt-cinsert-right = Int-insert-right[Transfer.transferred]
lemmas cInt-cinsert-right-if0[simp] = Int-insert-right-if0[Transfer.transferred]
lemmas cInt-cinsert-right-if1[simp] = Int-insert-right-if1[Transfer.transferred]
lemmas cUn-cInt-distrib = Un-Int-distrib[Transfer.transferred]
lemmas cUn-cInt-distrib2 = Un-Int-distrib2[Transfer.transferred]
lemmas cUn-cInt-crazy = Un-Int-crazy[Transfer.transferred]
lemmas csubset-cUn-eq = subset-Un-eq[Transfer.transferred]
lemmas cUn-empty[iff] = Un-empty[Transfer.transferred]
lemmas cUn-csubset-iff[no-atp, simp] = Un-subset-iff[Transfer.transferred]
lemmas cUn-cDiff-cInt = Un-Diff-Int[Transfer.transferred]
lemmas cDiff-cInt2 = Diff-Int2[Transfer.transferred]
lemmas cUn-cInt-assoc-eq = Un-Int-assoc-eq[Transfer.transferred]
lemmas cBall-cUn = ball-Un[Transfer.transferred]
lemmas cBex-cUn = bex-Un[Transfer.transferred]
lemmas cDiff-eq-empty-iff[simp, no-atp] = Diff-eq-empty-iff[Transfer.transferred]
lemmas cDiff-cancel[simp] = Diff-cancel[Transfer.transferred]
lemmas cDiff-idemp[simp] = Diff-idemp[Transfer.transferred]
lemmas cDiff-triv = Diff-triv[Transfer.transferred]
lemmas cempty-cDiff[simp] = empty-Diff[Transfer.transferred]
lemmas cDiff-empty[simp] = Diff-empty[Transfer.transferred]
lemmas cDiff-cinsert0[simp, no-atp] = Diff-insert0[Transfer.transferred]
lemmas cDiff-cinsert = Diff-insert[Transfer.transferred]
lemmas cDiff-cinsert2 = Diff-insert2[Transfer.transferred]
lemmas cinsert-cDiff-if = insert-Diff-if[Transfer.transferred]
lemmas cinsert-cDiff1[simp] = insert-Diff1[Transfer.transferred]
lemmas cinsert-cDiff-single[simp] = insert-Diff-single[Transfer.transferred]
lemmas cinsert-cDiff = insert-Diff[Transfer.transferred]
lemmas cDiff-cinsert-absorb = Diff-insert-absorb[Transfer.transferred]
lemmas cDiff-disjoint[simp] = Diff-disjoint[Transfer.transferred]
lemmas cDiff-partition = Diff-partition[Transfer.transferred]
lemmas double-cDiff = double-diff[Transfer.transferred]
lemmas cUn-cDiff-cancel[simp] = Un-Diff-cancel[Transfer.transferred]
lemmas cUn-cDiff-cancel2[simp] = Un-Diff-cancel2[Transfer.transferred]
lemmas cDiff-cUn = Diff-Un[Transfer.transferred]
lemmas cDiff-cInt = Diff-Int[Transfer.transferred]
lemmas cUn-cDiff = Un-Diff[Transfer.transferred]
lemmas cInt-cDiff = Int-Diff[Transfer.transferred]
lemmas cDiff-cInt-distrib = Diff-Int-distrib[Transfer.transferred]
lemmas cDiff-cInt-distrib2 = Diff-Int-distrib2[Transfer.transferred]
lemmas cset-eq-csubset = set-eq-subset[Transfer.transferred]

```

```

lemmas csubset-iff[no-atp] = subset-iff[Transfer.transferred]
lemmas csubset-iff-pfsubset-eq = subset-iff-pfsubset-eq[Transfer.transferred]
lemmas all-not-cin-conv[simp] = all-not-in-conv[Transfer.transferred]
lemmas ex-cin-conv = ex-in-conv[Transfer.transferred]
lemmas cimage-mono = image-mono[Transfer.transferred]
lemmas cinsert-mono = insert-mono[Transfer.transferred]
lemmas cunion-mono = Un-mono[Transfer.transferred]
lemmas cinter-mono = Int-mono[Transfer.transferred]
lemmas cminus-mono = Diff-mono[Transfer.transferred]
lemmas cin-mono = in-mono[Transfer.transferred]
lemmas cLeast-mono = Least-mono[Transfer.transferred]
lemmas cequalityI = equalityI[Transfer.transferred]
lemmas cUN-iff [simp] = UN-iff[Transfer.transferred]
lemmas cUN-I [intro] = UN-I[Transfer.transferred]
lemmas cUN-E [elim!] = UN-E[Transfer.transferred]
lemmas cUN-upper = UN-upper[Transfer.transferred]
lemmas cUN-least = UN-least[Transfer.transferred]
lemmas cUN-cinsert-distrib = UN-insert-distrib[Transfer.transferred]
lemmas cUN-empty [simp] = UN-empty[Transfer.transferred]
lemmas cUN-empty2 [simp] = UN-empty2[Transfer.transferred]
lemmas cUN-absorb = UN-absorb[Transfer.transferred]
lemmas cUN-cinsert [simp] = UN-insert[Transfer.transferred]
lemmas cUN-cUn [simp] = UN-Un[Transfer.transferred]
lemmas cUN-cUN-flatten = UN-UN-flatten[Transfer.transferred]
lemmas cUN-csubset-iff = UN-subset-iff[Transfer.transferred]
lemmas cUN-constant [simp] = UN-constant[Transfer.transferred]
lemmas cimage-cUnion = image-Union[Transfer.transferred]
lemmas cUNION-cempty-conv [simp] = UNION-empty-conv[Transfer.transferred]
lemmas cBall-cUN = ball-UN[Transfer.transferred]
lemmas cBex-cUN = bex-UN[Transfer.transferred]
lemmas cUn-eq-cUN = Un-eq-UN[Transfer.transferred]
lemmas cUN-mono = UN-mono[Transfer.transferred]
lemmas cimage-cUN = image-UN[Transfer.transferred]
lemmas cUN-csingleton [simp] = UN-singleton[Transfer.transferred]

```

23.3 Additional lemmas

23.3.1 *cempty*

lemma cemptyE [elim!]: *cin a cempty* \implies *P* \langle proof \rangle

23.3.2 *cinsert*

lemma countable-insert-iff: *countable (insert x A)* \longleftrightarrow *countable A* \langle proof \rangle

lemma set-cinsert:

assumes *cin x A*

obtains *B* **where** *A = cinsert x B* **and** \neg *cin x B*

\langle proof \rangle

lemma *mk-disjoint-cinsert*: $\text{cin } a \ A \implies \exists B. \ A = \text{cinsert } a \ B \wedge \neg \text{cin } a \ B$
 ⟨proof⟩

23.3.3 *cimage*

lemma *subset-cimage-iff*: $\text{csubset-eq } B \ (\text{cimage } f \ A) \longleftrightarrow (\exists AA. \ \text{csubset-eq } AA \ A \wedge B = \text{cimage } f \ AA)$
 ⟨proof⟩

23.3.4 bounded quantification

lemma *cBex-simps* [*simp*, *no-atp*]:
 $\bigwedge A \ P \ Q. \ \text{cBex } A \ (\lambda x. \ P \ x \wedge Q) = (\text{cBex } A \ P \wedge Q)$
 $\bigwedge A \ P \ Q. \ \text{cBex } A \ (\lambda x. \ P \wedge Q \ x) = (P \wedge \text{cBex } A \ Q)$
 $\bigwedge P. \ \text{cBex } \text{empty } P = \text{False}$
 $\bigwedge a \ B \ P. \ \text{cBex } (\text{cinsert } a \ B) \ P = (P \ a \vee \text{cBex } B \ P)$
 $\bigwedge A \ P \ f. \ \text{cBex } (\text{cimage } f \ A) \ P = \text{cBex } A \ (\lambda x. \ P \ (f \ x))$
 $\bigwedge A \ P. \ (\neg \text{cBex } A \ P) = \text{cBall } A \ (\lambda x. \neg P \ x)$
 ⟨proof⟩

lemma *cBall-simps* [*simp*, *no-atp*]:
 $\bigwedge A \ P \ Q. \ \text{cBall } A \ (\lambda x. \ P \ x \vee Q) = (\text{cBall } A \ P \vee Q)$
 $\bigwedge A \ P \ Q. \ \text{cBall } A \ (\lambda x. \ P \vee Q \ x) = (P \vee \text{cBall } A \ Q)$
 $\bigwedge A \ P \ Q. \ \text{cBall } A \ (\lambda x. \ P \longrightarrow Q \ x) = (P \longrightarrow \text{cBall } A \ Q)$
 $\bigwedge A \ P \ Q. \ \text{cBall } A \ (\lambda x. \ P \ x \longrightarrow Q) = (\text{cBex } A \ P \longrightarrow Q)$
 $\bigwedge P. \ \text{cBall } \text{empty } P = \text{True}$
 $\bigwedge a \ B \ P. \ \text{cBall } (\text{cinsert } a \ B) \ P = (P \ a \wedge \text{cBall } B \ P)$
 $\bigwedge A \ P \ f. \ \text{cBall } (\text{cimage } f \ A) \ P = \text{cBall } A \ (\lambda x. \ P \ (f \ x))$
 $\bigwedge A \ P. \ (\neg \text{cBall } A \ P) = \text{cBex } A \ (\lambda x. \neg P \ x)$
 ⟨proof⟩

lemma *atomize-cBall*:
 $(\bigwedge x. \ \text{cin } x \ A \implies P \ x) == \text{Trueprop } (\text{cBall } A \ (\lambda x. \ P \ x))$
 ⟨proof⟩

23.3.5 *cUnion*

lemma *cUNION-cimage*: $\text{cUNION } (\text{cimage } f \ A) \ g = \text{cUNION } A \ (g \circ f)$
 ⟨proof⟩

23.4 Setup for Lifting/Transfer

23.4.1 Relator and predicator properties

lift-definition *rel-cset* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \ \text{cset} \Rightarrow 'b \ \text{cset} \Rightarrow \text{bool}$
 is *rel-set parametric rel-set-transfer* ⟨proof⟩

lemma *rel-cset-alt-def*:
 $\text{rel-cset } R \ a \ b \longleftrightarrow$

$(\forall t \in \text{rcset } a. \exists u \in \text{rcset } b. R \ t \ u) \wedge$
 $(\forall t \in \text{rcset } b. \exists u \in \text{rcset } a. R \ u \ t)$
 $\langle \text{proof} \rangle$

lemma *rel-cset-iff*:

$\text{rel-cset } R \ a \ b \longleftrightarrow$
 $(\forall t. \text{cin } t \ a \longrightarrow (\exists u. \text{cin } u \ b \wedge R \ t \ u)) \wedge$
 $(\forall t. \text{cin } t \ b \longrightarrow (\exists u. \text{cin } u \ a \wedge R \ u \ t))$
 $\langle \text{proof} \rangle$

lemma *rel-cset-cUNION*:

$\llbracket \text{rel-cset } Q \ A \ B; \text{rel-fun } Q \ (\text{rel-cset } R) \ f \ g \rrbracket$
 $\implies \text{rel-cset } R \ (\text{cUnion } (\text{cimage } f \ A)) \ (\text{cUnion } (\text{cimage } g \ B))$
 $\langle \text{proof} \rangle$

lemma *rel-cset-csingle-iff [simp]*: $\text{rel-cset } R \ (\text{csingle } x) \ (\text{csingle } y) \longleftrightarrow R \ x \ y$
 $\langle \text{proof} \rangle$

23.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemmas *empty-parametric [transfer-rule]* = *empty-transfer[Transfer.transferred]*

lemma *cinsert-parametric [transfer-rule]*:

$(A \implies \text{rel-cset } A \implies \text{rel-cset } A) \text{ cinsert cinsert}$
 $\langle \text{proof} \rangle$

lemma *cUn-parametric [transfer-rule]*:

$(\text{rel-cset } A \implies \text{rel-cset } A \implies \text{rel-cset } A) \text{ cUn cUn}$
 $\langle \text{proof} \rangle$

lemma *cUnion-parametric [transfer-rule]*:

$(\text{rel-cset } (\text{rel-cset } A) \implies \text{rel-cset } A) \text{ cUnion cUnion}$
 $\langle \text{proof} \rangle$

lemma *cimage-parametric [transfer-rule]*:

$((A \implies B) \implies \text{rel-cset } A \implies \text{rel-cset } B) \text{ cimage cimage}$
 $\langle \text{proof} \rangle$

lemma *cBall-parametric [transfer-rule]*:

$(\text{rel-cset } A \implies (A \implies (=)) \implies (=)) \text{ cBall cBall}$
 $\langle \text{proof} \rangle$

lemma *cBex-parametric [transfer-rule]*:

$(\text{rel-cset } A \implies (A \implies (=)) \implies (=)) \text{ cBex cBex}$
 $\langle \text{proof} \rangle$

lemma *rel-cset-parametric* [transfer-rule]:
 $((A ==> B ==> (=)) ==> \text{rel-cset } A ==> \text{rel-cset } B ==> (=))$
rel-cset rel-cset
 ⟨proof⟩

Rules requiring bi-unique, bi-total or right-total relations

lemma *cin-parametric* [transfer-rule]:
 $\text{bi-unique } A \implies (A ==> \text{rel-cset } A ==> (=)) \text{ cin cin}$
 ⟨proof⟩

lemma *cInt-parametric* [transfer-rule]:
 $\text{bi-unique } A \implies (\text{rel-cset } A ==> \text{rel-cset } A ==> \text{rel-cset } A) \text{ cInt cInt}$
 ⟨proof⟩

lemma *cDiff-parametric* [transfer-rule]:
 $\text{bi-unique } A \implies (\text{rel-cset } A ==> \text{rel-cset } A ==> \text{rel-cset } A) \text{ cDiff cDiff}$
 ⟨proof⟩

lemma *csubset-parametric* [transfer-rule]:
 $\text{bi-unique } A \implies (\text{rel-cset } A ==> \text{rel-cset } A ==> (=)) \text{ csubset-eq csubset-eq}$
 ⟨proof⟩

end

lifting-update *cset.lifting*

lifting-forget *cset.lifting*

23.5 Registration as BNF

context

includes *cardinal-syntax*

begin

lemma *card-of-countable-sets-range*:
 fixes $A :: 'a \text{ set}$
 shows $|\{X. X \subseteq A \wedge \text{countable } X \wedge X \neq \{\}\}| \leq_o |\{f::\text{nat} \Rightarrow 'a. \text{range } f \subseteq A\}|$
 ⟨proof⟩

lemma *card-of-countable-sets-Func*:
 $|\{X. X \subseteq A \wedge \text{countable } X \wedge X \neq \{\}\}| \leq_o |A| \wedge^c \text{natLeq}$
 ⟨proof⟩

lemma *ordLeq-countable-subsets*:
 $|A| \leq_o |\{X. X \subseteq A \wedge \text{countable } X\}|$
 ⟨proof⟩

end

lemma *finite-countable-subset*:

finite $\{X. X \subseteq A \wedge \text{countable } X\} \longleftrightarrow \text{finite } A$
 <proof>

lemma *rcset-to-rcset*: $\text{countable } A \implies \text{rcset } (\text{the-inv rcset } A) = A$
including *cset.lifting*
 <proof>

lemma *Collect-Int-Times*: $\{(x, y). R \ x \ y\} \cap A \times B = \{(x, y). R \ x \ y \wedge x \in A \wedge y \in B\}$
 <proof>

lemma *rel-cset-aux*:
 $(\forall t \in \text{rcset } a. \exists u \in \text{rcset } b. R \ t \ u) \wedge (\forall t \in \text{rcset } b. \exists u \in \text{rcset } a. R \ u \ t) \longleftrightarrow$
 $((\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} (\text{cimage fst}))^{-1-1} \text{ OO}$
 $\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} (\text{cimage snd})) \ a \ b \ (\text{is ?L = ?R})$
 <proof> **including** *cset.lifting*
 <proof>

context
includes *cardinal-syntax*
begin

bnf 'a cset
map: *cimage*
sets: *rcset*
bd: *card-suc natLeq*
wits: *empty*
rel: *rel-cset*
 <proof> **including** *cset.lifting* <proof> **including** *cset.lifting* <proof> **including**
cset.lifting <proof>

end

end

24 Debugging facilities for code generated towards Isabelle/ML

theory *Debug*
imports *Main*
begin

context
begin

qualified definition *trace* :: *String.literal* \Rightarrow *unit* **where**
 [*simp*]: *trace* *s* = ()

qualified definition *tracing* :: *String.literal* \Rightarrow 'a \Rightarrow 'a **where**
 [simp]: *tracing* s = id

lemma [code]:
tracing s = (let u = trace s in id)
 ⟨proof⟩ **definition** *flush* :: 'a \Rightarrow unit **where**
 [simp]: *flush* x = ()

qualified definition *flushing* :: 'a \Rightarrow 'b \Rightarrow 'b **where**
 [simp]: *flushing* x = id

lemma [code, code-unfold]:
flushing x = (let u = flush x in id)
 ⟨proof⟩ **definition** *timing* :: *String.literal* \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b **where**
 [simp]: *timing* s f x = f x

end

code-printing

constant *Debug.trace* \rightarrow (Eval) *Output.tracing*
 | **constant** *Debug.flush* \rightarrow (Eval) *Output.tracing*/ (@{make'-string} -) — note
 indirection via antiquotation
 | **constant** *Debug.timing* \rightarrow (Eval) *Timing.timeap'-msg*

code-reserved (Eval) *Output Timing*

end

25 Sequence of Properties on Subsequences

theory *Diagonal-Subsequence*
imports *Complex-Main*
begin

locale *subseqs* =
fixes *P*::nat \Rightarrow (nat \Rightarrow nat) \Rightarrow bool
assumes *ex-subseq*: $\bigwedge n s. \text{strict-mono } (s::\text{nat}\Rightarrow\text{nat}) \implies \exists r'. \text{strict-mono } r' \wedge$
P n (*s* \circ *r'*)
begin

definition *reduce* **where** *reduce* s n = (SOME *r'*::nat \Rightarrow nat. *strict-mono* *r'* \wedge *P* n
 (*s* \circ *r'*))

lemma *subseq-reduce*[intro, simp]:
strict-mono s \implies *strict-mono* (*reduce* s n)
 ⟨proof⟩

lemma *reduce-holds*:

strict-mono $s \implies P\ n\ (s \circ \text{reduce}\ s\ n)$
 $\langle \text{proof} \rangle$

primrec *seqseq* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{seqseq}\ 0 = \text{id}$
 $|\ \text{seqseq}\ (\text{Suc}\ n) = \text{seqseq}\ n \circ \text{reduce}\ (\text{seqseq}\ n)\ n$

lemma *subseq-seqseq*[*intro, simp*]: *strict-mono* (*seqseq* n)
 $\langle \text{proof} \rangle$

lemma *seqseq-holds*:
 $P\ n\ (\text{seqseq}\ (\text{Suc}\ n))$
 $\langle \text{proof} \rangle$

definition *diagseq* :: $\text{nat} \Rightarrow \text{nat}$ **where** *diagseq* $i = \text{seqseq}\ i\ i$

lemma *diagseq-mono*: *diagseq* $n < \text{diagseq}\ (\text{Suc}\ n)$
 $\langle \text{proof} \rangle$

lemma *subseq-diagseq*: *strict-mono* *diagseq*
 $\langle \text{proof} \rangle$

primrec *fold-reduce* **where**
 $\text{fold-reduce}\ n\ 0 = \text{id}$
 $|\ \text{fold-reduce}\ n\ (\text{Suc}\ k) = \text{fold-reduce}\ n\ k \circ \text{reduce}\ (\text{seqseq}\ (n + k))\ (n + k)$

lemma *subseq-fold-reduce*[*intro, simp*]: *strict-mono* (*fold-reduce* $n\ k$)
 $\langle \text{proof} \rangle$

lemma *ex-subseq-reduce-index*: $\text{seqseq}\ (n + k) = \text{seqseq}\ n \circ \text{fold-reduce}\ n\ k$
 $\langle \text{proof} \rangle$

lemma *seqseq-fold-reduce*: $\text{seqseq}\ n = \text{fold-reduce}\ 0\ n$
 $\langle \text{proof} \rangle$

lemma *diagseq-fold-reduce*: $\text{diagseq}\ n = \text{fold-reduce}\ 0\ n\ n$
 $\langle \text{proof} \rangle$

lemma *fold-reduce-add*: $\text{fold-reduce}\ 0\ (m + n) = \text{fold-reduce}\ 0\ m \circ \text{fold-reduce}\ m\ n$
 $\langle \text{proof} \rangle$

lemma *diagseq-add*: $\text{diagseq}\ (k + n) = (\text{seqseq}\ k \circ (\text{fold-reduce}\ k\ n))\ (k + n)$
 $\langle \text{proof} \rangle$

lemma *diagseq-sub*:
assumes $m \leq n$ **shows** $\text{diagseq}\ n = (\text{seqseq}\ m \circ (\text{fold-reduce}\ m\ (n - m)))\ n$
 $\langle \text{proof} \rangle$

lemma *subseq-diagonal-rest*: *strict-mono* $(\lambda x. \text{fold-reduce } k \ x \ (k + x))$
 $\langle \text{proof} \rangle$

lemma *diagseq-seqseq*: $\text{diagseq} \circ ((+) \ k) = (\text{seqseq } k \circ (\lambda x. \text{fold-reduce } k \ x \ (k + x)))$
 $\langle \text{proof} \rangle$

lemma *diagseq-holds*:
assumes *subseq-stable*: $\bigwedge r \ s \ n. \text{strict-mono } r \implies P \ n \ s \implies P \ n \ (s \circ r)$
shows $P \ k \ (\text{diagseq} \circ ((+) \ (\text{Suc } k)))$
 $\langle \text{proof} \rangle$

end

end

26 Common discrete functions

theory *Discrete-Functions*
imports *Complex-Main*
begin

26.1 Discrete logarithm

fun *floor-log* :: $\text{nat} \Rightarrow \text{nat}$
where $[\text{simp del}]$: $\text{floor-log } n = (\text{if } n < 2 \text{ then } 0 \text{ else } \text{Suc } (\text{floor-log } (n \text{ div } 2)))$

lemma *floor-log-induct* [*consumes 1, case-names one double*]:
fixes $n :: \text{nat}$
assumes $n > 0$
assumes *one*: $P \ 1$
assumes *double*: $\bigwedge n. n \geq 2 \implies P \ (n \text{ div } 2) \implies P \ n$
shows $P \ n$
 $\langle \text{proof} \rangle$

lemma *floor-log-zero* [*simp*]: $\text{floor-log } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *floor-log-one* [*simp*]: $\text{floor-log } 1 = 0$
 $\langle \text{proof} \rangle$

lemma *floor-log-Suc-zero* [*simp*]: $\text{floor-log } (\text{Suc } 0) = 0$
 $\langle \text{proof} \rangle$

lemma *floor-log-rec*: $n \geq 2 \implies \text{floor-log } n = \text{Suc } (\text{floor-log } (n \text{ div } 2))$
 $\langle \text{proof} \rangle$

lemma *floor-log-twice* [*simp*]: $n \neq 0 \implies \text{floor-log } (2 * n) = \text{Suc } (\text{floor-log } n)$
 $\langle \text{proof} \rangle$

lemma *floor-log-half* [*simp*]: $\text{floor-log } (n \text{ div } 2) = \text{floor-log } n - 1$
 $\langle \text{proof} \rangle$

lemma *floor-log-power* [*simp*]: $\text{floor-log } (2^n) = n$
 $\langle \text{proof} \rangle$

lemma *floor-log-mono*: *mono floor-log*
 $\langle \text{proof} \rangle$

lemma *floor-log-exp2-le*:
assumes $n > 0$
shows $2^{\text{floor-log } n} \leq n$
 $\langle \text{proof} \rangle$

lemma *floor-log-exp2-gt*: $2 * 2^{\text{floor-log } n} > n$
 $\langle \text{proof} \rangle$

lemma *floor-log-exp2-ge*: $2 * 2^{\text{floor-log } n} \geq n$
 $\langle \text{proof} \rangle$

lemma *floor-log-le-iff*: $m \leq n \implies \text{floor-log } m \leq \text{floor-log } n$
 $\langle \text{proof} \rangle$

lemma *floor-log-eqI*:
assumes $n > 0$ $2^k \leq n$ $n < 2 * 2^k$
shows $\text{floor-log } n = k$
 $\langle \text{proof} \rangle$

lemma *floor-log-altdef*: $\text{floor-log } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \text{nat } \lfloor \log 2 (\text{real-of-nat } n) \rfloor)$
 $\langle \text{proof} \rangle$

26.2 Discrete square root

definition *floor-sqrt* :: $\text{nat} \Rightarrow \text{nat}$
where $\text{floor-sqrt } n = \text{Max } \{m. m^2 \leq n\}$

lemma *floor-sqrt-aux*:
fixes $n :: \text{nat}$
shows $\text{finite } \{m. m^2 \leq n\}$ **and** $\{m. m^2 \leq n\} \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-unique*:
assumes $m^2 \leq n$ $n < (\text{Suc } m)^2$
shows $\text{floor-sqrt } n = m$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-inverse-power2* [*simp*]: $\text{floor-sqrt } (n^2) = n$

$\langle \text{proof} \rangle$

lemma *floor-sqrt-zero* [*simp*]: *floor-sqrt* 0 = 0
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-one* [*simp*]: *floor-sqrt* 1 = 1
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-Suc-0* [*simp*]:
 $\langle \text{floor-sqrt (Suc 0) = 1} \rangle$
 $\langle \text{proof} \rangle$

lemma *mono-floor-sqrt*: *mono floor-sqrt*
 $\langle \text{proof} \rangle$

lemma *mono-floor-sqrt'*: $m \leq n \implies \text{floor-sqrt } m \leq \text{floor-sqrt } n$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-greater-zero-iff* [*simp*]: *floor-sqrt* n > 0 \longleftrightarrow n > 0
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-power2-le* [*simp*]: $(\text{floor-sqrt } n)^2 \leq n$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-le*: *floor-sqrt* n \leq n
 $\langle \text{proof} \rangle$

Additional facts about the discrete square root, thanks to Julian Bien-darra, Manuel Eberl

lemma *Suc-floor-sqrt-power2-gt*: $n < (\text{Suc } (\text{floor-sqrt } n))^2$
 $\langle \text{proof} \rangle$

lemma *le-floor-sqrt-iff*: $x \leq \text{floor-sqrt } y \longleftrightarrow x^2 \leq y$
 $\langle \text{proof} \rangle$

lemma *le-floor-sqrtI*: $x^2 \leq y \implies x \leq \text{floor-sqrt } y$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-le-iff*:
 $\langle \text{floor-sqrt } y \leq x \longleftrightarrow (\forall z. z^2 \leq y \longrightarrow z \leq x) \rangle$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-leI*:
 $(\bigwedge z. z^2 \leq y \implies z \leq x) \implies \text{floor-sqrt } y \leq x$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-less-eq-half*:
 $\langle \text{floor-sqrt } n \leq \text{Suc } n \text{ div } 2 \rangle$
 $\langle \text{proof} \rangle$

lemma *floor-sqrt-Suc*:

floor-sqrt (*Suc* *n*) = (if $\exists m. \text{Suc } n = m^2$ then *Suc* (*floor-sqrt* *n*) else *floor-sqrt* *n*)
 ⟨proof⟩

Computation by divide and conquer

definition *floor-sqrt-between* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$

where *floor-sqrt-between-eq*:

$\langle \text{floor-sqrt-between } m \ q \ n =$
 (if *floor-sqrt* *n* $\in \{m..<m + q\}$ then *floor-sqrt* *n* else 0)⟩

— The 0 is not for relevant regular computation and can be chosen arbitrarily.

lemma *floor-sqrt-between-out-of-bounds*:

$\langle \text{floor-sqrt-between } m \ 0 \ n = 0 \rangle$
 ⟨proof⟩

lemma *floor-sqrt-between-singleton*:

$\langle \text{floor-sqrt-between } m \ (\text{Suc } 0) \ n =$
 (if $m^2 \leq n \wedge n < (\text{Suc } m)^2$ then *m* else 0)⟩
 ⟨proof⟩

lemma *floor-sqrt-between-rec*:

$\langle \text{floor-sqrt-between } m \ q \ n = ($
 let
 $r = q \text{ div } 2;$
 $p = m + r;$
 $s = p^2$
 in
 if $s = n$
 then *p*
 else if $s < n$
 then *floor-sqrt-between* (*m* + *r*) (*q* − *r*) *n*
 else *floor-sqrt-between* *m* *r* *n*
)⟩ if $\langle q > 0 \rangle$
 ⟨proof⟩

lemma *floor-sqrt-between-code* [code]:

$\langle \text{floor-sqrt-between } m \ q \ n = ($
 if $q = 0$ then 0
 else if $q = 1$
 then if $m^2 \leq n \wedge n < (\text{Suc } m)^2$
 then *m*
 else 0
 else
 let
 $r = q \text{ div } 2;$
 $p = m + r;$
 $s = p^2$


```

    in
      if s = n
      then p
      else if s < n
        then floor-sqrt-between (m + r) (q - r) n
        else floor-sqrt-between m r n
      )
  <proof>

lemma [code]:
  <floor-sqrt n = floor-sqrt-between 0 (Suc (Suc n div 2)) n>
  <proof>

end

```

27 Pi and Function Sets

```

theory FuncSet
  imports Main
  abbrevs PiE = PiE
  and PIE = ΠE
begin

definition Pi :: 'a set ⇒ ('a ⇒ 'b set) ⇒ ('a ⇒ 'b) set
  where Pi A B = {f. ∀ x. x ∈ A ⟶ f x ∈ B x}

definition extensional :: 'a set ⇒ ('a ⇒ 'b) set
  where extensional A = {f. ∀ x. x ∉ A ⟶ f x = undefined}

definition restrict :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'a ⇒ 'b
  where restrict f A = (λx. if x ∈ A then f x else undefined)

abbreviation funcset :: 'a set ⇒ 'b set ⇒ ('a ⇒ 'b) set
  where funcset A B ≡ Pi A (λ-. B)

open-bundle funcset-syntax
begin
  notation funcset (infixr <→> 60)
end

syntax
  -Pi :: pttrn ⇒ 'a set ⇒ 'b set ⇒ ('a ⇒ 'b) set
    (⟨⟨indent=3 notation=⟨binder Π∈⟩Π -∈-./ -⟩ 10⟩)
  -lam :: pttrn ⇒ 'a set ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
    (⟨⟨indent=3 notation=⟨binder λ∈⟩λ-∈-./ -⟩ [0, 0, 3] 3⟩)
syntax-consts
  -Pi ⇒ Pi and
  -lam ⇒ restrict
translations

```

$\Pi x \in A. B \Rightarrow \text{CONST } Pi \ A \ (\lambda x. B)$
 $\lambda x \in A. f \Rightarrow \text{CONST } restrict \ (\lambda x. f) \ A$

definition *compose* :: 'a set \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c)
where *compose* $A \ g \ f = (\lambda x \in A. g \ (f \ x))$

27.1 Basic Properties of Pi

lemma *Pi-I[intro!]*: $(\bigwedge x. x \in A \Longrightarrow f \ x \in B \ x) \Longrightarrow f \in Pi \ A \ B$
 $\langle proof \rangle$

lemma *Pi-I'[simp]*: $(\bigwedge x. x \in A \longrightarrow f \ x \in B \ x) \Longrightarrow f \in Pi \ A \ B$
 $\langle proof \rangle$

lemma *funcsetI*: $(\bigwedge x. x \in A \Longrightarrow f \ x \in B) \Longrightarrow f \in A \rightarrow B$
 $\langle proof \rangle$

lemma *Pi-mem*: $f \in Pi \ A \ B \Longrightarrow x \in A \Longrightarrow f \ x \in B \ x$
 $\langle proof \rangle$

lemma *Pi-iff*: $f \in Pi \ I \ X \longleftrightarrow (\forall i \in I. f \ i \in X \ i)$
 $\langle proof \rangle$

lemma *PiE [elim]*: $f \in Pi \ A \ B \Longrightarrow (f \ x \in B \ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
 $\langle proof \rangle$

lemma *Pi-cong*: $(\bigwedge w. w \in A \Longrightarrow f \ w = g \ w) \Longrightarrow f \in Pi \ A \ B \longleftrightarrow g \in Pi \ A \ B$
 $\langle proof \rangle$

lemma *funcset-id [simp]*: $(\lambda x. x) \in A \rightarrow A$
 $\langle proof \rangle$

lemma *funcset-mem*: $f \in A \rightarrow B \Longrightarrow x \in A \Longrightarrow f \ x \in B$
 $\langle proof \rangle$

lemma *funcset-image*: $f \in A \rightarrow B \Longrightarrow f \ ' \ A \subseteq B$
 $\langle proof \rangle$

lemma *image-subset-iff-funcset*: $F \ ' \ A \subseteq B \longleftrightarrow F \in A \rightarrow B$
 $\langle proof \rangle$

lemma *funcset-to-empty-iff*: $A \rightarrow \{\} = (\text{if } A = \{\} \text{ then } UNIV \text{ else } \{\})$
 $\langle proof \rangle$

lemma *Pi-eq-empty[simp]*: $(\Pi x \in A. B \ x) = \{\} \longleftrightarrow (\exists x \in A. B \ x = \{\})$
 $\langle proof \rangle$

lemma *Pi-empty [simp]*: $Pi \ \{\} \ B = UNIV$
 $\langle proof \rangle$

lemma *Pi-Int*: $Pi\ I\ E \cap Pi\ I\ F = (\Pi\ i \in I. E\ i \cap F\ i)$
 $\langle proof \rangle$

lemma *Pi-UN*:
fixes $A :: nat \Rightarrow 'i \Rightarrow 'a\ set$
assumes *finite I*
and *mono*: $\bigwedge i\ n\ m. i \in I \implies n \leq m \implies A\ n\ i \subseteq A\ m\ i$
shows $(\bigcup n. Pi\ I\ (A\ n)) = (\Pi\ i \in I. \bigcup n. A\ n\ i)$
 $\langle proof \rangle$

lemma *Pi-UNIV [simp]*: $A \rightarrow UNIV = UNIV$
 $\langle proof \rangle$

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x. x \in A \implies B\ x \subseteq C\ x) \implies Pi\ A\ B \subseteq Pi\ A\ C$
 $\langle proof \rangle$

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \implies Pi\ A\ B \subseteq Pi\ A'\ B$
 $\langle proof \rangle$

lemma *prod-final*:
assumes $1: fst \circ f \in Pi\ A\ B$
and $2: snd \circ f \in Pi\ A\ C$
shows $f \in (\Pi\ z \in A. B\ z \times C\ z)$
 $\langle proof \rangle$

lemma *Pi-split-domain[simp]*: $x \in Pi\ (I \cup J)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x \in Pi\ J\ X$
 $\langle proof \rangle$

lemma *Pi-split-insert-domain[simp]*: $x \in Pi\ (insert\ i\ I)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x\ i \in X\ i$
 $\langle proof \rangle$

lemma *Pi-cancel-fupd-range[simp]*: $i \notin I \implies x \in Pi\ I\ (B(i := b)) \longleftrightarrow x \in Pi\ I\ B$
 $\langle proof \rangle$

lemma *Pi-cancel-fupd[simp]*: $i \notin I \implies x(i := a) \in Pi\ I\ B \longleftrightarrow x \in Pi\ I\ B$
 $\langle proof \rangle$

lemma *Pi-fupd-iff*: $i \in I \implies f \in Pi\ I\ (B(i := A)) \longleftrightarrow f \in Pi\ (I - \{i\})\ B \wedge f\ i \in A$
 $\langle proof \rangle$

lemma *fst-Pi*: $fst \in A \times B \rightarrow A$ **and** *snd-Pi*: $snd \in A \times B \rightarrow B$
 $\langle proof \rangle$

27.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \rightarrow B \implies g \in B \rightarrow C \implies \text{compose } A \ g \ f \in A \rightarrow C$
 $\langle \text{proof} \rangle$

lemma *compose-assoc*:
assumes $f \in A \rightarrow B$
shows $\text{compose } A \ h \ (\text{compose } A \ g \ f) = \text{compose } A \ (\text{compose } B \ h \ g) \ f$
 $\langle \text{proof} \rangle$

lemma *compose-eq*: $x \in A \implies \text{compose } A \ g \ f \ x = g \ (f \ x)$
 $\langle \text{proof} \rangle$

lemma *surj-compose*: $f \text{ ‘ } A = B \implies g \text{ ‘ } B = C \implies \text{compose } A \ g \ f \text{ ‘ } A = C$
 $\langle \text{proof} \rangle$

27.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \implies (\bigwedge i. i \in J \implies f \ i = g \ i) \implies \text{restrict } f \ I = \text{restrict } g \ J$
 $\langle \text{proof} \rangle$

lemma *restrictI[intro!]*: $(\bigwedge x. x \in A \implies f \ x \in B \ x) \implies (\lambda x \in A. f \ x) \in \text{Pi } A \ B$
 $\langle \text{proof} \rangle$

lemma *restrict-apply[simp]*: $(\lambda y \in A. f \ y) \ x = (\text{if } x \in A \text{ then } f \ x \text{ else undefined})$
 $\langle \text{proof} \rangle$

lemma *restrict-apply'*: $x \in A \implies (\lambda y \in A. f \ y) \ x = f \ x$
 $\langle \text{proof} \rangle$

lemma *restrict-ext*: $(\bigwedge x. x \in A \implies f \ x = g \ x) \implies (\lambda x \in A. f \ x) = (\lambda x \in A. g \ x)$
 $\langle \text{proof} \rangle$

lemma *restrict-UNIV*: $\text{restrict } f \ \text{UNIV} = f$
 $\langle \text{proof} \rangle$

lemma *inj-on-restrict-eq [simp]*: $\text{inj-on } (\text{restrict } f \ A) \ A \longleftrightarrow \text{inj-on } f \ A$
 $\langle \text{proof} \rangle$

lemma *inj-on-restrict-iff*: $A \subseteq B \implies \text{inj-on } (\text{restrict } f \ B) \ A \longleftrightarrow \text{inj-on } f \ A$
 $\langle \text{proof} \rangle$

lemma *Id-compose*: $f \in A \rightarrow B \implies f \in \text{extensional } A \implies \text{compose } A \ (\lambda y \in B. y) \ f = f$
 $\langle \text{proof} \rangle$

lemma *compose-Id*: $g \in A \rightarrow B \implies g \in \text{extensional } A \implies \text{compose } A \ g \ (\lambda x \in A. x) = g$

$\langle proof \rangle$

lemma *image-restrict-eq* [simp]: $(restrict\ f\ A) \circ A = f \circ A$
 $\langle proof \rangle$

lemma *restrict-restrict*[simp]: $restrict\ (restrict\ f\ A)\ B = restrict\ f\ (A \cap B)$
 $\langle proof \rangle$

lemma *restrict-fupd*[simp]: $i \notin I \implies restrict\ (f\ (i := x))\ I = restrict\ f\ I$
 $\langle proof \rangle$

lemma *restrict-upd*[simp]: $i \notin I \implies (restrict\ f\ I)(i := y) = restrict\ (f(i := y))$
 $(insert\ i\ I)$
 $\langle proof \rangle$

lemma *restrict-Pi-cancel*: $restrict\ x\ I \in Pi\ I\ A \longleftrightarrow x \in Pi\ I\ A$
 $\langle proof \rangle$

lemma *sum-restrict'* [simp]: $sum'\ (\lambda i \in I. g\ i)\ I = sum'\ (\lambda i. g\ i)\ I$
 $\langle proof \rangle$

lemma *prod-restrict'* [simp]: $prod'\ (\lambda i \in I. g\ i)\ I = prod'\ (\lambda i. g\ i)\ I$
 $\langle proof \rangle$

27.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

lemma *bij-betwI*:
assumes $f \in A \rightarrow B$
and $g \in B \rightarrow A$
and $g \circ f: \bigwedge x. x \in A \implies g\ (f\ x) = x$
and $f \circ g: \bigwedge y. y \in B \implies f\ (g\ y) = y$
shows *bij-betw* $f\ A\ B$
 $\langle proof \rangle$

lemma *bij-betw-imp-funcset*: *bij-betw* $f\ A\ B \implies f \in A \rightarrow B$
 $\langle proof \rangle$

lemma *inj-on-compose*: *bij-betw* $f\ A\ B \implies inj-on\ g\ B \implies inj-on\ (compose\ A\ g\ f)$
 A
 $\langle proof \rangle$

lemma *bij-betw-compose*: *bij-betw* $f\ A\ B \implies bij-betw\ g\ B\ C \implies bij-betw\ (compose\ A\ g\ f)\ A\ C$
 $\langle proof \rangle$

lemma *bij-betw-restrict-eq* [simp]: *bij-betw* $(restrict\ f\ A)\ A\ B = bij-betw\ f\ A\ B$
 $\langle proof \rangle$

27.5 Extensionality

lemma *extensional-empty*[simp]: *extensional* $\{\}$ = $\{\lambda x. \text{undefined}\}$
 ⟨proof⟩

lemma *extensional-arb*: $f \in \text{extensional } A \implies x \notin A \implies f\ x = \text{undefined}$
 ⟨proof⟩

lemma *restrict-extensional* [simp]: *restrict* $f\ A \in \text{extensional } A$
 ⟨proof⟩

lemma *compose-extensional* [simp]: *compose* $A\ f\ g \in \text{extensional } A$
 ⟨proof⟩

lemma *extensionalityI*:
 assumes $f \in \text{extensional } A$
 and $g \in \text{extensional } A$
 and $\bigwedge x. x \in A \implies f\ x = g\ x$
 shows $f = g$
 ⟨proof⟩

lemma *extensional-restrict*: $f \in \text{extensional } A \implies \text{restrict } f\ A = f$
 ⟨proof⟩

lemma *extensional-subset*: $f \in \text{extensional } A \implies A \subseteq B \implies f \in \text{extensional } B$
 ⟨proof⟩

lemma *inv-into-funcset*: $f \text{ ‘ } A = B \implies (\lambda x \in B. \text{inv-into } A\ f\ x) \in B \rightarrow A$
 ⟨proof⟩

lemma *compose-inv-into-id*: $\text{bij-betw } f\ A\ B \implies \text{compose } A\ (\lambda y \in B. \text{inv-into } A\ f\ y)$
 $f = (\lambda x \in A. x)$
 ⟨proof⟩

lemma *compose-id-inv-into*: $f \text{ ‘ } A = B \implies \text{compose } B\ f\ (\lambda y \in B. \text{inv-into } A\ f\ y)$
 $= (\lambda x \in B. x)$
 ⟨proof⟩

lemma *extensional-insert*[intro, simp]:
 assumes $a \in \text{extensional } (\text{insert } i\ I)$
 shows $a(i := b) \in \text{extensional } (\text{insert } i\ I)$
 ⟨proof⟩

lemma *extensional-Int*[simp]: *extensional* $I \cap \text{extensional } I' = \text{extensional } (I \cap I')$
 ⟨proof⟩

lemma *extensional-UNIV*[simp]: *extensional* $UNIV = UNIV$
 ⟨proof⟩

lemma *restrict-extensional-sub*[intro]: $A \subseteq B \implies \text{restrict } f \ A \in \text{extensional } B$
 ⟨proof⟩

lemma *extensional-insert-undefined*[intro, simp]:
 $a \in \text{extensional } (\text{insert } i \ I) \implies a(i := \text{undefined}) \in \text{extensional } I$
 ⟨proof⟩

lemma *extensional-insert-cancel*[intro, simp]:
 $a \in \text{extensional } I \implies a \in \text{extensional } (\text{insert } i \ I)$
 ⟨proof⟩

27.6 Cardinality

lemma *card-inj*: $f \in A \rightarrow B \implies \text{inj-on } f \ A \implies \text{finite } B \implies \text{card } A \leq \text{card } B$
 ⟨proof⟩

lemma *card-bij*:
 assumes $f \in A \rightarrow B$ *inj-on* $f \ A$
 and $g \in B \rightarrow A$ *inj-on* $g \ B$
 and *finite* A *finite* B
 shows $\text{card } A = \text{card } B$
 ⟨proof⟩

27.7 Extensional Function Spaces

definition $PiE :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b \text{ set}) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
 where $PiE \ S \ T = Pi \ S \ T \cap \text{extensional } S$

abbreviation $Pi_E \ A \ B \equiv PiE \ A \ B$

syntax
 $-PiE :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a \Rightarrow 'b) \text{ set}$
 (⟨⟨indent=3 notation=binder $\Pi_E \in \rangle \Pi_E \ - \in \cdot / \ - \rangle \ 10$)

syntax-consts

$-PiE \equiv Pi_E$

translations

$\Pi_E \ x \in A. B \equiv CONST \ Pi_E \ A \ (\lambda x. B)$

abbreviation *extensional-funcset* :: $'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a \Rightarrow 'b) \text{ set}$ (**infixr** \hookrightarrow_E)
 60)

where $A \rightarrow_E B \equiv (\Pi_E \ i \in A. B)$

lemma *extensional-funcset-def*: $\text{extensional-funcset } S \ T = (S \rightarrow T) \cap \text{extensional } S$
 ⟨proof⟩

lemma *PiE-empty-domain*[simp]: $Pi_E \ \{\} \ T = \{\lambda x. \text{undefined}\}$
 ⟨proof⟩

lemma *PiE-UNIV-domain*: $Pi_E \ UNIV \ T = Pi \ UNIV \ T$

$\langle proof \rangle$

lemma *PiE-empty-range[simp]*: $i \in I \implies F\ i = \{\} \implies (\Pi_E\ i \in I. F\ i) = \{\}$
 $\langle proof \rangle$

lemma *PiE-eq-empty-iff*: $Pi_E\ I\ F = \{\} \longleftrightarrow (\exists\ i \in I. F\ i = \{\})$
 $\langle proof \rangle$

lemma *PiE-arb*: $f \in Pi_E\ S\ T \implies x \notin S \implies f\ x = undefined$
 $\langle proof \rangle$

lemma *PiE-mem*: $f \in Pi_E\ S\ T \implies x \in S \implies f\ x \in T$
 $\langle proof \rangle$

lemma *PiE-fun-upd*: $y \in T\ x \implies f \in Pi_E\ S\ T \implies f(x := y) \in Pi_E\ (insert\ x\ S)$
 T
 $\langle proof \rangle$

lemma *fun-upd-in-PiE*: $x \notin S \implies f \in Pi_E\ (insert\ x\ S)\ T \implies f(x := undefined) \in Pi_E\ S\ T$
 $\langle proof \rangle$

lemma *PiE-insert-eq*: $Pi_E\ (insert\ x\ S)\ T = (\lambda(y, g). g(x := y))\ ` (T\ x \times Pi_E\ S\ T)$
 $\langle proof \rangle$

lemma *PiE-Int*: $Pi_E\ I\ A \cap Pi_E\ I\ B = Pi_E\ I\ (\lambda x. A\ x \cap B\ x)$
 $\langle proof \rangle$

lemma *PiE-cong*: $(\bigwedge i. i \in I \implies A\ i = B\ i) \implies Pi_E\ I\ A = Pi_E\ I\ B$
 $\langle proof \rangle$

lemma *PiE-E [elim]*:
assumes $f \in Pi_E\ A\ B$
obtains $x \in A$ **and** $f\ x \in B\ x$
 $\mid x \notin A$ **and** $f\ x = undefined$
 $\langle proof \rangle$

lemma *PiE-I[intro!]*:
 $(\bigwedge x. x \in A \implies f\ x \in B\ x) \implies (\bigwedge x. x \notin A \implies f\ x = undefined) \implies f \in Pi_E\ A\ B$
 $\langle proof \rangle$

lemma *PiE-mono*: $(\bigwedge x. x \in A \implies B\ x \subseteq C\ x) \implies Pi_E\ A\ B \subseteq Pi_E\ A\ C$
 $\langle proof \rangle$

lemma *PiE-iff*: $f \in Pi_E\ I\ X \longleftrightarrow (\forall i \in I. f\ i \in X\ i) \wedge f \in extensional\ I$
 $\langle proof \rangle$

lemma *restrict-PiE-iff*: $\text{restrict } f \ I \in \text{Pi}_E \ I \ X \longleftrightarrow (\forall i \in I. f \ i \in X \ i)$
 $\langle \text{proof} \rangle$

lemma *ext-funcset-to-sing-iff* [simp]: $A \rightarrow_E \{a\} = \{\lambda x \in A. a\}$
 $\langle \text{proof} \rangle$

lemma *PiE-restrict*[simp]: $f \in \text{Pi}_E \ A \ B \implies \text{restrict } f \ A = f$
 $\langle \text{proof} \rangle$

lemma *restrict-PiE*[simp]: $\text{restrict } f \ I \in \text{Pi}_E \ I \ S \longleftrightarrow f \in \text{Pi} \ I \ S$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-subset*:
 assumes *ne*: $\bigwedge i. i \in I \implies F \ i \neq \{\}$ $\bigwedge i. i \in I \implies F' \ i \neq \{\}$
 and *eq*: $\text{Pi}_E \ I \ F = \text{Pi}_E \ I \ F'$
 and $i \in I$
 shows $F \ i \subseteq F' \ i$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-iff-not-empty*:
 assumes *ne*: $\bigwedge i. i \in I \implies F \ i \neq \{\}$ $\bigwedge i. i \in I \implies F' \ i \neq \{\}$
 shows $\text{Pi}_E \ I \ F = \text{Pi}_E \ I \ F' \longleftrightarrow (\forall i \in I. F \ i = F' \ i)$
 $\langle \text{proof} \rangle$

lemma *PiE-eq-iff*: $\text{Pi}_E \ I \ F = \text{Pi}_E \ I \ F' \longleftrightarrow (\forall i \in I. F \ i = F' \ i) \vee ((\exists i \in I. F \ i = \{\}) \wedge (\exists i \in I. F' \ i = \{\}))$
 $\langle \text{proof} \rangle$

lemma *extensional-funcset-fun-upd-restricts-rangeI*:
 $\forall y \in S. f \ x \neq f \ y \implies f \in (\text{insert } x \ S) \rightarrow_E T \implies f(x := \text{undefined}) \in S \rightarrow_E (T - \{f \ x\})$
 $\langle \text{proof} \rangle$

lemma *extensional-funcset-fun-upd-extends-rangeI*:
 assumes $a \in T \ f \in S \rightarrow_E (T - \{a\})$
 shows $f(x := a) \in \text{insert } x \ S \rightarrow_E T$
 $\langle \text{proof} \rangle$

lemma *subset-PiE*:
 $\text{Pi}_E \ I \ S \subseteq \text{Pi}_E \ I \ T \longleftrightarrow \text{Pi}_E \ I \ S = \{\} \vee (\forall i \in I. S \ i \subseteq T \ i)$ (**is** ?lhs \longleftrightarrow - \vee ?rhs)
 $\langle \text{proof} \rangle$

lemma *PiE-eq*: $\text{Pi}_E \ I \ S = \text{Pi}_E \ I \ T \longleftrightarrow \text{Pi}_E \ I \ S = \{\} \wedge \text{Pi}_E \ I \ T = \{\} \vee (\forall i \in I. S \ i = T \ i)$
 $\langle \text{proof} \rangle$

lemma *PiE-UNIV* [simp]: $\text{Pi}_E \ \text{UNIV} \ (\lambda i. \ \text{UNIV}) = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *image-projection-PiE*:

$(\lambda f. f \ i) \text{ ‘ } (PiE \ I \ S) = (if \ PiE \ I \ S = \{\} \text{ then } \{\} \text{ else if } i \in I \text{ then } S \ i \text{ else } \{undefined\})$
 $\langle proof \rangle$

lemma *PiE-singleton*:

assumes $f \in \text{extensional } A$
shows $PiE \ A \ (\lambda x. \{f \ x\}) = \{f\}$
 $\langle proof \rangle$

lemma *PiE-eq-singleton*: $(\Pi_E \ i \in I. S \ i) = \{\lambda i \in I. f \ i\} \longleftrightarrow (\forall i \in I. S \ i = \{f \ i\})$
 $\langle proof \rangle$

lemma *PiE-over-singleton-iff*: $(\Pi_E \ x \in \{a\}. B \ x) = (\bigcup b \in B \ a. \{\lambda x \in \{a\}. b\})$
 $\langle proof \rangle$

lemma *all-PiE-elements*:

$(\forall z \in PiE \ I \ S. \forall i \in I. P \ i \ (z \ i)) \longleftrightarrow PiE \ I \ S = \{\} \vee (\forall i \in I. \forall x \in S \ i. P \ i \ x)$
(is ?lhs = ?rhs)
 $\langle proof \rangle$

lemma *PiE-ext*: $\llbracket x \in PiE \ k \ s; y \in PiE \ k \ s; \bigwedge i. i \in k \implies x \ i = y \ i \rrbracket \implies x = y$
 $\langle proof \rangle$

27.7.1 Injective Extensional Function Spaces

lemma *extensional-funcset-fun-upd-inj-onI*:

assumes $f \in S \rightarrow_E (T - \{a\})$
and $\text{inj-on } f \ S$
shows $\text{inj-on } (f(x := a)) \ S$
 $\langle proof \rangle$

lemma *extensional-funcset-extend-domain-inj-on-eq*:

assumes $x \notin S$
shows $\{f. f \in (\text{insert } x \ S) \rightarrow_E T \wedge \text{inj-on } f \ (\text{insert } x \ S)\} =$
 $\{(\lambda(y, g). g(x:=y)) \text{ ‘ } \{(y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge \text{inj-on } g \ S\}\}$
 $\langle proof \rangle$

lemma *extensional-funcset-extend-domain-inj-onI*:

assumes $x \notin S$
shows $\text{inj-on } (\lambda(y, g). g(x := y)) \ \{(y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge \text{inj-on } g \ S\}$
 $\langle proof \rangle$

27.7.2 Misc properties of functions, composition and restriction from HOL Light

lemma *function-factors-left-gen*:

$(\forall x \ y. P \ x \wedge P \ y \wedge g \ x = g \ y \longrightarrow f \ x = f \ y) \longleftrightarrow (\exists h. \forall x. P \ x \longrightarrow f \ x = h(g \ x))$

(is ?lhs = ?rhs)
 ⟨proof⟩

lemma *function-factors-left*: $(\forall x y. (g x = g y) \longrightarrow (f x = f y)) \longleftrightarrow (\exists h. f = h \circ g)$
 ⟨proof⟩

lemma *function-factors-right-gen*: $(\forall x. P x \longrightarrow (\exists y. g y = f x)) \longleftrightarrow (\exists h. \forall x. P x \longrightarrow f x = g(h x))$
 ⟨proof⟩

lemma *function-factors-right*: $(\forall x. \exists y. g y = f x) \longleftrightarrow (\exists h. f = g \circ h)$
 ⟨proof⟩

lemma *restrict-compose-right*: $\text{restrict } (g \circ \text{restrict } f S) S = \text{restrict } (g \circ f) S$
 ⟨proof⟩

lemma *restrict-compose-left*: $f \restriction S \subseteq T \implies \text{restrict } (\text{restrict } g T \circ f) S = \text{restrict } (g \circ f) S$
 ⟨proof⟩

27.7.3 Cardinality

lemma *finite-PiE*: $\text{finite } S \implies (\bigwedge i. i \in S \implies \text{finite } (T i)) \implies \text{finite } (\Pi_E i \in S. T i)$
 ⟨proof⟩

lemma *inj-combinator*: $x \notin S \implies \text{inj-on } (\lambda(y, g). g(x := y)) (T x \times \text{Pi}_E S T)$
 ⟨proof⟩

lemma *card-PiE*: $\text{finite } S \implies \text{card } (\Pi_E i \in S. T i) = (\prod_{i \in S. \text{card } (T i)})$
 ⟨proof⟩

lemma *card-funcsetE*: $\text{finite } A \implies \text{card } (A \rightarrow_E B) = \text{card } B \wedge \text{card } A$
 ⟨proof⟩

lemma *card-inj-on-subset-funcset*:

assumes *finB*: $\text{finite } B$

and *finC*: $\text{finite } C$

and *AB*: $A \subseteq B$

shows $\text{card } \{f \in B \rightarrow_E C. \text{inj-on } f A\} =$

$\text{card } C \setminus (\text{card } B - \text{card } A) * \text{prod } ((-) (\text{card } C)) \{0 ..< \text{card } A\}$

 ⟨proof⟩

27.8 The pigeonhole principle

An alternative formulation of this is that for a function mapping a finite set A of cardinality m to a finite set B of cardinality n , there exists an element $y \in B$ that is hit at least $\lceil \frac{m}{n} \rceil$ times. However, since we do not have real

numbers or rounding yet, we state it in the following equivalent form:

lemma *pigeonhole-card*:

assumes $f \in A \rightarrow B$ *finite* A *finite* B $B \neq \{\}$

shows $\exists y \in B. \text{card } (f^{-1} \{y\} \cap A) * \text{card } B \geq \text{card } A$

<proof>

27.9 Products of sums

lemma *prod-sum-PiE*:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c :: \text{comm-semiring-1}$

assumes *finite*: *finite* A **and** *finite*: $\bigwedge x. x \in A \implies \text{finite } (B\ x)$

shows $(\prod x \in A. \sum y \in B\ x. f\ x\ y) = (\sum g \in \text{PiE } A\ B. \prod x \in A. f\ x\ (g\ x))$

<proof>

end

28 Partitions and Disjoint Sets

theory *Disjoint-Sets*

imports *FuncSet*

begin

lemma *mono-imp-UN-eq-last*: $\text{mono } A \implies (\bigcup_{i \leq n. A\ i} = A\ n$

<proof>

28.1 Set of Disjoint Sets

abbreviation *disjoint* :: $'a \text{ set set} \Rightarrow \text{bool}$ **where** *disjoint* \equiv *pairwise disjoint*

lemma *disjoint-def*: $\text{disjoint } A \longleftrightarrow (\forall a \in A. \forall b \in A. a \neq b \longrightarrow a \cap b = \{\})$

<proof>

lemma *disjointI*:

$(\bigwedge a\ b. a \in A \implies b \in A \implies a \neq b \implies a \cap b = \{\}) \implies \text{disjoint } A$

<proof>

lemma *disjointD*:

$\text{disjoint } A \implies a \in A \implies b \in A \implies a \neq b \implies a \cap b = \{\}$

<proof>

lemma *disjoint-image*: $\text{inj-on } f\ (\bigcup A) \implies \text{disjoint } A \implies \text{disjoint } ((\cdot) f^{-1} A)$

<proof>

lemma **assumes** *disjoint* $(A \cup B)$

shows *disjoint-unionD1*: *disjoint* A **and** *disjoint-unionD2*: *disjoint* B

<proof>

lemma *disjoint-INT*:

assumes $*$: $\bigwedge i. i \in I \implies \text{disjoint } (F\ i)$

shows *disjoint* $\{\bigcap_{i \in I}. X\ i \mid X. \forall i \in I. X\ i \in F\ i\}$
 $\langle proof \rangle$

lemma *diff-Union-pairwise-disjoint*:
assumes *pairwise disjnt* $\mathcal{A}\ \mathcal{B} \subseteq \mathcal{A}$
shows $\bigcup \mathcal{A} - \bigcup \mathcal{B} = \bigcup (\mathcal{A} - \mathcal{B})$
 $\langle proof \rangle$

lemma *Int-Union-pairwise-disjoint*:
assumes *pairwise disjnt* $(\mathcal{A} \cup \mathcal{B})$
shows $\bigcup \mathcal{A} \cap \bigcup \mathcal{B} = \bigcup (\mathcal{A} \cap \mathcal{B})$
 $\langle proof \rangle$

lemma *psubset-Union-pairwise-disjoint*:
assumes \mathcal{B} : *pairwise disjnt* \mathcal{B} and $\mathcal{A} \subset \mathcal{B} - \{\{\}\}$
shows $\bigcup \mathcal{A} \subset \bigcup \mathcal{B}$
 $\langle proof \rangle$

28.1.1 Family of Disjoint Sets

definition *disjoint-family-on* :: $('i \Rightarrow 'a\ set) \Rightarrow 'i\ set \Rightarrow bool$ **where**
disjoint-family-on $A\ S \longleftrightarrow (\forall m \in S. \forall n \in S. m \neq n \longrightarrow A\ m \cap A\ n = \{\})$

abbreviation *disjoint-family* $A \equiv disjoint-family-on\ A\ UNIV$

lemma *disjoint-family-elem-disjnt*:
assumes *infinite* A *finite* C
and df : *disjoint-family-on* $B\ A$
obtains x **where** $x \in A$ *disjnt* $C\ (B\ x)$
 $\langle proof \rangle$

lemma *disjoint-family-onD*:
disjoint-family-on $A\ I \Longrightarrow i \in I \Longrightarrow j \in I \Longrightarrow i \neq j \Longrightarrow A\ i \cap A\ j = \{\}$
 $\langle proof \rangle$

lemma *disjoint-family-subset*: *disjoint-family* $A \Longrightarrow (\bigwedge x. B\ x \subseteq A\ x) \Longrightarrow disjoint-family\ B$
 $\langle proof \rangle$

lemma *disjoint-family-on-insert*:
 $i \notin I \Longrightarrow disjoint-family-on\ A\ (insert\ i\ I) \longleftrightarrow A\ i \cap (\bigcup_{i \in I}. A\ i) = \{\} \wedge disjoint-family-on\ A\ I$
 $\langle proof \rangle$

lemma *disjoint-family-on-bisimulation*:
assumes *disjoint-family-on* $f\ S$
and $\bigwedge n\ m. n \in S \Longrightarrow m \in S \Longrightarrow n \neq m \Longrightarrow f\ n \cap f\ m = \{\} \Longrightarrow g\ n \cap g\ m = \{\}$
shows *disjoint-family-on* $g\ S$

$\langle \text{proof} \rangle$

lemma *disjoint-family-on-mono*:

$A \subseteq B \implies \text{disjoint-family-on } f B \implies \text{disjoint-family-on } f A$

$\langle \text{proof} \rangle$

lemma *disjoint-family-Suc*:

$(\bigwedge n. A n \subseteq A (\text{Suc } n)) \implies \text{disjoint-family } (\lambda i. A (\text{Suc } i) - A i)$

$\langle \text{proof} \rangle$

lemma *disjoint-family-on-disjoint-image*:

$\text{disjoint-family-on } A I \implies \text{disjoint } (A \text{ ‘ } I)$

$\langle \text{proof} \rangle$

lemma *disjoint-family-on-vimageI*: $\text{disjoint-family-on } F I \implies \text{disjoint-family-on}$

$(\lambda i. f - \text{‘ } F i) I$

$\langle \text{proof} \rangle$

lemma *disjoint-image-disjoint-family-on*:

assumes d : $\text{disjoint } (A \text{ ‘ } I)$ **and** i : $\text{inj-on } A I$

shows $\text{disjoint-family-on } A I$

$\langle \text{proof} \rangle$

lemma *disjoint-family-on-iff-disjoint-image*:

assumes $\bigwedge i. i \in I \implies A i \neq \{\}$

shows $\text{disjoint-family-on } A I \longleftrightarrow \text{disjoint } (A \text{ ‘ } I) \wedge \text{inj-on } A I$

$\langle \text{proof} \rangle$

lemma *card-UN-disjoint'*:

assumes $\text{disjoint-family-on } A I \wedge \bigwedge i. i \in I \implies \text{finite } (A i) \text{ finite } I$

shows $\text{card } (\bigcup_{i \in I. A i}) = (\sum_{i \in I. \text{card } (A i)})$

$\langle \text{proof} \rangle$

lemma *disjoint-UN*:

assumes F : $\bigwedge i. i \in I \implies \text{disjoint } (F i)$ **and** $*$: $\text{disjoint-family-on } (\lambda i. \bigcup (F i))$

I

shows $\text{disjoint } (\bigcup_{i \in I. F i})$

$\langle \text{proof} \rangle$

lemma *distinct-list-bind*:

assumes $\text{distinct } xs \wedge x. x \in \text{set } xs \implies \text{distinct } (f x)$

$\text{disjoint-family-on } (\text{set } \circ f) (\text{set } xs)$

shows $\text{distinct } (\text{List.bind } xs f)$

$\langle \text{proof} \rangle$

lemma *bij-betw-UNION-disjoint*:

assumes disj : $\text{disjoint-family-on } A' I$

assumes bij : $\bigwedge i. i \in I \implies \text{bij-betw } f (A i) (A' i)$

shows $\text{bij-betw } f (\bigcup_{i \in I. A i}) (\bigcup_{i \in I. A' i})$

<proof>

lemma *disjoint-union*: $\text{disjoint } C \implies \text{disjoint } B \implies \bigcup C \cap \bigcup B = \{\} \implies \text{disjoint } (C \cup B)$
<proof>

Sum/product of the union of a finite disjoint family

context *comm-monoid-set*
begin

lemma *UNION-disjoint-family*:
assumes *finite I and $\forall i \in I. \text{finite } (A \ i)$*
and *disjoint-family-on A I*
shows $F \ g \ (\bigcup (A \ ' I)) = F \ (\lambda x. F \ g \ (A \ x)) \ I$
<proof>

lemma *Union-disjoint-sets*:
assumes $\forall A \in C. \text{finite } A$ **and** *disjoint C*
shows $F \ g \ (\bigcup C) = (F \circ F) \ g \ C$
<proof>

end

The union of an infinite disjoint family of non-empty sets is infinite.

lemma *infinite-disjoint-family-imp-infinite-UNION*:
assumes $\neg \text{finite } A \wedge x. x \in A \implies f \ x \neq \{\}$ *disjoint-family-on f A*
shows $\neg \text{finite } (\bigcup (f \ ' A))$
<proof>

28.2 Construct Disjoint Sequences

definition *disjointed* :: $(\text{nat} \Rightarrow 'a \text{ set}) \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$ **where**
 $\text{disjointed } A \ n = A \ n - (\bigcup i \in \{0..<n\}. A \ i)$

lemma *finite-UN-disjointed-eq*: $(\bigcup i \in \{0..<n\}. \text{disjointed } A \ i) = (\bigcup i \in \{0..<n\}. A \ i)$
<proof>

lemma *UN-disjointed-eq*: $(\bigcup i. \text{disjointed } A \ i) = (\bigcup i. A \ i)$
<proof>

lemma *less-disjoint-disjointed*: $m < n \implies \text{disjointed } A \ m \cap \text{disjointed } A \ n = \{\}$
<proof>

lemma *disjoint-family-disjointed*: *disjoint-family (disjointed A)*
<proof>

lemma *disjointed-subset*: $\text{disjointed } A \ n \subseteq A \ n$
<proof>

lemma *disjointed-0[simp]*: *disjointed* A $0 = A$ 0
 ⟨proof⟩

lemma *disjointed-mono*: *mono* $A \implies$ *disjointed* A $(\text{Suc } n) = A$ $(\text{Suc } n) - A$ n
 ⟨proof⟩

28.3 Partitions

Partitions P of a set A . We explicitly disallow empty sets.

definition *partition-on* :: 'a set \Rightarrow 'a set set \Rightarrow bool

where

partition-on A $P \longleftrightarrow \bigcup P = A \wedge \text{disjoint } P \wedge \{\} \notin P$

lemma *partition-onI*:

$\bigcup P = A \implies (\bigwedge p q. p \in P \implies q \in P \implies p \neq q \implies \text{disjnt } p \ q) \implies \{\} \notin P$
 $\implies \text{partition-on } A \ P$
 ⟨proof⟩

lemma *partition-onD1*: *partition-on* A $P \implies A = \bigcup P$
 ⟨proof⟩

lemma *partition-onD2*: *partition-on* A $P \implies \text{disjoint } P$
 ⟨proof⟩

lemma *partition-onD3*: *partition-on* A $P \implies \{\} \notin P$
 ⟨proof⟩

28.4 Constructions of partitions

lemma *partition-on-empty*: *partition-on* $\{\}$ $P \longleftrightarrow P = \{\}$
 ⟨proof⟩

lemma *partition-on-space*: $A \neq \{\} \implies \text{partition-on } A \ \{A\}$
 ⟨proof⟩

lemma *partition-on-singletons*: *partition-on* A $((\lambda x. \{x\}) \text{ ` } A)$
 ⟨proof⟩

lemma *partition-on-transform*:

assumes P : *partition-on* A P

assumes F -UN: $\bigcup (F \text{ ` } P) = F (\bigcup P)$ **and** F -disjnt: $\bigwedge p q. p \in P \implies q \in P$
 $\implies \text{disjnt } p \ q \implies \text{disjnt } (F \ p) \ (F \ q)$

shows *partition-on* $(F \ A)$ $(F \text{ ` } P - \{\{\}\})$

⟨proof⟩

lemma *partition-on-restrict*: *partition-on* A $P \implies \text{partition-on } (B \cap A) ((\cap) B \text{ ` } P - \{\{\}\})$
 ⟨proof⟩

lemma *partition-on-vimage*: $\text{partition-on } A \ P \implies \text{partition-on } (f \text{ -- } A) \ ((- \circ) f \text{ -- } P - \{\{\}\})$
 ⟨proof⟩

lemma *partition-on-inj-image*:
 assumes P : $\text{partition-on } A \ P$ and f : $\text{inj-on } f \ A$
 shows $\text{partition-on } (f \text{ -- } A) \ ((\circ) f \text{ -- } P - \{\{\}\})$
 ⟨proof⟩

lemma *partition-on-insert*:
 assumes $\text{disjnt } p \ (\bigcup P)$
 shows $\text{partition-on } A \ (\text{insert } p \ P) \longleftrightarrow \text{partition-on } (A - p) \ P \wedge p \subseteq A \wedge p \neq \{\}$
 ⟨proof⟩

28.5 Finiteness of partitions

lemma *finitely-many-partition-on*:
 assumes $\text{finite } A$
 shows $\text{finite } \{P. \text{partition-on } A \ P\}$
 ⟨proof⟩

lemma *finite-elements*: $\text{finite } A \implies \text{partition-on } A \ P \implies \text{finite } P$
 ⟨proof⟩

lemma *product-partition*:
 assumes $\text{partition-on } A \ P$ and $\bigwedge p. p \in P \implies \text{finite } p$
 shows $\text{card } A = (\sum p \in P. \text{card } p)$
 ⟨proof⟩

28.6 Equivalence of partitions and equivalence classes

lemma *partition-on-quotient*:
 assumes r : $\text{equiv } A \ r$
 shows $\text{partition-on } A \ (A \ /\ / r)$
 ⟨proof⟩

lemma *equiv-partition-on*:
 assumes P : $\text{partition-on } A \ P$
 shows $\text{equiv } A \ \{(x, y). \exists p \in P. x \in p \wedge y \in p\}$
 ⟨proof⟩

lemma *partition-on-eq-quotient*:
 assumes P : $\text{partition-on } A \ P$
 shows $A \ /\ / \{(x, y). \exists p \in P. x \in p \wedge y \in p\} = P$
 ⟨proof⟩

lemma *partition-on-alt*: $\text{partition-on } A \ P \longleftrightarrow (\exists r. \text{equiv } A \ r \wedge P = A \ /\ / r)$
 ⟨proof⟩

lemma (in *comm-monoid-set*) *partition*:

assumes *finite X partition-on X A*
shows $F\ g\ X = F\ (\lambda B. F\ g\ B)\ A$
 $\langle proof \rangle$

If h is an involution on X with no fixed points in X and $f(h(x)) = -f(x)$ then $\sum_{x \in X} f(x) = 0$.

This is easy to show in a ring with characteristic not equal to 2, since then we can do

$$\sum_{x \in X} f(x) = \sum_{x \in X} f(h(x)) = - \sum_{x \in X} f(x)$$

and therefore $2 \sum_{x \in X} f(x) = 0$.

However, the following proof also works in rings of characteristic 2. The idea is to simply partition X into a disjoint union of doubleton sets of the form $\{x, h(x)\}$.

lemma *sum-involution-eq-0*:

assumes $f\text{-}h: \bigwedge x. x \in X \implies f(h\ x) + f\ x = 0$
assumes $h: \bigwedge x. x \in X \implies h\ x \in X \bigwedge x. x \in X \implies h(h\ x) = x \bigwedge x. x \in X \implies h\ x \neq x$
shows $(\sum_{x \in X}. f\ x) = 0$
 $\langle proof \rangle$

28.7 Refinement of partitions

definition *refines* :: 'a set \Rightarrow 'a set set \Rightarrow 'a set set \Rightarrow bool

where *refines* $A\ P\ Q \equiv$
 $partition\text{-}on\ A\ P \wedge partition\text{-}on\ A\ Q \wedge (\forall X \in P. \exists Y \in Q. X \subseteq Y)$

lemma *refines-refl*: $partition\text{-}on\ A\ P \implies refines\ A\ P\ P$
 $\langle proof \rangle$

lemma *refines-asym1*:

assumes $refines\ A\ P\ Q\ refines\ A\ Q\ P$
shows $P \subseteq Q$
 $\langle proof \rangle$

lemma *refines-asym*: $\llbracket refines\ A\ P\ Q; refines\ A\ Q\ P \rrbracket \implies P=Q$
 $\langle proof \rangle$

lemma *refines-trans*: $\llbracket refines\ A\ P\ Q; refines\ A\ Q\ R \rrbracket \implies refines\ A\ P\ R$
 $\langle proof \rangle$

lemma *refines-obtains-subset*:

assumes $refines\ A\ P\ Q\ q \in Q$
shows $partition\text{-}on\ q\ \{p \in P. p \subseteq q\}$
 $\langle proof \rangle$

28.8 The coarsest common refinement of a set of partitions

definition *common-refinement* :: 'a set set set \Rightarrow 'a set set

where *common-refinement* $\mathcal{P} \equiv (\bigcup f \in (\Pi_E P \in \mathcal{P}. P). \{\bigcap (f \text{ ' } \mathcal{P})\}) - \{\{\}\}$

With non-extensional function space

lemma *common-refinement*: *common-refinement* $\mathcal{P} = (\bigcup f \in (\Pi P \in \mathcal{P}. P). \{\bigcap (f \text{ ' } \mathcal{P})\}) - \{\{\}\}$

(**is** ?lhs = ?rhs)

<proof>

lemma *common-refinement-exists*: $\llbracket X \in \text{common-refinement } \mathcal{P}; P \in \mathcal{P} \rrbracket \Longrightarrow \exists R \in P. X \subseteq R$

<proof>

lemma *Union-common-refinement*: $\bigcup (\text{common-refinement } \mathcal{P}) = (\bigcap P \in \mathcal{P}. \bigcup P)$

<proof>

lemma *partition-on-common-refinement*:

assumes $A: \bigwedge P. P \in \mathcal{P} \Longrightarrow \text{partition-on } A P$ **and** $\mathcal{P} \neq \{\}$

shows *partition-on* $A (\text{common-refinement } \mathcal{P})$

<proof>

lemma *refines-common-refinement*:

assumes $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{partition-on } A P$ $P \in \mathcal{P}$

shows *refines* $A (\text{common-refinement } \mathcal{P}) P$

<proof>

The common refinement is itself refined by any other

lemma *common-refinement-coarsest*:

assumes $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{partition-on } A P$ $R \text{ partition-on } A R$ $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{refines } A R P$ $\mathcal{P} \neq \{\}$

shows *refines* $A R (\text{common-refinement } \mathcal{P})$

<proof>

lemma *finite-common-refinement*:

assumes *finite* \mathcal{P} $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{finite } P$

shows *finite* $(\text{common-refinement } \mathcal{P})$

<proof>

lemma *card-common-refinement*:

assumes *finite* \mathcal{P} $\bigwedge P. P \in \mathcal{P} \Longrightarrow \text{finite } P$

shows *card* $(\text{common-refinement } \mathcal{P}) \leq (\prod P \in \mathcal{P}. \text{card } P)$

<proof>

end

29 Type of finite sets defined as a subtype of sets

theory *FSet*

```
imports Main Countable
begin
```

29.1 Definition of the type

```
typedef 'a fset = {A :: 'a set. finite A} morphisms fset Abs-fset
⟨proof⟩
```

```
setup-lifting type-definition-fset
```

29.2 Basic operations and type class instantiations

```
instantiation fset :: (finite) finite
begin
instance ⟨proof⟩
end
```

```
instantiation fset :: (type) {bounded-lattice-bot, distrib-lattice, minus}
begin
```

```
lift-definition bot-fset :: 'a fset is {} parametric empty-transfer ⟨proof⟩
```

```
lift-definition less-eq-fset :: 'a fset ⇒ 'a fset ⇒ bool is subset-eq parametric
subset-transfer
⟨proof⟩
```

```
definition less-fset :: 'a fset ⇒ 'a fset ⇒ bool where xs < ys ≡ xs ≤ ys ∧ xs ≠
(ys :: 'a fset)
```

```
lemma less-fset-transfer[transfer-rule]:
  includes lifting-syntax
  assumes [transfer-rule]: bi-unique A
  shows ((pcr-fset A) == => (pcr-fset A) == => (=)) (⊂) (<)
  ⟨proof⟩
```

```
lift-definition sup-fset :: 'a fset ⇒ 'a fset ⇒ 'a fset is union parametric union-transfer
⟨proof⟩
```

```
lift-definition inf-fset :: 'a fset ⇒ 'a fset ⇒ 'a fset is inter parametric inter-transfer
⟨proof⟩
```

```
lift-definition minus-fset :: 'a fset ⇒ 'a fset ⇒ 'a fset is minus parametric
Diff-transfer
⟨proof⟩
```

```
instance
  ⟨proof⟩
```

end

abbreviation *fempty* :: 'a fset ($\langle \{\} \rangle$) **where** $\{\} \equiv \text{bot}$

abbreviation *fsubset-eq* :: 'a fset \Rightarrow 'a fset \Rightarrow bool (**infix** $\langle |\subseteq| \rangle$ 50) **where** $xs |\subseteq| ys \equiv xs \leq ys$

abbreviation *fsubset* :: 'a fset \Rightarrow 'a fset \Rightarrow bool (**infix** $\langle |\subset| \rangle$ 50) **where** $xs |\subset| ys \equiv xs < ys$

abbreviation *funion* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset (**infixl** $\langle |\cup| \rangle$ 65) **where** $xs |\cup| ys \equiv \text{sup } xs \text{ } ys$

abbreviation *finter* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset (**infixl** $\langle |\cap| \rangle$ 65) **where** $xs |\cap| ys \equiv \text{inf } xs \text{ } ys$

abbreviation *fminus* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset (**infixl** $\langle |-| \rangle$ 65) **where** $xs |-| ys \equiv \text{minus } xs \text{ } ys$

instantiation *fset* :: (equal) equal

begin

definition *HOL.equal* $A \ B \longleftrightarrow A |\subseteq| B \wedge B |\subseteq| A$

instance $\langle \text{proof} \rangle$

end

instantiation *fset* :: (type) conditionally-complete-lattice

begin

context includes *lifting-syntax*

begin

lemma *right-total-Inf-fset-transfer*:

assumes [*transfer-rule*]: bi-unique A **and** [*transfer-rule*]: right-total A

shows (*rel-set* (*rel-set* A) \implies *rel-set* A)

($\lambda S.$ if finite ($\bigcap S \cap \text{Collect } (\text{Domainp } A)$) then $\bigcap S \cap \text{Collect } (\text{Domainp } A)$ else $\{\}$)

($\lambda S.$ if finite (*Inf* S) then *Inf* S else $\{\}$)

$\langle \text{proof} \rangle$

lemma *Inf-fset-transfer*:

assumes [*transfer-rule*]: bi-unique A **and** [*transfer-rule*]: bi-total A

shows (*rel-set* (*rel-set* A) \implies *rel-set* A) ($\lambda A.$ if finite (*Inf* A) then *Inf* A else $\{\}$)

($\lambda A.$ if finite (*Inf* A) then *Inf* A else $\{\}$)

$\langle \text{proof} \rangle$

lift-definition *Inf-fset* :: 'a fset set \Rightarrow 'a fset **is** $\lambda A.$ if finite (*Inf* A) then *Inf* A else $\{\}$

parametric *right-total-Inf-fset-transfer* *Inf-fset-transfer* $\langle \text{proof} \rangle$

lemma *Sup-fset-transfer*:

assumes [*transfer-rule*]: bi-unique A

shows (*rel-set* (*rel-set* A) \implies *rel-set* A) ($\lambda A.$ if finite (*Sup* A) then *Sup* A else $\{\}$)

$(\lambda A. \text{ if finite } (Sup\ A) \text{ then } Sup\ A \text{ else } \{\}) \langle proof \rangle$

lift-definition *Sup-fset* :: 'a fset set \Rightarrow 'a fset **is** $\lambda A. \text{ if finite } (Sup\ A) \text{ then } Sup\ A \text{ else } \{\}$

parametric *Sup-fset-transfer* $\langle proof \rangle$

lemma *finite-Sup*: $\exists z. \text{ finite } z \wedge (\forall a. a \in X \longrightarrow a \leq z) \Longrightarrow \text{ finite } (Sup\ X)$
 $\langle proof \rangle$

lemma *transfer-bdd-below*[*transfer-rule*]: $(rel\text{-}set\ (pcr\text{-}fset\ (=)) \implies (=))\ bdd\text{-}below$
 $\langle proof \rangle$

end

instance

$\langle proof \rangle$

end

instantiation *fset* :: (*finite*) *complete-lattice*

begin

lift-definition *top-fset* :: 'a fset **is** *UNIV* **parametric** *right-total-UNIV-transfer*
UNIV-transfer
 $\langle proof \rangle$

instance

$\langle proof \rangle$

end

instantiation *fset* :: (*finite*) *complete-boolean-algebra*

begin

lift-definition *uminus-fset* :: 'a fset \Rightarrow 'a fset **is** *uminus*
parametric *right-total-Compl-transfer* *Compl-transfer* $\langle proof \rangle$

instance

$\langle proof \rangle$

end

abbreviation *fUNIV* :: 'a::finite fset **where** *fUNIV* \equiv *top*

abbreviation *fuminus* :: 'a::finite fset \Rightarrow 'a fset $(\lhd | - | \rightarrow [81]\ 80)$ **where** $| - |$ $x \equiv$
uminus x

declare *top-fset.rep-eq*[*simp*]

29.3 Other operations

lift-definition $\text{fininsert} :: 'a \Rightarrow 'a \text{ fset} \Rightarrow 'a \text{ fset}$ **is insert parametric** *Lifting-Set.insert-transfer*
<proof>

syntax

$\text{-fset} :: \text{args} \Rightarrow 'a \text{ fset}$ ($\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix finite set enumeration} \rangle \rangle \{ | - | \} \rangle$)

syntax-consts

$\text{-fset} \Rightarrow \text{fininsert}$

translations

$\{ | x, xs | \} == \text{CONST fininsert } x \{ | xs | \}$
 $\{ | x | \} == \text{CONST fininsert } x \{ | | \}$

abbreviation $\text{fmember} :: 'a \Rightarrow 'a \text{ fset} \Rightarrow \text{bool}$ (**infix** $\langle | \in | \rangle 50$) **where**
 $x | \in | X \equiv x \in \text{fset } X$

abbreviation $\text{not-fmember} :: 'a \Rightarrow 'a \text{ fset} \Rightarrow \text{bool}$ (**infix** $\langle | \notin | \rangle 50$) **where**
 $x | \notin | X \equiv x \notin \text{fset } X$

context

begin

qualified abbreviation $\text{Ball} :: 'a \text{ fset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{Ball } X \equiv \text{Set.Ball } (\text{fset } X)$

alias $\text{fBall} = \text{FSet.Ball}$

qualified abbreviation $\text{Bex} :: 'a \text{ fset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{Bex } X \equiv \text{Set.Bex } (\text{fset } X)$

alias $\text{fBex} = \text{FSet.Bex}$

end

syntax (*input*)

$\text{-fBall} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite !} \rangle \rangle \langle - / | : | - \rangle / - \rangle [0, 0, 10] 10$)

$\text{-fBex} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite ?} \rangle \rangle \langle - / | : | - \rangle / - \rangle [0, 0, 10] 10$)

$\text{-fBex1} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite ?!} \rangle \rangle \langle - / | : | - \rangle / - \rangle [0, 0, 10] 10$)

syntax

$\text{-fBall} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \forall \rangle \rangle \langle - / | \in | - \rangle / - \rangle [0, 0, 10] 10$)

$\text{-fBex} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \exists \rangle \rangle \langle - / | \in | - \rangle / - \rangle [0, 0, 10] 10$)

$\text{-fBnex} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \# \rangle \rangle \langle - / | \in | - \rangle / - \rangle [0, 0, 10] 10$)

$\text{-fBex1} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \# \rangle \rangle \langle - / | \in | - \rangle / - \rangle [0, 0, 10] 10$)

$\exists ! \gg \exists ! (-/| \in |-) ./ -) \gg [0, 0, 10] 10)$

syntax-consts

$-fBall -fBnex \Rightarrow fBall$ **and**
 $-fBex \Rightarrow fBex$ **and**
 $-fBex1 \Rightarrow Ex1$

translations

$\forall x | \in | A. P \Rightarrow CONST FSet.Ball A (\lambda x. P)$
 $\exists x | \in | A. P \Rightarrow CONST FSet.Bex A (\lambda x. P)$
 $\nexists x | \in | A. P \Rightarrow CONST fBall A (\lambda x. \neg P)$
 $\exists ! x | \in | A. P \rightarrow \exists ! x. x | \in | A \wedge P$

$\langle ML \rangle$

syntax

$-setlessfAll :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 notation=\langle binder finite \forall \gg \forall -| \subset |- / -) \gg [0, 0, 10] 10) \rangle$
 $-setlessfEx :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 notation=\langle binder finite \exists \gg \exists -| \subset |- / -) \gg [0, 0, 10] 10) \rangle$
 $-setlefAll :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 notation=\langle binder finite \forall \gg \forall -| \subseteq |- / -) \gg [0, 0, 10] 10) \rangle$
 $-setlefEx :: [idt, 'a, bool] \Rightarrow bool \quad (\langle (\langle indent=3 notation=\langle binder finite \exists \gg \exists -| \subseteq |- / -) \gg [0, 0, 10] 10) \rangle$

syntax-consts

$-setlessfAll -setlefAll \Rightarrow All$ **and**
 $-setlessfEx -setlefEx \Rightarrow Ex$

translations

$\forall A | \subset | B. P \rightarrow \forall A. A | \subset | B \rightarrow P$
 $\exists A | \subset | B. P \rightarrow \exists A. A | \subset | B \wedge P$
 $\forall A | \subseteq | B. P \rightarrow \forall A. A | \subseteq | B \rightarrow P$
 $\exists A | \subseteq | B. P \rightarrow \exists A. A | \subseteq | B \wedge P$

context includes *lifting-syntax*
begin

lemma *fmember-transfer0*[*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique A*
shows ($A \Longrightarrow pcr-fset A \Longrightarrow (=)$) (\in) ($| \in |$)
 $\langle proof \rangle$

lemma *fBall-transfer0*[*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique A*
shows ($pcr-fset A \Longrightarrow (A \Longrightarrow (=)) \Longrightarrow (=)$) (*Ball*) (*fBall*)
 $\langle proof \rangle$

lemma *fBex-transfer0*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*pcr-fset A == => (A == => (=)) == => (=)*) (*Bex*) (*fBex*)
<proof>

lift-definition *ffilter* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a fset* \Rightarrow *'a fset* **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* *<proof>*

lift-definition *fPow* :: *'a fset* \Rightarrow *'a fset fset* **is** *Pow* **parametric** *Pow-transfer*
<proof>

lift-definition *fcard* :: *'a fset* \Rightarrow *nat* **is** *card* **parametric** *card-transfer* *<proof>*

lift-definition *fimage* :: (*'a* \Rightarrow *'b*) \Rightarrow *'a fset* \Rightarrow *'b fset* (**infixr** $\langle | \rangle$ 90) **is** *image*
parametric *image-transfer* *<proof>*

lift-definition *fthe-elem* :: *'a fset* \Rightarrow *'a* **is** *the-elem* *<proof>*

lift-definition *fbind* :: *'a fset* \Rightarrow (*'a* \Rightarrow *'b fset*) \Rightarrow *'b fset* **is** *Set.bind* **parametric**
bind-transfer
<proof>

lift-definition *ffUnion* :: *'a fset fset* \Rightarrow *'a fset* **is** *Union* **parametric** *Union-transfer*
<proof>

lift-definition *ffold* :: (*'a* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow *'b* \Rightarrow *'a fset* \Rightarrow *'b* **is** *Finite-Set.fold* *<proof>*

lift-definition *fset-of-list* :: *'a list* \Rightarrow *'a fset* **is** *set* *<proof>*

lift-definition *sorted-list-of-fset* :: *'a::linorder fset* \Rightarrow *'a list* **is** *sorted-list-of-set*
<proof>

29.4 Transferred lemmas from Set.thy

lemma *fset-eqI*: ($\bigwedge x. (x \in A) = (x \in B)$) $\Longrightarrow A = B$
<proof>

lemma *fset-eq-iff*[*no-atp*]: ($A = B$) = ($\forall x. (x \in A) = (x \in B)$)
<proof>

lemma *fBallI*[*no-atp*]: ($\bigwedge x. x \in A \Longrightarrow P x$) $\Longrightarrow fBall A P$
<proof>

lemma *fbspec*[*no-atp*]: $fBall A P \Longrightarrow x \in A \Longrightarrow P x$
<proof>

lemma *fBallE*[*no-atp*]: $fBall A P \Longrightarrow (P x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
<proof>

lemma $fBexI[no-atp]$: $P\ x \Longrightarrow x\ |\in|\ A \Longrightarrow fBex\ A\ P$
 $\langle proof \rangle$

lemma $rev-fBexI[no-atp]$: $x\ |\in|\ A \Longrightarrow P\ x \Longrightarrow fBex\ A\ P$
 $\langle proof \rangle$

lemma $fBexCI[no-atp]$: $(fBall\ A\ (\lambda x. \neg P\ x) \Longrightarrow P\ a) \Longrightarrow a\ |\in|\ A \Longrightarrow fBex\ A\ P$
 $\langle proof \rangle$

lemma $fBexE[no-atp]$: $fBex\ A\ P \Longrightarrow (\bigwedge x. x\ |\in|\ A \Longrightarrow P\ x \Longrightarrow Q) \Longrightarrow Q$
 $\langle proof \rangle$

lemma $fBall-triv[no-atp]$: $fBall\ A\ (\lambda x. P) = ((\exists x. x\ |\in|\ A) \longrightarrow P)$
 $\langle proof \rangle$

lemma $fBex-triv[no-atp]$: $fBex\ A\ (\lambda x. P) = ((\exists x. x\ |\in|\ A) \wedge P)$
 $\langle proof \rangle$

lemma $fBex-triv-one-point1[no-atp]$: $fBex\ A\ (\lambda x. x = a) = (a\ |\in|\ A)$
 $\langle proof \rangle$

lemma $fBex-triv-one-point2[no-atp]$: $fBex\ A\ ((=)\ a) = (a\ |\in|\ A)$
 $\langle proof \rangle$

lemma $fBex-one-point1[no-atp]$: $fBex\ A\ (\lambda x. x = a \wedge P\ x) = (a\ |\in|\ A \wedge P\ a)$
 $\langle proof \rangle$

lemma $fBex-one-point2[no-atp]$: $fBex\ A\ (\lambda x. a = x \wedge P\ x) = (a\ |\in|\ A \wedge P\ a)$
 $\langle proof \rangle$

lemma $fBall-one-point1[no-atp]$: $fBall\ A\ (\lambda x. x = a \longrightarrow P\ x) = (a\ |\in|\ A \longrightarrow P\ a)$
 $\langle proof \rangle$

lemma $fBall-one-point2[no-atp]$: $fBall\ A\ (\lambda x. a = x \longrightarrow P\ x) = (a\ |\in|\ A \longrightarrow P\ a)$
 $\langle proof \rangle$

lemma $fBall-conj-distrib$: $fBall\ A\ (\lambda x. P\ x \wedge Q\ x) = (fBall\ A\ P \wedge fBall\ A\ Q)$
 $\langle proof \rangle$

lemma $fBex-disj-distrib$: $fBex\ A\ (\lambda x. P\ x \vee Q\ x) = (fBex\ A\ P \vee fBex\ A\ Q)$
 $\langle proof \rangle$

lemma $fBall-cong[fundef-cong]$: $A = B \Longrightarrow (\bigwedge x. x\ |\in|\ B \Longrightarrow P\ x = Q\ x) \Longrightarrow fBall\ A\ P = fBall\ B\ Q$
 $\langle proof \rangle$

lemma $fBex-cong[fundef-cong]$: $A = B \Longrightarrow (\bigwedge x. x\ |\in|\ B \Longrightarrow P\ x = Q\ x) \Longrightarrow fBex\ A\ P = fBex\ B\ Q$

$A \text{ } P = fBex \text{ } B \text{ } Q$
 $\langle proof \rangle$

lemma *fsubsetI*[*intro!*]: $(\bigwedge x. x \in A \implies x \in B) \implies A \subseteq B$
 $\langle proof \rangle$

lemma *fsubsetD*[*elim, intro?*]: $A \subseteq B \implies c \in A \implies c \in B$
 $\langle proof \rangle$

lemma *rev-fsubsetD*[*no-atp, intro?*]: $c \in A \implies A \subseteq B \implies c \in B$
 $\langle proof \rangle$

lemma *fsubsetCE*[*no-atp, elim*]: $A \subseteq B \implies (c \notin A \implies P) \implies (c \in B \implies P) \implies P$
 $\langle proof \rangle$

lemma *fsubset-eq*[*no-atp*]: $(A \subseteq B) = fBall \text{ } A \text{ } (\lambda x. x \in B)$
 $\langle proof \rangle$

lemma *contra-fsubsetD*[*no-atp*]: $A \subseteq B \implies c \notin B \implies c \notin A$
 $\langle proof \rangle$

lemma *fsubset-refl*: $A \subseteq A$
 $\langle proof \rangle$

lemma *fsubset-trans*: $A \subseteq B \implies B \subseteq C \implies A \subseteq C$
 $\langle proof \rangle$

lemma *fset-rev-mp*: $c \in A \implies A \subseteq B \implies c \in B$
 $\langle proof \rangle$

lemma *fset-mp*: $A \subseteq B \implies c \in A \implies c \in B$
 $\langle proof \rangle$

lemma *fsubset-not-fsubset-eq*[*code*]: $(A \subset B) = (A \subseteq B \wedge \neg B \subseteq A)$
 $\langle proof \rangle$

lemma *eq-fmem-trans*: $a = b \implies b \in A \implies a \in A$
 $\langle proof \rangle$

lemma *fsubset-antisym*[*intro!*]: $A \subseteq B \implies B \subseteq A \implies A = B$
 $\langle proof \rangle$

lemma *fequalityD1*: $A = B \implies A \subseteq B$
 $\langle proof \rangle$

lemma *fequalityD2*: $A = B \implies B \subseteq A$
 $\langle proof \rangle$

lemma *fequalityE*: $A = B \implies (A \mid\subseteq\mid B \implies B \mid\subseteq\mid A \implies P) \implies P$
 $\langle proof \rangle$

lemma *fequalityCE[elim]*:
 $A = B \implies (c \mid\in\mid A \implies c \mid\in\mid B \implies P) \implies (c \mid\notin\mid A \implies c \mid\notin\mid B \implies P) \implies P$
 $\langle proof \rangle$

lemma *eqfset-imp-iff*: $A = B \implies (x \mid\in\mid A) = (x \mid\in\mid B)$
 $\langle proof \rangle$

lemma *eqfelem-imp-iff*: $x = y \implies (x \mid\in\mid A) = (y \mid\in\mid A)$
 $\langle proof \rangle$

lemma *fempty-iff[simp]*: $(c \mid\in\mid \{\mid\}) = False$
 $\langle proof \rangle$

lemma *fempty-fsubsetI[iff]*: $\{\mid\} \mid\subseteq\mid x$
 $\langle proof \rangle$

lemma *equalsffemptyI*: $(\bigwedge y. y \mid\in\mid A \implies False) \implies A = \{\mid\}$
 $\langle proof \rangle$

lemma *equalsffemptyD*: $A = \{\mid\} \implies a \mid\notin\mid A$
 $\langle proof \rangle$

lemma *fBall-fempty[simp]*: $fBall \{\mid\} P = True$
 $\langle proof \rangle$

lemma *fBex-fempty[simp]*: $fBex \{\mid\} P = False$
 $\langle proof \rangle$

lemma *fPow-iff[iff]*: $(A \mid\in\mid fPow B) = (A \mid\subseteq\mid B)$
 $\langle proof \rangle$

lemma *fPowI*: $A \mid\subseteq\mid B \implies A \mid\in\mid fPow B$
 $\langle proof \rangle$

lemma *fPowD*: $A \mid\in\mid fPow B \implies A \mid\subseteq\mid B$
 $\langle proof \rangle$

lemma *fPow-bottom*: $\{\mid\} \mid\in\mid fPow B$
 $\langle proof \rangle$

lemma *fPow-top*: $A \mid\in\mid fPow A$
 $\langle proof \rangle$

lemma *fPow-not-fempty*: $fPow A \neq \{\mid\}$
 $\langle proof \rangle$

lemma *finter-iff[simp]*: $(c \mid\in\mid A \mid\cap\mid B) = (c \mid\in\mid A \wedge c \mid\in\mid B)$
 $\langle proof \rangle$

lemma *finterI[intro!]*: $c \mid\in\mid A \implies c \mid\in\mid B \implies c \mid\in\mid A \mid\cap\mid B$
 $\langle proof \rangle$

lemma *finterD1*: $c \mid\in\mid A \mid\cap\mid B \implies c \mid\in\mid A$
 $\langle proof \rangle$

lemma *finterD2*: $c \mid\in\mid A \mid\cap\mid B \implies c \mid\in\mid B$
 $\langle proof \rangle$

lemma *finterE[elim!]*: $c \mid\in\mid A \mid\cap\mid B \implies (c \mid\in\mid A \implies c \mid\in\mid B \implies P) \implies P$
 $\langle proof \rangle$

lemma *funion-iff[simp]*: $(c \mid\in\mid A \mid\cup\mid B) = (c \mid\in\mid A \vee c \mid\in\mid B)$
 $\langle proof \rangle$

lemma *funionI1[elim?]*: $c \mid\in\mid A \implies c \mid\in\mid A \mid\cup\mid B$
 $\langle proof \rangle$

lemma *funionI2[elim?]*: $c \mid\in\mid B \implies c \mid\in\mid A \mid\cup\mid B$
 $\langle proof \rangle$

lemma *funionCI[intro!]*: $(c \not\mid\in\mid B \implies c \mid\in\mid A) \implies c \mid\in\mid A \mid\cup\mid B$
 $\langle proof \rangle$

lemma *funionE[elim!]*: $c \mid\in\mid A \mid\cup\mid B \implies (c \mid\in\mid A \implies P) \implies (c \mid\in\mid B \implies P) \implies P$
 $\langle proof \rangle$

lemma *fminus-iff[simp]*: $(c \mid\in\mid A \mid-\mid B) = (c \mid\in\mid A \wedge c \not\mid\in\mid B)$
 $\langle proof \rangle$

lemma *fminusI[intro!]*: $c \mid\in\mid A \implies c \not\mid\in\mid B \implies c \mid\in\mid A \mid-\mid B$
 $\langle proof \rangle$

lemma *fminusD1*: $c \mid\in\mid A \mid-\mid B \implies c \mid\in\mid A$
 $\langle proof \rangle$

lemma *fminusD2*: $c \mid\in\mid A \mid-\mid B \implies c \not\mid\in\mid B \implies P$
 $\langle proof \rangle$

lemma *fminusE[elim!]*: $c \mid\in\mid A \mid-\mid B \implies (c \mid\in\mid A \implies c \not\mid\in\mid B \implies P) \implies P$
 $\langle proof \rangle$

lemma *finsert-iff[simp]*: $(a \mid\in\mid finsert\ b\ A) = (a = b \vee a \mid\in\mid A)$
 $\langle proof \rangle$

lemma *fininsertI1*: $a \in \mid fininsert\ a\ B$
 $\langle proof \rangle$

lemma *fininsertI2*: $a \in \mid B \implies a \in \mid fininsert\ b\ B$
 $\langle proof \rangle$

lemma *fininsertE[elim!]*: $a \in \mid fininsert\ b\ A \implies (a = b \implies P) \implies (a \in \mid A \implies P) \implies P$
 $\langle proof \rangle$

lemma *fininsertCI[intro!]*: $(a \notin \mid B \implies a = b) \implies a \in \mid fininsert\ b\ B$
 $\langle proof \rangle$

lemma *fsubset-fininsert-iff*:
 $(A \subseteq \mid fininsert\ x\ B) = (if\ x \in \mid A\ then\ A \mid - \mid \{x\} \subseteq \mid B\ else\ A \subseteq \mid B)$
 $\langle proof \rangle$

lemma *fininsert-ident*: $x \notin \mid A \implies x \notin \mid B \implies (fininsert\ x\ A = fininsert\ x\ B) = (A = B)$
 $\langle proof \rangle$

lemma *fsingletonI[intro!,no-atp]*: $a \in \mid \{|a|\}$
 $\langle proof \rangle$

lemma *fsingletonD[dest!,no-atp]*: $b \in \mid \{|a|\} \implies b = a$
 $\langle proof \rangle$

lemma *fsingleton-iff*: $(b \in \mid \{|a|\}) = (b = a)$
 $\langle proof \rangle$

lemma *fsingleton-inject[dest!]*: $\{|a|\} = \{|b|\} \implies a = b$
 $\langle proof \rangle$

lemma *fsingleton-fininsert-inj-eq[iff,no-atp]*: $(\{|b|\} = fininsert\ a\ A) = (a = b \wedge A \subseteq \mid \{|b|\})$
 $\langle proof \rangle$

lemma *fsingleton-fininsert-inj-eq'[iff,no-atp]*: $(fininsert\ a\ A = \{|b|\}) = (a = b \wedge A \subseteq \mid \{|b|\})$
 $\langle proof \rangle$

lemma *fsubset-fsingletonD*: $A \subseteq \mid \{|x|\} \implies A = \{|\}\ \vee\ A = \{|x|\}$
 $\langle proof \rangle$

lemma *fminus-single-fininsert*: $A \mid - \mid \{|x|\} \subseteq \mid B \implies A \subseteq \mid fininsert\ x\ B$
 $\langle proof \rangle$

lemma *fdoubleton-eq-iff*: $(\{|a,\ b|\} = \{|c,\ d|\}) = (a = c \wedge b = d \vee a = d \wedge b = c)$

$\langle proof \rangle$

lemma *funion-fsingleton-iff*:

$$(A \mid \cup \mid B = \{|x|\}) = (A = \{|\}\} \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{|\}\} \vee A = \{|x|\} \wedge B = \{|x|\})$$

$\langle proof \rangle$

lemma *fsingleton-funion-iff*:

$$(\{|x|\} = A \mid \cup \mid B) = (A = \{|\}\} \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{|\}\} \vee A = \{|x|\} \wedge B = \{|x|\})$$

$\langle proof \rangle$

lemma *fimage-eqI[simp, intro]*: $b = f\ x \implies x \mid \in \mid A \implies b \mid \in \mid f \mid \upharpoonright A$

$\langle proof \rangle$

lemma *fimageI*: $x \mid \in \mid A \implies f\ x \mid \in \mid f \mid \upharpoonright A$

$\langle proof \rangle$

lemma *rev-fimage-eqI*: $x \mid \in \mid A \implies b = f\ x \implies b \mid \in \mid f \mid \upharpoonright A$

$\langle proof \rangle$

lemma *fimageE[elim!]*: $b \mid \in \mid f \mid \upharpoonright A \implies (\bigwedge x. b = f\ x \implies x \mid \in \mid A \implies thesis) \implies thesis$

$\langle proof \rangle$

lemma *Compr-fimage-eq*: $\{x. x \mid \in \mid f \mid \upharpoonright A \wedge P\ x\} = f\ ' \{x. x \mid \in \mid A \wedge P\ (f\ x)\}$

$\langle proof \rangle$

lemma *fimage-funion*: $f \mid \upharpoonright (A \mid \cup \mid B) = f \mid \upharpoonright A \mid \cup \mid f \mid \upharpoonright B$

$\langle proof \rangle$

lemma *fimage-iff*: $(z \mid \in \mid f \mid \upharpoonright A) = fBex\ A\ (\lambda x. z = f\ x)$

$\langle proof \rangle$

lemma *fimage-fsubset-iff[no-atp]*: $(f \mid \upharpoonright A \mid \subseteq \mid B) = fBall\ A\ (\lambda x. f\ x \mid \in \mid B)$

$\langle proof \rangle$

lemma *fimage-fsubsetI*: $(\bigwedge x. x \mid \in \mid A \implies f\ x \mid \in \mid B) \implies f \mid \upharpoonright A \mid \subseteq \mid B$

$\langle proof \rangle$

lemma *fimage-ident[simp]*: $(\lambda x. x) \mid \upharpoonright Y = Y$

$\langle proof \rangle$

lemma *if-split-fmem1*: $((if\ Q\ then\ x\ else\ y) \mid \in \mid b) = ((Q \longrightarrow x \mid \in \mid b) \wedge (\neg Q \longrightarrow y \mid \in \mid b))$

$\langle proof \rangle$

lemma *if-split-fmem2*: $(a \mid \in \mid (if\ Q\ then\ x\ else\ y)) = ((Q \longrightarrow a \mid \in \mid x) \wedge (\neg Q \longrightarrow a \mid \in \mid y))$

$\langle proof \rangle$

lemma *pfssubsetI[intro!,no-atp]*: $A \sqsubseteq B \implies A \neq B \implies A \sqsubset B$
 $\langle proof \rangle$

lemma *pfssubsetE[elim!,no-atp]*: $A \sqsubset B \implies (A \sqsubseteq B \implies \neg B \sqsubseteq A \implies R) \implies R$
 $\langle proof \rangle$

lemma *pfssubset-finsert-iff*:
 $(A \sqsubset \text{finsert } x \ B) =$
 $(\text{if } x \in B \text{ then } A \sqsubset B \text{ else if } x \in A \text{ then } A \dashv \{x\} \sqsubset B \text{ else } A \sqsubseteq B)$
 $\langle proof \rangle$

lemma *pfssubset-eq*: $(A \sqsubset B) = (A \sqsubseteq B \wedge A \neq B)$
 $\langle proof \rangle$

lemma *pfssubset-imp-fsubset*: $A \sqsubset B \implies A \sqsubseteq B$
 $\langle proof \rangle$

lemma *pfssubset-trans*: $A \sqsubset B \implies B \sqsubset C \implies A \sqsubset C$
 $\langle proof \rangle$

lemma *pfssubsetD*: $A \sqsubset B \implies c \in A \implies c \in B$
 $\langle proof \rangle$

lemma *pfssubset-fsubset-trans*: $A \sqsubset B \implies B \sqsubseteq C \implies A \sqsubset C$
 $\langle proof \rangle$

lemma *fsubset-pfssubset-trans*: $A \sqsubseteq B \implies B \sqsubset C \implies A \sqsubset C$
 $\langle proof \rangle$

lemma *pfssubset-imp-ex-fmem*: $A \sqsubset B \implies \exists b. b \in B \dashv A$
 $\langle proof \rangle$

lemma *fimage-fPow-mono*: $f \restriction A \sqsubseteq B \implies (f \restriction) f \restriction fPow A \sqsubseteq fPow B$
 $\langle proof \rangle$

lemma *fimage-fPow-surj*: $f \restriction A = B \implies (f \restriction) f \restriction fPow A = fPow B$
 $\langle proof \rangle$

lemma *fsubset-finsertI*: $B \sqsubseteq \text{finsert } a \ B$
 $\langle proof \rangle$

lemma *fsubset-finsertI2*: $A \sqsubseteq B \implies A \sqsubseteq \text{finsert } b \ B$
 $\langle proof \rangle$

lemma *fsubset-finsert*: $x \notin A \implies (A \sqsubseteq \text{finsert } x \ B) = (A \sqsubseteq B)$
 $\langle proof \rangle$

lemma *funion-upper1*: $A \mid\subseteq\mid A \mid\cup\mid B$
 $\langle proof \rangle$

lemma *funion-upper2*: $B \mid\subseteq\mid A \mid\cup\mid B$
 $\langle proof \rangle$

lemma *funion-least*: $A \mid\subseteq\mid C \implies B \mid\subseteq\mid C \implies A \mid\cup\mid B \mid\subseteq\mid C$
 $\langle proof \rangle$

lemma *finter-lower1*: $A \mid\cap\mid B \mid\subseteq\mid A$
 $\langle proof \rangle$

lemma *finter-lower2*: $A \mid\cap\mid B \mid\subseteq\mid B$
 $\langle proof \rangle$

lemma *finter-greatest*: $C \mid\subseteq\mid A \implies C \mid\subseteq\mid B \implies C \mid\subseteq\mid A \mid\cap\mid B$
 $\langle proof \rangle$

lemma *fminus-fsubset*: $A \mid-\mid B \mid\subseteq\mid A$
 $\langle proof \rangle$

lemma *fminus-fsubset-conv*: $(A \mid-\mid B \mid\subseteq\mid C) = (A \mid\subseteq\mid B \mid\cup\mid C)$
 $\langle proof \rangle$

lemma *fsubset-fempty[simp]*: $(A \mid\subseteq\mid \{\mid\}) = (A = \{\mid\})$
 $\langle proof \rangle$

lemma *not-pfsubset-fempty[iff]*: $\neg A \mid\subset\mid \{\mid\}$
 $\langle proof \rangle$

lemma *finsert-is-funion*: $finsert\ a\ A = \{\mid a\mid\} \mid\cup\mid A$
 $\langle proof \rangle$

lemma *finsert-not-fempty[simp]*: $finsert\ a\ A \neq \{\mid\}$
 $\langle proof \rangle$

lemma *fempty-not-finsert*: $\{\mid\} \neq finsert\ a\ A$
 $\langle proof \rangle$

lemma *finsert-absorb*: $a \mid\in\mid A \implies finsert\ a\ A = A$
 $\langle proof \rangle$

lemma *finsert-absorb2[simp]*: $finsert\ x\ (finsert\ x\ A) = finsert\ x\ A$
 $\langle proof \rangle$

lemma *finsert-commute*: $finsert\ x\ (finsert\ y\ A) = finsert\ y\ (finsert\ x\ A)$
 $\langle proof \rangle$

lemma *fininsert-fsubset[simp]*: $(\text{fininsert } x \ A \ \subseteq \ B) = (x \in \ B \wedge A \subseteq \ B)$
 $\langle \text{proof} \rangle$

lemma *fininsert-inter-fininsert[simp]*: $\text{fininsert } a \ A \ \cap \ \text{fininsert } a \ B = \text{fininsert } a \ (A \ \cap \ B)$
 $\langle \text{proof} \rangle$

lemma *fininsert-disjoint[simp,no-atp]*:
 $(\text{fininsert } a \ A \ \cap \ B = \{\}) = (a \notin \ B \wedge A \cap \ B = \{\})$
 $(\{\} = \text{fininsert } a \ A \ \cap \ B) = (a \notin \ B \wedge \{\} = A \cap \ B)$
 $\langle \text{proof} \rangle$

lemma *disjoint-fininsert[simp,no-atp]*:
 $(B \cap \ \text{fininsert } a \ A = \{\}) = (a \notin \ B \wedge B \cap \ A = \{\})$
 $(\{\} = A \cap \ \text{fininsert } b \ B) = (b \notin \ A \wedge \{\} = A \cap \ B)$
 $\langle \text{proof} \rangle$

lemma *fimage-fempty[simp]*: $f \ \mid \! \! \mid \ \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *fimage-fininsert[simp]*: $f \ \mid \! \! \mid \ \text{fininsert } a \ B = \text{fininsert } (f \ a) \ (f \ \mid \! \! \mid \ B)$
 $\langle \text{proof} \rangle$

lemma *fimage-constant*: $x \in \ A \implies (\lambda x. \ c) \ \mid \! \! \mid \ A = \{c\}$
 $\langle \text{proof} \rangle$

lemma *fimage-constant-conv*: $(\lambda x. \ c) \ \mid \! \! \mid \ A = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \{c\})$
 $\langle \text{proof} \rangle$

lemma *fimage-fimage*: $f \ \mid \! \! \mid \ g \ \mid \! \! \mid \ A = (\lambda x. \ f \ (g \ x)) \ \mid \! \! \mid \ A$
 $\langle \text{proof} \rangle$

lemma *fininsert-fimage[simp]*: $x \in \ A \implies \text{fininsert } (f \ x) \ (f \ \mid \! \! \mid \ A) = f \ \mid \! \! \mid \ A$
 $\langle \text{proof} \rangle$

lemma *fimage-is-fempty[iff]*: $(f \ \mid \! \! \mid \ A = \{\}) = (A = \{\})$
 $\langle \text{proof} \rangle$

lemma *fempty-is-fimage[iff]*: $(\{\} = f \ \mid \! \! \mid \ A) = (A = \{\})$
 $\langle \text{proof} \rangle$

lemma *fimage-cong*: $M = N \implies (\bigwedge x. \ x \in \ N \implies f \ x = g \ x) \implies f \ \mid \! \! \mid \ M = g \ \mid \! \! \mid \ N$
 $\langle \text{proof} \rangle$

lemma *fimage-finter-fsubset*: $f \ \mid \! \! \mid \ (A \ \cap \ B) \subseteq f \ \mid \! \! \mid \ A \ \cap \ f \ \mid \! \! \mid \ B$
 $\langle \text{proof} \rangle$

lemma *fimage-fminus-fsubset*: $f \ \mid \! \! \mid \ A \ \mid \! \! \mid \ f \ \mid \! \! \mid \ B \subseteq f \ \mid \! \! \mid \ (A \ \mid \! \! \mid \ B)$
 $\langle \text{proof} \rangle$

lemma *finter-absorb*: $A \mid\cap\mid A = A$
 $\langle proof \rangle$

lemma *finter-left-absorb*: $A \mid\cap\mid (A \mid\cap\mid B) = A \mid\cap\mid B$
 $\langle proof \rangle$

lemma *finter-commute*: $A \mid\cap\mid B = B \mid\cap\mid A$
 $\langle proof \rangle$

lemma *finter-left-commute*: $A \mid\cap\mid (B \mid\cap\mid C) = B \mid\cap\mid (A \mid\cap\mid C)$
 $\langle proof \rangle$

lemma *finter-assoc*: $A \mid\cap\mid B \mid\cap\mid C = A \mid\cap\mid (B \mid\cap\mid C)$
 $\langle proof \rangle$

lemma *finter-ac*:
 $A \mid\cap\mid B \mid\cap\mid C = A \mid\cap\mid (B \mid\cap\mid C)$
 $A \mid\cap\mid (A \mid\cap\mid B) = A \mid\cap\mid B$
 $A \mid\cap\mid B = B \mid\cap\mid A$
 $A \mid\cap\mid (B \mid\cap\mid C) = B \mid\cap\mid (A \mid\cap\mid C)$
 $\langle proof \rangle$

lemma *finter-absorb1*: $B \mid\subseteq\mid A \implies A \mid\cap\mid B = B$
 $\langle proof \rangle$

lemma *finter-absorb2*: $A \mid\subseteq\mid B \implies A \mid\cap\mid B = A$
 $\langle proof \rangle$

lemma *finter-fempty-left*: $\{\mid\mid\} \mid\cap\mid B = \{\mid\mid\}$
 $\langle proof \rangle$

lemma *finter-fempty-right*: $A \mid\cap\mid \{\mid\mid\} = \{\mid\mid\}$
 $\langle proof \rangle$

lemma *disjoint-iff-fnot-equal*: $(A \mid\cap\mid B = \{\mid\mid\}) = fBall\ A\ (\lambda x. fBall\ B\ ((\neq)\ x))$
 $\langle proof \rangle$

lemma *finter-funion-distrib*: $A \mid\cap\mid (B \mid\cup\mid C) = A \mid\cap\mid B \mid\cup\mid (A \mid\cap\mid C)$
 $\langle proof \rangle$

lemma *finter-funion-distrib2*: $B \mid\cup\mid C \mid\cap\mid A = B \mid\cap\mid A \mid\cup\mid (C \mid\cap\mid A)$
 $\langle proof \rangle$

lemma *finter-fsubset-iff*[*no-atp*, *simp*]: $(C \mid\subseteq\mid A \mid\cap\mid B) = (C \mid\subseteq\mid A \wedge C \mid\subseteq\mid B)$
 $\langle proof \rangle$

lemma *funion-absorb*: $A \mid\cup\mid A = A$
 $\langle proof \rangle$

lemma *funion-left-absorb*: $A \mid \cup \mid (A \mid \cup \mid B) = A \mid \cup \mid B$
 $\langle proof \rangle$

lemma *funion-commute*: $A \mid \cup \mid B = B \mid \cup \mid A$
 $\langle proof \rangle$

lemma *funion-left-commute*: $A \mid \cup \mid (B \mid \cup \mid C) = B \mid \cup \mid (A \mid \cup \mid C)$
 $\langle proof \rangle$

lemma *funion-assoc*: $A \mid \cup \mid B \mid \cup \mid C = A \mid \cup \mid (B \mid \cup \mid C)$
 $\langle proof \rangle$

lemma *funion-ac*:
 $A \mid \cup \mid B \mid \cup \mid C = A \mid \cup \mid (B \mid \cup \mid C)$
 $A \mid \cup \mid (A \mid \cup \mid B) = A \mid \cup \mid B$
 $A \mid \cup \mid B = B \mid \cup \mid A$
 $A \mid \cup \mid (B \mid \cup \mid C) = B \mid \cup \mid (A \mid \cup \mid C)$
 $\langle proof \rangle$

lemma *funion-absorb1*: $A \mid \subseteq \mid B \implies A \mid \cup \mid B = B$
 $\langle proof \rangle$

lemma *funion-absorb2*: $B \mid \subseteq \mid A \implies A \mid \cup \mid B = A$
 $\langle proof \rangle$

lemma *funion-fempty-left*: $\{\mid\} \mid \cup \mid B = B$
 $\langle proof \rangle$

lemma *funion-fempty-right*: $A \mid \cup \mid \{\mid\} = A$
 $\langle proof \rangle$

lemma *funion-finsert-left[simp]*: $finsert\ a\ B \mid \cup \mid C = finsert\ a\ (B \mid \cup \mid C)$
 $\langle proof \rangle$

lemma *funion-finsert-right[simp]*: $A \mid \cup \mid finsert\ a\ B = finsert\ a\ (A \mid \cup \mid B)$
 $\langle proof \rangle$

lemma *finter-finsert-left*: $finsert\ a\ B \mid \cap \mid C = (if\ a \mid \in \mid C\ then\ finsert\ a\ (B \mid \cap \mid C)\ else\ B \mid \cap \mid C)$
 $\langle proof \rangle$

lemma *finter-finsert-left-iffempty[simp]*: $a \mid \notin \mid C \implies finsert\ a\ B \mid \cap \mid C = B \mid \cap \mid C$
 $\langle proof \rangle$

lemma *finter-finsert-left-if1[simp]*: $a \mid \in \mid C \implies finsert\ a\ B \mid \cap \mid C = finsert\ a\ (B \mid \cap \mid C)$
 $\langle proof \rangle$

lemma *finter-finsert-right*:

$$A \mid \cap \mid \text{finsert } a \ B = (\text{if } a \mid \in \mid A \text{ then } \text{finsert } a \ (A \mid \cap \mid B) \text{ else } A \mid \cap \mid B)$$

<proof>

lemma *finter-finsert-right-iffempty[simp]*: $a \mid \notin \mid A \implies A \mid \cap \mid \text{finsert } a \ B = A \mid \cap \mid B$

<proof>

lemma *finter-finsert-right-if1[simp]*: $a \mid \in \mid A \implies A \mid \cap \mid \text{finsert } a \ B = \text{finsert } a \ (A \mid \cap \mid B)$

<proof>

lemma *funion-finter-distrib*: $A \mid \cup \mid (B \mid \cap \mid C) = A \mid \cup \mid B \mid \cap \mid (A \mid \cup \mid C)$

<proof>

lemma *funion-finter-distrib2*: $B \mid \cap \mid C \mid \cup \mid A = B \mid \cup \mid A \mid \cap \mid (C \mid \cup \mid A)$

<proof>

lemma *funion-finter-crazy*:

$$A \mid \cap \mid B \mid \cup \mid (B \mid \cap \mid C) \mid \cup \mid (C \mid \cap \mid A) = A \mid \cup \mid B \mid \cap \mid (B \mid \cup \mid C) \mid \cap \mid (C \mid \cup \mid A)$$

<proof>

lemma *fsubset-funion-eq*: $(A \mid \subseteq \mid B) = (A \mid \cup \mid B = B)$

<proof>

lemma *funion-fempty[iff]*: $(A \mid \cup \mid B = \{\mid\}) = (A = \{\mid\} \wedge B = \{\mid\})$

<proof>

lemma *funion-fsubset-iff[no-atp, simp]*: $(A \mid \cup \mid B \mid \subseteq \mid C) = (A \mid \subseteq \mid C \wedge B \mid \subseteq \mid C)$

<proof>

lemma *funion-fminus-finter*: $A \mid - \mid B \mid \cup \mid (A \mid \cap \mid B) = A$

<proof>

lemma *ffunion-empty[simp]*: $\text{ffUnion } \{\mid\} = \{\mid\}$

<proof>

lemma *ffunion-mono*: $A \mid \subseteq \mid B \implies \text{ffUnion } A \mid \subseteq \mid \text{ffUnion } B$

<proof>

lemma *ffunion-insert[simp]*: $\text{ffUnion } (\text{finsert } a \ B) = a \mid \cup \mid \text{ffUnion } B$

<proof>

lemma *fminus-finter2*: $A \mid \cap \mid C \mid - \mid (B \mid \cap \mid C) = A \mid \cap \mid C \mid - \mid B$

<proof>

lemma *funion-finter-assoc-eq*: $(A \mid \cap \mid B \mid \cup \mid C = A \mid \cap \mid (B \mid \cup \mid C)) = (C \mid \subseteq \mid A)$

<proof>

lemma *fBall-funion*: $fBall (A \mid\cup\mid B) P = (fBall A P \wedge fBall B P)$
 $\langle proof \rangle$

lemma *fBex-funion*: $fBex (A \mid\cup\mid B) P = (fBex A P \vee fBex B P)$
 $\langle proof \rangle$

lemma *fminus-eq-fempty-iff[simp,no-atp]*: $(A \mid-\mid B = \{\mid\}) = (A \mid\subseteq\mid B)$
 $\langle proof \rangle$

lemma *fminus-cancel[simp]*: $A \mid-\mid A = \{\mid\}$
 $\langle proof \rangle$

lemma *fminus-idemp[simp]*: $A \mid-\mid B \mid-\mid B = A \mid-\mid B$
 $\langle proof \rangle$

lemma *fminus-triv*: $A \mid\cap\mid B = \{\mid\} \implies A \mid-\mid B = A$
 $\langle proof \rangle$

lemma *fempty-fminus[simp]*: $\{\mid\} \mid-\mid A = \{\mid\}$
 $\langle proof \rangle$

lemma *fminus-fempty[simp]*: $A \mid-\mid \{\mid\} = A$
 $\langle proof \rangle$

lemma *fminus-finsertffempty[simp,no-atp]*: $x \notin A \implies A \mid-\mid finsert x B = A \mid-\mid B$
 $\langle proof \rangle$

lemma *fminus-finsert*: $A \mid-\mid finsert a B = A \mid-\mid B \mid-\mid \{|a|\}$
 $\langle proof \rangle$

lemma *fminus-finsert2*: $A \mid-\mid finsert a B = A \mid-\mid \{|a|\} \mid-\mid B$
 $\langle proof \rangle$

lemma *finsert-fminus-if*: $finsert x A \mid-\mid B = (if x \in B then A \mid-\mid B else finsert x (A \mid-\mid B))$
 $\langle proof \rangle$

lemma *finsert-fminus1[simp]*: $x \in B \implies finsert x A \mid-\mid B = A \mid-\mid B$
 $\langle proof \rangle$

lemma *finsert-fminus-single[simp]*: $finsert a (A \mid-\mid \{|a|\}) = finsert a A$
 $\langle proof \rangle$

lemma *finsert-fminus*: $a \in A \implies finsert a (A \mid-\mid \{|a|\}) = A$
 $\langle proof \rangle$

lemma *fminus-finsert-absorb*: $x \notin A \implies finsert x A \mid-\mid \{|x|\} = A$
 $\langle proof \rangle$

lemma *fminus-disjoint[simp]*: $A \mid \cap \mid (B \mid - \mid A) = \{\mid\}$
 $\langle proof \rangle$

lemma *fminus-partition*: $A \mid \subseteq \mid B \implies A \mid \cup \mid (B \mid - \mid A) = B$
 $\langle proof \rangle$

lemma *double-fminus*: $A \mid \subseteq \mid B \implies B \mid \subseteq \mid C \implies B \mid - \mid (C \mid - \mid A) = A$
 $\langle proof \rangle$

lemma *funion-fminus-cancel[simp]*: $A \mid \cup \mid (B \mid - \mid A) = A \mid \cup \mid B$
 $\langle proof \rangle$

lemma *funion-fminus-cancel2[simp]*: $B \mid - \mid A \mid \cup \mid A = B \mid \cup \mid A$
 $\langle proof \rangle$

lemma *fminus-funion*: $A \mid - \mid (B \mid \cup \mid C) = A \mid - \mid B \mid \cap \mid (A \mid - \mid C)$
 $\langle proof \rangle$

lemma *fminus-finter*: $A \mid - \mid (B \mid \cap \mid C) = A \mid - \mid B \mid \cup \mid (A \mid - \mid C)$
 $\langle proof \rangle$

lemma *funion-fminus*: $A \mid \cup \mid B \mid - \mid C = A \mid - \mid C \mid \cup \mid (B \mid - \mid C)$
 $\langle proof \rangle$

lemma *finter-fminus*: $A \mid \cap \mid B \mid - \mid C = A \mid \cap \mid (B \mid - \mid C)$
 $\langle proof \rangle$

lemma *fminus-finter-distrib*: $C \mid \cap \mid (A \mid - \mid B) = C \mid \cap \mid A \mid - \mid (C \mid \cap \mid B)$
 $\langle proof \rangle$

lemma *fminus-finter-distrib2*: $A \mid - \mid B \mid \cap \mid C = A \mid \cap \mid C \mid - \mid (B \mid \cap \mid C)$
 $\langle proof \rangle$

lemma *fUNIV-bool[no-atp]*: $fUNIV = \{\mid False, \mid True\}$
 $\langle proof \rangle$

lemma *fPow-fempty[simp]*: $fPow \{\mid\} = \{\{\mid\}\}$
 $\langle proof \rangle$

lemma *fPow-finsert*: $fPow (finsert a A) = fPow A \mid \cup \mid finsert a \mid \mid fPow A$
 $\langle proof \rangle$

lemma *funion-fPow-fsubset*: $fPow A \mid \cup \mid fPow B \mid \subseteq \mid fPow (A \mid \cup \mid B)$
 $\langle proof \rangle$

lemma *fPow-finter-eq[simp]*: $fPow (A \mid \cap \mid B) = fPow A \mid \cap \mid fPow B$
 $\langle proof \rangle$

lemma *fset-eq-fsubset*: $(A = B) = (A \sqsubseteq B \wedge B \sqsubseteq A)$
 $\langle \text{proof} \rangle$

lemma *fsubset-iff[no-atp]*: $(A \sqsubseteq B) = (\forall t. t \in A \longrightarrow t \in B)$
 $\langle \text{proof} \rangle$

lemma *fsubset-iff-pfssubset-eq*: $(A \sqsubseteq B) = (A \sqsubset B \vee A = B)$
 $\langle \text{proof} \rangle$

lemma *all-not-fin-conv[simp]*: $(\forall x. x \notin A) = (A = \{\})$
 $\langle \text{proof} \rangle$

lemma *ex-fin-conv*: $(\exists x. x \in A) = (A \neq \{\})$
 $\langle \text{proof} \rangle$

lemma *fimage-mono*: $A \sqsubseteq B \Longrightarrow f \mid^{\cdot} A \sqsubseteq f \mid^{\cdot} B$
 $\langle \text{proof} \rangle$

lemma *fPow-mono*: $A \sqsubseteq B \Longrightarrow fPow A \sqsubseteq fPow B$
 $\langle \text{proof} \rangle$

lemma *finsert-mono*: $C \sqsubseteq D \Longrightarrow finsert a C \sqsubseteq finsert a D$
 $\langle \text{proof} \rangle$

lemma *funion-mono*: $A \sqsubseteq C \Longrightarrow B \sqsubseteq D \Longrightarrow A \sqcup B \sqsubseteq C \sqcup D$
 $\langle \text{proof} \rangle$

lemma *finter-mono*: $A \sqsubseteq C \Longrightarrow B \sqsubseteq D \Longrightarrow A \sqcap B \sqsubseteq C \sqcap D$
 $\langle \text{proof} \rangle$

lemma *fminus-mono*: $A \sqsubseteq C \Longrightarrow D \sqsubseteq B \Longrightarrow A \sqcap B \sqsubseteq C \sqcap D$
 $\langle \text{proof} \rangle$

lemma *fin-mono*: $A \sqsubseteq B \Longrightarrow x \in A \longrightarrow x \in B$
 $\langle \text{proof} \rangle$

lemma *fthe-felem-eq[simp]*: $fthe\text{-}elem \{x\} = x$
 $\langle \text{proof} \rangle$

lemma *fLeast-mono*:
 $mono f \Longrightarrow fBex S (\lambda x. fBall S ((\leq) x)) \Longrightarrow (LEAST y. y \in f \mid^{\cdot} S) = f$
 $(LEAST x. x \in S)$
 $\langle \text{proof} \rangle$

lemma *fbind-fbind*: $fbind (fbind A B) C = fbind A (\lambda x. fbind (B x) C)$
 $\langle \text{proof} \rangle$

lemma *fempty-fbind[simp]*: $fbind \{\} f = \{\}$
 $\langle \text{proof} \rangle$

lemma *nonempty-fbind-const*: $A \neq \{\mid\} \implies \text{fbind } A (\lambda-. B) = B$
 $\langle \text{proof} \rangle$

lemma *fbind-const*: $\text{fbind } A (\lambda-. B) = (\text{if } A = \{\mid\} \text{ then } \{\mid\} \text{ else } B)$
 $\langle \text{proof} \rangle$

lemma *ffmember-filter[simp]*: $(x \mid\in \text{ffilter } P A) = (x \mid\in A \wedge P x)$
 $\langle \text{proof} \rangle$

lemma *fequalityI*: $A \mid\subseteq B \implies B \mid\subseteq A \implies A = B$
 $\langle \text{proof} \rangle$

lemma *fset-of-list-simps[simp]*:
 $\text{fset-of-list } [] = \{\mid\}$
 $\text{fset-of-list } (x21 \# x22) = \text{fininsert } x21 (\text{fset-of-list } x22)$
 $\langle \text{proof} \rangle$

lemma *fset-of-list-append[simp]*: $\text{fset-of-list } (xs @ ys) = \text{fset-of-list } xs \mid\cup \text{fset-of-list } ys$
 $\langle \text{proof} \rangle$

lemma *fset-of-list-rev[simp]*: $\text{fset-of-list } (\text{rev } xs) = \text{fset-of-list } xs$
 $\langle \text{proof} \rangle$

lemma *fset-of-list-map[simp]*: $\text{fset-of-list } (\text{map } f xs) = f \mid' \text{fset-of-list } xs$
 $\langle \text{proof} \rangle$

29.5 Additional lemmas

29.5.1 *ffUnion*

lemma *ffUnion-union-distrib[simp]*: $\text{ffUnion } (A \mid\cup B) = \text{ffUnion } A \mid\cup \text{ffUnion } B$
 $\langle \text{proof} \rangle$

29.5.2 *fbind*

lemma *fbind-cong[fundef-cong]*: $A = B \implies (\bigwedge x. x \mid\in B \implies f x = g x) \implies \text{fbind } A f = \text{fbind } B g$
 $\langle \text{proof} \rangle$

29.5.3 *fsingleton*

lemma *fsingletonE*: $b \mid\in \{ \mid a \} \implies (b = a \implies \text{thesis}) \implies \text{thesis}$
 $\langle \text{proof} \rangle$

29.5.4 *fempty*

lemma *fempty-ffilter[simp]*: $\text{ffilter } (\lambda-. \text{False}) A = \{\mid\}$

$\langle proof \rangle$

lemma *femptyE* [*elim!*]: $a \in \{\} \implies P$
 $\langle proof \rangle$

29.5.5 *fset*

lemma *fset-simps*[*simp*]:
 $fset \{\} = \{\}$
 $fset (insert\ x\ X) = insert\ x\ (fset\ X)$
 $\langle proof \rangle$

lemma *finite-fset* [*simp*]:
shows *finite* (*fset* *S*)
 $\langle proof \rangle$

lemmas *fset-cong* = *fset-inject*

lemma *filter-fset* [*simp*]:
shows $fset\ (ffilter\ P\ xs) = Collect\ P \cap fset\ xs$
 $\langle proof \rangle$

lemma *inter-fset*[*simp*]: $fset\ (A \cap B) = fset\ A \cap fset\ B$
 $\langle proof \rangle$

lemma *union-fset*[*simp*]: $fset\ (A \cup B) = fset\ A \cup fset\ B$
 $\langle proof \rangle$

lemma *minus-fset*[*simp*]: $fset\ (A - B) = fset\ A - fset\ B$
 $\langle proof \rangle$

29.5.6 *ffilter*

lemma *subset-ffilter*:
 $ffilter\ P\ A \subseteq ffilter\ Q\ A = (\forall\ x. x \in A \implies P\ x \implies Q\ x)$
 $\langle proof \rangle$

lemma *eq-ffilter*:
 $(ffilter\ P\ A = ffilter\ Q\ A) = (\forall\ x. x \in A \implies P\ x = Q\ x)$
 $\langle proof \rangle$

lemma *pfssubset-ffilter*:
 $(\bigwedge x. x \in A \implies P\ x \implies Q\ x) \implies (x \in A \wedge \neg P\ x \wedge Q\ x) \implies$
 $ffilter\ P\ A \subset ffilter\ Q\ A$
 $\langle proof \rangle$

29.5.7 *fset-of-list*

lemma *fset-of-list-filter*[*simp*]:

$fset\text{-of-list } (filter\ P\ xs) = ffilter\ P\ (fset\text{-of-list}\ xs)$
 $\langle proof \rangle$

lemma $fset\text{-of-list-subset}[intro]$:
 $set\ xs \subseteq set\ ys \implies fset\text{-of-list}\ xs \mid\subseteq\ fset\text{-of-list}\ ys$
 $\langle proof \rangle$

lemma $fset\text{-of-list-elem}$: $(x \mid\in\ fset\text{-of-list}\ xs) \longleftrightarrow (x \in set\ xs)$
 $\langle proof \rangle$

29.5.8 $fininsert$

lemma $set\text{-}fininsert$:
assumes $x \mid\in\ A$
obtains B **where** $A = fininsert\ x\ B$ **and** $x \nmid\in\ B$
 $\langle proof \rangle$

lemma $mk\text{-disjoint}\text{-}fininsert$: $a \mid\in\ A \implies \exists B. A = fininsert\ a\ B \wedge a \nmid\in\ B$
 $\langle proof \rangle$

lemma $fininsert\text{-eq}\text{-iff}$:
assumes $a \nmid\in\ A$ **and** $b \nmid\in\ B$
shows $(fininsert\ a\ A = fininsert\ b\ B) =$
 $(if\ a = b\ then\ A = B\ else\ \exists C. A = fininsert\ b\ C \wedge b \nmid\in\ C \wedge B = fininsert\ a\ C \wedge$
 $a \nmid\in\ C)$
 $\langle proof \rangle$

29.5.9 $fimage$

lemma $subset\text{-}fimage\text{-iff}$: $(B \mid\subseteq\ f \mid^! A) = (\exists AA. AA \mid\subseteq\ A \wedge B = f \mid^! AA)$
 $\langle proof \rangle$

lemma $fimage\text{-strict}\text{-mono}$:
assumes $inj\text{-on}\ f\ (fset\ B)$ **and** $A \mid\subset\ B$
shows $f \mid^! A \mid\subset\ f \mid^! B$
 — TODO: Configure transfer framework to lift $\llbracket inj\text{-on}\ ?f\ ?B; ?A \subset ?B \rrbracket \implies ?f$
 $\langle proof \rangle$

29.5.10 bounded quantification

lemma $bex\text{-simps}$ $[simp, no\text{-}atp]$:
 $\bigwedge A\ P\ Q. fBex\ A\ (\lambda x. P\ x \wedge Q) = (fBex\ A\ P \wedge Q)$
 $\bigwedge A\ P\ Q. fBex\ A\ (\lambda x. P \wedge Q\ x) = (P \wedge fBex\ A\ Q)$
 $\bigwedge P. fBex\ \{\mid\}\ P = False$
 $\bigwedge a\ B\ P. fBex\ (fininsert\ a\ B)\ P = (P\ a \vee fBex\ B\ P)$
 $\bigwedge A\ P\ f. fBex\ (f \mid^! A)\ P = fBex\ A\ (\lambda x. P\ (f\ x))$
 $\bigwedge A\ P. (\neg fBex\ A\ P) = fBall\ A\ (\lambda x. \neg P\ x)$
 $\langle proof \rangle$

lemma *ball-simps* [*simp*, *no-atp*]:

$$\begin{aligned}
&\bigwedge A P Q. fBall A (\lambda x. P x \vee Q) = (fBall A P \vee Q) \\
&\bigwedge A P Q. fBall A (\lambda x. P \vee Q x) = (P \vee fBall A Q) \\
&\bigwedge A P Q. fBall A (\lambda x. P \longrightarrow Q x) = (P \longrightarrow fBall A Q) \\
&\bigwedge A P Q. fBall A (\lambda x. P x \longrightarrow Q) = (fBex A P \longrightarrow Q) \\
&\bigwedge P. fBall \{\|\} P = True \\
&\bigwedge a B P. fBall (fininsert a B) P = (P a \wedge fBall B P) \\
&\bigwedge A P f. fBall (f \mid^{\cdot} A) P = fBall A (\lambda x. P (f x)) \\
&\bigwedge A P. (\neg fBall A P) = fBex A (\lambda x. \neg P x)
\end{aligned}$$

<proof>

lemma *atomize-fBall*:

$$(\bigwedge x. x \mid^{\cdot} A \implies P x) == Trueprop (fBall A (\lambda x. P x))$$

<proof>

lemma *fBall-mono*[*mono*]: $P \leq Q \implies fBall S P \leq fBall S Q$

<proof>

lemma *fBex-mono*[*mono*]: $P \leq Q \implies fBex S P \leq fBex S Q$

<proof>

end

29.5.11 *fcard*

lemma *fcard-fempty*:

$$fcard \{\|\} = 0$$

<proof>

lemma *fcard-fininsert-disjoint*:

$$x \notin A \implies fcard (fininsert x A) = Suc (fcard A)$$

<proof>

lemma *fcard-fininsert-if*:

$$fcard (fininsert x A) = (if x \mid^{\cdot} A then fcard A else Suc (fcard A))$$

<proof>

lemma *fcard-0-eq* [*simp*, *no-atp*]:

$$fcard A = 0 \longleftrightarrow A = \{\|\}$$

<proof>

lemma *fcard-Suc-fminus1*:

$$x \mid^{\cdot} A \implies Suc (fcard (A \mid^{\cdot} \{|x|\})) = fcard A$$

<proof>

lemma *fcard-fminus-fsingleton*:

$$x \mid^{\cdot} A \implies fcard (A \mid^{\cdot} \{|x|\}) = fcard A - 1$$

<proof>

lemma *fcard-fminus-fsingleton-if*:

$fcard (A \mid - \mid \{|x|\}) = (if\ x \mid \in \mid A\ then\ fcard\ A - 1\ else\ fcard\ A)$
 $\langle proof \rangle$

lemma *fcard-fminus-finsert[simp]*:

assumes $a \mid \in \mid A$ **and** $a \mid \notin \mid B$
shows $fcard (A \mid - \mid finsert\ a\ B) = fcard (A \mid - \mid B) - 1$
 $\langle proof \rangle$

lemma *fcard-finsert*: $fcard (finsert\ x\ A) = Suc\ (fcard (A \mid - \mid \{|x|\}))$
 $\langle proof \rangle$

lemma *fcard-finsert-le*: $fcard\ A \leq fcard (finsert\ x\ A)$
 $\langle proof \rangle$

lemma *fcard-mono*:

$A \mid \subseteq \mid B \implies fcard\ A \leq fcard\ B$
 $\langle proof \rangle$

lemma *fcard-seteq*: $A \mid \subseteq \mid B \implies fcard\ B \leq fcard\ A \implies A = B$
 $\langle proof \rangle$

lemma *pfssubset-fcard-mono*: $A \mid \subset \mid B \implies fcard\ A < fcard\ B$
 $\langle proof \rangle$

lemma *fcard-union-finter*:

$fcard\ A + fcard\ B = fcard (A \mid \cup \mid B) + fcard (A \mid \cap \mid B)$
 $\langle proof \rangle$

lemma *fcard-union-disjoint*:

$A \mid \cap \mid B = \{|\}\implies fcard (A \mid \cup \mid B) = fcard\ A + fcard\ B$
 $\langle proof \rangle$

lemma *fcard-union-fsubset*:

$B \mid \subseteq \mid A \implies fcard (A \mid - \mid B) = fcard\ A - fcard\ B$
 $\langle proof \rangle$

lemma *diff-fcard-le-fcard-fminus*:

$fcard\ A - fcard\ B \leq fcard (A \mid - \mid B)$
 $\langle proof \rangle$

lemma *fcard-fminus1-less*: $x \mid \in \mid A \implies fcard (A \mid - \mid \{|x|\}) < fcard\ A$
 $\langle proof \rangle$

lemma *fcard-fminus2-less*:

$x \mid \in \mid A \implies y \mid \in \mid A \implies fcard (A \mid - \mid \{|x|\} \mid - \mid \{|y|\}) < fcard\ A$
 $\langle proof \rangle$

lemma *fcard-fminus1-le*: $fcard (A \mid - \mid \{|x|\}) \leq fcard\ A$

$\langle proof \rangle$

lemma *fcard-pfssubset*: $A \mid \subseteq \mid B \implies \text{fcard } A < \text{fcard } B \implies A < B$
 $\langle proof \rangle$

29.5.12 *sorted-list-of-fset*

lemma *sorted-list-of-fset-simps*[*simp*]:
 $\text{set } (\text{sorted-list-of-fset } S) = \text{fset } S$
 $\text{fset-of-list } (\text{sorted-list-of-fset } S) = S$
 $\langle proof \rangle$

29.5.13 *ffold*

context *comp-fun-commute*

begin

lemma *ffold-empty*[*simp*]: $\text{ffold } f \ z \ \{\mid\} = z$
 $\langle proof \rangle$

lemma *ffold-finsert* [*simp*]:
assumes $x \mid \notin \mid A$
shows $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ A)$
 $\langle proof \rangle$

lemma *ffold-fun-left-comm*:
 $f \ x \ (\text{ffold } f \ z \ A) = \text{ffold } f \ (f \ x \ z) \ A$
 $\langle proof \rangle$

lemma *ffold-finsert2*:
 $x \mid \notin \mid A \implies \text{ffold } f \ z \ (\text{finsert } x \ A) = \text{ffold } f \ (f \ x \ z) \ A$
 $\langle proof \rangle$

lemma *ffold-rec*:
assumes $x \mid \in \mid A$
shows $\text{ffold } f \ z \ A = f \ x \ (\text{ffold } f \ z \ (A \mid - \mid \{|x|\}))$
 $\langle proof \rangle$

lemma *ffold-finsert-fremove*:
 $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ (A \mid - \mid \{|x|\}))$
 $\langle proof \rangle$

end

lemma *ffold-fimage*:
assumes *inj-on* $g \ (\text{fset } A)$
shows $\text{ffold } f \ z \ (g \mid ^\flat \mid A) = \text{ffold } (f \circ g) \ z \ A$
 $\langle proof \rangle$

lemma *ffold-cong*:
assumes *comp-fun-commute* f *comp-fun-commute* g
 $\bigwedge x. x \mid \in \mid A \implies f \ x = g \ x$

and $s = t$ **and** $A = B$
shows $\text{ffold } f \ s \ A = \text{ffold } g \ t \ B$
 $\langle \text{proof} \rangle$

context *comp-fun-idem*
begin

lemma *ffold-finsert-idem*:
 $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ A)$
 $\langle \text{proof} \rangle$

declare *ffold-finsert* [*simp del*] *ffold-finsert-idem* [*simp*]

lemma *ffold-finsert-idem2*:
 $\text{ffold } f \ z \ (\text{finsert } x \ A) = \text{ffold } f \ (f \ x \ z) \ A$
 $\langle \text{proof} \rangle$

end

29.5.14 $(|\subset|)$

lemma *wfP-pfsubset*: $\text{wfP } (|\subset|)$
 $\langle \text{proof} \rangle$

29.5.15 Group operations

locale *comm-monoid-fset* = *comm-monoid*
begin

sublocale *set*: *comm-monoid-set* $\langle \text{proof} \rangle$

lift-definition $F :: ('b \Rightarrow 'a) \Rightarrow 'b \text{ fset} \Rightarrow 'a \text{ is set}.F$ $\langle \text{proof} \rangle$

lemma *cong[fundef-cong]*: $A = B \Longrightarrow (\bigwedge x. x \in B \Longrightarrow g \ x = h \ x) \Longrightarrow F \ g \ A = F \ h \ B$
 $\langle \text{proof} \rangle$

lemma *cong-simp[cong]*:
 $\llbracket A = B; \bigwedge x. x \in B =_{\text{simp}} \Rightarrow g \ x = h \ x \rrbracket \Longrightarrow F \ g \ A = F \ h \ B$
 $\langle \text{proof} \rangle$

end

context *comm-monoid-add* **begin**

sublocale *fsum*: *comm-monoid-fset plus 0*
rewrites *comm-monoid-set.F plus 0 = sum*
defines *fsum* = *fsum.F*
 $\langle \text{proof} \rangle$

end

29.5.16 Semilattice operations

locale *semilattice-fset* = *semilattice*
begin

sublocale *set*: *semilattice-set* \langle *proof* \rangle

lift-definition $F :: 'a \text{ fset} \Rightarrow 'a \text{ is set}.F$ \langle *proof* \rangle

lemma *eq-fold*: $F (\text{finsert } x \ A) = \text{ffold } f \ x \ A$
 \langle *proof* \rangle

lemma *singleton* [*simp*]: $F \{|x|\} = x$
 \langle *proof* \rangle

lemma *insert-not-elem*: $x \notin A \Longrightarrow A \neq \{|\}\Longrightarrow F (\text{finsert } x \ A) = x * F \ A$
 \langle *proof* \rangle

lemma *in-idem*: $x \in A \Longrightarrow x * F \ A = F \ A$
 \langle *proof* \rangle

lemma *insert* [*simp*]: $A \neq \{|\}\Longrightarrow F (\text{finsert } x \ A) = x * F \ A$
 \langle *proof* \rangle

end

locale *semilattice-order-fset* = *binary?*: *semilattice-order* + *semilattice-fset*
begin

end

context *linorder* **begin**

sublocale *fMin*: *semilattice-order-fset* *min* *less-eq* *less*
rewrites *semilattice-set.F* *min* = *Min*
defines *fMin* = *fMin.F*
 \langle *proof* \rangle

sublocale *fMax*: *semilattice-order-fset* *max* *greater-eq* *greater*
rewrites *semilattice-set.F* *max* = *Max*
defines *fMax* = *fMax.F*
 \langle *proof* \rangle

end

lemma *mono-fMax-commute*: $\text{mono } f \Longrightarrow A \neq \{|\}\Longrightarrow f (fMax \ A) = fMax \ (f \mid \cdot)$

A)
 $\langle proof \rangle$

lemma *mono-fMin-commute*: $mono\ f \implies A \neq \{\mid\} \implies f\ (fMin\ A) = fMin\ (f\ |\ ^\dagger A)$
 $\langle proof \rangle$

lemma *fMax-in[simp]*: $A \neq \{\mid\} \implies fMax\ A\ |\in\ A$
 $\langle proof \rangle$

lemma *fMin-in[simp]*: $A \neq \{\mid\} \implies fMin\ A\ |\in\ A$
 $\langle proof \rangle$

lemma *fMax-ge[simp]*: $x\ |\in\ A \implies x \leq fMax\ A$
 $\langle proof \rangle$

lemma *fMin-le[simp]*: $x\ |\in\ A \implies fMin\ A \leq x$
 $\langle proof \rangle$

lemma *fMax-eqI*: $(\bigwedge y. y\ |\in\ A \implies y \leq x) \implies x\ |\in\ A \implies fMax\ A = x$
 $\langle proof \rangle$

lemma *fMin-eqI*: $(\bigwedge y. y\ |\in\ A \implies x \leq y) \implies x\ |\in\ A \implies fMin\ A = x$
 $\langle proof \rangle$

lemma *fMax-finsert[simp]*: $fMax\ (finsert\ x\ A) = (if\ A = \{\mid\}\ then\ x\ else\ max\ x\ (fMax\ A))$
 $\langle proof \rangle$

lemma *fMin-finsert[simp]*: $fMin\ (finsert\ x\ A) = (if\ A = \{\mid\}\ then\ x\ else\ min\ x\ (fMin\ A))$
 $\langle proof \rangle$

context *linorder* **begin**

lemma *fset-linorder-max-induct[case-names fempty finsert]*:
assumes $P\ \{\mid\}$
and $\bigwedge x\ S. \llbracket \forall y. y\ |\in\ S \longrightarrow y < x; P\ S \rrbracket \implies P\ (finsert\ x\ S)$
shows $P\ S$
 $\langle proof \rangle$

lemma *fset-linorder-min-induct[case-names fempty finsert]*:
assumes $P\ \{\mid\}$
and $\bigwedge x\ S. \llbracket \forall y. y\ |\in\ S \longrightarrow y > x; P\ S \rrbracket \implies P\ (finsert\ x\ S)$
shows $P\ S$
 $\langle proof \rangle$

end

29.6 Choice in fsets

lemma *fset-choice*:

assumes $\forall x. x \in A \longrightarrow (\exists y. P\ x\ y)$
shows $\exists f. \forall x. x \in A \longrightarrow P\ x\ (f\ x)$
 $\langle proof \rangle$

29.7 Induction and Cases rules for fsets

lemma *fset-exhaust* [*case-names empty insert, cases type: fset*]:

assumes *fempty-case*: $S = \{\} \implies P$
and *finsert-case*: $\bigwedge x\ S'. S = \text{finsert}\ x\ S' \implies P$
shows P
 $\langle proof \rangle$

lemma *fset-induct* [*case-names empty insert*]:

assumes *fempty-case*: $P\ \{\}$
and *finsert-case*: $\bigwedge x\ S. P\ S \implies P\ (\text{finsert}\ x\ S)$
shows $P\ S$
 $\langle proof \rangle$

lemma *fset-induct-stronger* [*case-names empty insert, induct type: fset*]:

assumes *empty-fset-case*: $P\ \{\}$
and *insert-fset-case*: $\bigwedge x\ S. \llbracket x \notin S; P\ S \rrbracket \implies P\ (\text{finsert}\ x\ S)$
shows $P\ S$
 $\langle proof \rangle$

lemma *fset-card-induct*:

assumes *empty-fset-case*: $P\ \{\}$
and *card-fset-Suc-case*: $\bigwedge S\ T. \text{Suc}\ (\text{fcard}\ S) = (\text{fcard}\ T) \implies P\ S \implies P\ T$
shows $P\ S$
 $\langle proof \rangle$

lemma *fset-strong-cases*:

obtains $xs = \{\}$
 $\mid ys\ x\ \textbf{where}\ x \notin ys\ \textbf{and}\ xs = \text{finsert}\ x\ ys$
 $\langle proof \rangle$

lemma *fset-induct2*:

$P\ \{\}\ \{\} \implies$
 $(\bigwedge x\ xs. x \notin xs \implies P\ (\text{finsert}\ x\ xs)\ \{\}) \implies$
 $(\bigwedge y\ ys. y \notin ys \implies P\ \{\}\ (\text{finsert}\ y\ ys)) \implies$
 $(\bigwedge x\ xs\ y\ ys. \llbracket P\ xs\ ys; x \notin xs; y \notin ys \rrbracket \implies P\ (\text{finsert}\ x\ xs)\ (\text{finsert}\ y\ ys)) \implies$
 $P\ xs\ a\ ys\ a$
 $\langle proof \rangle$

29.8 Lemmas depending on induction

lemma *ffUnion-fsubset-iff*: $\text{ffUnion}\ A \subseteq B \longleftrightarrow \text{fBall}\ A\ (\lambda x. x \subseteq B)$

$\langle proof \rangle$

29.9 Setup for Lifting/Transfer

29.9.1 Relator and predicator properties

lift-definition $rel\text{-}fset :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ fset \Rightarrow 'b\ fset \Rightarrow bool$ **is** $rel\text{-}set$
parametric $rel\text{-}set\text{-}transfer$ $\langle proof \rangle$

lemma $rel\text{-}fset\text{-}alt\text{-}def$: $rel\text{-}fset\ R = (\lambda A\ B. (\forall x. \exists y. x \in A \longrightarrow y \in B \wedge R\ x\ y) \wedge (\forall y. \exists x. y \in B \longrightarrow x \in A \wedge R\ x\ y))$
 $\langle proof \rangle$

lemma $finite\text{-}rel\text{-}set$:

assumes fin : $finite\ X\ finite\ Z$

assumes $R\text{-}S$: $rel\text{-}set\ (R\ OO\ S)\ X\ Z$

shows $\exists Y. finite\ Y \wedge rel\text{-}set\ R\ X\ Y \wedge rel\text{-}set\ S\ Y\ Z$

$\langle proof \rangle$

29.9.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes $lifting\text{-}syntax$

begin

lemma $fempty\text{-}transfer$ $[transfer\text{-}rule]$:

$rel\text{-}fset\ A\ \{\{\}\}\ \{\{\}\}$

$\langle proof \rangle$

lemma $finsert\text{-}transfer$ $[transfer\text{-}rule]$:

$(A\ ==>\ rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ A)\ finsert\ finsert$

$\langle proof \rangle$

lemma $funion\text{-}transfer$ $[transfer\text{-}rule]$:

$(rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ A)\ funion\ funion$

$\langle proof \rangle$

lemma $ffUnion\text{-}transfer$ $[transfer\text{-}rule]$:

$(rel\text{-}fset\ (rel\text{-}fset\ A)\ ==>\ rel\text{-}fset\ A)\ ffUnion\ ffUnion$

$\langle proof \rangle$

lemma $fimage\text{-}transfer$ $[transfer\text{-}rule]$:

$((A\ ==>\ B)\ ==>\ rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ B)\ fimage\ fimage$

$\langle proof \rangle$

lemma $fBall\text{-}transfer$ $[transfer\text{-}rule]$:

$(rel\text{-}fset\ A\ ==>\ (A\ ==>\ (=))\ ==>\ (=))\ fBall\ fBall$

$\langle proof \rangle$

lemma $fBex\text{-}transfer$ $[transfer\text{-}rule]$:

$(rel\text{-}fset\ A\ ==>\ (A\ ==>\ (=))\ ==>\ (=))\ fBex\ fBex$

$\langle proof \rangle$

lemma *fPow-transfer* [*transfer-rule*]:
 $(\text{rel-fset } A \implies \text{rel-fset } (\text{rel-fset } A)) \text{ fPow fPow}$
 ⟨*proof*⟩

lemma *rel-fset-transfer* [*transfer-rule*]:
 $((A \implies B \implies (=)) \implies \text{rel-fset } A \implies \text{rel-fset } B \implies (=))$
 rel-fset rel-fset
 ⟨*proof*⟩

lemma *bind-transfer* [*transfer-rule*]:
 $(\text{rel-fset } A \implies (A \implies \text{rel-fset } B) \implies \text{rel-fset } B) \text{ fbind fbind}$
 ⟨*proof*⟩

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(A \implies \text{rel-fset } A \implies (=)) (|\in|) (|\in|)$
 ⟨*proof*⟩

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(\text{rel-fset } A \implies \text{rel-fset } A \implies \text{rel-fset } A) \text{ finter finter}$
 ⟨*proof*⟩

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(\text{rel-fset } A \implies \text{rel-fset } A \implies \text{rel-fset } A) (|-|) (|-|)$
 ⟨*proof*⟩

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows $(\text{rel-fset } A \implies \text{rel-fset } A \implies (=)) (|\subseteq|) (|\subseteq|)$
 ⟨*proof*⟩

lemma *fSup-transfer* [*transfer-rule*]:
 $\text{bi-unique } A \implies (\text{rel-set } (\text{rel-fset } A) \implies \text{rel-fset } A) \text{ Sup Sup}$
 ⟨*proof*⟩

lemma *fInf-transfer* [*transfer-rule*]:
assumes *bi-unique* *A* **and** *bi-total* *A*
shows $(\text{rel-set } (\text{rel-fset } A) \implies \text{rel-fset } A) \text{ Inf Inf}$
 ⟨*proof*⟩

lemma *ffilter-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*

shows $((A ==> (=)) ==> \text{rel-fset } A ==> \text{rel-fset } A) \text{ ffilter ffilter}$
 $\langle \text{proof} \rangle$

lemma *card-transfer* [*transfer-rule*]:
 $\text{bi-unique } A \implies (\text{rel-fset } A ==> (=)) \text{ fcard fcard}$
 $\langle \text{proof} \rangle$

end

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

29.10 BNF setup

context
includes *fset.lifting*
begin

lemma *rel-fset-alt*:
 $\text{rel-fset } R \ a \ b \longleftrightarrow (\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u) \wedge (\forall t \in \text{fset } b. \exists u \in \text{fset } a. R \ u \ t)$
 $\langle \text{proof} \rangle$

lemma *fset-to-fset*: $\text{finite } A \implies \text{fset } (\text{the-inv fset } A) = A$
 $\langle \text{proof} \rangle$

lemma *rel-fset-aux*:
 $(\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u) \wedge (\forall u \in \text{fset } b. \exists t \in \text{fset } a. R \ t \ u) \longleftrightarrow$
 $((\text{BNF-Def.Grp } \{a. \text{fset } a \subseteq \{(a, b). R \ a \ b\}\} (\text{fimage fst}))^{-1-1} \text{ OO}$
 $\text{BNF-Def.Grp } \{a. \text{fset } a \subseteq \{(a, b). R \ a \ b\}\} (\text{fimage snd})) \ a \ b \ (\text{is } ?L = ?R)$
 $\langle \text{proof} \rangle$

bnf 'a *fset*
 map: fimage
 sets: fset
 bd: natLeq
 $\text{wits: \{\|\}}$
 rel: rel-fset
 $\langle \text{proof} \rangle$

lemma *rel-fset-fset*: $\text{rel-set } \chi \ (\text{fset } A1) \ (\text{fset } A2) = \text{rel-fset } \chi \ A1 \ A2$
 $\langle \text{proof} \rangle$

end

declare
 $\text{fset.map-comp}[\text{simp}]$
 $\text{fset.map-id}[\text{simp}]$
 $\text{fset.set-map}[\text{simp}]$

29.11 Size setup

context includes *fset.lifting*

begin

lift-definition *size-fset* :: (*'a* \Rightarrow *nat*) \Rightarrow *'a fset* \Rightarrow *nat* **is** $\lambda f. \text{sum } (Suc \circ f) \langle \text{proof} \rangle$

end

instantiation *fset* :: (*type*) *size*

begin

definition *size-fset* **where**

size-fset-overloaded-def: *size-fset* = *FSet.size-fset* ($\lambda \cdot. 0$)

instance $\langle \text{proof} \rangle$

end

lemma *size-fset-simps*[*simp*]: *size-fset* *f* *X* = $(\sum x \in \text{fset } X. \text{Suc } (f \ x))$
 $\langle \text{proof} \rangle$

lemma *size-fset-overloaded-simps*[*simp*]: *size* *X* = $(\sum x \in \text{fset } X. \text{Suc } 0)$
 $\langle \text{proof} \rangle$

lemma *fset-size-o-map*: *inj* *f* $\implies \text{size-fset } g \circ \text{fimage } f = \text{size-fset } (g \circ f)$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lifting-update *fset.lifting*

lifting-forget *fset.lifting*

29.12 Advanced relator customization

Set vs. sum relators:

lemma *rel-set-rel-sum*[*simp*]:

rel-set (*rel-sum* χ φ) *A1* *A2* \longleftrightarrow

rel-set χ (*Inl* $- ' A1$) (*Inl* $- ' A2$) \wedge *rel-set* φ (*Inr* $- ' A1$) (*Inr* $- ' A2$)

(**is** $?L \longleftrightarrow ?Rl \wedge ?Rr$)

$\langle \text{proof} \rangle$

29.12.1 Countability

lemma *exists-fset-of-list*: $\exists xs. \text{fset-of-list } xs = S$

including *fset.lifting*

$\langle \text{proof} \rangle$

lemma *fset-of-list-surj*[*simp*, *intro*]: *surj* *fset-of-list*

$\langle \text{proof} \rangle$

instance *fset* :: (*countable*) *countable*

$\langle \text{proof} \rangle$

29.13 Quickcheck setup

Setup adapted from sets.

notation *Quickcheck-Exhaustive.orelse* (**infixr** $\langle \text{orelse} \rangle$ 55)

context

includes *term-syntax*

begin

definition [*code-unfold*]:

valterm-femptyset = *Code-Evaluation.valtermify* ($\{\|\}$) :: ($'a :: \text{typerep}$) *fset*)

definition [*code-unfold*]:

valtermify-finsert $x\ s$ = *Code-Evaluation.valtermify* *finsert* $\{\cdot\}$ ($x :: ('a :: \text{typerep} * -))\ \{\cdot\}\ s$

end

instantiation *fset* :: (*exhaustive*) *exhaustive*

begin

fun *exhaustive-fset* **where**

exhaustive-fset $f\ i$ = (if $i = 0$ then *None* else ($f\ \{\|\}$ *orelse* *exhaustive-fset* ($\lambda A. f\ A$ *orelse* *Quickcheck-Exhaustive.exhaustive* ($\lambda x. \text{if } x \in A \text{ then } \text{None} \text{ else } f\ (\text{finsert } x\ A))\ (i - 1))\ (i - 1))$))

instance $\langle \text{proof} \rangle$

end

instantiation *fset* :: (*full-exhaustive*) *full-exhaustive*

begin

fun *full-exhaustive-fset* **where**

full-exhaustive-fset $f\ i$ = (if $i = 0$ then *None* else ($f\ \text{valterm-femptyset}$ *orelse* *full-exhaustive-fset* ($\lambda A. f\ A$ *orelse* *Quickcheck-Exhaustive.full-exhaustive* ($\lambda x. \text{if } x \in A \text{ then } \text{None} \text{ else } f\ (\text{valtermify-finsert } x\ A))\ (i - 1))\ (i - 1))$))

instance $\langle \text{proof} \rangle$

end

no-notation *Quickcheck-Exhaustive.orelse* (**infixr** $\langle \text{orelse} \rangle$ 55)

instantiation *fset* :: (*random*) *random*

begin

context

includes *state-combinator-syntax*

begin

```
fun random-aux-fset :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural  $\times$  natural  $\Rightarrow$  ('a fset  $\times$  (unit
 $\Rightarrow$  term))  $\times$  natural  $\times$  natural where
random-aux-fset 0 j = Quickcheck-Random.collapse (Random.select-weight [(1, Pair
valterm-femptyset)]) |
random-aux-fset (Code-Numeral.Suc i) j =
  Quickcheck-Random.collapse (Random.select-weight
    [(1, Pair valterm-femptyset),
     (Code-Numeral.Suc i,
      Quickcheck-Random.random j  $\circ \rightarrow$  ( $\lambda x$ . random-aux-fset i j  $\circ \rightarrow$  ( $\lambda s$ . Pair
        (valtermify-finset x s)))))])
```

lemma [code]:

```
random-aux-fset i j =
  Quickcheck-Random.collapse (Random.select-weight [(1, Pair valterm-femptyset),
    (i, Quickcheck-Random.random j  $\circ \rightarrow$  ( $\lambda x$ . random-aux-fset (i - 1) j  $\circ \rightarrow$  ( $\lambda s$ .
    Pair (valtermify-finset x s)))))]
<proof>
```

definition random-fset i = random-aux-fset i i

instance <proof>

end

end

29.14 Code Generation Setup

The following *code-unfold* lemmas are so the pre-processor of the code generator will perform conversions like, e.g., $(x \mid \in \mid f \mid \mid \mid \text{fset-of-list } xs) = (x \in f \text{ 'set } xs)$.

declare

```
ffilter.rep-eq[code-unfold]
fimage.rep-eq[code-unfold]
finset.rep-eq[code-unfold]
fset-of-list.rep-eq[code-unfold]
inf-fset.rep-eq[code-unfold]
minus-fset.rep-eq[code-unfold]
sup-fset.rep-eq[code-unfold]
uminus-fset.rep-eq[code-unfold]
```

end

30 Type of finite maps defined as a subtype of maps

```

theory Finite-Map
  imports FSet AList Conditional-Parametricity
  abbrevs (= =  $\subseteq_f$ )
begin

```

30.1 Auxiliary constants and lemmas over *map*

```

parametric-constant map-add-transfer[transfer-rule]: map-add-def
parametric-constant map-of-transfer[transfer-rule]: map-of-def

```

```

context includes lifting-syntax begin

```

```

abbreviation rel-map :: ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'c)  $\Rightarrow$  bool where
rel-map f  $\equiv$  (=)  $\implies$  rel-option f

```

```

lemma ran-transfer[transfer-rule]: (rel-map A  $\implies$  rel-set A) ran ran
<proof>

```

```

lemma ran-alt-def: ran m = (the  $\circ$  m) ' dom m
<proof>

```

```

parametric-constant dom-transfer[transfer-rule]: dom-def

```

```

definition map-upd :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) where
map-upd k v m = m(k  $\mapsto$  v)

```

```

parametric-constant map-upd-transfer[transfer-rule]: map-upd-def

```

```

definition map-filter :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) where
map-filter P m = ( $\lambda x$ . if P x then m x else None)

```

```

parametric-constant map-filter-transfer[transfer-rule]: map-filter-def

```

```

lemma map-filter-map-of[simp]: map-filter P (map-of m) = map-of [(k, -)  $\leftarrow$  m.
P k]
<proof>

```

```

lemma map-filter-finite[intro]:
  assumes finite (dom m)
  shows finite (dom (map-filter P m))
<proof>

```

```

definition map-drop :: 'a  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) where
map-drop a = map-filter ( $\lambda a'$ . a'  $\neq$  a)

```

```

parametric-constant map-drop-transfer[transfer-rule]: map-drop-def

```

definition *map-drop-set* :: *'a set* \Rightarrow (*'a* \rightarrow *'b*) \Rightarrow (*'a* \rightarrow *'b*) **where**
map-drop-set *A* = *map-filter* ($\lambda a. a \notin A$)

parametric-constant *map-drop-set-transfer*[*transfer-rule*]: *map-drop-set-def*

definition *map-restrict-set* :: *'a set* \Rightarrow (*'a* \rightarrow *'b*) \Rightarrow (*'a* \rightarrow *'b*) **where**
map-restrict-set *A* = *map-filter* ($\lambda a. a \in A$)

parametric-constant *map-restrict-set-transfer*[*transfer-rule*]: *map-restrict-set-def*

definition *map-pred* :: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*'a* \rightarrow *'b*) \Rightarrow *bool* **where**
map-pred *P m* \longleftrightarrow ($\forall x. \text{case } m \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } y \Rightarrow P \ x \ y$)

parametric-constant *map-pred-transfer*[*transfer-rule*]: *map-pred-def*

definition *rel-map-on-set* :: *'a set* \Rightarrow (*'b* \Rightarrow *'c* \Rightarrow *bool*) \Rightarrow (*'a* \rightarrow *'b*) \Rightarrow (*'a* \rightarrow *'c*)
 \Rightarrow *bool* **where**
rel-map-on-set *S P* = *eq-onp* ($\lambda x. x \in S$) $==>$ *rel-option P*

definition *set-of-map* :: (*'a* \rightarrow *'b*) \Rightarrow (*'a* \times *'b*) *set* **where**
set-of-map *m* = $\{(k, v) \mid k \ v. \ m \ k = \text{Some } v\}$

lemma *set-of-map-alt-def*: *set-of-map* *m* = ($\lambda k. (k, \text{the } (m \ k))$) ‘ *dom m*
 $\langle \text{proof} \rangle$

lemma *set-of-map-finite*: *finite* (*dom m*) \implies *finite* (*set-of-map m*)
 $\langle \text{proof} \rangle$

lemma *set-of-map-inj*: *inj set-of-map*
 $\langle \text{proof} \rangle$

lemma *dom-comp*: *dom* (*m* \circ_m *n*) \subseteq *dom n*
 $\langle \text{proof} \rangle$

lemma *dom-comp-finite*: *finite* (*dom n*) \implies *finite* (*dom* (*map-comp m n*))
 $\langle \text{proof} \rangle$

parametric-constant *map-comp-transfer*[*transfer-rule*]: *map-comp-def*

end

30.2 Abstract characterisation

typedef (*'a*, *'b*) *fmap* = {*m. finite* (*dom m*)} :: (*'a* \rightarrow *'b*) *set*
morphisms *fmlookup* *Abs-fmap*
 $\langle \text{proof} \rangle$

setup-lifting *type-definition-fmap*

lemma *dom-fmlookup-finite*[*intro, simp*]: *finite (dom (fmlookup m))*
 ⟨*proof*⟩

lemma *fmap-ext*:
 assumes $\bigwedge x. \text{fmlookup } m \ x = \text{fmlookup } n \ x$
 shows $m = n$
 ⟨*proof*⟩

30.3 Operations

context
 includes *fset.lifting*
begin

lift-definition *fmran* :: (*'a*, *'b*) *fmap* \Rightarrow *'b fset*
 is *ran*
parametric *ran-transfer*
 ⟨*proof*⟩

lemma *fmlookup-ran-iff*: $y \in | \text{fmran } m \longleftrightarrow (\exists x. \text{fmlookup } m \ x = \text{Some } y)$
 ⟨*proof*⟩

lemma *fmranI*: $\text{fmlookup } m \ x = \text{Some } y \Longrightarrow y \in | \text{fmran } m$ ⟨*proof*⟩

lemma *fmranE*[*elim*]:
 assumes $y \in | \text{fmran } m$
 obtains x **where** $\text{fmlookup } m \ x = \text{Some } y$
 ⟨*proof*⟩

lift-definition *fmdom* :: (*'a*, *'b*) *fmap* \Rightarrow *'a fset*
 is *dom*
parametric *dom-transfer*
 ⟨*proof*⟩

lemma *fmlookup-dom-iff*: $x \in | \text{fmdom } m \longleftrightarrow (\exists a. \text{fmlookup } m \ x = \text{Some } a)$
 ⟨*proof*⟩

lemma *fmdom-notI*: $\text{fmlookup } m \ x = \text{None} \Longrightarrow x \notin | \text{fmdom } m$ ⟨*proof*⟩

lemma *fmdomI*: $\text{fmlookup } m \ x = \text{Some } y \Longrightarrow x \in | \text{fmdom } m$ ⟨*proof*⟩

lemma *fmdom-notD*[*dest*]: $x \notin | \text{fmdom } m \Longrightarrow \text{fmlookup } m \ x = \text{None}$ ⟨*proof*⟩

lemma *fmdomE*[*elim*]:
 assumes $x \in | \text{fmdom } m$
 obtains y **where** $\text{fmlookup } m \ x = \text{Some } y$
 ⟨*proof*⟩

lift-definition *fmdom'* :: (*'a*, *'b*) *fmap* \Rightarrow *'a set*
 is *dom*

parametric dom-transfer
 $\langle \text{proof} \rangle$

lemma *fmlookup-dom'-iff*: $x \in \text{fmdom}' m \longleftrightarrow (\exists a. \text{fmlookup } m \ x = \text{Some } a)$
 $\langle \text{proof} \rangle$

lemma *fmdom'-notI*: $\text{fmlookup } m \ x = \text{None} \implies x \notin \text{fmdom}' m$ $\langle \text{proof} \rangle$

lemma *fmdom'I*: $\text{fmlookup } m \ x = \text{Some } y \implies x \in \text{fmdom}' m$ $\langle \text{proof} \rangle$

lemma *fmdom'-notD[dest]*: $x \notin \text{fmdom}' m \implies \text{fmlookup } m \ x = \text{None}$ $\langle \text{proof} \rangle$

lemma *fmdom'E[elim]*:
 assumes $x \in \text{fmdom}' m$
 obtains $x \ y$ where $\text{fmlookup } m \ x = \text{Some } y$
 $\langle \text{proof} \rangle$

lemma *fmdom'-alt-def*: $\text{fmdom}' m = \text{fset } (\text{fmdom } m)$
 $\langle \text{proof} \rangle$

lemma *finite-fmdom'[simp]*: $\text{finite } (\text{fmdom}' m)$
 $\langle \text{proof} \rangle$

lemma *dom-fmlookup[simp]*: $\text{dom } (\text{fmlookup } m) = \text{fmdom}' m$
 $\langle \text{proof} \rangle$

lift-definition *fmempty* :: $('a, 'b) \text{ fmap}$
 is *Map.empty*
 $\langle \text{proof} \rangle$

lemma *fmempty-lookup[simp]*: $\text{fmlookup } \text{fmempty } x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *fmdom-empty[simp]*: $\text{fmdom } \text{fmempty} = \{\mid\}$ $\langle \text{proof} \rangle$

lemma *fmdom'-empty[simp]*: $\text{fmdom}' \text{fmempty} = \{\}$ $\langle \text{proof} \rangle$

lemma *fmran-empty[simp]*: $\text{fmran } \text{fmempty} = \text{fempty}$ $\langle \text{proof} \rangle$

lift-definition *fmupd* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow ('a, 'b) \text{ fmap}$
 is *map-upd*
parametric map-upd-transfer
 $\langle \text{proof} \rangle$

lemma *fmupd-lookup[simp]*: $\text{fmlookup } (\text{fmupd } a \ b \ m) \ a' = (\text{if } a = a' \text{ then } \text{Some } b \text{ else } \text{fmlookup } m \ a')$
 $\langle \text{proof} \rangle$

lemma *fmdom-fmupd[simp]*: $\text{fmdom } (\text{fmupd } a \ b \ m) = \text{finsert } a \ (\text{fmdom } m)$ $\langle \text{proof} \rangle$

lemma *fmdom'-fmupd[simp]*: $\text{fmdom}' (\text{fmupd } a \ b \ m) = \text{insert } a \ (\text{fmdom}' m)$ $\langle \text{proof} \rangle$

lemma *fmupd-reorder-neq*:
 assumes $a \neq b$

shows $fmupd\ a\ x\ (fmupd\ b\ y\ m) = fmupd\ b\ y\ (fmupd\ a\ x\ m)$
 $\langle proof \rangle$

lemma $fmupd-idem[simp]$: $fmupd\ a\ x\ (fmupd\ a\ y\ m) = fmupd\ a\ x\ m$
 $\langle proof \rangle$

lift-definition $fmfilter :: ('a \Rightarrow bool) \Rightarrow ('a, 'b)\ fmap \Rightarrow ('a, 'b)\ fmap$
is $map-filter$
parametric $map-filter-transfer$
 $\langle proof \rangle$

lemma $fmdom-filter[simp]$: $fmdom\ (fmfilter\ P\ m) = ffilter\ P\ (fmdom\ m)$
 $\langle proof \rangle$

lemma $fmdom'-filter[simp]$: $fmdom'\ (fmfilter\ P\ m) = Set.filter\ P\ (fmdom'\ m)$
 $\langle proof \rangle$

lemma $fmlookup-filter[simp]$: $fmlookup\ (fmfilter\ P\ m)\ x = (if\ P\ x\ then\ fmlookup\ m\ x\ else\ None)$
 $\langle proof \rangle$

lemma $fmfilter-empty[simp]$: $fmfilter\ P\ fmempty = fmempty$
 $\langle proof \rangle$

lemma $fmfilter-true[simp]$:
assumes $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \implies P\ x$
shows $fmfilter\ P\ m = m$
 $\langle proof \rangle$

lemma $fmfilter-false[simp]$:
assumes $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \implies \neg P\ x$
shows $fmfilter\ P\ m = fmempty$
 $\langle proof \rangle$

lemma $fmfilter-comp[simp]$: $fmfilter\ P\ (fmfilter\ Q\ m) = fmfilter\ (\lambda x. P\ x \wedge Q\ x)\ m$
 $\langle proof \rangle$

lemma $fmfilter-comm$: $fmfilter\ P\ (fmfilter\ Q\ m) = fmfilter\ Q\ (fmfilter\ P\ m)$
 $\langle proof \rangle$

lemma $fmfilter-cong[cong]$:
assumes $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \implies P\ x = Q\ x$
shows $fmfilter\ P\ m = fmfilter\ Q\ m$
 $\langle proof \rangle$

lemma $fmfilter-cong'[fundef-cong]$:
assumes $m = n \wedge x. x \in fmdom'\ m \implies P\ x = Q\ x$
shows $fmfilter\ P\ m = fmfilter\ Q\ n$

⟨proof⟩

lemma *fmfilter-upd[simp]*:

fmfilter *P* (*fmupd* *x y m*) = (if *P x* then *fmupd* *x y* (*fmfilter* *P m*) else *fmfilter* *P m*)

⟨proof⟩

lift-definition *fmdrop* :: 'a ⇒ ('a, 'b) *fmap* ⇒ ('a, 'b) *fmap*

is *map-drop*

parametric *map-drop-transfer*

⟨proof⟩

lemma *fmdrop-lookup[simp]*: *fmlookup* (*fmdrop* *a m*) *a* = *None*

⟨proof⟩

lift-definition *fmdrop-set* :: 'a *set* ⇒ ('a, 'b) *fmap* ⇒ ('a, 'b) *fmap*

is *map-drop-set*

parametric *map-drop-set-transfer*

⟨proof⟩

lift-definition *fmdrop-fset* :: 'a *fset* ⇒ ('a, 'b) *fmap* ⇒ ('a, 'b) *fmap*

is *map-drop-set*

parametric *map-drop-set-transfer*

⟨proof⟩

lift-definition *fmrestrict-set* :: 'a *set* ⇒ ('a, 'b) *fmap* ⇒ ('a, 'b) *fmap*

is *map-restrict-set*

parametric *map-restrict-set-transfer*

⟨proof⟩

lift-definition *fmrestrict-fset* :: 'a *fset* ⇒ ('a, 'b) *fmap* ⇒ ('a, 'b) *fmap*

is *map-restrict-set*

parametric *map-restrict-set-transfer*

⟨proof⟩

lemma *fmfilter-alt-defs*:

fmdrop *a* = *fmfilter* (λ*a'*. *a' ≠ a*)

fmdrop-set *A* = *fmfilter* (λ*a*. *a ∉ A*)

fmdrop-fset *B* = *fmfilter* (λ*a*. *a ∉ B*)

fmrestrict-set *A* = *fmfilter* (λ*a*. *a ∈ A*)

fmrestrict-fset *B* = *fmfilter* (λ*a*. *a ∈ B*)

⟨proof⟩

lemma *fmdom-drop[simp]*: *fmdom* (*fmdrop* *a m*) = *fmdom* *m* − {*a*} ⟨proof⟩

lemma *fmdom'-drop[simp]*: *fmdom'* (*fmdrop* *a m*) = *fmdom'* *m* − {*a*} ⟨proof⟩

lemma *fmdom'-drop-set[simp]*: *fmdom'* (*fmdrop-set* *A m*) = *fmdom'* *m* − *A* ⟨proof⟩

lemma *fmdom-drop-fset[simp]*: *fmdom* (*fmdrop-fset* *A m*) = *fmdom* *m* − *A* ⟨proof⟩

lemma *fmdom'-restrict-set*: *fmdom'* (*fmrestrict-set* *A m*) ⊆ *A* ⟨proof⟩

lemma *fmdom-restrict-fset*: *fmdom* (*fmrestrict-fset* *A m*) |⊆| *A* ⟨proof⟩

lemma *fmdrop-fmupd*: $fmdrop\ x\ (fmupd\ y\ z\ m) = (if\ x = y\ then\ fmdrop\ x\ m\ else\ fmupd\ y\ z\ (fmdrop\ x\ m))$
 $\langle proof \rangle$

lemma *fmdrop-idle*: $x \notin fmdom\ B \implies fmdrop\ x\ B = B$
 $\langle proof \rangle$

lemma *fmdrop-idle'*: $x \notin fmdom'\ B \implies fmdrop\ x\ B = B$
 $\langle proof \rangle$

lemma *fmdrop-fmupd-same*: $fmdrop\ x\ (fmupd\ x\ y\ m) = fmdrop\ x\ m$
 $\langle proof \rangle$

lemma *fmdom'-restrict-set-precise*: $fmdom'\ (fmrestrict\ set\ A\ m) = fmdom'\ m \cap A$
 $\langle proof \rangle$

lemma *fmdom'-restrict-fset-precise*: $fmdom\ (fmrestrict\ fset\ A\ m) = fmdom\ m \upharpoonright A$
 $\langle proof \rangle$

lemma *fmdom'-drop-fset[simp]*: $fmdom'\ (fmdrop\ fset\ A\ m) = fmdom'\ m - fset\ A$
 $\langle proof \rangle$

lemma *fmdom'-restrict-fset*: $fmdom'\ (fmrestrict\ fset\ A\ m) \subseteq fset\ A$
 $\langle proof \rangle$

lemma *fmlookup-drop[simp]*:
 $fmlookup\ (fmdrop\ a\ m)\ x = (if\ x \neq a\ then\ fmlookup\ m\ x\ else\ None)$
 $\langle proof \rangle$

lemma *fmlookup-drop-set[simp]*:
 $fmlookup\ (fmdrop\ set\ A\ m)\ x = (if\ x \notin A\ then\ fmlookup\ m\ x\ else\ None)$
 $\langle proof \rangle$

lemma *fmlookup-drop-fset[simp]*:
 $fmlookup\ (fmdrop\ fset\ A\ m)\ x = (if\ x \notin A\ then\ fmlookup\ m\ x\ else\ None)$
 $\langle proof \rangle$

lemma *fmlookup-restrict-set[simp]*:
 $fmlookup\ (fmrestrict\ set\ A\ m)\ x = (if\ x \in A\ then\ fmlookup\ m\ x\ else\ None)$
 $\langle proof \rangle$

lemma *fmlookup-restrict-fset[simp]*:
 $fmlookup\ (fmrestrict\ fset\ A\ m)\ x = (if\ x \in A\ then\ fmlookup\ m\ x\ else\ None)$
 $\langle proof \rangle$

lemma *fmrestrict-set-dom[simp]*: $fmrestrict\ set\ (fmdom'\ m)\ m = m$

$\langle \text{proof} \rangle$

lemma $\text{fmrestrict-fset-dom}[\text{simp}]: \text{fmrestrict-fset} (\text{fmdom } m) \ m = m$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-empty}[\text{simp}]: \text{fmdrop } a \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-set-empty}[\text{simp}]: \text{fmdrop-set } A \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-fset-empty}[\text{simp}]: \text{fmdrop-fset } A \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-fset-fmdom}[\text{simp}]: \text{fmdrop-fset} (\text{fmdom } A) \ A = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-set-fmdom}[\text{simp}]: \text{fmdrop-set} (\text{fmdom}' A) \ A = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmrestrict-set-empty}[\text{simp}]: \text{fmrestrict-set } A \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmrestrict-fset-empty}[\text{simp}]: \text{fmrestrict-fset } A \ \text{fmempty} = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-set-null}[\text{simp}]: \text{fmdrop-set } \{\} \ m = m$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-fset-null}[\text{simp}]: \text{fmdrop-fset } \{|\} \ m = m$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-set-single}[\text{simp}]: \text{fmdrop-set } \{a\} \ m = \text{fmdrop } a \ m$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-fset-single}[\text{simp}]: \text{fmdrop-fset } \{|a|\} \ m = \text{fmdrop } a \ m$
 $\langle \text{proof} \rangle$

lemma $\text{fmrestrict-set-null}[\text{simp}]: \text{fmrestrict-set } \{\} \ m = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmrestrict-fset-null}[\text{simp}]: \text{fmrestrict-fset } \{|\} \ m = \text{fmempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-comm}: \text{fmdrop } a \ (\text{fmdrop } b \ m) = \text{fmdrop } b \ (\text{fmdrop } a \ m)$
 $\langle \text{proof} \rangle$

lemma $\text{fmdrop-set-insert}[\text{simp}]: \text{fmdrop-set} (\text{insert } x \ S) \ m = \text{fmdrop } x \ (\text{fmdrop-set } S \ m)$

$\langle proof \rangle$

lemma $fmdrop\text{-}fset\text{-}insert[simp]$: $fmdrop\text{-}fset (finsert\ x\ S)\ m = fmdrop\ x\ (fmdrop\text{-}fset\ S\ m)$

$\langle proof \rangle$

lemma $fmrestrict\text{-}set\text{-}twice[simp]$: $fmrestrict\text{-}set\ S\ (fmrestrict\text{-}set\ T\ m) = fmrestrict\text{-}set\ (S \cap T)\ m$

$\langle proof \rangle$

lemma $fmrestrict\text{-}fset\text{-}twice[simp]$: $fmrestrict\text{-}fset\ S\ (fmrestrict\text{-}fset\ T\ m) = fmrestrict\text{-}fset\ (S \mid\cap\mid T)\ m$

$\langle proof \rangle$

lemma $fmrestrict\text{-}set\text{-}drop[simp]$: $fmrestrict\text{-}set\ S\ (fmdrop\ b\ m) = fmrestrict\text{-}set\ (S - \{b\})\ m$

$\langle proof \rangle$

lemma $fmrestrict\text{-}fset\text{-}drop[simp]$: $fmrestrict\text{-}fset\ S\ (fmdrop\ b\ m) = fmrestrict\text{-}fset\ (S - \{\mid b \mid\})\ m$

$\langle proof \rangle$

lemma $fmdrop\text{-}fmrestrict\text{-}set[simp]$: $fmdrop\ b\ (fmrestrict\text{-}set\ S\ m) = fmrestrict\text{-}set\ (S - \{b\})\ m$

$\langle proof \rangle$

lemma $fmdrop\text{-}fmrestrict\text{-}fset[simp]$: $fmdrop\ b\ (fmrestrict\text{-}fset\ S\ m) = fmrestrict\text{-}fset\ (S - \{\mid b \mid\})\ m$

$\langle proof \rangle$

lemma $fmdrop\text{-}idem[simp]$: $fmdrop\ a\ (fmdrop\ a\ m) = fmdrop\ a\ m$

$\langle proof \rangle$

lemma $fmdrop\text{-}set\text{-}twice[simp]$: $fmdrop\text{-}set\ S\ (fmdrop\text{-}set\ T\ m) = fmdrop\text{-}set\ (S \cup T)\ m$

$\langle proof \rangle$

lemma $fmdrop\text{-}fset\text{-}twice[simp]$: $fmdrop\text{-}fset\ S\ (fmdrop\text{-}fset\ T\ m) = fmdrop\text{-}fset\ (S \mid\cup\mid T)\ m$

$\langle proof \rangle$

lemma $fmdrop\text{-}set\text{-}fmdrop[simp]$: $fmdrop\text{-}set\ S\ (fmdrop\ b\ m) = fmdrop\text{-}set\ (insert\ b\ S)\ m$

$\langle proof \rangle$

lemma $fmdrop\text{-}fset\text{-}fmdrop[simp]$: $fmdrop\text{-}fset\ S\ (fmdrop\ b\ m) = fmdrop\text{-}fset\ (finsert\ b\ S)\ m$

$\langle proof \rangle$

lift-definition $fmadd :: ('a, 'b) fmap \Rightarrow ('a, 'b) fmap \Rightarrow ('a, 'b) fmap$ (**infixl** $\langle ++_f \rangle$ 100)
is $map\text{-}add$
parametric $map\text{-}add\text{-}transfer$
 $\langle proof \rangle$

lemma $fmlookup\text{-}add[simp]$:
 $fmlookup (m ++_f n) x = (if x \in |fmdom\ n| then fmlookup\ n\ x else fmlookup\ m\ x)$
 $\langle proof \rangle$

lemma $fmdom\text{-}add[simp]$: $fmdom (m ++_f n) = fmdom\ m \cup |fmdom\ n|$ $\langle proof \rangle$
lemma $fmdom'\text{-}add[simp]$: $fmdom' (m ++_f n) = fmdom'\ m \cup fmdom'\ n$ $\langle proof \rangle$

lemma $fmadd\text{-}drop\text{-}left\text{-}dom$: $fmdrop\text{-}fset (fmdom\ n) m ++_f n = m ++_f n$
 $\langle proof \rangle$

lemma $fmadd\text{-}restrict\text{-}right\text{-}dom$: $fmrestrict\text{-}fset (fmdom\ n) (m ++_f n) = n$
 $\langle proof \rangle$

lemma $fmfilter\text{-}add\text{-}distrib[simp]$: $fmfilter\ P\ (m ++_f n) = fmfilter\ P\ m ++_f fmfilter\ P\ n$
 $\langle proof \rangle$

lemma $fmdrop\text{-}add\text{-}distrib[simp]$: $fmdrop\ a\ (m ++_f n) = fmdrop\ a\ m ++_f fmdrop\ a\ n$
 $\langle proof \rangle$

lemma $fmdrop\text{-}set\text{-}add\text{-}distrib[simp]$: $fmdrop\text{-}set\ A\ (m ++_f n) = fmdrop\text{-}set\ A\ m ++_f fmdrop\text{-}set\ A\ n$
 $\langle proof \rangle$

lemma $fmdrop\text{-}fset\text{-}add\text{-}distrib[simp]$: $fmdrop\text{-}fset\ A\ (m ++_f n) = fmdrop\text{-}fset\ A\ m ++_f fmdrop\text{-}fset\ A\ n$
 $\langle proof \rangle$

lemma $fmrestrict\text{-}set\text{-}add\text{-}distrib[simp]$:
 $fmrestrict\text{-}set\ A\ (m ++_f n) = fmrestrict\text{-}set\ A\ m ++_f fmrestrict\text{-}set\ A\ n$
 $\langle proof \rangle$

lemma $fmrestrict\text{-}fset\text{-}add\text{-}distrib[simp]$:
 $fmrestrict\text{-}fset\ A\ (m ++_f n) = fmrestrict\text{-}fset\ A\ m ++_f fmrestrict\text{-}fset\ A\ n$
 $\langle proof \rangle$

lemma $fmadd\text{-}empty[simp]$: $fmempty ++_f m = m ++_f fmempty = m$
 $\langle proof \rangle$

lemma $fmadd\text{-}idempotent[simp]$: $m ++_f m = m$
 $\langle proof \rangle$

lemma *fmadd-assoc[simp]*: $m ++_f (n ++_f p) = m ++_f n ++_f p$
 ⟨proof⟩

lemma *fmadd-fmupd[simp]*: $m ++_f \text{fmupd } a \ b \ n = \text{fmupd } a \ b \ (m ++_f n)$
 ⟨proof⟩

lift-definition *fmpr* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \text{fmap} \Rightarrow \text{bool}$
is *map-pred*
parametric *map-pred-transfer*
 ⟨proof⟩

lemma *fmpr*I[*intro*]:
assumes $\bigwedge x \ y. \text{fmlookup } m \ x = \text{Some } y \implies P \ x \ y$
shows *fmpr* $P \ m$
 ⟨proof⟩

lemma *fmpr*D[*dest*]: *fmpr* $P \ m \implies \text{fmlookup } m \ x = \text{Some } y \implies P \ x \ y$
 ⟨proof⟩

lemma *fmpr*-iff: *fmpr* $P \ m \longleftrightarrow (\forall x \ y. \text{fmlookup } m \ x = \text{Some } y \longrightarrow P \ x \ y)$
 ⟨proof⟩

lemma *fmpr*-alt-def: *fmpr* $P \ m \longleftrightarrow \text{fBall } (\text{fmdom } m) (\lambda x. P \ x \ (\text{the } (\text{fmlookup } m \ x)))$
 ⟨proof⟩

lemma *fmpr*-mono-strong:
assumes $\bigwedge x \ y. \text{fmlookup } m \ x = \text{Some } y \implies P \ x \ y \implies Q \ x \ y$
shows *fmpr* $P \ m \implies \text{fmpr } Q \ m$
 ⟨proof⟩

lemma *fmpr*-mono[*mono*]: $P \leq Q \implies \text{fmpr } P \leq \text{fmpr } Q$
 ⟨proof⟩

lemma *fmpr*-empty[*intro!*, *simp*]: *fmpr* $P \ \text{fmempty}$
 ⟨proof⟩

lemma *fmpr*-upd[*intro*]: *fmpr* $P \ m \implies P \ x \ y \implies \text{fmpr } P \ (\text{fmupd } x \ y \ m)$
 ⟨proof⟩

lemma *fmpr*-updD[*dest*]: *fmpr* $P \ (\text{fmupd } x \ y \ m) \implies P \ x \ y$
 ⟨proof⟩

lemma *fmpr*-add[*intro*]: *fmpr* $P \ m \implies \text{fmpr } P \ n \implies \text{fmpr } P \ (m ++_f n)$
 ⟨proof⟩

lemma *fmpr*-filter[*intro*]: *fmpr* $P \ m \implies \text{fmpr } P \ (\text{fmfilter } Q \ m)$
 ⟨proof⟩

lemma *fmpred-drop*[intro]: $fmpred\ P\ m \implies fmpred\ P\ (fmdrop\ a\ m)$
 ⟨proof⟩

lemma *fmpred-drop-set*[intro]: $fmpred\ P\ m \implies fmpred\ P\ (fmdrop-set\ A\ m)$
 ⟨proof⟩

lemma *fmpred-drop-fset*[intro]: $fmpred\ P\ m \implies fmpred\ P\ (fmdrop-fset\ A\ m)$
 ⟨proof⟩

lemma *fmpred-restrict-set*[intro]: $fmpred\ P\ m \implies fmpred\ P\ (fmrestrict-set\ A\ m)$
 ⟨proof⟩

lemma *fmpred-restrict-fset*[intro]: $fmpred\ P\ m \implies fmpred\ P\ (fmrestrict-fset\ A\ m)$
 ⟨proof⟩

lemma *fmpred-cases*[consumes 1]:
 assumes $fmpred\ P\ m$
 obtains $(none)\ fmlookup\ m\ x = None \mid (some)\ y\ \mathbf{where}\ fmlookup\ m\ x = Some\ y\ P\ x\ y$
 ⟨proof⟩

lift-definition *fmsubset* :: $('a,\ 'b)\ fmap \Rightarrow ('a,\ 'b)\ fmap \Rightarrow bool$ (**infix** \subseteq_f 50)
 is *map-le*
 ⟨proof⟩

lemma *fmsubset-alt-def*: $m \subseteq_f n \longleftrightarrow fmpred\ (\lambda k\ v.\ fmlookup\ n\ k = Some\ v)\ m$
 ⟨proof⟩

lemma *fmsubset-pred*: $fmpred\ P\ m \implies n \subseteq_f m \implies fmpred\ P\ n$
 ⟨proof⟩

lemma *fmsubset-filter-mono*: $m \subseteq_f n \implies fmfiter\ P\ m \subseteq_f fmfiter\ P\ n$
 ⟨proof⟩

lemma *fmsubset-drop-mono*: $m \subseteq_f n \implies fmdrop\ a\ m \subseteq_f fmdrop\ a\ n$
 ⟨proof⟩

lemma *fmsubset-drop-set-mono*: $m \subseteq_f n \implies fmdrop-set\ A\ m \subseteq_f fmdrop-set\ A\ n$
 ⟨proof⟩

lemma *fmsubset-drop-fset-mono*: $m \subseteq_f n \implies fmdrop-fset\ A\ m \subseteq_f fmdrop-fset\ A\ n$
 ⟨proof⟩

lemma *fmsubset-restrict-set-mono*: $m \subseteq_f n \implies fmrestrict-set\ A\ m \subseteq_f fmrestrict-set\ A\ n$
 ⟨proof⟩

lemma *fmsubset-restrict-fset-mono*: $m \subseteq_f n \implies \text{fmrestrict-fset } A \ m \subseteq_f \text{fmrestrict-fset } A \ n$

<proof>

lemma *fmfilter-subset[simp]*: $\text{fmfilter } P \ m \subseteq_f m$

<proof>

lemma *fmsubset-drop[simp]*: $\text{fmdrop } a \ m \subseteq_f m$

<proof>

lemma *fmsubset-drop-set[simp]*: $\text{fmdrop-set } S \ m \subseteq_f m$

<proof>

lemma *fmsubset-drop-fset[simp]*: $\text{fmdrop-fset } S \ m \subseteq_f m$

<proof>

lemma *fmsubset-restrict-set[simp]*: $\text{fmrestrict-set } S \ m \subseteq_f m$

<proof>

lemma *fmsubset-restrict-fset[simp]*: $\text{fmrestrict-fset } S \ m \subseteq_f m$

<proof>

lift-definition *fset-of-fmap* :: $('a, 'b) \text{ fmap} \Rightarrow ('a \times 'b) \text{ fset}$ **is** *set-of-map*

<proof>

lemma *fset-of-fmap-inj[intro, simp]*: $\text{inj } \text{fset-of-fmap}$

<proof>

lemma *fset-of-fmap-iff[simp]*: $(a, b) \in \text{fset-of-fmap } m \longleftrightarrow \text{fmlookup } m \ a = \text{Some } b$

<proof>

lemma *fset-of-fmap-iff'*: $(a, b) \in \text{fset } (\text{fset-of-fmap } m) \longleftrightarrow \text{fmlookup } m \ a = \text{Some } b$

<proof>

lift-definition *fmap-of-list* :: $('a \times 'b) \text{ list} \Rightarrow ('a, 'b) \text{ fmap}$

is *map-of*

parametric *map-of-transfer*

<proof>

lemma *fmap-of-list-simps[simp]*:

$\text{fmap-of-list } [] = \text{fmempty}$

$\text{fmap-of-list } ((k, v) \# kvs) = \text{fmupd } k \ v \ (\text{fmap-of-list } kvs)$

<proof>

lemma *fmap-of-list-app[simp]*: $\text{fmap-of-list } (xs @ ys) = \text{fmap-of-list } ys \ ++_f \text{fmap-of-list } xs$

<proof>

lemma *fmupd-alt-def*: $\text{fmupd } k \ v \ m = m ++_f \text{fmap-of-list } [(k, v)]$
 ⟨proof⟩

lemma *fmprered-of-list[intro]*:
 assumes $\bigwedge k \ v. (k, v) \in \text{set } xs \implies P \ k \ v$
 shows $\text{fmprered } P \ (\text{fmap-of-list } xs)$
 ⟨proof⟩

lemma *fmap-of-list-SomeD*: $\text{fmlookup } (\text{fmap-of-list } xs) \ k = \text{Some } v \implies (k, v) \in \text{set } xs$
 ⟨proof⟩

lemma *fmdom-fmap-of-list[simp]*: $\text{fmdom } (\text{fmap-of-list } xs) = \text{fset-of-list } (\text{map fst } xs)$
 ⟨proof⟩

lift-definition *fmrel-on-fset* :: $'a \ \text{fset} \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \ \text{fmap} \Rightarrow ('a, 'c) \ \text{fmap} \Rightarrow \text{bool}$
 is *rel-map-on-set*
 ⟨proof⟩

lemma *fmrel-on-fset-alt-def*: $\text{fmrel-on-fset } S \ P \ m \ n \longleftrightarrow \text{fBall } S \ (\lambda x. \text{rel-option } P \ (\text{fmlookup } m \ x) \ (\text{fmlookup } n \ x))$
 ⟨proof⟩

lemma *fmrel-on-fsetI[intro]*:
 assumes $\bigwedge x. x \in S \implies \text{rel-option } P \ (\text{fmlookup } m \ x) \ (\text{fmlookup } n \ x)$
 shows $\text{fmrel-on-fset } S \ P \ m \ n$
 ⟨proof⟩

lemma *fmrel-on-fset-mono[mono]*: $R \leq Q \implies \text{fmrel-on-fset } S \ R \leq \text{fmrel-on-fset } S \ Q$
 ⟨proof⟩

lemma *fmrel-on-fsetD*: $x \in S \implies \text{fmrel-on-fset } S \ P \ m \ n \implies \text{rel-option } P \ (\text{fmlookup } m \ x) \ (\text{fmlookup } n \ x)$
 ⟨proof⟩

lemma *fmrel-on-fsubset*: $\text{fmrel-on-fset } S \ R \ m \ n \implies T \subseteq S \implies \text{fmrel-on-fset } T \ R \ m \ n$
 ⟨proof⟩

lemma *fmrel-on-fset-unionI*:
 $\text{fmrel-on-fset } A \ R \ m \ n \implies \text{fmrel-on-fset } B \ R \ m \ n \implies \text{fmrel-on-fset } (A \cup B) \ R \ m \ n$
 ⟨proof⟩

lemma *fmrel-on-fset-updateI*:

assumes $\text{fmrel-on-fset } S \ P \ m \ n \ P \ v_1 \ v_2$
shows $\text{fmrel-on-fset } (\text{finsert } k \ S) \ P \ (\text{fmupd } k \ v_1 \ m) \ (\text{fmupd } k \ v_2 \ n)$
 $\langle \text{proof} \rangle$

lift-definition $\text{fmimage} :: ('a, 'b) \text{fmap} \Rightarrow 'a \text{fset} \Rightarrow 'b \text{fset}$ **is** $\lambda m \ S. \{b \mid a \ b. \ m \ a = \text{Some } b \wedge a \in S\}$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-alt-def}: \text{fmimage } m \ S = \text{fmran } (\text{fmrestrict-fset } S \ m)$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-empty[simp]}: \text{fmimage } m \ \text{fempty} = \text{fempty}$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-subset-ran[simp]}: \text{fmimage } m \ S \subseteq \text{fmran } m$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-dom[simp]}: \text{fmimage } m \ (\text{fmdom } m) = \text{fmran } m$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-inter}: \text{fmimage } m \ (A \ |\cap| \ B) \subseteq \text{fmimage } m \ A \ |\cap| \ \text{fmimage } m \ B$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-inter-dom[simp]}:$
 $\text{fmimage } m \ (\text{fmdom } m \ |\cap| \ A) = \text{fmimage } m \ A$
 $\text{fmimage } m \ (A \ |\cap| \ \text{fmdom } m) = \text{fmimage } m \ A$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-union[simp]}: \text{fmimage } m \ (A \ |\cup| \ B) = \text{fmimage } m \ A \ |\cup| \ \text{fmimage } m \ B$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-Union[simp]}: \text{fmimage } m \ (\text{ffUnion } A) = \text{ffUnion } (\text{fmimage } m \ `| \ A)$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-filter[simp]}: \text{fmimage } (\text{fmfilter } P \ m) \ A = \text{fmimage } m \ (\text{ffilter } P \ A)$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-drop[simp]}: \text{fmimage } (\text{fmdrop } a \ m) \ A = \text{fmimage } m \ (A - \{|a|\})$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-drop-fset[simp]}: \text{fmimage } (\text{fmdrop-fset } B \ m) \ A = \text{fmimage } m \ (A - B)$
 $\langle \text{proof} \rangle$

lemma $\text{fmimage-restrict-fset[simp]}: \text{fmimage } (\text{fmrestrict-fset } B \ m) \ A = \text{fmimage } m \ (A \ |\cap| \ B)$
 $\langle \text{proof} \rangle$

lemma *fmfilter-ran[simp]*: $\text{fmran } (\text{fmfilter } P \ m) = \text{fmimage } m \ (\text{ffilter } P \ (\text{fndom } m))$
 ⟨proof⟩

lemma *fmran-drop[simp]*: $\text{fmran } (\text{fmdrop } a \ m) = \text{fmimage } m \ (\text{fndom } m - \{|a|\})$
 ⟨proof⟩

lemma *fmran-drop-fset[simp]*: $\text{fmran } (\text{fmdrop-fset } A \ m) = \text{fmimage } m \ (\text{fndom } m - A)$
 ⟨proof⟩

lemma *fmran-restrict-fset*: $\text{fmran } (\text{fmrestrict-fset } A \ m) = \text{fmimage } m \ (\text{fndom } m \cap A)$
 ⟨proof⟩

lemma *fmlookup-image-iff*: $y \in \text{fmimage } m \ A \longleftrightarrow (\exists x. \text{fmlookup } m \ x = \text{Some } y \wedge x \in A)$
 ⟨proof⟩

lemma *fmimageI*: $\text{fmlookup } m \ x = \text{Some } y \implies x \in A \implies y \in \text{fmimage } m \ A$
 ⟨proof⟩

lemma *fmimageE[elim]*:
 assumes $y \in \text{fmimage } m \ A$
 obtains x where $\text{fmlookup } m \ x = \text{Some } y \wedge x \in A$
 ⟨proof⟩

lift-definition *fmcomp* :: $('b, 'c) \text{ fmap} \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow ('a, 'c) \text{ fmap}$ (**infixl** \circ_f 55)
 is *map-comp*
 parametric *map-comp-transfer*
 ⟨proof⟩

lemma *fmlookup-comp[simp]*: $\text{fmlookup } (m \circ_f n) \ x = \text{Option.bind } (\text{fmlookup } n \ x) \ (\text{fmlookup } m)$
 ⟨proof⟩

end

30.4 BNF setup

lift-bnf $('a, \text{fmran}' : 'b) \text{ fmap}$ [*wits*: *Map.empty*]
 for *map*: *fmmmap*
 rel: *fmrel*
 ⟨proof⟩

declare *fmap.pred-mono*[*mono*]

lemma *fmran'-alt-def*: $\text{fmran}'\ m = \text{fset}\ (\text{fmran}\ m)$
including *fset.lifting*
 $\langle \text{proof} \rangle$

lemma *fmlookup-ran'-iff*: $y \in \text{fmran}'\ m \longleftrightarrow (\exists x. \text{fmlookup}\ m\ x = \text{Some}\ y)$
 $\langle \text{proof} \rangle$

lemma *fmran'I*: $\text{fmlookup}\ m\ x = \text{Some}\ y \implies y \in \text{fmran}'\ m$
 $\langle \text{proof} \rangle$

lemma *fmran'E[elim]*:
assumes $y \in \text{fmran}'\ m$
obtains x **where** $\text{fmlookup}\ m\ x = \text{Some}\ y$
 $\langle \text{proof} \rangle$

lemma *fmrel-iff*: $\text{fmrel}\ R\ m\ n \longleftrightarrow (\forall x. \text{rel-option}\ R\ (\text{fmlookup}\ m\ x)\ (\text{fmlookup}\ n\ x))$
 $\langle \text{proof} \rangle$

lemma *fmrelI[intro]*:
assumes $\bigwedge x. \text{rel-option}\ R\ (\text{fmlookup}\ m\ x)\ (\text{fmlookup}\ n\ x)$
shows $\text{fmrel}\ R\ m\ n$
 $\langle \text{proof} \rangle$

lemma *fmrel-upd[intro]*: $\text{fmrel}\ P\ m\ n \implies P\ x\ y \implies \text{fmrel}\ P\ (\text{fmupd}\ k\ x\ m)\ (\text{fmupd}\ k\ y\ n)$
 $\langle \text{proof} \rangle$

lemma *fmrelD[dest]*: $\text{fmrel}\ P\ m\ n \implies \text{rel-option}\ P\ (\text{fmlookup}\ m\ x)\ (\text{fmlookup}\ n\ x)$
 $\langle \text{proof} \rangle$

lemma *fmrel-addI[intro]*:
assumes $\text{fmrel}\ P\ m\ n\ \text{fmrel}\ P\ a\ b$
shows $\text{fmrel}\ P\ (m\ ++_f\ a)\ (n\ ++_f\ b)$
 $\langle \text{proof} \rangle$

lemma *fmrel-cases[consumes 1]*:
assumes $\text{fmrel}\ P\ m\ n$
obtains $(\text{none})\ \text{fmlookup}\ m\ x = \text{None}\ \text{fmlookup}\ n\ x = \text{None}$
 $\quad \mid (\text{some})\ a\ b$ **where** $\text{fmlookup}\ m\ x = \text{Some}\ a\ \text{fmlookup}\ n\ x = \text{Some}\ b\ P\ a\ b$
 $\langle \text{proof} \rangle$

lemma *fmrel-filter[intro]*: $\text{fmrel}\ P\ m\ n \implies \text{fmrel}\ P\ (\text{fmfilter}\ Q\ m)\ (\text{fmfilter}\ Q\ n)$
 $\langle \text{proof} \rangle$

lemma *fmrel-drop[intro]*: $\text{fmrel}\ P\ m\ n \implies \text{fmrel}\ P\ (\text{fmdrop}\ a\ m)\ (\text{fmdrop}\ a\ n)$
 $\langle \text{proof} \rangle$

lemma *fmrel-drop-set[intro]*: $\text{fmrel } P \ m \ n \implies \text{fmrel } P \ (\text{fmdrop-set } A \ m) \ (\text{fmdrop-set } A \ n)$
 $\langle \text{proof} \rangle$

lemma *fmrel-drop-fset[intro]*: $\text{fmrel } P \ m \ n \implies \text{fmrel } P \ (\text{fmdrop-fset } A \ m) \ (\text{fmdrop-fset } A \ n)$
 $\langle \text{proof} \rangle$

lemma *fmrel-restrict-set[intro]*: $\text{fmrel } P \ m \ n \implies \text{fmrel } P \ (\text{fmrestrict-set } A \ m) \ (\text{fmrestrict-set } A \ n)$
 $\langle \text{proof} \rangle$

lemma *fmrel-restrict-fset[intro]*: $\text{fmrel } P \ m \ n \implies \text{fmrel } P \ (\text{fmrestrict-fset } A \ m) \ (\text{fmrestrict-fset } A \ n)$
 $\langle \text{proof} \rangle$

lemma *fmrel-on-fset-fmrel-restrict*:
 $\text{fmrel-on-fset } S \ P \ m \ n \longleftrightarrow \text{fmrel } P \ (\text{fmrestrict-fset } S \ m) \ (\text{fmrestrict-fset } S \ n)$
 $\langle \text{proof} \rangle$

lemma *fmrel-on-fset-refl-strong*:
assumes $\bigwedge x \ y. \ x \in S \implies \text{fmlookup } m \ x = \text{Some } y \implies P \ y \ y$
shows $\text{fmrel-on-fset } S \ P \ m \ m$
 $\langle \text{proof} \rangle$

lemma *fmrel-on-fset-addI*:
assumes $\text{fmrel-on-fset } S \ P \ m \ n \ \text{fmrel-on-fset } S \ P \ a \ b$
shows $\text{fmrel-on-fset } S \ P \ (m ++_f a) \ (n ++_f b)$
 $\langle \text{proof} \rangle$

lemma *fmrel-fmdom-eq*:
assumes $\text{fmrel } P \ x \ y$
shows $\text{fmdom } x = \text{fmdom } y$
 $\langle \text{proof} \rangle$

lemma *fmrel-fmdom'-eq*: $\text{fmrel } P \ x \ y \implies \text{fmdom}' x = \text{fmdom}' y$
 $\langle \text{proof} \rangle$

lemma *fmrel-rel-fmran*:
assumes $\text{fmrel } P \ x \ y$
shows $\text{rel-fset } P \ (\text{fmran } x) \ (\text{fmran } y)$
 $\langle \text{proof} \rangle$

lemma *fmrel-rel-fmran'*: $\text{fmrel } P \ x \ y \implies \text{rel-set } P \ (\text{fmran}' x) \ (\text{fmran}' y)$
 $\langle \text{proof} \rangle$

lemma *pred-fmap-fmpred[simp]*: $\text{pred-fmap } P = \text{fmpred } (\lambda \cdot. P)$
 $\langle \text{proof} \rangle$

lemma *pred-fmap-id[simp]*: $\text{pred-fmap id (fmap f m)} \longleftrightarrow \text{pred-fmap f m}$
 ⟨proof⟩

lemma *pred-fmapD*: $\text{pred-fmap P m} \implies x \in \text{fmap m} \implies P x$
 ⟨proof⟩

lemma *fmllookup-map[simp]*: $\text{fmllookup (fmap f m) x} = \text{map-option f (fmllookup m x)}$
 ⟨proof⟩

lemma *fmpred-map[simp]*: $\text{fmpred P (fmap f m)} \longleftrightarrow \text{fmpred } (\lambda k v. P k (f v)) m$
 ⟨proof⟩

lemma *fmpred-id[simp]*: $\text{fmpred } (\lambda \cdot. \text{id}) (\text{fmap f m}) \longleftrightarrow \text{fmpred } (\lambda \cdot. f) m$
 ⟨proof⟩

lemma *fmap-add[simp]*: $\text{fmap f (m ++}_f \text{ n)} = \text{fmap f m ++}_f \text{ fmap f n}$
 ⟨proof⟩

lemma *fmap-empty[simp]*: $\text{fmap f fempty} = \text{fempty}$
 ⟨proof⟩

lemma *fmdom-map[simp]*: $\text{fmdom (fmap f m)} = \text{fmdom m}$
including *fset.lifting*
 ⟨proof⟩

lemma *fmdom'-map[simp]*: $\text{fmdom' (fmap f m)} = \text{fmdom' m}$
 ⟨proof⟩

lemma *fmran-fmap[simp]*: $\text{fmran (fmap f m)} = f \mid \text{fmran m}$
including *fset.lifting*
 ⟨proof⟩

lemma *fmran'-fmap[simp]*: $\text{fmran' (fmap f m)} = f \text{ ' fmran' m}$
 ⟨proof⟩

lemma *fmfilter-fmap[simp]*: $\text{fmfilter P (fmap f m)} = \text{fmap f (fmfilter P m)}$
 ⟨proof⟩

lemma *fmdrop-fmap[simp]*: $\text{fmdrop a (fmap f m)} = \text{fmap f (fmdrop a m)}$
 ⟨proof⟩

lemma *fmdrop-set-fmap[simp]*: $\text{fmdrop-set A (fmap f m)} = \text{fmap f (fmdrop-set A m)}$
 ⟨proof⟩

lemma *fmdrop-fset-fmap[simp]*: $\text{fmdrop-fset A (fmap f m)} = \text{fmap f (fmdrop-fset A m)}$
 ⟨proof⟩

lemma *fmrestrict-set-fmmap[simp]*: *fmrestrict-set* A (*fmmap* f m) = *fmmap* f (*fmrestrict-set* A m)
 ⟨*proof*⟩

lemma *fmrestrict-fset-fmmap[simp]*: *fmrestrict-fset* A (*fmmap* f m) = *fmmap* f (*fmrestrict-fset* A m)
 ⟨*proof*⟩

lemma *fmmap-subset[intro]*: $m \subseteq_f n \implies \text{fmmap } f \ m \subseteq_f \text{fmmap } f \ n$
 ⟨*proof*⟩

lemma *fmmap-fset-of-fmap*: *fset-of-fmap* (*fmmap* f m) = $(\lambda(k, v). (k, f \ v)) \mid \uparrow$
fset-of-fmap m
including *fset.lifting*
 ⟨*proof*⟩

lemma *fmmap-fmupd*: *fmmap* f (*fmupd* x y m) = *fmupd* x ($f \ y$) (*fmmap* f m)
 ⟨*proof*⟩

30.5 size setup

definition *size-fmap* :: $('a \Rightarrow \text{nat}) \Rightarrow ('b \Rightarrow \text{nat}) \Rightarrow ('a, 'b) \text{fmap} \Rightarrow \text{nat}$ **where**
[simp]: *size-fmap* f g m = *size-fset* $(\lambda(a, b). f \ a + g \ b)$ (*fset-of-fmap* m)

instantiation *fmap* :: $(\text{type}, \text{type}) \text{size}$ **begin**

definition *size-fmap where*

size-fmap-overloaded-def: *size-fmap* = *Finite-Map.size-fmap* $(\lambda-. 0)$ $(\lambda-. 0)$

instance ⟨*proof*⟩

end

lemma *size-fmap-overloaded-simps[simp]*: *size* x = *size* (*fset-of-fmap* x)
 ⟨*proof*⟩

lemma *fmap-size-o-map*: *size-fmap* f $g \circ \text{fmmap } h$ = *size-fmap* f $(g \circ h)$
 ⟨*proof*⟩

⟨*ML*⟩

30.6 Additional operations

lift-definition *fmmap-keys* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b) \text{fmap} \Rightarrow ('a, 'c) \text{fmap}$ **is**
 $\lambda f \ m \ a. \text{map-option } (f \ a) \ (m \ a)$
 ⟨*proof*⟩

lemma *fmprered-fmmap-keys[simp]*: *fmprered* P (*fmmap-keys* f m) = *fmprered* $(\lambda a \ b. P \ a \ (f \ a \ b)) \ m$

$\langle \text{proof} \rangle$

lemma *fmdom-fmmap-keys[simp]*: $\text{fmdom} (\text{fmmap-keys } f \ m) = \text{fmdom } m$
including *fset.lifting*
 $\langle \text{proof} \rangle$

lemma *fmlookup-fmmap-keys[simp]*: $\text{fmlookup} (\text{fmmap-keys } f \ m) \ x = \text{map-option} (f \ x) (\text{fmlookup } m \ x)$
 $\langle \text{proof} \rangle$

lemma *fmfilter-fmmap-keys[simp]*: $\text{fmfilter } P (\text{fmmap-keys } f \ m) = \text{fmmap-keys } f (\text{fmfilter } P \ m)$
 $\langle \text{proof} \rangle$

lemma *fmdrop-fmmap-keys[simp]*: $\text{fmdrop } a (\text{fmmap-keys } f \ m) = \text{fmmap-keys } f (\text{fmdrop } a \ m)$
 $\langle \text{proof} \rangle$

lemma *fmdrop-set-fmmap-keys[simp]*: $\text{fmdrop-set } A (\text{fmmap-keys } f \ m) = \text{fmmap-keys } f (\text{fmdrop-set } A \ m)$
 $\langle \text{proof} \rangle$

lemma *fmdrop-fset-fmmap-keys[simp]*: $\text{fmdrop-fset } A (\text{fmmap-keys } f \ m) = \text{fmmap-keys } f (\text{fmdrop-fset } A \ m)$
 $\langle \text{proof} \rangle$

lemma *fmrestrict-set-fmmap-keys[simp]*: $\text{fmrestrict-set } A (\text{fmmap-keys } f \ m) = \text{fmmap-keys } f (\text{fmrestrict-set } A \ m)$
 $\langle \text{proof} \rangle$

lemma *fmrestrict-fset-fmmap-keys[simp]*: $\text{fmrestrict-fset } A (\text{fmmap-keys } f \ m) = \text{fmmap-keys } f (\text{fmrestrict-fset } A \ m)$
 $\langle \text{proof} \rangle$

lemma *fmmap-keys-subset[intro]*: $m \subseteq_f n \implies \text{fmmap-keys } f \ m \subseteq_f \text{fmmap-keys } f \ n$
 $\langle \text{proof} \rangle$

definition *sorted-list-of-fmap* :: $('a::\text{linorder}, 'b) \text{ fmap} \Rightarrow ('a \times 'b) \text{ list}$ **where**
 $\text{sorted-list-of-fmap } m = \text{map } (\lambda k. (k, \text{the } (\text{fmlookup } m \ k))) (\text{sorted-list-of-fset } (\text{fmdom } m))$

lemma *list-all-sorted-list[simp]*: $\text{list-all } P (\text{sorted-list-of-fmap } m) = \text{fmpred } (\text{curry } P) \ m$
 $\langle \text{proof} \rangle$

lemma *map-of-sorted-list[simp]*: $\text{map-of } (\text{sorted-list-of-fmap } m) = \text{fmlookup } m$
 $\langle \text{proof} \rangle$
including *fset.lifting*

$\langle \text{proof} \rangle$

30.7 Additional properties

lemma *fmchoice'*:
 assumes *finite* *S* $\forall x \in S. \exists y. Q\ x\ y$
 shows $\exists m. \text{fmdom}'\ m = S \wedge \text{fmpred}\ Q\ m$
 $\langle \text{proof} \rangle$

30.8 Lifting/transfer setup

context includes *lifting-syntax* **begin**

lemma *fmempty-transfer*[*simp*, *intro*, *transfer-rule*]: *fmrel* *P* *fmempty* *fmempty*
 $\langle \text{proof} \rangle$

lemma *fmadd-transfer*[*transfer-rule*]:
 $(\text{fmrel}\ P \implies \text{fmrel}\ P \implies \text{fmrel}\ P)\ \text{fmadd}\ \text{fmadd}$
 $\langle \text{proof} \rangle$

lemma *fmupd-transfer*[*transfer-rule*]:
 $((=) \implies P \implies \text{fmrel}\ P \implies \text{fmrel}\ P)\ \text{fmupd}\ \text{fmupd}$
 $\langle \text{proof} \rangle$

end

lemma *Quotient-fmap-bnf*[*quot-map*]:
 assumes *Quotient* *R* *Abs* *Rep* *T*
 shows *Quotient* (*fmrel* *R*) (*fmap* *Abs*) (*fmap* *Rep*) (*fmrel* *T*)
 $\langle \text{proof} \rangle$

30.9 View as datatype

lemma *fmap-distinct*[*simp*]:
 $\text{fmempty} \neq \text{fmupd}\ k\ v\ m$
 $\text{fmupd}\ k\ v\ m \neq \text{fmempty}$
 $\langle \text{proof} \rangle$

lifting-update *fmap.lifting*

lemma *fmap-exhaust*[*cases type: fmap*]:
 obtains (*fmempty*) *m* = *fmempty*
 | (*fmupd*) *x* *y* *m'* **where** *m* = *fmupd* *x* *y* *m'* $x \notin \text{fmdom}\ m'$
 $\langle \text{proof} \rangle$ **including** *fmap.lifting* **and** *fset.lifting*
 $\langle \text{proof} \rangle$

lemma *fmap-induct*[*case-names fmempty fmupd*, *induct type: fmap*]:
 assumes *P* *fmempty*
 assumes $(\bigwedge x\ y\ m. P\ m \implies \text{fmlookup}\ m\ x = \text{None} \implies P\ (\text{fmupd}\ x\ y\ m))$
 shows *P* *m*

$\langle proof \rangle$

30.10 Code setup

instantiation $fmap :: (type, equal) \Rightarrow equal$ **begin**

definition $equal-fmap \equiv fmrel\ HOL.equal$

instance $\langle proof \rangle$

end

lemma $fBall\text{-}alt\text{-}def$: $fBall\ S\ P \longleftrightarrow (\forall x. x \in S \longrightarrow P\ x)$
 $\langle proof \rangle$

lemma $fmrel\text{-}code$:

$fmrel\ R\ m\ n \longleftrightarrow$
 $fBall\ (fmdom\ m)\ (\lambda x. rel\text{-}option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)) \wedge$
 $fBall\ (fmdom\ n)\ (\lambda x. rel\text{-}option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x))$
 $\langle proof \rangle$

lemmas $[code] =$

$fmrel\text{-}code$
 $fmran'\text{-}alt\text{-}def$
 $fmdom'\text{-}alt\text{-}def$
 $fmfilter\text{-}alt\text{-}defs$
 $pred\text{-}fmap\text{-}fmpred$
 $fmsubset\text{-}alt\text{-}def$
 $fmupd\text{-}alt\text{-}def$
 $fmrel\text{-}on\text{-}fset\text{-}alt\text{-}def$
 $fmpred\text{-}alt\text{-}def$

code-datatype $fmap\text{-}of\text{-}list$

quickcheck-generator $fmap$ *constructors*: $fmap\text{-}of\text{-}list$

context **includes** $fset.lifting$ **begin**

lemma $fmlookup\text{-}of\text{-}list[code]$: $fmlookup\ (fmap\text{-}of\text{-}list\ m) = map\text{-}of\ m$
 $\langle proof \rangle$

lemma $fmempty\text{-}of\text{-}list[code]$: $fmempty = fmap\text{-}of\text{-}list\ []$
 $\langle proof \rangle$

lemma $fmran\text{-}of\text{-}list[code]$: $fmran\ (fmap\text{-}of\text{-}list\ m) = snd\ |\cdot| \ fset\text{-}of\text{-}list\ (AList.clearjunk\ m)$
 $\langle proof \rangle$

lemma $fmdom\text{-}of\text{-}list[code]$: $fmdom\ (fmap\text{-}of\text{-}list\ m) = fst\ |\cdot| \ fset\text{-}of\text{-}list\ m$

<proof>

lemma *fmfilter-of-list*[code]: *fmfilter* *P* (*fmap-of-list* *m*) = *fmap-of-list* (*filter* ($\lambda(k, -). P\ k$) *m*)
<proof>

lemma *fmadd-of-list*[code]: *fmap-of-list* *m* ++_f *fmap-of-list* *n* = *fmap-of-list* (*AList.merge* *m* *n*)
<proof>

lemma *fmmmap-of-list*[code]: *fmmmap* *f* (*fmap-of-list* *m*) = *fmap-of-list* (*map* (*apsnd* *f*) *m*)
<proof>

lemma *fmmmap-keys-of-list*[code]:
fmmmap-keys *f* (*fmap-of-list* *m*) = *fmap-of-list* (*map* ($\lambda(a, b). (a, f\ a\ b)$) *m*)
<proof>

lemma *fmimage-of-list*[code]:
fmimage (*fmap-of-list* *m*) *A* = *fset-of-list* (*map snd* (*filter* ($\lambda(k, -). k \in A$) (*AList.clearjunk* *m*)))
<proof>

lemma *fmcomp-list*[code]:
fmap-of-list *m* \circ_f *fmap-of-list* *n* = *fmap-of-list* (*AList.compose* *n* *m*)
<proof>

end

30.11 Instances

lemma *exists-map-of*:
assumes *finite* (*dom* *m*) **shows** $\exists xs. \text{map-of } xs = m$
<proof>

lemma *exists-fmap-of-list*: $\exists xs. \text{fmap-of-list } xs = m$
<proof>

lemma *fmap-of-list-surj*[simp, intro]: *surj* *fmap-of-list*
<proof>

instance *fmap* :: (*countable*, *countable*) *countable*
<proof>

instance *fmap* :: (*finite*, *finite*) *finite*
<proof>

lifting-update *fmap.lifting*
lifting-forget *fmap.lifting*

30.12 Tests

export-code

*Ball fset fmrel fmran fmran' fmdom fmdom' fmpred pred-fmap fmsubset fmupd
 fmrel-on-fset
 fmdrop fmdrop-set fmdrop-fset fmrestrict-set fmrestrict-fset fmimage fmlookup
 fmempty
 fmfilter fmadd fmmmap fmmmap-keys fmcomp
 checking SML Scala Haskell? OCaml?*

— *lifting* through *fmap*

experiment begin

context includes *fset.lifting* begin

lift-definition *test1* :: ('a, 'b fset) fmap is *fmempty* :: ('a, 'b set) fmap
 ⟨*proof*⟩

lift-definition *test2* :: 'a ⇒ 'b ⇒ ('a, 'b fset) fmap is λa b. *fmupd* a {b} *fmempty*
 ⟨*proof*⟩

end

end

end

31 Disjoint FSets

theory *Disjoint-FSets*

imports

HOL-Library.Finite-Map

Disjoint-Sets

begin

context

includes *fset.lifting*

begin

lift-definition *fdisjnt* :: 'a fset ⇒ 'a fset ⇒ bool is *disjnt* ⟨*proof*⟩

lemma *fdisjnt-alt-def*: *fdisjnt* M N ⇔ (M |∩| N = {||})
 ⟨*proof*⟩

lemma *fdisjnt-insert*: x ∉ N ⇒ *fdisjnt* M N ⇒ *fdisjnt* (finsert x M) N
 ⟨*proof*⟩

lemma *fdisjnt-subset-right*: N' |⊆| N ⇒ *fdisjnt* M N ⇒ *fdisjnt* M N'

<proof>

lemma *fdisjnt-subset-left*: $N' \sqsubseteq N \implies \text{fdisjnt } N \ M \implies \text{fdisjnt } N' \ M$
<proof>

lemma *fdisjnt-union-right*: $\text{fdisjnt } M \ A \implies \text{fdisjnt } M \ B \implies \text{fdisjnt } M \ (A \mid\cup\mid B)$
<proof>

lemma *fdisjnt-union-left*: $\text{fdisjnt } A \ M \implies \text{fdisjnt } B \ M \implies \text{fdisjnt } (A \mid\cup\mid B) \ M$
<proof>

lemma *fdisjnt-swap*: $\text{fdisjnt } M \ N \implies \text{fdisjnt } N \ M$
including *fset.lifting* *<proof>*

lemma *distinct-append-fset*:
assumes *distinct xs distinct ys fdisjnt (fset-of-list xs) (fset-of-list ys)*
shows *distinct (xs @ ys)*
<proof>

lemma *fdisjnt-contrI*:
assumes $\bigwedge x. x \mid\in\mid M \implies x \mid\in\mid N \implies \text{False}$
shows $\text{fdisjnt } M \ N$
<proof>

lemma *fdisjnt-Union-left*: $\text{fdisjnt } (\text{ffUnion } S) \ T \longleftrightarrow \text{fBall } S \ (\lambda S. \text{fdisjnt } S \ T)$
<proof>

lemma *fdisjnt-Union-right*: $\text{fdisjnt } T \ (\text{ffUnion } S) \longleftrightarrow \text{fBall } S \ (\lambda S. \text{fdisjnt } T \ S)$
<proof>

lemma *fdisjnt-ge-max*: $\text{fBall } X \ (\lambda x. x > \text{fMax } Y) \implies \text{fdisjnt } X \ Y$
<proof>

end

lemma *fmadd-disjnt*: $\text{fdisjnt } (\text{fmdom } m) \ (\text{fmdom } n) \implies m \ ++_f \ n = n \ ++_f \ m$
<proof>
including *fset.lifting* **and** *fmap.lifting*
<proof>

end

32 Lists with elements distinct as canonical example for datatype invariants

theory *Dlist*
imports *Confluent-Quotient*

begin

32.1 The type of distinct lists

typedef 'a dlist = {xs::'a list. distinct xs}
morphisms list-of-dlist Abs-dlist
 ⟨proof⟩

context begin

qualified definition dlist-eq **where** dlist-eq = BNF-Def.vimage2p remdups remdups
 (=)

qualified lemma equivp-dlist-eq: equivp dlist-eq
 ⟨proof⟩ **definition** abs-dlist :: 'a list \Rightarrow 'a dlist **where** abs-dlist = Abs-dlist o
 remdups

definition qcr-dlist :: 'a list \Rightarrow 'a dlist \Rightarrow bool **where** qcr-dlist x y \longleftrightarrow y =
 abs-dlist x

qualified lemma Quotient-dlist-remdups: Quotient dlist-eq abs-dlist list-of-dlist
 qcr-dlist
 ⟨proof⟩

end

locale Quotient-dlist **begin**

setup-lifting Dlist.Quotient-dlist-remdups Dlist.equivp-dlist-eq[THEN equivp-reflp2]
end

setup-lifting type-definition-dlist

lemma dlist-eq-iff:
 dxs = dys \longleftrightarrow list-of-dlist dxs = list-of-dlist dys
 ⟨proof⟩

lemma dlist-eqI:
 list-of-dlist dxs = list-of-dlist dys \Longrightarrow dxs = dys
 ⟨proof⟩

Formal, totalized constructor for 'a dlist:

definition Dlist :: 'a list \Rightarrow 'a dlist **where**
 Dlist xs = Abs-dlist (remdups xs)

lemma distinct-list-of-dlist [simp, intro]:
 distinct (list-of-dlist dxs)
 ⟨proof⟩

lemma list-of-dlist-Dlist [simp]:
 list-of-dlist (Dlist xs) = remdups xs

$\langle proof \rangle$

lemma *remdups-list-of-dlist* [simp]:
 $remdups\ (list-of-dlist\ dxs) = list-of-dlist\ dxs$
 $\langle proof \rangle$

lemma *Dlist-list-of-dlist* [simp, code abstype]:
 $Dlist\ (list-of-dlist\ dxs) = dxs$
 $\langle proof \rangle$

Fundamental operations:

context
begin

qualified definition *empty* :: 'a dlist **where**
 $empty = Dlist\ []$

qualified definition *insert* :: 'a \Rightarrow 'a dlist \Rightarrow 'a dlist **where**
 $insert\ x\ dxs = Dlist\ (List.insert\ x\ (list-of-dlist\ dxs))$

qualified definition *remove* :: 'a \Rightarrow 'a dlist \Rightarrow 'a dlist **where**
 $remove\ x\ dxs = Dlist\ (remove1\ x\ (list-of-dlist\ dxs))$

qualified definition *map* :: ('a \Rightarrow 'b) \Rightarrow 'a dlist \Rightarrow 'b dlist **where**
 $map\ f\ dxs = Dlist\ (remdups\ (List.map\ f\ (list-of-dlist\ dxs)))$

qualified definition *filter* :: ('a \Rightarrow bool) \Rightarrow 'a dlist \Rightarrow 'a dlist **where**
 $filter\ P\ dxs = Dlist\ (List.filter\ P\ (list-of-dlist\ dxs))$

qualified definition *rotate* :: nat \Rightarrow 'a dlist \Rightarrow 'a dlist **where**
 $rotate\ n\ dxs = Dlist\ (List.rotate\ n\ (list-of-dlist\ dxs))$

end

Derived operations:

context
begin

qualified definition *null* :: 'a dlist \Rightarrow bool **where**
 $null\ dxs = List.null\ (list-of-dlist\ dxs)$

qualified definition *member* :: 'a dlist \Rightarrow 'a \Rightarrow bool **where**
 $member\ dxs = List.member\ (list-of-dlist\ dxs)$

qualified definition *length* :: 'a dlist \Rightarrow nat **where**
 $length\ dxs = List.length\ (list-of-dlist\ dxs)$

qualified definition *fold* :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a dlist \Rightarrow 'b \Rightarrow 'b **where**
 $fold\ f\ dxs = List.fold\ f\ (list-of-dlist\ dxs)$

qualified definition $\text{foldr} :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ dlist} \Rightarrow 'b \Rightarrow 'b$ **where**
 $\text{foldr } f \text{ dxs} = \text{List.foldr } f (\text{list-of-dlist dxs})$

end

32.2 Executable version obeying invariant

lemma $\text{list-of-dlist-empty}$ [*simp*, *code abstract*]:

$\text{list-of-dlist } \text{Dlist.empty} = []$

$\langle \text{proof} \rangle$

lemma $\text{list-of-dlist-insert}$ [*simp*, *code abstract*]:

$\text{list-of-dlist } (\text{Dlist.insert } x \text{ dxs}) = \text{List.insert } x (\text{list-of-dlist dxs})$

$\langle \text{proof} \rangle$

lemma $\text{list-of-dlist-remove}$ [*simp*, *code abstract*]:

$\text{list-of-dlist } (\text{Dlist.remove } x \text{ dxs}) = \text{remove1 } x (\text{list-of-dlist dxs})$

$\langle \text{proof} \rangle$

lemma list-of-dlist-map [*simp*, *code abstract*]:

$\text{list-of-dlist } (\text{Dlist.map } f \text{ dxs}) = \text{remdups } (\text{List.map } f (\text{list-of-dlist dxs}))$

$\langle \text{proof} \rangle$

lemma $\text{list-of-dlist-filter}$ [*simp*, *code abstract*]:

$\text{list-of-dlist } (\text{Dlist.filter } P \text{ dxs}) = \text{List.filter } P (\text{list-of-dlist dxs})$

$\langle \text{proof} \rangle$

lemma $\text{list-of-dlist-rotate}$ [*simp*, *code abstract*]:

$\text{list-of-dlist } (\text{Dlist.rotate } n \text{ dxs}) = \text{List.rotate } n (\text{list-of-dlist dxs})$

$\langle \text{proof} \rangle$

Explicit executable conversion

definition dlist-of-list [*simp*]:

$\text{dlist-of-list} = \text{Dlist}$

lemma [*code abstract*]:

$\text{list-of-dlist } (\text{dlist-of-list } xs) = \text{remdups } xs$

$\langle \text{proof} \rangle$

Equality

instantiation $\text{dlist} :: (\text{equal}) \text{ equal}$

begin

definition $\text{HOL.equal } \text{dxs } \text{dys} \longleftrightarrow \text{HOL.equal } (\text{list-of-dlist dxs}) (\text{list-of-dlist dys})$

instance

$\langle \text{proof} \rangle$

end

declare *equal-dlist-def* [*code*]

lemma [*code nbe*]: *HOL.equal* (*dxs* :: 'a::equal dlist) *dxs* \longleftrightarrow *True*
 ⟨*proof*⟩

32.3 Induction principle and case distinction

lemma *dlist-induct* [*case-names empty insert, induct type: dlist*]:
assumes *empty*: *P Dlist.empty*
assumes *insrt*: $\bigwedge x \text{ dxs. } \neg \text{Dlist.member dxs } x \implies P \text{ dxs} \implies P (\text{Dlist.insert } x \text{ dxs})$
shows *P dxs*
 ⟨*proof*⟩

lemma *dlist-case* [*cases type: dlist*]:
obtains (*empty*) *dxs* = *Dlist.empty*
 | (*insert*) *x dys* **where** $\neg \text{Dlist.member dys } x$ **and** *dxs* = *Dlist.insert x dys*
 ⟨*proof*⟩

32.4 Functorial structure

functor *map*: *map*
 ⟨*proof*⟩

32.5 Quickcheck generators

quickcheck-generator *dlist predicate: distinct constructors: Dlist.empty, Dlist.insert*

32.6 BNF instance

context *begin*

qualified inductive *double* :: 'a list \Rightarrow 'a list \Rightarrow bool **where**
double (*xs @ ys*) (*xs @ x # ys*) **if** *x* \in *set ys*

qualified lemma *strong-confluentp-double*: *strong-confluentp double*
 ⟨*proof*⟩ **lemma** *double-Cons1* [*simp*]: *double xs* (*x # xs*) **if** *x* \in *set xs*
 ⟨*proof*⟩ **lemma** *double-Cons-same* [*simp*]: *double xs ys* \implies *double* (*x # xs*) (*x # ys*)
 ⟨*proof*⟩ **lemma** *doubles-Cons-same*: *double** xs ys* \implies *double*** (*x # xs*) (*x # ys*)
 ⟨*proof*⟩ **lemma** *remdups-into-doubles*: *double** (remdups xs) xs*
 ⟨*proof*⟩ **lemma** *dlist-eq-into-doubles*: *Dlist.dlist-eq* \leq *equivclp double*
 ⟨*proof*⟩ **lemma** *factor-double-map*: *double (map f xs) ys* $\implies \exists zs. \text{Dlist.dlist-eq } xs \text{ } zs \wedge ys = \text{map } f \text{ } zs \wedge \text{set } zs \subseteq \text{set } xs$
 ⟨*proof*⟩ **lemma** *dlist-eq-set-eq*: *Dlist.dlist-eq xs ys* $\implies \text{set } xs = \text{set } ys$
 ⟨*proof*⟩ **lemma** *dlist-eq-map-respect*: *Dlist.dlist-eq xs ys* $\implies \text{Dlist.dlist-eq } (\text{map } f \text{ } xs) (\text{map } f \text{ } ys)$
 ⟨*proof*⟩ **lemma** *confluent-quotient-dlist*:

```

    confluent-quotient double Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq
    Dlist.dlist-eq

```

```

    (map fst) (map snd) (map fst) (map snd) list-all2 list-all2 list-all2 set set
    <proof>

```

```

lifting-update dlist.lifting

```

```

lifting-forget dlist.lifting

```

```

end

```

```

context begin

```

```

interpretation Quotient-dlist: Quotient-dlist <proof>

```

```

lift-bnf (plugins del: code) 'a dlist

```

```

    <proof> lemma list-of-dlist-transfer[transfer-rule]:

```

```

    bi-unique R  $\implies$  (rel-fun (Quotient-dlist.pcr-dlist R) (list-all2 R)) remdups list-of-dlist

```

```

    <proof>

```

```

lemma list-of-dlist-map-dlist[simp]:

```

```

    list-of-dlist (map-dlist f xs) = remdups (map f (list-of-dlist xs))

```

```

    <proof>

```

```

end

```

```

end

```

33 Type of dual ordered lattices

```

theory Dual-Ordered-Lattice

```

```

imports Main

```

```

begin

```

The *dual* of an ordered structure is an isomorphic copy of the underlying type, with the \leq relation defined as the inverse of the original one.

The class of lattices is closed under formation of dual structures. This means that for any theorem of lattice theory, the dualized statement holds as well; this important fact simplifies many proofs of lattice theory.

```

typedef 'a dual = UNIV :: 'a set

```

```

    morphisms undual dual <proof>

```

```

setup-lifting type-definition-dual

```

```

code-datatype dual

```

```

lemma dual-eqI:

```

```

    x = y if undual x = undual y

```

```

    <proof>

```

lemma *dual-eq-iff*:

$$x = y \longleftrightarrow \text{undual } x = \text{undual } y$$

<proof>

lemma *eq-dual-iff [iff]*:

$$\text{dual } x = \text{dual } y \longleftrightarrow x = y$$

<proof>

lemma *undual-dual [simp, code]*:

$$\text{undual } (\text{dual } x) = x$$

<proof>

lemma *dual-undual [simp]*:

$$\text{dual } (\text{undual } x) = x$$

<proof>

lemma *undual-comp-dual [simp]*:

$$\text{undual} \circ \text{dual} = \text{id}$$

<proof>

lemma *dual-comp-undual [simp]*:

$$\text{dual} \circ \text{undual} = \text{id}$$

<proof>

lemma *inj-dual*:

$$\text{inj } \text{dual}$$

<proof>

lemma *inj-undual*:

$$\text{inj } \text{undual}$$

<proof>

lemma *surj-dual*:

$$\text{surj } \text{dual}$$

<proof>

lemma *surj-undual*:

$$\text{surj } \text{undual}$$

<proof>

lemma *bij-dual*:

$$\text{bij } \text{dual}$$

<proof>

lemma *bij-undual*:

$$\text{bij } \text{undual}$$

<proof>

instance *dual* :: (*finite*) *finite*
 ⟨*proof*⟩

instantiation *dual* :: (*equal*) *equal*
begin

lift-definition *equal-dual* :: '*a dual* ⇒ '*a dual* ⇒ *bool*
 is *HOL.equal* ⟨*proof*⟩

instance
 ⟨*proof*⟩

end

33.1 Pointwise ordering

instantiation *dual* :: (*ord*) *ord*
begin

lift-definition *less-eq-dual* :: '*a dual* ⇒ '*a dual* ⇒ *bool*
 is (\geq) ⟨*proof*⟩

lift-definition *less-dual* :: '*a dual* ⇒ '*a dual* ⇒ *bool*
 is ($>$) ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *dual-less-eqI*:
 $x \leq y$ **if** *undual* $y \leq$ *undual* x
 ⟨*proof*⟩

lemma *dual-less-eq-iff*:
 $x \leq y \longleftrightarrow$ *undual* $y \leq$ *undual* x
 ⟨*proof*⟩

lemma *less-eq-dual-iff* [*iff*]:
 $dual\ x \leq dual\ y \longleftrightarrow y \leq x$
 ⟨*proof*⟩

lemma *dual-lessI*:
 $x < y$ **if** *undual* $y <$ *undual* x
 ⟨*proof*⟩

lemma *dual-less-iff*:
 $x < y \longleftrightarrow$ *undual* $y <$ *undual* x
 ⟨*proof*⟩

lemma *less-dual-iff* [iff]:
 $dual\ x < dual\ y \longleftrightarrow y < x$
 ⟨proof⟩

instance *dual* :: (preorder) preorder
 ⟨proof⟩

instance *dual* :: (order) order
 ⟨proof⟩

33.2 Binary infimum and supremum

instantiation *dual* :: (sup) inf
begin

lift-definition *inf-dual* :: 'a dual \Rightarrow 'a dual \Rightarrow 'a dual
 is sup ⟨proof⟩

instance ⟨proof⟩

end

lemma *undual-inf-eq* [simp]:
 $undual\ (inf\ x\ y) = sup\ (undual\ x)\ (undual\ y)$
 ⟨proof⟩

lemma *dual-sup-eq* [simp]:
 $dual\ (sup\ x\ y) = inf\ (dual\ x)\ (dual\ y)$
 ⟨proof⟩

instantiation *dual* :: (inf) sup
begin

lift-definition *sup-dual* :: 'a dual \Rightarrow 'a dual \Rightarrow 'a dual
 is inf ⟨proof⟩

instance ⟨proof⟩

end

lemma *undual-sup-eq* [simp]:
 $undual\ (sup\ x\ y) = inf\ (undual\ x)\ (undual\ y)$
 ⟨proof⟩

lemma *dual-inf-eq* [simp]:
 $dual\ (inf\ x\ y) = sup\ (dual\ x)\ (dual\ y)$
 ⟨proof⟩

instance *dual* :: (semilattice-sup) semilattice-inf

<proof>

instance *dual* :: (*semilattice-inf*) *semilattice-sup*
<proof>

instance *dual* :: (*lattice*) *lattice* *<proof>*

instance *dual* :: (*distrib-lattice*) *distrib-lattice*
<proof>

33.3 Top and bottom elements

instantiation *dual* :: (*top*) *bot*
begin

lift-definition *bot-dual* :: 'a *dual*
is *top* *<proof>*

instance *<proof>*

end

lemma *undual-bot-eq* [*simp*]:
undual bot = top
<proof>

lemma *dual-top-eq* [*simp*]:
dual top = bot
<proof>

instantiation *dual* :: (*bot*) *top*
begin

lift-definition *top-dual* :: 'a *dual*
is *bot* *<proof>*

instance *<proof>*

end

lemma *undual-top-eq* [*simp*]:
undual top = bot
<proof>

lemma *dual-bot-eq* [*simp*]:
dual bot = top
<proof>

instance *dual* :: (*order-top*) *order-bot*

$\langle proof \rangle$

instance $dual :: (order-bot) \ order-top$
 $\langle proof \rangle$

instance $dual :: (bounded-lattice-top) \ bounded-lattice-bot \ \langle proof \rangle$

instance $dual :: (bounded-lattice-bot) \ bounded-lattice-top \ \langle proof \rangle$

instance $dual :: (bounded-lattice) \ bounded-lattice \ \langle proof \rangle$

33.4 Complement

instantiation $dual :: (uminus) \ uminus$
begin

lift-definition $uminus-dual :: 'a \ dual \Rightarrow 'a \ dual$
is $uminus \ \langle proof \rangle$

instance $\langle proof \rangle$

end

lemma $undual-uminus-eq \ [simp]:$
 $undual \ (- \ x) = - \ undual \ x$
 $\langle proof \rangle$

lemma $dual-uminus-eq \ [simp]:$
 $dual \ (- \ x) = - \ dual \ x$
 $\langle proof \rangle$

instantiation $dual :: (boolean-algebra) \ boolean-algebra$
begin

lift-definition $minus-dual :: 'a \ dual \Rightarrow 'a \ dual \Rightarrow 'a \ dual$
is $\lambda x \ y. - \ (y - x) \ \langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma $undual-minus-eq \ [simp]:$
 $undual \ (x - y) = - \ (undual \ y - undual \ x)$
 $\langle proof \rangle$

lemma $dual-minus-eq \ [simp]:$
 $dual \ (x - y) = - \ (dual \ y - dual \ x)$
 $\langle proof \rangle$

33.5 Complete lattice operations

The class of complete lattices is closed under formation of dual structures.

instantiation $dual :: (Sup) Inf$
begin

lift-definition $Inf\text{-}dual :: 'a\ dual\ set \Rightarrow 'a\ dual$
is $Sup \langle proof \rangle$

instance $\langle proof \rangle$

end

lemma $undual\text{-}Inf\text{-}eq [simp]:$
 $undual (Inf A) = Sup (undual ' A)$
 $\langle proof \rangle$

lemma $dual\text{-}Sup\text{-}eq [simp]:$
 $dual (Sup A) = Inf (dual ' A)$
 $\langle proof \rangle$

instantiation $dual :: (Inf) Sup$
begin

lift-definition $Sup\text{-}dual :: 'a\ dual\ set \Rightarrow 'a\ dual$
is $Inf \langle proof \rangle$

instance $\langle proof \rangle$

end

lemma $undual\text{-}Sup\text{-}eq [simp]:$
 $undual (Sup A) = Inf (undual ' A)$
 $\langle proof \rangle$

lemma $dual\text{-}Inf\text{-}eq [simp]:$
 $dual (Inf A) = Sup (dual ' A)$
 $\langle proof \rangle$

instance $dual :: (complete\text{-}lattice) complete\text{-}lattice$
 $\langle proof \rangle$

context

fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
and $g :: 'a\ dual \Rightarrow 'a\ dual$
assumes $mono\ f$
defines $g \equiv dual \circ f \circ undual$
begin

private lemma *mono-dual*:

mono g
 $\langle \text{proof} \rangle$

lemma *lfp-dual-gfp*:

lfp f = undual (gfp g) (is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *gfp-dual-lfp*:

gfp f = undual (lfp g)
 $\langle \text{proof} \rangle$

end

Finally

lifting-update *dual.lifting*

lifting-forget *dual.lifting*

end

34 Equipollence and Other Relations Connected with Cardinality

theory *Equipollence*

imports *FuncSet Countable-Set*

begin

34.1 Eqpoll

definition *eqpoll* :: *'a set* \Rightarrow *'b set* \Rightarrow *bool* (**infixl** $\langle \approx \rangle$ 50)
 where *eqpoll* *A B* $\equiv \exists f. \text{bij-betw } f \text{ } A \text{ } B$

definition *lepoll* :: *'a set* \Rightarrow *'b set* \Rightarrow *bool* (**infixl** $\langle \lesssim \rangle$ 50)
 where *lepoll* *A B* $\equiv \exists f. \text{inj-on } f \text{ } A \wedge f \text{ ' } A \subseteq B$

definition *lesspoll* :: *'a set* \Rightarrow *'b set* \Rightarrow *bool* (**infixl** $\langle \prec \rangle$ 50)
 where *A* \prec *B* $\equiv A \lesssim B \wedge \sim(A \approx B)$

lemma *lepoll-def'*: *lepoll* *A B* $\equiv \exists f. \text{inj-on } f \text{ } A \wedge f \in A \rightarrow B$
 $\langle \text{proof} \rangle$

lemma *eqpoll-empty-iff-empty* [*simp*]: *A* $\approx \{\}$ $\longleftrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *lepoll-empty-iff-empty* [*simp*]: *A* $\lesssim \{\}$ $\longleftrightarrow A = \{\}$
 $\langle \text{proof} \rangle$

lemma *not-lesspoll-empty*: $\neg A \prec \{\}$

$\langle proof \rangle$

lemma *lepoll-relational-full*:

assumes $\bigwedge y. y \in B \implies \exists x. x \in A \wedge R\ x\ y$

and $\bigwedge x\ y\ y'. \llbracket x \in A; y \in B; y' \in B; R\ x\ y; R\ x\ y' \rrbracket \implies y = y'$

shows $B \lesssim A$

$\langle proof \rangle$

lemma *eqpoll-iff-card-of-ordIso*: $A \approx B \longleftrightarrow \text{ordIso2 } (\text{card-of } A) (\text{card-of } B)$

$\langle proof \rangle$

lemma *eqpoll-refl [iff]*: $A \approx A$

$\langle proof \rangle$

lemma *eqpoll-finite-iff*: $A \approx B \implies \text{finite } A \longleftrightarrow \text{finite } B$

$\langle proof \rangle$

lemma *eqpoll-iff-card*:

assumes $\text{finite } A\ \text{finite } B$

shows $A \approx B \longleftrightarrow \text{card } A = \text{card } B$

$\langle proof \rangle$

lemma *eqpoll-singleton-iff*: $A \approx \{x\} \longleftrightarrow (\exists u. A = \{u\})$

$\langle proof \rangle$

lemma *eqpoll-doubleton-iff*: $A \approx \{x, y\} \longleftrightarrow (\exists u\ v. A = \{u, v\} \wedge (u=v \longleftrightarrow x=y))$

$\langle proof \rangle$

lemma *lepoll-antisym*:

assumes $A \lesssim B\ B \lesssim A$ **shows** $A \approx B$

$\langle proof \rangle$

lemma *lepoll-trans [trans]*:

assumes $A \lesssim B\ B \lesssim C$ **shows** $A \lesssim C$

$\langle proof \rangle$

lemma *lepoll-trans1 [trans]*: $\llbracket A \approx B; B \lesssim C \rrbracket \implies A \lesssim C$

$\langle proof \rangle$

lemma *lepoll-trans2 [trans]*: $\llbracket A \lesssim B; B \approx C \rrbracket \implies A \lesssim C$

$\langle proof \rangle$

lemma *eqpoll-sym*: $A \approx B \implies B \approx A$

$\langle proof \rangle$

lemma *eqpoll-trans [trans]*: $\llbracket A \approx B; B \approx C \rrbracket \implies A \approx C$

$\langle proof \rangle$

lemma *eqpoll-imp-lepoll*: $A \approx B \implies A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *subset-imp-lepoll*: $A \subseteq B \implies A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *lepoll-refl [iff]*: $A \lesssim A$
 $\langle \text{proof} \rangle$

lemma *lepoll-iff*: $A \lesssim B \longleftrightarrow (\exists g. A \subseteq g \text{ ‘ } B)$
 $\langle \text{proof} \rangle$

lemma *empty-lepoll [iff]*: $\{\} \lesssim A$
 $\langle \text{proof} \rangle$

lemma *subset-image-lepoll*: $B \subseteq f \text{ ‘ } A \implies B \lesssim A$
 $\langle \text{proof} \rangle$

lemma *image-lepoll*: $f \text{ ‘ } A \lesssim A$
 $\langle \text{proof} \rangle$

lemma *infinite-le-lepoll*: $\text{infinite } A \longleftrightarrow (\text{UNIV}::\text{nat set}) \lesssim A$
 $\langle \text{proof} \rangle$

lemma *lepoll-Pow-self*: $A \lesssim \text{Pow } A$
 $\langle \text{proof} \rangle$

lemma *eqpoll-iff-bijections*:
 $A \approx B \longleftrightarrow (\exists f g. (\forall x \in A. f x \in B \wedge g(f x) = x) \wedge (\forall y \in B. g y \in A \wedge f(g y) = y))$
 $\langle \text{proof} \rangle$

lemma *lepoll-restricted-funspace*:
 $\{f. f \text{ ‘ } A \subseteq B \wedge \{x. f x \neq k x\} \subseteq A \wedge \text{finite } \{x. f x \neq k x\}\} \lesssim \text{Fpow } (A \times B)$
 $\langle \text{proof} \rangle$

lemma *singleton-lepoll*: $\{x\} \lesssim \text{insert } y A$
 $\langle \text{proof} \rangle$

lemma *singleton-eqpoll*: $\{x\} \approx \{y\}$
 $\langle \text{proof} \rangle$

lemma *subset-singleton-iff-lepoll*: $(\exists x. S \subseteq \{x\}) \longleftrightarrow S \lesssim \{\}$
 $\langle \text{proof} \rangle$

lemma *infinite-insert-lepoll*:
assumes *infinite* *A* **shows** *insert a A* $\lesssim A$
 $\langle \text{proof} \rangle$

lemma *infinite-insert-epoll*: $\text{infinite } A \implies \text{insert } a \ A \approx A$
 $\langle \text{proof} \rangle$

lemma *finite-lepoll-infinite*:
assumes *infinite A finite B* **shows** $B \lesssim A$
 $\langle \text{proof} \rangle$

lemma *countable-lepoll*: $\llbracket \text{countable } A; B \lesssim A \rrbracket \implies \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *countable-epoll*: $\llbracket \text{countable } A; B \approx A \rrbracket \implies \text{countable } B$
 $\langle \text{proof} \rangle$

34.2 The strict relation

lemma *lesspoll-not-refl* [iff]: $\sim (i \prec i)$
 $\langle \text{proof} \rangle$

lemma *lesspoll-imp-lepoll*: $A \prec B \implies A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *lepoll-iff-leqpoll*: $A \lesssim B \longleftrightarrow A \prec B \mid A \approx B$
 $\langle \text{proof} \rangle$

lemma *lesspoll-trans* [trans]: $\llbracket X \prec Y; Y \prec Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-trans1* [trans]: $\llbracket X \lesssim Y; Y \prec Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-trans2* [trans]: $\llbracket X \prec Y; Y \lesssim Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *eq-lesspoll-trans* [trans]: $\llbracket X \approx Y; Y \prec Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-eq-trans* [trans]: $\llbracket X \prec Y; Y \approx Z \rrbracket \implies X \prec Z$
 $\langle \text{proof} \rangle$

lemma *lesspoll-Pow-self*: $A \prec \text{Pow } A$
 $\langle \text{proof} \rangle$

lemma *finite-lesspoll-infinite*:
assumes *infinite A finite B* **shows** $B \prec A$
 $\langle \text{proof} \rangle$

lemma *countable-lesspoll*: $\llbracket \text{countable } A; B \prec A \rrbracket \implies \text{countable } B$
 $\langle \text{proof} \rangle$

lemma *lepoll-iff-card-le*: $\llbracket \text{finite } A; \text{ finite } B \rrbracket \implies A \lesssim B \longleftrightarrow \text{card } A \leq \text{card } B$
 $\langle \text{proof} \rangle$

lemma *lepoll-iff-finite-card*: $A \lesssim \{..<n::\text{nat}\} \longleftrightarrow \text{finite } A \wedge \text{card } A \leq n$
 $\langle \text{proof} \rangle$

lemma *eqpoll-iff-finite-card*: $A \approx \{..<n::\text{nat}\} \longleftrightarrow \text{finite } A \wedge \text{card } A = n$
 $\langle \text{proof} \rangle$

lemma *lesspoll-iff-finite-card*: $A \prec \{..<n::\text{nat}\} \longleftrightarrow \text{finite } A \wedge \text{card } A < n$
 $\langle \text{proof} \rangle$

34.3 Mapping by an injection

lemma *inj-on-image-eqpoll-self*: $\text{inj-on } f \ A \implies f \, ' \ A \approx A$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-lepoll-1* [simp]:
assumes *inj-on* $f \ A$ **shows** $f \, ' \ A \lesssim B \longleftrightarrow A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-lepoll-2* [simp]:
assumes *inj-on* $f \ B$ **shows** $A \lesssim f \, ' \ B \longleftrightarrow A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-lesspoll-1* [simp]:
assumes *inj-on* $f \ A$ **shows** $f \, ' \ A \prec B \longleftrightarrow A \prec B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-lesspoll-2* [simp]:
assumes *inj-on* $f \ B$ **shows** $A \prec f \, ' \ B \longleftrightarrow A \prec B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-eqpoll-1* [simp]:
assumes *inj-on* $f \ A$ **shows** $f \, ' \ A \approx B \longleftrightarrow A \approx B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-eqpoll-2* [simp]:
assumes *inj-on* $f \ B$ **shows** $A \approx f \, ' \ B \longleftrightarrow A \approx B$
 $\langle \text{proof} \rangle$

34.4 Inserting elements into sets

lemma *insert-lepoll-insertD*:
assumes *insert* $u \ A \lesssim \text{insert } v \ B$ $u \notin A$ $v \notin B$ **shows** $A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *insert-eqpoll-insertD*: $\llbracket \text{insert } u \ A \approx \text{insert } v \ B; u \notin A; v \notin B \rrbracket \implies A \approx B$
 $\langle \text{proof} \rangle$

lemma *insert-lepoll-cong*:

assumes $A \lesssim B$ $b \notin B$ **shows** $\text{insert } a \ A \lesssim \text{insert } b \ B$
 $\langle \text{proof} \rangle$

lemma *insert-epoll-cong*:

$\llbracket A \approx B; a \notin A; b \notin B \rrbracket \implies \text{insert } a \ A \approx \text{insert } b \ B$
 $\langle \text{proof} \rangle$

lemma *insert-epoll-insert-iff*:

$\llbracket a \notin A; b \notin B \rrbracket \implies \text{insert } a \ A \approx \text{insert } b \ B \longleftrightarrow A \approx B$
 $\langle \text{proof} \rangle$

lemma *insert-lepoll-insert-iff*:

$\llbracket a \notin A; b \notin B \rrbracket \implies (\text{insert } a \ A \lesssim \text{insert } b \ B) \longleftrightarrow (A \lesssim B)$
 $\langle \text{proof} \rangle$

lemma *less-imp-insert-lepoll*:

assumes $A \prec B$ **shows** $\text{insert } a \ A \lesssim B$
 $\langle \text{proof} \rangle$

lemma *finite-insert-lepoll*: $\text{finite } A \implies (\text{insert } a \ A \lesssim A) \longleftrightarrow (a \in A)$
 $\langle \text{proof} \rangle$

34.5 Binary sums and unions

lemma *Un-lepoll-mono*:

assumes $A \lesssim C$ $B \lesssim D$ $\text{disjnt } C \ D$ **shows** $A \cup B \lesssim C \cup D$
 $\langle \text{proof} \rangle$

lemma *Un-epoll-cong*: $\llbracket A \approx C; B \approx D; \text{disjnt } A \ B; \text{disjnt } C \ D \rrbracket \implies A \cup B \approx C \cup D$
 $\langle \text{proof} \rangle$

lemma *sum-lepoll-mono*:

assumes $A \lesssim C$ $B \lesssim D$ **shows** $A <+> B \lesssim C <+> D$
 $\langle \text{proof} \rangle$

lemma *sum-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A <+> B \approx C <+> D$
 $\langle \text{proof} \rangle$

34.6 Binary Cartesian products

lemma *times-square-lepoll*: $A \lesssim A \times A$
 $\langle \text{proof} \rangle$

lemma *times-commute-epoll*: $A \times B \approx B \times A$
 $\langle \text{proof} \rangle$

lemma *times-assoc-epoll*: $(A \times B) \times C \approx A \times (B \times C)$
 $\langle \text{proof} \rangle$

lemma *times-singleton-epoll*: $\{a\} \times A \approx A$
 $\langle proof \rangle$

lemma *times-lepoll-mono*:
assumes $A \lesssim C$ $B \lesssim D$ **shows** $A \times B \lesssim C \times D$
 $\langle proof \rangle$

lemma *times-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A \times B \approx C \times D$
 $\langle proof \rangle$

lemma
assumes $B \neq \{\}$ **shows** *lepoll-times1*: $A \lesssim A \times B$ **and** *lepoll-times2*: $A \lesssim B \times A$
 $\langle proof \rangle$

lemma *times-0-epoll*: $\{\} \times A \approx \{\}$
 $\langle proof \rangle$

lemma *Sigma-inj-lepoll-mono*:
assumes h : *inj-on* h A h ‘ $A \subseteq C$ **and** $\bigwedge x. x \in A \implies B\ x \lesssim D\ (h\ x)$
shows $\text{Sigma}\ A\ B \lesssim \text{Sigma}\ C\ D$
 $\langle proof \rangle$

lemma *Sigma-lepoll-mono*:
assumes $A \subseteq C$ $\bigwedge x. x \in A \implies B\ x \lesssim D\ x$ **shows** $\text{Sigma}\ A\ B \lesssim \text{Sigma}\ C\ D$
 $\langle proof \rangle$

lemma *sum-times-distrib-epoll*: $(A <+> B) \times C \approx (A \times C) <+> (B \times C)$
 $\langle proof \rangle$

lemma *Sigma-epoll-cong*:
assumes h : *bij-betw* h A C **and** BD : $\bigwedge x. x \in A \implies B\ x \approx D\ (h\ x)$
shows $\text{Sigma}\ A\ B \approx \text{Sigma}\ C\ D$
 $\langle proof \rangle$

lemma *prod-insert-epoll*:
assumes $a \notin A$ **shows** $\text{insert}\ a\ A \times B \approx B <+> A \times B$
 $\langle proof \rangle$

34.7 General Unions

lemma *Union-epoll-Times*:
assumes B : $\bigwedge x. x \in A \implies F\ x \approx B$ **and** *disj*: *pairwise* $(\lambda x\ y. \text{disjnt}\ (F\ x)\ (F\ y))\ A$
shows $(\bigcup_{x \in A} F\ x) \approx A \times B$
 $\langle proof \rangle$

lemma *UN-lepoll-UN*:

assumes $A: \bigwedge x. x \in A \implies B\ x \lesssim C\ x$
and $disj: pairwise\ (\lambda x\ y. disjnt\ (C\ x)\ (C\ y))\ A$
shows $\bigcup (B'A) \lesssim \bigcup (C'A)$
 $\langle proof \rangle$

lemma *UN-eqpoll-UN*:
assumes $A: \bigwedge x. x \in A \implies B\ x \approx C\ x$
and $B: pairwise\ (\lambda x\ y. disjnt\ (B\ x)\ (B\ y))\ A$
and $C: pairwise\ (\lambda x\ y. disjnt\ (C\ x)\ (C\ y))\ A$
shows $(\bigcup_{x \in A}. B\ x) \approx (\bigcup_{x \in A}. C\ x)$
 $\langle proof \rangle$

34.8 General Cartesian products (Pi)

lemma *PiE-sing-epoll-self*: $(\{a\} \rightarrow_E B) \approx B$
 $\langle proof \rangle$

lemma *lepoll-funcset-right*:
assumes $B \lesssim B'$ **shows** $A \rightarrow_E B \lesssim A \rightarrow_E B'$
 $\langle proof \rangle$

lemma *lepoll-funcset-left*:
assumes $B \neq \{\}$ $A \lesssim A'$
shows $A \rightarrow_E B \lesssim A' \rightarrow_E B$
 $\langle proof \rangle$

lemma *lepoll-funcset*:
 $\llbracket B \neq \{\}; A \lesssim A'; B \lesssim B' \rrbracket \implies A \rightarrow_E B \lesssim A' \rightarrow_E B'$
 $\langle proof \rangle$

lemma *lepoll-PiE*:
assumes $\bigwedge i. i \in A \implies B\ i \lesssim C\ i$
shows $PiE\ A\ B \lesssim PiE\ A\ C$
 $\langle proof \rangle$

lemma *card-le-PiE-subindex*:
assumes $A \subseteq A'\ PiE\ A'\ B \neq \{\}$
shows $PiE\ A\ B \lesssim PiE\ A'\ B$
 $\langle proof \rangle$

lemma *finite-restricted-funspace*:
assumes *finite* A *finite* B
shows *finite* $\{f. f\ ' A \subseteq B \wedge \{x. f\ x \neq k\ x\} \subseteq A\}$ (**is** *finite* ? F)
 $\langle proof \rangle$

proposition *finite-PiE-iff*:

$finite(PiE\ I\ S) \longleftrightarrow PiE\ I\ S = \{\} \vee finite\ \{i \in I. \sim(\exists a. S\ i \subseteq \{a\})\} \wedge (\forall i \in I. finite(S\ i))$
(is ?lhs = ?rhs)
 $\langle proof \rangle$

corollary *finite-funcset-iff:*

$finite(I \rightarrow_E S) \longleftrightarrow (\exists a. S \subseteq \{a\}) \vee I = \{\} \vee finite\ I \wedge finite\ S$
 $\langle proof \rangle$

34.9 Misc other resultd

lemma *lists-lepoll-mono:*

assumes $A \lesssim B$ **shows** $lists\ A \lesssim lists\ B$
 $\langle proof \rangle$

lemma *lepoll-lists:* $A \lesssim lists\ A$

$\langle proof \rangle$

Dedekind’s definition of infinite set

lemma *infinite-iff-psubset:* $infinite\ A \longleftrightarrow (\exists B. B \subset A \wedge A \approx B)$
 $\langle proof \rangle$

lemma *infinite-iff-psubset-le:* $infinite\ A \longleftrightarrow (\exists B. B \subset A \wedge A \lesssim B)$
 $\langle proof \rangle$

end

theory *Simps-Case-Conv*

imports *Case-Converter*

keywords *simps-of-case case-of-simps :: thy-decl*

abbrevs *simps-of-case case-of-simps =*

begin

$\langle ML \rangle$

end

theory *Extended*

imports *Simps-Case-Conv*

begin

datatype *'a extended = Fin 'a | Pinf (ι∞) | Minf (ι−∞)*

instantiation *extended :: (order)order*

begin

```

fun less-eq-extended :: 'a extended  $\Rightarrow$  'a extended  $\Rightarrow$  bool where
  Fin x  $\leq$  Fin y = (x  $\leq$  y) |
  -       $\leq$  Pinf = True |
  Minf  $\leq$  -      = True |
  (-::'a extended)  $\leq$  -      = False

```

case-of-simps less-eq-extended-case: less-eq-extended.simps

```

definition less-extended :: 'a extended  $\Rightarrow$  'a extended  $\Rightarrow$  bool where
  ((x::'a extended) < y) = (x  $\leq$  y  $\wedge$   $\neg$  y  $\leq$  x)

```

```

instance
  <proof>

```

end

```

instance extended :: (linorder)linorder
  <proof>

```

```

lemma Minf-le[simp]: Minf  $\leq$  y
  <proof>

```

```

lemma le-Pinf[simp]: x  $\leq$  Pinf
  <proof>

```

```

lemma le-Minf[simp]: x  $\leq$  Minf  $\longleftrightarrow$  x = Minf
  <proof>

```

```

lemma Pinf-le[simp]: Pinf  $\leq$  x  $\longleftrightarrow$  x = Pinf
  <proof>

```

```

lemma less-extended-simps[simp]:

```

```

  Fin x < Fin y = (x < y)
  Fin x < Pinf = True
  Fin x < Minf = False
  Pinf < h      = False
  Minf < Fin x = True
  Minf < Pinf = True
  l < Minf = False

```

<proof>

```

lemma min-extended-simps[simp]:

```

```

  min (Fin x) (Fin y) = Fin(min x y)
  min xx Pinf = xx
  min xx Minf = Minf
  min Pinf yy = yy
  min Minf yy = Minf

```

<proof>

```

lemma max-extended-simps[simp]:

```

```

  max (Fin x) (Fin y) = Fin(max x y)
  max xx Pinf = Pinf

```

```

    max xx      Minf    = xx
    max Pinf    yy      = Pinf
    max Minf    yy      = yy
  <proof>

```

```

instantiation extended :: (zero)zero
begin
definition 0 = Fin(0::'a)
instance <proof>
end

```

```

declare zero-extended-def[symmetric, code-post]

```

```

instantiation extended :: (one)one
begin
definition 1 = Fin(1::'a)
instance <proof>
end

```

```

declare one-extended-def[symmetric, code-post]

```

```

instantiation extended :: (plus)plus
begin

```

The following definition of addition is totalized to make it associative and commutative. Normally the sum of plus and minus infinity is undefined.

```

fun plus-extended where
  Fin x + Fin y = Fin(x+y) |
  Fin x + Pinf = Pinf |
  Pinf + Fin x = Pinf |
  Pinf + Pinf = Pinf |
  Minf + Fin y = Minf |
  Fin x + Minf = Minf |
  Minf + Minf = Minf |
  Minf + Pinf = Pinf |
  Pinf + Minf = Pinf

```

```

case-of-simps plus-case: plus-extended.simps

```

```

instance <proof>

```

```

end

```

```

instance extended :: (ab-semigroup-add)ab-semigroup-add
  <proof>

```


instance *extended* :: (*ordered-ab-semigroup-add*)*ordered-ab-semigroup-add*
 ⟨*proof*⟩

instance *extended* :: (*comm-monoid-add*)*comm-monoid-add*
 ⟨*proof*⟩

instantiation *extended* :: (*uminus*)*uminus*
begin

fun *uminus-extended* **where**
 – (*Fin* *x*) = *Fin* (– *x*) |
 – *Pinf* = *Minf* |
 – *Minf* = *Pinf*

instance ⟨*proof*⟩

end

instantiation *extended* :: (*ab-group-add*)*minus*
begin

definition $x - y = x + -(y::'a \text{ extended})$

instance ⟨*proof*⟩

end

lemma *minus-extended-simps*[*simp*]:

$Fin\ x - Fin\ y = Fin\ (x - y)$

$Fin\ x - Pinf = Minf$

$Fin\ x - Minf = Pinf$

$Pinf - Fin\ y = Pinf$

$Pinf - Minf = Pinf$

$Minf - Fin\ y = Minf$

$Minf - Pinf = Minf$

$Minf - Minf = Pinf$

$Pinf - Pinf = Pinf$

⟨*proof*⟩

Numerals:

instance *extended* :: ({*ab-semigroup-add,one*})*numeral* ⟨*proof*⟩

lemma *Fin-numeral*[*code-post*]: $Fin(\text{numeral } w) = \text{numeral } w$
 ⟨*proof*⟩

lemma *Fin-neg-numeral*[*code-post*]: $Fin\ (-\ \text{numeral } w) = -\ \text{numeral } w$
 ⟨*proof*⟩

instantiation *extended* :: (*lattice*)*bounded-lattice*
begin

definition *bot* = *Minf*

definition *top* = *Pinf*

fun *inf-extended* :: '*a extended* \Rightarrow '*a extended* \Rightarrow '*a extended* **where**
inf-extended (*Fin i*) (*Fin j*) = *Fin* (*inf i j*) |
inf-extended *a Minf* = *Minf* |
inf-extended *Minf a* = *Minf* |
inf-extended *Pinf a* = *a* |
inf-extended *a Pinf* = *a*

fun *sup-extended* :: '*a extended* \Rightarrow '*a extended* \Rightarrow '*a extended* **where**
sup-extended (*Fin i*) (*Fin j*) = *Fin* (*sup i j*) |
sup-extended *a Pinf* = *Pinf* |
sup-extended *Pinf a* = *Pinf* |
sup-extended *Minf a* = *a* |
sup-extended *a Minf* = *a*

case-of-simps *inf-extended-case*: *inf-extended.simps*

case-of-simps *sup-extended-case*: *sup-extended.simps*

instance

<proof>

end

end

35 Continuity and iterations

theory *Order-Continuity*

imports *Complex-Main Countable-Complete-Lattices*

begin

lemma *SUP-nat-binary*:

$(\text{sup } A (\text{SUP } x \in \text{Collect } ((<) (0::\text{nat})). B)) = (\text{sup } A B::'a::\text{countable-complete-lattice})$

<proof>

lemma *INF-nat-binary*:

$(\text{inf } A (\text{INF } x \in \text{Collect } ((<) (0::\text{nat})). B)) = (\text{inf } A B::'a::\text{countable-complete-lattice})$

<proof>

The name *continuous* is already taken in *Complex-Main*, so we use *sup-continuous* and *inf-continuous*. These names appear sometimes in literature and have the advantage that these names are duals.

named-theorems *order-continuous-intros*

35.1 Continuity for complete lattices

definition

$sup\text{-}continuous :: ('a::countable\text{-}complete\text{-}lattice \Rightarrow 'b::countable\text{-}complete\text{-}lattice) \Rightarrow bool$

where

$sup\text{-}continuous\ F \longleftrightarrow (\forall M::nat \Rightarrow 'a. mono\ M \longrightarrow F\ (SUP\ i. M\ i) = (SUP\ i. F\ (M\ i)))$

lemma $sup\text{-}continuousD$: $sup\text{-}continuous\ F \Longrightarrow mono\ M \Longrightarrow F\ (SUP\ i::nat. M\ i) = (SUP\ i. F\ (M\ i))$

$\langle proof \rangle$

lemma $sup\text{-}continuous\text{-}mono$:

$mono\ F$ **if** $sup\text{-}continuous\ F$

$\langle proof \rangle$

lemma $[order\text{-}continuous\text{-}intros]$:

shows $sup\text{-}continuous\text{-}const$: $sup\text{-}continuous\ (\lambda x. c)$

and $sup\text{-}continuous\text{-}id$: $sup\text{-}continuous\ (\lambda x. x)$

and $sup\text{-}continuous\text{-}apply$: $sup\text{-}continuous\ (\lambda f. f\ x)$

and $sup\text{-}continuous\text{-}fun$: $(\bigwedge s. sup\text{-}continuous\ (\lambda x. P\ x\ s)) \Longrightarrow sup\text{-}continuous$

P

and $sup\text{-}continuous\text{-}If$: $sup\text{-}continuous\ F \Longrightarrow sup\text{-}continuous\ G \Longrightarrow sup\text{-}continuous\ (\lambda f. if\ C\ then\ F\ f\ else\ G\ f)$

$\langle proof \rangle$

lemma $sup\text{-}continuous\text{-}compose$:

assumes f : $sup\text{-}continuous\ f$ **and** g : $sup\text{-}continuous\ g$

shows $sup\text{-}continuous\ (\lambda x. f\ (g\ x))$

$\langle proof \rangle$

lemma $sup\text{-}continuous\text{-}sup[order\text{-}continuous\text{-}intros]$:

$sup\text{-}continuous\ f \Longrightarrow sup\text{-}continuous\ g \Longrightarrow sup\text{-}continuous\ (\lambda x. sup\ (f\ x)\ (g\ x))$

$\langle proof \rangle$

lemma $sup\text{-}continuous\text{-}inf[order\text{-}continuous\text{-}intros]$:

fixes $P\ Q :: 'a :: countable\text{-}complete\text{-}lattice \Rightarrow 'b :: countable\text{-}complete\text{-}distrib\text{-}lattice$

assumes P : $sup\text{-}continuous\ P$ **and** Q : $sup\text{-}continuous\ Q$

shows $sup\text{-}continuous\ (\lambda x. inf\ (P\ x)\ (Q\ x))$

$\langle proof \rangle$

lemma $sup\text{-}continuous\text{-}and[order\text{-}continuous\text{-}intros]$:

$sup\text{-}continuous\ P \Longrightarrow sup\text{-}continuous\ Q \Longrightarrow sup\text{-}continuous\ (\lambda x. P\ x \wedge Q\ x)$

$\langle proof \rangle$

lemma $sup\text{-}continuous\text{-}or[order\text{-}continuous\text{-}intros]$:

$sup\text{-}continuous\ P \Longrightarrow sup\text{-}continuous\ Q \Longrightarrow sup\text{-}continuous\ (\lambda x. P\ x \vee Q\ x)$

$\langle proof \rangle$

lemma *sup-continuous-lfp*:

assumes *sup-continuous* F **shows** $\text{lfp } F = (\text{SUP } i. (F \text{ } \sim i) \text{ bot})$ (**is** $\text{lfp } F = ?U$)
 $\langle \text{proof} \rangle$

lemma *lfp-transfer-bounded*:

assumes $P: P \text{ bot } \wedge x. P x \implies P (f x) \wedge M. (\wedge i. P (M i)) \implies P (\text{SUP } i::\text{nat}. M i)$

assumes $\alpha: \wedge M. \text{mono } M \implies (\wedge i::\text{nat}. P (M i)) \implies \alpha (\text{SUP } i. M i) = (\text{SUP } i. \alpha (M i))$

assumes $f: \text{sup-continuous } f$ **and** $g: \text{sup-continuous } g$

assumes $[\text{simp}]: \wedge x. P x \implies x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)$

assumes $g\text{-bound}: \wedge x. \alpha \text{ bot } \leq g x$

shows $\alpha (\text{lfp } f) = \text{lfp } g$

$\langle \text{proof} \rangle$

lemma *lfp-transfer*:

sup-continuous $\alpha \implies \text{sup-continuous } f \implies \text{sup-continuous } g \implies$

$(\wedge x. \alpha \text{ bot } \leq g x) \implies (\wedge x. x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)) \implies \alpha (\text{lfp } f) =$

$\text{lfp } g$

$\langle \text{proof} \rangle$

definition

inf-continuous :: $('a::\text{countable-complete-lattice} \Rightarrow 'b::\text{countable-complete-lattice}) \Rightarrow \text{bool}$

where

inf-continuous $F \longleftrightarrow (\forall M::\text{nat} \Rightarrow 'a. \text{antimono } M \longrightarrow F (\text{INF } i. M i) = (\text{INF } i. F (M i)))$

lemma *inf-continuousD*: *inf-continuous* $F \implies \text{antimono } M \implies F (\text{INF } i::\text{nat}. M i) = (\text{INF } i. F (M i))$

$\langle \text{proof} \rangle$

lemma *inf-continuous-mono*:

mono F **if** *inf-continuous* F

$\langle \text{proof} \rangle$

lemma *[order-continuous-intros]*:

shows *inf-continuous-const*: *inf-continuous* $(\lambda x. c)$

and *inf-continuous-id*: *inf-continuous* $(\lambda x. x)$

and *inf-continuous-apply*: *inf-continuous* $(\lambda f. f x)$

and *inf-continuous-fun*: $(\wedge s. \text{inf-continuous } (\lambda x. P x s)) \implies \text{inf-continuous } P$

and *inf-continuous-If*: *inf-continuous* $F \implies \text{inf-continuous } G \implies \text{inf-continuous } (\lambda f. \text{if } C \text{ then } F f \text{ else } G f)$

$\langle \text{proof} \rangle$

lemma *inf-continuous-inf*[*order-continuous-intros*]:

inf-continuous $f \implies \text{inf-continuous } g \implies \text{inf-continuous } (\lambda x. \text{inf } (f x) (g x))$

$\langle \text{proof} \rangle$

lemma *inf-continuous-sup*[*order-continuous-intros*]:
fixes $P\ Q :: 'a :: \text{countable-complete-lattice} \Rightarrow 'b :: \text{countable-complete-distrib-lattice}$
assumes $P: \text{inf-continuous } P$ **and** $Q: \text{inf-continuous } Q$
shows $\text{inf-continuous } (\lambda x. \text{sup } (P\ x) (Q\ x))$
 $\langle \text{proof} \rangle$

lemma *inf-continuous-and*[*order-continuous-intros*]:
 $\text{inf-continuous } P \Longrightarrow \text{inf-continuous } Q \Longrightarrow \text{inf-continuous } (\lambda x. P\ x \wedge Q\ x)$
 $\langle \text{proof} \rangle$

lemma *inf-continuous-or*[*order-continuous-intros*]:
 $\text{inf-continuous } P \Longrightarrow \text{inf-continuous } Q \Longrightarrow \text{inf-continuous } (\lambda x. P\ x \vee Q\ x)$
 $\langle \text{proof} \rangle$

lemma *inf-continuous-compose*:
assumes $f: \text{inf-continuous } f$ **and** $g: \text{inf-continuous } g$
shows $\text{inf-continuous } (\lambda x. f\ (g\ x))$
 $\langle \text{proof} \rangle$

lemma *inf-continuous-gfp*:
assumes $\text{inf-continuous } F$ **shows** $\text{gfp } F = (\text{INF } i. (F \rightsquigarrow i)\ \text{top})$ (**is** $\text{gfp } F = ?U$)
 $\langle \text{proof} \rangle$

lemma *gfp-transfer*:
assumes $\alpha: \text{inf-continuous } \alpha$ **and** $f: \text{inf-continuous } f$ **and** $g: \text{inf-continuous } g$
assumes [*simp*]: $\alpha\ \text{top} = \text{top} \wedge x. \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (\text{gfp } f) = \text{gfp } g$
 $\langle \text{proof} \rangle$

lemma *gfp-transfer-bounded*:
assumes $P: P\ (f\ \text{top}) \wedge x. P\ x \Longrightarrow P\ (f\ x) \wedge M. \text{antimono } M \Longrightarrow (\bigwedge i. P\ (M\ i)) \Longrightarrow P\ (\text{INF } i::\text{nat. } M\ i)$
assumes $\alpha: \bigwedge M. \text{antimono } M \Longrightarrow (\bigwedge i::\text{nat. } P\ (M\ i)) \Longrightarrow \alpha\ (\text{INF } i. M\ i) = (\text{INF } i. \alpha\ (M\ i))$
assumes $f: \text{inf-continuous } f$ **and** $g: \text{inf-continuous } g$
assumes [*simp*]: $\bigwedge x. P\ x \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
assumes $g\text{-bound}: \bigwedge x. g\ x \leq \alpha\ (f\ \text{top})$
shows $\alpha\ (\text{gfp } f) = \text{gfp } g$
 $\langle \text{proof} \rangle$

35.1.1 Least fixed points in countable complete lattices

definition (**in** *countable-complete-lattice*) $\text{cclfp} :: ('a \Rightarrow 'a) \Rightarrow 'a$
where $\text{cclfp } f = (\text{SUP } i. (f \rightsquigarrow i)\ \text{bot})$

lemma *cclfp-unfold*:
assumes $\text{sup-continuous } F$ **shows** $\text{cclfp } F = F\ (\text{cclfp } F)$
 $\langle \text{proof} \rangle$

lemma *cclfp-lowerbound*: **assumes** f : *mono f* **and** A : $f\ A \leq A$ **shows** $cclfp\ f \leq A$
 ⟨*proof*⟩

lemma *cclfp-transfer*:
assumes *sup-continuous* α *mono f*
assumes $\alpha\ bot = bot \wedge x. \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (cclfp\ f) = cclfp\ g$
 ⟨*proof*⟩

end

36 Extended natural numbers (i.e. with infinity)

theory *Extended-Nat*
imports *Main Countable Order-Continuity*
begin

class *infinity* =
fixes *infinity* :: 'a ($\iota\infty$)

context
fixes f :: $nat \Rightarrow 'a::\{canonically-ordered-monoid-add, linorder-topology, complete-linorder\}$
begin

lemma *sums-SUP*[*simp, intro*]: $f\ sums\ (SUP\ n. \sum\ i<n. f\ i)$
 ⟨*proof*⟩

lemma *suminf-eq-SUP*: $suminf\ f = (SUP\ n. \sum\ i<n. f\ i)$
 ⟨*proof*⟩

end

36.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *enat* = *UNIV* :: *nat option set* ⟨*proof*⟩

TODO: introduce *enat* as coinductive datatype, *enat* is just *of-nat*

definition *enat* :: $nat \Rightarrow enat$ **where**
 $enat\ n = Abs-enat\ (Some\ n)$

instantiation *enat* :: *infinity*
begin

definition $\infty = Abs-enat\ None$
instance ⟨*proof*⟩

end

instance *enat* :: *countable*
 ⟨*proof*⟩

old-rep-datatype *enat* ∞ :: *enat*
 ⟨*proof*⟩

declare [[*coercion enat::nat⇒enat*]]

lemmas *enat2-cases* = *enat.exhaust*[*case-product enat.exhaust*]

lemmas *enat3-cases* = *enat.exhaust*[*case-product enat.exhaust enat.exhaust*]

lemma *not-infinity-eq* [iff]: $(x \neq \infty) = (\exists i. x = \text{enat } i)$
 ⟨*proof*⟩

lemma *not-enat-eq* [iff]: $(\forall y. x \neq \text{enat } y) = (x = \infty)$
 ⟨*proof*⟩

lemma *enat-ex-split*: $(\exists c::\text{enat}. P \ c) \longleftrightarrow P \ \infty \vee (\exists c::\text{nat}. P \ c)$
 ⟨*proof*⟩

primrec *the-enat* :: *enat* ⇒ *nat*
where *the-enat* (*enat* *n*) = *n*

36.2 Constructors and numbers

instantiation *enat* :: *zero-neq-one*
begin

definition
 $0 = \text{enat } 0$

definition
 $1 = \text{enat } 1$

instance
 ⟨*proof*⟩

end

definition *eSuc* :: *enat* ⇒ *enat* **where**
 $eSuc \ i = (\text{case } i \text{ of } \text{enat } n \Rightarrow \text{enat } (Suc \ n) \mid \infty \Rightarrow \infty)$

lemma *enat-0* [code-post]: *enat* 0 = 0
 ⟨*proof*⟩

lemma *enat-1* [code-post]: *enat* 1 = 1

$\langle proof \rangle$

lemma *enat-0-iff*: $enat\ x = 0 \longleftrightarrow x = 0 \ 0 = enat\ x \longleftrightarrow x = 0$
 $\langle proof \rangle$

lemma *enat-1-iff*: $enat\ x = 1 \longleftrightarrow x = 1 \ 1 = enat\ x \longleftrightarrow x = 1$
 $\langle proof \rangle$

lemma *one-eSuc*: $1 = eSuc\ 0$
 $\langle proof \rangle$

lemma *infinity-ne-i0* [simp]: $(\infty::enat) \neq 0$
 $\langle proof \rangle$

lemma *i0-ne-infinity* [simp]: $0 \neq (\infty::enat)$
 $\langle proof \rangle$

lemma *zero-one-enat-neq*:
 $\neg 0 = (1::enat)$
 $\neg 1 = (0::enat)$
 $\langle proof \rangle$

lemma *infinity-ne-i1* [simp]: $(\infty::enat) \neq 1$
 $\langle proof \rangle$

lemma *i1-ne-infinity* [simp]: $1 \neq (\infty::enat)$
 $\langle proof \rangle$

lemma *eSuc-enat*: $eSuc\ (enat\ n) = enat\ (Suc\ n)$
 $\langle proof \rangle$

lemma *eSuc-infinity* [simp]: $eSuc\ \infty = \infty$
 $\langle proof \rangle$

lemma *eSuc-ne-0* [simp]: $eSuc\ n \neq 0$
 $\langle proof \rangle$

lemma *zero-ne-eSuc* [simp]: $0 \neq eSuc\ n$
 $\langle proof \rangle$

lemma *eSuc-inject* [simp]: $eSuc\ m = eSuc\ n \longleftrightarrow m = n$
 $\langle proof \rangle$

lemma *eSuc-enat-iff*: $eSuc\ x = enat\ y \longleftrightarrow (\exists n. y = Suc\ n \wedge x = enat\ n)$
 $\langle proof \rangle$

lemma *enat-eSuc-iff*: $enat\ y = eSuc\ x \longleftrightarrow (\exists n. y = Suc\ n \wedge enat\ n = x)$
 $\langle proof \rangle$

36.3 Addition

instantiation *enat* :: *comm-monoid-add*
begin

definition *[nitpick-simp]*:

$m + n = (\text{case } m \text{ of } \infty \Rightarrow \infty \mid \text{enat } m \Rightarrow (\text{case } n \text{ of } \infty \Rightarrow \infty \mid \text{enat } n \Rightarrow \text{enat } (m + n)))$

lemma *plus-enat-simps* [*simp*, *code*]:

fixes *q* :: *enat*

shows *enat m + enat n = enat (m + n)*

and $\infty + q = \infty$

and $q + \infty = \infty$

<proof>

instance

<proof>

end

lemma *eSuc-plus-1*:

$e\text{Suc } n = n + 1$

<proof>

lemma *plus-1-eSuc*:

$1 + q = e\text{Suc } q$

$q + 1 = e\text{Suc } q$

<proof>

lemma *iadd-Suc*: $e\text{Suc } m + n = e\text{Suc } (m + n)$

<proof>

lemma *iadd-Suc-right*: $m + e\text{Suc } n = e\text{Suc } (m + n)$

<proof>

36.4 Multiplication

instantiation *enat* :: {*comm-semiring-1*, *semiring-no-zero-divisors*}
begin

definition *times-enat-def* [*nitpick-simp*]:

$m * n = (\text{case } m \text{ of } \infty \Rightarrow \text{if } n = 0 \text{ then } 0 \text{ else } \infty \mid \text{enat } m \Rightarrow (\text{case } n \text{ of } \infty \Rightarrow \text{if } m = 0 \text{ then } 0 \text{ else } \infty \mid \text{enat } n \Rightarrow \text{enat } (m * n)))$

lemma *times-enat-simps* [*simp*, *code*]:

$\text{enat } m * \text{enat } n = \text{enat } (m * n)$

$\infty * \infty = (\infty :: \text{enat})$

$\infty * \text{enat } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \infty)$

$\text{enat } m * \infty = (\text{if } m = 0 \text{ then } 0 \text{ else } \infty)$

$\langle proof \rangle$

instance

$\langle proof \rangle$

end

lemma *mult-eSuc*: $eSuc\ m * n = n + m * n$

$\langle proof \rangle$

lemma *mult-eSuc-right*: $m * eSuc\ n = m + m * n$

$\langle proof \rangle$

lemma *of-nat-eq-enat*: $of\text{-}nat\ n = enat\ n$

$\langle proof \rangle$

instance *enat* :: *semiring-char-0*

$\langle proof \rangle$

lemma *imult-is-infinity*: $((a::enat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$

$\langle proof \rangle$

36.5 Numerals

lemma *numeral-eq-enat*:

$numeral\ k = enat\ (numeral\ k)$

$\langle proof \rangle$

lemma *enat-numeral* [*code-abbrev*]:

$enat\ (numeral\ k) = numeral\ k$

$\langle proof \rangle$

lemma *infinity-ne-numeral* [*simp*]: $(\infty::enat) \neq numeral\ k$

$\langle proof \rangle$

lemma *numeral-ne-infinity* [*simp*]: $numeral\ k \neq (\infty::enat)$

$\langle proof \rangle$

lemma *eSuc-numeral* [*simp*]: $eSuc\ (numeral\ k) = numeral\ (k + Num.One)$

$\langle proof \rangle$

36.6 Subtraction

instantiation *enat* :: *minus*

begin

definition *diff-enat-def*:

$a - b = (case\ a\ of\ (enat\ x) \Rightarrow (case\ b\ of\ (enat\ y) \Rightarrow enat\ (x - y) \mid \infty \Rightarrow 0) \mid \infty \Rightarrow \infty)$

instance $\langle proof \rangle$

end

lemma *idiff-enat-enat* [*simp*, *code*]: $enat\ a - enat\ b = enat\ (a - b)$
 $\langle proof \rangle$

lemma *idiff-infinity* [*simp*, *code*]: $\infty - n = (\infty :: enat)$
 $\langle proof \rangle$

lemma *idiff-infinity-right* [*simp*, *code*]: $enat\ a - \infty = 0$
 $\langle proof \rangle$

lemma *idiff-0* [*simp*]: $(0 :: enat) - n = 0$
 $\langle proof \rangle$

lemmas *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

lemma *idiff-0-right* [*simp*]: $(n :: enat) - 0 = n$
 $\langle proof \rangle$

lemmas *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

lemma *idiff-self* [*simp*]: $n \neq \infty \implies (n :: enat) - n = 0$
 $\langle proof \rangle$

lemma *eSuc-minus-eSuc* [*simp*]: $eSuc\ n - eSuc\ m = n - m$
 $\langle proof \rangle$

lemma *eSuc-minus-1* [*simp*]: $eSuc\ n - 1 = n$
 $\langle proof \rangle$

36.7 Ordering

instantiation *enat* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:
 $m \leq n = (\text{case } n \text{ of } enat\ n1 \Rightarrow (\text{case } m \text{ of } enat\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow False) \mid \infty \Rightarrow True)$

definition [*nitpick-simp*]:
 $m < n = (\text{case } m \text{ of } enat\ m1 \Rightarrow (\text{case } n \text{ of } enat\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow True) \mid \infty \Rightarrow False)$

lemma *enat-ord-simps* [*simp*]:
 $enat\ m \leq enat\ n \iff m \leq n$
 $enat\ m < enat\ n \iff m < n$

$q \leq (\infty::\text{enat})$
 $q < (\infty::\text{enat}) \longleftrightarrow q \neq \infty$
 $(\infty::\text{enat}) \leq q \longleftrightarrow q = \infty$
 $(\infty::\text{enat}) < q \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *numeral-le-enat-iff* [simp]:
shows $\text{numeral } m \leq \text{enat } n \longleftrightarrow \text{numeral } m \leq n$
 $\langle \text{proof} \rangle$

lemma *numeral-less-enat-iff* [simp]:
shows $\text{numeral } m < \text{enat } n \longleftrightarrow \text{numeral } m < n$
 $\langle \text{proof} \rangle$

lemma *enat-ord-code* [code]:
 $\text{enat } m \leq \text{enat } n \longleftrightarrow m \leq n$
 $\text{enat } m < \text{enat } n \longleftrightarrow m < n$
 $q \leq (\infty::\text{enat}) \longleftrightarrow \text{True}$
 $\text{enat } m < \infty \longleftrightarrow \text{True}$
 $\infty \leq \text{enat } n \longleftrightarrow \text{False}$
 $(\infty::\text{enat}) < q \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

instance *enat :: dioid*
 $\langle \text{proof} \rangle$

instance *enat :: {linordered-nonzero-semiring, strict-ordered-comm-monoid-add}*
 $\langle \text{proof} \rangle$

lemma *add-diff-assoc-enat*: $z \leq y \implies x + (y - z) = x + y - (z::\text{enat})$
 $\langle \text{proof} \rangle$

lemma *enat-ord-number* [simp]:
 $(\text{numeral } m :: \text{enat}) \leq \text{numeral } n \longleftrightarrow (\text{numeral } m :: \text{nat}) \leq \text{numeral } n$
 $(\text{numeral } m :: \text{enat}) < \text{numeral } n \longleftrightarrow (\text{numeral } m :: \text{nat}) < \text{numeral } n$
 $\langle \text{proof} \rangle$

lemma *infinity-ileE* [elim!]: $\infty \leq \text{enat } m \implies R$
 $\langle \text{proof} \rangle$

lemma *infinity-ilessE* [elim!]: $\infty < \text{enat } m \implies R$
 $\langle \text{proof} \rangle$

lemma *eSuc-ile-mono* [simp]: $eSuc\ n \leq eSuc\ m \longleftrightarrow n \leq m$
 ⟨proof⟩

lemma *eSuc-mono* [simp]: $eSuc\ n < eSuc\ m \longleftrightarrow n < m$
 ⟨proof⟩

lemma *ile-eSuc* [simp]: $n \leq eSuc\ n$
 ⟨proof⟩

lemma *not-eSuc-ilei0* [simp]: $\neg eSuc\ n \leq 0$
 ⟨proof⟩

lemma *i0-iless-eSuc* [simp]: $0 < eSuc\ n$
 ⟨proof⟩

lemma *iless-eSuc0* [simp]: $(n < eSuc\ 0) = (n = 0)$
 ⟨proof⟩

lemma *ileI1*: $m < n \implies eSuc\ m \leq n$
 ⟨proof⟩

lemma *Suc-ile-eq*: $enat\ (Suc\ m) \leq n \longleftrightarrow enat\ m < n$
 ⟨proof⟩

lemma *iless-Suc-eq* [simp]: $enat\ m < eSuc\ n \longleftrightarrow enat\ m \leq n$
 ⟨proof⟩

lemma *imult-infinity*: $(0::enat) < n \implies \infty * n = \infty$
 ⟨proof⟩

lemma *imult-infinity-right*: $(0::enat) < n \implies n * \infty = \infty$
 ⟨proof⟩

lemma *enat-0-less-mult-iff*: $(0 < (m::enat) * n) = (0 < m \wedge 0 < n)$
 ⟨proof⟩

lemma *mono-eSuc*: *mono* *eSuc*
 ⟨proof⟩

lemma *min-enat-simps* [simp]:
 $\min\ (enat\ m)\ (enat\ n) = enat\ (\min\ m\ n)$
 $\min\ q\ 0 = 0$
 $\min\ 0\ q = 0$
 $\min\ q\ (\infty::enat) = q$
 $\min\ (\infty::enat)\ q = q$
 ⟨proof⟩

lemma *max-enat-simps* [simp]:

$\text{max } (\text{enat } m) (\text{enat } n) = \text{enat } (\text{max } m \ n)$
 $\text{max } q \ 0 = q$
 $\text{max } 0 \ q = q$
 $\text{max } q \ \infty = (\infty :: \text{enat})$
 $\text{max } \infty \ q = (\infty :: \text{enat})$
 $\langle \text{proof} \rangle$

lemma *enat-ile*: $n \leq \text{enat } m \implies \exists k. n = \text{enat } k$
 $\langle \text{proof} \rangle$

lemma *enat-iless*: $n < \text{enat } m \implies \exists k. n = \text{enat } k$
 $\langle \text{proof} \rangle$

lemma *iadd-le-enat-iff*:
 $x + y \leq \text{enat } n \iff (\exists y' \ x'. x = \text{enat } x' \wedge y = \text{enat } y' \wedge x' + y' \leq n)$
 $\langle \text{proof} \rangle$

lemma *chain-incr*: $\forall i. \exists j. Y \ i < Y \ j \implies \exists j. \text{enat } k < Y \ j$
 $\langle \text{proof} \rangle$

lemma *eSuc-max*: $e\text{Suc } (\text{max } x \ y) = \text{max } (e\text{Suc } x) (e\text{Suc } y)$
 $\langle \text{proof} \rangle$

lemma *eSuc-Max*:
assumes *finite A* $A \neq \{\}$
shows $e\text{Suc } (\text{Max } A) = \text{Max } (e\text{Suc } ` A)$
 $\langle \text{proof} \rangle$

instantiation *enat* :: $\{\text{order-bot}, \text{order-top}\}$
begin

definition *bot-enat* :: *enat* **where** *bot-enat* = 0
definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
 $\langle \text{proof} \rangle$

end

lemma *finite-enat-bounded*:
assumes *le-fin*: $\bigwedge y. y \in A \implies y \leq \text{enat } n$
shows *finite A*
 $\langle \text{proof} \rangle$

36.8 Cancellation simprocs

lemma *add-diff-cancel-enat[simp]*: $x \neq \infty \implies x + y - x = (y :: \text{enat})$
 $\langle \text{proof} \rangle$

lemma *enat-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::\text{enat}) \vee b = c$
 $\langle \text{proof} \rangle$

lemma *enat-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::\text{enat}) \vee b \leq c$
 $\langle \text{proof} \rangle$

lemma *enat-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::\text{enat}) \wedge b < c$
 $\langle \text{proof} \rangle$

lemma *plus-eq-infty-iff-enat*: $(m::\text{enat}) + n = \infty \longleftrightarrow m = \infty \vee n = \infty$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

36.9 Well-ordering

lemma *less-enatE*:
 $\llbracket n < \text{enat } m; \bigwedge k. \llbracket n = \text{enat } k; k < m \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *less-infinityE*:
 $\llbracket n < \infty; \bigwedge k. n = \text{enat } k \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *enat-less-induct*:
assumes $\bigwedge n. \forall m::\text{enat}. m < n \longrightarrow P \ m \implies P \ n$
shows $P \ n$
 $\langle \text{proof} \rangle$

instance *enat :: wellorder*
 $\langle \text{proof} \rangle$

36.10 Complete Lattice

instantiation *enat :: complete-lattice*
begin

definition *inf-enat* :: $\text{enat} \Rightarrow \text{enat} \Rightarrow \text{enat}$ **where**
 $\text{inf-enat} = \text{min}$

definition *sup-enat* :: $\text{enat} \Rightarrow \text{enat} \Rightarrow \text{enat}$ **where**
 $\text{sup-enat} = \text{max}$

definition *Inf-enat* :: $\text{enat set} \Rightarrow \text{enat}$ **where**
 $\text{Inf-enat } A = (\text{if } A = \{\} \text{ then } \infty \text{ else } (\text{LEAST } x. x \in A))$

definition *Sup-enat* :: *enat set* \Rightarrow *enat* **where**

Sup-enat A = (if *A* = {} then 0 else if finite *A* then Max *A* else ∞)

instance

<proof>

end

instance *enat* :: *complete-linorder* *<proof>*

lemma *eSuc-Sup*: $A \neq \{\} \implies eSuc (Sup A) = Sup (eSuc ` A)$

<proof>

lemma *sup-continuous-eSuc*: *sup-continuous f* \implies *sup-continuous* ($\lambda x. eSuc (f x)$)

<proof>

36.11 Traditional theorem names

lemmas *enat-defs* = *zero-enat-def one-enat-def eSuc-def*

plus-enat-def less-eq-enat-def less-enat-def

lemma *iadd-is-0*: $(m + n = (0::enat)) = (m = 0 \wedge n = 0)$

<proof>

lemma *i0-lb* : $(0::enat) \leq n$

<proof>

lemma *ile0-eq*: $n \leq (0::enat) \longleftrightarrow n = 0$

<proof>

lemma *not-iless0*: $\neg n < (0::enat)$

<proof>

lemma *i0-less[simp]*: $(0::enat) < n \longleftrightarrow n \neq 0$

<proof>

lemma *imult-is-0*: $((m::enat) * n = 0) = (m = 0 \vee n = 0)$

<proof>

end

37 Liminf and Limsup on conditionally complete lattices

theory *Liminf-Limsup*

imports *Complex-Main*

begin

lemma (in *conditionally-complete-linorder*) *le-cSup-iff*:

assumes $A \neq \{\}$ *bdd-above* A

shows $x \leq \text{Sup } A \longleftrightarrow (\forall y < x. \exists a \in A. y < a)$

<proof>

lemma (in *conditionally-complete-linorder*) *le-cSUP-iff*:

$A \neq \{\} \implies \text{bdd-above } (f' A) \implies x \leq \text{Sup } (f' A) \longleftrightarrow (\forall y < x. \exists i \in A. y < f i)$

<proof>

lemma *le-cSup-iff-less*:

fixes $x :: 'a :: \{\text{conditionally-complete-linorder}, \text{dense-linorder}\}$

shows $A \neq \{\} \implies \text{bdd-above } (f' A) \implies x \leq (\text{SUP } i \in A. f i) \longleftrightarrow (\forall y < x. \exists i \in A. y \leq f i)$

<proof>

lemma *le-Sup-iff-less*:

fixes $x :: 'a :: \{\text{complete-linorder}, \text{dense-linorder}\}$

shows $x \leq (\text{SUP } i \in A. f i) \longleftrightarrow (\forall y < x. \exists i \in A. y \leq f i)$ (is ?lhs = ?rhs)

<proof>

lemma (in *conditionally-complete-linorder*) *cInf-le-iff*:

assumes $A \neq \{\}$ *bdd-below* A

shows $\text{Inf } A \leq x \longleftrightarrow (\forall y > x. \exists a \in A. y > a)$

<proof>

lemma (in *conditionally-complete-linorder*) *cINF-le-iff*:

$A \neq \{\} \implies \text{bdd-below } (f' A) \implies \text{Inf } (f' A) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. y > f i)$

<proof>

lemma *cInf-le-iff-less*:

fixes $x :: 'a :: \{\text{conditionally-complete-linorder}, \text{dense-linorder}\}$

shows $A \neq \{\} \implies \text{bdd-below } (f' A) \implies (\text{INF } i \in A. f i) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. f i \leq y)$

<proof>

lemma *Inf-le-iff-less*:

fixes $x :: 'a :: \{\text{complete-linorder}, \text{dense-linorder}\}$

shows $(\text{INF } i \in A. f i) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. f i \leq y)$

<proof>

lemma *SUP-pair*:

fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

shows $(\text{SUP } i \in A. \text{SUP } j \in B. f i j) = (\text{SUP } p \in A \times B. f (\text{fst } p) (\text{snd } p))$

<proof>

lemma *INF-pair*:

fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

shows $(\text{INF } i \in A. \text{INF } j \in B. f i j) = (\text{INF } p \in A \times B. f (\text{fst } p) (\text{snd } p))$

<proof>

lemma *INF-Sigma*:

fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

shows $(\text{INF } i \in A. \text{ INF } j \in B \ i. f \ i \ j) = (\text{INF } p \in \text{Sigma } A \ B. f \ (\text{fst } p) \ (\text{snd } p))$
 $\langle \text{proof} \rangle$

37.0.1 *Liminf and Limsup*

definition *Liminf* :: $'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$ **where**

$\text{Liminf } F \ f = (\text{SUP } P \in \{P. \text{eventually } P \ F\}. \text{ INF } x \in \{x. P \ x\}. f \ x)$

definition *Limsup* :: $'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$ **where**

$\text{Limsup } F \ f = (\text{INF } P \in \{P. \text{eventually } P \ F\}. \text{ SUP } x \in \{x. P \ x\}. f \ x)$

abbreviation *liminf* $\equiv \text{Liminf sequentially}$

abbreviation *limsup* $\equiv \text{Limsup sequentially}$

lemma *Liminf-eqI*:

$(\bigwedge P. \text{eventually } P \ F \Longrightarrow \text{Inf } (f \ ' (\text{Collect } P)) \leq x) \Longrightarrow$
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \Longrightarrow \text{Inf } (f \ ' (\text{Collect } P)) \leq y) \Longrightarrow x \leq y) \Longrightarrow \text{Liminf}$
 $F \ f = x$
 $\langle \text{proof} \rangle$

lemma *Limsup-eqI*:

$(\bigwedge P. \text{eventually } P \ F \Longrightarrow x \leq \text{Sup } (f \ ' (\text{Collect } P))) \Longrightarrow$
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \Longrightarrow y \leq \text{Sup } (f \ ' (\text{Collect } P))) \Longrightarrow y \leq x) \Longrightarrow$
 $\text{Limsup } F \ f = x$
 $\langle \text{proof} \rangle$

lemma *liminf-SUP-INF*: $\text{liminf } f = (\text{SUP } n. \text{ INF } m \in \{n..\}. f \ m)$

$\langle \text{proof} \rangle$

lemma *limsup-INF-SUP*: $\text{limsup } f = (\text{INF } n. \text{ SUP } m \in \{n..\}. f \ m)$

$\langle \text{proof} \rangle$

lemma *mem-limsup-iff*: $x \in \text{limsup } A \longleftrightarrow (\exists_F n \text{ in sequentially. } x \in A \ n)$

$\langle \text{proof} \rangle$

lemma *mem-liminf-iff*: $x \in \text{liminf } A \longleftrightarrow (\forall_F n \text{ in sequentially. } x \in A \ n)$

$\langle \text{proof} \rangle$

lemma *Limsup-const*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Limsup } F \ (\lambda x. c) = c$

$\langle \text{proof} \rangle$

lemma *Liminf-const*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Liminf } F (\lambda x. c) = c$
 $\langle \text{proof} \rangle$

lemma *Liminf-mono*:
assumes $ev: \text{eventually } (\lambda x. f\ x \leq g\ x)\ F$
shows $\text{Liminf } F\ f \leq \text{Liminf } F\ g$
 $\langle \text{proof} \rangle$

lemma *Liminf-eq*:
assumes $\text{eventually } (\lambda x. f\ x = g\ x)\ F$
shows $\text{Liminf } F\ f = \text{Liminf } F\ g$
 $\langle \text{proof} \rangle$

lemma *Limsup-mono*:
assumes $ev: \text{eventually } (\lambda x. f\ x \leq g\ x)\ F$
shows $\text{Limsup } F\ f \leq \text{Limsup } F\ g$
 $\langle \text{proof} \rangle$

lemma *Limsup-eq*:
assumes $\text{eventually } (\lambda x. f\ x = g\ x)\ \text{net}$
shows $\text{Limsup } \text{net}\ f = \text{Limsup } \text{net}\ g$
 $\langle \text{proof} \rangle$

lemma *Liminf-bot[simp]*: $\text{Liminf } \text{bot}\ f = \text{top}$
 $\langle \text{proof} \rangle$

lemma *Limsup-bot[simp]*: $\text{Limsup } \text{bot}\ f = \text{bot}$
 $\langle \text{proof} \rangle$

lemma *Liminf-le-Limsup*:
assumes $\text{ntriv}: \neg \text{trivial-limit } F$
shows $\text{Liminf } F\ f \leq \text{Limsup } F\ f$
 $\langle \text{proof} \rangle$

lemma *Liminf-bounded*:
assumes $le: \text{eventually } (\lambda n. C \leq X\ n)\ F$
shows $C \leq \text{Liminf } F\ X$
 $\langle \text{proof} \rangle$

lemma *Limsup-bounded*:
assumes $le: \text{eventually } (\lambda n. X\ n \leq C)\ F$
shows $\text{Limsup } F\ X \leq C$
 $\langle \text{proof} \rangle$

lemma *le-Limsup*:
assumes $F: F \neq \text{bot}$ **and** $x: \forall_F x \text{ in } F. l \leq f\ x$
shows $l \leq \text{Limsup } F\ f$
 $\langle \text{proof} \rangle$

lemma *Liminf-le*:

assumes $F: F \neq \text{bot}$ **and** $x: \forall_F x \text{ in } F. f x \leq l$

shows $\text{Liminf } F f \leq l$

<proof>

lemma *le-Liminf-iff*:

fixes $X :: - \Rightarrow - :: \text{complete-linorder}$

shows $C \leq \text{Liminf } F X \longleftrightarrow (\forall y < C. \text{eventually } (\lambda x. y < X x) F)$

<proof>

lemma *Limsup-le-iff*:

fixes $X :: - \Rightarrow - :: \text{complete-linorder}$

shows $C \geq \text{Limsup } F X \longleftrightarrow (\forall y > C. \text{eventually } (\lambda x. y > X x) F)$

<proof>

lemma *less-LiminfD*:

$y < \text{Liminf } F (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f x > y) F$

<proof>

lemma *Limsup-lessD*:

$y > \text{Limsup } F (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f x < y) F$

<proof>

lemma *lim-imp-Liminf*:

fixes $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{ntriv}: \neg \text{trivial-limit } F$

assumes $\text{lim}: (f \longrightarrow f0) F$

shows $\text{Liminf } F f = f0$

<proof>

lemma *lim-imp-Limsup*:

fixes $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{ntriv}: \neg \text{trivial-limit } F$

assumes $\text{lim}: (f \longrightarrow f0) F$

shows $\text{Limsup } F f = f0$

<proof>

lemma *Liminf-eq-Limsup*:

fixes $f0 :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{ntriv}: \neg \text{trivial-limit } F$

and $\text{lim}: \text{Liminf } F f = f0 \text{ Limsup } F f = f0$

shows $(f \longrightarrow f0) F$

<proof>

lemma *tendsto-iff-Liminf-eq-Limsup*:

fixes $f0 :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

shows $\neg \text{trivial-limit } F \implies (f \longrightarrow f0) F \longleftrightarrow (\text{Liminf } F f = f0 \wedge \text{Limsup } F f = f0)$

<proof>

lemma *liminf-subseq-mono*:

fixes $X :: \text{nat} \Rightarrow 'a :: \text{complete-linorder}$

assumes *strict-mono* r

shows $\text{liminf } X \leq \text{liminf } (X \circ r)$

<proof>

lemma *limsup-subseq-mono*:

fixes $X :: \text{nat} \Rightarrow 'a :: \text{complete-linorder}$

assumes *strict-mono* r

shows $\text{limsup } (X \circ r) \leq \text{limsup } X$

<proof>

lemma *continuous-on-imp-continuous-within*:

continuous-on $s \ f \Longrightarrow t \subseteq s \Longrightarrow x \in s \Longrightarrow \text{continuous } (\text{at } x \text{ within } t) \ f$

<proof>

lemma *Liminf-compose-continuous-mono*:

fixes $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes c : *continuous-on UNIV* f **and** am : *mono* f **and** F : $F \neq \text{bot}$

shows $\text{Liminf } F \ (\lambda n. f \ (g \ n)) = f \ (\text{Liminf } F \ g)$

<proof>

lemma *Limsup-compose-continuous-mono*:

fixes $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes c : *continuous-on UNIV* f **and** am : *mono* f **and** F : $F \neq \text{bot}$

shows $\text{Limsup } F \ (\lambda n. f \ (g \ n)) = f \ (\text{Limsup } F \ g)$

<proof>

lemma *Liminf-compose-continuous-antimono*:

fixes $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes c : *continuous-on UNIV* f

and am : *antimono* f

and F : $F \neq \text{bot}$

shows $\text{Liminf } F \ (\lambda n. f \ (g \ n)) = f \ (\text{Limsup } F \ g)$

<proof>

lemma *Limsup-compose-continuous-antimono*:

fixes $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

assumes c : *continuous-on UNIV* f **and** am : *antimono* f **and** F : $F \neq \text{bot}$

shows $\text{Limsup } F \ (\lambda n. f \ (g \ n)) = f \ (\text{Liminf } F \ g)$

<proof>

lemma *Liminf-filtermap-le*: $\text{Liminf } (\text{filtermap } f \ F) \ g \leq \text{Liminf } F \ (\lambda x. g \ (f \ x))$

<proof>

lemma *Limsup-filtermap-ge*: $\text{Limsup } (\text{filtermap } f \ F) \ g \geq \text{Limsup } F \ (\lambda x. g \ (f \ x))$

<proof>

lemma *Liminf-least*: $(\bigwedge P. \text{eventually } P \ F \implies (\inf_{x \in \text{Collect } P} f \ x) \leq x) \implies \text{Liminf } F \ f \leq x$
 ⟨proof⟩

lemma *Limsup-greatest*: $(\bigwedge P. \text{eventually } P \ F \implies x \leq (\sup_{x \in \text{Collect } P} f \ x)) \implies \text{Limsup } F \ f \geq x$
 ⟨proof⟩

lemma *Liminf-filtermap-ge*: $\text{inj } f \implies \text{Liminf } (\text{filtermap } f \ F) \ g \geq \text{Liminf } F \ (\lambda x. g \ (f \ x))$
 ⟨proof⟩

lemma *Limsup-filtermap-le*: $\text{inj } f \implies \text{Limsup } (\text{filtermap } f \ F) \ g \leq \text{Limsup } F \ (\lambda x. g \ (f \ x))$
 ⟨proof⟩

lemma *Liminf-filtermap-eq*: $\text{inj } f \implies \text{Liminf } (\text{filtermap } f \ F) \ g = \text{Liminf } F \ (\lambda x. g \ (f \ x))$
 ⟨proof⟩

lemma *Limsup-filtermap-eq*: $\text{inj } f \implies \text{Limsup } (\text{filtermap } f \ F) \ g = \text{Limsup } F \ (\lambda x. g \ (f \ x))$
 ⟨proof⟩

37.1 More Limits

lemma *convergent-limsup-cl*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
 shows $\text{convergent } X \implies \text{limsup } X = \lim X$
 ⟨proof⟩

lemma *convergent-liminf-cl*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
 shows $\text{convergent } X \implies \text{liminf } X = \lim X$
 ⟨proof⟩

lemma *lim-increasing-cl*:
 assumes $\bigwedge n \ m. n \geq m \implies f \ n \geq f \ m$
 obtains l where $f \longrightarrow (l :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\})$
 ⟨proof⟩

lemma *lim-decreasing-cl*:
 assumes $\bigwedge n \ m. n \geq m \implies f \ n \leq f \ m$
 obtains l where $f \longrightarrow (l :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\})$
 ⟨proof⟩

lemma *compact-complete-linorder*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
 shows $\exists l \ r. \text{strict-mono } r \wedge (X \circ r) \longrightarrow l$

$\langle \text{proof} \rangle$

lemma *tendsto-Limsup*:

fixes $f :: - \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

shows $F \neq \text{bot} \implies \text{Limsup } F f = \text{Liminf } F f \implies (f \longrightarrow \text{Limsup } F f) F$

$\langle \text{proof} \rangle$

lemma *tendsto-Liminf*:

fixes $f :: - \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

shows $F \neq \text{bot} \implies \text{Limsup } F f = \text{Liminf } F f \implies (f \longrightarrow \text{Liminf } F f) F$

$\langle \text{proof} \rangle$

end

38 Extended real number line

theory *Extended-Real*

imports *Complex-Main Extended-Nat Liminf-Limsup*

begin

This should be part of *HOL-Library.Extended-Nat* or *HOL-Library.Order-Continuity*, but then the AFP-entry *Jinja-Thread* fails, as it does overload certain named from *Complex-Main*.

lemma *incseq-sumI2*:

fixes $f :: 'i \Rightarrow \text{nat} \Rightarrow 'a::\text{ordered-comm-monoid-add}$

shows $(\bigwedge n. n \in A \implies \text{mono } (f n)) \implies \text{mono } (\lambda i. \sum_{n \in A} f n i)$

$\langle \text{proof} \rangle$

lemma *incseq-sumI*:

fixes $f :: \text{nat} \Rightarrow 'a::\text{ordered-comm-monoid-add}$

assumes $\bigwedge i. 0 \leq f i$

shows $\text{incseq } (\lambda i. \text{sum } f \{..< i\})$

$\langle \text{proof} \rangle$

lemma *continuous-at-left-imp-sup-continuous*:

fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b::\{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $\text{mono } f \wedge x. \text{continuous } (\text{at-left } x) f$

shows $\text{sup-continuous } f$

$\langle \text{proof} \rangle$

lemma *sup-continuous-at-left*:

fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}, \text{first-countable-topology}\} \Rightarrow$

$'b::\{\text{complete-linorder}, \text{linorder-topology}\}$

assumes $f: \text{sup-continuous } f$

shows $\text{continuous } (\text{at-left } x) f$

$\langle \text{proof} \rangle$

lemma *sup-continuous-iff-at-left*:

fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}, \text{first-countable-topology}\} \Rightarrow$

$'b::\{\text{complete-linorder}, \text{linorder-topology}\}$
shows $\text{sup-continuous } f \longleftrightarrow (\forall x. \text{continuous } (\text{at-left } x) f) \wedge \text{mono } f$
 $\langle \text{proof} \rangle$

lemma *continuous-at-right-imp-inf-continuous*:
fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b::\{\text{complete-linorder}, \text{linorder-topology}\}$
assumes $\text{mono } f \wedge x. \text{continuous } (\text{at-right } x) f$
shows $\text{inf-continuous } f$
 $\langle \text{proof} \rangle$

lemma *inf-continuous-at-right*:
fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}, \text{first-countable-topology}\} \Rightarrow$
 $'b::\{\text{complete-linorder}, \text{linorder-topology}\}$
assumes $f: \text{inf-continuous } f$
shows $\text{continuous } (\text{at-right } x) f$
 $\langle \text{proof} \rangle$

lemma *inf-continuous-iff-at-right*:
fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}, \text{first-countable-topology}\} \Rightarrow$
 $'b::\{\text{complete-linorder}, \text{linorder-topology}\}$
shows $\text{inf-continuous } f \longleftrightarrow (\forall x. \text{continuous } (\text{at-right } x) f) \wedge \text{mono } f$
 $\langle \text{proof} \rangle$

instantiation $\text{enat} :: \text{linorder-topology}$
begin

definition $\text{open-enat} :: \text{enat set} \Rightarrow \text{bool}$ **where**
 $\text{open-enat} = \text{generate-topology } (\text{range lessThan} \cup \text{range greaterThan})$

instance
 $\langle \text{proof} \rangle$

end

lemma *open-enat*: $\text{open } \{\text{enat } n\}$
 $\langle \text{proof} \rangle$

lemma *open-enat-iff*:
fixes $A :: \text{enat set}$
shows $\text{open } A \longleftrightarrow (\infty \in A \longrightarrow (\exists n::\text{nat}. \{n <..\} \subseteq A))$
 $\langle \text{proof} \rangle$

lemma *nhds-enat*: $\text{nhds } x = (\text{if } x = \infty \text{ then } \text{INF } i. \text{principal } \{\text{enat } i..\} \text{ else } \text{principal } \{x\})$
 $\langle \text{proof} \rangle$

instance $\text{enat} :: \text{topological-comm-monoid-add}$
 $\langle \text{proof} \rangle$

For more lemmas about the extended real numbers see `~~/src/HOL/`

Analysis/Extended_Real_Limits.thy.

38.1 Definition and basic properties

datatype *ereal* = *ereal real* | *PInfy* | *MInfy*

instantiation *ereal* :: *uminus*
begin

fun *uminus-ereal* **where**
 – (*ereal* *r*) = *ereal* (– *r*)
 | – *PInfy* = *MInfy*
 | – *MInfy* = *PInfy*

instance $\langle proof \rangle$

end

instantiation *ereal* :: *infinity*
begin

definition ($\infty :: \text{ereal}$) = *PInfy*
instance $\langle proof \rangle$

end

declare $[[\text{coercion } \text{ereal} :: \text{real} \Rightarrow \text{ereal}]]$

lemma *ereal-uminus-uminus[simp]*:
fixes *a* :: *ereal*
shows – (– *a*) = *a*
 $\langle proof \rangle$

lemma
shows *PInfy-eq-infinity[simp]*: *PInfy* = ∞
and *MInfy-eq-minfinity[simp]*: *MInfy* = $-\infty$
and *MInfy-neq-PInfy[simp]*: $\infty \neq -(\infty :: \text{ereal})$ $-\infty \neq (\infty :: \text{ereal})$
and *MInfy-neq-ereal[simp]*: *ereal* *r* $\neq -\infty$ $-\infty \neq \text{ereal } r$
and *PInfy-neq-ereal[simp]*: *ereal* *r* $\neq \infty$ $\infty \neq \text{ereal } r$
and *PInfy-cases[simp]*: (case ∞ of *ereal* *r* \Rightarrow *f r* | *PInfy* \Rightarrow *y* | *MInfy* \Rightarrow *z*)
 = *y*
and *MInfy-cases[simp]*: (case $-\infty$ of *ereal* *r* \Rightarrow *f r* | *PInfy* \Rightarrow *y* | *MInfy* \Rightarrow *z*)
 = *z*
 $\langle proof \rangle$

declare
PInfy-eq-infinity[code-post]
MInfy-eq-minfinity[code-post]

lemma *[code-unfold]*:

$\infty = PInfty$
 $- PInfty = MInfty$
 $\langle proof \rangle$

lemma *inj-ereal[simp]*: *inj-on ereal A*
 $\langle proof \rangle$

lemma *ereal-cases[cases type: ereal]*:
obtains *(real) r where* $x = \text{ereal } r$
 $| (PInf) \ x = \infty$
 $| (MInf) \ x = -\infty$
 $\langle proof \rangle$

lemmas *ereal2-cases* = *ereal-cases[case-product ereal-cases]*
lemmas *ereal3-cases* = *ereal2-cases[case-product ereal-cases]*

lemma *ereal-all-split*: $\bigwedge P. (\forall x::\text{ereal}. P \ x) \longleftrightarrow P \ \infty \wedge (\forall x. P \ (\text{ereal } x)) \wedge P \ (-\infty)$
 $\langle proof \rangle$

lemma *ereal-ex-split*: $\bigwedge P. (\exists x::\text{ereal}. P \ x) \longleftrightarrow P \ \infty \vee (\exists x. P \ (\text{ereal } x)) \vee P \ (-\infty)$
 $\langle proof \rangle$

lemma *ereal-uminus-eq-iff[simp]*:
fixes $a \ b :: \text{ereal}$
shows $-a = -b \longleftrightarrow a = b$
 $\langle proof \rangle$

function *real-of-ereal* :: *ereal* \Rightarrow *real* **where**
 $\text{real-of-ereal } (\text{ereal } r) = r$
 $| \text{real-of-ereal } \infty = 0$
 $| \text{real-of-ereal } (-\infty) = 0$
 $\langle proof \rangle$
termination $\langle proof \rangle$

lemma *real-of-ereal[simp]*:
 $\text{real-of-ereal } (-x :: \text{ereal}) = -(\text{real-of-ereal } x)$
 $\langle proof \rangle$

lemma *range-ereal[simp]*: $\text{range } \text{ereal} = \text{UNIV} - \{\infty, -\infty\}$
 $\langle proof \rangle$

lemma *ereal-range-uminus[simp]*: $\text{range } \text{uminus} = (\text{UNIV}::\text{ereal set})$
 $\langle proof \rangle$

instantiation *ereal* :: *abs*
begin

function *abs-ereal* **where**

$|ereal\ r| = ereal\ |r|$
 $|-\infty| = (\infty::ereal)$
 $|\infty| = (\infty::ereal)$
 $\langle proof \rangle$
termination $\langle proof \rangle$

instance $\langle proof \rangle$

end

lemma *abs-eq-infinity-cases*[*elim!*]:

fixes $x :: ereal$
 assumes $|x| = \infty$
 obtains $x = \infty \mid x = -\infty$
 $\langle proof \rangle$

lemma *abs-neq-infinity-cases*[*elim!*]:

fixes $x :: ereal$
 assumes $|x| \neq \infty$
 obtains r **where** $x = ereal\ r$
 $\langle proof \rangle$

lemma *abs-ereal-uminus*[*simp*]:

fixes $x :: ereal$
 shows $|-x| = |x|$
 $\langle proof \rangle$

lemma *ereal-infinity-cases*:

fixes $a :: ereal$
 shows $a \neq \infty \implies a \neq -\infty \implies |a| \neq \infty$
 $\langle proof \rangle$

38.1.1 Addition

instantiation *ereal* :: $\{one, comm-monoid-add, zero-neq-one\}$
begin

definition $0 = ereal\ 0$

definition $1 = ereal\ 1$

function *plus-ereal* **where**

$ereal\ r + ereal\ p = ereal\ (r + p)$
 $|\infty + a = (\infty::ereal)$
 $|a + \infty = (\infty::ereal)$
 $|ereal\ r + -\infty = -\infty$
 $|- \infty + ereal\ p = -(\infty::ereal)$
 $|- \infty + -\infty = -(\infty::ereal)$

$\langle proof \rangle$

termination $\langle proof \rangle$

lemma *Infty-neq-0*[simp]:

$(\infty :: ereal) \neq 0 \quad 0 \neq (\infty :: ereal)$

$-(\infty :: ereal) \neq 0 \quad 0 \neq -(\infty :: ereal)$

$\langle proof \rangle$

lemma *ereal-eq-0*[simp]:

$ereal\ r = 0 \longleftrightarrow r = 0$

$0 = ereal\ r \longleftrightarrow r = 0$

$\langle proof \rangle$

lemma *ereal-eq-1*[simp]:

$ereal\ r = 1 \longleftrightarrow r = 1$

$1 = ereal\ r \longleftrightarrow r = 1$

$\langle proof \rangle$

instance

$\langle proof \rangle$

end

lemma *ereal-0-plus* [simp]: $ereal\ 0 + x = x$

and *plus-ereal-0* [simp]: $x + ereal\ 0 = x$

$\langle proof \rangle$

instance *ereal* :: *numeral* $\langle proof \rangle$

lemma *real-of-ereal-0*[simp]: $real\ of\ ereal\ (0 :: ereal) = 0$

$\langle proof \rangle$

lemma *abs-ereal-zero*[simp]: $|0| = (0 :: ereal)$

$\langle proof \rangle$

lemma *ereal-uminus-zero*[simp]: $- 0 = (0 :: ereal)$

$\langle proof \rangle$

lemma *ereal-uminus-zero-iff*[simp]:

fixes $a :: ereal$

shows $-a = 0 \longleftrightarrow a = 0$

$\langle proof \rangle$

lemma *ereal-plus-eq-PIInfty*[simp]:

fixes $a\ b :: ereal$

shows $a + b = \infty \longleftrightarrow a = \infty \vee b = \infty$

$\langle proof \rangle$

lemma *ereal-plus-eq-MInfty*[simp]:

fixes $a\ b :: \text{ereal}$
shows $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$
 $\langle \text{proof} \rangle$

lemma *ereal-add-cancel-left*:
fixes $a\ b :: \text{ereal}$
assumes $a \neq -\infty$
shows $a + b = a + c \longleftrightarrow a = \infty \vee b = c$
 $\langle \text{proof} \rangle$

lemma *ereal-add-cancel-right*:
fixes $a\ b :: \text{ereal}$
assumes $a \neq -\infty$
shows $b + a = c + a \longleftrightarrow a = \infty \vee b = c$
 $\langle \text{proof} \rangle$

lemma *ereal-real*: $\text{ereal} (\text{real-of-ereal } x) = (\text{if } |x| = \infty \text{ then } 0 \text{ else } x)$
 $\langle \text{proof} \rangle$

lemma *real-of-ereal-add*:
fixes $a\ b :: \text{ereal}$
shows $\text{real-of-ereal } (a + b) =$
 $(\text{if } (|a| = \infty) \wedge (|b| = \infty) \vee (|a| \neq \infty) \wedge (|b| \neq \infty) \text{ then } \text{real-of-ereal } a +$
 $\text{real-of-ereal } b \text{ else } 0)$
 $\langle \text{proof} \rangle$

38.1.2 Linear order on *ereal*

instantiation $\text{ereal} :: \text{linorder}$
begin

function *less-ereal*
where
 $\text{ereal } x < \text{ereal } y \longleftrightarrow x < y$
 $| (\infty :: \text{ereal}) < a \longleftrightarrow \text{False}$
 $| a < -(\infty :: \text{ereal}) \longleftrightarrow \text{False}$
 $| \text{ereal } x < \infty \longleftrightarrow \text{True}$
 $| -\infty < \text{ereal } r \longleftrightarrow \text{True}$
 $| -\infty < (\infty :: \text{ereal}) \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

definition $x \leq (y :: \text{ereal}) \longleftrightarrow x < y \vee x = y$

lemma *ereal-infity-less[simp]*:
fixes $x :: \text{ereal}$
shows $x < \infty \longleftrightarrow (x \neq \infty)$
 $-\infty < x \longleftrightarrow (x \neq -\infty)$
 $\langle \text{proof} \rangle$

lemma *ereal-infity-less-eq[simp]*:
fixes $x :: \text{ereal}$
shows $\infty \leq x \longleftrightarrow x = \infty$
and $x \leq -\infty \longleftrightarrow x = -\infty$
 $\langle \text{proof} \rangle$

lemma *ereal-less[simp]*:
 $\text{ereal } r < 0 \longleftrightarrow (r < 0)$
 $0 < \text{ereal } r \longleftrightarrow (0 < r)$
 $\text{ereal } r < 1 \longleftrightarrow (r < 1)$
 $1 < \text{ereal } r \longleftrightarrow (1 < r)$
 $0 < (\infty :: \text{ereal})$
 $-(\infty :: \text{ereal}) < 0$
 $\langle \text{proof} \rangle$

lemma *ereal-less-eq[simp]*:
 $x \leq (\infty :: \text{ereal})$
 $-(\infty :: \text{ereal}) \leq x$
 $\text{ereal } r \leq \text{ereal } p \longleftrightarrow r \leq p$
 $\text{ereal } r \leq 0 \longleftrightarrow r \leq 0$
 $0 \leq \text{ereal } r \longleftrightarrow 0 \leq r$
 $\text{ereal } r \leq 1 \longleftrightarrow r \leq 1$
 $1 \leq \text{ereal } r \longleftrightarrow 1 \leq r$
 $\langle \text{proof} \rangle$

lemma *ereal-infity-less-eq2*:
 $a \leq b \implies a = \infty \implies b = (\infty :: \text{ereal})$
 $a \leq b \implies b = -\infty \implies a = -(\infty :: \text{ereal})$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *ereal-dense2*: $x < y \implies \exists z. x < \text{ereal } z \wedge \text{ereal } z < y$
 $\langle \text{proof} \rangle$

instance *ereal :: dense-linorder*
 $\langle \text{proof} \rangle$

instance *ereal :: ordered-comm-monoid-add*
 $\langle \text{proof} \rangle$

lemma *ereal-one-not-less-zero-ereal[simp]*: $\neg 1 < (0 :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *real-of-ereal-positive-mono*:

fixes $x\ y :: \text{ereal}$
shows $0 \leq x \implies x \leq y \implies y \neq \infty \implies \text{real-of-ereal } x \leq \text{real-of-ereal } y$
 $\langle \text{proof} \rangle$

lemma *ereal-MInfty-lessI*[*intro, simp*]:
fixes $a :: \text{ereal}$
shows $a \neq -\infty \implies -\infty < a$
 $\langle \text{proof} \rangle$

lemma *ereal-less-PInfty*[*intro, simp*]:
fixes $a :: \text{ereal}$
shows $a \neq \infty \implies a < \infty$
 $\langle \text{proof} \rangle$

lemma *ereal-less-ereal-Ex*:
fixes $a\ b :: \text{ereal}$
shows $x < \text{ereal } r \longleftrightarrow x = -\infty \vee (\exists p. p < r \wedge x = \text{ereal } p)$
 $\langle \text{proof} \rangle$

lemma *less-PInf-Ex-of-nat*: $x \neq \infty \longleftrightarrow (\exists n::\text{nat}. x < \text{ereal } (\text{real } n))$
 $\langle \text{proof} \rangle$

lemma *ereal-add-strict-mono2*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $a < b$ **and** $c < d$
shows $a + c < b + d$
 $\langle \text{proof} \rangle$

lemma *ereal-minus-le-minus*[*simp*]:
fixes $a\ b :: \text{ereal}$
shows $-a \leq -b \longleftrightarrow b \leq a$
 $\langle \text{proof} \rangle$

lemma *ereal-minus-less-minus*[*simp*]:
fixes $a\ b :: \text{ereal}$
shows $-a < -b \longleftrightarrow b < a$
 $\langle \text{proof} \rangle$

lemma *ereal-le-real-iff*:
 $x \leq \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$
 $\langle \text{proof} \rangle$

lemma *real-le-ereal-iff*:
 $\text{real-of-ereal } y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$
 $\langle \text{proof} \rangle$

lemma *ereal-less-real-iff*:
 $x < \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x < y) \wedge (|y| = \infty \longrightarrow x < 0)$
 $\langle \text{proof} \rangle$

lemma *real-less-ereal-iff*:

real-of-ereal $y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y < \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 < x)$
 ⟨proof⟩

To help with inferences like $\llbracket a < \text{ereal } x; x < y \rrbracket \Longrightarrow a < \text{ereal } y$, where x and y are real.

lemma *le-ereal-le*: $a \leq \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a \leq \text{ereal } y$
 ⟨proof⟩

lemma *le-ereal-less*: $a \leq \text{ereal } x \Longrightarrow x < y \Longrightarrow a < \text{ereal } y$
 ⟨proof⟩

lemma *less-ereal-le*: $a < \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a < \text{ereal } y$
 ⟨proof⟩

lemma *ereal-le-le*: $\text{ereal } y \leq a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x \leq a$
 ⟨proof⟩

lemma *ereal-le-less*: $\text{ereal } y \leq a \Longrightarrow x < y \Longrightarrow \text{ereal } x < a$
 ⟨proof⟩

lemma *ereal-less-le*: $\text{ereal } y < a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x < a$
 ⟨proof⟩

lemma *real-of-ereal-pos*:

fixes $x :: \text{ereal}$

shows $0 \leq x \Longrightarrow 0 \leq \text{real-of-ereal } x$

⟨proof⟩

lemmas *real-of-ereal-ord-simps* =

ereal-le-real-iff *real-le-ereal-iff* *ereal-less-real-iff* *real-less-ereal-iff*

lemma *abs-ereal-ge0[simp]*: $0 \leq x \Longrightarrow |x :: \text{ereal}| = x$
 ⟨proof⟩

lemma *abs-ereal-less0[simp]*: $x < 0 \Longrightarrow |x :: \text{ereal}| = -x$
 ⟨proof⟩

lemma *abs-ereal-pos[simp]*: $0 \leq |x :: \text{ereal}|$
 ⟨proof⟩

lemma *ereal-abs-leI*:

fixes $x y :: \text{ereal}$

shows $\llbracket x \leq y; -x \leq y \rrbracket \Longrightarrow |x| \leq y$

⟨proof⟩

lemma *ereal-abs-add*:

fixes $a b :: \text{ereal}$

shows $\text{abs}(a+b) \leq \text{abs } a + \text{abs } b$
 $\langle \text{proof} \rangle$

lemma *real-of-ereal-le-0[simp]*: $\text{real-of-ereal } (x :: \text{ereal}) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$
 $\langle \text{proof} \rangle$

lemma *abs-real-of-ereal[simp]*: $|\text{real-of-ereal } (x :: \text{ereal})| = \text{real-of-ereal } |x|$
 $\langle \text{proof} \rangle$

lemma *zero-less-real-of-ereal*:
fixes $x :: \text{ereal}$
shows $0 < \text{real-of-ereal } x \longleftrightarrow 0 < x \wedge x \neq \infty$
 $\langle \text{proof} \rangle$

lemma *ereal-0-le-uminus-iff[simp]*:
fixes $a :: \text{ereal}$
shows $0 \leq -a \longleftrightarrow a \leq 0$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-le-0-iff[simp]*:
fixes $a :: \text{ereal}$
shows $-a \leq 0 \longleftrightarrow 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *ereal-add-strict-mono*:
fixes $a \ b \ c \ d :: \text{ereal}$
assumes $a \leq b$
and $0 \leq a$
and $a \neq \infty$
and $c < d$
shows $a + c < b + d$
 $\langle \text{proof} \rangle$

lemma *ereal-less-add*:
fixes $a \ b \ c :: \text{ereal}$
shows $|a| \neq \infty \implies c < b \implies a + c < a + b$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-eq-reorder*: $-a = b \longleftrightarrow a = (-b :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-less-reorder*: $-a < b \longleftrightarrow -b < a$
and *ereal-less-uminus-reorder*: $a < -b \longleftrightarrow b < -a$
and *ereal-uminus-le-reorder*: $-a \leq b \longleftrightarrow -b \leq a$ **for** $a :: \text{ereal}$
 $\langle \text{proof} \rangle$

lemmas *ereal-uminus-reorder* =
ereal-uminus-eq-reorder *ereal-uminus-less-reorder* *ereal-uminus-le-reorder*

lemma *ereal-bot*:

fixes $x :: \text{ereal}$

assumes $\bigwedge B. x \leq \text{ereal } B$

shows $x = -\infty$

$\langle \text{proof} \rangle$

lemma *ereal-top*:

fixes $x :: \text{ereal}$

assumes $\bigwedge B. x \geq \text{ereal } B$

shows $x = \infty$

$\langle \text{proof} \rangle$

lemma

shows *ereal-max[simp]*: $\text{ereal } (\max x y) = \max (\text{ereal } x) (\text{ereal } y)$

and *ereal-min[simp]*: $\text{ereal } (\min x y) = \min (\text{ereal } x) (\text{ereal } y)$

$\langle \text{proof} \rangle$

lemma *ereal-max-0*: $\max 0 (\text{ereal } r) = \text{ereal } (\max 0 r)$

$\langle \text{proof} \rangle$

lemma

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

shows *ereal-incseq-uminus[simp]*: $\text{incseq } (\lambda x. - f x) \longleftrightarrow \text{decseq } f$

and *ereal-decseq-uminus[simp]*: $\text{decseq } (\lambda x. - f x) \longleftrightarrow \text{incseq } f$

$\langle \text{proof} \rangle$

lemma *incseq-ereal*: $\text{incseq } f \Longrightarrow \text{incseq } (\lambda x. \text{ereal } (f x))$

$\langle \text{proof} \rangle$

lemma *sum-ereal[simp]*: $(\sum x \in A. \text{ereal } (f x)) = \text{ereal } (\sum x \in A. f x)$

$\langle \text{proof} \rangle$

lemma *sum-list-ereal [simp]*: $\text{sum-list } (\text{map } (\lambda x. \text{ereal } (f x)) xs) = \text{ereal } (\text{sum-list } (\text{map } f xs))$

$\langle \text{proof} \rangle$

lemma *sum-Pinfity*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $(\sum x \in P. f x) = \infty \longleftrightarrow \text{finite } P \wedge (\exists i \in P. f i = \infty)$

$\langle \text{proof} \rangle$

lemma *sum-Inf*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $|\text{sum } f A| = \infty \longleftrightarrow \text{finite } A \wedge (\exists i \in A. |f i| = \infty)$

$\langle \text{proof} \rangle$

lemma *sum-real-of-ereal*:

fixes $f :: 'i \Rightarrow \text{ereal}$

assumes $\bigwedge x. x \in S \Longrightarrow |f x| \neq \infty$

shows $(\sum_{x \in S}. \text{real-of-ereal } (f\ x)) = \text{real-of-ereal } (\text{sum } f\ S)$
 $\langle \text{proof} \rangle$

38.1.3 Multiplication

instantiation *ereal* :: {comm-monoid-mult,sgn}
begin

function *sgn-ereal* :: *ereal* \Rightarrow *ereal* **where**

sgn (*ereal* *r*) = *ereal* (*sgn* *r*)

| *sgn* ($\infty::\text{ereal}$) = 1

| *sgn* ($-\infty::\text{ereal}$) = -1

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

function *times-ereal* **where**

ereal *r* * *ereal* *p* = *ereal* (*r* * *p*)

| *ereal* *r* * ∞ = (if *r* = 0 then 0 else if *r* > 0 then ∞ else $-\infty$)

| ∞ * *ereal* *r* = (if *r* = 0 then 0 else if *r* > 0 then ∞ else $-\infty$)

| *ereal* *r* * $-\infty$ = (if *r* = 0 then 0 else if *r* > 0 then $-\infty$ else ∞)

| $-\infty$ * *ereal* *r* = (if *r* = 0 then 0 else if *r* > 0 then $-\infty$ else ∞)

| ($\infty::\text{ereal}$) * ∞ = ∞

| $-(\infty::\text{ereal})$ * ∞ = $-\infty$

| ($\infty::\text{ereal}$) * $-\infty$ = $-\infty$

| $-(\infty::\text{ereal})$ * $-\infty$ = ∞

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

instance

$\langle \text{proof} \rangle$

end

lemma [*simp*]:

shows *ereal-1-times*: *ereal* 1 * *x* = *x*

and *times-ereal-1*: *x* * *ereal* 1 = *x*

$\langle \text{proof} \rangle$

lemma *one-not-le-zero-ereal*[*simp*]: $\neg (1 \leq (0::\text{ereal}))$

$\langle \text{proof} \rangle$

lemma *real-ereal-1*[*simp*]: *real-of-ereal* (1::*ereal*) = 1

$\langle \text{proof} \rangle$

lemma *real-of-ereal-le-1*:

fixes *a* :: *ereal*

shows $a \leq 1 \implies \text{real-of-ereal } a \leq 1$

$\langle \text{proof} \rangle$

lemma *abs-ereal-one*[simp]: $|1| = (1::ereal)$
 $\langle proof \rangle$

lemma *ereal-mult-zero*[simp]:
fixes $a :: ereal$
shows $a * 0 = 0$
 $\langle proof \rangle$

lemma *ereal-zero-mult*[simp]:
fixes $a :: ereal$
shows $0 * a = 0$
 $\langle proof \rangle$

lemma *ereal-m1-less-0*[simp]: $-(1::ereal) < 0$
 $\langle proof \rangle$

lemma *ereal-times*[simp]:
 $1 \neq (\infty::ereal) \quad (\infty::ereal) \neq 1$
 $1 \neq -(\infty::ereal) \quad -(\infty::ereal) \neq 1$
 $\langle proof \rangle$

lemma *ereal-plus-1*[simp]:
 $1 + ereal\ r = ereal\ (r + 1)$
 $ereal\ r + 1 = ereal\ (r + 1)$
 $1 + -(\infty::ereal) = -\infty$
 $-(\infty::ereal) + 1 = -\infty$
 $\langle proof \rangle$

lemma *ereal-zero-times*[simp]:
fixes $a\ b :: ereal$
shows $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$
 $\langle proof \rangle$

lemma *ereal-mult-eq-PInfty*[simp]:
 $a * b = (\infty::ereal) \longleftrightarrow$
 $(a = \infty \wedge b > 0) \vee (a > 0 \wedge b = \infty) \vee (a = -\infty \wedge b < 0) \vee (a < 0 \wedge b =$
 $-\infty)$
 $\langle proof \rangle$

lemma *ereal-mult-eq-MInfty*[simp]:
 $a * b = -(\infty::ereal) \longleftrightarrow$
 $(a = \infty \wedge b < 0) \vee (a < 0 \wedge b = \infty) \vee (a = -\infty \wedge b > 0) \vee (a > 0 \wedge b =$
 $-\infty)$
 $\langle proof \rangle$

lemma *ereal-abs-mult*: $|x * y :: ereal| = |x| * |y|$
 $\langle proof \rangle$

lemma *ereal-0-less-1*[simp]: $0 < (1::ereal)$

$\langle \text{proof} \rangle$

lemma *ereal-mult-minus-left[simp]*:

fixes $a\ b :: \text{ereal}$

shows $-a * b = -(a * b)$

$\langle \text{proof} \rangle$

lemma *ereal-mult-minus-right[simp]*:

fixes $a\ b :: \text{ereal}$

shows $a * -b = -(a * b)$

$\langle \text{proof} \rangle$

lemma *ereal-mult-infity[simp]*:

$a * (\infty :: \text{ereal}) = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$

$\langle \text{proof} \rangle$

lemma *ereal-infity-mult[simp]*:

$(\infty :: \text{ereal}) * a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$

$\langle \text{proof} \rangle$

lemma *ereal-mult-strict-right-mono*:

assumes $a < b$

and $0 < c$

and $c < (\infty :: \text{ereal})$

shows $a * c < b * c$

$\langle \text{proof} \rangle$

lemma *ereal-mult-strict-left-mono*:

$a < b \implies 0 < c \implies c < (\infty :: \text{ereal}) \implies c * a < c * b$

$\langle \text{proof} \rangle$

lemma *ereal-mult-right-mono*:

fixes $a\ b\ c :: \text{ereal}$

assumes $a \leq b$ $0 \leq c$

shows $a * c \leq b * c$

$\langle \text{proof} \rangle$

lemma *ereal-mult-left-mono*:

fixes $a\ b\ c :: \text{ereal}$

shows $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

$\langle \text{proof} \rangle$

lemma *ereal-mult-mono*:

fixes $a\ b\ c\ d :: \text{ereal}$

assumes $b \geq 0$ $c \geq 0$ $a \leq b$ $c \leq d$

shows $a * c \leq b * d$

$\langle \text{proof} \rangle$

lemma *ereal-mult-mono'*:

fixes $a\ b\ c\ d :: \text{ereal}$
assumes $a \geq 0\ c \geq 0\ a \leq b\ c \leq d$
shows $a * c \leq b * d$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-mono-strict*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $b > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-mono-strict'*:
fixes $a\ b\ c\ d :: \text{ereal}$
assumes $a > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
 $\langle \text{proof} \rangle$

lemma *zero-less-one-ereal[simp]*: $0 \leq (1 :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-0-le-mult[simp]*: $0 \leq a \implies 0 \leq b \implies 0 \leq a * (b :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-right-distrib*:
fixes $r\ a\ b :: \text{ereal}$
shows $0 \leq a \implies 0 \leq b \implies r * (a + b) = r * a + r * b$
 $\langle \text{proof} \rangle$

lemma *ereal-left-distrib*:
fixes $r\ a\ b :: \text{ereal}$
shows $0 \leq a \implies 0 \leq b \implies (a + b) * r = a * r + b * r$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-le-0-iff*:
fixes $a\ b :: \text{ereal}$
shows $a * b \leq 0 \longleftrightarrow (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$
 $\langle \text{proof} \rangle$

lemma *ereal-zero-le-0-iff*:
fixes $a\ b :: \text{ereal}$
shows $0 \leq a * b \longleftrightarrow (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-less-0-iff*:
fixes $a\ b :: \text{ereal}$
shows $a * b < 0 \longleftrightarrow (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$
 $\langle \text{proof} \rangle$

lemma *ereal-zero-less-0-iff*:

fixes $a\ b :: \text{ereal}$
shows $0 < a * b \longleftrightarrow (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$
 $\langle \text{proof} \rangle$

lemma *ereal-left-mult-cong*:
fixes $a\ b\ c :: \text{ereal}$
shows $c = d \implies (d \neq 0 \implies a = b) \implies a * c = b * d$
 $\langle \text{proof} \rangle$

lemma *ereal-right-mult-cong*:
fixes $a\ b\ c :: \text{ereal}$
shows $c = d \implies (d \neq 0 \implies a = b) \implies c * a = d * b$
 $\langle \text{proof} \rangle$

lemma *ereal-distrib*:
fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$
and $|c| \neq \infty$
shows $(a + b) * c = a * c + b * c$
 $\langle \text{proof} \rangle$

lemma *numeral-eq-ereal* [simp]: $\text{numeral } w = \text{ereal } (\text{numeral } w)$
 $\langle \text{proof} \rangle$

lemma *m1-ereal-less-iff* [simp]:
 $((-1::\text{ereal}) < \text{numeral } a) \longleftrightarrow ((-1::\text{real}) < \text{numeral } a)$
 $\langle \text{proof} \rangle$

lemma *m1-ereal-le-iff* [simp]:
 $((-1::\text{ereal}) \leq \text{numeral } a) \longleftrightarrow ((-1::\text{real}) \leq \text{numeral } a)$
 $\langle \text{proof} \rangle$

lemma *m1-ereal-eq-iff* [simp]:
 $((-1::\text{ereal}) = \text{numeral } a) \longleftrightarrow ((-1::\text{real}) = \text{numeral } a)$
 $\langle \text{proof} \rangle$

lemma *ereal-less-m1-iff* [simp]:
 $(\text{numeral } a < (-1::\text{ereal})) \longleftrightarrow (\text{numeral } a < (-1::\text{real}))$
 $\langle \text{proof} \rangle$

lemma *ereal-le-m1-iff* [simp]:
 $(\text{numeral } a \leq (-1::\text{ereal})) \longleftrightarrow (\text{numeral } a \leq (-1::\text{real}))$
 $\langle \text{proof} \rangle$

lemma *ereal-eq-m1-iff* [simp]:
 $(\text{numeral } a = (-1::\text{ereal})) \longleftrightarrow (\text{numeral } a = (-1::\text{real}))$
 $\langle \text{proof} \rangle$

lemma *distrib-left-ereal-nn*:

$c \geq 0 \implies (x + y) * \text{ereal } c = x * \text{ereal } c + y * \text{ereal } c$

$\langle \text{proof} \rangle$

lemma *sum-ereal-right-distrib*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $(\bigwedge i. i \in A \implies 0 \leq f i) \implies r * \text{sum } f A = (\sum n \in A. r * f n)$

$\langle \text{proof} \rangle$

lemma *sum-ereal-left-distrib*:

$(\bigwedge i. i \in A \implies 0 \leq f i) \implies \text{sum } f A * r = (\sum n \in A. f n * r :: \text{ereal})$

$\langle \text{proof} \rangle$

lemma *sum-distrib-right-ereal*:

$c \geq 0 \implies \text{sum } f A * \text{ereal } c = (\sum x \in A. f x * c :: \text{ereal})$

$\langle \text{proof} \rangle$

lemma *ereal-le-epsilon*:

fixes $x y :: \text{ereal}$

assumes $\bigwedge e. 0 < e \implies x \leq y + e$

shows $x \leq y$

$\langle \text{proof} \rangle$

lemma *ereal-le-epsilon2*:

fixes $x y :: \text{ereal}$

assumes $\bigwedge e :: \text{real}. 0 < e \implies x \leq y + \text{ereal } e$

shows $x \leq y$

$\langle \text{proof} \rangle$

lemma *ereal-le-real*:

fixes $x y :: \text{ereal}$

assumes $\bigwedge z. x \leq \text{ereal } z \implies y \leq \text{ereal } z$

shows $y \leq x$

$\langle \text{proof} \rangle$

lemma *prod-ereal-0*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $(\prod i \in A. f i) = 0 \iff \text{finite } A \wedge (\exists i \in A. f i = 0)$

$\langle \text{proof} \rangle$

lemma *prod-ereal-pos*:

fixes $f :: 'a \Rightarrow \text{ereal}$

assumes $\bigwedge i. i \in I \implies 0 \leq f i$

shows $0 \leq (\prod i \in I. f i)$

$\langle \text{proof} \rangle$

lemma *prod-PInf*:

fixes $f :: 'a \Rightarrow \text{ereal}$

assumes $\bigwedge i. i \in I \implies 0 \leq f i$

shows $(\prod_{i \in I}. f\ i) = \infty \longleftrightarrow \text{finite } I \wedge (\exists i \in I. f\ i = \infty) \wedge (\forall i \in I. f\ i \neq 0)$
 $\langle \text{proof} \rangle$

lemma *prod-ereal*: $(\prod_{i \in A}. \text{ereal } (f\ i)) = \text{ereal } (\text{prod } f\ A)$
 $\langle \text{proof} \rangle$

38.1.4 Power

lemma *ereal-power[simp]*: $(\text{ereal } x) \wedge^n = \text{ereal } (x \wedge^n)$
 $\langle \text{proof} \rangle$

lemma *ereal-power-PInf[simp]*: $(\infty :: \text{ereal}) \wedge^n = (\text{if } n = 0 \text{ then } 1 \text{ else } \infty)$
 $\langle \text{proof} \rangle$

lemma *ereal-power-uminus[simp]*:
fixes $x :: \text{ereal}$
shows $(-x) \wedge^n = (\text{if even } n \text{ then } x \wedge^n \text{ else } -(x \wedge^n))$
 $\langle \text{proof} \rangle$

lemma *ereal-power-numeral[simp]*:
 $(\text{numeral } \text{num} :: \text{ereal}) \wedge^n = \text{ereal } (\text{numeral } \text{num} \wedge^n)$
 $\langle \text{proof} \rangle$

lemma *zero-le-power-ereal[simp]*:
fixes $a :: \text{ereal}$
assumes $0 \leq a$
shows $0 \leq a \wedge^n$
 $\langle \text{proof} \rangle$

38.1.5 Subtraction

lemma *ereal-minus-minus-image[simp]*:
fixes $S :: \text{ereal set}$
shows $\text{uminus } ' \text{uminus } ' S = S$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-lessThan[simp]*:
fixes $a :: \text{ereal}$
shows $\text{uminus } ' \{..<a\} = \{-a<..\}$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-greaterThan[simp]*: $\text{uminus } ' \{(a :: \text{ereal})<..\} = \{..<-a\}$
 $\langle \text{proof} \rangle$

instantiation *ereal* :: *minus*
begin

definition $x - y = x + -(y :: \text{ereal})$
instance $\langle \text{proof} \rangle$

end

lemma *ereal-minus[simp]*:

$$\text{ereal } r - \text{ereal } p = \text{ereal } (r - p)$$

$$-\infty - \text{ereal } r = -\infty$$

$$\text{ereal } r - \infty = -\infty$$

$$(\infty :: \text{ereal}) - x = \infty$$

$$-(\infty :: \text{ereal}) - \infty = -\infty$$

$$x - -y = x + y$$

$$x - 0 = x$$

$$0 - x = -x$$

<proof>

lemma *ereal-x-minus-x[simp]*: $x - x = (\text{if } |x| = \infty \text{ then } \infty \text{ else } 0 :: \text{ereal})$

<proof>

lemma *ereal-eq-minus-iff*:

fixes $x \ y \ z :: \text{ereal}$

shows $x = z - y \longleftrightarrow$

$$(|y| \neq \infty \longrightarrow x + y = z) \wedge$$

$$(y = -\infty \longrightarrow x = \infty) \wedge$$

$$(y = \infty \longrightarrow z = \infty \longrightarrow x = \infty) \wedge$$

$$(y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty)$$

<proof>

lemma *ereal-eq-minus*:

fixes $x \ y \ z :: \text{ereal}$

shows $|y| \neq \infty \implies x = z - y \longleftrightarrow x + y = z$

<proof>

lemma *ereal-less-minus-iff*:

fixes $x \ y \ z :: \text{ereal}$

shows $x < z - y \longleftrightarrow$

$$(y = \infty \longrightarrow z = \infty \wedge x \neq \infty) \wedge$$

$$(y = -\infty \longrightarrow x \neq \infty) \wedge$$

$$(|y| \neq \infty \longrightarrow x + y < z)$$

<proof>

lemma *ereal-less-minus*:

fixes $x \ y \ z :: \text{ereal}$

shows $|y| \neq \infty \implies x < z - y \longleftrightarrow x + y < z$

<proof>

lemma *ereal-le-minus-iff*:

fixes $x \ y \ z :: \text{ereal}$

shows $x \leq z - y \longleftrightarrow (y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty) \wedge (|y| \neq \infty \longrightarrow x + y \leq z)$

<proof>

lemma *ereal-le-minus:*

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x \leq z - y \longleftrightarrow x + y \leq z$

<proof>

lemma *ereal-minus-less-iff:*

fixes $x\ y\ z :: \text{ereal}$

shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$

<proof>

lemma *ereal-minus-less:*

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x - y < z \longleftrightarrow x < z + y$

<proof>

lemma *ereal-minus-le-iff:*

fixes $x\ y\ z :: \text{ereal}$

shows $x - y \leq z \longleftrightarrow$

$(y = -\infty \longrightarrow z = \infty) \wedge$

$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$

$(|y| \neq \infty \longrightarrow x \leq z + y)$

<proof>

lemma *ereal-minus-le:*

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x - y \leq z \longleftrightarrow x \leq z + y$

<proof>

lemma *ereal-minus-eq-minus-iff:*

fixes $a\ b\ c :: \text{ereal}$

shows $a - b = a - c \longleftrightarrow$

$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$

<proof>

lemma *ereal-add-le-add-iff:*

fixes $a\ b\ c :: \text{ereal}$

shows $c + a \leq c + b \longleftrightarrow$

$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

<proof>

lemma *ereal-add-le-add-iff2:*

fixes $a\ b\ c :: \text{ereal}$

shows $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

<proof>

lemma *ereal-mult-le-mult-iff:*

fixes $a\ b\ c :: \text{ereal}$

shows $|c| \neq \infty \implies c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$

$\langle \text{proof} \rangle$

lemma *ereal-minus-mono*:

fixes $A B C D :: \text{ereal}$ **assumes** $A \leq B \ D \leq C$

shows $A - C \leq B - D$

$\langle \text{proof} \rangle$

lemma *ereal-mono-minus-cancel*:

fixes $a b c :: \text{ereal}$

shows $c - a \leq c - b \implies 0 \leq c \implies c < \infty \implies b \leq a$

$\langle \text{proof} \rangle$

lemma *real-of-ereal-minus*:

fixes $a b :: \text{ereal}$

shows $\text{real-of-ereal } (a - b) = (\text{if } |a| = \infty \vee |b| = \infty \text{ then } 0 \text{ else } \text{real-of-ereal } a - \text{real-of-ereal } b)$

$\langle \text{proof} \rangle$

lemma *real-of-ereal-minus'*: $|x| = \infty \longleftrightarrow |y| = \infty \implies \text{real-of-ereal } x - \text{real-of-ereal } y = \text{real-of-ereal } (x - y :: \text{ereal})$

$\langle \text{proof} \rangle$

lemma *ereal-diff-positive*:

fixes $a b :: \text{ereal}$ **shows** $a \leq b \implies 0 \leq b - a$

$\langle \text{proof} \rangle$

lemma *ereal-between*:

fixes $x e :: \text{ereal}$

assumes $|x| \neq \infty$ **and** $0 < e$

shows $x - e < x$

and $x < x + e$

$\langle \text{proof} \rangle$

lemma *ereal-minus-eq-PIfty-iff*:

fixes $x y :: \text{ereal}$

shows $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$

$\langle \text{proof} \rangle$

lemma *ereal-diff-add-eq-diff-diff-swap*:

fixes $x y z :: \text{ereal}$

shows $|y| \neq \infty \implies x - (y + z) = x - y - z$

$\langle \text{proof} \rangle$

lemma *ereal-diff-add-assoc2*:

fixes $x y z :: \text{ereal}$

shows $x + y - z = x - z + y$

$\langle \text{proof} \rangle$

lemma *ereal-add-uminus-conv-diff*: **fixes** $x y z :: \text{ereal}$ **shows** $-x + y = y - x$

$\langle \text{proof} \rangle$

lemma *ereal-minus-diff-eq*:

fixes $x\ y :: \text{ereal}$

shows $\llbracket x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq -\infty \rrbracket \implies -(x - y) = y - x$

$\langle \text{proof} \rangle$

lemma *ediff-le-self [simp]*: $x - y \leq (x :: \text{enat})$

$\langle \text{proof} \rangle$

lemma *ereal-abs-diff*:

fixes $a\ b :: \text{ereal}$

shows $\text{abs}(a - b) \leq \text{abs}\ a + \text{abs}\ b$

$\langle \text{proof} \rangle$

38.1.6 Division

instantiation *ereal* :: *inverse*

begin

function *inverse-ereal* **where**

inverse (*ereal* r) = (if $r = 0$ then ∞ else *ereal* (*inverse* r))

| *inverse* ($\infty :: \text{ereal}$) = 0

| *inverse* ($-\infty :: \text{ereal}$) = 0

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

definition $x \text{ div } y = x * \text{inverse}\ (y :: \text{ereal})$

instance $\langle \text{proof} \rangle$

end

lemma *real-of-ereal-inverse[simp]*:

fixes $a :: \text{ereal}$

shows $\text{real-of-ereal}\ (\text{inverse}\ a) = 1 / \text{real-of-ereal}\ a$

$\langle \text{proof} \rangle$

lemma *ereal-inverse[simp]*:

inverse ($0 :: \text{ereal}$) = ∞

inverse ($1 :: \text{ereal}$) = 1

$\langle \text{proof} \rangle$

lemma *ereal-divide[simp]*:

ereal $r / \text{ereal}\ p = (\text{if } p = 0 \text{ then } \text{ereal}\ r * \infty \text{ else } \text{ereal}\ (r / p))$

$\langle \text{proof} \rangle$

lemma *ereal-divide-same[simp]*:

fixes $x :: \text{ereal}$

shows $x / x = (\text{if } |x| = \infty \vee x = 0 \text{ then } 0 \text{ else } 1)$
 $\langle \text{proof} \rangle$

lemma *ereal-inv-inv[simp]*:
fixes $x :: \text{ereal}$
shows $\text{inverse} (\text{inverse } x) = (\text{if } x \neq -\infty \text{ then } x \text{ else } \infty)$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-minus[simp]*:
fixes $x :: \text{ereal}$
shows $\text{inverse} (-x) = (\text{if } x = 0 \text{ then } \infty \text{ else } -\text{inverse } x)$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-divide[simp]*:
fixes $x y :: \text{ereal}$
shows $-x / y = -(x / y)$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-Infty[simp]*:
fixes $x :: \text{ereal}$
shows $x / \infty = 0$ $x / -\infty = 0$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-one[simp]*: $x / 1 = (x :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-ereal[simp]*: $\infty / \text{ereal } r = (\text{if } 0 \leq r \text{ then } \infty \text{ else } -\infty)$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-nonneg-iff*: $0 \leq \text{inverse } (x :: \text{ereal}) \longleftrightarrow 0 \leq x \vee x = -\infty$
 $\langle \text{proof} \rangle$

lemma *inverse-ereal-ge0I*: $0 \leq (x :: \text{ereal}) \implies 0 \leq \text{inverse } x$
 $\langle \text{proof} \rangle$

lemma *zero-le-divide-ereal[simp]*:
fixes $a :: \text{ereal}$
assumes $0 \leq a$ **and** $0 \leq b$
shows $0 \leq a / b$
 $\langle \text{proof} \rangle$

lemma *ereal-le-divide-pos*:
fixes $x y z :: \text{ereal}$
shows $x > 0 \implies x \neq \infty \implies y \leq z / x \longleftrightarrow x * y \leq z$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-le-pos*:
fixes $x y z :: \text{ereal}$
shows $x > 0 \implies x \neq \infty \implies z / x \leq y \longleftrightarrow z \leq x * y$

$\langle \text{proof} \rangle$

lemma *ereal-le-divide-neg*:

fixes $x\ y\ z :: \text{ereal}$

shows $x < 0 \implies x \neq -\infty \implies y \leq z \ / \ x \longleftrightarrow z \leq x * y$

$\langle \text{proof} \rangle$

lemma *ereal-divide-le-neg*:

fixes $x\ y\ z :: \text{ereal}$

shows $x < 0 \implies x \neq -\infty \implies z \ / \ x \leq y \longleftrightarrow x * y \leq z$

$\langle \text{proof} \rangle$

lemma *ereal-inverse-antimono-strict*:

fixes $x\ y :: \text{ereal}$

shows $0 \leq x \implies x < y \implies \text{inverse } y < \text{inverse } x$

$\langle \text{proof} \rangle$

lemma *ereal-inverse-antimono*:

fixes $x\ y :: \text{ereal}$

shows $0 \leq x \implies x \leq y \implies \text{inverse } y \leq \text{inverse } x$

$\langle \text{proof} \rangle$

lemma *inverse-inverse-Pinfy-iff[simp]*:

fixes $x :: \text{ereal}$

shows $\text{inverse } x = \infty \longleftrightarrow x = 0$

$\langle \text{proof} \rangle$

lemma *ereal-inverse-eq-0*:

fixes $x :: \text{ereal}$

shows $\text{inverse } x = 0 \longleftrightarrow x = \infty \vee x = -\infty$

$\langle \text{proof} \rangle$

lemma *ereal-0-gt-inverse*:

fixes $x :: \text{ereal}$

shows $0 < \text{inverse } x \longleftrightarrow x \neq \infty \wedge 0 \leq x$

$\langle \text{proof} \rangle$

lemma *ereal-inverse-le-0-iff*:

fixes $x :: \text{ereal}$

shows $\text{inverse } x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$

$\langle \text{proof} \rangle$

lemma *ereal-divide-eq-0-iff*: $x \ / \ y = 0 \longleftrightarrow x = 0 \vee |y :: \text{ereal}| = \infty$

$\langle \text{proof} \rangle$

lemma *ereal-mult-less-right*:

fixes $a\ b\ c :: \text{ereal}$

assumes $b * a < c * a \ 0 < a \ a < \infty$

shows $b < c$

$\langle proof \rangle$

lemma *ereal-mult-divide*:

fixes $a\ b :: \text{ereal}$

shows $0 < b \implies b < \infty \implies b * (a / b) = a$

$\langle proof \rangle$

lemma *ereal-power-divide*:

fixes $x\ y :: \text{ereal}$

shows $y \neq 0 \implies (x / y) ^ n = x ^ n / y ^ n$

$\langle proof \rangle$

lemma *ereal-le-mult-one-interval*:

fixes $x\ y :: \text{ereal}$

assumes $y: y \neq -\infty$

assumes $z: \bigwedge z. 0 < z \implies z < 1 \implies z * x \leq y$

shows $x \leq y$

$\langle proof \rangle$

lemma *ereal-divide-right-mono[simp]*:

fixes $x\ y\ z :: \text{ereal}$

assumes $x \leq y$

and $0 < z$

shows $x / z \leq y / z$

$\langle proof \rangle$

lemma *ereal-divide-left-mono[simp]*:

fixes $x\ y\ z :: \text{ereal}$

assumes $y \leq x$

and $0 < z$

and $0 < x * y$

shows $z / x \leq z / y$

$\langle proof \rangle$

lemma *ereal-divide-zero-left[simp]*:

fixes $a :: \text{ereal}$

shows $0 / a = 0$

$\langle proof \rangle$

lemma *ereal-times-divide-eq-left[simp]*:

fixes $a\ b\ c :: \text{ereal}$

shows $b / c * a = b * a / c$

$\langle proof \rangle$

lemma *ereal-times-divide-eq*: $a * (b / c :: \text{ereal}) = a * b / c$

$\langle proof \rangle$

lemma *ereal-inverse-real [simp]*: $|z| \neq \infty \implies z \neq 0 \implies \text{ereal } (\text{inverse } (\text{real-of-ereal } z)) = \text{inverse } z$

$\langle \text{proof} \rangle$

lemma *ereal-inverse-mult*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } (a * (b::\text{ereal})) = \text{inverse } a * \text{inverse } b$
 $\langle \text{proof} \rangle$

lemma *inverse-eq-infinity-iff-eq-zero* [simp]:

$1/(x::\text{ereal}) = \infty \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *ereal-distrib-left*:

fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$
and $|c| \neq \infty$
shows $c * (a + b) = c * a + c * b$
 $\langle \text{proof} \rangle$

lemma *ereal-distrib-minus-left*:

fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq \infty$
and $a \neq -\infty \vee b \neq -\infty$
and $|c| \neq \infty$
shows $c * (a - b) = c * a - c * b$
 $\langle \text{proof} \rangle$

lemma *ereal-distrib-minus-right*:

fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq \infty$
and $a \neq -\infty \vee b \neq -\infty$
and $|c| \neq \infty$
shows $(a - b) * c = a * c - b * c$
 $\langle \text{proof} \rangle$

38.2 Complete lattice

instantiation *ereal* :: *lattice*

begin

definition [simp]: $\text{sup } x\ y = (\text{max } x\ y :: \text{ereal})$

definition [simp]: $\text{inf } x\ y = (\text{min } x\ y :: \text{ereal})$

instance $\langle \text{proof} \rangle$

end

instantiation *ereal* :: *complete-lattice*

begin

definition $\text{bot} = (-\infty::\text{ereal})$

definition $top = (\infty :: ereal)$

definition $Sup\ S = (SOME\ x :: ereal. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z))$

definition $Inf\ S = (SOME\ x :: ereal. (\forall y \in S. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x))$

lemma *ereal-complete-Sup*:

fixes $S :: ereal\ set$

shows $\exists x. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z)$

<proof>

lemma *ereal-complete-uminus-eq*:

fixes $S :: ereal\ set$

shows $(\forall y \in uminus'S. y \leq x) \wedge (\forall z. (\forall y \in uminus'S. y \leq z) \longrightarrow x \leq z)$

$\longleftrightarrow (\forall y \in S. -x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq -x)$

<proof>

lemma *ereal-complete-Inf*:

$\exists x. (\forall y \in S :: ereal\ set. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x)$

<proof>

instance

<proof>

end

instance *ereal :: complete-linorder* *<proof>*

instance *ereal :: linear-continuum*

<proof>

lemma *min-PInf [simp]*: $min\ (\infty :: ereal)\ x = x$

<proof>

lemma *min-PInf2 [simp]*: $min\ x\ (\infty :: ereal) = x$

<proof>

lemma *max-PInf [simp]*: $max\ (\infty :: ereal)\ x = \infty$

<proof>

lemma *max-PInf2 [simp]*: $max\ x\ (\infty :: ereal) = \infty$

<proof>

lemma *min-MInf [simp]*: $min\ (-\infty :: ereal)\ x = -\infty$

<proof>

lemma *min-MInf2 [simp]*: $min\ x\ (-\infty :: ereal) = -\infty$

<proof>

lemma *max-MInf [simp]:* $\max (-\infty::ereal) x = x$
 ⟨proof⟩

lemma *max-MInf2 [simp]:* $\max x (-\infty::ereal) = x$
 ⟨proof⟩

38.3 Extended real intervals

lemma *real-greaterThanLessThan-infinity-eq:*
 $real-of-ereal \text{ ‘ } \{N::ereal <..
 (if $N = \infty$ then $\{\}$ else if $N = -\infty$ then *UNIV* else $\{real-of-ereal N <..\}$)
 ⟨proof⟩$

lemma *real-greaterThanLessThan-minus-infinity-eq:*
 $real-of-ereal \text{ ‘ } \{-\infty <.. $N::ereal\} =$
 (if $N = \infty$ then *UNIV* else if $N = -\infty$ then $\{\}$ else $\{.. $real-of-ereal N\}$)
 ⟨proof⟩$$

lemma *real-greaterThanLessThan-inter:*
 $real-of-ereal \text{ ‘ } \{N <.. $M::ereal\} = real-of-ereal \text{ ‘ } \{-\infty <.. $M\} \cap real-of-ereal \text{ ‘ } \{N <.. $\infty\}$
 ⟨proof⟩$$$

lemma *real-atLeastGreaterThan-eq:* $real-of-ereal \text{ ‘ } \{N <.. $M::ereal\} =$
 (if $N = \infty$ then $\{\}$ else
 if $N = -\infty$ then
 (if $M = \infty$ then *UNIV*
 else if $M = -\infty$ then $\{\}$
 else $\{.. $real-of-ereal M\}$)
 else if $M = -\infty$ then $\{\}$
 else if $M = \infty$ then $\{real-of-ereal N <..\}$
 else $\{real-of-ereal N <.. $real-of-ereal M\}$)
 ⟨proof⟩$$$

lemma *real-image-ereal-ivl:*
fixes $a b::ereal$
shows
 $real-of-ereal \text{ ‘ } \{a <.. $b\} =$
 (if $a < b$ then (if $a = -\infty$ then if $b = \infty$ then *UNIV* else $\{.. $real-of-ereal b\}$
 else if $b = \infty$ then $\{real-of-ereal a <..\}$ else $\{real-of-ereal a <.. $real-of-ereal b\}$)
 else $\{\}$)
 ⟨proof⟩$$$

lemma **fixes** $a b c::ereal$
shows *not-infityI:* $a < b \implies b < c \implies abs b \neq \infty$
 ⟨proof⟩

context

fixes $r\ s\ t::\text{real}$
begin

lemma *interval-Ioo-neq-Ioi*: $\{r <..
 $\langle\text{proof}\rangle$$

lemma *interval-Ioo-neq-Iio*: $\{r <..
 $\langle\text{proof}\rangle$$

lemma *interval-neq-ioo-UNIV*: $\{r <..
and *interval-Ioi-neq-UNIV*: $\{r <..\} \neq \text{UNIV}$
and *interval-Iio-neq-UNIV*: $\{..
 $\langle\text{proof}\rangle$$$

lemma *interval-Ioi-neq-Iio*: $\{r <..\} \neq \{..
 $\langle\text{proof}\rangle$$

lemma *interval-empty-neq-Ioi*: $\{\} \neq \{r <..\}$
and *interval-empty-neq-Iio*: $\{\} \neq \{..
 $\langle\text{proof}\rangle$$

end

lemmas *interval-neqs* = *interval-Ioo-neq-Ioi interval-Ioo-neq-Iio*
interval-neq-ioo-UNIV interval-Ioi-neq-Iio
interval-Ioi-neq-UNIV interval-Iio-neq-UNIV
interval-empty-neq-Ioi interval-empty-neq-Iio

lemma *greaterThanLessThan-eq-iff*:
fixes $r\ s\ t\ u::\text{real}$
shows $(\{r <..
 $\langle\text{proof}\rangle$$

lemma *real-of-ereal-image-greaterThanLessThan-iff*:
 $\text{real-of-ereal} \text{ ‘ } \{a <.. **= real-of-ereal ‘ } \{c <..
 $= c \wedge b = d)$
 $\langle\text{proof}\rangle$**$

lemma *uminus-image-real-of-ereal-image-greaterThanLessThan*:
 $\text{uminus} \text{ ‘ } \text{real-of-ereal} \text{ ‘ } \{l <.. -l\}$
 $\langle\text{proof}\rangle$

lemma *add-image-real-of-ereal-image-greaterThanLessThan*:
 $(+) \ c \text{ ‘ } \text{real-of-ereal} \text{ ‘ } \{l <.. c + u\}$
 $\langle\text{proof}\rangle$

lemma *add2-image-real-of-ereal-image-greaterThanLessThan*:
 $(\lambda x. x + c) \text{ ‘ } \text{real-of-ereal} \text{ ‘ } \{l <.. u + c\}$
 $\langle\text{proof}\rangle$

lemma *minus-image-real-of-ereal-image-greaterThanLessThan*:

(-) $c \in \text{real-of-ereal } \{l <..< u\} = \text{real-of-ereal } \{c - u <..< c - l\}$
 (is ?l = ?r)
 <proof>

lemma *real-ereal-bound-lemma-up*:

assumes $s \in \text{real-of-ereal } \{a <..< b\}$
assumes $t \notin \text{real-of-ereal } \{a <..< b\}$
assumes $s \leq t$
shows $b \neq \infty$
 <proof>

lemma *real-ereal-bound-lemma-down*:

assumes $s: s \in \text{real-of-ereal } \{a <..< b\}$
and $t: t \notin \text{real-of-ereal } \{a <..< b\}$
and $t \leq s$
shows $a \neq -\infty$
 <proof>

38.4 Topological space

instantiation *ereal* :: *linear-continuum-topology*
begin

definition *open-ereal* :: *ereal set* \Rightarrow *bool* **where**

open-ereal-generated: *open-ereal* = *generate-topology* (*range lessThan* \cup *range greaterThan*)

instance
 <proof>

end

lemma *continuous-on-ereal*[*continuous-intros*]:

assumes $f: \text{continuous-on } s \text{ f}$ **shows** *continuous-on* s $(\lambda x. \text{ereal } (f x))$
 <proof>

lemma *tendsto-ereal*[*tendsto-intros*, *simp*, *intro*]: $(f \longrightarrow x) F \Longrightarrow ((\lambda x. \text{ereal } (f x)) \longrightarrow \text{ereal } x) F$
 <proof>

lemma *tendsto-uminus-ereal*[*tendsto-intros*, *simp*, *intro*]:

assumes $(f \longrightarrow x) F$
shows $((\lambda x. - f x::\text{ereal}) \longrightarrow - x) F$
 <proof>

lemma *at-infty-ereal-eq-at-top*: $at\ \infty = filtermap\ ereal\ at_top$
 $\langle proof \rangle$

lemma *ereal-Lim-uminus*: $(f \longrightarrow f0)\ net \longleftrightarrow ((\lambda x. - f\ x::ereal) \longrightarrow - f0)$
 $\langle proof \rangle$

lemma *ereal-divide-less-iff*: $0 < (c::ereal) \implies c < \infty \implies a / c < b \longleftrightarrow a < b * c$
 $\langle proof \rangle$

lemma *ereal-less-divide-iff*: $0 < (c::ereal) \implies c < \infty \implies a < b / c \longleftrightarrow a * c < b$
 $\langle proof \rangle$

lemma *tendsto-cmult-ereal*[*tendsto-intros, simp, intro*]:
assumes $c: |c| \neq \infty$ **and** $f: (f \longrightarrow x)\ F$
shows $((\lambda x. c * f\ x::ereal) \longrightarrow c * x)\ F$
 $\langle proof \rangle$

lemma *tendsto-cmult-ereal-not-0*[*tendsto-intros, simp, intro*]:
assumes $x \neq 0$ **and** $f: (f \longrightarrow x)\ F$
shows $((\lambda x. c * f\ x::ereal) \longrightarrow c * x)\ F$
 $\langle proof \rangle$

lemma *tendsto-cadd-ereal*[*tendsto-intros, simp, intro*]:
assumes $c: y \neq -\infty\ x \neq -\infty$ **and** $f: (f \longrightarrow x)\ F$
shows $((\lambda x. f\ x + y::ereal) \longrightarrow x + y)\ F$
 $\langle proof \rangle$

lemma *tendsto-add-left-ereal*[*tendsto-intros, simp, intro*]:
assumes $c: |y| \neq \infty$ **and** $f: (f \longrightarrow x)\ F$
shows $((\lambda x. f\ x + y::ereal) \longrightarrow x + y)\ F$
 $\langle proof \rangle$

lemma *continuous-at-ereal*[*continuous-intros*]: $continuous\ F\ f \implies continuous\ F\ (\lambda x. ereal\ (f\ x))$
 $\langle proof \rangle$

lemma *ereal-Sup*:
assumes $*$: $|SUP\ a \in A. ereal\ a| \neq \infty$
shows $ereal\ (Sup\ A) = (SUP\ a \in A. ereal\ a)$
 $\langle proof \rangle$

lemma *ereal-SUP*: $|SUP\ a \in A. ereal\ (f\ a)| \neq \infty \implies ereal\ (SUP\ a \in A. f\ a) = (SUP\ a \in A. ereal\ (f\ a))$
 $\langle proof \rangle$

lemma *ereal-Inf*:

assumes *: $|INF\ a \in A.\ ereal\ a| \neq \infty$
shows $ereal\ (Inf\ A) = (INF\ a \in A.\ ereal\ a)$
 $\langle proof \rangle$

lemma *ereal-Inf'*:
assumes *: $bdd\text{-}below\ A\ A \neq \{\}$
shows $ereal\ (Inf\ A) = (INF\ a \in A.\ ereal\ a)$
 $\langle proof \rangle$

lemma *ereal-INF*: $|INF\ a \in A.\ ereal\ (f\ a)| \neq \infty \implies ereal\ (INF\ a \in A.\ f\ a) = (INF\ a \in A.\ ereal\ (f\ a))$
 $\langle proof \rangle$

lemma *ereal-Sup-uminus-image-eq*: $Sup\ (uminus\ 'S :: ereal\ set) = -\ Inf\ S$
 $\langle proof \rangle$

lemma *ereal-SUP-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(SUP\ x \in S.\ uminus\ (f\ x)) = -\ (INF\ x \in S.\ f\ x)$
 $\langle proof \rangle$

lemma *ereal-inj-on-uminus*[*intro, simp*]: $inj\text{-}on\ uminus\ (A :: ereal\ set)$
 $\langle proof \rangle$

lemma *ereal-Inf-uminus-image-eq*: $Inf\ (uminus\ 'S :: ereal\ set) = -\ Sup\ S$
 $\langle proof \rangle$

lemma *ereal-INF-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(INF\ x \in S.\ -\ f\ x) = -\ (SUP\ x \in S.\ f\ x)$
 $\langle proof \rangle$

lemma *ereal-SUP-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \implies l \neq -\infty \implies u \neq \infty \implies \forall a \in A.\ l \leq f\ a \wedge f\ a \leq u \implies |Sup\ (f\ 'A)| \neq \infty$
 $\langle proof \rangle$

lemma *ereal-INF-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \implies l \neq -\infty \implies u \neq \infty \implies \forall a \in A.\ l \leq f\ a \wedge f\ a \leq u \implies |Inf\ (f\ 'A)| \neq \infty$
 $\langle proof \rangle$

lemma *ereal-image-uminus-shift*:
fixes $X\ Y :: ereal\ set$
shows $uminus\ 'X = Y \longleftrightarrow X = uminus\ 'Y$
 $\langle proof \rangle$

lemma *Sup-eq-MInfty*:

fixes $S :: \text{ereal set}$

shows $\text{Sup } S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$

$\langle \text{proof} \rangle$

lemma *Inf-eq-PInfty*:

fixes $S :: \text{ereal set}$

shows $\text{Inf } S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$

$\langle \text{proof} \rangle$

lemma *Inf-eq-MInfty*:

fixes $S :: \text{ereal set}$

shows $-\infty \in S \implies \text{Inf } S = -\infty$

$\langle \text{proof} \rangle$

lemma *Sup-eq-PInfty*:

fixes $S :: \text{ereal set}$

shows $\infty \in S \implies \text{Sup } S = \infty$

$\langle \text{proof} \rangle$

lemma *not-MInfty-nonneg[simp]*: $0 \leq (x::\text{ereal}) \implies x \neq -\infty$

$\langle \text{proof} \rangle$

lemma *Sup-ereal-close*:

fixes $e :: \text{ereal}$

assumes $0 < e$

and $S: |\text{Sup } S| \neq \infty \ S \neq \{\}$

shows $\exists x \in S. \text{Sup } S - e < x$

$\langle \text{proof} \rangle$

lemma *Inf-ereal-close*:

fixes $e :: \text{ereal}$

assumes $|\text{Inf } X| \neq \infty$

and $0 < e$

shows $\exists x \in X. x < \text{Inf } X + e$

$\langle \text{proof} \rangle$

lemma *SUP-PInfty*:

$(\bigwedge n::\text{nat}. \exists i \in A. \text{ereal } (\text{real } n) \leq f i) \implies (\text{SUP } i \in A. f i :: \text{ereal}) = \infty$

$\langle \text{proof} \rangle$

lemma *SUP-nat-Infty*: $(\text{SUP } i. \text{ereal } (\text{real } i)) = \infty$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-add-left*:

assumes $I \neq \{\} \ c \neq -\infty$

shows $(\text{SUP } i \in I. f i + c :: \text{ereal}) = (\text{SUP } i \in I. f i) + c$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-add-right*:

fixes $c :: \text{ereal}$

shows $I \neq \{\} \implies c \neq -\infty \implies (\text{SUP } i \in I. c + f i) = c + (\text{SUP } i \in I. f i)$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-minus-right*:

assumes $I \neq \{\}$ $c \neq -\infty$

shows $(\text{SUP } i \in I. c - f i :: \text{ereal}) = c - (\text{INF } i \in I. f i)$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-minus-left*:

assumes $I \neq \{\}$ $c \neq \infty$

shows $(\text{SUP } i \in I. f i - c :: \text{ereal}) = (\text{SUP } i \in I. f i) - c$

$\langle \text{proof} \rangle$

lemma *INF-ereal-minus-right*:

assumes $I \neq \{\}$ **and** $|c| \neq \infty$

shows $(\text{INF } i \in I. c - f i) = c - (\text{SUP } i \in I. f i :: \text{ereal})$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-le-addI*:

fixes $f :: 'i \Rightarrow \text{ereal}$

assumes $\bigwedge i. f i + y \leq z$ **and** $y \neq -\infty$

shows $\text{Sup } (f \text{ ‘ UNIV}) + y \leq z$

$\langle \text{proof} \rangle$

lemma *SUP-combine*:

fixes $f :: 'a :: \text{semilattice-sup} \Rightarrow 'a :: \text{semilattice-sup} \Rightarrow 'b :: \text{complete-lattice}$

assumes *mono*: $\bigwedge a b c d. a \leq b \implies c \leq d \implies f a c \leq f b d$

shows $(\text{SUP } i \in \text{UNIV}. \text{SUP } j \in \text{UNIV}. f i j) = (\text{SUP } i. f i i)$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-add*:

fixes $f g :: \text{nat} \Rightarrow \text{ereal}$

assumes *inc*: $\text{incseq } f \text{ incseq } g$

and *pos*: $\bigwedge i. f i \neq -\infty \bigwedge i. g i \neq -\infty$

shows $(\text{SUP } i. f i + g i) = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$

$\langle \text{proof} \rangle$

lemma *INF-eq-minf*: $(\text{INF } i \in I. f i :: \text{ereal}) \neq -\infty \iff (\exists b > -\infty. \forall i \in I. b \leq f i)$

$\langle \text{proof} \rangle$

lemma *INF-ereal-add-left*:

assumes $I \neq \{\}$ $c \neq -\infty \bigwedge x. x \in I \implies 0 \leq f x$

shows $(\text{INF } i \in I. f i + c :: \text{ereal}) = (\text{INF } i \in I. f i) + c$

$\langle \text{proof} \rangle$

lemma *INF-ereal-add-right*:

assumes $I \neq \{\}$ $c \neq -\infty \bigwedge x. x \in I \implies 0 \leq f x$

shows $(\text{INF } i \in I. c + f i :: \text{ereal}) = c + (\text{INF } i \in I. f i)$
 $\langle \text{proof} \rangle$

lemma *INF-ereal-add-directed*:

fixes $f g :: 'a \Rightarrow \text{ereal}$
assumes *nonneg*: $\bigwedge i. i \in I \implies 0 \leq f i \wedge i. i \in I \implies 0 \leq g i$
assumes *directed*: $\bigwedge i j. i \in I \implies j \in I \implies \exists k \in I. f i + g j \geq f k + g k$
shows $(\text{INF } i \in I. f i + g i) = (\text{INF } i \in I. f i) + (\text{INF } i \in I. g i)$
 $\langle \text{proof} \rangle$

lemma *INF-ereal-add*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes *decseq* f *decseq* g
and *fin*: $\bigwedge i. f i \neq \infty \wedge i. g i \neq \infty$
shows $(\text{INF } i. f i + g i) = \text{Inf } (f ' \text{UNIV}) + \text{Inf } (g ' \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-add-pos*:

fixes $f g :: \text{nat} \Rightarrow \text{ereal}$
assumes *incseq* f *incseq* g
and $\bigwedge i. 0 \leq f i \wedge i. 0 \leq g i$
shows $(\text{SUP } i. f i + g i) = \text{Sup } (f ' \text{UNIV}) + \text{Sup } (g ' \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-sum*:

fixes $f g :: 'a \Rightarrow \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge n. n \in A \implies \text{incseq } (f n)$
and *pos*: $\bigwedge n i. n \in A \implies 0 \leq f n i$
shows $(\text{SUP } i. \sum n \in A. f n i) = (\sum n \in A. \text{Sup } ((f n) ' \text{UNIV}))$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-mult-left*:

fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $I \neq \{\}$
assumes f : $\bigwedge i. i \in I \implies 0 \leq f i$ **and** c : $0 \leq c$
shows $(\text{SUP } i \in I. c * f i) = c * (\text{SUP } i \in I. f i)$
 $\langle \text{proof} \rangle$

lemma *countable-approach*:

fixes $x :: \text{ereal}$
assumes $x \neq -\infty$
shows $\exists f. \text{incseq } f \wedge (\forall i :: \text{nat}. f i < x) \wedge (f \longrightarrow x)$
 $\langle \text{proof} \rangle$

lemma *Sup-countable-SUP*:

assumes $A \neq \{\}$
shows $\exists f :: \text{nat} \Rightarrow \text{ereal}. \text{incseq } f \wedge \text{range } f \subseteq A \wedge \text{Sup } A = (\text{SUP } i. f i)$
 $\langle \text{proof} \rangle$

lemma *Inf-countable-INF*:

assumes $A \neq \{\}$ **shows** $\exists f::nat \Rightarrow ereal. decseq f \wedge range f \subseteq A \wedge Inf A = (INF i. f i)$
 $\langle proof \rangle$

lemma *SUP-countable-SUP*:

$A \neq \{\} \implies \exists f::nat \Rightarrow ereal. range f \subseteq g'A \wedge Sup (g ' A) = Sup (f ' UNIV)$
 $\langle proof \rangle$

38.5 Relation to *enat*

definition *ereal-of-enat* $n = (case\ n\ of\ enat\ n \Rightarrow ereal\ (real\ n) \mid \infty \Rightarrow \infty)$

declare $[[coercion\ ereal-of-enat :: enat \Rightarrow ereal]]$

declare $[[coercion\ (\lambda n. ereal\ (real\ n)) :: nat \Rightarrow ereal]]$

lemma *ereal-of-enat-simps[simp]*:

$ereal-of-enat\ (enat\ n) = ereal\ n$
 $ereal-of-enat\ \infty = \infty$
 $\langle proof \rangle$

lemma *ereal-of-enat-le-iff[simp]*: $ereal-of-enat\ m \leq ereal-of-enat\ n \longleftrightarrow m \leq n$
 $\langle proof \rangle$

lemma *ereal-of-enat-less-iff[simp]*: $ereal-of-enat\ m < ereal-of-enat\ n \longleftrightarrow m < n$
 $\langle proof \rangle$

lemma *numeral-le-ereal-of-enat-iff[simp]*: $numeral\ m \leq ereal-of-enat\ n \longleftrightarrow numeral\ m \leq n$
 $\langle proof \rangle$

lemma *numeral-less-ereal-of-enat-iff[simp]*: $numeral\ m < ereal-of-enat\ n \longleftrightarrow numeral\ m < n$
 $\langle proof \rangle$

lemma *ereal-of-enat-ge-zero-cancel-iff[simp]*: $0 \leq ereal-of-enat\ n \longleftrightarrow 0 \leq n$
 $\langle proof \rangle$

lemma *ereal-of-enat-gt-zero-cancel-iff[simp]*: $0 < ereal-of-enat\ n \longleftrightarrow 0 < n$
 $\langle proof \rangle$

lemma *ereal-of-enat-zero[simp]*: $ereal-of-enat\ 0 = 0$
 $\langle proof \rangle$

lemma *ereal-of-enat-inf[simp]*: $ereal-of-enat\ n = \infty \longleftrightarrow n = \infty$
 $\langle proof \rangle$

lemma *ereal-of-enat-add*: $ereal-of-enat\ (m + n) = ereal-of-enat\ m + ereal-of-enat\ n$

$\langle \text{proof} \rangle$

lemma *ereal-of-enat-sub*:

assumes $n \leq m$

shows $\text{ereal-of-enat } (m - n) = \text{ereal-of-enat } m - \text{ereal-of-enat } n$

$\langle \text{proof} \rangle$

lemma *ereal-of-enat-mult*:

$\text{ereal-of-enat } (m * n) = \text{ereal-of-enat } m * \text{ereal-of-enat } n$

$\langle \text{proof} \rangle$

lemmas *ereal-of-enat-pushin* = *ereal-of-enat-add* *ereal-of-enat-sub* *ereal-of-enat-mult*

lemmas *ereal-of-enat-pushout* = *ereal-of-enat-pushin*[*symmetric*]

lemma *ereal-of-enat-nonneg*: $\text{ereal-of-enat } n \geq 0$

$\langle \text{proof} \rangle$

lemma *ereal-of-enat-Sup*:

assumes $A \neq \{\}$ **shows** $\text{ereal-of-enat } (\text{Sup } A) = (\text{SUP } a \in A. \text{ereal-of-enat } a)$

$\langle \text{proof} \rangle$

lemma *ereal-of-enat-SUP*:

$A \neq \{\} \implies \text{ereal-of-enat } (\text{SUP } a \in A. f a) = (\text{SUP } a \in A. \text{ereal-of-enat } (f a))$

$\langle \text{proof} \rangle$

38.6 Limits on *ereal*

lemma *open-PInfty*: $\text{open } A \implies \infty \in A \implies (\exists x. \{\text{ereal } x < ..\} \subseteq A)$

$\langle \text{proof} \rangle$

lemma *open-MInfty*: $\text{open } A \implies -\infty \in A \implies (\exists x. \{.. < \text{ereal } x\} \subseteq A)$

$\langle \text{proof} \rangle$

lemma *open-ereal-vimage*: $\text{open } S \implies \text{open } (\text{ereal } -' S)$

$\langle \text{proof} \rangle$

lemma *open-ereal*: $\text{open } S \implies \text{open } (\text{ereal } ' S)$

$\langle \text{proof} \rangle$

lemma *open-image-real-of-ereal*:

fixes $X :: \text{ereal set}$

assumes $\text{open } X$

assumes *infy*: $\infty \notin X \text{ } -\infty \notin X$

shows $\text{open } (\text{real-of-ereal } ' X)$

$\langle \text{proof} \rangle$

lemma *eventually-finite*:

fixes $x :: \text{ereal}$

assumes $|x| \neq \infty \text{ } (f \longrightarrow x) F$

shows eventually $(\lambda x. |f\ x| \neq \infty)\ F$
 $\langle proof \rangle$

lemma *open-ereal-def*:

$open\ A \longleftrightarrow open\ (ereal - 'A) \wedge (\infty \in A \longrightarrow (\exists x. \{ereal\ x < ..\} \subseteq A)) \wedge (-\infty \in A \longrightarrow (\exists x. \{.. < ereal\ x\} \subseteq A))$
 (is $open\ A \longleftrightarrow ?rhs$)
 $\langle proof \rangle$

lemma *open-PInfty2*:

assumes $open\ A$ **and** $\infty \in A$
obtains x **where** $\{ereal\ x < ..\} \subseteq A$
 $\langle proof \rangle$

lemma *open-MInfty2*:

assumes $open\ A$ **and** $-\infty \in A$
obtains x **where** $\{.. < ereal\ x\} \subseteq A$
 $\langle proof \rangle$

lemma *ereal-openE*:

assumes $open\ A$
obtains $x\ y$ **where** $open\ (ereal - 'A)$
and $\infty \in A \implies \{ereal\ x < ..\} \subseteq A$
and $-\infty \in A \implies \{.. < ereal\ y\} \subseteq A$
 $\langle proof \rangle$

lemmas $open-ereal-lessThan = open-lessThan[where\ 'a=ereal]$

lemmas $open-ereal-greaterThan = open-greaterThan[where\ 'a=ereal]$

lemmas $ereal-open-greaterThanLessThan = open-greaterThanLessThan[where\ 'a=ereal]$

lemmas $closed-ereal-atLeast = closed-atLeast[where\ 'a=ereal]$

lemmas $closed-ereal-atMost = closed-atMost[where\ 'a=ereal]$

lemmas $closed-ereal-atLeastAtMost = closed-atLeastAtMost[where\ 'a=ereal]$

lemmas $closed-ereal-singleton = closed-singleton[where\ 'a=ereal]$

lemma *ereal-open-cont-interval*:

fixes $S :: \text{ereal set}$
assumes $open\ S$
and $x \in S$
and $|x| \neq \infty$
obtains e **where** $e > 0$ **and** $\{x-e < .. < x+e\} \subseteq S$
 $\langle proof \rangle$

lemma *ereal-open-cont-interval2*:

fixes $S :: \text{ereal set}$
assumes $open\ S$ **and** $x \in S$ **and** $|x| \neq \infty$
obtains $a\ b$ **where** $a < x$ **and** $x < b$ **and** $\{a < .. < b\} \subseteq S$
 $\langle proof \rangle$

38.6.1 Convergent sequences

lemma *lim-real-of-ereal[simp]*:
assumes *lim*: $(f \longrightarrow \text{ereal } x) \text{ net}$
shows $((\lambda x. \text{real-of-ereal } (f \ x)) \longrightarrow x) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *lim-ereal[simp]*: $((\lambda n. \text{ereal } (f \ n)) \longrightarrow \text{ereal } x) \text{ net} \longleftrightarrow (f \longrightarrow x) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *convergent-real-imp-convergent-ereal*:
assumes *convergent a*
shows *convergent* $(\lambda n. \text{ereal } (a \ n))$ **and** $\lim (\lambda n. \text{ereal } (a \ n)) = \text{ereal } (\lim a)$
 $\langle \text{proof} \rangle$

lemma *tendsto-PInfy*: $(f \longrightarrow \infty) \ F \longleftrightarrow (\forall r. \text{eventually } (\lambda x. \text{ereal } r < f \ x) \ F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-PInfy'*: $(f \longrightarrow \infty) \ F = (\forall r > c. \text{eventually } (\lambda x. \text{ereal } r < f \ x) \ F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-PInfy-eq-at-top*:
 $((\lambda z. \text{ereal } (f \ z)) \longrightarrow \infty) \ F \longleftrightarrow (LIM \ z \ F. f \ z :> \text{at-top})$
 $\langle \text{proof} \rangle$

lemma *tendsto-MInfy*: $(f \longrightarrow -\infty) \ F \longleftrightarrow (\forall r. \text{eventually } (\lambda x. f \ x < \text{ereal } r) \ F)$
 $\langle \text{proof} \rangle$

lemma *tendsto-MInfy'*: $(f \longrightarrow -\infty) \ F = (\forall r < c. \text{eventually } (\lambda x. \text{ereal } r > f \ x) \ F)$
 $\langle \text{proof} \rangle$

lemma *Lim-PInfy*: $f \longrightarrow \infty \longleftrightarrow (\forall B. \exists N. \forall n \geq N. f \ n \geq \text{ereal } B)$
 $\langle \text{proof} \rangle$

lemma *Lim-MInfy*: $f \longrightarrow -\infty \longleftrightarrow (\forall B. \exists N. \forall n \geq N. \text{ereal } B \geq f \ n)$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded-PInfy*: $f \longrightarrow l \implies (\bigwedge n. f \ n \leq \text{ereal } B) \implies l \neq \infty$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded-MInfy*: $f \longrightarrow l \implies (\bigwedge n. \text{ereal } B \leq f \ n) \implies l \neq -\infty$
 $\langle \text{proof} \rangle$

lemma *tendsto-zero-erealI*:
assumes $\bigwedge e. e > 0 \implies \text{eventually } (\lambda x. |f \ x| < \text{ereal } e) \ F$
shows $(f \longrightarrow 0) \ F$
 $\langle \text{proof} \rangle$

lemma *Lim-bounded-PInfy2*: $f \longrightarrow l \implies \forall n \geq N. f\ n \leq \text{ereal } B \implies l \neq \infty$
 $\langle \text{proof} \rangle$

lemma *real-of-ereal-mult[simp]*:
fixes $a\ b :: \text{ereal}$
shows $\text{real-of-ereal } (a * b) = \text{real-of-ereal } a * \text{real-of-ereal } b$
 $\langle \text{proof} \rangle$

lemma *real-of-ereal-eq-0*:
fixes $x :: \text{ereal}$
shows $\text{real-of-ereal } x = 0 \longleftrightarrow x = \infty \vee x = -\infty \vee x = 0$
 $\langle \text{proof} \rangle$

lemma *tendsto-ereal-realD*:
fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $x \neq 0$
and $\text{tendsto}: ((\lambda x. \text{ereal } (\text{real-of-ereal } (f\ x))) \longrightarrow x) \text{ net}$
shows $(f \longrightarrow x) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *tendsto-ereal-realI*:
fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $x: |x| \neq \infty$ **and** $\text{tendsto}: (f \longrightarrow x) \text{ net}$
shows $((\lambda x. \text{ereal } (\text{real-of-ereal } (f\ x))) \longrightarrow x) \text{ net}$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-cancel-left*:
fixes $a\ b\ c :: \text{ereal}$
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
 $\langle \text{proof} \rangle$

lemma *tendsto-add-ereal*:
fixes $x\ y :: \text{ereal}$
assumes $x: |x| \neq \infty$ **and** $y: |y| \neq \infty$
assumes $f: (f \longrightarrow x) F$ **and** $g: (g \longrightarrow y) F$
shows $((\lambda x. f\ x + g\ x) \longrightarrow x + y) F$
 $\langle \text{proof} \rangle$

lemma *tendsto-add-ereal-nonneg*:
fixes $x\ y :: \text{ereal}$
assumes $x \neq -\infty$ $y \neq -\infty$ $(f \longrightarrow x) F$ $(g \longrightarrow y) F$
shows $((\lambda x. f\ x + g\ x) \longrightarrow x + y) F$
 $\langle \text{proof} \rangle$

lemma *ereal-inj-affinity*:
fixes $m\ t :: \text{ereal}$
assumes $|m| \neq \infty$
and $m \neq 0$

and $|t| \neq \infty$
shows *inj-on* $(\lambda x. m * x + t) A$
 $\langle \text{proof} \rangle$

lemma *ereal-PInfty-eq-plus[simp]*:
fixes $a b :: \text{ereal}$
shows $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$
 $\langle \text{proof} \rangle$

lemma *ereal-MInfty-eq-plus[simp]*:
fixes $a b :: \text{ereal}$
shows $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$
 $\langle \text{proof} \rangle$

lemma *ereal-less-divide-pos*:
fixes $x y :: \text{ereal}$
shows $x > 0 \implies x \neq \infty \implies y < z / x \longleftrightarrow x * y < z$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-less-pos*:
fixes $x y z :: \text{ereal}$
shows $x > 0 \implies x \neq \infty \implies y / x < z \longleftrightarrow y < x * z$
 $\langle \text{proof} \rangle$

lemma *ereal-divide-eq*:
fixes $a b c :: \text{ereal}$
shows $b \neq 0 \implies |b| \neq \infty \implies a / b = c \longleftrightarrow a = b * c$
 $\langle \text{proof} \rangle$

lemma *ereal-inverse-not-MInfty[simp]*: *inverse* $(a :: \text{ereal}) \neq -\infty$
 $\langle \text{proof} \rangle$

lemma *ereal-mult-m1[simp]*: $x * \text{ereal } (-1) = -x$
 $\langle \text{proof} \rangle$

lemma *ereal-real'*:
assumes $|x| \neq \infty$
shows $\text{ereal } (\text{real-of-ereal } x) = x$
 $\langle \text{proof} \rangle$

lemma *real-ereal-id*: $\text{real-of-ereal} \circ \text{ereal} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *open-image-ereal*: $\text{open}(UNIV - \{ \infty, (-\infty :: \text{ereal}) \})$
 $\langle \text{proof} \rangle$

lemma *ereal-le-distrib*:
fixes $a b c :: \text{ereal}$
shows $c * (a + b) \leq c * a + c * b$

$\langle \text{proof} \rangle$

lemma *ereal-pos-distrib*:

fixes $a\ b\ c :: \text{ereal}$

assumes $0 \leq c$

and $c \neq \infty$

shows $c * (a + b) = c * a + c * b$

$\langle \text{proof} \rangle$

lemma *ereal-LimI-finite*:

fixes $x :: \text{ereal}$

assumes $|x| \neq \infty$

and $\bigwedge r. 0 < r \implies \exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r$

shows $u \longrightarrow x$

$\langle \text{proof} \rangle$

lemma *tendsto-obtains-N*:

assumes $f \longrightarrow f0$ *open* S $f0 \in S$

obtains N **where** $\forall n \geq N. f\ n \in S$

$\langle \text{proof} \rangle$

lemma *ereal-LimI-finite-iff*:

fixes $x :: \text{ereal}$

assumes $|x| \neq \infty$

shows $u \longrightarrow x \longleftrightarrow (\forall r. 0 < r \longrightarrow (\exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r))$

(is ?lhs \longleftrightarrow ?rhs)

$\langle \text{proof} \rangle$

lemma *ereal-Limsup-uminus*:

fixes $f :: 'a \Rightarrow \text{ereal}$

shows $\text{Limsup}\ \text{net}\ (\lambda x. - (f\ x)) = -\ \text{Liminf}\ \text{net}\ f$

$\langle \text{proof} \rangle$

lemma *liminf-bounded-iff*:

fixes $x :: \text{nat} \Rightarrow \text{ereal}$

shows $C \leq \text{liminf}\ x \longleftrightarrow (\forall B < C. \exists N. \forall n \geq N. B < x\ n)$

(is ?lhs \longleftrightarrow ?rhs)

$\langle \text{proof} \rangle$

lemma *Liminf-add-le*:

fixes $f\ g :: - \Rightarrow \text{ereal}$

assumes $F: F \neq \text{bot}$

assumes *ev*: *eventually* $(\lambda x. 0 \leq f\ x)$ *F* *eventually* $(\lambda x. 0 \leq g\ x)$ *F*

shows $\text{Liminf}\ F\ f + \text{Liminf}\ F\ g \leq \text{Liminf}\ F\ (\lambda x. f\ x + g\ x)$

$\langle \text{proof} \rangle$

lemma *Sup-ereal-mult-right'*:

assumes *nonempty*: $Y \neq \{\}$

and $x: x \geq 0$
shows $(\text{SUP } i \in Y. f\ i) * \text{ereal } x = (\text{SUP } i \in Y. f\ i * \text{ereal } x)$ (**is** $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

lemma *Sup-ereal-mult-left'*:
 $\llbracket Y \neq \{\}; x \geq 0 \rrbracket \implies \text{ereal } x * (\text{SUP } i \in Y. f\ i) = (\text{SUP } i \in Y. \text{ereal } x * f\ i)$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-add[order-continuous-intros]*:
fixes $f\ g :: 'a :: \text{complete-lattice} \Rightarrow \text{ereal}$
assumes $nn: \bigwedge x. 0 \leq f\ x \bigwedge x. 0 \leq g\ x$ **and** $cont: \text{sup-continuous } f\ \text{sup-continuous } g$
shows $\text{sup-continuous } (\lambda x. f\ x + g\ x)$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-mult-right[order-continuous-intros]*:
 $0 \leq c \implies c < \infty \implies \text{sup-continuous } f \implies \text{sup-continuous } (\lambda x. f\ x * c :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-mult-left[order-continuous-intros]*:
 $0 \leq c \implies c < \infty \implies \text{sup-continuous } f \implies \text{sup-continuous } (\lambda x. c * f\ x :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-ereal-of-enat[order-continuous-intros]*:
assumes $f: \text{sup-continuous } f$
shows $\text{sup-continuous } (\lambda x. \text{ereal-of-enat } (f\ x))$
 $\langle \text{proof} \rangle$

38.6.2 Sums

lemma *sums-ereal-positive*:
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. 0 \leq f\ i$
shows $f\ \text{sums } (\text{SUP } n. \sum i < n. f\ i)$
 $\langle \text{proof} \rangle$

lemma *summable-ereal-pos*:
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. 0 \leq f\ i$
shows $\text{summable } f$
 $\langle \text{proof} \rangle$

lemma *sums-ereal*: $(\lambda x. \text{ereal } (f\ x))\ \text{sums } \text{ereal } x \longleftrightarrow f\ \text{sums } x$
 $\langle \text{proof} \rangle$

lemma *suminf-ereal-eq-SUP*:
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge i. 0 \leq f\ i$
shows $(\sum x. f\ x) = (\text{SUP } n. \sum i < n. f\ i)$

$\langle \text{proof} \rangle$

lemma *suminf-bound*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes $\forall N. (\sum n < N. f\ n) \leq x \wedge n. 0 \leq f\ n$

shows $\text{suminf}\ f \leq x$

$\langle \text{proof} \rangle$

lemma *suminf-bound-add*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes $\forall N. (\sum n < N. f\ n) + y \leq x$

and $\wedge n. 0 \leq f\ n$

and $y \neq -\infty$

shows $\text{suminf}\ f + y \leq x$

$\langle \text{proof} \rangle$

lemma *suminf-upper*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes $\wedge n. 0 \leq f\ n$

shows $(\sum n < N. f\ n) \leq (\sum n. f\ n)$

$\langle \text{proof} \rangle$

lemma *suminf-0-le*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes $\wedge n. 0 \leq f\ n$

shows $0 \leq (\sum n. f\ n)$

$\langle \text{proof} \rangle$

lemma *suminf-le-pos*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ereal}$

assumes $\wedge N. f\ N \leq g\ N$

and $\wedge N. 0 \leq f\ N$

shows $\text{suminf}\ f \leq \text{suminf}\ g$

$\langle \text{proof} \rangle$

lemma *suminf-half-series-ereal*: $(\sum n. (1/2 :: \text{ereal}) \wedge \text{Suc}\ n) = 1$

$\langle \text{proof} \rangle$

lemma *suminf-add-ereal*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ereal}$

assumes $\wedge i. 0 \leq f\ i \wedge i. 0 \leq g\ i$

shows $(\sum i. f\ i + g\ i) = \text{suminf}\ f + \text{suminf}\ g$

$\langle \text{proof} \rangle$

lemma *suminf-cmult-ereal*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ereal}$

assumes $\wedge i. 0 \leq f\ i$ **and** $0 \leq a$

shows $(\sum i. a * f\ i) = a * \text{suminf}\ f$

$\langle \text{proof} \rangle$

lemma *suminf-PInf*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes $\bigwedge i. 0 \leq f\ i$

and $\text{suminf}\ f \neq \infty$

shows $f\ i \neq \infty$

$\langle \text{proof} \rangle$

lemma *suminf-PInf-fun*:

assumes $\bigwedge i. 0 \leq f\ i$

and $\text{suminf}\ f \neq \infty$

shows $\exists f'. f = (\lambda x. \text{ereal}\ (f'\ x))$

$\langle \text{proof} \rangle$

lemma *summable-ereal*:

assumes $\bigwedge i. 0 \leq f\ i$

and $(\sum i. \text{ereal}\ (f\ i)) \neq \infty$

shows *summable* f

$\langle \text{proof} \rangle$

lemma *suminf-ereal*:

assumes $\bigwedge i. 0 \leq f\ i$

and $(\sum i. \text{ereal}\ (f\ i)) \neq \infty$

shows $(\sum i. \text{ereal}\ (f\ i)) = \text{ereal}\ (\text{suminf}\ f)$

$\langle \text{proof} \rangle$

lemma *suminf-ereal-minus*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ereal}$

assumes *ord*: $\bigwedge i. g\ i \leq f\ i \wedge i. 0 \leq g\ i$

and *fin*: $\text{suminf}\ f \neq \infty \wedge \text{suminf}\ g \neq \infty$

shows $(\sum i. f\ i - g\ i) = \text{suminf}\ f - \text{suminf}\ g$

$\langle \text{proof} \rangle$

lemma *suminf-ereal-PInf [simp]*: $(\sum x. \infty :: \text{ereal}) = \infty$

$\langle \text{proof} \rangle$

lemma *summable-real-of-ereal*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes *f*: $\bigwedge i. 0 \leq f\ i$

and *fin*: $(\sum i. f\ i) \neq \infty$

shows *summable* $(\lambda i. \text{real-of-ereal}\ (f\ i))$

$\langle \text{proof} \rangle$

lemma *suminf-SUP-eq*:

fixes $f :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ereal}$

assumes $\bigwedge i. \text{incseq}\ (\lambda n. f\ n\ i)$

and $\bigwedge n\ i. 0 \leq f\ n\ i$

shows $(\sum i. \text{SUP}\ n. f\ n\ i) = (\text{SUP}\ n. \sum i. f\ n\ i)$

$\langle \text{proof} \rangle$

lemma *suminf-sum-ereal*:

fixes $f :: - \Rightarrow - \Rightarrow \text{ereal}$

assumes *nonneg*: $\bigwedge i. a. a \in A \implies 0 \leq f\ i\ a$

shows $(\sum i. \sum_{a \in A}. f\ i\ a) = (\sum_{a \in A}. \sum i. f\ i\ a)$

$\langle \text{proof} \rangle$

lemma *suminf-ereal-eq-0*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes *nneg*: $\bigwedge i. 0 \leq f\ i$

shows $(\sum i. f\ i) = 0 \iff (\forall i. f\ i = 0)$

$\langle \text{proof} \rangle$

lemma *suminf-ereal-offset-le*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes *f*: $\bigwedge i. 0 \leq f\ i$

shows $(\sum i. f\ (i + k)) \leq \text{suminf}\ f$

$\langle \text{proof} \rangle$

lemma *sums-suminf-ereal*: $f\ \text{sums}\ x \implies (\sum i. \text{ereal}\ (f\ i)) = \text{ereal}\ x$

$\langle \text{proof} \rangle$

lemma *suminf-ereal'*: $\text{summable}\ f \implies (\sum i. \text{ereal}\ (f\ i)) = \text{ereal}\ (\sum i. f\ i)$

$\langle \text{proof} \rangle$

lemma *suminf-ereal-finite*: $\text{summable}\ f \implies (\sum i. \text{ereal}\ (f\ i)) \neq \infty$

$\langle \text{proof} \rangle$

lemma *suminf-ereal-finite-neg*:

assumes *summable* f

shows $(\sum x. \text{ereal}\ (f\ x)) \neq -\infty$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-add-directed*:

fixes $f\ g :: 'a \Rightarrow \text{ereal}$

assumes *nonneg*: $\bigwedge i. i \in I \implies 0 \leq f\ i \wedge i \in I \implies 0 \leq g\ i$

assumes *directed*: $\bigwedge i\ j. i \in I \implies j \in I \implies \exists k \in I. f\ i + g\ j \leq f\ k + g\ k$

shows $(\text{SUP } i \in I. f\ i + g\ i) = (\text{SUP } i \in I. f\ i) + (\text{SUP } i \in I. g\ i)$

$\langle \text{proof} \rangle$

lemma *SUP-ereal-sum-directed*:

fixes $f\ g :: 'a \Rightarrow 'b \Rightarrow \text{ereal}$

assumes $I \neq \{\}$

assumes *directed*: $\bigwedge N\ i\ j. N \subseteq A \implies i \in I \implies j \in I \implies \exists k \in I. \forall n \in N. f\ n\ i \leq f\ n\ k \wedge f\ n\ j \leq f\ n\ k$

assumes *nonneg*: $\bigwedge n\ i. i \in I \implies n \in A \implies 0 \leq f\ n\ i$

shows $(\text{SUP } i \in I. \sum n \in A. f\ n\ i) = (\sum n \in A. \text{SUP } i \in I. f\ n\ i)$

$\langle \text{proof} \rangle$

lemma *suminf-SUP-eq-directed*:

fixes $f :: - \Rightarrow \text{nat} \Rightarrow \text{ereal}$
assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N i j. i \in I \implies j \in I \implies \text{finite } N \implies \exists k \in I. \forall n \in N. f i n \leq f k n \wedge f j n \leq f k n$
assumes *nonneg*: $\bigwedge n i. 0 \leq f n i$
shows $(\sum i. \text{SUP } n \in I. f n i) = (\text{SUP } n \in I. \sum i. f n i)$
 $\langle \text{proof} \rangle$

lemma *ereal-dense3*:

fixes $x y :: \text{ereal}$
shows $x < y \implies \exists r :: \text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$
 $\langle \text{proof} \rangle$

lemma *continuous-within-ereal*[*intro, simp*]: $x \in A \implies \text{continuous (at } x \text{ within } A)$
 ereal
 $\langle \text{proof} \rangle$

lemma *ereal-open-uminus*:

fixes $S :: \text{ereal set}$
assumes *open* S
shows *open* $(\text{uminus } S)$
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-complement*:

fixes $S :: \text{ereal set}$
shows $\text{uminus } (- S) = - \text{uminus } S$
 $\langle \text{proof} \rangle$

lemma *ereal-closed-uminus*:

fixes $S :: \text{ereal set}$
assumes *closed* S
shows *closed* $(\text{uminus } S)$
 $\langle \text{proof} \rangle$

lemma *ereal-open-affinity-pos*:

fixes $S :: \text{ereal set}$
assumes *open* S
and $m: m \neq \infty \ 0 < m$
and $t: |t| \neq \infty$
shows *open* $((\lambda x. m * x + t) S)$
 $\langle \text{proof} \rangle$

lemma *ereal-open-affinity*:

fixes $S :: \text{ereal set}$
assumes *open* S
and $m: |m| \neq \infty \ m \neq 0$
and $t: |t| \neq \infty$
shows *open* $((\lambda x. m * x + t) S)$

$\langle proof \rangle$

lemma *open-uminus-iff*:

fixes $S :: ereal\ set$

shows $open\ (uminus\ 'S) \longleftrightarrow open\ S$

$\langle proof \rangle$

lemma *ereal-Liminf-uminus*:

fixes $f :: 'a \Rightarrow ereal$

shows $Liminf\ net\ (\lambda x. -\ (f\ x)) = -\ Limsup\ net\ f$

$\langle proof \rangle$

lemma *Liminf-PInfy*:

fixes $f :: 'a \Rightarrow ereal$

assumes $\neg\ trivial_limit\ net$

shows $(f \longrightarrow \infty)\ net \longleftrightarrow Liminf\ net\ f = \infty$

$\langle proof \rangle$

lemma *Limsup-MInfy*:

fixes $f :: 'a \Rightarrow ereal$

assumes $\neg\ trivial_limit\ net$

shows $(f \longrightarrow -\infty)\ net \longleftrightarrow Limsup\ net\ f = -\infty$

$\langle proof \rangle$

lemma *convergent-ereal*: — RENAME

fixes $X :: nat \Rightarrow 'a :: \{complete_linorder, linorder_topology\}$

shows $convergent\ X \longleftrightarrow limsup\ X = liminf\ X$

$\langle proof \rangle$

lemma *limsup-le-liminf-real*:

fixes $X :: nat \Rightarrow real$ **and** $L :: real$

assumes $1: limsup\ X \leq L$ **and** $2: L \leq liminf\ X$

shows $X \longrightarrow L$

$\langle proof \rangle$

lemma *liminf-PInfy*:

fixes $X :: nat \Rightarrow ereal$

shows $X \longrightarrow \infty \longleftrightarrow liminf\ X = \infty$

$\langle proof \rangle$

lemma *limsup-MInfy*:

fixes $X :: nat \Rightarrow ereal$

shows $X \longrightarrow -\infty \longleftrightarrow limsup\ X = -\infty$

$\langle proof \rangle$

lemma *SUP-eq-LIMSEQ*:

assumes $mono\ f$

shows $(SUP\ n. ereal\ (f\ n)) = ereal\ x \longleftrightarrow f \longrightarrow x$

$\langle proof \rangle$

lemma *liminf-ereal-cminus*:
fixes $f :: \text{nat} \Rightarrow \text{ereal}$
assumes $c \neq -\infty$
shows $\text{liminf } (\lambda x. c - f x) = c - \text{limsup } f$
 $\langle \text{proof} \rangle$

38.6.3 Continuity

lemma *continuous-at-of-ereal*:
 $|x0 :: \text{ereal}| \neq \infty \implies \text{continuous } (\text{at } x0) \text{ real-of-ereal}$
 $\langle \text{proof} \rangle$

lemma *nhds-ereal*: $\text{nhds } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{nhds } r)$
 $\langle \text{proof} \rangle$

lemma *at-ereal*: $\text{at } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at } r)$
 $\langle \text{proof} \rangle$

lemma *at-left-ereal*: $\text{at-left } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at-left } r)$
 $\langle \text{proof} \rangle$

lemma *at-right-ereal*: $\text{at-right } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at-right } r)$
 $\langle \text{proof} \rangle$

lemma
shows *at-left-PIInf*: $\text{at-left } \infty = \text{filtermap } \text{ereal } \text{at-top}$
and *at-right-MInf*: $\text{at-right } (-\infty) = \text{filtermap } \text{ereal } \text{at-bot}$
 $\langle \text{proof} \rangle$

lemma *ereal-tendsto-simps1*:
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-left } (\text{ereal } x)) \longleftrightarrow (f \longrightarrow y) (\text{at-left } x)$
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-right } (\text{ereal } x)) \longleftrightarrow (f \longrightarrow y) (\text{at-right } x)$
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-left } (\infty :: \text{ereal})) \longleftrightarrow (f \longrightarrow y) \text{at-top}$
 $((f \circ \text{real-of-ereal}) \longrightarrow y) (\text{at-right } (-\infty :: \text{ereal})) \longleftrightarrow (f \longrightarrow y) \text{at-bot}$
 $\langle \text{proof} \rangle$

lemma *ereal-tendsto-simps2*:
 $((\text{ereal} \circ f) \longrightarrow \text{ereal } a) F \longleftrightarrow (f \longrightarrow a) F$
 $((\text{ereal} \circ f) \longrightarrow \infty) F \longleftrightarrow (\text{LIM } x F. f x :> \text{at-top})$
 $((\text{ereal} \circ f) \longrightarrow -\infty) F \longleftrightarrow (\text{LIM } x F. f x :> \text{at-bot})$
 $\langle \text{proof} \rangle$

lemma *inverse-infty-ereal-tendsto-0*: $\text{inverse } -\infty \rightarrow (0 :: \text{ereal})$
 $\langle \text{proof} \rangle$

lemma *inverse-ereal-tendsto-pos*:
fixes $x :: \text{ereal}$ **assumes** $0 < x$
shows $\text{inverse } -x \rightarrow \text{inverse } x$

$\langle \text{proof} \rangle$

lemma *inverse-ereal-tendsto-at-right-0*: $(\text{inverse} \longrightarrow \infty) (\text{at-right } (0::\text{ereal}))$
 $\langle \text{proof} \rangle$

lemmas *ereal-tendsto-simps* = *ereal-tendsto-simps1* *ereal-tendsto-simps2*

lemma *continuous-at-iff-ereal*:
fixes $f :: 'a::t2\text{-space} \Rightarrow \text{real}$
shows $\text{continuous } (\text{at } x0 \text{ within } s) f \longleftrightarrow \text{continuous } (\text{at } x0 \text{ within } s) (\text{ereal} \circ f)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-iff-ereal*:
fixes $f :: 'a::t2\text{-space} \Rightarrow \text{real}$
assumes *open A*
shows $\text{continuous-on } A f \longleftrightarrow \text{continuous-on } A (\text{ereal} \circ f)$
 $\langle \text{proof} \rangle$

lemma *continuous-on-real*: $\text{continuous-on } (\text{UNIV} - \{\infty, -\infty::\text{ereal}\}) \text{real-of-ereal}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-iff-real*:
fixes $f :: 'a::t2\text{-space} \Rightarrow \text{ereal}$
assumes $\bigwedge x. x \in A \implies |f x| \neq \infty$
shows $\text{continuous-on } A f \longleftrightarrow \text{continuous-on } A (\text{real-of-ereal} \circ f)$
 $\langle \text{proof} \rangle$

lemma *continuous-uminus-ereal* [*continuous-intros*]: $\text{continuous-on } (A :: \text{ereal set})$
 uminus
 $\langle \text{proof} \rangle$

lemma *ereal-uminus-atMost* [*simp*]: $\text{uminus } \{..(a::\text{ereal})\} = \{-a..\}$
 $\langle \text{proof} \rangle$

lemma *continuous-on-inverse-ereal* [*continuous-intros*]:
 $\text{continuous-on } \{0::\text{ereal} ..\} \text{inverse}$
 $\langle \text{proof} \rangle$

lemma *continuous-inverse-ereal-nonpos*: $\text{continuous-on } (\{..<0\} :: \text{ereal set}) \text{inverse}$
 $\langle \text{proof} \rangle$

lemma *tendsto-inverse-ereal*:
assumes $(f \longrightarrow (c :: \text{ereal})) F$
assumes *eventually* $(\lambda x. f x \geq 0) F$
shows $((\lambda x. \text{inverse } (f x)) \longrightarrow \text{inverse } c) F$
 $\langle \text{proof} \rangle$

38.6.4 liminf and limsup**lemma** *Limsup-ereal-mult-right:***assumes** $F \neq \text{bot } (c::\text{real}) \geq 0$ **shows** $\text{Limsup } F (\lambda n. f n * \text{ereal } c) = \text{Limsup } F f * \text{ereal } c$ $\langle \text{proof} \rangle$ **lemma** *Liminf-ereal-mult-right:***assumes** $F \neq \text{bot } (c::\text{real}) \geq 0$ **shows** $\text{Liminf } F (\lambda n. f n * \text{ereal } c) = \text{Liminf } F f * \text{ereal } c$ $\langle \text{proof} \rangle$ **lemma** *Liminf-ereal-mult-left:***assumes** $F \neq \text{bot } (c::\text{real}) \geq 0$ **shows** $\text{Liminf } F (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{Liminf } F f$ $\langle \text{proof} \rangle$ **lemma** *Limsup-ereal-mult-left:***assumes** $F \neq \text{bot } (c::\text{real}) \geq 0$ **shows** $\text{Limsup } F (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{Limsup } F f$ $\langle \text{proof} \rangle$ **lemma** *limsup-ereal-mult-right:* $(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. f n * \text{ereal } c) = \text{limsup } f * \text{ereal } c$ $\langle \text{proof} \rangle$ **lemma** *limsup-ereal-mult-left:* $(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{limsup } f$ $\langle \text{proof} \rangle$ **lemma** *Limsup-add-ereal-right:* $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. g n + (c :: \text{ereal})) = \text{Limsup } F g +$ c $\langle \text{proof} \rangle$ **lemma** *Limsup-add-ereal-left:* $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Limsup } F$ g $\langle \text{proof} \rangle$ **lemma** *Liminf-add-ereal-right:* $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. g n + (c :: \text{ereal})) = \text{Liminf } F g + c$ $\langle \text{proof} \rangle$ **lemma** *Liminf-add-ereal-left:* $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Liminf } F g$ $\langle \text{proof} \rangle$ **lemma****assumes** $F \neq \text{bot}$

assumes *nonneg*: *eventually* $(\lambda x. f\ x \geq (0 :: ereal))\ F$
shows *Liminf-inverse-ereal*: $\text{Liminf } F\ (\lambda x. \text{inverse } (f\ x)) = \text{inverse } (\text{Limsup } F\ f)$
and *Limsup-inverse-ereal*: $\text{Limsup } F\ (\lambda x. \text{inverse } (f\ x)) = \text{inverse } (\text{Liminf } F\ f)$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-le-mono-left*: $\llbracket x \leq z; 0 \leq y \rrbracket \implies x - y \leq (z :: ereal)$
 $\langle \text{proof} \rangle$

lemma *neg-0-less-iff-less-erea* [*simp*]: $0 < -\ a \longleftrightarrow (a :: ereal) < 0$
 $\langle \text{proof} \rangle$

lemma *not-infity-ereal*: $|x| \neq \infty \longleftrightarrow (\exists x'. x = ereal\ x')$
 $\langle \text{proof} \rangle$

lemma *neg-PInf-trans*: **fixes** $x\ y :: ereal$ **shows** $\llbracket y \neq \infty; x \leq y \rrbracket \implies x \neq \infty$
 $\langle \text{proof} \rangle$

lemma *mult-2-ereal*: $ereal\ 2 * x = x + x$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-le-self*: $0 \leq y \implies x - y \leq (x :: ereal)$
 $\langle \text{proof} \rangle$

lemma *ereal-le-add-self*: $0 \leq y \implies x \leq x + (y :: ereal)$
 $\langle \text{proof} \rangle$

lemma *ereal-le-add-self2*: $0 \leq y \implies x \leq y + (x :: ereal)$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-nonpos*:
fixes $a\ b :: ereal$ **shows** $\llbracket a \leq b; a = \infty \implies b \neq \infty; a = -\infty \implies b \neq -\infty \rrbracket$
 $\implies a - b \leq 0$
 $\langle \text{proof} \rangle$

lemma *minus-ereal-0* [*simp*]: $x - ereal\ 0 = x$
 $\langle \text{proof} \rangle$

lemma *ereal-diff-eq-0-iff*: **fixes** $a\ b :: ereal$
shows $(|a| = \infty \implies |b| \neq \infty) \implies a - b = 0 \longleftrightarrow a = b$
 $\langle \text{proof} \rangle$

lemma *SUP-ereal-eq-0-iff-nonneg*:
fixes $f :: - \Rightarrow ereal$ **and** A
assumes *nonneg*: $\forall x \in A. f\ x \geq 0$
and $A : A \neq \{\}$
shows $(\text{SUP } x \in A. f\ x) = 0 \longleftrightarrow (\forall x \in A. f\ x = 0)$ (**is** *?lhs* \longleftrightarrow -)
 $\langle \text{proof} \rangle$

lemma *ereal-divide-le-posI*:

fixes $x\ y\ z :: \text{ereal}$

shows $x > 0 \implies z \neq -\infty \implies z \leq x * y \implies z / x \leq y$

$\langle \text{proof} \rangle$

lemma *add-diff-eq-ereal*:

fixes $x\ y\ z :: \text{ereal}$

shows $x + (y - z) = x + y - z$

$\langle \text{proof} \rangle$

lemma *ereal-diff-gr0*:

fixes $a\ b :: \text{ereal}$

shows $a < b \implies 0 < b - a$

$\langle \text{proof} \rangle$

lemma *ereal-minus-minus*:

fixes $x\ y\ z :: \text{ereal}$ **shows**

$(|y| = \infty \implies |z| \neq \infty) \implies x - (y - z) = x + z - y$

$\langle \text{proof} \rangle$

lemma *diff-diff-commute-ereal*:

fixes $x\ y\ z :: \text{ereal}$

shows $x - y - z = x - z - y$

$\langle \text{proof} \rangle$

lemma *ereal-diff-eq-MInfty-iff*:

fixes $x\ y :: \text{ereal}$

shows $x - y = -\infty \longleftrightarrow x = -\infty \wedge y \neq -\infty \vee y = \infty \wedge |x| \neq \infty$

$\langle \text{proof} \rangle$

lemma *ereal-diff-add-inverse*:

fixes $x\ y :: \text{ereal}$

shows $|x| \neq \infty \implies x + y - x = y$

$\langle \text{proof} \rangle$

lemma *tendsto-diff-ereal*:

fixes $x\ y :: \text{ereal}$

assumes $x: |x| \neq \infty$ **and** $y: |y| \neq \infty$

assumes $f: (f \longrightarrow x) F$ **and** $g: (g \longrightarrow y) F$

shows $((\lambda x. f\ x - g\ x) \longrightarrow x - y) F$

$\langle \text{proof} \rangle$

lemma *continuous-on-diff-ereal*:

$\text{continuous-on } A\ f \implies \text{continuous-on } A\ g \implies (\bigwedge x. x \in A \implies |f\ x| \neq \infty) \implies$

$(\bigwedge x. x \in A \implies |g\ x| \neq \infty) \implies \text{continuous-on } A\ (\lambda z. f\ z - g\ z :: \text{ereal})$

$\langle \text{proof} \rangle$

38.6.5 Tests for code generator

A small list of simple arithmetic expressions.

```

value  $-\infty :: \text{ereal}$ 
value  $|-\infty| :: \text{ereal}$ 
value  $4 + 5 / 4 - \text{ereal } 2 :: \text{ereal}$ 
value  $\text{ereal } 3 < \infty$ 
value  $\text{real-of-ereal } (\infty :: \text{ereal}) = 0$ 

```

end

39 Indicator Function

```

theory Indicator-Function
imports Complex-Main Disjoint-Sets
begin

```

definition $\text{indicator } S \ x = \text{of-bool } (x \in S)$

Type constrained version

abbreviation $\text{indicat-real} :: 'a \text{ set} \Rightarrow 'a \Rightarrow \text{real}$ **where** $\text{indicat-real } S \equiv \text{indicator } S$

lemma $\text{indicator-simps}[\text{simp}]$:
 $x \in S \Longrightarrow \text{indicator } S \ x = 1$
 $x \notin S \Longrightarrow \text{indicator } S \ x = 0$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-pos-le}[\text{intro}, \text{simp}]$: $(0 :: 'a :: \text{linordered-semidom}) \leq \text{indicator } S \ x$
and $\text{indicator-le-1}[\text{intro}, \text{simp}]$: $\text{indicator } S \ x \leq (1 :: 'a :: \text{linordered-semidom})$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-abs-le-1}$: $|\text{indicator } S \ x| \leq (1 :: 'a :: \text{linordered-idom})$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-eq-0-iff}$: $\text{indicator } A \ x = (0 :: 'a :: \text{zero-neq-one}) \longleftrightarrow x \notin A$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-eq-1-iff}$: $\text{indicator } A \ x = (1 :: 'a :: \text{zero-neq-one}) \longleftrightarrow x \in A$
 $\langle \text{proof} \rangle$

lemma $\text{indicator-UNIV} [\text{simp}]$: $\text{indicator } \text{UNIV} = (\lambda x. 1)$
 $\langle \text{proof} \rangle$

lemma indicator-leI :
 $(x \in A \Longrightarrow y \in B) \Longrightarrow (\text{indicator } A \ x :: 'a :: \text{linordered-nonzero-semiring}) \leq \text{indicator } B \ y$
 $\langle \text{proof} \rangle$

lemma *split-indicator*: $P \text{ (indicator } S \text{ } x) \longleftrightarrow ((x \in S \longrightarrow P \text{ } 1) \wedge (x \notin S \longrightarrow P \text{ } 0))$

<proof>

lemma *split-indicator-asm*: $P \text{ (indicator } S \text{ } x) \longleftrightarrow (\neg (x \in S \wedge \neg P \text{ } 1 \vee x \notin S \wedge \neg P \text{ } 0))$

<proof>

lemma *indicator-inter-arith*: $\text{indicator } (A \cap B) \text{ } x = \text{indicator } A \text{ } x * (\text{indicator } B \text{ } x :: 'a::\text{semiring-1})$

<proof>

lemma *indicator-union-arith*:

$\text{indicator } (A \cup B) \text{ } x = \text{indicator } A \text{ } x + \text{indicator } B \text{ } x - \text{indicator } A \text{ } x * (\text{indicator } B \text{ } x :: 'a::\text{ring-1})$

<proof>

lemma *indicator-inter-min*: $\text{indicator } (A \cap B) \text{ } x = \min (\text{indicator } A \text{ } x) (\text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

and *indicator-union-max*: $\text{indicator } (A \cup B) \text{ } x = \max (\text{indicator } A \text{ } x) (\text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

<proof>

lemma *indicator-disj-union*:

$A \cap B = \{\} \implies \text{indicator } (A \cup B) \text{ } x = (\text{indicator } A \text{ } x + \text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

<proof>

lemma *indicator-compl*: $\text{indicator } (- A) \text{ } x = 1 - (\text{indicator } A \text{ } x :: 'a::\text{ring-1})$

and *indicator-diff*: $\text{indicator } (A - B) \text{ } x = \text{indicator } A \text{ } x * (1 - \text{indicator } B \text{ } x :: 'a::\text{ring-1})$

<proof>

lemma *indicator-times*:

$\text{indicator } (A \times B) \text{ } x = \text{indicator } A \text{ } (\text{fst } x) * (\text{indicator } B \text{ } (\text{snd } x) :: 'a::\text{semiring-1})$

<proof>

lemma *indicator-sum*:

$\text{indicator } (A <+> B) \text{ } x = (\text{case } x \text{ of } \text{Inl } x \Rightarrow \text{indicator } A \text{ } x \mid \text{Inr } x \Rightarrow \text{indicator } B \text{ } x)$

<proof>

lemma *indicator-image*: $\text{inj } f \implies \text{indicator } (f ' X) \text{ } (f x) = (\text{indicator } X \text{ } x :: \text{zero-neq-one})$

<proof>

lemma *indicator-vimage*: $\text{indicator } (f - ' A) \text{ } x = \text{indicator } A \text{ } (f x)$

<proof>

lemma *mult-indicator-cong*:

fixes $f\ g :: - \Rightarrow 'a :: \text{semiring-1}$
shows $(\bigwedge x. x \in A \implies f\ x = g\ x) \implies \text{indicator } A\ x * f\ x = \text{indicator } A\ x * g\ x$
 $\langle \text{proof} \rangle$

lemma

fixes $f :: 'a \Rightarrow 'b :: \text{semiring-1}$
assumes $\text{finite } A$
shows $\text{sum-mult-indicator}[simp]: (\sum x \in A. f\ x * \text{indicator } B\ x) = (\sum x \in A \cap B. f\ x)$
and $\text{sum-indicator-mult}[simp]: (\sum x \in A. \text{indicator } B\ x * f\ x) = (\sum x \in A \cap B. f\ x)$
 $\langle \text{proof} \rangle$

lemma $\text{sum-indicator-eq-card}$:

assumes $\text{finite } A$
shows $(\sum x \in A. \text{indicator } B\ x) = \text{card } (A \text{ Int } B)$
 $\langle \text{proof} \rangle$

lemma $\text{sum-indicator-scaleR}[simp]$:

$\text{finite } A \implies$
 $(\sum x \in A. \text{indicator } (B\ x)\ (g\ x) *_{\mathbb{R}} f\ x) = (\sum x \in \{x \in A. g\ x \in B\ x\}. f\ x :: 'a :: \text{real-vector})$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-incseq}$:

assumes $\text{incseq } A$
shows $(\lambda i. \text{indicator } (A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcup i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-UN}$:

$(\lambda k. \text{indicator } (\bigcup i < k. A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcup i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-decseq}$:

assumes $\text{decseq } A$
shows $(\lambda i. \text{indicator } (A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcap i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma $\text{LIMSEQ-indicator-INT}$:

$(\lambda k. \text{indicator } (\bigcap i < k. A\ i)\ x :: 'a :: \{\text{topological-space}, \text{zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcap i. A\ i)\ x$
 $\langle \text{proof} \rangle$

lemma indicator-add :

$A \cap B = \{\} \implies (\text{indicator } A\ x :: \text{monoid-add}) + \text{indicator } B\ x = \text{indicator } (A \cup B)\ x$

<proof>

lemma *of-real-indicator*: *of-real (indicator A x) = indicator A x*
<proof>

lemma *real-of-nat-indicator*: *real (indicator A x :: nat) = indicator A x*
<proof>

lemma *abs-indicator*: *|indicator A x :: 'a::linordered-idom| = indicator A x*
<proof>

lemma *mult-indicator-subset*:
 $A \subseteq B \implies \text{indicator } A \ x * \text{indicator } B \ x = (\text{indicator } A \ x :: 'a::\text{comm-semiring-1})$
<proof>

lemma *indicator-times-eq-if*:
fixes $f :: 'a \Rightarrow 'b::\text{comm-ring-1}$
shows $\text{indicator } S \ x * f \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0) \ f \ x * \text{indicator } S \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0)$
<proof>

lemma *indicator-scaleR-eq-if*:
fixes $f :: 'a \Rightarrow 'b::\text{real-vector}$
shows $\text{indicator } S \ x *_R f \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0)$
<proof>

lemma *indicator-sums*:
assumes $\bigwedge i \ j. \ i \neq j \implies A \ i \cap A \ j = \{\}$
shows $(\lambda i. \text{indicator } (A \ i) \ x::\text{real}) \text{ sums indicator } (\bigcup i. A \ i) \ x$
<proof>

The indicator function of the union of a disjoint family of sets is the sum over all the individual indicators.

lemma *indicator-UN-disjoint*:
 $\text{finite } A \implies \text{disjoint-family-on } f \ A \implies \text{indicator } (\bigcup (f \ ` \ A)) \ x = (\sum y \in A. \text{indicator } (f \ y) \ x)$
<proof>

end

40 The type of non-negative extended real numbers

theory *Extended-Nonnegative-Real*
imports *Extended-Real Indicator-Function*
begin

lemma *ereal-ineq-diff-add*:

assumes $b \neq (-\infty::ereal) \ a \geq b$
shows $a = b + (a-b)$
 $\langle proof \rangle$

lemma *Limsup-const-add*:

fixes $c :: 'a::\{complete-linorder, linorder-topology, topological-monoid-add, ordered-ab-semigroup-add\}$
shows $F \neq bot \implies Limsup\ F\ (\lambda x. c + f\ x) = c + Limsup\ F\ f$
 $\langle proof \rangle$

lemma *Liminf-const-add*:

fixes $c :: 'a::\{complete-linorder, linorder-topology, topological-monoid-add, ordered-ab-semigroup-add\}$
shows $F \neq bot \implies Liminf\ F\ (\lambda x. c + f\ x) = c + Liminf\ F\ f$
 $\langle proof \rangle$

lemma *Liminf-add-const*:

fixes $c :: 'a::\{complete-linorder, linorder-topology, topological-monoid-add, ordered-ab-semigroup-add\}$
shows $F \neq bot \implies Liminf\ F\ (\lambda x. f\ x + c) = Liminf\ F\ f + c$
 $\langle proof \rangle$

lemma *sums-offset*:

fixes $f\ g :: nat \Rightarrow 'a :: \{t2-space, topological-comm-monoid-add\}$
assumes $(\lambda n. f\ (n + i))\ sums\ l$ **shows** $f\ sums\ (l + (\sum j < i. f\ j))$
 $\langle proof \rangle$

lemma *suminf-offset*:

fixes $f\ g :: nat \Rightarrow 'a :: \{t2-space, topological-comm-monoid-add\}$
shows $summable\ (\lambda j. f\ (j + i)) \implies suminf\ f = (\sum j. f\ (j + i)) + (\sum j < i. f\ j)$
 $\langle proof \rangle$

lemma *eventually-at-left-1*: $(\bigwedge z::real. 0 < z \implies z < 1 \implies P\ z) \implies eventually\ P\ (at-left\ 1)$
 $\langle proof \rangle$

lemma *mult-eq-1*:

fixes $a\ b :: 'a :: \{ordered-semiring, comm-monoid-mult\}$
shows $0 \leq a \implies a \leq 1 \implies b \leq 1 \implies a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$
 $\langle proof \rangle$

lemma *ereal-add-diff-cancel*:

fixes $a\ b :: ereal$
shows $|b| \neq \infty \implies (a + b) - b = a$
 $\langle proof \rangle$

lemma *add-top*:

fixes $x :: 'a::\{order-top, ordered-comm-monoid-add\}$
shows $0 \leq x \implies x + top = top$

$\langle \text{proof} \rangle$

lemma *top-add*:

fixes $x :: 'a::\{\text{order-top, ordered-comm-monoid-add}\}$

shows $0 \leq x \implies \text{top} + x = \text{top}$

$\langle \text{proof} \rangle$

lemma *le-lfp*: $\text{mono } f \implies x \leq \text{lfp } f \implies f x \leq \text{lfp } f$

$\langle \text{proof} \rangle$

lemma *lfp-transfer*:

assumes α : *sup-continuous* α **and** f : *sup-continuous* f **and** mg : *mono* g

assumes bot : $\alpha \text{ bot} \leq \text{lfp } g$ **and** eq : $\bigwedge x. x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)$

shows $\alpha (\text{lfp } f) = \text{lfp } g$

$\langle \text{proof} \rangle$

lemma *sup-continuous-applyD*: *sup-continuous* $f \implies \text{sup-continuous } (\lambda x. f x h)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-SUP*[*order-continuous-intros*]:

fixes $M :: - \Rightarrow - \Rightarrow 'a::\text{complete-lattice}$

assumes M : $\bigwedge i. i \in I \implies \text{sup-continuous } (M i)$

shows *sup-continuous* $(\text{SUP } i \in I. M i)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-apply-SUP*[*order-continuous-intros*]:

fixes $M :: - \Rightarrow - \Rightarrow 'a::\text{complete-lattice}$

shows $(\bigwedge i. i \in I \implies \text{sup-continuous } (M i)) \implies \text{sup-continuous } (\lambda x. \text{SUP } i \in I. M i x)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-lfp'*[*order-continuous-intros*]:

assumes 1: *sup-continuous* f

assumes 2: $\bigwedge g. \text{sup-continuous } g \implies \text{sup-continuous } (f g)$

shows *sup-continuous* $(\text{lfp } f)$

$\langle \text{proof} \rangle$

lemma *sup-continuous-lfp''*[*order-continuous-intros*]:

assumes 1: $\bigwedge s. \text{sup-continuous } (f s)$

assumes 2: $\bigwedge g. \text{sup-continuous } g \implies \text{sup-continuous } (\lambda s. f s (g s))$

shows *sup-continuous* $(\lambda x. \text{lfp } (f x))$

$\langle \text{proof} \rangle$

lemma *mono-INF-fun*:

$(\bigwedge x y. \text{mono } (F x y)) \implies \text{mono } (\lambda z x. \text{INF } y \in X x. F x y z :: 'a :: \text{complete-lattice})$

$\langle \text{proof} \rangle$

lemma *continuous-on-cmult-ereal*:

$|c::ereal| \neq \infty \implies \text{continuous-on } A \ f \implies \text{continuous-on } A \ (\lambda x. c * f x)$
 $\langle \text{proof} \rangle$

lemma *real-of-nat-Sup*:

assumes $A \neq \{\}$ *bdd-above* A

shows $\text{of-nat } (\text{Sup } A) = (\text{SUP } a \in A. \text{ of-nat } a :: \text{real})$

$\langle \text{proof} \rangle$

lemma (*in complete-lattice*) *SUP-sup-const1*:

$I \neq \{\} \implies (\text{SUP } i \in I. \text{ sup } c \ (f \ i)) = \text{sup } c \ (\text{SUP } i \in I. f \ i)$

$\langle \text{proof} \rangle$

lemma (*in complete-lattice*) *SUP-sup-const2*:

$I \neq \{\} \implies (\text{SUP } i \in I. \text{ sup } (f \ i) \ c) = \text{sup } (\text{SUP } i \in I. f \ i) \ c$

$\langle \text{proof} \rangle$

lemma *one-less-of-natD*:

assumes $(1::'a::\text{linordered-semidom}) < \text{of-nat } n$ **shows** $1 < n$

$\langle \text{proof} \rangle$

40.1 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = $\{x :: \text{ereal}. 0 \leq x\}$

morphisms *enn2ereal* *e2ennreal'*

$\langle \text{proof} \rangle$

definition *e2ennreal* $x = e2ennreal' \ (\max \ 0 \ x)$

lemma *enn2ereal-range*: *e2ennreal* ‘ $\{0..\}$ ’ = *UNIV*

$\langle \text{proof} \rangle$

lemma *type-definition-ennreal'*: *type-definition* *enn2ereal* *e2ennreal* $\{x. 0 \leq x\}$

$\langle \text{proof} \rangle$

setup-lifting *type-definition-ennreal'*

declare $[[\text{coercion } e2ennreal]]$

instantiation *ennreal* :: *complete-linorder*

begin

lift-definition *top-ennreal* :: *ennreal* **is** *top* $\langle \text{proof} \rangle$

lift-definition *bot-ennreal* :: *ennreal* **is** *0* $\langle \text{proof} \rangle$

lift-definition *sup-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** *sup* $\langle \text{proof} \rangle$

lift-definition *inf-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** *inf* $\langle \text{proof} \rangle$

lift-definition *Inf-ennreal* :: *ennreal* *set* \Rightarrow *ennreal* **is** *Inf*

$\langle \text{proof} \rangle$

lift-definition *Sup-ennreal* :: *ennreal set* \Rightarrow *ennreal* **is** *sup 0* \circ *Sup*
 $\langle proof \rangle$

lift-definition *less-eq-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *bool* **is** (\leq) $\langle proof \rangle$

lift-definition *less-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *bool* **is** $(<)$ $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma *pcr-ennreal-enn2ereal[simp]*: *pcr-ennreal* (*enn2ereal* *x*) *x*
 $\langle proof \rangle$

lemma *rel-fun-eq-pcr-ennreal*: *rel-fun* $(=)$ *pcr-ennreal* *f g* \longleftrightarrow *f* = *enn2ereal* \circ *g*
 $\langle proof \rangle$

instantiation *ennreal* :: *infinity*
begin

definition *infinity-ennreal* :: *ennreal*
where [*simp*]: $\infty = (top::ennreal)$

instance $\langle proof \rangle$

end

instantiation *ennreal* :: {*semiring-1-no-zero-divisors*, *comm-semiring-1*}
begin

lift-definition *one-ennreal* :: *ennreal* **is** 1 $\langle proof \rangle$

lift-definition *zero-ennreal* :: *ennreal* **is** 0 $\langle proof \rangle$

lift-definition *plus-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** $(+)$ $\langle proof \rangle$

lift-definition *times-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** $(*)$ $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

instantiation *ennreal* :: *minus*
begin

lift-definition *minus-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** $\lambda a b. \max 0 (a - b)$
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

instance *ennreal* :: *numeral* $\langle \text{proof} \rangle$

instantiation *ennreal* :: *inverse*
begin

lift-definition *inverse-ennreal* :: *ennreal* \Rightarrow *ennreal* **is** *inverse*
 $\langle \text{proof} \rangle$

definition *divide-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal*
where $x \text{ div } y = x * \text{inverse } (y :: \text{ennreal})$

instance $\langle \text{proof} \rangle$

end

lemma *ennreal-zero-less-one*: $0 < (1 :: \text{ennreal})$ — TODO: remove
 $\langle \text{proof} \rangle$

instance *ennreal* :: *dioid*
 $\langle \text{proof} \rangle$

instance *ennreal* :: *ordered-comm-semiring*
 $\langle \text{proof} \rangle$

instance *ennreal* :: *linordered-nonzero-semiring*
 $\langle \text{proof} \rangle$

instance *ennreal* :: *strict-ordered-ab-semigroup-add*
 $\langle \text{proof} \rangle$

declare $[[\text{coercion of-nat} :: \text{nat} \Rightarrow \text{ennreal}]]$

lemma *e2ennreal-neg*: $x \leq 0 \Longrightarrow e2ennreal\ x = 0$
 $\langle \text{proof} \rangle$

lemma *e2ennreal-mono*: $x \leq y \Longrightarrow e2ennreal\ x \leq e2ennreal\ y$
 $\langle \text{proof} \rangle$

lemma *enn2ereal-nonneg[simp]*: $0 \leq \text{enn2ereal } x$
 $\langle \text{proof} \rangle$

lemma *ereal-ennreal-cases*:
obtains b **where** $0 \leq a$ $a = \text{enn2ereal } b$ $|$ $a < 0$
 $\langle \text{proof} \rangle$

lemma *rel-fun-liminf[transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal*

liminf liminf
 $\langle \text{proof} \rangle$

lemma *rel-fun-limsup[transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal*
limsup limsup
 $\langle \text{proof} \rangle$

lemma *sum-enn2ereal[simp]*: $(\bigwedge i. i \in I \implies 0 \leq f i) \implies (\sum_{i \in I}. \text{enn2ereal } (f i))$
 $= \text{enn2ereal } (\text{sum } f I)$
 $\langle \text{proof} \rangle$

lemma *transfer-e2ennreal-sum [transfer-rule]*:
rel-fun (*rel-fun* (=) *pcr-ennreal*) (*rel-fun* (=) *pcr-ennreal*) *sum sum*
 $\langle \text{proof} \rangle$

lemma *enn2ereal-of-nat[simp]*: *enn2ereal* (*of-nat* *n*) = *ereal* *n*
 $\langle \text{proof} \rangle$

lemma *enn2ereal-numeral[simp]*: *enn2ereal* (*numeral* *a*) = *numeral* *a*
 $\langle \text{proof} \rangle$

lemma *transfer-numeral[transfer-rule]*: *pcr-ennreal* (*numeral* *a*) (*numeral* *a*)
 $\langle \text{proof} \rangle$

40.2 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty :: \text{ennreal}) \vee b = c$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty :: \text{ennreal}) \vee b \leq c$
 $\langle \text{proof} \rangle$

lemma *ereal-add-left-cancel-less*:
fixes *a b c* :: *ereal*
shows $0 \leq a \implies 0 \leq b \implies a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty :: \text{ennreal}) \wedge b < c$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

40.3 Order with top

lemma *ennreal-zero-less-top[simp]*: $0 < (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-one-less-top[simp]*: $1 < (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-zero-neq-top[simp]*: $0 \neq (top :: ennreal)$
 $\langle proof \rangle$

lemma *ennreal-top-neq-zero[simp]*: $(top :: ennreal) \neq 0$
 $\langle proof \rangle$

lemma *ennreal-top-neq-one[simp]*: $top \neq (1 :: ennreal)$
 $\langle proof \rangle$

lemma *ennreal-one-neq-top[simp]*: $1 \neq (top :: ennreal)$
 $\langle proof \rangle$

lemma *ennreal-add-less-top[simp]*:
fixes $a\ b :: ennreal$
shows $a + b < top \longleftrightarrow a < top \wedge b < top$
 $\langle proof \rangle$

lemma *ennreal-add-eq-top[simp]*:
fixes $a\ b :: ennreal$
shows $a + b = top \longleftrightarrow a = top \vee b = top$
 $\langle proof \rangle$

lemma *ennreal-sum-less-top[simp]*:
fixes $f :: 'a \Rightarrow ennreal$
shows $finite\ I \implies (\sum i \in I. f\ i) < top \longleftrightarrow (\forall i \in I. f\ i < top)$
 $\langle proof \rangle$

lemma *ennreal-sum-eq-top[simp]*:
fixes $f :: 'a \Rightarrow ennreal$
shows $finite\ I \implies (\sum i \in I. f\ i) = top \longleftrightarrow (\exists i \in I. f\ i = top)$
 $\langle proof \rangle$

lemma *ennreal-mult-eq-top-iff*:
fixes $a\ b :: ennreal$
shows $a * b = top \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
 $\langle proof \rangle$

lemma *ennreal-top-eq-mult-iff*:
fixes $a\ b :: ennreal$
shows $top = a * b \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
 $\langle proof \rangle$

lemma *ennreal-mult-less-top*:
fixes $a\ b :: ennreal$
shows $a * b < top \longleftrightarrow (a = 0 \vee b = 0 \vee (a < top \wedge b < top))$
 $\langle proof \rangle$

lemma *top-power-ennreal*: $top \wedge^n = (if\ n = 0\ then\ 1\ else\ top :: ennreal)$
 $\langle proof \rangle$

lemma *ennreal-prod-eq-0*[simp]:

fixes $f :: 'a \Rightarrow \text{ennreal}$

shows $(\text{prod } f \ A = 0) = (\text{finite } A \wedge (\exists i \in A. f \ i = 0))$

$\langle \text{proof} \rangle$

lemma *ennreal-prod-eq-top*:

fixes $f :: 'a \Rightarrow \text{ennreal}$

shows $(\prod i \in I. f \ i) = \text{top} \longleftrightarrow (\text{finite } I \wedge ((\forall i \in I. f \ i \neq 0) \wedge (\exists i \in I. f \ i = \text{top})))$

$\langle \text{proof} \rangle$

lemma *ennreal-top-mult*: $\text{top} * a = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-mult-top*: $a * \text{top} = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *enn2ereal-eq-top-iff*[simp]: $\text{enn2ereal } x = \infty \longleftrightarrow x = \text{top}$

$\langle \text{proof} \rangle$

lemma *enn2ereal-top*[simp]: $\text{enn2ereal } \text{top} = \infty$

$\langle \text{proof} \rangle$

lemma *e2ennreal-infty*[simp]: $e2ennreal \ \infty = \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-top-minus*[simp]: $\text{top} - x = (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *minus-top-ennreal*: $x - \text{top} = (\text{if } x = \text{top} \text{ then } \text{top} \text{ else } 0 :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *bot-ennreal*: $\text{bot} = (0 :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-of-nat-neq-top*[simp]: $\text{of-nat } i \neq (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *numeral-eq-of-nat*: $(\text{numeral } a :: \text{ennreal}) = \text{of-nat } (\text{numeral } a)$

$\langle \text{proof} \rangle$

lemma *of-nat-less-top*: $\text{of-nat } i < (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *top-neq-numeral*[simp]: $\text{top} \neq (\text{numeral } i :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-numeral-less-top*[simp]: $\text{numeral } i < (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-add-bot[simp]*: $\text{bot} + x = (x :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *add-top-right-ennreal [simp]*: $x + \text{top} = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *add-top-left-ennreal [simp]*: $\text{top} + x = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-top-mult-left [simp]*: $x \neq 0 \implies x * \text{top} = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-top-mult-right [simp]*: $x \neq 0 \implies \text{top} * x = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *power-top-ennreal [simp]*: $n > 0 \implies \text{top} ^ n = (\text{top} :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *power-eq-top-ennreal-iff*: $x ^ n = \text{top} \longleftrightarrow x = (\text{top} :: \text{ennreal}) \wedge n > 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-le-mult-iff*: $c \neq 0 \implies c \neq \text{top} \implies c * a \leq c * b \longleftrightarrow a \leq (b :: \text{ennreal})$
including *ennreal.lifting*
 $\langle \text{proof} \rangle$

lemma *power-mono-ennreal*: $x \leq y \implies x ^ n \leq (y ^ n :: \text{ennreal})$
 $\langle \text{proof} \rangle$

instance *ennreal :: semiring-char-0*
 $\langle \text{proof} \rangle$

40.4 Arithmetic

lemma *ennreal-minus-zero[simp]*: $a - (0 :: \text{ennreal}) = a$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-diff-cancel-right[simp]*:
fixes $x \ y \ z :: \text{ennreal}$ **shows** $y \neq \text{top} \implies (x + y) - y = x$
 $\langle \text{proof} \rangle$

lemma *ennreal-add-diff-cancel-left[simp]*:
fixes $x \ y \ z :: \text{ennreal}$ **shows** $y \neq \text{top} \implies (y + x) - y = x$
 $\langle \text{proof} \rangle$

lemma
fixes $a \ b :: \text{ennreal}$

shows $a - b = 0 \implies a \leq b$
 $\langle proof \rangle$

lemma *ennreal-minus-cancel*:

fixes $a\ b\ c :: ennreal$

shows $c \neq top \implies a \leq c \implies b \leq c \implies c - a = c - b \implies a = b$
 $\langle proof \rangle$

lemma *sup-const-add-ennreal*:

fixes $a\ b\ c :: ennreal$

shows $sup\ (c + a)\ (c + b) = c + sup\ a\ b$
 $\langle proof \rangle$

lemma *ennreal-diff-add-assoc*:

fixes $a\ b\ c :: ennreal$

shows $a \leq b \implies c + b - a = c + (b - a)$
 $\langle proof \rangle$

lemma *mult-divide-eq-ennreal*:

fixes $a\ b :: ennreal$

shows $b \neq 0 \implies b \neq top \implies (a * b) / b = a$
 $\langle proof \rangle$

lemma *divide-mult-eq*: $a \neq 0 \implies a \neq \infty \implies x * a / (b * a) = x / (b :: ennreal)$

$\langle proof \rangle$

lemma *ennreal-mult-divide-eq*:

fixes $a\ b :: ennreal$

shows $b \neq 0 \implies b \neq top \implies (a * b) / b = a$
 $\langle proof \rangle$

lemma *ennreal-add-diff-cancel*:

fixes $a\ b :: ennreal$

shows $b \neq \infty \implies (a + b) - b = a$
 $\langle proof \rangle$

lemma *ennreal-minus-eq-0*:

$a - b = 0 \implies a \leq (b :: ennreal)$

$\langle proof \rangle$

lemma *ennreal-mono-minus-cancel*:

fixes $a\ b\ c :: ennreal$

shows $a - b \leq a - c \implies a < top \implies b \leq a \implies c \leq a \implies c \leq b$
 $\langle proof \rangle$

lemma *ennreal-mono-minus*:

fixes $a\ b\ c :: ennreal$

shows $c \leq b \implies a - b \leq a - c$
 $\langle proof \rangle$

lemma *ennreal-minus-pos-iff*:

fixes $a\ b :: \text{ennreal}$

shows $a < \text{top} \vee b < \text{top} \implies 0 < a - b \implies b < a$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-top[simp]*: $\text{inverse } \text{top} = (0 :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-zero[simp]*: $\text{inverse } 0 = (\text{top} :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-top-divide*: $\text{top} / (x :: \text{ennreal}) = (\text{if } x = \text{top} \text{ then } 0 \text{ else } \text{top})$

$\langle \text{proof} \rangle$

lemma *ennreal-zero-divide[simp]*: $0 / (x :: \text{ennreal}) = 0$

$\langle \text{proof} \rangle$

lemma *ennreal-divide-zero[simp]*: $x / (0 :: \text{ennreal}) = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{top})$

$\langle \text{proof} \rangle$

lemma *ennreal-divide-top[simp]*: $x / (\text{top} :: \text{ennreal}) = 0$

$\langle \text{proof} \rangle$

lemma *ennreal-times-divide*: $a * (b / c) = a * b / (c :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-zero-less-divide*: $0 < a / b \longleftrightarrow (0 < a \wedge b < (\text{top} :: \text{ennreal}))$

$\langle \text{proof} \rangle$

lemma *add-divide-distrib-ennreal*: $(a + b) / c = a / c + b / (c :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *divide-right-mono-ennreal*:

fixes $a\ b\ c :: \text{ennreal}$

shows $a \leq b \implies a / c \leq b / c$

$\langle \text{proof} \rangle$

lemma *ennreal-mult-strict-right-mono*: $(a :: \text{ennreal}) < c \implies 0 < b \implies b < \text{top} \implies a * b < c * b$

$\langle \text{proof} \rangle$

lemma *ennreal-indicator-less[simp]*:

$\text{indicator } A\ x \leq (\text{indicator } B\ x :: \text{ennreal}) \longleftrightarrow (x \in A \longrightarrow x \in B)$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-positive*: $0 < \text{inverse } x \longleftrightarrow (x :: \text{ennreal}) \neq \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-mult'*: $((0 < b \vee a < \text{top}) \wedge (0 < a \vee b < \text{top})) \implies$
 $\text{inverse } (a * b::\text{ennreal}) = \text{inverse } a * \text{inverse } b$
 $\langle \text{proof} \rangle$

lemma *ennreal-inverse-mult*: $a < \text{top} \implies b < \text{top} \implies \text{inverse } (a * b::\text{ennreal}) =$
 $\text{inverse } a * \text{inverse } b$
 $\langle \text{proof} \rangle$

lemma *ennreal-inverse-1[simp]*: $\text{inverse } (1::\text{ennreal}) = 1$
 $\langle \text{proof} \rangle$

lemma *ennreal-inverse-eq-0-iff[simp]*: $\text{inverse } (a::\text{ennreal}) = 0 \longleftrightarrow a = \text{top}$
 $\langle \text{proof} \rangle$

lemma *ennreal-inverse-eq-top-iff[simp]*: $\text{inverse } (a::\text{ennreal}) = \text{top} \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-divide-eq-0-iff[simp]*: $(a::\text{ennreal}) / b = 0 \longleftrightarrow (a = 0 \vee b = \text{top})$
 $\langle \text{proof} \rangle$

lemma *ennreal-divide-eq-top-iff*: $(a::\text{ennreal}) / b = \text{top} \longleftrightarrow ((a \neq 0 \wedge b = 0) \vee$
 $(a = \text{top} \wedge b \neq \text{top}))$
 $\langle \text{proof} \rangle$

lemma *one-divide-one-divide-ennreal[simp]*: $1 / (1 / c) = (c::\text{ennreal})$
including *ennreal.lifting*
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-left-cong*:
 $((a::\text{ennreal}) \neq 0 \implies b = c) \implies a * b = a * c$
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-right-cong*:
 $((a::\text{ennreal}) \neq 0 \implies b = c) \implies b * a = c * a$
 $\langle \text{proof} \rangle$

lemma *ennreal-zero-less-mult-iff*: $0 < a * b \longleftrightarrow 0 < a \wedge 0 < (b::\text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *less-diff-eq-ennreal*:
fixes $a \ b \ c :: \text{ennreal}$
shows $b < \text{top} \vee c < \text{top} \implies a < b - c \longleftrightarrow a + c < b$
 $\langle \text{proof} \rangle$

lemma *diff-add-cancel-ennreal*:
fixes $a \ b :: \text{ennreal}$ **shows** $a \leq b \implies b - a + a = b$
 $\langle \text{proof} \rangle$

lemma *ennreal-diff-self[simp]*: $a \neq \text{top} \implies a - a = (0::\text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-minus-mono*:

fixes $a\ b\ c :: \text{ennreal}$

shows $a \leq c \implies d \leq b \implies a - b \leq c - d$

$\langle \text{proof} \rangle$

lemma *ennreal-minus-eq-top[simp]*: $a - (b :: \text{ennreal}) = \text{top} \longleftrightarrow a = \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-divide-self[simp]*: $a \neq 0 \implies a < \text{top} \implies a / a = (1 :: \text{ennreal})$

$\langle \text{proof} \rangle$

40.5 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: *real* \Rightarrow *ennreal* **is** $\text{sup } 0 \circ \text{ereal}$

$\langle \text{proof} \rangle$

declare $[[\text{coercion } \text{ennreal}]]$

lemma *ennreal-cong*: $x = y \implies \text{ennreal } x = \text{ennreal } y$

$\langle \text{proof} \rangle$

lemma *ennreal-cases*[*cases type: ennreal*]:

fixes $x :: \text{ennreal}$

obtains (*real*) $r :: \text{real}$ **where** $0 \leq r$ $x = \text{ennreal } r \mid (\text{top})\ x = \text{top}$

$\langle \text{proof} \rangle$

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]

lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neq-top[simp]*: $\text{ennreal } r \neq \text{top}$

$\langle \text{proof} \rangle$

lemma *top-neq-ennreal[simp]*: $\text{top} \neq \text{ennreal } r$

$\langle \text{proof} \rangle$

lemma *ennreal-less-top[simp]*: $\text{ennreal } x < \text{top}$

$\langle \text{proof} \rangle$

lemma *ennreal-neg*: $x \leq 0 \implies \text{ennreal } x = 0$

$\langle \text{proof} \rangle$

lemma *ennreal-inj[simp]*:

$0 \leq a \implies 0 \leq b \implies \text{ennreal } a = \text{ennreal } b \longleftrightarrow a = b$

$\langle \text{proof} \rangle$

lemma *ennreal-le-iff[simp]*: $0 \leq y \implies \text{ennreal } x \leq \text{ennreal } y \longleftrightarrow x \leq y$

$\langle \text{proof} \rangle$

lemma *le-ennreal-iff*: $0 \leq r \implies x \leq \text{ennreal } r \longleftrightarrow (\exists q \geq 0. x = \text{ennreal } q \wedge q \leq r)$
 $\langle \text{proof} \rangle$

lemma *ennreal-less-iff*: $0 \leq r \implies \text{ennreal } r < \text{ennreal } q \longleftrightarrow r < q$
 $\langle \text{proof} \rangle$

lemma *ennreal-eq-zero-iff[simp]*: $0 \leq x \implies \text{ennreal } x = 0 \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-less-zero-iff[simp]*: $0 < \text{ennreal } x \longleftrightarrow 0 < x$
 $\langle \text{proof} \rangle$

lemma *ennreal-lessI*: $0 < q \implies r < q \implies \text{ennreal } r < \text{ennreal } q$
 $\langle \text{proof} \rangle$

lemma *ennreal-leI*: $x \leq y \implies \text{ennreal } x \leq \text{ennreal } y$
 $\langle \text{proof} \rangle$

lemma *enn2ereal-ennreal[simp]*: $0 \leq x \implies \text{enn2ereal } (\text{ennreal } x) = x$
 $\langle \text{proof} \rangle$

lemma *e2ennreal-enn2ereal[simp]*: $e2ennreal (\text{enn2ereal } x) = x$
 $\langle \text{proof} \rangle$

lemma *enn2ereal-e2ennreal*: $x \geq 0 \implies \text{enn2ereal } (e2ennreal x) = x$
 $\langle \text{proof} \rangle$

lemma *e2ennreal-ereal [simp]*: $e2ennreal (\text{ereal } x) = \text{ennreal } x$
 $\langle \text{proof} \rangle$

lemma *ennreal-0[simp]*: $\text{ennreal } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-1[simp]*: $\text{ennreal } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *ennreal-eq-0-iff*: $\text{ennreal } x = 0 \longleftrightarrow x \leq 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-le-iff2*: $\text{ennreal } x \leq \text{ennreal } y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge y \leq 0))$
 $\langle \text{proof} \rangle$

lemma *ennreal-eq-1[simp]*: $\text{ennreal } x = 1 \longleftrightarrow x = 1$
 $\langle \text{proof} \rangle$

lemma *ennreal-le-1[simp]*: $\text{ennreal } x \leq 1 \longleftrightarrow x \leq 1$

$\langle \text{proof} \rangle$

lemma *ennreal-ge-1[simp]*: $\text{ennreal } x \geq 1 \longleftrightarrow x \geq 1$
 $\langle \text{proof} \rangle$

lemma *one-less-ennreal[simp]*: $1 < \text{ennreal } x \longleftrightarrow 1 < x$
 $\langle \text{proof} \rangle$

lemma *ennreal-plus[simp]*:
 $0 \leq a \implies 0 \leq b \implies \text{ennreal } (a + b) = \text{ennreal } a + \text{ennreal } b$
 $\langle \text{proof} \rangle$

lemma *add-mono-ennreal*: $x < \text{ennreal } y \implies x' < \text{ennreal } y' \implies x + x' < \text{ennreal } (y + y')$
 $\langle \text{proof} \rangle$

lemma *sum-ennreal[simp]*: $(\bigwedge i. i \in I \implies 0 \leq f i) \implies (\sum_{i \in I}. \text{ennreal } (f i)) = \text{ennreal } (\sum f I)$
 $\langle \text{proof} \rangle$

lemma *sum-list-ennreal[simp]*:
assumes $\bigwedge x. x \in \text{set } xs \implies f x \geq 0$
shows $\text{sum-list } (\text{map } (\lambda x. \text{ennreal } (f x)) xs) = \text{ennreal } (\text{sum-list } (\text{map } f xs))$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-nat-eq-real-of-nat*: $\text{of-nat } i = \text{ennreal } (\text{of-nat } i)$
 $\langle \text{proof} \rangle$

lemma *of-nat-le-ennreal-iff[simp]*: $0 \leq r \implies \text{of-nat } i \leq \text{ennreal } r \longleftrightarrow \text{of-nat } i \leq r$
 $\langle \text{proof} \rangle$

lemma *ennreal-le-of-nat-iff[simp]*: $\text{ennreal } r \leq \text{of-nat } i \longleftrightarrow r \leq \text{of-nat } i$
 $\langle \text{proof} \rangle$

lemma *ennreal-indicator*: $\text{ennreal } (\text{indicator } A x) = \text{indicator } A x$
 $\langle \text{proof} \rangle$

lemma *ennreal-numeral[simp]*: $\text{ennreal } (\text{numeral } n) = \text{numeral } n$
 $\langle \text{proof} \rangle$

lemma *ennreal-less-numeral-iff [simp]*: $\text{ennreal } n < \text{numeral } w \longleftrightarrow n < \text{numeral } w$
 $\langle \text{proof} \rangle$

lemma *numeral-less-ennreal-iff [simp]*: $\text{numeral } w < \text{ennreal } n \longleftrightarrow \text{numeral } w < n$
 $\langle \text{proof} \rangle$

lemma *numeral-le-ennreal-iff* [simp]: $\text{numeral } n \leq \text{ennreal } m \longleftrightarrow \text{numeral } n \leq m$
 ⟨proof⟩

lemma *min-ennreal*: $0 \leq x \implies 0 \leq y \implies \min (\text{ennreal } x) (\text{ennreal } y) = \text{ennreal } (\min x y)$
 ⟨proof⟩

lemma *ennreal-half*[simp]: $\text{ennreal } (1/2) = \text{inverse } 2$
 ⟨proof⟩

lemma *ennreal-minus*: $0 \leq q \implies \text{ennreal } r - \text{ennreal } q = \text{ennreal } (r - q)$
 ⟨proof⟩

lemma *ennreal-minus-top*[simp]: $\text{ennreal } a - \text{top} = 0$
 ⟨proof⟩

lemma *e2eenreal-enn2ereal-diff* [simp]:
 $e2\text{ennreal}(\text{enn2ereal } x - \text{enn2ereal } y) = x - y$ **for** $x y$
 ⟨proof⟩

lemma *ennreal-mult*: $0 \leq a \implies 0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
 ⟨proof⟩

lemma *ennreal-mult'*: $0 \leq a \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
 ⟨proof⟩

lemma *indicator-mult-ennreal*: $\text{indicator } A x * \text{ennreal } r = \text{ennreal } (\text{indicator } A x * r)$
 ⟨proof⟩

lemma *ennreal-mult''*: $0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
 ⟨proof⟩

lemma *numeral-mult-ennreal*: $0 \leq x \implies \text{numeral } b * \text{ennreal } x = \text{ennreal } (\text{numeral } b * x)$
 ⟨proof⟩

lemma *ennreal-power*: $0 \leq r \implies \text{ennreal } r ^ n = \text{ennreal } (r ^ n)$
 ⟨proof⟩

lemma *power-eq-top-ennreal*: $x ^ n = \text{top} \longleftrightarrow (n \neq 0 \wedge (x::\text{ennreal}) = \text{top})$
 ⟨proof⟩

lemma *inverse-ennreal*: $0 < r \implies \text{inverse } (\text{ennreal } r) = \text{ennreal } (\text{inverse } r)$
 ⟨proof⟩

lemma *divide-ennreal*: $0 \leq r \implies 0 < q \implies \text{ennreal } r / \text{ennreal } q = \text{ennreal } (r / q)$

$\langle \text{proof} \rangle$

lemma *ennreal-inverse-power*: $\text{inverse } (x \wedge n :: \text{ennreal}) = \text{inverse } x \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-divide-distrib-ennreal* [*algebra-simps*]:
 $(x / y) \wedge n = x \wedge n / (y \wedge n :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *ennreal-divide-numeral*: $0 \leq x \implies \text{ennreal } x / \text{numeral } b = \text{ennreal } (x / \text{numeral } b)$
 $\langle \text{proof} \rangle$

lemma *prod-ennreal*: $(\bigwedge i. i \in A \implies 0 \leq f i) \implies (\prod_{i \in A. \text{ennreal } (f i)} = \text{ennreal } (\prod f A))$
 $\langle \text{proof} \rangle$

lemma *prod-mono-ennreal*:
assumes $\bigwedge x. x \in A \implies f x \leq (g x :: \text{ennreal})$
shows $\text{prod } f A \leq \text{prod } g A$
 $\langle \text{proof} \rangle$

lemma *mult-right-ennreal-cancel*: $a * \text{ennreal } c = b * \text{ennreal } c \longleftrightarrow (a = b \vee c \leq 0)$
 $\langle \text{proof} \rangle$

lemma *ennreal-le-epsilon*:
 $(\bigwedge e :: \text{real}. y < \text{top} \implies 0 < e \implies x \leq y + \text{ennreal } e) \implies x \leq y$
 $\langle \text{proof} \rangle$

lemma *ennreal-rat-dense*:
fixes $x y :: \text{ennreal}$
shows $x < y \implies \exists r :: \text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$
 $\langle \text{proof} \rangle$

lemma *ennreal-Ex-less-of-nat*: $(x :: \text{ennreal}) < \text{top} \implies \exists n. x < \text{of-nat } n$
 $\langle \text{proof} \rangle$

40.6 Coercion from *ennreal* to *real*

definition *enn2real* $x = \text{real-of-ereal } (\text{enn2ereal } x)$

lemma *enn2real-nonneg*[*simp*]: $0 \leq \text{enn2real } x$
 $\langle \text{proof} \rangle$

lemma *enn2real-mono*: $a \leq b \implies b < \text{top} \implies \text{enn2real } a \leq \text{enn2real } b$
 $\langle \text{proof} \rangle$

lemma *enn2real-of-nat*[*simp*]: $\text{enn2real } (\text{of-nat } n) = n$

$\langle \text{proof} \rangle$

lemma *enn2real-ennreal[simp]*: $0 \leq r \implies \text{enn2real} (\text{ennreal } r) = r$
 $\langle \text{proof} \rangle$

lemma *ennreal-enn2real[simp]*: $r < \text{top} \implies \text{ennreal} (\text{enn2real } r) = r$
 $\langle \text{proof} \rangle$

lemma *real-of-ereal-enn2ereal[simp]*: $\text{real-of-ereal} (\text{enn2ereal } x) = \text{enn2real } x$
 $\langle \text{proof} \rangle$

lemma *enn2real-top[simp]*: $\text{enn2real } \text{top} = 0$
 $\langle \text{proof} \rangle$

lemma *enn2real-0[simp]*: $\text{enn2real } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *enn2real-1[simp]*: $\text{enn2real } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *enn2real-numeral[simp]*: $\text{enn2real} (\text{numeral } n) = (\text{numeral } n)$
 $\langle \text{proof} \rangle$

lemma *enn2real-mult*: $\text{enn2real} (a * b) = \text{enn2real } a * \text{enn2real } b$
 $\langle \text{proof} \rangle$

lemma *enn2real-leI*: $0 \leq B \implies x \leq \text{ennreal } B \implies \text{enn2real } x \leq B$
 $\langle \text{proof} \rangle$

lemma *enn2real-positive-iff*: $0 < \text{enn2real } x \longleftrightarrow (0 < x \wedge x < \text{top})$
 $\langle \text{proof} \rangle$

lemma *enn2real-eq-posreal-iff[simp]*: $c > 0 \implies \text{enn2real } x = c \longleftrightarrow x = c$
 $\langle \text{proof} \rangle$

lemma *ennreal-enn2real-if*: $\text{ennreal} (\text{enn2real } r) = (\text{if } r = \text{top} \text{ then } 0 \text{ else } r)$
 $\langle \text{proof} \rangle$

40.7 Coercion from *enat* to *ennreal*

definition *ennreal-of-enat* :: *enat* \Rightarrow *ennreal*

where

ennreal-of-enat *n* = (case *n* of $\infty \Rightarrow \text{top}$ | *enat* *n* \Rightarrow *of-nat* *n*)

declare [[*coercion* *ennreal-of-enat*]]

declare [[*coercion* *of-nat* :: *nat* \Rightarrow *ennreal*]]

lemma *ennreal-of-enat-infty[simp]*: *ennreal-of-enat* $\infty = \infty$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-enat[simp]*: $\text{ennreal-of-enat } (\text{enat } n) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-0[simp]*: $\text{ennreal-of-enat } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-1[simp]*: $\text{ennreal-of-enat } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *ennreal-top-neg-of-nat[simp]*: $(\text{top}::\text{ennreal}) \neq \text{of-nat } i$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-inj[simp]*: $\text{ennreal-of-enat } i = \text{ennreal-of-enat } j \longleftrightarrow i = j$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-le-iff[simp]*: $\text{ennreal-of-enat } m \leq \text{ennreal-of-enat } n \longleftrightarrow m \leq n$
 $\langle \text{proof} \rangle$

lemma *of-nat-less-ennreal-of-nat[simp]*: $\text{of-nat } n \leq \text{ennreal-of-enat } x \longleftrightarrow \text{of-nat } n \leq x$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-Sup*: $\text{ennreal-of-enat } (\text{Sup } X) = (\text{SUP } x \in X. \text{ennreal-of-enat } x)$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-eSuc[simp]*: $\text{ennreal-of-enat } (\text{eSuc } x) = 1 + \text{ennreal-of-enat } x$
 $\langle \text{proof} \rangle$

lemma *ennreal-of-enat-plus[simp]*: $\text{ennreal-of-enat } (a+b) = \text{ennreal-of-enat } a + \text{ennreal-of-enat } b$
 $\langle \text{proof} \rangle$

lemma *sum-ennreal-of-enat[simp]*: $(\sum i \in I. \text{ennreal-of-enat } (f i)) = \text{ennreal-of-enat } (\text{sum } f I)$
 $\langle \text{proof} \rangle$

40.8 Topology on *ennreal*

lemma *enn2ereal-Iio*: $\text{enn2ereal} - ' \{..<a\} = (\text{if } 0 \leq a \text{ then } \{..< \text{e2ennreal } a\} \text{ else } \{\})$
 $\langle \text{proof} \rangle$

lemma *enn2ereal-Ioi*: $\text{enn2ereal} - ' \{a <..\} = (\text{if } 0 \leq a \text{ then } \{\text{e2ennreal } a <..\})$

else UNIV)
⟨proof⟩

instantiation *ennreal* :: *linear-continuum-topology*
begin

definition *open-ennreal* :: *ennreal set* \Rightarrow *bool*
where (*open* :: *ennreal set* \Rightarrow *bool*) = *generate-topology* (*range lessThan* \cup *range greaterThan*)

instance
⟨proof⟩

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*
⟨proof⟩

lemma *continuous-at-e2ennreal*: *continuous (at x within A) e2ennreal*
⟨proof⟩

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*
⟨proof⟩

lemma *continuous-at-enn2ereal*: *continuous (at x within A) enn2ereal*
⟨proof⟩

lemma *sup-continuous-e2ennreal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* ($\lambda x. e2ennreal (f x)$)
⟨proof⟩

lemma *sup-continuous-enn2ereal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* ($\lambda x. enn2ereal (f x)$)
⟨proof⟩

lemma *sup-continuous-mult-left-ennreal'*:
fixes *c* :: *ennreal*
shows *sup-continuous* ($\lambda x. c * x$)
⟨proof⟩

lemma *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:
sup-continuous f \implies *sup-continuous* ($\lambda x. c * f x :: ennreal$)
⟨proof⟩

lemma *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:
sup-continuous f \implies *sup-continuous* ($\lambda x. f x * c :: ennreal$)
⟨proof⟩

lemma *sup-continuous-divide-ennreal*[*order-continuous-intros*]:

fixes $f\ g :: 'a::\text{complete-lattice} \Rightarrow \text{ennreal}$
shows $\text{sup-continuous } f \Longrightarrow \text{sup-continuous } (\lambda x. f\ x \ / \ c)$
 $\langle \text{proof} \rangle$

lemma $\text{transfer-enn2ereal-continuous-on}$ [*transfer-rule*]:
 $\text{rel-fun } (=) \ (\text{rel-fun } (\text{rel-fun } (=) \ \text{pcr-ennreal}) \ (=)) \ \text{continuous-on continuous-on}$
 $\langle \text{proof} \rangle$

lemma $\text{transfer-sup-continuous}$ [*transfer-rule*]:
 $(\text{rel-fun } (\text{rel-fun } (=) \ \text{pcr-ennreal}) \ (=)) \ \text{sup-continuous sup-continuous}$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-on-ennreal}$ [*tendsto-intros*]:
 $\text{continuous-on } A \ f \Longrightarrow \text{continuous-on } A \ (\lambda x. \text{ennreal } (f\ x))$
 $\langle \text{proof} \rangle$

lemma tendsto-ennrealD :
assumes $\text{lim}: ((\lambda x. \text{ennreal } (f\ x)) \longrightarrow \text{ennreal } x) \ F$
assumes $*$: $\forall_F\ x \text{ in } F. \ 0 \leq f\ x \text{ and } x: \ 0 \leq x$
shows $(f \longrightarrow x) \ F$
 $\langle \text{proof} \rangle$

lemma $\text{tendsto-ennreal-iff}$ [*simp*]:
 $\langle ((\lambda x. \text{ennreal } (f\ x)) \longrightarrow \text{ennreal } x) \ F \longleftrightarrow (f \longrightarrow x) \ F \rangle \ (\text{is } \langle ?P \longleftrightarrow ?Q \rangle)$
if $\langle \forall_F\ x \text{ in } F. \ 0 \leq f\ x \rangle \langle 0 \leq x \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{tendsto-enn2ereal-iff}$ [*simp*]: $((\lambda i. \text{enn2ereal } (f\ i)) \longrightarrow \text{enn2ereal } x) \ F \longleftrightarrow$
 $(f \longrightarrow x) \ F$
 $\langle \text{proof} \rangle$

lemma $\text{ennreal-tendsto-0-iff}$: $(\bigwedge n. f\ n \geq 0) \Longrightarrow ((\lambda n. \text{ennreal } (f\ n)) \longrightarrow 0)$
 $\longleftrightarrow (f \longrightarrow 0)$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-on-add-ennreal}$:
fixes $f\ g :: 'a::\text{topological-space} \Rightarrow \text{ennreal}$
shows $\text{continuous-on } A \ f \Longrightarrow \text{continuous-on } A \ g \Longrightarrow \text{continuous-on } A \ (\lambda x. f\ x$
 $+ \ g\ x)$
 $\langle \text{proof} \rangle$

lemma $\text{continuous-on-inverse-ennreal}$ [*continuous-intros*]:
fixes $f :: 'a::\text{topological-space} \Rightarrow \text{ennreal}$
shows $\text{continuous-on } A \ f \Longrightarrow \text{continuous-on } A \ (\lambda x. \text{inverse } (f\ x))$
 $\langle \text{proof} \rangle$

instance $\text{ennreal} :: \text{topological-comm-monoid-add}$
 $\langle \text{proof} \rangle$

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:

fixes $f\ g :: 'a::\text{complete-lattice} \Rightarrow \text{ennreal}$

shows $\text{sup-continuous } f \Longrightarrow \text{sup-continuous } g \Longrightarrow \text{sup-continuous } (\lambda x. f\ x + g\ x)$
 $\langle \text{proof} \rangle$

lemma *ennreal-suminf-lessD*: $(\sum i. f\ i :: \text{ennreal}) < x \Longrightarrow f\ i < x$

$\langle \text{proof} \rangle$

lemma *sums-ennreal*[*simp*]: $(\bigwedge i. 0 \leq f\ i) \Longrightarrow 0 \leq x \Longrightarrow (\lambda i. \text{ennreal } (f\ i)) \text{ sums } \text{ennreal } x \longleftrightarrow f \text{ sums } x$

$\langle \text{proof} \rangle$

lemma *summable-suminf-not-top*: $(\bigwedge i. 0 \leq f\ i) \Longrightarrow (\sum i. \text{ennreal } (f\ i)) \neq \text{top} \Longrightarrow \text{summable } f$

$\langle \text{proof} \rangle$

lemma *suminf-ennreal*[*simp*]:

$(\bigwedge i. 0 \leq f\ i) \Longrightarrow (\sum i. \text{ennreal } (f\ i)) \neq \text{top} \Longrightarrow (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } (\sum i. f\ i)$
 $\langle \text{proof} \rangle$

lemma *sums-enn2ereal*[*simp*]: $(\lambda i. \text{enn2ereal } (f\ i)) \text{ sums } \text{enn2ereal } x \longleftrightarrow f \text{ sums } x$

$\langle \text{proof} \rangle$

lemma *suminf-enn2ereal*[*simp*]: $(\sum i. \text{enn2ereal } (f\ i)) = \text{enn2ereal } (\text{suminf } f)$

$\langle \text{proof} \rangle$

lemma *transfer-e2ennreal-suminf* [*transfer-rule*]: $\text{rel-fun } (\text{rel-fun } (=) \text{ pcr-ennreal}) \text{ pcr-ennreal suminf suminf}$

$\langle \text{proof} \rangle$

lemma *ennreal-suminf-cmult*[*simp*]: $(\sum i. r * f\ i) = r * (\sum i. f\ i :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-suminf-multc*[*simp*]: $(\sum i. f\ i * r) = (\sum i. f\ i :: \text{ennreal}) * r$

$\langle \text{proof} \rangle$

lemma *ennreal-suminf-divide*[*simp*]: $(\sum i. f\ i / r) = (\sum i. f\ i :: \text{ennreal}) / r$

$\langle \text{proof} \rangle$

lemma *ennreal-suminf-neq-top*: $\text{summable } f \Longrightarrow (\bigwedge i. 0 \leq f\ i) \Longrightarrow (\sum i. \text{ennreal } (f\ i)) \neq \text{top}$

$\langle \text{proof} \rangle$

lemma *suminf-ennreal-eq*:

$(\bigwedge i. 0 \leq f\ i) \Longrightarrow f \text{ sums } x \Longrightarrow (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } x$

$\langle \text{proof} \rangle$

lemma *ennreal-suminf-bound-add*:

fixes $f :: \text{nat} \Rightarrow \text{ennreal}$

shows $(\bigwedge N. (\sum n < N. f\ n) + y \leq x) \Longrightarrow \text{suminf } f + y \leq x$

<proof>

lemma *ennreal-suminf-SUP-eq-directed*:

fixes $f :: 'a \Rightarrow \text{nat} \Rightarrow \text{ennreal}$

assumes $*$: $\bigwedge N\ i\ j. i \in I \Longrightarrow j \in I \Longrightarrow \text{finite } N \Longrightarrow \exists k \in I. \forall n \in N. f\ i\ n \leq f\ k\ n \wedge f\ j\ n \leq f\ k\ n$

shows $(\sum n. \text{SUP } i \in I. f\ i\ n) = (\text{SUP } i \in I. \sum n. f\ i\ n)$

<proof>

lemma *INF-ennreal-add-const*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$

shows $(\text{INF } i. f\ i + c) = (\text{INF } i. f\ i) + c$

<proof>

lemma *INF-ennreal-const-add*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$

shows $(\text{INF } i. c + f\ i) = c + (\text{INF } i. f\ i)$

<proof>

lemma *SUP-mult-left-ennreal*: $c * (\text{SUP } i \in I. f\ i) = (\text{SUP } i \in I. c * f\ i :: \text{ennreal})$

<proof>

lemma *SUP-mult-right-ennreal*: $(\text{SUP } i \in I. f\ i) * c = (\text{SUP } i \in I. f\ i * c :: \text{ennreal})$

<proof>

lemma *SUP-divide-ennreal*: $(\text{SUP } i \in I. f\ i) / c = (\text{SUP } i \in I. f\ i / c :: \text{ennreal})$

<proof>

lemma *ennreal-SUP-of-nat-eq-top*: $(\text{SUP } x. \text{of-nat } x :: \text{ennreal}) = \text{top}$

<proof>

lemma *ennreal-SUP-eq-top*:

fixes $f :: 'a \Rightarrow \text{ennreal}$

assumes $\bigwedge n. \exists i \in I. \text{of-nat } n \leq f\ i$

shows $(\text{SUP } i \in I. f\ i) = \text{top}$

<proof>

lemma *ennreal-INF-const-minus*:

fixes $f :: 'a \Rightarrow \text{ennreal}$

shows $I \neq \{\} \Longrightarrow (\text{SUP } x \in I. c - f\ x) = c - (\text{INF } x \in I. f\ x)$

<proof>

lemma *of-nat-Sup-ennreal*:

assumes $A \neq \{\}$ *bdd-above* A

shows $\text{of-nat } (\text{Sup } A) = (\text{SUP } a \in A. \text{of-nat } a :: \text{ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-tendsto-const-minus*:

fixes $g :: 'a \Rightarrow \text{ennreal}$

assumes $ae: \forall_F x \text{ in } F. g\ x \leq c$

assumes $g: ((\lambda x. c - g\ x) \longrightarrow 0) F$

shows $(g \longrightarrow c) F$

$\langle \text{proof} \rangle$

lemma *ennreal-SUP-add*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$

shows $\text{incseq } f \Longrightarrow \text{incseq } g \Longrightarrow (\text{SUP } i. f\ i + g\ i) = \text{Sup } (f\ ^\text{UNIV}) + \text{Sup } (g\ ^\text{UNIV})$

$\langle \text{proof} \rangle$

lemma *ennreal-SUP-sum*:

fixes $f :: 'a \Rightarrow \text{nat} \Rightarrow \text{ennreal}$

shows $(\bigwedge i. i \in I \Longrightarrow \text{incseq } (f\ i)) \Longrightarrow (\text{SUP } n. \sum_{i \in I} f\ i\ n) = (\sum_{i \in I} \text{SUP } n. f\ i\ n)$

$\langle \text{proof} \rangle$

lemma *ennreal-liminf-minus*:

fixes $f :: \text{nat} \Rightarrow \text{ennreal}$

shows $(\bigwedge n. f\ n \leq c) \Longrightarrow \text{liminf } (\lambda n. c - f\ n) = c - \text{limsup } f$

$\langle \text{proof} \rangle$

lemma *ennreal-continuous-on-cmult*:

$(c :: \text{ennreal}) < \text{top} \Longrightarrow \text{continuous-on } A\ f \Longrightarrow \text{continuous-on } A\ (\lambda x. c * f\ x)$

$\langle \text{proof} \rangle$

lemma *ennreal-tendsto-cmult*:

$(c :: \text{ennreal}) < \text{top} \Longrightarrow (f \longrightarrow x) F \Longrightarrow ((\lambda x. c * f\ x) \longrightarrow c * x) F$

$\langle \text{proof} \rangle$

lemma *tendsto-ennrealI*[intro, simp, tendsto-intros]:

$(f \longrightarrow x) F \Longrightarrow ((\lambda x. \text{ennreal } (f\ x)) \longrightarrow \text{ennreal } x) F$

$\langle \text{proof} \rangle$

lemma *tendsto-enn2erealI* [tendsto-intros]:

assumes $(f \longrightarrow l) F$

shows $((\lambda i. \text{enn2ereal}(f\ i)) \longrightarrow \text{enn2ereal } l) F$

$\langle \text{proof} \rangle$

lemma *tendsto-e2ennrealI* [tendsto-intros]:

assumes $(f \longrightarrow l) F$

shows $((\lambda i. \text{e2ennreal}(f\ i)) \longrightarrow \text{e2ennreal } l) F$

$\langle \text{proof} \rangle$

lemma *ennreal-suminf-minus*:

fixes $f\ g :: \text{nat} \Rightarrow \text{ennreal}$
shows $(\bigwedge i. g\ i \leq f\ i) \implies \text{suminf}\ f \neq \text{top} \implies \text{suminf}\ g \neq \text{top} \implies (\sum i. f\ i - g\ i) = \text{suminf}\ f - \text{suminf}\ g$
 $\langle \text{proof} \rangle$

lemma *ennreal-Sup-countable-SUP*:

$A \neq \{\}$ $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{incseq}\ f \wedge \text{range}\ f \subseteq A \wedge \text{Sup}\ A = (\text{SUP}\ i. f\ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-Inf-countable-INF*:

$A \neq \{\}$ $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{decseq}\ f \wedge \text{range}\ f \subseteq A \wedge \text{Inf}\ A = (\text{INF}\ i. f\ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-countable-SUP*:

$A \neq \{\}$ $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{range}\ f \subseteq g\ A \wedge \text{Sup}\ (g\ ` A) = \text{Sup}\ (f\ ` \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n :: \text{nat}. \text{of-nat}\ n :: \text{ennreal}) \longrightarrow \text{top}$
 $\langle \text{proof} \rangle$

lemma *SUP-sup-continuous-ennreal*:

fixes $f :: \text{ennreal} \Rightarrow 'a :: \text{complete-lattice}$
assumes f : *sup-continuous* f **and** $I \neq \{\}$
shows $(\text{SUP}\ i \in I. f\ (g\ i)) = f\ (\text{SUP}\ i \in I. g\ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-suminf-SUP-eq*:

fixes $f :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ennreal}$
shows $(\bigwedge i. \text{incseq}\ (\lambda n. f\ n\ i)) \implies (\sum i. \text{SUP}\ n. f\ n\ i) = (\text{SUP}\ n. \sum i. f\ n\ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-add-left*:

fixes $c :: \text{ennreal}$
shows $I \neq \{\} \implies (\text{SUP}\ i \in I. f\ i + c) = (\text{SUP}\ i \in I. f\ i) + c$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-const-minus*:

fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $I \neq \{\} \implies c < \text{top} \implies (\text{INF}\ x \in I. c - f\ x) = c - (\text{SUP}\ x \in I. f\ x)$
 $\langle \text{proof} \rangle$

lemma *isCont-ennreal[simp]*: $\langle \text{isCont}\ \text{ennreal}\ x \rangle$

$\langle \text{proof} \rangle$

lemma *isCont-ennreal-of-enat[simp]*: $\langle \text{isCont}\ \text{ennreal-of-enat}\ x \rangle$

$\langle \text{proof} \rangle$

40.9 Approximation lemmas

lemma *INF-approx-ennreal*:

fixes $x::ennreal$ **and** $e::real$
assumes $e > 0$
assumes $x = (\text{INF } i \in A. f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A. f\ i < x + e$
 $\langle \text{proof} \rangle$

lemma *SUP-approx-ennreal*:

fixes $x::ennreal$ **and** $e::real$
assumes $e > 0$ $A \neq \{\}$
assumes $\text{SUP}: x = (\text{SUP } i \in A. f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A. x < f\ i + e$
 $\langle \text{proof} \rangle$

lemma *ennreal-approx-SUP*:

fixes $x::ennreal$
assumes $f\text{-bound}: \bigwedge i. i \in A \implies f\ i \leq x$
assumes $\text{approx}: \bigwedge e. (e::real) > 0 \implies \exists i \in A. x \leq f\ i + e$
shows $x = (\text{SUP } i \in A. f\ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-approx-INF*:

fixes $x::ennreal$
assumes $f\text{-bound}: \bigwedge i. i \in A \implies x \leq f\ i$
assumes $\text{approx}: \bigwedge e. (e::real) > 0 \implies \exists i \in A. f\ i \leq x + e$
shows $x = (\text{INF } i \in A. f\ i)$
 $\langle \text{proof} \rangle$

lemma *ennreal-approx-unit*:

$(\bigwedge a::ennreal. 0 < a \implies a < 1 \implies a * z \leq y) \implies z \leq y$
 $\langle \text{proof} \rangle$

lemma *suminf-ennreal2*:

$(\bigwedge i. 0 \leq f\ i) \implies \text{summable } f \implies (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } (\sum i. f\ i)$
 $\langle \text{proof} \rangle$

lemma *less-top-ennreal*: $x < \text{top} \longleftrightarrow (\exists r \geq 0. x = \text{ennreal } r)$

$\langle \text{proof} \rangle$

lemma *enn2real-less-iff[simp]*: $x < \text{top} \implies \text{enn2real } x < c \longleftrightarrow x < c$

$\langle \text{proof} \rangle$

lemma *enn2real-le-iff[simp]*: $\llbracket x < \text{top}; c > 0 \rrbracket \implies \text{enn2real } x \leq c \longleftrightarrow x \leq c$

$\langle \text{proof} \rangle$

lemma *enn2real-less*:

assumes $enn2real\ e < r\ e \neq top$ **shows** $e < ennreal\ r$
 $\langle proof \rangle$

lemma *enn2real-le*:

assumes $enn2real\ e \leq r\ e \neq top$ **shows** $e \leq ennreal\ r$
 $\langle proof \rangle$

lemma *tendsto-top-iff-ennreal*:

fixes $f :: 'a \Rightarrow ennreal$
shows $(f \longrightarrow top)\ F \longleftrightarrow (\forall l \geq 0. eventually\ (\lambda x. ennreal\ l < f\ x)\ F)$
 $\langle proof \rangle$

lemma *ennreal-tendsto-top-eq-at-top*:

$((\lambda z. ennreal\ (f\ z)) \longrightarrow top)\ F \longleftrightarrow (LIM\ z\ F. f\ z :> at-top)$
 $\langle proof \rangle$

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:

fixes $f :: - \Rightarrow ennreal$
shows $Limsup\ F\ f = 0 \implies (f \longrightarrow 0)\ F$
 $\langle proof \rangle$

lemma *diff-le-self-ennreal[simp]*: $a - b \leq (a::ennreal)$

$\langle proof \rangle$

lemma *ennreal-ineq-diff-add*: $b \leq a \implies a = b + (a - b::ennreal)$

$\langle proof \rangle$

lemma *ennreal-mult-strict-left-mono*: $(a::ennreal) < c \implies 0 < b \implies b < top \implies$

$b * a < b * c$

$\langle proof \rangle$

lemma *ennreal-between*: $0 < e \implies 0 < x \implies x < top \implies x - e < (x::ennreal)$

$\langle proof \rangle$

lemma *minus-less-iff-ennreal*: $b < top \implies b \leq a \implies a - b < c \longleftrightarrow a < c +$

$(b::ennreal)$

$\langle proof \rangle$

lemma *tendsto-zero-ennreal*:

assumes $ev: \bigwedge r. 0 < r \implies \forall_F\ x\ in\ F. f\ x < ennreal\ r$

shows $(f \longrightarrow 0)\ F$

$\langle proof \rangle$

lifting-update *ennreal.lifting*

lifting-forget *ennreal.lifting*

40.10 *ennreal* theorems

lemma *neg-top-trans*: **fixes** $x\ y :: ennreal$ **shows** $\llbracket y \neq top; x \leq y \rrbracket \implies x \neq top$

$\langle \text{proof} \rangle$

lemma *diff-diff-ennreal*: **fixes** $a\ b :: \text{ennreal}$ **shows** $a \leq b \implies b \neq \infty \implies b - (b - a) = a$
 $\langle \text{proof} \rangle$

lemma *ennreal-less-one-iff[simp]*: $\text{ennreal } x < 1 \longleftrightarrow x < 1$
 $\langle \text{proof} \rangle$

lemma *SUP-const-minus-ennreal*:
fixes $f :: 'a \Rightarrow \text{ennreal}$ **shows** $I \neq \{\} \implies (\text{SUP } x \in I. c - f\ x) = c - (\text{INF } x \in I. f\ x)$
including *ennreal.lifting*
 $\langle \text{proof} \rangle$

lemma *zero-minus-ennreal[simp]*: $0 - (a :: \text{ennreal}) = 0$
including *ennreal.lifting*
 $\langle \text{proof} \rangle$

lemma *diff-diff-commute-ennreal*:
fixes $a\ b\ c :: \text{ennreal}$ **shows** $a - b - c = a - c - b$
 $\langle \text{proof} \rangle$

lemma *diff-gr0-ennreal*: $b < (a :: \text{ennreal}) \implies 0 < a - b$
including *ennreal.lifting* $\langle \text{proof} \rangle$

lemma *divide-le-posI-ennreal*:
fixes $x\ y\ z :: \text{ennreal}$
shows $x > 0 \implies z \leq x * y \implies z / x \leq y$
 $\langle \text{proof} \rangle$

lemma *add-diff-eq-ennreal*:
fixes $x\ y\ z :: \text{ennreal}$
shows $z \leq y \implies x + (y - z) = x + y - z$
 $\langle \text{proof} \rangle$

lemma *add-diff-inverse-ennreal*:
fixes $x\ y :: \text{ennreal}$ **shows** $x \leq y \implies x + (y - x) = y$
 $\langle \text{proof} \rangle$

lemma *add-diff-eq-iff-ennreal[simp]*:
fixes $x\ y :: \text{ennreal}$ **shows** $x + (y - x) = y \longleftrightarrow x \leq y$
 $\langle \text{proof} \rangle$

lemma *add-diff-le-ennreal*: $a + b - c \leq a + (b - c :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *diff-eq-0-ennreal*: $a < \text{top} \implies a \leq b \implies a - b = (0 :: \text{ennreal})$
 $\langle \text{proof} \rangle$

lemma *diff-diff-ennreal'*: **fixes** $x\ y\ z :: \text{ennreal}$ **shows** $z \leq y \implies y - z \leq x \implies x - (y - z) = x + z - y$
 ⟨proof⟩

lemma *diff-diff-ennreal''*: **fixes** $x\ y\ z :: \text{ennreal}$
shows $z \leq y \implies x - (y - z) = (\text{if } y - z \leq x \text{ then } x + z - y \text{ else } 0)$
 ⟨proof⟩

lemma *power-less-top-ennreal*: **fixes** $x :: \text{ennreal}$ **shows** $x \wedge n < \text{top} \longleftrightarrow x < \text{top} \vee n = 0$
 ⟨proof⟩

lemma *ennreal-divide-times*: $(a / b) * c = a * (c / b :: \text{ennreal})$
 ⟨proof⟩

lemma *diff-less-top-ennreal*: $a - b < \text{top} \longleftrightarrow a < (\text{top} :: \text{ennreal})$
 ⟨proof⟩

lemma *divide-less-ennreal*: $b \neq 0 \implies b < \text{top} \implies a / b < c \longleftrightarrow a < (c * b :: \text{ennreal})$
 ⟨proof⟩

lemma *one-less-numeral[simp]*: $1 < (\text{numeral } n :: \text{ennreal}) \longleftrightarrow (\text{num.One} < n)$
 ⟨proof⟩

lemma *divide-eq-1-ennreal*: $a / b = (1 :: \text{ennreal}) \longleftrightarrow (b \neq \text{top} \wedge b \neq 0 \wedge b = a)$
 ⟨proof⟩

lemma *ennreal-mult-cancel-left*: $(a * b = a * c) = (a = \text{top} \wedge b \neq 0 \wedge c \neq 0 \vee a = 0 \vee b = (c :: \text{ennreal}))$
 ⟨proof⟩

lemma *ennreal-minus-if*: $\text{ennreal } a - \text{ennreal } b = \text{ennreal } (\text{if } 0 \leq b \text{ then } (\text{if } b \leq a \text{ then } a - b \text{ else } 0) \text{ else } a)$
 ⟨proof⟩

lemma *ennreal-plus-if*: $\text{ennreal } a + \text{ennreal } b = \text{ennreal } (\text{if } 0 \leq a \text{ then } (\text{if } 0 \leq b \text{ then } a + b \text{ else } a) \text{ else } b)$
 ⟨proof⟩

lemma *ennreal-diff-le-mono-left*: $a \leq b \implies a - c \leq (b :: \text{ennreal})$
 ⟨proof⟩

lemma *ennreal-minus-le-iff*: $a - b \leq c \longleftrightarrow (a \leq b + (c :: \text{ennreal}) \wedge (a = \text{top} \wedge b = \text{top} \longrightarrow c = \text{top}))$
 ⟨proof⟩

lemma *ennreal-le-minus-iff*: $a \leq b - c \longleftrightarrow (a + c \leq (b :: \text{ennreal}) \vee (a = 0 \wedge b = c))$

$\leq c))$
 $\langle \text{proof} \rangle$

lemma *diff-add-eq-diff-diff-swap-ennreal*: $x - (y + z :: \text{ennreal}) = x - y - z$
 $\langle \text{proof} \rangle$

lemma *diff-add-assoc2-ennreal*: $b \leq a \implies (a - b + c :: \text{ennreal}) = a + c - b$
 $\langle \text{proof} \rangle$

lemma *diff-gt-0-iff-gt-ennreal*: $0 < a - b \longleftrightarrow (a = \text{top} \wedge b = \text{top} \vee b < (a :: \text{ennreal}))$
 $\langle \text{proof} \rangle$

lemma *diff-eq-0-iff-ennreal*: $(a - b :: \text{ennreal}) = 0 \longleftrightarrow (a < \text{top} \wedge a \leq b)$
 $\langle \text{proof} \rangle$

lemma *add-diff-self-ennreal*: $a + (b - a :: \text{ennreal}) = (\text{if } a \leq b \text{ then } b \text{ else } a)$
 $\langle \text{proof} \rangle$

lemma *diff-add-self-ennreal*: $(b - a + a :: \text{ennreal}) = (\text{if } a \leq b \text{ then } b \text{ else } a)$
 $\langle \text{proof} \rangle$

lemma *ennreal-minus-cancel-iff*:
fixes $a \ b \ c :: \text{ennreal}$
shows $a - b = a - c \longleftrightarrow (b = c \vee (a \leq b \wedge a \leq c) \vee a = \text{top})$
 $\langle \text{proof} \rangle$

The next lemma is wrong for $a = \text{top}$, for $b = c = 1$ for instance.

lemma *ennreal-right-diff-distrib*:
fixes $a \ b \ c :: \text{ennreal}$
assumes $a \neq \text{top}$
shows $a * (b - c) = a * b - a * c$
 $\langle \text{proof} \rangle$

lemma *SUP-diff-ennreal*:
 $c < \text{top} \implies (\text{SUP } i \in I. f \ i - c :: \text{ennreal}) = (\text{SUP } i \in I. f \ i) - c$
 $\langle \text{proof} \rangle$

lemma *ennreal-SUP-add-right*:
fixes $c :: \text{ennreal}$ **shows** $I \neq \{\}$ $\implies c + (\text{SUP } i \in I. f \ i) = (\text{SUP } i \in I. c + f \ i)$
 $\langle \text{proof} \rangle$

lemma *SUP-add-directed-ennreal*:
fixes $f \ g :: - \Rightarrow \text{ennreal}$
assumes *directed*: $\bigwedge i \ j. i \in I \implies j \in I \implies \exists k \in I. f \ i + g \ j \leq f \ k + g \ k$
shows $(\text{SUP } i \in I. f \ i + g \ i) = (\text{SUP } i \in I. f \ i) + (\text{SUP } i \in I. g \ i)$
 $\langle \text{proof} \rangle$

lemma *enn2real-eq-0-iff*: $\text{enn2real } x = 0 \longleftrightarrow x = 0 \vee x = \text{top}$

$\langle \text{proof} \rangle$

lemma *continuous-on-diff-ennreal*:

continuous-on $A \ f \implies \text{continuous-on } A \ g \implies (\bigwedge x. x \in A \implies f \ x \neq \text{top}) \implies$
 $(\bigwedge x. x \in A \implies g \ x \neq \text{top}) \implies \text{continuous-on } A \ (\lambda z. f \ z - g \ z :: \text{ennreal})$

including *ennreal.lifting*

$\langle \text{proof} \rangle$

lemma *tendsto-diff-ennreal*:

$(f \longrightarrow x) \ F \implies (g \longrightarrow y) \ F \implies x \neq \text{top} \implies y \neq \text{top} \implies ((\lambda z. f \ z - g \ z :: \text{ennreal}) \longrightarrow x - y) \ F$

$\langle \text{proof} \rangle$

declare *lim-real-of-ereal* [*tendsto-intros*]

lemma *tendsto-enn2real* [*tendsto-intros*]:

assumes $(u \longrightarrow \text{ennreal } l) \ F \ l \geq 0$

shows $((\lambda n. \text{enn2real } (u \ n)) \longrightarrow l) \ F$

$\langle \text{proof} \rangle$

end

41 Logarithm of Natural Numbers

theory *Log-Nat*

imports *Complex-Main*

begin

41.1 Preliminaries

lemma *divide-nat-diff-div-nat-less-one*:

real $x \ / \ \text{real } b - \text{real } (x \ \text{div } b) < 1$ **for** $x \ b :: \text{nat}$

$\langle \text{proof} \rangle$

41.2 Floorlog

definition *floorlog* $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where *floorlog* $b \ a = (\text{if } a > 0 \wedge b > 1 \text{ then } \text{nat } \lfloor \log b \ a \rfloor + 1 \text{ else } 0)$

lemma *floorlog-mono*: $x \leq y \implies \text{floorlog } b \ x \leq \text{floorlog } b \ y$

$\langle \text{proof} \rangle$

lemma *floorlog-bounds*:

$b \wedge (\text{floorlog } b \ x - 1) \leq x \wedge x < b \wedge (\text{floorlog } b \ x) \text{ if } x > 0 \ b > 1$

$\langle \text{proof} \rangle$

lemma *floorlog-power* [*simp*]:

floorlog $b \ (a * b \wedge c) = \text{floorlog } b \ a + c$ **if** $a > 0 \ b > 1$

$\langle \text{proof} \rangle$

lemma *floor-log-add-eqI*:

$\lfloor \log b (a + r) \rfloor = \lfloor \log b a \rfloor$ **if** $b > 1$ $a \geq 1$ $0 \leq r$ $r < 1$
for $a \ b :: \text{nat}$ **and** $r :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *floor-log-div*:

$\lfloor \log b x \rfloor = \lfloor \log b (x \text{ div } b) \rfloor + 1$ **if** $b > 1$ $x > 0$ $x \text{ div } b > 0$
for $b \ x :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *compute-floorlog* [code]:

$\text{floorlog } b \ x = (\text{if } x > 0 \wedge b > 1 \text{ then } \text{floorlog } b (x \text{ div } b) + 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *floor-log-eq-if*:

$\lfloor \log b x \rfloor = \lfloor \log b y \rfloor$ **if** $x \text{ div } b = y \text{ div } b$ $b > 1$ $x > 0$ $x \text{ div } b \geq 1$
for $b \ x \ y :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *floorlog-eq-if*:

$\text{floorlog } b \ x = \text{floorlog } b \ y$ **if** $x \text{ div } b = y \text{ div } b$ $b > 1$ $x > 0$ $x \text{ div } b \geq 1$
for $b \ x \ y :: \text{nat}$
 $\langle \text{proof} \rangle$

lemma *floorlog-leD*:

$\text{floorlog } b \ x \leq w \implies b > 1 \implies x < b^w$
 $\langle \text{proof} \rangle$

lemma *floorlog-leI*:

$x < b^w \implies 0 \leq w \implies b > 1 \implies \text{floorlog } b \ x \leq w$
 $\langle \text{proof} \rangle$

lemma *floorlog-eq-zero-iff*:

$\text{floorlog } b \ x = 0 \iff b \leq 1 \vee x \leq 0$
 $\langle \text{proof} \rangle$

lemma *floorlog-le-iff*:

$\text{floorlog } b \ x \leq w \iff b \leq 1 \vee b > 1 \wedge 0 \leq w \wedge x < b^w$
 $\langle \text{proof} \rangle$

lemma *floorlog-ge-SucI*:

$\text{Suc } w \leq \text{floorlog } b \ x$ **if** $b^w \leq x$ $b > 1$
 $\langle \text{proof} \rangle$

lemma *floorlog-geI*:

$w \leq \text{floorlog } b \ x$ **if** $b^w \leq x$ $b > 1$
 $\langle \text{proof} \rangle$

lemma *floorlog-geD*:

$b \wedge (w - 1) \leq x$ **if** $w \leq \text{floorlog } b \ x \ w > 0$
 $\langle \text{proof} \rangle$

41.3

definition *ceillog2* :: $\text{nat} \Rightarrow \text{nat}$ **where**

$\text{ceillog2 } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \text{nat } \lceil \log 2 (\text{real } n) \rceil)$

lemma *ceillog2-0* [simp]: $\text{ceillog2 } 0 = 0$

and *ceillog2-Suc-0* [simp]: $\text{ceillog2 } (\text{Suc } 0) = 0$

and *ceillog2-2* [simp]: $\text{ceillog2 } 2 = 1$

$\langle \text{proof} \rangle$

lemma *ceillog2-le1-eq-0* [simp]: $n \leq 1 \implies \text{ceillog2 } n = 0$

$\langle \text{proof} \rangle$

lemma *ceillog2-2-power* [simp]: $\text{ceillog2 } (2 \wedge n) = n$

$\langle \text{proof} \rangle$

lemma *ceillog2-ge-log*:

assumes $n > 0$

shows $\text{real } (\text{ceillog2 } n) \geq \log 2 (\text{real } n)$

$\langle \text{proof} \rangle$

lemma *ceillog2-less-log*:

assumes $n > 0$

shows $\text{real } (\text{ceillog2 } n) < \log 2 (\text{real } n) + 1$

$\langle \text{proof} \rangle$

lemma *ceillog2-le-iff*:

assumes $n > 0$

shows $\text{ceillog2 } n \leq l \iff n \leq 2 \wedge l$

$\langle \text{proof} \rangle$

lemma *ceillog2-ge-iff*:

assumes $n > 0$

shows $\text{ceillog2 } n \geq l \iff 2 \wedge l < 2 * n$

$\langle \text{proof} \rangle$

lemma *le-two-power-ceillog2*: $n \leq 2 \wedge \text{ceillog2 } n$

$\langle \text{proof} \rangle$

lemma *two-power-ceillog2-gt*:

assumes $n > 0$

shows $2 * n > 2 \wedge \text{ceillog2 } n$

$\langle \text{proof} \rangle$

lemma *ceillog2-eqI*:

assumes $n \leq 2 \wedge l \ 2 \wedge l < 2 * n$
shows $\text{ceillog2 } n = l$
 $\langle \text{proof} \rangle$

lemma *ceillog2-rec-even*:
assumes $k > 0$
shows $\text{ceillog2 } (2 * k) = \text{Suc } (\text{ceillog2 } k)$
 $\langle \text{proof} \rangle$

lemma *ceillog2-mono*:
assumes $m \leq n$
shows $\text{ceillog2 } m \leq \text{ceillog2 } n$
 $\langle \text{proof} \rangle$

lemma *ceillog2-rec-odd*:
assumes $k > 0$
shows $\text{ceillog2 } (\text{Suc } (2 * k)) = \text{Suc } (\text{ceillog2 } (\text{Suc } k))$
 $\langle \text{proof} \rangle$

lemma *ceillog2-rec*:
 $\text{ceillog2 } n = (\text{if } n \leq 1 \text{ then } 0 \text{ else } 1 + \text{ceillog2 } ((n + 1) \text{ div } 2))$
 $\langle \text{proof} \rangle$

lemma *funpow-div2-ceillog2-le-1*:
 $((\lambda n. (n + 1) \text{ div } 2) \text{ `` } \text{ceillog2 } n) \ n \leq 1$
 $\langle \text{proof} \rangle$

fun *ceillog2-aux* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{ceillog2-aux } \text{acc } n = (\text{if } n \leq 1 \text{ then } \text{acc} \text{ else } \text{ceillog2-aux } (\text{acc} + 1) ((n + 1) \text{ div } 2))$

lemmas $[\text{simp del}] = \text{ceillog2-aux.simps}$

lemma *ceillog2-aux-correct*: $\text{ceillog2-aux } \text{acc } n = \text{ceillog2 } n + \text{acc}$
 $\langle \text{proof} \rangle$

lemma *ceillog2-code* [code]: $\text{ceillog2 } n = \text{ceillog2-aux } 0 \ n$
 $\langle \text{proof} \rangle$

41.4 Bitlen

definition *bitlen* :: $\text{int} \Rightarrow \text{int}$
where $\text{bitlen } a = \text{floorlog } 2 \ (\text{nat } a)$

lemma *bitlen-alt-def*:
 $\text{bitlen } a = (\text{if } a > 0 \text{ then } \lfloor \log 2 \ a \rfloor + 1 \text{ else } 0)$

$\langle \text{proof} \rangle$

lemma *bitlen-zero* [*simp*]:

$\text{bitlen } 0 = 0$

$\langle \text{proof} \rangle$

lemma *bitlen-nonneg*:

$0 \leq \text{bitlen } x$

$\langle \text{proof} \rangle$

lemma *bitlen-bounds*:

$2^{\text{nat } (\text{bitlen } x - 1)} \leq x \wedge x < 2^{\text{nat } (\text{bitlen } x)} \text{ if } x > 0$
 $\langle \text{proof} \rangle$

lemma *bitlen-pow2* [*simp*]:

$\text{bitlen } (b * 2^c) = \text{bitlen } b + c \text{ if } b > 0$

$\langle \text{proof} \rangle$

lemma *compute-bitlen* [*code*]:

$\text{bitlen } x = (\text{if } x > 0 \text{ then } \text{bitlen } (x \text{ div } 2) + 1 \text{ else } 0)$

$\langle \text{proof} \rangle$

lemma *bitlen-eq-zero-iff*:

$\text{bitlen } x = 0 \iff x \leq 0$

$\langle \text{proof} \rangle$

lemma *bitlen-div*:

$1 \leq \text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)}$

and $\text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)} < 2 \text{ if } 0 < m$
 $\langle \text{proof} \rangle$

lemma *bitlen-le-iff-floorlog*:

$\text{bitlen } x \leq w \iff w \geq 0 \wedge \text{floorlog } 2 (\text{nat } x) \leq \text{nat } w$

$\langle \text{proof} \rangle$

lemma *bitlen-le-iff-power*:

$\text{bitlen } x \leq w \iff w \geq 0 \wedge x < 2^{\text{nat } w}$

$\langle \text{proof} \rangle$

lemma *less-power-nat-iff-bitlen*:

$x < 2^w \iff \text{bitlen } (\text{int } x) \leq w$

$\langle \text{proof} \rangle$

lemma *bitlen-ge-iff-power*:

$w \leq \text{bitlen } x \iff w \leq 0 \vee 2^{\text{nat } w - 1} \leq x$

$\langle \text{proof} \rangle$

lemma *bitlen-twopow-add-eq*:

$\text{bitlen } (2^w + b) = w + 1 \text{ if } 0 \leq b < 2^w$

$\langle proof \rangle$

end

42 Various algebraic structures combined with a lattice

theory *Lattice-Algebras*
imports *Complex-Main*
begin

class *semilattice-inf-ab-group-add* = *ordered-ab-group-add* + *semilattice-inf*
begin

lemma *add-inf-distrib-left*: $a + \inf b\ c = \inf (a + b)\ (a + c)$ (**is** ?L=?R)
 $\langle proof \rangle$

lemma *add-inf-distrib-right*: $\inf a\ b + c = \inf (a + c)\ (b + c)$
 $\langle proof \rangle$

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*: $a + \sup b\ c = \sup (a + b)\ (a + c)$ (**is** ?L = ?R)
 $\langle proof \rangle$

lemma *add-sup-distrib-right*: $\sup a\ b + c = \sup (a + c)\ (b + c)$
 $\langle proof \rangle$

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*
begin

subclass *semilattice-inf-ab-group-add* $\langle proof \rangle$

subclass *semilattice-sup-ab-group-add* $\langle proof \rangle$

lemmas *add-sup-inf-distrib* =
add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

lemma *inf-eq-neg-sup*: $\inf a\ b = - \sup (- a)\ (- b)$
 $\langle proof \rangle$

lemma *sup-eq-neg-inf*: $\sup a\ b = - \inf (- a)\ (- b)$
 $\langle proof \rangle$

lemma *neg-inf-eq-sup*: $- \inf a \ b = \sup (- a) (- b)$
 $\langle \text{proof} \rangle$

lemma *diff-inf-eq-sup*: $a - \inf b \ c = a + \sup (- b) (- c)$
 $\langle \text{proof} \rangle$

lemma *neg-sup-eq-inf*: $- \sup a \ b = \inf (- a) (- b)$
 $\langle \text{proof} \rangle$

lemma *diff-sup-eq-inf*: $a - \sup b \ c = a + \inf (- b) (- c)$
 $\langle \text{proof} \rangle$

lemma *add-eq-inf-sup*: $a + b = \sup a \ b + \inf a \ b$
 $\langle \text{proof} \rangle$

42.1 Positive Part, Negative Part, Absolute Value

definition *npert* :: $'a \Rightarrow 'a$
where *npert* $x = \inf x \ 0$

definition *pprt* :: $'a \Rightarrow 'a$
where *pprt* $x = \sup x \ 0$

lemma *pprt-neg*: $\text{pprt } (- x) = - \text{npert } x$
 $\langle \text{proof} \rangle$

lemma *npert-neg*: $\text{npert } (- x) = - \text{pprt } x$
 $\langle \text{proof} \rangle$

lemma *prts*: $a = \text{pprt } a + \text{npert } a$
 $\langle \text{proof} \rangle$

lemma *zero-le-pprt[simp]*: $0 \leq \text{pprt } a$
 $\langle \text{proof} \rangle$

lemma *npert-le-zero[simp]*: $\text{npert } a \leq 0$
 $\langle \text{proof} \rangle$

lemma *le-eq-neg*: $a \leq - b \longleftrightarrow a + b \leq 0$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *pprt-0[simp]*: $\text{pprt } 0 = 0$ $\langle \text{proof} \rangle$

lemma *npert-0[simp]*: $\text{npert } 0 = 0$ $\langle \text{proof} \rangle$

lemma *pprt-eq-id [simp, no-atp]*: $0 \leq x \implies \text{pprt } x = x$
 $\langle \text{proof} \rangle$

lemma *npert-eq-id [simp, no-atp]*: $x \leq 0 \implies \text{npert } x = x$

$\langle \text{proof} \rangle$

lemma *pprt-eq-0* [*simp*, *no-atp*]: $x \leq 0 \implies \text{pprt } x = 0$
 $\langle \text{proof} \rangle$

lemma *nprrt-eq-0* [*simp*, *no-atp*]: $0 \leq x \implies \text{nprrt } x = 0$
 $\langle \text{proof} \rangle$

lemma *sup-0-imp-0*:
 assumes $\text{sup } a \ (-\ a) = 0$
 shows $a = 0$
 $\langle \text{proof} \rangle$

lemma *inf-0-imp-0*: $\text{inf } a \ (-\ a) = 0 \implies a = 0$
 $\langle \text{proof} \rangle$

lemma *inf-0-eq-0* [*simp*]: $\text{inf } a \ (-\ a) = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *sup-0-eq-0* [*simp*]: $\text{sup } a \ (-\ a) = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: $0 \leq a + a \longleftrightarrow 0 \leq a$
 (is $?lhs \longleftrightarrow ?rhs$)
 $\langle \text{proof} \rangle$

lemma *double-zero* [*simp*]: $a + a = 0 \longleftrightarrow a = 0$
 $\langle \text{proof} \rangle$

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]: $0 < a + a \longleftrightarrow 0 < a$
 $\langle \text{proof} \rangle$

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: $a + a \leq 0 \longleftrightarrow a \leq 0$
 $\langle \text{proof} \rangle$

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]: $a + a < 0 \longleftrightarrow a < 0$
 $\langle \text{proof} \rangle$

declare *neg-inf-eq-sup* [*simp*]
 and *neg-sup-eq-inf* [*simp*]
 and *diff-inf-eq-sup* [*simp*]
 and *diff-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq -\ a \longleftrightarrow a \leq 0$
 $\langle \text{proof} \rangle$

lemma *minus-le-self-iff*: $-\ a \leq a \longleftrightarrow 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow \text{nprt } a = 0$
 ⟨proof⟩

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow \text{pprt } a = 0$
 ⟨proof⟩

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow \text{pprt } a = a$
 ⟨proof⟩

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow \text{nprt } a = a$
 ⟨proof⟩

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \implies \text{pprt } a \leq \text{pprt } b$
 ⟨proof⟩

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \implies \text{nprt } a \leq \text{nprt } b$
 ⟨proof⟩

end

lemmas *add-sup-inf-distrib* =
add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

class *lattice-ab-group-add-abs* = *lattice-ab-group-add* + *abs* +
assumes *abs-lattice*: $|a| = \text{sup } a \ (-\ a)$
begin

lemma *abs-prts*: $|a| = \text{pprt } a - \text{nprt } a$
 ⟨proof⟩

subclass *ordered-ab-group-add-abs*
 ⟨proof⟩

end

lemma *sup-eq-if*:
fixes $a :: 'a :: \{\text{lattice-ab-group-add}, \text{linorder}\}$
shows $\text{sup } a \ (-\ a) = (\text{if } a < 0 \text{ then } -\ a \text{ else } a)$
 ⟨proof⟩

lemma *abs-if-lattice*:
fixes $a :: 'a :: \{\text{lattice-ab-group-add-abs}, \text{linorder}\}$
shows $|a| = (\text{if } a < 0 \text{ then } -\ a \text{ else } a)$
 ⟨proof⟩

lemma *estimate-by-abs*:
fixes $a\ b\ c :: 'a :: \text{lattice-ab-group-add-abs}$
assumes $a + b \leq c$

shows $a \leq c + |b|$
 $\langle proof \rangle$

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
begin

subclass *semilattice-inf-ab-group-add* $\langle proof \rangle$
subclass *semilattice-sup-ab-group-add* $\langle proof \rangle$

end

lemma *abs-le-mult*:
fixes $a\ b :: 'a::lattice-ring$
shows $|a * b| \leq |a| * |b|$
 $\langle proof \rangle$

instance *lattice-ring* \subseteq *ordered-ring-abs*
 $\langle proof \rangle$

lemma *mult-le-prts*:
fixes $a\ b :: 'a::lattice-ring$
assumes $a1 \leq a$
and $a \leq a2$
and $b1 \leq b$
and $b \leq b2$
shows $a * b \leq$
 $pprt\ a2 * pprt\ b2 + pprt\ a1 * nprt\ b2 + nprt\ a2 * pprt\ b1 + nprt\ a1 * nprt$
 $b1$
 $\langle proof \rangle$

lemma *mult-ge-prts*:
fixes $a\ b :: 'a::lattice-ring$
assumes $a1 \leq a$
and $a \leq a2$
and $b1 \leq b$
and $b \leq b2$
shows $a * b \geq$
 $nprrt\ a1 * pprt\ b2 + nprrt\ a2 * nprrt\ b2 + pprt\ a1 * pprt\ b1 + pprt\ a2 * nprrt$
 $b1$
 $\langle proof \rangle$

instance *int* :: *lattice-ring*
 $\langle proof \rangle$

instance *real* :: *lattice-ring*
 $\langle proof \rangle$

end

43 Floating-Point Numbers

```

theory Float
imports Log-Nat Lattice-Algebras
begin

definition float = {m * 2 powr e | (m :: int) (e :: int). True}

typedef float = float
  morphisms real-of-float float-of
    ⟨proof⟩

setup-lifting type-definition-float

declare real-of-float [code-unfold]

lemmas float-of-inject[simp]

declare [[coercion real-of-float :: float ⇒ real]]

lemma real-of-float-eq: f1 = f2  $\longleftrightarrow$  real-of-float f1 = real-of-float f2 for f1 f2 ::
  float
  ⟨proof⟩

declare real-of-float-inverse[simp] float-of-inverse [simp]
declare real-of-float [simp]



### 43.1 Real operations preserving the representation as floating point number

lemma floatI: m * 2 powr e = x  $\implies$  x ∈ float for m e :: int
  ⟨proof⟩

lemma zero-float[simp]: 0 ∈ float
  ⟨proof⟩

lemma one-float[simp]: 1 ∈ float
  ⟨proof⟩

lemma numeral-float[simp]: numeral i ∈ float
  ⟨proof⟩

lemma neg-numeral-float[simp]: − numeral i ∈ float
  ⟨proof⟩

lemma real-of-int-float[simp]: real-of-int x ∈ float for x :: int
  ⟨proof⟩

lemma real-of-nat-float[simp]: real x ∈ float for x :: nat

```

<proof>

lemma *two-powr-int-float[simp]*: $2^{\text{powr } (\text{real-of-int } i)} \in \text{float}$ **for** $i :: \text{int}$
<proof>

lemma *two-powr-nat-float[simp]*: $2^{\text{powr } (\text{real } i)} \in \text{float}$ **for** $i :: \text{nat}$
<proof>

lemma *two-powr-minus-int-float[simp]*: $2^{\text{powr } - (\text{real-of-int } i)} \in \text{float}$ **for** $i :: \text{int}$
<proof>

lemma *two-powr-minus-nat-float[simp]*: $2^{\text{powr } - (\text{real } i)} \in \text{float}$ **for** $i :: \text{nat}$
<proof>

lemma *two-powr-numeral-float[simp]*: $2^{\text{powr } \text{numeral } i} \in \text{float}$
<proof>

lemma *two-powr-neg-numeral-float[simp]*: $2^{\text{powr } - \text{numeral } i} \in \text{float}$
<proof>

lemma *two-pow-float[simp]*: $2^{\wedge n} \in \text{float}$
<proof>

lemma *plus-float[simp]*: $r \in \text{float} \implies p \in \text{float} \implies r + p \in \text{float}$
<proof>

lemma *uminus-float[simp]*: $x \in \text{float} \implies -x \in \text{float}$
<proof>

lemma *times-float[simp]*: $x \in \text{float} \implies y \in \text{float} \implies x * y \in \text{float}$
<proof>

lemma *minus-float[simp]*: $x \in \text{float} \implies y \in \text{float} \implies x - y \in \text{float}$
<proof>

lemma *abs-float[simp]*: $x \in \text{float} \implies |x| \in \text{float}$
<proof>

lemma *sgn-of-float[simp]*: $x \in \text{float} \implies \text{sgn } x \in \text{float}$
<proof>

lemma *div-power-2-float[simp]*: $x \in \text{float} \implies x / 2^{\wedge d} \in \text{float}$
<proof>

lemma *div-power-2-int-float[simp]*: $x \in \text{float} \implies x / (2::\text{int})^{\wedge d} \in \text{float}$
<proof>

lemma *div-numeral-Bit0-float[simp]*:

assumes $x / \text{numeral } n \in \text{float}$
shows $x / (\text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *div-neg-numeral-Bit0-float[simp]*:
assumes $x / \text{numeral } n \in \text{float}$
shows $x / (- \text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$
 $\langle \text{proof} \rangle$

lemma *power-float[simp]*:
assumes $a \in \text{float}$
shows $a ^ b \in \text{float}$
 $\langle \text{proof} \rangle$

lift-definition *Float* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{float}$ **is** $\lambda(m::\text{int}) (e::\text{int}). m * 2^{\text{powr } e}$
 $\langle \text{proof} \rangle$
declare *Float.rep-eq[simp]*

code-datatype *Float*

lemma *compute-real-of-float[code]*:
 $\text{real-of-float } (\text{Float } m \ e) = (\text{if } e \geq 0 \text{ then } m * 2^{\text{nat } e} \text{ else } m / 2^{\text{nat } (-e)})$
 $\langle \text{proof} \rangle$

43.2 Arithmetic operations on floating point numbers

instantiation *float* :: $\{\text{ring-1}, \text{linorder}, \text{linordered-ring}, \text{linordered-idom}, \text{numeral}, \text{equal}\}$
begin

lift-definition *zero-float* :: *float* **is** 0 $\langle \text{proof} \rangle$
declare *zero-float.rep-eq[simp]*

lift-definition *one-float* :: *float* **is** 1 $\langle \text{proof} \rangle$
declare *one-float.rep-eq[simp]*

lift-definition *plus-float* :: *float* \Rightarrow *float* \Rightarrow *float* **is** (+) $\langle \text{proof} \rangle$
declare *plus-float.rep-eq[simp]*

lift-definition *times-float* :: *float* \Rightarrow *float* \Rightarrow *float* **is** (*) $\langle \text{proof} \rangle$
declare *times-float.rep-eq[simp]*

lift-definition *minus-float* :: *float* \Rightarrow *float* \Rightarrow *float* **is** (-) $\langle \text{proof} \rangle$
declare *minus-float.rep-eq[simp]*

lift-definition *uminus-float* :: *float* \Rightarrow *float* **is** *uminus* $\langle \text{proof} \rangle$
declare *uminus-float.rep-eq[simp]*

lift-definition *abs-float* :: *float* \Rightarrow *float* **is** *abs* $\langle \text{proof} \rangle$
declare *abs-float.rep-eq[simp]*

```

lift-definition sgn-float :: float  $\Rightarrow$  float is sgn  $\langle proof \rangle$ 
declare sgn-float.rep-eq[simp]

lift-definition equal-float :: float  $\Rightarrow$  float  $\Rightarrow$  bool is  $(=)$  :: real  $\Rightarrow$  real  $\Rightarrow$  bool
 $\langle proof \rangle$ 

lift-definition less-eq-float :: float  $\Rightarrow$  float  $\Rightarrow$  bool is  $(\leq)$   $\langle proof \rangle$ 
declare less-eq-float.rep-eq[simp]

lift-definition less-float :: float  $\Rightarrow$  float  $\Rightarrow$  bool is  $(<)$   $\langle proof \rangle$ 
declare less-float.rep-eq[simp]

instance
   $\langle proof \rangle$ 

end

lemma real-of-float [simp]: real-of-float (of-nat n) = of-nat n
   $\langle proof \rangle$ 

lemma real-of-float-of-int-eq [simp]: real-of-float (of-int z) = of-int z
   $\langle proof \rangle$ 

lemma Float-0-eq-0[simp]: Float 0 e = 0
   $\langle proof \rangle$ 

lemma real-of-float-power[simp]: real-of-float ( $f^n$ ) = real-of-float  $f^n$  for f :: float
   $\langle proof \rangle$ 

lemma real-of-float-min: real-of-float (min x y) = min (real-of-float x) (real-of-float y)
and real-of-float-max: real-of-float (max x y) = max (real-of-float x) (real-of-float y)
for x y :: float
   $\langle proof \rangle$ 

instance float :: unbounded-dense-linorder
   $\langle proof \rangle$ 

instantiation float :: lattice-ab-group-add
begin

definition inf-float :: float  $\Rightarrow$  float  $\Rightarrow$  float
  where inf-float a b = min a b

definition sup-float :: float  $\Rightarrow$  float  $\Rightarrow$  float
  where sup-float a b = max a b

```

instance
 $\langle proof \rangle$

end

lemma *float-numeral*[simp]: *real-of-float* (*numeral* *x* :: *float*) = *numeral* *x*
 $\langle proof \rangle$

lemma *transfer-numeral* [*transfer-rule*]:
rel-fun (=) *pcr-float* (*numeral* :: - \Rightarrow *real*) (*numeral* :: - \Rightarrow *float*)
 $\langle proof \rangle$

lemma *float-neg-numeral*[simp]: *real-of-float* (- *numeral* *x* :: *float*) = - *numeral* *x*
 $\langle proof \rangle$

lemma *transfer-neg-numeral* [*transfer-rule*]:
rel-fun (=) *pcr-float* (- *numeral* :: - \Rightarrow *real*) (- *numeral* :: - \Rightarrow *float*)
 $\langle proof \rangle$

lemma *float-of-numeral*: *numeral* *k* = *float-of* (*numeral* *k*)
and *float-of-neg-numeral*: - *numeral* *k* = *float-of* (- *numeral* *k*)
 $\langle proof \rangle$

43.3 Quickcheck

instantiation *float* :: *exhaustive*
begin

definition *exhaustive-float* **where**
exhaustive-float *f* *d* =
Quickcheck-Exhaustive.exhaustive ($\lambda x.$ *Quickcheck-Exhaustive.exhaustive* ($\lambda y.$ *f*
(*Float* *x* *y*)) *d*) *d*

instance $\langle proof \rangle$

end

context
includes *term-syntax*
begin

definition [*code-unfold*]:
valtermify-float *x* *y* = *Code-Evaluation.valtermify* *Float* {·} *x* {·} *y*

end

instantiation *float* :: *full-exhaustive*
begin

definition

full-exhaustive-float $f\ d =$
Quickcheck-Exhaustive.full-exhaustive
 $(\lambda x. \text{Quickcheck-Exhaustive.full-exhaustive } (\lambda y. f \text{ (valtermify-float } x\ y))\ d)\ d$

instance $\langle \text{proof} \rangle$

end

instantiation *float* :: *random*
begin

definition *Quickcheck-Random.random* $i =$

scomp (*Quickcheck-Random.random* $(2^{\wedge} \text{nat-of-natural } i)$)
 $(\lambda \text{man. scomp } (\text{Quickcheck-Random.random } i) (\lambda \text{exp. Pair } (\text{valtermify-float } \text{man } \text{exp})))$

instance $\langle \text{proof} \rangle$

end

43.4 Represent floats as unique mantissa and exponent

lemma *int-induct-abs[case-names less]*:

fixes $j :: \text{int}$
assumes $H: \bigwedge n. (\bigwedge i. |i| < |n| \implies P\ i) \implies P\ n$
shows $P\ j$
 $\langle \text{proof} \rangle$

lemma *int-cancel-factors*:

fixes $n :: \text{int}$
assumes $1 < r$
shows $n = 0 \vee (\exists k\ i. n = k * r^{\wedge} i \wedge \neg r\ \text{dvd } k)$
 $\langle \text{proof} \rangle$

lemma *mult-powr-eq-mult-powr-iff-asym*:

fixes $m1\ m2\ e1\ e2 :: \text{int}$
assumes $m1: \neg 2\ \text{dvd } m1$
and $e1 \leq e2$
shows $m1 * 2^{\text{powr } e1} = m2 * 2^{\text{powr } e2} \longleftrightarrow m1 = m2 \wedge e1 = e2$
 $(\text{is } ?lhs \longleftrightarrow ?rhs)$
 $\langle \text{proof} \rangle$

lemma *mult-powr-eq-mult-powr-iff*:

$\neg 2\ \text{dvd } m1 \implies \neg 2\ \text{dvd } m2 \implies m1 * 2^{\text{powr } e1} = m2 * 2^{\text{powr } e2} \longleftrightarrow m1 = m2 \wedge e1 = e2$
for $m1\ m2\ e1\ e2 :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *floatE-normed*:

assumes $x: x \in \text{float}$

obtains $(\text{zero})\ x = 0$

| $(\text{powr})\ m\ e :: \text{int}$ **where** $x = m * 2^{\text{powr } e} \wedge 2 \nmid m \wedge x \neq 0$
 $\langle \text{proof} \rangle$

lemma *float-normed-cases*:

fixes $f :: \text{float}$

obtains $(\text{zero})\ f = 0$

| $(\text{powr})\ m\ e :: \text{int}$ **where** $\text{real-of-float } f = m * 2^{\text{powr } e} \wedge 2 \nmid m \wedge f \neq 0$
 $\langle \text{proof} \rangle$

definition *mantissa* $:: \text{float} \Rightarrow \text{int}$

where *mantissa* $f =$

$\text{fst } (\text{SOME } p :: \text{int} \times \text{int}. (f = 0 \wedge \text{fst } p = 0 \wedge \text{snd } p = 0)) \vee$

$(f \neq 0 \wedge \text{real-of-float } f = \text{real-of-int } (\text{fst } p) * 2^{\text{powr } \text{real-of-int } (\text{snd } p)} \wedge 2 \nmid \text{fst } p))$

definition *exponent* $:: \text{float} \Rightarrow \text{int}$

where *exponent* $f =$

$\text{snd } (\text{SOME } p :: \text{int} \times \text{int}. (f = 0 \wedge \text{fst } p = 0 \wedge \text{snd } p = 0)) \vee$

$(f \neq 0 \wedge \text{real-of-float } f = \text{real-of-int } (\text{fst } p) * 2^{\text{powr } \text{real-of-int } (\text{snd } p)} \wedge 2 \nmid \text{fst } p))$

lemma *exponent-0[simp]*: $\text{exponent } 0 = 0$ (**is** ?E)

and *mantissa-0[simp]*: $\text{mantissa } 0 = 0$ (**is** ?M)

$\langle \text{proof} \rangle$

lemma *mantissa-exponent*: $\text{real-of-float } f = \text{mantissa } f * 2^{\text{powr } \text{exponent } f}$ (**is** ?E)

and *mantissa-not-dvd*: $f \neq 0 \implies 2 \nmid \text{mantissa } f$ (**is** - \implies ?D)

$\langle \text{proof} \rangle$

lemma *mantissa-noteq-0*: $f \neq 0 \implies \text{mantissa } f \neq 0$

$\langle \text{proof} \rangle$

lemma *mantissa-eq-zero-iff*: $\text{mantissa } x = 0 \longleftrightarrow x = 0$

(**is** ?lhs \longleftrightarrow ?rhs)

$\langle \text{proof} \rangle$

lemma *mantissa-pos-iff*: $0 < \text{mantissa } x \longleftrightarrow 0 < x$

$\langle \text{proof} \rangle$

lemma *mantissa-nonneg-iff*: $0 \leq \text{mantissa } x \longleftrightarrow 0 \leq x$

$\langle \text{proof} \rangle$

lemma *mantissa-neg-iff*: $0 > \text{mantissa } x \longleftrightarrow 0 > x$

$\langle \text{proof} \rangle$

```

lemma
  fixes  $m\ e :: \text{int}$ 
  defines  $f \equiv \text{float-of } (m * 2^{\text{powr } e})$ 
  assumes  $\text{dvd}: \neg 2 \text{ dvd } m$ 
  shows  $\text{mantissa-float}: \text{mantissa } f = m$  (is ?M)
  and  $\text{exponent-float}: m \neq 0 \implies \text{exponent } f = e$  (is -  $\implies$  ?E)
  <proof>

```

43.5 Compute arithmetic operations

```

lemma  $\text{Float-mantissa-exponent}: \text{Float } (\text{mantissa } f) (\text{exponent } f) = f$ 
  <proof>

```

```

lemma  $\text{Float-cases } [\text{cases type: float}]:$ 
  fixes  $f :: \text{float}$ 
  obtains  $(\text{Float})\ m\ e :: \text{int}$  where  $f = \text{Float } m\ e$ 
  <proof>

```

```

lemma  $\text{denormalize-shift}:$ 
  assumes  $f\text{-def}: f = \text{Float } m\ e$ 
  and  $\text{not-0}: f \neq 0$ 
  obtains  $i$  where  $m = \text{mantissa } f * 2^i$   $e = \text{exponent } f - i$ 
  <proof>

```

```

context
begin

```

```

qualified lemma  $\text{compute-float-zero}[\text{code-unfold, code}]: 0 = \text{Float } 0\ 0$ 
  <proof> lemma  $\text{compute-float-one}[\text{code-unfold, code}]: 1 = \text{Float } 1\ 0$ 
  <proof>

```

```

lift-definition  $\text{normfloat} :: \text{float} \Rightarrow \text{float}$  is  $\lambda x. x$  <proof>
lemma  $\text{normfloat-id}[\text{simp}]: \text{normfloat } x = x$  <proof> lemma  $\text{compute-normfloat}[\text{code}]:$ 
   $\text{normfloat } (\text{Float } m\ e) =$ 
    (if  $m \bmod 2 = 0 \wedge m \neq 0$  then  $\text{normfloat } (\text{Float } (m \text{ div } 2) (e + 1))$ 
     else if  $m = 0$  then 0 else  $\text{Float } m\ e$ )
  <proof> lemma  $\text{compute-float-numeral}[\text{code-abbrev}]: \text{Float } (\text{numeral } k)\ 0 = \text{numeral } k$ 
  <proof> lemma  $\text{compute-float-neg-numeral}[\text{code-abbrev}]: \text{Float } (- \text{numeral } k)\ 0 = - \text{numeral } k$ 
  <proof> lemma  $\text{compute-float-uminus}[\text{code}]: - \text{Float } m1\ e1 = \text{Float } (- m1)\ e1$ 
  <proof> lemma  $\text{compute-float-times}[\text{code}]: \text{Float } m1\ e1 * \text{Float } m2\ e2 = \text{Float } (m1 * m2) (e1 + e2)$ 
  <proof> lemma  $\text{compute-float-plus}[\text{code}]:$ 
     $\text{Float } m1\ e1 + \text{Float } m2\ e2 =$ 
    (if  $m1 = 0$  then  $\text{Float } m2\ e2$ 
     else if  $m2 = 0$  then  $\text{Float } m1\ e1$ 
     else if  $e1 \leq e2$  then  $\text{Float } (m1 + m2 * 2^{\text{nat } (e2 - e1)})\ e1$ 

```



```

    else Float (m2 + m1 * 2nat (e1 - e2)) e2)
  <proof> lemma compute-float-minus[code]:  $f - g = f + (-g)$  for  $f\ g :: \text{float}$ 
  <proof> lemma compute-float-sgn[code]:
     $\text{sgn} (\text{Float } m1\ e1) = (\text{if } 0 < m1 \text{ then } 1 \text{ else if } m1 < 0 \text{ then } -1 \text{ else } 0)$ 
  <proof>

lift-definition is-float-pos ::  $\text{float} \Rightarrow \text{bool}$  is  $(<) 0 :: \text{real} \Rightarrow \text{bool}$  <proof> lemma
compute-is-float-pos[code]:  $\text{is-float-pos} (\text{Float } m\ e) \longleftrightarrow 0 < m$ 
  <proof>

lift-definition is-float-nonneg ::  $\text{float} \Rightarrow \text{bool}$  is  $(\leq) 0 :: \text{real} \Rightarrow \text{bool}$  <proof>
lemma compute-is-float-nonneg[code]:  $\text{is-float-nonneg} (\text{Float } m\ e) \longleftrightarrow 0 \leq m$ 
  <proof>

lift-definition is-float-zero ::  $\text{float} \Rightarrow \text{bool}$  is  $(=) 0 :: \text{real} \Rightarrow \text{bool}$  <proof> lemma
compute-is-float-zero[code]:  $\text{is-float-zero} (\text{Float } m\ e) \longleftrightarrow 0 = m$ 
  <proof> lemma compute-float-abs[code]:  $|\text{Float } m\ e| = \text{Float } |m|\ e$ 
  <proof> lemma compute-float-eq[code]:  $\text{equal-class.equal } f\ g = \text{is-float-zero } (f - g)$ 
  <proof>

end

```

43.6 Lemmas for types *real*, *nat*, *int*

```

lemmas real-of-ints =
  of-int-add
  of-int-minus
  of-int-diff
  of-int-mult
  of-int-power
  of-int-numeral of-int-neg-numeral

```

```

lemmas int-of-reals = real-of-ints[symmetric]

```

43.7 Rounding Real Numbers

```

definition round-down ::  $\text{int} \Rightarrow \text{real} \Rightarrow \text{real}$ 
  where round-down prec  $x = \lfloor x * 2^{\text{powr } \text{prec}} \rfloor * 2^{\text{powr } -\text{prec}}$ 

```

```

definition round-up ::  $\text{int} \Rightarrow \text{real} \Rightarrow \text{real}$ 
  where round-up prec  $x = \lceil x * 2^{\text{powr } \text{prec}} \rceil * 2^{\text{powr } -\text{prec}}$ 

```

```

lemma round-down-float[simp]:  $\text{round-down } \text{prec } x \in \text{float}$ 
  <proof>

```

```

lemma round-up-float[simp]:  $\text{round-up } \text{prec } x \in \text{float}$ 
  <proof>

```

```

lemma round-up:  $x \leq \text{round-up } \text{prec } x$ 

```

$\langle \text{proof} \rangle$

lemma *round-down*: $\text{round-down prec } x \leq x$
 $\langle \text{proof} \rangle$

lemma *round-up-0[simp]*: $\text{round-up } p \ 0 = 0$
 $\langle \text{proof} \rangle$

lemma *round-down-0[simp]*: $\text{round-down } p \ 0 = 0$
 $\langle \text{proof} \rangle$

lemma *round-up-diff-round-down*: $\text{round-up prec } x - \text{round-down prec } x \leq 2^{\text{powr}} - \text{prec}$
 $\langle \text{proof} \rangle$

lemma *round-down-shift*: $\text{round-down } p \ (x * 2^{\text{powr } k}) = 2^{\text{powr } k} * \text{round-down } (p + k) \ x$
 $\langle \text{proof} \rangle$

lemma *round-up-shift*: $\text{round-up } p \ (x * 2^{\text{powr } k}) = 2^{\text{powr } k} * \text{round-up } (p + k) \ x$
 $\langle \text{proof} \rangle$

lemma *round-up-uminus-eq*: $\text{round-up } p \ (-x) = - \text{round-down } p \ x$
and *round-down-uminus-eq*: $\text{round-down } p \ (-x) = - \text{round-up } p \ x$
 $\langle \text{proof} \rangle$

lemma *round-up-mono*: $x \leq y \implies \text{round-up } p \ x \leq \text{round-up } p \ y$
 $\langle \text{proof} \rangle$

lemma *round-up-le1*:
assumes $x \leq 1 \ \text{prec} \geq 0$
shows $\text{round-up prec } x \leq 1$
 $\langle \text{proof} \rangle$

lemma *round-up-less1*:
assumes $x < 1 \ / \ 2^p > 0$
shows $\text{round-up } p \ x < 1$
 $\langle \text{proof} \rangle$

lemma *round-down-ge1*:
assumes $x: x \geq 1$
assumes *prec*: $p \geq - \log 2 \ x$
shows $1 \leq \text{round-down } p \ x$
 $\langle \text{proof} \rangle$

lemma *round-up-le0*: $x \leq 0 \implies \text{round-up } p \ x \leq 0$
 $\langle \text{proof} \rangle$

43.8 Rounding Floats

definition $\text{div-twoPow} :: \text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$
where $[\text{simp}]$: $\text{div-twoPow } x \ n = x \text{ div } (2 \wedge n)$

definition $\text{mod-twoPow} :: \text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$
where $[\text{simp}]$: $\text{mod-twoPow } x \ n = x \text{ mod } (2 \wedge n)$

lemma $\text{compute-div-twoPow}[\text{code}]$:
 $\text{div-twoPow } x \ n = (\text{if } x = 0 \vee x = -1 \vee n = 0 \text{ then } x \text{ else } \text{div-twoPow } (x \text{ div } 2) (n - 1))$
 $\langle \text{proof} \rangle$

lemma $\text{compute-mod-twoPow}[\text{code}]$:
 $\text{mod-twoPow } x \ n = (\text{if } n = 0 \text{ then } 0 \text{ else } x \text{ mod } 2 + 2 * \text{mod-twoPow } (x \text{ div } 2) (n - 1))$
 $\langle \text{proof} \rangle$

lift-definition $\text{float-up} :: \text{int} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** round-up $\langle \text{proof} \rangle$
declare $\text{float-up.rep-eq}[\text{simp}]$

lemma round-up-correct : $\text{round-up } e \ f - f \in \{0..2^{\text{powr } -e}\}$
 $\langle \text{proof} \rangle$

lemma float-up-correct : $\text{real-of-float } (\text{float-up } e \ f) - \text{real-of-float } f \in \{0..2^{\text{powr } -e}\}$
 $\langle \text{proof} \rangle$

lift-definition $\text{float-down} :: \text{int} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** round-down $\langle \text{proof} \rangle$
declare $\text{float-down.rep-eq}[\text{simp}]$

lemma $\text{round-down-correct}$: $f - (\text{round-down } e \ f) \in \{0..2^{\text{powr } -e}\}$
 $\langle \text{proof} \rangle$

lemma $\text{float-down-correct}$: $\text{real-of-float } f - \text{real-of-float } (\text{float-down } e \ f) \in \{0..2^{\text{powr } -e}\}$
 $\langle \text{proof} \rangle$

context
begin

qualified lemma $\text{compute-float-down}[\text{code}]$:
 $\text{float-down } p \ (\text{Float } m \ e) =$
 $(\text{if } p + e < 0 \text{ then } \text{Float } (\text{div-twoPow } m \ (\text{nat } (-(p + e)))) (-p) \text{ else } \text{Float } m \ e)$
 $\langle \text{proof} \rangle$

lemma abs-round-down-le : $|f - (\text{round-down } e \ f)| \leq 2^{\text{powr } -e}$
 $\langle \text{proof} \rangle$

lemma abs-round-up-le : $|f - (\text{round-up } e \ f)| \leq 2^{\text{powr } -e}$

$\langle \text{proof} \rangle$

lemma *round-down-nonneg*: $0 \leq s \implies 0 \leq \text{round-down } p \ s$
 $\langle \text{proof} \rangle$

lemma *ceil-divide-floor-conv*:

assumes $b \neq 0$

shows $\lceil \text{real-of-int } a / \text{real-of-int } b \rceil =$

$(\text{if } b \text{ dvd } a \text{ then } a \text{ div } b \text{ else } \lfloor \text{real-of-int } a / \text{real-of-int } b \rfloor + 1)$

$\langle \text{proof} \rangle$ **lemma** *compute-float-up[code]*: $\text{float-up } p \ x = - \text{float-down } p \ (-x)$
 $\langle \text{proof} \rangle$

end

lemma *bitlen-Float*:

fixes $m \ e$

defines $[THEN \text{ meta-eq-to-obj-eq}]: f \equiv \text{Float } m \ e$

shows $\text{bitlen } |\text{mantissa } f| + \text{exponent } f = (\text{if } m = 0 \text{ then } 0 \text{ else } \text{bitlen } |m| + e)$
 $\langle \text{proof} \rangle$

lemma *float-gt1-scale*:

assumes $1 \leq \text{Float } m \ e$

shows $0 \leq e + (\text{bitlen } m - 1)$

$\langle \text{proof} \rangle$

43.9 Truncating Real Numbers

definition *truncate-down::nat \Rightarrow real \Rightarrow real*

where $\text{truncate-down } \text{prec } x = \text{round-down } (\text{prec} - \lfloor \log 2 |x| \rfloor) \ x$

lemma *truncate-down*: $\text{truncate-down } \text{prec } x \leq x$
 $\langle \text{proof} \rangle$

lemma *truncate-down-le*: $x \leq y \implies \text{truncate-down } \text{prec } x \leq y$
 $\langle \text{proof} \rangle$

lemma *truncate-down-zero[simp]*: $\text{truncate-down } \text{prec } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *truncate-down-float[simp]*: $\text{truncate-down } p \ x \in \text{float}$
 $\langle \text{proof} \rangle$

definition *truncate-up::nat \Rightarrow real \Rightarrow real*

where $\text{truncate-up } \text{prec } x = \text{round-up } (\text{prec} - \lfloor \log 2 |x| \rfloor) \ x$

lemma *truncate-up*: $x \leq \text{truncate-up } \text{prec } x$
 $\langle \text{proof} \rangle$

lemma *truncate-up-le*: $x \leq y \implies x \leq \text{truncate-up } \text{prec } y$
 ⟨proof⟩

lemma *truncate-up-zero[simp]*: $\text{truncate-up } \text{prec } 0 = 0$
 ⟨proof⟩

lemma *truncate-up-uminus-eq*: $\text{truncate-up } \text{prec } (-x) = - \text{truncate-down } \text{prec } x$
and *truncate-down-uminus-eq*: $\text{truncate-down } \text{prec } (-x) = - \text{truncate-up } \text{prec } x$
 ⟨proof⟩

lemma *truncate-up-float[simp]*: $\text{truncate-up } p \ x \in \text{float}$
 ⟨proof⟩

lemma *mult-powr-eq*: $0 < b \implies b \neq 1 \implies 0 < x \implies x * b^{\text{powr } y} = b^{\text{powr } (y + \log b \ x)}$
 ⟨proof⟩

lemma *truncate-down-pos*:
assumes $x > 0$
shows $\text{truncate-down } p \ x > 0$
 ⟨proof⟩

lemma *truncate-down-nonneg*: $0 \leq y \implies 0 \leq \text{truncate-down } \text{prec } y$
 ⟨proof⟩

lemma *truncate-down-ge1*: $1 \leq x \implies 1 \leq \text{truncate-down } p \ x$
 ⟨proof⟩

lemma *truncate-up-nonpos*: $x \leq 0 \implies \text{truncate-up } \text{prec } x \leq 0$
 ⟨proof⟩

lemma *truncate-up-le1*:
assumes $x \leq 1$
shows $\text{truncate-up } p \ x \leq 1$
 ⟨proof⟩

lemma *truncate-down-shift-int*:
 $\text{truncate-down } p \ (x * 2^{\text{powr } \text{real-of-int } k}) = \text{truncate-down } p \ x * 2^{\text{powr } k}$
 ⟨proof⟩

lemma *truncate-down-shift-nat*: $\text{truncate-down } p \ (x * 2^{\text{powr } \text{real } k}) = \text{truncate-down } p \ x * 2^{\text{powr } k}$
 ⟨proof⟩

lemma *truncate-up-shift-int*: $\text{truncate-up } p \ (x * 2^{\text{powr } \text{real-of-int } k}) = \text{truncate-up } p \ x * 2^{\text{powr } k}$
 ⟨proof⟩

lemma *truncate-up-shift-nat*: $\text{truncate-up } p \ (x * 2^{\text{powr } \text{real } k}) = \text{truncate-up } p \ x$

* 2 powr k
 ⟨proof⟩

43.10 Truncating Floats

lift-definition *float-round-up* :: nat \Rightarrow float \Rightarrow float **is** *truncate-up*
 ⟨proof⟩

lemma *float-round-up*: *real-of-float* $x \leq$ *real-of-float* (*float-round-up* prec x)
 ⟨proof⟩

lemma *float-round-up-zero[simp]*: *float-round-up* prec 0 = 0
 ⟨proof⟩

lift-definition *float-round-down* :: nat \Rightarrow float \Rightarrow float **is** *truncate-down*
 ⟨proof⟩

lemma *float-round-down*: *real-of-float* (*float-round-down* prec x) \leq *real-of-float* x
 ⟨proof⟩

lemma *float-round-down-zero[simp]*: *float-round-down* prec 0 = 0
 ⟨proof⟩

lemmas *float-round-up-le* = *order-trans*[*OF* - *float-round-up*]
 and *float-round-down-le* = *order-trans*[*OF* *float-round-down*]

lemma *minus-float-round-up-eq*: - *float-round-up* prec x = *float-round-down* prec
 (- x)
 and *minus-float-round-down-eq*: - *float-round-down* prec x = *float-round-up* prec
 (- x)
 ⟨proof⟩

context
begin

qualified lemma *compute-float-round-down*[code]:
float-round-down prec (*Float* m e) =
 (let $d = \text{bitlen } |m| - \text{int } \text{prec} - 1$ in
 if $0 < d$ then *Float* (*div-two*pow m (nat d)) ($e + d$)
 else *Float* m e)
 ⟨proof⟩ **lemma** *compute-float-round-up*[code]:
float-round-up prec x = - *float-round-down* prec (- x)
 ⟨proof⟩

end

lemma *truncate-up-nonneg-mono*:
 assumes $0 \leq x \leq y$
 shows *truncate-up* prec $x \leq$ *truncate-up* prec y

$\langle \text{proof} \rangle$

lemma *truncate-up-switch-sign-mono*:

assumes $x \leq 0$ $0 \leq y$

shows $\text{truncate-up prec } x \leq \text{truncate-up prec } y$

$\langle \text{proof} \rangle$

lemma *truncate-down-switch-sign-mono*:

assumes $x \leq 0$

and $0 \leq y$

and $x \leq y$

shows $\text{truncate-down prec } x \leq \text{truncate-down prec } y$

$\langle \text{proof} \rangle$

lemma *truncate-down-nonneg-mono*:

assumes $0 \leq x$ $x \leq y$

shows $\text{truncate-down prec } x \leq \text{truncate-down prec } y$

$\langle \text{proof} \rangle$

lemma *truncate-down-eq-truncate-up*: $\text{truncate-down } p \ x = - \text{truncate-up } p \ (-x)$

and *truncate-up-eq-truncate-down*: $\text{truncate-up } p \ x = - \text{truncate-down } p \ (-x)$

$\langle \text{proof} \rangle$

lemma *truncate-down-mono*: $x \leq y \implies \text{truncate-down } p \ x \leq \text{truncate-down } p \ y$

$\langle \text{proof} \rangle$

lemma *truncate-up-mono*: $x \leq y \implies \text{truncate-up } p \ x \leq \text{truncate-up } p \ y$

$\langle \text{proof} \rangle$

lemma *truncate-up-nonneg*: $0 \leq \text{truncate-up } p \ x$ **if** $0 \leq x$

$\langle \text{proof} \rangle$

lemma *truncate-up-pos*: $0 < \text{truncate-up } p \ x$ **if** $0 < x$

$\langle \text{proof} \rangle$

lemma *truncate-up-less-zero-iff[simp]*: $\text{truncate-up } p \ x < 0 \longleftrightarrow x < 0$

$\langle \text{proof} \rangle$

lemma *truncate-up-nonneg-iff[simp]*: $\text{truncate-up } p \ x \geq 0 \longleftrightarrow x \geq 0$

$\langle \text{proof} \rangle$

lemma *truncate-down-less-zero-iff[simp]*: $\text{truncate-down } p \ x < 0 \longleftrightarrow x < 0$

$\langle \text{proof} \rangle$

lemma *truncate-down-nonneg-iff[simp]*: $\text{truncate-down } p \ x \geq 0 \longleftrightarrow x \geq 0$

$\langle \text{proof} \rangle$

lemma *truncate-down-eq-zero-iff[simp]*: $\text{truncate-down prec } x = 0 \longleftrightarrow x = 0$

$\langle \text{proof} \rangle$

lemma *truncate-up-eq-zero-iff*[simp]: *truncate-up prec x = 0 \longleftrightarrow x = 0*
 ⟨proof⟩

43.11 Approximation of positive rationals

lemma *div-mult-twopow-eq*: *a div ((2::nat) ^ n) div b = a div (b * 2 ^ n)* **for** *a b :: nat*
 ⟨proof⟩

lemma *real-div-nat-eq-floor-of-divide*: *a div b = real-of-int $\lfloor a / b \rfloor$* **for** *a b :: nat*
 ⟨proof⟩

definition *rat-precision prec x y =*
(let d = bitlen x - bitlen y
in int prec - d + (if Float (abs x) 0 < Float (abs y) d then 1 else 0))

lemma *floor-log-divide-eq*:
assumes *i > 0 j > 0 p > 1*
shows $\lfloor \log p (i / j) \rfloor = \text{floor } (\log p i) - \text{floor } (\log p j) -$
*(if $i \geq j * p^{\text{floor } (\log p i) - \text{floor } (\log p j)}$ then 0 else 1)*
 ⟨proof⟩

lemma *truncate-down-rat-precision*:
truncate-down prec (real x / real y) = round-down (rat-precision prec x y) (real x / real y)
and *truncate-up-rat-precision*:
truncate-up prec (real x / real y) = round-up (rat-precision prec x y) (real x / real y)
 ⟨proof⟩

lift-definition *lapprox-posrat :: nat \Rightarrow nat \Rightarrow nat \Rightarrow float*
is $\lambda \text{prec } (x::\text{nat}) (y::\text{nat}). \text{truncate-down prec } (x / y)$
 ⟨proof⟩

context
begin

qualified lemma *compute-lapprox-posrat*[code]:
lapprox-posrat prec x y =
(let
l = rat-precision prec x y;
*d = if $0 \leq l$ then $x * 2^{\text{nat } l} \text{ div } y$ else $x \text{ div } 2^{\text{nat } (- l)} \text{ div } y$*
in normfloat (Float d (- l)))
 ⟨proof⟩

end

lift-definition *rapprox-posrat :: nat \Rightarrow nat \Rightarrow nat \Rightarrow float*

is $\lambda \text{prec } (x::\text{nat}) (y::\text{nat}). \text{truncate-up } \text{prec } (x / y)$
 $\langle \text{proof} \rangle$

context
begin

qualified lemma *compute-rapprox-posrat*[code]:
fixes $\text{prec } x \ y$
defines $l \equiv \text{rat-precision } \text{prec } x \ y$
shows $\text{rapprox-posrat } \text{prec } x \ y =$
 $(\text{let}$
 $\quad l = l;$
 $\quad (r, s) = \text{if } 0 \leq l \text{ then } (x * 2^{\text{nat } l}, y) \text{ else } (x, y * 2^{\text{nat } (-l)});$
 $\quad d = r \text{ div } s;$
 $\quad m = r \text{ mod } s$
 $\quad \text{in normfloat } (\text{Float } (d + (\text{if } m = 0 \vee y = 0 \text{ then } 0 \text{ else } 1)) (-l)))$
 $\langle \text{proof} \rangle$

end

lemma *rat-precision-pos*:
assumes $0 \leq x$
and $0 < y$
and $2 * x < y$
shows $\text{rat-precision } n \ (\text{int } x) \ (\text{int } y) > 0$
 $\langle \text{proof} \rangle$

lemma *rapprox-posrat-less1*:
 $0 \leq x \implies 0 < y \implies 2 * x < y \implies \text{real-of-float } (\text{rapprox-posrat } n \ x \ y) < 1$
 $\langle \text{proof} \rangle$

lift-definition *lapprox-rat* :: $\text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{float}$ **is**
 $\lambda \text{prec } (x::\text{int}) (y::\text{int}). \text{truncate-down } \text{prec } (x / y)$
 $\langle \text{proof} \rangle$

context
begin

qualified lemma *compute-lapprox-rat*[code]:
 $\text{lapprox-rat } \text{prec } x \ y =$
 $(\text{if } y = 0 \text{ then } 0$
 $\quad \text{else if } 0 \leq x \text{ then}$
 $\quad (\text{if } 0 < y \text{ then } \text{lapprox-posrat } \text{prec } (\text{nat } x) (\text{nat } y)$
 $\quad \quad \text{else } - (\text{rapprox-posrat } \text{prec } (\text{nat } x) (\text{nat } (-y))))$
 $\quad \text{else}$
 $\quad (\text{if } 0 < y$
 $\quad \quad \text{then } - (\text{rapprox-posrat } \text{prec } (\text{nat } (-x)) (\text{nat } y))$
 $\quad \quad \text{else } \text{lapprox-posrat } \text{prec } (\text{nat } (-x)) (\text{nat } (-y))))$
 $\langle \text{proof} \rangle$

lift-definition *rapprox-rat* :: *nat* \Rightarrow *int* \Rightarrow *int* \Rightarrow *float* **is**

$\lambda \text{prec } (x::\text{int}) \ (y::\text{int}). \text{truncate-up } \text{prec } (x / y)$
 $\langle \text{proof} \rangle$

lemma *rapprox-rat* = *rapprox-posrat*

$\langle \text{proof} \rangle$

lemma *lapprox-rat* = *lapprox-posrat*

$\langle \text{proof} \rangle$ **lemma** *compute-rapprox-rat*[code]:

rapprox-rat *prec* *x* *y* = $- \text{lapprox-rat } \text{prec } (-x) \ y$

$\langle \text{proof} \rangle$ **lemma** *compute-truncate-down*[code]:

truncate-down *p* (*Ratreal* *r*) = (let (*a*, *b*) = *quotient-of* *r* in *lapprox-rat* *p* *a* *b*)

$\langle \text{proof} \rangle$ **lemma** *compute-truncate-up*[code]:

truncate-up *p* (*Ratreal* *r*) = (let (*a*, *b*) = *quotient-of* *r* in *rapprox-rat* *p* *a* *b*)

$\langle \text{proof} \rangle$

end

43.12 Division

definition *real-divl* *prec* *a* *b* = *truncate-down* *prec* (*a* / *b*)

definition *real-divr* *prec* *a* *b* = *truncate-up* *prec* (*a* / *b*)

lift-definition *float-divl* :: *nat* \Rightarrow *float* \Rightarrow *float* \Rightarrow *float* **is** *real-divl*

$\langle \text{proof} \rangle$

context

begin

qualified lemma *compute-float-divl*[code]:

float-divl *prec* (*Float* *m1* *s1*) (*Float* *m2* *s2*) = *lapprox-rat* *prec* *m1* *m2* * *Float* 1
 (*s1* - *s2*)

$\langle \text{proof} \rangle$

lift-definition *float-divr* :: *nat* \Rightarrow *float* \Rightarrow *float* \Rightarrow *float* **is** *real-divr*

$\langle \text{proof} \rangle$ **lemma** *compute-float-divr*[code]:

float-divr *prec* *x* *y* = $- \text{float-divl } \text{prec } (-x) \ y$

$\langle \text{proof} \rangle$

end

43.13 Approximate Addition

definition *plus-down* *prec* *x* *y* = *truncate-down* *prec* (*x* + *y*)

definition *plus-up* *prec* *x* *y* = *truncate-up* *prec* (*x* + *y*)

lemma *float-plus-down-float*[*intro, simp*]: $x \in \text{float} \implies y \in \text{float} \implies \text{plus-down } p \ x \ y \in \text{float}$
 ⟨*proof*⟩

lemma *float-plus-up-float*[*intro, simp*]: $x \in \text{float} \implies y \in \text{float} \implies \text{plus-up } p \ x \ y \in \text{float}$
 ⟨*proof*⟩

lift-definition *float-plus-down* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *plus-down* ⟨*proof*⟩

lift-definition *float-plus-up* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *plus-up* ⟨*proof*⟩

lemma *plus-down*: $\text{plus-down } p \ x \ y \leq x + y$
and *plus-up*: $x + y \leq \text{plus-up } p \ x \ y$
 ⟨*proof*⟩

lemma *float-plus-down*: $\text{real-of-float } (\text{float-plus-down } p \ x \ y) \leq x + y$
and *float-plus-up*: $x + y \leq \text{real-of-float } (\text{float-plus-up } p \ x \ y)$
 ⟨*proof*⟩

lemmas *plus-down-le* = *order-trans*[*OF plus-down*]
and *plus-up-le* = *order-trans*[*OF - plus-up*]
and *float-plus-down-le* = *order-trans*[*OF float-plus-down*]
and *float-plus-up-le* = *order-trans*[*OF - float-plus-up*]

lemma *compute-plus-up*[*code*]: $\text{plus-up } p \ x \ y = - \text{plus-down } p \ (-x) \ (-y)$
 ⟨*proof*⟩

lemma *truncate-down-log2-eqI*:
assumes $\lfloor \log 2 \ |x| \rfloor = \lfloor \log 2 \ |y| \rfloor$
assumes $\lfloor x * 2^{\text{powr } (p - \lfloor \log 2 \ |x| \rfloor)} \rfloor = \lfloor y * 2^{\text{powr } (p - \lfloor \log 2 \ |x| \rfloor)} \rfloor$
shows $\text{truncate-down } p \ x = \text{truncate-down } p \ y$
 ⟨*proof*⟩

lemma *sum-neq-zeroI*:
 $|a| \geq k \implies |b| < k \implies a + b \neq 0$
 $|a| > k \implies |b| \leq k \implies a + b \neq 0$
for $a \ k :: \text{real}$
 ⟨*proof*⟩

lemma *abs-real-le-2-powr-bitlen*[*simp*]: $|\text{real-of-int } m2| < 2^{\text{powr } \text{real-of-int } (\text{bitlen } |m2|)}$
 ⟨*proof*⟩

lemma *floor-sum-times-2-powr-sgn-eq*:
fixes $a_i \ p \ q :: \text{int}$
and $a \ b :: \text{real}$
assumes $a * 2^{\text{powr } p} = a_i$
and $b \leq 1$: $|b * 2^{\text{powr } (p + 1)}| \leq 1$

and *leqp*: $q \leq p$
 shows $\lfloor (a + b) * 2^{\text{powr } q} \rfloor = \lfloor (2 * ai + \text{sgn } b) * 2^{\text{powr } (q - p - 1)} \rfloor$
 $\langle \text{proof} \rangle$

lemma *log2-abs-int-add-less-half-sgn-eq*:
 fixes *ai* :: *int*
 and *b* :: *real*
 assumes $|b| \leq 1/2$
 and $ai \neq 0$
 shows $\lfloor \log 2 \mid \text{real-of-int } ai + b \rfloor = \lfloor \log 2 \mid ai + \text{sgn } b / 2 \rfloor$
 $\langle \text{proof} \rangle$

context
begin

qualified lemma *compute-far-float-plus-down*:
 fixes *m1 e1 m2 e2* :: *int*
 and *p* :: *nat*
 defines $k1 \equiv \text{Suc } p - \text{nat } (\text{bitlen } |m1|)$
 assumes $H: \text{bitlen } |m2| \leq e1 - e2 - k1 - 2$ $m1 \neq 0$ $m2 \neq 0$ $e1 \geq e2$
 shows $\text{float-plus-down } p \text{ (Float } m1 \text{ } e1) \text{ (Float } m2 \text{ } e2) =$
 $\text{float-round-down } p \text{ (Float } (m1 * 2^{\wedge} (\text{Suc } (\text{Suc } k1)) + \text{sgn } m2) (e1 - \text{int } k1$
 $- 2))$
 $\langle \text{proof} \rangle$

lemma *compute-float-plus-down-naive*: $\text{float-plus-down } p \text{ } x \text{ } y = \text{float-round-down } p \text{ } (x + y)$
 $\langle \text{proof} \rangle$ **lemma** *compute-float-plus-down[code]*:
 fixes *p::nat* and *m1 e1 m2 e2::int*
 shows $\text{float-plus-down } p \text{ (Float } m1 \text{ } e1) \text{ (Float } m2 \text{ } e2) =$
 $(\text{if } m1 = 0 \text{ then } \text{float-round-down } p \text{ (Float } m2 \text{ } e2)$
 $\text{else if } m2 = 0 \text{ then } \text{float-round-down } p \text{ (Float } m1 \text{ } e1)$
 else
 $(\text{if } e1 \geq e2 \text{ then}$
 $(\text{let } k1 = \text{Suc } p - \text{nat } (\text{bitlen } |m1|) \text{ in}$
 $\text{if } \text{bitlen } |m2| > e1 - e2 - k1 - 2$
 $\text{then } \text{float-round-down } p \text{ ((Float } m1 \text{ } e1) + (\text{Float } m2 \text{ } e2))$
 $\text{else } \text{float-round-down } p \text{ (Float } (m1 * 2^{\wedge} (\text{Suc } (\text{Suc } k1)) + \text{sgn } m2) (e1$
 $- \text{int } k1 - 2)))$
 $\text{else } \text{float-plus-down } p \text{ (Float } m2 \text{ } e2) \text{ (Float } m1 \text{ } e1)))$
 $\langle \text{proof} \rangle$ **lemma** *compute-float-plus-up[code]*: $\text{float-plus-up } p \text{ } x \text{ } y = - \text{float-plus-down } p \text{ } (-x) \text{ } (-y)$
 $\langle \text{proof} \rangle$

lemma *mantissa-zero*: $\text{mantissa } 0 = 0$
 $\langle \text{proof} \rangle$ **lemma** *compute-float-less[code]*: $a < b \longleftrightarrow \text{is-float-pos } (\text{float-plus-down } 0 \text{ } b \text{ } (-a))$
 $\langle \text{proof} \rangle$ **lemma** *compute-float-le[code]*: $a \leq b \longleftrightarrow \text{is-float-nonneg } (\text{float-plus-down } 0 \text{ } b \text{ } (-a))$

$\langle \text{proof} \rangle$

end

lemma *plus-down-mono*: $\text{plus-down } p \ a \ b \leq \text{plus-down } p \ c \ d$ **if** $a + b \leq c + d$
 $\langle \text{proof} \rangle$

lemma *plus-up-mono*: $\text{plus-up } p \ a \ b \leq \text{plus-up } p \ c \ d$ **if** $a + b \leq c + d$
 $\langle \text{proof} \rangle$

43.14 Approximate Multiplication

lemma *mult-mono-nonpos-nonneg*: $a * b \leq c * d$
if $a \leq c \ a \leq 0 \ 0 \leq d \ d \leq b$ **for** $a \ b \ c \ d :: 'a :: \text{ordered-ring}$
 $\langle \text{proof} \rangle$

lemma *mult-mono-nonneg-nonpos*: $b * a \leq d * c$
if $a \leq c \ c \leq 0 \ 0 \leq d \ d \leq b$ **for** $a \ b \ c \ d :: 'a :: \text{ordered-ring}$
 $\langle \text{proof} \rangle$

lemma *mult-mono-nonpos-nonpos*: $a * b \leq c * d$
if $a \geq c \ a \leq 0 \ b \geq d \ d \leq 0$ **for** $a \ b \ c \ d :: \text{real}$
 $\langle \text{proof} \rangle$

lemma *mult-float-mono1*:
shows $a \leq b \implies ab \leq bb \implies$
 $aa \leq a \implies$
 $b \leq ba \implies$
 $ac \leq ab \implies$
 $bb \leq bc \implies$
 $\text{plus-down prec } (\text{nprt } aa * \text{pprt } bc)$
 $(\text{plus-down prec } (\text{nprt } ba * \text{nprt } bc))$
 $(\text{plus-down prec } (\text{pprt } aa * \text{pprt } ac))$
 $(\text{pprt } ba * \text{nprt } ac)))$
 $\leq \text{plus-down prec } (\text{nprt } a * \text{pprt } bb)$
 $(\text{plus-down prec } (\text{nprt } b * \text{nprt } bb))$
 $(\text{plus-down prec } (\text{pprt } a * \text{pprt } ab))$
 $(\text{pprt } b * \text{nprt } ab)))$
 $\langle \text{proof} \rangle$

lemma *mult-float-mono2*:
shows $a \leq b \implies$
 $ab \leq bb \implies$
 $aa \leq a \implies$
 $b \leq ba \implies$
 $ac \leq ab \implies$
 $bb \leq bc \implies$
 $\text{plus-up prec } (\text{pprt } b * \text{pprt } bb)$
 $(\text{plus-up prec } (\text{pprt } a * \text{nprt } bb))$

$$\begin{aligned}
& (plus-up\ prec\ (np\!rt\ b * pp\!rt\ ab) \\
& \quad (np\!rt\ a * np\!rt\ ab))) \\
\leq & plus-up\ prec\ (pp\!rt\ ba * pp\!rt\ bc) \\
& (plus-up\ prec\ (pp\!rt\ aa * np\!rt\ bc) \\
& \quad (plus-up\ prec\ (np\!rt\ ba * pp\!rt\ ac) \\
& \quad \quad (np\!rt\ aa * np\!rt\ ac))) \\
\langle proof \rangle
\end{aligned}$$

43.15 Approximate Power

lemma *div2-less-self*[*termination-simp*]: $odd\ n \implies n\ div\ 2 < n$ **for** $n :: nat$
 $\langle proof \rangle$

fun *power-down* :: $nat \Rightarrow real \Rightarrow nat \Rightarrow real$
where

$$\begin{aligned}
& power-down\ p\ x\ 0 = 1 \\
| & power-down\ p\ x\ (Suc\ n) = \\
& \quad (if\ odd\ n\ then\ truncate-down\ (Suc\ p)\ ((power-down\ p\ x\ (Suc\ n\ div\ 2))^2) \\
& \quad \quad else\ truncate-down\ (Suc\ p)\ (x * power-down\ p\ x\ n))
\end{aligned}$$

fun *power-up* :: $nat \Rightarrow real \Rightarrow nat \Rightarrow real$
where

$$\begin{aligned}
& power-up\ p\ x\ 0 = 1 \\
| & power-up\ p\ x\ (Suc\ n) = \\
& \quad (if\ odd\ n\ then\ truncate-up\ p\ ((power-up\ p\ x\ (Suc\ n\ div\ 2))^2) \\
& \quad \quad else\ truncate-up\ p\ (x * power-up\ p\ x\ n))
\end{aligned}$$

lift-definition *power-up-fl* :: $nat \Rightarrow float \Rightarrow nat \Rightarrow float$ **is** *power-up*
 $\langle proof \rangle$

lift-definition *power-down-fl* :: $nat \Rightarrow float \Rightarrow nat \Rightarrow float$ **is** *power-down*
 $\langle proof \rangle$

lemma *power-float-transfer*[*transfer-rule*]:
 $(rel_fun\ pcr_float\ (rel_fun\ (=)\ pcr_float))\ (\preceq)\ (\preceq)$
 $\langle proof \rangle$

lemma *compute-power-up-fl*[*code*]:

$$\begin{aligned}
& power-up-fl\ p\ x\ 0 = 1 \\
& power-up-fl\ p\ x\ (Suc\ n) = \\
& \quad (if\ odd\ n\ then\ float-round-up\ p\ ((power-up-fl\ p\ x\ (Suc\ n\ div\ 2))^2) \\
& \quad \quad else\ float-round-up\ p\ (x * power-up-fl\ p\ x\ n))
\end{aligned}$$

and *compute-power-down-fl*[*code*]:

$$\begin{aligned}
& power-down-fl\ p\ x\ 0 = 1 \\
& power-down-fl\ p\ x\ (Suc\ n) = \\
& \quad (if\ odd\ n\ then\ float-round-down\ (Suc\ p)\ ((power-down-fl\ p\ x\ (Suc\ n\ div\ 2))^2) \\
& \quad \quad else\ float-round-down\ (Suc\ p)\ (x * power-down-fl\ p\ x\ n))
\end{aligned}$$

$\langle proof \rangle$

lemma *power-down-pos*: $0 < x \implies 0 < \text{power-down } p \ x \ n$
 ⟨proof⟩

lemma *power-down-nonneg*: $0 \leq x \implies 0 \leq \text{power-down } p \ x \ n$
 ⟨proof⟩

lemma *power-down*: $0 \leq x \implies \text{power-down } p \ x \ n \leq x \wedge n$
 ⟨proof⟩

lemma *power-up*: $0 \leq x \implies x \wedge n \leq \text{power-up } p \ x \ n$
 ⟨proof⟩

lemmas *power-up-le* = *order-trans*[*OF* - *power-up*]
 and *power-up-less* = *less-le-trans*[*OF* - *power-up*]
 and *power-down-le* = *order-trans*[*OF* *power-down*]

lemma *power-down-fl*: $0 \leq x \implies \text{power-down-fl } p \ x \ n \leq x \wedge n$
 ⟨proof⟩

lemma *power-up-fl*: $0 \leq x \implies x \wedge n \leq \text{power-up-fl } p \ x \ n$
 ⟨proof⟩

lemma *real-power-up-fl*: *real-of-float* (*power-up-fl* *p* *x* *n*) = *power-up* *p* *x* *n*
 ⟨proof⟩

lemma *real-power-down-fl*: *real-of-float* (*power-down-fl* *p* *x* *n*) = *power-down* *p* *x* *n*
 ⟨proof⟩

lemmas [*simp del*] = *power-down.simps*(2) *power-up.simps*(2)

lemmas *power-down-simp* = *power-down.simps*(2)

lemmas *power-up-simp* = *power-up.simps*(2)

lemma *power-down-even-nonneg*: *even* *n* $\implies 0 \leq \text{power-down } p \ x \ n$
 ⟨proof⟩

lemma *power-down-eq-zero-iff*[*simp*]: *power-down* *prec* *b* *n* = 0 $\longleftrightarrow b = 0 \wedge n \neq 0$
 ⟨proof⟩

lemma *power-down-nonneg-iff*[*simp*]:
power-down *prec* *b* *n* $\geq 0 \longleftrightarrow \text{even } n \vee b \geq 0$
 ⟨proof⟩

lemma *power-down-neg-iff*[*simp*]:
power-down *prec* *b* *n* $< 0 \longleftrightarrow$
 $b < 0 \wedge \text{odd } n$
 ⟨proof⟩

lemma *power-down-nonpos-iff*[simp]:

notes [simp del] = *power-down-neg-iff power-down-eq-zero-iff*

shows $\text{power-down prec } b \ n \leq 0 \longleftrightarrow b < 0 \wedge \text{odd } n \vee b = 0 \wedge n \neq 0$
 <proof>

lemma *power-down-mono*:

power-down prec a n ≤ power-down prec b n

if $((0 \leq a \wedge a \leq b) \vee (\text{odd } n \wedge a \leq b) \vee (\text{even } n \wedge a \leq 0 \wedge b \leq a))$
 <proof>

lemma *power-up-even-nonneg*: $\text{even } n \implies 0 \leq \text{power-up } p \ x \ n$

<proof>

lemma *power-up-eq-zero-iff*[simp]: $\text{power-up prec } b \ n = 0 \longleftrightarrow b = 0 \wedge n \neq 0$

<proof>

lemma *power-up-nonneg-iff*[simp]:

power-up prec b n ≥ 0 ↔ even n ∨ b ≥ 0

<proof>

lemma *power-up-neg-iff*[simp]:

power-up prec b n < 0 ↔ b < 0 ∧ odd n

<proof>

lemma *power-up-nonpos-iff*[simp]:

notes [simp del] = *power-up-neg-iff power-up-eq-zero-iff*

shows $\text{power-up prec } b \ n \leq 0 \longleftrightarrow b < 0 \wedge \text{odd } n \vee b = 0 \wedge n \neq 0$
 <proof>

lemma *power-up-mono*:

power-up prec a n ≤ power-up prec b n

if $((0 \leq a \wedge a \leq b) \vee (\text{odd } n \wedge a \leq b) \vee (\text{even } n \wedge a \leq 0 \wedge b \leq a))$
 <proof>

43.16 Lemmas needed by Approximate

lemma *Float-num*[simp]:

real-of-float (Float 1 0) = 1

real-of-float (Float 1 1) = 2

real-of-float (Float 1 2) = 4

real-of-float (Float 1 (− 1)) = 1/2

real-of-float (Float 1 (− 2)) = 1/4

real-of-float (Float 1 (− 3)) = 1/8

real-of-float (Float (− 1) 0) = −1

real-of-float (Float (numeral n) 0) = numeral n

real-of-float (Float (− numeral n) 0) = − numeral n

<proof>

lemma *real-of-Float-int[simp]*: *real-of-float (Float n 0) = real n*
 ⟨*proof*⟩

lemma *float-zero[simp]*: *real-of-float (Float 0 e) = 0*
 ⟨*proof*⟩

lemma *abs-div-2-less*: $a \neq 0 \implies a \neq -1 \implies |(a::int) \text{ div } 2| < |a|$
 ⟨*proof*⟩

lemma *lapprox-rat*: *real-of-float (lapprox-rat prec x y) ≤ real-of-int x / real-of-int y*
 ⟨*proof*⟩

lemma *mult-div-le*:
 fixes $a\ b :: int$
 assumes $b > 0$
 shows $a \geq b * (a \text{ div } b)$
 ⟨*proof*⟩

lemma *lapprox-rat-nonneg*:
 assumes $0 \leq x$ and $0 \leq y$
 shows $0 \leq \text{real-of-float (lapprox-rat } n\ x\ y)$
 ⟨*proof*⟩

lemma *rapprox-rat*: *real-of-int x / real-of-int y ≤ real-of-float (rapprox-rat prec x y)*
 ⟨*proof*⟩

lemma *rapprox-rat-le1*:
 assumes $0 \leq x$ and $0 < y$ and $x \leq y$
 shows $\text{real-of-float (rapprox-rat } n\ x\ y) \leq 1$
 ⟨*proof*⟩

lemma *rapprox-rat-nonneg-nonpos*: $0 \leq x \implies y \leq 0 \implies \text{real-of-float (rapprox-rat } n\ x\ y) \leq 0$
 ⟨*proof*⟩

lemma *rapprox-rat-nonpos-nonneg*: $x \leq 0 \implies 0 \leq y \implies \text{real-of-float (rapprox-rat } n\ x\ y) \leq 0$
 ⟨*proof*⟩

lemma *real-divl*: *real-divl prec x y ≤ x / y*
 ⟨*proof*⟩

lemma *real-divr*: *x / y ≤ real-divr prec x y*
 ⟨*proof*⟩

lemma *float-divl*: *real-of-float (float-divl prec x y) ≤ x / y*
 ⟨*proof*⟩

lemma *real-divl-lower-bound*: $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-divl } \text{prec } x \ y$
 ⟨proof⟩

lemma *float-divl-lower-bound*: $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-of-float } (\text{float-divl } \text{prec } x \ y)$
 ⟨proof⟩

lemma *exponent-1*: $\text{exponent } 1 = 0$
 ⟨proof⟩

lemma *mantissa-1*: $\text{mantissa } 1 = 1$
 ⟨proof⟩

lemma *bitlen-1*: $\text{bitlen } 1 = 1$
 ⟨proof⟩

lemma *float-upper-bound*: $x \leq 2^{\text{powr } (\text{bitlen } |\text{mantissa } x| + \text{exponent } x)}$
 ⟨proof⟩

lemma *real-divl-pos-less1-bound*:
 assumes $0 < x \ x \leq 1$
 shows $1 \leq \text{real-divl } \text{prec } 1 \ x$
 ⟨proof⟩

lemma *float-divl-pos-less1-bound*:
 $0 < \text{real-of-float } x \implies \text{real-of-float } x \leq 1 \implies \text{prec} \geq 1 \implies$
 $1 \leq \text{real-of-float } (\text{float-divl } \text{prec } 1 \ x)$
 ⟨proof⟩

lemma *float-divr*: $\text{real-of-float } x / \text{real-of-float } y \leq \text{real-of-float } (\text{float-divr } \text{prec } x \ y)$
 ⟨proof⟩

lemma *real-divr-pos-less1-lower-bound*:
 assumes $0 < x$
 and $x \leq 1$
 shows $1 \leq \text{real-divr } \text{prec } 1 \ x$
 ⟨proof⟩

lemma *float-divr-pos-less1-lower-bound*: $0 < x \implies x \leq 1 \implies 1 \leq \text{float-divr } \text{prec } 1 \ x$
 ⟨proof⟩

lemma *real-divr-nonpos-pos-upper-bound*: $x \leq 0 \implies 0 \leq y \implies \text{real-divr } \text{prec } x \ y \leq 0$
 ⟨proof⟩

lemma *float-divr-nonpos-pos-upper-bound*:

real-of-float $x \leq 0 \implies 0 \leq \text{real-of-float } y \implies \text{real-of-float } (\text{float-divr prec } x \ y) \leq 0$
 ⟨proof⟩

lemma *real-divr-nonneg-neg-upper-bound*: $0 \leq x \implies y \leq 0 \implies \text{real-divr prec } x \ y \leq 0$
 ⟨proof⟩

lemma *float-divr-nonneg-neg-upper-bound*:
 $0 \leq \text{real-of-float } x \implies \text{real-of-float } y \leq 0 \implies \text{real-of-float } (\text{float-divr prec } x \ y) \leq 0$
 ⟨proof⟩

lemma *Float-le-zero-iff*: $\text{Float } a \ b \leq 0 \longleftrightarrow a \leq 0$
 ⟨proof⟩

lemma *real-of-float-pprt[simp]*:
fixes $a :: \text{float}$
shows $\text{real-of-float } (\text{pprt } a) = \text{pprt } (\text{real-of-float } a)$
 ⟨proof⟩

lemma *real-of-float-nprt[simp]*:
fixes $a :: \text{float}$
shows $\text{real-of-float } (\text{nprt } a) = \text{nprt } (\text{real-of-float } a)$
 ⟨proof⟩

context
begin

lift-definition *int-floor-fl* :: $\text{float} \Rightarrow \text{int}$ **is** *floor* ⟨proof⟩ **lemma** *compute-int-floor-fl[code]*:
 $\text{int-floor-fl } (\text{Float } m \ e) = (\text{if } 0 \leq e \text{ then } m * 2^{\text{nat } e} \text{ else } m \text{ div } (2^{\text{nat } (-e)}))$
 ⟨proof⟩

lift-definition *floor-fl* :: $\text{float} \Rightarrow \text{float}$ **is** $\lambda x. \text{real-of-int } \lfloor x \rfloor$
 ⟨proof⟩ **lemma** *compute-floor-fl[code]*:
 $\text{floor-fl } (\text{Float } m \ e) = (\text{if } 0 \leq e \text{ then } \text{Float } m \ e \text{ else } \text{Float } (m \text{ div } (2^{\text{nat } (-e)})))$
 0)
 ⟨proof⟩

end

lemma *floor-fl*: $\text{real-of-float } (\text{floor-fl } x) \leq \text{real-of-float } x$
 ⟨proof⟩

lemma *int-floor-fl*: $\text{real-of-int } (\text{int-floor-fl } x) \leq \text{real-of-float } x$
 ⟨proof⟩

lemma *floor-pos-exp*: $\text{exponent } (\text{floor-fl } x) \geq 0$

$\langle proof \rangle$

lemma *compute-mantissa*[code]:

mantissa (Float *m e*) =
 (if *m* = 0 then 0 else if 2 dvd *m* then *mantissa* (normfloat (Float *m e*)) else *m*)
 $\langle proof \rangle$

lemma *compute-exponent*[code]:

exponent (Float *m e*) =
 (if *m* = 0 then 0 else if 2 dvd *m* then *exponent* (normfloat (Float *m e*)) else *e*)
 $\langle proof \rangle$

lifting-update *Float.float.lifting*

lifting-forget *Float.float.lifting*

end

44 Pointwise instantiation of functions to algebra type classes

theory *Function-Algebras*

imports *Main*

begin

Pointwise operations

instantiation *fun* :: (*type*, *plus*) *plus*
begin

definition $f + g = (\lambda x. f\ x + g\ x)$

instance $\langle proof \rangle$

end

lemma *plus-fun-apply* [simp]:

$(f + g)\ x = f\ x + g\ x$
 $\langle proof \rangle$

instantiation *fun* :: (*type*, *zero*) *zero*
begin

definition $0 = (\lambda x. 0)$

instance $\langle proof \rangle$

end

lemma *zero-fun-apply* [simp]:

$0\ x = 0$
 $\langle proof \rangle$

instantiation *fun* :: (*type*, *times*) *times*
begin

definition $f * g = (\lambda x. f\ x * g\ x)$
instance $\langle proof \rangle$

end

lemma *times-fun-apply* [*simp*]:
 $(f * g)\ x = f\ x * g\ x$
 $\langle proof \rangle$

instantiation *fun* :: (*type*, *one*) *one*
begin

definition $1 = (\lambda x. 1)$
instance $\langle proof \rangle$

end

lemma *one-fun-apply* [*simp*]:
 $1\ x = 1$
 $\langle proof \rangle$

Additive structures

instance *fun* :: (*type*, *semigroup-add*) *semigroup-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *cancel-semigroup-add*) *cancel-semigroup-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *ab-semigroup-add*) *ab-semigroup-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *monoid-add*) *monoid-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add* $\langle proof \rangle$

instance *fun* :: (*type*, *group-add*) *group-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *ab-group-add*) *ab-group-add*
 ⟨*proof*⟩

Multiplicative structures

instance *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*
 ⟨*proof*⟩

instance *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*
 ⟨*proof*⟩

instance *fun* :: (*type*, *monoid-mult*) *monoid-mult*
 ⟨*proof*⟩

instance *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*
 ⟨*proof*⟩

Misc

instance *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* ⟨*proof*⟩

instance *fun* :: (*type*, *mult-zero*) *mult-zero*
 ⟨*proof*⟩

instance *fun* :: (*type*, *zero-neq-one*) *zero-neq-one*
 ⟨*proof*⟩

Ring structures

instance *fun* :: (*type*, *semiring*) *semiring*
 ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring*) *comm-semiring*
 ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0*) *semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0*) *comm-semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0-cancel*) *semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-1*) *semiring-1* ⟨*proof*⟩

lemma *numeral-fun*:
 ⟨*numeral* *n* = (λ*x*::'a. numeral *n*)⟩
 ⟨*proof*⟩

lemma *numeral-fun-apply* [*simp*]:
 ⟨*numeral* *n* *x* = numeral *n*⟩
 ⟨*proof*⟩

lemma *of-nat-fun*: $of\text{-}nat\ n = (\lambda x::'a. of\text{-}nat\ n)$
 $\langle proof \rangle$

lemma *of-nat-fun-apply* [*simp*]:
 $of\text{-}nat\ n\ x = of\text{-}nat\ n$
 $\langle proof \rangle$

instance *fun* :: (*type*, *comm-semiring-1*) *comm-semiring-1* $\langle proof \rangle$

instance *fun* :: (*type*, *semiring-1-cancel*) *semiring-1-cancel* $\langle proof \rangle$

instance *fun* :: (*type*, *comm-semiring-1-cancel*) *comm-semiring-1-cancel*
 $\langle proof \rangle$

instance *fun* :: (*type*, *semiring-char-0*) *semiring-char-0*
 $\langle proof \rangle$

instance *fun* :: (*type*, *ring*) *ring* $\langle proof \rangle$

instance *fun* :: (*type*, *comm-ring*) *comm-ring* $\langle proof \rangle$

instance *fun* :: (*type*, *ring-1*) *ring-1* $\langle proof \rangle$

instance *fun* :: (*type*, *comm-ring-1*) *comm-ring-1* $\langle proof \rangle$

instance *fun* :: (*type*, *ring-char-0*) *ring-char-0* $\langle proof \rangle$

Ordered structures

instance *fun* :: (*type*, *ordered-ab-semigroup-add*) *ordered-ab-semigroup-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *ordered-cancel-ab-semigroup-add*) *ordered-cancel-ab-semigroup-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *ordered-ab-semigroup-add-imp-le*) *ordered-ab-semigroup-add-imp-le*
 $\langle proof \rangle$

instance *fun* :: (*type*, *ordered-comm-monoid-add*) *ordered-comm-monoid-add* $\langle proof \rangle$

instance *fun* :: (*type*, *ordered-cancel-comm-monoid-add*) *ordered-cancel-comm-monoid-add*
 $\langle proof \rangle$

instance *fun* :: (*type*, *ordered-ab-group-add*) *ordered-ab-group-add* $\langle proof \rangle$

instance *fun* :: (*type*, *ordered-semiring*) *ordered-semiring*
 $\langle proof \rangle$

instance *fun* :: (*type*, *diod*) *diod*

<proof>

instance *fun* :: (*type*, *ordered-comm-semiring*) *ordered-comm-semiring*
<proof>

instance *fun* :: (*type*, *ordered-cancel-semiring*) *ordered-cancel-semiring* *<proof>*

instance *fun* :: (*type*, *ordered-cancel-comm-semiring*) *ordered-cancel-comm-semiring*
<proof>

instance *fun* :: (*type*, *ordered-ring*) *ordered-ring* *<proof>*

instance *fun* :: (*type*, *ordered-comm-ring*) *ordered-comm-ring* *<proof>*

lemmas *func-plus* = *plus-fun-def*
lemmas *func-zero* = *zero-fun-def*
lemmas *func-times* = *times-fun-def*
lemmas *func-one* = *one-fun-def*

end

45 Pointwise instantiation of functions to division

theory *Function-Division*
imports *Function-Algebras*
begin

45.1 Syntactic with division

instantiation *fun* :: (*type*, *inverse*) *inverse*
begin

definition *inverse f* = *inverse* \circ *f*

definition *f div g* = ($\lambda x. f\ x \ / \ g\ x$)

instance *<proof>*

end

lemma *inverse-fun-apply* [*simp*]:
inverse f x = *inverse* (*f x*)
<proof>

lemma *divide-fun-apply* [*simp*]:
 $(f \ / \ g)\ x$ = *f x* / *g x*
<proof>

Unfortunately, we cannot lift this operations to algebraic type classes for division: being different from the constant zero function $f \neq 0$ is too weak as precondition. So we must introduce our own set of lemmas.

abbreviation *zero-free* :: ($'b \Rightarrow 'a::field$) \Rightarrow *bool* **where**
 $zero\text{-}free\ f \equiv \neg (\exists x. f\ x = 0)$

lemma *fun-left-inverse*:
fixes $f :: 'b \Rightarrow 'a::field$
shows $zero\text{-}free\ f \Longrightarrow inverse\ f * f = 1$
 $\langle proof \rangle$

lemma *fun-right-inverse*:
fixes $f :: 'b \Rightarrow 'a::field$
shows $zero\text{-}free\ f \Longrightarrow f * inverse\ f = 1$
 $\langle proof \rangle$

lemma *fun-divide-inverse*:
fixes $f\ g :: 'b \Rightarrow 'a::field$
shows $f / g = f * inverse\ g$
 $\langle proof \rangle$

Feel free to extend this.

Another possibility would be a reformulation of the division type classes to use a *zero-free* predicate rather than a direct $a \neq 0$ condition.

end

46 Lexicographic order on functions

theory *Fun-Lexorder*
imports *Main*
begin

definition *less-fun* :: ($'a::linorder \Rightarrow 'b::linorder$) \Rightarrow ($'a \Rightarrow 'b$) \Rightarrow *bool*
where
 $less\text{-}fun\ f\ g \longleftrightarrow (\exists k. f\ k < g\ k \wedge (\forall k' < k. f\ k' = g\ k'))$

lemma *less-funI*:
assumes $\exists k. f\ k < g\ k \wedge (\forall k' < k. f\ k' = g\ k')$
shows $less\text{-}fun\ f\ g$
 $\langle proof \rangle$

lemma *less-funE*:
assumes $less\text{-}fun\ f\ g$
obtains k **where** $f\ k < g\ k$ **and** $\bigwedge k'. k' < k \Longrightarrow f\ k' = g\ k'$
 $\langle proof \rangle$

lemma *less-fun-asym*:

```

assumes less-fun f g
shows  $\neg$  less-fun g f
⟨proof⟩

```

```

lemma less-fun-irrefl:
   $\neg$  less-fun f f
⟨proof⟩

```

```

lemma less-fun-trans:
  assumes less-fun f g and less-fun g h
  shows less-fun f h
⟨proof⟩

```

```

lemma order-less-fun:
  class.order ( $\lambda f g. \text{less-fun } f g \vee f = g$ ) less-fun
⟨proof⟩

```

```

lemma less-fun-trichotomy:
  assumes finite {k. f k  $\neq$  g k}
  shows less-fun f g  $\vee f = g \vee$  less-fun g f
⟨proof⟩

```

```

end

```

47 The going-to filter

```

theory Going-To-Filter
  imports Complex-Main
begin

```

```

definition going-to-within :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b filter  $\Rightarrow$  'a set  $\Rightarrow$  'a filter
  (⟨⟨open-block notation=⟨mixfix going-to-within⟩(-)/ going'-to (-)/ within (-)⟩
  [1000,60,60] 60)
  where f going-to F within A = inf (filtercomap f F) (principal A)

```

```

abbreviation going-to :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b filter  $\Rightarrow$  'a filter
  (infix ⟨going'-to⟩ 60)
  where f going-to F  $\equiv$  f going-to F within UNIV

```

The *going-to* filter is, in a sense, the opposite of *filtermap*. It corresponds to the intuition of, given a function $f : A \rightarrow B$ and a filter F on the range of B , looking at such values of x that $f(x)$ approaches F . This can be written as f *going-to* F .

A classic example is the *at-infinity* filter, which describes the neighbourhood of infinity (i. e. all values sufficiently far away from the zero). This can also be written as *norm going-to at-top*.

Additionally, the *going-to* filter can be restricted with an optional ‘within’ parameter. For instance, if one would want to consider the filter of

complex numbers near infinity that do not lie on the negative real line, one could write *cmof going-to at-top within – complex-of-real* ‘ $\{..0\}$ ’.

A third, less mathematical example lies in the complexity analysis of algorithms. Suppose we wanted to say that an algorithm on lists takes $O(n^2)$ time where n is the length of the input list. We can write this using the Landau symbols from the AFP, where the underlying filter is *length going-to sequentially*. If, on the other hand, we want to look the complexity of the algorithm on sorted lists, we could use the filter *length going-to sequentially within Collect sorted*.

lemma *going-to-def*: $f \text{ going-to } F = \text{filtercomap } f \ F$
 ⟨proof⟩

lemma *eventually-going-toI* [intro]:
 assumes *eventually* $P \ F$
 shows *eventually* $(\lambda x. P \ (f \ x)) \ (f \text{ going-to } F)$
 ⟨proof⟩

lemma *filterlim-going-toI-weak* [intro]: $\text{filterlim } f \ F \ (f \text{ going-to } F \text{ within } A)$
 ⟨proof⟩

lemma *going-to-mono*: $F \leq G \implies A \subseteq B \implies f \text{ going-to } F \text{ within } A \leq f \text{ going-to } G \text{ within } B$
 ⟨proof⟩

lemma *going-to-inf*:
 $f \text{ going-to } (\inf F \ G) \text{ within } A = \inf (f \text{ going-to } F \text{ within } A) \ (f \text{ going-to } G \text{ within } A)$
 ⟨proof⟩

lemma *going-to-sup*:
 $f \text{ going-to } (\sup F \ G) \text{ within } A \geq \sup (f \text{ going-to } F \text{ within } A) \ (f \text{ going-to } G \text{ within } A)$
 ⟨proof⟩

lemma *going-to-top* [simp]: $f \text{ going-to top within } A = \text{principal } A$
 ⟨proof⟩

lemma *going-to-bot* [simp]: $f \text{ going-to bot within } A = \text{bot}$
 ⟨proof⟩

lemma *going-to-principal*:
 $f \text{ going-to principal } A \text{ within } B = \text{principal } (f - ' A \cap B)$
 ⟨proof⟩

lemma *going-to-within-empty* [simp]: $f \text{ going-to } F \text{ within } \{\} = \text{bot}$
 ⟨proof⟩

lemma *going-to-within-union* [simp]:

f going-to F within $(A \cup B) = \sup (f \text{ going-to } F \text{ within } A) (f \text{ going-to } F \text{ within } B)$

<proof>

lemma *eventually-going-to-at-top-linorder:*

fixes *f :: 'a \Rightarrow 'b :: linorder*

shows *eventually P (f going-to at-top within A) \longleftrightarrow ($\exists C. \forall x \in A. f\ x \geq C \longrightarrow P\ x$)*

<proof>

lemma *eventually-going-to-at-bot-linorder:*

fixes *f :: 'a \Rightarrow 'b :: linorder*

shows *eventually P (f going-to at-bot within A) \longleftrightarrow ($\exists C. \forall x \in A. f\ x \leq C \longrightarrow P\ x$)*

<proof>

lemma *eventually-going-to-at-top-dense:*

fixes *f :: 'a \Rightarrow 'b :: {linorder,no-top}*

shows *eventually P (f going-to at-top within A) \longleftrightarrow ($\exists C. \forall x \in A. f\ x > C \longrightarrow P\ x$)*

<proof>

lemma *eventually-going-to-at-bot-dense:*

fixes *f :: 'a \Rightarrow 'b :: {linorder,no-bot}*

shows *eventually P (f going-to at-bot within A) \longleftrightarrow ($\exists C. \forall x \in A. f\ x < C \longrightarrow P\ x$)*

<proof>

lemma *eventually-going-to-nhds:*

eventually P (f going-to nhds a within A) \longleftrightarrow

($\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f\ x \in S \longrightarrow P\ x)$)

<proof>

lemma *eventually-going-to-at:*

eventually P (f going-to (at a within B) within A) \longleftrightarrow

($\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f\ x \in B \cap S - \{a\} \longrightarrow P\ x)$)

<proof>

lemma *norm-going-to-at-top-eq: norm going-to at-top = at-infinity*

<proof>

lemmas *at-infinity-altdef = norm-going-to-at-top-eq [symmetric]*

end

48 Big sum and product over function bodies

theory *Groups-Big-Fun*

imports

Main
begin

48.1 Abstract product

locale *comm-monoid-fun* = *comm-monoid*
begin

definition $G :: ('b \Rightarrow 'a) \Rightarrow 'a$
where
expand-set: $G\ g = \text{comm-monoid-set}.F\ f\ \mathbf{1}\ g\ \{a. g\ a \neq \mathbf{1}\}$

interpretation F : *comm-monoid-set* $f\ \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *expand-superset*:
assumes *finite* A **and** $\{a. g\ a \neq \mathbf{1}\} \subseteq A$
shows $G\ g = F.F\ g\ A$
 $\langle \text{proof} \rangle$

lemma *conditionalize*:
assumes *finite* A
shows $F.F\ g\ A = G\ (\lambda a. \text{if } a \in A \text{ then } g\ a \text{ else } \mathbf{1})$
 $\langle \text{proof} \rangle$

lemma *neutral* [*simp*]:
 $G\ (\lambda a. \mathbf{1}) = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *update* [*simp*]:
assumes *finite* $\{a. g\ a \neq \mathbf{1}\}$
assumes $g\ a = \mathbf{1}$
shows $G\ (g(a := b)) = b * G\ g$
 $\langle \text{proof} \rangle$

lemma *infinite* [*simp*]:
 $\neg \text{finite } \{a. g\ a \neq \mathbf{1}\} \implies G\ g = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *cong* [*cong*]:
assumes $\bigwedge a. g\ a = h\ a$
shows $G\ g = G\ h$
 $\langle \text{proof} \rangle$

lemma *not-neutral-obtains-not-neutral*:
assumes $G\ g \neq \mathbf{1}$
obtains a **where** $g\ a \neq \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *reindex-cong*:

assumes *bij* *l*
 assumes $g \circ l = h$
 shows $G\ g = G\ h$

$\langle proof \rangle$

lemma *distrib*:

assumes *finite* $\{a. g\ a \neq 1\}$ and *finite* $\{a. h\ a \neq 1\}$
 shows $G\ (\lambda a. g\ a * h\ a) = G\ g * G\ h$

$\langle proof \rangle$

lemma *swap*:

assumes *finite* *C*
 assumes *subset*: $\{a. \exists b. g\ a\ b \neq 1\} \times \{b. \exists a. g\ a\ b \neq 1\} \subseteq C$ (is $?A \times ?B \subseteq C$)
 shows $G\ (\lambda a. G\ (g\ a)) = G\ (\lambda b. G\ (\lambda a. g\ a\ b))$

$\langle proof \rangle$

lemma *cartesian-product*:

assumes *finite* *C*
 assumes *subset*: $\{a. \exists b. g\ a\ b \neq 1\} \times \{b. \exists a. g\ a\ b \neq 1\} \subseteq C$ (is $?A \times ?B \subseteq C$)
 shows $G\ (\lambda a. G\ (g\ a)) = G\ (\lambda(a, b). g\ a\ b)$

$\langle proof \rangle$

lemma *cartesian-product2*:

assumes *fin*: *finite* *D*
 assumes *subset*: $\{(a, b). \exists c. g\ a\ b\ c \neq 1\} \times \{c. \exists a\ b. g\ a\ b\ c \neq 1\} \subseteq D$ (is $?AB \times ?C \subseteq D$)
 shows $G\ (\lambda(a, b). G\ (g\ a\ b)) = G\ (\lambda(a, b, c). g\ a\ b\ c)$

$\langle proof \rangle$

lemma *delta [simp]*:

$G\ (\lambda b. \text{if } b = a \text{ then } g\ b \text{ else } 1) = g\ a$

$\langle proof \rangle$

lemma *delta' [simp]*:

$G\ (\lambda b. \text{if } a = b \text{ then } g\ b \text{ else } 1) = g\ a$

$\langle proof \rangle$

end

48.2 Concrete sum

context *comm-monoid-add*

begin

sublocale *Sum-any: comm-monoid-fun plus 0*

```

rewrites comm-monoid-set.F plus 0 = sum
defines Sum-any = Sum-any.G
⟨proof⟩

end

syntax (ASCII)
  -Sum-any :: pttrn ⇒ 'a ⇒ 'a::comm-monoid-add  (⟨(⟨indent=3 notation=⟨binder
SUM⟩⟩SUM -. -)⟩ [0, 10] 10)
syntax
  -Sum-any :: pttrn ⇒ 'a ⇒ 'a::comm-monoid-add  (⟨(⟨indent=2 notation=⟨binder
Σ⟩⟩Σ -. -)⟩ [0, 10] 10)
syntax-consts
  -Sum-any ⇒ Sum-any
translations
   $\sum a. b \Rightarrow \text{CONST } \textit{Sum-any} (\lambda a. b)$ 

lemma Sum-any-left-distrib:
  fixes r :: 'a :: semiring-0
  assumes finite {a. g a ≠ 0}
  shows Sum-any g * r = ( $\sum n. g\ n * r$ )
  ⟨proof⟩

lemma Sum-any-right-distrib:
  fixes r :: 'a :: semiring-0
  assumes finite {a. g a ≠ 0}
  shows r * Sum-any g = ( $\sum n. r * g\ n$ )
  ⟨proof⟩

lemma Sum-any-product:
  fixes f g :: 'b ⇒ 'a::semiring-0
  assumes finite {a. f a ≠ 0} and finite {b. g b ≠ 0}
  shows Sum-any f * Sum-any g = ( $\sum a. \sum b. f\ a * g\ b$ )
  ⟨proof⟩

lemma Sum-any-eq-zero-iff [simp]:
  fixes f :: 'a ⇒ nat
  assumes finite {a. f a ≠ 0}
  shows Sum-any f = 0 ⟷ f = ( $\lambda -. 0$ )
  ⟨proof⟩

```

48.3 Concrete product

```

context comm-monoid-mult
begin

sublocale Prod-any: comm-monoid-fun times 1
  rewrites comm-monoid-set.F times 1 = prod
  defines Prod-any = Prod-any.G

```

<proof>

end

syntax (*ASCII*)

-*Prod-any* :: *pttrn* \Rightarrow '*a* \Rightarrow '*a*::*comm-monoid-mult* (\langle (\langle *indent*=4 *notation*= \langle *binder*
PROD $\rangle\rangle$ *PROD* -. -) \rangle [*0*, *10*] *10*)

syntax

-*Prod-any* :: *pttrn* \Rightarrow '*a* \Rightarrow '*a*::*comm-monoid-mult* (\langle (\langle *indent*=2 *notation*= \langle *binder*
 \prod $\rangle\rangle$ \prod -. -) \rangle [*0*, *10*] *10*)

syntax-consts

-*Prod-any* == *Prod-any*

translations

$\prod a. b == \text{CONST } \text{Prod-any } (\lambda a. b)$

lemma *Prod-any-zero*:

fixes *f* :: '*b* \Rightarrow '*a* :: *comm-semiring-1*

assumes *finite* {*a*. *f a* \neq 1}

assumes *f a* = 0

shows ($\prod a. f a$) = 0

<proof>

lemma *Prod-any-not-zero*:

fixes *f* :: '*b* \Rightarrow '*a* :: *comm-semiring-1*

assumes *finite* {*a*. *f a* \neq 1}

assumes ($\prod a. f a$) \neq 0

shows *f a* \neq 0

<proof>

lemma *power-Sum-any*:

assumes *finite* {*a*. *f a* \neq 0}

shows *c* \wedge ($\sum a. f a$) = ($\prod a. c \wedge f a$)

<proof>

end

49 Infinite Type Class

The type class of infinite sets (originally from the Incredible Proof Machine)

theory *Infinite-Typeclass*

imports *Complex-Main*

begin

class *infinite* =

assumes *infinite-UNIV*: *infinite* (*UNIV*::'*a set*)

begin

lemma *arb-element*: $\text{finite } Y \implies \exists x :: 'a. x \notin Y$
 $\langle \text{proof} \rangle$

lemma *arb-finite-subset*: $\text{finite } Y \implies \exists X :: 'a \text{ set}. Y \cap X = \{\} \wedge \text{finite } X \wedge n \leq \text{card } X$
 $\langle \text{proof} \rangle$

lemma *arb-inj-on-finite-infinite*: $\text{finite}(A :: 'b \text{ set}) \implies \exists f :: 'b \Rightarrow 'a. \text{inj-on } f \ A$
 $\langle \text{proof} \rangle$

lemma *arb-countable-map*: $\text{finite } Y \implies \exists f :: (\text{nat} \Rightarrow 'a). \text{inj } f \wedge \text{range } f \subseteq \text{UNIV} - Y$
 $\langle \text{proof} \rangle$

end

instance *nat* :: *infinite*
 $\langle \text{proof} \rangle$

instance *int* :: *infinite*
 $\langle \text{proof} \rangle$

instance *rat* :: *infinite*
 $\langle \text{proof} \rangle$

instance *real* :: *infinite*
 $\langle \text{proof} \rangle$

instance *complex* :: *infinite*
 $\langle \text{proof} \rangle$

instance *option* :: (*infinite*) *infinite*
 $\langle \text{proof} \rangle$

instance *prod* :: (*infinite*, *type*) *infinite*
 $\langle \text{proof} \rangle$

instance *list* :: (*type*) *infinite*
 $\langle \text{proof} \rangle$

end

50 Algebraic operations on sets

theory *Set-Algebras*
imports *Main*
begin

This library lifts operations like addition and multiplication to sets. It

was designed to support asymptotic calculations for the now-obsolete BigO theory, but has other uses.

instantiation *set* :: (*plus*) *plus*
begin

definition *plus-set* :: 'a::plus *set* \Rightarrow 'a *set* \Rightarrow 'a *set*
where *set-plus-def*: $A + B = \{c. \exists a \in A. \exists b \in B. c = a + b\}$

instance \langle *proof* \rangle

end

instantiation *set* :: (*times*) *times*
begin

definition *times-set* :: 'a::times *set* \Rightarrow 'a *set* \Rightarrow 'a *set*
where *set-times-def*: $A * B = \{c. \exists a \in A. \exists b \in B. c = a * b\}$

instance \langle *proof* \rangle

end

instantiation *set* :: (*zero*) *zero*
begin

definition *set-zero[simp]*: $(0::'a::zero \text{ set}) = \{0\}$

instance \langle *proof* \rangle

end

instantiation *set* :: (*one*) *one*
begin

definition *set-one[simp]*: $(1::'a::one \text{ set}) = \{1\}$

instance \langle *proof* \rangle

end

definition *elt-set-plus* :: 'a::plus \Rightarrow 'a *set* \Rightarrow 'a *set* (**infixl** $\langle +_o \rangle$ 70)
where $a +_o B = \{c. \exists b \in B. c = a + b\}$

definition *elt-set-times* :: 'a::times \Rightarrow 'a *set* \Rightarrow 'a *set* (**infixl** $\langle *_o \rangle$ 80)
where $a *_o B = \{c. \exists b \in B. c = a * b\}$

abbreviation (*input*) *elt-set-eq* :: 'a \Rightarrow 'a *set* \Rightarrow bool (**infix** $\langle =_o \rangle$ 50)
where $x =_o A \equiv x \in A$

instance *set* :: (*semigroup-add*) *semigroup-add*
 ⟨*proof*⟩

instance *set* :: (*ab-semigroup-add*) *ab-semigroup-add*
 ⟨*proof*⟩

instance *set* :: (*monoid-add*) *monoid-add*
 ⟨*proof*⟩

instance *set* :: (*comm-monoid-add*) *comm-monoid-add*
 ⟨*proof*⟩

instance *set* :: (*semigroup-mult*) *semigroup-mult*
 ⟨*proof*⟩

instance *set* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
 ⟨*proof*⟩

instance *set* :: (*monoid-mult*) *monoid-mult*
 ⟨*proof*⟩

instance *set* :: (*comm-monoid-mult*) *comm-monoid-mult*
 ⟨*proof*⟩

lemma *sumset-empty* [*simp*]: $A + \{\} = \{\} \{\} + A = \{\}$
 ⟨*proof*⟩

lemma *Un-set-plus*: $(A \cup B) + C = (A+C) \cup (B+C)$ **and** *set-plus-Un*: $C + (A \cup B) = (C+A) \cup (C+B)$
 ⟨*proof*⟩

lemma
fixes $A :: 'a::comm-monoid-add$ *set*
shows *insert-set-plus*: $(insert\ a\ A) + B = (A+B) \cup (((+)\ a)\ 'B)$ **and** *set-plus-insert*:
 $B + (insert\ a\ A) = (B+A) \cup (((+)\ a)\ 'B)$
 ⟨*proof*⟩

lemma *set-add-0* [*simp*]:
fixes $A :: 'a::comm-monoid-add$ *set*
shows $\{0\} + A = A$
 ⟨*proof*⟩

lemma *set-add-0-right* [*simp*]:
fixes $A :: 'a::comm-monoid-add$ *set*
shows $A + \{0\} = A$
 ⟨*proof*⟩

lemma *card-plus-sing*:
fixes $A :: 'a::ab-group-add$ *set*

shows $\text{card } (A + \{a\}) = \text{card } A$
 $\langle \text{proof} \rangle$

lemma *set-plus-intro* [intro]: $a \in C \implies b \in D \implies a + b \in C + D$
 $\langle \text{proof} \rangle$

lemma *set-plus-elim*:
assumes $x \in A + B$
obtains $a \ b$ **where** $x = a + b$ **and** $a \in A$ **and** $b \in B$
 $\langle \text{proof} \rangle$

lemma *set-plus-intro2* [intro]: $b \in C \implies a + b \in a + o \ C$
 $\langle \text{proof} \rangle$

lemma *set-plus-rearrange*: $(a + o \ C) + (b + o \ D) = (a + b) + o \ (C + D)$
for $a \ b :: 'a :: \text{comm-monoid-add}$
 $\langle \text{proof} \rangle$

lemma *set-plus-rearrange2*: $a + o \ (b + o \ C) = (a + b) + o \ C$
for $a \ b :: 'a :: \text{semigroup-add}$
 $\langle \text{proof} \rangle$

lemma *set-plus-rearrange3*: $(a + o \ B) + C = a + o \ (B + C)$
for $a :: 'a :: \text{semigroup-add}$
 $\langle \text{proof} \rangle$

theorem *set-plus-rearrange4*: $C + (a + o \ D) = a + o \ (C + D)$
for $a :: 'a :: \text{comm-monoid-add}$
 $\langle \text{proof} \rangle$

lemmas *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2 set-plus-rearrange3 set-plus-rearrange4*

lemma *set-plus-mono* [intro!]: $C \subseteq D \implies a + o \ C \subseteq a + o \ D$
 $\langle \text{proof} \rangle$

lemma *set-plus-mono2* [intro]: $C \subseteq D \implies E \subseteq F \implies C + E \subseteq D + F$
for $C \ D \ E \ F :: 'a :: \text{plus set}$
 $\langle \text{proof} \rangle$

lemma *set-plus-mono3* [intro]: $a \in C \implies a + o \ D \subseteq C + D$
 $\langle \text{proof} \rangle$

lemma *set-plus-mono4* [intro]: $a \in C \implies a + o \ D \subseteq D + C$
for $a :: 'a :: \text{comm-monoid-add}$
 $\langle \text{proof} \rangle$

lemma *set-plus-mono5*: $a \in C \implies B \subseteq D \implies a + o \ B \subseteq C + D$
 $\langle \text{proof} \rangle$

lemma *set-plus-mono-b*: $C \subseteq D \implies x \in a +_o C \implies x \in a +_o D$
 ⟨proof⟩

lemma *set-zero-plus* [simp]: $0 +_o C = C$
for $C :: 'a::comm-monoid-add\ set$
 ⟨proof⟩

lemma *set-zero-plus2*: $0 \in A \implies B \subseteq A + B$
for $A\ B :: 'a::comm-monoid-add\ set$
 ⟨proof⟩

lemma *set-plus-imp-minus*: $a \in b +_o C \implies a - b \in C$
for $a\ b :: 'a::ab-group-add$
 ⟨proof⟩

lemma *set-minus-imp-plus*: $a - b \in C \implies a \in b +_o C$
for $a\ b :: 'a::ab-group-add$
 ⟨proof⟩

lemma *set-minus-plus*: $a - b \in C \longleftrightarrow a \in b +_o C$
for $a\ b :: 'a::ab-group-add$
 ⟨proof⟩

lemma *set-times-intro* [intro]: $a \in C \implies b \in D \implies a * b \in C * D$
 ⟨proof⟩

lemma *set-times-elim*:
assumes $x \in A * B$
obtains $a\ b$ **where** $x = a * b$ **and** $a \in A$ **and** $b \in B$
 ⟨proof⟩

lemma *set-times-intro2* [intro!]: $b \in C \implies a * b \in a *_o C$
 ⟨proof⟩

lemma *set-times-rearrange*: $(a *_o C) * (b *_o D) = (a * b) *_o (C * D)$
for $a\ b :: 'a::comm-monoid-mult$
 ⟨proof⟩

lemma *set-times-rearrange2*: $a *_o (b *_o C) = (a * b) *_o C$
for $a\ b :: 'a::semigroup-mult$
 ⟨proof⟩

lemma *set-times-rearrange3*: $(a *_o B) * C = a *_o (B * C)$
for $a :: 'a::semigroup-mult$
 ⟨proof⟩

theorem *set-times-rearrange4*: $C * (a *_o D) = a *_o (C * D)$
for $a :: 'a::comm-monoid-mult$

$\langle \text{proof} \rangle$

lemmas *set-times-rearranges* = *set-times-rearrange set-times-rearrange2*
set-times-rearrange3 set-times-rearrange4

lemma *set-times-mono* [intro]: $C \subseteq D \implies a * o C \subseteq a * o D$
 $\langle \text{proof} \rangle$

lemma *set-times-mono2* [intro]: $C \subseteq D \implies E \subseteq F \implies C * E \subseteq D * F$
for $C D E F :: 'a::\text{times set}$
 $\langle \text{proof} \rangle$

lemma *set-times-mono3* [intro]: $a \in C \implies a * o D \subseteq C * D$
 $\langle \text{proof} \rangle$

lemma *set-times-mono4* [intro]: $a \in C \implies a * o D \subseteq D * C$
for $a :: 'a::\text{comm-monoid-mult}$
 $\langle \text{proof} \rangle$

lemma *set-times-mono5*: $a \in C \implies B \subseteq D \implies a * o B \subseteq C * D$
 $\langle \text{proof} \rangle$

lemma *set-one-times* [simp]: $1 * o C = C$
for $C :: 'a::\text{comm-monoid-mult set}$
 $\langle \text{proof} \rangle$

lemma *set-times-plus-distrib*: $a * o (b + o C) = (a * b) + o (a * o C)$
for $a b :: 'a::\text{semiring}$
 $\langle \text{proof} \rangle$

lemma *set-times-plus-distrib2*: $a * o (B + C) = (a * o B) + (a * o C)$
for $a :: 'a::\text{semiring}$
 $\langle \text{proof} \rangle$

lemma *set-times-plus-distrib3*: $(a + o C) * D \subseteq a * o D + C * D$
for $a :: 'a::\text{semiring}$
 $\langle \text{proof} \rangle$

lemmas *set-times-plus-distribs* =
set-times-plus-distrib
set-times-plus-distrib2

lemma *set-neg-intro*: $a \in (- 1) * o C \implies - a \in C$
for $a :: 'a::\text{ring-1}$
 $\langle \text{proof} \rangle$

lemma *set-neg-intro2*: $a \in C \implies - a \in (- 1) * o C$
for $a :: 'a::\text{ring-1}$
 $\langle \text{proof} \rangle$

lemma *set-plus-image*: $S + T = (\lambda(x, y). x + y) \text{ ` } (S \times T)$
 ⟨proof⟩

lemma *set-times-image*: $S * T = (\lambda(x, y). x * y) \text{ ` } (S \times T)$
 ⟨proof⟩

lemma *finite-set-plus*: $\text{finite } s \implies \text{finite } t \implies \text{finite } (s + t)$
 ⟨proof⟩

lemma *finite-set-times*: $\text{finite } s \implies \text{finite } t \implies \text{finite } (s * t)$
 ⟨proof⟩

lemma *set-sum-alt*:
 assumes *fin*: $\text{finite } I$
 shows $\text{sum } S \text{ } I = \{\text{sum } s \text{ } I \mid s. \forall i \in I. s \text{ } i \in S \text{ } i\}$
 (is $- = ?\text{sum } I$)
 ⟨proof⟩

lemma *sum-set-cond-linear*:
 fixes $f :: 'a::\text{comm-monoid-add set} \Rightarrow 'b::\text{comm-monoid-add set}$
 assumes [intro!]: $\bigwedge A \ B. P \ A \implies P \ B \implies P \ (A + B) \ P \ \{0\}$
 and f : $\bigwedge A \ B. P \ A \implies P \ B \implies f \ (A + B) = f \ A + f \ B \ f \ \{0\} = \{0\}$
 assumes *all*: $\bigwedge i. i \in I \implies P \ (S \text{ } i)$
 shows $f \ (\text{sum } S \text{ } I) = \text{sum } (f \circ S) \text{ } I$
 ⟨proof⟩

lemma *sum-set-linear*:
 fixes $f :: 'a::\text{comm-monoid-add set} \Rightarrow 'b::\text{comm-monoid-add set}$
 assumes $\bigwedge A \ B. f(A) + f(B) = f(A + B) \ f \ \{0\} = \{0\}$
 shows $f \ (\text{sum } S \text{ } I) = \text{sum } (f \circ S) \text{ } I$
 ⟨proof⟩

lemma *set-times-Un-distrib*:
 $A * (B \cup C) = A * B \cup A * C$
 $(A \cup B) * C = A * C \cup B * C$
 ⟨proof⟩

lemma *set-times-UNION-distrib*:
 $A * \bigcup (M \text{ ` } I) = (\bigcup i \in I. A * M \text{ } i)$
 $\bigcup (M \text{ ` } I) * A = (\bigcup i \in I. M \text{ } i * A)$
 ⟨proof⟩

end

51 Interval Type

theory *Interval*
 imports

Complex-Main
Lattice-Algebras
Set-Algebras

begin

A type of non-empty, closed intervals.

typedef (overloaded) 'a interval =
 {(a::'a::preorder, b). a ≤ b}
morphisms bounds-of-interval Interval
 ⟨proof⟩

setup-lifting type-definition-interval

lift-definition lower::('a::preorder) interval ⇒ 'a is fst ⟨proof⟩

lift-definition upper::('a::preorder) interval ⇒ 'a is snd ⟨proof⟩

lemma interval-eq-iff: a = b ⟷ lower a = lower b ∧ upper a = upper b
 ⟨proof⟩

lemma interval-eqI: lower a = lower b ⟹ upper a = upper b ⟹ a = b
 ⟨proof⟩

lemma lower-le-upper[simp]: lower i ≤ upper i
 ⟨proof⟩

lift-definition set-of :: 'a::preorder interval ⇒ 'a set is λx. {fst x .. snd x} ⟨proof⟩

lemma set-of-eq: set-of x = {lower x .. upper x}
 ⟨proof⟩

context notes [[typedef-overloaded]] **begin**

lift-definition(code-dt) Interval'::'a::preorder ⇒ 'a::preorder ⇒ 'a interval option
 is λa b. if a ≤ b then Some (a, b) else None
 ⟨proof⟩

lemma Interval'-split:
 P (Interval' a b) ⟷
 (∀ ivl. a ≤ b ⟶ lower ivl = a ⟶ upper ivl = b ⟶ P (Some ivl)) ∧ (¬a ≤ b
 ⟶ P None)
 ⟨proof⟩

lemma Interval'-split-asm:
 P (Interval' a b) ⟷
 ¬((∃ ivl. a ≤ b ∧ lower ivl = a ∧ upper ivl = b ∧ ¬P (Some ivl)) ∨ (¬a ≤ b ∧
 ¬P None))
 ⟨proof⟩

lemmas *Interval'-splits* = *Interval'-split* *Interval'-split-asm*

lemma *Interval'-eq-Some*: *Interval' a b = Some i \implies lower i = a \wedge upper i = b*
<proof>

end

instantiation *interval* :: (*{preorder, equal}*) *equal*
begin

definition *equal-class.equal a b* \equiv (*lower a = lower b*) \wedge (*upper a = upper b*)

instance *<proof>*
end

instantiation *interval* :: (*preorder*) *ord* **begin**

definition *less-eq-interval* :: '*a interval* \Rightarrow '*a interval* \Rightarrow *bool*
where *less-eq-interval a b* \longleftrightarrow *lower b \leq lower a \wedge upper a \leq upper b*

definition *less-interval* :: '*a interval* \Rightarrow '*a interval* \Rightarrow *bool*
where *less-interval x y* = (*x \leq y \wedge \neg y \leq x*)

instance *<proof>*
end

instantiation *interval* :: (*lattice*) *semilattice-sup*
begin

lift-definition *sup-interval* :: '*a interval* \Rightarrow '*a interval* \Rightarrow '*a interval*
is $\lambda(a, b) (c, d). (inf\ a\ c, sup\ b\ d)$
<proof>

lemma *lower-sup[simp]*: *lower (sup A B) = inf (lower A) (lower B)*
<proof>

lemma *upper-sup[simp]*: *upper (sup A B) = sup (upper A) (upper B)*
<proof>

instance *<proof>*
end

lemma *set-of-interval-union*: *set-of A \cup set-of B \subseteq set-of (sup A B)* **for** *A::'a::lattice interval*
<proof>

lemma *interval-union-commute*: *sup A B = sup B A* **for** *A::'a::lattice interval*
<proof>

lemma *interval-union-mono1*: *set-of* *a* \subseteq *set-of* (*sup* *a* *A*) **for** *A* :: '*a*::*lattice interval*

⟨*proof*⟩

lemma *interval-union-mono2*: *set-of* *A* \subseteq *set-of* (*sup* *a* *A*) **for** *A* :: '*a*::*lattice interval*

⟨*proof*⟩

lift-definition *interval-of* :: '*a*::*preorder* \Rightarrow '*a interval* **is** $\lambda x. (x, x)$

⟨*proof*⟩

lemma *lower-interval-of[simp]*: *lower* (*interval-of* *a*) = *a*

⟨*proof*⟩

lemma *upper-interval-of[simp]*: *upper* (*interval-of* *a*) = *a*

⟨*proof*⟩

definition *width* :: '*a*::{*preorder, minus*} *interval* \Rightarrow '*a*

where *width* *i* = *upper* *i* – *lower* *i*

instantiation *interval* :: (*ordered-ab-semigroup-add*) *ab-semigroup-add*

begin

lift-definition *plus-interval*::'*a interval* \Rightarrow '*a interval* \Rightarrow '*a interval*

is $\lambda(a, b). \lambda(c, d). (a + c, b + d)$

⟨*proof*⟩

lemma *lower-plus[simp]*: *lower* (*plus* *A* *B*) = *plus* (*lower* *A*) (*lower* *B*)

⟨*proof*⟩

lemma *upper-plus[simp]*: *upper* (*plus* *A* *B*) = *plus* (*upper* *A*) (*upper* *B*)

⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *interval* :: ({*ordered-ab-semigroup-add, lattice*}) *ordered-ab-semigroup-add*

⟨*proof*⟩

instantiation *interval* :: ({*preorder, zero*}) *zero*

begin

lift-definition *zero-interval*::'*a interval* **is** (*0*, *0*) ⟨*proof*⟩

lemma *lower-zero[simp]*: *lower* *0* = *0*

⟨*proof*⟩

lemma *upper-zero[simp]*: *upper* *0* = *0*

⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *interval* :: ($\{\text{ordered-comm-monoid-add}\}$) *comm-monoid-add*
 $\langle \text{proof} \rangle$

instance *interval* :: ($\{\text{ordered-comm-monoid-add}, \text{lattice}\}$) *ordered-comm-monoid-add*
 $\langle \text{proof} \rangle$

instantiation *interval* :: ($\{\text{ordered-ab-group-add}\}$) *uminus*
begin

lift-definition *uminus-interval*::'a *interval* \Rightarrow 'a *interval* **is** $\lambda(a, b). (-b, -a)$
 $\langle \text{proof} \rangle$

lemma *lower-uminus[simp]*: *lower* $(- A) = - \text{upper } A$
 $\langle \text{proof} \rangle$

lemma *upper-uminus[simp]*: *upper* $(- A) = - \text{lower } A$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$
end

instantiation *interval* :: ($\{\text{ordered-ab-group-add}\}$) *minus*
begin

definition *minus-interval*::'a *interval* \Rightarrow 'a *interval* \Rightarrow 'a *interval*
where *minus-interval* *a b* = *a* + - *b*

lemma *lower-minus[simp]*: *lower* (*minus A B*) = *minus* (*lower A*) (*upper B*)
 $\langle \text{proof} \rangle$

lemma *upper-minus[simp]*: *upper* (*minus A B*) = *minus* (*upper A*) (*lower B*)
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$
end

instantiation *interval* :: ($\{\text{times}, \text{linorder}\}$) *times*
begin

lift-definition *times-interval* :: 'a *interval* \Rightarrow 'a *interval* \Rightarrow 'a *interval*
is $\lambda(a1, a2). \lambda(b1, b2).$
 $(\text{let } x1 = a1 * b1; x2 = a1 * b2; x3 = a2 * b1; x4 = a2 * b2$
 $\text{in } (\text{min } x1 (\text{min } x2 (\text{min } x3 x4)), \text{max } x1 (\text{max } x2 (\text{max } x3 x4))))$
 $\langle \text{proof} \rangle$

lemma *lower-times*:

lower (*times A B*) = *Min* {*lower A* * *lower B*, *lower A* * *upper B*, *upper A* *
lower B, *upper A* * *upper B*}
 $\langle \text{proof} \rangle$

lemma *upper-times*:

upper (*times A B*) = *Max* {*lower A* * *lower B*, *lower A* * *upper B*, *upper A* *
lower B, *upper A* * *upper B*}
 $\langle \text{proof} \rangle$

instance $\langle proof \rangle$
end

lemma *interval-eq-set-of-iff*: $X = Y \longleftrightarrow \text{set-of } X = \text{set-of } Y$ **for** $X\ Y :: 'a :: \text{order interval}$
 $\langle proof \rangle$

51.1 Membership

abbreviation (**in** *preorder*) *in-interval* $(\langle \langle \text{notation} = \langle infix \in_i \rangle \rangle - / \in_i - \rangle [51, 51]$
 $50)$
where *in-interval* $x\ X \equiv x \in \text{set-of } X$

lemma *in-interval-to-interval[intro!]*: $a \in_i \text{interval-of } a$
 $\langle proof \rangle$

lemma *plus-in-intervalI*:
fixes $x\ y :: 'a :: \text{ordered-ab-semigroup-add}$
shows $x \in_i X \Longrightarrow y \in_i Y \Longrightarrow x + y \in_i X + Y$
 $\langle proof \rangle$

lemma *connected-set-of[intro, simp]*:
 $\text{connected } (\text{set-of } X)$ **for** $X :: 'a :: \text{linear-continuum-topology interval}$
 $\langle proof \rangle$

lemma *ex-sum-in-interval-lemma*: $\exists xa \in \{la .. ua\}. \exists xb \in \{lb .. ub\}. x = xa + xb$
if $la \leq ua\ lb \leq ub\ la + lb \leq x\ x \leq ua + ub$
 $ua - la \leq ub - lb$
for $la\ b\ c\ d :: 'a :: \text{linordered-ab-group-add}$
 $\langle proof \rangle$

lemma *ex-sum-in-interval*: $\exists xa \geq la. xa \leq ua \wedge (\exists xb \geq lb. xb \leq ub \wedge x = xa + xb)$
if $a: la \leq ua$ **and** $b: lb \leq ub$ **and** $x: la + lb \leq x \leq ua + ub$
for $la\ b\ c\ d :: 'a :: \text{linordered-ab-group-add}$
 $\langle proof \rangle$

lemma *Icc-plus-Icc*:
 $\{a .. b\} + \{c .. d\} = \{a + c .. b + d\}$
if $a \leq b\ c \leq d$
for $a\ b\ c\ d :: 'a :: \text{linordered-ab-group-add}$
 $\langle proof \rangle$

lemma *set-of-plus*:
fixes $A :: 'a :: \text{linordered-ab-group-add interval}$
shows $\text{set-of } (A + B) = \text{set-of } A + \text{set-of } B$
 $\langle proof \rangle$

lemma *plus-in-intervalE*:

fixes $xy :: 'a :: \text{linordered-ab-group-add}$

assumes $xy \in_i X + Y$

obtains $x \ y$ **where** $xy = x + y$ $x \in_i X$ $y \in_i Y$

$\langle \text{proof} \rangle$

lemma *set-of-uminus*: $\text{set-of } (-X) = \{-x \mid x. x \in \text{set-of } X\}$

for $X :: 'a :: \text{ordered-ab-group-add interval}$

$\langle \text{proof} \rangle$

lemma *uminus-in-intervalI*:

fixes $x :: 'a :: \text{ordered-ab-group-add}$

shows $x \in_i X \implies -x \in_i -X$

$\langle \text{proof} \rangle$

lemma *uminus-in-intervalD*:

fixes $x :: 'a :: \text{ordered-ab-group-add}$

shows $x \in_i -X \implies -x \in_i X$

$\langle \text{proof} \rangle$

lemma *minus-in-intervalI*:

fixes $x \ y :: 'a :: \text{ordered-ab-group-add}$

shows $x \in_i X \implies y \in_i Y \implies x - y \in_i X - Y$

$\langle \text{proof} \rangle$

lemma *set-of-minus*: $\text{set-of } (X - Y) = \{x - y \mid x \ y. x \in \text{set-of } X \wedge y \in \text{set-of } Y\}$

for $X \ Y :: 'a :: \text{linordered-ab-group-add interval}$

$\langle \text{proof} \rangle$

lemma *times-in-intervalI*:

fixes $x \ y :: 'a :: \text{linordered-ring}$

assumes $x \in_i X$ $y \in_i Y$

shows $x * y \in_i X * Y$

$\langle \text{proof} \rangle$

lemma *times-in-intervalE*:

fixes $xy :: 'a :: \{\text{linorder}, \text{real-normed-algebra}, \text{linear-continuum-topology}\}$

— TODO: linear continuum topology is pretty strong

assumes $xy \in_i X * Y$

obtains $x \ y$ **where** $xy = x * y$ $x \in_i X$ $y \in_i Y$

$\langle \text{proof} \rangle$

thm *times-in-intervalE*[of $1 :: \text{real}$]

lemma *set-of-times*: $\text{set-of } (X * Y) = \{x * y \mid x \ y. x \in \text{set-of } X \wedge y \in \text{set-of } Y\}$

for $X \ Y :: 'a :: \{\text{linordered-ring}, \text{real-normed-algebra}, \text{linear-continuum-topology}\}$

interval

$\langle \text{proof} \rangle$

instance *interval* :: (linordered-idom) *cancel-semigroup-add*

⟨proof⟩

lemma *interval-mul-commute*: $A * B = B * A$ **for** $A B :: 'a::linordered-idom$ *interval*
 ⟨proof⟩

lemma *interval-times-zero-right[simp]*: $A * 0 = 0$ **for** $A :: 'a::linordered-ring$ *interval*
 ⟨proof⟩

lemma *interval-times-zero-left[simp]*:
 $0 * A = 0$ **for** $A :: 'a::linordered-ring$ *interval*
 ⟨proof⟩

instantiation *interval* :: ($\{preorder, one\}$) *one*
begin

lift-definition *one-interval*:: $'a$ *interval* **is** $(1, 1)$ ⟨proof⟩

lemma *lower-one[simp]*: $lower\ 1 = 1$
 ⟨proof⟩

lemma *upper-one[simp]*: $upper\ 1 = 1$
 ⟨proof⟩

instance ⟨proof⟩
end

instance *interval* :: ($\{one, preorder, linorder, times\}$) *power*
 ⟨proof⟩

lemma *set-of-one[simp]*: $set-of\ (1::'a::\{one, order\}\ interval) = \{1\}$
 ⟨proof⟩

instance *interval* ::
 ($\{linordered-idom, real-normed-algebra, linear-continuum-topology\}$) *monoid-mult*
 ⟨proof⟩

lemma *one-times-ivl-left[simp]*: $1 * A = A$ **for** $A :: 'a::linordered-idom$ *interval*
 ⟨proof⟩

lemma *one-times-ivl-right[simp]*: $A * 1 = A$ **for** $A :: 'a::linordered-idom$ *interval*
 ⟨proof⟩

lemma *set-of-power-mono*: $a^{\hat{n}} \in set-of\ (A^{\hat{n}})$ **if** $a \in set-of\ A$
for $a :: 'a::linordered-idom$
 ⟨proof⟩

lemma *set-of-add-cong*:
 $set-of\ (A + B) = set-of\ (A' + B')$
if $set-of\ A = set-of\ A'$ $set-of\ B = set-of\ B'$
for $A :: 'a::linordered-ab-group-add$ *interval*

$\langle proof \rangle$

lemma *set-of-add-inc-left*:
 $set-of (A + B) \subseteq set-of (A' + B)$
if $set-of A \subseteq set-of A'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-add-inc-right*:
 $set-of (A + B) \subseteq set-of (A + B')$
if $set-of B \subseteq set-of B'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-add-inc*:
 $set-of (A + B) \subseteq set-of (A' + B')$
if $set-of A \subseteq set-of A' set-of B \subseteq set-of B'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-neg-inc*:
 $set-of (-A) \subseteq set-of (-A')$
if $set-of A \subseteq set-of A'$
for $A :: 'a::ordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-sub-inc-left*:
 $set-of (A - B) \subseteq set-of (A' - B)$
if $set-of A \subseteq set-of A'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-sub-inc-right*:
 $set-of (A - B) \subseteq set-of (A - B')$
if $set-of B \subseteq set-of B'$
for $A :: 'a::linordered-ab-group-add interval$
 $\langle proof \rangle$

lemma *set-of-sub-inc*:
 $set-of (A - B) \subseteq set-of (A' - B')$
if $set-of A \subseteq set-of A' set-of B \subseteq set-of B'$
for $A :: 'a::linordered-idom interval$
 $\langle proof \rangle$

lemma *set-of-mul-inc-right*:
 $set-of (A * B) \subseteq set-of (A * B')$
if $set-of B \subseteq set-of B'$
for $A :: 'a::linordered-ring interval$
 $\langle proof \rangle$

lemma *set-of-distrib-left:*

set-of $(B * (A1 + A2)) \subseteq \text{set-of } (B * A1 + B * A2)$
for $A1 :: 'a::\text{linordered-ring interval}$
<proof>

lemma *set-of-distrib-right:*

set-of $((A1 + A2) * B) \subseteq \text{set-of } (A1 * B + A2 * B)$
for $A1\ A2\ B :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$
interval
<proof>

lemma *set-of-mul-inc-left:*

set-of $(A * B) \subseteq \text{set-of } (A' * B)$
if *set-of* $A \subseteq \text{set-of } A'$
for $A :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *set-of-mul-inc:*

set-of $(A * B) \subseteq \text{set-of } (A' * B')$
if *set-of* $A \subseteq \text{set-of } A'$ *set-of* $B \subseteq \text{set-of } B'$
for $A :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *set-of-pow-inc:*

set-of $(A^{\wedge}n) \subseteq \text{set-of } (A'^{\wedge}n)$
if *set-of* $A \subseteq \text{set-of } A'$
for $A :: 'a::\{\text{linordered-idom, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *set-of-distrib-right-left:*

set-of $((A1 + A2) * (B1 + B2)) \subseteq \text{set-of } (A1 * B1 + A1 * B2 + A2 * B1 + A2 * B2)$
for $A1 :: 'a::\{\text{linordered-idom, real-normed-algebra, linear-continuum-topology}\}$ *interval*
<proof>

lemma *mult-bounds-enclose-zero1:*

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$
 $0 \leq \max (la * lb) (\max (la * ub) (\max (lb * ua) (ua * ub)))$
if $la \leq 0\ 0 \leq ua$
for $la\ lb\ ua\ ub:: 'a::\text{linordered-idom}$
<proof>

lemma *mult-bounds-enclose-zero2:*

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$


```

0 ≤ max (la * lb) (max (la * ub) (max (lb * ua) (ua * ub)))
if lb ≤ 0 0 ≤ ub
for la lb ua ub:: 'a::linordered-idom
⟨proof⟩

```

lemma *set-of-mul-contains-zero*:

```

0 ∈ set-of (A * B)
if 0 ∈ set-of A ∨ 0 ∈ set-of B
for A :: 'a::linordered-idom interval
⟨proof⟩

```

instance *interval* :: ($\{\text{linordered-semiring, zero, times}\}$) *mult-zero*

```

⟨proof⟩

```

lift-definition *min-interval*::'a::linorder interval \Rightarrow 'a interval \Rightarrow 'a interval **is**

```

λ(l1, u1). λ(l2, u2). (min l1 l2, min u1 u2)
⟨proof⟩

```

lemma *lower-min-interval[simp]*: lower (min-interval x y) = min (lower x) (lower y)

```

⟨proof⟩

```

lemma *upper-min-interval[simp]*: upper (min-interval x y) = min (upper x) (upper y)

```

⟨proof⟩

```

lemma *min-intervalI*:

```

a ∈i A  $\Rightarrow$  b ∈i B  $\Rightarrow$  min a b ∈i min-interval A B
⟨proof⟩

```

lift-definition *max-interval*::'a::linorder interval \Rightarrow 'a interval \Rightarrow 'a interval **is**

```

λ(l1, u1). λ(l2, u2). (max l1 l2, max u1 u2)
⟨proof⟩

```

lemma *lower-max-interval[simp]*: lower (max-interval x y) = max (lower x) (lower y)

```

⟨proof⟩

```

lemma *upper-max-interval[simp]*: upper (max-interval x y) = max (upper x) (upper y)

```

⟨proof⟩

```

lemma *max-intervalI*:

```

a ∈i A  $\Rightarrow$  b ∈i B  $\Rightarrow$  max a b ∈i max-interval A B
⟨proof⟩

```

lift-definition *abs-interval*::'a::linordered-idom interval \Rightarrow 'a interval **is**

```

(λ(l,u). (if l < 0 ∧ 0 < u then 0 else min |l| |u|, max |l| |u|))
⟨proof⟩

```

lemma *lower-abs-interval[simp]*:

```

lower (abs-interval x) = (if lower x < 0 ∧ 0 < upper x then 0 else min |lower x|
|upper x|)

```

```

    <proof>
lemma upper-abs-interval[simp]: upper (abs-interval x) = max |lower x| |upper x|
    <proof>

lemma in-abs-intervalI1:
  lx < 0  $\implies$  0 < ux  $\implies$  0  $\leq$  xa  $\implies$  xa  $\leq$  max (- lx) (ux)  $\implies$  xa  $\in$  abs ‘ {lx..ux}
  for xa::'a::linordered-idom
  <proof>

lemma in-abs-intervalI2:
  min (|lx|) |ux|  $\leq$  xa  $\implies$  xa  $\leq$  max |lx| |ux|  $\implies$  lx  $\leq$  ux  $\implies$  0  $\leq$  lx  $\vee$  ux  $\leq$  0
 $\implies$ 
  xa  $\in$  abs ‘ {lx..ux}
  for xa::'a::linordered-idom
  <proof>

lemma set-of-abs-interval: set-of (abs-interval x) = abs ‘ set-of x
  <proof>

fun split-domain :: ('a::preorder interval  $\Rightarrow$  'a interval list)  $\Rightarrow$  'a interval list  $\Rightarrow$ 
'a interval list list
  where split-domain split [] = [[]]
  | split-domain split (I#Is) = (
    let S = split I;
    D = split-domain split Is
    in concat (map ( $\lambda$ d. map ( $\lambda$ s. s # d) S) D)
  )

context notes [[typedef-overloaded]] begin
lift-definition(code-dt) split-interval::'a::linorder interval  $\Rightarrow$  'a  $\Rightarrow$  ('a interval  $\times$ 
'a interval)
  is  $\lambda(l, u) x. ((\min l x, \max l x), (\min u x, \max u x))$ 
  <proof>
end

lemma split-domain-nonempty:
  assumes  $\bigwedge I. \text{split } I \neq []$ 
  shows split-domain split I  $\neq []$ 
  <proof>

lemma lower-split-interval1: lower (fst (split-interval X m)) = min (lower X) m
  and lower-split-interval2: lower (snd (split-interval X m)) = min (upper X) m
  and upper-split-interval1: upper (fst (split-interval X m)) = max (lower X) m
  and upper-split-interval2: upper (snd (split-interval X m)) = max (upper X) m
  <proof>

lemma split-intervalD: split-interval X x = (A, B)  $\implies$  set-of X  $\subseteq$  set-of A  $\cup$  set-of
B
  <proof>

```

instantiation *interval* :: (*{topological-space, preorder}*) *topological-space*
begin

definition *open-interval-def*[*code del*]: *open* (*X*::'*a interval set*) =
 ($\forall x \in X.$
 $\exists A B.$
 $\text{open } A \wedge$
 $\text{open } B \wedge$
 $\text{lower } x \in A \wedge \text{upper } x \in B \wedge \text{Interval } (A \times B) \subseteq X$)

instance
<proof>

end

51.2 Quickcheck

lift-definition *Ivl*::'*a* \Rightarrow '*a*::*preorder* \Rightarrow '*a interval* **is** $\lambda a b. (\text{min } a b, b)$
<proof>

instantiation *interval* :: (*{exhaustive,preorder}*) *exhaustive*
begin

definition *exhaustive-interval*::('a *interval* \Rightarrow (*bool* \times *term list*) *option*)
 \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*
where
 $\text{exhaustive-interval } f d =$
 $\text{Quickcheck-Exhaustive.exhaustive } (\lambda x. \text{Quickcheck-Exhaustive.exhaustive } (\lambda y. f$
 $(\text{Ivl } x y)) d) d$

instance *<proof>*

end

context
 includes *term-syntax*
begin

definition [*code-unfold*]:
 $\text{valtermify-interval } x y = \text{Code-Evaluation.valtermify } (\text{Ivl}::'\text{a}::\{\text{preorder}, \text{typerep}\} \Rightarrow -)$
 $\{\cdot\} x \{\cdot\} y$

end

instantiation *interval* :: (*{full-exhaustive,preorder,typerep}*) *full-exhaustive*
begin

definition *full-exhaustive-interval*::

```

('a interval × (unit ⇒ term) ⇒ (bool × term list) option)
  ⇒ natural ⇒ (bool × term list) option where
full-exhaustive-interval f d =
  Quickcheck-Exhaustive.full-exhaustive
    (λx. Quickcheck-Exhaustive.full-exhaustive (λy. f (valtermify-interval x y)) d)
d

```

```
instance ⟨proof⟩
```

```
end
```

```

instantiation interval :: ({random,preorder,typerep}) random
begin

```

```
definition random-interval ::
```

```

  natural
  ⇒ natural × natural
  ⇒ ('a interval × (unit ⇒ term)) × natural × natural where
random-interval i =
  scomp (Quickcheck-Random.random i)
    (λman. scomp (Quickcheck-Random.random i) (λexp. Pair (valtermify-interval
man exp)))

```

```
instance ⟨proof⟩
```

```
end
```

```

lifting-update interval.lifting
lifting-forget interval.lifting

```

```
end
```

52 Approximate Operations on Intervals of Floating Point Numbers

```
theory Interval-Float
```

```
imports
```

```
  Interval
```

```
  Float
```

```
begin
```

```
definition mid :: float interval ⇒ float
```

```
where mid i = (lower i + upper i) * Float 1 (-1)
```

```

lemma mid-in-interval: mid i ∈i i
  ⟨proof⟩

```

```
lemma mid-le: lower i ≤ mid i mid i ≤ upper i
```

<proof>

definition *centered* :: float interval \Rightarrow float interval
where *centered* *i* = *i* - interval-of (mid *i*)

definition *split-float-interval* *x* = *split-interval* *x* ((lower *x* + upper *x*) * Float 1 (-1))

lemma *split-float-intervalD*: *split-float-interval* *X* = (*A*, *B*) \implies set-of *X* \subseteq set-of *A* \cup set-of *B*
<proof>

lemma *split-float-interval-bounds*:
shows

lower-split-float-interval1: lower (fst (split-float-interval *X*)) = lower *X*
and *lower-split-float-interval2*: lower (snd (split-float-interval *X*)) = mid *X*
and *upper-split-float-interval1*: upper (fst (split-float-interval *X*)) = mid *X*
and *upper-split-float-interval2*: upper (snd (split-float-interval *X*)) = upper *X*
<proof>

lemmas *float-round-down-le[intro]* = order-trans[OF float-round-down]
and *float-round-up-ge[intro]* = order-trans[OF - float-round-up]

TODO: many of the lemmas should move to theories Float or Approximation (the latter should be based on type *interval*).

52.1 Intervals with Floating Point Bounds

context includes *interval.lifting* **begin**

lift-definition *round-interval* :: nat \Rightarrow float interval \Rightarrow float interval
is $\lambda p. \lambda(l, u). (\text{float-round-down } p \ l, \text{float-round-up } p \ u)$
<proof>

lemma *lower-round-ivl[simp]*: lower (round-interval *p* *x*) = float-round-down *p* (lower *x*)
<proof>

lemma *upper-round-ivl[simp]*: upper (round-interval *p* *x*) = float-round-up *p* (upper *x*)
<proof>

lemma *round-ivl-correct*: set-of *A* \subseteq set-of (round-interval prec *A*)
<proof>

lift-definition *truncate-ivl* :: nat \Rightarrow real interval \Rightarrow real interval
is $\lambda p. \lambda(l, u). (\text{truncate-down } p \ l, \text{truncate-up } p \ u)$
<proof>

lemma *lower-truncate-ivl[simp]*: lower (truncate-ivl *p* *x*) = truncate-down *p* (lower *x*)

$\langle \text{proof} \rangle$
lemma *upper-truncate-ivl[simp]*: $\text{upper } (\text{truncate-ivl } p \ x) = \text{truncate-up } p \ (\text{upper } x)$
 $\langle \text{proof} \rangle$

lemma *truncate-ivl-correct*: $\text{set-of } A \subseteq \text{set-of } (\text{truncate-ivl } \text{prec } A)$
 $\langle \text{proof} \rangle$

lift-definition *real-interval::float interval* \Rightarrow *real interval*
is $\lambda(l, u). (\text{real-of-float } l, \text{real-of-float } u)$
 $\langle \text{proof} \rangle$

lemma *lower-real-interval[simp]*: $\text{lower } (\text{real-interval } x) = \text{lower } x$
 $\langle \text{proof} \rangle$

lemma *upper-real-interval[simp]*: $\text{upper } (\text{real-interval } x) = \text{upper } x$
 $\langle \text{proof} \rangle$

definition *set-of'* $x = (\text{case } x \text{ of } \text{None} \Rightarrow \text{UNIV} \mid \text{Some } i \Rightarrow \text{set-of } (\text{real-interval } i))$

lemma *real-interval-min-interval[simp]*:
 $\text{real-interval } (\text{min-interval } a \ b) = \text{min-interval } (\text{real-interval } a) \ (\text{real-interval } b)$
 $\langle \text{proof} \rangle$

lemma *real-interval-max-interval[simp]*:
 $\text{real-interval } (\text{max-interval } a \ b) = \text{max-interval } (\text{real-interval } a) \ (\text{real-interval } b)$
 $\langle \text{proof} \rangle$

lemma *in-intervalI*:
 $x \in_i X$ **if** $\text{lower } X \leq x \leq \text{upper } X$
 $\langle \text{proof} \rangle$

abbreviation *in-real-interval* $(\langle \langle \text{notation} = \langle \text{infix } \in_r \rangle \rangle - / \in_r - \rangle \ [51, 51] \ 50)$
where $x \in_r X \equiv x \in_i \text{real-interval } X$

lemma *in-real-intervalI*:
 $x \in_r X$ **if** $\text{lower } X \leq x \leq \text{upper } X$ **for** $x::\text{real}$ **and** $X::\text{float interval}$
 $\langle \text{proof} \rangle$

52.2 intros for *real-interval*

lemma *in-round-intervalI*: $x \in_r A \implies x \in_r (\text{round-interval } \text{prec } A)$
 $\langle \text{proof} \rangle$

lemma *zero-in-float-intervalI*: $0 \in_r 0$
 $\langle \text{proof} \rangle$

lemma *plus-in-float-intervalI*: $a + b \in_r A + B$ **if** $a \in_r A$ $b \in_r B$
 $\langle \text{proof} \rangle$

lemma *minus-in-float-interval* I : $a - b \in_r A - B$ **if** $a \in_r A$ $b \in_r B$
 $\langle \text{proof} \rangle$

lemma *uminus-in-float-interval* I : $-a \in_r -A$ **if** $a \in_r A$
 $\langle \text{proof} \rangle$

lemma *real-interval-times*: $\text{real-interval } (A * B) = \text{real-interval } A * \text{real-interval } B$
 $\langle \text{proof} \rangle$

lemma *times-in-float-interval* I : $a * b \in_r A * B$ **if** $a \in_r A$ $b \in_r B$
 $\langle \text{proof} \rangle$

lemma *real-interval-abs*: $\text{real-interval } (\text{abs-interval } A) = \text{abs-interval } (\text{real-interval } A)$
 $\langle \text{proof} \rangle$

lemma *abs-in-float-interval* I : $\text{abs } a \in_r \text{abs-interval } A$ **if** $a \in_r A$
 $\langle \text{proof} \rangle$

lemma *interval-of* $[\text{intro}, \text{simp}]$: $x \in_r \text{interval-of } x$
 $\langle \text{proof} \rangle$

lemma *split-float-interval-real* D : $\text{split-float-interval } X = (A, B) \implies x \in_r X \implies x \in_r A \vee x \in_r B$
 $\langle \text{proof} \rangle$

52.3 bounds for lists

lemma *lower-Interval*: $\text{lower } (\text{Interval } x) = \text{fst } x$
and *upper-Interval*: $\text{upper } (\text{Interval } x) = \text{snd } x$
if $\text{fst } x \leq \text{snd } x$
 $\langle \text{proof} \rangle$

definition *all-in-i* :: $'a::\text{preorder list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool}$
 $(\text{infix } \langle (\text{all}'\text{-in}_i) \rangle 50)$
where $x \text{ all-in}_i I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_i I ! i))$

definition *all-in* :: $\text{real list} \Rightarrow \text{float interval list} \Rightarrow \text{bool}$
 $(\text{infix } \langle (\text{all}'\text{-in}) \rangle 50)$
where $x \text{ all-in } I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_r I ! i))$

definition *all-subset* :: $'a::\text{order interval list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool}$
 $(\text{infix } \langle (\text{all}'\text{-subset}) \rangle 50)$
where $I \text{ all-subset } J = (\text{length } I = \text{length } J \wedge (\forall i < \text{length } I. \text{set-of } (I!i) \subseteq \text{set-of } (J!i)))$

lemmas $[\text{simp}] = \text{all-in-def all-subset-def}$

lemma *all-subsetD*:

assumes *I all-subset J*

assumes *x all-in I*

shows *x all-in J*

<proof>

lemma *round-interval-mono*: *set-of (round-interval prec X) \subseteq set-of (round-interval prec Y)*

if *set-of X \subseteq set-of Y*

<proof>

lemma *Ivl-simps[simp]*: *lower (Ivl a b) = min a b upper (Ivl a b) = b*

<proof>

lemma *set-of-subset-iff*: *set-of X \subseteq set-of Y \longleftrightarrow lower Y \leq lower X \wedge upper X \leq upper Y*

for *X Y :: 'a :: linorder interval*

<proof>

lemma *set-of-subset-iff'*:

set-of a \subseteq set-of (b :: 'a :: linorder interval) \longleftrightarrow a \leq b

<proof>

lemma *bounds-of-interval-eq-lower-upper*:

bounds-of-interval ivl = (lower ivl, upper ivl) if lower ivl \leq upper ivl

<proof>

lemma *real-interval-Ivl*: *real-interval (Ivl a b) = Ivl a b*

<proof>

lemma *set-of-mul-contains-real-zero*:

*0 \in_r (A * B) if 0 \in_r A \vee 0 \in_r B*

<proof>

fun *subdivide-interval* :: *nat \Rightarrow float interval \Rightarrow float interval list*

where *subdivide-interval 0 I = [I]*

| *subdivide-interval (Suc n) I = (*

let m = mid I

in (subdivide-interval n (Ivl (lower I) m)) @ (subdivide-interval n (Ivl m

(upper I)))

)

lemma *subdivide-interval-length*:

shows *length (subdivide-interval n I) = 2ⁿ*

<proof>

lemma *lower-le-mid*: *lower x \leq mid x real-of-float (lower x) \leq mid x*

and *mid-le-upper*: *mid x \leq upper x real-of-float (mid x) \leq upper x*

<proof>

lemma *subdivide-interval-correct:*

list-ex ($\lambda i. x \in_r i$) (*subdivide-interval* n I) **if** $x \in_r I$ **for** $x::real$
<proof>

fun *interval-list-union* :: $'a::lattice$ *interval list* \Rightarrow $'a$ *interval*

where *interval-list-union* [] = *undefined*
 | *interval-list-union* [I] = I
 | *interval-list-union* (I#Is) = *sup* I (*interval-list-union* Is)

lemma *interval-list-union-correct:*

assumes $S \neq []$
assumes $i < \text{length } S$
shows *set-of* ($S!i$) \subseteq *set-of* (*interval-list-union* S)
<proof>

lemma *split-domain-correct:*

fixes $x :: real$ *list*
assumes x *all-in* I
assumes *split-correct*: $\bigwedge x a I. x \in_r I \implies \text{list-ex } (\lambda i::float \text{ interval}. x \in_r i) (\text{split } I)$
shows *list-ex* ($\lambda s. x$ *all-in* s) (*split-domain split* I)
<proof>

lift-definition(*code-dt*) *inverse-float-interval*:: $nat \Rightarrow float$ *interval* \Rightarrow $float$ *interval*
option is

$\lambda prec (l, u). \text{ if } (0 < l \vee u < 0) \text{ then Some } (float-divl \text{ prec } 1 \ u, float-divr \text{ prec } 1 \ l) \text{ else None}$
<proof>

lemma *inverse-float-interval-eq-Some-conv:*

defines $one \equiv (1::float)$
shows
 $\text{inverse-float-interval } p \ X = \text{Some } R \longleftrightarrow$
 $(\text{lower } X > 0 \vee \text{upper } X < 0) \wedge$
 $\text{lower } R = \text{float-divl } p \ one \ (\text{upper } X) \wedge$
 $\text{upper } R = \text{float-divr } p \ one \ (\text{lower } X)$
<proof>

lemma *inverse-float-interval:*

inverse ‘ *set-of* (*real-interval* X) \subseteq *set-of* (*real-interval* Y)
if *inverse-float-interval* p X = *Some* Y
<proof>

lemma *inverse-float-intervalI:*

$x \in_r X \implies \text{inverse } x \in \text{set-of}' (\text{inverse-float-interval } p \ X)$
<proof>

lemma *inverse-float-interval-eqI*: *inverse-float-interval* p $X = \text{Some } IVL \implies x \in_r X \implies \text{inverse } x \in_r IVL$
 ⟨proof⟩

lemma *real-interval-abs-interval[simp]*:
real-interval (*abs-interval* x) = *abs-interval* (*real-interval* x)
 ⟨proof⟩

lift-definition *floor-float-interval::float interval \Rightarrow float interval* **is**
 $\lambda(l, u). (\text{floor-fl } l, \text{floor-fl } u)$
 ⟨proof⟩

lemma *lower-floor-float-interval[simp]*: *lower* (*floor-float-interval* x) = *floor-fl* (*lower* x)
 ⟨proof⟩

lemma *upper-floor-float-interval[simp]*: *upper* (*floor-float-interval* x) = *floor-fl* (*upper* x)
 ⟨proof⟩

lemma *floor-float-intervalI*: $\lfloor x \rfloor \in_r \text{floor-float-interval } X$ **if** $x \in_r X$
 ⟨proof⟩

end

52.4 constants for code generation

definition *lowerF::float interval \Rightarrow float* **where** *lowerF* = *lower*
definition *upperF::float interval \Rightarrow float* **where** *upperF* = *upper*

end

53 Immutable Arrays with Code Generation

theory *IArray*
imports *Main*
begin

53.1 Fundamental operations

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Could be extended to other target languages and more operations.

context
begin

datatype *'a iarray* = *IArray 'a list*

qualified primrec *list-of* :: *'a iarray* \Rightarrow *'a list* **where**
list-of (*IArray xs*) = *xs*

qualified definition *of-fun* :: (*nat* \Rightarrow *'a*) \Rightarrow *nat* \Rightarrow *'a iarray* **where**
[simp]: *of-fun f n* = *IArray (map f [0..*n*])*

qualified definition *sub* :: *'a iarray* \Rightarrow *nat* \Rightarrow *'a* (**infixl** $\langle !! \rangle$ 100) **where**
[simp]: *as !! n* = *IArray.list-of as ! n*

qualified definition *length* :: *'a iarray* \Rightarrow *nat* **where**
[simp]: *length as* = *List.length (IArray.list-of as)*

qualified definition *all* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a iarray* \Rightarrow *bool* **where**
[simp]: *all p as* \longleftrightarrow ($\forall a \in \text{set } (\text{list-of } as). p a$)

qualified definition *exists* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a iarray* \Rightarrow *bool* **where**
[simp]: *exists p as* \longleftrightarrow ($\exists a \in \text{set } (\text{list-of } as). p a$)

lemma *of-fun-nth*:
IArray.of-fun f n !! i = *f i* **if** *i* < *n*
 $\langle \text{proof} \rangle$

end

53.2 Generic code equations

lemma [*code*]:
size (as :: 'a iarray) = *Suc (IArray.length as)*
 $\langle \text{proof} \rangle$

lemma [*code*]:
size-iarray f as = *Suc (size-list f (IArray.list-of as))*
 $\langle \text{proof} \rangle$

lemma [*code*]:
rec-iarray f as = *f (IArray.list-of as)*
 $\langle \text{proof} \rangle$

lemma [*code*]:
case-iarray f as = *f (IArray.list-of as)*
 $\langle \text{proof} \rangle$

lemma [*code*]:
set-iarray as = *set (IArray.list-of as)*
 $\langle \text{proof} \rangle$

lemma [*code*]:

map-iarray f $as = IArray$ (*map* f (*IArray.list-of* as))
 ⟨*proof*⟩

lemma [*code*]:
rel-iarray r as $bs = list-all2$ r (*IArray.list-of* as) (*IArray.list-of* bs)
 ⟨*proof*⟩

lemma *list-of-code* [*code*]:
IArray.list-of $as = map$ ($\lambda n. as !! n$) [$0 ..< IArray.length$ as]
 ⟨*proof*⟩

lemma [*code*]:
HOL.equal as $bs \longleftrightarrow HOL.equal$ (*IArray.list-of* as) (*IArray.list-of* bs)
 ⟨*proof*⟩

lemma [*code*]:
IArray.all $p = Not \circ IArray.exists$ ($Not \circ p$)
 ⟨*proof*⟩

context
includes *term-syntax*
begin

lemma [*code*]:
Code-Evaluation.term-of ($as :: 'a::typerep$ *iarray*) =
Code-Evaluation.Const (*STR* "*IArray.iarray.IArray*") (*TYPEREP* ('*a list* \Rightarrow '*a*
iarray)) $<\cdot>$ (*Code-Evaluation.term-of* (*IArray.list-of* as))
 ⟨*proof*⟩

end

53.3 Auxiliary operations for code generation

context
begin

qualified primrec *tabulate* :: *integer* \times (*integer* \Rightarrow '*a*) \Rightarrow '*a iarray* **where**
tabulate (n, f) = *IArray* (*map* ($f \circ integer-of-nat$) [$0..<nat-of-integer$ n])

lemma [*code*]:
IArray.of-fun f $n = IArray.tabulate$ (*integer-of-nat* $n, f \circ nat-of-integer$)
 ⟨*proof*⟩ **primrec** *sub'* :: '*a iarray* \times *integer* \Rightarrow '*a* **where**
sub' (as, n) = $as !! nat-of-integer$ n

lemma [*code*]:
IArray.sub' (*IArray* as, n) = $as ! nat-of-integer$ n
 ⟨*proof*⟩

lemma [*code*]:

```

as !! n = IArray.sub' (as, integer-of-nat n)
⟨proof⟩ definition length' :: 'a iarray ⇒ integer where
[simp]: length' as = integer-of-nat (List.length (IArray.list-of as))

lemma [code]:
  IArray.length' (IArray as) = integer-of-nat (List.length as)
  ⟨proof⟩

lemma [code]:
  IArray.length as = nat-of-integer (IArray.length' as)
  ⟨proof⟩ definition exists-upto :: ('a ⇒ bool) ⇒ integer ⇒ 'a iarray ⇒ bool where
[simp]: exists-upto p k as ⇔ (∃ l. 0 ≤ l ∧ l < k ∧ p (sub' (as, l)))

lemma exists-upto-of-nat:
  exists-upto p (of-nat n) as ⇔ (∃ m < n. p (as !! m))
  including integer.lifting ⟨proof⟩

lemma [code]:
  exists-upto p k as ⇔ (if k ≤ 0 then False else
    let l = k - 1 in p (sub' (as, l)) ∨ exists-upto p l as)
  ⟨proof⟩
  including integer.lifting ⟨proof⟩

lemma [code]:
  IArray.exists p as ⇔ exists-upto p (length' as) as
  including integer.lifting ⟨proof⟩

end

```

53.4 Code Generation for SML

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

code-reserved (SML) *Vector*

code-printing

```

type-constructor iarray → (SML) - Vector.vector
| constant IArray → (SML) Vector.fromList
| constant IArray.all → (SML) Vector.all
| constant IArray.exists → (SML) Vector.exists
| constant IArray.tabulate → (SML) Vector.tabulate
| constant IArray.sub' → (SML) Vector.sub
| constant IArray.length' → (SML) Vector.length

```

53.5 Code Generation for Haskell

We map 'a iarrays in Isabelle/HOL to *Data.Array.IArray.array* in Haskell. Performance mapping to *Data.Array.Unboxed.Array* and *Data.Array.Array*

is similar.

code-printing

```
code-module IArray  $\rightarrow$  (Haskell)  $\langle$ 
module IArray(IArray, tabulate, of-list, sub, length) where {

  import Prelude (Bool(True, False), not, Maybe(Nothing, Just),
    Integer, (+), (-), (<), fromInteger, toInteger, map, seq, (..));
  import qualified Prelude;
  import qualified Data.Array.IArray;
  import qualified Data.Array.Base;
  import qualified Data.Ix;

  newtype IArray e = IArray (Data.Array.IArray.Array Integer e);

  tabulate :: (Integer, (Integer  $\rightarrow$  e))  $\rightarrow$  IArray e;
  tabulate (k, f) = IArray (Data.Array.IArray.array (0, k - 1) (map (\i  $\rightarrow$  let
    fi = f i in fi 'seq' (i, fi)) [0..k - 1]));

  of-list :: [e]  $\rightarrow$  IArray e;
  of-list l = IArray (Data.Array.IArray.listArray (0, (toInteger . Prelude.length) l
    - 1) l);

  sub :: (IArray e, Integer)  $\rightarrow$  e;
  sub (IArray v, i) = v 'Data.Array.Base.unsafeAt' fromInteger i;

  length :: IArray e  $\rightarrow$  Integer;
  length (IArray v) = toInteger (Data.Ix.rangeSize (Data.Array.IArray.bounds v));

} for type-constructor iarray constant IArray IArray.tabulate IArray.sub' IAr-
ray.length'
```

code-reserved (Haskell) IArray-Impl

code-printing

```
type-constructor iarray  $\rightarrow$  (Haskell) IArray.IArray -
| constant IArray  $\rightarrow$  (Haskell) IArray.of'-list
| constant IArray.tabulate  $\rightarrow$  (Haskell) IArray.tabulate
| constant IArray.sub'  $\rightarrow$  (Haskell) IArray.sub
| constant IArray.length'  $\rightarrow$  (Haskell) IArray.length
```

end

54 Definition of Landau symbols

theory Landau-Symbols

imports

Complex-Main

begin

lemma *eventually-subst'*:

eventually $(\lambda x. f\ x = g\ x)\ F \implies \text{eventually } (\lambda x. P\ x\ (f\ x))\ F = \text{eventually } (\lambda x. P\ x\ (g\ x))\ F$
<proof>

54.1 Definition of Landau symbols

Our Landau symbols are sign-oblivious, i.e. any function always has the same growth as its absolute. This has the advantage of making some cancelling rules for sums nicer, but introduces some problems in other places. Nevertheless, we found this definition more convenient to work with.

definition *bigo* :: 'a filter \Rightarrow ('a \Rightarrow ('b :: real-normed-field)) \Rightarrow ('a \Rightarrow 'b) set
 ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix bigo} \rangle O[-]'(-) \rangle \rangle$)
where *bigo* $F\ g = \{f. (\exists c>0. \text{eventually } (\lambda x. \text{norm } (f\ x) \leq c * \text{norm } (g\ x))\ F)\}$

definition *smallo* :: 'a filter \Rightarrow ('a \Rightarrow ('b :: real-normed-field)) \Rightarrow ('a \Rightarrow 'b) set
 ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix smallo} \rangle o[-]'(-) \rangle \rangle$)
where *smallo* $F\ g = \{f. (\forall c>0. \text{eventually } (\lambda x. \text{norm } (f\ x) \leq c * \text{norm } (g\ x))\ F)\}$

definition *bigomega* :: 'a filter \Rightarrow ('a \Rightarrow ('b :: real-normed-field)) \Rightarrow ('a \Rightarrow 'b) set
 ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix bigomega} \rangle \Omega[-]'(-) \rangle \rangle$)
where *bigomega* $F\ g = \{f. (\exists c>0. \text{eventually } (\lambda x. \text{norm } (f\ x) \geq c * \text{norm } (g\ x))\ F)\}$

definition *smallomega* :: 'a filter \Rightarrow ('a \Rightarrow ('b :: real-normed-field)) \Rightarrow ('a \Rightarrow 'b) set
 ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix smallomega} \rangle \omega[-]'(-) \rangle \rangle$)
where *smallomega* $F\ g = \{f. (\forall c>0. \text{eventually } (\lambda x. \text{norm } (f\ x) \geq c * \text{norm } (g\ x))\ F)\}$

definition *bigtheta* :: 'a filter \Rightarrow ('a \Rightarrow ('b :: real-normed-field)) \Rightarrow ('a \Rightarrow 'b) set
 ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix bigtheta} \rangle \Theta[-]'(-) \rangle \rangle$)
where *bigtheta* $F\ g = \text{bigo } F\ g \cap \text{bigomega } F\ g$

abbreviation *bigo-at-top* ($\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigo} \rangle O'(-) \rangle \rangle$)
where $O(g) \equiv \text{bigo at-top } g$

abbreviation *smallo-at-top* ($\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix smallo} \rangle o'(-) \rangle \rangle$)
where $o(g) \equiv \text{smallo at-top } g$

abbreviation *bigomega-at-top* ($\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigomega} \rangle \Omega'(-) \rangle \rangle$)
where $\Omega(g) \equiv \text{bigomega at-top } g$

abbreviation *smallomega-at-top* ($\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix smallomega} \rangle \omega'(-) \rangle \rangle$)
where $\omega(g) \equiv \text{smallomega at-top } g$

abbreviation *bigtheta-at-top* ($\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigtheta} \rangle \Theta'(-) \rangle \rangle$)

where $\Theta(g) \equiv \text{bigtheta at-top } g$

The following is a set of properties that all Landau symbols satisfy.

named-theorems *landau-divide-simps*

locale *landau-symbol* =

fixes $L :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$

and $L' :: 'c \text{ filter} \Rightarrow ('c \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('c \Rightarrow 'b) \text{ set}$

and $Lr :: 'a \text{ filter} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real}) \text{ set}$

assumes *bot'*: $L \text{ bot } f = \text{UNIV}$

assumes *filter-mono'*: $F1 \leq F2 \Longrightarrow L F2 f \subseteq L F1 f$

assumes *in-filtermap-iff*:

$f' \in L (\text{filtermap } h' F') g' \longleftrightarrow (\lambda x. f' (h' x)) \in L' F' (\lambda x. g' (h' x))$

assumes *filtercomap*:

$f' \in L F'' g' \Longrightarrow (\lambda x. f' (h' x)) \in L' (\text{filtercomap } h' F'') (\lambda x. g' (h' x))$

assumes *sup*: $f \in L F1 g \Longrightarrow f \in L F2 g \Longrightarrow f \in L (\text{sup } F1 F2) g$

assumes *in-cong*: $\text{eventually } (\lambda x. f x = g x) F \Longrightarrow f \in L F (h) \longleftrightarrow g \in L F (h)$

assumes *cong*: $\text{eventually } (\lambda x. f x = g x) F \Longrightarrow L F (f) = L F (g)$

assumes *cong-bigtheta*: $f \in \Theta[F](g) \Longrightarrow L F (f) = L F (g)$

assumes *in-cong-bigtheta*: $f \in \Theta[F](g) \Longrightarrow f \in L F (h) \longleftrightarrow g \in L F (h)$

assumes *cmult [simp]*: $c \neq 0 \Longrightarrow L F (\lambda x. c * f x) = L F (f)$

assumes *cmult-in-iff [simp]*: $c \neq 0 \Longrightarrow (\lambda x. c * f x) \in L F (g) \longleftrightarrow f \in L F (g)$

assumes *mult-left [simp]*: $f \in L F (g) \Longrightarrow (\lambda x. h x * f x) \in L F (\lambda x. h x * g x)$

assumes *inverse*: $\text{eventually } (\lambda x. f x \neq 0) F \Longrightarrow \text{eventually } (\lambda x. g x \neq 0) F$

\Longrightarrow

$f \in L F (g) \Longrightarrow (\lambda x. \text{inverse } (g x)) \in L F (\lambda x. \text{inverse } (f x))$

assumes *subsetI*: $f \in L F (g) \Longrightarrow L F (f) \subseteq L F (g)$

assumes *plus-subset1*: $f \in o[F](g) \Longrightarrow L F (g) \subseteq L F (\lambda x. f x + g x)$

assumes *trans*: $f \in L F (g) \Longrightarrow g \in L F (h) \Longrightarrow f \in L F (h)$

assumes *compose*: $f \in L F (g) \Longrightarrow \text{filterlim } h' F G \Longrightarrow (\lambda x. f (h' x)) \in L' G$

$(\lambda x. g (h' x))$

assumes *norm-iff [simp]*: $(\lambda x. \text{norm } (f x)) \in Lr F (\lambda x. \text{norm } (g x)) \longleftrightarrow f \in L$

$F (g)$

assumes *abs [simp]*: $Lr Fr (\lambda x. |fr x|) = Lr Fr fr$

assumes *abs-in-iff [simp]*: $(\lambda x. |fr x|) \in Lr Fr gr \longleftrightarrow fr \in Lr Fr gr$

begin

lemma *bot [simp]*: $f \in L \text{ bot } g \langle \text{proof} \rangle$

lemma *filter-mono*: $F1 \leq F2 \Longrightarrow f \in L F2 g \Longrightarrow f \in L F1 g$

$\langle \text{proof} \rangle$

lemma *cong-ex*:

$\text{eventually } (\lambda x. f1 x = f2 x) F \Longrightarrow \text{eventually } (\lambda x. g1 x = g2 x) F \Longrightarrow$

$f1 \in L F (g1) \longleftrightarrow f2 \in L F (g2)$

$\langle \text{proof} \rangle$

lemma *cong-ex-bigtheta*:

$$f1 \in \Theta[F](f2) \implies g1 \in \Theta[F](g2) \implies f1 \in L F (g1) \longleftrightarrow f2 \in L F (g2)$$

<proof>

lemma *bigheta-trans1*:

$$f \in L F (g) \implies g \in \Theta[F](h) \implies f \in L F (h)$$

<proof>

lemma *bigheta-trans2*:

$$f \in \Theta[F](g) \implies g \in L F (h) \implies f \in L F (h)$$

<proof>

lemma *cmult'* [simp]: $c \neq 0 \implies L F (\lambda x. f x * c) = L F (f)$

<proof>

lemma *cmult-in-iff'* [simp]: $c \neq 0 \implies (\lambda x. f x * c) \in L F (g) \longleftrightarrow f \in L F (g)$

<proof>

lemma *cdiv* [simp]: $c \neq 0 \implies L F (\lambda x. f x / c) = L F (f)$

<proof>

lemma *cdiv-in-iff'* [simp]: $c \neq 0 \implies (\lambda x. f x / c) \in L F (g) \longleftrightarrow f \in L F (g)$

<proof>

lemma *uminus* [simp]: $L F (\lambda x. -g x) = L F (g)$ *<proof>*

lemma *uminus-in-iff'* [simp]: $(\lambda x. -f x) \in L F (g) \longleftrightarrow f \in L F (g)$

<proof>

lemma *const*: $c \neq 0 \implies L F (\lambda-. c) = L F (\lambda-. 1)$

<proof>

lemma *const'* [simp]: *NO-MATCH* $1 c \implies c \neq 0 \implies L F (\lambda-. c) = L F (\lambda-. 1)$

<proof>

lemma *const-in-iff'*: $c \neq 0 \implies (\lambda-. c) \in L F (f) \longleftrightarrow (\lambda-. 1) \in L F (f)$

<proof>

lemma *const-in-iff'* [simp]: *NO-MATCH* $1 c \implies c \neq 0 \implies (\lambda-. c) \in L F (f) \longleftrightarrow (\lambda-. 1) \in L F (f)$

<proof>

lemma *plus-subset2*: $g \in o[F](f) \implies L F (f) \subseteq L F (\lambda x. f x + g x)$

<proof>

lemma *mult-right* [simp]: $f \in L F (g) \implies (\lambda x. f x * h x) \in L F (\lambda x. g x * h x)$

<proof>

lemma *mult*: $f1 \in L F (g1) \implies f2 \in L F (g2) \implies (\lambda x. f1 x * f2 x) \in L F (\lambda x.$

$g1\ x * g2\ x)$
 $\langle proof \rangle$

lemma *inverse-cancel*:

assumes *eventually* $(\lambda x. f\ x \neq 0)\ F$
assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
shows $(\lambda x. \text{inverse}\ (f\ x)) \in L\ F\ (\lambda x. \text{inverse}\ (g\ x)) \longleftrightarrow g \in L\ F\ (f)$
 $\langle proof \rangle$

lemma *divide-right*:

assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$
assumes $f \in L\ F\ (g)$
shows $(\lambda x. f\ x / h\ x) \in L\ F\ (\lambda x. g\ x / h\ x)$
 $\langle proof \rangle$

lemma *divide-right-iff*:

assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$
shows $(\lambda x. f\ x / h\ x) \in L\ F\ (\lambda x. g\ x / h\ x) \longleftrightarrow f \in L\ F\ (g)$
 $\langle proof \rangle$

lemma *divide-left*:

assumes *eventually* $(\lambda x. f\ x \neq 0)\ F$
assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
assumes $g \in L\ F\ (f)$
shows $(\lambda x. h\ x / f\ x) \in L\ F\ (\lambda x. h\ x / g\ x)$
 $\langle proof \rangle$

lemma *divide-left-iff*:

assumes *eventually* $(\lambda x. f\ x \neq 0)\ F$
assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$
shows $(\lambda x. h\ x / f\ x) \in L\ F\ (\lambda x. h\ x / g\ x) \longleftrightarrow g \in L\ F\ (f)$
 $\langle proof \rangle$

lemma *divide*:

assumes *eventually* $(\lambda x. g1\ x \neq 0)\ F$
assumes *eventually* $(\lambda x. g2\ x \neq 0)\ F$
assumes $f1 \in L\ F\ (f2)\ g2 \in L\ F\ (g1)$
shows $(\lambda x. f1\ x / g1\ x) \in L\ F\ (\lambda x. f2\ x / g2\ x)$
 $\langle proof \rangle$

lemma *divide-eq1*:

assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$
shows $f \in L\ F\ (\lambda x. g\ x / h\ x) \longleftrightarrow (\lambda x. f\ x * h\ x) \in L\ F\ (g)$
 $\langle proof \rangle$

lemma *divide-eq2*:

assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$
shows $(\lambda x. f\ x / h\ x) \in L\ F\ (\lambda x. g\ x) \longleftrightarrow f \in L\ F\ (\lambda x. g\ x * h\ x)$

$\langle proof \rangle$

lemma *inverse-eq1*:

assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$

shows $f \in L\ F\ (\lambda x. \text{inverse}\ (g\ x)) \longleftrightarrow (\lambda x. f\ x * g\ x) \in L\ F\ (\lambda -. 1)$

$\langle proof \rangle$

lemma *inverse-eq2*:

assumes *eventually* $(\lambda x. f\ x \neq 0)\ F$

shows $(\lambda x. \text{inverse}\ (f\ x)) \in L\ F\ (g) \longleftrightarrow (\lambda x. 1) \in L\ F\ (\lambda x. f\ x * g\ x)$

$\langle proof \rangle$

lemma *inverse-flip*:

assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$

assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$

assumes $(\lambda x. \text{inverse}\ (g\ x)) \in L\ F\ (h)$

shows $(\lambda x. \text{inverse}\ (h\ x)) \in L\ F\ (g)$

$\langle proof \rangle$

lemma *lift-trans*:

assumes $f \in L\ F\ (g)$

assumes $(\lambda x. t\ x\ (g\ x)) \in L\ F\ (h)$

assumes $\bigwedge f\ g. f \in L\ F\ (g) \implies (\lambda x. t\ x\ (f\ x)) \in L\ F\ (\lambda x. t\ x\ (g\ x))$

shows $(\lambda x. t\ x\ (f\ x)) \in L\ F\ (h)$

$\langle proof \rangle$

lemma *lift-trans'*:

assumes $f \in L\ F\ (\lambda x. t\ x\ (g\ x))$

assumes $g \in L\ F\ (h)$

assumes $\bigwedge g\ h. g \in L\ F\ (h) \implies (\lambda x. t\ x\ (g\ x)) \in L\ F\ (\lambda x. t\ x\ (h\ x))$

shows $f \in L\ F\ (\lambda x. t\ x\ (h\ x))$

$\langle proof \rangle$

lemma *lift-trans-bigtheta*:

assumes $f \in L\ F\ (g)$

assumes $(\lambda x. t\ x\ (g\ x)) \in \Theta[F](h)$

assumes $\bigwedge f\ g. f \in L\ F\ (g) \implies (\lambda x. t\ x\ (f\ x)) \in L\ F\ (\lambda x. t\ x\ (g\ x))$

shows $(\lambda x. t\ x\ (f\ x)) \in L\ F\ (h)$

$\langle proof \rangle$

lemma *lift-trans-bigtheta'*:

assumes $f \in L\ F\ (\lambda x. t\ x\ (g\ x))$

assumes $g \in \Theta[F](h)$

assumes $\bigwedge g\ h. g \in \Theta[F](h) \implies (\lambda x. t\ x\ (g\ x)) \in \Theta[F](\lambda x. t\ x\ (h\ x))$

shows $f \in L\ F\ (\lambda x. t\ x\ (h\ x))$

$\langle proof \rangle$

lemma (in *landau-symbol*) *mult-in-1*:

assumes $f \in L\ F\ (\lambda -. 1)\ g \in L\ F\ (\lambda -. 1)$

shows $(\lambda x. f\ x * g\ x) \in L\ F\ (\lambda x. 1)$
 $\langle proof \rangle$

lemma (in *landau-symbol*) *of-real-cancel*:
 $(\lambda x. of\text{-}real\ (f\ x)) \in L\ F\ (\lambda x. of\text{-}real\ (g\ x)) \implies f \in Lr\ F\ g$
 $\langle proof \rangle$

lemma (in *landau-symbol*) *of-real-iff*:
 $(\lambda x. of\text{-}real\ (f\ x)) \in L\ F\ (\lambda x. of\text{-}real\ (g\ x)) \longleftrightarrow f \in Lr\ F\ g$
 $\langle proof \rangle$

lemmas [*landau-divide-simps*] =
inverse-cancel divide-left-iff divide-eq1 divide-eq2 inverse-eq1 inverse-eq2

end

The symbols O and o and Ω and ω are dual, so for many rules, replacing O with Ω , o with ω , and \leq with \geq in a theorem yields another valid theorem. The following locale captures this fact.

locale *landau-pair* =
fixes $L\ l :: 'a\ filter \Rightarrow ('a \Rightarrow ('b :: real\text{-}normed\text{-}field)) \Rightarrow ('a \Rightarrow 'b)\ set$
fixes $L'\ l' :: 'c\ filter \Rightarrow ('c \Rightarrow ('b :: real\text{-}normed\text{-}field)) \Rightarrow ('c \Rightarrow 'b)\ set$
fixes $Lr\ lr :: 'a\ filter \Rightarrow ('a \Rightarrow real) \Rightarrow ('a \Rightarrow real)\ set$
and $R :: real \Rightarrow real \Rightarrow bool$
assumes $L\text{-}def: L\ F\ g = \{f. \exists c>0. eventually\ (\lambda x. R\ (norm\ (f\ x))\ (c * norm\ (g\ x)))\ F\}$
and $l\text{-}def: l\ F\ g = \{f. \forall c>0. eventually\ (\lambda x. R\ (norm\ (f\ x))\ (c * norm\ (g\ x)))\ F\}$
and $L'\text{-}def: L'\ F'\ g' = \{f. \exists c>0. eventually\ (\lambda x. R\ (norm\ (f\ x))\ (c * norm\ (g'\ x)))\ F'\}$
and $l'\text{-}def: l'\ F'\ g' = \{f. \forall c>0. eventually\ (\lambda x. R\ (norm\ (f\ x))\ (c * norm\ (g'\ x)))\ F'\}$
and $Lr\text{-}def: Lr\ F''\ g'' = \{f. \exists c>0. eventually\ (\lambda x. R\ (norm\ (f\ x))\ (c * norm\ (g''\ x)))\ F''\}$
and $lr\text{-}def: lr\ F''\ g'' = \{f. \forall c>0. eventually\ (\lambda x. R\ (norm\ (f\ x))\ (c * norm\ (g''\ x)))\ F''\}$
and $R: R = (\leq) \vee R = (\geq)$

interpretation *landau-o*:
landau-pair bigo smallo bigo smallo bigo smallo (\leq)
 $\langle proof \rangle$

interpretation *landau-omega*:
landau-pair bigomega smallomega bigomega smallomega bigomega smallomega
 (\geq)
 $\langle proof \rangle$

context *landau-pair*

begin

lemmas $R\text{-}E = \text{disj}E \text{ [} OF \text{ } R, \text{ case-names } le \text{ } ge \text{]}$

lemma *bigI*:

$c > 0 \implies \text{eventually } (\lambda x. R \text{ (norm } (f \text{ } x)) \text{ (} c * \text{norm } (g \text{ } x))) \text{ } F \implies f \in L \text{ } F \text{ (} g \text{)}$
 $\langle \text{proof} \rangle$

lemma *bigE*:

assumes $f \in L \text{ } F \text{ (} g \text{)}$

obtains c **where** $c > 0$ $\text{eventually } (\lambda x. R \text{ (norm } (f \text{ } x)) \text{ (} c * \text{norm } (g \text{ } x))) \text{ } F$
 $\langle \text{proof} \rangle$

lemma *smallI*:

$(\bigwedge c. c > 0 \implies \text{eventually } (\lambda x. R \text{ (norm } (f \text{ } x)) \text{ (} c * \text{norm } (g \text{ } x)))) \text{ } F \implies f \in$
 $l \text{ } F \text{ (} g \text{)}$
 $\langle \text{proof} \rangle$

lemma *smallD*:

$f \in l \text{ } F \text{ (} g \text{)} \implies c > 0 \implies \text{eventually } (\lambda x. R \text{ (norm } (f \text{ } x)) \text{ (} c * \text{norm } (g \text{ } x))) \text{ } F$
 $\langle \text{proof} \rangle$

lemma *bigE-nonneg-real*:

assumes $f \in Lr \text{ } F \text{ (} g \text{)}$ $\text{eventually } (\lambda x. f \text{ } x \geq 0) \text{ } F$

obtains c **where** $c > 0$ $\text{eventually } (\lambda x. R \text{ (} f \text{ } x \text{) (} c * |g \text{ } x|)) \text{ } F$
 $\langle \text{proof} \rangle$

lemma *smallD-nonneg-real*:

assumes $f \in lr \text{ } F \text{ (} g \text{)}$ $\text{eventually } (\lambda x. f \text{ } x \geq 0) \text{ } F$ $c > 0$

shows $\text{eventually } (\lambda x. R \text{ (} f \text{ } x \text{) (} c * |g \text{ } x|)) \text{ } F$
 $\langle \text{proof} \rangle$

lemma *small-imp-big*: $f \in l \text{ } F \text{ (} g \text{)} \implies f \in L \text{ } F \text{ (} g \text{)}$

$\langle \text{proof} \rangle$

lemma *small-subset-big*: $l \text{ } F \text{ (} g \text{)} \subseteq L \text{ } F \text{ (} g \text{)}$

$\langle \text{proof} \rangle$

lemma *R-refl* [*simp*]: $R \text{ } x \text{ } x$ $\langle \text{proof} \rangle$

lemma *R-linear*: $\neg R \text{ } x \text{ } y \implies R \text{ } y \text{ } x$

$\langle \text{proof} \rangle$

lemma *R-trans* [*trans*]: $R \text{ } a \text{ } b \implies R \text{ } b \text{ } c \implies R \text{ } a \text{ } c$

$\langle \text{proof} \rangle$

lemma *R-mult-left-mono*: $R \text{ } a \text{ } b \implies c \geq 0 \implies R \text{ (} c * a \text{) (} c * b \text{)}$

$\langle \text{proof} \rangle$

lemma *R-mult-right-mono*: $R\ a\ b \implies c \geq 0 \implies R\ (a*c)\ (b*c)$
 $\langle proof \rangle$

lemma *big-trans*:
assumes $f \in L\ F\ (g)\ g \in L\ F\ (h)$
shows $f \in L\ F\ (h)$
 $\langle proof \rangle$

lemma *big-small-trans*:
assumes $f \in L\ F\ (g)\ g \in l\ F\ (h)$
shows $f \in l\ F\ (h)$
 $\langle proof \rangle$

lemma *small-big-trans*:
assumes $f \in l\ F\ (g)\ g \in L\ F\ (h)$
shows $f \in l\ F\ (h)$
 $\langle proof \rangle$

lemma *small-trans*:
 $f \in l\ F\ (g) \implies g \in l\ F\ (h) \implies f \in l\ F\ (h)$
 $\langle proof \rangle$

lemma *small-big-trans'*:
 $f \in l\ F\ (g) \implies g \in L\ F\ (h) \implies f \in L\ F\ (h)$
 $\langle proof \rangle$

lemma *big-small-trans'*:
 $f \in L\ F\ (g) \implies g \in l\ F\ (h) \implies f \in L\ F\ (h)$
 $\langle proof \rangle$

lemma *big-subsetI [intro]*: $f \in L\ F\ (g) \implies L\ F\ (f) \subseteq L\ F\ (g)$
 $\langle proof \rangle$

lemma *small-subsetI [intro]*: $f \in L\ F\ (g) \implies l\ F\ (f) \subseteq l\ F\ (g)$
 $\langle proof \rangle$

lemma *big-refl [simp]*: $f \in L\ F\ (f)$
 $\langle proof \rangle$

lemma *small-refl-iff*: $f \in l\ F\ (f) \longleftrightarrow eventually\ (\lambda x. f\ x = 0)\ F$
 $\langle proof \rangle$

lemma *big-small-asymmetric*: $f \in L\ F\ (g) \implies g \in l\ F\ (f) \implies eventually\ (\lambda x. f\ x = 0)\ F$
 $\langle proof \rangle$

lemma *small-big-asymmetric*: $f \in l\ F\ (g) \implies g \in L\ F\ (f) \implies eventually\ (\lambda x. f\ x = 0)\ F$
 $\langle proof \rangle$

lemma *small-asymmetric*: $f \in l\ F\ (g) \implies g \in l\ F\ (f) \implies \text{eventually } (\lambda x. f\ x = 0) \ F$
 $\langle \text{proof} \rangle$

lemma *plus-aux*:
assumes $f \in o[F](g)$
shows $g \in L\ F\ (\lambda x. f\ x + g\ x)$
 $\langle \text{proof} \rangle$

end

lemma *summable-comparison-test-bigo*:
fixes $f :: \text{nat} \Rightarrow \text{real}$
assumes $\text{summable } (\lambda n. \text{norm } (g\ n)) \ f \in O(g)$
shows $\text{summable } f$
 $\langle \text{proof} \rangle$

lemma *bigomega-iff-bigo*: $g \in \Omega[F](f) \longleftrightarrow f \in O[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallomega-iff-smallo*: $g \in \omega[F](f) \longleftrightarrow f \in o[F](g)$
 $\langle \text{proof} \rangle$

context *landau-pair*
begin

lemma *big-mono*:
 $\text{eventually } (\lambda x. R\ (\text{norm } (f\ x))\ (\text{norm } (g\ x))) \ F \implies f \in L\ F\ (g)$
 $\langle \text{proof} \rangle$

lemma *big-mult*:
assumes $f1 \in L\ F\ (g1) \ f2 \in L\ F\ (g2)$
shows $(\lambda x. f1\ x * f2\ x) \in L\ F\ (\lambda x. g1\ x * g2\ x)$
 $\langle \text{proof} \rangle$

lemma *small-big-mult*:
assumes $f1 \in l\ F\ (g1) \ f2 \in L\ F\ (g2)$
shows $(\lambda x. f1\ x * f2\ x) \in l\ F\ (\lambda x. g1\ x * g2\ x)$
 $\langle \text{proof} \rangle$

lemma *big-small-mult*:
 $f1 \in L\ F\ (g1) \implies f2 \in l\ F\ (g2) \implies (\lambda x. f1\ x * f2\ x) \in l\ F\ (\lambda x. g1\ x * g2\ x)$
 $\langle \text{proof} \rangle$

lemma *small-mult*: $f1 \in l\ F\ (g1) \implies f2 \in l\ F\ (g2) \implies (\lambda x. f1\ x * f2\ x) \in l\ F\ (\lambda x. g1\ x * g2\ x)$

$\langle \text{proof} \rangle$

lemmas $\text{mult} = \text{big-mult} \text{ small-big-mult} \text{ big-small-mult} \text{ small-mult}$

lemma *big-power*:

assumes $f \in L F (g)$

shows $(\lambda x. f x \wedge m) \in L F (\lambda x. g x \wedge m)$

$\langle \text{proof} \rangle$

lemma (*in landau-pair*) *small-power*:

assumes $f \in l F (g) \ m > 0$

shows $(\lambda x. f x \wedge m) \in l F (\lambda x. g x \wedge m)$

$\langle \text{proof} \rangle$

lemma *big-power-increasing*:

assumes $(\lambda-. 1) \in L F f \ m \leq n$

shows $(\lambda x. f x \wedge m) \in L F (\lambda x. f x \wedge n)$

$\langle \text{proof} \rangle$

lemma *small-power-increasing*:

assumes $(\lambda-. 1) \in l F f \ m < n$

shows $(\lambda x. f x \wedge m) \in l F (\lambda x. f x \wedge n)$

$\langle \text{proof} \rangle$

sublocale *big*: *landau-symbol* $L \ L' \ Lr$

$\langle \text{proof} \rangle$

sublocale *small*: *landau-symbol* $l \ l' \ lr$

$\langle \text{proof} \rangle$

These rules allow chaining of Landau symbol propositions in Isar with "also".

lemma *big-mult-1*: $f \in L F (g) \implies (\lambda-. 1) \in L F (h) \implies f \in L F (\lambda x. g x * h x)$

and *big-mult-1'*: $(\lambda-. 1) \in L F (g) \implies f \in L F (h) \implies f \in L F (\lambda x. g x * h x)$

and *small-mult-1*: $f \in l F (g) \implies (\lambda-. 1) \in L F (h) \implies f \in l F (\lambda x. g x * h x)$

and *small-mult-1'*: $(\lambda-. 1) \in L F (g) \implies f \in l F (h) \implies f \in l F (\lambda x. g x * h x)$

and *small-mult-1''*: $f \in L F (g) \implies (\lambda-. 1) \in l F (h) \implies f \in l F (\lambda x. g x * h x)$

and *small-mult-1'''*: $(\lambda-. 1) \in l F (g) \implies f \in L F (h) \implies f \in l F (\lambda x. g x * h x)$

$\langle \text{proof} \rangle$

lemma *big-1-mult*: $f \in L F (g) \implies h \in L F (\lambda-. 1) \implies (\lambda x. f x * h x) \in L F (g)$

and *big-1-mult'*: $h \in L F (\lambda-. 1) \implies f \in L F (g) \implies (\lambda x. f x * h x) \in L F (g)$

and *small-1-mult*: $f \in l\ F\ (g) \implies h \in L\ F\ (\lambda\cdot.\ 1) \implies (\lambda x. f\ x * h\ x) \in l\ F\ (g)$
and *small-1-mult'*: $h \in L\ F\ (\lambda\cdot.\ 1) \implies f \in l\ F\ (g) \implies (\lambda x. f\ x * h\ x) \in l\ F\ (g)$
and *small-1-mult''*: $f \in L\ F\ (g) \implies h \in l\ F\ (\lambda\cdot.\ 1) \implies (\lambda x. f\ x * h\ x) \in l\ F\ (g)$
and *small-1-mult'''*: $h \in l\ F\ (\lambda\cdot.\ 1) \implies f \in L\ F\ (g) \implies (\lambda x. f\ x * h\ x) \in l\ F\ (g)$
 <proof>

lemmas *mult-1-trans* =

big-mult-1 big-mult-1' small-mult-1 small-mult-1' small-mult-1'' small-mult-1'''
big-1-mult big-1-mult' small-1-mult small-1-mult' small-1-mult'' small-1-mult'''

lemma *big-equal-iff-bigtheta*: $L\ F\ (f) = L\ F\ (g) \longleftrightarrow f \in \Theta[F](g)$
 <proof>

lemma *big-prod*:

assumes $\bigwedge x. x \in A \implies f\ x \in L\ F\ (g\ x)$
shows $(\lambda y. \prod_{x \in A}. f\ x\ y) \in L\ F\ (\lambda y. \prod_{x \in A}. g\ x\ y)$
 <proof>

lemma *big-prod-in-1*:

assumes $\bigwedge x. x \in A \implies f\ x \in L\ F\ (\lambda\cdot.\ 1)$
shows $(\lambda y. \prod_{x \in A}. f\ x\ y) \in L\ F\ (\lambda\cdot.\ 1)$
 <proof>

end

context *landau-symbol*

begin

lemma *plus-absorb1*:

assumes $f \in o[F](g)$
shows $L\ F\ (\lambda x. f\ x + g\ x) = L\ F\ (g)$
 <proof>

lemma *plus-absorb2*: $g \in o[F](f) \implies L\ F\ (\lambda x. f\ x + g\ x) = L\ F\ (f)$
 <proof>

lemma *diff-absorb1*: $f \in o[F](g) \implies L\ F\ (\lambda x. f\ x - g\ x) = L\ F\ (g)$
 <proof>

lemma *diff-absorb2*: $g \in o[F](f) \implies L\ F\ (\lambda x. f\ x - g\ x) = L\ F\ (f)$
 <proof>

lemmas *absorb* = *plus-absorb1 plus-absorb2 diff-absorb1 diff-absorb2*

end

lemma *bighetaI* [intro]: $f \in O[F](g) \implies f \in \Omega[F](g) \implies f \in \Theta[F](g)$
 ⟨proof⟩

lemma *bighetaD1* [dest]: $f \in \Theta[F](g) \implies f \in O[F](g)$
and *bighetaD2* [dest]: $f \in \Theta[F](g) \implies f \in \Omega[F](g)$
 ⟨proof⟩

lemma *bigheta-refl* [simp]: $f \in \Theta[F](f)$
 ⟨proof⟩

lemma *bigheta-sym*: $f \in \Theta[F](g) \longleftrightarrow g \in \Theta[F](f)$
 ⟨proof⟩

lemmas *landau-flip* =
bigomega-iff-bigo[symmetric] *smallomega-iff-smallo*[symmetric]
bigomega-iff-bigo *smallomega-iff-smallo* *bigheta-sym*

interpretation *landau-theta*: *landau-symbol* *bigheta* *bigheta* *bigheta*
 ⟨proof⟩

lemmas *landau-symbols* =
landau-o.big.landau-symbol-axioms *landau-o.small.landau-symbol-axioms*
landau-omega.big.landau-symbol-axioms *landau-omega.small.landau-symbol-axioms*
landau-theta.landau-symbol-axioms

lemma *bigoI* [intro]:
assumes *eventually* $(\lambda x. (\text{norm } (f x)) \leq c * (\text{norm } (g x))) F$
shows $f \in O[F](g)$
 ⟨proof⟩

lemma *smallomegaD* [dest]:
assumes $f \in \omega[F](g)$
shows *eventually* $(\lambda x. (\text{norm } (f x)) \geq c * (\text{norm } (g x))) F$
 ⟨proof⟩

lemma *bighetaI'*:
assumes $c1 > 0$ $c2 > 0$
assumes *eventually* $(\lambda x. c1 * (\text{norm } (g x)) \leq (\text{norm } (f x)) \wedge (\text{norm } (f x)) \leq c2$
 $* (\text{norm } (g x))) F$
shows $f \in \Theta[F](g)$
 ⟨proof⟩

lemma *bighetaI-cong*: *eventually* $(\lambda x. f x = g x) F \implies f \in \Theta[F](g)$
 ⟨proof⟩

lemma (in *landau-symbol*) *ev-eq-trans1*:

$f \in L F (\lambda x. g x (h x)) \implies \text{eventually } (\lambda x. h x = h' x) F \implies f \in L F (\lambda x. g x (h' x))$
 $\langle \text{proof} \rangle$

lemma (in *landau-symbol*) *ev-eq-trans2*:

$\text{eventually } (\lambda x. f x = f' x) F \implies (\lambda x. g x (f' x)) \in L F (h) \implies (\lambda x. g x (f x)) \in L F (h)$
 $\langle \text{proof} \rangle$

declare *landau-o.smallI landau-omega.bigI landau-omega.smallI* [intro]

declare *landau-o.bigE landau-omega.bigE* [elim]

declare *landau-o.smallD*

lemma (in *landau-symbol*) *bigheta-trans1'*:

$f \in L F (g) \implies h \in \Theta[F](g) \implies f \in L F (h)$
 $\langle \text{proof} \rangle$

lemma (in *landau-symbol*) *bigheta-trans2'*:

$g \in \Theta[F](f) \implies g \in L F (h) \implies f \in L F (h)$
 $\langle \text{proof} \rangle$

lemma *bigo-bigomega-trans*: $f \in O[F](g) \implies h \in \Omega[F](g) \implies f \in O[F](h)$

and *bigo-smallomega-trans*: $f \in O[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$

and *smallo-bigomega-trans*: $f \in o[F](g) \implies h \in \Omega[F](g) \implies f \in o[F](h)$

and *smallo-smallomega-trans*: $f \in o[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$

and *bigomega-bigo-trans*: $f \in \Omega[F](g) \implies h \in O[F](g) \implies f \in \Omega[F](h)$

and *bigomega-smallo-trans*: $f \in \Omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$

and *smallomega-bigo-trans*: $f \in \omega[F](g) \implies h \in O[F](g) \implies f \in \omega[F](h)$

and *smallomega-smallo-trans*: $f \in \omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$

$\langle \text{proof} \rangle$

lemmas *landau-trans-lift* [trans] =

landau-symbols[*THEN landau-symbol.lift-trans*]

landau-symbols[*THEN landau-symbol.lift-trans*']

landau-symbols[*THEN landau-symbol.lift-trans-bigheta*]

landau-symbols[*THEN landau-symbol.lift-trans-bigheta*']

lemmas *landau-mult-1-trans* [trans] =

landau-o.mult-1-trans landau-omega.mult-1-trans

lemmas *landau-trans* [trans] =

landau-symbols[*THEN landau-symbol.bigheta-trans1*]

landau-symbols[*THEN landau-symbol.bigheta-trans2*]

landau-symbols[*THEN landau-symbol.bigheta-trans1*']

landau-symbols[*THEN landau-symbol.bigheta-trans2*']

landau-symbols[*THEN landau-symbol.ev-eq-trans1*]

landau-symbols[*THEN landau-symbol.ev-eq-trans2*]

landau-o.big-trans landau-o.small-trans landau-o.small-big-trans landau-o.big-small-trans

landau-omega.big-trans landau-omega.small-trans
landau-omega.small-big-trans landau-omega.big-small-trans

bigo-bigomega-trans bigo-smallomega-trans smallo-bigomega-trans smallo-smallomega-trans
bigomega-bigo-trans bigomega-smallo-trans smallomega-bigo-trans smallomega-smallo-trans

lemma *bigtheta-inverse [simp]*:
shows $(\lambda x. \text{inverse } (f x)) \in \Theta[F](\lambda x. \text{inverse } (g x)) \longleftrightarrow f \in \Theta[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigtheta-divide*:
assumes $f1 \in \Theta(f2)$ $g1 \in \Theta(g2)$
shows $(\lambda x. f1 x / g1 x) \in \Theta(\lambda x. f2 x / g2 x)$
 $\langle \text{proof} \rangle$

lemma *eventually-nonzero-bigtheta*:
assumes $f \in \Theta[F](g)$
shows $\text{eventually } (\lambda x. f x \neq 0) F \longleftrightarrow \text{eventually } (\lambda x. g x \neq 0) F$
 $\langle \text{proof} \rangle$

54.2 Landau symbols and limits

lemma *bigoI-tendsto-norm*:
fixes $f g$
assumes $((\lambda x. \text{norm } (f x / g x)) \longrightarrow c) F$
assumes $\text{eventually } (\lambda x. g x \neq 0) F$
shows $f \in O[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigoI-tendsto*:
assumes $((\lambda x. f x / g x) \longrightarrow c) F$
assumes $\text{eventually } (\lambda x. g x \neq 0) F$
shows $f \in O[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigomegaI-tendsto-norm*:
assumes $c\text{-not-0: } (c::\text{real}) \neq 0$
assumes $\text{lim: } ((\lambda x. \text{norm } (f x / g x)) \longrightarrow c) F$
shows $f \in \Omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigomegaI-tendsto*:
assumes $c\text{-not-0: } (c::\text{real}) \neq 0$
assumes $\text{lim: } ((\lambda x. f x / g x) \longrightarrow c) F$
shows $f \in \Omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallomegaI-filterlim-at-top-norm*:
assumes $\text{lim: filterlim } (\lambda x. \text{norm } (f x / g x)) \text{ at-top } F$

shows $f \in \omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallomegaI-filterlim-at-infinity*:
assumes $\text{lim: filterlim } (\lambda x. f\ x / g\ x) \text{ at-infinity } F$
shows $f \in \omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallomegaD-filterlim-at-top-norm*:
assumes $f \in \omega[F](g)$
assumes $\text{eventually } (\lambda x. g\ x \neq 0) F$
shows $\text{LIM } x\ F. \text{ norm } (f\ x / g\ x) :> \text{ at-top}$
 $\langle \text{proof} \rangle$

lemma *smallomegaD-filterlim-at-infinity*:
assumes $f \in \omega[F](g)$
assumes $\text{eventually } (\lambda x. g\ x \neq 0) F$
shows $\text{LIM } x\ F. f\ x / g\ x :> \text{ at-infinity}$
 $\langle \text{proof} \rangle$

lemma *smallomega-1-conv-filterlim*: $f \in \omega[F](\lambda-. 1) \longleftrightarrow \text{filterlim } f \text{ at-infinity } F$
 $\langle \text{proof} \rangle$

lemma *smalloI-tendsto*:
assumes $\text{lim: } ((\lambda x. f\ x / g\ x) \longrightarrow 0) F$
assumes $\text{eventually } (\lambda x. g\ x \neq 0) F$
shows $f \in o[F](g)$
 $\langle \text{proof} \rangle$

lemma *smalloD-tendsto*:
assumes $f \in o[F](g)$
shows $((\lambda x. f\ x / g\ x) \longrightarrow 0) F$
 $\langle \text{proof} \rangle$

lemma *bigthetaI-tendsto-norm*:
assumes $c\text{-not-0: } (c::\text{real}) \neq 0$
assumes $\text{lim: } ((\lambda x. \text{norm } (f\ x / g\ x)) \longrightarrow c) F$
shows $f \in \Theta[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigthetaI-tendsto*:
assumes $c\text{-not-0: } (c::\text{real}) \neq 0$
assumes $\text{lim: } ((\lambda x. f\ x / g\ x) \longrightarrow c) F$
shows $f \in \Theta[F](g)$
 $\langle \text{proof} \rangle$

lemma *tendsto-add-smallo*:
assumes $(f1 \longrightarrow a) F$
assumes $f2 \in o[F](f1)$

shows $((\lambda x. f1\ x + f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

lemma *tendsto-diff-smallo*:

shows $(f1 \longrightarrow a)\ F \implies f2 \in o[F](f1) \implies ((\lambda x. f1\ x - f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

lemma *tendsto-add-smallo-iff*:

assumes $f2 \in o[F](f1)$
shows $(f1 \longrightarrow a)\ F \longleftrightarrow ((\lambda x. f1\ x + f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

lemma *tendsto-diff-smallo-iff*:

shows $f2 \in o[F](f1) \implies (f1 \longrightarrow a)\ F \longleftrightarrow ((\lambda x. f1\ x - f2\ x) \longrightarrow a)\ F$
 $\langle proof \rangle$

lemma *tendsto-divide-smallo*:

assumes $((\lambda x. f1\ x / g1\ x) \longrightarrow a)\ F$
assumes $f2 \in o[F](f1)\ g2 \in o[F](g1)$
assumes *eventually* $(\lambda x. g1\ x \neq 0)\ F$
shows $((\lambda x. (f1\ x + f2\ x) / (g1\ x + g2\ x)) \longrightarrow a)\ F$ (**is** $(?f \longrightarrow -)\ -$)
 $\langle proof \rangle$

lemma *bigO-powr*:

fixes $f :: 'a \Rightarrow real$
assumes $f \in O[F](g)\ p \geq 0$
shows $(\lambda x. |f\ x|^{powr\ p}) \in O[F](\lambda x. |g\ x|^{powr\ p})$
 $\langle proof \rangle$

lemma *smallo-powr*:

fixes $f :: 'a \Rightarrow real$
assumes $f \in o[F](g)\ p > 0$
shows $(\lambda x. |f\ x|^{powr\ p}) \in o[F](\lambda x. |g\ x|^{powr\ p})$
 $\langle proof \rangle$

lemma *smallo-powr-nonneg*:

fixes $f :: 'a \Rightarrow real$
assumes $f \in o[F](g)\ p > 0$ *eventually* $(\lambda x. f\ x \geq 0)\ F$ *eventually* $(\lambda x. g\ x \geq 0)\ F$
shows $(\lambda x. f\ x^{powr\ p}) \in o[F](\lambda x. g\ x^{powr\ p})$
 $\langle proof \rangle$

lemma *bigtheta-powr*:

fixes $f :: 'a \Rightarrow real$
shows $f \in \Theta[F](g) \implies (\lambda x. |f\ x|^{powr\ p}) \in \Theta[F](\lambda x. |g\ x|^{powr\ p})$
 $\langle proof \rangle$

lemma *bigO-powr-nonneg*:

fixes $f :: 'a \Rightarrow \text{real}$
assumes $f \in O[F](g)$ $p \geq 0$ *eventually* $(\lambda x. f\ x \geq 0)$ F *eventually* $(\lambda x. g\ x \geq 0)$
 F
shows $(\lambda x. f\ x \text{ powr } p) \in O[F](\lambda x. g\ x \text{ powr } p)$
 $\langle \text{proof} \rangle$

lemma *zero-in-smallo* [simp]: $(\lambda-. 0) \in o[F](f)$
 $\langle \text{proof} \rangle$

lemma *zero-in-bigo* [simp]: $(\lambda-. 0) \in O[F](f)$
 $\langle \text{proof} \rangle$

lemma *in-bigomega-zero* [simp]: $f \in \Omega[F](\lambda x. 0)$
 $\langle \text{proof} \rangle$

lemma *in-smallomega-zero* [simp]: $f \in \omega[F](\lambda x. 0)$
 $\langle \text{proof} \rangle$

lemma *in-smallo-zero-iff* [simp]: $f \in o[F](\lambda-. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) F$
 $\langle \text{proof} \rangle$

lemma *in-bigo-zero-iff* [simp]: $f \in O[F](\lambda-. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) F$
 $\langle \text{proof} \rangle$

lemma *zero-in-smallomega-iff* [simp]: $(\lambda-. 0) \in \omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) F$
 $\langle \text{proof} \rangle$

lemma *zero-in-bigomega-iff* [simp]: $(\lambda-. 0) \in \Omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) F$
 $\langle \text{proof} \rangle$

lemma *zero-in-bigtheta-iff* [simp]: $(\lambda-. 0) \in \Theta[F](f) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) F$
 $\langle \text{proof} \rangle$

lemma *in-bigtheta-zero-iff* [simp]: $f \in \Theta[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) F$
 $\langle \text{proof} \rangle$

lemma *cmult-in-bigo-iff* [simp]: $(\lambda x. c * f\ x) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$
and *cmult-in-bigo-iff'* [simp]: $(\lambda x. f\ x * c) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$
and *cmult-in-smallo-iff* [simp]: $(\lambda x. c * f\ x) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
and *cmult-in-smallo-iff'* [simp]: $(\lambda x. f\ x * c) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
 $\langle \text{proof} \rangle$

lemma *bigo-const* [simp]: $(\lambda-. c) \in O[F](\lambda-. 1) \langle \text{proof} \rangle$

lemma *bigo-const-iff* [simp]: $(\lambda-. c1) \in O[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 = 0 \vee c2 \neq 0$
 $\langle \text{proof} \rangle$

lemma *bigomega-const-iff* [simp]: $(\lambda-. c1) \in \Omega[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 \neq 0 \vee c2 = 0$
 $\langle \text{proof} \rangle$

lemma *smallo-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in o(g) \Longrightarrow (\lambda x::\text{nat}. f (\text{real } x)) \in o(\lambda x. g (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *bigo-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in O(g) \Longrightarrow (\lambda x::\text{nat}. f (\text{real } x)) \in O(\lambda x. g (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *smallomega-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in \omega(g) \Longrightarrow (\lambda x::\text{nat}. f (\text{real } x)) \in \omega(\lambda x. g (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *bigomega-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in \Omega(g) \Longrightarrow (\lambda x::\text{nat}. f (\text{real } x)) \in \Omega(\lambda x. g (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *bigtheta-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in \Theta(g) \Longrightarrow (\lambda x::\text{nat}. f (\text{real } x)) \in \Theta(\lambda x. g (\text{real } x))$
 $\langle \text{proof} \rangle$

lemmas *landau-real-nat-transfer* [intro] =

bigo-real-nat-transfer *smallo-real-nat-transfer* *bigomega-real-nat-transfer*
smallomega-real-nat-transfer *bigtheta-real-nat-transfer*

lemma *landau-symbol-if-at-top-eq* [simp]:

assumes *landau-symbol* $L L' Lr$

shows $L \text{ at-top } (\lambda x::'a::\text{linordered-semidom}. \text{if } x = a \text{ then } f \ x \text{ else } g \ x) = L$
 $\text{at-top } (g)$
 $\langle \text{proof} \rangle$

lemmas *landau-symbols-if-at-top-eq* [simp] = *landau-symbols*[*THEN* *landau-symbol-if-at-top-eq*]

lemma *sum-in-smallo*:

assumes $f \in o[F](h) \ g \in o[F](h)$

shows $(\lambda x. f \ x + g \ x) \in o[F](h) \ (\lambda x. f \ x - g \ x) \in o[F](h)$
 $\langle \text{proof} \rangle$

lemma *big-sum-in-smallo*:

assumes $\bigwedge x. x \in A \implies f\ x \in o[F](g)$
shows $(\lambda x. \text{sum } (\lambda y. f\ y\ x)\ A) \in o[F](g)$
 $\langle \text{proof} \rangle$

lemma *sum-in-bigo*:

assumes $f \in O[F](h)$ $g \in O[F](h)$
shows $(\lambda x. f\ x + g\ x) \in O[F](h)$ $(\lambda x. f\ x - g\ x) \in O[F](h)$
 $\langle \text{proof} \rangle$

lemma *big-sum-in-bigo*:

assumes $\bigwedge x. x \in A \implies f\ x \in O[F](g)$
shows $(\lambda x. \text{sum } (\lambda y. f\ y\ x)\ A) \in O[F](g)$
 $\langle \text{proof} \rangle$

lemma *smallo-multiples*:

assumes $f: f \in o(\text{real})$ **and** $k > 0$
shows $(\lambda n. f\ (k * n)) \in o(\text{real})$
 $\langle \text{proof} \rangle$

lemma *maxmin-in-smallo*:

assumes $f \in o[F](h)$ $g \in o[F](h)$
shows $(\lambda k. \text{max } (f\ k)\ (g\ k)) \in o[F](h)$ $(\lambda k. \text{min } (f\ k)\ (g\ k)) \in o[F](h)$
 $\langle \text{proof} \rangle$

lemma *le-imp-bigo-real*:

assumes $c \geq 0$ *eventually* $(\lambda x. f\ x \leq c * (g\ x :: \text{real}))\ F$ *eventually* $(\lambda x. 0 \leq f\ x)\ F$
shows $f \in O[F](g)$
 $\langle \text{proof} \rangle$

context *landau-symbol*

begin

lemma *mult-cancel-left*:

assumes $f1 \in \Theta[F](g1)$ **and** *eventually* $(\lambda x. g1\ x \neq 0)\ F$
notes $[trans] = \text{bigtheta-trans1 bigtheta-trans2}$
shows $(\lambda x. f1\ x * f2\ x) \in L\ F\ (\lambda x. g1\ x * g2\ x) \longleftrightarrow f2 \in L\ F\ (g2)$
 $\langle \text{proof} \rangle$

lemma *mult-cancel-right*:

assumes $f2 \in \Theta[F](g2)$ **and** *eventually* $(\lambda x. g2\ x \neq 0)\ F$
shows $(\lambda x. f1\ x * f2\ x) \in L\ F\ (\lambda x. g1\ x * g2\ x) \longleftrightarrow f1 \in L\ F\ (g1)$
 $\langle \text{proof} \rangle$

lemma *divide-cancel-right*:

assumes $f2 \in \Theta[F](g2)$ **and** *eventually* $(\lambda x. g2\ x \neq 0)\ F$
shows $(\lambda x. f1\ x / f2\ x) \in L\ F\ (\lambda x. g1\ x / g2\ x) \longleftrightarrow f1 \in L\ F\ (g1)$

$\langle \text{proof} \rangle$

lemma *divide-cancel-left*:

assumes $f1 \in \Theta[F](g1)$ **and** *eventually* $(\lambda x. g1\ x \neq 0)\ F$

shows $(\lambda x. f1\ x / f2\ x) \in L\ F\ (\lambda x. g1\ x / g2\ x) \longleftrightarrow$
 $(\lambda x. \text{inverse}\ (f2\ x)) \in L\ F\ (\lambda x. \text{inverse}\ (g2\ x))$

$\langle \text{proof} \rangle$

end

lemma *powr-smallo-iff*:

assumes *filterlim* g *at-top* $F\ F \neq \text{bot}$

shows $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in o[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p < q$

$\langle \text{proof} \rangle$

lemma *powr-bigo-iff*:

assumes *filterlim* g *at-top* $F\ F \neq \text{bot}$

shows $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in O[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p \leq q$

$\langle \text{proof} \rangle$

lemma *powr-bigtheta-iff*:

assumes *filterlim* g *at-top* $F\ F \neq \text{bot}$

shows $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in \Theta[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p = q$

$\langle \text{proof} \rangle$

54.3 Flatness of real functions

Given two real-valued functions f and g , we say that f is flatter than g if any power of $f(x)$ is asymptotically dominated by any positive power of $g(x)$. This is a useful notion since, given two products of powers of functions sorted by flatness, we can compare them asymptotically by simply comparing the exponent lists lexicographically.

A simple sufficient criterion for flatness is that $\ln f(x) \in o(\ln g(x))$, which we show now.

lemma *ln-smallo-imp-flat*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes *lim-f*: *filterlim* f *at-top* *at-top*

assumes *lim-g*: *filterlim* g *at-top* *at-top*

assumes *ln-o-ln*: $(\lambda x. \ln (f\ x)) \in o(\lambda x. \ln (g\ x))$

assumes $q: q > 0$

shows $(\lambda x. f\ x\ \text{powr}\ p) \in o(\lambda x. g\ x\ \text{powr}\ q)$

$\langle \text{proof} \rangle$

lemma *ln-smallo-imp-flat'*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$

assumes *lim-f*: *filterlim* f *at-top* *at-top*

assumes *lim-g*: *filterlim* g *at-top* *at-top*

assumes *ln-o-ln*: $(\lambda x. \text{ln } (f x)) \in o(\lambda x. \text{ln } (g x))$
assumes *q*: $q < 0$
shows $(\lambda x. g x \text{ powr } q) \in o(\lambda x. f x \text{ powr } p)$
 <proof>

54.4 Asymptotic Equivalence

named-theorems *asympt-equiv-intros*
named-theorems *asympt-equiv-simps*

definition *asympt-equiv* :: $('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow 'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$
 $(\langle \langle \text{open-block notation} = \langle \text{mixfix } \text{asympt-equiv} \rangle \rangle \sim [-] -) \rangle [51, 10, 51] 50)$
where $f \sim[F] g \longleftrightarrow ((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1) F$

abbreviation (*input*) *asympt-equiv-at-top* **where**
asympt-equiv-at-top $f g \equiv f \sim[at-top] g$

bundle *asympt-equiv-syntax*
begin
notation *asympt-equiv-at-top* (**infix** $\langle \sim \rangle$ 50)
end

lemma *asympt-equivI*: $((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1)$
 $F \Longrightarrow f \sim[F] g$
 <proof>

lemma *asympt-equivD*: $f \sim[F] g \Longrightarrow ((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1) F$
 <proof>

lemma *asympt-equiv-filtermap-iff*:
 $f \sim[\text{filtermap } h F] g \longleftrightarrow (\lambda x. f (h x)) \sim[F] (\lambda x. g (h x))$
 <proof>

lemma *asympt-equiv-refl* [*simp*, *asympt-equiv-intros*]: $f \sim[F] f$
 <proof>

lemma *asympt-equiv-symI*:
assumes $f \sim[F] g$
shows $g \sim[F] f$
 <proof>

lemma *asympt-equiv-sym*: $f \sim[F] g \longleftrightarrow g \sim[F] f$
 <proof>

lemma *asympt-equivI'*:
assumes $((\lambda x. f x / g x) \longrightarrow 1) F$
shows $f \sim[F] g$

$\langle proof \rangle$

lemma *tendsto-imp-asymp-equiv-const*:

assumes $(f \longrightarrow c) \ F \ c \neq 0$

shows $f \sim[F] (\lambda\cdot. c)$

$\langle proof \rangle$

lemma *asymp-equiv-cong*:

assumes *eventually* $(\lambda x. f1\ x = f2\ x) \ F$ *eventually* $(\lambda x. g1\ x = g2\ x) \ F$

shows $f1 \sim[F] g1 \longleftrightarrow f2 \sim[F] g2$

$\langle proof \rangle$

lemma *asymp-equiv-eventually-zeros*:

fixes $f\ g :: 'a \Rightarrow 'b :: \text{real-normed-field}$

assumes $f \sim[F] g$

shows *eventually* $(\lambda x. f\ x = 0 \longleftrightarrow g\ x = 0) \ F$

$\langle proof \rangle$

lemma *asymp-equiv-transfer*:

assumes $f1 \sim[F] g1$ *eventually* $(\lambda x. f1\ x = f2\ x) \ F$ *eventually* $(\lambda x. g1\ x = g2\ x) \ F$

F

shows $f2 \sim[F] g2$

$\langle proof \rangle$

lemma *asymp-equiv-transfer-trans* $[trans]$:

assumes $(\lambda x. f\ x (h1\ x)) \sim[F] (\lambda x. g\ x (h1\ x))$

assumes *eventually* $(\lambda x. h1\ x = h2\ x) \ F$

shows $(\lambda x. f\ x (h2\ x)) \sim[F] (\lambda x. g\ x (h2\ x))$

$\langle proof \rangle$

lemma *asymp-equiv-trans* $[trans]$:

fixes $f\ g\ h$

assumes $f \sim[F] g \ g \sim[F] h$

shows $f \sim[F] h$

$\langle proof \rangle$

lemma *asymp-equiv-trans-lift1* $[trans]$:

assumes $a \sim[F] f \ b \ b \sim[F] c \ \bigwedge c \ d. \ c \sim[F] d \implies f\ c \sim[F] f\ d$

shows $a \sim[F] f\ c$

$\langle proof \rangle$

lemma *asymp-equiv-trans-lift2* $[trans]$:

assumes $f\ a \sim[F] b \ a \sim[F] c \ \bigwedge c \ d. \ c \sim[F] d \implies f\ c \sim[F] f\ d$

shows $f\ c \sim[F] b$

$\langle proof \rangle$

lemma *asymp-equivD-const*:

assumes $f \sim[F] (\lambda\cdot. c)$

shows $(f \longrightarrow c) \ F$

$\langle proof \rangle$

lemma *asympt-equiv-refl-ev*:

assumes *eventually* $(\lambda x. f\ x = g\ x)\ F$

shows $f \sim[F] g$

$\langle proof \rangle$

lemma *asympt-equiv-nhds-iff*: $f \sim[nhds\ (z :: 'a :: t1-space)]\ g \longleftrightarrow f \sim[at\ z]\ g \wedge f\ z = g\ z$

$\langle proof \rangle$

lemma *asympt-equiv-sandwich*:

fixes $f\ g\ h :: 'a \Rightarrow 'b :: \{real-normed-field, order-topology, linordered-field\}$

assumes *eventually* $(\lambda x. f\ x \geq 0)\ F$

assumes *eventually* $(\lambda x. f\ x \leq g\ x)\ F$

assumes *eventually* $(\lambda x. g\ x \leq h\ x)\ F$

assumes $f \sim[F] h$

shows $g \sim[F] f\ g \sim[F] h$

$\langle proof \rangle$

lemma *asympt-equiv-imp-eventually-same-sign*:

fixes $f\ g :: real \Rightarrow real$

assumes $f \sim[F] g$

shows *eventually* $(\lambda x. sgn\ (f\ x) = sgn\ (g\ x))\ F$

$\langle proof \rangle$

lemma

fixes $f\ g :: - \Rightarrow real$

assumes $f \sim[F] g$

shows *asympt-equiv-eventually-same-sign*: *eventually* $(\lambda x. sgn\ (f\ x) = sgn\ (g\ x))\ F$ (**is** ?th1)

and *asympt-equiv-eventually-neg-iff*: *eventually* $(\lambda x. f\ x < 0 \longleftrightarrow g\ x < 0)$ *F* (**is** ?th2)

and *asympt-equiv-eventually-pos-iff*: *eventually* $(\lambda x. f\ x > 0 \longleftrightarrow g\ x > 0)$ *F* (**is** ?th3)

$\langle proof \rangle$

lemma *asympt-equiv-tendsto-transfer*:

assumes $f \sim[F] g$ **and** $(f \longrightarrow c)\ F$

shows $(g \longrightarrow c)\ F$

$\langle proof \rangle$

lemma *tendsto-asympt-equiv-cong*:

assumes $f \sim[F] g$

shows $(f \longrightarrow c)\ F \longleftrightarrow (g \longrightarrow c)\ F$

$\langle proof \rangle$

lemma *smallo-imp-eventually-sgn*:

fixes $f\ g :: \text{real} \Rightarrow \text{real}$
assumes $g \in o(f)$
shows *eventually* $(\lambda x. \text{sgn } (f\ x + g\ x) = \text{sgn } (f\ x)) \text{ at-top}$
 $\langle \text{proof} \rangle$

context
begin

private lemma *asympt-equiv-add-rightI*:

assumes $f \sim[F] g \ h \in o[F](g)$
shows $(\lambda x. f\ x + h\ x) \sim[F] g$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-add-right* [*asympt-equiv-simps*]:

assumes $h \in o[F](g)$
shows $(\lambda x. f\ x + h\ x) \sim[F] g \longleftrightarrow f \sim[F] g$
 $\langle \text{proof} \rangle$

end

lemma *asympt-equiv-add-left* [*asympt-equiv-simps*]:

assumes $h \in o[F](g)$
shows $(\lambda x. h\ x + f\ x) \sim[F] g \longleftrightarrow f \sim[F] g$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-add-right'* [*asympt-equiv-simps*]:

assumes $h \in o[F](g)$
shows $g \sim[F] (\lambda x. f\ x + h\ x) \longleftrightarrow g \sim[F] f$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-add-left'* [*asympt-equiv-simps*]:

assumes $h \in o[F](g)$
shows $g \sim[F] (\lambda x. h\ x + f\ x) \longleftrightarrow g \sim[F] f$
 $\langle \text{proof} \rangle$

lemma *smallo-imp-asympt-equiv*:

assumes $(\lambda x. f\ x - g\ x) \in o[F](g)$
shows $f \sim[F] g$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-uminus* [*asympt-equiv-intros*]:

$f \sim[F] g \implies (\lambda x. -f\ x) \sim[F] (\lambda x. -g\ x)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-uminus-iff* [*asympt-equiv-simps*]:

$(\lambda x. -f\ x) \sim[F] g \longleftrightarrow f \sim[F] (\lambda x. -g\ x)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-mult* [*asympt-equiv-intros*]:

fixes $f1\ f2\ g1\ g2 :: 'a \Rightarrow 'b :: \text{real-normed-field}$
assumes $f1 \sim[F] g1\ f2 \sim[F] g2$
shows $(\lambda x. f1\ x * f2\ x) \sim[F] (\lambda x. g1\ x * g2\ x)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-power* [*asympt-equiv-intros*]:
 $f \sim[F] g \implies (\lambda x. f\ x \wedge^n) \sim[F] (\lambda x. g\ x \wedge^n)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-inverse* [*asympt-equiv-intros*]:
assumes $f \sim[F] g$
shows $(\lambda x. \text{inverse}\ (f\ x)) \sim[F] (\lambda x. \text{inverse}\ (g\ x))$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-inverse-iff* [*asympt-equiv-simps*]:
 $(\lambda x. \text{inverse}\ (f\ x)) \sim[F] (\lambda x. \text{inverse}\ (g\ x)) \iff f \sim[F] g$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-divide* [*asympt-equiv-intros*]:
assumes $f1 \sim[F] g1\ f2 \sim[F] g2$
shows $(\lambda x. f1\ x / f2\ x) \sim[F] (\lambda x. g1\ x / g2\ x)$
 $\langle \text{proof} \rangle$

lemma *asympt-equivD-strong*:
assumes $f \sim[F] g$ *eventually* $(\lambda x. f\ x \neq 0 \vee g\ x \neq 0)\ F$
shows $((\lambda x. f\ x / g\ x) \longrightarrow 1)\ F$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-compose* [*asympt-equiv-intros*]:
assumes $f \sim[G] g$ *filterlim* $h\ G\ F$
shows $f \circ h \sim[F] g \circ h$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-compose'*:
assumes $f \sim[G] g$ *filterlim* $h\ G\ F$
shows $(\lambda x. f\ (h\ x)) \sim[F] (\lambda x. g\ (h\ x))$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-powr-real* [*asympt-equiv-intros*]:
fixes $f\ g :: 'a \Rightarrow \text{real}$
assumes $f \sim[F] g$ *eventually* $(\lambda x. f\ x \geq 0)\ F$ *eventually* $(\lambda x. g\ x \geq 0)\ F$
shows $(\lambda x. f\ x \text{ powr } y) \sim[F] (\lambda x. g\ x \text{ powr } y)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-norm* [*asympt-equiv-intros*]:
fixes $f\ g :: 'a \Rightarrow 'b :: \text{real-normed-field}$
assumes $f \sim[F] g$
shows $(\lambda x. \text{norm}\ (f\ x)) \sim[F] (\lambda x. \text{norm}\ (g\ x))$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-abs-real* [*asympt-equiv-intros*]:

fixes $f\ g :: 'a \Rightarrow \text{real}$
assumes $f \sim[F] g$
shows $(\lambda x. |f\ x|) \sim[F] (\lambda x. |g\ x|)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-eventually-le*:

assumes $f \sim[F] g$ $c > 1$
shows *eventually* $(\lambda x. \text{norm}\ (f\ x) \leq c * \text{norm}\ (g\ x))\ F$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-eventually-ge*:

assumes $f \sim[F] g$ $c < 1$
shows *eventually* $(\lambda x. \text{norm}\ (f\ x) \geq c * \text{norm}\ (g\ x))\ F$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-bigo*:

assumes $f \sim[F] g$
shows $f \in O[F](g)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-bigomega*:

$f \sim[F] g \implies f \in \Omega[F](g)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-imp-bigtheta*:

$f \sim[F] g \implies f \in \Theta[F](g)$
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-at-infinity-transfer*:

assumes $f \sim[F] g$ *filterlim* f *at-infinity* F
shows *filterlim* g *at-infinity* F
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-at-top-transfer*:

fixes $f\ g :: - \Rightarrow \text{real}$
assumes $f \sim[F] g$ *filterlim* f *at-top* F
shows *filterlim* g *at-top* F
 $\langle \text{proof} \rangle$

lemma *asympt-equiv-at-bot-transfer*:

fixes $f\ g :: - \Rightarrow \text{real}$
assumes $f \sim[F] g$ *filterlim* f *at-bot* F
shows *filterlim* g *at-bot* F
 $\langle \text{proof} \rangle$

lemma *asympt-equivI'-const*:

assumes $(\lambda x. f\ x / g\ x) \longrightarrow c$ $F\ c \neq 0$

shows $f \sim[F] (\lambda x. c * g x)$
 $\langle proof \rangle$

lemma *asympt-equivI'-inverse-const:*

assumes $((\lambda x. f x / g x) \longrightarrow \text{inverse } c) \ F \ c \neq 0$
shows $(\lambda x. c * f x) \sim[F] g$
 $\langle proof \rangle$

lemma *filterlim-at-bot-imp-at-infinity:* $\text{filterlim } f \text{ at-bot } F \implies \text{filterlim } f \text{ at-infinity } F$

for $f :: - \Rightarrow \text{real}$ $\langle proof \rangle$

lemma *asympt-equiv-imp-diff-smallo:*

assumes $f \sim[F] g$
shows $(\lambda x. f x - g x) \in o[F](g)$
 $\langle proof \rangle$

lemma *asympt-equiv-altdef:*

$f \sim[F] g \longleftrightarrow (\lambda x. f x - g x) \in o[F](g)$
 $\langle proof \rangle$

lemma *asympt-equiv-0-left-iff [simp]:* $(\lambda -. 0) \sim[F] f \longleftrightarrow \text{eventually } (\lambda x. f x = 0) \ F$

and *asympt-equiv-0-right-iff [simp]:* $f \sim[F] (\lambda -. 0) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) \ F$
 $\langle proof \rangle$

lemma *asympt-equiv-sandwich-real:*

fixes $f g l u :: 'a \Rightarrow \text{real}$
assumes $l \sim[F] g \ u \sim[F] g \text{ eventually } (\lambda x. f x \in \{l x..u x\}) \ F$
shows $f \sim[F] g$
 $\langle proof \rangle$

lemma *asympt-equiv-sandwich-real':*

fixes $f g l u :: 'a \Rightarrow \text{real}$
assumes $f \sim[F] l \ f \sim[F] u \text{ eventually } (\lambda x. g x \in \{l x..u x\}) \ F$
shows $f \sim[F] g$
 $\langle proof \rangle$

lemma *asympt-equiv-sandwich-real'':*

fixes $f g l u :: 'a \Rightarrow \text{real}$
assumes $l1 \sim[F] u1 \ u1 \sim[F] l2 \ l2 \sim[F] u2$
 $\text{eventually } (\lambda x. f x \in \{l1 x..u1 x\}) \ F \text{ eventually } (\lambda x. g x \in \{l2 x..u2 x\}) \ F$
shows $f \sim[F] g$
 $\langle proof \rangle$

end

55 Values extended by a bottom element

theory *Lattice-Constructions*

imports *Main*

begin

datatype *'a bot* = *Value 'a* | *Bot*

instantiation *bot* :: (*preorder*) *preorder*

begin

definition *less-eq-bot* **where**

$x \leq y \longleftrightarrow (\text{case } x \text{ of } Bot \Rightarrow True \mid \text{Value } x \Rightarrow (\text{case } y \text{ of } Bot \Rightarrow False \mid \text{Value } y \Rightarrow x \leq y))$

definition *less-bot* **where**

$x < y \longleftrightarrow (\text{case } y \text{ of } Bot \Rightarrow False \mid \text{Value } y \Rightarrow (\text{case } x \text{ of } Bot \Rightarrow True \mid \text{Value } x \Rightarrow x < y))$

lemma *less-eq-bot-Bot* [*simp*]: $Bot \leq x$

<proof>

lemma *less-eq-bot-Bot-code* [*code*]: $Bot \leq x \longleftrightarrow True$

<proof>

lemma *less-eq-bot-Bot-is-Bot*: $x \leq Bot \implies x = Bot$

<proof>

lemma *less-eq-bot-Value-Bot* [*simp*, *code*]: $\text{Value } x \leq Bot \longleftrightarrow False$

<proof>

lemma *less-eq-bot-Value* [*simp*, *code*]: $\text{Value } x \leq \text{Value } y \longleftrightarrow x \leq y$

<proof>

lemma *less-bot-Bot* [*simp*, *code*]: $x < Bot \longleftrightarrow False$

<proof>

lemma *less-bot-Bot-is-Value*: $Bot < x \implies \exists z. x = \text{Value } z$

<proof>

lemma *less-bot-Bot-Value* [*simp*]: $Bot < \text{Value } x$

<proof>

lemma *less-bot-Bot-Value-code* [*code*]: $Bot < \text{Value } x \longleftrightarrow True$

<proof>

lemma *less-bot-Value* [*simp*, *code*]: $\text{Value } x < \text{Value } y \longleftrightarrow x < y$

<proof>

```

instance
   $\langle proof \rangle$ 

end

instance bot :: (order) order
   $\langle proof \rangle$ 

instance bot :: (linorder) linorder
   $\langle proof \rangle$ 

instantiation bot :: (order) bot
begin
  definition bot = Bot
  instance  $\langle proof \rangle$ 
end

instantiation bot :: (top) top
begin
  definition top = Value top
  instance  $\langle proof \rangle$ 
end

instantiation bot :: (semilattice-inf) semilattice-inf
begin

definition inf-bot
where
  inf x y =
    (case x of
      Bot  $\Rightarrow$  Bot
    | Value v  $\Rightarrow$ 
      (case y of
        Bot  $\Rightarrow$  Bot
      | Value v'  $\Rightarrow$  Value (inf v v')))

instance
   $\langle proof \rangle$ 

end

instantiation bot :: (semilattice-sup) semilattice-sup
begin

definition sup-bot
where
  sup x y =
    (case x of
      Bot  $\Rightarrow$  y

```

```

| Value v ⇒
  (case y of
    Bot ⇒ x
  | Value v' ⇒ Value (sup v v'))

```

instance
 ⟨proof⟩

end

instance bot :: (lattice) bounded-lattice-bot
 ⟨proof⟩

55.1 Values extended by a top element

datatype 'a top = Value 'a | Top

instantiation top :: (preorder) preorder
begin

definition less-eq-top **where**

$$x \leq y \longleftrightarrow (\text{case } y \text{ of } Top \Rightarrow True \mid \text{Value } y \Rightarrow (\text{case } x \text{ of } Top \Rightarrow False \mid \text{Value } x \Rightarrow x \leq y))$$

definition less-top **where**

$$x < y \longleftrightarrow (\text{case } x \text{ of } Top \Rightarrow False \mid \text{Value } x \Rightarrow (\text{case } y \text{ of } Top \Rightarrow True \mid \text{Value } y \Rightarrow x < y))$$

lemma less-eq-top-Top [simp]: $x \leq Top$
 ⟨proof⟩

lemma less-eq-top-Top-code [code]: $x \leq Top \longleftrightarrow True$
 ⟨proof⟩

lemma less-eq-top-is-Top: $Top \leq x \Longrightarrow x = Top$
 ⟨proof⟩

lemma less-eq-top-Top-Value [simp, code]: $Top \leq \text{Value } x \longleftrightarrow False$
 ⟨proof⟩

lemma less-eq-top-Value-Value [simp, code]: $\text{Value } x \leq \text{Value } y \longleftrightarrow x \leq y$
 ⟨proof⟩

lemma less-top-Top [simp, code]: $Top < x \longleftrightarrow False$
 ⟨proof⟩

lemma less-top-Top-is-Value: $x < Top \Longrightarrow \exists z. x = \text{Value } z$
 ⟨proof⟩

lemma *less-top-Value-Top* [*simp*]: *Value x < Top*
 ⟨*proof*⟩

lemma *less-top-Value-Top-code* [*code*]: *Value x < Top* \longleftrightarrow *True*
 ⟨*proof*⟩

lemma *less-top-Value* [*simp*, *code*]: *Value x < Value y* \longleftrightarrow *x < y*
 ⟨*proof*⟩

instance
 ⟨*proof*⟩

end

instance *top* :: (*order*) *order*
 ⟨*proof*⟩

instance *top* :: (*linorder*) *linorder*
 ⟨*proof*⟩

instantiation *top* :: (*order*) *top*
begin
definition *top* = *Top*
instance ⟨*proof*⟩
end

instantiation *top* :: (*bot*) *bot*
begin
definition *bot* = *Value bot*
instance ⟨*proof*⟩
end

instantiation *top* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-top*
where
inf x y =
 (*case x of*
 Top \Rightarrow *y*
 | *Value v* \Rightarrow
 (*case y of*
 Top \Rightarrow *x*
 | *Value v'* \Rightarrow *Value (inf v v')*))

instance
 ⟨*proof*⟩

end

instantiation *top* :: (*semilattice-sup*) *semilattice-sup*
begin

definition *sup-top*
where

$$\begin{aligned} \text{sup } x \ y = & \\ & (\text{case } x \text{ of} \\ & \quad \text{Top} \Rightarrow \text{Top} \\ & | \text{Value } v \Rightarrow \\ & \quad (\text{case } y \text{ of} \\ & \quad \quad \text{Top} \Rightarrow \text{Top} \\ & \quad | \text{Value } v' \Rightarrow \text{Value } (\text{sup } v \ v')))) \end{aligned}$$

instance
 ⟨*proof*⟩

end

instance *top* :: (*lattice*) *bounded-lattice-top*
 ⟨*proof*⟩

55.2 Values extended by a top and a bottom element

datatype *'a flat-complete-lattice* = *Value 'a* | *Bot* | *Top*

instantiation *flat-complete-lattice* :: (*type*) *order*
begin

definition *less-eq-flat-complete-lattice*
where

$$\begin{aligned} x \leq y \equiv & \\ & (\text{case } x \text{ of} \\ & \quad \text{Bot} \Rightarrow \text{True} \\ & | \text{Value } v1 \Rightarrow \\ & \quad (\text{case } y \text{ of} \\ & \quad \quad \text{Bot} \Rightarrow \text{False} \\ & \quad | \text{Value } v2 \Rightarrow v1 = v2 \\ & \quad | \text{Top} \Rightarrow \text{True}) \\ & | \text{Top} \Rightarrow y = \text{Top}) \end{aligned}$$

definition *less-flat-complete-lattice*
where

$$\begin{aligned} x < y = & \\ & (\text{case } x \text{ of} \\ & \quad \text{Bot} \Rightarrow y \neq \text{Bot} \\ & | \text{Value } v1 \Rightarrow y = \text{Top} \\ & | \text{Top} \Rightarrow \text{False}) \end{aligned}$$

lemma *[simp]*: $Bot \leq y$
 $\langle proof \rangle$

lemma *[simp]*: $y \leq Top$
 $\langle proof \rangle$

lemma *greater-than-two-values*:
assumes $a \neq b$ *Value* $a \leq z$ *Value* $b \leq z$
shows $z = Top$
 $\langle proof \rangle$

lemma *lesser-than-two-values*:
assumes $a \neq b$ $z \leq \text{Value } a$ $z \leq \text{Value } b$
shows $z = Bot$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

instantiation *flat-complete-lattice* :: (type) *bot*
begin
definition *bot* = *Bot*
instance $\langle proof \rangle$
end

instantiation *flat-complete-lattice* :: (type) *top*
begin
definition *top* = *Top*
instance $\langle proof \rangle$
end

instantiation *flat-complete-lattice* :: (type) *lattice*
begin

definition *inf-flat-complete-lattice*
where
 $inf\ x\ y =$
 $(case\ x\ of$
 $\quad Bot \Rightarrow Bot$
 $\mid Value\ v1 \Rightarrow$
 $\quad (case\ y\ of$
 $\quad\quad Bot \Rightarrow Bot$
 $\quad\mid Value\ v2 \Rightarrow if\ v1 = v2\ then\ x\ else\ Bot$
 $\quad\mid Top \Rightarrow x)$
 $\mid Top \Rightarrow y)$

definition *sup-flat-complete-lattice*

where

```

sup x y =
  (case x of
    Bot ⇒ y
  | Value v1 ⇒
    (case y of
      Bot ⇒ x
    | Value v2 ⇒ if v1 = v2 then x else Top
    | Top ⇒ Top)
  | Top ⇒ Top)

```

instance

⟨proof⟩

end

instantiation *flat-complete-lattice* :: (type) complete-lattice
begin

definition *Sup-flat-complete-lattice*

where

```

Sup A =
  (if A = {} ∨ A = {Bot} then Bot
   else if ∃ v. A - {Bot} = {Value v} then Value (THE v. A - {Bot} = {Value
v})
   else Top)

```

definition *Inf-flat-complete-lattice*

where

```

Inf A =
  (if A = {} ∨ A = {Top} then Top
   else if ∃ v. A - {Top} = {Value v} then Value (THE v. A - {Top} = {Value
v})
   else Bot)

```

instance

⟨proof⟩

end

end

56 Infinite Streams

theory *Stream*

imports *Nat-Bijection*

begin

codatatype (*sset: 'a*) *stream* =

SCons (*shd*: 'a) (*stl*: 'a stream) (**infixr** <##> 65)

for

map: *smap*

rel: *stream-all2*

context

begin

— for code generation only

qualified definition *smember* :: 'a \Rightarrow 'a stream \Rightarrow bool **where**

[*code-abbrev*]: *smember* *x s* $\longleftrightarrow x \in \text{sset } s$

lemma *smember-code*[*code*, *simp*]: *smember* *x* (*y* ## *s*) = (if *x* = *y* then True else *smember* *x s*)

<proof>

end

lemmas *smap-simps*[*simp*] = *stream.map-sel*

lemmas *shd-sset* = *stream.set-sel*(1)

lemmas *stl-sset* = *stream.set-sel*(2)

theorem *sset-induct*[*consumes* 1, *case-names* *shd stl*, *induct* *set*: *sset*]:

assumes *y* $\in \text{sset } s$ **and** $\bigwedge s. P (\text{shd } s) s$ **and** $\bigwedge s y. \llbracket y \in \text{sset } (\text{stl } s); P y (\text{stl } s) \rrbracket$
 $\implies P y s$

shows *P y s*

<proof>

lemma *smap-ctr*: *smap* *f s* = *x* ## *s'* $\longleftrightarrow f (\text{shd } s) = x \wedge \text{smap } f (\text{stl } s) = s'$

<proof>

56.1 prepend list to stream

primrec *shift* :: 'a list \Rightarrow 'a stream \Rightarrow 'a stream (**infixr** <@-> 65) **where**

shift [] *s* = *s*

| *shift* (*x* # *xs*) *s* = *x* ## *shift* *xs s*

lemma *smap-shift*[*simp*]: *smap* *f* (*xs* @- *s*) = *map* *f* *xs* @- *smap* *f s*

<proof>

lemma *shift-append*[*simp*]: (*xs* @ *ys*) @- *s* = *xs* @- *ys* @- *s*

<proof>

lemma *shift-simps*[*simp*]:

shd (*xs* @- *s*) = (if *xs* = [] then *shd s* else *hd xs*)

stl (*xs* @- *s*) = (if *xs* = [] then *stl s* else *tl xs* @- *s*)

<proof>

lemma *sset-shift*[*simp*]: *sset* (*xs* @- *s*) = *set* *xs* \cup *sset s*

$\langle proof \rangle$

lemma *shift-left-inj*[*simp*]: $xs @- s1 = xs @- s2 \longleftrightarrow s1 = s2$
 $\langle proof \rangle$

56.2 set of streams with elements in some fixed set

context

notes [[*inductive-internals*]]

begin

coinductive-set

streams :: 'a set \Rightarrow 'a stream set

for *A* :: 'a set

where

Stream[*intro!*, *simp*, *no-atp*]: $\llbracket a \in A; s \in \text{streams } A \rrbracket \Longrightarrow a \#\# s \in \text{streams } A$

end

lemma *in-streams*: $stl\ s \in \text{streams } S \Longrightarrow shd\ s \in S \Longrightarrow s \in \text{streams } S$
 $\langle proof \rangle$

lemma *streamsE*: $s \in \text{streams } A \Longrightarrow (shd\ s \in A \Longrightarrow stl\ s \in \text{streams } A \Longrightarrow P) \Longrightarrow P$
 $\langle proof \rangle$

lemma *Stream-image*: $x \#\# y \in ((\#\#) x') \text{ ' } Y \longleftrightarrow x = x' \wedge y \in Y$
 $\langle proof \rangle$

lemma *shift-streams*: $\llbracket w \in \text{lists } A; s \in \text{streams } A \rrbracket \Longrightarrow w @- s \in \text{streams } A$
 $\langle proof \rangle$

lemma *streams-Stream*: $x \#\# s \in \text{streams } A \longleftrightarrow x \in A \wedge s \in \text{streams } A$
 $\langle proof \rangle$

lemma *streams-stl*: $s \in \text{streams } A \Longrightarrow stl\ s \in \text{streams } A$
 $\langle proof \rangle$

lemma *streams-shd*: $s \in \text{streams } A \Longrightarrow shd\ s \in A$
 $\langle proof \rangle$

lemma *sset-streams*:

assumes *sset* $s \subseteq A$

shows $s \in \text{streams } A$

$\langle proof \rangle$

lemma *streams-sset*:

assumes $s \in \text{streams } A$

shows *sset* $s \subseteq A$

$\langle \text{proof} \rangle$

lemma *streams-iff-sset*: $s \in \text{streams } A \longleftrightarrow \text{sset } s \subseteq A$
 $\langle \text{proof} \rangle$

lemma *streams-mono*: $s \in \text{streams } A \implies A \subseteq B \implies s \in \text{streams } B$
 $\langle \text{proof} \rangle$

lemma *streams-mono2*: $S \subseteq T \implies \text{streams } S \subseteq \text{streams } T$
 $\langle \text{proof} \rangle$

lemma *smap-streams*: $s \in \text{streams } A \implies (\bigwedge x. x \in A \implies f x \in B) \implies \text{smap } f s \in \text{streams } B$
 $\langle \text{proof} \rangle$

lemma *streams-empty*: $\text{streams } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *streams-UNIV[simp]*: $\text{streams } \text{UNIV} = \text{UNIV}$
 $\langle \text{proof} \rangle$

56.3 nth, take, drop for streams

primrec *snth* :: 'a stream \Rightarrow nat \Rightarrow 'a (**infixl** $\langle !! \rangle$ 100) **where**
 $s !! 0 = \text{shd } s$
 $| s !! \text{Suc } n = \text{stl } s !! n$

lemma *snth-Stream*: $(x \#\# s) !! \text{Suc } i = s !! i$
 $\langle \text{proof} \rangle$

lemma *snth-smap[simp]*: $\text{smap } f s !! n = f (s !! n)$
 $\langle \text{proof} \rangle$

lemma *shift-snth-less[simp]*: $p < \text{length } xs \implies (xs @- s) !! p = xs ! p$
 $\langle \text{proof} \rangle$

lemma *shift-snth-ge[simp]*: $p \geq \text{length } xs \implies (xs @- s) !! p = s !! (p - \text{length } xs)$
 $\langle \text{proof} \rangle$

lemma *shift-snth*: $(xs @- s) !! n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } s !! (n - \text{length } xs))$
 $\langle \text{proof} \rangle$

lemma *snth-sset[simp]*: $s !! n \in \text{sset } s$
 $\langle \text{proof} \rangle$

lemma *sset-range*: $\text{sset } s = \text{range } (\text{snth } s)$
 $\langle \text{proof} \rangle$

lemma *streams-iff-snth*: $s \in \text{streams } X \longleftrightarrow (\forall n. s !! n \in X)$
 ⟨proof⟩

lemma *snth-in*: $s \in \text{streams } X \implies s !! n \in X$
 ⟨proof⟩

primrec *stake* :: $\text{nat} \Rightarrow 'a \text{ stream} \Rightarrow 'a \text{ list}$ **where**
 stake 0 $s = []$
 | *stake* (Suc n) $s = \text{shd } s \# \text{stake } n (\text{stl } s)$

lemma *length-stake[simp]*: $\text{length } (\text{stake } n \ s) = n$
 ⟨proof⟩

lemma *stake-smap[simp]*: $\text{stake } n (\text{smap } f \ s) = \text{map } f (\text{stake } n \ s)$
 ⟨proof⟩

lemma *take-stake*: $\text{take } n (\text{stake } m \ s) = \text{stake } (\min n \ m) \ s$
 ⟨proof⟩

primrec *sdrop* :: $\text{nat} \Rightarrow 'a \text{ stream} \Rightarrow 'a \text{ stream}$ **where**
 sdrop 0 $s = s$
 | *sdrop* (Suc n) $s = \text{sdrop } n (\text{stl } s)$

lemma *sdrop-simps[simp]*:
 $\text{shd } (\text{sdrop } n \ s) = s !! n \ \text{stl } (\text{sdrop } n \ s) = \text{sdrop } (\text{Suc } n) \ s$
 ⟨proof⟩

lemma *sdrop-smap[simp]*: $\text{sdrop } n (\text{smap } f \ s) = \text{smap } f (\text{sdrop } n \ s)$
 ⟨proof⟩

lemma *sdrop-stl*: $\text{sdrop } n (\text{stl } s) = \text{stl } (\text{sdrop } n \ s)$
 ⟨proof⟩

lemma *drop-stake*: $\text{drop } n (\text{stake } m \ s) = \text{stake } (m - n) (\text{sdrop } n \ s)$
 ⟨proof⟩

lemma *stake-sdrop*: $\text{stake } n \ s @- \text{sdrop } n \ s = s$
 ⟨proof⟩

lemma *id-stake-snth-sdrop*:
 $s = \text{stake } i \ s @- s !! i \ \#\# \text{sdrop } (\text{Suc } i) \ s$
 ⟨proof⟩

lemma *smap-alt*: $\text{smap } f \ s = s' \longleftrightarrow (\forall n. f (s !! n) = s' !! n) \ (\text{is } ?L = ?R)$
 ⟨proof⟩

lemma *stake-invert-Nil[iff]*: $\text{stake } n \ s = [] \longleftrightarrow n = 0$
 ⟨proof⟩

lemma *sdrop-shift*: $\text{sdrop } i \ (w \ @- \ s) = \text{drop } i \ w \ @- \ \text{sdrop } (i - \text{length } w) \ s$
 $\langle \text{proof} \rangle$

lemma *stake-shift*: $\text{stake } i \ (w \ @- \ s) = \text{take } i \ w \ @ \ \text{stake } (i - \text{length } w) \ s$
 $\langle \text{proof} \rangle$

lemma *stake-add[simp]*: $\text{stake } m \ s \ @ \ \text{stake } n \ (\text{sdrop } m \ s) = \text{stake } (m + n) \ s$
 $\langle \text{proof} \rangle$

lemma *sdrop-add[simp]*: $\text{sdrop } n \ (\text{sdrop } m \ s) = \text{sdrop } (m + n) \ s$
 $\langle \text{proof} \rangle$

lemma *sdrop-snth*: $\text{sdrop } n \ s \ !! \ m = s \ !! \ (n + m)$
 $\langle \text{proof} \rangle$

partial-function (*tailrec*) *sdrop-while* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \ \text{stream} \Rightarrow 'a \ \text{stream}$
where

$\text{sdrop-while } P \ s = (\text{if } P \ (\text{shd } s) \text{ then } \text{sdrop-while } P \ (\text{stl } s) \text{ else } s)$

lemma *sdrop-while-SCons[code]*:
 $\text{sdrop-while } P \ (a \ ## \ s) = (\text{if } P \ a \text{ then } \text{sdrop-while } P \ s \text{ else } a \ ## \ s)$
 $\langle \text{proof} \rangle$

lemma *sdrop-while-sdrop-LEAST*:
assumes $\exists n. P \ (s \ !! \ n)$
shows $\text{sdrop-while } (\text{Not} \circ P) \ s = \text{sdrop } (\text{LEAST } n. P \ (s \ !! \ n)) \ s$
 $\langle \text{proof} \rangle$

primcorec *sfilter* **where**

$\text{shd } (\text{sfilter } P \ s) = \text{shd } (\text{sdrop-while } (\text{Not} \circ P) \ s)$
 $| \ \text{stl } (\text{sfilter } P \ s) = \text{sfilter } P \ (\text{stl } (\text{sdrop-while } (\text{Not} \circ P) \ s))$

lemma *sfilter-Stream*: $\text{sfilter } P \ (x \ ## \ s) = (\text{if } P \ x \text{ then } x \ ## \ \text{sfilter } P \ s \text{ else } \text{sfilter } P \ s)$
 $\langle \text{proof} \rangle$

56.4 unary predicates lifted to streams

definition *stream-all* $P \ s = (\forall p. P \ (s \ !! \ p))$

lemma *stream-all-iff[iff]*: $\text{stream-all } P \ s \longleftrightarrow \text{Ball } (\text{sset } s) \ P$
 $\langle \text{proof} \rangle$

lemma *stream-all-shift[simp]*: $\text{stream-all } P \ (xs \ @- \ s) = (\text{list-all } P \ xs \wedge \text{stream-all } P \ s)$
 $\langle \text{proof} \rangle$

lemma *stream-all-Stream*: $\text{stream-all } P \ (x \ ## \ X) \longleftrightarrow P \ x \wedge \text{stream-all } P \ X$
 $\langle \text{proof} \rangle$

56.5 recurring stream out of a list

primcorec $\text{cycle} :: 'a \text{ list} \Rightarrow 'a \text{ stream}$ **where**

$\text{shd } (\text{cycle } xs) = \text{hd } xs$
 $| \text{stl } (\text{cycle } xs) = \text{cycle } (\text{tl } xs @ [\text{hd } xs])$

lemma cycle-decomp : $u \neq [] \implies \text{cycle } u = u @- \text{cycle } u$
 $\langle \text{proof} \rangle$

lemma cycle-Cons[code] : $\text{cycle } (x \# xs) = x \#\# \text{cycle } (xs @ [x])$
 $\langle \text{proof} \rangle$

lemma cycle-rotated : $\llbracket v \neq []; \text{cycle } u = v @- s \rrbracket \implies \text{cycle } (\text{tl } u @ [\text{hd } u]) = \text{tl } v @- s$
 $\langle \text{proof} \rangle$

lemma stake-append : $\text{stake } n (u @- s) = \text{take } (\min (\text{length } u) n) u @ \text{stake } (n - \text{length } u) s$
 $\langle \text{proof} \rangle$

lemma $\text{stake-cycle-le[simp]}$:
assumes $u \neq [] \ n < \text{length } u$
shows $\text{stake } n (\text{cycle } u) = \text{take } n u$
 $\langle \text{proof} \rangle$

lemma $\text{stake-cycle-eq[simp]}$: $u \neq [] \implies \text{stake } (\text{length } u) (\text{cycle } u) = u$
 $\langle \text{proof} \rangle$

lemma $\text{sdrop-cycle-eq[simp]}$: $u \neq [] \implies \text{sdrop } (\text{length } u) (\text{cycle } u) = \text{cycle } u$
 $\langle \text{proof} \rangle$

lemma $\text{stake-cycle-eq-mod-0[simp]}$: $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies \text{stake } n (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u) u)$
 $\langle \text{proof} \rangle$

lemma $\text{sdrop-cycle-eq-mod-0[simp]}$: $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies \text{sdrop } n (\text{cycle } u) = \text{cycle } u$
 $\langle \text{proof} \rangle$

lemma stake-cycle : $u \neq [] \implies \text{stake } n (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u) u) @ \text{take } (n \bmod \text{length } u) u$
 $\langle \text{proof} \rangle$

lemma sdrop-cycle : $u \neq [] \implies \text{sdrop } n (\text{cycle } u) = \text{cycle } (\text{rotate } (n \bmod \text{length } u) u)$
 $\langle \text{proof} \rangle$

lemma sset-cycle[simp] :
assumes $xs \neq []$

shows $sset\ (cycle\ xs) = set\ xs$
 $\langle proof \rangle$

56.6 iterated application of a function

primcorec *siterate* **where**
 $shd\ (siterate\ f\ x) = x$
 $|\ stl\ (siterate\ f\ x) = siterate\ f\ (f\ x)$

lemma *stake-Suc*: $stake\ (Suc\ n)\ s = stake\ n\ s\ @\ [s\ !!\ n]$
 $\langle proof \rangle$

lemma *snth-siterate[simp]*: $siterate\ f\ x\ !!\ n = (f\ \frown\ n)\ x$
 $\langle proof \rangle$

lemma *sdrop-siterate[simp]*: $sdrop\ n\ (siterate\ f\ x) = siterate\ f\ ((f\ \frown\ n)\ x)$
 $\langle proof \rangle$

lemma *stake-siterate[simp]*: $stake\ n\ (siterate\ f\ x) = map\ (\lambda n. (f\ \frown\ n)\ x)\ [0\ ..<\ n]$
 $\langle proof \rangle$

lemma *sset-siterate*: $sset\ (siterate\ f\ x) = \{(f\ \frown\ n)\ x \mid n. True\}$
 $\langle proof \rangle$

lemma *smap-siterate*: $smap\ f\ (siterate\ f\ x) = siterate\ f\ (f\ x)$
 $\langle proof \rangle$

56.7 stream repeating a single element

abbreviation $sconst \equiv siterate\ id$

lemma *shift-replicate-sconst[simp]*: $replicate\ n\ x\ @- sconst\ x = sconst\ x$
 $\langle proof \rangle$

lemma *sset-sconst[simp]*: $sset\ (sconst\ x) = \{x\}$
 $\langle proof \rangle$

lemma *sconst-alt*: $s = sconst\ x \longleftrightarrow sset\ s = \{x\}$
 $\langle proof \rangle$

lemma *sconst-cycle*: $sconst\ x = cycle\ [x]$
 $\langle proof \rangle$

lemma *smap-sconst*: $smap\ f\ (sconst\ x) = sconst\ (f\ x)$
 $\langle proof \rangle$

lemma *sconst-streams*: $x \in A \implies sconst\ x \in streams\ A$
 $\langle proof \rangle$

lemma *streams-empty-iff*: $streams\ S = \{\} \longleftrightarrow S = \{\}$

$\langle proof \rangle$

56.8 stream of natural numbers

abbreviation $fromN \equiv siterate\ Suc$

abbreviation $nats \equiv fromN\ 0$

lemma $sset-fromN[simp]$: $sset\ (fromN\ n) = \{n\ ..\}$
 $\langle proof \rangle$

lemma $stream-smap-fromN$: $s = smap\ (\lambda j. let\ i = j - n\ in\ s\ !!\ i)\ (fromN\ n)$
 $\langle proof \rangle$

lemma $stream-smap-nats$: $s = smap\ (snth\ s)\ nats$
 $\langle proof \rangle$

56.9 flatten a stream of lists

primcorec $flat$ **where**

$shd\ (flat\ ws) = hd\ (shd\ ws)$
 $| stl\ (flat\ ws) = flat\ (if\ tl\ (shd\ ws) = []\ then\ stl\ ws\ else\ tl\ (shd\ ws)\ \#\# \ stl\ ws)$

lemma $flat-Cons[simp, code]$: $flat\ ((x\ \#\# \ xs)\ \#\# \ ws) = x\ \#\# \ flat\ (if\ xs = []\ then\ ws\ else\ xs\ \#\# \ ws)$
 $\langle proof \rangle$

lemma $flat-Stream[simp]$: $xs \neq [] \implies flat\ (xs\ \#\# \ ws) = xs\ @-\ flat\ ws$
 $\langle proof \rangle$

lemma $flat-unfold$: $shd\ ws \neq [] \implies flat\ ws = shd\ ws\ @-\ flat\ (stl\ ws)$
 $\langle proof \rangle$

lemma $flat-snth$: $\forall xs \in sset\ s. xs \neq [] \implies flat\ s\ !!\ n = (if\ n < length\ (shd\ s)\ then\ shd\ s\ !\ n\ else\ flat\ (stl\ s)\ !!\ (n - length\ (shd\ s)))$
 $\langle proof \rangle$

lemma $sset-flat[simp]$: $\forall xs \in sset\ s. xs \neq [] \implies$
 $sset\ (flat\ s) = (\bigcup xs \in sset\ s. set\ xs)\ (is\ ?P \implies ?L = ?R)$
 $\langle proof \rangle$

56.10 merge a stream of streams

definition $smerge :: 'a\ stream\ stream \Rightarrow 'a\ stream$ **where**

$smerge\ ss = flat\ (smap\ (\lambda n. map\ (\lambda s. s\ !!\ n)\ (stake\ (Suc\ n)\ ss)\ @\ stake\ n\ (ss\ !!\ n))\ nats)$

lemma $stake-nth[simp]$: $m < n \implies stake\ n\ s\ !\ m = s\ !!\ m$
 $\langle proof \rangle$

lemma *snth-sset-smerge*: $ss \text{ !! } n \text{ !! } m \in \text{sset } (\text{smerge } ss)$
 $\langle \text{proof} \rangle$

lemma *sset-smerge*: $\text{sset } (\text{smerge } ss) = \bigcup (\text{sset } ' (\text{sset } ss))$
 $\langle \text{proof} \rangle$

56.11 product of two streams

definition *sproduct* :: $'a \text{ stream} \Rightarrow 'b \text{ stream} \Rightarrow ('a \times 'b) \text{ stream}$ **where**
 $\text{sproduct } s1 \ s2 = \text{smerge } (\text{smap } (\lambda x. \text{smap } (\text{Pair } x) \ s2) \ s1)$

lemma *sset-sproduct*: $\text{sset } (\text{sproduct } s1 \ s2) = \text{sset } s1 \times \text{sset } s2$
 $\langle \text{proof} \rangle$

56.12 interleave two streams

primcorec *sinterleave* **where**
 $\text{shd } (\text{sinterleave } s1 \ s2) = \text{shd } s1$
 $| \text{stl } (\text{sinterleave } s1 \ s2) = \text{sinterleave } s2 \ (\text{stl } s1)$

lemma *sinterleave-code*[code]:
 $\text{sinterleave } (x \ \#\# \ s1) \ s2 = x \ \#\# \ \text{sinterleave } s2 \ s1$
 $\langle \text{proof} \rangle$

lemma *sinterleave-snth*[simp]:
 $\text{even } n \implies \text{sinterleave } s1 \ s2 \text{ !! } n = s1 \text{ !! } (n \text{ div } 2)$
 $\text{odd } n \implies \text{sinterleave } s1 \ s2 \text{ !! } n = s2 \text{ !! } (n \text{ div } 2)$
 $\langle \text{proof} \rangle$

lemma *sset-sinterleave*: $\text{sset } (\text{sinterleave } s1 \ s2) = \text{sset } s1 \cup \text{sset } s2$
 $\langle \text{proof} \rangle$

56.13 zip

primcorec *szip* **where**
 $\text{shd } (\text{szip } s1 \ s2) = (\text{shd } s1, \text{shd } s2)$
 $| \text{stl } (\text{szip } s1 \ s2) = \text{szip } (\text{stl } s1) \ (\text{stl } s2)$

lemma *szip-unfold*[code]: $\text{szip } (a \ \#\# \ s1) \ (b \ \#\# \ s2) = (a, b) \ \#\# \ (\text{szip } s1 \ s2)$
 $\langle \text{proof} \rangle$

lemma *snth-szip*[simp]: $\text{szip } s1 \ s2 \text{ !! } n = (s1 \text{ !! } n, s2 \text{ !! } n)$
 $\langle \text{proof} \rangle$

lemma *stake-szip*[simp]:
 $\text{stake } n \ (\text{szip } s1 \ s2) = \text{szip } (\text{stake } n \ s1) \ (\text{stake } n \ s2)$
 $\langle \text{proof} \rangle$

lemma *sdrop-szip*[simp]: $\text{sdrop } n \ (\text{szip } s1 \ s2) = \text{szip } (\text{sdrop } n \ s1) \ (\text{sdrop } n \ s2)$
 $\langle \text{proof} \rangle$

lemma *smap-szip-fst*:

smap ($\lambda x. f (fst\ x)$) (*szip* *s1* *s2*) = *smap* *f* *s1*
 ⟨*proof*⟩

lemma *smap-szip-snd*:

smap ($\lambda x. g (snd\ x)$) (*szip* *s1* *s2*) = *smap* *g* *s2*
 ⟨*proof*⟩

56.14 zip via function

primcorec *smap2* **where**

shd (*smap2* *f* *s1* *s2*) = *f* (*shd* *s1*) (*shd* *s2*)
 | *stl* (*smap2* *f* *s1* *s2*) = *smap2* *f* (*stl* *s1*) (*stl* *s2*)

lemma *smap2-unfold*[*code*]:

smap2 *f* (*a* ## *s1*) (*b* ## *s2*) = *f* *a* *b* ## (*smap2* *f* *s1* *s2*)
 ⟨*proof*⟩

lemma *smap2-szip*:

smap2 *f* *s1* *s2* = *smap* (*case-prod* *f*) (*szip* *s1* *s2*)
 ⟨*proof*⟩

lemma *smap-smap2*[*simp*]:

smap *f* (*smap2* *g* *s1* *s2*) = *smap2* ($\lambda x\ y. f (g\ x\ y)$) *s1* *s2*
 ⟨*proof*⟩

lemma *smap2-alt*:

(*smap2* *f* *s1* *s2* = *s*) = ($\forall n. f (s1\ !!\ n) (s2\ !!\ n) = s\ !!\ n$)
 ⟨*proof*⟩

lemma *snth-smap2*[*simp*]:

smap2 *f* *s1* *s2* !! *n* = *f* (*s1* !! *n*) (*s2* !! *n*)
 ⟨*proof*⟩

lemma *stake-smap2*[*simp*]:

stake *n* (*smap2* *f* *s1* *s2*) = *map* (*case-prod* *f*) (*zip* (*stake* *n* *s1*) (*stake* *n* *s2*))
 ⟨*proof*⟩

lemma *sdrop-smap2*[*simp*]:

sdrop *n* (*smap2* *f* *s1* *s2*) = *smap2* *f* (*sdrop* *n* *s1*) (*sdrop* *n* *s2*)
 ⟨*proof*⟩

end

57 List prefixes, suffixes, and homeomorphic embedding

```
theory Sublist
imports Main
begin
```

57.1 Prefix order on lists

```
definition prefix :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool
  where prefix xs ys  $\longleftrightarrow (\exists zs. ys = xs @ zs)$ 
```

```
definition strict-prefix :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool
  where strict-prefix xs ys  $\longleftrightarrow$  prefix xs ys  $\wedge$  xs  $\neq$  ys
```

```
global-interpretation prefix-order: ordering prefix strict-prefix
  <proof>
```

```
interpretation prefix-order: order prefix strict-prefix
  <proof>
```

```
global-interpretation prefix-bot: ordering-top  $\langle \lambda xs\ ys. prefix\ ys\ xs \rangle$   $\langle \lambda xs\ ys. strict-prefix\ ys\ xs \rangle$   $\langle [] \rangle$ 
  <proof>
```

```
interpretation prefix-bot: order-bot Nil prefix strict-prefix
  <proof>
```

```
lemma prefixI [intro?]: ys = xs @ zs  $\implies$  prefix xs ys
  <proof>
```

```
lemma prefixE [elim?]:
  assumes prefix xs ys
  obtains zs where ys = xs @ zs
  <proof>
```

```
lemma strict-prefixI' [intro?]: ys = xs @ z # zs  $\implies$  strict-prefix xs ys
  <proof>
```

```
lemma strict-prefixE' [elim?]:
  assumes strict-prefix xs ys
  obtains z zs where ys = xs @ z # zs
  <proof>
```

```
lemma strict-prefixI [intro?]: prefix xs ys  $\implies$  xs  $\neq$  ys  $\implies$  strict-prefix xs ys
  <proof>
```

```
lemma strict-prefixE [elim?]:
```

fixes $xs\ ys :: 'a\ list$
assumes $strict_prefix\ xs\ ys$
obtains $prefix\ xs\ ys$ **and** $xs \neq ys$
 $\langle proof \rangle$

57.2 Basic properties of prefixes

theorem $Nil_prefix\ [simp]: prefix\ []\ xs$
 $\langle proof \rangle$

theorem $prefix_Nil\ [simp]: (prefix\ xs\ []) = (xs = [])$
 $\langle proof \rangle$

lemma $prefix_snoc\ [simp]: prefix\ xs\ (ys\ @\ [y]) \longleftrightarrow xs = ys\ @\ [y] \vee prefix\ xs\ ys$
 $\langle proof \rangle$

lemma $Cons_prefix_Cons\ [simp]: prefix\ (x\ \# \ xs)\ (y\ \# \ ys) = (x = y \wedge prefix\ xs\ ys)$
 $\langle proof \rangle$

lemma $prefix_code\ [code]:$
 $prefix\ []\ xs \longleftrightarrow True$
 $prefix\ (x\ \# \ xs)\ [] \longleftrightarrow False$
 $prefix\ (x\ \# \ xs)\ (y\ \# \ ys) \longleftrightarrow x = y \wedge prefix\ xs\ ys$
 $\langle proof \rangle$

lemma $same_prefix_prefix\ [simp]: prefix\ (xs\ @\ ys)\ (xs\ @\ zs) = prefix\ ys\ zs$
 $\langle proof \rangle$

lemma $same_prefix_nil\ [simp]: prefix\ (xs\ @\ ys)\ xs = (ys = [])$
 $\langle proof \rangle$

lemma $prefix_prefix\ [simp]: prefix\ xs\ ys \implies prefix\ xs\ (ys\ @\ zs)$
 $\langle proof \rangle$

lemma $append_prefixD: prefix\ (xs\ @\ ys)\ zs \implies prefix\ xs\ zs$
 $\langle proof \rangle$

theorem $prefix_Cons: prefix\ xs\ (y\ \# \ ys) = (xs = [] \vee (\exists\ zs.\ xs = y\ \# \ zs \wedge prefix\ zs\ ys))$
 $\langle proof \rangle$

theorem $prefix_append:$
 $prefix\ xs\ (ys\ @\ zs) = (prefix\ xs\ ys \vee (\exists\ us.\ xs = ys\ @\ us \wedge prefix\ us\ zs))$
 $\langle proof \rangle$

lemma $append_one_prefix:$
 $prefix\ xs\ ys \implies length\ xs < length\ ys \implies prefix\ (xs\ @\ [ys\ !\ length\ xs])\ ys$
 $\langle proof \rangle$

theorem *prefix-length-le*: $\text{prefix } xs \ ys \implies \text{length } xs \leq \text{length } ys$
 ⟨proof⟩

lemma *prefix-same-cases*:
 $\text{prefix } (xs_1 :: 'a \ \text{list}) \ ys \implies \text{prefix } xs_2 \ ys \implies \text{prefix } xs_1 \ xs_2 \vee \text{prefix } xs_2 \ xs_1$
 ⟨proof⟩

lemma *prefix-length-prefix*:
 $\text{prefix } ps \ xs \implies \text{prefix } qs \ xs \implies \text{length } ps \leq \text{length } qs \implies \text{prefix } ps \ qs$
 ⟨proof⟩

lemma *set-mono-prefix*: $\text{prefix } xs \ ys \implies \text{set } xs \subseteq \text{set } ys$
 ⟨proof⟩

lemma *take-is-prefix*: $\text{prefix } (\text{take } n \ xs) \ xs$
 ⟨proof⟩

lemma *takeWhile-is-prefix*: $\text{prefix } (\text{takeWhile } P \ xs) \ xs$
 ⟨proof⟩

lemma *prefixeq-butlast*: $\text{prefix } (\text{butlast } xs) \ xs$
 ⟨proof⟩

lemma *prefix-map-rightE*:
 assumes $\text{prefix } xs \ (\text{map } f \ ys)$
 shows $\exists xs'. \text{prefix } xs' \ ys \wedge xs = \text{map } f \ xs'$
 ⟨proof⟩

lemma *map-mono-prefix*: $\text{prefix } xs \ ys \implies \text{prefix } (\text{map } f \ xs) \ (\text{map } f \ ys)$
 ⟨proof⟩

lemma *filter-mono-prefix*: $\text{prefix } xs \ ys \implies \text{prefix } (\text{filter } P \ xs) \ (\text{filter } P \ ys)$
 ⟨proof⟩

lemma *sorted-antimono-prefix*: $\text{prefix } xs \ ys \implies \text{sorted } ys \implies \text{sorted } xs$
 ⟨proof⟩

lemma *prefix-length-less*: $\text{strict-prefix } xs \ ys \implies \text{length } xs < \text{length } ys$
 ⟨proof⟩

lemma *prefix-snocD*: $\text{prefix } (xs@[x]) \ ys \implies \text{strict-prefix } xs \ ys$
 ⟨proof⟩

lemma *strict-prefix-simps* [simp, code]:
 $\text{strict-prefix } xs \ [] \longleftrightarrow \text{False}$
 $\text{strict-prefix } [] \ (x \# xs) \longleftrightarrow \text{True}$
 $\text{strict-prefix } (x \# xs) \ (y \# ys) \longleftrightarrow x = y \wedge \text{strict-prefix } xs \ ys$
 ⟨proof⟩

lemma *take-strict-prefix*: $\text{strict-prefix } xs \ ys \implies \text{strict-prefix } (\text{take } n \ xs) \ ys$
 <proof>

lemma *prefix-takeWhile*:
 assumes $\text{prefix } xs \ ys$
 shows $\text{prefix } (\text{takeWhile } P \ xs) \ (\text{takeWhile } P \ ys)$
 <proof>

lemma *prefix-dropWhile*:
 assumes $\text{prefix } xs \ ys$
 shows $\text{prefix } (\text{dropWhile } P \ xs) \ (\text{dropWhile } P \ ys)$
 <proof>

lemma *prefix-remdups-adj*:
 assumes $\text{prefix } xs \ ys$
 shows $\text{prefix } (\text{remdups-adj } xs) \ (\text{remdups-adj } ys)$
 <proof>

lemma *not-prefix-cases*:
 assumes $\text{pfx}: \neg \text{prefix } ps \ ls$
 obtains
 $(c1) \ ps \neq [] \text{ and } ls = []$
 $| \ (c2) \ a \ as \ x \ xs \text{ where } ps = a\#as \text{ and } ls = x\#xs \text{ and } x = a \text{ and } \neg \text{prefix } as \ xs$
 $| \ (c3) \ a \ as \ x \ xs \text{ where } ps = a\#as \text{ and } ls = x\#xs \text{ and } x \neq a$
 <proof>

lemma *not-prefix-induct* [consumes 1, case-names Nil Neg Eq]:
 assumes $np: \neg \text{prefix } ps \ ls$
 and $\text{base}: \bigwedge x \ xs. P \ (x\#xs) \ []$
 and $r1: \bigwedge x \ xs \ y \ ys. x \neq y \implies P \ (x\#xs) \ (y\#ys)$
 and $r2: \bigwedge x \ xs \ y \ ys. \llbracket x = y; \neg \text{prefix } xs \ ys; P \ xs \ ys \rrbracket \implies P \ (x\#xs) \ (y\#ys)$
 shows $P \ ps \ ls$ <proof>

57.3 Prefixes

primrec *prefixes* **where**
 $\text{prefixes } [] = [[]] \ |$
 $\text{prefixes } (x\#xs) = [] \ \# \ \text{map } ((\#) \ x) \ (\text{prefixes } xs)$

lemma *in-set-prefixes[simp]*: $xs \in \text{set } (\text{prefixes } ys) \longleftrightarrow \text{prefix } xs \ ys$
 <proof>

lemma *length-prefixes[simp]*: $\text{length } (\text{prefixes } xs) = \text{length } xs + 1$
 <proof>

lemma *distinct-prefixes [intro]*: $\text{distinct } (\text{prefixes } xs)$
 <proof>

lemma *prefixes-snoc [simp]*: $\text{prefixes } (xs@[x]) = \text{prefixes } xs \ @ \ [xs@[x]]$

$\langle proof \rangle$

lemma *prefixes-not-Nil* [simp]: $prefixes\ xs \neq []$
 $\langle proof \rangle$

lemma *hd-prefixes* [simp]: $hd\ (prefixes\ xs) = []$
 $\langle proof \rangle$

lemma *last-prefixes* [simp]: $last\ (prefixes\ xs) = xs$
 $\langle proof \rangle$

lemma *prefixes-append*:
 $prefixes\ (xs\ @\ ys) = prefixes\ xs\ @\ map\ (\lambda ys'.\ xs\ @\ ys')\ (tl\ (prefixes\ ys))$
 $\langle proof \rangle$

lemma *prefixes-eq-snoc*:
 $prefixes\ ys = xs\ @\ [x] \longleftrightarrow$
 $(ys = [] \wedge xs = [] \vee (\exists z\ zs.\ ys = zs@[z] \wedge xs = prefixes\ zs)) \wedge x = ys$
 $\langle proof \rangle$

lemma *prefixes-tailrec* [code]:
 $prefixes\ xs = rev\ (snd\ (foldl\ (\lambda(acc1,\ acc2)\ x.\ (x\#acc1,\ rev\ (x\#acc1)\#acc2))$
 $([],[])\ xs))$
 $\langle proof \rangle$

lemma *set-prefixes-eq*: $set\ (prefixes\ xs) = \{ys.\ prefix\ ys\ xs\}$
 $\langle proof \rangle$

lemma *card-set-prefixes* [simp]: $card\ (set\ (prefixes\ xs)) = Suc\ (length\ xs)$
 $\langle proof \rangle$

lemma *set-prefixes-append*:
 $set\ (prefixes\ (xs\ @\ ys)) = set\ (prefixes\ xs) \cup \{xs\ @\ ys' \mid ys'.\ ys' \in set\ (prefixes\ ys)\}$
 $\langle proof \rangle$

57.4 Longest Common Prefix

definition *Longest-common-prefix* :: 'a list set \Rightarrow 'a list **where**
Longest-common-prefix $L = (ARG-MAX\ length\ ps.\ \forall xs \in L.\ prefix\ ps\ xs)$

lemma *Longest-common-prefix-ex*: $L \neq \{\} \implies$
 $\exists ps.\ (\forall xs \in L.\ prefix\ ps\ xs) \wedge (\forall qs.\ (\forall xs \in L.\ prefix\ qs\ xs) \longrightarrow size\ qs \leq size\ ps)$
 $(is\ - \implies \exists ps.\ ?P\ L\ ps)$
 $\langle proof \rangle$

lemma *Longest-common-prefix-unique*:
 $\langle \exists! ps.\ (\forall xs \in L.\ prefix\ ps\ xs) \wedge (\forall qs.\ (\forall xs \in L.\ prefix\ qs\ xs) \longrightarrow length\ qs \leq$

length ps)›
 if $\langle L \neq \{\} \rangle$
 ‹proof›

lemma Longest-common-prefix-eq:

[[$L \neq \{\}$; $\forall xs \in L. \text{prefix } ps \ xs$;
 $\forall qs. (\forall xs \in L. \text{prefix } qs \ xs) \longrightarrow \text{size } qs \leq \text{size } ps$]]
 $\implies \text{Longest-common-prefix } L = ps$
 ‹proof›

lemma Longest-common-prefix-prefix:

$xs \in L \implies \text{prefix } (\text{Longest-common-prefix } L) \ xs$
 ‹proof›

lemma Longest-common-prefix-longest:

$L \neq \{\} \implies \forall xs \in L. \text{prefix } ps \ xs \implies \text{length } ps \leq \text{length } (\text{Longest-common-prefix } L)$
 ‹proof›

lemma Longest-common-prefix-max-prefix:

$L \neq \{\} \implies \forall xs \in L. \text{prefix } ps \ xs \implies \text{prefix } ps \ (\text{Longest-common-prefix } L)$
 ‹proof›

lemma Longest-common-prefix-Nil: $[] \in L \implies \text{Longest-common-prefix } L = []$

‹proof›

lemma Longest-common-prefix-image-Cons:

assumes $L \neq \{\}$

shows $\text{Longest-common-prefix } ((\#) \ x \ 'L) = x \ \# \ \text{Longest-common-prefix } L$
 ‹proof›

lemma Longest-common-prefix-eq-Cons: **assumes** $L \neq \{\}$ $[] \notin L$ $\forall xs \in L. \text{hd } xs = x$

shows $\text{Longest-common-prefix } L = x \ \# \ \text{Longest-common-prefix } \{ys. x \ \# \ ys \in L\}$
 ‹proof›

lemma Longest-common-prefix-eq-Nil:

[[$x \ \# \ ys \in L$; $y \ \# \ zs \in L$; $x \neq y$]] $\implies \text{Longest-common-prefix } L = []$
 ‹proof›

fun longest-common-prefix :: 'a list \Rightarrow 'a list \Rightarrow 'a list **where**

longest-common-prefix (x#xs) (y#ys) =
 (if x=y then x # longest-common-prefix xs ys else []) |
 longest-common-prefix - - = []

lemma longest-common-prefix-prefix1:

prefix (longest-common-prefix xs ys) xs
 ‹proof›

lemma *longest-common-prefix-prefix2*:
 $\text{prefix } (\text{longest-common-prefix } xs \ ys) \ ys$
 $\langle \text{proof} \rangle$

lemma *longest-common-prefix-max-prefix*:
 $\llbracket \text{prefix } ps \ xs; \text{prefix } ps \ ys \rrbracket$
 $\implies \text{prefix } ps \ (\text{longest-common-prefix } xs \ ys)$
 $\langle \text{proof} \rangle$

57.5 Parallel lists

definition *parallel* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ (**infixl** $\langle \parallel \rangle$ 50)
where $(xs \parallel ys) = (\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs)$

lemma *parallelI* [intro]: $\neg \text{prefix } xs \ ys \implies \neg \text{prefix } ys \ xs \implies xs \parallel ys$
 $\langle \text{proof} \rangle$

lemma *parallelE* [elim]:
assumes $xs \parallel ys$
obtains $\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs$
 $\langle \text{proof} \rangle$

theorem *prefix-cases*:
obtains $\text{prefix } xs \ ys \mid \text{strict-prefix } ys \ xs \mid xs \parallel ys$
 $\langle \text{proof} \rangle$

lemma *parallel-cancel*: $a \# xs \parallel a \# ys \implies xs \parallel ys$
 $\langle \text{proof} \rangle$

theorem *parallel-decomp*:
 $xs \parallel ys \implies \exists as \ b \ bs \ c \ cs. b \neq c \wedge xs = as @ b \# bs \wedge ys = as @ c \# cs$
 $\langle \text{proof} \rangle$

lemma *parallel-append*: $a \parallel b \implies a @ c \parallel b @ d$
 $\langle \text{proof} \rangle$

lemma *parallel-appendI*: $xs \parallel ys \implies x = xs @ xs' \implies y = ys @ ys' \implies x \parallel y$
 $\langle \text{proof} \rangle$

lemma *parallel-commute*: $a \parallel b \longleftrightarrow b \parallel a$
 $\langle \text{proof} \rangle$

57.6 Suffix order on lists

definition *suffix* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$
where $\text{suffix } xs \ ys = (\exists zs. ys = zs @ xs)$

definition *strict-suffix* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$
where $\text{strict-suffix } xs \ ys \longleftrightarrow \text{suffix } xs \ ys \wedge xs \neq ys$

global-interpretation *suffix-order: ordering suffix strict-suffix*
 $\langle proof \rangle$

interpretation *suffix-order: order suffix strict-suffix*
 $\langle proof \rangle$

global-interpretation *suffix-bot: ordering-top $\langle \lambda xs\ ys.\ suffix\ ys\ xs \rangle \langle \lambda xs\ ys.\ strict-suffix\ ys\ xs \rangle \langle [] \rangle$*
 $\langle proof \rangle$

interpretation *suffix-bot: order-bot Nil suffix strict-suffix*
 $\langle proof \rangle$

lemma *suffixI [intro?]: $ys = zs @ xs \implies suffix\ xs\ ys$*
 $\langle proof \rangle$

lemma *suffixE [elim?]:*
assumes *suffix xs ys*
obtains *zs where $ys = zs @ xs$*
 $\langle proof \rangle$

lemma *suffix-tl [simp]: $suffix\ (tl\ xs)\ xs$*
 $\langle proof \rangle$

lemma *strict-suffix-tl [simp]: $xs \neq [] \implies strict-suffix\ (tl\ xs)\ xs$*
 $\langle proof \rangle$

lemma *Nil-suffix [simp]: $suffix\ []\ xs$*
 $\langle proof \rangle$

lemma *suffix-Nil [simp]: $(suffix\ xs\ []) = (xs = [])$*
 $\langle proof \rangle$

lemma *suffix-ConsI: $suffix\ xs\ ys \implies suffix\ xs\ (y \# ys)$*
 $\langle proof \rangle$

lemma *suffix-ConsD: $suffix\ (x \# xs)\ ys \implies suffix\ xs\ ys$*
 $\langle proof \rangle$

lemma *suffix-appendI: $suffix\ xs\ ys \implies suffix\ xs\ (zs @ ys)$*
 $\langle proof \rangle$

lemma *suffix-appendD: $suffix\ (zs @ xs)\ ys \implies suffix\ xs\ ys$*
 $\langle proof \rangle$

lemma *strict-suffix-set-subset: $strict-suffix\ xs\ ys \implies set\ xs \subseteq set\ ys$*
 $\langle proof \rangle$

lemma *set-mono-suffix: $suffix\ xs\ ys \implies set\ xs \subseteq set\ ys$*

$\langle proof \rangle$

lemma *sorted-antimono-suffix*: $suffix\ xs\ ys \implies sorted\ ys \implies sorted\ xs$
 $\langle proof \rangle$

lemma *suffix-ConsD2*: $suffix\ (x \# xs)\ (y \# ys) \implies suffix\ xs\ ys$
 $\langle proof \rangle$

lemma *suffix-to-prefix* [code]: $suffix\ xs\ ys \longleftrightarrow prefix\ (rev\ xs)\ (rev\ ys)$
 $\langle proof \rangle$

lemma *strict-suffix-to-prefix* [code]: $strict-suffix\ xs\ ys \longleftrightarrow strict-prefix\ (rev\ xs)\ (rev\ ys)$
 $\langle proof \rangle$

lemma *distinct-suffix*: $distinct\ ys \implies suffix\ xs\ ys \implies distinct\ xs$
 $\langle proof \rangle$

lemma *map-mono-suffix*: $suffix\ xs\ ys \implies suffix\ (map\ f\ xs)\ (map\ f\ ys)$
 $\langle proof \rangle$

lemma *map-mono-strict-suffix*: $strict-suffix\ xs\ ys \implies strict-suffix\ (map\ f\ xs)\ (map\ f\ ys)$
 $\langle proof \rangle$

lemma *filter-mono-suffix*: $suffix\ xs\ ys \implies suffix\ (filter\ P\ xs)\ (filter\ P\ ys)$
 $\langle proof \rangle$

lemma *suffix-drop*: $suffix\ (drop\ n\ as)\ as$
 $\langle proof \rangle$

lemma *suffix-dropWhile*: $suffix\ (dropWhile\ P\ xs)\ xs$
 $\langle proof \rangle$

lemma *suffix-take*: $suffix\ xs\ ys \implies ys = take\ (length\ ys - length\ xs)\ ys @ xs$
 $\langle proof \rangle$

lemma *strict-suffix-reflclp-conv*: $strict-suffix^{==} = suffix$
 $\langle proof \rangle$

lemma *suffix-lists*: $suffix\ xs\ ys \implies ys \in lists\ A \implies xs \in lists\ A$
 $\langle proof \rangle$

lemma *suffix-snoc* [simp]: $suffix\ xs\ (ys @ [y]) \longleftrightarrow xs = [] \vee (\exists zs. xs = zs @ [y] \wedge suffix\ zs\ ys)$
 $\langle proof \rangle$

lemma *snoc-suffix-snoc* [simp]: $suffix\ (xs @ [x])\ (ys @ [y]) = (x = y \wedge suffix\ xs\ ys)$

$\langle \text{proof} \rangle$

lemma *same-suffix-suffix* [simp]: $\text{suffix } (ys @ xs) (zs @ xs) = \text{suffix } ys zs$
 $\langle \text{proof} \rangle$

lemma *same-suffix-nil* [simp]: $\text{suffix } (ys @ xs) xs = (ys = [])$
 $\langle \text{proof} \rangle$

theorem *suffix-Cons*: $\text{suffix } xs (y \# ys) \longleftrightarrow xs = y \# ys \vee \text{suffix } xs ys$
 $\langle \text{proof} \rangle$

theorem *suffix-append*:
 $\text{suffix } xs (ys @ zs) \longleftrightarrow \text{suffix } xs zs \vee (\exists xs'. xs = xs' @ zs \wedge \text{suffix } xs' ys)$
 $\langle \text{proof} \rangle$

theorem *suffix-length-le*: $\text{suffix } xs ys \implies \text{length } xs \leq \text{length } ys$
 $\langle \text{proof} \rangle$

lemma *suffix-same-cases*:
 $\text{suffix } (xs_1 :: 'a \text{ list}) ys \implies \text{suffix } xs_2 ys \implies \text{suffix } xs_1 xs_2 \vee \text{suffix } xs_2 xs_1$
 $\langle \text{proof} \rangle$

lemma *suffix-length-suffix*:
 $\text{suffix } ps xs \implies \text{suffix } qs xs \implies \text{length } ps \leq \text{length } qs \implies \text{suffix } ps qs$
 $\langle \text{proof} \rangle$

lemma *suffix-length-less*: $\text{strict-suffix } xs ys \implies \text{length } xs < \text{length } ys$
 $\langle \text{proof} \rangle$

lemma *suffix-ConsD'*: $\text{suffix } (x \# xs) ys \implies \text{strict-suffix } xs ys$
 $\langle \text{proof} \rangle$

lemma *drop-strict-suffix*: $\text{strict-suffix } xs ys \implies \text{strict-suffix } (\text{drop } n \text{ } xs) ys$
 $\langle \text{proof} \rangle$

lemma *suffix-map-rightE*:
assumes $\text{suffix } xs (\text{map } f \text{ } ys)$
shows $\exists xs'. \text{suffix } xs' ys \wedge xs = \text{map } f \text{ } xs'$
 $\langle \text{proof} \rangle$

lemma *suffix-remdups-adj*: $\text{suffix } xs ys \implies \text{suffix } (\text{remdups-adj } xs) (\text{remdups-adj } ys)$
 $\langle \text{proof} \rangle$

lemma *not-suffix-cases*:
assumes $\text{pfx}: \neg \text{suffix } ps \text{ } ls$
obtains
 (c1) $ps \neq []$ **and** $ls = []$
 | (c2) $a \text{ as } x \text{ xs}$ **where** $ps = \text{as} @ [a]$ **and** $ls = \text{xs} @ [x]$ **and** $x = a$ **and** $\neg \text{suffix } as$

xs
 | (*c3*) *a as x xs* **where** *ps = as@[a]* **and** *ls = xs@[x]* **and** *x ≠ a*
 ⟨*proof*⟩

lemma *not-suffix-induct* [*consumes 1, case-names Nil Neq Eq*]:

assumes *np*: $\neg \text{suffix } ps \text{ } ls$
and *base*: $\bigwedge x \text{ } xs. P \text{ } (xs@[x]) \text{ } []$
and *r1*: $\bigwedge x \text{ } xs \text{ } y \text{ } ys. x \neq y \implies P \text{ } (xs@[x]) \text{ } (ys@[y])$
and *r2*: $\bigwedge x \text{ } xs \text{ } y \text{ } ys. [x = y; \neg \text{suffix } xs \text{ } ys; P \text{ } xs \text{ } ys] \implies P \text{ } (xs@[x]) \text{ } (ys@[y])$
shows $P \text{ } ps \text{ } ls$ ⟨*proof*⟩

lemma *parallelD1*: $x \parallel y \implies \neg \text{prefix } x \text{ } y$
 ⟨*proof*⟩

lemma *parallelD2*: $x \parallel y \implies \neg \text{prefix } y \text{ } x$
 ⟨*proof*⟩

lemma *parallel-Nil1* [*simp*]: $\neg x \parallel []$
 ⟨*proof*⟩

lemma *parallel-Nil2* [*simp*]: $\neg [] \parallel x$
 ⟨*proof*⟩

lemma *Cons-parallelI1*: $a \neq b \implies a \# as \parallel b \# bs$
 ⟨*proof*⟩

lemma *Cons-parallelI2*: $[a = b; as \parallel bs] \implies a \# as \parallel b \# bs$
 ⟨*proof*⟩

lemma *not-equal-is-parallel*:
assumes *neq*: $xs \neq ys$
and *len*: $\text{length } xs = \text{length } ys$
shows $xs \parallel ys$
 ⟨*proof*⟩

57.7 Suffixes

primrec *suffixes* **where**

suffixes $[] = [[]]$
 | *suffixes* $(x \# xs) = \text{suffixes } xs \text{ } @ [x \# xs]$

lemma *in-set-suffixes* [*simp*]: $xs \in \text{set } (\text{suffixes } ys) \longleftrightarrow \text{suffix } xs \text{ } ys$
 ⟨*proof*⟩

lemma *distinct-suffixes* [*intro*]: $\text{distinct } (\text{suffixes } xs)$
 ⟨*proof*⟩

lemma *length-suffixes* [*simp*]: $\text{length } (\text{suffixes } xs) = \text{Suc } (\text{length } xs)$

$\langle proof \rangle$

lemma *suffixes-snoc* [simp]: $suffixes (xs @ [x]) = [] \# map (\lambda ys. ys @ [x]) (suffixes xs)$
 $\langle proof \rangle$

lemma *suffixes-not-Nil* [simp]: $suffixes xs \neq []$
 $\langle proof \rangle$

lemma *hd-suffixes* [simp]: $hd (suffixes xs) = []$
 $\langle proof \rangle$

lemma *last-suffixes* [simp]: $last (suffixes xs) = xs$
 $\langle proof \rangle$

lemma *suffixes-append*:
 $suffixes (xs @ ys) = suffixes ys @ map (\lambda xs'. xs' @ ys) (tl (suffixes xs))$
 $\langle proof \rangle$

lemma *suffixes-eq-snoc*:
 $suffixes ys = xs @ [x] \longleftrightarrow$
 $(ys = [] \wedge xs = [] \vee (\exists z zs. ys = z \# zs \wedge xs = suffixes zs)) \wedge x = ys$
 $\langle proof \rangle$

lemma *suffixes-tailrec* [code]:
 $suffixes xs = rev (snd (foldl (\lambda (acc1, acc2) x. (x \# acc1, (x \# acc1) \# acc2)) ([], []))$
 $(rev xs)))$
 $\langle proof \rangle$

lemma *set-suffixes-eq*: $set (suffixes xs) = \{ys. suffix ys xs\}$
 $\langle proof \rangle$

lemma *card-set-suffixes* [simp]: $card (set (suffixes xs)) = Suc (length xs)$
 $\langle proof \rangle$

lemma *set-suffixes-append*:
 $set (suffixes (xs @ ys)) = set (suffixes ys) \cup \{xs' @ ys \mid xs'. xs' \in set (suffixes xs)\}$
 $\langle proof \rangle$

lemma *suffixes-conv-prefixes*: $suffixes xs = map rev (prefixes (rev xs))$
 $\langle proof \rangle$

lemma *prefixes-conv-suffixes*: $prefixes xs = map rev (suffixes (rev xs))$
 $\langle proof \rangle$

lemma *prefixes-rev*: $prefixes (rev xs) = map rev (suffixes xs)$
 $\langle proof \rangle$

lemma *suffixes-rev*: $\text{suffixes } (\text{rev } xs) = \text{map rev } (\text{prefixes } xs)$
 ⟨proof⟩

57.8 Homeomorphic embedding on lists

inductive *list-emb* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$
for $P :: ('a \Rightarrow 'a \Rightarrow \text{bool})$

where

list-emb-Nil [intro, simp]: $\text{list-emb } P \ [] \ ys$
 | *list-emb-Cons* [intro] : $\text{list-emb } P \ xs \ ys \Longrightarrow \text{list-emb } P \ xs \ (y\#ys)$
 | *list-emb-Cons2* [intro]: $P \ x \ y \Longrightarrow \text{list-emb } P \ xs \ ys \Longrightarrow \text{list-emb } P \ (x\#xs) \ (y\#ys)$

lemma *list-emb-mono*:

assumes $\bigwedge x \ y. P \ x \ y \longrightarrow Q \ x \ y$
shows $\text{list-emb } P \ xs \ ys \longrightarrow \text{list-emb } Q \ xs \ ys$
 ⟨proof⟩

lemma *list-emb-Nil2* [simp]:

assumes $\text{list-emb } P \ xs \ []$ **shows** $xs = []$
 ⟨proof⟩

lemma *list-emb-refl*:

assumes $\bigwedge x. x \in \text{set } xs \Longrightarrow P \ x \ x$
shows $\text{list-emb } P \ xs \ xs$
 ⟨proof⟩

lemma *list-emb-Cons-Nil* [simp]: $\text{list-emb } P \ (x\#xs) \ [] = \text{False}$
 ⟨proof⟩

lemma *list-emb-append2* [intro]: $\text{list-emb } P \ xs \ ys \Longrightarrow \text{list-emb } P \ xs \ (zs @ ys)$
 ⟨proof⟩

lemma *list-emb-prefix* [intro]:

assumes $\text{list-emb } P \ xs \ ys$ **shows** $\text{list-emb } P \ xs \ (ys @ zs)$
 ⟨proof⟩

lemma *list-emb-ConsD*:

assumes $\text{list-emb } P \ (x\#xs) \ ys$
shows $\exists us \ v \ vs. ys = us @ v \ \# \ vs \wedge P \ x \ v \wedge \text{list-emb } P \ xs \ vs$
 ⟨proof⟩

lemma *list-emb-appendD*:

assumes $\text{list-emb } P \ (xs @ ys) \ zs$
shows $\exists us \ vs. zs = us @ vs \wedge \text{list-emb } P \ xs \ us \wedge \text{list-emb } P \ ys \ vs$
 ⟨proof⟩

lemma *list-emb-strict-suffix*:

assumes $\text{list-emb } P \ xs \ ys$ **and** *strict-suffix* $ys \ zs$

shows *list-emb* P xs zs
 $\langle proof \rangle$

lemma *list-emb-suffix*:
assumes *list-emb* P xs ys **and** *suffix* ys zs
shows *list-emb* P xs zs
 $\langle proof \rangle$

lemma *list-emb-length*: *list-emb* P xs $ys \implies \text{length } xs \leq \text{length } ys$
 $\langle proof \rangle$

lemma *list-emb-trans*:
assumes $\bigwedge x y z. \llbracket x \in \text{set } xs; y \in \text{set } ys; z \in \text{set } zs; P x y; P y z \rrbracket \implies P x z$
shows $\llbracket \text{list-emb } P xs ys; \text{list-emb } P ys zs \rrbracket \implies \text{list-emb } P xs zs$
 $\langle proof \rangle$

lemma *list-emb-set*:
assumes *list-emb* P xs ys **and** $x \in \text{set } xs$
obtains y **where** $y \in \text{set } ys$ **and** $P x y$
 $\langle proof \rangle$

lemma *list-emb-Cons-iff1* [simp]:
assumes $P x y$
shows *list-emb* P $(x\#xs)$ $(y\#ys) \longleftrightarrow \text{list-emb } P xs ys$
 $\langle proof \rangle$

lemma *list-emb-Cons-iff2* [simp]:
assumes $\neg P x y$
shows *list-emb* P $(x\#xs)$ $(y\#ys) \longleftrightarrow \text{list-emb } P (x\#xs) ys$
 $\langle proof \rangle$

lemma *list-emb-code* [code]:
list-emb P [] $ys \longleftrightarrow \text{True}$
list-emb P $(x\#xs)$ [] $\longleftrightarrow \text{False}$
list-emb P $(x\#xs)$ $(y\#ys) \longleftrightarrow (\text{if } P x y \text{ then } \text{list-emb } P xs ys \text{ else } \text{list-emb } P (x\#xs) ys)$
 $\langle proof \rangle$

57.9 Subsequences (special case of homeomorphic embedding)

abbreviation *subseq* :: 'a list \Rightarrow 'a list \Rightarrow bool
where *subseq* xs $ys \equiv \text{list-emb } (=) xs ys$

definition *strict-subseq* **where** *strict-subseq* xs $ys \longleftrightarrow xs \neq ys \wedge \text{subseq } xs ys$

lemma *subseq-Cons2*: *subseq* xs $ys \implies \text{subseq } (x\#xs) (x\#ys)$ $\langle proof \rangle$

lemma *subseq-same-length*:

assumes *subseq xs ys and length xs = length ys* **shows** *xs = ys*
 ⟨proof⟩

lemma *not-subseq-length* [simp]: *length ys < length xs $\implies \neg$ subseq xs ys*
 ⟨proof⟩

lemma *subseq-Cons'*: *subseq (x#xs) ys \implies subseq xs ys*
 ⟨proof⟩

lemma *subseq-Cons2'*:
assumes *subseq (x#xs) (y#ys)* **shows** *subseq xs ys*
 ⟨proof⟩

lemma *subseq-Cons2-neg*:
assumes *subseq (x#xs) (y#ys)*
shows *x \neq y \implies subseq (x#xs) ys*
 ⟨proof⟩

lemma *subseq-Cons2-iff* [simp]:
subseq (x#xs) (y#ys) = (if x = y then subseq xs ys else subseq (x#xs) ys)
 ⟨proof⟩

lemma *subseq-append'*: *subseq (zs @ xs) (zs @ ys) \longleftrightarrow subseq xs ys*
 ⟨proof⟩

global-interpretation *subseq-order*: *ordering subseq strict-subseq*
 ⟨proof⟩

interpretation *subseq-order*: *order subseq strict-subseq*
 ⟨proof⟩

lemma *in-set-subseqs* [simp]: *xs \in set (subseqs ys) \longleftrightarrow subseq xs ys*
 ⟨proof⟩

lemma *set-subseqs-eq*: *set (subseqs ys) = {xs. subseq xs ys}*
 ⟨proof⟩

lemma *subseq-append-le-same-iff*: *subseq (xs @ ys) ys \longleftrightarrow xs = []*
 ⟨proof⟩

lemma *subseq-singleton-left*: *subseq [x] ys \longleftrightarrow x \in set ys*
 ⟨proof⟩

lemma *list-emb-append-mono*:
 $\llbracket \text{list-emb } P \text{ xs xs'; list-emb } P \text{ ys ys'} \rrbracket \implies \text{list-emb } P \text{ (xs@ys) (xs'@ys')}$
 ⟨proof⟩

lemma *prefix-imp-subseq* [intro]: *prefix xs ys \implies subseq xs ys*
 ⟨proof⟩

lemma *suffix-imp-subseq* [intro]: $\text{suffix } xs \ ys \implies \text{subseq } xs \ ys$
 ⟨proof⟩

a subsequence of a sorted list

lemma *sorted-subset-imp-subseq*:
fixes $xs :: 'a::\text{order list}$
assumes $\text{set } xs \subseteq \text{set } ys$ *sorted-wrt* ($<$) xs *sorted-wrt* (\leq) ys
shows $\text{subseq } xs \ ys$
 ⟨proof⟩

57.10 Appending elements

lemma *subseq-append* [simp]:
 $\text{subseq } (xs @ zs) \ (ys @ zs) \longleftrightarrow \text{subseq } xs \ ys \ (\text{is } ?l = ?r)$
 ⟨proof⟩

lemma *subseq-append-iff*:
 $\text{subseq } xs \ (ys @ zs) \longleftrightarrow (\exists xs1 \ xs2. xs = xs1 @ xs2 \wedge \text{subseq } xs1 \ ys \wedge \text{subseq } xs2 \ zs)$
 (is ?lhs = ?rhs)
 ⟨proof⟩

lemma *subseq-appendE* [case-names append]:
assumes $\text{subseq } xs \ (ys @ zs)$
obtains $xs1 \ xs2$ **where** $xs = xs1 @ xs2$ $\text{subseq } xs1 \ ys$ $\text{subseq } xs2 \ zs$
 ⟨proof⟩

lemma *subseq-drop-many*: $\text{subseq } xs \ ys \implies \text{subseq } xs \ (zs @ ys)$
 ⟨proof⟩

lemma *subseq-rev-drop-many*: $\text{subseq } xs \ ys \implies \text{subseq } xs \ (ys @ zs)$
 ⟨proof⟩

57.11 Relation to standard list operations

lemma *subseq-map*:
assumes $\text{subseq } xs \ ys$ **shows** $\text{subseq } (\text{map } f \ xs) \ (\text{map } f \ ys)$
 ⟨proof⟩

lemma *subseq-filter-left* [simp]: $\text{subseq } (\text{filter } P \ xs) \ xs$
 ⟨proof⟩

lemma *subseq-filter* [simp]:
assumes $\text{subseq } xs \ ys$ **shows** $\text{subseq } (\text{filter } P \ xs) \ (\text{filter } P \ ys)$
 ⟨proof⟩

lemma *subseq-conv-nths*: $\text{subseq } xs \ ys \longleftrightarrow (\exists N. xs = \text{nths } ys \ N)$
 (is ?L = ?R)
 ⟨proof⟩

57.12 Contiguous sublists

57.12.1 *sublist*

definition *sublist* :: 'a list \Rightarrow 'a list \Rightarrow bool **where**
sublist *xs* *ys* = (\exists *ps ss*. *ys* = *ps* @ *xs* @ *ss*)

definition *strict-sublist* :: 'a list \Rightarrow 'a list \Rightarrow bool **where**
strict-sublist *xs* *ys* \longleftrightarrow *sublist* *xs* *ys* \wedge *xs* \neq *ys*

interpretation *sublist-order*: order *sublist* *strict-sublist*
 <proof>

lemma *sublist-Nil-left* [*simp*, *intro*]: *sublist* [] *ys*
 <proof>

lemma *sublist-Cons-Nil* [*simp*]: \neg *sublist* (*x* # *xs*) []
 <proof>

lemma *sublist-Nil-right* [*simp*]: *sublist* *xs* [] \longleftrightarrow *xs* = []
 <proof>

lemma *sublist-appendI* [*simp*, *intro*]: *sublist* *xs* (*ps* @ *xs* @ *ss*)
 <proof>

lemma *sublist-append-leftI* [*simp*, *intro*]: *sublist* *xs* (*ps* @ *xs*)
 <proof>

lemma *sublist-append-rightI* [*simp*, *intro*]: *sublist* *xs* (*xs* @ *ss*)
 <proof>

lemma *sublist-altdef*: *sublist* *xs* *ys* \longleftrightarrow (\exists *ys'*. *prefix* *ys'* *ys* \wedge *suffix* *xs* *ys'*)
 <proof>

lemma *sublist-altdef'*: *sublist* *xs* *ys* \longleftrightarrow (\exists *ys'*. *suffix* *ys'* *ys* \wedge *prefix* *xs* *ys'*)
 <proof>

lemma *sublist-Cons-right*: *sublist* *xs* (*y* # *ys*) \longleftrightarrow *prefix* *xs* (*y* # *ys*) \vee *sublist* *xs* *ys*
 <proof>

lemma *sublist-code* [*code*]:
sublist [] *ys* \longleftrightarrow *True*
sublist (*x* # *xs*) [] \longleftrightarrow *False*
sublist (*x* # *xs*) (*y* # *ys*) \longleftrightarrow *prefix* (*x* # *xs*) (*y* # *ys*) \vee *sublist* (*x* # *xs*) *ys*
 <proof>

lemma *sublist-append*:
sublist *xs* (*ys* @ *zs*) \longleftrightarrow
sublist *xs* *ys* \vee *sublist* *xs* *zs* \vee (\exists *xs1 xs2*. *xs* = *xs1* @ *xs2* \wedge *suffix* *xs1* *ys* \wedge

prefix xs2 zs)
 $\langle \text{proof} \rangle$

lemma *map-mono-sublist*:
 assumes *sublist xs ys*
 shows *sublist (map f xs) (map f ys)*
 $\langle \text{proof} \rangle$

lemma *sublist-length-le*: *sublist xs ys \implies length xs \leq length ys*
 $\langle \text{proof} \rangle$

lemma *set-mono-sublist*: *sublist xs ys \implies set xs \subseteq set ys*
 $\langle \text{proof} \rangle$

lemma *prefix-imp-sublist* [*simp, intro*]: *prefix xs ys \implies sublist xs ys*
 $\langle \text{proof} \rangle$

lemma *suffix-imp-sublist* [*simp, intro*]: *suffix xs ys \implies sublist xs ys*
 $\langle \text{proof} \rangle$

lemma *sublist-take* [*simp, intro*]: *sublist (take n xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-takeWhile* [*simp, intro*]: *sublist (takeWhile P xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-drop* [*simp, intro*]: *sublist (drop n xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-dropWhile* [*simp, intro*]: *sublist (dropWhile P xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-tl* [*simp, intro*]: *sublist (tl xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-butlast* [*simp, intro*]: *sublist (butlast xs) xs*
 $\langle \text{proof} \rangle$

lemma *sublist-rev* [*simp*]: *sublist (rev xs) (rev ys) = sublist xs ys*
 $\langle \text{proof} \rangle$

lemma *sublist-rev-left*: *sublist (rev xs) ys = sublist xs (rev ys)*
 $\langle \text{proof} \rangle$

lemma *sublist-rev-right*: *sublist xs (rev ys) = sublist (rev xs) ys*
 $\langle \text{proof} \rangle$

lemma *snoc-sublist-snoc*:
sublist (xs @ [x]) (ys @ [y]) \longleftrightarrow

$(x = y \wedge \text{suffix } xs \text{ } ys \vee \text{sublist } (xs @ [x]) \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *sublist-snoc*:

$\text{sublist } xs \text{ } (ys @ [y]) \longleftrightarrow \text{suffix } xs \text{ } (ys @ [y]) \vee \text{sublist } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *sublist-imp-subseq* [intro]: $\text{sublist } xs \text{ } ys \implies \text{subseq } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *sublist-map-rightE*:

assumes $\text{sublist } xs \text{ } (\text{map } f \text{ } ys)$
shows $\exists xs'. \text{sublist } xs' \text{ } ys \wedge xs = \text{map } f \text{ } xs'$
 $\langle \text{proof} \rangle$

lemma *sublist-remdups-adj*:

assumes $\text{sublist } xs \text{ } ys$
shows $\text{sublist } (\text{remdups-adj } xs) \text{ } (\text{remdups-adj } ys)$
 $\langle \text{proof} \rangle$

57.12.2 sublists

primrec *sublists* :: 'a list \Rightarrow 'a list list **where**

$\text{sublists } [] = [[]]$
 $| \text{sublists } (x \# xs) = \text{sublists } xs @ \text{map } ((\#) \text{ } x) \text{ } (\text{prefixes } xs)$

lemma *in-set-sublists* [simp]: $xs \in \text{set } (\text{sublists } ys) \longleftrightarrow \text{sublist } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *set-sublists-eq*: $\text{set } (\text{sublists } xs) = \{ys. \text{sublist } ys \text{ } xs\}$
 $\langle \text{proof} \rangle$

lemma *length-sublists* [simp]: $\text{length } (\text{sublists } xs) = \text{Suc } (\text{length } xs * \text{Suc } (\text{length } xs) \text{ div } 2)$
 $\langle \text{proof} \rangle$

57.13 Parametricity

context *includes lifting-syntax*
begin

private lemma *prefix-primrec*:

$\text{prefix} = \text{rec-list } (\lambda xs. \text{True}) \text{ } (\lambda x \text{ } xs \text{ } xsa \text{ } ys.$
 $\text{case } ys \text{ of } [] \Rightarrow \text{False} \mid y \# ys \Rightarrow x = y \wedge xsa \text{ } ys)$

$\langle \text{proof} \rangle$ **lemma** *sublist-primrec*:

$\text{sublist} = (\lambda xs \text{ } ys. \text{rec-list } (\lambda xs. xs = []) \text{ } (\lambda y \text{ } ys \text{ } ysa \text{ } xs. \text{prefix } xs \text{ } (y \# ys) \vee ysa \text{ } xs) \text{ } ys \text{ } xs)$

$\langle \text{proof} \rangle$ **lemma** *list-emb-primrec*:

$\text{list-emb} = (\lambda uu \text{ } l' \text{ } l. \text{rec-list } (\lambda P \text{ } xs. \text{List.null } xs) \text{ } (\lambda y \text{ } ys \text{ } ysa \text{ } P \text{ } xs. \text{case } xs \text{ of } [] \Rightarrow \text{True}$

$| x \# xs \Rightarrow \text{if } P \ x \ y \text{ then } \text{ysa } P \ xs \text{ else } \text{ysa } P \ (x \# xs)) \ l \ \text{uu } l'$
 $\langle \text{proof} \rangle$

lemma *prefix-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ prefix prefix}$
 $\langle \text{proof} \rangle$

lemma *suffix-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ suffix suffix}$
 $\langle \text{proof} \rangle$

lemma *sublist-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ sublist sublist}$
 $\langle \text{proof} \rangle$

lemma *parallel-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ parallel parallel}$
 $\langle \text{proof} \rangle$

lemma *list-emb-transfer* [transfer-rule]:
 $((A \implies A \implies (=)) \implies \text{list-all2 } A \implies \text{list-all2 } A \implies (=))$
list-emb list-emb
 $\langle \text{proof} \rangle$

lemma *strict-prefix-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ strict-prefix strict-prefix}$
 $\langle \text{proof} \rangle$

lemma *strict-suffix-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ strict-suffix strict-suffix}$
 $\langle \text{proof} \rangle$

lemma *strict-subseq-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ strict-subseq strict-subseq}$
 $\langle \text{proof} \rangle$

lemma *strict-sublist-transfer* [transfer-rule]:
assumes [transfer-rule]: *bi-unique A*
shows $(\text{list-all2 } A \implies \text{list-all2 } A \implies (=)) \text{ strict-sublist strict-sublist}$
 $\langle \text{proof} \rangle$

lemma *prefixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A == => list-all2 (list-all2 A)*) *prefixes prefixes*
 ⟨*proof*⟩

lemma *suffixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A == => list-all2 (list-all2 A)*) *suffixes suffixes*
 ⟨*proof*⟩

lemma *sublists-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A == => list-all2 (list-all2 A)*) *sublists sublists*
 ⟨*proof*⟩

end

end

58 Linear Temporal Logic on Streams

theory *Linear-Temporal-Logic-on-Streams*
imports *Stream Sublist Extended-Nat Infinite-Set*
begin

59 Preliminaries

lemma *shift-prefix*:
assumes *xl @- xs = yl @- ys* **and** *length xl ≤ length yl*
shows *prefix xl yl*
 ⟨*proof*⟩

lemma *shift-prefix-cases*:
assumes *xl @- xs = yl @- ys*
shows *prefix xl yl ∨ prefix yl xl*
 ⟨*proof*⟩

60 Linear temporal logic

Propositional connectives:

abbreviation (*input*) *IMPL* (**infix** ⟨*impl*⟩ 60)
where $\varphi \text{ impl } \psi \equiv \lambda xs. \varphi xs \longrightarrow \psi xs$

abbreviation (*input*) *OR* (**infix** ⟨*or*⟩ 60)
where $\varphi \text{ or } \psi \equiv \lambda xs. \varphi xs \vee \psi xs$

abbreviation (*input*) *AND* (**infix** $\langle \text{aand} \rangle$ 60)

where $\varphi \text{ aand } \psi \equiv \lambda xs. \varphi xs \wedge \psi xs$

abbreviation (*input*) *not* **where** $\text{not } \varphi \equiv \lambda xs. \neg \varphi xs$

abbreviation (*input*) *true* $\equiv \lambda xs. \text{True}$

abbreviation (*input*) *false* $\equiv \lambda xs. \text{False}$

lemma *impl-not-or*: $\varphi \text{ impl } \psi = (\text{not } \varphi) \text{ or } \psi$
 $\langle \text{proof} \rangle$

lemma *not-or*: $\text{not } (\varphi \text{ or } \psi) = (\text{not } \varphi) \text{ aand } (\text{not } \psi)$
 $\langle \text{proof} \rangle$

lemma *not-aand*: $\text{not } (\varphi \text{ aand } \psi) = (\text{not } \varphi) \text{ or } (\text{not } \psi)$
 $\langle \text{proof} \rangle$

lemma *non-not[simp]*: $\text{not } (\text{not } \varphi) = \varphi$ $\langle \text{proof} \rangle$

Temporal (LTL) connectives:

fun *holds* **where** $\text{holds } P xs \longleftrightarrow P (\text{shd } xs)$

fun *next* **where** $\text{next } \varphi xs = \varphi (\text{stl } xs)$

definition *HLD* $s = \text{holds } (\lambda x. x \in s)$

abbreviation *HLD-next* (**infixr** $\langle \cdot \rangle$ 65) **where**
 $s \cdot P \equiv \text{HLD } s \text{ aand } \text{next } P$

context

notes $[[\text{inductive-internals}]]$

begin

inductive *ev* **for** φ **where**

base: $\varphi xs \Longrightarrow \text{ev } \varphi xs$

|

step: $\text{ev } \varphi (\text{stl } xs) \Longrightarrow \text{ev } \varphi xs$

coinductive *alw* **for** φ **where**

alw: $\llbracket \varphi xs; \text{alw } \varphi (\text{stl } xs) \rrbracket \Longrightarrow \text{alw } \varphi xs$

— weak until:

coinductive *UNTIL* (**infix** $\langle \text{until} \rangle$ 60) **for** $\varphi \psi$ **where**

base: $\psi xs \Longrightarrow (\varphi \text{ until } \psi) xs$

|

step: $\llbracket \varphi xs; (\varphi \text{ until } \psi) (\text{stl } xs) \rrbracket \Longrightarrow (\varphi \text{ until } \psi) xs$

end

lemma *holds-mono*:

assumes *holds*: *holds* P xs **and** 0 : $\bigwedge x. P\ x \implies Q\ x$

shows *holds* Q xs

$\langle proof \rangle$

lemma *holds-aand*:

$(holds\ P\ aand\ holds\ Q) \ steps \longleftrightarrow holds\ (\lambda\ step. P\ step \wedge Q\ step) \ steps \ \langle proof \rangle$

lemma *HLD-iff*: $HLD\ s\ \omega \longleftrightarrow shd\ \omega \in s$

$\langle proof \rangle$

lemma *HLD-Stream[simp]*: $HLD\ X\ (x\ \#\#\ \omega) \longleftrightarrow x \in X$

$\langle proof \rangle$

lemma *next-mono*:

assumes *next*: *next* $\varphi\ xs$ **and** 0 : $\bigwedge xs. \varphi\ xs \implies \psi\ xs$

shows *next* $\psi\ xs$

$\langle proof \rangle$

declare *ev.intros*[*intro*]

declare *alw.cases*[*elim*]

lemma *ev-induct-strong*[*consumes 1, case-names base step*]:

$ev\ \varphi\ x \implies (\bigwedge xs. \varphi\ xs \implies P\ xs) \implies (\bigwedge xs. ev\ \varphi\ (stl\ xs) \implies \neg\ \varphi\ xs \implies P\ (stl\ xs) \implies P\ xs) \implies P\ x$

$\langle proof \rangle$

lemma *alw-coinduct*[*consumes 1, case-names alw stl*]:

$X\ x \implies (\bigwedge x. X\ x \implies \varphi\ x) \implies (\bigwedge x. X\ x \implies \neg\ alw\ \varphi\ (stl\ x) \implies X\ (stl\ x)) \implies alw\ \varphi\ x$

$\langle proof \rangle$

lemma *ev-mono*:

assumes *ev*: *ev* $\varphi\ xs$ **and** 0 : $\bigwedge xs. \varphi\ xs \implies \psi\ xs$

shows *ev* $\psi\ xs$

$\langle proof \rangle$

lemma *alw-mono*:

assumes *alw*: *alw* $\varphi\ xs$ **and** 0 : $\bigwedge xs. \varphi\ xs \implies \psi\ xs$

shows *alw* $\psi\ xs$

$\langle proof \rangle$

lemma *until-monoL*:

assumes *until*: $(\varphi 1\ until\ \psi)\ xs$ **and** 0 : $\bigwedge xs. \varphi 1\ xs \implies \varphi 2\ xs$

shows $(\varphi 2\ until\ \psi)\ xs$

$\langle proof \rangle$

lemma *until-monoR*:

assumes *until*: $(\varphi\ until\ \psi 1)\ xs$ **and** 0 : $\bigwedge xs. \psi 1\ xs \implies \psi 2\ xs$

shows $(\varphi \text{ until } \psi_2) \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *until-mono*:
assumes *until*: $(\varphi_1 \text{ until } \psi_1) \text{ xs}$ **and**
 $0: \bigwedge \text{xs}. \varphi_1 \text{ xs} \implies \varphi_2 \text{ xs} \bigwedge \text{xs}. \psi_1 \text{ xs} \implies \psi_2 \text{ xs}$
shows $(\varphi_2 \text{ until } \psi_2) \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *until-false*: $\varphi \text{ until false} = \text{alw } \varphi$
 $\langle \text{proof} \rangle$

lemma *ev-nxt*: $\text{ev } \varphi = (\varphi \text{ or } \text{nxt } (\text{ev } \varphi))$
 $\langle \text{proof} \rangle$

lemma *alw-nxt*: $\text{alw } \varphi = (\varphi \text{ aand } \text{nxt } (\text{alw } \varphi))$
 $\langle \text{proof} \rangle$

lemma *ev-ev[simp]*: $\text{ev } (\text{ev } \varphi) = \text{ev } \varphi$
 $\langle \text{proof} \rangle$

lemma *alw-alw[simp]*: $\text{alw } (\text{alw } \varphi) = \text{alw } \varphi$
 $\langle \text{proof} \rangle$

lemma *ev-shift*:
assumes *ev* $\varphi \text{ xs}$
shows $\text{ev } \varphi (xl @- \text{xs})$
 $\langle \text{proof} \rangle$

lemma *ev-imp-shift*:
assumes *ev* $\varphi \text{ xs}$ **shows** $\exists \text{ xl xs2}. \text{xs} = \text{xl} @- \text{xs2} \wedge \varphi \text{ xs2}$
 $\langle \text{proof} \rangle$

lemma *alw-ev-shift*: $\text{alw } \varphi \text{ xs1} \implies \text{ev } (\text{alw } \varphi) (\text{xl} @- \text{xs1})$
 $\langle \text{proof} \rangle$

lemma *alw-shift*:
assumes *alw* $\varphi (xl @- \text{xs})$
shows $\text{alw } \varphi \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *ev-ex-nxt*:
assumes *ev* $\varphi \text{ xs}$
shows $\exists n. (\text{nxt} \rightsquigarrow n) \varphi \text{ xs}$
 $\langle \text{proof} \rangle$

lemma *alw-sdrop*:
assumes *alw* $\varphi \text{ xs}$ **shows** $\text{alw } \varphi (\text{sdrop } n \text{ xs})$
 $\langle \text{proof} \rangle$

lemma *next-sdrop*: $(next \leadsto n) \varphi xs \longleftrightarrow \varphi (sdrop\ n\ xs)$
 $\langle proof \rangle$

definition *wait* $\varphi\ xs \equiv LEAST\ n. (next \leadsto n) \varphi\ xs$

lemma *next-wait*:
assumes $ev\ \varphi\ xs$ **shows** $(next \leadsto (wait\ \varphi\ xs)) \varphi\ xs$
 $\langle proof \rangle$

lemma *next-wait-least*:
assumes $ev: ev\ \varphi\ xs$ **and** $next: (next \leadsto n) \varphi\ xs$ **shows** $wait\ \varphi\ xs \leq n$
 $\langle proof \rangle$

lemma *sdrop-wait*:
assumes $ev\ \varphi\ xs$ **shows** $\varphi (sdrop\ (wait\ \varphi\ xs)\ xs)$
 $\langle proof \rangle$

lemma *sdrop-wait-least*:
assumes $ev: ev\ \varphi\ xs$ **and** $next: \varphi (sdrop\ n\ xs)$ **shows** $wait\ \varphi\ xs \leq n$
 $\langle proof \rangle$

lemma *next-ev*: $(next \leadsto n) \varphi\ xs \implies ev\ \varphi\ xs$
 $\langle proof \rangle$

lemma *not-ev*: $not\ (ev\ \varphi) = alw\ (not\ \varphi)$
 $\langle proof \rangle$

lemma *not-alw*: $not\ (alw\ \varphi) = ev\ (not\ \varphi)$
 $\langle proof \rangle$

lemma *not-ev-not[simp]*: $not\ (ev\ (not\ \varphi)) = alw\ \varphi$
 $\langle proof \rangle$

lemma *not-alw-not[simp]*: $not\ (alw\ (not\ \varphi)) = ev\ \varphi$
 $\langle proof \rangle$

lemma *alw-ev-sdrop*:
assumes $alw\ (ev\ \varphi)\ (sdrop\ m\ xs)$
shows $alw\ (ev\ \varphi)\ xs$
 $\langle proof \rangle$

lemma *ev-alw-imp-alw-ev*:
assumes $ev\ (alw\ \varphi)\ xs$ **shows** $alw\ (ev\ \varphi)\ xs$
 $\langle proof \rangle$

lemma *alw-aand*: $alw\ (\varphi\ aand\ \psi) = alw\ \varphi\ aand\ alw\ \psi$
 $\langle proof \rangle$

lemma *ev-or*: $ev (\varphi \text{ or } \psi) = ev \varphi \text{ or } ev \psi$
 $\langle proof \rangle$

lemma *ev-alw-aand*:
assumes $\varphi: ev (alw \varphi) \text{ xs}$ **and** $\psi: ev (alw \psi) \text{ xs}$
shows $ev (alw (\varphi \text{ aand } \psi)) \text{ xs}$
 $\langle proof \rangle$

lemma *ev-alw-alw-impl*:
assumes $ev (alw \varphi) \text{ xs}$ **and** $alw (alw \varphi \text{ impl } ev \psi) \text{ xs}$
shows $ev \psi \text{ xs}$
 $\langle proof \rangle$

lemma *ev-alw-stl[simp]*: $ev (alw \varphi) (stl \text{ } x) \longleftrightarrow ev (alw \varphi) x$
 $\langle proof \rangle$

lemma *alw-alw-impl-ev*:
 $alw (alw \varphi \text{ impl } ev \psi) = (ev (alw \varphi) \text{ impl } alw (ev \psi)) \text{ (is } ?A = ?B)$
 $\langle proof \rangle$

lemma *ev-alw-impl*:
assumes $ev \varphi \text{ xs}$ **and** $alw (\varphi \text{ impl } \psi) \text{ xs}$ **shows** $ev \psi \text{ xs}$
 $\langle proof \rangle$

lemma *ev-alw-impl-ev*:
assumes $ev \varphi \text{ xs}$ **and** $alw (\varphi \text{ impl } ev \psi) \text{ xs}$ **shows** $ev \psi \text{ xs}$
 $\langle proof \rangle$

lemma *alw-mp*:
assumes $alw \varphi \text{ xs}$ **and** $alw (\varphi \text{ impl } \psi) \text{ xs}$
shows $alw \psi \text{ xs}$
 $\langle proof \rangle$

lemma *all-imp-alw*:
assumes $\bigwedge xs. \varphi \text{ xs}$ **shows** $alw \varphi \text{ xs}$
 $\langle proof \rangle$

lemma *alw-impl-ev-alw*:
assumes $alw (\varphi \text{ impl } ev \psi) \text{ xs}$
shows $alw (ev \varphi \text{ impl } ev \psi) \text{ xs}$
 $\langle proof \rangle$

lemma *ev-holds-sset*:
 $ev (holds P) \text{ xs} \longleftrightarrow (\exists x \in sset \text{ xs}. P x) \text{ (is } ?L \longleftrightarrow ?R)$
 $\langle proof \rangle$

LTL as a program logic:

lemma *alw-invar*:
assumes $\varphi \text{ xs}$ **and** $alw (\varphi \text{ impl } next \varphi) \text{ xs}$

shows $alw\ \varphi\ xs$
 $\langle proof \rangle$

lemma *variance*:

assumes 1: $\varphi\ xs$ **and** 2: $alw\ (\varphi\ impl\ (\psi\ or\ nxt\ \varphi))\ xs$

shows $(alw\ \varphi\ or\ ev\ \psi)\ xs$

$\langle proof \rangle$

lemma *ev-alw-imp-nxt*:

assumes $e: ev\ \varphi\ xs$ **and** $a: alw\ (\varphi\ impl\ (nxt\ \varphi))\ xs$

shows $ev\ (alw\ \varphi)\ xs$

$\langle proof \rangle$

inductive $ev-at :: ('a\ stream \Rightarrow bool) \Rightarrow nat \Rightarrow 'a\ stream \Rightarrow bool$ **for** $P :: 'a\ stream \Rightarrow bool$ **where**

base: $P\ \omega \Longrightarrow ev-at\ P\ 0\ \omega$

| *step*: $\neg P\ \omega \Longrightarrow ev-at\ P\ n\ (stl\ \omega) \Longrightarrow ev-at\ P\ (Suc\ n)\ \omega$

inductive-simps $ev-at-0[simp]: ev-at\ P\ 0\ \omega$

inductive-simps $ev-at-Suc[simp]: ev-at\ P\ (Suc\ n)\ \omega$

lemma *ev-at-imp-snth*: $ev-at\ P\ n\ \omega \Longrightarrow P\ (sdrop\ n\ \omega)$

$\langle proof \rangle$

lemma *ev-at-HLD-imp-snth*: $ev-at\ (HLD\ X)\ n\ \omega \Longrightarrow \omega !! n \in X$

$\langle proof \rangle$

lemma *ev-at-HLD-single-imp-snth*: $ev-at\ (HLD\ \{x\})\ n\ \omega \Longrightarrow \omega !! n = x$

$\langle proof \rangle$

lemma *ev-at-unique*: $ev-at\ P\ n\ \omega \Longrightarrow ev-at\ P\ m\ \omega \Longrightarrow n = m$

$\langle proof \rangle$

lemma *ev-iff-ev-at*: $ev\ P\ \omega \longleftrightarrow (\exists n. ev-at\ P\ n\ \omega)$

$\langle proof \rangle$

lemma *ev-at-shift*: $ev-at\ (HLD\ X)\ i\ (stake\ (Suc\ i)\ \omega @- \omega' :: 's\ stream) \longleftrightarrow ev-at\ (HLD\ X)\ i\ \omega$

$\langle proof \rangle$

lemma *ev-iff-ev-at-unique*: $ev\ P\ \omega \longleftrightarrow (\exists! n. ev-at\ P\ n\ \omega)$

$\langle proof \rangle$

lemma *alw-HLD-iff-streams*: $alw\ (HLD\ X)\ \omega \longleftrightarrow \omega \in streams\ X$

$\langle proof \rangle$

lemma *not-HLD*: $not\ (HLD\ X) = HLD\ (-\ X)$

$\langle proof \rangle$

lemma *not-alw-iff*: $\neg (alw\ P\ \omega) \longleftrightarrow ev\ (not\ P)\ \omega$
 $\langle proof \rangle$

lemma *not-ev-iff*: $\neg (ev\ P\ \omega) \longleftrightarrow alw\ (not\ P)\ \omega$
 $\langle proof \rangle$

lemma *ev-Stream*: $ev\ P\ (x\ \#\#\ s) \longleftrightarrow P\ (x\ \#\#\ s) \vee ev\ P\ s$
 $\langle proof \rangle$

lemma *alw-ev-imp-ev-alw*:
assumes $alw\ (ev\ P)\ \omega$ **shows** $ev\ (P\ a\ and\ alw\ (ev\ P))\ \omega$
 $\langle proof \rangle$

lemma *ev-False*: $ev\ (\lambda x. False)\ \omega \longleftrightarrow False$
 $\langle proof \rangle$

lemma *alw-False*: $alw\ (\lambda x. False)\ \omega \longleftrightarrow False$
 $\langle proof \rangle$

lemma *ev-iff-sdrop*: $ev\ P\ \omega \longleftrightarrow (\exists m. P\ (sdrop\ m\ \omega))$
 $\langle proof \rangle$

lemma *alw-iff-sdrop*: $alw\ P\ \omega \longleftrightarrow (\forall m. P\ (sdrop\ m\ \omega))$
 $\langle proof \rangle$

lemma *infinite-iff-alw-ev*: $infinite\ \{m. P\ (sdrop\ m\ \omega)\} \longleftrightarrow alw\ (ev\ P)\ \omega$
 $\langle proof \rangle$

lemma *alw-inv*:
assumes $stl: \bigwedge s. f\ (stl\ s) = stl\ (f\ s)$
shows $alw\ P\ (f\ s) \longleftrightarrow alw\ (\lambda x. P\ (f\ x))\ s$
 $\langle proof \rangle$

lemma *ev-inv*:
assumes $stl: \bigwedge s. f\ (stl\ s) = stl\ (f\ s)$
shows $ev\ P\ (f\ s) \longleftrightarrow ev\ (\lambda x. P\ (f\ x))\ s$
 $\langle proof \rangle$

lemma *alw-smap*: $alw\ P\ (smap\ f\ s) \longleftrightarrow alw\ (\lambda x. P\ (smap\ f\ x))\ s$
 $\langle proof \rangle$

lemma *ev-smap*: $ev\ P\ (smap\ f\ s) \longleftrightarrow ev\ (\lambda x. P\ (smap\ f\ x))\ s$
 $\langle proof \rangle$

lemma *alw-cong*:
assumes $P: alw\ P\ \omega$ **and** $eq: \bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$
shows $alw\ Q1\ \omega \longleftrightarrow alw\ Q2\ \omega$
 $\langle proof \rangle$

lemma *ev-cong*:

assumes P : $alw\ P\ \omega$ **and** eq : $\bigwedge\omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$

shows $ev\ Q1\ \omega \longleftrightarrow ev\ Q2\ \omega$

$\langle proof \rangle$

lemma *alwD*: $alw\ P\ x \implies P\ x$

$\langle proof \rangle$

lemma *alw-alwD*: $alw\ P\ \omega \implies alw\ (alw\ P)\ \omega$

$\langle proof \rangle$

lemma *alw-ev-stl*: $alw\ (ev\ P)\ (stl\ \omega) \longleftrightarrow alw\ (ev\ P)\ \omega$

$\langle proof \rangle$

lemma *holds-Stream*: $holds\ P\ (x\ \#\#\ s) \longleftrightarrow P\ x$

$\langle proof \rangle$

lemma *holds-eq1[simp]*: $holds\ ((=)\ x) = HLD\ \{x\}$

$\langle proof \rangle$

lemma *holds-eq2[simp]*: $holds\ (\lambda y. y = x) = HLD\ \{x\}$

$\langle proof \rangle$

lemma *not-holds-eq[simp]*: $holds\ (-\ (=)\ x) = not\ (HLD\ \{x\})$

$\langle proof \rangle$

Strong until

context

notes $[[inductive-internals]]$

begin

inductive *suntil* (**infix** $\langle suntil \rangle\ 60$) **for** $\varphi\ \psi$ **where**

base: $\psi\ \omega \implies (\varphi\ suntil\ \psi)\ \omega$

| *step*: $\varphi\ \omega \implies (\varphi\ suntil\ \psi)\ (stl\ \omega) \implies (\varphi\ suntil\ \psi)\ \omega$

inductive-simps *suntil-Stream*: $(\varphi\ suntil\ \psi)\ (x\ \#\#\ s)$

end

lemma *suntil-induct-strong[consumes 1, case-names base step]*:

$(\varphi\ suntil\ \psi)\ x \implies$

$(\bigwedge\omega. \psi\ \omega \implies P\ \omega) \implies$

$(\bigwedge\omega. \varphi\ \omega \implies \neg\ \psi\ \omega \implies (\varphi\ suntil\ \psi)\ (stl\ \omega) \implies P\ (stl\ \omega) \implies P\ \omega) \implies P\ x$

$\langle proof \rangle$

lemma *ev-suntil*: $(\varphi\ suntil\ \psi)\ \omega \implies ev\ \psi\ \omega$

$\langle proof \rangle$

lemma *suntil-inv*:

assumes *stl*: $\bigwedge s. f (stl\ s) = stl\ (f\ s)$
shows $(P\ suntil\ Q)\ (f\ s) \longleftrightarrow ((\lambda x. P\ (f\ x))\ suntil\ (\lambda x. Q\ (f\ x)))\ s$
 $\langle proof \rangle$

lemma *suntil-smap*: $(P\ suntil\ Q)\ (smap\ f\ s) \longleftrightarrow ((\lambda x. P\ (smap\ f\ x))\ suntil\ (\lambda x. Q\ (smap\ f\ x)))\ s$
 $\langle proof \rangle$

lemma *hld-smap*: $HLD\ x\ (smap\ f\ s) = holds\ (\lambda y. f\ y \in x)\ s$
 $\langle proof \rangle$

lemma *suntil-mono*:

assumes *eq*: $\bigwedge \omega. P\ \omega \implies Q1\ \omega \implies Q2\ \omega \bigwedge \omega. P\ \omega \implies R1\ \omega \implies R2\ \omega$
assumes *: $(Q1\ suntil\ R1)\ \omega\ alw\ P\ \omega$ **shows** $(Q2\ suntil\ R2)\ \omega$
 $\langle proof \rangle$

lemma *suntil-cong*:

$alw\ P\ \omega \implies (\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega) \implies (\bigwedge \omega. P\ \omega \implies R1\ \omega \longleftrightarrow R2\ \omega) \implies$
 $(Q1\ suntil\ R1)\ \omega \longleftrightarrow (Q2\ suntil\ R2)\ \omega$
 $\langle proof \rangle$

lemma *ev-suntil-iff*: $ev\ (P\ suntil\ Q)\ \omega \longleftrightarrow ev\ Q\ \omega$
 $\langle proof \rangle$

lemma *true-suntil*: $((\lambda -. True)\ suntil\ P) = ev\ P$
 $\langle proof \rangle$

lemma *suntil-lfp*: $(\varphi\ suntil\ \psi) = lfp\ (\lambda P\ s. \psi\ s \vee (\varphi\ s \wedge P\ (stl\ s)))$
 $\langle proof \rangle$

lemma *sfilter-P[simp]*: $P\ (shd\ s) \implies sfilter\ P\ s = shd\ s \#\#\ sfilter\ P\ (stl\ s)$
 $\langle proof \rangle$

lemma *sfilter-not-P[simp]*: $\neg P\ (shd\ s) \implies sfilter\ P\ s = sfilter\ P\ (stl\ s)$
 $\langle proof \rangle$

lemma *sfilter-eq*:

assumes *ev* (*holds* *P*) *s*
shows $sfilter\ P\ s = x \#\#\ s' \longleftrightarrow$
 $P\ x \wedge (not\ (holds\ P)\ suntil\ (HLD\ \{x\}\ aand\ next\ (\lambda s. sfilter\ P\ s = s')))\ s$
 $\langle proof \rangle$

lemma *sfilter-streams*:

$alw\ (ev\ (holds\ P))\ \omega \implies \omega \in streams\ A \implies sfilter\ P\ \omega \in streams\ \{x \in A. P\ x\}$
 $\langle proof \rangle$

lemma *alw-sfilter*:

assumes *: $alw (ev (holds P)) s$
shows $alw Q (sfilter P s) \longleftrightarrow alw (\lambda x. Q (sfilter P x)) s$
 $\langle proof \rangle$

lemma *ev-sfilter*:
assumes *: $alw (ev (holds P)) s$
shows $ev Q (sfilter P s) \longleftrightarrow ev (\lambda x. Q (sfilter P x)) s$
 $\langle proof \rangle$

lemma *holds-sfilter*:
assumes $ev (holds Q) s$ **shows** $holds P (sfilter Q s) \longleftrightarrow (not (holds Q) \text{ until } (holds (Q \text{ aand } P))) s$
 $\langle proof \rangle$

lemma *suntil-aand-nxt*:
 $(\varphi \text{ until } (\varphi \text{ aand } \text{next } \psi)) \omega \longleftrightarrow (\varphi \text{ aand } \text{next } (\varphi \text{ until } \psi)) \omega$
 $\langle proof \rangle$

lemma *alw-sconst*: $alw P (sconst x) \longleftrightarrow P (sconst x)$
 $\langle proof \rangle$

lemma *ev-sconst*: $ev P (sconst x) \longleftrightarrow P (sconst x)$
 $\langle proof \rangle$

lemma *suntil-sconst*: $(\varphi \text{ until } \psi) (sconst x) \longleftrightarrow \psi (sconst x)$
 $\langle proof \rangle$

lemma *hld-smap'*: $HLD x (smap f s) = HLD (f -' x) s$
 $\langle proof \rangle$

lemma *pigeonhole-stream*:
assumes $alw (HLD s) \omega$
assumes *finite* s
shows $\exists x \in s. alw (ev (HLD \{x\})) \omega$
 $\langle proof \rangle$

lemma *ev-eq-suntil*: $ev P \omega \longleftrightarrow (not P \text{ until } P) \omega$
 $\langle proof \rangle$

61 Weak vs. strong until (contributed by Michael Foster, University of Sheffield)

lemma *suntil-implies-until*: $(\varphi \text{ until } \psi) \omega \implies (\varphi \text{ until } \psi) \omega$
 $\langle proof \rangle$

lemma *alw-implies-until*: $alw \varphi \omega \implies (\varphi \text{ until } \psi) \omega$
 $\langle proof \rangle$

lemma *until-ev-suntil*: $(\varphi \text{ until } \psi) \omega \implies \text{ev } \psi \omega \implies (\varphi \text{ suntil } \psi) \omega$
 $\langle \text{proof} \rangle$

lemma *suntil-as-until*: $(\varphi \text{ suntil } \psi) \omega = ((\varphi \text{ until } \psi) \omega \wedge \text{ev } \psi \omega)$
 $\langle \text{proof} \rangle$

lemma *until-not-released-now*: $(\varphi \text{ until } \psi) \omega \implies \neg \psi \omega \implies \varphi \omega$
 $\langle \text{proof} \rangle$

lemma *until-must-release-ev*: $(\varphi \text{ until } \psi) \omega \implies \text{ev } (\text{not } \varphi) \omega \implies \text{ev } \psi \omega$
 $\langle \text{proof} \rangle$

lemma *until-as-suntil*: $(\varphi \text{ until } \psi) \omega = ((\varphi \text{ suntil } \psi) \text{ or } (\text{alw } \varphi)) \omega$
 $\langle \text{proof} \rangle$

lemma *alw-holds*: $\text{alw } (\text{holds } P) (h \# \# t) = (P \ h \wedge \text{alw } (\text{holds } P) \ t)$
 $\langle \text{proof} \rangle$

lemma *alw-holds2*: $\text{alw } (\text{holds } P) \ ss = (P \ (\text{shd } ss) \wedge \text{alw } (\text{holds } P) \ (\text{stl } ss))$
 $\langle \text{proof} \rangle$

lemma *alw-eq-sconst*: $(\text{alw } (\text{HLD } \{h\}) \ t) = (t = \text{sconst } h)$
 $\langle \text{proof} \rangle$

lemma *sdrop-if-suntil*: $(p \text{ suntil } q) \omega \implies \exists j. q \ (\text{sdrop } j \ \omega) \wedge (\forall k < j. p \ (\text{sdrop } k \ \omega))$
 $\langle \text{proof} \rangle$

lemma *not-suntil*: $(\neg (p \text{ suntil } q) \ \omega) = (\neg (p \text{ until } q) \ \omega \vee \text{alw } (\text{not } q) \ \omega)$
 $\langle \text{proof} \rangle$

lemma *sdrop-until*: $q \ (\text{sdrop } j \ \omega) \implies \forall k < j. p \ (\text{sdrop } k \ \omega) \implies (p \text{ until } q) \ \omega$
 $\langle \text{proof} \rangle$

lemma *sdrop-suntil*: $q \ (\text{sdrop } j \ \omega) \implies (\forall k < j. p \ (\text{sdrop } k \ \omega)) \implies (p \text{ suntil } q) \ \omega$
 $\langle \text{proof} \rangle$

lemma *suntil-iff-sdrop*: $(p \text{ suntil } q) \ \omega = (\exists j. q \ (\text{sdrop } j \ \omega) \wedge (\forall k < j. p \ (\text{sdrop } k \ \omega)))$
 $\langle \text{proof} \rangle$

end

62 Lists as vectors

```
theory ListVector
  imports Main
begin
```

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation $scale :: ('a::times) \Rightarrow 'a\ list \Rightarrow 'a\ list$ (**infix** $\langle *_{\text{s}} \rangle$ 70)
where $x *_{\text{s}} xs \equiv map\ ((*)\ x)\ xs$

lemma $scaleI[simp]: (1::'a::monoid-mult) *_{\text{s}} xs = xs$
 $\langle proof \rangle$

62.1 + and −

fun $zipwith0 :: ('a::zero \Rightarrow 'b::zero \Rightarrow 'c) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list$
where
 $zipwith0\ f\ []\ [] = []\ |$
 $zipwith0\ f\ (x\#xs)\ (y\#ys) = f\ x\ y\ \# \ zipwith0\ f\ xs\ ys\ |$
 $zipwith0\ f\ (x\#xs)\ [] = f\ x\ 0\ \# \ zipwith0\ f\ xs\ []\ |$
 $zipwith0\ f\ []\ (y\#ys) = f\ 0\ y\ \# \ zipwith0\ f\ []\ ys$

instantiation $list :: (\{zero, plus\})\ plus$
begin

definition
 $list-add-def: (+) = zipwith0\ (+)$

instance $\langle proof \rangle$

end

instantiation $list :: (\{zero, uminus\})\ uminus$
begin

definition
 $list-uminus-def: uminus = map\ uminus$

instance $\langle proof \rangle$

end

instantiation $list :: (\{zero, minus\})\ minus$
begin

definition
 $list-diff-def: (-) = zipwith0\ (-)$

instance $\langle proof \rangle$

end

lemma *zipwith0-Nil[simp]*: $\text{zipwith0 } f \ [] \ ys = \text{map } (f \ 0) \ ys$
 $\langle \text{proof} \rangle$

lemma *list-add-Nil[simp]*: $[] + xs = (xs :: 'a :: \text{monoid-add list})$
 $\langle \text{proof} \rangle$

lemma *list-add-Nil2[simp]*: $xs + [] = (xs :: 'a :: \text{monoid-add list})$
 $\langle \text{proof} \rangle$

lemma *list-add-Cons[simp]*: $(x \# xs) + (y \# ys) = (x + y) \# (xs + ys)$
 $\langle \text{proof} \rangle$

lemma *list-diff-Nil[simp]*: $[] - xs = -(xs :: 'a :: \text{group-add list})$
 $\langle \text{proof} \rangle$

lemma *list-diff-Nil2[simp]*: $xs - [] = (xs :: 'a :: \text{group-add list})$
 $\langle \text{proof} \rangle$

lemma *list-diff-Cons-Cons[simp]*: $(x \# xs) - (y \# ys) = (x - y) \# (xs - ys)$
 $\langle \text{proof} \rangle$

lemma *list-uminus-Cons[simp]*: $-(x \# xs) = (-x) \# (-xs)$
 $\langle \text{proof} \rangle$

lemma *self-list-diff*:
 $xs - xs = \text{replicate } (\text{length}(xs :: 'a :: \text{group-add list})) \ 0$
 $\langle \text{proof} \rangle$

lemma *list-add-assoc*:
fixes $xs :: 'a :: \text{monoid-add list}$
shows $(xs + ys) + zs = xs + (ys + zs)$
 $\langle \text{proof} \rangle$

62.2 Inner product

definition *iprod* :: $'a :: \text{ring list} \Rightarrow 'a \text{ list} \Rightarrow 'a$ ($\langle \langle \text{open-block notation} = \langle \text{mixfix iprod} \rangle \langle -, - \rangle \rangle$)
where $\langle xs, ys \rangle = (\sum (x, y) \leftarrow \text{zip } xs \ ys. \ x * y)$

lemma *iprod-Nil[simp]*: $\langle [], ys \rangle = 0$
 $\langle \text{proof} \rangle$

lemma *iprod-Nil2[simp]*: $\langle xs, [] \rangle = 0$
 $\langle \text{proof} \rangle$

lemma *iprod-Cons[simp]*: $\langle x \# xs, y \# ys \rangle = x * y + \langle xs, ys \rangle$
 $\langle \text{proof} \rangle$

lemma *iprod0-if-coeffs0*: $\forall c \in \text{set } cs. \ c = 0 \implies \langle cs, xs \rangle = 0$

$\langle proof \rangle$

lemma *iprod-uminus[simp]*: $\langle -xs, ys \rangle = -\langle xs, ys \rangle$
 $\langle proof \rangle$

lemma *iprod-left-add-distrib*: $\langle xs + ys, zs \rangle = \langle xs, zs \rangle + \langle ys, zs \rangle$
 $\langle proof \rangle$

lemma *iprod-left-diff-distrib*: $\langle xs - ys, zs \rangle = \langle xs, zs \rangle - \langle ys, zs \rangle$
 $\langle proof \rangle$

lemma *iprod-assoc*: $\langle x *_s xs, ys \rangle = x * \langle xs, ys \rangle$
 $\langle proof \rangle$

end

63 Definitions of Least Upper Bounds and Greatest Lower Bounds

theory *Lub-Glb*
imports *Complex-Main*
begin

Thanks to suggestions by James Margetson

definition *setle* :: $'a \text{ set} \Rightarrow 'a::ord \Rightarrow bool$ (**infixl** $\langle * \leq \rangle$ 70)
where $S * \leq x = (\forall y \in S. y \leq x)$

definition *setge* :: $'a::ord \Rightarrow 'a \text{ set} \Rightarrow bool$ (**infixl** $\langle \leq * \rangle$ 70)
where $x \leq * S = (\forall y \in S. x \leq y)$

63.1 Rules for the Relations $* \leq$ and $\leq *$

lemma *setleI*: $\forall y \in S. y \leq x \Longrightarrow S * \leq x$
 $\langle proof \rangle$

lemma *setleD*: $S * \leq x \Longrightarrow y \in S \Longrightarrow y \leq x$
 $\langle proof \rangle$

lemma *setgeI*: $\forall y \in S. x \leq y \Longrightarrow x \leq * S$
 $\langle proof \rangle$

lemma *setgeD*: $x \leq * S \Longrightarrow y \in S \Longrightarrow x \leq y$
 $\langle proof \rangle$

definition *leastP* :: $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$
where $leastP P x = (P x \wedge x \leq * Collect P)$

definition $isUb :: 'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where $isUb\ R\ S\ x = (S\ *\leq\ x \wedge x \in R)$

definition $isLub :: 'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where $isLub\ R\ S\ x = leastP\ (isUb\ R\ S)\ x$

definition $ubs :: 'a\ set \Rightarrow 'a::ord\ set \Rightarrow 'a\ set$
where $ubs\ R\ S = Collect\ (isUb\ R\ S)$

63.2 Rules about the Operators $leastP$, ub and lub

lemma $leastPD1: leastP\ P\ x \Longrightarrow P\ x$
 $\langle proof \rangle$

lemma $leastPD2: leastP\ P\ x \Longrightarrow x\ *\leq\ Collect\ P$
 $\langle proof \rangle$

lemma $leastPD3: leastP\ P\ x \Longrightarrow y \in Collect\ P \Longrightarrow x \leq y$
 $\langle proof \rangle$

lemma $isLubD1: isLub\ R\ S\ x \Longrightarrow S\ *\leq\ x$
 $\langle proof \rangle$

lemma $isLubD1a: isLub\ R\ S\ x \Longrightarrow x \in R$
 $\langle proof \rangle$

lemma $isLub-isUb: isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ x$
 $\langle proof \rangle$

lemma $isLubD2: isLub\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \leq x$
 $\langle proof \rangle$

lemma $isLubD3: isLub\ R\ S\ x \Longrightarrow leastP\ (isUb\ R\ S)\ x$
 $\langle proof \rangle$

lemma $isLubI1: leastP\ (isUb\ R\ S)\ x \Longrightarrow isLub\ R\ S\ x$
 $\langle proof \rangle$

lemma $isLubI2: isUb\ R\ S\ x \Longrightarrow x\ *\leq\ Collect\ (isUb\ R\ S) \Longrightarrow isLub\ R\ S\ x$
 $\langle proof \rangle$

lemma $isUbD: isUb\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \leq x$
 $\langle proof \rangle$

lemma $isUbD2: isUb\ R\ S\ x \Longrightarrow S\ *\leq\ x$
 $\langle proof \rangle$

lemma $isUbD2a: isUb\ R\ S\ x \Longrightarrow x \in R$
 $\langle proof \rangle$

lemma *isUbI*: $S * \leq x \implies x \in R \implies \text{isUb } R \ S \ x$
 $\langle \text{proof} \rangle$

lemma *isLub-le-isUb*: $\text{isLub } R \ S \ x \implies \text{isUb } R \ S \ y \implies x \leq y$
 $\langle \text{proof} \rangle$

lemma *isLub-ubs*: $\text{isLub } R \ S \ x \implies x \leq * \text{ubs } R \ S$
 $\langle \text{proof} \rangle$

lemma *isLub-unique*: $[\text{isLub } R \ S \ x; \text{isLub } R \ S \ y] \implies x = (y::'a::\text{linorder})$
 $\langle \text{proof} \rangle$

lemma *isUb-UNIV-I*: $(\bigwedge y. y \in S \implies y \leq u) \implies \text{isUb } UNIV \ S \ u$
 $\langle \text{proof} \rangle$

definition *greatestP* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a::\text{ord} \Rightarrow \text{bool}$
where *greatestP* $P \ x = (P \ x \wedge \text{Collect } P * \leq x)$

definition *isLb* :: $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a::\text{ord} \Rightarrow \text{bool}$
where *isLb* $R \ S \ x = (x \leq * S \wedge x \in R)$

definition *isGlb* :: $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a::\text{ord} \Rightarrow \text{bool}$
where *isGlb* $R \ S \ x = \text{greatestP } (\text{isLb } R \ S) \ x$

definition *lbs* :: $'a \text{ set} \Rightarrow 'a::\text{ord} \text{ set} \Rightarrow 'a \text{ set}$
where *lbs* $R \ S = \text{Collect } (\text{isLb } R \ S)$

63.3 Rules about the Operators *greatestP*, *isLb* and *isGlb*

lemma *greatestPD1*: $\text{greatestP } P \ x \implies P \ x$
 $\langle \text{proof} \rangle$

lemma *greatestPD2*: $\text{greatestP } P \ x \implies \text{Collect } P * \leq x$
 $\langle \text{proof} \rangle$

lemma *greatestPD3*: $\text{greatestP } P \ x \implies y \in \text{Collect } P \implies x \geq y$
 $\langle \text{proof} \rangle$

lemma *isGlbD1*: $\text{isGlb } R \ S \ x \implies x \leq * S$
 $\langle \text{proof} \rangle$

lemma *isGlbD1a*: $\text{isGlb } R \ S \ x \implies x \in R$
 $\langle \text{proof} \rangle$

lemma *isGlb-isLb*: $\text{isGlb } R \ S \ x \implies \text{isLb } R \ S \ x$
 $\langle \text{proof} \rangle$

lemma *isGlbD2*: $\text{isGlb } R \ S \ x \Longrightarrow y \in S \Longrightarrow y \geq x$
 ⟨proof⟩

lemma *isGlbD3*: $\text{isGlb } R \ S \ x \Longrightarrow \text{greatestP } (\text{isLb } R \ S) \ x$
 ⟨proof⟩

lemma *isGlbI1*: $\text{greatestP } (\text{isLb } R \ S) \ x \Longrightarrow \text{isGlb } R \ S \ x$
 ⟨proof⟩

lemma *isGlbI2*: $\text{isLb } R \ S \ x \Longrightarrow \text{Collect } (\text{isLb } R \ S) \ * \leq x \Longrightarrow \text{isGlb } R \ S \ x$
 ⟨proof⟩

lemma *isLbD*: $\text{isLb } R \ S \ x \Longrightarrow y \in S \Longrightarrow y \geq x$
 ⟨proof⟩

lemma *isLbD2*: $\text{isLb } R \ S \ x \Longrightarrow x \leq^* S$
 ⟨proof⟩

lemma *isLbD2a*: $\text{isLb } R \ S \ x \Longrightarrow x \in R$
 ⟨proof⟩

lemma *isLbI*: $x \leq^* S \Longrightarrow x \in R \Longrightarrow \text{isLb } R \ S \ x$
 ⟨proof⟩

lemma *isGlb-le-isLb*: $\text{isGlb } R \ S \ x \Longrightarrow \text{isLb } R \ S \ y \Longrightarrow x \geq y$
 ⟨proof⟩

lemma *isGlb-ubs*: $\text{isGlb } R \ S \ x \Longrightarrow \text{lbs } R \ S \ * \leq x$
 ⟨proof⟩

lemma *isGlb-unique*: $[\text{isGlb } R \ S \ x; \text{isGlb } R \ S \ y] \Longrightarrow x = (y::'a::\text{linorder})$
 ⟨proof⟩

lemma *bdd-above-settle*: $\text{bdd-above } A \longleftrightarrow (\exists a. A \leq^* a)$
 ⟨proof⟩

lemma *bdd-below-setge*: $\text{bdd-below } A \longleftrightarrow (\exists a. a \leq^* A)$
 ⟨proof⟩

lemma *isLub-cSup*:
 $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \Longrightarrow (\exists b. S \leq^* b) \Longrightarrow \text{isLub}$
 $\text{UNIV } S \ (\text{Sup } S)$
 ⟨proof⟩

lemma *isGlb-cInf*:
 $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \Longrightarrow (\exists b. b \leq^* S) \Longrightarrow \text{isGlb}$
 $\text{UNIV } S \ (\text{Inf } S)$
 ⟨proof⟩

lemma *cSup-le*: $(S :: 'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies S * \leq b \implies \text{Sup } S \leq b$
 <proof>

lemma *cInf-ge*: $(S :: 'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies b \leq * S \implies \text{Inf } S \geq b$
 <proof>

lemma *cSup-bounds*:

fixes $S :: 'a :: \text{conditionally-complete-lattice set}$

shows $S \neq \{\} \implies a \leq * S \implies S * \leq b \implies a \leq \text{Sup } S \wedge \text{Sup } S \leq b$

<proof>

lemma *cSup-unique*: $(S :: 'a :: \{\text{conditionally-complete-linorder, no-bot}\} \text{ set}) * \leq b \implies (\forall b' < b. \exists x \in S. b' < x) \implies \text{Sup } S = b$
 <proof>

lemma *cInf-unique*: $b \leq * (S :: 'a :: \{\text{conditionally-complete-linorder, no-top}\} \text{ set}) \implies (\forall b' > b. \exists x \in S. b' > x) \implies \text{Inf } S = b$
 <proof>

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: $\exists X. X \in S \implies \exists Y. \text{isUb } (\text{UNIV} :: \text{real set}) S Y \implies \exists t. \text{isLub } (\text{UNIV} :: \text{real set}) S t$
 <proof>

lemma *Bseq-isUb*: $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \implies \exists U. \text{isUb } (\text{UNIV} :: \text{real set}) \{x. \exists n. X n = x\} U$
 <proof>

lemma *Bseq-isLub*: $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \implies \exists U. \text{isLub } (\text{UNIV} :: \text{real set}) \{x. \exists n. X n = x\} U$
 <proof>

lemma *isLub-mono-imp-LIMSEQ*:

fixes $X :: \text{nat} \Rightarrow \text{real}$

assumes $u: \text{isLub } \text{UNIV } \{x. \exists n. X n = x\} u$

assumes $X: \forall m n. m \leq n \longrightarrow X m \leq X n$

shows $X \longrightarrow u$

<proof>

lemmas *real-isGlb-unique* = *isGlb-unique*[**where** 'a=real]

lemma *real-le-inf-subset*: $t \neq \{\} \implies t \subseteq s \implies \exists b. b \leq * s \implies \text{Inf } s \leq \text{Inf } (t :: \text{real set})$
 <proof>

lemma *real-ge-sup-subset*: $t \neq \{\} \implies t \subseteq s \implies \exists b. s * \leq b \implies \text{Sup } s \geq \text{Sup } t$

```
(t::real set)
  ⟨proof⟩
```

```
end
```

64 An abstract view on maps for code generation.

```
theory Mapping
imports Main AList
begin
```

64.1 Parametricity transfer rules

```
lemma map-of-foldr: map-of xs = foldr (λ(k, v) m. m(k ↦ v)) xs Map.empty
  ⟨proof⟩
```

```
context includes lifting-syntax
begin
```

```
lemma empty-parametric: (A ==> rel-option B) Map.empty Map.empty
  ⟨proof⟩
```

```
lemma lookup-parametric: ((A ==> B) ==> A ==> B) (λm k. m k) (λm
k. m k)
  ⟨proof⟩
```

```
lemma update-parametric:
  assumes [transfer-rule]: bi-unique A
  shows (A ==> B ==> (A ==> rel-option B) ==> A ==> rel-option
B)
  (λk v m. m(k ↦ v)) (λk v m. m(k ↦ v))
  ⟨proof⟩
```

```
lemma delete-parametric:
  assumes [transfer-rule]: bi-unique A
  shows (A ==> (A ==> rel-option B) ==> A ==> rel-option B)
  (λk m. m(k := None)) (λk m. m(k := None))
  ⟨proof⟩
```

```
lemma is-none-parametric [transfer-rule]:
  (rel-option A ==> HOL.eq) Option.is-none Option.is-none
  ⟨proof⟩
```

```
lemma dom-parametric:
  assumes [transfer-rule]: bi-total A
  shows ((A ==> rel-option B) ==> rel-set A) dom dom
  ⟨proof⟩
```

```
lemma graph-parametric:
```

assumes *bi-total A*
shows $((A \text{====>} \text{rel-option } B) \text{====>} \text{rel-set } (\text{rel-prod } A \ B)) \text{ Map.graph Map.graph}$
 $\langle \text{proof} \rangle$

lemma *map-of-parametric [transfer-rule]:*
assumes *[transfer-rule]: bi-unique R1*
shows $(\text{list-all2 } (\text{rel-prod } R1 \ R2) \text{====>} R1 \text{====>} \text{rel-option } R2) \text{ map-of}$
 map-of
 $\langle \text{proof} \rangle$

lemma *map-entry-parametric [transfer-rule]:*
assumes *[transfer-rule]: bi-unique A*
shows $(A \text{====>} (B \text{====>} B) \text{====>} (A \text{====>} \text{rel-option } B) \text{====>} A$
 $\text{====>} \text{rel-option } B)$
 $(\lambda k \ f \ m. (\text{case } m \ k \ \text{of } \text{None} \Rightarrow m$
 $\quad | \ \text{Some } v \Rightarrow m \ (k \mapsto (f \ v)))) (\lambda k \ f \ m. (\text{case } m \ k \ \text{of } \text{None} \Rightarrow m$
 $\quad | \ \text{Some } v \Rightarrow m \ (k \mapsto (f \ v))))$
 $\langle \text{proof} \rangle$

lemma *tabulate-parametric:*
assumes *[transfer-rule]: bi-unique A*
shows $(\text{list-all2 } A \text{====>} (A \text{====>} B) \text{====>} A \text{====>} \text{rel-option } B)$
 $(\lambda ks \ f. (\text{map-of } (\text{map } (\lambda k. (k, f \ k)) \ ks))) (\lambda ks \ f. (\text{map-of } (\text{map } (\lambda k. (k, f \ k))$
 $\ ks)))$
 $\langle \text{proof} \rangle$

lemma *bulkload-parametric:*
 $(\text{list-all2 } A \text{====>} \text{HOL.eq} \text{====>} \text{rel-option } A)$
 $(\lambda xs \ k. \text{if } k < \text{length } xs \text{ then } \text{Some } (xs \ ! \ k) \ \text{else } \text{None})$
 $(\lambda xs \ k. \text{if } k < \text{length } xs \text{ then } \text{Some } (xs \ ! \ k) \ \text{else } \text{None})$
 $\langle \text{proof} \rangle$

lemma *map-parametric:*
 $((A \text{====>} B) \text{====>} (C \text{====>} D) \text{====>} (B \text{====>} \text{rel-option } C) \text{====>} A$
 $\text{====>} \text{rel-option } D)$
 $(\lambda f \ g \ m. (\text{map-option } g \circ m \circ f)) (\lambda f \ g \ m. (\text{map-option } g \circ m \circ f))$
 $\langle \text{proof} \rangle$

lemma *combine-with-key-parametric:*
 $((A \text{====>} B \text{====>} B \text{====>} B) \text{====>} (A \text{====>} \text{rel-option } B) \text{====>} (A$
 $\text{====>} \text{rel-option } B) \text{====>}$
 $(A \text{====>} \text{rel-option } B)) (\lambda f \ m1 \ m2 \ x. \text{combine-options } (f \ x) \ (m1 \ x) \ (m2 \ x))$
 $(\lambda f \ m1 \ m2 \ x. \text{combine-options } (f \ x) \ (m1 \ x) \ (m2 \ x))$
 $\langle \text{proof} \rangle$

lemma *combine-parametric:*
 $((B \text{====>} B \text{====>} B) \text{====>} (A \text{====>} \text{rel-option } B) \text{====>} (A \text{====>} \text{rel-option } B) \text{====>}$
 $\text{rel-option } B) \text{====>}$
 $(A \text{====>} \text{rel-option } B)) (\lambda f \ m1 \ m2 \ x. \text{combine-options } f \ (m1 \ x) \ (m2 \ x))$

```

    ( $\lambda f\ m1\ m2\ x.$  combine-options  $f\ (m1\ x)\ (m2\ x)$ )
  <proof>

```

end

64.2 Type definition and primitive operations

```

typedef ('a, 'b) mapping = UNIV :: ('a  $\rightarrow$  'b) set
  morphisms rep Mapping <proof>

```

setup-lifting type-definition-mapping

```

lift-definition empty :: ('a, 'b) mapping
  is Map.empty parametric empty-parametric <proof>

```

```

lift-definition lookup :: ('a, 'b) mapping  $\Rightarrow$  'a  $\Rightarrow$  'b option
  is  $\lambda m\ k.$  m k parametric lookup-parametric <proof>

```

```

definition lookup-default d m k = (case Mapping.lookup m k of None  $\Rightarrow$  d | Some
  v  $\Rightarrow$  v)

```

```

lift-definition update :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping
  is  $\lambda k\ v\ m.$  m(k  $\mapsto$  v) parametric update-parametric <proof>

```

```

lift-definition delete :: 'a  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping
  is  $\lambda k\ m.$  m(k := None) parametric delete-parametric <proof>

```

```

lift-definition filter :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping
  is  $\lambda P\ m\ k.$  case m k of None  $\Rightarrow$  None | Some v  $\Rightarrow$  if P k v then Some v else None
  <proof>

```

```

lift-definition keys :: ('a, 'b) mapping  $\Rightarrow$  'a set
  is dom parametric dom-parametric <proof>

```

```

lift-definition entries :: ('a, 'b) mapping  $\Rightarrow$  ('a  $\times$  'b) set
  is Map.graph parametric graph-parametric <proof>

```

```

lift-definition tabulate :: 'a list  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) mapping
  is  $\lambda ks\ f.$  (map-of (List.map ( $\lambda k.$  (k, f k)) ks)) parametric tabulate-parametric
  <proof>

```

```

lift-definition bulkload :: 'a list  $\Rightarrow$  (nat, 'a) mapping
  is  $\lambda xs\ k.$  if k < length xs then Some (xs ! k) else None parametric bulk-
  load-parametric <proof>

```

```

lift-definition map :: ('c  $\Rightarrow$  'a)  $\Rightarrow$  ('b  $\Rightarrow$  'd)  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('c, 'd) mapping
  is  $\lambda f\ g\ m.$  (map-option g  $\circ$  m  $\circ$  f) parametric map-parametric <proof>

```

```

lift-definition map-values :: ('c  $\Rightarrow$  'a  $\Rightarrow$  'b)  $\Rightarrow$  ('c, 'a) mapping  $\Rightarrow$  ('c, 'b) mapping

```

is $\lambda f m x. \text{map-option } (f x) (m x) \langle \text{proof} \rangle$

lift-definition *combine-with-key* ::

$(\text{'a} \Rightarrow \text{'b} \Rightarrow \text{'b} \Rightarrow \text{'b}) \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$
 is $\lambda f m1 m2 x. \text{combine-options } (f x) (m1 x) (m2 x) \text{ parametric combine-with-key-parametric}$
 $\langle \text{proof} \rangle$

lift-definition *combine* ::

$(\text{'b} \Rightarrow \text{'b} \Rightarrow \text{'b}) \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$
 is $\lambda f m1 m2 x. \text{combine-options } f (m1 x) (m2 x) \text{ parametric combine-parametric}$
 $\langle \text{proof} \rangle$

definition *All-mapping* $m P \longleftrightarrow$

$(\forall x. \text{case Mapping.lookup } m x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } y \Rightarrow P x y)$

declare $[[\text{code drop: map}]]$

64.3 Functorial structure

functor *map*: *map*

$\langle \text{proof} \rangle$

64.4 Derived operations

definition *ordered-keys* :: $(\text{'a}::\text{linorder}, \text{'b}) \text{ mapping} \Rightarrow \text{'a list}$

where *ordered-keys* $m = (\text{if finite } (\text{keys } m) \text{ then sorted-list-of-set } (\text{keys } m) \text{ else } [])$

definition *ordered-entries* :: $(\text{'a}::\text{linorder}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a} \times \text{'b}) \text{ list}$

where *ordered-entries* $m = (\text{if finite } (\text{entries } m) \text{ then sorted-key-list-of-set fst } (\text{entries } m) \text{ else } [])$

definition *fold* :: $(\text{'a}::\text{linorder} \Rightarrow \text{'b} \Rightarrow \text{'c} \Rightarrow \text{'c}) \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow \text{'c} \Rightarrow \text{'c}$

where *fold* $f m a = \text{List.fold } (\text{case-prod } f) (\text{ordered-entries } m) a$

definition *is-empty* :: $(\text{'a}, \text{'b}) \text{ mapping} \Rightarrow \text{bool}$

where *is-empty* $m \longleftrightarrow \text{keys } m = \{\}$

definition *size* :: $(\text{'a}, \text{'b}) \text{ mapping} \Rightarrow \text{nat}$

where *size* $m = (\text{if finite } (\text{keys } m) \text{ then card } (\text{keys } m) \text{ else } 0)$

definition *replace* :: $\text{'a} \Rightarrow \text{'b} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$

where *replace* $k v m = (\text{if } k \in \text{keys } m \text{ then update } k v m \text{ else } m)$

definition *default* :: $\text{'a} \Rightarrow \text{'b} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$

where *default* $k v m = (\text{if } k \in \text{keys } m \text{ then } m \text{ else update } k v m)$

Manual derivation of transfer rule is non-trivial

lift-definition *map-entry* :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) *mapping* \Rightarrow ('a, 'b) *mapping*
is

$\lambda k f m.$
 (case *m k* of
 None \Rightarrow *m*
 | *Some v* \Rightarrow *m* (*k* \mapsto (*f v*))) **parametric** *map-entry-parametric* \langle *proof* \rangle

lemma *map-entry-code* [*code*]:

map-entry k f m =
 (case *lookup m k* of
 None \Rightarrow *m*
 | *Some v* \Rightarrow *update k (f v) m*)
 \langle *proof* \rangle

definition *map-default* :: 'a \Rightarrow 'b \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) *mapping* \Rightarrow ('a, 'b) *mapping*

where *map-default k v f m* = *map-entry k f (default k v m)*

definition *of-alist* :: ('k \times 'v) *list* \Rightarrow ('k, 'v) *mapping*

where *of-alist xs* = *foldr* ($\lambda(k, v) m. \text{update } k v m$) *xs empty*

instantiation *mapping* :: (*type*, *type*) *equal*

begin

definition *HOL.equal m1 m2* \longleftrightarrow ($\forall k. \text{lookup } m1 k = \text{lookup } m2 k$)

instance

\langle *proof* \rangle

end

context includes *lifting-syntax*

begin

lemma [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-total A*

and [*transfer-rule*]: *bi-unique B*

shows (*pcr-mapping A B* \implies *pcr-mapping A B* \implies (=)) *HOL.eq HOL.equal*

\langle *proof* \rangle

lemma *of-alist-transfer* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique R1*

shows (*list-all2 (rel-prod R1 R2)* \implies *pcr-mapping R1 R2*) *map-of of-alist*

\langle *proof* \rangle

end

64.5 Properties

lemma *mapping-eqI*: $(\bigwedge x. \text{lookup } m \ x = \text{lookup } m' \ x) \implies m = m'$
 $\langle \text{proof} \rangle$

lemma *mapping-eqI'*:

assumes $\bigwedge x. x \in \text{Mapping.keys } m \implies \text{Mapping.lookup-default } d \ m \ x = \text{Mapping.lookup-default } d \ m' \ x$

and $\text{Mapping.keys } m = \text{Mapping.keys } m'$

shows $m = m'$

$\langle \text{proof} \rangle$

lemma *lookup-update[simp]*: $\text{lookup } (\text{update } k \ v \ m) \ k = \text{Some } v$
 $\langle \text{proof} \rangle$

lemma *lookup-update-neq[simp]*: $k \neq k' \implies \text{lookup } (\text{update } k \ v \ m) \ k' = \text{lookup } m \ k'$
 $\langle \text{proof} \rangle$

lemma *lookup-update'*: $\text{lookup } (\text{update } k \ v \ m) \ k' = (\text{if } k = k' \text{ then } \text{Some } v \text{ else } \text{lookup } m \ k')$
 $\langle \text{proof} \rangle$

lemma *lookup-empty[simp]*: $\text{lookup empty } k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *lookup-delete[simp]*: $\text{lookup } (\text{delete } k \ m) \ k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *lookup-delete-neq[simp]*: $k \neq k' \implies \text{lookup } (\text{delete } k \ m) \ k' = \text{lookup } m \ k'$
 $\langle \text{proof} \rangle$

lemma *lookup-filter*:

$\text{lookup } (\text{filter } P \ m) \ k =$

$(\text{case } \text{lookup } m \ k \ \text{of}$

$\text{None} \Rightarrow \text{None}$

$| \text{Some } v \Rightarrow \text{if } P \ k \ v \ \text{then } \text{Some } v \ \text{else } \text{None})$

$\langle \text{proof} \rangle$

lemma *lookup-map-values*: $\text{lookup } (\text{map-values } f \ m) \ k = \text{map-option } (f \ k) \ (\text{lookup } m \ k)$
 $\langle \text{proof} \rangle$

lemma *lookup-default-empty*: $\text{lookup-default } d \ \text{empty } k = d$
 $\langle \text{proof} \rangle$

lemma *lookup-default-update*: $\text{lookup-default } d \ (\text{update } k \ v \ m) \ k = v$
 $\langle \text{proof} \rangle$

lemma *lookup-default-update-neq*:

$k \neq k' \implies \text{lookup-default } d \text{ (update } k \text{ } v \text{ } m) \text{ } k' = \text{lookup-default } d \text{ } m \text{ } k'$
 ⟨proof⟩

lemma *lookup-default-update'*:

$\text{lookup-default } d \text{ (update } k \text{ } v \text{ } m) \text{ } k' = (\text{if } k = k' \text{ then } v \text{ else } \text{lookup-default } d \text{ } m \text{ } k')$
 ⟨proof⟩

lemma *lookup-default-filter*:

$\text{lookup-default } d \text{ (filter } P \text{ } m) \text{ } k =$
 $(\text{if } P \text{ } k \text{ (lookup-default } d \text{ } m \text{ } k) \text{ then } \text{lookup-default } d \text{ } m \text{ } k \text{ else } d)$
 ⟨proof⟩

lemma *lookup-default-map-values*:

$\text{lookup-default } (f \text{ } k \text{ } d) \text{ (map-values } f \text{ } m) \text{ } k = f \text{ } k \text{ (lookup-default } d \text{ } m \text{ } k)$
 ⟨proof⟩

lemma *lookup-combine-with-key*:

$\text{Mapping.lookup (combine-with-key } f \text{ } m1 \text{ } m2) \text{ } x =$
 $\text{combine-options } (f \text{ } x) \text{ (Mapping.lookup } m1 \text{ } x) \text{ (Mapping.lookup } m2 \text{ } x)$
 ⟨proof⟩

lemma *combine-altdef*: $\text{combine } f \text{ } m1 \text{ } m2 = \text{combine-with-key } (\lambda -. f) \text{ } m1 \text{ } m2$

⟨proof⟩

lemma *lookup-combine*:

$\text{Mapping.lookup (combine } f \text{ } m1 \text{ } m2) \text{ } x =$
 $\text{combine-options } f \text{ (Mapping.lookup } m1 \text{ } x) \text{ (Mapping.lookup } m2 \text{ } x)$
 ⟨proof⟩

lemma *lookup-default-neutral-combine-with-key*:

assumes $\bigwedge x. f \text{ } k \text{ } d \text{ } x = x \bigwedge x. f \text{ } k \text{ } x \text{ } d = x$
shows $\text{Mapping.lookup-default } d \text{ (combine-with-key } f \text{ } m1 \text{ } m2) \text{ } k =$
 $f \text{ } k \text{ (Mapping.lookup-default } d \text{ } m1 \text{ } k) \text{ (Mapping.lookup-default } d \text{ } m2 \text{ } k)$
 ⟨proof⟩

lemma *lookup-default-neutral-combine*:

assumes $\bigwedge x. f \text{ } d \text{ } x = x \bigwedge x. f \text{ } x \text{ } d = x$
shows $\text{Mapping.lookup-default } d \text{ (combine } f \text{ } m1 \text{ } m2) \text{ } x =$
 $f \text{ (Mapping.lookup-default } d \text{ } m1 \text{ } x) \text{ (Mapping.lookup-default } d \text{ } m2 \text{ } x)$
 ⟨proof⟩

lemma *lookup-map-entry*: $\text{lookup (map-entry } x \text{ } f \text{ } m) \text{ } x = \text{map-option } f \text{ (lookup } m \text{ } x)$

⟨proof⟩

lemma *lookup-map-entry-neq*: $x \neq y \implies \text{lookup (map-entry } x \text{ } f \text{ } m) \text{ } y = \text{lookup } m \text{ } y$

⟨proof⟩

lemma *lookup-map-entry'*:

lookup (map-entry x f m) y =
(if x = y then map-option f (lookup m y) else lookup m y)
⟨proof⟩

lemma *lookup-default*: *lookup (default x d m) x = Some (lookup-default d m x)*
⟨proof⟩

lemma *lookup-default-neq*: *x ≠ y ⟹ lookup (default x d m) y = lookup m y*
⟨proof⟩

lemma *lookup-default'*:

lookup (default x d m) y =
(if x = y then Some (lookup-default d m x) else lookup m y)
⟨proof⟩

lemma *lookup-map-default*: *lookup (map-default x d f m) x = Some (f (lookup-default d m x))*
⟨proof⟩

lemma *lookup-map-default-neq*: *x ≠ y ⟹ lookup (map-default x d f m) y = lookup m y*
⟨proof⟩

lemma *lookup-map-default'*:

lookup (map-default x d f m) y =
(if x = y then Some (f (lookup-default d m x)) else lookup m y)
⟨proof⟩

lemma *lookup-tabulate*:

assumes *distinct xs*

shows *Mapping.lookup (Mapping.tabulate xs f) x = (if x ∈ set xs then Some (f x) else None)*
⟨proof⟩

lemma *lookup-of-alist*: *lookup (of-alist xs) k = map-of xs k*
⟨proof⟩

lemma *keys-is-none-rep* [code-unfold]: *k ∈ keys m ⟷ ¬ (Option.is-none (lookup m k))*
⟨proof⟩

lemma *update-update*:

update k v (update k w m) = update k v m
k ≠ l ⟹ update k v (update l w m) = update l w (update k v m)
⟨proof⟩

lemma *update-delete* [simp]: *update k v (delete k m) = update k v m*
⟨proof⟩

lemma *delete-update*:

$delete\ k\ (update\ k\ v\ m) = delete\ k\ m$
 $k \neq l \implies delete\ k\ (update\ l\ v\ m) = update\ l\ v\ (delete\ k\ m)$
 ⟨proof⟩

lemma *delete-empty [simp]*: $delete\ k\ empty = empty$

⟨proof⟩

lemma *Mapping-delete-if-notin-keys[simp]*:

$k \notin keys\ m \implies delete\ k\ m = m$
 ⟨proof⟩

lemma *replace-update*:

$k \notin keys\ m \implies replace\ k\ v\ m = m$
 $k \in keys\ m \implies replace\ k\ v\ m = update\ k\ v\ m$
 ⟨proof⟩

lemma *map-values-update*: $map-values\ f\ (update\ k\ v\ m) = update\ k\ (f\ k\ v)\ (map-values\ f\ m)$

⟨proof⟩

lemma *size-mono*: $finite\ (keys\ m') \implies keys\ m \subseteq keys\ m' \implies size\ m \leq size\ m'$

⟨proof⟩

lemma *size-empty [simp]*: $size\ empty = 0$

⟨proof⟩

lemma *size-update*:

$finite\ (keys\ m) \implies size\ (update\ k\ v\ m) =$
 $(if\ k \in keys\ m\ then\ size\ m\ else\ Suc\ (size\ m))$
 ⟨proof⟩

lemma *size-delete*: $size\ (delete\ k\ m) = (if\ k \in keys\ m\ then\ size\ m - 1\ else\ size\ m)$

⟨proof⟩

lemma *size-tabulate [simp]*: $size\ (tabulate\ ks\ f) = length\ (remdups\ ks)$

⟨proof⟩

lemma *keys-filter*: $keys\ (filter\ P\ m) \subseteq keys\ m$

⟨proof⟩

lemma *size-filter*: $finite\ (keys\ m) \implies size\ (filter\ P\ m) \leq size\ m$

⟨proof⟩

lemma *bulkload-tabulate*: $bulkload\ xs = tabulate\ [0..<length\ xs]\ (nth\ xs)$

⟨proof⟩

lemma *is-empty-empty [simp]*: $is-empty\ empty$

$\langle proof \rangle$

lemma *is-empty-update* [simp]: $\neg is_empty (update\ k\ v\ m)$
 $\langle proof \rangle$

lemma *is-empty-delete*: $is_empty (delete\ k\ m) \longleftrightarrow is_empty\ m \vee keys\ m = \{k\}$
 $\langle proof \rangle$

lemma *is-empty-replace* [simp]: $is_empty (replace\ k\ v\ m) \longleftrightarrow is_empty\ m$
 $\langle proof \rangle$

lemma *is-empty-default* [simp]: $\neg is_empty (default\ k\ v\ m)$
 $\langle proof \rangle$

lemma *is-empty-map-entry* [simp]: $is_empty (map_entry\ k\ f\ m) \longleftrightarrow is_empty\ m$
 $\langle proof \rangle$

lemma *is-empty-map-values* [simp]: $is_empty (map_values\ f\ m) \longleftrightarrow is_empty\ m$
 $\langle proof \rangle$

lemma *is-empty-map-default* [simp]: $\neg is_empty (map_default\ k\ v\ f\ m)$
 $\langle proof \rangle$

lemma *keys-dom-lookup*: $keys\ m = dom\ (Mapping.lookup\ m)$
 $\langle proof \rangle$

lemma *keys-empty* [simp]: $keys\ empty = \{\}$
 $\langle proof \rangle$

lemma *in-keysD*: $k \in keys\ m \implies \exists v. lookup\ m\ k = Some\ v$
 $\langle proof \rangle$

lemma *keys-update* [simp]: $keys (update\ k\ v\ m) = insert\ k (keys\ m)$
 $\langle proof \rangle$

lemma *keys-delete* [simp]: $keys (delete\ k\ m) = keys\ m - \{k\}$
 $\langle proof \rangle$

lemma *keys-replace* [simp]: $keys (replace\ k\ v\ m) = keys\ m$
 $\langle proof \rangle$

lemma *keys-default* [simp]: $keys (default\ k\ v\ m) = insert\ k (keys\ m)$
 $\langle proof \rangle$

lemma *keys-map-entry* [simp]: $keys (map_entry\ k\ f\ m) = keys\ m$
 $\langle proof \rangle$

lemma *keys-map-default* [simp]: $keys (map_default\ k\ v\ f\ m) = insert\ k (keys\ m)$
 $\langle proof \rangle$

lemma *keys-map-values* [simp]: $\text{keys } (\text{map-values } f \ m) = \text{keys } m$
 ⟨proof⟩

lemma *keys-combine-with-key* [simp]:
 $\text{Mapping.keys } (\text{combine-with-key } f \ m1 \ m2) = \text{Mapping.keys } m1 \cup \text{Mapping.keys } m2$
 ⟨proof⟩

lemma *keys-combine* [simp]: $\text{Mapping.keys } (\text{combine } f \ m1 \ m2) = \text{Mapping.keys } m1 \cup \text{Mapping.keys } m2$
 ⟨proof⟩

lemma *keys-tabulate* [simp]: $\text{keys } (\text{tabulate } ks \ f) = \text{set } ks$
 ⟨proof⟩

lemma *keys-of-alist* [simp]: $\text{keys } (\text{of-alist } xs) = \text{set } (\text{List.map } \text{fst } xs)$
 ⟨proof⟩

lemma *keys-bulkload* [simp]: $\text{keys } (\text{bulkload } xs) = \{0..<\text{length } xs\}$
 ⟨proof⟩

lemma *finite-keys-update*[simp]:
 $\text{finite } (\text{keys } (\text{update } k \ v \ m)) = \text{finite } (\text{keys } m)$
 ⟨proof⟩

lemma *set-ordered-keys*[simp]:
 $\text{finite } (\text{Mapping.keys } m) \implies \text{set } (\text{Mapping.ordered-keys } m) = \text{Mapping.keys } m$
 ⟨proof⟩

lemma *distinct-ordered-keys* [simp]: $\text{distinct } (\text{ordered-keys } m)$
 ⟨proof⟩

lemma *ordered-keys-infinite* [simp]: $\neg \text{finite } (\text{keys } m) \implies \text{ordered-keys } m = []$
 ⟨proof⟩

lemma *ordered-keys-empty* [simp]: $\text{ordered-keys } \text{empty} = []$
 ⟨proof⟩

lemma *sorted-ordered-keys*[simp]: $\text{sorted } (\text{ordered-keys } m)$
 ⟨proof⟩

lemma *ordered-keys-update* [simp]:
 $k \in \text{keys } m \implies \text{ordered-keys } (\text{update } k \ v \ m) = \text{ordered-keys } m$
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies$
 $\text{ordered-keys } (\text{update } k \ v \ m) = \text{insert } k \ (\text{ordered-keys } m)$
 ⟨proof⟩

lemma *ordered-keys-delete* [simp]: $\text{ordered-keys } (\text{delete } k \ m) = \text{remove1 } k \ (\text{ordered-keys } m)$

$m)$
 $\langle \text{proof} \rangle$

lemma *ordered-keys-replace* [simp]: $\text{ordered-keys } (\text{replace } k \ v \ m) = \text{ordered-keys } m$
 $\langle \text{proof} \rangle$

lemma *ordered-keys-default* [simp]:
 $k \in \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{ordered-keys } m$
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{insert } k$
 $(\text{ordered-keys } m)$
 $\langle \text{proof} \rangle$

lemma *ordered-keys-map-entry* [simp]: $\text{ordered-keys } (\text{map-entry } k \ f \ m) = \text{ordered-keys } m$
 $\langle \text{proof} \rangle$

lemma *ordered-keys-map-default* [simp]:
 $k \in \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{ordered-keys } m$
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{insert } k$
 $(\text{ordered-keys } m)$
 $\langle \text{proof} \rangle$

lemma *ordered-keys-tabulate* [simp]: $\text{ordered-keys } (\text{tabulate } ks \ f) = \text{sort } (\text{remdups } ks)$
 $\langle \text{proof} \rangle$

lemma *ordered-keys-bulkload* [simp]: $\text{ordered-keys } (\text{bulkload } ks) = [0..<\text{length } ks]$
 $\langle \text{proof} \rangle$

lemma *tabulate-fold*: $\text{tabulate } xs \ f = \text{List.fold } (\lambda k \ m. \text{update } k \ (f \ k) \ m) \ xs \ \text{empty}$
 $\langle \text{proof} \rangle$

lemma *All-mapping-mono*:
 $(\bigwedge k \ v. k \in \text{keys } m \implies P \ k \ v \implies Q \ k \ v) \implies \text{All-mapping } m \ P \implies \text{All-mapping } m \ Q$
 $\langle \text{proof} \rangle$

lemma *All-mapping-empty* [simp]: $\text{All-mapping } \text{Mapping.empty} \ P$
 $\langle \text{proof} \rangle$

lemma *All-mapping-update-iff*:
 $\text{All-mapping } (\text{Mapping.update } k \ v \ m) \ P \longleftrightarrow P \ k \ v \wedge \text{All-mapping } m \ (\lambda k' \ v'. k = k' \vee P \ k' \ v')$
 $\langle \text{proof} \rangle$

lemma *All-mapping-update*:
 $P \ k \ v \implies \text{All-mapping } m \ (\lambda k' \ v'. k = k' \vee P \ k' \ v') \implies \text{All-mapping } (\text{Mapping.update } k \ v \ m) \ P$
 $\langle \text{proof} \rangle$

lemma *All-mapping-filter-iff*: *All-mapping (filter P m) Q \longleftrightarrow All-mapping m (λk v. P k v \longrightarrow Q k v)*
 $\langle \text{proof} \rangle$

lemma *All-mapping-filter*: *All-mapping m Q \implies All-mapping (filter P m) Q*
 $\langle \text{proof} \rangle$

lemma *All-mapping-map-values*: *All-mapping (map-values f m) P \longleftrightarrow All-mapping m (λk v. P k (f k v))*
 $\langle \text{proof} \rangle$

lemma *All-mapping-tabulate*: *($\forall x \in \text{set } xs. P x (f x)$) \implies All-mapping (Mapping.tabulate xs f) P*
 $\langle \text{proof} \rangle$

lemma *All-mapping-alist*:
 $(\bigwedge k v. (k, v) \in \text{set } xs \implies P k v) \implies \text{All-mapping (Mapping.of-alist } xs) P$
 $\langle \text{proof} \rangle$

lemma *combine-empty [simp]*: *combine f Mapping.empty y = y combine f y Mapping.empty = y*
 $\langle \text{proof} \rangle$

lemma (*in abel-semigroup*) *comm-monoid-set-combine*: *comm-monoid-set (combine f) Mapping.empty*
 $\langle \text{proof} \rangle$

locale *combine-mapping-abel-semigroup* = *abel-semigroup*
begin

sublocale *combine*: *comm-monoid-set combine f Mapping.empty*
 $\langle \text{proof} \rangle$

lemma *fold-combine-code*:
 $\text{combine.F } g (\text{set } xs) = \text{foldr } (\lambda x. \text{combine } f (g x)) (\text{remdups } xs) \text{ Mapping.empty}$
 $\langle \text{proof} \rangle$

lemma *keys-fold-combine*: *finite A \implies Mapping.keys (combine.F g A) = ($\bigcup x \in A. \text{Mapping.keys } (g x)$)*
 $\langle \text{proof} \rangle$

end

64.5.1 entries, ordered-entries, and fold

context *linorder*
begin

sublocale *folding-Map-graph*: *folding-insort-key* (\leq) ($<$) *Map.graph* *m* *fst* **for** *m*
 ⟨*proof*⟩

end

lemma *sorted-fst-list-of-set-insort-Map-graph*[*simp*]:
 assumes *finite* (*dom m*) *fst* $x \notin \text{dom } m$
 shows *sorted-key-list-of-set* *fst* (*insert* *x* (*Map.graph* *m*))
 = *insort-key* *fst* *x* (*sorted-key-list-of-set* *fst* (*Map.graph* *m*))
 ⟨*proof*⟩

lemma *sorted-fst-list-of-set-insort-insert-Map-graph*[*simp*]:
 assumes *finite* (*dom m*) *fst* $x \notin \text{dom } m$
 shows *sorted-key-list-of-set* *fst* (*insert* *x* (*Map.graph* *m*))
 = *insort-insert-key* *fst* *x* (*sorted-key-list-of-set* *fst* (*Map.graph* *m*))
 ⟨*proof*⟩

lemma *linorder-finite-Map-induct*[*consumes 1, case-names empty update*]:
 fixes *m* :: 'a::linorder \rightarrow 'b
 assumes *finite* (*dom m*)
 assumes *P* *Map.empty*
 assumes $\bigwedge k \ v \ m. \llbracket \text{finite } (\text{dom } m); k \notin \text{dom } m; (\bigwedge k'. k' \in \text{dom } m \implies k' \leq k);$
 $P \ m \rrbracket \implies P \ (m(k \mapsto v))$
 shows *P* *m*
 ⟨*proof*⟩

lemma *delete-insort-fst*[*simp*]: *AList.delete* *k* (*insort-key* *fst* (*k*, *v*) *xs*) = *AL-*
ist.delete *k* *xs*
 ⟨*proof*⟩

lemma *insort-fst-delete*: $\llbracket \text{fst } x \neq k2; \text{sorted } (\text{List.map } \text{fst } xs) \rrbracket$
 $\implies \text{insort-key } \text{fst } x \ (\text{AList.delete } k2 \ xs) = \text{AList.delete } k2 \ (\text{insort-key } \text{fst } x \ xs)$
 ⟨*proof*⟩

lemma *sorted-fst-list-of-set-Map-graph-fun-upd-None*[*simp*]:
sorted-key-list-of-set *fst* (*Map.graph* (*m*(*k* := *None*)))
 = *AList.delete* *k* (*sorted-key-list-of-set* *fst* (*Map.graph* *m*))
 ⟨*proof*⟩

lemma *entries-empty*[*simp*]: *entries empty* = {}
 ⟨*proof*⟩

lemma *entries-lookup*: *entries* *m* = *Map.graph* (*lookup* *m*)
 ⟨*proof*⟩

lemma *in-entriesI*: *lookup* *m* *k* = *Some* *v* $\implies (k, v) \in \text{entries } m$
 ⟨*proof*⟩

lemma *in-entriesD*: $(k, v) \in \text{entries } m \implies \text{lookup } m \ k = \text{Some } v$
 ⟨proof⟩

lemma *fst-image-entries-eq-keys[simp]*: $\text{fst} \ ` \ \text{Mapping.entries } m = \text{Mapping.keys } m$
 ⟨proof⟩

lemma *finite-entries-iff-finite-keys[simp]*:
 $\text{finite } (\text{entries } m) = \text{finite } (\text{keys } m)$
 ⟨proof⟩

lemma *entries-update*:
 $\text{entries } (\text{update } k \ v \ m) = \text{insert } (k, v) \ (\text{entries } (\text{delete } k \ m))$
 ⟨proof⟩

lemma *entries-delete*:
 $\text{entries } (\text{delete } k \ m) = \{e \in \text{entries } m. \text{fst } e \neq k\}$
 ⟨proof⟩

lemma *entries-of-alist[simp]*:
 $\text{distinct } (\text{List.map } \text{fst } xs) \implies \text{entries } (\text{of-alist } xs) = \text{set } xs$
 ⟨proof⟩

lemma *entries-keysD*:
 $x \in \text{entries } m \implies \text{fst } x \in \text{keys } m$
 ⟨proof⟩

lemma *set-ordered-entries[simp]*:
 $\text{finite } (\text{keys } m) \implies \text{set } (\text{ordered-entries } m) = \text{entries } m$
 ⟨proof⟩

lemma *distinct-ordered-entries[simp]*: $\text{distinct } (\text{List.map } \text{fst } (\text{ordered-entries } m))$
 ⟨proof⟩

lemma *sorted-ordered-entries[simp]*: $\text{sorted } (\text{List.map } \text{fst } (\text{ordered-entries } m))$
 ⟨proof⟩

lemma *ordered-entries-infinite[simp]*:
 $\neg \text{finite } (\text{Mapping.keys } m) \implies \text{ordered-entries } m = []$
 ⟨proof⟩

lemma *ordered-entries-empty[simp]*: $\text{ordered-entries } \text{empty} = []$
 ⟨proof⟩

lemma *ordered-entries-update[simp]*:
 assumes $\text{finite } (\text{keys } m)$
 shows $\text{ordered-entries } (\text{update } k \ v \ m)$
 $= \text{insort-insert-key } \text{fst } (k, v) \ (\text{AList.delete } k \ (\text{ordered-entries } m))$
 ⟨proof⟩


```

lemma ordered-entries-delete[simp]:
  ordered-entries (delete k m) = AList.delete k (ordered-entries m)
  ⟨proof⟩

lemma map-fst-ordered-entries[simp]:
  List.map fst (ordered-entries m) = ordered-keys m
  ⟨proof⟩

lemma fold-empty[simp]: fold f empty a = a
  ⟨proof⟩

lemma insort-key-is-snoc-if-sorted-and-distinct:
  assumes sorted (List.map f xs) f y  $\notin$  f ‘ set xs  $\forall$  x  $\in$  set xs. f x  $\leq$  f y
  shows insort-key f y xs = xs @ [y]
  ⟨proof⟩

lemma fold-update:
  assumes finite (keys m)
  assumes  $k \notin \text{keys } m \wedge k'. k' \in \text{keys } m \implies k' \leq k$ 
  shows fold f (update k v m) a = f k v (fold f m a)
  ⟨proof⟩

lemma linorder-finite-Mapping-induct[consumes 1, case-names empty update]:
  fixes m :: ('a::linorder, 'b) mapping
  assumes finite (keys m)
  assumes P empty
  assumes  $\bigwedge k \ v \ m.$ 
     $\llbracket \text{finite } (\text{keys } m); k \notin \text{keys } m; (\bigwedge k'. k' \in \text{keys } m \implies k' \leq k); P \ m \rrbracket$ 
     $\implies P \ (\text{update } k \ v \ m)$ 
  shows P m
  ⟨proof⟩

```

64.6 Code generator setup

```

hide-const (open) empty is-empty rep lookup lookup-default filter update delete
ordered-keys
  keys size replace default map-entry map-default tabulate bulkload map map-values
combine of-alist
  entries ordered-entries fold

```

end

65 Monad notation for arbitrary types

```

theory Monad-Syntax
  imports Main
begin

```

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

consts

bind :: 'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd (**infixl** $\langle \gg \rangle$ 54)

notation (ASCII)

bind (**infixl** $\langle > \Rightarrow \rangle$ 54)

abbreviation (do-notation)

bind-do :: 'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd

where *bind-do* \equiv *bind*

notation (output)

bind-do (**infixl** $\langle \gg \rangle$ 54)

notation (ASCII output)

bind-do (**infixl** $\langle > \Rightarrow \rangle$ 54)

nonterminal do-binds and do-bind**syntax**

-do-block :: *do-binds* \Rightarrow 'a

($\langle \langle \text{open-block notation} = \langle \text{mixfix do block} \rangle \rangle \text{do} \{ // (2 \text{ -}) // \} \rangle$ [12] 62)

-do-bind :: [*pttrn*, 'a] \Rightarrow *do-bind*

($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix do bind} \rangle \rangle \text{ - } \leftarrow / \text{ -} \rangle$ 13)

-do-let :: [*pttrn*, 'a] \Rightarrow *do-bind*

($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix do let} \rangle \rangle \text{let - } = / \text{ -} \rangle$ [1000, 13] 13)

-do-then :: 'a \Rightarrow *do-bind* ($\langle \cdot \rangle$ [14] 13)

-do-final :: 'a \Rightarrow *do-binds* ($\langle \cdot \rangle$)

-do-cons :: [*do-bind*, *do-binds*] \Rightarrow *do-binds*

($\langle \langle \text{open-block notation} = \langle \text{infix do next} \rangle \rangle \text{ ; } // \text{ -} \rangle$ [13, 12] 12)

-thenM :: ['a, 'b] \Rightarrow 'c (**infixl** $\langle \gg \rangle$ 54)

syntax (ASCII)

-do-bind :: [*pttrn*, 'a] \Rightarrow *do-bind*

($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix do bind} \rangle \rangle \text{ - } \leftarrow / \text{ -} \rangle$ 13)

-thenM :: ['a, 'b] \Rightarrow 'c (**infixl** $\langle > \rangle$ 54)

syntax-consts

-do-block -do-cons -do-bind -do-then \Rightarrow *bind* **and**

-do-let \Rightarrow *Let*

translations

-do-block (*-do-cons* (*-do-then* *t*) (*-do-final* *e*))

\Rightarrow *CONST bind-do t* ($\lambda \text{-} . e$)

-do-block (*-do-cons* (*-do-bind* *p t*) (*-do-final* *e*))

\Rightarrow *CONST bind-do t* ($\lambda p . e$)

-do-block (*-do-cons* (*-do-let* *p t*) *bs*)

$$\begin{aligned}
& \Rightarrow \text{let } p = t \text{ in } \text{-do-block } bs \\
& \text{-do-block } (\text{-do-cons } b \text{ } (\text{-do-cons } c \text{ } cs)) \\
& \Rightarrow \text{-do-block } (\text{-do-cons } b \text{ } (\text{-do-final } (\text{-do-block } (\text{-do-cons } c \text{ } cs)))) \\
& \text{-do-cons } (\text{-do-let } p \text{ } t) \text{ } (\text{-do-final } s) \\
& \Rightarrow \text{-do-final } (\text{let } p = t \text{ in } s) \\
& \text{-do-block } (\text{-do-final } e) \rightarrow e \\
& (m \gg n) \rightarrow (m \gg (\lambda \cdot n))
\end{aligned}$$
adhoc-overloading

$$\text{bind} \Rightarrow \text{Set.bind Predicate.bind Option.bind List.bind}$$
end**66 Less common functions on lists****theory** *More-List***imports** *Main***begin****definition** *strip-while* :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'a list**where**

$$\text{strip-while } P = \text{rev} \circ \text{dropWhile } P \circ \text{rev}$$
lemma *strip-while-rev* [simp]:
$$\text{strip-while } P \text{ } (\text{rev } xs) = \text{rev } (\text{dropWhile } P \text{ } xs)$$
*<proof>***lemma** *strip-while-Nil* [simp]:
$$\text{strip-while } P \text{ } [] = []$$
*<proof>***lemma** *strip-while-append* [simp]:
$$\neg P \text{ } x \Longrightarrow \text{strip-while } P \text{ } (xs @ [x]) = xs @ [x]$$
*<proof>***lemma** *strip-while-append-rec* [simp]:
$$P \text{ } x \Longrightarrow \text{strip-while } P \text{ } (xs @ [x]) = \text{strip-while } P \text{ } xs$$
*<proof>***lemma** *strip-while-Cons* [simp]:
$$\neg P \text{ } x \Longrightarrow \text{strip-while } P \text{ } (x \# xs) = x \# \text{strip-while } P \text{ } xs$$
*<proof>***lemma** *strip-while-eq-Nil* [simp]:
$$\text{strip-while } P \text{ } xs = [] \longleftrightarrow (\forall x \in \text{set } xs. P \text{ } x)$$
*<proof>***lemma** *strip-while-eq-Cons-rec*:
$$\text{strip-while } P \text{ } (x \# xs) = x \# \text{strip-while } P \text{ } xs \longleftrightarrow \neg (P \text{ } x \wedge (\forall x \in \text{set } xs. P \text{ } x))$$

$\langle \text{proof} \rangle$

lemma *split-strip-while-append*:

fixes $xs :: 'a \text{ list}$

obtains $ys \ zs :: 'a \text{ list}$

where $\text{strip-while } P \ xs = ys$ **and** $\forall x \in \text{set } zs. P \ x$ **and** $xs = ys @ zs$
 $\langle \text{proof} \rangle$

lemma *strip-while-snoc* [simp]:

$\text{strip-while } P \ (xs @ [x]) = (\text{if } P \ x \text{ then } \text{strip-while } P \ xs \text{ else } xs @ [x])$

$\langle \text{proof} \rangle$

lemma *strip-while-map*:

$\text{strip-while } P \ (\text{map } f \ xs) = \text{map } f \ (\text{strip-while } (P \circ f) \ xs)$

$\langle \text{proof} \rangle$

lemma *strip-while-dropWhile-commute*:

$\text{strip-while } P \ (\text{dropWhile } Q \ xs) = \text{dropWhile } Q \ (\text{strip-while } P \ xs)$

$\langle \text{proof} \rangle$

lemma *dropWhile-strip-while-commute*:

$\text{dropWhile } P \ (\text{strip-while } Q \ xs) = \text{strip-while } Q \ (\text{dropWhile } P \ xs)$

$\langle \text{proof} \rangle$

definition *no-leading* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$

where

$\text{no-leading } P \ xs \longleftrightarrow (xs \neq [] \longrightarrow \neg P \ (\text{hd } xs))$

lemma *no-leading-Nil* [iff]:

$\text{no-leading } P \ []$

$\langle \text{proof} \rangle$

lemma *no-leading-Cons* [iff]:

$\text{no-leading } P \ (x \# xs) \longleftrightarrow \neg P \ x$

$\langle \text{proof} \rangle$

lemma *no-leading-append* [simp]:

$\text{no-leading } P \ (xs @ ys) \longleftrightarrow \text{no-leading } P \ xs \wedge (xs = [] \longrightarrow \text{no-leading } P \ ys)$

$\langle \text{proof} \rangle$

lemma *no-leading-dropWhile* [simp]:

$\text{no-leading } P \ (\text{dropWhile } P \ xs)$

$\langle \text{proof} \rangle$

lemma *dropWhile-eq-obtain-leading*:

assumes $\text{dropWhile } P \ xs = ys$

obtains zs **where** $xs = zs @ ys$ **and** $\bigwedge z. z \in \text{set } zs \Longrightarrow P \ z$ **and** $\text{no-leading } P \ ys$

$\langle proof \rangle$

lemma *dropWhile-idem-iff*:

$dropWhile\ P\ xs = xs \longleftrightarrow no-leading\ P\ xs$

$\langle proof \rangle$

abbreviation *no-trailing* :: ($'a \Rightarrow bool$) $\Rightarrow 'a\ list \Rightarrow bool$

where

$no-trailing\ P\ xs \equiv no-leading\ P\ (rev\ xs)$

lemma *no-trailing-unfold*:

$no-trailing\ P\ xs \longleftrightarrow (xs \neq [] \longrightarrow \neg P\ (last\ xs))$

$\langle proof \rangle$

lemma *no-trailing-Nil [iff]*:

$no-trailing\ P\ []$

$\langle proof \rangle$

lemma *no-trailing-Cons [simp]*:

$no-trailing\ P\ (x \# xs) \longleftrightarrow no-trailing\ P\ xs \wedge (xs = [] \longrightarrow \neg P\ x)$

$\langle proof \rangle$

lemma *no-trailing-append*:

$no-trailing\ P\ (xs @ ys) \longleftrightarrow no-trailing\ P\ ys \wedge (ys = [] \longrightarrow no-trailing\ P\ xs)$

$\langle proof \rangle$

lemma *no-trailing-append-Cons [simp]*:

$no-trailing\ P\ (xs @ y \# ys) \longleftrightarrow no-trailing\ P\ (y \# ys)$

$\langle proof \rangle$

lemma *no-trailing-strip-while [simp]*:

$no-trailing\ P\ (strip-while\ P\ xs)$

$\langle proof \rangle$

lemma *strip-while-idem [simp]*:

$no-trailing\ P\ xs \Longrightarrow strip-while\ P\ xs = xs$

$\langle proof \rangle$

lemma *strip-while-eq-obtain-trailing*:

assumes $strip-while\ P\ xs = ys$

obtains zs **where** $xs = ys @ zs$ **and** $\bigwedge z. z \in set\ zs \Longrightarrow P\ z$ **and** $no-trailing\ P$

ys

$\langle proof \rangle$

lemma *strip-while-idem-iff*:

$strip-while\ P\ xs = xs \longleftrightarrow no-trailing\ P\ xs$

$\langle proof \rangle$

lemma *no-trailing-map*:

no-trailing P ($\text{map } f \text{ } xs$) \longleftrightarrow *no-trailing* ($P \circ f$) xs
 $\langle \text{proof} \rangle$

lemma *no-trailing-drop* [*simp*]:

no-trailing P ($\text{drop } n \text{ } xs$) **if** *no-trailing* $P \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *no-trailing-upt* [*simp*]:

no-trailing P [$n..<m$] \longleftrightarrow ($n < m \longrightarrow \neg P (m - 1)$)
 $\langle \text{proof} \rangle$

definition *nth-default* :: $'a \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a$

where

nth-default $\text{dflt } xs \text{ } n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } \text{dflt})$

lemma *nth-default-nth*:

$n < \text{length } xs \implies \text{nth-default } \text{dflt } xs \text{ } n = xs ! n$
 $\langle \text{proof} \rangle$

lemma *nth-default-beyond*:

$\text{length } xs \leq n \implies \text{nth-default } \text{dflt } xs \text{ } n = \text{dflt}$
 $\langle \text{proof} \rangle$

lemma *nth-default-Nil* [*simp*]:

nth-default $\text{dflt } [] \text{ } n = \text{dflt}$
 $\langle \text{proof} \rangle$

lemma *nth-default-Cons*:

nth-default $\text{dflt } (x \# xs) \text{ } n = (\text{case } n \text{ of } 0 \Rightarrow x \mid \text{Suc } n' \Rightarrow \text{nth-default } \text{dflt } xs \text{ } n')$
 $\langle \text{proof} \rangle$

lemma *nth-default-Cons-0* [*simp*]:

nth-default $\text{dflt } (x \# xs) \text{ } 0 = x$
 $\langle \text{proof} \rangle$

lemma *nth-default-Cons-Suc* [*simp*]:

nth-default $\text{dflt } (x \# xs) \text{ } (\text{Suc } n) = \text{nth-default } \text{dflt } xs \text{ } n$
 $\langle \text{proof} \rangle$

lemma *nth-default-replicate-dflt* [*simp*]:

nth-default $\text{dflt } (\text{replicate } n \text{ } \text{dflt}) \text{ } m = \text{dflt}$
 $\langle \text{proof} \rangle$

lemma *nth-default-append*:

nth-default $\text{dflt } (xs @ ys) \text{ } n =$
 $(\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } \text{nth-default } \text{dflt } ys \text{ } (n - \text{length } xs))$
 $\langle \text{proof} \rangle$

lemma *nth-default-append-trailing* [simp]:

$nth\text{-default}\ dflt\ (xs\ @\ replicate\ n\ dflt) = nth\text{-default}\ dflt\ xs$
 ⟨proof⟩

lemma *nth-default-snoc-default* [simp]:

$nth\text{-default}\ dflt\ (xs\ @\ [dflt]) = nth\text{-default}\ dflt\ xs$
 ⟨proof⟩

lemma *nth-default-eq-dflt-iff*:

$nth\text{-default}\ dflt\ xs\ k = dflt \longleftrightarrow (k < length\ xs \longrightarrow xs\ !\ k = dflt)$
 ⟨proof⟩

lemma *nth-default-take-eq*:

$nth\text{-default}\ dflt\ (take\ m\ xs)\ n =$
 (if $n < m$ then $nth\text{-default}\ dflt\ xs\ n$ else $dflt$)
 ⟨proof⟩

lemma *in-enumerate-iff-nth-default-eq*:

$x \neq dflt \implies (n, x) \in set\ (enumerate\ 0\ xs) \longleftrightarrow nth\text{-default}\ dflt\ xs\ n = x$
 ⟨proof⟩

lemma *last-conv-nth-default*:

assumes $xs \neq []$
shows $last\ xs = nth\text{-default}\ dflt\ xs\ (length\ xs - 1)$
 ⟨proof⟩

lemma *nth-default-map-eq*:

$f\ dflt' = dflt \implies nth\text{-default}\ dflt\ (map\ f\ xs)\ n = f\ (nth\text{-default}\ dflt'\ xs\ n)$
 ⟨proof⟩

lemma *finite-nth-default-neq-default* [simp]:

$finite\ \{k. nth\text{-default}\ dflt\ xs\ k \neq dflt\}$
 ⟨proof⟩

lemma *sorted-list-of-set-nth-default*:

$sorted\text{-list-of-set}\ \{k. nth\text{-default}\ dflt\ xs\ k \neq dflt\} = map\ fst\ (filter\ (\lambda(-, x). x \neq dflt)\ (enumerate\ 0\ xs))$
 ⟨proof⟩

lemma *map-nth-default*:

$map\ (nth\text{-default}\ x\ xs)\ [0..<length\ xs] = xs$
 ⟨proof⟩

lemma *range-nth-default* [simp]:

$range\ (nth\text{-default}\ dflt\ xs) = insert\ dflt\ (set\ xs)$
 ⟨proof⟩

lemma *nth-strip-while*:

assumes $n < \text{length } (\text{strip-while } P \text{ } xs)$
shows $\text{strip-while } P \text{ } xs ! n = xs ! n$
 $\langle \text{proof} \rangle$

lemma *length-strip-while-le*:
 $\text{length } (\text{strip-while } P \text{ } xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *nth-default-strip-while-dflt* [simp]:
 $\text{nth-default dflt } (\text{strip-while } ((=) \text{ dflt}) \text{ } xs) = \text{nth-default dflt } xs$
 $\langle \text{proof} \rangle$

lemma *nth-default-eq-iff*:
 $\text{nth-default dflt } xs = \text{nth-default dflt } ys$
 $\longleftrightarrow \text{strip-while } (HOL.eq \text{ dflt}) \text{ } xs = \text{strip-while } (HOL.eq \text{ dflt}) \text{ } ys$ **(is** $?P \longleftrightarrow ?Q)$
 $\langle \text{proof} \rangle$

lemma *nth-default-map2*:
 $\langle \text{nth-default d } (\text{map2 } f \text{ } xs \text{ } ys) \text{ } n = f (\text{nth-default d1 } xs \text{ } n) (\text{nth-default d2 } ys \text{ } n) \rangle$
if $\langle \text{length } xs = \text{length } ys \rangle$ **and** $\langle f \text{ d1 d2} = d \rangle$ **for** $bs \text{ } cs$
 $\langle \text{proof} \rangle$

end

theory *Cancellation*
imports *Main*
begin

named-theorems *cancelation-simproc-pre* \langle These theorems are here to normalise the term. Special handling of constructors should be here. Remark that only the simproc @{term NO-MATCH} is also included. \rangle

named-theorems *cancelation-simproc-post* \langle These theorems are here to normalise the term, after the cancelation simproc. Normalisation of $\langle \text{iterate-add} \rangle$ back to the normale representation should be put here. \rangle

named-theorems *cancelation-simproc-eq-elim* \langle These theorems are here to help deriving contradiction (e.g., $\langle \text{Suc } - = 0 \rangle$). \rangle

definition *iterate-add* :: $\langle \text{nat} \Rightarrow 'a :: \text{cancel-comm-monoid-add} \Rightarrow 'a \rangle$ **where**
 $\langle \text{iterate-add } n \text{ } a = (((+) \text{ } a) \text{ } \sim n) \text{ } 0 \rangle$

lemma *iterate-add-simps*[simp]:

$\langle \text{iterate-add } 0 \ a = 0 \rangle$
 $\langle \text{iterate-add } (\text{Suc } n) \ a = a + \text{iterate-add } n \ a \rangle$
 $\langle \text{proof} \rangle$

lemma *iterate-add-empty*[simp]: $\langle \text{iterate-add } n \ 0 = 0 \rangle$

$\langle \text{proof} \rangle$

lemma *iterate-add-distrib*[simp]: $\langle \text{iterate-add } (m+n) \ a = \text{iterate-add } m \ a + \text{iterate-add } n \ a \rangle$

$\langle \text{proof} \rangle$

lemma *iterate-add-Numeral1*: $\langle \text{iterate-add } n \ \text{Numeral1} = \text{of-nat } n \rangle$

$\langle \text{proof} \rangle$

lemma *iterate-add-1*: $\langle \text{iterate-add } n \ 1 = \text{of-nat } n \rangle$

$\langle \text{proof} \rangle$

lemma *iterate-add-eq-add-iff1*:

$\langle i \leq j \implies (\text{iterate-add } j \ u + m = \text{iterate-add } i \ u + n) = (\text{iterate-add } (j - i) \ u + m = n) \rangle$
 $\langle \text{proof} \rangle$

lemma *iterate-add-eq-add-iff2*:

$\langle i \leq j \implies (\text{iterate-add } i \ u + m = \text{iterate-add } j \ u + n) = (m = \text{iterate-add } (j - i) \ u + n) \rangle$
 $\langle \text{proof} \rangle$

lemma *iterate-add-less-iff1*:

$j \leq (i::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add}, \text{ordered-ab-semigroup-add-imp-le}\}) + m < \text{iterate-add } j \ u + n) = (\text{iterate-add } (i-j) \ u + m < n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-less-iff2*:

$i \leq (j::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add}, \text{ordered-ab-semigroup-add-imp-le}\}) + m < \text{iterate-add } j \ u + n) = (m < \text{iterate-add } (j - i) \ u + n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-less-eq-iff1*:

$j \leq (i::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add}, \text{ordered-ab-semigroup-add-imp-le}\}) + m \leq \text{iterate-add } j \ u + n) = (\text{iterate-add } (i-j) \ u + m \leq n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-less-eq-iff2*:

$i \leq (j::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add}, \text{ordered-ab-semigroup-add-imp-le}\}) + m \leq \text{iterate-add } j \ u + n) = (m \leq \text{iterate-add } (j - i) \ u + n)$
 $\langle \text{proof} \rangle$

lemma *iterate-add-add-eq1*:

$j \leq (i::nat) \implies ((iterate-add\ i\ u + m) - (iterate-add\ j\ u + n)) = ((iterate-add\ (i-j)\ u + m) - n)$
 ⟨proof⟩

lemma *iterate-add-diff-add-eq2*:

$i \leq (j::nat) \implies ((iterate-add\ i\ u + m) - (iterate-add\ j\ u + n)) = (m - (iterate-add\ (j-i)\ u + n))$
 ⟨proof⟩

Simproc Set-Up

⟨ML⟩

end

67 (Finite) Multisets

theory *Multiset*

imports *Cancellation*

begin

67.1 The type of multisets

typedef *'a multiset* = $\langle \{f :: 'a \Rightarrow nat.\ finite\ \{x.\ f\ x > 0\}\} \rangle$
morphisms *count Abs-multiset*
 ⟨proof⟩

setup-lifting *type-definition-multiset*

lemma *count-Abs-multiset*:

$\langle count\ (Abs-multiset\ f) = f \rangle$ **if** $\langle finite\ \{x.\ f\ x > 0\} \rangle$
 ⟨proof⟩

lemma *multiset-eq-iff*: $M = N \longleftrightarrow (\forall a.\ count\ M\ a = count\ N\ a)$
 ⟨proof⟩

lemma *multiset-eqI*: $(\bigwedge x.\ count\ A\ x = count\ B\ x) \implies A = B$
 ⟨proof⟩

Preservation of the representing set *multiset*.

lemma *diff-preserves-multiset*:

$\langle finite\ \{x.\ 0 < M\ x - N\ x\} \rangle$ **if** $\langle finite\ \{x.\ 0 < M\ x\} \rangle$ **for** $M\ N :: 'a \Rightarrow nat$
 ⟨proof⟩

lemma *filter-preserves-multiset*:

$\langle finite\ \{x.\ 0 < (if\ P\ x\ then\ M\ x\ else\ 0)\} \rangle$ **if** $\langle finite\ \{x.\ 0 < M\ x\} \rangle$ **for** $M\ N :: 'a \Rightarrow nat$
 ⟨proof⟩

lemmas *in-multiset = diff-preserves-multiset filter-preserves-multiset*

67.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (type) cancel-comm-monoid-add
begin

lift-definition *zero-multiset* :: $\langle 'a \text{ multiset} \rangle$
 is $\langle \lambda a. 0 \rangle$
 $\langle \text{proof} \rangle$

abbreviation *empty-mset* :: $\langle 'a \text{ multiset} \rangle$ ($\langle \{\#\} \rangle$)
 where $\langle \text{empty-mset} \equiv 0 \rangle$

lift-definition *plus-multiset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$
 is $\langle \lambda M N a. M a + N a \rangle$
 $\langle \text{proof} \rangle$

lift-definition *minus-multiset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$
 is $\langle \lambda M N a. M a - N a \rangle$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

context
begin

qualified definition *is-empty* :: $'a \text{ multiset} \Rightarrow \text{bool}$ **where**
 $[\text{code-abbrev}]: \text{is-empty } A \longleftrightarrow A = \{\#\}$

end

lemma *add-mset-in-multiset*:
 $\langle \text{finite } \{x. 0 < (\text{if } x = a \text{ then } \text{Suc } (M x) \text{ else } M x)\} \rangle$
 if $\langle \text{finite } \{x. 0 < M x\} \rangle$
 $\langle \text{proof} \rangle$

lift-definition *add-mset* :: $'a \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$ **is**
 $\lambda a M b. \text{if } b = a \text{ then } \text{Suc } (M b) \text{ else } M b$
 $\langle \text{proof} \rangle$

syntax
 -multiset :: $\text{args} \Rightarrow 'a \text{ multiset}$ ($\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{mixfix multiset enumeration} \rangle \{\#\text{-}\#\} \rangle) \rangle$)
syntax-consts
 -multiset $\equiv \text{add-mset}$
translations

$\{\#x, xs\# \} == \text{CONST add-mset } x \ \{\#xs\# \}$
 $\{\#x\# \} == \text{CONST add-mset } x \ \{\# \}$

lemma *count-empty* [simp]: *count* $\{\# \}$ *a* = 0
 ⟨proof⟩

lemma *count-add-mset* [simp]:
count (*add-mset* *b* *A*) *a* = (if *b* = *a* then *Suc* (*count* *A* *a*) else *count* *A* *a*)
 ⟨proof⟩

lemma *count-single*: *count* $\{\#b\# \}$ *a* = (if *b* = *a* then 1 else 0)
 ⟨proof⟩

lemma
add-mset-not-empty [simp]: $\langle \text{add-mset } a \ A \neq \{\# \} \rangle$ and
empty-not-add-mset [simp]: $\{\# \} \neq \text{add-mset } a \ A$
 ⟨proof⟩

lemma *add-mset-add-mset-same-iff* [simp]:
add-mset *a* *A* = *add-mset* *a* *B* \longleftrightarrow *A* = *B*
 ⟨proof⟩

lemma *add-mset-commute*:
add-mset *x* (*add-mset* *y* *M*) = *add-mset* *y* (*add-mset* *x* *M*)
 ⟨proof⟩

67.3 Basic operations

67.3.1 Conversion to set and membership

definition *set-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ set} \rangle$
 where $\langle \text{set-mset } M = \{x. \text{count } M \ x > 0\} \rangle$

abbreviation *member-mset* :: $\langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$
 where $\langle \text{member-mset } a \ M \equiv a \in \text{set-mset } M \rangle$

notation
member-mset $(\langle '(\in \#') \rangle)$ and
member-mset $(\langle (\langle \text{notation} = \langle \text{infix } \in \# \rangle \rangle - / \in \# -) \rangle [50, 51] \ 50)$

notation (ASCII)
member-mset $(\langle '(:\#') \rangle)$ and
member-mset $(\langle (\langle \text{notation} = \langle \text{infix } :\# \rangle \rangle - / :\# -) \rangle [50, 51] \ 50)$

abbreviation *not-member-mset* :: $\langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$
 where $\langle \text{not-member-mset } a \ M \equiv a \notin \text{set-mset } M \rangle$

notation
not-member-mset $(\langle '(\notin \#') \rangle)$ and
not-member-mset $(\langle (\langle \text{notation} = \langle \text{infix } \notin \# \rangle \rangle - / \notin \# -) \rangle [50, 51] \ 50)$

notation (ASCII)

not-member-mset ($\langle \langle \sim : \# \rangle \rangle$) **and**

not-member-mset ($\langle \langle \text{notation} = \langle \text{infix } \sim : \# \rangle \rangle - / \sim : \# - \rangle [50, 51] 50$)

context

begin

qualified abbreviation $\text{Ball} :: 'a \text{ multiset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where $\text{Ball } M \equiv \text{Set.Ball } (\text{set-mset } M)$

qualified abbreviation $\text{Bex} :: 'a \text{ multiset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where $\text{Bex } M \equiv \text{Set.Bex } (\text{set-mset } M)$

end

syntax

-MBall $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool}$

($\langle \langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \forall \rangle \rangle \forall - \in \# - / - \rangle [0, 0, 10] 10$)

-MBex $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool}$

($\langle \langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \exists \rangle \rangle \exists - \in \# - / - \rangle [0, 0, 10] 10$)

syntax (ASCII)

-MBall $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool}$

($\langle \langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \forall \rangle \rangle \forall - : \# - / - \rangle [0, 0, 10] 10$)

-MBex $:: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool}$

($\langle \langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \exists \rangle \rangle \exists - : \# - / - \rangle [0, 0, 10] 10$)

syntax-consts

-MBall $\equiv \text{Multiset.Ball}$ **and**

-MBex $\equiv \text{Multiset.Bex}$

translations

$\forall x \in \# A. P \equiv \text{CONST Multiset.Ball } A (\lambda x. P)$

$\exists x \in \# A. P \equiv \text{CONST Multiset.Bex } A (\lambda x. P)$

$\langle \text{ML} \rangle$

lemma *count-eq-zero-iff*:

$\text{count } M x = 0 \longleftrightarrow x \notin \# M$

$\langle \text{proof} \rangle$

lemma *not-in-iff*:

$x \notin \# M \longleftrightarrow \text{count } M x = 0$

$\langle \text{proof} \rangle$

lemma *count-greater-zero-iff* [simp]:

$\text{count } M x > 0 \longleftrightarrow x \in \# M$

$\langle \text{proof} \rangle$

lemma *count-inI*:

assumes $\text{count } M x = 0 \implies \text{False}$

shows $x \in \# M$
 $\langle \text{proof} \rangle$

lemma *in-countE*:
assumes $x \in \# M$
obtains n **where** $\text{count } M \ x = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *count-greater-eq-Suc-zero-iff* [simp]:
 $\text{count } M \ x \geq \text{Suc } 0 \longleftrightarrow x \in \# M$
 $\langle \text{proof} \rangle$

lemma *count-greater-eq-one-iff* [simp]:
 $\text{count } M \ x \geq 1 \longleftrightarrow x \in \# M$
 $\langle \text{proof} \rangle$

lemma *set-mset-empty* [simp]:
 $\text{set-mset } \{\#\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *set-mset-single*:
 $\text{set-mset } \{\#b\# \} = \{b\}$
 $\langle \text{proof} \rangle$

lemma *set-mset-eq-empty-iff* [simp]:
 $\text{set-mset } M = \{\} \longleftrightarrow M = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *finite-set-mset* [iff]:
 $\text{finite } (\text{set-mset } M)$
 $\langle \text{proof} \rangle$

lemma *set-mset-add-mset-insert* [simp]: $\langle \text{set-mset } (\text{add-mset } a \ A) = \text{insert } a \ (\text{set-mset } A) \rangle$
 $\langle \text{proof} \rangle$

lemma *multiset-nonemptyE* [elim]:
assumes $A \neq \{\#\}$
obtains x **where** $x \in \# A$
 $\langle \text{proof} \rangle$

lemma *count-gt-imp-in-mset*: $\text{count } M \ x > n \implies x \in \# M$
 $\langle \text{proof} \rangle$

67.3.2 Union

lemma *count-union* [simp]:
 $\text{count } (M + N) \ a = \text{count } M \ a + \text{count } N \ a$
 $\langle \text{proof} \rangle$

lemma *set-mset-union* [simp]:
 $set\text{-}mset\ (M + N) = set\text{-}mset\ M \cup set\text{-}mset\ N$
 ⟨proof⟩

lemma *union-mset-add-mset-left* [simp]:
 $add\text{-}mset\ a\ A + B = add\text{-}mset\ a\ (A + B)$
 ⟨proof⟩

lemma *union-mset-add-mset-right* [simp]:
 $A + add\text{-}mset\ a\ B = add\text{-}mset\ a\ (A + B)$
 ⟨proof⟩

lemma *add-mset-add-single*: $\langle add\text{-}mset\ a\ A = A + \{\#a\# \} \rangle$
 ⟨proof⟩

67.3.3 Difference

instance *multiset* :: (type) *comm-monoid-diff*
 ⟨proof⟩

lemma *count-diff* [simp]:
 $count\ (M - N)\ a = count\ M\ a - count\ N\ a$
 ⟨proof⟩

lemma *add-mset-diff-bothsides*:
 $\langle add\text{-}mset\ a\ M - add\text{-}mset\ a\ A = M - A \rangle$
 ⟨proof⟩

lemma *in-diff-count*:
 $a \in\# M - N \longleftrightarrow count\ N\ a < count\ M\ a$
 ⟨proof⟩

lemma *count-in-diffI*:
assumes $\bigwedge n. count\ N\ x = n + count\ M\ x \implies False$
shows $x \in\# M - N$
 ⟨proof⟩

lemma *in-diff-countE*:
assumes $x \in\# M - N$
obtains n **where** $count\ M\ x = Suc\ n + count\ N\ x$
 ⟨proof⟩

lemma *in-diffD*:
assumes $a \in\# M - N$
shows $a \in\# M$
 ⟨proof⟩

lemma *set-mset-diff*:

set-mset ($M - N$) = $\{a. \text{count } N \ a < \text{count } M \ a\}$

<proof>

lemma *diff-empty [simp]*: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$

<proof>

lemma *diff-cancel*: $A - A = \{\#\}$

<proof>

lemma *diff-union-cancelR*: $M + N - N = (M :: 'a \text{ multiset})$

<proof>

lemma *diff-union-cancelL*: $N + M - N = (M :: 'a \text{ multiset})$

<proof>

lemma *diff-right-commute*:

fixes $M \ N \ Q :: 'a \text{ multiset}$

shows $M - N - Q = M - Q - N$

<proof>

lemma *diff-add*:

fixes $M \ N \ Q :: 'a \text{ multiset}$

shows $M - (N + Q) = M - N - Q$

<proof>

lemma *insert-DiffM [simp]*: $x \in \# \ M \implies \text{add-mset } x \ (M - \{\#x\}) = M$

<proof>

lemma *insert-DiffM2*: $x \in \# \ M \implies (M - \{\#x\}) + \{\#x\} = M$

<proof>

lemma *diff-union-swap*: $a \neq b \implies \text{add-mset } b \ (M - \{\#a\}) = \text{add-mset } b \ M - \{\#a\}$

<proof>

lemma *diff-add-mset-swap [simp]*: $b \notin \# \ A \implies \text{add-mset } b \ M - A = \text{add-mset } b \ (M - A)$

<proof>

lemma *diff-union-swap2 [simp]*: $y \in \# \ M \implies \text{add-mset } x \ M - \{\#y\} = \text{add-mset } x \ (M - \{\#y\})$

<proof>

lemma *diff-diff-add-mset [simp]*: $(M :: 'a \text{ multiset}) - N - P = M - (N + P)$

<proof>

lemma *diff-union-single-conv*:

$a \in \# \ J \implies I + J - \{\#a\} = I + (J - \{\#a\})$

$\langle \text{proof} \rangle$

lemma *mset-add* [*elim?*]:

assumes $a \in\# A$

obtains B where $A = \text{add-mset } a B$

$\langle \text{proof} \rangle$

lemma *union-iff*:

$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$

$\langle \text{proof} \rangle$

lemma *count-minus-inter-lt-count-minus-inter-iff*:

$\text{count } (M2 - M1) y < \text{count } (M1 - M2) y \longleftrightarrow y \in\# M1 - M2$

$\langle \text{proof} \rangle$

lemma *minus-inter-eq-minus-inter-iff*:

$(M1 - M2) = (M2 - M1) \longleftrightarrow \text{set-mset } (M1 - M2) = \text{set-mset } (M2 - M1)$

$\langle \text{proof} \rangle$

67.3.4 Min and Max

abbreviation *Min-mset* :: $'a::\text{linorder multiset} \Rightarrow 'a$ where

$\text{Min-mset } m \equiv \text{Min } (\text{set-mset } m)$

abbreviation *Max-mset* :: $'a::\text{linorder multiset} \Rightarrow 'a$ where

$\text{Max-mset } m \equiv \text{Max } (\text{set-mset } m)$

lemma

Min-in-mset: $M \neq \{\#\} \Longrightarrow \text{Min-mset } M \in\# M$ and

Max-in-mset: $M \neq \{\#\} \Longrightarrow \text{Max-mset } M \in\# M$

$\langle \text{proof} \rangle$

67.3.5 Equality of multisets

lemma *single-eq-single* [*simp*]: $\{\#a\# \} = \{\#b\# \} \longleftrightarrow a = b$

$\langle \text{proof} \rangle$

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$

$\langle \text{proof} \rangle$

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$

$\langle \text{proof} \rangle$

lemma *multi-self-add-other-not-self* [*simp*]: $M = \text{add-mset } x M \longleftrightarrow \text{False}$

$\langle \text{proof} \rangle$

lemma *add-mset-remove-trivial* [*simp*]: $\langle \text{add-mset } x M - \{\#x\# \} = M \rangle$

$\langle \text{proof} \rangle$

lemma *diff-single-trivial*: $\neg x \in\# M \Longrightarrow M - \{\#x\# \} = M$

$\langle proof \rangle$

lemma *diff-single-eq-union*: $x \in\# M \implies M - \{\#x\# \} = N \longleftrightarrow M = \text{add-mset } x N$
 $\langle proof \rangle$

lemma *union-single-eq-diff*: $\text{add-mset } x M = N \implies M = N - \{\#x\# \}$
 $\langle proof \rangle$

lemma *union-single-eq-member*: $\text{add-mset } x M = N \implies x \in\# N$
 $\langle proof \rangle$

lemma *add-mset-remove-trivial-If*:
 $\text{add-mset } a (N - \{\#a\# \}) = (\text{if } a \in\# N \text{ then } N \text{ else } \text{add-mset } a N)$
 $\langle proof \rangle$

lemma *add-mset-remove-trivial-eq*: $\langle N = \text{add-mset } a (N - \{\#a\# \}) \longleftrightarrow a \in\# N \rangle$
 $\langle proof \rangle$

lemma *union-is-single*:
 $M + N = \{\#a\# \} \longleftrightarrow M = \{\#a\# \} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\# \}$
 $(\text{is } ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *single-is-union*: $\{\#a\# \} = M + N \longleftrightarrow \{\#a\# \} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\# \} = N$
 $\langle proof \rangle$

lemma *add-eq-conv-diff*:
 $\text{add-mset } a M = \text{add-mset } b N \longleftrightarrow M = N \wedge a = b \vee M = \text{add-mset } b (N - \{\#a\# \}) \wedge N = \text{add-mset } a (M - \{\#b\# \})$
 $(\text{is } ?lhs \longleftrightarrow ?rhs)$

$\langle proof \rangle$

lemma *add-mset-eq-single [iff]*: $\text{add-mset } b M = \{\#a\# \} \longleftrightarrow b = a \wedge M = \{\#\}$
 $\langle proof \rangle$

lemma *single-eq-add-mset [iff]*: $\{\#a\# \} = \text{add-mset } b M \longleftrightarrow b = a \wedge M = \{\#\}$
 $\langle proof \rangle$

lemma *insert-noteq-member*:
assumes *BC*: $\text{add-mset } b B = \text{add-mset } c C$
and *bnotc*: $b \neq c$
shows $c \in\# B$
 $\langle proof \rangle$

lemma *add-eq-conv-ex*:

$(\text{add-mset } a \ M = \text{add-mset } b \ N) =$
 $(M = N \wedge a = b \vee (\exists K. M = \text{add-mset } b \ K \wedge N = \text{add-mset } a \ K))$
 $\langle \text{proof} \rangle$

lemma *multi-member-split*: $x \in\# \ M \implies \exists A. M = \text{add-mset } x \ A$
 $\langle \text{proof} \rangle$

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# \ B$
and $b \neq c$
shows $\text{add-mset } b \ (B - \{\#c\}) = \text{add-mset } b \ B - \{\#c\}$
 $\langle \text{proof} \rangle$

lemma *add-mset-eq-singleton-iff*[*iff*]:
 $\text{add-mset } x \ M = \{\#y\} \longleftrightarrow M = \{\#\} \wedge x = y$
 $\langle \text{proof} \rangle$

67.3.6 Pointwise ordering induced by count

definition *subseteq-mset* :: $'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \Rightarrow \text{bool}$ (**infix** $\langle \subseteq\# \rangle$ 50)
where $A \subseteq\# \ B \longleftrightarrow (\forall a. \text{count } A \ a \leq \text{count } B \ a)$

definition *subset-mset* :: $'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \Rightarrow \text{bool}$ (**infix** $\langle \subset\# \rangle$ 50)
where $A \subset\# \ B \longleftrightarrow A \subseteq\# \ B \wedge A \neq B$

abbreviation (*input*) *supseteq-mset* :: $'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \Rightarrow \text{bool}$ (**infix** $\langle \supseteq\# \rangle$ 50)
where $\text{supseteq-mset } A \ B \equiv B \subseteq\# \ A$

abbreviation (*input*) *supset-mset* :: $'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \Rightarrow \text{bool}$ (**infix** $\langle \supset\# \rangle$ 50)
where $\text{supset-mset } A \ B \equiv B \subset\# \ A$

notation (*input*)
 subseq-mset (**infix** $\langle \leq\# \rangle$ 50) **and**
 supseq-mset (**infix** $\langle \geq\# \rangle$ 50)

notation (*ASCII*)
 subseq-mset (**infix** $\langle \leq\# \rangle$ 50) **and**
 subset-mset (**infix** $\langle <\# \rangle$ 50) **and**
 supseq-mset (**infix** $\langle \geq\# \rangle$ 50) **and**
 supset-mset (**infix** $\langle >\# \rangle$ 50)

global-interpretation *subset-mset*: *ordering* $\langle (\subseteq\#) \rangle \langle (\subset\#) \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le* $\langle (+) \rangle \langle (-) \rangle \langle (\subseteq\#) \rangle$
 $\langle (\subset\#) \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset*: *ordered-ab-semigroup-monoid-add-imp-le* (+) 0 (−)
 $(\subseteq\#) (\subset\#)$
 $\langle proof \rangle$

lemma *mset-subset-eqI*:
 $(\bigwedge a. \text{count } A \ a \leq \text{count } B \ a) \implies A \subseteq\# B$
 $\langle proof \rangle$

lemma *mset-subset-eq-count*:
 $A \subseteq\# B \implies \text{count } A \ a \leq \text{count } B \ a$
 $\langle proof \rangle$

lemma *mset-subset-eq-exists-conv*: $(A::'a \text{ multiset}) \subseteq\# B \longleftrightarrow (\exists C. B = A + C)$
 $\langle proof \rangle$

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff* (+) 0 $(\subseteq\#) (\subset\#)$
 $(-)$
 $\langle proof \rangle$

declare *subset-mset.add-diff-assoc[simp]* *subset-mset.add-diff-assoc2[simp]*

lemma *mset-subset-eq-mono-add-right-cancel*: $(A::'a \text{ multiset}) + C \subseteq\# B + C$
 $\longleftrightarrow A \subseteq\# B$
 $\langle proof \rangle$

lemma *mset-subset-eq-mono-add-left-cancel*: $C + (A::'a \text{ multiset}) \subseteq\# C + B \longleftrightarrow$
 $A \subseteq\# B$
 $\langle proof \rangle$

lemma *mset-subset-eq-mono-add*: $(A::'a \text{ multiset}) \subseteq\# B \implies C \subseteq\# D \implies A +$
 $C \subseteq\# B + D$
 $\langle proof \rangle$

lemma *mset-subset-eq-add-left*: $(A::'a \text{ multiset}) \subseteq\# A + B$
 $\langle proof \rangle$

lemma *mset-subset-eq-add-right*: $B \subseteq\# (A::'a \text{ multiset}) + B$
 $\langle proof \rangle$

lemma *single-subset-iff [simp]*:
 $\{\#a\# \} \subseteq\# M \longleftrightarrow a \in\# M$
 $\langle proof \rangle$

lemma *mset-subset-eq-single*: $a \in\# B \implies \{\#a\# \} \subseteq\# B$
 $\langle proof \rangle$

lemma *mset-subset-eq-add-mset-cancel*: $\langle \text{add-mset } a \ A \subseteq\# \text{ add-mset } a \ B \longleftrightarrow A$
 $\subseteq\# B \rangle$

$\langle \text{proof} \rangle$

lemma *multiset-diff-union-assoc*:

fixes $A\ B\ C\ D :: 'a\ \text{multiset}$

shows $C \subseteq\# B \implies A + B - C = A + (B - C)$

$\langle \text{proof} \rangle$

lemma *mset-subset-eq-multiset-union-diff-commute*:

fixes $A\ B\ C\ D :: 'a\ \text{multiset}$

shows $B \subseteq\# A \implies A - B + C = A + C - B$

$\langle \text{proof} \rangle$

lemma *diff-subset-eq-self[simp]*:

$(M :: 'a\ \text{multiset}) - N \subseteq\# M$

$\langle \text{proof} \rangle$

lemma *mset-subset-eqD*:

assumes $A \subseteq\# B$ **and** $x \in\# A$

shows $x \in\# B$

$\langle \text{proof} \rangle$

lemma *mset-subsetD*:

$A \subset\# B \implies x \in\# A \implies x \in\# B$

$\langle \text{proof} \rangle$

lemma *set-mset-mono*:

$A \subseteq\# B \implies \text{set-mset } A \subseteq \text{set-mset } B$

$\langle \text{proof} \rangle$

lemma *mset-subset-eq-insertD*:

assumes $\text{add-mset } x\ A \subseteq\# B$

shows $x \in\# B \wedge A \subset\# B$

$\langle \text{proof} \rangle$

lemma *mset-subset-insertD*:

$\text{add-mset } x\ A \subset\# B \implies x \in\# B \wedge A \subset\# B$

$\langle \text{proof} \rangle$

lemma *mset-subset-of-empty[simp]*: $A \subset\# \{\#\} \longleftrightarrow \text{False}$

$\langle \text{proof} \rangle$

lemma *empty-subset-add-mset[simp]*: $\{\#\} \subset\# \text{add-mset } x\ M$

$\langle \text{proof} \rangle$

lemma *empty-le*: $\{\#\} \subseteq\# A$

$\langle \text{proof} \rangle$

lemma *insert-subset-eq-iff*:

$\text{add-mset } a\ A \subseteq\# B \longleftrightarrow a \in\# B \wedge A \subseteq\# B - \{\#a\# \}$

$\langle \text{proof} \rangle$

lemma *insert-union-subset-iff*:

$$\text{add-mset } a \ A \subset\# B \longleftrightarrow a \in\# B \wedge A \subset\# B - \{\#a\# \}$$

$\langle \text{proof} \rangle$

lemma *subset-eq-diff-conv*:

$$A - C \subseteq\# B \longleftrightarrow A \subseteq\# B + C$$

$\langle \text{proof} \rangle$

lemma *multi-psub-of-add-self* [simp]: $A \subset\# \text{add-mset } x \ A$

$\langle \text{proof} \rangle$

lemma *multi-psub-self*: $A \subset\# A = \text{False}$

$\langle \text{proof} \rangle$

lemma *mset-subset-add-mset* [simp]: $\text{add-mset } x \ N \subset\# \text{add-mset } x \ M \longleftrightarrow N \subset\# M$

$\langle \text{proof} \rangle$

lemma *mset-subset-diff-self*: $c \in\# B \implies B - \{\#c\# \} \subset\# B$

$\langle \text{proof} \rangle$

lemma *Diff-eq-empty-iff-mset*: $A - B = \{\#\} \longleftrightarrow A \subseteq\# B$

$\langle \text{proof} \rangle$

lemma *add-mset-subseteq-single-iff*[iff]: $\text{add-mset } a \ M \subseteq\# \{\#b\# \} \longleftrightarrow M = \{\#\} \wedge a = b$

$\langle \text{proof} \rangle$

lemma *nonempty-subseteq-mset-eq-single*: $M \neq \{\#\} \implies M \subseteq\# \{\#x\# \} \implies M = \{\#x\# \}$

$\langle \text{proof} \rangle$

lemma *nonempty-subseteq-mset-iff-single*: $(M \neq \{\#\} \wedge M \subseteq\# \{\#x\# \} \wedge P) \longleftrightarrow M = \{\#x\# \} \wedge P$

$\langle \text{proof} \rangle$

67.3.7 Intersection and bounded union

definition *inter-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ (**infixl** $\langle \cap\# \rangle$ 70)

where $\langle A \cap\# B = A - (A - B) \rangle$

lemma *count-inter-mset* [simp]:

$$\langle \text{count } (A \cap\# B) \ x = \min (\text{count } A \ x) (\text{count } B \ x) \rangle$$

$\langle \text{proof} \rangle$

interpretation *subset-mset: semilattice-inf* $\langle (\cap \#) \rangle \langle (\subseteq \#) \rangle \langle (\subset \#) \rangle$
 $\langle \text{proof} \rangle$

definition *union-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ (**infixl** $\cup \#$)
 70)
where $\langle A \cup \# B = A + (B - A) \rangle$

lemma *count-union-mset* [*simp*]:
 $\langle \text{count } (A \cup \# B) \ x = \max (\text{count } A \ x) (\text{count } B \ x) \rangle$
 $\langle \text{proof} \rangle$

global-interpretation *subset-mset: semilattice-neutr-order* $\langle (\cup \#) \rangle \langle \{ \# \} \rangle \langle (\sup \#) \rangle$
 $\langle (\sup \#) \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset: semilattice-sup* $\langle (\cup \#) \rangle \langle (\subseteq \#) \rangle \langle (\subset \#) \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset: bounded-lattice-bot* $(\cap \#) (\subseteq \#) (\subset \#)$
 $(\cup \#) \{ \# \}$
 $\langle \text{proof} \rangle$

67.3.8 Additional intersection facts

lemma *set-mset-inter* [*simp*]:
 $\text{set-mset } (A \cap \# B) = \text{set-mset } A \cap \text{set-mset } B$
 $\langle \text{proof} \rangle$

lemma *diff-intersect-left-idem* [*simp*]:
 $M - M \cap \# N = M - N$
 $\langle \text{proof} \rangle$

lemma *diff-intersect-right-idem* [*simp*]:
 $M - N \cap \# M = M - N$
 $\langle \text{proof} \rangle$

lemma *multiset-inter-single* [*simp*]: $a \neq b \implies \{ \# a \# \} \cap \# \{ \# b \# \} = \{ \# \}$
 $\langle \text{proof} \rangle$

lemma *multiset-union-diff-commute*:
assumes $B \cap \# C = \{ \# \}$
shows $A + B - C = A - C + B$
 $\langle \text{proof} \rangle$

lemma *disjunct-not-in*:
 $A \cap \# B = \{ \# \} \longleftrightarrow (\forall a. a \notin \# A \vee a \notin \# B)$
 $\langle \text{proof} \rangle$

lemma *inter-mset-empty-distrib-right*: $A \cap\# (B + C) = \{\#\} \longleftrightarrow A \cap\# B = \{\#\} \wedge A \cap\# C = \{\#\}$
 ⟨proof⟩

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$
 ⟨proof⟩

lemma *add-mset-inter-add-mset* [simp]:
 $\text{add-mset } a \ A \cap\# \text{ add-mset } a \ B = \text{add-mset } a \ (A \cap\# B)$
 ⟨proof⟩

lemma *add-mset-disjoint* [simp]:
 $\text{add-mset } a \ A \cap\# B = \{\#\} \longleftrightarrow a \notin\# B \wedge A \cap\# B = \{\#\}$
 $\{\#\} = \text{add-mset } a \ A \cap\# B \longleftrightarrow a \notin\# B \wedge \{\#\} = A \cap\# B$
 ⟨proof⟩

lemma *disjoint-add-mset* [simp]:
 $B \cap\# \text{ add-mset } a \ A = \{\#\} \longleftrightarrow a \notin\# B \wedge B \cap\# A = \{\#\}$
 $\{\#\} = A \cap\# \text{ add-mset } b \ B \longleftrightarrow b \notin\# A \wedge \{\#\} = A \cap\# B$
 ⟨proof⟩

lemma *inter-add-left1*: $\neg x \in\# N \implies (\text{add-mset } x \ M) \cap\# N = M \cap\# N$
 ⟨proof⟩

lemma *inter-add-left2*: $x \in\# N \implies (\text{add-mset } x \ M) \cap\# N = \text{add-mset } x \ (M \cap\# (N - \{\#x\#}))$
 ⟨proof⟩

lemma *inter-add-right1*: $\neg x \in\# N \implies N \cap\# (\text{add-mset } x \ M) = N \cap\# M$
 ⟨proof⟩

lemma *inter-add-right2*: $x \in\# N \implies N \cap\# (\text{add-mset } x \ M) = \text{add-mset } x \ ((N - \{\#x\#}) \cap\# M)$
 ⟨proof⟩

lemma *disjunct-set-mset-diff*:
 assumes $M \cap\# N = \{\#\}$
 shows $\text{set-mset } (M - N) = \text{set-mset } M$
 ⟨proof⟩

lemma *at-most-one-mset-mset-diff*:
 assumes $a \notin\# M - \{\#a\#$
 shows $\text{set-mset } (M - \{\#a\#}) = \text{set-mset } M - \{a\}$
 ⟨proof⟩

lemma *more-than-one-mset-mset-diff*:
 assumes $a \in\# M - \{\#a\#$
 shows $\text{set-mset } (M - \{\#a\#}) = \text{set-mset } M$

$\langle proof \rangle$

lemma *inter-iff*:

$$a \in\# A \cap\# B \longleftrightarrow a \in\# A \wedge a \in\# B$$

$\langle proof \rangle$

lemma *inter-union-distrib-left*:

$$A \cap\# B + C = (A + C) \cap\# (B + C)$$

$\langle proof \rangle$

lemma *inter-union-distrib-right*:

$$C + A \cap\# B = (C + A) \cap\# (C + B)$$

$\langle proof \rangle$

lemma *inter-subset-eq-union*:

$$A \cap\# B \subseteq\# A + B$$

$\langle proof \rangle$

67.3.9 Additional bounded union facts

lemma *set-mset-sup* [simp]:

$$\langle set-mset (A \cup\# B) = set-mset A \cup set-mset B \rangle$$

$\langle proof \rangle$

lemma *sup-union-left1* [simp]: $\neg x \in\# N \implies (add-mset x M) \cup\# N = add-mset x (M \cup\# N)$

$\langle proof \rangle$

lemma *sup-union-left2*: $x \in\# N \implies (add-mset x M) \cup\# N = add-mset x (M \cup\# (N - \{\#x\}))$

$\langle proof \rangle$

lemma *sup-union-right1* [simp]: $\neg x \in\# N \implies N \cup\# (add-mset x M) = add-mset x (N \cup\# M)$

$\langle proof \rangle$

lemma *sup-union-right2*: $x \in\# N \implies N \cup\# (add-mset x M) = add-mset x ((N - \{\#x\}) \cup\# M)$

$\langle proof \rangle$

lemma *sup-union-distrib-left*:

$$A \cup\# B + C = (A + C) \cup\# (B + C)$$

$\langle proof \rangle$

lemma *union-sup-distrib-right*:

$$C + A \cup\# B = (C + A) \cup\# (C + B)$$

$\langle proof \rangle$

lemma *union-diff-inter-eq-sup*:

$$A + B - A \cap \# B = A \cup \# B$$

<proof>

lemma *union-diff-sup-eq-inter*:
 $A + B - A \cup \# B = A \cap \# B$
<proof>

lemma *add-mset-union*:
 $\langle \text{add-mset } a \ A \cup \# \text{ add-mset } a \ B = \text{add-mset } a \ (A \cup \# B) \rangle$
<proof>

67.4 Replicate and repeat operations

definition *replicate-mset* :: $\text{nat} \Rightarrow 'a \Rightarrow 'a \text{ multiset}$ **where**
 $\text{replicate-mset } n \ x = (\text{add-mset } x \ \frown \ n) \ \{\#\}$

lemma *replicate-mset-0[simp]*: $\text{replicate-mset } 0 \ x = \{\#\}$
<proof>

lemma *replicate-mset-Suc [simp]*: $\text{replicate-mset } (\text{Suc } n) \ x = \text{add-mset } x \ (\text{replicate-mset } n \ x)$
<proof>

lemma *count-replicate-mset[simp]*: $\text{count } (\text{replicate-mset } n \ x) \ y = (\text{if } y = x \text{ then } n \text{ else } 0)$
<proof>

lift-definition *repeat-mset* :: $\langle \text{nat} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$
is $\langle \lambda n \ M \ a. \ n * M \ a \rangle$ *<proof>*

lemma *count-repeat-mset [simp]*: $\text{count } (\text{repeat-mset } i \ A) \ a = i * \text{count } A \ a$
<proof>

lemma *repeat-mset-0 [simp]*:
 $\langle \text{repeat-mset } 0 \ M = \{\#\} \rangle$
<proof>

lemma *repeat-mset-Suc [simp]*:
 $\langle \text{repeat-mset } (\text{Suc } n) \ M = M + \text{repeat-mset } n \ M \rangle$
<proof>

lemma *repeat-mset-right [simp]*: $\text{repeat-mset } a \ (\text{repeat-mset } b \ A) = \text{repeat-mset } (a * b) \ A$
<proof>

lemma *left-diff-repeat-mset-distrib'*: $\langle \text{repeat-mset } (i - j) \ u = \text{repeat-mset } i \ u - \text{repeat-mset } j \ u \rangle$
<proof>

lemma *left-add-mult-distrib-mset*:

$$\text{repeat-mset } i \ u + (\text{repeat-mset } j \ u + k) = \text{repeat-mset } (i+j) \ u + k$$

<proof>

lemma *repeat-mset-distrib*:

$$\text{repeat-mset } (m + n) \ A = \text{repeat-mset } m \ A + \text{repeat-mset } n \ A$$

<proof>

lemma *repeat-mset-distrib2[simp]*:

$$\text{repeat-mset } n \ (A + B) = \text{repeat-mset } n \ A + \text{repeat-mset } n \ B$$

<proof>

lemma *repeat-mset-replicate-mset[simp]*:

$$\text{repeat-mset } n \ \{\#a\# \} = \text{replicate-mset } n \ a$$

<proof>

lemma *repeat-mset-distrib-add-mset[simp]*:

$$\text{repeat-mset } n \ (\text{add-mset } a \ A) = \text{replicate-mset } n \ a + \text{repeat-mset } n \ A$$

<proof>

lemma *repeat-mset-empty[simp]*: $\text{repeat-mset } n \ \{\# \} = \{\# \}$

<proof>

lemma *set-mset-sum*: $\text{finite } A \implies \text{set-mset } (\sum x \in A. f \ x) = (\bigcup x \in A. \text{set-mset } (f \ x))$

<proof>

67.4.1 Simprocs

lemma *repeat-mset-iterate-add*: $\langle \text{repeat-mset } n \ M = \text{iterate-add } n \ M \rangle$

<proof>

lemma *mset-subseteq-add-iff1*:

$$j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subseteq\# n)$$

<proof>

lemma *mset-subseteq-add-iff2*:

$$i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{repeat-mset } j \ u + n) = (m \subseteq\# \text{repeat-mset } (j-i) \ u + n)$$

<proof>

lemma *mset-subset-add-iff1*:

$$j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subset\# n)$$

<proof>

lemma *mset-subset-add-iff2*:

$$i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{repeat-mset } j \ u + n) = (m \subset\# \text{repeat-mset } (j-i) \ u + n)$$

repeat-mset (*j-i*) *u + n*)
 ⟨*proof*⟩

⟨*ML*⟩

lemma *add-mset-replicate-mset-safe*[*cancelation-simproc-pre*]: ⟨*NO-MATCH* {#}
 $M \implies \text{add-mset } a \ M = \{\#a\# \} + M$ ⟩
 ⟨*proof*⟩

declare *repeat-mset-iterate-add*[*cancelation-simproc-pre*]

declare *iterate-add-distrib*[*cancelation-simproc-pre*]

declare *repeat-mset-iterate-add*[*symmetric, cancelation-simproc-post*]

declare *add-mset-not-empty*[*cancelation-simproc-eq-elim*]
empty-not-add-mset[*cancelation-simproc-eq-elim*]
subset-mset.le-zero-eq[*cancelation-simproc-eq-elim*]
empty-not-add-mset[*cancelation-simproc-eq-elim*]
add-mset-not-empty[*cancelation-simproc-eq-elim*]
subset-mset.le-zero-eq[*cancelation-simproc-eq-elim*]
le-zero-eq[*cancelation-simproc-eq-elim*]

⟨*ML*⟩

67.4.2 Conditionally complete lattice

instantiation *multiset* :: (*type*) *Inf*
begin

lift-definition *Inf-multiset* :: '*a multiset* *set* \Rightarrow '*a multiset* **is**
 $\lambda A \ i. \text{ if } A = \{\} \text{ then } 0 \text{ else } \text{Inf } ((\lambda f. f \ i) \ 'A)$
 ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *Inf-multiset-empty*: *Inf* {#} = {#}
 ⟨*proof*⟩

lemma *count-Inf-multiset-nonempty*: $A \neq \{\} \implies \text{count } (\text{Inf } A) \ x = \text{Inf } ((\lambda X. \text{count } X \ x) \ 'A)$
 ⟨*proof*⟩

instantiation *multiset* :: (*type*) *Sup*
begin

definition *Sup-multiset* :: '*a multiset* *set* \Rightarrow '*a multiset* **where**

Sup-multiset $A = (\text{if } A \neq \{\} \wedge \text{subset-mset.bdd-above } A \text{ then}$
Abs-multiset $(\lambda i. \text{Sup } ((\lambda X. \text{count } X \ i) \ 'A)) \text{ else } \{\#\})$

lemma *Sup-multiset-empty*: $\text{Sup } \{\} = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *Sup-multiset-unbounded*: $\neg \text{subset-mset.bdd-above } A \implies \text{Sup } A = \{\#\}$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *bdd-above-multiset-imp-bdd-above-count*:
assumes $\text{subset-mset.bdd-above } (A :: 'a \text{ multiset set})$
shows $\text{bdd-above } ((\lambda X. \text{count } X \ x) \ 'A)$
 $\langle \text{proof} \rangle$

lemma *bdd-above-multiset-imp-finite-support*:
assumes $A \neq \{\}$ $\text{subset-mset.bdd-above } (A :: 'a \text{ multiset set})$
shows $\text{finite } (\bigcup X \in A. \{x. \text{count } X \ x > 0\})$
 $\langle \text{proof} \rangle$

lemma *Sup-multiset-in-multiset*:
 $\langle \text{finite } \{i. 0 < (\text{SUP } M \in A. \text{count } M \ i)\} \rangle$
if $\langle A \neq \{\} \rangle \langle \text{subset-mset.bdd-above } A \rangle$
 $\langle \text{proof} \rangle$

lemma *count-Sup-multiset-nonempty*:
 $\langle \text{count } (\text{Sup } A) \ x = (\text{SUP } X \in A. \text{count } X \ x) \rangle$
if $\langle A \neq \{\} \rangle \langle \text{subset-mset.bdd-above } A \rangle$
 $\langle \text{proof} \rangle$

interpretation *subset-mset: conditionally-complete-lattice* $\text{Inf Sup } (\cap\#) (\subseteq\#) (\subset\#)$
 $(\cup\#)$
 $\langle \text{proof} \rangle$

lemma *set-mset-Inf*:
assumes $A \neq \{\}$
shows $\text{set-mset } (\text{Inf } A) = (\bigcap X \in A. \text{set-mset } X)$
 $\langle \text{proof} \rangle$

lemma *in-Inf-multiset-iff*:
assumes $A \neq \{\}$
shows $x \in\# \text{Inf } A \longleftrightarrow (\forall X \in A. x \in\# X)$
 $\langle \text{proof} \rangle$

lemma *in-Inf-multisetD*: $x \in\# \text{Inf } A \implies X \in A \implies x \in\# X$
 $\langle \text{proof} \rangle$

lemma *set-mset-Sup*:

assumes *subset-mset.bdd-above A*

shows $\text{set-mset } (\text{Sup } A) = (\bigcup X \in A. \text{set-mset } X)$

<proof>

lemma *in-Sup-multiset-iff*:

assumes *subset-mset.bdd-above A*

shows $x \in \# \text{Sup } A \longleftrightarrow (\exists X \in A. x \in \# X)$

<proof>

lemma *in-Sup-multisetD*:

assumes $x \in \# \text{Sup } A$

shows $\exists X \in A. x \in \# X$

<proof>

interpretation *subset-mset*: *distrib-lattice* $(\cap \#)$ $(\subseteq \#)$ $(\subset \#)$ $(\cup \#)$

<proof>

67.4.3 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$

is $\lambda P M. \lambda x. \text{if } P x \text{ then } M x \text{ else } 0$

<proof>

syntax (*ASCII*)

-MCollect :: $\text{pttrn} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \Rightarrow 'a \text{ multiset}$

$(\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - : \# - / - \# \} \rangle)$

syntax

-MCollect :: $\text{pttrn} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \Rightarrow 'a \text{ multiset}$

$(\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - \in \# - / - \# \} \rangle)$

syntax-consts

-MCollect == *filter-mset*

translations

$\{ \# x \in \# M. P \# \} == \text{CONST filter-mset } (\lambda x. P) M$

lemma *count-filter-mset [simp]*:

$\text{count } (\text{filter-mset } P M) a = (\text{if } P a \text{ then count } M a \text{ else } 0)$

<proof>

lemma *set-mset-filter [simp]*:

$\text{set-mset } (\text{filter-mset } P M) = \{ a \in \text{set-mset } M. P a \}$

<proof>

lemma *filter-empty-mset [simp]*: $\text{filter-mset } P \{ \# \} = \{ \# \}$

<proof>

lemma *filter-single-mset*: $\text{filter-mset } P \{ \# x \# \} = (\text{if } P x \text{ then } \{ \# x \# \} \text{ else } \{ \# \})$

$\langle \text{proof} \rangle$

lemma *filter-union-mset* [simp]: $\text{filter-mset } P \ (M + N) = \text{filter-mset } P \ M + \text{filter-mset } P \ N$
 $\langle \text{proof} \rangle$

lemma *filter-diff-mset* [simp]: $\text{filter-mset } P \ (M - N) = \text{filter-mset } P \ M - \text{filter-mset } P \ N$
 $\langle \text{proof} \rangle$

lemma *filter-inter-mset* [simp]: $\text{filter-mset } P \ (M \cap\# N) = \text{filter-mset } P \ M \cap\# \text{filter-mset } P \ N$
 $\langle \text{proof} \rangle$

lemma *filter-sup-mset*[simp]: $\text{filter-mset } P \ (A \cup\# B) = \text{filter-mset } P \ A \cup\# \text{filter-mset } P \ B$
 $\langle \text{proof} \rangle$

lemma *filter-mset-add-mset* [simp]:
 $\text{filter-mset } P \ (\text{add-mset } x \ A) =$
 $(\text{if } P \ x \ \text{then } \text{add-mset } x \ (\text{filter-mset } P \ A) \ \text{else } \text{filter-mset } P \ A)$
 $\langle \text{proof} \rangle$

lemma *multiset-filter-subset*[simp]: $\text{filter-mset } f \ M \subseteq\# M$
 $\langle \text{proof} \rangle$

lemma *filter-mset-mono-strong*:
assumes $A \subseteq\# B \ \wedge x. x \in\# A \implies P \ x \implies Q \ x$
shows $\text{filter-mset } P \ A \subseteq\# \text{filter-mset } Q \ B$
 $\langle \text{proof} \rangle$

lemma *multiset-filter-mono*:
assumes $A \subseteq\# B$
shows $\text{filter-mset } f \ A \subseteq\# \text{filter-mset } f \ B$
 $\langle \text{proof} \rangle$

lemma *filter-mset-eq-conv*:
 $\text{filter-mset } P \ M = N \longleftrightarrow N \subseteq\# M \wedge (\forall b \in\# N. P \ b) \wedge (\forall a \in\# M - N. \neg P \ a)$
(is ?P \longleftrightarrow ?Q)
 $\langle \text{proof} \rangle$

lemma *filter-mset-eq-mempty-iff*[simp]: $\text{filter-mset } P \ A = \{\#\} \longleftrightarrow (\forall x. x \in\# A \longrightarrow \neg P \ x)$
 $\langle \text{proof} \rangle$

lemma *filter-filter-mset*: $\text{filter-mset } P \ (\text{filter-mset } Q \ M) = \{\#x \in\# M. Q \ x \wedge P \ x\# \}$
 $\langle \text{proof} \rangle$

lemma

filter-mset-True[simp]: $\{\#y \in \# M. \text{True}\# \} = M$ **and**
filter-mset-False[simp]: $\{\#y \in \# M. \text{False}\# \} = \{\#\}$
 ⟨proof⟩

lemma *filter-mset-cong0*:

assumes $\bigwedge x. x \in \# M \implies f\ x \longleftrightarrow g\ x$
shows *filter-mset* $f\ M = \text{filter-mset}\ g\ M$
 ⟨proof⟩

lemma *filter-mset-cong*:

assumes $M = M'$ **and** $\bigwedge x. x \in \# M' \implies f\ x \longleftrightarrow g\ x$
shows *filter-mset* $f\ M = \text{filter-mset}\ g\ M'$
 ⟨proof⟩

lemma *filter-eq-replicate-mset*: $\{\#y \in \# D. y = x\# \} = \text{replicate-mset}\ (\text{count}\ D\ x)$
 x
 ⟨proof⟩

67.4.4 Size

definition *wcount* **where** *wcount* $f\ M = (\lambda x. \text{count}\ M\ x * \text{Suc}\ (f\ x))$

lemma *wcount-union*: *wcount* $f\ (M + N)\ a = \text{wcount}\ f\ M\ a + \text{wcount}\ f\ N\ a$
 ⟨proof⟩

lemma *wcount-add-mset*:

wcount $f\ (\text{add-mset}\ x\ M)\ a = (\text{if}\ x = a\ \text{then}\ \text{Suc}\ (f\ a)\ \text{else}\ 0) + \text{wcount}\ f\ M\ a$
 ⟨proof⟩

definition *size-multiset* :: $('a \Rightarrow \text{nat}) \Rightarrow 'a\ \text{multiset} \Rightarrow \text{nat}$ **where**
size-multiset $f\ M = \text{sum}\ (\text{wcount}\ f\ M)\ (\text{set-mset}\ M)$

lemmas *size-multiset-eq* = *size-multiset-def*[*unfolded wcount-def*]

instantiation *multiset* :: $(\text{type})\ \text{size}$
begin

definition *size-multiset* **where**

size-multiset-overloaded-def: *size-multiset* = *Multiset.size-multiset* $(\lambda -. 0)$

instance ⟨proof⟩

end

lemmas *size-multiset-overloaded-eq* =

size-multiset-overloaded-def[*THEN fun-cong, unfolded size-multiset-eq, simplified*]

lemma *size-multiset-empty* [simp]: *size-multiset* $f\ \{\#\} = 0$

$\langle \text{proof} \rangle$

lemma *size-empty* [simp]: $\text{size } \{\#\} = 0$
 $\langle \text{proof} \rangle$

lemma *size-multiset-single* : $\text{size-multiset } f \ \{\#b\# \} = \text{Suc } (f \ b)$
 $\langle \text{proof} \rangle$

lemma *size-single*: $\text{size } \{\#b\# \} = 1$
 $\langle \text{proof} \rangle$

lemma *sum-wcount-Int*:
 $\text{finite } A \implies \text{sum } (\text{wcount } f \ N) \ (A \cap \text{set-mset } N) = \text{sum } (\text{wcount } f \ N) \ A$
 $\langle \text{proof} \rangle$

lemma *size-multiset-union* [simp]:
 $\text{size-multiset } f \ (M + N :: 'a \text{ multiset}) = \text{size-multiset } f \ M + \text{size-multiset } f \ N$
 $\langle \text{proof} \rangle$

lemma *size-multiset-add-mset* [simp]:
 $\text{size-multiset } f \ (\text{add-mset } a \ M) = \text{Suc } (f \ a) + \text{size-multiset } f \ M$
 $\langle \text{proof} \rangle$

lemma *size-add-mset* [simp]: $\text{size } (\text{add-mset } a \ A) = \text{Suc } (\text{size } A)$
 $\langle \text{proof} \rangle$

lemma *size-union* [simp]: $\text{size } (M + N :: 'a \text{ multiset}) = \text{size } M + \text{size } N$
 $\langle \text{proof} \rangle$

lemma *size-multiset-eq-0-iff-empty* [iff]:
 $\text{size-multiset } f \ M = 0 \longleftrightarrow M = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *size-eq-0-iff-empty* [iff]: $(\text{size } M = 0) = (M = \{\#\})$
 $\langle \text{proof} \rangle$

lemma *nonempty-has-size*: $(S \neq \{\#\}) = (0 < \text{size } S)$
 $\langle \text{proof} \rangle$

lemma *size-eq-Suc-imp-elem*: $\text{size } M = \text{Suc } n \implies \exists a. a \in\# M$
 $\langle \text{proof} \rangle$

lemma *size-eq-Suc-imp-eq-union*:
assumes $\text{size } M = \text{Suc } n$
shows $\exists a \ N. M = \text{add-mset } a \ N$
 $\langle \text{proof} \rangle$

lemma *size-mset-mono*:
fixes $A \ B :: 'a \text{ multiset}$

assumes $A \subseteq\# B$
shows $\text{size } A \leq \text{size } B$
 $\langle \text{proof} \rangle$

lemma *size-filter-mset-lesseq[simp]*: $\text{size } (\text{filter-mset } f \ M) \leq \text{size } M$
 $\langle \text{proof} \rangle$

lemma *size-Diff-submset*:
 $M \subseteq\# M' \implies \text{size } (M' - M) = \text{size } M' - \text{size}(M::'a \text{ multiset})$
 $\langle \text{proof} \rangle$

lemma *size-lt-imp-ex-count-lt*: $\text{size } M < \text{size } N \implies \exists x \in\# N. \text{count } M \ x < \text{count } N \ x$
 $\langle \text{proof} \rangle$

67.5 Induction and case splits

theorem *multiset-induct* [*case-names empty add, induct type: multiset*]:
assumes *empty*: $P \ \{\#\}$
assumes *add*: $\bigwedge x \ M. P \ M \implies P \ (\text{add-mset } x \ M)$
shows $P \ M$
 $\langle \text{proof} \rangle$

lemma *multiset-induct-min* [*case-names empty add*]:
fixes $M :: 'a::\text{linorder multiset}$
assumes
 empty: $P \ \{\#\}$ **and**
 add: $\bigwedge x \ M. P \ M \implies (\forall y \in\# M. y \geq x) \implies P \ (\text{add-mset } x \ M)$
shows $P \ M$
 $\langle \text{proof} \rangle$

lemma *multiset-induct-max* [*case-names empty add*]:
fixes $M :: 'a::\text{linorder multiset}$
assumes
 empty: $P \ \{\#\}$ **and**
 add: $\bigwedge x \ M. P \ M \implies (\forall y \in\# M. y \leq x) \implies P \ (\text{add-mset } x \ M)$
shows $P \ M$
 $\langle \text{proof} \rangle$

lemma *multi-nonempty-split*: $M \neq \{\#\} \implies \exists A \ a. M = \text{add-mset } a \ A$
 $\langle \text{proof} \rangle$

lemma *multiset-cases* [*cases type*]:
obtains $(\text{empty}) \ M = \{\#\} \mid (\text{add}) \ x \ N$ **where** $M = \text{add-mset } x \ N$
 $\langle \text{proof} \rangle$

lemma *multi-drop-mem-not-eq*: $c \in\# B \implies B - \{\#c\} \neq B$
 $\langle \text{proof} \rangle$

lemma *union-filter-mset-complement*[simp]:

$\forall x. P\ x = (\neg Q\ x) \implies \text{filter-mset } P\ M + \text{filter-mset } Q\ M = M$

<proof>

lemma *multiset-partition*: $M = \{\#x \in \# M. P\ x\# \} + \{\#x \in \# M. \neg P\ x\# \}$

<proof>

lemma *mset-subset-size*: $A \subset \# B \implies \text{size } A < \text{size } B$

<proof>

lemma *size-1-singleton-mset*: $\text{size } M = 1 \implies \exists a. M = \{\#a\# \}$

<proof>

lemma *set-mset-subset-singletonD*:

assumes *set-mset* $A \subseteq \{x\}$

shows $A = \text{replicate-mset } (\text{size } A)\ x$

<proof>

lemma *count-conv-size-mset*: $\text{count } A\ x = \text{size } (\text{filter-mset } (\lambda y. y = x)\ A)$

<proof>

lemma *size-conv-count-bool-mset*: $\text{size } A = \text{count } A\ \text{True} + \text{count } A\ \text{False}$

<proof>

67.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-subset-mset-rel*: $\text{wf } \{(M, N :: 'a\ \text{multiset}). M \subset \# N\}$

<proof>

lemma *wfp-subset-mset*[simp]: $\text{wfp } (\subset \#)$

<proof>

lemma *full-multiset-induct* [case-names less]:

assumes *ih*: $\bigwedge B. \forall (A :: 'a\ \text{multiset}). A \subset \# B \longrightarrow P\ A \implies P\ B$

shows $P\ B$

<proof>

lemma *multi-subset-induct* [consumes 2, case-names empty add]:

assumes $F \subseteq \# A$

and *empty*: $P\ \{\#\}$

and *insert*: $\bigwedge a\ F. a \in \# A \implies P\ F \implies P\ (\text{add-mset } a\ F)$

shows $P\ F$

<proof>

67.6 Least and greatest elements

context begin

qualified lemma**assumes** $M \neq \{\#\}$ **and** $\text{transp-on } (\text{set-mset } M) \ R$ **and** $\text{totalp-on } (\text{set-mset } M) \ R$ **shows** $\text{bex-least-element: } (\exists l \in \# M. \forall x \in \# M. x \neq l \longrightarrow R \ l \ x)$ **and** $\text{bex-greatest-element: } (\exists g \in \# M. \forall x \in \# M. x \neq g \longrightarrow R \ x \ g)$ $\langle \text{proof} \rangle$ **end****67.7 The fold combinator****definition** $\text{fold-mset} :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ multiset} \Rightarrow 'b$ **where** $\text{fold-mset } f \ s \ M = \text{Finite-Set.fold } (\lambda x. f \ x \ \frown \ \text{count } M \ x) \ s \ (\text{set-mset } M)$ **lemma** fold-mset-empty $[\text{simp}]$: $\text{fold-mset } f \ s \ \{\#\} = s$ $\langle \text{proof} \rangle$ **lemma** fold-mset-single $[\text{simp}]$: $\text{fold-mset } f \ s \ \{\#x\# \} = f \ x \ s$ $\langle \text{proof} \rangle$ **context** comp-fun-commute **begin****lemma** $\text{fold-mset-add-mset}$ $[\text{simp}]$: $\text{fold-mset } f \ s \ (\text{add-mset } x \ M) = f \ x \ (\text{fold-mset } f \ s \ M)$ $\langle \text{proof} \rangle$ **lemma** $\text{fold-mset-fun-left-comm}$: $f \ x \ (\text{fold-mset } f \ s \ M) = \text{fold-mset } f \ (f \ x \ s) \ M$ $\langle \text{proof} \rangle$ **lemma** fold-mset-union $[\text{simp}]$: $\text{fold-mset } f \ s \ (M + N) = \text{fold-mset } f \ (\text{fold-mset } f \ s \ M) \ N$ $\langle \text{proof} \rangle$ **lemma** fold-mset-fusion :**assumes** $\text{comp-fun-commute } g$ **and** $*$: $\bigwedge x \ y. h \ (g \ x \ y) = f \ x \ (h \ y)$ **shows** $h \ (\text{fold-mset } g \ w \ A) = \text{fold-mset } f \ (h \ w) \ A$ $\langle \text{proof} \rangle$ **end****lemma** $\text{union-fold-mset-add-mset}$: $A + B = \text{fold-mset add-mset } A \ B$ $\langle \text{proof} \rangle$

A note on code generation: When defining some function containing a

subterm *fold-mset* F , code generation is not automatic. When interpreting locale *left-commutative* with F , the would be code thms for *fold-mset* become thms like *fold-mset* F z $\{\#\}$ $= z$ where F is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for F . See the image operator below.

67.8 Image

definition *image-mset* $:: ('a \Rightarrow 'b) \Rightarrow 'a \text{ multiset} \Rightarrow 'b \text{ multiset}$ **where**
image-mset $f = \text{fold-mset } (\text{add-mset} \circ f) \{\#\}$

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* $(\text{add-mset} \circ f)$
 $\langle \text{proof} \rangle$

lemma *image-mset-empty* [simp]: *image-mset* $f \{\#\} = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *image-mset-single*: *image-mset* $f \{\#x\# \} = \{\#f x\# \}$
 $\langle \text{proof} \rangle$

lemma *image-mset-union* [simp]: *image-mset* $f (M + N) = \text{image-mset } f M + \text{image-mset } f N$
 $\langle \text{proof} \rangle$

corollary *image-mset-add-mset* [simp]:
image-mset $f (\text{add-mset } a M) = \text{add-mset } (f a) (\text{image-mset } f M)$
 $\langle \text{proof} \rangle$

lemma *set-image-mset* [simp]: *set-mset* $(\text{image-mset } f M) = \text{image } f (\text{set-mset } M)$
 $\langle \text{proof} \rangle$

lemma *size-image-mset* [simp]: *size* $(\text{image-mset } f M) = \text{size } M$
 $\langle \text{proof} \rangle$

lemma *image-mset-is-empty-iff* [simp]: *image-mset* $f M = \{\#\} \longleftrightarrow M = \{\#\}$
 $\langle \text{proof} \rangle$

lemma *image-mset-If*:
image-mset $(\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) A =$
image-mset $f (\text{filter-mset } P A) + \text{image-mset } g (\text{filter-mset } (\lambda x. \neg P x) A)$
 $\langle \text{proof} \rangle$

lemma *filter-mset-image-mset*:
 $\text{filter-mset } P (\text{image-mset } f A) = \text{image-mset } f (\text{filter-mset } (\lambda x. P (f x)) A)$
 $\langle \text{proof} \rangle$

lemma *image-mset-Diff*:
assumes $B \subseteq\# A$
shows *image-mset* $f (A - B) = \text{image-mset } f A - \text{image-mset } f B$

$\langle proof \rangle$

lemma *minus-add-mset-if-not-in-lhs*[simp]: $x \notin \# A \implies A - \text{add-mset } x B = A - B$
 $\langle proof \rangle$

lemma *image-mset-diff-if-inj*:
fixes $f A B$
assumes *inj* f
shows $\text{image-mset } f (A - B) = \text{image-mset } f A - \text{image-mset } f B$
 $\langle proof \rangle$

lemma *count-image-mset*:
 $\langle \text{count } (\text{image-mset } f A) x = (\sum y \in f^{-1} \{x\} \cap \text{set-mset } A. \text{count } A y) \rangle$
 $\langle proof \rangle$

lemma *count-image-mset'*:
 $\langle \text{count } (\text{image-mset } f X) y = (\sum x \mid x \in \# X \wedge y = f x. \text{count } X x) \rangle$
 $\langle proof \rangle$

lemma *image-mset-subseteq-mono*: $A \subseteq \# B \implies \text{image-mset } f A \subseteq \# \text{image-mset } f B$
 $\langle proof \rangle$

lemma *image-mset-subset-mono*: $M \subset \# N \implies \text{image-mset } f M \subset \# \text{image-mset } f N$
 $\langle proof \rangle$

syntax (ASCII)

-comprehension-mset :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset}$
 $(\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - / . - : \# - \# \} \rangle)$

syntax

-comprehension-mset :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset}$
 $(\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - / . - \in \# - \# \} \rangle)$

syntax-consts

-comprehension-mset $\equiv \text{image-mset}$

translations

$\{ \# e. x \in \# M \# \} \equiv \text{CONST image-mset } (\lambda x. e) M$

syntax (ASCII)

-comprehension-mset' :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow \text{bool} \Rightarrow 'a \text{ multiset}$
 $(\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - / \mid - : \# - / - \# \} \rangle)$

syntax

-comprehension-mset' :: $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow \text{bool} \Rightarrow 'a \text{ multiset}$
 $(\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - / \mid - \in \# - / - \# \} \rangle)$

syntax-consts

-comprehension-mset' $\equiv \text{image-mset}$

translations

$\{ \# e \mid x \in \# M. P \# \} \mapsto \{ \# e. x \in \# \{ \# x \in \# M. P \# \} \# \}$

This allows to write not just filters like $\{\#x \in \# M. x < c\# \}$ but also images like $\{\#x + x. x \in \# M\# \}$ and $\{\#x+x | x \in \# M. x < c\# \}$, where the latter is currently displayed as $\{\#x + x. x \in \# \{\#x \in \# M. x < c\# \}\# \}$.

lemma *in-image-mset*: $y \in \# \{\#f x. x \in \# M\# \} \longleftrightarrow y \in f \text{ ' set-mset } M$
 $\langle \text{proof} \rangle$

functor *image-mset*: *image-mset*
 $\langle \text{proof} \rangle$

declare
image-mset.id [simp]
image-mset.identity [simp]

lemma *image-mset-id*[simp]: *image-mset id* $x = x$
 $\langle \text{proof} \rangle$

lemma *image-mset-cong*: $(\bigwedge x. x \in \# M \implies f x = g x) \implies \{\#f x. x \in \# M\# \} = \{\#g x. x \in \# M\# \}$
 $\langle \text{proof} \rangle$

lemma *image-mset-cong-pair*:
 $(\forall x y. (x, y) \in \# M \longrightarrow f x y = g x y) \implies \{\#f x y. (x, y) \in \# M\# \} = \{\#g x y. (x, y) \in \# M\# \}$
 $\langle \text{proof} \rangle$

lemma *image-mset-const-eq*:
 $\{\#c. a \in \# M\# \} = \text{replicate-mset (size } M) c$
 $\langle \text{proof} \rangle$

lemma *image-mset-filter-mset-swap*:
image-mset f (*filter-mset* $(\lambda x. P (f x))$ M) = *filter-mset* P (*image-mset* f M)
 $\langle \text{proof} \rangle$

lemma *image-mset-eq-plusD*:
image-mset f $A = B + C \implies \exists B' C'. A = B' + C' \wedge B = \text{image-mset } f B' \wedge C = \text{image-mset } f C'$
 $\langle \text{proof} \rangle$

lemma *image-mset-eq-image-mset-plusD*:
assumes *image-mset* f $A = \text{image-mset } f B + C$ **and** *inj-f*: *inj-on* f (*set-mset* $A \cup \text{set-mset } B$)
shows $\exists C'. A = B + C' \wedge C = \text{image-mset } f C'$
 $\langle \text{proof} \rangle$

lemma *image-mset-eq-plus-image-msetD*:
image-mset f $A = B + \text{image-mset } f C \implies \text{inj-on } f (\text{set-mset } A \cup \text{set-mset } C) \implies \exists B'. A = B' + C \wedge B = \text{image-mset } f B'$
 $\langle \text{proof} \rangle$

67.9 Further conversions

primrec *mset* :: 'a list \Rightarrow 'a multiset **where**

mset [] = {#} |
mset (a # x) = add-mset a (*mset* x)

lemma *in-multiset-in-set*:

$x \in\# \text{ mset } xs \longleftrightarrow x \in \text{ set } xs$
 <proof>

lemma *count-mset*:

$\text{count } (\text{mset } xs) \ x = \text{count-list } xs \ x$
 <proof>

lemma *mset-zero-iff[simp]*: $(\text{mset } x = \{\#\}) = (x = [])$

<proof>

lemma *mset-zero-iff-right[simp]*: $(\{\#\} = \text{mset } x) = (x = [])$

<proof>

lemma *mset-replicate [simp]*: $\text{mset } (\text{replicate } n \ x) = \text{replicate-mset } n \ x$

<proof>

lemma *count-mset-gt-0*: $x \in \text{ set } xs \implies \text{count } (\text{mset } xs) \ x > 0$

<proof>

lemma *count-mset-0-iff [simp]*: $\text{count } (\text{mset } xs) \ x = 0 \longleftrightarrow x \notin \text{ set } xs$

<proof>

lemma *mset-single-iff[iff]*: $\text{mset } xs = \{\#x\#\} \longleftrightarrow xs = [x]$

<proof>

lemma *mset-single-iff-right[iff]*: $\{\#x\#\} = \text{mset } xs \longleftrightarrow xs = [x]$

<proof>

lemma *set-mset-mset[simp]*: $\text{set-mset } (\text{mset } xs) = \text{set } xs$

<proof>

lemma *set-mset-comp-mset [simp]*: $\text{set-mset} \circ \text{mset} = \text{set}$

<proof>

lemma *size-mset [simp]*: $\text{size } (\text{mset } xs) = \text{length } xs$

<proof>

lemma *mset-append [simp]*: $\text{mset } (xs @ ys) = \text{mset } xs + \text{mset } ys$

<proof>

lemma *mset-filter[simp]*: $\text{mset } (\text{filter } P \ xs) = \{\#x \in\# \text{ mset } xs. P \ x \#\}$

<proof>

lemma *mset-rev [simp]*:

$mset\ (rev\ xs) = mset\ xs$

$\langle proof \rangle$

lemma *surj-mset*: *surj mset*

$\langle proof \rangle$

lemma *distinct-count-atmost-1*:

$distinct\ x = (\forall a.\ count\ (mset\ x)\ a = (if\ a \in set\ x\ then\ 1\ else\ 0))$

$\langle proof \rangle$

lemma *mset-eq-setD*:

assumes $mset\ xs = mset\ ys$

shows $set\ xs = set\ ys$

$\langle proof \rangle$

lemma *set-eq-iff-mset-eq-distinct*:

$\langle distinct\ x \implies distinct\ y \implies set\ x = set\ y \longleftrightarrow mset\ x = mset\ y \rangle$

$\langle proof \rangle$

lemma *set-eq-iff-mset-remdups-eq*:

$\langle set\ x = set\ y \longleftrightarrow mset\ (remdups\ x) = mset\ (remdups\ y) \rangle$

$\langle proof \rangle$

lemma *mset-eq-imp-distinct-iff*:

$\langle distinct\ xs \longleftrightarrow distinct\ ys \rangle$ **if** $\langle mset\ xs = mset\ ys \rangle$

$\langle proof \rangle$

lemma *nth-mem-mset*: $i < length\ ls \implies (ls\ !\ i) \in\# mset\ ls$

$\langle proof \rangle$

lemma *mset-remove1 [simp]*: $mset\ (remove1\ a\ xs) = mset\ xs - \{ \#a\# \}$

$\langle proof \rangle$

lemma *mset-eq-length*:

assumes $mset\ xs = mset\ ys$

shows $length\ xs = length\ ys$

$\langle proof \rangle$

lemma *mset-eq-length-filter*:

assumes $mset\ xs = mset\ ys$

shows $count-list\ xs\ z = count-list\ ys\ z$

$\langle proof \rangle$

lemma *fold-multiset-equiv*:

$\langle List.fold\ f\ xs = List.fold\ f\ ys \rangle$

if f : $\langle \bigwedge x\ y.\ x \in set\ xs \implies y \in set\ xs \implies f\ x \circ f\ y = f\ y \circ f\ x \rangle$

and $\langle mset\ xs = mset\ ys \rangle$

$\langle proof \rangle$

lemma *fold-permuted-eq*:

$\langle \text{List.fold } (\odot) \text{ } xs \text{ } z = \text{List.fold } (\odot) \text{ } ys \text{ } z \rangle$
if $\langle mset \text{ } xs = mset \text{ } ys \rangle$
and $\langle P \text{ } z \rangle$ **and** $P: \langle \bigwedge x \text{ } z. x \in set \text{ } xs \implies P \text{ } z \implies P \text{ } (x \odot z) \rangle$
and $f: \langle \bigwedge x \text{ } y \text{ } z. x \in set \text{ } xs \implies y \in set \text{ } xs \implies P \text{ } z \implies x \odot (y \odot z) = y \odot (x \odot z) \rangle$
for f (**infixl** \odot 70)
 $\langle proof \rangle$

lemma *mset-shuffles*: $zs \in shuffles \text{ } xs \text{ } ys \implies mset \text{ } zs = mset \text{ } xs + mset \text{ } ys$
 $\langle proof \rangle$

lemma *mset-insort [simp]*: $mset \text{ } (insort \text{ } x \text{ } xs) = add\text{-}mset \text{ } x \text{ } (mset \text{ } xs)$
 $\langle proof \rangle$

lemma *mset-map[simp]*: $mset \text{ } (map \text{ } f \text{ } xs) = image\text{-}mset \text{ } f \text{ } (mset \text{ } xs)$
 $\langle proof \rangle$

lemma *mset-removeAll-eq*:
 $\langle mset \text{ } (removeAll \text{ } x \text{ } xs) = filter\text{-}mset \text{ } ((\neq) \text{ } x) \text{ } (mset \text{ } xs) \rangle$
 $\langle proof \rangle$

lemma *singleton-set-mset-subset*: **fixes** $X \text{ } Y :: 'a \text{ list set}$
assumes $\forall xs \in X. set \text{ } xs \subseteq \{a\}$ $mset \text{ } 'X \subseteq mset \text{ } 'Y$
shows $X \subseteq Y$
 $\langle proof \rangle$

lemma *singleton-set-mset-eq*: **fixes** $X \text{ } Y :: 'a \text{ list set}$
assumes $\forall xs \in X. set \text{ } xs \subseteq \{a\}$ $mset \text{ } 'X = mset \text{ } 'Y$
shows $X = Y$
 $\langle proof \rangle$

global-interpretation *mset-set*: *folding* $add\text{-}mset \text{ } \{\#\}$
defines $mset\text{-}set = folding\text{-}on.F \text{ } add\text{-}mset \text{ } \{\#\}$
 $\langle proof \rangle$

lemma *sum-multiset-singleton [simp]*: $sum \text{ } (\lambda n. \{\#n\# \}) \text{ } A = mset\text{-}set \text{ } A$
 $\langle proof \rangle$

lemma *count-mset-set [simp]*:
 $finite \text{ } A \implies x \in A \implies count \text{ } (mset\text{-}set \text{ } A) \text{ } x = 1$ (**is** *PROP* ?P)
 $\neg finite \text{ } A \implies count \text{ } (mset\text{-}set \text{ } A) \text{ } x = 0$ (**is** *PROP* ?Q)
 $x \notin A \implies count \text{ } (mset\text{-}set \text{ } A) \text{ } x = 0$ (**is** *PROP* ?R)
 $\langle proof \rangle$

lemma *elem-mset-set[simp, intro]*: $finite \text{ } A \implies x \in \# \text{ } mset\text{-}set \text{ } A \longleftrightarrow x \in A$
 $\langle proof \rangle$

lemma *mset-set-Union*:

finite A \implies finite B $\implies A \cap B = \{\}$ \implies mset-set (A \cup B) = mset-set A + mset-set B
<proof>

lemma *filter-mset-mset-set* [simp]:

finite A \implies filter-mset P (mset-set A) = mset-set {x \in A. P x}
<proof>

lemma *mset-set-Diff*:

assumes finite A B \subseteq A
shows mset-set (A - B) = mset-set A - mset-set B
<proof>

lemma *mset-minus-list-mset*[simp]: *mset(minus-list-mset xs ys) = mset xs - mset ys*
<proof>

lemma *mset-set-set*: *distinct xs \implies mset-set (set xs) = mset xs*
<proof>

lemma *count-mset-set'*: *count (mset-set A) x = (if finite A \wedge x \in A then 1 else 0)*
<proof>

lemma *subset-imp-msubset-mset-set*:

assumes A \subseteq B finite B
shows mset-set A $\subseteq\#$ mset-set B
<proof>

lemma *mset-set-set-mset-msubset*: *mset-set (set-mset A) $\subseteq\#$ A*
<proof>

lemma *mset-set-upto-eq-mset-upto*:

<mset-set {.. n } = mset [0.. n]
<proof>

context *linorder*

begin

definition *sorted-list-of-multiset* :: *'a multiset \Rightarrow 'a list*

where

sorted-list-of-multiset M = fold-mset insert [] M

lemma *sorted-list-of-multiset-empty* [simp]:

sorted-list-of-multiset {#} = []
<proof>

lemma *sorted-list-of-multiset-singleton* [simp]:

sorted-list-of-multiset $\{\#x\# \} = [x]$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-multiset-insert* [simp]:
sorted-list-of-multiset (add-mset x M) = *List.insert* x (*sorted-list-of-multiset* M)
 $\langle \text{proof} \rangle$

end

lemma *mset-sorted-list-of-multiset*[simp]: *mset* (*sorted-list-of-multiset* M) = M
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-multiset-mset*[simp]: *sorted-list-of-multiset* (*mset* xs) = *sort* xs
 $\langle \text{proof} \rangle$

lemma *finite-set-mset-mset-set*[simp]: *finite* $A \implies \text{set-mset}$ (*mset-set* A) = A
 $\langle \text{proof} \rangle$

lemma *mset-set-empty-iff*: *mset-set* $A = \{\#\} \longleftrightarrow A = \{\} \vee \text{infinite } A$
 $\langle \text{proof} \rangle$

lemma *infinite-set-mset-mset-set*: $\neg \text{finite } A \implies \text{set-mset}$ (*mset-set* A) = $\{\}$
 $\langle \text{proof} \rangle$

lemma *set-sorted-list-of-multiset* [simp]:
set (*sorted-list-of-multiset* M) = *set-mset* M
 $\langle \text{proof} \rangle$

lemma *sorted-sorted-list-of-multiset* [iff]:
 $\langle \text{sorted} \text{ (sorted-list-of-multiset } M) \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-mset-set* [simp]:
sorted-list-of-multiset (*mset-set* A) = *sorted-list-of-set* A
 $\langle \text{proof} \rangle$

lemma *mset-upt* [simp]: *mset* [$m..<n$] = *mset-set* $\{m..<n\}$
 $\langle \text{proof} \rangle$

lemma *image-mset-map-of*:
 $\text{distinct} \text{ (map fst } xs) \implies \{\# \text{the (map-of } xs \text{ } i). i \in \# \text{ mset (map fst } xs) \# \} = \text{mset (map snd } xs)$
 $\langle \text{proof} \rangle$

lemma *msubset-mset-set-iff*[simp]:
assumes *finite* A *finite* B
shows *mset-set* $A \subseteq \# \text{ mset-set } B \longleftrightarrow A \subseteq B$
 $\langle \text{proof} \rangle$

lemma *mset-set-eq-iff* [simp]:
 assumes *finite A finite B*
 shows *mset-set A = mset-set B \longleftrightarrow A = B*
<proof>

lemma *image-mset-mset-set*:
 assumes *inj-on f A*
 shows *image-mset f (mset-set A) = mset-set (f ` A)*
<proof>

67.10 More properties of the replicate, repeat, and image operations

lemma *in-replicate-mset* [simp]: *x \in # replicate-mset n y \longleftrightarrow n > 0 \wedge x = y*
<proof>

lemma *set-mset-replicate-mset-subset* [simp]: *set-mset (replicate-mset n x) = (if n = 0 then {} else {x})*
<proof>

lemma *size-replicate-mset* [simp]: *size (replicate-mset n M) = n*
<proof>

lemma *size-repeat-mset* [simp]: *size (repeat-mset n A) = n * size A*
<proof>

lemma *size-multiset-sum* [simp]: *size ($\sum x \in A. f x :: 'a$ multiset) = ($\sum x \in A. size (f x)$)*
<proof>

lemma *size-multiset-sum-list* [simp]: *size ($\sum X \leftarrow Xs. X :: 'a$ multiset) = ($\sum X \leftarrow Xs. size X$)*
<proof>

lemma *count-le-replicate-mset-subset-eq*: *n \leq count M x \longleftrightarrow replicate-mset n x \subseteq # M*
<proof>

lemma *replicate-count-mset-eq-filter-eq*: *replicate (count (mset xs) k) k = filter (HOL.eq k) xs*
<proof>

lemma *replicate-mset-eq-empty-iff* [simp]: *replicate-mset n a = {} \longleftrightarrow n = 0*
<proof>

lemma *replicate-mset-eq-iff*:
replicate-mset m a = replicate-mset n b \longleftrightarrow m = 0 \wedge n = 0 \vee m = n \wedge a = b
<proof>

lemma *repeat-mset-cancel1*: $\text{repeat-mset } a \ A = \text{repeat-mset } a \ B \longleftrightarrow A = B \vee a = 0$
 ⟨proof⟩

lemma *repeat-mset-cancel2*: $\text{repeat-mset } a \ A = \text{repeat-mset } b \ A \longleftrightarrow a = b \vee A = \{\#\}$
 ⟨proof⟩

lemma *repeat-mset-eq-empty-iff*: $\text{repeat-mset } n \ A = \{\#\} \longleftrightarrow n = 0 \vee A = \{\#\}$
 ⟨proof⟩

lemma *image-replicate-mset* [simp]:
 $\text{image-mset } f \ (\text{replicate-mset } n \ a) = \text{replicate-mset } n \ (f \ a)$
 ⟨proof⟩

lemma *replicate-mset-msubseteq-iff*:
 $\text{replicate-mset } m \ a \subseteq\# \text{replicate-mset } n \ b \longleftrightarrow m = 0 \vee a = b \wedge m \leq n$
 ⟨proof⟩

lemma *msubseteq-replicate-msetE*:
 assumes $A \subseteq\# \text{replicate-mset } n \ a$
 obtains m where $m \leq n$ and $A = \text{replicate-mset } m \ a$
 ⟨proof⟩

lemma *count-image-mset-lt-imp-lt-raw*:
 assumes
 $\text{finite } A$ and
 $A = \text{set-mset } M \cup \text{set-mset } N$ and
 $\text{count } (\text{image-mset } f \ M) \ b < \text{count } (\text{image-mset } f \ N) \ b$
 shows $\exists x. f \ x = b \wedge \text{count } M \ x < \text{count } N \ x$
 ⟨proof⟩

lemma *count-image-mset-lt-imp-lt*:
 assumes $\text{cnt-b: count } (\text{image-mset } f \ M) \ b < \text{count } (\text{image-mset } f \ N) \ b$
 shows $\exists x. f \ x = b \wedge \text{count } M \ x < \text{count } N \ x$
 ⟨proof⟩

lemma *count-image-mset-le-imp-lt-raw*:
 assumes
 $\text{finite } A$ and
 $A = \text{set-mset } M \cup \text{set-mset } N$ and
 $\text{count } (\text{image-mset } f \ M) \ (f \ a) + \text{count } N \ a < \text{count } (\text{image-mset } f \ N) \ (f \ a) + \text{count } M \ a$
 shows $\exists b. f \ b = f \ a \wedge \text{count } M \ b < \text{count } N \ b$
 ⟨proof⟩

lemma *count-image-mset-le-imp-lt*:
 assumes

$\text{count } (\text{image-mset } f \ M) \ (f \ a) \leq \text{count } (\text{image-mset } f \ N) \ (f \ a)$ **and**
 $\text{count } M \ a > \text{count } N \ a$
shows $\exists b. f \ b = f \ a \wedge \text{count } M \ b < \text{count } N \ b$
 $\langle \text{proof} \rangle$

lemma *size-filter-unsat-elim*:
assumes $x \in\# \ M$ **and** $\neg P \ x$
shows $\text{size } \{\#x \in\# \ M. \ P \ x\# \} < \text{size } M$
 $\langle \text{proof} \rangle$

lemma *size-filter-ne-elim*: $x \in\# \ M \implies \text{size } \{\#y \in\# \ M. \ y \neq x\# \} < \text{size } M$
 $\langle \text{proof} \rangle$

lemma *size-eq-ex-count-lt*:
assumes $\text{size } M = \text{size } N$ **and** $M \neq N$
shows $\exists x. \text{count } M \ x < \text{count } N \ x$
 $\langle \text{proof} \rangle$

67.11 Big operators

locale *comm-monoid-mset* = *comm-monoid*
begin

interpretation *comp-fun-commute* *f*
 $\langle \text{proof} \rangle$

interpretation *comp?*: *comp-fun-commute* $f \circ g$
 $\langle \text{proof} \rangle$

context
begin

definition $F :: 'a \text{ multiset} \Rightarrow 'a$
where *eq-fold*: $F \ M = \text{fold-mset } f \ \mathbf{1} \ M$

lemma *empty* [*simp*]: $F \ \{\#\} = \mathbf{1}$
 $\langle \text{proof} \rangle$

lemma *singleton* [*simp*]: $F \ \{\#x\# \} = x$
 $\langle \text{proof} \rangle$

lemma *union* [*simp*]: $F \ (M + N) = F \ M * F \ N$
 $\langle \text{proof} \rangle$

lemma *add-mset* [*simp*]: $F \ (\text{add-mset } x \ N) = x * F \ N$
 $\langle \text{proof} \rangle$

lemma *insert* [*simp*]:
shows $F \ (\text{image-mset } g \ (\text{add-mset } x \ A)) = g \ x * F \ (\text{image-mset } g \ A)$

$\langle proof \rangle$

lemma *remove*:

assumes $x \in\# A$

shows $F A = x * F (A - \{\#x\# \})$

$\langle proof \rangle$

lemma *neutral*:

$\forall x \in\# A. x = \mathbf{1} \implies F A = \mathbf{1}$

$\langle proof \rangle$

lemma *neutral-const* [simp]:

$F (\text{image-mset } (\lambda_. \mathbf{1}) A) = \mathbf{1}$

$\langle proof \rangle$ **lemma** *F-image-mset-product*:

$F \{\#g\ x\ j * F \{\#g\ i\ j. i \in\# A\#\}. j \in\# B\#\} =$

$F (\text{image-mset } (g\ x)\ B) * F \{\#F \{\#g\ i\ j. i \in\# A\#\}. j \in\# B\#\}$

$\langle proof \rangle$

lemma *swap*:

$F (\text{image-mset } (\lambda i. F (\text{image-mset } (g\ i)\ B)) A) =$

$F (\text{image-mset } (\lambda j. F (\text{image-mset } (\lambda i. g\ i\ j) A)) B)$

$\langle proof \rangle$

lemma *distrib*: $F (\text{image-mset } (\lambda x. g\ x * h\ x) A) = F (\text{image-mset } g\ A) * F (\text{image-mset } h\ A)$

$\langle proof \rangle$

lemma *union-disjoint*:

$A \cap\# B = \{\#\} \implies F (\text{image-mset } g\ (A \cup\# B)) = F (\text{image-mset } g\ A) * F (\text{image-mset } g\ B)$

$\langle proof \rangle$

end

end

lemma *comp-fun-commute-plus-mset*[simp]: *comp-fun-commute* $((+) :: 'a\ \text{multiset} \Rightarrow - \Rightarrow -)$

$\langle proof \rangle$

declare *comp-fun-commute.fold-mset-add-mset*[OF *comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[iff]: $x \in\# \text{fold-mset } (+)\ M\ NN \longleftrightarrow x \in\# M \vee (\exists N. N \in\# NN \wedge x \in\# N)$

$\langle proof \rangle$

context *comm-monoid-add*

begin

sublocale *sum-mset*: *comm-monoid-mset* plus 0
defines *sum-mset* = *sum-mset.F* $\langle \text{proof} \rangle$

lemma *sum-unfold-sum-mset*:
 $\text{sum } f \ A = \text{sum-mset } (\text{image-mset } f \ (\text{mset-set } A))$
 $\langle \text{proof} \rangle$

end

notation *sum-mset* ($\langle \sum \# \rangle$)

syntax (*ASCII*)
 $\text{-sum-mset-image} :: \text{pttrn} \Rightarrow 'b \ \text{set} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-add}$
 $(\langle \langle \text{indent}=3 \ \text{notation}=\langle \text{binder } SUM \rangle \rangle SUM \text{ :-}\#-. \ - \rangle \ [0, 51, 10] \ 10)$

syntax
 $\text{-sum-mset-image} :: \text{pttrn} \Rightarrow 'b \ \text{set} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-add}$
 $(\langle \langle \text{indent}=3 \ \text{notation}=\langle \text{binder } \sum \rangle \rangle \sum \text{ -}\in\#-. \ - \rangle \ [0, 51, 10] \ 10)$

syntax-consts
 $\text{-sum-mset-image} \Rightarrow \text{sum-mset}$

translations
 $\sum i \in\# \ A. \ b \Rightarrow \text{CONST } \text{sum-mset } (\text{CONST } \text{image-mset } (\lambda i. \ b) \ A)$

context *comm-monoid-add*
begin

lemma *sum-mset-sum-list*:
 $\text{sum-mset } (\text{mset } xs) = \text{sum-list } xs$
 $\langle \text{proof} \rangle$

end

context *canonically-ordered-monoid-add*
begin

lemma *sum-mset-0-iff* [*simp*]:
 $\text{sum-mset } M = 0 \iff (\forall x \in \text{set-mset } M. \ x = 0)$
 $\langle \text{proof} \rangle$

end

context *ordered-comm-monoid-add*
begin

lemma *sum-mset-mono*:
 $\text{sum-mset } (\text{image-mset } f \ K) \leq \text{sum-mset } (\text{image-mset } g \ K)$
if $\bigwedge i. \ i \in\# \ K \implies f \ i \leq g \ i$
 $\langle \text{proof} \rangle$

end

context *cancel-comm-monoid-add*
begin

lemma *sum-mset-diff*:

$sum\text{-}mset\ (M - N) = sum\text{-}mset\ M - sum\text{-}mset\ N$ **if** $N \subseteq\# M$ **for** $M\ N :: 'a$
multiset
 ⟨*proof*⟩

end

context *semiring-0*
begin

lemma *sum-mset-distrib-left*:

$c * (\sum x \in\# M. f\ x) = (\sum x \in\# M. c * f(x))$
 ⟨*proof*⟩

lemma *sum-mset-distrib-right*:

$(\sum x \in\# M. f\ x) * c = (\sum x \in\# M. f\ x * c)$
 ⟨*proof*⟩

end

lemma *sum-mset-product*:

fixes $f :: 'a :: \{comm\text{-}monoid\text{-}add, times\} \Rightarrow 'b :: semiring\text{-}0$
shows $(\sum i \in\# A. f\ i) * (\sum i \in\# B. g\ i) = (\sum i \in\# A. \sum j \in\# B. f\ i * g\ j)$
 ⟨*proof*⟩

context *semiring-1*
begin

lemma *sum-mset-replicate-mset* [*simp*]:

$sum\text{-}mset\ (replicate\text{-}mset\ n\ a) = of\text{-}nat\ n * a$
 ⟨*proof*⟩

lemma *sum-mset-delta*:

$sum\text{-}mset\ (image\text{-}mset\ (\lambda x. if\ x = y\ then\ c\ else\ 0)\ A) = c * of\text{-}nat\ (count\ A\ y)$
 ⟨*proof*⟩

lemma *sum-mset-delta'*:

$sum\text{-}mset\ (image\text{-}mset\ (\lambda x. if\ y = x\ then\ c\ else\ 0)\ A) = c * of\text{-}nat\ (count\ A\ y)$
 ⟨*proof*⟩

end

lemma *of-nat-sum-mset* [*simp*]:

$of\text{-}nat\ (sum\text{-}mset\ A) = sum\text{-}mset\ (image\text{-}mset\ of\text{-}nat\ A)$
 ⟨*proof*⟩

lemma *size-eq-sum-mset*:

$$\text{size } M = (\sum_{a \in \#M}. 1)$$

<proof>

lemma *size-mset-set* [simp]:

$$\text{size } (\text{mset-set } A) = \text{card } A$$

<proof>

lemma *sum-mset-constant* [simp]:

fixes $y :: 'b::\text{semiring-1}$

shows $\langle (\sum_{x \in \#A}. y) = \text{of-nat } (\text{size } A) * y \rangle$

<proof>

lemma *set-mset-Union-mset*[simp]: $\text{set-mset } (\sum_{\#} MM) = (\bigcup M \in \text{set-mset } MM. \text{set-mset } M)$

<proof>

lemma *in-Union-mset-iff*[iff]: $x \in \# \sum_{\#} MM \longleftrightarrow (\exists M. M \in \# MM \wedge x \in \# M)$

<proof>

lemma *count-sum*:

$$\text{count } (\text{sum } f A) x = \text{sum } (\lambda a. \text{count } (f a) x) A$$

<proof>

lemma *sum-eq-empty-iff*:

assumes *finite* A

shows $\text{sum } f A = \{\#\} \longleftrightarrow (\forall a \in A. f a = \{\#\})$

<proof>

lemma *mset-concat*: $\text{mset } (\text{concat } xss) = (\sum_{xs \leftarrow xss. \text{mset } xs})$

<proof>

lemma *set-mset-sum-list* [simp]: $\text{set-mset } (\text{sum-list } xs) = (\bigcup x \in \text{set } xs. \text{set-mset } x)$

<proof>

lemma *filter-mset-sum-list*: $\text{filter-mset } P (\text{sum-list } xs) = \text{sum-list } (\text{map } (\text{filter-mset } P) xs)$

<proof>

lemma *sum-mset-singleton-mset* [simp]: $(\sum_{x \in \#A. \{\#f x\}}) = \text{image-mset } f A$

<proof>

lemma *sum-list-singleton-mset* [simp]: $(\sum_{x \leftarrow xs. \{\#f x\}}) = \text{image-mset } f (\text{mset } xs)$

<proof>

lemma *Union-mset-empty-conv*[simp]: $\sum_{\#} M = \{\#\} \longleftrightarrow (\forall i \in \#M. i = \{\#\})$

<proof>

lemma *Union-image-single-mset*[simp]: $\sum_{\#} (\text{image-mset } (\lambda x. \{\#x\# \}) \ m) = m$
 ⟨proof⟩

lemma *size-mset-sum-mset-conv* [simp]: $\text{size } (\sum_{\#} A :: 'a \text{ multiset}) = (\sum X \in \# A. \text{size } X)$
 ⟨proof⟩

lemma *sum-mset-image-mset-mono-strong*:
 assumes $A \subseteq_{\#} B$ and *f-subeq-g*: $\bigwedge x. x \in \# A \implies f \ x \subseteq_{\#} g \ x$
 shows $(\sum x \in \# A. f \ x) \subseteq_{\#} (\sum x \in \# B. g \ x)$
 ⟨proof⟩

context *comm-monoid-mult*
begin

sublocale *prod-mset*: *comm-monoid-mset times 1*
defines *prod-mset* = *prod-mset.F* ⟨proof⟩

lemma *prod-mset-empty*:
 $\text{prod-mset } \{\#\} = 1$
 ⟨proof⟩

lemma *prod-mset-singleton*:
 $\text{prod-mset } \{\#x\# \} = x$
 ⟨proof⟩

lemma *prod-mset-Un*:
 $\text{prod-mset } (A + B) = \text{prod-mset } A * \text{prod-mset } B$
 ⟨proof⟩

lemma *prod-mset-prod-list*:
 $\text{prod-mset } (\text{mset } xs) = \text{prod-list } xs$
 ⟨proof⟩

lemma *prod-mset-replicate-mset* [simp]:
 $\text{prod-mset } (\text{replicate-mset } n \ a) = a \wedge^n$
 ⟨proof⟩

lemma *prod-unfold-prod-mset*:
 $\text{prod } f \ A = \text{prod-mset } (\text{image-mset } f \ (\text{mset-set } A))$
 ⟨proof⟩

lemma *prod-mset-multiplicity*:
 $\text{prod-mset } M = \text{prod } (\lambda x. x \wedge \text{count } M \ x) \ (\text{set-mset } M)$
 ⟨proof⟩

lemma *prod-mset-delta*: $\text{prod-mset } (\text{image-mset } (\lambda x. \text{if } x = y \text{ then } c \text{ else } 1) \ A) = c \wedge \text{count } A \ y$

$\langle \text{proof} \rangle$

lemma *prod-mset-delta'*: $\text{prod-mset } (\text{image-mset } (\lambda x. \text{if } y = x \text{ then } c \text{ else } 1) A) = c \hat{\ } \text{count } A \ y$
 $\langle \text{proof} \rangle$

lemma *prod-mset-subset-imp-dvd*:
assumes $A \subseteq\# B$
shows $\text{prod-mset } A \ \text{dvd} \ \text{prod-mset } B$
 $\langle \text{proof} \rangle$

lemma *dvd-prod-mset*:
assumes $x \in\# A$
shows $x \ \text{dvd} \ \text{prod-mset } A$
 $\langle \text{proof} \rangle$

end

notation *prod-mset* $(\langle \prod\# \rangle)$

syntax (*ASCII*)
 $\text{-prod-mset-image} :: \text{pttrn} \Rightarrow 'b \ \text{set} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-mult}$
 $(\langle \langle \text{indent}=3 \ \text{notation}=\langle \text{binder } \text{PROD} \rangle \text{PROD} \text{-}\# \text{-} \rangle [0, 51, 10] \ 10)$

syntax
 $\text{-prod-mset-image} :: \text{pttrn} \Rightarrow 'b \ \text{set} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-mult}$
 $(\langle \langle \text{indent}=3 \ \text{notation}=\langle \text{binder } \prod \rangle \prod \text{-}\# \text{-} \rangle [0, 51, 10] \ 10)$

syntax-consts
 $\text{-prod-mset-image} \Rightarrow \text{prod-mset}$

translations
 $\prod i \in\# A. b \Rightarrow \text{CONST prod-mset } (\text{CONST image-mset } (\lambda i. b) A)$

lemma *prod-mset-constant* [*simp*]: $(\prod \text{-}\in\# A. c) = c \hat{\ } \text{size } A$
 $\langle \text{proof} \rangle$

lemma (*in semidom*) *prod-mset-zero-iff* [*iff*]:
 $\text{prod-mset } A = 0 \longleftrightarrow 0 \in\# A$
 $\langle \text{proof} \rangle$

lemma (*in semidom-divide*) *prod-mset-diff*:
assumes $B \subseteq\# A$ **and** $0 \notin\# B$
shows $\text{prod-mset } (A - B) = \text{prod-mset } A \ \text{div} \ \text{prod-mset } B$
 $\langle \text{proof} \rangle$

lemma (*in semidom-divide*) *prod-mset-minus*:
assumes $a \in\# A$ **and** $a \neq 0$
shows $\text{prod-mset } (A - \{\#a\}) = \text{prod-mset } A \ \text{div} \ a$
 $\langle \text{proof} \rangle$

lemma (*in normalization-semidom*) *normalize-prod-mset-normalize*:

normalize (prod-mset (image-mset normalize A)) = normalize (prod-mset A)
 ⟨proof⟩

lemma (in algebraic-semidom) *is-unit-prod-mset-iff*:
is-unit (prod-mset A) \longleftrightarrow ($\forall x \in \# A. \text{is-unit } x$)
 ⟨proof⟩

lemma (in normalization-semidom-multiplicative) *normalize-prod-mset*:
normalize (prod-mset A) = prod-mset (image-mset normalize A)
 ⟨proof⟩

lemma (in normalization-semidom-multiplicative) *normalized-prod-msetI*:
assumes $\bigwedge a. a \in \# A \implies \text{normalize } a = a$
shows normalize (prod-mset A) = prod-mset A
 ⟨proof⟩

lemma *image-prod-mset-multiplicity*:
prod-mset (image-mset f M) = prod ($\lambda x. f x \wedge \text{count } M x$) (set-mset M)
 ⟨proof⟩

67.12 Multiset as order-ignorant lists

context *linorder*
begin

lemma *mset-insort [simp]*:
mset (insort-key k x xs) = add-mset x (mset xs)
 ⟨proof⟩

lemma *mset-sort [simp]*:
mset (sort-key k xs) = mset xs
 ⟨proof⟩

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma *properties-for-sort-key*:
assumes mset ys = mset xs
and $\bigwedge k. k \in \text{set } ys \implies \text{filter } (\lambda x. f\ k = f\ x)\ ys = \text{filter } (\lambda x. f\ k = f\ x)\ xs$
and sorted (map f ys)
shows sort-key f xs = ys
 ⟨proof⟩

lemma *properties-for-sort*:
assumes multiset: mset ys = mset xs
and sorted ys
shows sort xs = ys
 ⟨proof⟩

lemma *sort-key-inj-key-eq*:

assumes *mset-equal*: $mset\ xs = mset\ ys$
and *inj-on* $f\ (set\ xs)$
and *sorted* $(map\ f\ ys)$
shows *sort-key* $f\ xs = ys$
 $\langle proof \rangle$

lemma *sort-key-eq-sort-key*:
assumes $mset\ xs = mset\ ys$
and *inj-on* $f\ (set\ xs)$
shows *sort-key* $f\ xs = sort\ key\ f\ ys$
 $\langle proof \rangle$

lemma *sort-key-by-quicksort*:
 $sort\ key\ f\ xs = sort\ key\ f\ [x \leftarrow xs. f\ x < f\ (xs\ !\ (length\ xs\ div\ 2))]$
 $\quad @\ [x \leftarrow xs. f\ x = f\ (xs\ !\ (length\ xs\ div\ 2))]$
 $\quad @\ sort\ key\ f\ [x \leftarrow xs. f\ x > f\ (xs\ !\ (length\ xs\ div\ 2))]\ (is\ sort\ key\ f\ ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *sort-by-quicksort*:
 $sort\ xs = sort\ [x \leftarrow xs. x < xs\ !\ (length\ xs\ div\ 2)]$
 $\quad @\ [x \leftarrow xs. x = xs\ !\ (length\ xs\ div\ 2)]$
 $\quad @\ sort\ [x \leftarrow xs. x > xs\ !\ (length\ xs\ div\ 2)]\ (is\ sort\ ?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *sort-append*:
assumes $\bigwedge x\ y. x \in set\ xs \implies y \in set\ ys \implies x \leq y$
shows $sort\ (xs\ @\ ys) = sort\ xs\ @\ sort\ ys$
 $\langle proof \rangle$

lemma *sort-append-replicate-left*:
 $(\bigwedge y. y \in set\ xs \implies x \leq y) \implies sort\ (replicate\ n\ x\ @\ xs) = replicate\ n\ x\ @\ sort\ xs$
 $\langle proof \rangle$

lemma *sort-append-replicate-right*:
 $(\bigwedge y. y \in set\ xs \implies x \geq y) \implies sort\ (xs\ @\ replicate\ n\ x) = sort\ xs\ @\ replicate\ n\ x$
 $\langle proof \rangle$

A stable parameterized quicksort

definition *part* :: $('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'b\ list \times 'b\ list \times 'b\ list$ **where**
 $part\ f\ pivot\ xs = ([x \leftarrow xs. f\ x < pivot], [x \leftarrow xs. f\ x = pivot], [x \leftarrow xs. pivot < f\ x])$

lemma *part-code* [code]:
 $part\ f\ pivot\ [] = ([], [], [])$
 $part\ f\ pivot\ (x \# xs) = (let\ (lts, eqs, gts) = part\ f\ pivot\ xs; x' = f\ x\ in$
 $\quad if\ x' < pivot\ then\ (x \# lts, eqs, gts)$
 $\quad else\ if\ x' > pivot\ then\ (lts, eqs, x \# gts)$

else (lts, x # eqs, gts))
 ⟨proof⟩

lemma *sort-key-by-quicksort-code* [code]:
 sort-key f xs =
 (case xs of
 | [] ⇒ []
 | [x] ⇒ xs
 | [x, y] ⇒ (if f x ≤ f y then xs else [y, x])
 | _ ⇒
 let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
 in sort-key f lts @ eqs @ sort-key f gts)
 ⟨proof⟩

end

hide-const (open) part

lemma *sort-sorted-list-of-multiset-eq* [simp]:
 ⟨sort (sorted-list-of-multiset M) = sorted-list-of-multiset M⟩ **for** M :: ⟨'a::linorder
 multiset⟩
 ⟨proof⟩

lemma *mset-remdups-subset-eq*: mset (remdups xs) ⊆# mset xs
 ⟨proof⟩

lemma *mset-update*:
 i < length ls ⇒ mset (ls[i := v]) = add-mset v (mset ls - {#ls ! i#})
 ⟨proof⟩

lemma *mset-swap*:
 i < length ls ⇒ j < length ls ⇒
 mset (ls[j := ls ! i, i := ls ! j]) = mset ls
 ⟨proof⟩

lemma *mset-eq-finite*:
 ⟨finite {ys. mset ys = mset xs}⟩
 ⟨proof⟩

67.13 The multiset order

definition *mult1* :: ('a × 'a) set ⇒ ('a multiset × 'a multiset) set **where**
 mult1 r = {(N, M). ∃ a M0 K. M = add-mset a M0 ∧ N = M0 + K ∧
 (∀ b. b ∈# K ⇒ (b, a) ∈ r)}

definition *mult* :: ('a × 'a) set ⇒ ('a multiset × 'a multiset) set **where**
 mult r = (mult1 r)⁺

definition *multp* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ bool **where**

$$\text{multp } r \ M \ N \longleftrightarrow (M, N) \in \text{mult } \{(x, y). r \ x \ y\}$$

declare *multp-def*[*pred-set-conv*]

lemma *mult1I*:

assumes $M = \text{add-mset } a \ M0$ **and** $N = M0 + K$ **and** $\bigwedge b. b \in \# K \implies (b, a) \in r$
shows $(N, M) \in \text{mult1 } r$
 $\langle \text{proof} \rangle$

lemma *mult1E*:

assumes $(N, M) \in \text{mult1 } r$
obtains $a \ M0 \ K$ **where** $M = \text{add-mset } a \ M0$ $N = M0 + K$ $\bigwedge b. b \in \# K \implies (b, a) \in r$
 $\langle \text{proof} \rangle$

lemma *mono-mult1*:

assumes $r \subseteq r'$ **shows** $\text{mult1 } r \subseteq \text{mult1 } r'$
 $\langle \text{proof} \rangle$

lemma *mono-mult*:

assumes $r \subseteq r'$ **shows** $\text{mult } r \subseteq \text{mult } r'$
 $\langle \text{proof} \rangle$

lemma *mono-multp*[*mono*]: $r \leq r' \implies \text{multp } r \leq \text{multp } r'$
 $\langle \text{proof} \rangle$

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin \text{mult1 } r$
 $\langle \text{proof} \rangle$

67.13.1 Well-foundedness

lemma *less-add*:

assumes $\text{mult1}: (N, \text{add-mset } a \ M0) \in \text{mult1 } r$
shows
 $(\exists M. (M, M0) \in \text{mult1 } r \wedge N = \text{add-mset } a \ M) \vee$
 $(\exists K. (\forall b. b \in \# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
 $\langle \text{proof} \rangle$

lemma *all-accessible*:

assumes $\text{wf } r$
shows $\forall M. M \in \text{Wellfounded.acc } (\text{mult1 } r)$
 $\langle \text{proof} \rangle$

lemma *wf-mult1*: $\text{wf } r \implies \text{wf } (\text{mult1 } r)$
 $\langle \text{proof} \rangle$

lemma *wf-mult*: $\text{wf } r \implies \text{wf } (\text{mult } r)$
 $\langle \text{proof} \rangle$

lemma *wfp-multip*: $wfp\ r \implies wfp\ (multp\ r)$
 $\langle proof \rangle$

67.13.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:

assumes

trans: $trans\ r$ **and**

$MN: (M, N) \in mult\ r$

shows $\exists I\ J\ K. N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in set-mset\ K. \exists j \in set-mset\ J. (k, j) \in r)$
 $\langle proof \rangle$

lemma *multp-implies-one-step*:

$transp\ R \implies multp\ R\ M\ N \implies \exists I\ J\ K. N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in \#K. \exists x \in \#J. R\ k\ x)$
 $\langle proof \rangle$

lemma *one-step-implies-mult*:

assumes

$J \neq \{\#\}$ **and**

$\forall k \in set-mset\ K. \exists j \in set-mset\ J. (k, j) \in r$

shows $(I + K, I + J) \in mult\ r$

$\langle proof \rangle$

lemma *one-step-implies-multp*:

$J \neq \{\#\} \implies \forall k \in \#K. \exists j \in \#J. R\ k\ j \implies multp\ R\ (I + K)\ (I + J)$

$\langle proof \rangle$

lemma *subset-implies-mult*:

assumes *sub*: $A \subset \# B$

shows $(A, B) \in mult\ r$

$\langle proof \rangle$

lemma *subset-implies-multp*: $A \subset \# B \implies multp\ r\ A\ B$

$\langle proof \rangle$

lemma *multp-repeat-mset-repeat-msetI*:

assumes *transp* R **and** *multp* $R\ A\ B$ **and** $n \neq 0$

shows $multp\ R\ (repeat-mset\ n\ A)\ (repeat-mset\ n\ B)$

$\langle proof \rangle$

67.13.3 Monotonicity

lemma *multp-mono-strong*:

assumes *multp* $R\ M1\ M2$ **and** *transp* R **and**

S-if-R: $\bigwedge x\ y. x \in set-mset\ M1 \implies y \in set-mset\ M2 \implies R\ x\ y \implies S\ x\ y$

shows *multp S M1 M2*
 $\langle \text{proof} \rangle$

lemma *mult-mono-strong*:

assumes $(M1, M2) \in \text{mult } r$ **and** *trans r* **and**
S-if-R: $\bigwedge x y. x \in \text{set-mset } M1 \implies y \in \text{set-mset } M2 \implies (x, y) \in r \implies (x, y) \in s$
shows $(M1, M2) \in \text{mult } s$
 $\langle \text{proof} \rangle$

lemma *monotone-on-multip-multip-image-mset*:

assumes *monotone-on A orda ordb f* **and** *transp orda*
shows *monotone-on* $\{M. \text{set-mset } M \subseteq A\}$ *(multip orda) (multip ordb) (image-mset f)*
 $\langle \text{proof} \rangle$

lemma *monotone-multip-multip-image-mset*:

assumes *monotone orda ordb f* **and** *transp orda*
shows *monotone* *(multip orda) (multip ordb) (image-mset f)*
 $\langle \text{proof} \rangle$

lemma *multip-image-mset-image-msetI*:

assumes *multip* $(\lambda x y. R (f x) (f y))$ *M1 M2* **and** *transp R*
shows *multip R (image-mset f M1) (image-mset f M2)*
 $\langle \text{proof} \rangle$

lemma *multip-image-mset-image-msetD*:

assumes
multip R (image-mset f A) (image-mset f B) **and**
transp R **and**
inj-on-f: *inj-on f (set-mset A \cup set-mset B)*
shows *multip* $(\lambda x y. R (f x) (f y))$ *A B*
 $\langle \text{proof} \rangle$

67.13.4 The multiset extension is cancellative for multiset union

lemma *mult-cancel*:

assumes *trans s* **and** *irrefl-on (set-mset Z) s*
shows $(X + Z, Y + Z) \in \text{mult } s \longleftrightarrow (X, Y) \in \text{mult } s$ **(is ?L \longleftrightarrow ?R)**
 $\langle \text{proof} \rangle$

lemma *multip-cancel*:

transp R \implies irreflp-on (set-mset Z) R \implies multip R (X + Z) (Y + Z) \longleftrightarrow multip R X Y
 $\langle \text{proof} \rangle$

lemma *mult-cancel-add-mset*:

trans r \implies irrefl-on {z} r \implies
 $((\text{add-mset } z X, \text{add-mset } z Y) \in \text{mult } r) = ((X, Y) \in \text{mult } r)$

$\langle \text{proof} \rangle$

lemma *multp-cancel-add-mset*:

transp $R \implies \text{irreflp-on } \{z\} R \implies \text{multp } R (\text{add-mset } z X) (\text{add-mset } z Y) = \text{multp } R X Y$
 $\langle \text{proof} \rangle$

lemma *mult-cancel-max0*:

assumes *trans* s **and** *irrefl-on* $(\text{set-mset } X \cap \text{set-mset } Y) s$
shows $(X, Y) \in \text{mult } s \iff (X - X \cap \# Y, Y - X \cap \# Y) \in \text{mult } s$ (**is** ? $L \iff ?R$)
 $\langle \text{proof} \rangle$

lemma *mult-cancel-max*:

trans $r \implies \text{irrefl-on } (\text{set-mset } X \cap \text{set-mset } Y) r \implies$
 $(X, Y) \in \text{mult } r \iff (X - Y, Y - X) \in \text{mult } r$
 $\langle \text{proof} \rangle$

lemma *multp-cancel-max*:

transp $R \implies \text{irreflp-on } (\text{set-mset } X \cap \text{set-mset } Y) R \implies \text{multp } R X Y \iff \text{multp } R (X - Y) (Y - X)$
 $\langle \text{proof} \rangle$

67.13.5 Strict partial-order properties

lemma *mult1-lessE*:

assumes $(N, M) \in \text{mult1 } \{(a, b). r a b\}$ **and** *asympt* r
obtains $a M0 K$ **where** $M = \text{add-mset } a M0$ $N = M0 + K$
 $a \notin \# K \wedge b. b \in \# K \implies r b a$
 $\langle \text{proof} \rangle$

lemma *trans-on-mult*:

assumes *trans-on* $A r$ **and** $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$
shows *trans-on* $B (\text{mult } r)$
 $\langle \text{proof} \rangle$

lemma *trans-mult*: *trans* $r \implies \text{trans } (\text{mult } r)$

$\langle \text{proof} \rangle$

lemma *transp-on-multip*:

assumes *transp-on* $A r$ **and** $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$
shows *transp-on* $B (\text{multp } r)$
 $\langle \text{proof} \rangle$

lemma *transp-multip*: *transp* $r \implies \text{transp } (\text{multp } r)$

$\langle \text{proof} \rangle$

lemma *irrefl-mult*:

assumes *trans* r *irrefl* r

shows *irrefl* (*mult* *r*)
 $\langle \text{proof} \rangle$

lemma *irreflp-multp*: *transp* *R* \implies *irreflp* *R* \implies *irreflp* (*multp* *R*)
 $\langle \text{proof} \rangle$

instantiation *multiset* :: (*preorder*) *order* **begin**

definition *less-multiset* :: '*a* *multiset* \Rightarrow '*a* *multiset* \Rightarrow *bool*
where *M* < *N* \longleftrightarrow *multp* (<) *M* *N*

definition *less-eq-multiset* :: '*a* *multiset* \Rightarrow '*a* *multiset* \Rightarrow *bool*
where *less-eq-multiset* *M* *N* \longleftrightarrow *M* < *N* \vee *M* = *N*

instance
 $\langle \text{proof} \rangle$

end

lemma *mset-le-irrefl* [*elim!*]:
fixes *M* :: '*a*::*preorder multiset*
shows *M* < *M* \implies *R*
 $\langle \text{proof} \rangle$

lemma *wfp-less-multiset*[*simp*]:
assumes *wf*: *wfp* (<) :: ('*a* :: *preorder*) \Rightarrow '*a* \Rightarrow *bool*)
shows *wfp* ((<) :: '*a* *multiset* \Rightarrow '*a* *multiset* \Rightarrow *bool*)
 $\langle \text{proof} \rangle$

67.13.6 Strict total-order properties

lemma *total-on-mult*:
assumes *total-on* *A* *r* **and** *trans* *r* **and** $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$
shows *total-on* *B* (*mult* *r*)
 $\langle \text{proof} \rangle$

lemma *total-mult*: *total* *r* \implies *trans* *r* \implies *total* (*mult* *r*)
 $\langle \text{proof} \rangle$

lemma *totalp-on-multp*:
totalp-on *A* *R* \implies *transp* *R* \implies ($\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$) \implies *totalp-on*
B (*multp* *R*)
 $\langle \text{proof} \rangle$

lemma *totalp-multp*: *totalp* *R* \implies *transp* *R* \implies *totalp* (*multp* *R*)
 $\langle \text{proof} \rangle$

67.14 Quasi-executable version of the multiset extension

Predicate variants of *mult* and the reflexive closure of *mult*, which are executable whenever the given predicate *P* is. Together with the standard code equations for $(\cap\#)$ and $(-)$ this should yield quadratic (with respect to calls to *P*) implementations of *multp-code* and *multeqp-code*.

definition *multp-code* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$
where

$$\begin{aligned} \text{multp-code } P \ N \ M = \\ (\text{let } Z = M \cap\# \ N; \ X = M - Z \text{ in} \\ X \neq \{\#\} \wedge (\text{let } Y = N - Z \text{ in } (\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P \ y \ x))) \end{aligned}$$

definition *multeqp-code* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$
where

$$\begin{aligned} \text{multeqp-code } P \ N \ M = \\ (\text{let } Z = M \cap\# \ N; \ X = M - Z; \ Y = N - Z \text{ in} \\ (\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P \ y \ x)) \end{aligned}$$

lemma *multp-code-iff-mult*:

assumes *irrefl-on* $(\text{set-mset } N \cap \text{set-mset } M) \ R$ **and** *trans* *R* **and**

[*simp*]: $\bigwedge x \ y. P \ x \ y \longleftrightarrow (x, y) \in R$

shows $\text{multp-code } P \ N \ M \longleftrightarrow (N, M) \in \text{mult } R$ (**is** $?L \longleftrightarrow ?R$)

<proof>

lemma *multp-code-iff-multp*:

irreflp-on $(\text{set-mset } M \cap \text{set-mset } N) \ R \implies \text{transp } R \implies \text{multp-code } R \ M \ N$
 $\longleftrightarrow \text{multp } R \ M \ N$

<proof>

lemma *multp-code-eq-multp*:

assumes *irreflp* *R* **and** *transp* *R*

shows $\text{multp-code } R = \text{multp } R$

<proof>

lemma *multeqp-code-iff-reflcl-mult*:

assumes *irrefl-on* $(\text{set-mset } N \cap \text{set-mset } M) \ R$ **and** *trans* *R* **and** $\bigwedge x \ y. P \ x \ y$
 $\longleftrightarrow (x, y) \in R$

shows $\text{multeqp-code } P \ N \ M \longleftrightarrow (N, M) \in (\text{mult } R)^=$

<proof>

lemma *multeqp-code-iff-reflclp-multp*:

irreflp-on $(\text{set-mset } M \cap \text{set-mset } N) \ R \implies \text{transp } R \implies \text{multeqp-code } R \ M \ N$
 $\longleftrightarrow (\text{multp } R)^= \ M \ N$

<proof>

lemma *multeqp-code-eq-reflclp-multp*:

assumes *irreflp* *R* **and** *transp* *R*

shows $\text{multeqp-code } R = (\text{multp } R)^{==}$

<proof>

67.14.1 Monotonicity of multiset union

lemma *mult1-union*: $(B, D) \in \text{mult1 } r \implies (C + B, C + D) \in \text{mult1 } r$
 $\langle \text{proof} \rangle$

lemma *union-le-mono2*: $B < D \implies C + B < C + (D :: 'a :: \text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *union-le-mono1*: $B < D \implies B + C < D + (C :: 'a :: \text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *union-less-mono*:
fixes $A B C D :: 'a :: \text{preorder multiset}$
shows $A < C \implies B < D \implies A + B < C + D$
 $\langle \text{proof} \rangle$

instantiation *multiset* :: $(\text{preorder}) \text{ ordered-ab-semigroup-add}$
begin
instance
 $\langle \text{proof} \rangle$
end

67.14.2 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in \# XS \implies x \in \# \{\# y \# \} + XS$
and *multi-member-this*: $x \in \# \{\# x \# \} + XS$
and *multi-member-last*: $x \in \# \{\# x \# \}$
 $\langle \text{proof} \rangle$

definition *ms-strict* = *mult pair-less*

definition *ms-weak* = *ms-strict* \cup *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
 $\langle \text{proof} \rangle$

lemma *smsI*:
 $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z + B) \in \text{ms-strict}$
 $\langle \text{proof} \rangle$

lemma *wmsI*:
 $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee A = \{\#\} \wedge B = \{\#\}$
 $\implies (Z + A, Z + B) \in \text{ms-weak}$
 $\langle \text{proof} \rangle$

inductive *pw-leq*

where

pw-leq-empty: $\text{pw-leq } \{\#\} \{\#\}$
 $| \text{pw-leq-step}: \llbracket (x, y) \in \text{pair-leq}; \text{pw-leq } X Y \rrbracket \implies \text{pw-leq } (\{\#x\# \} + X) (\{\#y\# \} + Y)$

lemma *pw-leq-lstep*:

$(x, y) \in \text{pair-leq} \implies \text{pw-leq } \{ \#x\# \} \{ \#y\# \}$
 $\langle \text{proof} \rangle$

lemma *pw-leq-split*:

assumes *pw-leq* $X\ Y$
shows $\exists A\ B\ Z. X = A + Z \wedge Y = B + Z \wedge ((\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee (B = \{ \# \} \wedge A = \{ \# \}))$
 $\langle \text{proof} \rangle$

lemma

assumes *pwleq*: *pw-leq* $Z\ Z'$
shows *ms-strictI*: $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B) \in \text{ms-strict}$
and *ms-weakI1*: $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B) \in \text{ms-weak}$
and *ms-weakI2*: $(Z + \{ \# \}, Z' + \{ \# \}) \in \text{ms-weak}$
 $\langle \text{proof} \rangle$

lemma *empty-neutral*: $\{ \# \} + x = x\ x + \{ \# \} = x$

and *nonempty-plus*: $\{ \#\ x\ \# \} + rs \neq \{ \# \}$

and *nonempty-single*: $\{ \#\ x\ \# \} \neq \{ \# \}$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

67.15 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ \text{multiset})$

$\langle \text{proof} \rangle$

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ \text{multiset}))$

$\langle \text{proof} \rangle$

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ \text{multiset}))$

$\langle \text{proof} \rangle$

lemmas *union-ac* = *union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ \text{multiset})$

$\langle \text{proof} \rangle$

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ \text{multiset})$

$\langle \text{proof} \rangle$

lemma *multi-union-self-other-eq*: $(A::'a\ \text{multiset}) + X = A + Y \implies X = Y$

$\langle \text{proof} \rangle$

lemma *mset-subset-trans*: $(M::'a \text{ multiset}) \subset\# K \implies K \subset\# N \implies M \subset\# N$
 $\langle \text{proof} \rangle$

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
 $\langle \text{proof} \rangle$

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
 $\langle \text{proof} \rangle$

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
 $\langle \text{proof} \rangle$

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg M < (M::'a::\text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *mset-le-trans*: $K < M \implies M < N \implies K < (N::'a::\text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *mset-le-not-sym*: $M < N \implies \neg N < (M::'a::\text{preorder multiset})$
 $\langle \text{proof} \rangle$

lemma *mset-le-asy*: $M < N \implies (\neg P \implies N < (M::'a::\text{preorder multiset})) \implies P$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

67.16 Naive implementation using lists

code-datatype *mset*

lemma [code]: $\{\#\} = \text{mset } []$
 $\langle \text{proof} \rangle$

lemma [code]: $\text{add-mset } x (\text{mset } xs) = \text{mset } (x \# xs)$
 $\langle \text{proof} \rangle$

lemma [code]: $\text{Multiset.is-empty } (\text{mset } xs) \longleftrightarrow \text{List.null } xs$
 $\langle \text{proof} \rangle$

lemma *union-code* [code]: $\text{mset } xs + \text{mset } ys = \text{mset } (xs @ ys)$
 $\langle \text{proof} \rangle$

lemma [code]: $\text{image-mset } f \text{ (mset } xs) = \text{mset (map } f \text{ } xs)$
 ⟨proof⟩

lemma [code]: $\text{filter-mset } f \text{ (mset } xs) = \text{mset (filter } f \text{ } xs)$
 ⟨proof⟩

lemma [code]: $\text{mset } xs - \text{mset } ys = \text{mset (minus-list-mset } xs \text{ } ys)$
 ⟨proof⟩

lemma [code]:
 $\text{mset } xs \cap \# \text{mset } ys =$
 $\text{mset (snd (fold } (\lambda x \text{ (ys, zs).}$
 $\text{if } x \in \text{set } ys \text{ then (remove1 } x \text{ } ys, x \# \text{ } zs) \text{ else (ys, zs)) } xs \text{ (ys, []))})$
 ⟨proof⟩

lemma [code]:
 $\text{mset } xs \cup \# \text{mset } ys =$
 $\text{mset (case-prod append (fold } (\lambda x \text{ (ys, zs). (remove1 } x \text{ } ys, x \# \text{ } zs)) } xs \text{ (ys, []))})$
 ⟨proof⟩

declare *in-multiset-in-set* [code-unfold]

lemma [code]: $\text{count (mset } xs) \text{ } x = \text{fold } (\lambda y. \text{if } x = y \text{ then Suc else id}) \text{ } xs \text{ } 0$
 ⟨proof⟩

declare *set-mset-mset* [code]

declare *sorted-list-of-multiset-mset* [code]

lemma [code]: — not very efficient, but representation-ignorant!
 $\text{mset-set } A = \text{mset (sorted-list-of-set } A)$
 ⟨proof⟩

declare *size-mset* [code]

fun *subset-eq-mset-impl* :: 'a list \Rightarrow 'a list \Rightarrow bool option **where**
 $\text{subset-eq-mset-impl [] } ys = \text{Some (ys } \neq \text{ [])}$
 $| \text{subset-eq-mset-impl (Cons } x \text{ } xs) \text{ } ys = (\text{case List.extract ((=) } x) \text{ } ys \text{ of}$
 $\text{None } \Rightarrow \text{None}$
 $| \text{Some (ys1, -, ys2)} \Rightarrow \text{subset-eq-mset-impl } xs \text{ (ys1 @ ys2))}$

lemma *subset-eq-mset-impl*: $(\text{subset-eq-mset-impl } xs \text{ } ys = \text{None} \longleftrightarrow \neg \text{mset } xs$
 $\subseteq \# \text{mset } ys) \wedge$
 $(\text{subset-eq-mset-impl } xs \text{ } ys = \text{Some True} \longleftrightarrow \text{mset } xs \subseteq \# \text{mset } ys) \wedge$
 $(\text{subset-eq-mset-impl } xs \text{ } ys = \text{Some False} \longrightarrow \text{mset } xs = \text{mset } ys)$
 ⟨proof⟩

lemma [code]: $\text{mset } xs \subseteq \# \text{mset } ys \longleftrightarrow \text{subset-eq-mset-impl } xs \text{ } ys \neq \text{None}$
 ⟨proof⟩

lemma [code]: $\text{mset } xs \subset\# \text{ mset } ys \longleftrightarrow \text{subset-eq-mset-impl } xs \text{ } ys = \text{Some True}$
 ⟨proof⟩

instantiation *multiset* :: (equal) equal
begin

definition

[code del]: $\text{HOL.equal } A \text{ } (B :: 'a \text{ multiset}) \longleftrightarrow A = B$

lemma [code]: $\text{HOL.equal } (\text{mset } xs) \text{ } (\text{mset } ys) \longleftrightarrow \text{subset-eq-mset-impl } xs \text{ } ys = \text{Some False}$
 ⟨proof⟩

instance
 ⟨proof⟩

end

declare *sum-mset-sum-list* [code]

lemma [code]: $\text{prod-mset } (\text{mset } xs) = \text{fold times } xs \text{ } 1$
 ⟨proof⟩

Exercise for the casual reader: add implementations for (\leq) and $(<)$ (multiset order).

Quickcheck generators

context

includes *term-syntax*

begin

definition

$\text{msetify} :: 'a::\text{typerep list} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term})$
 $\Rightarrow 'a \text{ multiset} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term})$ **where**

[code-unfold]: $\text{msetify } xs = \text{Code-Evaluation.valtermify mset } \{\cdot\} \text{ } xs$

end

instantiation *multiset* :: (random) random
begin

context

includes *state-combinator-syntax*

begin

definition

$\text{Quickcheck-Random.random } i = \text{Quickcheck-Random.random } i \circ \rightarrow (\lambda xs. \text{Pair } (\text{msetify } xs))$

instance ⟨proof⟩

end

end

instantiation *multiset* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-multiset* :: ('a multiset × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option

where

full-exhaustive-multiset *f* *i* = *Quickcheck-Exhaustive.full-exhaustive* (λ*xs*. *f* (*msetify* *xs*)) *i*

instance ⟨*proof*⟩

end

hide-const (**open**) *msetify*

67.17 BNF setup

definition *rel-mset* **where**

rel-mset *R* *X* *Y* ⇔ (∃ *xs* *ys*. *mset xs* = *X* ∧ *mset ys* = *Y* ∧ *list-all2 R xs ys*)

lemma *mset-zip-take-Cons-drop-twice*:

assumes *length xs* = *length ys* *j* ≤ *length xs*

shows *mset* (*zip* (*take j xs* @ *x* # *drop j xs*) (*take j ys* @ *y* # *drop j ys*)) =
add-mset (*x*, *y*) (*mset* (*zip xs ys*))

⟨*proof*⟩

lemma *ex-mset-zip-left*:

assumes *length xs* = *length ys* *mset xs'* = *mset xs*

shows ∃ *ys'*. *length ys'* = *length xs'* ∧ *mset* (*zip xs' ys'*) = *mset* (*zip xs ys*)

⟨*proof*⟩

lemma *list-all2-reorder-left-invariance*:

assumes *rel*: *list-all2 R xs ys* **and** *ms-x*: *mset xs'* = *mset xs*

shows ∃ *ys'*. *list-all2 R xs' ys'* ∧ *mset ys'* = *mset ys*

⟨*proof*⟩

lemma *ex-mset*: ∃ *xs*. *mset xs* = *X*

⟨*proof*⟩

inductive *pred-mset* :: ('a ⇒ bool) ⇒ 'a multiset ⇒ bool

where

pred-mset *P* {#}

| ⟦*P* *a*; *pred-mset P M*⟧ ⇒ *pred-mset P* (*add-mset a M*)

lemma *pred-mset-iff*: — TODO: alias for *Multiset.Ball*
 $\langle \text{pred-mset } P \ M \longleftrightarrow \text{Multiset.Ball } M \ P \rangle$ (**is** $\langle ?P \longleftrightarrow ?Q \rangle$)
 $\langle \text{proof} \rangle$

bnf *'a multiset*
map: *image-mset*
sets: *set-mset*
bd: *natLeq*
wits: $\{\#\}$
rel: *rel-mset*
pred: *pred-mset*
 $\langle \text{proof} \rangle$

inductive *rel-mset'* :: $\langle ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'b \text{ multiset} \Rightarrow \text{bool} \rangle$
where

Zero[*intro*]: *rel-mset'* *R* $\{\#\}$ $\{\#\}$
| *Plus*[*intro*]: $\llbracket R \ a \ b; \text{rel-mset}' \ R \ M \ N \rrbracket \Longrightarrow \text{rel-mset}' \ R \ (\text{add-mset } a \ M) \ (\text{add-mset } b \ N)$

lemma *rel-mset-Zero*: *rel-mset* *R* $\{\#\}$ $\{\#\}$
 $\langle \text{proof} \rangle$

declare *multiset.count*[*simp*]
declare *count-Abs-multiset*[*simp*]
declare *multiset.count-inverse*[*simp*]

lemma *rel-mset-Plus*:
assumes *ab*: *R* *a* *b*
and *MN*: *rel-mset* *R* *M* *N*
shows *rel-mset* *R* (*add-mset* *a* *M*) (*add-mset* *b* *N*)
 $\langle \text{proof} \rangle$

lemma *rel-mset'-imp-rel-mset*: *rel-mset'* *R* *M* *N* \Longrightarrow *rel-mset* *R* *M* *N*
 $\langle \text{proof} \rangle$

lemma *rel-mset-size*: *rel-mset* *R* *M* *N* \Longrightarrow *size* *M* = *size* *N*
 $\langle \text{proof} \rangle$

lemma *rel-mset-Zero-iff* [*simp*]:
shows *rel-mset* *rel* $\{\#\}$ *Y* \longleftrightarrow *Y* = $\{\#\}$ **and** *rel-mset* *rel* *X* $\{\#\}$ \longleftrightarrow *X* = $\{\#\}$
 $\langle \text{proof} \rangle$

lemma *multiset-induct2*[*case-names empty addL addR*]:
assumes *empty*: *P* $\{\#\}$ $\{\#\}$
and *addL*: $\bigwedge a \ M \ N. \ P \ M \ N \Longrightarrow P \ (\text{add-mset } a \ M) \ N$
and *addR*: $\bigwedge a \ M \ N. \ P \ M \ N \Longrightarrow P \ M \ (\text{add-mset } a \ N)$
shows *P* *M* *N*
 $\langle \text{proof} \rangle$

lemma *multiset-induct2-size*[*consumes 1, case-names empty add*]:
assumes *c: size M = size N*
and *empty: P {#} {#}*
and *add: $\bigwedge a b M N a b. P M N \implies P (\text{add-mset } a M) (\text{add-mset } b N)$*
shows *P M N*
 $\langle \text{proof} \rangle$

lemma *msed-map-invL*:
assumes *image-mset f (add-mset a M) = N*
shows $\exists N1. N = \text{add-mset } (f a) N1 \wedge \text{image-mset } f M = N1$
 $\langle \text{proof} \rangle$

lemma *msed-map-invR*:
assumes *image-mset f M = add-mset b N*
shows $\exists M1 a. M = \text{add-mset } a M1 \wedge f a = b \wedge \text{image-mset } f M1 = N$
 $\langle \text{proof} \rangle$

lemma *msed-rel-invL*:
assumes *rel-mset R (add-mset a M) N*
shows $\exists N1 b. N = \text{add-mset } b N1 \wedge R a b \wedge \text{rel-mset } R M N1$
 $\langle \text{proof} \rangle$

lemma *msed-rel-invR*:
assumes *rel-mset R M (add-mset b N)*
shows $\exists M1 a. M = \text{add-mset } a M1 \wedge R a b \wedge \text{rel-mset } R M1 N$
 $\langle \text{proof} \rangle$

lemma *rel-mset-imp-rel-mset'*:
assumes *rel-mset R M N*
shows *rel-mset' R M N*
 $\langle \text{proof} \rangle$

lemma *rel-mset-rel-mset'*: *rel-mset R M N = rel-mset' R M N*
 $\langle \text{proof} \rangle$

The main end product for *rel-mset*: inductive characterization:

lemmas *rel-mset-induct*[*case-names empty add, induct pred: rel-mset*] =
rel-mset'.induct[*unfolded rel-mset-rel-mset'*[*symmetric*]]

67.18 Size setup

lemma *size-multiset-o-map*: *size-multiset g \circ image-mset f = size-multiset (g \circ f)*
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

67.19 Lemmas about Size

lemma *size-mset-SucE*: *size A = Suc n \implies ($\bigwedge a B. A = \{ \# a \# \} + B \implies \text{size } B = n \implies P$) $\implies P$*

<proof>

lemma *size-Suc-Diff1*: $x \in\# M \implies \text{Suc} (\text{size} (M - \{\#x\})) = \text{size} M$
<proof>

lemma *size-Diff-singleton*: $x \in\# M \implies \text{size} (M - \{\#x\}) = \text{size} M - 1$
<proof>

lemma *size-Diff-singleton-if*: $\text{size} (A - \{\#x\}) = (\text{if } x \in\# A \text{ then } \text{size} A - 1 \text{ else } \text{size} A)$
<proof>

lemma *size-Un-Int*: $\text{size} A + \text{size} B = \text{size} (A \cup\# B) + \text{size} (A \cap\# B)$
<proof>

lemma *size-Un-disjoint*: $A \cap\# B = \{\#\} \implies \text{size} (A \cup\# B) = \text{size} A + \text{size} B$
<proof>

lemma *size-Diff-subset-Int*: $\text{size} (M - M') = \text{size} M - \text{size} (M \cap\# M')$
<proof>

lemma *diff-size-le-size-Diff*: $\text{size} (M :: \text{- multiset}) - \text{size} M' \leq \text{size} (M - M')$
<proof>

lemma *size-Diff1-less*: $x \in\# M \implies \text{size} (M - \{\#x\}) < \text{size} M$
<proof>

lemma *size-Diff2-less*: $x \in\# M \implies y \in\# M \implies \text{size} (M - \{\#x\} - \{\#y\}) < \text{size} M$
<proof>

lemma *size-Diff1-le*: $\text{size} (M - \{\#x\}) \leq \text{size} M$
<proof>

lemma *size-psubset*: $M \subseteq\# M' \implies \text{size} M < \text{size} M' \implies M \subset\# M'$
<proof>

lifting-update *multiset.lifting*

lifting-forget *multiset.lifting*

hide-const (**open**) *wcount*

67.20 The set of multisets of a given size

The following operator gives the set of all multisets consisting of n elements drawn from the set A . In other words: all the different ways to put n unlabelled balls into the labelled bins A .

definition *multisets-of-size* :: 'a set \Rightarrow nat \Rightarrow 'a multiset set **where**
multisets-of-size $A\ n = \{X. \text{set-mset } X \subseteq A \wedge \text{size } X = n\}$

lemma

assumes $X \in \text{multisets-of-size } A \ n$
shows $\text{multisets-of-size-subset: set-mset } X \subseteq A$
and $\text{multisets-of-size-size: size } X = n$
 $\langle \text{proof} \rangle$

lemma *multisets-of-size-mono:*

assumes $A \subseteq B$
shows $\text{multisets-of-size } A \ n \subseteq \text{multisets-of-size } B \ n$
 $\langle \text{proof} \rangle$

lemma *multisets-of-size-0 [simp]: multisets-of-size A 0 = {{#}}*
 $\langle \text{proof} \rangle$

lemma *multisets-of-size-empty [simp]: $n > 0 \implies \text{multisets-of-size } \{\} \ n = \{\}$*
 $\langle \text{proof} \rangle$

lemma *count-le-size: count X x ≤ size X*
 $\langle \text{proof} \rangle$

lemma *bij-betw-multisets-of-size-insert:*

assumes $a \notin A$
shows $\text{bij-betw } (\lambda(m,X). X + \text{replicate-mset } m \ a)$
 $(\text{SIGMA } m:\{0..n\}. \text{multisets-of-size } A \ (n - m)) \ (\text{multisets-of-size } (\text{insert } a \ A) \ n)$
 $\langle \text{proof} \rangle$

lemma *multisets-of-size-insert:*

assumes $a \notin A$
shows $\text{multisets-of-size } (\text{insert } a \ A) \ n =$
 $(\bigcup_{m \leq n. (\lambda X. X + \text{replicate-mset } m \ a)) \ ' \text{multisets-of-size } A \ (n - m))$
 $\langle \text{proof} \rangle$

primrec *multisets-of-size-list :: 'a list \Rightarrow nat \Rightarrow 'a list list where*

multisets-of-size-list [] n = (if n = 0 then [[]] else [])
 $| \text{multisets-of-size-list } (x \# xs) \ n =$
 $[\text{replicate } m \ x \ @ \ ys \ . \ m \leftarrow [0..<n+1], \ ys \leftarrow \text{multisets-of-size-list } xs \ (n - m)]$

lemma *multisets-of-size-list-correct:*

assumes *distinct xs*
shows $\text{mset } ' \text{set } (\text{multisets-of-size-list } xs \ n) = \text{multisets-of-size } (\text{set } xs) \ n$
 $\langle \text{proof} \rangle$

lemma *multisets-of-size-code [code]:*

$\text{multisets-of-size } (\text{set } xs) \ n = \text{set } (\text{map mset } (\text{multisets-of-size-list } (\text{remdups } xs) \ n))$
 $\langle \text{proof} \rangle$

lemma *finite-multisets-of-size* [intro]:

assumes *finite A*

shows *finite (multisets-of-size A n)*

<proof>

lemma *card-multisets-of-size*:

assumes *finite A*

shows *card (multisets-of-size A n) = (card A + n - 1) choose n*

<proof>

end

68 More Theorems about the Multiset Order

theory *Multiset-Order*

imports *Multiset*

begin

68.1 Alternative Characterizations

68.1.1 The Dershowitz–Manna Ordering

definition *multp_{DM}* where

multp_{DM} r M N \longleftrightarrow

($\exists X Y. X \neq \{\#\} \wedge X \subseteq \# N \wedge M = (N - X) + Y \wedge (\forall k. k \in \# Y \longrightarrow (\exists a. a \in \# X \wedge r k a))$)

lemma *multp_{DM}-imp-multp*:

multp_{DM} r M N \implies multp r M N

<proof>

68.1.2 The Huet–Oppen Ordering

definition *multp_{HO}* where

multp_{HO} r M N $\longleftrightarrow M \neq N \wedge (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x. r y x \wedge \text{count } M x < \text{count } N x))$

lemma *multp-imp-multp_{HO}*:

assumes *asyp r and transp r*

shows *multp r M N \implies multp_{HO} r M N*

<proof>

lemma *multp_{HO}-imp-multp_{DM}*: *multp_{HO} r M N \implies multp_{DM} r M N*

<proof>

lemma *multp-eq-multp_{DM}*: *asyp r \implies transp r \implies multp r = multp_{DM} r*

<proof>

lemma *multp-eq-multp_{HO}*: *asyp r \implies transp r \implies multp r = multp_{HO} r*

$\langle \text{proof} \rangle$

lemma *multp_{DM}-plus-plusI[simp]*:
assumes *multp_{DM} R M1 M2*
shows *multp_{DM} R (M + M1) (M + M2)*
 $\langle \text{proof} \rangle$

lemma *multp_{HO}-plus-plus[simp]*: *multp_{HO} R (M + M1) (M + M2) \longleftrightarrow multp_{HO} R M1 M2*
 $\langle \text{proof} \rangle$

lemma *strict-subset-implies-multp_{DM}*: *A $\subset\#$ B \implies multp_{DM} r A B*
 $\langle \text{proof} \rangle$

lemma *strict-subset-implies-multp_{HO}*: *A $\subset\#$ B \implies multp_{HO} r A B*
 $\langle \text{proof} \rangle$

lemma *multp_{HO}-implies-one-step-strong*:
assumes *multp_{HO} R A B*
defines *J \equiv B - A and K \equiv A - B*
shows *J $\neq \{\#\}$ and $\forall k \in\# K. \exists x \in\# J. R k x$*
 $\langle \text{proof} \rangle$

lemma *multp_{HO}-minus-inter-minus-inter-iff*:
fixes *M1 M2 :: - multiset*
shows *multp_{HO} R (M1 - M2) (M2 - M1) \longleftrightarrow multp_{HO} R M1 M2*
 $\langle \text{proof} \rangle$

lemma *multp_{HO}-iff-set-mset-less_{HO}-set-mset*:
*multp_{HO} R M1 M2 \longleftrightarrow (set-mset (M1 - M2) \neq set-mset (M2 - M1) \wedge
 $(\forall y \in\# M1 - M2. (\exists x \in\# M2 - M1. R y x)))$*
 $\langle \text{proof} \rangle$

68.1.3 Monotonicity

lemma *multp_{DM}-mono-strong*:
*multp_{DM} R M1 M2 \implies ($\bigwedge x y. x \in\# M1 \implies y \in\# M2 \implies R x y \implies S x y$)
 \implies multp_{DM} S M1 M2*
 $\langle \text{proof} \rangle$

lemma *multp_{HO}-mono-strong*:
*multp_{HO} R M1 M2 \implies ($\bigwedge x y. x \in\# M1 \implies y \in\# M2 \implies R x y \implies S x y$)
 \implies multp_{HO} S M1 M2*
 $\langle \text{proof} \rangle$

68.1.4 Properties of Orders

Asymmetry The following lemma is a negative result stating that asymmetry of an arbitrary binary relation cannot be simply lifted to *multp_{HO}*.

It suffices to have four distinct values to build a counterexample.

lemma *asympt-not-liftable-to-multp_{HO}*:

fixes $a\ b\ c\ d :: 'a$

assumes *distinct* $[a, b, c, d]$

shows $\neg (\forall (R :: 'a \Rightarrow 'a \Rightarrow \text{bool}). \text{asympt } R \longrightarrow \text{asympt } (\text{multp}_{HO} R))$

<proof>

However, if the binary relation is both asymmetric and transitive, then *multp_{HO}* is also asymmetric.

lemma *asympt-on-multp_{HO}*:

assumes *asympt-on* $A\ R$ **and** *transp-on* $A\ R$ **and**

$B\text{-sub-}A: \bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$

shows *asympt-on* $B\ (\text{multp}_{HO} R)$

<proof>

lemma *asympt-multp_{HO}*:

assumes *asympt* R **and** *transp* R

shows *asympt* $(\text{multp}_{HO} R)$

<proof>

Irreflexivity **lemma** *irreflp-on-multp_{HO}[simp]*: *irreflp-on* $B\ (\text{multp}_{HO} R)$

<proof>

Transitivity **lemma** *transp-on-multp_{HO}*:

assumes *asympt-on* $A\ R$ **and** *transp-on* $A\ R$ **and** $B\text{-sub-}A: \bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$

shows *transp-on* $B\ (\text{multp}_{HO} R)$

<proof>

lemma *transp-multp_{HO}*:

assumes *asympt* R **and** *transp* R

shows *transp* $(\text{multp}_{HO} R)$

<proof>

Totality **lemma** *totalp-on-multp_{DM}*:

totalp-on $A\ R \implies (\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A) \implies \text{totalp-on } B\ (\text{multp}_{DM} R)$

<proof>

lemma *totalp-multp_{DM}*: *totalp* $R \implies \text{totalp } (\text{multp}_{DM} R)$

<proof>

lemma *totalp-on-multp_{HO}*:

totalp-on $A\ R \implies (\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A) \implies \text{totalp-on } B\ (\text{multp}_{HO} R)$

<proof>

lemma *totalp-multp_{HO}*: *totalp* $R \implies \text{totalp } (\text{multp}_{HO} R)$

<proof>

Type Classes *context preorder*
begin

lemma *order-mult: class.order*
 $(\lambda M N. (M, N) \in \text{mult } \{(x, y). x < y\} \vee M = N)$
 $(\lambda M N. (M, N) \in \text{mult } \{(x, y). x < y\})$
(is class.order ?le ?less)
 $\langle \text{proof} \rangle$

The Dershowitz–Manna ordering:

definition *less-multiset_{DM}* **where**
 $\text{less-multiset}_{DM} M N \longleftrightarrow$
 $(\exists X Y. X \neq \{\#\} \wedge X \subseteq\# N \wedge M = (N - X) + Y \wedge (\forall k. k \in\# Y \longrightarrow (\exists a.$
 $a \in\# X \wedge k < a)))$

The Huet–Oppen ordering:

definition *less-multiset_{HO}* **where**
 $\text{less-multiset}_{HO} M N \longleftrightarrow M \neq N \wedge (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x. y <$
 $x \wedge \text{count } M x < \text{count } N x))$

lemma *mult-imp-less-multiset_{HO}*:
 $(M, N) \in \text{mult } \{(x, y). x < y\} \implies \text{less-multiset}_{HO} M N$
 $\langle \text{proof} \rangle$

lemma *less-multiset_{DM}-imp-mult*:
 $\text{less-multiset}_{DM} M N \implies (M, N) \in \text{mult } \{(x, y). x < y\}$
 $\langle \text{proof} \rangle$

lemma *less-multiset_{HO}-imp-less-multiset_{DM}*: $\text{less-multiset}_{HO} M N \implies \text{less-multiset}_{DM} M N$
 $\langle \text{proof} \rangle$

lemma *mult-less-multiset_{DM}*: $(M, N) \in \text{mult } \{(x, y). x < y\} \longleftrightarrow \text{less-multiset}_{DM} M N$
 $\langle \text{proof} \rangle$

lemma *mult-less-multiset_{HO}*: $(M, N) \in \text{mult } \{(x, y). x < y\} \longleftrightarrow \text{less-multiset}_{HO} M N$
 $\langle \text{proof} \rangle$

lemmas $\text{mult}_{DM} = \text{mult-less-multiset}_{DM}[\text{unfolded less-multiset}_{DM}\text{-def}]$
lemmas $\text{mult}_{HO} = \text{mult-less-multiset}_{HO}[\text{unfolded less-multiset}_{HO}\text{-def}]$

end

lemma *less-multiset-less-multiset_{HO}*: $M < N \longleftrightarrow \text{less-multiset}_{HO} M N$
 $\langle \text{proof} \rangle$

lemma *less-multiset_{DM}*:

$M < N \longleftrightarrow (\exists X Y. X \neq \{\#\} \wedge X \subseteq\# N \wedge M = N - X + Y \wedge (\forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge k < a)))$
 <proof>

lemma *less-multiset_{HO}*:

$M < N \longleftrightarrow M \neq N \wedge (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x>y. \text{count } M x < \text{count } N x))$
 <proof>

lemma *subset-eq-imp-le-multiset*:

shows $M \subseteq\# N \Longrightarrow M \leq N$
 <proof>

lemma *le-multiset-right-total*: $M < \text{add-mset } x M$

<proof>

lemma *less-eq-multiset-empty-left[simp]*: $\{\#\} \leq M$

<proof>

lemma *ex-gt-imp-less-multiset*: $(\exists y. y \in\# N \wedge (\forall x. x \in\# M \longrightarrow x < y)) \Longrightarrow M < N$

<proof>

lemma *less-eq-multiset-empty-right[simp]*: $M \neq \{\#\} \Longrightarrow \neg M \leq \{\#\}$

<proof>

lemma *le-multiset-empty-left[simp]*: $M \neq \{\#\} \Longrightarrow \{\#\} < M$

<proof>

lemma *le-multiset-empty-right[simp]*: $\neg M < \{\#\}$

<proof>

lemma *union-le-diff-plus*: $P \subseteq\# M \Longrightarrow N < P \Longrightarrow M - P + N < M$

<proof>

instantiation *multiset* :: (preorder) ordered-ab-semigroup-monoid-add-imp-le
begin

lemma *less-eq-multiset_{HO}*:

$M \leq N \longleftrightarrow (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x. y < x \wedge \text{count } M x < \text{count } N x))$
 <proof>

instance <proof>

lemma**fixes** $M\ N :: 'a\ multiset$ **shows** *less-eq-multiset-plus-left*: $N \leq (M + N)$ **and** *less-eq-multiset-plus-right*: $M \leq (M + N)$ $\langle proof \rangle$ **lemma****fixes** $M\ N :: 'a\ multiset$ **shows** *le-multiset-plus-left-nonempty*: $M \neq \{\#\} \implies N < M + N$ **and** *le-multiset-plus-right-nonempty*: $N \neq \{\#\} \implies M < M + N$ $\langle proof \rangle$ **end****lemma** *all-lt-Max-imp-lt-mset*: $N \neq \{\#\} \implies (\forall a \in \# M. a < \text{Max}(\text{set-mset } N))$ $\implies M < N$ $\langle proof \rangle$ **lemma** *lt-imp-ex-count-lt*: $M < N \implies \exists y. \text{count } M\ y < \text{count } N\ y$ $\langle proof \rangle$ **lemma** *subset-imp-less-mset*: $A \subset \# B \implies A < B$ $\langle proof \rangle$ **lemma** *image-mset-strict-mono*:**assumes** *mono-f*: $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f\ x < f\ y$ **and** *less*: $M < N$ **shows** *image-mset f M < image-mset f N* $\langle proof \rangle$ **lemma** *image-mset-mono*:**assumes** *mono-f*: $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f\ x < f\ y$ **and** *less*: $M \leq N$ **shows** *image-mset f M ≤ image-mset f N* $\langle proof \rangle$ **lemma** *mset-lt-single-right-iff[simp]*: $M < \{\#y\# \} \longleftrightarrow (\forall x \in \# M. x < y)$ **for** y $:: 'a::\text{linorder}$ $\langle proof \rangle$ **lemma** *mset-le-single-right-iff[simp]*: $M \leq \{\#y\# \} \longleftrightarrow M = \{\#y\# \} \vee (\forall x \in \# M. x < y)$ **for** $y :: 'a::\text{linorder}$ $\langle proof \rangle$

68.1.5 Simplifications

lemma *multp_{HO}-repeat-mset-repeat-mset[simp]*:**assumes** $n \neq 0$ **shows** $\text{multp}_{HO}\ R\ (\text{repeat-mset } n\ A)\ (\text{repeat-mset } n\ B) \longleftrightarrow \text{multp}_{HO}\ R\ A\ B$

$\langle proof \rangle$

lemma *multp_{HO}-double-double[simp]*: $multp_{HO} \ R \ (A + A) \ (B + B) \longleftrightarrow multp_{HO} \ R \ A \ B$
 $\langle proof \rangle$

68.2 Simprocs

lemma *mset-le-add-iff1*:
 $j \leq (i::nat) \implies (repeat_mset \ i \ u + m \leq repeat_mset \ j \ u + n) = (repeat_mset \ (i-j) \ u + m \leq n)$
 $\langle proof \rangle$

lemma *mset-le-add-iff2*:
 $i \leq (j::nat) \implies (repeat_mset \ i \ u + m \leq repeat_mset \ j \ u + n) = (m \leq repeat_mset \ (j-i) \ u + n)$
 $\langle proof \rangle$

$\langle ML \rangle$

68.3 Additional facts and instantiations

lemma *ex-gt-count-imp-le-multiset*:
 $(\forall y :: 'a :: order. y \in \# \ M + N \longrightarrow y \leq x) \implies count \ M \ x < count \ N \ x \implies M < N$
 $\langle proof \rangle$

lemma *mset-lt-single-iff[iff]*: $\{\#x\# \} < \{\#y\# \} \longleftrightarrow x < y$
 $\langle proof \rangle$

lemma *mset-le-single-iff[iff]*: $\{\#x\# \} \leq \{\#y\# \} \longleftrightarrow x \leq y$ **for** $x \ y :: 'a :: order$
 $\langle proof \rangle$

instance *multiset* :: (*linorder*) *linordered-cancel-ab-semigroup-add*
 $\langle proof \rangle$

lemma *less-eq-multiset-total*: $\neg \ M \leq N \implies N \leq M$ **for** $M \ N :: 'a :: linorder$
multiset
 $\langle proof \rangle$

instantiation *multiset* :: (*wellorder*) *wellorder*
begin

lemma *wf-less-multiset*: $wf \ \{(M :: 'a \ multiset, N). M < N\}$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

instantiation *multiset* :: (*preorder*) *order-bot*
begin

definition *bot-multiset* :: 'a *multiset* **where** *bot-multiset* = {#}

instance *<proof>*

end

instance *multiset* :: (*preorder*) *no-top*
<proof>

instance *multiset* :: (*preorder*) *ordered-cancel-comm-monoid-add*
<proof>

instantiation *multiset* :: (*linorder*) *distrib-lattice*
begin

definition *inf-multiset* :: 'a *multiset* \Rightarrow 'a *multiset* \Rightarrow 'a *multiset* **where**
inf-multiset *A B* = (if *A* < *B* then *A* else *B*)

definition *sup-multiset* :: 'a *multiset* \Rightarrow 'a *multiset* \Rightarrow 'a *multiset* **where**
sup-multiset *A B* = (if *B* > *A* then *B* else *A*)

instance
<proof>

end

lemma *add-mset-lt-left-lt*: $a < b \implies \text{add-mset } a \ A < \text{add-mset } b \ A$
<proof>

lemma *add-mset-le-left-le*: $a \leq b \implies \text{add-mset } a \ A \leq \text{add-mset } b \ A$ **for** *a* :: 'a ::
linorder
<proof>

lemma *add-mset-lt-right-lt*: $A < B \implies \text{add-mset } a \ A < \text{add-mset } a \ B$
<proof>

lemma *add-mset-le-right-le*: $A \leq B \implies \text{add-mset } a \ A \leq \text{add-mset } a \ B$
<proof>

lemma *add-mset-lt-lt-lt*:
assumes *a-lt-b*: $a < b$ **and** *A-le-B*: $A < B$
shows $\text{add-mset } a \ A < \text{add-mset } b \ B$
<proof>

lemma *add-mset-lt-lt-le*: $a < b \implies A \leq B \implies \text{add-mset } a \ A < \text{add-mset } b \ B$

$\langle proof \rangle$

lemma *add-mset-lt-le-lt*: $a \leq b \implies A < B \implies \text{add-mset } a \ A < \text{add-mset } b \ B$ **for**
 $a :: 'a :: \text{linorder}$
 $\langle proof \rangle$

lemma *add-mset-le-le-le*:
fixes $a :: 'a :: \text{linorder}$
assumes $a\text{-le-}b$: $a \leq b$ **and** $A\text{-le-}B$: $A \leq B$
shows $\text{add-mset } a \ A \leq \text{add-mset } b \ B$
 $\langle proof \rangle$

lemma *Max-lt-imp-lt-mset*:
assumes $n\text{-nemp}$: $N \neq \{\#\}$ **and** max : $\text{Max-mset } M < \text{Max-mset } N$ (**is** $?max\text{-}M < ?max\text{-}N$)
shows $M < N$
 $\langle proof \rangle$

end

69 Fixed Length Lists

theory *NList*
imports *Main*
begin

definition *nlists* :: $\text{nat} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ list set}$
where $nlists \ n \ A = \{xs. \text{size } xs = n \wedge \text{set } xs \subseteq A\}$

lemma *nlistsI*: $\llbracket \text{size } xs = n; \text{set } xs \subseteq A \rrbracket \implies xs \in nlists \ n \ A$
 $\langle proof \rangle$

These [simp] attributes are double-edged. Many proofs in Jinja rely on it but they can degrade performance.

lemma *nlistsE-length* [simp]: $xs \in nlists \ n \ A \implies \text{size } xs = n$
 $\langle proof \rangle$

lemma *in-nlists-UNIV*: $xs \in nlists \ k \ UNIV \longleftrightarrow \text{length } xs = k$
 $\langle proof \rangle$

lemma *less-lengthI*: $\llbracket xs \in nlists \ n \ A; p < n \rrbracket \implies p < \text{size } xs$
 $\langle proof \rangle$

lemma *nlistsE-set*[simp]: $xs \in nlists \ n \ A \implies \text{set } xs \subseteq A$
 $\langle proof \rangle$

lemma *nlists-mono*:
assumes $A \subseteq B$ **shows** $nlists \ n \ A \subseteq nlists \ n \ B$
 $\langle proof \rangle$

lemma *nlists-singleton*: $nlists\ n\ \{a\} = \{replicate\ n\ a\}$
 $\langle proof \rangle$

lemma *nlists-n-0 [simp]*: $nlists\ 0\ A = \{\}\}$
 $\langle proof \rangle$

lemma *in-nlists-Suc-iff*: $(xs \in nlists\ (Suc\ n)\ A) = (\exists y \in A. \exists ys \in nlists\ n\ A. xs = y \# ys)$
 $\langle proof \rangle$

lemma *Cons-in-nlists-Suc [iff]*: $(x \# xs \in nlists\ (Suc\ n)\ A) \longleftrightarrow (x \in A \wedge xs \in nlists\ n\ A)$
 $\langle proof \rangle$

lemma *nlists-Suc*: $nlists\ (Suc\ n)\ A = (\bigcup a \in A. (\#)\ a\ ' nlists\ n\ A)$
 $\langle proof \rangle$

lemma *nlists-not-empty*: $A \neq \{\} \implies \exists xs. xs \in nlists\ n\ A$
 $\langle proof \rangle$

lemma *nlistsE-nth-in*: $\llbracket xs \in nlists\ n\ A; i < n \rrbracket \implies xs!i \in A$
 $\langle proof \rangle$

lemma *nlists-Cons-Suc [elim!]*:
 $l \# xs \in nlists\ n\ A \implies (\bigwedge n'. n = Suc\ n' \implies l \in A \implies xs \in nlists\ n'\ A \implies P) \implies P$
 $\langle proof \rangle$

lemma *nlists-appendE [elim!]*:
 $a @ b \in nlists\ n\ A \implies (\bigwedge n1\ n2. n = n1 + n2 \implies a \in nlists\ n1\ A \implies b \in nlists\ n2\ A \implies P) \implies P$
 $\langle proof \rangle$

lemma *nlists-update-in-list [simp, intro!]*:
 $\llbracket xs \in nlists\ n\ A; x \in A \rrbracket \implies xs[i := x] \in nlists\ n\ A$
 $\langle proof \rangle$

lemma *nlists-appendI [intro?]*:
 $\llbracket a \in nlists\ n\ A; b \in nlists\ m\ A \rrbracket \implies a @ b \in nlists\ (n+m)\ A$
 $\langle proof \rangle$

lemma *nlists-append*:
 $xs @ ys \in nlists\ k\ A \longleftrightarrow$
 $k = length(xs @ ys) \wedge xs \in nlists\ (length\ xs)\ A \wedge ys \in nlists\ (length\ ys)\ A$
 $\langle proof \rangle$

lemma *nlists-map [simp]*: $(map\ f\ xs \in nlists\ (size\ xs)\ A) = (f\ ' set\ xs \subseteq A)$

<proof>

lemma *nlists-replicateI* [intro]: $x \in A \implies \text{replicate } n \ x \in \text{nlists } n \ A$
<proof>

Link to an executable version on lists in List.

lemma *nlists-set*[code]: $\text{nlists } n \ (\text{set } xs) = \text{set}(\text{List.n-lists } n \ xs)$
<proof>

end

70 Non-negative, non-positive integers and reals

theory *Nonpos-Ints*
imports *Complex-Main*
begin

70.1 Non-positive integers

The set of non-positive integers on a ring. (in analogy to the set of non-negative integers \mathbb{N}) This is useful e.g. for the Gamma function.

definition *nonpos-Ints* ($\langle \mathbb{Z}_{\leq 0} \rangle$) **where** $\mathbb{Z}_{\leq 0} = \{\text{of-int } n \mid n. n \leq 0\}$

lemma *zero-in-nonpos-Ints* [simp,intro]: $0 \in \mathbb{Z}_{\leq 0}$
<proof>

lemma *neg-one-in-nonpos-Ints* [simp,intro]: $-1 \in \mathbb{Z}_{\leq 0}$
<proof>

lemma *neg-numeral-in-nonpos-Ints* [simp,intro]: $-\text{numeral } n \in \mathbb{Z}_{\leq 0}$
<proof>

lemma *one-notin-nonpos-Ints* [simp]: $(1 :: 'a :: \text{ring-char-0}) \notin \mathbb{Z}_{\leq 0}$
<proof>

lemma *numeral-notin-nonpos-Ints* [simp]: $(\text{numeral } n :: 'a :: \text{ring-char-0}) \notin \mathbb{Z}_{\leq 0}$
<proof>

lemma *minus-of-nat-in-nonpos-Ints* [simp, intro]: $-\text{of-nat } n \in \mathbb{Z}_{\leq 0}$
<proof>

lemma *of-nat-in-nonpos-Ints-iff*: $(\text{of-nat } n :: 'a :: \{\text{ring-1, ring-char-0}\}) \in \mathbb{Z}_{\leq 0} \iff n = 0$
<proof>

lemma *nonpos-Ints-of-int*: $n \leq 0 \implies \text{of-int } n \in \mathbb{Z}_{\leq 0}$
<proof>

lemma *nonpos-IntsI*:

$x \in \mathbb{Z} \implies x \leq 0 \implies (x :: 'a :: \text{linordered-idom}) \in \mathbb{Z}_{\leq 0}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-subset-Ints*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-nonpos* [dest]: $x \in \mathbb{Z}_{\leq 0} \implies x \leq (0 :: 'a :: \text{linordered-idom})$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-Int* [dest]: $x \in \mathbb{Z}_{\leq 0} \implies x \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-cases*:

assumes $x \in \mathbb{Z}_{\leq 0}$

obtains n **where** $x = \text{of-int } n$ $n \leq 0$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-cases'*:

assumes $x \in \mathbb{Z}_{\leq 0}$

obtains n **where** $x = -\text{of-nat } n$

$\langle \text{proof} \rangle$

lemma *of-real-in-nonpos-Ints-iff*: $(\text{of-real } x :: 'a :: \text{real-algebra-1}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-altdef*: $\mathbb{Z}_{\leq 0} = \{n \in \mathbb{Z}. (n :: 'a :: \text{linordered-idom}) \leq 0\}$

$\langle \text{proof} \rangle$

lemma *uminus-in-Nats-iff*: $-x \in \mathbb{N} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$

$\langle \text{proof} \rangle$

lemma *uminus-in-nonpos-Ints-iff*: $-x \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{N}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-mult*: $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{Z}_{\leq 0} \implies x * y \in \mathbb{N}$

$\langle \text{proof} \rangle$

lemma *Nats-mult-nonpos-Ints*: $x \in \mathbb{N} \implies y \in \mathbb{Z}_{\leq 0} \implies x * y \in \mathbb{Z}_{\leq 0}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-mult-Nats*:

$x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{N} \implies x * y \in \mathbb{Z}_{\leq 0}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-add*:

$x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{Z}_{\leq 0} \implies x + y \in \mathbb{Z}_{\leq 0}$

$\langle \text{proof} \rangle$

lemma *nonpos-Ints-diff-Nats*:

$x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{N} \implies x - y \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *Nats-diff-nonpos-Ints*:

$x \in \mathbb{N} \implies y \in \mathbb{Z}_{\leq 0} \implies x - y \in \mathbb{N}$
 $\langle \text{proof} \rangle$

lemma *plus-of-nat-eq-0-imp*: $z + \text{of-nat } n = 0 \implies z \in \mathbb{Z}_{\leq 0}$

$\langle \text{proof} \rangle$

70.2 Non-negative reals

definition *nonneg-Reals* :: ‘a::real-algebra-1 set ($\langle \mathbb{R}_{\geq 0} \rangle$)

where $\mathbb{R}_{\geq 0} = \{\text{of-real } r \mid r. r \geq 0\}$

lemma *nonneg-Reals-of-real-iff* [simp]: $\text{of-real } r \in \mathbb{R}_{\geq 0} \longleftrightarrow r \geq 0$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-subset-Reals*: $\mathbb{R}_{\geq 0} \subseteq \mathbb{R}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-Real* [dest]: $x \in \mathbb{R}_{\geq 0} \implies x \in \mathbb{R}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-of-nat-I* [simp]: $\text{of-nat } n \in \mathbb{R}_{\geq 0}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-cases*:

assumes $x \in \mathbb{R}_{\geq 0}$

obtains r **where** $x = \text{of-real } r$ $r \geq 0$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-zero-I* [simp]: $0 \in \mathbb{R}_{\geq 0}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-one-I* [simp]: $1 \in \mathbb{R}_{\geq 0}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-minus-one-I* [simp]: $-1 \notin \mathbb{R}_{\geq 0}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-numeral-I* [simp]: $\text{numeral } w \in \mathbb{R}_{\geq 0}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-minus-numeral-I* [simp]: $-\text{numeral } w \notin \mathbb{R}_{\geq 0}$

$\langle \text{proof} \rangle$

lemma *nonneg-Reals-add-I* [simp]: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a + b \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-mult-I* [simp]: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a * b \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-inverse-I* [simp]:
 fixes $a :: 'a::\text{real-div-algebra}$
 shows $a \in \mathbb{R}_{\geq 0} \implies \text{inverse } a \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-divide-I* [simp]:
 fixes $a :: 'a::\text{real-div-algebra}$
 shows $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a / b \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonneg-Reals-pow-I* [simp]: $a \in \mathbb{R}_{\geq 0} \implies a^{\wedge n} \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *complex-nonneg-Reals-iff*: $z \in \mathbb{R}_{\geq 0} \longleftrightarrow \text{Re } z \geq 0 \wedge \text{Im } z = 0$
 ⟨proof⟩

lemma *ii-not-nonneg-Reals* [iff]: $i \notin \mathbb{R}_{\geq 0}$
 ⟨proof⟩

70.3 Non-positive reals

definition *nonpos-Reals* :: $'a::\text{real-algebra-1}$ set ($\langle \mathbb{R}_{\leq 0} \rangle$)
 where $\mathbb{R}_{\leq 0} = \{\text{of-real } r \mid r. r \leq 0\}$

lemma *nonpos-Reals-of-real-iff* [simp]: $\text{of-real } r \in \mathbb{R}_{\leq 0} \longleftrightarrow r \leq 0$
 ⟨proof⟩

lemma *nonpos-Reals-subset-Reals*: $\mathbb{R}_{\leq 0} \subseteq \mathbb{R}$
 ⟨proof⟩

lemma *nonpos-Ints-subset-nonpos-Reals*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-of-nat-iff* [simp]: $\text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow n=0$
 ⟨proof⟩

lemma *nonpos-Reals-Real* [dest]: $x \in \mathbb{R}_{\leq 0} \implies x \in \mathbb{R}$
 ⟨proof⟩

lemma *nonpos-Reals-cases*:
 assumes $x \in \mathbb{R}_{\leq 0}$
 obtains r where $x = \text{of-real } r$ $r \leq 0$
 ⟨proof⟩

lemma *uminus-nonneg-Reals-iff* [simp]: $-x \in \mathbb{R}_{\geq 0} \longleftrightarrow x \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *uminus-nonpos-Reals-iff* [simp]: $-x \in \mathbb{R}_{\leq 0} \longleftrightarrow x \in \mathbb{R}_{\geq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-zero-I* [simp]: $0 \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-one-I* [simp]: $1 \notin \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-numeral-I* [simp]: *numeral* $w \notin \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-add-I* [simp]: $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a + b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-mult-I1*: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-mult-I2*: $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-mult-of-nat-iff*:
fixes $a :: 'a :: \text{real-div-algebra}$ **shows** $a * \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n=0$
 ⟨proof⟩

lemma *nonpos-Reals-inverse-I*:
fixes $a :: 'a :: \text{real-div-algebra}$
shows $a \in \mathbb{R}_{\leq 0} \implies \text{inverse } a \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-divide-I1*:
fixes $a :: 'a :: \text{real-div-algebra}$
shows $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-divide-I2*:
fixes $a :: 'a :: \text{real-div-algebra}$
shows $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$
 ⟨proof⟩

lemma *nonpos-Reals-divide-of-nat-iff*:
fixes $a :: 'a :: \text{real-div-algebra}$ **shows** $a / \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n=0$
 ⟨proof⟩

lemma *nonpos-Reals-inverse-iff* [simp]:

fixes $a :: 'a :: \text{real-div-algebra}$
shows $\text{inverse } a \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *nonpos-Reals-pow-I*: $\llbracket a \in \mathbb{R}_{\leq 0}; \text{ odd } n \rrbracket \implies a^{\wedge n} \in \mathbb{R}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *complex-nonpos-Reals-iff*: $z \in \mathbb{R}_{\leq 0} \longleftrightarrow \text{Re } z \leq 0 \wedge \text{Im } z = 0$
 $\langle \text{proof} \rangle$

lemma *ii-not-nonpos-Reals [iff]*: $i \notin \mathbb{R}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *plus-one-in-nonpos-Ints-imp*: $z + 1 \in \mathbb{Z}_{\leq 0} \implies z \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *of-int-in-nonpos-Ints-iff*:
 $(\text{of-int } n :: 'a :: \text{ring-char-0}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow n \leq 0$
 $\langle \text{proof} \rangle$

lemma *one-plus-of-int-in-nonpos-Ints-iff*:
 $(1 + \text{of-int } n :: 'a :: \text{ring-char-0}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow n \leq -1$
 $\langle \text{proof} \rangle$

lemma *one-minus-of-nat-in-nonpos-Ints-iff*:
 $(1 - \text{of-nat } n :: 'a :: \text{ring-char-0}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow n > 0$
 $\langle \text{proof} \rangle$

lemma *fraction-not-in-Nats*:
assumes $\neg n \text{ dvd } m \text{ } n \neq 0$
shows $\text{of-int } m / \text{of-int } n \notin (\mathbb{N} :: 'a :: \{\text{division-ring, ring-char-0}\} \text{ set})$
 $\langle \text{proof} \rangle$

lemma *not-in-Ints-imp-not-in-nonpos-Ints*: $z \notin \mathbb{Z} \implies z \notin \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *double-in-nonpos-Ints-imp*:
assumes $2 * (z :: 'a :: \text{field-char-0}) \in \mathbb{Z}_{\leq 0}$
shows $z \in \mathbb{Z}_{\leq 0} \vee z + 1/2 \in \mathbb{Z}_{\leq 0}$
 $\langle \text{proof} \rangle$

lemma *fraction-numeral-Ints-iff [simp]*:
 $\text{numeral } a / \text{numeral } b \in (\mathbb{Z} :: 'a :: \{\text{division-ring, ring-char-0}\} \text{ set})$
 $\longleftrightarrow (\text{numeral } b :: \text{int}) \text{ dvd numeral } a \text{ (is ?L=?R)}$
 $\langle \text{proof} \rangle$

lemma *fraction-numeral-Ints-iff1 [simp]*:
 $1 / \text{numeral } b \in (\mathbb{Z} :: 'a :: \{\text{division-ring, ring-char-0}\} \text{ set})$
 $\longleftrightarrow b = \text{Num.One (is ?L=?R)}$

<proof>

lemma *fraction-numeral-Nats-iff* [simp]:
 $\text{numeral } a / \text{numeral } b \in (\mathbb{N} :: 'a :: \{\text{division-ring, ring-char-0}\} \text{ set})$
 $\longleftrightarrow (\text{numeral } b :: \text{int}) \text{ dvd numeral } a \text{ (is ?L=?R)}$
<proof>

lemma *fraction-numeral-Nats-iff1* [simp]:
 $1 / \text{numeral } b \in (\mathbb{N} :: 'a :: \{\text{division-ring, ring-char-0}\} \text{ set})$
 $\longleftrightarrow b = \text{Num.One (is ?L=?R)}$
<proof>

end

71 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

71.1 Numeral Types

typedef *num0* = *UNIV* :: *nat* set *<proof>*
typedef *num1* = *UNIV* :: *unit* set *<proof>*

typedef *'a bit0* = $\{0 \dots 2 * \text{int CARD}('a::\text{finite})\}$
<proof>

typedef *'a bit1* = $\{0 \dots 1 + 2 * \text{int CARD}('a::\text{finite})\}$
<proof>

lemma *card-num0* [simp]: $\text{CARD}(\text{num0}) = 0$
<proof>

lemma *infinite-num0*: $\neg \text{finite}(\text{UNIV} :: \text{num0 set})$
<proof>

lemma *card-num1* [simp]: $\text{CARD}(\text{num1}) = 1$
<proof>

lemma *card-bit0* [simp]: $\text{CARD}('a \text{ bit0}) = 2 * \text{CARD}('a::\text{finite})$
<proof>

lemma *card-bit1* [simp]: $\text{CARD}('a \text{ bit1}) = \text{Suc}(2 * \text{CARD}('a::\text{finite}))$
<proof>

71.2 *num1*

instance *num1* :: *finite*

$\langle proof \rangle$

instantiation *num1* :: *CARD-1*
begin

instance
 $\langle proof \rangle$

end

lemma *num1-eq-iff*: $(x::num1) = (y::num1) \longleftrightarrow True$
 $\langle proof \rangle$

instantiation *num1* :: $\{comm-ring, comm-monoid-mult, numeral\}$
begin

instance
 $\langle proof \rangle$

end

lemma *num1-eqI*:
fixes *a::num1* **shows** $a = b$
 $\langle proof \rangle$

lemma *num1-eq1* [*simp*]:
fixes *a::num1* **shows** $a = 1$
 $\langle proof \rangle$

lemma *forall-1* [*simp*]: $(\forall i::num1. P\ i) \longleftrightarrow P\ 1$
 $\langle proof \rangle$

lemma *ex-1* [*simp*]: $(\exists x::num1. P\ x) \longleftrightarrow P\ 1$
 $\langle proof \rangle$

instantiation *num1* :: *linorder* **begin**
definition $a < b \longleftrightarrow Rep-num1\ a < Rep-num1\ b$
definition $a \leq b \longleftrightarrow Rep-num1\ a \leq Rep-num1\ b$
instance
 $\langle proof \rangle$
end

instance *num1* :: *wellorder*
 $\langle proof \rangle$

instance *bit0* :: (*finite*) *card2*
 $\langle proof \rangle$

```
instance bit1 :: (finite) card2
⟨proof⟩
```

71.3 Locales for modular arithmetic subtypes

```
locale mod-type =
  fixes n :: int
  and Rep :: 'a::{zero,one,plus,times,uminus,minus} ⇒ int
  and Abs :: int ⇒ 'a::{zero,one,plus,times,uminus,minus}
  assumes type: type-definition Rep Abs {0.. $n$ }
  and size1: 1 < n
  and zero-def: 0 = Abs 0
  and one-def: 1 = Abs 1
  and add-def:  $x + y = Abs ((Rep\ x + Rep\ y) \bmod n)$ 
  and mult-def:  $x * y = Abs ((Rep\ x * Rep\ y) \bmod n)$ 
  and diff-def:  $x - y = Abs ((Rep\ x - Rep\ y) \bmod n)$ 
  and minus-def:  $-x = Abs ((- Rep\ x) \bmod n)$ 
begin
```

```
lemma size0: 0 < n
⟨proof⟩
```

```
lemmas definitions =
  zero-def one-def add-def mult-def minus-def diff-def
```

```
lemma Rep-less-n: Rep x < n
⟨proof⟩
```

```
lemma Rep-le-n: Rep x ≤ n
⟨proof⟩
```

```
lemma Rep-inject-sym:  $x = y \longleftrightarrow Rep\ x = Rep\ y$ 
⟨proof⟩
```

```
lemma Rep-inverse: Abs (Rep x) = x
⟨proof⟩
```

```
lemma Abs-inverse:  $m \in \{0.. $n$ \} \implies Rep\ (Abs\ m) = m$ 
⟨proof⟩
```

```
lemma Rep-Abs-mod: Rep (Abs (m mod n)) = m mod n
⟨proof⟩
```

```
lemma Rep-Abs-0: Rep (Abs 0) = 0
⟨proof⟩
```

```
lemma Rep-0: Rep 0 = 0
⟨proof⟩
```

lemma *Rep-Abs-1*: $\text{Rep } (\text{Abs } 1) = 1$
 $\langle \text{proof} \rangle$

lemma *Rep-1*: $\text{Rep } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *Rep-mod*: $\text{Rep } x \text{ mod } n = \text{Rep } x$
 $\langle \text{proof} \rangle$

lemmas *Rep-simps* =
Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

lemma *comm-ring-1*: $\text{OFCLASS}('a, \text{comm-ring-1-class})$
 $\langle \text{proof} \rangle$

end

locale *mod-ring* = *mod-type* n *Rep* *Abs*
for $n :: \text{int}$
and $\text{Rep} :: 'a :: \{\text{comm-ring-1}\} \Rightarrow \text{int}$
and $\text{Abs} :: \text{int} \Rightarrow 'a :: \{\text{comm-ring-1}\}$
begin

lemma *of-nat-eq*: $\text{of-nat } k = \text{Abs } (\text{int } k \text{ mod } n)$
 $\langle \text{proof} \rangle$

lemma *of-int-eq*: $\text{of-int } z = \text{Abs } (z \text{ mod } n)$
 $\langle \text{proof} \rangle$

lemma *Rep-numeral*: $\text{Rep } (\text{numeral } w) = \text{numeral } w \text{ mod } n$
 $\langle \text{proof} \rangle$

lemma *iszero-numeral*:
 $\text{iszero } (\text{numeral } w :: 'a) \longleftrightarrow \text{numeral } w \text{ mod } n = 0$
 $\langle \text{proof} \rangle$

lemma *cases*:
assumes $1: \bigwedge z. \llbracket (x :: 'a) = \text{of-int } z; 0 \leq z; z < n \rrbracket \Longrightarrow P$
shows P
 $\langle \text{proof} \rangle$

lemma *induct*:
 $(\bigwedge z. \llbracket 0 \leq z; z < n \rrbracket \Longrightarrow P (\text{of-int } z)) \Longrightarrow P (x :: 'a)$
 $\langle \text{proof} \rangle$

lemma *UNIV-eq*: $(\text{UNIV} :: 'a \text{ set}) = \text{Abs } ' \{0..<n\}$
 $\langle \text{proof} \rangle$

lemma *CARD-eq*: $\text{CARD}('a) = \text{nat } n$

$\langle proof \rangle$

lemma *CHAR-eq* [*simp*]: $CHAR('a) = CARD('a)$

$\langle proof \rangle$

end

71.4 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation

bit0 and *bit1* :: (*finite*) {*zero,one,plus,times,uminus,minus*}
begin

definition *Abs-bit0'* :: *int* \Rightarrow '*a bit0* **where**

Abs-bit0' *x* = *Abs-bit0* (*x mod int CARD('a bit0)*)

definition *Abs-bit1'* :: *int* \Rightarrow '*a bit1* **where**

Abs-bit1' *x* = *Abs-bit1* (*x mod int CARD('a bit1)*)

definition 0 = *Abs-bit0* 0

definition 1 = *Abs-bit0* 1

definition *x* + *y* = *Abs-bit0'* (*Rep-bit0* *x* + *Rep-bit0* *y*)

definition *x* * *y* = *Abs-bit0'* (*Rep-bit0* *x* * *Rep-bit0* *y*)

definition *x* - *y* = *Abs-bit0'* (*Rep-bit0* *x* - *Rep-bit0* *y*)

definition - *x* = *Abs-bit0'* (- *Rep-bit0* *x*)

definition 0 = *Abs-bit1* 0

definition 1 = *Abs-bit1* 1

definition *x* + *y* = *Abs-bit1'* (*Rep-bit1* *x* + *Rep-bit1* *y*)

definition *x* * *y* = *Abs-bit1'* (*Rep-bit1* *x* * *Rep-bit1* *y*)

definition *x* - *y* = *Abs-bit1'* (*Rep-bit1* *x* - *Rep-bit1* *y*)

definition - *x* = *Abs-bit1'* (- *Rep-bit1* *x*)

instance $\langle proof \rangle$

end

interpretation *bit0*:

mod-type int CARD('a::finite bit0)

Rep-bit0 :: '*a::finite bit0* \Rightarrow *int*

Abs-bit0 :: *int* \Rightarrow '*a::finite bit0*

$\langle proof \rangle$

interpretation *bit1*:

mod-type int CARD('a::finite bit1)

Rep-bit1 :: '*a::finite bit1* \Rightarrow *int*

Abs-bit1 :: *int* \Rightarrow '*a::finite bit1*

```

    <proof>

instance bit0 :: (finite) comm-ring-1
    <proof>

instance bit1 :: (finite) comm-ring-1
    <proof>

interpretation bit0:
    mod-ring int CARD('a::finite bit0)
    Rep-bit0 :: 'a::finite bit0  $\Rightarrow$  int
    Abs-bit0 :: int  $\Rightarrow$  'a::finite bit0
    <proof>

interpretation bit1:
    mod-ring int CARD('a::finite bit1)
    Rep-bit1 :: 'a::finite bit1  $\Rightarrow$  int
    Abs-bit1 :: int  $\Rightarrow$  'a::finite bit1
    <proof>

    Set up cases, induction, and arithmetic

lemmas bit0-cases [case-names of-int, cases type: bit0] = bit0.cases
lemmas bit1-cases [case-names of-int, cases type: bit1] = bit1.cases

lemmas bit0-induct [case-names of-int, induct type: bit0] = bit0.induct
lemmas bit1-induct [case-names of-int, induct type: bit1] = bit1.induct

lemmas bit0-iszero-numeral [simp] = bit0.iszero-numeral
lemmas bit1-iszero-numeral [simp] = bit1.iszero-numeral

lemmas [simp] = eq-numeral-iff-iszero [where 'a='a bit0] for dummy :: 'a::finite
lemmas [simp] = eq-numeral-iff-iszero [where 'a='a bit1] for dummy :: 'a::finite

```

71.5 Order instances

```

instantiation bit0 and bit1 :: (finite) linorder begin
definition a < b  $\longleftrightarrow$  Rep-bit0 a < Rep-bit0 b
definition a  $\leq$  b  $\longleftrightarrow$  Rep-bit0 a  $\leq$  Rep-bit0 b
definition a < b  $\longleftrightarrow$  Rep-bit1 a < Rep-bit1 b
definition a  $\leq$  b  $\longleftrightarrow$  Rep-bit1 a  $\leq$  Rep-bit1 b

instance
    <proof>
end

instance bit0 and bit1 :: (finite) wellorder
    <proof>

```

71.6 Code setup and type classes for code generation

Code setup for *num0* and *num1*

```
definition Num0 :: num0 where Num0 = Abs-num0 0
code-datatype Num0
```

```
instantiation num0 :: equal begin
definition equal-num0 :: num0  $\Rightarrow$  num0  $\Rightarrow$  bool
  where equal-num0 = (=)
instance  $\langle$ proof $\rangle$ 
end
```

```
lemma equal-num0-code [code]:
  equal-class.equal Num0 Num0 = True
 $\langle$ proof $\rangle$ 
```

```
code-datatype 1 :: num1
```

```
instantiation num1 :: equal begin
definition equal-num1 :: num1  $\Rightarrow$  num1  $\Rightarrow$  bool
  where equal-num1 = (=)
instance  $\langle$ proof $\rangle$ 
end
```

```
lemma equal-num1-code [code]:
  equal-class.equal (1 :: num1) 1 = True
 $\langle$ proof $\rangle$ 
```

```
instantiation num1 :: enum begin
definition enum-class.enum = [1 :: num1]
definition enum-class.enum-all P = P (1 :: num1)
definition enum-class.enum-ex P = P (1 :: num1)
instance
   $\langle$ proof $\rangle$ 
end
```

```
instantiation num0 and num1 :: card-UNIV begin
definition finite-UNIV = Phantom(num0) False
definition card-UNIV = Phantom(num0) 0
definition finite-UNIV = Phantom(num1) True
definition card-UNIV = Phantom(num1) 1
instance
   $\langle$ proof $\rangle$ 
end
```

Code setup for 'a bit0 and 'a bit1

```
declare
  bit0.Rep-inverse[code abstype]
  bit0.Rep-0[code abstract]
```

bit0.Rep-1[code abstract]

lemma *Abs-bit0'-code* [code abstract]:

Rep-bit0 (*Abs-bit0'* $x :: 'a :: \text{finite } \text{bit0}$) = $x \bmod \text{int } (\text{CARD}('a \text{ bit0}))$
 ⟨proof⟩

lemma *inj-on-Abs-bit0*:

inj-on (*Abs-bit0* :: $\text{int} \Rightarrow 'a \text{ bit0}$) { $0..<2 * \text{int } \text{CARD}('a :: \text{finite})$ }
 ⟨proof⟩

declare

bit1.Rep-inverse[code abstype]
bit1.Rep-0[code abstract]
bit1.Rep-1[code abstract]

lemma *Abs-bit1'-code* [code abstract]:

Rep-bit1 (*Abs-bit1'* $x :: 'a :: \text{finite } \text{bit1}$) = $x \bmod \text{int } (\text{CARD}('a \text{ bit1}))$
 ⟨proof⟩

lemma *inj-on-Abs-bit1*:

inj-on (*Abs-bit1* :: $\text{int} \Rightarrow 'a \text{ bit1}$) { $0..<1 + 2 * \text{int } \text{CARD}('a :: \text{finite})$ }
 ⟨proof⟩

instantiation *bit0* and *bit1* :: (*finite*) *equal* **begin**

definition *equal-class.equal* $x y \longleftrightarrow \text{Rep-bit0 } x = \text{Rep-bit0 } y$

definition *equal-class.equal* $x y \longleftrightarrow \text{Rep-bit1 } x = \text{Rep-bit1 } y$

instance

⟨proof⟩

end

instantiation *bit0* :: (*finite*) *enum* **begin**

definition (*enum-class.enum* :: $'a \text{ bit0 list}$) = $\text{map } (\text{Abs-bit0}' \circ \text{int}) (\text{upt } 0 (\text{CARD}('a \text{ bit0})))$

definition *enum-class.enum-all* $P = (\forall b :: 'a \text{ bit0} \in \text{set } \text{enum-class.enum}. P b)$

definition *enum-class.enum-ex* $P = (\exists b :: 'a \text{ bit0} \in \text{set } \text{enum-class.enum}. P b)$

instance ⟨proof⟩

end

instantiation *bit1* :: (*finite*) *enum* **begin**

definition (*enum-class.enum* :: $'a \text{ bit1 list}$) = $\text{map } (\text{Abs-bit1}' \circ \text{int}) (\text{upt } 0 (\text{CARD}('a \text{ bit1})))$

definition *enum-class.enum-all* $P = (\forall b :: 'a \text{ bit1} \in \text{set } \text{enum-class.enum}. P b)$

definition *enum-class.enum-ex* $P = (\exists b :: 'a \text{ bit1} \in \text{set } \text{enum-class.enum}. P b)$

instance
 $\langle proof \rangle$

end

instantiation *bit0* and *bit1* :: (*finite*) *finite-UNIV* **begin**

definition *finite-UNIV* = *Phantom*('a *bit0*) *True*

definition *finite-UNIV* = *Phantom*('a *bit1*) *True*

instance $\langle proof \rangle$

end

instantiation *bit0* and *bit1* :: (*{finite, card-UNIV}*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('a *bit0*) (*2 * of-phantom* (*card-UNIV* :: 'a *card-UNIV*))

definition *card-UNIV* = *Phantom*('a *bit1*) (*1 + 2 * of-phantom* (*card-UNIV* :: 'a *card-UNIV*))

instance $\langle proof \rangle$

end

71.7 Syntax

syntax

-*NumeralType* :: *num-token* => *type* ($\langle \langle \text{open-block notation} = \langle \text{type-literal number} \rangle \rangle \rangle$)

-*NumeralType0* :: *type* ($\langle \langle \text{open-block notation} = \langle \text{type-literal number} \rangle \rangle 0 \rangle$)

-*NumeralType1* :: *type* ($\langle \langle \text{open-block notation} = \langle \text{type-literal number} \rangle \rangle 1 \rangle$)

translations

(*type*) *1* == (*type*) *num1*

(*type*) *0* == (*type*) *num0*

$\langle ML \rangle$

71.8 Examples

lemma *CARD*(*0*) = *0* $\langle proof \rangle$

lemma *CARD*(*17*) = *17* $\langle proof \rangle$

lemma *CHAR*(*23*) = *23* $\langle proof \rangle$

lemma *8 * 11 ^ 3 - 6* = (*2::5*) $\langle proof \rangle$

end

72 ω -words

theory *Omega-Words-Fun*

imports *Infinite-Set*

begin

Note: This theory is based on Stefan Merz’s work.

Automata recognize languages, which are sets of words. For the theory of ω -automata, we are mostly interested in ω -words, but it is sometimes useful to reason about finite words, too. We are modeling finite words as lists; this lets us benefit from the existing library. Other formalizations could be investigated, such as representing words as functions whose domains are initial intervals of the natural numbers.

72.1 Type declaration and elementary operations

We represent ω -words as functions from the natural numbers to the alphabet type. Other possible formalizations include a coinductive definition or a uniform encoding of finite and infinite words, as studied by Müller et al.

type-synonym

$'a \text{ word} = \text{nat} \Rightarrow 'a$

We can prefix a finite word to an ω -word, and a way to obtain an ω -word from a finite, non-empty word is by ω -iteration.

definition

$\text{conc} :: ['a \text{ list}, 'a \text{ word}] \Rightarrow 'a \text{ word}$ (**infixr** \frown 65)
where $w \frown x == \lambda n. \text{if } n < \text{length } w \text{ then } w!n \text{ else } x (n - \text{length } w)$

definition

$\text{iter} :: 'a \text{ list} \Rightarrow 'a \text{ word}$ ($\langle \langle \text{notation} = \langle \text{postfix } \omega \rangle \rangle^\omega \rangle$ [1000])
where $\text{iter } w == \text{if } w = [] \text{ then undefined else } (\lambda n. w!(n \bmod (\text{length } w)))$

lemma conc-empty[simp] : $[] \frown w = w$

$\langle \text{proof} \rangle$

lemma conc-fst[simp] : $n < \text{length } w \implies (w \frown x) n = w!n$

$\langle \text{proof} \rangle$

lemma conc-snd[simp] : $\neg(n < \text{length } w) \implies (w \frown x) n = x (n - \text{length } w)$

$\langle \text{proof} \rangle$

lemma iter-nth [simp] : $0 < \text{length } w \implies w^\omega n = w!(n \bmod (\text{length } w))$

$\langle \text{proof} \rangle$

lemma conc-conc[simp] : $u \frown v \frown w = (u @ v) \frown w$ (**is** $?lhs = ?rhs$)

$\langle \text{proof} \rangle$

lemma range-conc[simp] : $\text{range } (w_1 \frown w_2) = \text{set } w_1 \cup \text{range } w_2$

$\langle \text{proof} \rangle$

lemma iter-unroll : $0 < \text{length } w \implies w^\omega = w \frown w^\omega$

$\langle \text{proof} \rangle$

72.2 Subsequence, Prefix, and Suffix

definition $\text{suffix} :: [\text{nat}, 'a \text{ word}] \Rightarrow 'a \text{ word}$
where $\text{suffix } k \ x \equiv \lambda n. \ x \ (k+n)$

definition $\text{subsequence} :: 'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ list}$
 $(\langle (\langle \text{open-block notation} = \langle \text{mixfix subsequence} \rangle - [- \rightarrow -]) \rangle \ 900)$
where $\text{subsequence } w \ i \ j \equiv \text{map } w \ [i..<j]$

abbreviation $\text{prefix} :: \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ list}$
where $\text{prefix } n \ w \equiv \text{subsequence } w \ 0 \ n$

lemma $\text{suffix-nth} [\text{simp}]: (\text{suffix } k \ x) \ n = x \ (k+n)$
 $\langle \text{proof} \rangle$

lemma $\text{suffix-0} [\text{simp}]: \text{suffix } 0 \ x = x$
 $\langle \text{proof} \rangle$

lemma $\text{suffix-suffix} [\text{simp}]: \text{suffix } m \ (\text{suffix } k \ x) = \text{suffix } (k+m) \ x$
 $\langle \text{proof} \rangle$

lemma $\text{subsequence-append}: \text{prefix } (i + j) \ w = \text{prefix } i \ w \ @ \ (w \ [i \rightarrow i + j])$
 $\langle \text{proof} \rangle$

lemma $\text{subsequence-drop} [\text{simp}]: \text{drop } i \ (w \ [j \rightarrow k]) = w \ [j + i \rightarrow k]$
 $\langle \text{proof} \rangle$

lemma $\text{subsequence-empty} [\text{simp}]: w \ [i \rightarrow j] = [] \longleftrightarrow j \leq i$
 $\langle \text{proof} \rangle$

lemma $\text{subsequence-length} [\text{simp}]: \text{length } (\text{subsequence } w \ i \ j) = j - i$
 $\langle \text{proof} \rangle$

lemma $\text{subsequence-nth} [\text{simp}]: k < j - i \implies (w \ [i \rightarrow j]) ! k = w \ (i + k)$
 $\langle \text{proof} \rangle$

lemma $\text{subseq-to-zero} [\text{simp}]: w[i \rightarrow 0] = []$
 $\langle \text{proof} \rangle$

lemma $\text{subseq-to-smaller} [\text{simp}]: i \geq j \implies w[i \rightarrow j] = []$
 $\langle \text{proof} \rangle$

lemma $\text{subseq-to-Suc} [\text{simp}]: i \leq j \implies w \ [i \rightarrow \text{Suc } j] = w \ [i \rightarrow j] \ @ \ [w \ j]$
 $\langle \text{proof} \rangle$

lemma $\text{subsequence-singleton} [\text{simp}]: w \ [i \rightarrow \text{Suc } i] = [w \ i]$
 $\langle \text{proof} \rangle$

lemma $\text{subsequence-prefix-suffix}: \text{prefix } (j - i) \ (\text{suffix } i \ w) = w \ [i \rightarrow j]$

$\langle proof \rangle$

lemma *prefix-suffix*: $x = \text{prefix } n \ x \frown (\text{suffix } n \ x)$
 $\langle proof \rangle$

declare *prefix-suffix*[*symmetric*, *simp*]

lemma *word-split*: **obtains** $v_1 \ v_2$ **where** $v = v_1 \frown v_2$ *length* $v_1 = k$
 $\langle proof \rangle$

lemma *set-subsequence*[*simp*]: $\text{set } (w[i \rightarrow j]) = w^{\{i..<j\}}$
 $\langle proof \rangle$

lemma *subsequence-take*[*simp*]: $\text{take } i \ (w [j \rightarrow k]) = w [j \rightarrow \min (j + i) \ k]$
 $\langle proof \rangle$

lemma *subsequence-shift*[*simp*]: $(\text{suffix } i \ w) [j \rightarrow k] = w [i + j \rightarrow i + k]$
 $\langle proof \rangle$

lemma *suffix-subseq-join*[*simp*]: $i \leq j \implies v [i \rightarrow j] \frown \text{suffix } j \ v = \text{suffix } i \ v$
 $\langle proof \rangle$

lemma *prefix-conc-fst*[*simp*]:
assumes $j \leq \text{length } w$
shows $\text{prefix } j \ (w \frown w') = \text{take } j \ w$
 $\langle proof \rangle$

lemma *prefix-conc-snd*[*simp*]:
assumes $n \geq \text{length } u$
shows $\text{prefix } n \ (u \frown v) = u @ \text{prefix } (n - \text{length } u) \ v$
 $\langle proof \rangle$

lemma *prefix-conc-length*[*simp*]: $\text{prefix } (\text{length } w) \ (w \frown w') = w$
 $\langle proof \rangle$

lemma *suffix-conc-fst*[*simp*]:
assumes $n \leq \text{length } u$
shows $\text{suffix } n \ (u \frown v) = \text{drop } n \ u \frown v$
 $\langle proof \rangle$

lemma *suffix-conc-snd*[*simp*]:
assumes $n \geq \text{length } u$
shows $\text{suffix } n \ (u \frown v) = \text{suffix } (n - \text{length } u) \ v$
 $\langle proof \rangle$

lemma *suffix-conc-length*[*simp*]: $\text{suffix } (\text{length } w) \ (w \frown w') = w'$
 $\langle proof \rangle$

lemma *concat-eq*[*iff*]:
assumes $\text{length } v_1 = \text{length } v_2$
shows $v_1 \frown u_1 = v_2 \frown u_2 \longleftrightarrow v_1 = v_2 \wedge u_1 = u_2$
(is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma *same-concat-eq*[*iff*]: $u \frown v = u \frown w \longleftrightarrow v = w$
 $\langle \text{proof} \rangle$

lemma *comp-concat*[*simp*]: $f \circ u \frown v = \text{map } f \, u \frown (f \circ v)$
 $\langle \text{proof} \rangle$

72.3 Prepending

primrec *build* :: $'a \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}$ (**infixr** $\langle \#\# \rangle$ 65)
where $(a \#\# w) \, 0 = a \mid (a \#\# w) \, (\text{Suc } i) = w \, i$

lemma *build-eq*[*iff*]: $a_1 \#\# w_1 = a_2 \#\# w_2 \longleftrightarrow a_1 = a_2 \wedge w_1 = w_2$
 $\langle \text{proof} \rangle$

lemma *build-cons*[*simp*]: $(a \# u) \frown v = a \#\# u \frown v$
 $\langle \text{proof} \rangle$

lemma *build-append*[*simp*]: $(w @ a \# u) \frown v = w \frown a \#\# u \frown v$
 $\langle \text{proof} \rangle$

lemma *build-first*[*simp*]: $w \, 0 \#\# \text{suffix } (\text{Suc } 0) \, w = w$
 $\langle \text{proof} \rangle$

lemma *build-split*[*intro*]: $w = w \, 0 \#\# \text{suffix } 1 \, w$
 $\langle \text{proof} \rangle$

lemma *build-range*[*simp*]: $\text{range } (a \#\# w) = \text{insert } a \, (\text{range } w)$
 $\langle \text{proof} \rangle$

lemma *suffix-singleton-suffix*[*simp*]: $w \, i \#\# \text{suffix } (\text{Suc } i) \, w = \text{suffix } i \, w$
 $\langle \text{proof} \rangle$

Find the first occurrence of a letter from a given set

lemma *word-first-split-set*:
assumes $A \cap \text{range } w \neq \{\}$
obtains $u \, a \, v$ **where** $w = u \frown [a] \frown v \wedge A \cap \text{set } u = \{\}$ $a \in A$
 $\langle \text{proof} \rangle$

72.4 The limit set of an ω -word

The limit set (also called infinity set) of an ω -word is the set of letters that appear infinitely often in the word. This set plays an important role in

defining acceptance conditions of ω -automata.

definition *limit* :: 'a word \Rightarrow 'a set
where *limit* $x \equiv \{a . \exists_{\infty} n . x\ n = a\}$

lemma *limit-iff-frequent*: $a \in \text{limit } x \longleftrightarrow (\exists_{\infty} n . x\ n = a)$
 <proof>

The following is a different way to define the limit, using the reverse image, making the laws about reverse image applicable to the limit set. (Might want to change the definition above?)

lemma *limit-vimage*: $(a \in \text{limit } x) = \text{infinite } (x - ' \{a\})$
 <proof>

lemma *two-in-limit-iff*:
 $(\{a, b\} \subseteq \text{limit } x) =$
 $((\exists n. x\ n = a) \wedge (\forall n. x\ n = a \longrightarrow (\exists m > n. x\ m = b)) \wedge (\forall m. x\ m = b \longrightarrow$
 $(\exists n > m. x\ n = a)))$
 (is ?lhs = (?r1 \wedge ?r2 \wedge ?r3))
 <proof>

For ω -words over a finite alphabet, the limit set is non-empty. Moreover, from some position onward, any such word contains only letters from its limit set.

lemma *limit-nonempty*:
assumes *fin*: *finite* (range x)
shows $\exists a. a \in \text{limit } x$
 <proof>

lemmas *limit-nonemptyE* = *limit-nonempty*[*THEN exE*]

lemma *limit-inter-INF*:
assumes *hyp*: *limit* $w \cap S \neq \{\}$
shows $\exists_{\infty} n. w\ n \in S$
 <proof>

The reverse implication is true only if S is finite.

lemma *INF-limit-inter*:
assumes *hyp*: $\exists_{\infty} n. w\ n \in S$
and *fin*: *finite* ($S \cap \text{range } w$)
shows $\exists a. a \in \text{limit } w \cap S$
 <proof>

lemma *fin-ex-inf-eq-limit*: *finite* $A \implies (\exists_{\infty} i. w\ i \in A) \longleftrightarrow \text{limit } w \cap A \neq \{\}$
 <proof>

lemma *limit-in-range-suffix*: *limit* $x \subseteq \text{range } (\text{suffix } k\ x)$
 <proof>

lemma *limit-in-range*: $\text{limit } r \subseteq \text{range } r$
 $\langle \text{proof} \rangle$

lemmas *limit-in-range-suffix* $D = \text{limit-in-range-suffix}[\text{THEN subset} D]$

lemma *limit-subset*: $\text{limit } f \subseteq f \text{ ‘ } \{n..\}$
 $\langle \text{proof} \rangle$

theorem *limit-is-suffix*:
assumes *fin*: *finite* (*range x*)
shows $\exists k. \text{limit } x = \text{range } (\text{suffix } k \ x)$
 $\langle \text{proof} \rangle$

lemmas *limit-is-suffixE* $= \text{limit-is-suffix}[\text{THEN exE}]$

The limit set enjoys some simple algebraic laws with respect to concatenation, suffixes, iteration, and renaming.

theorem *limit-conc* [*simp*]: $\text{limit } (w \frown x) = \text{limit } x$
 $\langle \text{proof} \rangle$

theorem *limit-suffix* [*simp*]: $\text{limit } (\text{suffix } n \ x) = \text{limit } x$
 $\langle \text{proof} \rangle$

theorem *limit-iter* [*simp*]:
assumes *nempty*: $0 < \text{length } w$
shows $\text{limit } w^\omega = \text{set } w$
 $\langle \text{proof} \rangle$

lemma *limit-o* [*simp*]:
assumes *a*: $a \in \text{limit } w$
shows $f \ a \in \text{limit } (f \circ w)$
 $\langle \text{proof} \rangle$

The converse relation is not true in general: $f(a)$ can be in the limit of $f \circ w$ even though a is not in the limit of w . However, *limit* commutes with renaming if the function is injective. More generally, if $f(a)$ is the image of only finitely many elements, some of these must be in the limit of w .

lemma *limit-o-inv*:
assumes *fin*: *finite* ($f \text{ ‘ } \{x\}$)
and *x*: $x \in \text{limit } (f \circ w)$
shows $\exists a \in (f \text{ ‘ } \{x\}). a \in \text{limit } w$
 $\langle \text{proof} \rangle$

theorem *limit-inj* [*simp*]:
assumes *inj*: *inj* *f*
shows $\text{limit } (f \circ w) = f \text{ ‘ } (\text{limit } w)$
 $\langle \text{proof} \rangle$

lemma *limit-inter-empty*:

assumes *fin*: *finite* (*range w*)
assumes *hyp*: *limit w* \cap *S* = {}
shows $\forall_{\infty} n. w\ n \notin S$
 <proof>

If the limit is the suffix of the sequence’s range, we may increase the suffix index arbitrarily

lemma *limit-range-suffix-incr*:
assumes *limit r* = *range* (*suffix i r*)
assumes $j \geq i$
shows *limit r* = *range* (*suffix j r*)
 (**is** ?*lhs* = ?*rhs*)
 <proof>

For two finite sequences, we can find a common suffix index such that the limits can be represented as these suffixes’ ranges.

lemma *common-range-limit*:
assumes *finite* (*range x*)
and *finite* (*range y*)
obtains *i* **where** *limit x* = *range* (*suffix i x*)
and *limit y* = *range* (*suffix i y*)
 <proof>

72.5 Index sequences and piecewise definitions

A word can be defined piecewise: given a sequence of words w_0, w_1, \dots and a strictly increasing sequence of integers i_0, i_1, \dots where $i_0 = 0$, a single word is obtained by concatenating subwords of the w_n as given by the integers: the resulting word is

$$(w_0)_{i_0} \dots (w_0)_{i_1-1} (w_1)_{i_1} \dots (w_1)_{i_2-1} \dots$$

We prepare the field by proving some trivial facts about such sequences of indexes.

definition *idx-sequence* :: *nat word* \Rightarrow *bool*
where *idx-sequence idx* \equiv (*idx 0* = 0) \wedge ($\forall n. \text{idx } n < \text{idx } (\text{Suc } n)$)

lemma *idx-sequence-less*:
assumes *iseq*: *idx-sequence idx*
shows *idx n* < *idx* (*Suc*(*n+k*))
 <proof>

lemma *idx-sequence-inj*:
assumes *iseq*: *idx-sequence idx*
and *eq*: *idx m* = *idx n*
shows *m* = *n*
 <proof>

lemma *idx-sequence-mono*:
assumes *iseq*: *idx-sequence idx*
and *m*: $m \leq n$
shows $\text{idx } m \leq \text{idx } n$
 $\langle \text{proof} \rangle$

Given an index sequence, every natural number is contained in the interval defined by two adjacent indexes, and in fact this interval is determined uniquely.

lemma *idx-sequence-idx*:
assumes *idx-sequence idx*
shows $\text{idx } k \in \{\text{idx } k ..< \text{idx } (\text{Suc } k)\}$
 $\langle \text{proof} \rangle$

lemma *idx-sequence-interval*:
assumes *iseq*: *idx-sequence idx*
shows $\exists k. n \in \{\text{idx } k ..< \text{idx } (\text{Suc } k)\}$
(is ?P n is $\exists k. ?in n k$)
 $\langle \text{proof} \rangle$

lemma *idx-sequence-interval-unique*:
assumes *iseq*: *idx-sequence idx*
and *k*: $n \in \{\text{idx } k ..< \text{idx } (\text{Suc } k)\}$
and *m*: $n \in \{\text{idx } m ..< \text{idx } (\text{Suc } m)\}$
shows $k = m$
 $\langle \text{proof} \rangle$

lemma *idx-sequence-unique-interval*:
assumes *iseq*: *idx-sequence idx*
shows $\exists! k. n \in \{\text{idx } k ..< \text{idx } (\text{Suc } k)\}$
 $\langle \text{proof} \rangle$

Now we can define the piecewise construction of a word using an index sequence.

definition *merge* :: 'a word word \Rightarrow nat word \Rightarrow 'a word
where $\text{merge } ws \text{ idx} \equiv \lambda n. \text{let } i = \text{THE } i. n \in \{\text{idx } i ..< \text{idx } (\text{Suc } i)\} \text{ in } ws \ i \ n$

lemma *merge*:
assumes *idx*: *idx-sequence idx*
and *n*: $n \in \{\text{idx } i ..< \text{idx } (\text{Suc } i)\}$
shows $\text{merge } ws \text{ idx } n = ws \ i \ n$
 $\langle \text{proof} \rangle$

lemma *merge0*:
assumes *idx*: *idx-sequence idx*
shows $\text{merge } ws \text{ idx } 0 = ws \ 0 \ 0$
 $\langle \text{proof} \rangle$

lemma *merge-Suc*:

```

assumes idx: idx-sequence idx
and n:  $n \in \{idx\ i \ ..< \ idx\ (Suc\ i)\}$ 
shows  $merge\ ws\ idx\ (Suc\ n) = (if\ Suc\ n = idx\ (Suc\ i)\ then\ ws\ (Suc\ i)\ else\ ws\ i)\ (Suc\ n)$ 
 $\langle proof \rangle$ 

end

```

73 Combinator syntax for generic, open state monads (single-threaded monads)

```

theory Open-State-Syntax
imports Main
begin

context
includes state-combinator-syntax
begin

```

73.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threadedly).

To enter from the Haskell world, https://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

73.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

“yielding” transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, “yielding” a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and $\alpha, \beta, \gamma, \dots$ for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ ,

the former value should not be used again. To achieve this, we use a set of monad combinators:

Given two transformations f and g , they may be directly composed using the $(\circ>)$ combinator, forming a forward composition: $(f \circ> g) s = f (g s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the $(\circ\rightarrow)$ combinator: $(f \circ\rightarrow (\lambda x. g)) s = (let (x, s') = f s in g s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *return* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units (“open monad”).
- The type of states may change due to a transformation.

73.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemmas *monad-simp* = *Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse* = *monad-simp fcomp-apply scomp-apply split-beta*

end

73.4 Do-syntax

nonterminal *sdo-binds* and *sdo-bind*

syntax

-sdo-block :: *sdo-binds* \Rightarrow *'a*

($\langle \langle \text{open-block notation} = \langle \text{mixfix exec block} \rangle \rangle \text{exec} \{ // (2 \text{ -}) // \} \rangle$ [12] 62)

-sdo-bind :: [*pttrn*, *'a*] \Rightarrow *sdo-bind*

```

  (⟨⟨indent=2 notation=⟨infix exec bind⟩⟩- <- / -⟩ 13)
-sdo-let :: [pttrn, 'a] ⇒ sdo-bind
  (⟨⟨indent=2 notation=⟨infix exec let⟩⟩let - = / -⟩ [1000, 13] 13)
-sdo-then :: 'a ⇒ sdo-bind (⟨-⟩ [14] 13)
-sdo-final :: 'a ⇒ sdo-binds (⟨-⟩)
-sdo-cons :: [sdo-bind, sdo-binds] ⇒ sdo-binds
  (⟨⟨open-block notation=⟨infix exec next⟩⟩-; / -⟩ [13, 12] 12)

```

syntax (*ASCII*)

```

-sdo-bind :: [pttrn, 'a] ⇒ sdo-bind
  (⟨⟨indent=2 notation=⟨infix exec bind⟩⟩- <- / -⟩ 13)

```

syntax-consts

```

-sdo-let == Let

```

translations

```

-sdo-block (-sdo-cons (-sdo-bind p t) (-sdo-final e))
  == CONST scomp t (λp. e)
-sdo-block (-sdo-cons (-sdo-then t) (-sdo-final e))
  => CONST fcomp t e
-sdo-final (-sdo-block (-sdo-cons (-sdo-then t) (-sdo-final e)))
  <= -sdo-final (CONST fcomp t e)
-sdo-block (-sdo-cons (-sdo-then t) e)
  <= CONST fcomp t (-sdo-block e)
-sdo-block (-sdo-cons (-sdo-let p t) bs)
  == let p = t in -sdo-block bs
-sdo-block (-sdo-cons b (-sdo-cons c cs))
  == -sdo-block (-sdo-cons b (-sdo-final (-sdo-block (-sdo-cons c cs))))
-sdo-cons (-sdo-let p t) (-sdo-final s)
  == -sdo-final (let p = t in s)
-sdo-block (-sdo-final e) => e

```

For an example, see `~/src/HOL/Proofs/Extraction/Higman_Extraction.thy`.

end

74 Canonical order on option type

theory *Option-ord*

imports *Main*

begin

unbundle *lattice-syntax*

instantiation *option* :: (*preorder*) *preorder*

begin

definition *less-eq-option* **where**

```

  x ≤ y ⟷ (case x of None ⇒ True | Some x ⇒ (case y of None ⇒ False | Some
y ⇒ x ≤ y))

```

definition *less-option* **where**

$x < y \longleftrightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } y \Rightarrow (\text{case } x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow x < y))$

lemma *less-eq-option-None* [*simp*]: $\text{None} \leq x$
 $\langle \text{proof} \rangle$

lemma *less-eq-option-None-code* [*code*]: $\text{None} \leq x \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

lemma *less-eq-option-None-is-None*: $x \leq \text{None} \implies x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *less-eq-option-Some-None* [*simp*, *code*]: $\text{Some } x \leq \text{None} \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *less-eq-option-Some* [*simp*, *code*]: $\text{Some } x \leq \text{Some } y \longleftrightarrow x \leq y$
 $\langle \text{proof} \rangle$

lemma *less-option-None* [*simp*, *code*]: $x < \text{None} \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *less-option-None-is-Some*: $\text{None} < x \implies \exists z. x = \text{Some } z$
 $\langle \text{proof} \rangle$

lemma *less-option-None-Some* [*simp*]: $\text{None} < \text{Some } x$
 $\langle \text{proof} \rangle$

lemma *less-option-None-Some-code* [*code*]: $\text{None} < \text{Some } x \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

lemma *less-option-Some* [*simp*, *code*]: $\text{Some } x < \text{Some } y \longleftrightarrow x < y$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

instance *option* :: (*order*) *order*
 $\langle \text{proof} \rangle$

instance *option* :: (*linorder*) *linorder*
 $\langle \text{proof} \rangle$

instantiation *option* :: (*order*) *order-bot*
begin

definition *bot-option* **where** $\perp = \text{None}$

instance
 $\langle \text{proof} \rangle$

end

instantiation *option* :: (*order-top*) *order-top*
begin

definition *top-option* **where** $\top = \text{Some } \top$

instance
 $\langle \text{proof} \rangle$

end

instance *option* :: (*wellorder*) *wellorder*
 $\langle \text{proof} \rangle$

instantiation *option* :: (*inf*) *inf*
begin

definition *inf-option* **where**

$x \sqcap y = (\text{case } x \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } y \Rightarrow \text{Some } (x \sqcap y)))$

lemma *inf-None-1* [*simp*, *code*]: $\text{None} \sqcap y = \text{None}$
 $\langle \text{proof} \rangle$

lemma *inf-None-2* [*simp*, *code*]: $x \sqcap \text{None} = \text{None}$
 $\langle \text{proof} \rangle$

lemma *inf-Some* [*simp*, *code*]: $\text{Some } x \sqcap \text{Some } y = \text{Some } (x \sqcap y)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instantiation *option* :: (*sup*) *sup*
begin

definition *sup-option* **where**

$x \sqcup y = (\text{case } x \text{ of } \text{None} \Rightarrow y \mid \text{Some } x' \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow x \mid \text{Some } y \Rightarrow \text{Some } (x' \sqcup y)))$

lemma *sup-None-1* [*simp*, *code*]: $\text{None} \sqcup y = y$
 $\langle \text{proof} \rangle$

lemma *sup-None-2* [*simp*, *code*]: $x \sqcup \text{None} = x$
 $\langle \text{proof} \rangle$

lemma *sup-Some* [*simp*, *code*]: $\text{Some } x \sqcup \text{Some } y = \text{Some } (x \sqcup y)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

instance *option* :: (*semilattice-inf*) *semilattice-inf*
 $\langle \text{proof} \rangle$

instance *option* :: (*semilattice-sup*) *semilattice-sup*
 $\langle \text{proof} \rangle$

instance *option* :: (*lattice*) *lattice* $\langle \text{proof} \rangle$

instance *option* :: (*lattice*) *bounded-lattice-bot* $\langle \text{proof} \rangle$

instance *option* :: (*bounded-lattice-top*) *bounded-lattice-top* $\langle \text{proof} \rangle$

instance *option* :: (*bounded-lattice-top*) *bounded-lattice* $\langle \text{proof} \rangle$

instance *option* :: (*distrib-lattice*) *distrib-lattice*
 $\langle \text{proof} \rangle$

instantiation *option* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-option* :: 'a *option set* \Rightarrow 'a *option* **where**
 $\sqcap A = (\text{if } \text{None} \in A \text{ then } \text{None} \text{ else } \text{Some } (\sqcap \text{Option.these } A))$

lemma *None-in-Inf* [*simp*]: $\text{None} \in A \implies \sqcap A = \text{None}$
 $\langle \text{proof} \rangle$

definition *Sup-option* :: 'a *option set* \Rightarrow 'a *option* **where**
 $\sqcup A = (\text{if } A = \{\} \vee A = \{\text{None}\} \text{ then } \text{None} \text{ else } \text{Some } (\sqcup \text{Option.these } A))$

lemma *empty-Sup* [*simp*]: $\sqcup \{\} = \text{None}$
 $\langle \text{proof} \rangle$

lemma *singleton-None-Sup* [*simp*]: $\sqcup \{\text{None}\} = \text{None}$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *Some-Inf*:

$\text{Some } (\prod A) = \prod (\text{Some } 'A)$
 $\langle \text{proof} \rangle$

lemma *Some-Sup*:

$A \neq \{\} \implies \text{Some } (\sqcup A) = \sqcup (\text{Some } 'A)$
 $\langle \text{proof} \rangle$

lemma *Some-INF*:

$\text{Some } (\prod_{x \in A}. f\ x) = (\prod_{x \in A}. \text{Some } (f\ x))$
 $\langle \text{proof} \rangle$

lemma *Some-SUP*:

$A \neq \{\} \implies \text{Some } (\sqcup_{x \in A}. f\ x) = (\sqcup_{x \in A}. \text{Some } (f\ x))$
 $\langle \text{proof} \rangle$

lemma *option-Inf-Sup*: $\prod (\text{Sup } 'A) \leq \sqcup (\text{Inf } ' \{f\ 'A \mid f. \forall Y \in A. f\ Y \in Y\})$

for $A :: ('a :: \text{complete-distrib-lattice } \text{option}) \text{ set set}$
 $\langle \text{proof} \rangle$

instance *option* :: (*complete-distrib-lattice*) *complete-distrib-lattice*
 $\langle \text{proof} \rangle$

instance *option* :: (*complete-linorder*) *complete-linorder* $\langle \text{proof} \rangle$

unbundle *no lattice-syntax*

end

75 Futures and parallel lists for code generated towards Isabelle/ML

theory *Parallel*

imports *Main*

begin

75.1 Futures

datatype $'a \text{ future} = \text{fork unit} \Rightarrow 'a$

primrec *join* :: $'a \text{ future} \Rightarrow 'a$ **where**
 $\text{join } (\text{fork } f) = f\ ()$

lemma *future-eqI* [*intro!*]:

assumes $\text{join } f = \text{join } g$
shows $f = g$

<proof>

code-printing

type-constructor *future* \rightarrow (*Eval*) - *future*
| **constant** *fork* \rightarrow (*Eval*) *Future.fork*
| **constant** *join* \rightarrow (*Eval*) *Future.join*

code-reserved (*Eval*) *Future future*

75.2 Parallel lists

definition *map* :: (*a* \Rightarrow *b*) \Rightarrow *a list* \Rightarrow *b list* **where**
[simp]: map = List.map

definition *forall* :: (*a* \Rightarrow *bool*) \Rightarrow *a list* \Rightarrow *bool* **where**
forall = list-all

lemma *forall-all* *[simp]:*
forall P xs \longleftrightarrow ($\forall x \in \text{set } xs. P x$)
<proof>

definition *exists* :: (*a* \Rightarrow *bool*) \Rightarrow *a list* \Rightarrow *bool* **where**
exists = list-ex

lemma *exists-ex* *[simp]:*
exists P xs \longleftrightarrow ($\exists x \in \text{set } xs. P x$)
<proof>

code-printing

constant *map* \rightarrow (*Eval*) *Par'-List.map*
| **constant** *forall* \rightarrow (*Eval*) *Par'-List.forall*
| **constant** *exists* \rightarrow (*Eval*) *Par'-List.exists*

code-reserved (*Eval*) *Par-List*

hide-const (**open**) *fork join map exists forall*

end

76 Input syntax for pattern aliases (or “as-patterns” in Haskell)

theory *Pattern-Aliases*
imports *Main*
begin

Most functional languages (Haskell, ML, Scala) support aliases in patterns. This allows to refer to a subpattern with a variable name. This theory

implements this using a check phase. It works well for function definitions (see usage below). All features are packed into a **bundle**.

The following caveats should be kept in mind:

- The translation expects a term of the form $f\ x\ y = rhs$, where x and y are patterns that may contain aliases. The result of the translation is a nested *Let*-expression on the right hand side. The code generator *does not* print Isabelle pattern aliases to target language pattern aliases.
- The translation does not process nested equalities; only the top-level equality is translated.
- Terms that do not adhere to the above shape may either stay untranslated or produce an error message. The **fun** command will complain if pattern aliases are left untranslated. In particular, there are no checks whether the patterns are wellformed or linear.
- The corresponding uncheck phase attempts to reverse the translation (no guarantee). The additionally introduced variables are bound using a “fake quantifier” that does not appear in the output.
- To obtain reasonable induction principles in function definitions, the bundle also declares a custom congruence rule for *Let* that only affects **fun**. This congruence rule might lead to an explosion in term size (although that is rare)! In some circumstances (using *let* to destructure tuples), the internal construction of functions stumbles over this rule and prints an error. To mitigate this, either
 - activate the bundle locally (**context includes ... begin**) or
 - rewrite the *let*-expression to use *case*: $let\ (a, b) = x\ in\ (b, a)$ becomes $case\ x\ of\ (a, b) \Rightarrow (b, a)$.
- The bundle also adds the $Let\ ?s\ ?f \equiv ?f\ ?s$ rule to the simpset.

76.1 Definition

consts

$as :: 'a \Rightarrow 'a \Rightarrow 'a$

$fake-quant :: ('a \Rightarrow prop) \Rightarrow prop$

lemma *let-cong-unfolding*: $M = N \Longrightarrow f\ N = g\ N \Longrightarrow Let\ M\ f = Let\ N\ g$

<proof>

translations $P <= CONST\ fake-quant\ (\lambda x. P)$

<ML>

bundle *pattern-aliases* **begin**

notation *as* (**infixr** $\langle =: \rangle$ 1)

$\langle ML \rangle$

declare *let-cong-unfolding* [*fundef-cong*]

declare *Let-def* [*simp*]

end

hide-const *as*

hide-const *fake-quant*

76.2 Usage

context **includes** *pattern-aliases* **begin**

Not very useful for plain definitions, but works anyway.

private definition *test-1* $x (y =: z) = y + z$

lemma *test-1* $x y = y + y$

$\langle proof \rangle$

Very useful for function definitions.

private fun *test-2* **where**

test-2 $(y \# (y' \# ys =: x') =: x) = x @ x' @ x' \mid$

test-2 - = []

lemma *test-2* $(y \# y' \# ys) = (y \# y' \# ys) @ (y' \# ys) @ (y' \# ys)$

$\langle proof \rangle$

$\langle ML \rangle$

end

end

77 Periodic Functions

theory *Periodic-Fun*

imports *Complex-Main*

begin

A locale for periodic functions. The idea is that one proves $f(x + p) = f(x)$ for some period p and gets derived results like $f(x - p) = f(x)$ and $f(x + 2p) = f(x)$ for free.

g and gm are “plus/minus k periods” functions. $g1$ and $gn1$ are “plus/minus one period” functions. This is useful e.g. if the period is one; the lemmas one gets are then $f (x + 1) = f x$ instead of $f (x + 1 * 1) = f x$ etc.

```

locale periodic-fun =
  fixes  $f :: ('a :: \{ring-1\}) \Rightarrow 'b$  and  $g\ gm :: 'a \Rightarrow 'a \Rightarrow 'a$  and  $g1\ gn1 :: 'a \Rightarrow 'a$ 
  assumes plus-1:  $f (g1\ x) = f\ x$ 
  assumes periodic-arg-plus-0:  $g\ x\ 0 = x$ 
  assumes periodic-arg-plus-distrib:  $g\ x\ (of-int\ (m + n)) = g\ (g\ x\ (of-int\ n))\ (of-int\ m)$ 
  assumes plus-1-eq:  $g\ x\ 1 = g1\ x$  and minus-1-eq:  $g\ x\ (-1) = gn1\ x$ 
  and minus-eq:  $g\ x\ (-y) = gm\ x\ y$ 
begin

```

```

lemma plus-of-nat:  $f (g\ x\ (of-nat\ n)) = f\ x$ 
   $\langle proof \rangle$ 

```

```

lemma minus-of-nat:  $f (gm\ x\ (of-nat\ n)) = f\ x$ 
   $\langle proof \rangle$ 

```

```

lemma plus-of-int:  $f (g\ x\ (of-int\ n)) = f\ x$ 
   $\langle proof \rangle$ 

```

```

lemma minus-of-int:  $f (gm\ x\ (of-int\ n)) = f\ x$ 
   $\langle proof \rangle$ 

```

```

lemma plus-numeral:  $f (g\ x\ (numeral\ n)) = f\ x$ 
   $\langle proof \rangle$ 

```

```

lemma minus-numeral:  $f (gm\ x\ (numeral\ n)) = f\ x$ 
   $\langle proof \rangle$ 

```

```

lemma minus-1:  $f (gn1\ x) = f\ x$ 
   $\langle proof \rangle$ 

```

```

lemmas periodic-simps = plus-of-nat minus-of-nat plus-of-int minus-of-int
  plus-numeral minus-numeral plus-1 minus-1

```

```

end

```

Specialised case of the *periodic-fun* locale for periods that are not 1. Gives lemmas $f (x - period) = f x$ etc.

```

locale periodic-fun-simple =
  fixes  $f :: ('a :: \{ring-1\}) \Rightarrow 'b$  and  $period :: 'a$ 
  assumes plus-period:  $f (x + period) = f\ x$ 
begin
sublocale periodic-fun  $f\ \lambda z\ x.\ z + x * period\ \lambda z\ x.\ z - x * period$ 
   $\lambda z.\ z + period\ \lambda z.\ z - period$ 
   $\langle proof \rangle$ 
end

```

Specialised case of the *periodic-fun* locale for period 1. Gives lemmas $f(x - 1) = f x$ etc.

locale *periodic-fun-simple'* =

fixes $f :: ('a :: \{\text{ring-1}\}) \Rightarrow 'b$

assumes *plus-period*: $f(x + 1) = f x$

begin

sublocale *periodic-fun* $f \lambda z x. z + x \lambda z x. z - x \lambda z. z + 1 \lambda z. z - 1$

$\langle \text{proof} \rangle$

lemma *of-nat*: $f(\text{of-nat } n) = f 0 \langle \text{proof} \rangle$

lemma *uminus-of-nat*: $f(-\text{of-nat } n) = f 0 \langle \text{proof} \rangle$

lemma *of-int*: $f(\text{of-int } n) = f 0 \langle \text{proof} \rangle$

lemma *uminus-of-int*: $f(-\text{of-int } n) = f 0 \langle \text{proof} \rangle$

lemma *of-numeral*: $f(\text{numeral } n) = f 0 \langle \text{proof} \rangle$

lemma *of-neg-numeral*: $f(-\text{numeral } n) = f 0 \langle \text{proof} \rangle$

lemma *of-1*: $f 1 = f 0 \langle \text{proof} \rangle$

lemma *of-neg-1*: $f(-1) = f 0 \langle \text{proof} \rangle$

lemmas *periodic-simps'* =

of-nat uminus-of-nat of-int uminus-of-int of-numeral of-neg-numeral of-1 of-neg-1

end

lemma *sin-plus-pi*: $\sin((z :: 'a :: \{\text{real-normed-field}, \text{banach}\}) + \text{of-real } \pi) = -\sin z$
 $\langle \text{proof} \rangle$

lemma *cos-plus-pi*: $\cos((z :: 'a :: \{\text{real-normed-field}, \text{banach}\}) + \text{of-real } \pi) = -\cos z$
 $\langle \text{proof} \rangle$

interpretation *sin*: *periodic-fun-simple* $\sin 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

interpretation *cos*: *periodic-fun-simple* $\cos 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

interpretation *tan*: *periodic-fun-simple* $\tan 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

interpretation *cot*: *periodic-fun-simple* $\cot 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
 $\langle \text{proof} \rangle$

lemma *cos-eq-neg-periodic-intro*:

assumes $x - y = 2 * (\text{of-int } k) * \pi + \pi \vee x + y = 2 * (\text{of-int } k) * \pi + \pi$

shows $\cos x = -\cos y \langle \text{proof} \rangle$

lemma *cos-eq-periodic-intro*:

assumes $x - y = 2 * (\text{of-int } k) * \pi \vee x + y = 2 * (\text{of-int } k) * \pi$

shows $\cos x = \cos y$
 $\langle \text{proof} \rangle$

lemma *cos-eq-arccos-Ex*:
 $\cos x = y \iff -1 \leq y \wedge y \leq 1 \wedge (\exists k::\text{int}. x = \arccos y + 2*k*\pi \vee x = -\arccos y + 2*k*\pi)$ (**is** $?L=?R$)
 $\langle \text{proof} \rangle$

end

78 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view

theory *Poly-Mapping*
imports *Groups-Big-Fun Fun-Lexorder More-List*
begin

78.1 Preliminary: auxiliary operations for *almost everywhere zero*

A central notion for polynomials are functions being *almost everywhere zero*. For these we provide some auxiliary definitions and lemmas.

lemma *finite-mult-not-eq-zero-leftI*:
fixes $f :: 'b \Rightarrow 'a :: \text{mult-zero}$
assumes $\text{finite } \{a. f\ a \neq 0\}$
shows $\text{finite } \{a. g\ a * f\ a \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-mult-not-eq-zero-rightI*:
fixes $f :: 'b \Rightarrow 'a :: \text{mult-zero}$
assumes $\text{finite } \{a. f\ a \neq 0\}$
shows $\text{finite } \{a. f\ a * g\ a \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-mult-not-eq-zero-prodI*:
fixes $f\ g :: 'a \Rightarrow 'b::\text{semiring-0}$
assumes $\text{finite } \{a. f\ a \neq 0\}$ (**is** $\text{finite } ?F$)
assumes $\text{finite } \{b. g\ b \neq 0\}$ (**is** $\text{finite } ?G$)
shows $\text{finite } \{(a, b). f\ a * g\ b \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-not-eq-zero-sumI*:
fixes $f\ g :: 'a::\text{monoid-add} \Rightarrow 'b::\text{semiring-0}$
assumes $\text{finite } \{a. f\ a \neq 0\}$ (**is** $\text{finite } ?F$)
assumes $\text{finite } \{b. g\ b \neq 0\}$ (**is** $\text{finite } ?G$)
shows $\text{finite } \{a + b \mid a\ b. f\ a \neq 0 \wedge g\ b \neq 0\}$ (**is** $\text{finite } ?FG$)
 $\langle \text{proof} \rangle$

lemma *finite-mult-not-eq-zero-sumI*:
fixes $f\ g :: 'a::\text{monoid-add} \Rightarrow 'b::\text{semiring-0}$
assumes $\text{finite } \{a. f\ a \neq 0\}$
assumes $\text{finite } \{b. g\ b \neq 0\}$
shows $\text{finite } \{a + b \mid a\ b. f\ a * g\ b \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-Sum-any-not-eq-zero-weakenI*:
assumes $\text{finite } \{a. \exists b. f\ a\ b \neq 0\}$
shows $\text{finite } \{a. \text{Sum-any } (f\ a) \neq 0\}$
 $\langle \text{proof} \rangle$

context *zero*
begin

definition $\text{when} :: 'a \Rightarrow \text{bool} \Rightarrow 'a$ (**infixl** $\langle \text{when} \rangle$ 20)
where
 $(a\ \text{when}\ P) = (\text{if } P \text{ then } a \text{ else } 0)$

Case distinctions always complicate matters, particularly when nested. The (when) operation allows to minimise these if 0 is the false-case value and makes proof obligations much more readable.

lemma *when [simp]*:
 $P \Longrightarrow (a\ \text{when}\ P) = a$
 $\neg P \Longrightarrow (a\ \text{when}\ P) = 0$
 $\langle \text{proof} \rangle$

lemma *when-simps [simp]*:
 $(a\ \text{when}\ \text{True}) = a$
 $(a\ \text{when}\ \text{False}) = 0$
 $\langle \text{proof} \rangle$

lemma *when-cong*:
assumes $P \longleftrightarrow Q$
and $Q \Longrightarrow a = b$
shows $(a\ \text{when}\ P) = (b\ \text{when}\ Q)$
 $\langle \text{proof} \rangle$

lemma *zero-when [simp]*:
 $(0\ \text{when}\ P) = 0$
 $\langle \text{proof} \rangle$

lemma *when-when*:
 $(a\ \text{when}\ P\ \text{when}\ Q) = (a\ \text{when}\ P \wedge Q)$
 $\langle \text{proof} \rangle$

lemma *when-commute*:
 $(a\ \text{when}\ Q\ \text{when}\ P) = (a\ \text{when}\ P\ \text{when}\ Q)$

$\langle \text{proof} \rangle$

lemma *when-neg-zero* [simp]:

$$(a \text{ when } P) \neq 0 \iff P \wedge a \neq 0$$

$\langle \text{proof} \rangle$

end

context *monoid-add*

begin

lemma *when-add-distrib*:

$$(a + b \text{ when } P) = (a \text{ when } P) + (b \text{ when } P)$$

$\langle \text{proof} \rangle$

end

context *semiring-1*

begin

lemma *zero-power-eq*:

$$0 \wedge n = (1 \text{ when } n = 0)$$

$\langle \text{proof} \rangle$

end

context *comm-monoid-add*

begin

lemma *Sum-any-when-equal* [simp]:

$$(\sum a. (f a \text{ when } a = b)) = f b$$

$\langle \text{proof} \rangle$

lemma *Sum-any-when-equal'* [simp]:

$$(\sum a. (f a \text{ when } b = a)) = f b$$

$\langle \text{proof} \rangle$

lemma *Sum-any-when-independent*:

$$(\sum a. g a \text{ when } P) = ((\sum a. g a) \text{ when } P)$$

$\langle \text{proof} \rangle$

lemma *Sum-any-when-dependent-prod-right*:

$$(\sum (a, b). g a \text{ when } b = h a) = (\sum a. g a)$$

$\langle \text{proof} \rangle$

lemma *Sum-any-when-dependent-prod-left*:

$$(\sum (a, b). g b \text{ when } a = h b) = (\sum b. g b)$$

$\langle \text{proof} \rangle$

end

context *cancel-comm-monoid-add*
begin

lemma *when-diff-distrib*:
 $(a - b \text{ when } P) = (a \text{ when } P) - (b \text{ when } P)$
 $\langle \text{proof} \rangle$

end

context *group-add*
begin

lemma *when-uminus-distrib*:
 $(- a \text{ when } P) = - (a \text{ when } P)$
 $\langle \text{proof} \rangle$

end

context *mult-zero*
begin

lemma *mult-when*:
 $a * (b \text{ when } P) = (a * b \text{ when } P)$
 $\langle \text{proof} \rangle$

lemma *when-mult*:
 $(a \text{ when } P) * b = (a * b \text{ when } P)$
 $\langle \text{proof} \rangle$

end

78.2 Type definition

The following type is of central importance:

typedef (**overloaded**) (*'a, 'b*) *poly-mapping* ($\langle (- \Rightarrow_0 / -) \rangle [1, 0] 0$) =
 $\{f :: 'a \Rightarrow 'b :: \text{zero. finite } \{x. f x \neq 0\}\}$
morphisms *lookup Abs-poly-mapping*
 $\langle \text{proof} \rangle$

declare *lookup-inverse* [*simp*]
declare *lookup-inject* [*simp*]

lemma *lookup-Abs-poly-mapping* [*simp*]:
 $\text{finite } \{x. f x \neq 0\} \implies \text{lookup } (\text{Abs-poly-mapping } f) = f$
 $\langle \text{proof} \rangle$

lemma *finite-lookup* [*simp*]:

finite $\{k. \text{lookup } f \ k \neq 0\}$
 $\langle \text{proof} \rangle$

lemma *finite-lookup-nat* [simp]:
fixes $f :: 'a \Rightarrow_0 \text{nat}$
shows *finite* $\{k. 0 < \text{lookup } f \ k\}$
 $\langle \text{proof} \rangle$

lemma *poly-mapping-eqI*:
assumes $\bigwedge k. \text{lookup } f \ k = \text{lookup } g \ k$
shows $f = g$
 $\langle \text{proof} \rangle$

lemma *poly-mapping-eq-iff*: $a = b \longleftrightarrow \text{lookup } a = \text{lookup } b$
 $\langle \text{proof} \rangle$

We model the universe of functions being *almost everywhere zero* by means of a separate type $'a \Rightarrow_0 'b$. For convenience we provide a suggestive infix syntax which is a variant of the usual function space syntax. Conversion between both types happens through the morphisms

lookup

Abs-poly-mapping

satisfying

Abs-poly-mapping (*lookup* $?x$) = $?x$

finite $\{x. ?f \ x \neq 0\} \implies \text{lookup } (\text{Abs-poly-mapping } ?f) = ?f$

Luckily, we have rarely to deal with those low-level morphisms explicitly but rely on Isabelle’s *lifting* package with its method *transfer* and its specification tool *lift-definition*.

setup-lifting *type-definition-poly-mapping*

code-datatype *Abs-poly-mapping*— FIXME? workaround for preventing *code-abstype* setup

$'a \Rightarrow_0 'b$ serves distinctive purposes:

1. A clever nesting as $(\text{nat} \Rightarrow_0 \text{nat}) \Rightarrow_0 'a$ later in theory *MPoly* gives a suitable representation type for polynomials *almost for free*: Interpreting $\text{nat} \Rightarrow_0 \text{nat}$ as a mapping from variable identifiers to exponents yields monomials, and the whole type maps monomials to coefficients. Lets call this the *ultimate interpretation*.
2. A further more specialised type isomorphic to $\text{nat} \Rightarrow_0 'a$ is apt to direct implementation using code generation [1].

Note that despite the names *mapping* and *lookup* suggest something implementation-near, it is best to keep $'a \Rightarrow_0 'b$ as an abstract *algebraic* type providing operations like *addition*, *multiplication* without any notion of key-order, data structures etc. This implementations-specific notions are easily introduced later for particular implementations but do not provide any gain for specifying logical properties of polynomials.

78.3 Additive structure

The additive structure covers the usual operations 0 , $+$ and (unary and binary) $-$. Recalling the ultimate interpretation, it is obvious that these have just lift the corresponding operations on values to mappings.

Isabelle has a rich hierarchy of algebraic type classes, and in such situations of pointwise lifting a typical pattern is to have instantiations for a considerable number of type classes.

The operations themselves are specified using *lift-definition*, where the proofs of the *almost everywhere zero* property can be significantly involved.

The *lookup* operation is supposed to be usable explicitly (unless in other situations where the morphisms between types are somehow internal to the *lifting* package). Hence it is good style to provide explicit rewrite rules how *lookup* acts on operations immediately.

instantiation *poly-mapping* :: (type, zero) zero
begin

lift-definition *zero-poly-mapping* :: $'a \Rightarrow_0 'b$
 is $\lambda k. 0$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *Abs-poly-mapping* [simp]: *Abs-poly-mapping* ($\lambda k. 0$) = 0
 $\langle \text{proof} \rangle$

lemma *lookup-zero* [simp]: *lookup* 0 k = 0
 $\langle \text{proof} \rangle$

instantiation *poly-mapping* :: (type, monoid-add) monoid-add
begin

lift-definition *plus-poly-mapping* ::
 $('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow_0 'b$
 is $\lambda f1\ f2\ k. f1\ k + f2\ k$
 $\langle \text{proof} \rangle$

instance
 $\langle proof \rangle$

end

lemma *lookup-add*: $lookup\ (f + g)\ k = lookup\ f\ k + lookup\ g\ k$
 $\langle proof \rangle$

instance *poly-mapping* :: (type, comm-monoid-add) comm-monoid-add
 $\langle proof \rangle$

lemma *lookup-sum*: $lookup\ (sum\ pp\ X)\ i = sum\ (\lambda x. lookup\ (pp\ x)\ i)\ X$
 $\langle proof \rangle$

instantiation *poly-mapping* :: (type, cancel-comm-monoid-add) cancel-comm-monoid-add
begin

lift-definition *minus-poly-mapping* :: ($'a \Rightarrow_0 'b$) \Rightarrow ($'a \Rightarrow_0 'b$) \Rightarrow $'a \Rightarrow_0 'b$
is $\lambda f1\ f2\ k. f1\ k - f2\ k$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

instantiation *poly-mapping* :: (type, ab-group-add) ab-group-add
begin

lift-definition *uminus-poly-mapping* :: ($'a \Rightarrow_0 'b$) \Rightarrow $'a \Rightarrow_0 'b$
is *uminus*
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma *lookup-uminus* [*simp*]:
 $lookup\ (-\ f)\ k = -\ lookup\ f\ k$
 $\langle proof \rangle$

lemma *lookup-minus*:
 $lookup\ (f - g)\ k = lookup\ f\ k - lookup\ g\ k$
 $\langle proof \rangle$

78.4 Multiplicative structure

instantiation *poly-mapping* :: (zero, zero-neq-one) zero-neq-one
begin

lift-definition *one-poly-mapping* :: 'a \Rightarrow_0 'b
 is $\lambda k. 1$ when $k = 0$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma *lookup-one*: *lookup 1 k = (1 when k = 0)*
 $\langle proof \rangle$

lemma *lookup-one-zero* [simp]:
lookup 1 0 = 1
 $\langle proof \rangle$

definition *prod-fun* :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a::monoid-add \Rightarrow 'b::semiring-0
where
*prod-fun f1 f2 k = ($\sum l. f1\ l * (\sum q. (f2\ q$ when $k = l + q))$)*

lemma *prod-fun-unfold-prod*:
fixes *f g* :: 'a :: monoid-add \Rightarrow 'b::semiring-0
assumes *fin-f*: *finite {a. f a \neq 0}*
assumes *fin-g*: *finite {b. g b \neq 0}*
shows *prod-fun f g k = ($\sum (a, b). f\ a * g\ b$ when $k = a + b$)*
 $\langle proof \rangle$

lemma *finite-prod-fun*:
fixes *f1 f2* :: 'a :: monoid-add \Rightarrow 'b :: semiring-0
assumes *fin1*: *finite {l. f1 l \neq 0}*
and *fin2*: *finite {q. f2 q \neq 0}*
shows *finite {k. prod-fun f1 f2 k \neq 0}*
 $\langle proof \rangle$

instantiation *poly-mapping* :: (monoid-add, semiring-0) semiring-0
begin

lift-definition *times-poly-mapping* :: ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow_0 'b
 is *prod-fun*
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma *lookup-mult*:

lookup ($f * g$) $k = (\sum l. \text{lookup } f \ l * (\sum q. \text{lookup } g \ q \text{ when } k = l + q))$
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0*) *comm-semiring-0*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*monoid-add*, *semiring-0-cancel*) *semiring-0-cancel*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*monoid-add*, *semiring-1*) *semiring-1*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-1*) *comm-semiring-1*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*monoid-add*, *semiring-1-cancel*) *semiring-1-cancel*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*monoid-add*, *ring*) *ring*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring*) *comm-ring*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*monoid-add*, *ring-1*) *ring-1*
 $\langle \text{proof} \rangle$

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring-1*) *comm-ring-1*
 $\langle \text{proof} \rangle$

78.5 Single-point mappings

lift-definition *single* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_0 'b :: \text{zero}$
 $\text{is } \lambda k \ v \ k'. (v \text{ when } k = k')$
 $\langle \text{proof} \rangle$

lemma *inj-single* [*iff*]:
 $\text{inj } (\text{single } k)$
 $\langle \text{proof} \rangle$

lemma *lookup-single*:
 $\text{lookup } (\text{single } k \ v) \ k' = (v \text{ when } k = k')$
 $\langle \text{proof} \rangle$

lemma *lookup-single-eq* [*simp*]:
 $\text{lookup } (\text{single } k \ v) \ k = v$
 $\langle \text{proof} \rangle$

lemma *lookup-single-not-eq*:
 $k \neq k' \implies \text{lookup } (\text{single } k \ v) \ k' = 0$
 $\langle \text{proof} \rangle$

lemma *single-zero* [*simp*]:
 $\text{single } k \ 0 = 0$
 $\langle \text{proof} \rangle$

lemma *single-one* [*simp*]:
 $\text{single } 0 \ 1 = 1$
 $\langle \text{proof} \rangle$

lemma *single-add*:
 $\text{single } k \ (a + b) = \text{single } k \ a + \text{single } k \ b$
 $\langle \text{proof} \rangle$

lemma *single-uminus*:
 $\text{single } k \ (-a) = - \text{single } k \ a$
 $\langle \text{proof} \rangle$

lemma *single-diff*:
 $\text{single } k \ (a - b) = \text{single } k \ a - \text{single } k \ b$
 $\langle \text{proof} \rangle$

lemma *single-numeral* [*simp*]:
 $\text{single } 0 \ (\text{numeral } n) = \text{numeral } n$
 $\langle \text{proof} \rangle$

lemma *lookup-numeral*:
 $\text{lookup } (\text{numeral } n) \ k = (\text{numeral } n \text{ when } k = 0)$
 $\langle \text{proof} \rangle$

lemma *single-of-nat* [*simp*]:
 $\text{single } 0 \ (\text{of-nat } n) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *lookup-of-nat*:
 $\text{lookup } (\text{of-nat } n) \ k = (\text{of-nat } n \text{ when } k = 0)$
 $\langle \text{proof} \rangle$

lemma *of-nat-single*:
 $\text{of-nat} = \text{single } 0 \circ \text{of-nat}$
 $\langle \text{proof} \rangle$

lemma *mult-single*:

single *k a * single l b = single (k + l) (a * b)*
 ⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-char-0*) *semiring-char-0*
 ⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *ring-char-0*) *ring-char-0*
 ⟨*proof*⟩

lemma *single-of-int* [*simp*]:
single 0 (of-int k) = of-int k
 ⟨*proof*⟩

lemma *lookup-of-int*:
lookup (of-int l) k = (of-int l when k = 0)
 ⟨*proof*⟩

78.6 Integral domains

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *semiring-no-zero-divisors*)
semiring-no-zero-divisors

The *linorder* constraint is a pragmatic device for the proof — maybe it can be dropped

⟨*proof*⟩

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *ring-no-zero-divisors*)
ring-no-zero-divisors
 ⟨*proof*⟩

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *ring-1-no-zero-divisors*)
ring-1-no-zero-divisors
 ⟨*proof*⟩

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *idom*) *idom*
 ⟨*proof*⟩

78.7 Mapping order

instantiation *poly-mapping* :: (*linorder*, {*zero*, *linorder*}) *linorder*
begin

lift-definition *less-poly-mapping* :: ('*a* ⇒₀ '*b*) ⇒ ('*a* ⇒₀ '*b*) ⇒ *bool*
is *less-fun*
 ⟨*proof*⟩

lift-definition *less-eq-poly-mapping* :: ('*a* ⇒₀ '*b*) ⇒ ('*a* ⇒₀ '*b*) ⇒ *bool*
is λ*f g*. *less-fun f g* ∨ *f = g*
 ⟨*proof*⟩

instance $\langle proof \rangle$

end

instance $poly\text{-}mapping :: (linorder, \{ordered\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, linorder\}) ordered\text{-}ab\text{-}semigroup\text{-}add$
 $\langle proof \rangle$

instance $poly\text{-}mapping :: (linorder, \{ordered\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, cancel\text{-}comm\text{-}monoid\text{-}add, linorder\}) linordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add$
 $\langle proof \rangle$

instance $poly\text{-}mapping :: (linorder, \{ordered\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, cancel\text{-}comm\text{-}monoid\text{-}add, linorder\}) ordered\text{-}comm\text{-}monoid\text{-}add$
 $\langle proof \rangle$

instance $poly\text{-}mapping :: (linorder, \{ordered\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, cancel\text{-}comm\text{-}monoid\text{-}add, linorder\}) ordered\text{-}cancel\text{-}comm\text{-}monoid\text{-}add$
 $\langle proof \rangle$

instance $poly\text{-}mapping :: (linorder, linordered\text{-}ab\text{-}group\text{-}add) linordered\text{-}ab\text{-}group\text{-}add$
 $\langle proof \rangle$

For pragmatism we leave out the final elements in the hierarchy: *linordered-ring*, *linordered-ring-strict*, *linordered-idom*; remember that the order instance is a mere technical device, not a deeper algebraic property.

78.8 Fundamental mapping notions

lift-definition $keys :: ('a \Rightarrow_0 'b::zero) \Rightarrow 'a\ set$
 $is\ \lambda f. \{k. f\ k \neq 0\} \langle proof \rangle$

lift-definition $range :: ('a \Rightarrow_0 'b::zero) \Rightarrow 'b\ set$
 $is\ \lambda f :: 'a \Rightarrow 'b. Set.range\ f - \{0\} \langle proof \rangle$

lemma *finite-keys* [simp]:
 $finite\ (keys\ f)$
 $\langle proof \rangle$

lemma *not-in-keys-iff-lookup-eq-zero*:
 $k \notin keys\ f \longleftrightarrow lookup\ f\ k = 0$
 $\langle proof \rangle$

lemma *lookup-not-eq-zero-eq-in-keys*:
 $lookup\ f\ k \neq 0 \longleftrightarrow k \in keys\ f$
 $\langle proof \rangle$

lemma *lookup-eq-zero-in-keys-contradict* [dest]:
 $lookup\ f\ k = 0 \implies \neg k \in keys\ f$

$\langle \text{proof} \rangle$

lemma *finite-range* [simp]: *finite* (*Poly-Mapping.range* *p*)
 $\langle \text{proof} \rangle$

lemma *in-keys-lookup-in-range* [simp]:
 $k \in \text{keys } f \implies \text{lookup } f \ k \in \text{range } f$
 $\langle \text{proof} \rangle$

lemma *in-keys-iff*: $x \in (\text{keys } s) = (\text{lookup } s \ x \neq 0)$
 $\langle \text{proof} \rangle$

lemma *keys-zero* [simp]:
 $\text{keys } 0 = \{\}$
 $\langle \text{proof} \rangle$

lemma *range-zero* [simp]:
 $\text{range } 0 = \{\}$
 $\langle \text{proof} \rangle$

lemma *keys-add*:
 $\text{keys } (f + g) \subseteq \text{keys } f \cup \text{keys } g$
 $\langle \text{proof} \rangle$

lemma *keys-one* [simp]:
 $\text{keys } 1 = \{0\}$
 $\langle \text{proof} \rangle$

lemma *range-one* [simp]:
 $\text{range } 1 = \{1\}$
 $\langle \text{proof} \rangle$

lemma *keys-single* [simp]:
 $\text{keys } (\text{single } k \ v) = (\text{if } v = 0 \text{ then } \{\} \text{ else } \{k\})$
 $\langle \text{proof} \rangle$

lemma *range-single* [simp]:
 $\text{range } (\text{single } k \ v) = (\text{if } v = 0 \text{ then } \{\} \text{ else } \{v\})$
 $\langle \text{proof} \rangle$

lemma *keys-mult*:
 $\text{keys } (f * g) \subseteq \{a + b \mid a \ b. \ a \in \text{keys } f \wedge b \in \text{keys } g\}$
 $\langle \text{proof} \rangle$

lemma *setsum-keys-plus-distrib*:
assumes *hom-0*: $\bigwedge k. \ f \ k \ 0 = 0$
and *hom-plus*: $\bigwedge k. \ k \in \text{Poly-Mapping.keys } p \cup \text{Poly-Mapping.keys } q \implies f \ k$
 $(\text{Poly-Mapping.lookup } p \ k + \text{Poly-Mapping.lookup } q \ k) = f \ k \ (\text{Poly-Mapping.lookup } p \ k) + f \ k \ (\text{Poly-Mapping.lookup } q \ k)$

shows

$$\begin{aligned} & (\sum_{k \in \text{Poly-Mapping.keys } (p + q)}. f \ k \ (\text{Poly-Mapping.lookup } (p + q) \ k)) = \\ & (\sum_{k \in \text{Poly-Mapping.keys } p}. f \ k \ (\text{Poly-Mapping.lookup } p \ k)) + \\ & (\sum_{k \in \text{Poly-Mapping.keys } q}. f \ k \ (\text{Poly-Mapping.lookup } q \ k)) \\ & \text{(is ?lhs = ?p + ?q)} \\ & \langle \text{proof} \rangle \end{aligned}$$

78.9 Degree

definition *degree* :: (nat \Rightarrow_0 'a::zero) \Rightarrow nat

where

degree *f* = Max (insert 0 (Suc ‘ keys *f*))

lemma *degree-zero* [simp]:

degree 0 = 0

$\langle \text{proof} \rangle$

lemma *degree-one* [simp]:

degree 1 = 1

$\langle \text{proof} \rangle$

lemma *degree-single-zero* [simp]:

degree (single *k* 0) = 0

$\langle \text{proof} \rangle$

lemma *degree-single-not-zero* [simp]:

$v \neq 0 \implies \text{degree } (\text{single } k \ v) = \text{Suc } k$

$\langle \text{proof} \rangle$

lemma *degree-zero-iff* [simp]:

degree *f* = 0 \longleftrightarrow *f* = 0

$\langle \text{proof} \rangle$

lemma *degree-greater-zero-in-keys*:

assumes 0 < *degree* *f*

shows *degree* *f* - 1 \in keys *f*

$\langle \text{proof} \rangle$

lemma *in-keys-less-degree*:

$n \in \text{keys } f \implies n < \text{degree } f$

$\langle \text{proof} \rangle$

lemma *beyond-degree-lookup-zero*:

degree *f* \leq *n* \implies lookup *f* *n* = 0

$\langle \text{proof} \rangle$

lemma *degree-add*:

degree (*f* + *g*) \leq max (*degree* *f*) (Poly-Mapping.degree *g*)

$\langle \text{proof} \rangle$

lemma *sorted-list-of-set-keys*:

sorted-list-of-set (*keys f*) = *filter* ($\lambda k. k \in \text{keys } f$) [*0*..*degree f*] (**is** - = ?*r*)
 ⟨*proof*⟩

78.10 Inductive structure

lift-definition *update* :: '*a* \Rightarrow '*b* \Rightarrow ('*a* \Rightarrow_0 '*b*::*zero*) \Rightarrow '*a* \Rightarrow_0 '*b*
is $\lambda k \ v \ f. f(k := v)$
 ⟨*proof*⟩

lemma *update-induct* [*case-names const update*]:

assumes *const*': *P 0*
assumes *update*': $\bigwedge f \ a \ b. a \notin \text{keys } f \implies b \neq 0 \implies P \ f \implies P \ (\text{update } a \ b \ f)$
shows *P f*
 ⟨*proof*⟩

lemma *lookup-update*:

lookup (*update k v f*) *k*' = (if *k* = *k*' then *v* else *lookup f k*)
 ⟨*proof*⟩

lemma *keys-update*:

keys (*update k v f*) = (if *v* = 0 then *keys f* - {*k*} else *insert k* (*keys f*))
 ⟨*proof*⟩

78.11 Quasi-functorial structure

lift-definition *map* :: ('*b*::*zero* \Rightarrow '*c*::*zero*)
 \Rightarrow ('*a* \Rightarrow_0 '*b*) \Rightarrow ('*a* \Rightarrow_0 '*c*::*zero*)
is $\lambda g \ f \ k. g \ (f \ k)$ when *f k* $\neq 0$
 ⟨*proof*⟩

context

fixes *f* :: '*b* \Rightarrow '*a*
assumes *inj-f*: *inj f*
begin

lift-definition *map-key* :: ('*a* \Rightarrow_0 '*c*::*zero*) \Rightarrow '*b* \Rightarrow_0 '*c*
is $\lambda p. p \circ f$
 ⟨*proof*⟩

end

lemma *map-key-compose*:

assumes [*transfer-rule*]: *inj f inj g*
shows *map-key f* (*map-key g p*) = *map-key* (*g* \circ *f*) *p*
 ⟨*proof*⟩

lemma *map-key-id*:

map-key ($\lambda x. x$) *p* = *p*

$\langle proof \rangle$

context

fixes $f :: 'a \Rightarrow 'b$

assumes $inj\text{-}f$ [transfer-rule]: $inj\ f$

begin

lemma *map-key-map*:

$map\text{-}key\ f\ (map\ g\ p) = map\ g\ (map\text{-}key\ f\ p)$

$\langle proof \rangle$

lemma *map-key-plus*:

$map\text{-}key\ f\ (p + q) = map\text{-}key\ f\ p + map\text{-}key\ f\ q$

$\langle proof \rangle$

lemma *keys-map-key*:

$keys\ (map\text{-}key\ f\ p) = f\ \text{--}\text{'}\ keys\ p$

$\langle proof \rangle$

lemma *map-key-zero* [simp]:

$map\text{-}key\ f\ 0 = 0$

$\langle proof \rangle$

lemma *map-key-single* [simp]:

$map\text{-}key\ f\ (single\ (f\ k)\ v) = single\ k\ v$

$\langle proof \rangle$

end

lemma *mult-map-scale-conv-mult*: $map\ ((*)\ s)\ p = single\ 0\ s * p$

$\langle proof \rangle$

lemma *map-single* [simp]:

$(c = 0 \implies f\ 0 = 0) \implies map\ f\ (single\ x\ c) = single\ x\ (f\ c)$

$\langle proof \rangle$

lemma *map-eq-zero-iff*: $map\ f\ p = 0 \longleftrightarrow (\forall k \in keys\ p. f\ (lookup\ p\ k) = 0)$

$\langle proof \rangle$

78.12 Canonical dense representation of $nat \Rightarrow_0 'a$

abbreviation *no-trailing-zeros* :: $'a :: zero\ list \Rightarrow bool$

where

$no\text{-}trailing\text{-}zeros \equiv no\text{-}trailing\ ((=)\ 0)$

lift-definition *nth* :: $'a\ list \Rightarrow (nat \Rightarrow_0 'a::zero)$

is *nth-default* 0

$\langle proof \rangle$

The opposite direction is directly specified on (later) type *nat-mapping*.

lemma *nth-Nil* [simp]:

$$\text{nth } [] = 0$$

$\langle \text{proof} \rangle$

lemma *nth-singleton* [simp]:

$$\text{nth } [v] = \text{single } 0 \ v$$

$\langle \text{proof} \rangle$

lemma *nth-replicate* [simp]:

$$\text{nth } (\text{replicate } n \ 0 \ @ \ [v]) = \text{single } n \ v$$

$\langle \text{proof} \rangle$

lemma *nth-strip-while* [simp]:

$$\text{nth } (\text{strip-while } ((=) \ 0) \ xs) = \text{nth } xs$$

$\langle \text{proof} \rangle$

lemma *nth-strip-while'* [simp]:

$$\text{nth } (\text{strip-while } (\lambda k. \ k = 0) \ xs) = \text{nth } xs$$

$\langle \text{proof} \rangle$

lemma *nth-eq-iff*:

$$\text{nth } xs = \text{nth } ys \longleftrightarrow \text{strip-while } (\text{HOL.eq } 0) \ xs = \text{strip-while } (\text{HOL.eq } 0) \ ys$$

$\langle \text{proof} \rangle$

lemma *lookup-nth* [simp]:

$$\text{lookup } (\text{nth } xs) = \text{nth-default } 0 \ xs$$

$\langle \text{proof} \rangle$

lemma *keys-nth* [simp]:

$$\text{keys } (\text{nth } xs) = \text{fst } ' \{ (n, v) \in \text{set } (\text{enumerate } 0 \ xs). \ v \neq 0 \}$$

$\langle \text{proof} \rangle$

lemma *range-nth* [simp]:

$$\text{range } (\text{nth } xs) = \text{set } xs - \{0\}$$

$\langle \text{proof} \rangle$

lemma *degree-nth*:

$$\text{no-trailing-zeros } xs \implies \text{degree } (\text{nth } xs) = \text{length } xs$$

$\langle \text{proof} \rangle$

lemma *nth-trailing-zeros* [simp]:

$$\text{nth } (xs \ @ \ \text{replicate } n \ 0) = \text{nth } xs$$

$\langle \text{proof} \rangle$

lemma *nth-idem*:

$$\text{nth } (\text{List.map } (\text{lookup } f) \ [0..<\text{degree } f]) = f$$

$\langle \text{proof} \rangle$

lemma *nth-idem-bound*:

assumes $\text{degree } f \leq n$
shows $\text{nth } (List.map (lookup f) [0..<n]) = f$
 $\langle \text{proof} \rangle$

78.13 Canonical sparse representation of $'a \Rightarrow_0 'b$

lift-definition $\text{the-value} :: ('a \times 'b) \text{ list} \Rightarrow 'a \Rightarrow_0 'b::\text{zero}$
is $\lambda xs \ k. \text{case map-of } xs \ k \text{ of } None \Rightarrow 0 \mid Some \ v \Rightarrow v$
 $\langle \text{proof} \rangle$

definition $\text{items} :: ('a::\text{linorder} \Rightarrow_0 'b::\text{zero}) \Rightarrow ('a \times 'b) \text{ list}$
where

$\text{items } f = List.map (\lambda k. (k, lookup f k)) (\text{sorted-list-of-set } (keys f))$

For the canonical sparse representation we provide both directions of morphisms since the specification of ordered association lists in theory *OAL-ist* will support arbitrary linear orders *linorder* as keys, not just natural numbers *nat*.

lemma $\text{the-value-items} [simp]$:
 $\text{the-value } (\text{items } f) = f$
 $\langle \text{proof} \rangle$

lemma lookup-the-value :
 $\text{lookup } (\text{the-value } xs) \ k = (\text{case map-of } xs \ k \text{ of } None \Rightarrow 0 \mid Some \ v \Rightarrow v)$
 $\langle \text{proof} \rangle$

lemma items-the-value :
assumes $\text{sorted } (List.map \text{fst } xs)$ **and** $\text{distinct } (List.map \text{fst } xs)$ **and** $0 \notin \text{snd 'set } xs$
shows $\text{items } (\text{the-value } xs) = xs$
 $\langle \text{proof} \rangle$

lemma $\text{the-value-Nil} [simp]$:
 $\text{the-value } [] = 0$
 $\langle \text{proof} \rangle$

lemma $\text{the-value-Cons} [simp]$:
 $\text{the-value } (x \# xs) = \text{update } (\text{fst } x) (\text{snd } x) (\text{the-value } xs)$
 $\langle \text{proof} \rangle$

lemma $\text{items-zero} [simp]$:
 $\text{items } 0 = []$
 $\langle \text{proof} \rangle$

lemma $\text{items-one} [simp]$:
 $\text{items } 1 = [(0, 1)]$
 $\langle \text{proof} \rangle$

lemma $\text{items-single} [simp]$:

$items\ (single\ k\ v) = (if\ v = 0\ then\ []\ else\ [(k,\ v)])$
 $\langle proof \rangle$

lemma *in-set-items-iff* [simp]:
 $(k,\ v) \in set\ (items\ f) \longleftrightarrow k \in keys\ f \wedge lookup\ f\ k = v$
 $\langle proof \rangle$

78.14 Size estimation

context
fixes $f :: 'a \Rightarrow nat$
and $g :: 'b :: zero \Rightarrow nat$
begin

definition *poly-mapping-size* :: $('a \Rightarrow_0 'b) \Rightarrow nat$
where
 $poly_mapping_size\ m = g\ 0 + (\sum k \in keys\ m.\ Suc\ (f\ k + g\ (lookup\ m\ k)))$

lemma *poly-mapping-size-0* [simp]:
 $poly_mapping_size\ 0 = g\ 0$
 $\langle proof \rangle$

lemma *poly-mapping-size-single* [simp]:
 $poly_mapping_size\ (single\ k\ v) = (if\ v = 0\ then\ g\ 0\ else\ g\ 0 + f\ k + g\ v + 1)$
 $\langle proof \rangle$

lemma *keys-less-poly-mapping-size*:
 $k \in keys\ m \implies f\ k + g\ (lookup\ m\ k) < poly_mapping_size\ m$
 $\langle proof \rangle$

lemma *lookup-le-poly-mapping-size*:
 $g\ (lookup\ m\ k) \leq poly_mapping_size\ m$
 $\langle proof \rangle$

lemma *poly-mapping-size-estimation*:
 $k \in keys\ m \implies y \leq f\ k + g\ (lookup\ m\ k) \implies y < poly_mapping_size\ m$
 $\langle proof \rangle$

lemma *poly-mapping-size-estimation2*:
assumes $v \in range\ m$ **and** $y \leq g\ v$
shows $y < poly_mapping_size\ m$
 $\langle proof \rangle$

end

lemma *poly-mapping-size-one* [simp]:
 $poly_mapping_size\ f\ g\ 1 = g\ 0 + f\ 0 + g\ 1 + 1$
 $\langle proof \rangle$

lemma *poly-mapping-size-cong* [fundef-cong]:
 $m = m' \implies g \ 0 = g' \ 0 \implies (\bigwedge k. k \in \text{keys } m' \implies f \ k = f' \ k)$
 $\implies (\bigwedge v. v \in \text{range } m' \implies g \ v = g' \ v)$
 $\implies \text{poly-mapping-size } f \ g \ m = \text{poly-mapping-size } f' \ g' \ m'$
 ⟨proof⟩

instantiation *poly-mapping* :: (type, zero) size
begin

definition *size* = *poly-mapping-size* ($\lambda\cdot. 0$) ($\lambda\cdot. 0$)

instance ⟨proof⟩

end

78.15 Further mapping operations and properties

It is like in algebra: there are many definitions, some are also used

lift-definition *mapp* ::
 $('a \Rightarrow 'b :: \text{zero} \Rightarrow 'c :: \text{zero}) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c)$
is $\lambda f \ p \ k. (\text{if } k \in \text{keys } p \text{ then } f \ k \ (\text{lookup } p \ k) \text{ else } 0)$
 ⟨proof⟩

lemma *mapp-cong* [fundef-cong]:
 $\llbracket m = m'; \bigwedge k. k \in \text{keys } m' \implies f \ k \ (\text{lookup } m' \ k) = f' \ k \ (\text{lookup } m' \ k) \rrbracket$
 $\implies \text{mapp } f \ m = \text{mapp } f' \ m'$
 ⟨proof⟩

lemma *lookup-mapp*:
 $\text{lookup } (\text{mapp } f \ p) \ k = (f \ k \ (\text{lookup } p \ k) \text{ when } k \in \text{keys } p)$
 ⟨proof⟩

lemma *keys-mapp-subset*: $\text{keys } (\text{mapp } f \ p) \subseteq \text{keys } p$
 ⟨proof⟩

78.16 Free Abelian Groups Over a Type

abbreviation *frag-of* :: $'a \Rightarrow 'a \Rightarrow_0 \text{int}$
where *frag-of* $c \equiv \text{Poly-Mapping.single } c \ (1::\text{int})$

lemma *lookup-frag-of* [simp]:
 $\text{Poly-Mapping.lookup}(\text{frag-of } c) = (\lambda x. \text{if } x = c \text{ then } 1 \text{ else } 0)$
 ⟨proof⟩

lemma *frag-of-nonzero* [simp]: $\text{frag-of } a \neq 0$
 ⟨proof⟩

definition *frag-cmul* :: $\text{int} \Rightarrow ('a \Rightarrow_0 \text{int}) \Rightarrow ('a \Rightarrow_0 \text{int})$
where *frag-cmul* $c \ a = \text{Abs-poly-mapping } (\lambda x. c * \text{Poly-Mapping.lookup } a \ x)$

lemma *frag-cmul-zero* [*simp*]: *frag-cmul* 0 *x* = 0
 ⟨*proof*⟩

lemma *frag-cmul-zero2* [*simp*]: *frag-cmul* *c* 0 = 0
 ⟨*proof*⟩

lemma *frag-cmul-one* [*simp*]: *frag-cmul* 1 *x* = *x*
 ⟨*proof*⟩

lemma *frag-cmul-minus-one* [*simp*]: *frag-cmul* (−1) *x* = −*x*
 ⟨*proof*⟩

lemma *frag-cmul-cmul* [*simp*]: *frag-cmul* *c* (*frag-cmul* *d* *x*) = *frag-cmul* (*c***d*) *x*
 ⟨*proof*⟩

lemma *lookup-frag-cmul* [*simp*]: *poly-mapping.lookup* (*frag-cmul* *c* *x*) *i* = *c* * *poly-mapping.lookup* *x* *i*
 ⟨*proof*⟩

lemma *minus-frag-cmul* [*simp*]: − *frag-cmul* *k* *x* = *frag-cmul* (−*k*) *x*
 ⟨*proof*⟩

lemma *keys-frag-of*: *Poly-Mapping.keys*(*frag-of* *a*) = {*a*}
 ⟨*proof*⟩

lemma *finite-cmul-nonzero*: *finite* {*x*. *c* * *Poly-Mapping.lookup* *a* *x* ≠ (0::int)}

lemma *keys-cmul*: *Poly-Mapping.keys*(*frag-cmul* *c* *a*) ⊆ *Poly-Mapping.keys* *a*
 ⟨*proof*⟩

lemma *keys-cmul-iff* [*iff*]: *i* ∈ *Poly-Mapping.keys* (*frag-cmul* *c* *x*) ↔ *i* ∈ *Poly-Mapping.keys* *x* ∧ *c* ≠ 0
 ⟨*proof*⟩

lemma *keys-minus* [*simp*]: *Poly-Mapping.keys*(−*a*) = *Poly-Mapping.keys* *a*
 ⟨*proof*⟩

lemma *keys-diff*:
Poly-Mapping.keys(*a* − *b*) ⊆ *Poly-Mapping.keys* *a* ∪ *Poly-Mapping.keys* *b*
 ⟨*proof*⟩

lemma *keys-eq-empty* [*simp*]: *Poly-Mapping.keys* *c* = {} ↔ *c* = 0
 ⟨*proof*⟩

lemma *frag-cmul-eq-0-iff* [*simp*]: *frag-cmul* *k* *c* = 0 ↔ *k*=0 ∨ *c*=0
 ⟨*proof*⟩

lemma *frag-of-eq*: $\text{frag-of } x = \text{frag-of } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-distrib*: $\text{frag-cmul } (c+d) \ a = \text{frag-cmul } c \ a + \text{frag-cmul } d \ a$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-distrib2*: $\text{frag-cmul } c \ (a+b) = \text{frag-cmul } c \ a + \text{frag-cmul } c \ b$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-diff-distrib*: $\text{frag-cmul } (a - b) \ c = \text{frag-cmul } a \ c - \text{frag-cmul } b \ c$
 $\langle \text{proof} \rangle$

lemma *frag-cmul-sum*:
 $\text{frag-cmul } a \ (\text{sum } b \ I) = (\sum_{i \in I}. \text{frag-cmul } a \ (b \ i))$
 $\langle \text{proof} \rangle$

lemma *keys-sum*: $\text{Poly-Mapping.keys}(\text{sum } b \ I) \subseteq (\bigcup_{i \in I}. \text{Poly-Mapping.keys}(b \ i))$
 $\langle \text{proof} \rangle$

definition *frag-extend* :: $('b \Rightarrow 'a \Rightarrow_0 \text{int}) \Rightarrow ('b \Rightarrow_0 \text{int}) \Rightarrow 'a \Rightarrow_0 \text{int}$
where $\text{frag-extend } b \ x \equiv (\sum_{i \in \text{Poly-Mapping.keys } x}. \text{frag-cmul } (\text{Poly-Mapping.lookup } x \ i) \ (b \ i))$

lemma *frag-extend-0* [simp]: $\text{frag-extend } b \ 0 = 0$
 $\langle \text{proof} \rangle$

lemma *frag-extend-of* [simp]: $\text{frag-extend } f \ (\text{frag-of } a) = f \ a$
 $\langle \text{proof} \rangle$

lemma *frag-extend-cmul*:
 $\text{frag-extend } f \ (\text{frag-cmul } c \ x) = \text{frag-cmul } c \ (\text{frag-extend } f \ x)$
 $\langle \text{proof} \rangle$

lemma *frag-extend-minus*:
 $\text{frag-extend } f \ (-x) = -(\text{frag-extend } f \ x)$
 $\langle \text{proof} \rangle$

lemma *frag-extend-add*:
 $\text{frag-extend } f \ (a+b) = (\text{frag-extend } f \ a) + (\text{frag-extend } f \ b)$
 $\langle \text{proof} \rangle$

lemma *frag-extend-diff*:
 $\text{frag-extend } f \ (a-b) = (\text{frag-extend } f \ a) - (\text{frag-extend } f \ b)$
 $\langle \text{proof} \rangle$

lemma *frag-extend-sum*:

finite $I \implies \text{frag-extend } f \ (\sum_{i \in I}. g \ i) = \text{sum } (\text{frag-extend } f \ o \ g) \ I$
 ⟨proof⟩

lemma *frag-extend-eq*:

$(\bigwedge f. f \in \text{Poly-Mapping.keys } c \implies g \ f = h \ f) \implies \text{frag-extend } g \ c = \text{frag-extend } h \ c$
 ⟨proof⟩

lemma *frag-extend-eq-0*:

$(\bigwedge x. x \in \text{Poly-Mapping.keys } c \implies f \ x = 0) \implies \text{frag-extend } f \ c = 0$
 ⟨proof⟩

lemma *keys-frag-extend*: $\text{Poly-Mapping.keys}(\text{frag-extend } f \ c) \subseteq (\bigcup x \in \text{Poly-Mapping.keys } c. \text{Poly-Mapping.keys}(f \ x))$
 ⟨proof⟩

lemma *frag-expansion*: $a = \text{frag-extend } \text{frag-of } a$
 ⟨proof⟩

lemma *frag-closure-minus-cmul*:

assumes $P \ 0$ **and** $P: \bigwedge x \ y. \llbracket P \ x; \ P \ y \rrbracket \implies P(x - y) \ P \ c$
shows $P(\text{frag-cmul } k \ c)$
 ⟨proof⟩

lemma *frag-induction* [*consumes 1, case-names zero one diff*]:

assumes $\text{supp: Poly-Mapping.keys } c \subseteq S$
and $0: P \ 0$ **and** $\text{sing: } \bigwedge x. x \in S \implies P(\text{frag-of } x)$
and $\text{diff: } \bigwedge a \ b. \llbracket P \ a; \ P \ b \rrbracket \implies P(a - b)$
shows $P \ c$
 ⟨proof⟩

lemma *frag-extend-compose*:

$\text{frag-extend } f \ (\text{frag-extend } (\text{frag-of } o \ g) \ c) = \text{frag-extend } (f \ o \ g) \ c$
 ⟨proof⟩

lemma *frag-split*:

fixes $c :: 'a \Rightarrow_0 \text{int}$
assumes $\text{Poly-Mapping.keys } c \subseteq S \cup T$
obtains $d \ e$ **where** $\text{Poly-Mapping.keys } d \subseteq S \ \text{Poly-Mapping.keys } e \subseteq T \ d + e = c$
 ⟨proof⟩

hide-const (**open**) *lookup single update keys range map map-key degree nth the-value items foldr mapp*

end

79 Exponentiation by Squaring

```

theory Power-By-Squaring
  imports Main
begin

context
  fixes  $f :: 'a \Rightarrow 'a \Rightarrow 'a$ 
begin

function efficient-funpow ::  $'a \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a$  where
  efficient-funpow  $y\ x\ 0 = y$ 
| efficient-funpow  $y\ x\ (\text{Suc } 0) = f\ x\ y$ 
|  $n \neq 0 \implies \text{even } n \implies \text{efficient-funpow } y\ x\ n = \text{efficient-funpow } y\ (f\ x\ x)\ (n\ \text{div } 2)$ 
|  $n \neq 1 \implies \text{odd } n \implies \text{efficient-funpow } y\ x\ n = \text{efficient-funpow } (f\ x\ y)\ (f\ x\ x)\ (n\ \text{div } 2)$ 
   $\langle \text{proof} \rangle$ 
termination  $\langle \text{proof} \rangle$ 

lemma efficient-funpow-code [code]:
  efficient-funpow  $y\ x\ n =$ 
    (if  $n = 0$  then  $y$ 
     else if  $n = 1$  then  $f\ x\ y$ 
     else if even  $n$  then efficient-funpow  $y\ (f\ x\ x)\ (n\ \text{div } 2)$ 
     else efficient-funpow  $(f\ x\ y)\ (f\ x\ x)\ (n\ \text{div } 2)$ )
   $\langle \text{proof} \rangle$ 

end

lemma efficient-funpow-correct:
  assumes  $f\text{-assoc}: \bigwedge x\ z. f\ x\ (f\ x\ z) = f\ (f\ x\ x)\ z$ 
  shows efficient-funpow  $f\ y\ x\ n = (f\ x\ \frown n)\ y$ 
   $\langle \text{proof} \rangle$ 

context monoid-mult
begin

lemma power-by-squaring: efficient-funpow  $(*)\ (1 :: 'a) = (\frown)$ 
   $\langle \text{proof} \rangle$ 

end

end

```

80 Preorders with explicit equivalence relation

```

theory Preorder

```

imports *Main*
begin

class *preorder-equiv* = *preorder*
begin

definition *equiv* :: 'a \Rightarrow 'a \Rightarrow bool
where *equiv* *x y* \longleftrightarrow *x* \leq *y* \wedge *y* \leq *x*

notation
equiv ($\langle'(\approx)'\rangle$) **and**
equiv ($\langle(\langle notation = infix \approx \rangle) - / \approx - \rangle$) [51, 51] 50)

lemma *equivD1*: *x* \leq *y* **if** *x* \approx *y*
 $\langle proof \rangle$

lemma *equivD2*: *y* \leq *x* **if** *x* \approx *y*
 $\langle proof \rangle$

lemma *equiv-refl* [*iff*]: *x* \approx *x*
 $\langle proof \rangle$

lemma *equiv-sym*: *x* \approx *y* \longleftrightarrow *y* \approx *x*
 $\langle proof \rangle$

lemma *equiv-trans*: *x* \approx *y* \Longrightarrow *y* \approx *z* \Longrightarrow *x* \approx *z*
 $\langle proof \rangle$

lemma *equiv-antisym*: *x* \leq *y* \Longrightarrow *y* \leq *x* \Longrightarrow *x* \approx *y*
 $\langle proof \rangle$

lemma *less-le*: *x* $<$ *y* \longleftrightarrow *x* \leq *y* \wedge \neg *x* \approx *y*
 $\langle proof \rangle$

lemma *le-less*: *x* \leq *y* \longleftrightarrow *x* $<$ *y* \vee *x* \approx *y*
 $\langle proof \rangle$

lemma *le-imp-less-or-equiv*: *x* \leq *y* \Longrightarrow *x* $<$ *y* \vee *x* \approx *y*
 $\langle proof \rangle$

lemma *less-imp-not-equiv*: *x* $<$ *y* \Longrightarrow \neg *x* \approx *y*
 $\langle proof \rangle$

lemma *not-equiv-le-trans*: \neg *a* \approx *b* \Longrightarrow *a* \leq *b* \Longrightarrow *a* $<$ *b*
 $\langle proof \rangle$

lemma *le-not-equiv-trans*: *a* \leq *b* \Longrightarrow \neg *a* \approx *b* \Longrightarrow *a* $<$ *b*
 $\langle proof \rangle$

lemma *antisym-conv*: $y \leq x \implies x \leq y \longleftrightarrow x \approx y$
 $\langle proof \rangle$

end

$\langle ML \rangle$

end

81 Additive group operations on product types

theory *Product-Plus*

imports *Main*

begin

81.1 Operations

instantiation *prod* :: (*zero*, *zero*) *zero*
begin

definition *zero-prod-def*: $0 = (0, 0)$

instance $\langle proof \rangle$
end

instantiation *prod* :: (*plus*, *plus*) *plus*
begin

definition *plus-prod-def*:
 $x + y = (fst\ x + fst\ y, snd\ x + snd\ y)$

instance $\langle proof \rangle$
end

instantiation *prod* :: (*minus*, *minus*) *minus*
begin

definition *minus-prod-def*:
 $x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$

instance $\langle proof \rangle$
end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*
begin

definition *uminus-prod-def*:
 $- x = (-\ fst\ x, -\ snd\ x)$

instance $\langle proof \rangle$
end

lemma *fst-zero* [*simp*]: $fst\ 0 = 0$
 $\langle proof \rangle$

lemma *snd-zero* [*simp*]: $snd\ 0 = 0$
 $\langle proof \rangle$

lemma *fst-add* [*simp*]: $fst\ (x + y) = fst\ x + fst\ y$
 $\langle proof \rangle$

lemma *snd-add* [*simp*]: $snd\ (x + y) = snd\ x + snd\ y$
 $\langle proof \rangle$

lemma *fst-diff* [*simp*]: $fst\ (x - y) = fst\ x - fst\ y$
 $\langle proof \rangle$

lemma *snd-diff* [*simp*]: $snd\ (x - y) = snd\ x - snd\ y$
 $\langle proof \rangle$

lemma *fst-uminus* [*simp*]: $fst\ (-\ x) = -\ fst\ x$
 $\langle proof \rangle$

lemma *snd-uminus* [*simp*]: $snd\ (-\ x) = -\ snd\ x$
 $\langle proof \rangle$

lemma *add-Pair* [*simp*]: $(a, b) + (c, d) = (a + c, b + d)$
 $\langle proof \rangle$

lemma *diff-Pair* [*simp*]: $(a, b) - (c, d) = (a - c, b - d)$
 $\langle proof \rangle$

lemma *uminus-Pair* [*simp*, *code*]: $-(a, b) = (-\ a, -\ b)$
 $\langle proof \rangle$

81.2 Class instances

instance *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
 $\langle proof \rangle$

instance *prod* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
 $\langle proof \rangle$

instance *prod* :: (*monoid-add*, *monoid-add*) *monoid-add*
 $\langle proof \rangle$

instance *prod* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
 $\langle proof \rangle$


```

instance prod :: (cancel-semigroup-add, cancel-semigroup-add) cancel-semigroup-add
  ⟨proof⟩

instance prod :: (cancel-ab-semigroup-add, cancel-ab-semigroup-add) cancel-ab-semigroup-add
  ⟨proof⟩

instance prod :: (cancel-comm-monoid-add, cancel-comm-monoid-add) cancel-comm-monoid-add
  ⟨proof⟩

instance prod :: (group-add, group-add) group-add
  ⟨proof⟩

instance prod :: (ab-group-add, ab-group-add) ab-group-add
  ⟨proof⟩

lemma fst-sum: fst (∑ x∈A. f x) = (∑ x∈A. fst (f x))
  ⟨proof⟩

lemma snd-sum: snd (∑ x∈A. f x) = (∑ x∈A. snd (f x))
  ⟨proof⟩

lemma sum-prod: (∑ x∈A. (f x, g x)) = (∑ x∈A. f x, ∑ x∈A. g x)
  ⟨proof⟩

end

```

82 Roots of real quadratics

```

theory Quadratic-Discriminant
imports Complex-Main
begin

```

```

definition discrim :: real ⇒ real ⇒ real ⇒ real
  where discrim a b c ≡ b2 - 4 * a * c

```

```

lemma complete-square:
  a ≠ 0 ⇒ a * x2 + b * x + c = 0 ⟷ (2 * a * x + b)2 = discrim a b c
  ⟨proof⟩

```

```

lemma discriminant-negative:
  fixes a b c x :: real
  assumes a ≠ 0
  and discrim a b c < 0
  shows a * x2 + b * x + c ≠ 0
  ⟨proof⟩

```

```

lemma plus-or-minus-sqrt:
  fixes x y :: real

```

assumes $y \geq 0$
 shows $x^2 = y \longleftrightarrow x = \text{sqrt } y \vee x = - \text{sqrt } y$
 $\langle \text{proof} \rangle$

lemma *divide-non-zero*:
 fixes $x \ y \ z :: \text{real}$
 assumes $x \neq 0$
 shows $x * y = z \longleftrightarrow y = z / x$
 $\langle \text{proof} \rangle$

lemma *discriminant-nonneg*:
 fixes $a \ b \ c \ x :: \text{real}$
 assumes $a \neq 0$
 and $\text{discrim } a \ b \ c \geq 0$
 shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
 $x = (-b + \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a) \vee$
 $x = (-b - \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a)$
 $\langle \text{proof} \rangle$

lemma *discriminant-zero*:
 fixes $a \ b \ c \ x :: \text{real}$
 assumes $a \neq 0$
 and $\text{discrim } a \ b \ c = 0$
 shows $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b / (2 * a)$
 $\langle \text{proof} \rangle$

theorem *discriminant-iff*:
 fixes $a \ b \ c \ x :: \text{real}$
 assumes $a \neq 0$
 shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
 $\text{discrim } a \ b \ c \geq 0 \wedge$
 $(x = (-b + \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a) \vee$
 $x = (-b - \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a))$
 $\langle \text{proof} \rangle$

lemma *discriminant-nonneg-ex*:
 fixes $a \ b \ c :: \text{real}$
 assumes $a \neq 0$
 and $\text{discrim } a \ b \ c \geq 0$
 shows $\exists x. a * x^2 + b * x + c = 0$
 $\langle \text{proof} \rangle$

lemma *discriminant-pos-ex*:
 fixes $a \ b \ c :: \text{real}$
 assumes $a \neq 0$
 and $\text{discrim } a \ b \ c > 0$
 shows $\exists x \ y. x \neq y \wedge a * x^2 + b * x + c = 0 \wedge a * y^2 + b * y + c = 0$
 $\langle \text{proof} \rangle$

```

lemma discriminant-pos-distinct:
  fixes  $a\ b\ c\ x :: \text{real}$ 
  assumes  $a \neq 0$ 
  and  $\text{discrim } a\ b\ c > 0$ 
  shows  $\exists y. x \neq y \wedge a * y^2 + b * y + c = 0$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma Rats-solution-QE:
  assumes  $a \in \mathbb{Q}\ b \in \mathbb{Q}\ a \neq 0$ 
  and  $a * x^2 + b * x + c = 0$ 
  and  $\text{sqr}t\ (\text{discrim } a\ b\ c) \in \mathbb{Q}$ 
  shows  $x \in \mathbb{Q}$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma Rats-solution-QE-converse:
  assumes  $a \in \mathbb{Q}\ b \in \mathbb{Q}$ 
  and  $a * x^2 + b * x + c = 0$ 
  and  $x \in \mathbb{Q}$ 
  shows  $\text{sqr}t\ (\text{discrim } a\ b\ c) \in \mathbb{Q}$ 
   $\langle \text{proof} \rangle$ 

```

```

end

```

83 Pretty syntax for Quotient operations

```

theory Quotient-Syntax
imports Main
begin

notation
  rel-conj (infixr  $\langle \text{OOO} \rangle$  75) and
  map-fun (infixr  $\langle \text{---} \rangle$  55) and
  rel-fun (infixr  $\langle \text{===} \rangle$  55)

end

```

84 Quotient infrastructure for the set type

```

theory Quotient-Set
imports Quotient-Syntax
begin

```

84.1 Contravariant set map (vimage) and set relator, rules for the Quotient package

```

definition rel-vset  $R\ xs\ ys \equiv \forall x\ y. R\ x\ y \longrightarrow x \in xs \longleftrightarrow y \in ys$ 

```

```

lemma rel-vset-eq [id-simps]:

```

rel-vset (=) = (=)
 ⟨proof⟩

lemma *rel-vset-equivp*:
 assumes *e*: *equivp R*
 shows *rel-vset R xs ys* \longleftrightarrow *xs = ys* \wedge ($\forall x y. x \in xs \longrightarrow R x y \longrightarrow y \in xs$)
 ⟨proof⟩

lemma *set-quotient* [*quot-thm*]:
 assumes *Quotient3 R Abs Rep*
 shows *Quotient3 (rel-vset R) (vimage Rep) (vimage Abs)*
 ⟨proof⟩

declare [[*mapQ3 set = (rel-vset, set-quotient)*]]

lemma *empty-set-rsp*[*quot-respect*]:
rel-vset R {} {}
 ⟨proof⟩

lemma *collect-rsp*[*quot-respect*]:
 assumes *Quotient3 R Abs Rep*
 shows $((R ==> (=)) ==> \textit{rel-vset R Collect Collect})$
 ⟨proof⟩

lemma *collect-prs*[*quot-preserve*]:
 assumes *Quotient3 R Abs Rep*
 shows $((\textit{Abs} \dashrightarrow \textit{id}) \dashrightarrow (-') \textit{Rep}) \textit{Collect} = \textit{Collect}$
 ⟨proof⟩

lemma *union-rsp*[*quot-respect*]:
 assumes *Quotient3 R Abs Rep*
 shows $(\textit{rel-vset R} ==> \textit{rel-vset R} ==> \textit{rel-vset R}) (\cup) (\cup)$
 ⟨proof⟩

lemma *union-prs*[*quot-preserve*]:
 assumes *Quotient3 R Abs Rep*
 shows $((-') \textit{Abs} \dashrightarrow (-') \textit{Abs} \dashrightarrow (-') \textit{Rep}) (\cup) = (\cup)$
 ⟨proof⟩

lemma *diff-rsp*[*quot-respect*]:
 assumes *Quotient3 R Abs Rep*
 shows $(\textit{rel-vset R} ==> \textit{rel-vset R} ==> \textit{rel-vset R}) (-) (-)$
 ⟨proof⟩

lemma *diff-prs*[*quot-preserve*]:
 assumes *Quotient3 R Abs Rep*
 shows $((-') \textit{Abs} \dashrightarrow (-') \textit{Abs} \dashrightarrow (-') \textit{Rep}) (-) = (-)$
 ⟨proof⟩

lemma *inter-rsp*[*quot-respect*]:
assumes *Quotient3* *R Abs Rep*
shows $(rel\text{-}vset\ R ==> rel\text{-}vset\ R ==> rel\text{-}vset\ R)\ (\cap)\ (\cap)$
 $\langle proof \rangle$

lemma *inter-prs*[*quot-preserve*]:
assumes *Quotient3* *R Abs Rep*
shows $((-\cdot) Abs \text{----}> (-\cdot) Abs \text{----}> (-\cdot) Rep)\ (\cap) = (\cap)$
 $\langle proof \rangle$

lemma *mem-prs*[*quot-preserve*]:
assumes *Quotient3* *R Abs Rep*
shows $(Rep \text{----}> (-\cdot) Abs \text{----}> id)\ (\in) = (\in)$
 $\langle proof \rangle$

lemma *mem-rsp*[*quot-respect*]:
shows $(R ==> rel\text{-}vset\ R ==> (=))\ (\in)\ (\in)$
 $\langle proof \rangle$

end

85 Quotient infrastructure for the product type

theory *Quotient-Product*
imports *Quotient-Syntax*
begin

85.1 Rules for the Quotient package

lemma *map-prod-id* [*id-simps*]:
shows *map-prod id id = id*
 $\langle proof \rangle$

lemma *rel-prod-eq* [*id-simps*]:
shows *rel-prod (=) (=) = (=)*
 $\langle proof \rangle$

lemma *prod-equivp* [*quot-equiv*]:
assumes *equivp* *R1*
assumes *equivp* *R2*
shows *equivp* $(rel\text{-}prod\ R1\ R2)$
 $\langle proof \rangle$

lemma *prod-quotient* [*quot-thm*]:
assumes *Quotient3* *R1 Abs1 Rep1*
assumes *Quotient3* *R2 Abs2 Rep2*
shows *Quotient3* $(rel\text{-}prod\ R1\ R2)\ (map\text{-}prod\ Abs1\ Abs2)\ (map\text{-}prod\ Rep1\ Rep2)$
 $\langle proof \rangle$

declare $[[\text{mapQ3 prod} = (\text{rel-prod}, \text{prod-quotient})]]$

lemma *Pair-rsp* [quot-respect]:

assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$

assumes $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $(R1 ==> R2 ==> \text{rel-prod } R1 \text{ } R2) \text{ Pair Pair}$

$\langle \text{proof} \rangle$

lemma *Pair-prs* [quot-preserve]:

assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$

assumes $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $(\text{Rep1} \text{ ---} > \text{Rep2} \text{ ---} > (\text{map-prod Abs1 Abs2})) \text{ Pair} = \text{Pair}$

$\langle \text{proof} \rangle$

lemma *fst-rsp* [quot-respect]:

assumes $\text{Quotient3 } R1 \text{ Abs1 Rep1}$

assumes $\text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $(\text{rel-prod } R1 \text{ } R2 ==> R1) \text{ fst fst}$

$\langle \text{proof} \rangle$

lemma *fst-prs* [quot-preserve]:

assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$

assumes $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $(\text{map-prod Rep1 Rep2} \text{ ---} > \text{Abs1}) \text{ fst} = \text{fst}$

$\langle \text{proof} \rangle$

lemma *snd-rsp* [quot-respect]:

assumes $\text{Quotient3 } R1 \text{ Abs1 Rep1}$

assumes $\text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $(\text{rel-prod } R1 \text{ } R2 ==> R2) \text{ snd snd}$

$\langle \text{proof} \rangle$

lemma *snd-prs* [quot-preserve]:

assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$

assumes $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $(\text{map-prod Rep1 Rep2} \text{ ---} > \text{Abs2}) \text{ snd} = \text{snd}$

$\langle \text{proof} \rangle$

lemma *case-prod-rsp* [quot-respect]:

shows $((R1 ==> R2 ==> (=)) ==> (\text{rel-prod } R1 \text{ } R2) ==> (=))$

case-prod case-prod

$\langle \text{proof} \rangle$

lemma *split-prs* [quot-preserve]:

assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$

and $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $((\text{Abs1} \text{ ---} > \text{Abs2} \text{ ---} > \text{id}) \text{ ---} > \text{map-prod Rep1 Rep2} \text{ ---} > \text{id})$

case-prod = *case-prod*

$\langle \text{proof} \rangle$

lemma *[quot-respect]*:

shows $((R2 \text{ ==== } R2 \text{ ==== } (=)) \text{ ==== } (R1 \text{ ==== } R1 \text{ ==== } (=)) \text{ ==== } \text{rel-prod } R2 \text{ } R1 \text{ ==== } \text{rel-prod } R2 \text{ } R1 \text{ ==== } (=)) \text{ rel-prod rel-prod}$
 $\langle \text{proof} \rangle$

lemma *[quot-preserve]*:

assumes $q1: \text{Quotient3 } R1 \text{ abs1 rep1}$
and $q2: \text{Quotient3 } R2 \text{ abs2 rep2}$
shows $((\text{abs1} \text{ ---- } \text{abs1} \text{ ---- } id) \text{ ---- } (\text{abs2} \text{ ---- } \text{abs2} \text{ ---- } id) \text{ ---- } \text{map-prod rep1 rep2 ---- } \text{map-prod rep1 rep2 ---- } id) \text{ rel-prod} = \text{rel-prod}$
 $\langle \text{proof} \rangle$

lemma *[quot-preserve]*:

shows $(\text{rel-prod } ((\text{rep1} \text{ ---- } \text{rep1} \text{ ---- } id) \text{ } R1) ((\text{rep2} \text{ ---- } \text{rep2} \text{ ---- } id) \text{ } R2) (l1, l2) (r1, r2)) = (R1 (\text{rep1 } l1) (\text{rep1 } r1) \wedge R2 (\text{rep2 } l2) (\text{rep2 } r2))$
 $\langle \text{proof} \rangle$

declare *prod.inject**[quot-preserve]*

end

86 Quotient infrastructure for the option type

theory *Quotient-Option*
imports *Quotient-Syntax*
begin

86.1 Rules for the Quotient package

lemma *rel-option-map1*:

$\text{rel-option } R (\text{map-option } f \text{ } x) \text{ } y \longleftrightarrow \text{rel-option } (\lambda x. R (f \text{ } x)) \text{ } x \text{ } y$
 $\langle \text{proof} \rangle$

lemma *rel-option-map2*:

$\text{rel-option } R \text{ } x (\text{map-option } f \text{ } y) \longleftrightarrow \text{rel-option } (\lambda x \text{ } y. R \text{ } x (f \text{ } y)) \text{ } x \text{ } y$
 $\langle \text{proof} \rangle$

declare

map-option.id *[id-simps]*
option.rel-eq *[id-simps]*

lemma *reflp-rel-option*:

$\text{reflp } R \implies \text{reflp } (\text{rel-option } R)$
 $\langle \text{proof} \rangle$

lemma *option-symp*:

```

  symp R  $\implies$  symp (rel-option R)
  <proof>

lemma option-transp:
  transp R  $\implies$  transp (rel-option R)
  <proof>

lemma option-equivp [quot-equiv]:
  equivp R  $\implies$  equivp (rel-option R)
  <proof>

lemma option-quotient [quot-thm]:
  assumes Quotient3 R Abs Rep
  shows Quotient3 (rel-option R) (map-option Abs) (map-option Rep)
  <proof>

declare [[mapQ3 option = (rel-option, option-quotient)]]

lemma option-None-rsp [quot-respect]:
  assumes q: Quotient3 R Abs Rep
  shows rel-option R None None
  <proof>

lemma option-Some-rsp [quot-respect]:
  assumes q: Quotient3 R Abs Rep
  shows (R  $\implies$  rel-option R) Some Some
  <proof>

lemma option-None-prs [quot-preserve]:
  assumes q: Quotient3 R Abs Rep
  shows map-option Abs None = None
  <proof>

lemma option-Some-prs [quot-preserve]:
  assumes q: Quotient3 R Abs Rep
  shows (Rep  $\dashv\dashv$  map-option Abs) Some = Some
  <proof>

end

```

87 Quotient infrastructure for the list type

```

theory Quotient-List
imports Quotient-Set Quotient-Product Quotient-Option
begin

```

87.1 Rules for the Quotient package

```

lemma map-id [id-simps]:

```


map id = id
 $\langle \text{proof} \rangle$

lemma *list-all2-eq [id-simps]*:
list-all2 (=) = (=)
 $\langle \text{proof} \rangle$

lemma *reflp-list-all2*:
assumes *reflp R*
shows *reflp (list-all2 R)*
 $\langle \text{proof} \rangle$

lemma *list-symp*:
assumes *symp R*
shows *symp (list-all2 R)*
 $\langle \text{proof} \rangle$

lemma *list-transp*:
assumes *transp R*
shows *transp (list-all2 R)*
 $\langle \text{proof} \rangle$

lemma *list-equivp [quot-equiv]*:
equivp R \implies equivp (list-all2 R)
 $\langle \text{proof} \rangle$

lemma *list-quotient3 [quot-thm]*:
assumes *Quotient3 R Abs Rep*
shows *Quotient3 (list-all2 R) (map Abs) (map Rep)*
 $\langle \text{proof} \rangle$

declare $[[\text{mapQ3 list} = (\text{list-all2}, \text{list-quotient3})]]$

lemma *cons-prs [quot-preserve]*:
assumes *q: Quotient3 R Abs Rep*
shows $(\text{Rep} \dashrightarrow (\text{map Rep}) \dashrightarrow (\text{map Abs})) (\#) = (\#)$
 $\langle \text{proof} \rangle$

lemma *cons-rsp [quot-respect]*:
assumes *q: Quotient3 R Abs Rep*
shows $(R \implies \text{list-all2 } R \implies \text{list-all2 } R) (\#) (\#)$
 $\langle \text{proof} \rangle$

lemma *nil-prs [quot-preserve]*:
assumes *q: Quotient3 R Abs Rep*
shows $\text{map Abs } [] = []$
 $\langle \text{proof} \rangle$

lemma *nil-rsp [quot-respect]*:

assumes q : *Quotient3* R Abs Rep
shows $list-all2\ R\ []\ []$
 $\langle proof \rangle$

lemma *map-prs-aux*:

assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $(map\ abs2)\ (map\ ((abs1\ ---->\ rep2)\ f)\ (map\ rep1\ l)) = map\ f\ l$
 $\langle proof \rangle$

lemma *map-prs [quot-preserve]*:

assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $((abs1\ ---->\ rep2)\ ---->\ (map\ rep1)\ ---->\ (map\ abs2))\ map = map$
and $((abs1\ ---->\ id)\ ---->\ map\ rep1\ ---->\ id)\ map = map$
 $\langle proof \rangle$

lemma *map-rsp [quot-respect]*:

assumes $q1$: *Quotient3* $R1$ $Abs1$ $Rep1$
and $q2$: *Quotient3* $R2$ $Abs2$ $Rep2$
shows $((R1\ ===>\ R2)\ ===>\ (list-all2\ R1)\ ===>\ list-all2\ R2)\ map\ map$
and $((R1\ ===>\ (=))\ ===>\ (list-all2\ R1)\ ===>\ (=))\ map\ map$
 $\langle proof \rangle$

lemma *foldr-prs-aux*:

assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $abs2\ (foldr\ ((abs1\ ---->\ abs2\ ---->\ rep2)\ f)\ (map\ rep1\ l)\ (rep2\ e))$
 $= foldr\ f\ l\ e$
 $\langle proof \rangle$

lemma *foldr-prs [quot-preserve]*:

assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $((abs1\ ---->\ abs2\ ---->\ rep2)\ ---->\ (map\ rep1)\ ---->\ rep2\ ---->\ abs2)\ foldr = foldr$
 $\langle proof \rangle$

lemma *foldl-prs-aux*:

assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $abs1\ (foldl\ ((abs1\ ---->\ abs2\ ---->\ rep1)\ f)\ (rep1\ e)\ (map\ rep2\ l)) = foldl\ f\ e\ l$
 $\langle proof \rangle$

lemma *foldl-prs [quot-preserve]*:

assumes a : *Quotient3* $R1$ $abs1$ $rep1$
and b : *Quotient3* $R2$ $abs2$ $rep2$
shows $((abs1\ ---->\ abs2\ ---->\ rep1)\ ---->\ rep1\ ---->\ (map\ rep2)\ ---->$

abs1) *foldl* = *foldl*
 ⟨*proof*⟩

lemma *foldl-rsp*[*quot-respect*]:
 assumes *q1*: *Quotient3* *R1* *Abs1* *Rep1*
 and *q2*: *Quotient3* *R2* *Abs2* *Rep2*
 shows ((*R1* ==> *R2* ==> *R1*) ==> *R1* ==> *list-all2* *R2* ==> *R1*)
foldl foldl
 ⟨*proof*⟩

lemma *foldr-rsp*[*quot-respect*]:
 assumes *q1*: *Quotient3* *R1* *Abs1* *Rep1*
 and *q2*: *Quotient3* *R2* *Abs2* *Rep2*
 shows ((*R1* ==> *R2* ==> *R2*) ==> *list-all2* *R1* ==> *R2* ==> *R2*)
foldr foldr
 ⟨*proof*⟩

lemma *list-all2-rsp*:
 assumes *r*: $\forall x y. R\ x\ y \longrightarrow (\forall a\ b. R\ a\ b \longrightarrow S\ x\ a = T\ y\ b)$
 and *l1*: *list-all2* *R* *x* *y*
 and *l2*: *list-all2* *R* *a* *b*
 shows *list-all2* *S* *x* *a* = *list-all2* *T* *y* *b*
 ⟨*proof*⟩

lemma [*quot-respect*]:
 ((*R* ==> *R* ==> (=)) ==> *list-all2* *R* ==> *list-all2* *R* ==> (=))
list-all2 list-all2
 ⟨*proof*⟩

lemma [*quot-preserve*]:
 assumes *a*: *Quotient3* *R* *abs1* *rep1*
 shows ((*abs1* ----> *abs1* ----> *id*) ----> *map* *rep1* ----> *map* *rep1* ---->
id) *list-all2* = *list-all2*
 ⟨*proof*⟩

lemma [*quot-preserve*]:
 assumes *a*: *Quotient3* *R* *abs1* *rep1*
 shows (*list-all2* ((*rep1* ----> *rep1* ----> *id*) *R*) *l* *m*) = (*l* = *m*)
 ⟨*proof*⟩

lemma *list-all2-find-element*:
 assumes *a*: *x* ∈ *set a*
 and *b*: *list-all2* *R* *a* *b*
 shows $\exists y. (y \in \text{set } b \wedge R\ x\ y)$
 ⟨*proof*⟩

lemma *list-all2-refl*:
 assumes *a*: $\bigwedge x\ y. R\ x\ y = (R\ x = R\ y)$
 shows *list-all2* *R* *x* *x*

$\langle \text{proof} \rangle$

end

88 Quotient infrastructure for the sum type

```
theory Quotient-Sum
imports Quotient-Syntax
begin
```

88.1 Rules for the Quotient package

lemma *rel-sum-map1*:

$$\text{rel-sum } R1 \ R2 \ (\text{map-sum } f1 \ f2 \ x) \ y \longleftrightarrow \text{rel-sum } (\lambda x. R1 \ (f1 \ x)) \ (\lambda x. R2 \ (f2 \ x))$$

$x \ y$
 $\langle \text{proof} \rangle$

lemma *rel-sum-map2*:

$$\text{rel-sum } R1 \ R2 \ x \ (\text{map-sum } f1 \ f2 \ y) \longleftrightarrow \text{rel-sum } (\lambda x \ y. R1 \ x \ (f1 \ y)) \ (\lambda x \ y. R2 \ x \ (f2 \ y))$$

$x \ y$
 $\langle \text{proof} \rangle$

lemma *map-sum-id* [*id-simps*]:

$$\text{map-sum } id \ id = id$$

$\langle \text{proof} \rangle$

lemma *rel-sum-eq* [*id-simps*]:

$$\text{rel-sum } (=) \ (=) = (=)$$

$\langle \text{proof} \rangle$

lemma *reflp-rel-sum*:

$$\text{reflp } R1 \implies \text{reflp } R2 \implies \text{reflp } (\text{rel-sum } R1 \ R2)$$

$\langle \text{proof} \rangle$

lemma *sum-symp*:

$$\text{symp } R1 \implies \text{symp } R2 \implies \text{symp } (\text{rel-sum } R1 \ R2)$$

$\langle \text{proof} \rangle$

lemma *sum-transp*:

$$\text{transp } R1 \implies \text{transp } R2 \implies \text{transp } (\text{rel-sum } R1 \ R2)$$

$\langle \text{proof} \rangle$

lemma *sum-equivp* [*quot-equiv*]:

$$\text{equivp } R1 \implies \text{equivp } R2 \implies \text{equivp } (\text{rel-sum } R1 \ R2)$$

$\langle \text{proof} \rangle$

lemma *sum-quotient* [*quot-thm*]:

```
assumes q1: Quotient3 R1 Abs1 Rep1
assumes q2: Quotient3 R2 Abs2 Rep2
```

```

shows Quotient3 (rel-sum R1 R2) (map-sum Abs1 Abs2) (map-sum Rep1 Rep2)
  ⟨proof⟩

declare [[mapQ3 sum = (rel-sum, sum-quotient)]]

lemma sum-Inl-rsp [quot-respect]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (R1 ==> rel-sum R1 R2) Inl Inl
  ⟨proof⟩

lemma sum-Inr-rsp [quot-respect]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (R2 ==> rel-sum R1 R2) Inr Inr
  ⟨proof⟩

lemma sum-Inl-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (Rep1 ---> map-sum Abs1 Abs2) Inl = Inl
  ⟨proof⟩

lemma sum-Inr-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (Rep2 ---> map-sum Abs1 Abs2) Inr = Inr
  ⟨proof⟩

end

```

89 Quotient types

```

theory Quotient-Type
imports Main
begin

```

We introduce the notion of quotient types over equivalence relations via type classes.

89.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a \Rightarrow 'a \Rightarrow \text{bool}$.

```

class eqv =
  fixes eqv :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infixl  $\langle \sim \rangle$  50)

```

```

class equiv = eqv +
  assumes equiv-refl [intro]:  $x \sim x$ 

```

and *equiv-trans* [*trans*]: $x \sim y \implies y \sim z \implies x \sim z$
and *equiv-sym* [*sym*]: $x \sim y \implies y \sim x$
begin

lemma *equiv-not-sym* [*sym*]: $\neg x \sim y \implies \neg y \sim x$
 $\langle \text{proof} \rangle$

lemma *not-equiv-trans1* [*trans*]: $\neg x \sim y \implies y \sim z \implies \neg x \sim z$
 $\langle \text{proof} \rangle$

lemma *not-equiv-trans2* [*trans*]: $x \sim y \implies \neg y \sim z \implies \neg x \sim z$
 $\langle \text{proof} \rangle$

end

The quotient type *'a quot* consists of all *equivalence classes* over elements of the base type *'a*.

definition (**in** *eqv*) *quot* = $\{\{x. a \sim x\} \mid a. \text{True}\}$

typedef (**overloaded**) *'a quot* = *quot* :: *'a::eqv set set*
 $\langle \text{proof} \rangle$

lemma *quotI* [*intro*]: $\{x. a \sim x\} \in \text{quot}$
 $\langle \text{proof} \rangle$

lemma *quotE* [*elim*]:
assumes $R \in \text{quot}$
obtains *a* **where** $R = \{x. a \sim x\}$
 $\langle \text{proof} \rangle$

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition *class* :: *'a::equiv* \Rightarrow *'a quot* ($\langle \langle \text{open-block notation} = \langle \text{mixfix class} \rangle \rangle [-] \rangle$)
where $[a] = \text{Abs-quot } \{x. a \sim x\}$

theorem *quot-exhaust*: $\exists a. A = [a]$
 $\langle \text{proof} \rangle$

lemma *quot-cases* [*cases type: quot*]:
obtains *a* **where** $A = [a]$
 $\langle \text{proof} \rangle$

89.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $[a] = [b] \longleftrightarrow a \sim b$
 $\langle \text{proof} \rangle$

89.3 Picking representing elements

definition $\text{pick} :: 'a::\text{equiv quot} \Rightarrow 'a$
where $\text{pick } A = (\text{SOME } a. A = \lfloor a \rfloor)$

theorem $\text{pick-equiv [intro]: pick } \lfloor a \rfloor \sim a$
 $\langle \text{proof} \rangle$

theorem $\text{pick-inverse [intro]: } \lfloor \text{pick } A \rfloor = A$
 $\langle \text{proof} \rangle$

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

theorem $\text{quot-cond-function:}$
assumes $\text{eq: } \bigwedge X Y. P X Y \Longrightarrow f X Y \equiv g (\text{pick } X) (\text{pick } Y)$
and $\text{cong: } \bigwedge x x' y y'. \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$
 $\Longrightarrow P \lfloor x \rfloor \lfloor y \rfloor \Longrightarrow P \lfloor x' \rfloor \lfloor y' \rfloor \Longrightarrow g x y = g x' y'$
and $P: P \lfloor a \rfloor \lfloor b \rfloor$
shows $f \lfloor a \rfloor \lfloor b \rfloor = g a b$
 $\langle \text{proof} \rangle$

theorem quot-function:
assumes $\bigwedge X Y. f X Y \equiv g (\text{pick } X) (\text{pick } Y)$
and $\bigwedge x x' y y'. \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor \Longrightarrow g x y = g x' y'$
shows $f \lfloor a \rfloor \lfloor b \rfloor = g a b$
 $\langle \text{proof} \rangle$

theorem quot-function':
 $(\bigwedge X Y. f X Y \equiv g (\text{pick } X) (\text{pick } Y)) \Longrightarrow$
 $(\bigwedge x x' y y'. x \sim x' \Longrightarrow y \sim y' \Longrightarrow g x y = g x' y') \Longrightarrow$
 $f \lfloor a \rfloor \lfloor b \rfloor = g a b$
 $\langle \text{proof} \rangle$

end

90 Ramsey’s Theorem

theory *Ramsey*
imports *Infinite-Set Equipollence FuncSet*
begin

90.1 Preliminary definitions

abbreviation $\text{strict-sorted} :: 'a::\text{linorder list} \Rightarrow \text{bool}$ **where**
 $\text{strict-sorted} \equiv \text{sorted-wrt } (<)$

90.1.1 The n -element subsets of a set A

definition $nsets :: [a \ set, \ nat] \Rightarrow 'a \ set \ set \ (\langle \langle notation = \langle mixfix \ nsets \rangle \rangle [-] \rangle [0,999] \ 999)$

where $nsets \ A \ n \equiv \{N. \ N \subseteq A \wedge finite \ N \wedge card \ N = n\}$

lemma $finite\text{-}imp\text{-}finite\text{-}nsets$: $finite \ A \Longrightarrow finite \ ([A]^k)$
 $\langle proof \rangle$

lemma $nsets\text{-}mono$: $A \subseteq B \Longrightarrow nsets \ A \ n \subseteq nsets \ B \ n$
 $\langle proof \rangle$

lemma $nsets\text{-}Pi\text{-}contra$: $A' \subseteq A \Longrightarrow Pi \ ([A]^n) \ B \subseteq Pi \ ([A']^n) \ B$
 $\langle proof \rangle$

lemma $nsets\text{-}2\text{-}eq$: $[A]^2 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \{\{x, y\}\})$
 $\langle proof \rangle$

lemma $nsets\text{-}2\text{-}E$:
assumes $e \in [A]^2$
obtains $x \ y$ **where** $e = \{x, y\} \ x \in A \ y \in A \ x \neq y$
 $\langle proof \rangle$

lemma $nsets\text{-}doubleton\text{-}2\text{-}eq$ $[simp]$: $[\{x, y\}]^2 = (if \ x=y \ then \ \{\} \ else \ \{\{x, y\}\})$
 $\langle proof \rangle$

lemma $doubleton\text{-}in\text{-}nsets\text{-}2$ $[simp]$: $\{x, y\} \in [A]^2 \longleftrightarrow x \in A \wedge y \in A \wedge x \neq y$
 $\langle proof \rangle$

lemma $nsets\text{-}3\text{-}eq$: $[A]^3 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \bigcup z \in A - \{x, y\}. \{\{x, y, z\}\})$
 $\langle proof \rangle$

lemma $nsets\text{-}4\text{-}eq$: $[A]^4 = (\bigcup u \in A. \bigcup x \in A - \{u\}. \bigcup y \in A - \{u, x\}. \bigcup z \in A - \{u, x, y\}. \{\{u, x, y, z\}\})$
(is - = ?rhs)
 $\langle proof \rangle$

lemma $nsets\text{-}disjoint\text{-}2$:
 $X \cap Y = \{\} \Longrightarrow [X \cup Y]^2 = [X]^2 \cup [Y]^2 \cup (\bigcup x \in X. \bigcup y \in Y. \{\{x, y\}\})$
 $\langle proof \rangle$

lemma $ordered\text{-}nsets\text{-}2\text{-}eq$:
fixes $A :: 'a::linorder \ set$
shows $[A]^2 = \{\{x, y\} \mid x \ y. \ x \in A \wedge y \in A \wedge x < y\}$
(is - = ?rhs)
 $\langle proof \rangle$

lemma $ordered\text{-}nsets\text{-}3\text{-}eq$:
fixes $A :: 'a::linorder \ set$
shows $[A]^3 = \{\{x, y, z\} \mid x \ y \ z. \ x \in A \wedge y \in A \wedge z \in A \wedge x < y \wedge y < z\}$

(is - = ?rhs)
 ⟨proof⟩

lemma *ordered-nsets-4-eq*:
 fixes $A :: 'a::linorder\ set$
 defines $rhs \equiv \lambda U. \exists u\ x\ y\ z. U = \{u, x, y, z\} \wedge u \in A \wedge x \in A \wedge y \in A \wedge z \in A$
 $\wedge u < x \wedge x < y \wedge y < z$
 shows $[A]^4 = Collect\ rhs$
 ⟨proof⟩

lemma *ordered-nsets-5-eq*:
 fixes $A :: 'a::linorder\ set$
 defines $rhs \equiv \lambda U. \exists u\ v\ x\ y\ z. U = \{u, v, x, y, z\} \wedge u \in A \wedge v \in A \wedge x \in A \wedge y$
 $\in A \wedge z \in A \wedge u < v \wedge v < x \wedge x < y \wedge y < z$
 shows $[A]^5 = Collect\ rhs$
 ⟨proof⟩

lemma *binomial-eq-nsets*: $n\ choose\ k = card\ (nsets\ \{0..<n\}\ k)$
 ⟨proof⟩

lemma *nsets-eq-empty-iff*: $nsets\ A\ r = \{\} \longleftrightarrow finite\ A \wedge card\ A < r$
 ⟨proof⟩

lemma *nsets-eq-empty*: $\llbracket finite\ A; card\ A < r \rrbracket \Longrightarrow nsets\ A\ r = \{\}$
 ⟨proof⟩

lemma *nsets-empty-iff*: $nsets\ \{\}\ r = (if\ r=0\ then\ \{\{\}\}\ else\ \{\})$
 ⟨proof⟩

lemma *nsets-singleton-iff*: $nsets\ \{a\}\ r = (if\ r=0\ then\ \{\{\}\}\ else\ if\ r=1\ then\ \{\{a\}\}$
 $else\ \{\})$
 ⟨proof⟩

lemma *nsets-self* [simp]: $nsets\ \{..<m\}\ m = \{\{..<m\}\}$
 ⟨proof⟩

lemma *nsets-zero* [simp]: $nsets\ A\ 0 = \{\{\}\}$
 ⟨proof⟩

lemma *nsets-one*: $nsets\ A\ (Suc\ 0) = (\lambda x. \{x\})\ 'A$
 ⟨proof⟩

lemma *inj-on-nsets*:
 assumes *inj-on* $f\ A$
 shows *inj-on* $(\lambda X. f\ 'X)\ ([A]^n)$
 ⟨proof⟩

lemma *bij-betw-nsets*:
 assumes *bij-betw* $f\ A\ B$

shows *bij-betw* $(\lambda X. f \text{ ‘ } X) ([A]^n) ([B]^n)$
 $\langle \textit{proof} \rangle$

lemma *nset-image-obtains*:
assumes $X \in [f \text{ ‘ } A]^k$ *inj-on* f A
obtains Y **where** $Y \in [A]^k$ $X = f \text{ ‘ } Y$
 $\langle \textit{proof} \rangle$

lemma *nsets-image-funcset*:
assumes $g \in S \rightarrow T$ **and** *inj-on* g S
shows $(\lambda X. g \text{ ‘ } X) \in [S]^k \rightarrow [T]^k$
 $\langle \textit{proof} \rangle$

lemma *nsets-compose-image-funcset*:
assumes $f: f \in [T]^k \rightarrow D$ **and** $g \in S \rightarrow T$ **and** *inj-on* g S
shows $f \circ (\lambda X. g \text{ ‘ } X) \in [S]^k \rightarrow D$
 $\langle \textit{proof} \rangle$

90.1.2 Further properties, involving equipollence

lemma *nsets-lepoll-cong*:
assumes $A \lesssim B$
shows $[A]^k \lesssim [B]^k$
 $\langle \textit{proof} \rangle$

lemma *nsets-epoll-cong*:
assumes $A \approx B$
shows $[A]^k \approx [B]^k$
 $\langle \textit{proof} \rangle$

lemma *infinite-imp-infinite-nsets*:
assumes *inf*: *infinite* A **and** $k > 0$
shows *infinite* $([A]^k)$
 $\langle \textit{proof} \rangle$

lemma *finite-nsets-iff*:
assumes $k > 0$
shows *finite* $([A]^k) \longleftrightarrow \textit{finite } A$
 $\langle \textit{proof} \rangle$

lemma *card-nsets [simp]*: $\textit{card } (nsets \ A \ k) = \textit{card } A$ *choose* k
 $\langle \textit{proof} \rangle$

90.1.3 Partition predicates

definition *monochromatic* $\equiv \lambda \beta \ \alpha \ \gamma \ i. \exists H \in nsets \ \beta \ \alpha. f \text{ ‘ } (nsets \ H \ \gamma) \subseteq \{i\}$

uniform partition sizes

definition *partn* $:: 'a \ \textit{set} \Rightarrow \textit{nat} \Rightarrow \textit{nat} \Rightarrow 'b \ \textit{set} \Rightarrow \textit{bool}$

where $\text{partn } \beta \alpha \gamma \delta \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \delta. \exists \xi \in \delta. \text{monochromatic } \beta \alpha \gamma f \xi$
 partition sizes enumerated in a list

definition $\text{partn-lst} :: 'a \text{ set} \Rightarrow \text{nat list} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where $\text{partn-lst } \beta \alpha \gamma \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \{..<\text{length } \alpha\}. \exists i < \text{length } \alpha. \text{monochromatic } \beta (\alpha!i) \gamma f i$

There’s always a 0-clique

lemma $\text{partn-lst-0}: \gamma > 0 \implies \text{partn-lst } \beta (0\#\alpha) \gamma$
 $\langle \text{proof} \rangle$

lemma $\text{partn-lst-0'}: \gamma > 0 \implies \text{partn-lst } \beta (a\#0\#\alpha) \gamma$
 $\langle \text{proof} \rangle$

lemma $\text{partn-lst-greater-resource}$:

fixes $M::\text{nat}$

assumes $M: \text{partn-lst } \{..<M\} \alpha \gamma$ **and** $M \leq N$

shows $\text{partn-lst } \{..<N\} \alpha \gamma$

$\langle \text{proof} \rangle$

lemma $\text{partn-lst-fewer-colours}$:

assumes $\text{major}: \text{partn-lst } \beta (n\#\alpha) \gamma$ **and** $n \geq \gamma$

shows $\text{partn-lst } \beta \alpha \gamma$

$\langle \text{proof} \rangle$

lemma $\text{partn-lst-eq-partn}: \text{partn-lst } \{..<n\} [m,m] 2 = \text{partn } \{..<n\} m 2 \{..<2::\text{nat}\}$
 $\langle \text{proof} \rangle$

lemma partn-lstE :

assumes $\text{partn-lst } \beta \alpha \gamma f \in \text{nsets } \beta \gamma \rightarrow \{..<l\} \text{length } \alpha = l$

obtains $i H$ **where** $i < \text{length } \alpha$ $H \in \text{nsets } \beta (\alpha!i) f ' (\text{nsets } H \gamma) \subseteq \{i\}$

$\langle \text{proof} \rangle$

lemma partn-lst-less :

assumes $M: \text{partn-lst } \beta \alpha n$ **and** $\text{eq}: \text{length } \alpha' = \text{length } \alpha$

and $\text{le}: \bigwedge i. i < \text{length } \alpha \implies \alpha!i \leq \alpha!i$

shows $\text{partn-lst } \beta \alpha' n$

$\langle \text{proof} \rangle$

90.2 Finite versions of Ramsey’s theorem

To distinguish the finite and infinite ones, lower and upper case names are used (ramsey vs Ramsey).

90.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers

The Erds–Szekeres bound, essentially extracted from the proof

```

fun ES :: [nat,nat,nat]  $\Rightarrow$  nat
  where ES 0 k l = max k l
    |   ES (Suc r) k l =
      (if r=0 then k+l-1
       else if k=0  $\vee$  l=0 then 1 else Suc (ES r (ES (Suc r) (k-1) l) (ES (Suc
r) k (l-1))))

```

```

declare ES.simps [simp del]

```

```

lemma ES-0 [simp]: ES 0 k l = max k l
  <proof>

```

```

lemma ES-1 [simp]: ES 1 k l = k+l-1
  <proof>

```

```

lemma ES-2: ES 2 k l = (if k=0  $\vee$  l=0 then 1 else ES 2 (k-1) l + ES 2 k (l-1))
  <proof>

```

The Erds–Szekeres upper bound

```

lemma ES2-choose: ES 2 k l = (k+l) choose k
  <proof>

```

90.2.2 Trivial cases

Vacuous, since we are dealing with 0-sets!

```

lemma ramsey0:  $\exists N::nat.$  partn-lst {..N} [q1,q2] 0
  <proof>

```

Just the pigeon hole principle, since we are dealing with 1-sets

```

lemma ramsey1-explicit: partn-lst {..q0 + q1 - Suc 0} [q0,q1] 1
  <proof>

```

```

lemma ramsey1:  $\exists N::nat.$  partn-lst {..N} [q0,q1] 1
  <proof>

```

90.2.3 Ramsey’s theorem with TWO colours and arbitrary exponents (hypergraph version)

```

lemma ramsey-induction-step:

```

```

  fixes p::nat
    assumes p1: partn-lst {..p1} [q1-1,q2] (Suc r) and p2: partn-lst {..p2}
[q1,q2-1] (Suc r)
    and p: partn-lst {..p} [p1,p2] r
    and q1>0 q2>0
  shows partn-lst {..Suc p} [q1, q2] (Suc r)
  <proof>

```

```

proposition ramsey2-full: partn-lst {..ES r q1 q2} [q1,q2] r
  <proof>

```

90.2.4 Full Ramsey’s theorem with multiple colours and arbitrary exponents

theorem *ramsey-full*: $\exists N::nat. \text{partn-lst } \{..<N\} \text{ qs } r$
 $\langle \text{proof} \rangle$

90.2.5 Simple graph version

This is the most basic version in terms of cliques and independent sets, i.e. the version for graphs and 2 colours.

definition *clique* $V E \longleftrightarrow (\forall v \in V. \forall w \in V. v \neq w \longrightarrow \{v, w\} \in E)$

definition *indep* $V E \longleftrightarrow (\forall v \in V. \forall w \in V. v \neq w \longrightarrow \{v, w\} \notin E)$

lemma *clique-Un*: $\llbracket \text{clique } K F; \text{clique } L F; \forall v \in K. \forall w \in L. v \neq w \longrightarrow \{v, w\} \in F \rrbracket$
 $\implies \text{clique } (K \cup L) F$
 $\langle \text{proof} \rangle$

lemma *null-clique[simp]*: *clique* $\{\} E$ **and** *null-indep[simp]*: *indep* $\{\} E$
 $\langle \text{proof} \rangle$

lemma *smaller-clique*: $\llbracket \text{clique } R E; R' \subseteq R \rrbracket \implies \text{clique } R' E$
 $\langle \text{proof} \rangle$

lemma *smaller-indep*: $\llbracket \text{indep } R E; R' \subseteq R \rrbracket \implies \text{indep } R' E$
 $\langle \text{proof} \rangle$

lemma *ramsey2*:
 $\exists r \geq 1. \forall (V::'a \text{ set}) (E::'a \text{ set set}). \text{finite } V \wedge \text{card } V \geq r \longrightarrow$
 $(\exists R \subseteq V. \text{card } R = m \wedge \text{clique } R E \vee \text{card } R = n \wedge \text{indep } R E)$
 $\langle \text{proof} \rangle$

90.3 Preliminaries for the infinitary version

90.3.1 “Axiom” of Dependent Choice

primrec *choice* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow \text{nat} \Rightarrow 'a$

where — An integer-indexed chain of choices

choice-0: *choice* $P r 0 = (\text{SOME } x. P x)$

| *choice-Suc*: *choice* $P r (\text{Suc } n) = (\text{SOME } y. P y \wedge (\text{choice } P r n, y) \in r)$

lemma *choice-n*:
assumes $P0$: $P x0$
and $P\text{step}$: $\bigwedge x. P x \implies \exists y. P y \wedge (x, y) \in r$
shows $P (\text{choice } P r n)$
 $\langle \text{proof} \rangle$

lemma *dependent-choice*:
assumes *trans*: *trans* r
and $P0$: $P x0$
and $P\text{step}$: $\bigwedge x. P x \implies \exists y. P y \wedge (x, y) \in r$

obtains $f :: \text{nat} \Rightarrow 'a \text{ where } \bigwedge n. P(f\ n) \text{ and } \bigwedge n\ m. n < m \implies (f\ n, f\ m) \in r$
 $\langle \text{proof} \rangle$

90.3.2 Partition functions

definition $\text{part-fn} :: \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ set} \Rightarrow ('a \text{ set} \Rightarrow \text{nat}) \Rightarrow \text{bool}$

— the function f partitions the r -subsets of the typically infinite set Y into s distinct categories.

where $\text{part-fn}\ r\ s\ Y\ f \longleftrightarrow (f \in \text{nsets}\ Y\ r \rightarrow \{..<s\})$

For induction, we decrease the value of r in partitions.

lemma $\text{part-fn-Suc-imp-part-fn}$:

$\llbracket \text{infinite}\ Y; \text{part-fn}\ (\text{Suc}\ r)\ s\ Y\ f; y \in Y \rrbracket \implies \text{part-fn}\ r\ s\ (Y - \{y\})\ (\lambda u. f\ (\text{insert}\ y\ u))$
 $\langle \text{proof} \rangle$

lemma part-fn-subset : $\text{part-fn}\ r\ s\ YY\ f \implies Y \subseteq YY \implies \text{part-fn}\ r\ s\ Y\ f$
 $\langle \text{proof} \rangle$

90.4 Ramsey’s Theorem: Infinitary Version

lemma Ramsey-induction :

fixes $s\ r :: \text{nat}$

and $YY :: 'a \text{ set}$

and $f :: 'a \text{ set} \Rightarrow \text{nat}$

assumes $\text{infinite}\ YY\ \text{part-fn}\ r\ s\ YY\ f$

shows $\exists Y'\ t'. Y' \subseteq YY \wedge \text{infinite}\ Y' \wedge t' < s \wedge (\forall X. X \subseteq Y' \wedge \text{finite}\ X \wedge \text{card}\ X = r \longrightarrow f\ X = t')$
 $\langle \text{proof} \rangle$

theorem Ramsey :

fixes $s\ r :: \text{nat}$

and $Z :: 'a \text{ set}$

and $f :: 'a \text{ set} \Rightarrow \text{nat}$

shows

$\llbracket \text{infinite}\ Z;$
 $\forall X. X \subseteq Z \wedge \text{finite}\ X \wedge \text{card}\ X = r \longrightarrow f\ X < s \rrbracket$
 $\implies \exists Y\ t. Y \subseteq Z \wedge \text{infinite}\ Y \wedge t < s$
 $\wedge (\forall X. X \subseteq Y \wedge \text{finite}\ X \wedge \text{card}\ X = r \longrightarrow f\ X = t)$
 $\langle \text{proof} \rangle$

corollary Ramsey2 :

fixes $s :: \text{nat}$

and $Z :: 'a \text{ set}$

and $f :: 'a \text{ set} \Rightarrow \text{nat}$

assumes $\text{inf}Z$: $\text{infinite}\ Z$

and part : $\forall x \in Z. \forall y \in Z. x \neq y \longrightarrow f\ \{x, y\} < s$

shows $\exists Y\ t. Y \subseteq Z \wedge \text{infinite}\ Y \wedge t < s \wedge (\forall x \in Y. \forall y \in Y. x \neq y \longrightarrow f\ \{x, y\} = t)$

<proof>

corollary *Ramsey-nsets:*

fixes $f :: 'a \text{ set} \Rightarrow \text{nat}$

assumes $\text{infinite } Z \text{ } f \text{ ' nsets } Z \text{ } r \subseteq \{..<s\}$

obtains $Y \text{ } t \text{ where } Y \subseteq Z \text{ infinite } Y \text{ } t < s \text{ } f \text{ ' nsets } Y \text{ } r \subseteq \{t\}$

<proof>

90.5 Disjunctive Well-Foundedness

An application of Ramsey’s theorem to program termination. See [4].

definition $\text{disj-wf} :: ('a \times 'a) \text{ set} \Rightarrow \text{bool}$

where $\text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i<n. \text{wf } (T \text{ } i)) \wedge r = (\bigcup i<n. T \text{ } i))$

definition $\text{transition-idx} :: (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow ('a \times 'a) \text{ set}) \Rightarrow \text{nat set} \Rightarrow \text{nat}$

where $\text{transition-idx } s \text{ } T \text{ } A = (\text{LEAST } k. \exists i \text{ } j. A = \{i, j\} \wedge i < j \wedge (s \text{ } j, s \text{ } i) \in T \text{ } k)$

lemma *transition-idx-less:*

assumes $i < j \text{ } (s \text{ } j, s \text{ } i) \in T \text{ } k \text{ } k < n$

shows $\text{transition-idx } s \text{ } T \text{ } \{i, j\} < n$

<proof>

lemma *transition-idx-in:*

assumes $i < j \text{ } (s \text{ } j, s \text{ } i) \in T \text{ } k$

shows $(s \text{ } j, s \text{ } i) \in T \text{ } (\text{transition-idx } s \text{ } T \text{ } \{i, j\})$

<proof>

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf:* $\text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i<n. \text{wf } (T \text{ } i)) \wedge r \subseteq (\bigcup i<n. T \text{ } i))$

<proof>

theorem *trans-disj-wf-implies-wf:*

assumes $\text{trans } r$

and $\text{disj-wf } r$

shows $\text{wf } r$

<proof>

end

91 Modulo and congruence on the reals

theory *Real-Mod*

imports *Complex-Main*

begin

definition $rmod :: real \Rightarrow real \Rightarrow real$ (**infixl** $\langle rmod \rangle$ 70) **where**
 $x \ rmod \ y = x - |y| * of_int \lfloor x / |y| \rfloor$

lemma $rmod_conv_frac$: $y \neq 0 \implies x \ rmod \ y = frac \ (x / |y|) * |y|$
 $\langle proof \rangle$

lemma $rmod_conv_frac'$: $x \ rmod \ y = (if \ y = 0 \ then \ x \ else \ frac \ (x / |y|) * |y|)$
 $\langle proof \rangle$

lemma $rmod_rmod$ [simp]: $(x \ rmod \ y) \ rmod \ y = x \ rmod \ y$
 $\langle proof \rangle$

lemma $rmod_0_right$ [simp]: $x \ rmod \ 0 = x$
 $\langle proof \rangle$

lemma $rmod_less$: $m > 0 \implies x \ rmod \ m < m$
 $\langle proof \rangle$

lemma $rmod_less_abs$: $m \neq 0 \implies x \ rmod \ m < |m|$
 $\langle proof \rangle$

lemma $rmod_le$: $m > 0 \implies x \ rmod \ m \leq m$
 $\langle proof \rangle$

lemma $rmod_nonneg$: $m \neq 0 \implies x \ rmod \ m \geq 0$
 $\langle proof \rangle$

lemma $rmod_unique$:
assumes $z \in \{0..<|y|\}$ $x = z + of_int \ n * y$
shows $x \ rmod \ y = z$
 $\langle proof \rangle$

lemma $rmod_0$ [simp]: $0 \ rmod \ z = 0$
 $\langle proof \rangle$

lemma $rmod_add$: $(x \ rmod \ z + y \ rmod \ z) \ rmod \ z = (x + y) \ rmod \ z$
 $\langle proof \rangle$

lemma $rmod_diff$: $(x \ rmod \ z - y \ rmod \ z) \ rmod \ z = (x - y) \ rmod \ z$
 $\langle proof \rangle$

lemma $rmod_self$ [simp]: $x \ rmod \ x = 0$
 $\langle proof \rangle$

lemma $rmod_self_multiple_int$ [simp]: $(of_int \ n * x) \ rmod \ x = 0$
 $\langle proof \rangle$

lemma *rmod-self-multiple-nat* [simp]: $(\text{of-nat } n * x) \text{ rmod } x = 0$
 ⟨proof⟩

lemma *rmod-self-multiple-numeral* [simp]: $(\text{numeral } n * x) \text{ rmod } x = 0$
 ⟨proof⟩

lemma *rmod-self-multiple-int'* [simp]: $(x * \text{of-int } n) \text{ rmod } x = 0$
 ⟨proof⟩

lemma *rmod-self-multiple-nat'* [simp]: $(x * \text{of-nat } n) \text{ rmod } x = 0$
 ⟨proof⟩

lemma *rmod-self-multiple-numeral'* [simp]: $(x * \text{numeral } n) \text{ rmod } x = 0$
 ⟨proof⟩

lemma *rmod-idem* [simp]: $x \in \{0..<|y|\} \implies x \text{ rmod } y = x$
 ⟨proof⟩

definition *rcong* :: $\text{real} \Rightarrow \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}$
 ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix } rcong \rangle [- = -] '(\text{' rmod -'}) \rangle \rangle$)
where $[x = y] (\text{rmod } m) \longleftrightarrow x \text{ rmod } m = y \text{ rmod } m$

named-theorems *rcong-intros*

lemma *rcong-0-right* [simp]: $[x = y] (\text{rmod } 0) \longleftrightarrow x = y$
 ⟨proof⟩

lemma *rcong-0-iff*: $[x = 0] (\text{rmod } m) \longleftrightarrow x \text{ rmod } m = 0$
and *rcong-0-iff'*: $[0 = x] (\text{rmod } m) \longleftrightarrow x \text{ rmod } m = 0$
 ⟨proof⟩

lemma *rcong-refl* [simp, intro!, *rcong-intros*]: $[x = x] (\text{rmod } m)$
 ⟨proof⟩

lemma *rcong-sym*: $[y = x] (\text{rmod } m) \implies [x = y] (\text{rmod } m)$
 ⟨proof⟩

lemma *rcong-sym-iff*: $[y = x] (\text{rmod } m) \longleftrightarrow [x = y] (\text{rmod } m)$
 ⟨proof⟩

lemma *rcong-trans* [trans]: $[x = y] (\text{rmod } m) \implies [y = z] (\text{rmod } m) \implies [x = z]$
 (*rmod m*)
 ⟨proof⟩

lemma *rcong-add* [*rcong-intros*]:
 $[a = b] (\text{rmod } m) \implies [c = d] (\text{rmod } m) \implies [a + c = b + d] (\text{rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-diff* [*rcong-intros*]:

$$[a = b] \text{ (rmod } m) \implies [c = d] \text{ (rmod } m) \implies [a - c = b - d] \text{ (rmod } m)$$

$\langle \text{proof} \rangle$

lemma *rcong-uminus* [*rcong-intros*]:

$$[a = b] \text{ (rmod } m) \implies [-a = -b] \text{ (rmod } m)$$

$\langle \text{proof} \rangle$

lemma *rcong-uminus-uminus-iff* [*simp*]: $[-x = -y] \text{ (rmod } m) \longleftrightarrow [x = y] \text{ (rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-uminus-left-iff*: $[-x = y] \text{ (rmod } m) \longleftrightarrow [x = -y] \text{ (rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-add-right-cancel* [*simp*]: $[a + c = b + c] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-add-left-cancel* [*simp*]: $[c + a = c + b] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-diff-right-cancel* [*simp*]: $[a - c = b - c] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-diff-left-cancel* [*simp*]: $[c - a = c - b] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-rmod-right-iff* [*simp*]: $[a = (b \text{ rmod } m)] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$

and *rcong-rmod-left-iff* [*simp*]: $[(a \text{ rmod } m) = b] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$

$\langle \text{proof} \rangle$

lemma *rcong-rmod-left* [*rcong-intros*]: $[a = b] \text{ (rmod } m) \implies [(a \text{ rmod } m) = b]$

and *rcong-rmod-right* [*rcong-intros*]: $[a = b] \text{ (rmod } m) \implies [a = (b \text{ rmod } m)]$

$\langle \text{proof} \rangle$

lemma *rcong-mult-of-int-0-left-left* [*rcong-intros*]: $[0 = \text{of-int } n * m] \text{ (rmod } m)$

and *rcong-mult-of-int-0-right-left* [*rcong-intros*]: $[0 = m * \text{of-int } n] \text{ (rmod } m)$

and *rcong-mult-of-int-0-left-right* [*rcong-intros*]: $[\text{of-int } n * m = 0] \text{ (rmod } m)$

and *rcong-mult-of-int-0-right-right* [*rcong-intros*]: $[m * \text{of-int } n = 0] \text{ (rmod } m)$

$\langle proof \rangle$

lemma *rcong-altdef*: $[a = b] \text{ (rmod } m) \longleftrightarrow (\exists n. b = a + \text{ of-int } n * m)$
 $\langle proof \rangle$

lemma *rcong-conv-diff-rmod-eq-0*: $[x = y] \text{ (rmod } m) \longleftrightarrow (x - y) \text{ rmod } m = 0$
 $\langle proof \rangle$

lemma *rcong-imp-eq*:
assumes $[x = y] \text{ (rmod } m) \mid x - y| < |m|$
shows $x = y$
 $\langle proof \rangle$

lemma *rcong-mult-modulus*:
assumes $[a = b] \text{ (rmod } (m / c)) \ c \neq 0$
shows $[a * c = b * c] \text{ (rmod } m)$
 $\langle proof \rangle$

lemma *rcong-divide-modulus*:
assumes $[a = b] \text{ (rmod } (m * c)) \ c \neq 0$
shows $[a / c = b / c] \text{ (rmod } m)$
 $\langle proof \rangle$

lemma *sin-rmod [simp]*: $\sin (x \text{ rmod } (2 * \pi)) = \sin x$
and *cos-rmod [simp]*: $\cos (x \text{ rmod } (2 * \pi)) = \cos x$
 $\langle proof \rangle$

lemma *tan-rmod [simp]*: $\tan (x \text{ rmod } (2 * \pi)) = \tan x$
and *cot-rmod [simp]*: $\cot (x \text{ rmod } (2 * \pi)) = \cot x$
and *cis-rmod [simp]*: $\text{cis } (x \text{ rmod } (2 * \pi)) = \text{cis } x$
and *rcis-rmod [simp]*: $\text{rcis } r \text{ (} x \text{ rmod } (2 * \pi)) = \text{rcis } r \ x$
 $\langle proof \rangle$

lemma *cis-eq-iff*: $\text{cis } a = \text{cis } b \longleftrightarrow [a = b] \text{ (rmod } (2 * \pi))$
 $\langle proof \rangle$

lemma *cis-eq-1-iff*: $\text{cis } a = 1 \longleftrightarrow (\exists n. a = \text{ of-int } n * (2 * \pi))$
 $\langle proof \rangle$

lemma *cis-cong*:
assumes $[a = b] \text{ (rmod } 2 * \pi)$
shows $\text{cis } a = \text{cis } b$
 $\langle proof \rangle$

lemma *rcis-cong*:
assumes $[a = b] \text{ (rmod } 2 * \pi)$
shows $\text{rcis } r \ a = \text{rcis } r \ b$
 $\langle proof \rangle$

lemma *sin-rcong*: $[x = y] \text{ (rmod } (2 * \pi)) \implies \sin x = \sin y$
and *cos-rcong*: $[x = y] \text{ (rmod } (2 * \pi)) \implies \cos x = \cos y$
 $\langle \text{proof} \rangle$

lemma *sin-eq-sin-conv-rmod*:
assumes $\sin x = \sin y$
shows $[x = y] \text{ (rmod } 2 * \pi) \vee [x = \pi - y] \text{ (rmod } 2 * \pi)$
 $\langle \text{proof} \rangle$

lemma *cos-eq-cos-conv-rmod*:
assumes $\cos x = \cos y$
shows $[x = y] \text{ (rmod } 2 * \pi) \vee [x = -y] \text{ (rmod } 2 * \pi)$
 $\langle \text{proof} \rangle$

lemma *sin-eq-sin-conv-rmod-iff*:
 $\sin x = \sin y \longleftrightarrow [x = y] \text{ (rmod } 2 * \pi) \vee [x = \pi - y] \text{ (rmod } 2 * \pi)$
 $\langle \text{proof} \rangle$

lemma *cos-eq-cos-conv-rmod-iff*:
 $\cos x = \cos y \longleftrightarrow [x = y] \text{ (rmod } 2 * \pi) \vee [x = -y] \text{ (rmod } 2 * \pi)$
 $\langle \text{proof} \rangle$

end

92 Generic reflection and reification

theory *Reflection*
imports *Main*
begin

$\langle \text{ML} \rangle$

end

theory *Rewrite*
imports *Main*
begin

consts *rewrite-HOLE* :: $'a::\{\}$ $(\lhd \sqcap \rhd)$

lemma *eta-expand*:
fixes $f :: 'a::\{\} \Rightarrow 'b::\{\}$
shows $f \equiv \lambda x. f\ x$ $\langle \text{proof} \rangle$

lemma *imp-cong-eq*:
 $(\text{PROP } A \implies (\text{PROP } B \implies \text{PROP } C)) \equiv (\text{PROP } B' \implies \text{PROP } C') \equiv$
 $((\text{PROP } B \implies \text{PROP } A \implies \text{PROP } C) \equiv (\text{PROP } B' \implies \text{PROP } A \implies \text{PROP } C'))$

⟨proof⟩

⟨ML⟩

end

93 Assigning lengths to types by type classes

```
theory Type-Length
imports Numeral-Type
begin
```

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in `Numeral_Type.thy`.

```
class len0 =
  fixes len-of :: 'a itself ⇒ nat

syntax -type-length :: type ⇒ nat  (⟨(1LENGTH/(1'(-'))⟩)⟩)
syntax-consts -type-length ⇒ len-of
translations LENGTH('a) ↦ CONST len-of TYPE('a)
⟨ML⟩
```

Some theorems are only true on words with length greater 0.

```
class len = len0 +
  assumes len-gt-0 [iff]: 0 < LENGTH('a)
begin

lemma len-not-eq-0 [simp]:
  LENGTH('a) ≠ 0
  ⟨proof⟩

end
```

```
instantiation num0 and num1 :: len0
begin
```

```
definition len-num0: len-of (- :: num0 itself) = 0
definition len-num1: len-of (- :: num1 itself) = 1
```

```
instance ⟨proof⟩
```

end

```
instantiation bit0 and bit1 :: (len0) len0
begin
```

```
definition len-bit0: len-of (- :: 'a::len0 bit0 itself) = 2 * LENGTH('a)
definition len-bit1: len-of (- :: 'a::len0 bit1 itself) = 2 * LENGTH('a) + 1
```

instance $\langle proof \rangle$

end

lemmas *len-of-numeral-defs* [simp] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len*

$\langle proof \rangle$

instance *bit0* :: (*len*) *len*

$\langle proof \rangle$

instance *bit1* :: (*len0*) *len*

$\langle proof \rangle$

instantiation *Enum.finite-1* :: *len*

begin

definition

len-of-finite-1 (*x* :: *Enum.finite-1* *itself*) \equiv (*1* :: *nat*)

instance

$\langle proof \rangle$

end

instantiation *Enum.finite-2* :: *len*

begin

definition

len-of-finite-2 (*x* :: *Enum.finite-2* *itself*) \equiv (*2* :: *nat*)

instance

$\langle proof \rangle$

end

instantiation *Enum.finite-3* :: *len*

begin

definition

len-of-finite-3 (*x* :: *Enum.finite-3* *itself*) \equiv (*4* :: *nat*)

instance

$\langle proof \rangle$

end

lemma *length-less-eq-Suc-0-iff* [simp]:

$\langle LENGTH('a::len) \leq Suc\ 0 \longleftrightarrow LENGTH('a) = Suc\ 0 \rangle$

$\langle \text{proof} \rangle$

lemma *length-not-greater-eq-2-iff* [simp]:
 $\langle \neg 2 \leq \text{LENGTH}('a::\text{len}) \longleftrightarrow \text{LENGTH}('a) = \text{Suc } 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *less-eq-decr-length-iff* [simp]:
 $\langle n \leq \text{LENGTH}('a::\text{len}) - \text{Suc } 0 \longleftrightarrow n < \text{LENGTH}('a) \rangle$
 $\langle \text{proof} \rangle$

lemma *decr-length-less-iff* [simp]:
 $\langle \text{LENGTH}('a::\text{len}) - \text{Suc } 0 < n \longleftrightarrow \text{LENGTH}('a) \leq n \rangle$
 $\langle \text{proof} \rangle$

context *linordered-idom*
begin

lemma *two-less-eq-exp-length* [simp]:
 $\langle 2 \leq 2 \wedge \text{LENGTH}('b::\text{len}) \rangle$
 $\langle \text{proof} \rangle$

end

end

94 Saturated arithmetic

theory *Saturated*
imports *Natural-Type Type-Length*
begin

94.1 The type of saturated naturals

typedef (overloaded) $('a::\text{len}) \text{ sat} = \{.. \text{LENGTH}('a)\}$
morphisms *nat-of Abs-sat*
 $\langle \text{proof} \rangle$

lemma *sat-eqI*:
 $\text{nat-of } m = \text{nat-of } n \implies m = n$
 $\langle \text{proof} \rangle$

lemma *sat-eq-iff*:
 $m = n \longleftrightarrow \text{nat-of } m = \text{nat-of } n$
 $\langle \text{proof} \rangle$

lemma *Abs-sat-nat-of* [code abstype]:
 $\text{Abs-sat } (\text{nat-of } n) = n$
 $\langle \text{proof} \rangle$

definition $Abs\text{-}sat' :: nat \Rightarrow 'a::len\ sat$ **where**
 $Abs\text{-}sat' n = Abs\text{-}sat (\min (LENGTH('a)) n)$

lemma $nat\text{-}of\text{-}Abs\text{-}sat'$ [simp]:
 $nat\text{-}of (Abs\text{-}sat' n :: ('a::len) sat) = \min (LENGTH('a)) n$
 $\langle proof \rangle$

lemma $nat\text{-}of\text{-}le\text{-}len\text{-}of$ [simp]:
 $nat\text{-}of (n :: ('a::len) sat) \leq LENGTH('a)$
 $\langle proof \rangle$

lemma $min\text{-}len\text{-}of\text{-}nat\text{-}of$ [simp]:
 $\min (LENGTH('a)) (nat\text{-}of (n::('a::len) sat)) = nat\text{-}of n$
 $\langle proof \rangle$

lemma $min\text{-}nat\text{-}of\text{-}len\text{-}of$ [simp]:
 $\min (nat\text{-}of (n::('a::len) sat)) (LENGTH('a)) = nat\text{-}of n$
 $\langle proof \rangle$

lemma $Abs\text{-}sat'\text{-}nat\text{-}of$ [simp]:
 $Abs\text{-}sat' (nat\text{-}of n) = n$
 $\langle proof \rangle$

instantiation $sat :: (len) linorder$
begin

definition
 $less\text{-}eq\text{-}sat\text{-}def: x \leq y \longleftrightarrow nat\text{-}of x \leq nat\text{-}of y$

definition
 $less\text{-}sat\text{-}def: x < y \longleftrightarrow nat\text{-}of x < nat\text{-}of y$

instance
 $\langle proof \rangle$

end

instantiation $sat :: (len) \{minus, comm\text{-}semiring\text{-}1\}$
begin

definition
 $0 = Abs\text{-}sat' 0$

definition
 $1 = Abs\text{-}sat' 1$

lemma $nat\text{-}of\text{-}zero\text{-}sat$ [simp, code abstract]:
 $nat\text{-}of 0 = 0$
 $\langle proof \rangle$

lemma *nat-of-one-sat* [*simp*, *code abstract*]:

$\text{nat-of } 1 = \min 1 (\text{LENGTH}('a))$
 $\langle \text{proof} \rangle$

definition

$x + y = \text{Abs-sat}' (\text{nat-of } x + \text{nat-of } y)$

lemma *nat-of-plus-sat* [*simp*, *code abstract*]:

$\text{nat-of } (x + y) = \min (\text{nat-of } x + \text{nat-of } y) (\text{LENGTH}('a))$
 $\langle \text{proof} \rangle$

definition

$x - y = \text{Abs-sat}' (\text{nat-of } x - \text{nat-of } y)$

lemma *nat-of-minus-sat* [*simp*, *code abstract*]:

$\text{nat-of } (x - y) = \text{nat-of } x - \text{nat-of } y$
 $\langle \text{proof} \rangle$

definition

$x * y = \text{Abs-sat}' (\text{nat-of } x * \text{nat-of } y)$

lemma *nat-of-times-sat* [*simp*, *code abstract*]:

$\text{nat-of } (x * y) = \min (\text{nat-of } x * \text{nat-of } y) (\text{LENGTH}('a))$
 $\langle \text{proof} \rangle$

instance

$\langle \text{proof} \rangle$

end

instantiation *sat* :: (*len*) *ordered-comm-semiring*

begin

instance

$\langle \text{proof} \rangle$

end

lemma *Abs-sat'-eq-of-nat*: $\text{Abs-sat}' n = \text{of-nat } n$

$\langle \text{proof} \rangle$

abbreviation *Sat* :: $\text{nat} \Rightarrow 'a::\text{len } \text{sat}$ **where**

$\text{Sat} \equiv \text{of-nat}$

lemma *nat-of-Sat* [*simp*]:

$\text{nat-of } (\text{Sat } n :: ('a::\text{len}) \text{ sat}) = \min (\text{LENGTH}('a)) n$
 $\langle \text{proof} \rangle$

```

lemma [code-abbrev]:
  of-nat (numeral k) = (numeral k :: 'a::len sat)
  ⟨proof⟩

context
begin

qualified definition sat-of-nat :: nat ⇒ ('a::len) sat
  where [code-abbrev]: sat-of-nat = of-nat

lemma [code abstract]:
  nat-of (sat-of-nat n :: ('a::len) sat) = min (LENGTH('a)) n
  ⟨proof⟩

end

instance sat :: (len) finite
  ⟨proof⟩

instantiation sat :: (len) equal
begin

definition HOL.equal A B ⟷ nat-of A = nat-of B

instance
  ⟨proof⟩

end

instantiation sat :: (len) {bounded-lattice, distrib-lattice}
begin

definition (inf :: 'a sat ⇒ 'a sat ⇒ 'a sat) = min
definition (sup :: 'a sat ⇒ 'a sat ⇒ 'a sat) = max
definition bot = (0 :: 'a sat)
definition top = Sat (LENGTH('a))

instance
  ⟨proof⟩

end

instantiation sat :: (len) {Inf, Sup}
begin

global-interpretation Inf-sat: semilattice-neutr-set min ⟨top :: 'a sat⟩
defines Inf-sat = Inf-sat.F
  ⟨proof⟩

```

```

global-interpretation Sup-sat: semilattice-neutr-set max ⟨bot :: 'a sat
  defines Sup-sat = Sup-sat.F
  ⟨proof⟩

instance ⟨proof⟩

end

instance sat :: (len) complete-lattice
  ⟨proof⟩

```

94.2 Enumeration

```

lemma inj-on-sat-of-nat:
  shows inj-on (of-nat :: nat ⇒ 'a::len sat) {0..LENGTH('a)}
  ⟨proof⟩

lemma UNIV-sat-eq-of-nat:
  shows (UNIV :: 'a::len sat set) = of-nat ‘ {0..LENGTH('a)} (is ?lhs = ?rhs)
  ⟨proof⟩

instantiation sat :: (len) enum
begin

definition enum-sat :: 'a sat list where
  enum-sat = map of-nat [0..Suc(LENGTH('a))]

definition enum-all-sat :: ('a sat ⇒ bool) ⇒ bool where
  enum-all-sat = All

definition enum-ex-sat :: ('a sat ⇒ bool) ⇒ bool where
  enum-ex-sat = Ex

instance
  ⟨proof⟩

end

```

```

lemma enum-sat-code [code]:
  fixes P :: 'a::len sat ⇒ bool
  shows Enum.enum-all P ⟷ list-all P Enum.enum
  and Enum.enum-ex P ⟷ list-ex P Enum.enum
  ⟨proof⟩

end

```

95 Set Idioms

```

theory Set-Idioms

```

imports *Countable-Set*

begin

95.1 Idioms for being a suitable union/intersection of something

definition *union-of* :: ('a set set \Rightarrow bool) \Rightarrow ('a set \Rightarrow bool) \Rightarrow 'a set \Rightarrow bool
 (infixr <union'-of> 60)
 where $P \text{ union-of } Q \equiv \lambda S. \exists \mathcal{U}. P \mathcal{U} \wedge \mathcal{U} \subseteq \text{Collect } Q \wedge \bigcup \mathcal{U} = S$

definition *intersection-of* :: ('a set set \Rightarrow bool) \Rightarrow ('a set \Rightarrow bool) \Rightarrow 'a set \Rightarrow bool
 (infixr <intersection'-of> 60)
 where $P \text{ intersection-of } Q \equiv \lambda S. \exists \mathcal{U}. P \mathcal{U} \wedge \mathcal{U} \subseteq \text{Collect } Q \wedge \bigcap \mathcal{U} = S$

definition *arbitrary*:: 'a set set \Rightarrow bool where *arbitrary* $\mathcal{U} \equiv \text{True}$

lemma *union-of-inc*: $\llbracket P \{S\}; Q S \rrbracket \Longrightarrow (P \text{ union-of } Q) S$
 <proof>

lemma *intersection-of-inc*:
 $\llbracket P \{S\}; Q S \rrbracket \Longrightarrow (P \text{ intersection-of } Q) S$
 <proof>

lemma *union-of-mono*:
 $\llbracket (P \text{ union-of } Q) S; \bigwedge x. Q x \Longrightarrow Q' x \rrbracket \Longrightarrow (P \text{ union-of } Q') S$
 <proof>

lemma *intersection-of-mono*:
 $\llbracket (P \text{ intersection-of } Q) S; \bigwedge x. Q x \Longrightarrow Q' x \rrbracket \Longrightarrow (P \text{ intersection-of } Q') S$
 <proof>

lemma *all-union-of*:
 $(\forall S. (P \text{ union-of } Q) S \longrightarrow R S) \longleftrightarrow (\forall T. P T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcup T))$
 <proof>

lemma *all-intersection-of*:
 $(\forall S. (P \text{ intersection-of } Q) S \longrightarrow R S) \longleftrightarrow (\forall T. P T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcap T))$
 <proof>

lemma *intersection-ofE*:
 $\llbracket (P \text{ intersection-of } Q) S; \bigwedge T. \llbracket P T; T \subseteq \text{Collect } Q \rrbracket \Longrightarrow R(\bigcap T) \rrbracket \Longrightarrow R S$
 <proof>

lemma *union-of-empty*:
 $P \{\} \Longrightarrow (P \text{ union-of } Q) \{\}$
 <proof>

lemma *intersection-of-empty*:

$$P \{\} \implies (P \text{ intersection-of } Q) \text{ UNIV}$$

<proof>

The arbitrary and finite cases

lemma *arbitrary-union-of-alt*:

$$(arbitrary \text{ union-of } Q) S \longleftrightarrow (\forall x \in S. \exists U. Q U \wedge x \in U \wedge U \subseteq S)$$

(is ?lhs = ?rhs)

<proof>

lemma *arbitrary-union-of-empty [simp]*: $(arbitrary \text{ union-of } P) \{\}$

<proof>

lemma *arbitrary-intersection-of-empty [simp]*:

$$(arbitrary \text{ intersection-of } P) \text{ UNIV}$$

<proof>

lemma *arbitrary-union-of-inc*:

$$P S \implies (arbitrary \text{ union-of } P) S$$

<proof>

lemma *arbitrary-intersection-of-inc*:

$$P S \implies (arbitrary \text{ intersection-of } P) S$$

<proof>

lemma *arbitrary-union-of-complement*:

$$(arbitrary \text{ union-of } P) S \longleftrightarrow (arbitrary \text{ intersection-of } (\lambda S. P(- S))) (- S)$$

(is ?lhs = ?rhs)

<proof>

lemma *arbitrary-intersection-of-complement*:

$$(arbitrary \text{ intersection-of } P) S \longleftrightarrow (arbitrary \text{ union-of } (\lambda S. P(- S))) (- S)$$

<proof>

lemma *arbitrary-union-of-idempot [simp]*:

$$arbitrary \text{ union-of } arbitrary \text{ union-of } P = arbitrary \text{ union-of } P$$

<proof>

lemma *arbitrary-intersection-of-idempot*:

$$arbitrary \text{ intersection-of } arbitrary \text{ intersection-of } P = arbitrary \text{ intersection-of } P$$

(is ?lhs = ?rhs)

<proof>

lemma *arbitrary-union-of-Union*:

$$(\bigwedge S. S \in \mathcal{U} \implies (arbitrary \text{ union-of } P) S) \implies (arbitrary \text{ union-of } P) (\bigcup \mathcal{U})$$

<proof>

lemma *arbitrary-union-of-Un*:

$$\llbracket (arbitrary \text{ union-of } P) S; (arbitrary \text{ union-of } P) T \rrbracket$$

$$\begin{array}{c} \implies (\text{arbitrary union-of } P) (S \cup T) \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-intersection-of-Inter:*

$$\begin{array}{c} (\bigwedge S. S \in \mathcal{U} \implies (\text{arbitrary intersection-of } P) S) \implies (\text{arbitrary intersection-of } P) (\bigcap \mathcal{U}) \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-intersection-of-Int:*

$$\begin{array}{c} \llbracket (\text{arbitrary intersection-of } P) S; (\text{arbitrary intersection-of } P) T \rrbracket \\ \implies (\text{arbitrary intersection-of } P) (S \cap T) \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-union-of-Int-eq:*

$$\begin{array}{c} (\forall S T. (\text{arbitrary union-of } P) S \wedge (\text{arbitrary union-of } P) T \\ \longrightarrow (\text{arbitrary union-of } P) (S \cap T)) \\ \longleftrightarrow (\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary union-of } P) (S \cap T)) \text{ (is ?lhs = ?rhs)} \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-intersection-of-Un-eq:*

$$\begin{array}{c} (\forall S T. (\text{arbitrary intersection-of } P) S \wedge (\text{arbitrary intersection-of } P) T \\ \longrightarrow (\text{arbitrary intersection-of } P) (S \cup T)) \longleftrightarrow \\ (\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary intersection-of } P) (S \cup T)) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-empty [simp]:* $(\text{finite union-of } P) \{\}$
 $\langle \text{proof} \rangle$

lemma *finite-intersection-of-empty [simp]:* $(\text{finite intersection-of } P) \text{ UNIV}$
 $\langle \text{proof} \rangle$

lemma *finite-union-of-inc:*

$$\begin{array}{c} P S \implies (\text{finite union-of } P) S \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-intersection-of-inc:*

$$\begin{array}{c} P S \implies (\text{finite intersection-of } P) S \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-complement:*

$$\begin{array}{c} (\text{finite union-of } P) S \longleftrightarrow (\text{finite intersection-of } (\lambda S. P(- S))) (- S) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-intersection-of-complement:*

$$\begin{array}{c} (\text{finite intersection-of } P) S \longleftrightarrow (\text{finite union-of } (\lambda S. P(- S))) (- S) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-idempot [simp]:*

$$\text{finite union-of finite union-of } P = \text{finite union-of } P$$

$\langle \text{proof} \rangle$

lemma *finite-intersection-of-idempot* [simp]:

finite intersection-of finite intersection-of P = finite intersection-of P

$\langle \text{proof} \rangle$

lemma *finite-union-of-Union*:

$\llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{finite union-of } P) S \rrbracket \implies (\text{finite union-of } P) (\bigcup \mathcal{U})$

$\langle \text{proof} \rangle$

lemma *finite-union-of-Un*:

$\llbracket (\text{finite union-of } P) S; (\text{finite union-of } P) T \rrbracket \implies (\text{finite union-of } P) (S \cup T)$

$\langle \text{proof} \rangle$

lemma *finite-intersection-of-Inter*:

$\llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{finite intersection-of } P) S \rrbracket \implies (\text{finite intersection-of } P) (\bigcap \mathcal{U})$

$\langle \text{proof} \rangle$

lemma *finite-intersection-of-Int*:

$\llbracket (\text{finite intersection-of } P) S; (\text{finite intersection-of } P) T \rrbracket$
 $\implies (\text{finite intersection-of } P) (S \cap T)$

$\langle \text{proof} \rangle$

lemma *finite-union-of-Int-eq*:

$(\forall S T. (\text{finite union-of } P) S \wedge (\text{finite union-of } P) T \longrightarrow (\text{finite union-of } P) (S \cap T))$

$\longleftrightarrow (\forall S T. P S \wedge P T \longrightarrow (\text{finite union-of } P) (S \cap T))$

(is ?lhs = ?rhs)

$\langle \text{proof} \rangle$

lemma *finite-intersection-of-Un-eq*:

$(\forall S T. (\text{finite intersection-of } P) S \wedge$
 $(\text{finite intersection-of } P) T$
 $\longrightarrow (\text{finite intersection-of } P) (S \cup T)) \longleftrightarrow$
 $(\forall S T. P S \wedge P T \longrightarrow (\text{finite intersection-of } P) (S \cup T))$

$\langle \text{proof} \rangle$

abbreviation *finite'* :: 'a set \Rightarrow bool

where *finite'* A \equiv *finite* A \wedge A \neq {}

lemma *finite'-intersection-of-Int*:

$\llbracket (\text{finite}' \text{ intersection-of } P) S; (\text{finite}' \text{ intersection-of } P) T \rrbracket$
 $\implies (\text{finite}' \text{ intersection-of } P) (S \cap T)$

$\langle \text{proof} \rangle$

lemma *finite'-intersection-of-inc*:

$P S \implies (\text{finite}' \text{ intersection-of } P) S$

$\langle \text{proof} \rangle$

95.2 The “Relative to” operator

A somewhat cheap but handy way of getting localized forms of various topological concepts (open, closed, borel, fsigma, gdelta etc.)

definition *relative-to* :: [*'a set* \Rightarrow *bool*, *'a set*, *'a set*] \Rightarrow *bool* (**infixl** $\langle \text{relative}'\text{-to} \rangle$ 55)

where $P \text{ relative-to } S \equiv \lambda T. \exists U. P \ U \wedge S \cap U = T$

lemma *relative-to-UNIV* [*simp*]: $(P \text{ relative-to } \text{UNIV}) \ S \longleftrightarrow P \ S$
 $\langle \text{proof} \rangle$

lemma *relative-to-imp-subset*:
 $(P \text{ relative-to } S) \ T \Longrightarrow T \subseteq S$
 $\langle \text{proof} \rangle$

lemma *all-relative-to*: $(\forall S. (P \text{ relative-to } U) \ S \longrightarrow Q \ S) \longleftrightarrow (\forall S. P \ S \longrightarrow Q(U \cap S))$
 $\langle \text{proof} \rangle$

lemma *relative-toE*: $\llbracket (P \text{ relative-to } U) \ S; \bigwedge S. P \ S \Longrightarrow Q(U \cap S) \rrbracket \Longrightarrow Q \ S$
 $\langle \text{proof} \rangle$

lemma *relative-to-inc*:
 $P \ S \Longrightarrow (P \text{ relative-to } U) \ (U \cap S)$
 $\langle \text{proof} \rangle$

lemma *relative-to-relative-to* [*simp*]:
 $P \text{ relative-to } S \text{ relative-to } T = P \text{ relative-to } (S \cap T)$
 $\langle \text{proof} \rangle$

lemma *relative-to-compl*:
 $S \subseteq U \Longrightarrow ((P \text{ relative-to } U) \ (U - S) \longleftrightarrow ((\lambda c. P(- \ c)) \text{ relative-to } U) \ S)$
 $\langle \text{proof} \rangle$

lemma *relative-to-subset-trans*:
 $\llbracket (P \text{ relative-to } U) \ S; S \subseteq T; T \subseteq U \rrbracket \Longrightarrow (P \text{ relative-to } T) \ S$
 $\langle \text{proof} \rangle$

lemma *relative-to-mono*:
 $\llbracket (P \text{ relative-to } U) \ S; \bigwedge S. P \ S \Longrightarrow Q \ S \rrbracket \Longrightarrow (Q \text{ relative-to } U) \ S$
 $\langle \text{proof} \rangle$

lemma *relative-to-subset-inc*: $\llbracket S \subseteq U; P \ S \rrbracket \Longrightarrow (P \text{ relative-to } U) \ S$
 $\langle \text{proof} \rangle$

lemma *relative-to-Int*:
 $\llbracket (P \text{ relative-to } S) \ C; (P \text{ relative-to } S) \ D; \bigwedge X \ Y. \llbracket P \ X; P \ Y \rrbracket \Longrightarrow P(X \cap Y) \rrbracket$

$$\begin{array}{l} \implies (P \text{ relative-to } S) (C \cap D) \\ \langle \text{proof} \rangle \end{array}$$

lemma *relative-to-Un:*

$$\begin{array}{l} \llbracket (P \text{ relative-to } S) C; (P \text{ relative-to } S) D; \bigwedge X Y. \llbracket P X; P Y \rrbracket \implies P(X \cup Y) \rrbracket \\ \implies (P \text{ relative-to } S) (C \cup D) \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-union-of-relative-to:*

$$\begin{array}{l} ((\text{arbitrary union-of } P) \text{ relative-to } U) = (\text{arbitrary union-of } (P \text{ relative-to } U)) \\ (\text{is } ?lhs = ?rhs) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-union-of-relative-to:*

$$\begin{array}{l} ((\text{finite union-of } P) \text{ relative-to } U) = (\text{finite union-of } (P \text{ relative-to } U)) \text{ (is } ?lhs \\ = ?rhs) \\ \langle \text{proof} \rangle \end{array}$$

lemma *countable-union-of-relative-to:*

$$\begin{array}{l} ((\text{countable union-of } P) \text{ relative-to } U) = (\text{countable union-of } (P \text{ relative-to } U)) \\ (\text{is } ?lhs = ?rhs) \\ \langle \text{proof} \rangle \end{array}$$

lemma *arbitrary-intersection-of-relative-to:*

$$\begin{array}{l} ((\text{arbitrary intersection-of } P) \text{ relative-to } U) = ((\text{arbitrary intersection-of } (P \text{ rel-} \\ \text{ative-to } U)) \text{ relative-to } U) \text{ (is } ?lhs = ?rhs) \\ \langle \text{proof} \rangle \end{array}$$

lemma *finite-intersection-of-relative-to:*

$$\begin{array}{l} ((\text{finite intersection-of } P) \text{ relative-to } U) = ((\text{finite intersection-of } (P \text{ relative-to } \\ U)) \text{ relative-to } U) \text{ (is } ?lhs = ?rhs) \\ \langle \text{proof} \rangle \end{array}$$

lemma *countable-intersection-of-relative-to:*

$$\begin{array}{l} ((\text{countable intersection-of } P) \text{ relative-to } U) = ((\text{countable intersection-of } (P \\ \text{relative-to } U)) \text{ relative-to } U) \text{ (is } ?lhs = ?rhs) \\ \langle \text{proof} \rangle \end{array}$$

$$\begin{array}{l} \text{lemma countable-union-of-empty [simp]: } (\text{countable union-of } P) \{\} \\ \langle \text{proof} \rangle \end{array}$$

$$\begin{array}{l} \text{lemma countable-intersection-of-empty [simp]: } (\text{countable intersection-of } P) \text{ UNIV} \\ \langle \text{proof} \rangle \end{array}$$

$$\begin{array}{l} \text{lemma countable-union-of-inc: } P S \implies (\text{countable union-of } P) S \\ \langle \text{proof} \rangle \end{array}$$

$$\text{lemma countable-intersection-of-inc: } P S \implies (\text{countable intersection-of } P) S$$

$\langle \text{proof} \rangle$

lemma *countable-union-of-complement*:

$(\text{countable union-of } P) S \longleftrightarrow (\text{countable intersection-of } (\lambda S. P(-S))) (-S)$
 (is ?lhs=?rhs)

$\langle \text{proof} \rangle$

lemma *countable-intersection-of-complement*:

$(\text{countable intersection-of } P) S \longleftrightarrow (\text{countable union-of } (\lambda S. P(-S))) (-S)$

$\langle \text{proof} \rangle$

lemma *countable-union-of-explicit*:

assumes $P \{\}$

shows $(\text{countable union-of } P) S \longleftrightarrow$

$(\exists T. (\forall n::\text{nat}. P(T\ n)) \wedge \bigcup (\text{range } T) = S) \text{ (is ?lhs=?rhs)}$

$\langle \text{proof} \rangle$

lemma *countable-union-of-ascending*:

assumes *empty*: $P \{\}$ and $Un: \bigwedge T\ U. \llbracket P\ T; P\ U \rrbracket \implies P(T \cup U)$

shows $(\text{countable union-of } P) S \longleftrightarrow$

$(\exists T. (\forall n. P(T\ n)) \wedge (\forall n. T\ n \subseteq T(\text{Suc } n)) \wedge \bigcup (\text{range } T) = S) \text{ (is$

?lhs=?rhs)

$\langle \text{proof} \rangle$

lemma *countable-union-of-idem [simp]*:

$\text{countable union-of countable union-of } P = \text{countable union-of } P \text{ (is ?lhs=?rhs)}$

$\langle \text{proof} \rangle$

lemma *countable-intersection-of-idem [simp]*:

$\text{countable intersection-of countable intersection-of } P =$
 $\text{countable intersection-of } P$

$\langle \text{proof} \rangle$

lemma *countable-union-of-Union*:

$\llbracket \text{countable } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{countable union-of } P) S \rrbracket$
 $\implies (\text{countable union-of } P) (\bigcup \mathcal{U})$

$\langle \text{proof} \rangle$

lemma *countable-union-of-UN*:

$\llbracket \text{countable } I; \bigwedge i. i \in I \implies (\text{countable union-of } P) (U\ i) \rrbracket$
 $\implies (\text{countable union-of } P) (\bigcup_{i \in I} U\ i)$

$\langle \text{proof} \rangle$

lemma *countable-union-of-Un*:

$\llbracket (\text{countable union-of } P) S; (\text{countable union-of } P) T \rrbracket$
 $\implies (\text{countable union-of } P) (S \cup T)$

$\langle \text{proof} \rangle$

lemma *countable-intersection-of-Inter*:

$\llbracket \text{countable } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{countable intersection-of } P) S \rrbracket$
 $\implies (\text{countable intersection-of } P) (\bigcap \mathcal{U})$
 $\langle \text{proof} \rangle$

lemma *countable-intersection-of-INT:*

$\llbracket \text{countable } I; \bigwedge i. i \in I \implies (\text{countable intersection-of } P) (U i) \rrbracket$
 $\implies (\text{countable intersection-of } P) (\bigcap_{i \in I}. U i)$
 $\langle \text{proof} \rangle$

lemma *countable-intersection-of-inter:*

$\llbracket (\text{countable intersection-of } P) S; (\text{countable intersection-of } P) T \rrbracket$
 $\implies (\text{countable intersection-of } P) (S \cap T)$
 $\langle \text{proof} \rangle$

lemma *countable-union-of-Int:*

assumes S : $(\text{countable union-of } P) S$ **and** T : $(\text{countable union-of } P) T$
and Int : $\bigwedge S T. P S \wedge P T \implies P(S \cap T)$
shows $(\text{countable union-of } P) (S \cap T)$
 $\langle \text{proof} \rangle$

lemma *countable-intersection-of-union:*

assumes S : $(\text{countable intersection-of } P) S$ **and** T : $(\text{countable intersection-of } P) T$
and Un : $\bigwedge S T. P S \wedge P T \implies P(S \cup T)$
shows $(\text{countable intersection-of } P) (S \cup T)$
 $\langle \text{proof} \rangle$

end

96 Signed division: negative results rounded towards zero rather than minus infinity.

theory *Signed-Division*

imports *Main*

begin

class *signed-divide* =

fixes *signed-divide* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixl** $\langle \text{sdiv} \rangle$ 70)

class *signed-modulo* =

fixes *signed-modulo* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$ (**infixl** $\langle \text{smod} \rangle$ 70)

class *signed-division* = *comm-semiring-1-cancel* + *signed-divide* + *signed-modulo* +

assumes *sdiv-mult-smod-eq*: $\langle a \text{ sdiv } b * b + a \text{ smod } b = a \rangle$

begin

lemma *mult-sdiv-smod-eq*:

$\langle b * (a \text{ sdiv } b) + a \text{ smod } b = a \rangle$
 $\langle \text{proof} \rangle$

lemma *smod-sdiv-mult-eq*:
 $\langle a \text{ smod } b + a \text{ sdiv } b * b = a \rangle$
 $\langle \text{proof} \rangle$

lemma *smod-mult-sdiv-eq*:
 $\langle a \text{ smod } b + b * (a \text{ sdiv } b) = a \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-sdiv-mult-eq-smod*:
 $\langle a - a \text{ sdiv } b * b = a \text{ smod } b \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-mult-sdiv-eq-smod*:
 $\langle a - b * (a \text{ sdiv } b) = a \text{ smod } b \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-smod-eq-sdiv-mult*:
 $\langle a - a \text{ smod } b = a \text{ sdiv } b * b \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-smod-eq-mult-sdiv*:
 $\langle a - a \text{ smod } b = b * (a \text{ sdiv } b) \rangle$
 $\langle \text{proof} \rangle$

end

The following specification of division is named “T-division” in [2]. It is motivated by ISO C99, which in turn adopted the typical behavior of hardware modern in the beginning of the 1990ies; but note ISO C99 describes the instance on machine words, not mathematical integers.

instantiation *int :: signed-division*
begin

definition *signed-divide-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ sdiv } l = \text{sgn } k * \text{sgn } l * (|k| \text{ div } |l|) \rangle$ **for** $k \ l :: \text{int}$

definition *signed-modulo-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ smod } l = \text{sgn } k * (|k| \text{ mod } |l|) \rangle$ **for** $k \ l :: \text{int}$

instance $\langle \text{proof} \rangle$

end

lemma *divide-int-eq-signed-divide-int*:
 $\langle k \text{ div } l = k \text{ sdiv } l - \text{of_bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$

$\langle \text{proof} \rangle$

lemma *signed-divide-int-eq-divide-int*:

$\langle k \text{ sdiv } l = k \text{ div } l + \text{of-bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *modulo-int-eq-signed-modulo-int*:

$\langle k \text{ mod } l = k \text{ smod } l + l * \text{of-bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *signed-modulo-int-eq-modulo-int*:

$\langle k \text{ smod } l = k \text{ mod } l - l * \text{of-bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *sdiv-int-div-0*:

$(x :: \text{int}) \text{ sdiv } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *sdiv-int-0-div [simp]*:

$0 \text{ sdiv } (x :: \text{int}) = 0$
 $\langle \text{proof} \rangle$

lemma *smod-int-alt-def*:

$(a :: \text{int}) \text{ smod } b = \text{sgn } (a) * (\text{abs } a \text{ mod } \text{abs } b)$
 $\langle \text{proof} \rangle$

lemma *int-sdiv-simps [simp]*:

$(a :: \text{int}) \text{ sdiv } 1 = a$
 $(a :: \text{int}) \text{ sdiv } 0 = 0$
 $(a :: \text{int}) \text{ sdiv } -1 = -a$
 $\langle \text{proof} \rangle$

lemma *smod-int-mod-0 [simp]*:

$x \text{ smod } (0 :: \text{int}) = x$
 $\langle \text{proof} \rangle$

lemma *smod-int-0-mod [simp]*:

$0 \text{ smod } (x :: \text{int}) = 0$
 $\langle \text{proof} \rangle$

lemma *sgn-sdiv-eq-sgn-mult*:

$a \text{ sdiv } b \neq 0 \implies \text{sgn } ((a :: \text{int}) \text{ sdiv } b) = \text{sgn } (a * b)$
 $\langle \text{proof} \rangle$

lemma *int-sdiv-same-is-1 [simp]*:

assumes $a \neq 0$

shows $((a :: \text{int}) \text{ sdiv } b = a) = (b = 1)$
 $\langle \text{proof} \rangle$

lemma *int-sdiv-negated-is-minus1* [simp]:
 $a \neq 0 \implies ((a :: \text{int}) \text{ sdiv } b = -a) = (b = -1)$
 $\langle \text{proof} \rangle$

lemma *sdiv-int-range*:
 $\langle a \text{ sdiv } b \in \{-|a|..|a|\} \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *smod-int-range*:
 $\langle a \text{ smod } b \in \{-|b| + 1..|b| - 1\} \rangle$
if $\langle b \neq 0 \rangle$ **for** $a \ b :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *smod-int-compares*:
 $\llbracket 0 \leq a; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b < b$
 $\llbracket 0 \leq a; 0 < b \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; 0 < b \rrbracket \implies -b < (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b < -b$
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$
 $\llbracket a \leq 0; b < 0 \rrbracket \implies b \leq (a :: \text{int}) \text{ smod } b$
 $\langle \text{proof} \rangle$

lemma *smod-mod-positive*:
 $\llbracket 0 \leq (a :: \text{int}); 0 \leq b \rrbracket \implies a \text{ smod } b = a \text{ mod } b$
 $\langle \text{proof} \rangle$

lemma *minus-sdiv-eq* [simp]:
 $\langle -k \text{ sdiv } l = -(k \text{ sdiv } l) \rangle$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *sdiv-minus-eq* [simp]:
 $\langle k \text{ sdiv } -l = -(k \text{ sdiv } l) \rangle$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *sdiv-int-numeral-numeral* [simp]:
 $\langle \text{numeral } m \text{ sdiv numeral } n = \text{numeral } m \text{ div } (\text{numeral } n :: \text{int}) \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-smod-eq* [simp]:
 $\langle -k \text{ smod } l = -(k \text{ smod } l) \rangle$ **for** $k \ l :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *smod-minus-eq* [simp]:
 $\langle k \text{ smod } -l = k \text{ smod } l \rangle$ **for** $k \ l :: \text{int}$

$\langle \text{proof} \rangle$

lemma *smod-int-numeral-numeral* [simp]:

$\langle \text{numeral } m \text{ smod numeral } n = \text{numeral } m \bmod (\text{numeral } n :: \text{int}) \rangle$

$\langle \text{proof} \rangle$

end

97 State monad

theory *State-Monad*

imports *Monad-Syntax*

begin

datatype $('s, 'a) \text{ state} = \text{State } (\text{run-state}: 's \Rightarrow ('a \times 's))$

lemma *set-state-iff*: $x \in \text{set-state } m \iff (\exists s \ s'. \text{run-state } m \ s = (x, s'))$

$\langle \text{proof} \rangle$

lemma *pred-stateI*[intro]:

assumes $\bigwedge a \ s \ s'. \text{run-state } m \ s = (a, s') \implies P \ a$

shows *pred-state* $P \ m$

$\langle \text{proof} \rangle$

lemma *pred-stateD*[dest]:

assumes *pred-state* $P \ m$ $\text{run-state } m \ s = (a, s')$

shows $P \ a$

$\langle \text{proof} \rangle$

lemma *pred-state-run-state*: *pred-state* $P \ m \implies P \ (\text{fst } (\text{run-state } m \ s))$

$\langle \text{proof} \rangle$

definition *state-io-rel* :: $('s \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow ('s, 'a) \text{ state} \Rightarrow \text{bool}$ **where**
state-io-rel $P \ m = (\forall s. P \ s \ (\text{snd } (\text{run-state } m \ s)))$

lemma *state-io-relI*[intro]:

assumes $\bigwedge a \ s \ s'. \text{run-state } m \ s = (a, s') \implies P \ s \ s'$

shows *state-io-rel* $P \ m$

$\langle \text{proof} \rangle$

lemma *state-io-relD*[dest]:

assumes *state-io-rel* $P \ m$ $\text{run-state } m \ s = (a, s')$

shows $P \ s \ s'$

$\langle \text{proof} \rangle$

lemma *state-io-rel-mono*[mono]: $P \leq Q \implies \text{state-io-rel } P \leq \text{state-io-rel } Q$

$\langle \text{proof} \rangle$

lemma *state-ext*:

assumes $\bigwedge s. \text{run-state } m \ s = \text{run-state } n \ s$
shows $m = n$
 $\langle \text{proof} \rangle$

context begin

qualified definition $\text{return} :: 'a \Rightarrow ('s, 'a) \text{ state}$ **where**
 $\text{return } a = \text{State } (\text{Pair } a)$

lemma $\text{run-state-return}[\text{simp}]$: $\text{run-state } (\text{return } x) \ s = (x, s)$
 $\langle \text{proof} \rangle$ **definition** $\text{ap} :: ('s, 'a \Rightarrow 'b) \text{ state} \Rightarrow ('s, 'a) \text{ state} \Rightarrow ('s, 'b) \text{ state}$ **where**
 $\text{ap } f \ x = \text{State } (\lambda s. \text{case run-state } f \ s \text{ of } (g, s') \Rightarrow \text{case run-state } x \ s' \text{ of } (y, s'') \Rightarrow (g \ y, s''))$

lemma $\text{run-state-ap}[\text{simp}]$:
 $\text{run-state } (\text{ap } f \ x) \ s = (\text{case run-state } f \ s \text{ of } (g, s') \Rightarrow \text{case run-state } x \ s' \text{ of } (y, s'') \Rightarrow (g \ y, s''))$
 $\langle \text{proof} \rangle$ **definition** $\text{bind} :: ('s, 'a) \text{ state} \Rightarrow ('a \Rightarrow ('s, 'b) \text{ state}) \Rightarrow ('s, 'b) \text{ state}$
where
 $\text{bind } x \ f = \text{State } (\lambda s. \text{case run-state } x \ s \text{ of } (a, s') \Rightarrow \text{run-state } (f \ a) \ s')$

lemma $\text{run-state-bind}[\text{simp}]$:
 $\text{run-state } (\text{bind } x \ f) \ s = (\text{case run-state } x \ s \text{ of } (a, s') \Rightarrow \text{run-state } (f \ a) \ s')$
 $\langle \text{proof} \rangle$

ad hoc-overloading $\text{Monad-Syntax.bind} \equiv \text{bind}$

lemma $\text{bind-left-identity}[\text{simp}]$: $\text{bind } (\text{return } a) \ f = f \ a$
 $\langle \text{proof} \rangle$

lemma $\text{bind-right-identity}[\text{simp}]$: $\text{bind } m \ \text{return} = m$
 $\langle \text{proof} \rangle$

lemma $\text{bind-assoc}[\text{simp}]$: $\text{bind } (\text{bind } m \ f) \ g = \text{bind } m \ (\lambda x. \text{bind } (f \ x) \ g)$
 $\langle \text{proof} \rangle$

lemma $\text{bind-predI}[\text{intro}]$:
assumes $\text{pred-state } (\lambda x. \text{pred-state } P \ (f \ x)) \ m$
shows $\text{pred-state } P \ (\text{bind } m \ f)$
 $\langle \text{proof} \rangle$ **definition** $\text{get} :: ('s, 's) \text{ state}$ **where**
 $\text{get} = \text{State } (\lambda s. (s, s))$

lemma $\text{run-state-get}[\text{simp}]$: $\text{run-state } \text{get} \ s = (s, s)$
 $\langle \text{proof} \rangle$ **definition** $\text{set} :: 's \Rightarrow ('s, \text{unit}) \text{ state}$ **where**
 $\text{set } s' = \text{State } (\lambda _. ((), s'))$

lemma $\text{run-state-set}[\text{simp}]$: $\text{run-state } (\text{set } s') \ s = ((), s')$
 $\langle \text{proof} \rangle$

lemma *get-set[simp]*: $\text{bind } \text{get } \text{set} = \text{return } ()$
 $\langle \text{proof} \rangle$

lemma *set-set[simp]*: $\text{bind } (\text{set } s) (\lambda -. \text{set } s') = \text{set } s'$
 $\langle \text{proof} \rangle$

lemma *get-bind-set[simp]*: $\text{bind } \text{get } (\lambda s. \text{bind } (\text{set } s) (f s)) = \text{bind } \text{get } (\lambda s. f s ())$
 $\langle \text{proof} \rangle$

lemma *get-const[simp]*: $\text{bind } \text{get } (\lambda -. m) = m$
 $\langle \text{proof} \rangle$

fun *traverse-list* :: $('a \Rightarrow ('b, 'c) \text{ state}) \Rightarrow 'a \text{ list} \Rightarrow ('b, 'c \text{ list}) \text{ state}$ **where**
traverse-list - [] = *return* [] |
traverse-list *f* (*x* # *xs*) = *do* {
 x ← *f* *x*;
 xs ← *traverse-list* *f* *xs*;
 return (*x* # *xs*)
}

lemma *traverse-list-app[simp]*: $\text{traverse-list } f (xs @ ys) = \text{do } \{$
 xs ← *traverse-list* *f* *xs*;
 ys ← *traverse-list* *f* *ys*;
 return (*xs* @ *ys*)
 $\}$
 $\langle \text{proof} \rangle$

lemma *traverse-comp[simp]*: $\text{traverse-list } (g \circ f) xs = \text{traverse-list } g (\text{map } f xs)$
 $\langle \text{proof} \rangle$

abbreviation *mono-state* :: $('s::\text{preorder}, 'a) \text{ state} \Rightarrow \text{bool}$ **where**
mono-state $\equiv \text{state-io-rel } (\leq)$

abbreviation *strict-mono-state* :: $('s::\text{preorder}, 'a) \text{ state} \Rightarrow \text{bool}$ **where**
strict-mono-state $\equiv \text{state-io-rel } (<)$

corollary *strict-mono-implies-mono*: $\text{strict-mono-state } m \implies \text{mono-state } m$
 $\langle \text{proof} \rangle$

lemma *return-mono[simp, intro]*: $\text{mono-state } (\text{return } x)$
 $\langle \text{proof} \rangle$

lemma *get-mono[simp, intro]*: $\text{mono-state } \text{get}$
 $\langle \text{proof} \rangle$

lemma *put-mono*:
 assumes $\bigwedge x. s' \geq x$
 shows $\text{mono-state } (\text{set } s')$
 $\langle \text{proof} \rangle$

lemma *map-mono[intro]*: *mono-state* $m \implies \text{mono-state } (\text{map-state } f \ m)$
 $\langle \text{proof} \rangle$

lemma *map-strict-mono[intro]*: *strict-mono-state* $m \implies \text{strict-mono-state } (\text{map-state } f \ m)$
 $\langle \text{proof} \rangle$

lemma *bind-mono-strong*:
 assumes *mono-state* m
 assumes $\bigwedge x \ s \ s'. \text{run-state } m \ s = (x, \ s') \implies \text{mono-state } (f \ x)$
 shows *mono-state* $(\text{bind } m \ f)$
 $\langle \text{proof} \rangle$

lemma *bind-strict-mono-strong1*:
 assumes *mono-state* m
 assumes $\bigwedge x \ s \ s'. \text{run-state } m \ s = (x, \ s') \implies \text{strict-mono-state } (f \ x)$
 shows *strict-mono-state* $(\text{bind } m \ f)$
 $\langle \text{proof} \rangle$

lemma *bind-strict-mono-strong2*:
 assumes *strict-mono-state* m
 assumes $\bigwedge x \ s \ s'. \text{run-state } m \ s = (x, \ s') \implies \text{mono-state } (f \ x)$
 shows *strict-mono-state* $(\text{bind } m \ f)$
 $\langle \text{proof} \rangle$

corollary *bind-strict-mono-strong*:
 assumes *strict-mono-state* m
 assumes $\bigwedge x \ s \ s'. \text{run-state } m \ s = (x, \ s') \implies \text{strict-mono-state } (f \ x)$
 shows *strict-mono-state* $(\text{bind } m \ f)$
 $\langle \text{proof} \rangle$ **definition** *update* :: $('s \Rightarrow 's) \Rightarrow ('s, \text{unit}) \text{ state}$ **where**
 $\text{update } f = \text{bind } \text{get } (\text{set } \circ f)$

lemma *update-id[simp]*: *update* $(\lambda x. \ x) = \text{return } ()$
 $\langle \text{proof} \rangle$

lemma *update-comp[simp]*: *bind* $(\text{update } f) \ (\lambda -. \ \text{update } g) = \text{update } (g \circ f)$
 $\langle \text{proof} \rangle$

lemma *set-update[simp]*: *bind* $(\text{set } s) \ (\lambda -. \ \text{update } f) = \text{set } (f \ s)$
 $\langle \text{proof} \rangle$

lemma *set-bind-update[simp]*: *bind* $(\text{set } s) \ (\lambda -. \ \text{bind } (\text{update } f) \ g) = \text{bind } (\text{set } (f \ s)) \ g$
 $\langle \text{proof} \rangle$

lemma *update-mono*:
 assumes $\bigwedge x. \ x \leq f \ x$
 shows *mono-state* $(\text{update } f)$

$\langle proof \rangle$

lemma *update-strict-mono*:

assumes $\bigwedge x. x < f\ x$

shows *strict-mono-state* (*update f*)

$\langle proof \rangle$

end

end

theory *Comparator*

imports *Main*

begin

98 Comparators on linear quasi-orders

98.1 Basic properties

datatype *comp* = *Less* | *Equiv* | *Greater*

locale *comparator* =

fixes *cmp* :: $\langle 'a \Rightarrow 'a \Rightarrow comp \rangle$

assumes *refl* [*simp*]: $\langle \bigwedge a. cmp\ a\ a = Equiv \rangle$

and *trans-equiv*: $\langle \bigwedge a\ b\ c. cmp\ a\ b = Equiv \implies cmp\ b\ c = Equiv \implies cmp\ a\ c = Equiv \rangle$

assumes *trans-less*: $\langle cmp\ a\ b = Less \implies cmp\ b\ c = Less \implies cmp\ a\ c = Less \rangle$

and *greater-iff-sym-less*: $\langle \bigwedge b\ a. cmp\ b\ a = Greater \longleftrightarrow cmp\ a\ b = Less \rangle$

begin

Dual properties

lemma *trans-greater*:

$\langle cmp\ a\ c = Greater \rangle$ **if** $\langle cmp\ a\ b = Greater \rangle$ $\langle cmp\ b\ c = Greater \rangle$

$\langle proof \rangle$

lemma *less-iff-sym-greater*:

$\langle cmp\ b\ a = Less \longleftrightarrow cmp\ a\ b = Greater \rangle$

$\langle proof \rangle$

The equivalence part

lemma *sym*:

$\langle cmp\ b\ a = Equiv \longleftrightarrow cmp\ a\ b = Equiv \rangle$

$\langle proof \rangle$

lemma *reflp*:

$\langle reflp\ (\lambda a\ b. cmp\ a\ b = Equiv) \rangle$

$\langle proof \rangle$

lemma *symp*:
 $\langle \text{symp } (\lambda a b. \text{cmp } a b = \text{Equiv}) \rangle$
 $\langle \text{proof} \rangle$

lemma *transp*:
 $\langle \text{transp } (\lambda a b. \text{cmp } a b = \text{Equiv}) \rangle$
 $\langle \text{proof} \rangle$

lemma *equivp*:
 $\langle \text{equivp } (\lambda a b. \text{cmp } a b = \text{Equiv}) \rangle$
 $\langle \text{proof} \rangle$

The strict part

lemma *irreflp-less*:
 $\langle \text{irreflp } (\lambda a b. \text{cmp } a b = \text{Less}) \rangle$
 $\langle \text{proof} \rangle$

lemma *irreflp-greater*:
 $\langle \text{irreflp } (\lambda a b. \text{cmp } a b = \text{Greater}) \rangle$
 $\langle \text{proof} \rangle$

lemma *asym-less*:
 $\langle \text{cmp } b a \neq \text{Less} \rangle \text{ if } \langle \text{cmp } a b = \text{Less} \rangle$
 $\langle \text{proof} \rangle$

lemma *asym-greater*:
 $\langle \text{cmp } b a \neq \text{Greater} \rangle \text{ if } \langle \text{cmp } a b = \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *asym-less*:
 $\langle \text{asym } (\lambda a b. \text{cmp } a b = \text{Less}) \rangle$
 $\langle \text{proof} \rangle$

lemma *asym-greater*:
 $\langle \text{asym } (\lambda a b. \text{cmp } a b = \text{Greater}) \rangle$
 $\langle \text{proof} \rangle$

lemma *trans-equiv-less*:
 $\langle \text{cmp } a c = \text{Less} \rangle \text{ if } \langle \text{cmp } a b = \text{Equiv} \rangle \text{ and } \langle \text{cmp } b c = \text{Less} \rangle$
 $\langle \text{proof} \rangle$

lemma *trans-less-equiv*:
 $\langle \text{cmp } a c = \text{Less} \rangle \text{ if } \langle \text{cmp } a b = \text{Less} \rangle \text{ and } \langle \text{cmp } b c = \text{Equiv} \rangle$
 $\langle \text{proof} \rangle$

lemma *trans-equiv-greater*:
 $\langle \text{cmp } a c = \text{Greater} \rangle \text{ if } \langle \text{cmp } a b = \text{Equiv} \rangle \text{ and } \langle \text{cmp } b c = \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *trans-greater-equiv*:

$\langle \text{cmp } a \ c = \text{Greater} \rangle$ **if** $\langle \text{cmp } a \ b = \text{Greater} \rangle$ **and** $\langle \text{cmp } b \ c = \text{Equiv} \rangle$
 $\langle \text{proof} \rangle$

lemma *transp-less*:

$\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b = \text{Less}) \rangle$
 $\langle \text{proof} \rangle$

lemma *transp-greater*:

$\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b = \text{Greater}) \rangle$
 $\langle \text{proof} \rangle$

The reflexive part

lemma *reflp-not-less*:

$\langle \text{reflp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Less}) \rangle$
 $\langle \text{proof} \rangle$

lemma *reflp-not-greater*:

$\langle \text{reflp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Greater}) \rangle$
 $\langle \text{proof} \rangle$

lemma *quasisym-not-less*:

$\langle \text{cmp } a \ b = \text{Equiv} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Less} \rangle$ **and** $\langle \text{cmp } b \ a \neq \text{Less} \rangle$
 $\langle \text{proof} \rangle$

lemma *quasisym-not-greater*:

$\langle \text{cmp } a \ b = \text{Equiv} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Greater} \rangle$ **and** $\langle \text{cmp } b \ a \neq \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *trans-not-less*:

$\langle \text{cmp } a \ c \neq \text{Less} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Less} \rangle$ $\langle \text{cmp } b \ c \neq \text{Less} \rangle$
 $\langle \text{proof} \rangle$

lemma *trans-not-greater*:

$\langle \text{cmp } a \ c \neq \text{Greater} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Greater} \rangle$ $\langle \text{cmp } b \ c \neq \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *transp-not-less*:

$\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Less}) \rangle$
 $\langle \text{proof} \rangle$

lemma *transp-not-greater*:

$\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Greater}) \rangle$
 $\langle \text{proof} \rangle$

Substitution under equivalences

lemma *equiv-subst-left*:

$\langle \text{cmp } z \ y = \text{comp} \longleftrightarrow \text{cmp } x \ y = \text{comp} \rangle$ **if** $\langle \text{cmp } z \ x = \text{Equiv} \rangle$ **for** *comp*
 $\langle \text{proof} \rangle$

lemma *equiv-subst-right*:

⟨*cmp* *x z* = *comp* \longleftrightarrow *cmp* *x y* = *comp*⟩ **if** ⟨*cmp* *z y* = *Equiv*⟩ **for** *comp*
 ⟨*proof*⟩

end

typedef *'a comparator* = ⟨{*cmp* :: *'a* \Rightarrow *'a* \Rightarrow *comp. comparator cmp*}⟩
morphisms *compare Abs-comparator*
 ⟨*proof*⟩

setup-lifting *type-definition-comparator*

global-interpretation *compare: comparator* ⟨*compare cmp*⟩
 ⟨*proof*⟩

lift-definition *flat* :: ⟨*'a comparator*⟩
is ⟨ λ - *-*. *Equiv*⟩ ⟨*proof*⟩

instantiation *comparator* :: (*linorder*) *default*
begin

lift-definition *default-comparator* :: ⟨*'a comparator*⟩
is ⟨ λ *x y. if* *x* < *y* *then Less* *else if* *x* > *y* *then Greater* *else Equiv*⟩
 ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *compare-default-eq-Less-iff* [*simp*]:
 ⟨*compare default x y* = *Less* \longleftrightarrow *x* < *y*⟩
 ⟨*proof*⟩

lemma *compare-default-eq-Equiv-iff* [*simp*]:
 ⟨*compare default x y* = *Equiv* \longleftrightarrow *x* = *y*⟩
 ⟨*proof*⟩

lemma *compare-default-eq-Greater-iff* [*simp*]:
 ⟨*compare default x y* = *Greater* \longleftrightarrow *x* > *y*⟩
 ⟨*proof*⟩

A rudimentary quickcheck setup

instantiation *comparator* :: (*enum*) *equal*
begin

lift-definition *equal-comparator* :: ⟨*'a comparator* \Rightarrow *'a comparator* \Rightarrow *bool*⟩
is ⟨ λ *f g. \forall *x* \in set Enum.enum. *f x* = *g x**⟩ ⟨*proof*⟩

instance
 ⟨*proof*⟩

end

lemma [*code nbe*]:
 ⟨*HOL.equal* (*cmp* :: 'a::enum comparator) *cmp* \longleftrightarrow *True*⟩
 ⟨*proof*⟩

lemma [*code*]:
 ⟨*HOL.equal* *cmp1 cmp2* \longleftrightarrow *Enum.enum-all* ($\lambda x.$ *compare cmp1 x = compare cmp2 x*)⟩
 ⟨*proof*⟩

instantiation *comparator* :: ($\{\textit{linorder}, \textit{typerep}\}$) *full-exhaustive*
begin

definition *full-exhaustive-comparator* ::
 ⟨('a comparator \times (unit \Rightarrow term) \Rightarrow (bool \times term list) option)
 \Rightarrow *natural* \Rightarrow (bool \times term list) option)⟩
where ⟨*full-exhaustive-comparator* *f s* =
 Quickcheck-Exhaustive.orelse
 (*f* (*flat*, ($\lambda u.$ *Code-Evaluation.Const* (*STR "Comparator.flat"*) *TYPEREPE*('a
 comparator))))
 (*f* (*default*, ($\lambda u.$ *Code-Evaluation.Const* (*STR "HOL.default-class.default"*)
 TYPEREPE('a *comparator*))))⟩

instance ⟨*proof*⟩

end

98.2 Fundamental comparator combinators

lift-definition *reversed* :: ⟨'a comparator \Rightarrow 'a comparator⟩
is ⟨ $\lambda \textit{cmp} a b.$ *cmp b a*⟩
 ⟨*proof*⟩

lemma *compare-reversed-apply* [*simp*]:
 ⟨*compare* (*reversed cmp*) *x y* = *compare cmp y x*⟩
 ⟨*proof*⟩

lift-definition *key* :: ⟨('b \Rightarrow 'a) \Rightarrow 'a comparator \Rightarrow 'b comparator⟩
is ⟨ $\lambda f \textit{cmp} a b.$ *cmp* (*f a*) (*f b*)⟩
 ⟨*proof*⟩

lemma *compare-key-apply* [*simp*]:
 ⟨*compare* (*key f cmp*) *x y* = *compare cmp* (*f x*) (*f y*)⟩
 ⟨*proof*⟩

lift-definition *prod-lex* :: $\langle 'a \text{ comparator} \Rightarrow 'b \text{ comparator} \Rightarrow ('a \times 'b) \text{ comparator} \rangle$
is $\langle \lambda f g (a, c) (b, d). \text{case } f a b \text{ of } \text{Less} \Rightarrow \text{Less} \mid \text{Equiv} \Rightarrow g c d \mid \text{Greater} \Rightarrow \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *compare-prod-lex-apply*:
 $\langle \text{compare } (\text{prod-lex } \text{cmp1 } \text{cmp2}) p q =$
 $(\text{case } \text{compare } (\text{key fst } \text{cmp1}) p q \text{ of } \text{Less} \Rightarrow \text{Less} \mid \text{Equiv} \Rightarrow \text{compare } (\text{key snd } \text{cmp2}) p q \mid \text{Greater} \Rightarrow \text{Greater}) \rangle$
 $\langle \text{proof} \rangle$

98.3 Direct implementations for linear orders on selected types

definition *comparator-bool* :: $\langle \text{bool comparator} \rangle$
where [*simp*, *code-abbrev*]: $\langle \text{comparator-bool} = \text{default} \rangle$

lemma *compare-comparator-bool* [*code abstract*]:
 $\langle \text{compare } \text{comparator-bool} = (\lambda p q.$
 $\text{if } p \text{ then if } q \text{ then } \text{Equiv} \text{ else } \text{Greater}$
 $\text{else if } q \text{ then } \text{Less} \text{ else } \text{Equiv}) \rangle$
 $\langle \text{proof} \rangle$

definition *raw-comparator-nat* :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{comp} \rangle$
where [*simp*]: $\langle \text{raw-comparator-nat} = \text{compare default} \rangle$

lemma *default-comparator-nat* [*simp*, *code*]:
 $\langle \text{raw-comparator-nat } (0::\text{nat}) 0 = \text{Equiv} \rangle$
 $\langle \text{raw-comparator-nat } (\text{Suc } m) 0 = \text{Greater} \rangle$
 $\langle \text{raw-comparator-nat } 0 (\text{Suc } n) = \text{Less} \rangle$
 $\langle \text{raw-comparator-nat } (\text{Suc } m) (\text{Suc } n) = \text{raw-comparator-nat } m n \rangle$
 $\langle \text{proof} \rangle$

definition *comparator-nat* :: $\langle \text{nat comparator} \rangle$
where [*simp*, *code-abbrev*]: $\langle \text{comparator-nat} = \text{default} \rangle$

lemma *compare-comparator-nat* [*code abstract*]:
 $\langle \text{compare } \text{comparator-nat} = \text{raw-comparator-nat} \rangle$
 $\langle \text{proof} \rangle$

definition *comparator-linordered-group* :: $\langle 'a::\text{linordered-ab-group-add comparator} \rangle$
where [*simp*, *code-abbrev*]: $\langle \text{comparator-linordered-group} = \text{default} \rangle$

lemma *comparator-linordered-group* [*code abstract*]:
 $\langle \text{compare } \text{comparator-linordered-group} = (\lambda a b.$
 $\text{let } c = a - b \text{ in if } c < 0 \text{ then } \text{Less}$
 $\text{else if } c = 0 \text{ then } \text{Equiv} \text{ else } \text{Greater}) \rangle$
 $\langle \text{proof} \rangle$

end

theory *Sorting-Algorithms*
imports *Main Multiset Comparator*
begin

99 Stably sorted lists

abbreviation (*input*) *stable-segment* :: $\langle 'a \text{ comparator} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$
where $\langle \text{stable-segment cmp } x \equiv \text{filter } (\lambda y. \text{compare cmp } x \ y = \text{Equiv}) \rangle$

fun *sorted* :: $\langle 'a \text{ comparator} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$
where *sorted-Nil*: $\langle \text{sorted cmp } [] \longleftrightarrow \text{True} \rangle$
| *sorted-single*: $\langle \text{sorted cmp } [x] \longleftrightarrow \text{True} \rangle$
| *sorted-rec*: $\langle \text{sorted cmp } (y \# x \# xs) \longleftrightarrow \text{compare cmp } y \ x \neq \text{Greater} \wedge \text{sorted cmp } (x \# xs) \rangle$

lemma *sorted-ConsI*:
 $\langle \text{sorted cmp } (x \# xs) \rangle$ **if** $\langle \text{sorted cmp } xs \rangle$
and $\langle \bigwedge y \text{ ys. } xs = y \# ys \implies \text{compare cmp } x \ y \neq \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-Cons-imp-sorted*:
 $\langle \text{sorted cmp } xs \rangle$ **if** $\langle \text{sorted cmp } (x \# xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-Cons-imp-not-less*:
 $\langle \text{compare cmp } y \ x \neq \text{Greater} \rangle$ **if** $\langle \text{sorted cmp } (y \# xs) \rangle$
and $\langle x \in \text{set } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-induct* [*consumes 1, case-names Nil Cons, induct pred: sorted*]:
 $\langle P \ xs \rangle$ **if** $\langle \text{sorted cmp } xs \rangle$ **and** $\langle P \ [] \rangle$
and *: $\langle \bigwedge x \ xs. \text{sorted cmp } xs \implies P \ xs \rangle$
 $\implies \langle \bigwedge y. y \in \text{set } xs \implies \text{compare cmp } x \ y \neq \text{Greater} \implies P \ (x \# xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-induct-remove1* [*consumes 1, case-names Nil minimum*]:
 $\langle P \ xs \rangle$ **if** $\langle \text{sorted cmp } xs \rangle$ **and** $\langle P \ [] \rangle$
and *: $\langle \bigwedge x \ xs. \text{sorted cmp } xs \implies P \ (\text{remove1 } x \ xs) \rangle$
 $\implies x \in \text{set } xs \implies \text{hd } (\text{stable-segment cmp } x \ xs) = x \implies \langle \bigwedge y. y \in \text{set } xs \implies \text{compare cmp } x \ y \neq \text{Greater} \rangle$
 $\implies P \ xs \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-remove1*:
 $\langle \text{sorted cmp } (\text{remove1 } x \ xs) \rangle$ **if** $\langle \text{sorted cmp } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-stable-segment*:
 $\langle \text{sorted cmp } (\text{stable-segment cmp } x \text{ } xs) \rangle$
 $\langle \text{proof} \rangle$

primrec *insort* :: $\langle 'a \text{ comparator} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$
where $\langle \text{insort cmp } y \ [] = [y] \rangle$
 $\mid \langle \text{insort cmp } y (x \# xs) = (\text{if compare cmp } y \ x \neq \text{Greater}$
 $\text{then } y \# x \# xs$
 $\text{else } x \# \text{insort cmp } y \ xs) \rangle$

lemma *mset-insort [simp]*:
 $\langle \text{mset } (\text{insort cmp } x \text{ } xs) = \text{add-mset } x \ (\text{mset } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *length-insort [simp]*:
 $\langle \text{length } (\text{insort cmp } x \text{ } xs) = \text{Suc } (\text{length } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-insort*:
 $\langle \text{sorted cmp } (\text{insort cmp } x \text{ } xs) \rangle \text{ if } \langle \text{sorted cmp } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *stable-insort-equiv*:
 $\langle \text{stable-segment cmp } y (\text{insort cmp } x \text{ } xs) = x \# \text{stable-segment cmp } y \ xs \rangle$
if $\langle \text{compare cmp } y \ x = \text{Equiv} \rangle$
 $\langle \text{proof} \rangle$

lemma *stable-insort-not-equiv*:
 $\langle \text{stable-segment cmp } y (\text{insort cmp } x \text{ } xs) = \text{stable-segment cmp } y \ xs \rangle$
if $\langle \text{compare cmp } y \ x \neq \text{Equiv} \rangle$
 $\langle \text{proof} \rangle$

lemma *remove1-insort-same-eq [simp]*:
 $\langle \text{remove1 } x (\text{insort cmp } x \text{ } xs) = xs \rangle$
 $\langle \text{proof} \rangle$

lemma *insort-eq-ConsI*:
 $\langle \text{insort cmp } x \text{ } xs = x \# xs \rangle$
if $\langle \text{sorted cmp } xs \rangle \langle \bigwedge y. y \in \text{set } xs \implies \text{compare cmp } x \ y \neq \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *remove1-insort-not-same-eq [simp]*:
 $\langle \text{remove1 } y (\text{insort cmp } x \text{ } xs) = \text{insort cmp } x (\text{remove1 } y \ xs) \rangle$
if $\langle \text{sorted cmp } xs \rangle \langle x \neq y \rangle$
 $\langle \text{proof} \rangle$

lemma *insort-remove1-same-eq*:
 $\langle \text{insort cmp } x (\text{remove1 } x \text{ } xs) = xs \rangle$

if $\langle \text{sorted cmp } xs \rangle$ **and** $\langle x \in \text{set } xs \rangle$ **and** $\langle \text{hd } (\text{stable-segment cmp } x xs) = x \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-append-iff*:

$\langle \text{sorted cmp } (xs @ ys) \longleftrightarrow \text{sorted cmp } xs \wedge \text{sorted cmp } ys$
 $\wedge (\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{compare cmp } x y \neq \text{Greater}) \rangle$ **(is** $\langle ?P \longleftrightarrow ?R \wedge$
 $?S \wedge ?Q \rangle$
 $\langle \text{proof} \rangle$

definition *sort* :: $\langle 'a \text{ comparator} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$
where $\langle \text{sort cmp } xs = \text{foldr } (\text{insort cmp}) xs [] \rangle$

lemma *sort-simps* [*simp*]:

$\langle \text{sort cmp } [] = [] \rangle$
 $\langle \text{sort cmp } (x \# xs) = \text{insort cmp } x (\text{sort cmp } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *mset-sort* [*simp*]:

$\langle \text{mset } (\text{sort cmp } xs) = \text{mset } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *length-sort* [*simp*]:

$\langle \text{length } (\text{sort cmp } xs) = \text{length } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-sort* [*simp*]:

$\langle \text{sorted cmp } (\text{sort cmp } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *stable-sort*:

$\langle \text{stable-segment cmp } x (\text{sort cmp } xs) = \text{stable-segment cmp } x xs \rangle$
 $\langle \text{proof} \rangle$

lemma *sort-remove1-eq* [*simp*]:

$\langle \text{sort cmp } (\text{remove1 } x xs) = \text{remove1 } x (\text{sort cmp } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *set-insort* [*simp*]:

$\langle \text{set } (\text{insort cmp } x xs) = \text{insert } x (\text{set } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *set-sort* [*simp*]:

$\langle \text{set } (\text{sort cmp } xs) = \text{set } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *sort-eqI*:

$\langle \text{sort cmp } ys = xs \rangle$
if *permutation*: $\langle \text{mset } ys = \text{mset } xs \rangle$
and *sorted*: $\langle \text{sorted cmp } xs \rangle$

and *stable*: $\langle \bigwedge y. y \in \text{set } ys \implies$
 $\text{stable-segment cmp } y \text{ } ys = \text{stable-segment cmp } y \text{ } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *filter-insort*:
 $\langle \text{filter } P (\text{insort cmp } x \text{ } xs) = \text{insort cmp } x (\text{filter } P \text{ } xs) \rangle$
if $\langle \text{sorted cmp } xs \rangle$ **and** $\langle P \text{ } x \rangle$
 $\langle \text{proof} \rangle$

lemma *filter-insort-triv*:
 $\langle \text{filter } P (\text{insort cmp } x \text{ } xs) = \text{filter } P \text{ } xs \rangle$
if $\langle \neg P \text{ } x \rangle$
 $\langle \text{proof} \rangle$

lemma *filter-sort*:
 $\langle \text{filter } P (\text{sort cmp } xs) = \text{sort cmp } (\text{filter } P \text{ } xs) \rangle$
 $\langle \text{proof} \rangle$

100 Alternative sorting algorithms

100.1 Quicksort

definition *quicksort* :: $\langle 'a \text{ comparator} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$
where *quicksort-is-sort* [*simp*]: $\langle \text{quicksort} = \text{sort} \rangle$

lemma *sort-by-quicksort*:
 $\langle \text{sort} = \text{quicksort} \rangle$
 $\langle \text{proof} \rangle$

lemma *sort-by-quicksort-rec*:
 $\langle \text{sort cmp } xs = \text{sort cmp } [x \leftarrow xs. \text{compare cmp } x (xs ! (\text{length } xs \text{ div } 2)) = \text{Less}]$
 $\text{@ stable-segment cmp } (xs ! (\text{length } xs \text{ div } 2)) \text{ } xs$
 $\text{@ sort cmp } [x \leftarrow xs. \text{compare cmp } x (xs ! (\text{length } xs \text{ div } 2)) = \text{Greater}] \rangle$ (**is** $\langle - =$
 $?rhs \rangle$)
 $\langle \text{proof} \rangle$

context
begin

qualified definition *partition* :: $\langle 'a \text{ comparator} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \times 'a \text{ list}$
 $\times 'a \text{ list} \rangle$
where $\langle \text{partition cmp pivot } xs =$
 $([x \leftarrow xs. \text{compare cmp } x \text{ pivot} = \text{Less}], \text{stable-segment cmp pivot } xs, [x \leftarrow xs.$
 $\text{compare cmp } x \text{ pivot} = \text{Greater}]) \rangle$

qualified lemma *partition-code* [*code*]:
 $\langle \text{partition cmp pivot } [] = ([], [], []) \rangle$
 $\langle \text{partition cmp pivot } (x \# xs) =$
 $(\text{let } (lts, eqs, gts) = \text{partition cmp pivot } xs$

```

    in case compare cmp x pivot of
      Less  $\Rightarrow$  ( $x \# lts$ , eqs, gts)
    | Equiv  $\Rightarrow$  (lts,  $x \# eqs$ , gts)
    | Greater  $\Rightarrow$  (lts, eqs,  $x \# gts$ )
  <proof>

```

lemma quicksort-code [code]:

```

  <quicksort cmp xs =
    (case xs of
      []  $\Rightarrow$  []
    | [x]  $\Rightarrow$  xs
    | [x, y]  $\Rightarrow$  (if compare cmp x y  $\neq$  Greater then xs else [y, x])
    | -  $\Rightarrow$ 
      let (lts, eqs, gts) = partition cmp (xs ! (length xs div 2)) xs
      in quicksort cmp lts @ eqs @ quicksort cmp gts)
  <proof>

```

end

100.2 Mergesort

definition mergesort :: $\langle 'a \text{ comparator} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$

where mergesort-is-sort [simp]: $\langle \text{mergesort} = \text{sort} \rangle$

lemma sort-by-mergesort:

```

  <sort = mergesort>
  <proof>

```

context

fixes cmp :: $\langle 'a \text{ comparator} \rangle$

begin

qualified function merge :: $\langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$

```

  where <merge [] ys = ys>
  | <merge xs [] = xs>
  | <merge (x # xs) (y # ys) = (if compare cmp x y = Greater
    then y # merge (x # xs) ys else x # merge xs (y # ys))>
  <proof> termination <proof>

```

lemma mset-merge:

```

  <mset (merge xs ys) = mset xs + mset ys>
  <proof>

```

lemma merge-eq-Cons-imp:

```

  <xs  $\neq$  []  $\wedge$  z = hd xs  $\vee$  ys  $\neq$  []  $\wedge$  z = hd ys>
  if <merge xs ys = z # zs>
  <proof>

```

lemma filter-merge:

$\langle \text{filter } P \text{ (merge } xs \text{ } ys) = \text{merge (filter } P \text{ } xs) \text{ (filter } P \text{ } ys) \rangle$
 $\text{if } \langle \text{sorted cmp } xs \rangle \text{ and } \langle \text{sorted cmp } ys \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-merge*:

$\langle \text{sorted cmp (merge } xs \text{ } ys) \rangle \text{ if } \langle \text{sorted cmp } xs \rangle \text{ and } \langle \text{sorted cmp } ys \rangle$
 $\langle \text{proof} \rangle$

lemma *merge-eq-appendI*:

$\langle \text{merge } xs \text{ } ys = xs @ ys \rangle$
 $\text{if } \langle \bigwedge x y. x \in \text{set } xs \implies y \in \text{set } ys \implies \text{compare cmp } x \text{ } y \neq \text{Greater} \rangle$
 $\langle \text{proof} \rangle$

lemma *merge-stable-segments*:

$\langle \text{merge (stable-segment cmp } l \text{ } xs) \text{ (stable-segment cmp } l \text{ } ys) =$
 $\text{stable-segment cmp } l \text{ } xs @ \text{stable-segment cmp } l \text{ } ys \rangle$
 $\langle \text{proof} \rangle$

lemma *sort-by-mergesort-rec*:

$\langle \text{sort cmp } xs =$
 $\text{merge (sort cmp (take (length } xs \text{ div } 2) \text{ } xs))$
 $\text{(sort cmp (drop (length } xs \text{ div } 2) \text{ } xs))} \rangle \text{ (is } \langle - = ?rhs \rangle)$
 $\langle \text{proof} \rangle$

lemma *mergesort-code* [code]:

$\langle \text{mergesort cmp } xs =$
 $(\text{case } xs \text{ of}$
 $\quad [] \Rightarrow []$
 $\quad | [x] \Rightarrow xs$
 $\quad | [x, y] \Rightarrow (\text{if compare cmp } x \text{ } y \neq \text{Greater then } xs \text{ else } [y, x])$
 $\quad | - \Rightarrow$
 $\quad \text{let}$
 $\quad \quad \text{half} = \text{length } xs \text{ div } 2;$
 $\quad \quad \text{ys} = \text{take half } xs;$
 $\quad \quad \text{zs} = \text{drop half } xs$
 $\quad \text{in merge (mergesort cmp } ys) \text{ (mergesort cmp } zs)) \rangle$
 $\langle \text{proof} \rangle$

end

100.3 Lexicographic products

lemma *sorted-prod-lex-imp-sorted-fst*:

$\langle \text{sorted (key fst cmp1) } ps \rangle \text{ if } \langle \text{sorted (prod-lex cmp1 cmp2) } ps \rangle$
 $\langle \text{proof} \rangle$

lemma *sorted-prod-lex-imp-sorted-snd*:

$\langle \text{sorted (key snd cmp2) } ps \rangle \text{ if } \langle \text{sorted (prod-lex cmp1 cmp2) } ps \rangle \langle \bigwedge a' b'. (a', b') \in \text{set } ps \implies \text{compare cmp1 } a \text{ } a' = \text{Equiv} \rangle$

⟨proof⟩

lemma *sort-comp-fst-snd-eq-sort-prod-lex*:

⟨*sort* (*key fst cmp1*) \circ *sort* (*key snd cmp2*) = *sort* (*prod-lex cmp1 cmp2*)⟩ (is

⟨*sort* ?*cmp1* \circ *sort* ?*cmp2* = *sort* ?*cmp*⟩)

⟨proof⟩

end

101 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

theory *Sum-of-Squares*

imports *Complex-Main*

begin

⟨ML⟩

end

theory *Time-Commands*

imports *Main*

keywords *time-fun* :: *thy-decl*

and *time-function* :: *thy-decl*

and *time-definition* :: *thy-decl*

and *time-partial-function* :: *thy-decl*

and *equations*

and *time-fun-0* :: *thy-decl*

begin

⟨ML⟩

declare [[*time-prefix* = *T*]]

This theory provides commands for the automatic definition of step-counting running-time functions from HOL functions following the translation described in Section 1.5, Running Time, of the book "Functional Data Structures and Algorithms. A Proof Assistant Approach." See <https://functional-algorithms-verified.org>

Command *time-fun* *f* retrieves the definition of *f* and defines a corresponding step-counting running-time function *T*-*f*. For all auxiliary functions used by *f* (excluding constructors), running time functions must already have been defined. If the definition of the function requires a manual termination proof, use *time-function* accompanied by a *termination* com-

mand.

The pre-defined functions below are assumed to have constant running time. In fact, we make that constant 0. This does not change the asymptotic running time of user-defined functions using the pre-defined functions because 1 is added for every user-defined function call.

Many of the functions below are polymorphic and reside in type classes. The constant-time assumption is justified only for those types where the hardware offers suitable support, e.g. numeric types. The argument size is implicitly bounded, too.

The constant-time assumption for $(=)$ is justified for recursive data types such as lists and trees as long as the comparison is of the form $t = c$ where c is a constant term, for example $xs = []$.

Users of this running time framework need to ensure that 0-time functions are used only within the above restrictions.

```

time-fun-0 min
time-fun-0 max
time-fun-0 (+)
time-fun-0 (-)
time-fun-0 (*)
time-fun-0 (/)
time-fun-0 (div)
time-fun-0 (<)
time-fun-0 (≤)
time-fun-0 Not
time-fun-0 (∧)
time-fun-0 (∨)
time-fun-0 Num.numeral-class.numeral
time-fun-0 (=)

end

```

102 Time functions for various standard library operations. Also defines *itrev*.

```

theory Time-Functions
  imports Time-Commands
begin

time-fun fst
time-fun snd

time-fun (@)

lemma T-append: T-append xs ys = length xs + 1
  <proof>

```



```

class T-size =
  fixes T-size :: 'a  $\Rightarrow$  nat

instantiation list :: (-) T-size
begin

time-fun length

instance  $\langle$ proof $\rangle$ 

end

abbreviation T-length :: 'a list  $\Rightarrow$  nat where
  T-length  $\equiv$  T-size

lemma T-length: T-length xs = length xs + 1
   $\langle$ proof $\rangle$ 

lemmas [simp del] = T-size-list.simps

time-fun map

lemma T-map-simps [simp,code]:
  T-map T-f [] = 1
  T-map T-f (x # xs) = T-f x + T-map T-f xs + 1
   $\langle$ proof $\rangle$ 

lemma T-map: T-map T-f xs = ( $\sum x \leftarrow xs.$  T-f x) + length xs + 1
   $\langle$ proof $\rangle$ 

lemmas [simp del] = T-map-simps

time-fun filter

lemma T-filter-simps [code]:
  T-filter T-P [] = 1
  T-filter T-P (x # xs) = T-P x + T-filter T-P xs + 1
   $\langle$ proof $\rangle$ 

lemma T-filter: T-filter T-P xs = ( $\sum x \leftarrow xs.$  T-P x) + length xs + 1
   $\langle$ proof $\rangle$ 

time-fun nth

lemma T-nth: n < length xs  $\Longrightarrow$  T-nth xs n = n + 1
   $\langle$ proof $\rangle$ 

lemmas [simp del] = T-nth.simps

```

time-fun *take*
time-fun *drop*

lemma *T-take*: $T\text{-take } n \text{ } xs = \min n \text{ (length } xs) + 1$
 $\langle \text{proof} \rangle$

lemma *T-drop*: $T\text{-drop } n \text{ } xs = \min n \text{ (length } xs) + 1$
 $\langle \text{proof} \rangle$

time-fun *rev*

lemma *T-rev*: $T\text{-rev } xs \leq (\text{length } xs + 1)^2$
 $\langle \text{proof} \rangle$

fun *itrev* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
itrev [] *ys* = *ys* |
itrev (*x* # *xs*) *ys* = *itrev xs (x # ys)*

lemma *itrev*: $\text{itrev } xs \text{ } ys = \text{rev } xs @ ys$
 $\langle \text{proof} \rangle$

lemma *itrev-Nil*: $\text{itrev } xs [] = \text{rev } xs$
 $\langle \text{proof} \rangle$

time-fun *itrev*

lemma *T-itrev*: $T\text{-itrev } xs \text{ } ys = \text{length } xs + 1$
 $\langle \text{proof} \rangle$

time-fun *tl*

lemma *T-tl*: $T\text{-tl } xs = 0$
 $\langle \text{proof} \rangle$

declare *T-tl.simps*[*simp del*]

end

103 A table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtranc1-path* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \Rightarrow \text{bool}$
for *r* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$
where

base: $rtrancl\text{-}path\ r\ x\ []\ x$
 | step: $r\ x\ y \implies rtrancl\text{-}path\ r\ y\ ys\ z \implies rtrancl\text{-}path\ r\ x\ (y\ \# \ ys)\ z$

lemma $rtranclp\text{-}eq\text{-}rtrancl\text{-}path$: $r^{**}\ x\ y \longleftrightarrow (\exists\ xs.\ rtrancl\text{-}path\ r\ x\ xs\ y)$
 $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}trans$:
 assumes xy : $rtrancl\text{-}path\ r\ x\ xs\ y$
 and yz : $rtrancl\text{-}path\ r\ y\ ys\ z$
 shows $rtrancl\text{-}path\ r\ x\ (xs\ @\ ys)\ z$ $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}appendE$:
 assumes xz : $rtrancl\text{-}path\ r\ x\ (xs\ @\ y\ \# \ ys)\ z$
 obtains $rtrancl\text{-}path\ r\ x\ (xs\ @\ [y])\ y$ and $rtrancl\text{-}path\ r\ y\ ys\ z$
 $\langle proof \rangle$

lemma $rtrancl\text{-}path\text{-}distinct$:
 assumes xy : $rtrancl\text{-}path\ r\ x\ xs\ y$
 obtains xs' where $rtrancl\text{-}path\ r\ x\ xs'\ y$ and $distinct\ (x\ \# \ xs')$ and $set\ xs' \subseteq set\ xs$
 $\langle proof \rangle$

inductive $rtrancl\text{-}tab$:: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
 for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where

base: $rtrancl\text{-}tab\ r\ xs\ x\ x$
 | step: $x \notin set\ xs \implies r\ x\ y \implies rtrancl\text{-}tab\ r\ (x\ \# \ xs)\ y\ z \implies rtrancl\text{-}tab\ r\ xs\ x\ z$

lemma $rtrancl\text{-}path\text{-}imp\text{-}rtrancl\text{-}tab$:
 assumes $path$: $rtrancl\text{-}path\ r\ x\ xs\ y$
 and x : $distinct\ (x\ \# \ xs)$
 and ys : $(\{x\} \cup set\ xs) \cap set\ ys = \{\}$
 shows $rtrancl\text{-}tab\ r\ ys\ x\ y$
 $\langle proof \rangle$

lemma $rtrancl\text{-}tab\text{-}imp\text{-}rtrancl\text{-}path$:
 assumes tab : $rtrancl\text{-}tab\ r\ ys\ x\ y$
 obtains xs where $rtrancl\text{-}path\ r\ x\ xs\ y$
 $\langle proof \rangle$

lemma $rtranclp\text{-}eq\text{-}rtrancl\text{-}tab\text{-}nil$: $r^{**}\ x\ y \longleftrightarrow rtrancl\text{-}tab\ r\ []\ x\ y$
 $\langle proof \rangle$

declare $rtranclp\text{-}rtrancl\text{-}eq$ [code del]
declare $rtranclp\text{-}eq\text{-}rtrancl\text{-}tab\text{-}nil$ [THEN iffD2, code-pred-intro]

code-pred $rtranclp$
 $\langle proof \rangle$

lemma *rtrancl-path-Range*: $\llbracket \text{rtrancl-path } R \ x \ xs \ y; z \in \text{set } xs \rrbracket \implies \text{Range } R \ z$
 $\langle \text{proof} \rangle$

lemma *rtrancl-path-Range-end*: $\llbracket \text{rtrancl-path } R \ x \ xs \ y; xs \neq [] \rrbracket \implies \text{Range } R \ y$
 $\langle \text{proof} \rangle$

lemma *rtrancl-path-nth*:
 $\llbracket \text{rtrancl-path } R \ x \ xs \ y; i < \text{length } xs \rrbracket \implies R \ ((x \# xs) ! i) \ (xs ! i)$
 $\langle \text{proof} \rangle$

lemma *rtrancl-path-last*: $\llbracket \text{rtrancl-path } R \ x \ xs \ y; xs \neq [] \rrbracket \implies \text{last } xs = y$
 $\langle \text{proof} \rangle$

lemma *rtrancl-path-mono*:
 $\llbracket \text{rtrancl-path } R \ x \ p \ y; \bigwedge x \ y. R \ x \ y \implies S \ x \ y \rrbracket \implies \text{rtrancl-path } S \ x \ p \ y$
 $\langle \text{proof} \rangle$

end

104 Binary Tree

theory *Tree*
imports *Main*
begin

datatype *'a tree* =
 $\text{Leaf } (\langle \rangle) \mid$
 $\text{Node } 'a \ \text{tree } (\text{value: } 'a) \ 'a \ \text{tree } (\langle \langle \text{indent}=1 \ \text{notation}=\langle \text{mixfix Node} \rangle \rangle \langle -, / -, / - \rangle \rangle)$
datatype-compact *tree*

primrec *left* :: *'a tree* \Rightarrow *'a tree* **where**
 $\text{left } (\text{Node } l \ v \ r) = l \mid$
 $\text{left } \text{Leaf} = \text{Leaf}$

primrec *right* :: *'a tree* \Rightarrow *'a tree* **where**
 $\text{right } (\text{Node } l \ v \ r) = r \mid$
 $\text{right } \text{Leaf} = \text{Leaf}$

Counting the number of leaves rather than nodes:

fun *size1* :: *'a tree* \Rightarrow *nat* **where**
 $\text{size1 } \langle \rangle = 1 \mid$
 $\text{size1 } \langle l, x, r \rangle = \text{size1 } l + \text{size1 } r$

fun *subtrees* :: *'a tree* \Rightarrow *'a tree set* **where**
 $\text{subtrees } \langle \rangle = \{ \langle \rangle \} \mid$
 $\text{subtrees } (\langle l, a, r \rangle) = \{ \langle l, a, r \rangle \} \cup \text{subtrees } l \cup \text{subtrees } r$

fun *mirror* :: *'a tree* \Rightarrow *'a tree* **where**
 $\text{mirror } \langle \rangle = \text{Leaf} \mid$

$\text{mirror } \langle l, x, r \rangle = \langle \text{mirror } r, x, \text{mirror } l \rangle$

class *height* = **fixes** *height* :: 'a \Rightarrow nat

instantiation *tree* :: (type)*height*
begin

fun *height-tree* :: 'a *tree* \Rightarrow nat **where**
height Leaf = 0 |
height (Node *l* *a* *r*) = max (*height* *l*) (*height* *r*) + 1

instance $\langle \text{proof} \rangle$

end

fun *min-height* :: 'a *tree* \Rightarrow nat **where**
min-height Leaf = 0 |
min-height (Node *l* - *r*) = min (*min-height* *l*) (*min-height* *r*) + 1

fun *complete* :: 'a *tree* \Rightarrow bool **where**
complete Leaf = True |
complete (Node *l* *x* *r*) = (*height* *l* = *height* *r* \wedge *complete* *l* \wedge *complete* *r*)

Almost complete:

definition *acomplete* :: 'a *tree* \Rightarrow bool **where**
acomplete *t* = (*height* *t* - *min-height* *t* \leq 1)

Weight balanced:

fun *wbalanced* :: 'a *tree* \Rightarrow bool **where**
wbalanced Leaf = True |
wbalanced (Node *l* *x* *r*) = (abs(int(size *l*) - int(size *r*)) \leq 1 \wedge *wbalanced* *l* \wedge *wbalanced* *r*)

Internal path length:

fun *ipl* :: 'a *tree* \Rightarrow nat **where**
ipl Leaf = 0 |
ipl (Node *l* - *r*) = *ipl* *l* + size *l* + *ipl* *r* + size *r*

fun *preorder* :: 'a *tree* \Rightarrow 'a *list* **where**
preorder $\langle \rangle$ = [] |
preorder $\langle l, x, r \rangle$ = *x* # *preorder* *l* @ *preorder* *r*

fun *inorder* :: 'a *tree* \Rightarrow 'a *list* **where**
inorder $\langle \rangle$ = [] |
inorder $\langle l, x, r \rangle$ = *inorder* *l* @ [*x*] @ *inorder* *r*

A linear version avoiding append:

fun *inorder2* :: 'a *tree* \Rightarrow 'a *list* \Rightarrow 'a *list* **where**
inorder2 $\langle \rangle$ *xs* = *xs* |

$inorder2 \langle l, x, r \rangle xs = inorder2 \ l \ (x \# \ inorder2 \ r \ xs)$

fun *postorder* :: 'a tree \Rightarrow 'a list **where**
postorder $\langle \rangle = []$ |
postorder $\langle l, x, r \rangle = \text{postorder } l @ \text{postorder } r @ [x]$

Binary Search Tree:

fun *bst-wrt* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a tree \Rightarrow bool **where**
bst-wrt *P* $\langle \rangle \longleftrightarrow \text{True}$ |
bst-wrt *P* $\langle l, a, r \rangle \longleftrightarrow$
 $(\forall x \in \text{set-tree } l. \ P \ x \ a) \wedge (\forall x \in \text{set-tree } r. \ P \ a \ x) \wedge \text{bst-wrt } P \ l \wedge \text{bst-wrt } P \ r$

abbreviation *bst* :: ('a::linorder) tree \Rightarrow bool **where**
bst $\equiv \text{bst-wrt } (<)$

fun (in *linorder*) *heap* :: 'a tree \Rightarrow bool **where**
heap *Leaf* = *True* |
heap (Node *l m r*) =
 $((\forall x \in \text{set-tree } l \cup \text{set-tree } r. \ m \leq x) \wedge \text{heap } l \wedge \text{heap } r)$

104.1 map-tree

lemma *eq-map-tree-Leaf[simp]*: *map-tree f t = Leaf* $\longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *eq-Leaf-map-tree[simp]*: *Leaf = map-tree f t* $\longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

104.2 size

lemma *size1-size*: *size1 t = size t + 1*
 $\langle \text{proof} \rangle$

lemma *size1-ge0[simp]*: $0 < \text{size1 } t$
 $\langle \text{proof} \rangle$

lemma *eq-size-0[simp]*: *size t = 0* $\longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *eq-0-size[simp]*: $0 = \text{size } t \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *neq-Leaf-iff*: $(t \neq \langle \rangle) = (\exists l \ a \ r. \ t = \langle l, a, r \rangle)$
 $\langle \text{proof} \rangle$

lemma *size-map-tree[simp]*: *size (map-tree f t) = size t*
 $\langle \text{proof} \rangle$

lemma *size1-map-tree[simp]*: *size1 (map-tree f t) = size1 t*
 $\langle \text{proof} \rangle$

104.3 *set-tree*

lemma *eq-set-tree-empty[simp]*: $\text{set-tree } t = \{\} \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *eq-empty-set-tree[simp]*: $\{\} = \text{set-tree } t \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *finite-set-tree[simp]*: $\text{finite}(\text{set-tree } t)$
 $\langle \text{proof} \rangle$

104.4 *subtrees*

lemma *neq-subtrees-empty[simp]*: $\text{subtrees } t \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *neq-empty-subtrees[simp]*: $\{\} \neq \text{subtrees } t$
 $\langle \text{proof} \rangle$

lemma *size-subtrees*: $s \in \text{subtrees } t \implies \text{size } s \leq \text{size } t$
 $\langle \text{proof} \rangle$

lemma *set-treeE*: $a \in \text{set-tree } t \implies \exists l r. \langle l, a, r \rangle \in \text{subtrees } t$
 $\langle \text{proof} \rangle$

lemma *Node-notin-subtrees-if[simp]*: $a \notin \text{set-tree } t \implies \text{Node } l a r \notin \text{subtrees } t$
 $\langle \text{proof} \rangle$

lemma *in-set-tree-if*: $\langle l, a, r \rangle \in \text{subtrees } t \implies a \in \text{set-tree } t$
 $\langle \text{proof} \rangle$

104.5 *height and min-height*

lemma *eq-height-0[simp]*: $\text{height } t = 0 \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *eq-0-height[simp]*: $0 = \text{height } t \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *height-map-tree[simp]*: $\text{height } (\text{map-tree } f t) = \text{height } t$
 $\langle \text{proof} \rangle$

lemma *height-le-size-tree*: $\text{height } t \leq \text{size } (t::'a \text{ tree})$
 $\langle \text{proof} \rangle$

lemma *size1-height*: $\text{size1 } t \leq 2 \wedge \text{height } (t::'a \text{ tree})$
 $\langle \text{proof} \rangle$

corollary *size-height*: $\text{size } t \leq 2 \wedge \text{height } (t::'a \text{ tree}) - 1$
 $\langle \text{proof} \rangle$

lemma *height-subtrees*: $s \in \text{subtrees } t \implies \text{height } s \leq \text{height } t$
 ⟨proof⟩

lemma *min-height-le-height*: $\text{min-height } t \leq \text{height } t$
 ⟨proof⟩

lemma *min-height-map-tree[simp]*: $\text{min-height } (\text{map-tree } f \ t) = \text{min-height } t$
 ⟨proof⟩

lemma *min-height-size1*: $2^{\text{min-height } t} \leq \text{size1 } t$
 ⟨proof⟩

104.6 complete

lemma *complete-iff-height*: $\text{complete } t \iff (\text{min-height } t = \text{height } t)$
 ⟨proof⟩

lemma *size1-if-complete*: $\text{complete } t \implies \text{size1 } t = 2^{\text{height } t}$
 ⟨proof⟩

lemma *size-if-complete*: $\text{complete } t \implies \text{size } t = 2^{\text{height } t} - 1$
 ⟨proof⟩

lemma *size1-height-if-incomplete*:
 $\neg \text{complete } t \implies \text{size1 } t < 2^{\text{height } t}$
 ⟨proof⟩

lemma *complete-iff-min-height*: $\text{complete } t \iff (\text{height } t = \text{min-height } t)$
 ⟨proof⟩

lemma *min-height-size1-if-incomplete*:
 $\neg \text{complete } t \implies 2^{\text{min-height } t} < \text{size1 } t$
 ⟨proof⟩

lemma *complete-if-size1-height*: $\text{size1 } t = 2^{\text{height } t} \implies \text{complete } t$
 ⟨proof⟩

lemma *complete-if-size1-min-height*: $\text{size1 } t = 2^{\text{min-height } t} \implies \text{complete } t$
 ⟨proof⟩

lemma *complete-iff-size1*: $\text{complete } t \iff \text{size1 } t = 2^{\text{height } t}$
 ⟨proof⟩

104.7 acomplete

lemma *acomplete-subtreeL*: $\text{acomplete } (\text{Node } l \ x \ r) \implies \text{acomplete } l$
 ⟨proof⟩

lemma *acomplete-subtreeR*: $acomplete\ (Node\ l\ x\ r) \implies acomplete\ r$
 $\langle proof \rangle$

lemma *acomplete-subtrees*: $\llbracket acomplete\ t; s \in subtrees\ t \rrbracket \implies acomplete\ s$
 $\langle proof \rangle$

Balanced trees have optimal height:

lemma *acomplete-optimal*:
fixes $t :: 'a\ tree$ **and** $t' :: 'b\ tree$
assumes $acomplete\ t$ **size** $t \leq size\ t'$ **shows** $height\ t \leq height\ t'$
 $\langle proof \rangle$

104.8 *wbalanced*

lemma *wbalanced-subtrees*: $\llbracket wbalanced\ t; s \in subtrees\ t \rrbracket \implies wbalanced\ s$
 $\langle proof \rangle$

104.9 *ipl*

The internal path length of a tree:

lemma *ipl-if-complete-int*:
 $complete\ t \implies int(ipl\ t) = (int(height\ t) - 2) * 2^{height\ t} + 2$
 $\langle proof \rangle$

104.10 List of entries

lemma *eq-inorder-Nil[simp]*: $inorder\ t = [] \longleftrightarrow t = Leaf$
 $\langle proof \rangle$

lemmas *eq-Nil-inorder[simp]* = *eq-inorder-Nil[THEN eq-iff-swap]*

lemma *set-inorder[simp]*: $set\ (inorder\ t) = set-tree\ t$
 $\langle proof \rangle$

lemma *preorder-eq-Nil-iff[simp]*: $(preorder\ t = []) = (t = \langle \rangle)$
 $\langle proof \rangle$

lemmas *Nil-eq-preorder-iff [simp]* = *preorder-eq-Nil-iff[THEN eq-iff-swap]*

lemma *preorder-eq-Cons-iff*:
 $preorder\ t = x \# xs \longleftrightarrow (\exists\ l\ r. t = \langle l, x, r \rangle \wedge xs = preorder\ l @ preorder\ r)$
 $\langle proof \rangle$

lemmas *Cons-eq-preorder-iff* = *preorder-eq-Cons-iff[THEN eq-iff-swap]*

lemma *set-preorder[simp]*: $set\ (preorder\ t) = set-tree\ t$
 $\langle proof \rangle$

lemma *set-postorder[simp]*: $set\ (postorder\ t) = set-tree\ t$

$\langle \text{proof} \rangle$

lemma *length-preorder[simp]*: $\text{length } (\text{preorder } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *length-inorder[simp]*: $\text{length } (\text{inorder } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *length-postorder[simp]*: $\text{length } (\text{postorder } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *preorder-map*: $\text{preorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{preorder } t)$
 $\langle \text{proof} \rangle$

lemma *inorder-map*: $\text{inorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{inorder } t)$
 $\langle \text{proof} \rangle$

lemma *postorder-map*: $\text{postorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{postorder } t)$
 $\langle \text{proof} \rangle$

lemma *inorder2-inorder*: $\text{inorder2 } t \ xs = \text{inorder } t \ @ \ xs$
 $\langle \text{proof} \rangle$

104.11 Binary Search Tree

lemma *bst-wrt-mono*: $(\bigwedge x \ y. P \ x \ y \implies Q \ x \ y) \implies \text{bst-wrt } P \ t \implies \text{bst-wrt } Q \ t$
 $\langle \text{proof} \rangle$

lemma *bst-wrt-le-if-bst*: $\text{bst } t \implies \text{bst-wrt } (\leq) \ t$
 $\langle \text{proof} \rangle$

lemma *bst-wrt-le-iff-sorted*: $\text{bst-wrt } (\leq) \ t \longleftrightarrow \text{sorted } (\text{inorder } t)$
 $\langle \text{proof} \rangle$

lemma *bst-iff-sorted-wrt-less*: $\text{bst } t \longleftrightarrow \text{sorted-wrt } (<) \ (\text{inorder } t)$
 $\langle \text{proof} \rangle$

104.12 heap

104.13 mirror

lemma *mirror-Leaf[simp]*: $\text{mirror } t = \langle \rangle \longleftrightarrow t = \langle \rangle$
 $\langle \text{proof} \rangle$

lemma *Leaf-mirror[simp]*: $\langle \rangle = \text{mirror } t \longleftrightarrow t = \langle \rangle$
 $\langle \text{proof} \rangle$

lemma *size-mirror[simp]*: $\text{size}(\text{mirror } t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *size1-mirror*[simp]: $\text{size1}(\text{mirror } t) = \text{size1 } t$
 $\langle \text{proof} \rangle$

lemma *height-mirror*[simp]: $\text{height}(\text{mirror } t) = \text{height } t$
 $\langle \text{proof} \rangle$

lemma *min-height-mirror* [simp]: $\text{min-height}(\text{mirror } t) = \text{min-height } t$
 $\langle \text{proof} \rangle$

lemma *ipl-mirror* [simp]: $\text{ipl}(\text{mirror } t) = \text{ipl } t$
 $\langle \text{proof} \rangle$

lemma *inorder-mirror*: $\text{inorder}(\text{mirror } t) = \text{rev}(\text{inorder } t)$
 $\langle \text{proof} \rangle$

lemma *map-mirror*: $\text{map-tree } f(\text{mirror } t) = \text{mirror}(\text{map-tree } f t)$
 $\langle \text{proof} \rangle$

lemma *mirror-mirror*[simp]: $\text{mirror}(\text{mirror } t) = t$
 $\langle \text{proof} \rangle$

end

105 Multiset of Elements of Binary Tree

theory *Tree-Multiset*
imports *Multiset Tree*
begin

Kept separate from theory *HOL-Library.Tree* to avoid importing all of theory *HOL-Library.Multiset* into *HOL-Library.Tree*. Should be merged if *HOL-Library.Multiset* ever becomes part of *Main*.

fun *mset-tree* :: $'a \text{ tree} \Rightarrow 'a \text{ multiset}$ **where**
mset-tree Leaf = $\{\#\}$ |
mset-tree (Node $l \ a \ r$) = $\{\#a\# \} + \text{mset-tree } l + \text{mset-tree } r$

fun *subtrees-mset* :: $'a \text{ tree} \Rightarrow 'a \text{ tree multiset}$ **where**
subtrees-mset Leaf = $\{\# \text{Leaf} \# \}$ |
subtrees-mset (Node $l \ x \ r$) = $\text{add-mset}(\text{Node } l \ x \ r) (\text{subtrees-mset } l + \text{subtrees-mset } r)$

lemma *mset-tree-empty-iff*[simp]: $\text{mset-tree } t = \{\#\} \longleftrightarrow t = \text{Leaf}$
 $\langle \text{proof} \rangle$

lemma *set-mset-tree*[simp]: $\text{set-mset}(\text{mset-tree } t) = \text{set-tree } t$
 $\langle \text{proof} \rangle$

lemma *size-mset-tree*[simp]: $\text{size}(\text{mset-tree } t) = \text{size } t$

$\langle proof \rangle$

lemma *mset-map-tree*: $mset-tree (map-tree f t) = image-mset f (mset-tree t)$
 $\langle proof \rangle$

lemma *mset-iff-set-tree*: $x \in \# mset-tree t \longleftrightarrow x \in set-tree t$
 $\langle proof \rangle$

lemma *mset-preorder[simp]*: $mset (preorder t) = mset-tree t$
 $\langle proof \rangle$

lemma *mset-inorder[simp]*: $mset (inorder t) = mset-tree t$
 $\langle proof \rangle$

lemma *map-mirror*: $mset-tree (mirror t) = mset-tree t$
 $\langle proof \rangle$

lemma *in-subtrees-mset-iff[simp]*: $s \in \# subtrees-mset t \longleftrightarrow s \in subtrees t$
 $\langle proof \rangle$

end

theory *Tree-Real*
imports
 Complex-Main
 Tree
begin

This theory is separate from *HOL-Library.Tree* because the former is discrete and builds on *Main* whereas this theory builds on *Complex-Main*.

lemma *size1-height-log*: $\log 2 (size1 t) \leq height t$
 $\langle proof \rangle$

lemma *min-height-size1-log*: $min-height t \leq \log 2 (size1 t)$
 $\langle proof \rangle$

lemma *size1-log-if-complete*: $complete t \implies height t = \log 2 (size1 t)$
 $\langle proof \rangle$

lemma *min-height-size1-log-if-incomplete*:
 $\neg complete t \implies min-height t < \log 2 (size1 t)$
 $\langle proof \rangle$

lemma *min-height-acomplete*: **assumes** *acomplete t*
shows $min-height t = nat(floor(\log 2 (size1 t)))$
 $\langle proof \rangle$

lemma *height-acomplete*: **assumes** *acomplete t*
shows *height t = nat(ceiling(log 2 (size1 t)))*
 $\langle \text{proof} \rangle$

lemma *acomplete-Node-if-wbal1*:
assumes *acomplete l acomplete r size l = size r + 1*
shows *acomplete $\langle l, x, r \rangle$*
 $\langle \text{proof} \rangle$

lemma *acomplete-sym*: *acomplete $\langle l, x, r \rangle \implies \text{acomplete } \langle r, y, l \rangle$*
 $\langle \text{proof} \rangle$

lemma *acomplete-Node-if-wbal2*:
assumes *acomplete l acomplete r abs(int(size l) - int(size r)) ≤ 1*
shows *acomplete $\langle l, x, r \rangle$*
 $\langle \text{proof} \rangle$

lemma *acomplete-if-wbalanced*: *wbalanced t $\implies \text{acomplete } t$*
 $\langle \text{proof} \rangle$

end

106 Unordered pairs

theory *Uprod* **imports** *Main* **begin**

typedef (*'a*, *'b*) *commute* = {*f* :: *'a* \Rightarrow *'a* \Rightarrow *'b*. $\forall x y. f x y = f y x$ }
morphisms *apply-commute* *Abs-commute*
 $\langle \text{proof} \rangle$

setup-lifting *type-definition-commute*

lemma *apply-commute-commute*: *apply-commute f x y = apply-commute f y x*
 $\langle \text{proof} \rangle$

context **includes** *lifting-syntax* **begin**

lift-definition *rel-commute* :: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*'c* \Rightarrow *'d* \Rightarrow *bool*) \Rightarrow (*'a*, *'c*)
commute \Rightarrow (*'b*, *'d*) *commute* \Rightarrow *bool*
is $\lambda A B. A \implies B \implies A \implies B$ $\langle \text{proof} \rangle$

end

definition *eq-upair* :: (*'a* \times *'a*) \Rightarrow (*'a* \times *'a*) \Rightarrow *bool*
where *eq-upair* = ($\lambda(a, b) (c, d). a = c \wedge b = d \vee a = d \wedge b = c$)

lemma *eq-upair-simps* [*simp*]:
eq-upair (a, b) (c, d) $\longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
 $\langle \text{proof} \rangle$

lemma *equivp-eq-upair*: *equivp eq-upair*
 $\langle \text{proof} \rangle$

quotient-type *'a uprod* = *'a* \times *'a* / *eq-upair* $\langle \text{proof} \rangle$

lift-definition *Upair* :: *'a* \Rightarrow *'a* \Rightarrow *'a uprod* **is** *Pair* **parametric** *Pair-transfer*[*of A A for A*] $\langle \text{proof} \rangle$

lemma *uprod-exhaust* [*case-names Upair, cases type: uprod*]:
obtains *a b* **where** *x* = *Upair a b*
 $\langle \text{proof} \rangle$

lemma *Upair-inject* [*simp*]: *Upair a b* = *Upair c d* \longleftrightarrow *a* = *c* \wedge *b* = *d* \vee *a* = *d* \wedge *b* = *c*
 $\langle \text{proof} \rangle$

code-datatype *Upair*

lift-definition *case-uprod* :: (*'a, 'b*) *commute* \Rightarrow *'a uprod* \Rightarrow *'b* **is** *case-prod*
parametric *case-prod-transfer*[*of A A for A*] $\langle \text{proof} \rangle$

lemma *case-uprod-simps* [*simp, code*]: *case-uprod f (Upair x y)* = *apply-commute f x y*
 $\langle \text{proof} \rangle$

lemma *uprod-split*: *P (case-uprod f x)* \longleftrightarrow ($\forall a b. x = \text{Upair } a b \longrightarrow P (\text{apply-commute } f a b)$)
 $\langle \text{proof} \rangle$

lemma *uprod-split-asm*: *P (case-uprod f x)* \longleftrightarrow $\neg (\exists a b. x = \text{Upair } a b \wedge \neg P (\text{apply-commute } f a b))$
 $\langle \text{proof} \rangle$

lift-definition *not-equal* :: (*'a, bool*) *commute* **is** (\neq) $\langle \text{proof} \rangle$

lemma *apply-not-equal* [*simp*]: *apply-commute not-equal x y* \longleftrightarrow *x* \neq *y*
 $\langle \text{proof} \rangle$

definition *proper-uprod* :: *'a uprod* \Rightarrow *bool*
where *proper-uprod* = *case-uprod not-equal*

lemma *proper-uprod-simps* [*simp, code*]: *proper-uprod (Upair x y)* \longleftrightarrow *x* \neq *y*
 $\langle \text{proof} \rangle$

context includes *lifting-syntax* **begin**

private lemma *set-uprod-parametric'*:
(*rel-prod A A* \implies *rel-set A*) ($\lambda(a, b). \{a, b\}$) ($\lambda(a, b). \{a, b\}$)

<proof>

lift-definition *set-uprod* :: 'a uprod \Rightarrow 'a set **is** $\lambda(a, b). \{a, b\}$
parametric *set-uprod-parametric'* *<proof>*

lemma *set-uprod-simps* [simp, code]: *set-uprod* (Upair *x y*) = {*x, y*}
<proof>

lemma *finite-set-uprod* [simp]: *finite* (*set-uprod x*)
<proof> **lemma** *map-uprod-parametric'*:
 $((A \implies B) \implies \text{rel-prod } A \ A \implies \text{rel-prod } B \ B) (\lambda f. \text{map-prod } f \ f) (\lambda f. \text{map-prod } f \ f)$
<proof>

lift-definition *map-uprod* :: ('a \Rightarrow 'b) \Rightarrow 'a uprod \Rightarrow 'b uprod **is** $\lambda f. \text{map-prod } f \ f$
parametric *map-uprod-parametric'* *<proof>*

lemma *map-uprod-simps* [simp, code]: *map-uprod* *f* (Upair *x y*) = Upair (*f x*) (*f y*)
<proof> **lemma** *rel-uprod-transfer'*:
 $((A \implies B \implies (=)) \implies \text{rel-prod } A \ A \implies \text{rel-prod } B \ B \implies (=))$
 $(\lambda R \ (a, b) \ (c, d). R \ a \ c \wedge R \ b \ d \vee R \ a \ d \wedge R \ b \ c) (\lambda R \ (a, b) \ (c, d). R \ a \ c \wedge R \ b \ d \vee R \ a \ d \wedge R \ b \ c)$
<proof>

lift-definition *rel-uprod* :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a uprod \Rightarrow 'b uprod \Rightarrow bool
is $\lambda R \ (a, b) \ (c, d). R \ a \ c \wedge R \ b \ d \vee R \ a \ d \wedge R \ b \ c$ **parametric** *rel-uprod-transfer'*
<proof>

lemma *rel-uprod-simps* [simp, code]:
 $\text{rel-uprod } R \ (\text{Upair } a \ b) \ (\text{Upair } c \ d) \longleftrightarrow R \ a \ c \wedge R \ b \ d \vee R \ a \ d \wedge R \ b \ c$
<proof>

lemma *Upair-parametric* [transfer-rule]: $(A \implies A \implies \text{rel-uprod } A) \text{ Upair Upair}$
<proof>

lemma *case-uprod-parametric* [transfer-rule]:
 $(\text{rel-commute } A \ B \implies \text{rel-uprod } A \implies B) \text{ case-uprod case-uprod}$
<proof>

end

bnf *uprod*: 'a uprod
map: map-uprod
sets: set-uprod
bd: natLeq
rel: rel-uprod

<proof>

lemma *pred-uprod-code* [*simp*, *code*]: *pred-uprod* *P* (*Upair* *x y*) \longleftrightarrow *P* *x* \wedge *P* *y*
<proof>

instantiation *uprod* :: (*equal*) *equal* **begin**

definition *equal-uprod* :: '*a* *uprod* \Rightarrow '*a* *uprod* \Rightarrow *bool*
where *equal-uprod* = (=)

lemma *equal-uprod-code* [*code*]:
HOL.equal (*Upair* *x y*) (*Upair* *z u*) \longleftrightarrow *x* = *z* \wedge *y* = *u* \vee *x* = *u* \wedge *y* = *z*
<proof>

instance *<proof>*
end

quickcheck-generator *uprod* *constructors*: *Upair*

lemma *UNIV-uprod*: *UNIV* = ($\lambda x.$ *Upair* *x x*) ‘ *UNIV* \cup ($\lambda(x, y).$ *Upair* *x y*) ‘
Sigma *UNIV* ($\lambda x.$ *UNIV* - {*x*})
<proof>

context **begin**
private lift-definition *upair-inv* :: '*a* *uprod* \Rightarrow '*a*
is $\lambda(x, y).$ *if* *x* = *y* *then* *x* *else* *undefined* *<proof>*

lemma *finite-UNIV-prod* [*simp*]:
finite (*UNIV* :: '*a* *uprod* *set*) \longleftrightarrow *finite* (*UNIV* :: '*a* *set*) (**is** ?*lhs* = ?*rhs*)
<proof>

end

lemma *card-UNIV-uprod*:
 $\text{card } (\text{UNIV} :: 'a \text{ uprod set}) = \text{card } (\text{UNIV} :: 'a \text{ set}) * (\text{card } (\text{UNIV} :: 'a \text{ set}) + 1) \text{ div } 2$
(**is** ?*UPROD* = ?*A* * - *div* -)
<proof>

end

107 A type of finite bit strings

theory *Word*
imports
HOL-Library.Type-Length
begin

107.1 Preliminaries

lemma *signed-take-bit-decr-length-iff*:

$\langle \text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - \text{Suc } 0) \ k = \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \ l$
 $\longleftrightarrow \text{take-bit } \text{LENGTH}('a) \ k = \text{take-bit } \text{LENGTH}('a) \ l \rangle$
 $\langle \text{proof} \rangle$

107.2 Fundamentals

107.2.1 Type definition

quotient-type (overloaded) $'a \text{ word} = \text{int} / \langle \lambda k \ l. \text{take-bit } \text{LENGTH}('a) \ k = \text{take-bit } \text{LENGTH}('a::\text{len}) \ l \rangle$
morphisms *rep* *Word* $\langle \text{proof} \rangle$

hide-const (**open**) *rep* — only for foundational purpose

hide-const (**open**) *Word* — only for code generation

107.2.2 Basic arithmetic

instantiation *word* :: (*len*) *comm-ring-1*
begin

lift-definition *zero-word* :: $\langle 'a \text{ word} \rangle$
is *0* $\langle \text{proof} \rangle$

lift-definition *one-word* :: $\langle 'a \text{ word} \rangle$
is *1* $\langle \text{proof} \rangle$

lift-definition *plus-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle (+) \rangle$
 $\langle \text{proof} \rangle$

lift-definition *minus-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle (-) \rangle$
 $\langle \text{proof} \rangle$

lift-definition *uminus-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *uminus*
 $\langle \text{proof} \rangle$

lift-definition *times-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle (*) \rangle$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

```

context
  includes lifting-syntax
  notes
    power-transfer [transfer-rule]
    transfer-rule-of-bool [transfer-rule]
    transfer-rule-numeral [transfer-rule]
    transfer-rule-of-nat [transfer-rule]
    transfer-rule-of-int [transfer-rule]
begin

lemma power-transfer-word [transfer-rule]:
  ⟨(pcr-word ==> (=) ==> pcr-word) (∧) (∧)⟩
  ⟨proof⟩

lemma [transfer-rule]:
  ⟨((=) ==> pcr-word) of-bool of-bool⟩
  ⟨proof⟩

lemma [transfer-rule]:
  ⟨((=) ==> pcr-word) numeral numeral⟩
  ⟨proof⟩

lemma [transfer-rule]:
  ⟨((=) ==> pcr-word) int of-nat⟩
  ⟨proof⟩

lemma [transfer-rule]:
  ⟨((=) ==> pcr-word) (λk. k) of-int⟩
  ⟨proof⟩

lemma [transfer-rule]:
  ⟨(pcr-word ==> (⟷)) even ((dvd) 2 :: 'a::len word ⇒ bool)⟩
  ⟨proof⟩

end

lemma exp-eq-zero-iff [simp]:
  ⟨2 ^ n = (0 :: 'a::len word) ⟷ n ≥ LENGTH('a)⟩
  ⟨proof⟩

lemma word-exp-length-eq-0 [simp]:
  ⟨(2 :: 'a::len word) ^ LENGTH('a) = 0⟩
  ⟨proof⟩

```

107.2.3 Basic tool setup

⟨*ML*⟩

107.2.4 Basic code generation setup**context****begin**

qualified lift-definition *the-int* :: $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$
is $\langle \text{take-bit LENGTH}('a) \rangle \langle \text{proof} \rangle$

end

lemma [*code abstype*]:
 $\langle \text{Word.Word} (\text{Word.the-int } w) = w \rangle$
 $\langle \text{proof} \rangle$

lemma *Word-eq-word-of-int* [*code-post, simp*]:
 $\langle \text{Word.Word} = \text{of-int} \rangle$
 $\langle \text{proof} \rangle$

quickcheck-generator *word*
constructors:
 $\langle 0 :: 'a::\text{len word} \rangle,$
 $\langle \text{numeral} :: \text{num} \Rightarrow 'a::\text{len word} \rangle$

instantiation *word* :: (*len*) *equal***begin**

lift-definition *equal-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$
is $\langle \lambda k l. \text{take-bit LENGTH}('a) k = \text{take-bit LENGTH}('a) l \rangle$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$ **end**

lemma [*code*]:
 $\langle \text{HOL.equal } v w \longleftrightarrow \text{HOL.equal} (\text{Word.the-int } v) (\text{Word.the-int } w) \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle \text{Word.the-int } 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle \text{Word.the-int } 1 = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\langle \text{Word.the-int } (v + w) = \text{take-bit LENGTH}('a) (\text{Word.the-int } v + \text{Word.the-int } w) \rangle$

for $v\ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (-\ w) = (\text{let } k = \text{Word.the-int } w \text{ in if } w = 0 \text{ then } 0 \text{ else } 2^{\wedge} \text{LENGTH('a)} - k) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (v - w) = \text{take-bit LENGTH('a)} (\text{Word.the-int } v - \text{Word.the-int } w) \rangle$
for $v\ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (v * w) = \text{take-bit LENGTH('a)} (\text{Word.the-int } v * \text{Word.the-int } w) \rangle$
for $v\ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

107.2.5 Basic conversions

abbreviation $\text{word-of-nat} :: \langle \text{nat} \Rightarrow 'a::\text{len word} \rangle$
where $\langle \text{word-of-nat} \equiv \text{of-nat} \rangle$

abbreviation $\text{word-of-int} :: \langle \text{int} \Rightarrow 'a::\text{len word} \rangle$
where $\langle \text{word-of-int} \equiv \text{of-int} \rangle$

lemma $\text{word-of-nat-eq-iff}$:
 $\langle \text{word-of-nat } m = (\text{word-of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a)}\ m = \text{take-bit LENGTH('a)}\ n \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-of-int-eq-iff}$:
 $\langle \text{word-of-int } k = (\text{word-of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a)}\ k = \text{take-bit LENGTH('a)}\ l \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-of-nat-eq-0-iff}$:
 $\langle \text{word-of-nat } n = 0 :: 'a::\text{len word} \longleftrightarrow 2^{\wedge} \text{LENGTH('a)}\ \text{dvd } n \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{word-of-int-eq-0-iff}$:
 $\langle \text{word-of-int } k = 0 :: 'a::\text{len word} \longleftrightarrow 2^{\wedge} \text{LENGTH('a)}\ \text{dvd } k \rangle$
 $\langle \text{proof} \rangle$

context semiring-1
begin

lift-definition *unsigned* :: $\langle 'b::len\ word \Rightarrow 'a \rangle$
 is $\langle of_nat \circ nat \circ take_bit\ LENGTH('b) \rangle$
 $\langle proof \rangle$

lemma *unsigned-0* [simp]:
 $\langle unsigned\ 0 = 0 \rangle$
 $\langle proof \rangle$

lemma *unsigned-1* [simp]:
 $\langle unsigned\ 1 = 1 \rangle$
 $\langle proof \rangle$

lemma *unsigned-numeral* [simp]:
 $\langle unsigned\ (numeral\ n :: 'b::len\ word) = of_nat\ (take_bit\ LENGTH('b)\ (numeral\ n)) \rangle$
 $\langle proof \rangle$

lemma *unsigned-neg-numeral* [simp]:
 $\langle unsigned\ (-\ numeral\ n :: 'b::len\ word) = of_nat\ (nat\ (take_bit\ LENGTH('b)\ (-\ numeral\ n))) \rangle$
 $\langle proof \rangle$

end

context *semiring-1*
begin

lemma *unsigned-of-nat*:
 $\langle unsigned\ (word_of_nat\ n :: 'b::len\ word) = of_nat\ (take_bit\ LENGTH('b)\ n) \rangle$
 $\langle proof \rangle$

lemma *unsigned-of-int*:
 $\langle unsigned\ (word_of_int\ k :: 'b::len\ word) = of_nat\ (nat\ (take_bit\ LENGTH('b)\ k)) \rangle$
 $\langle proof \rangle$

end

context *semiring-char-0*
begin

lemma *unsigned-word-eqI*:
 $\langle v = w \rangle$ **if** $\langle unsigned\ v = unsigned\ w \rangle$
 $\langle proof \rangle$

lemma *word-eq-iff-unsigned*:
 $\langle v = w \iff unsigned\ v = unsigned\ w \rangle$
 $\langle proof \rangle$

lemma *inj-unsigned* [simp]:

⟨*inj unsigned*⟩

⟨*proof*⟩

lemma *unsigned-eq-0-iff*:

⟨*unsigned* $w = 0 \longleftrightarrow w = 0$ ⟩

⟨*proof*⟩

end

context *ring-1*

begin

lift-definition *signed* :: $\langle 'b::\text{len word} \Rightarrow 'a \rangle$

is $\langle \text{of-int} \circ \text{signed-take-bit } (\text{LENGTH}('b) - \text{Suc } 0) \rangle$

⟨*proof*⟩

lemma *signed-0* [simp]:

⟨*signed* $0 = 0$ ⟩

⟨*proof*⟩

lemma *signed-1* [simp]:

⟨*signed* $(1 :: 'b::\text{len word}) = (\text{if } \text{LENGTH}('b) = 1 \text{ then } - 1 \text{ else } 1)$ ⟩

⟨*proof*⟩

lemma *signed-minus-1* [simp]:

⟨*signed* $(- 1 :: 'b::\text{len word}) = - 1$ ⟩

⟨*proof*⟩

lemma *signed-numeral* [simp]:

⟨*signed* (*numeral* $n :: 'b::\text{len word}$) = *of-int* (*signed-take-bit* ($\text{LENGTH}('b) - 1$) (*numeral* n))⟩

⟨*proof*⟩

lemma *signed-neg-numeral* [simp]:

⟨*signed* $(- \text{numeral } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - 1) (- \text{numeral } n))$ ⟩

⟨*proof*⟩

lemma *signed-of-nat*:

⟨*signed* (*word-of-nat* $n :: 'b::\text{len word}$) = *of-int* (*signed-take-bit* ($\text{LENGTH}('b) - \text{Suc } 0$) (*int* n))⟩

⟨*proof*⟩

lemma *signed-of-int*:

⟨*signed* (*word-of-int* $n :: 'b::\text{len word}$) = *of-int* (*signed-take-bit* ($\text{LENGTH}('b) - \text{Suc } 0$) n)⟩

⟨*proof*⟩

end

context *ring-char-0*
begin

lemma *signed-word-eqI*:
 $\langle v = w \rangle$ **if** $\langle \text{signed } v = \text{signed } w \rangle$
 $\langle \text{proof} \rangle$

lemma *word-eq-iff-signed*:
 $\langle v = w \iff \text{signed } v = \text{signed } w \rangle$
 $\langle \text{proof} \rangle$

lemma *inj-signed* [*simp*]:
 $\langle \text{inj signed} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-eq-0-iff*:
 $\langle \text{signed } w = 0 \iff w = 0 \rangle$
 $\langle \text{proof} \rangle$

end

abbreviation *unat* :: $\langle 'a::\text{len word} \Rightarrow \text{nat} \rangle$
where $\langle \text{unat} \equiv \text{unsigned} \rangle$

abbreviation *uint* :: $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$
where $\langle \text{uint} \equiv \text{unsigned} \rangle$

abbreviation *sint* :: $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$
where $\langle \text{sint} \equiv \text{signed} \rangle$

abbreviation *ucast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
where $\langle \text{ucast} \equiv \text{unsigned} \rangle$

abbreviation *scast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
where $\langle \text{scast} \equiv \text{signed} \rangle$

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
 $\langle (\text{pcr-word} ==> (=)) (\text{nat} \circ \text{take-bit LENGTH}('a)) (\text{unat} :: 'a::\text{len word} \Rightarrow \text{nat}) \rangle$
 $\langle \text{proof} \rangle$

lemma [*transfer-rule*]:
 $\langle (\text{pcr-word} ==> (=)) (\text{take-bit LENGTH}('a)) (\text{uint} :: 'a::\text{len word} \Rightarrow \text{int}) \rangle$

⟨proof⟩

lemma [transfer-rule]:

⟨(pcr-word ==> (=)) (signed-take-bit (LENGTH('a) - Suc 0)) (sint :: 'a::len word ⇒ int)⟩
 ⟨proof⟩

lemma [transfer-rule]:

⟨(pcr-word ==> pcr-word) (take-bit LENGTH('a)) (ucast :: 'a::len word ⇒ 'b::len word)⟩
 ⟨proof⟩

lemma [transfer-rule]:

⟨(pcr-word ==> pcr-word) (signed-take-bit (LENGTH('a) - Suc 0)) (scast :: 'a::len word ⇒ 'b::len word)⟩
 ⟨proof⟩

end

lemma of-nat-unat [simp]:

⟨of-nat (unat w) = unsigned w⟩
 ⟨proof⟩

lemma of-int-uint [simp]:

⟨of-int (uint w) = unsigned w⟩
 ⟨proof⟩

lemma of-int-sint [simp]:

⟨of-int (sint a) = signed a⟩
 ⟨proof⟩

lemma nat-uint-eq [simp]:

⟨nat (uint w) = unat w⟩
 ⟨proof⟩

lemma nat-of-natural-unsigned-eq [simp]:

⟨nat-of-natural (unsigned w) = unat w⟩
 ⟨proof⟩

lemma int-of-integer-unsigned-eq [simp]:

⟨int-of-integer (unsigned w) = uint w⟩
 ⟨proof⟩

lemma int-of-integer-signed-eq [simp]:

⟨int-of-integer (signed w) = sint w⟩
 ⟨proof⟩

lemma sgn-uint-eq [simp]:

⟨sgn (uint w) = of-bool (w ≠ 0)⟩

⟨proof⟩

Aliases only for code generation

context

begin

qualified lift-definition *of-int* :: ⟨*int* ⇒ 'a::len word⟩

is ⟨*take-bit LENGTH('a)*⟩ ⟨proof⟩ **lift-definition** *of-nat* :: ⟨*nat* ⇒ 'a::len word⟩

is ⟨*int* ○ *take-bit LENGTH('a)*⟩ ⟨proof⟩ **lift-definition** *the-nat* :: ⟨'a::len word ⇒ *nat*⟩

is ⟨*nat* ○ *take-bit LENGTH('a)*⟩ ⟨proof⟩ **lift-definition** *the-signed-int* :: ⟨'a::len word ⇒ *int*⟩

is ⟨*signed-take-bit (LENGTH('a) - Suc 0)*⟩ ⟨proof⟩ **lift-definition** *cast* :: ⟨'a::len word ⇒ 'b::len word⟩

is ⟨*take-bit LENGTH('a)*⟩ ⟨proof⟩ **lift-definition** *signed-cast* :: ⟨'a::len word ⇒ 'b::len word⟩

is ⟨*signed-take-bit (LENGTH('a) - Suc 0)*⟩ ⟨proof⟩

end

lemma [*code-abbrev, simp*]:

⟨*Word.the-int* = *uint*⟩

⟨proof⟩

lemma [*code*]:

⟨*Word.the-int* (*Word.of-int k* :: 'a::len word) = *take-bit LENGTH('a) k*⟩

⟨proof⟩

lemma [*code-abbrev, simp*]:

⟨*Word.of-int* = *word-of-int*⟩

⟨proof⟩

lemma [*code*]:

⟨*Word.the-int* (*Word.of-nat n* :: 'a::len word) = *take-bit LENGTH('a) (int n)*⟩

⟨proof⟩

lemma [*code-abbrev, simp*]:

⟨*Word.of-nat* = *word-of-nat*⟩

⟨proof⟩

lemma [*code*]:

⟨*Word.the-nat w* = *nat (Word.the-int w)*⟩

⟨proof⟩

lemma [*code-abbrev, simp*]:

⟨*Word.the-nat* = *unat*⟩

⟨proof⟩

lemma [*code*]:

```

  ⟨ Word.the-signed-int w = (let k = Word.the-int w
    in if bit k (LENGTH('a) - Suc 0) then k + push-bit LENGTH('a) (- 1) else
    k) ⟩
  for w :: ⟨'a::len word⟩
  ⟨proof⟩

```

```

lemma [code-abbrev, simp]:
  ⟨ Word.the-signed-int = sint ⟩
  ⟨proof⟩

```

```

lemma [code]:
  ⟨ Word.the-int (Word.cast w :: 'b::len word) = take-bit LENGTH('b) (Word.the-int
    w) ⟩
  for w :: ⟨'a::len word⟩
  ⟨proof⟩

```

```

lemma [code-abbrev, simp]:
  ⟨ Word.cast = ucast ⟩
  ⟨proof⟩

```

```

lemma [code]:
  ⟨ Word.the-int (Word.signed-cast w :: 'b::len word) = take-bit LENGTH('b) (Word.the-signed-int
    w) ⟩
  for w :: ⟨'a::len word⟩
  ⟨proof⟩

```

```

lemma [code-abbrev, simp]:
  ⟨ Word.signed-cast = scast ⟩
  ⟨proof⟩

```

```

lemma [code]:
  ⟨ unsigned w = of-nat (nat (Word.the-int w)) ⟩
  ⟨proof⟩

```

```

lemma [code]:
  ⟨ signed w = of-int (Word.the-signed-int w) ⟩
  ⟨proof⟩

```

107.3 Elementary case distinctions

```

lemma word-length-one [case-names zero minus-one length-beyond]:
  fixes w :: ⟨'a::len word⟩
  obtains (zero) ⟨LENGTH('a) = Suc 0⟩ ⟨w = 0⟩
  | (minus-one) ⟨LENGTH('a) = Suc 0⟩ ⟨w = - 1⟩
  | (length-beyond) ⟨2 ≤ LENGTH('a)⟩
  ⟨proof⟩

```

107.3.1 Basic ordering

```

instantiation word :: (len) linorder

```

begin

lift-definition *less-eq-word* :: 'a word \Rightarrow 'a word \Rightarrow bool
is $\lambda a b. \text{take-bit LENGTH('a)} a \leq \text{take-bit LENGTH('a)} b$
 $\langle \text{proof} \rangle$

lift-definition *less-word* :: 'a word \Rightarrow 'a word \Rightarrow bool
is $\lambda a b. \text{take-bit LENGTH('a)} a < \text{take-bit LENGTH('a)} b$
 $\langle \text{proof} \rangle$

instance
 $\langle \text{proof} \rangle$

end

interpretation *word-order*: *ordering-top* $\langle (\leq) \rangle \langle (<) \rangle \langle - 1 :: 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

interpretation *word-coorder*: *ordering-top* $\langle (\geq) \rangle \langle (>) \rangle \langle 0 :: 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-of-nat-less-eq-iff*:
 $\langle \text{word-of-nat } m \leq (\text{word-of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a)} m$
 $\leq \text{take-bit LENGTH('a)} n \rangle$
 $\langle \text{proof} \rangle$

lemma *word-of-int-less-eq-iff*:
 $\langle \text{word-of-int } k \leq (\text{word-of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a)} k \leq$
 $\text{take-bit LENGTH('a)} l \rangle$
 $\langle \text{proof} \rangle$

lemma *word-of-nat-less-iff*:
 $\langle \text{word-of-nat } m < (\text{word-of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a)} m$
 $< \text{take-bit LENGTH('a)} n \rangle$
 $\langle \text{proof} \rangle$

lemma *word-of-int-less-iff*:
 $\langle \text{word-of-int } k < (\text{word-of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a)} k <$
 $\text{take-bit LENGTH('a)} l \rangle$
 $\langle \text{proof} \rangle$

lemma *word-le-def* [code]:
 $a \leq b \longleftrightarrow \text{uint } a \leq \text{uint } b$
 $\langle \text{proof} \rangle$

lemma *word-less-def* [code]:
 $a < b \longleftrightarrow \text{uint } a < \text{uint } b$
 $\langle \text{proof} \rangle$

lemma *word-greater-zero-iff*:

$\langle a > 0 \iff a \neq 0 \rangle$ **for** $a :: \langle 'a::len \text{ word} \rangle$

$\langle proof \rangle$

lemma *of-nat-word-less-eq-iff*:

$\langle of\text{-}nat\ m \leq (of\text{-}nat\ n :: 'a::len \text{ word}) \iff take\text{-}bit\ LENGTH('a)\ m \leq take\text{-}bit\ LENGTH('a)\ n \rangle$

$\langle proof \rangle$

lemma *of-nat-word-less-iff*:

$\langle of\text{-}nat\ m < (of\text{-}nat\ n :: 'a::len \text{ word}) \iff take\text{-}bit\ LENGTH('a)\ m < take\text{-}bit\ LENGTH('a)\ n \rangle$

$\langle proof \rangle$

lemma *of-int-word-less-eq-iff*:

$\langle of\text{-}int\ k \leq (of\text{-}int\ l :: 'a::len \text{ word}) \iff take\text{-}bit\ LENGTH('a)\ k \leq take\text{-}bit\ LENGTH('a)\ l \rangle$

$\langle proof \rangle$

lemma *of-int-word-less-iff*:

$\langle of\text{-}int\ k < (of\text{-}int\ l :: 'a::len \text{ word}) \iff take\text{-}bit\ LENGTH('a)\ k < take\text{-}bit\ LENGTH('a)\ l \rangle$

$\langle proof \rangle$

instantiation $word :: (len)\ order\text{-}bot$
begin

lift-definition $bot\text{-}word :: \langle 'a \text{ word} \rangle$

is $0 \langle proof \rangle$

instance

$\langle proof \rangle$

end

lemma *bot-word-eq*:

$\langle bot = (0 :: 'a::len \text{ word}) \rangle$

$\langle proof \rangle$

instantiation $word :: (len)\ order\text{-}top$
begin

lift-definition $top\text{-}word :: \langle 'a \text{ word} \rangle$

is $\leftarrow 1 \langle proof \rangle$

instance

$\langle proof \rangle$

end

lemma *top-word-eq*:
 $\langle \text{top} = (-\ 1 :: 'a::\text{len word}) \rangle$
 $\langle \text{proof} \rangle$

107.4 Enumeration

lemma *inj-on-word-of-nat*:
 $\langle \text{inj-on } (\text{word-of-nat} :: \text{nat} \Rightarrow 'a::\text{len word}) \{0..\<2 \wedge \text{LENGTH}('a)\} \rangle$
 $\langle \text{proof} \rangle$

lemma *UNIV-word-eq-word-of-nat*:
 $\langle (\text{UNIV} :: 'a::\text{len word set}) = \text{word-of-nat } ' \{0..\<2 \wedge \text{LENGTH}('a)\} \rangle \text{ (is } \langle - = ?A \rangle)$
 $\langle \text{proof} \rangle$

instantiation *word* :: (len) enum
begin

definition *enum-word* :: $\langle 'a \text{ word list} \rangle$
where $\langle \text{enum-word} = \text{map word-of-nat } [0..\<2 \wedge \text{LENGTH}('a)] \rangle$

definition *enum-all-word* :: $\langle ('a \text{ word} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$
where $\langle \text{enum-all-word} = \text{All} \rangle$

definition *enum-ex-word* :: $\langle ('a \text{ word} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$
where $\langle \text{enum-ex-word} = \text{Ex} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma [code]:
 $\langle \text{Enum.enum-all } P \longleftrightarrow \text{list-all } P \text{ Enum.enum} \rangle$
 $\langle \text{Enum.enum-ex } P \longleftrightarrow \text{list-ex } P \text{ Enum.enum} \rangle$ **for** $P :: \langle 'a::\text{len word} \Rightarrow \text{bool} \rangle$
 $\langle \text{proof} \rangle$

107.5 Bit-wise operations

The following specification of word division just lifts the pre-existing division on integers named “F-Division” in [2].

instantiation *word* :: (len) semiring-modulo
begin

lift-definition *divide-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda a \ b. \text{take-bit LENGTH}('a) \ a \ \text{div} \ \text{take-bit LENGTH}('a) \ b \rangle$
 $\langle \text{proof} \rangle$

lift-definition *modulo-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda a \ b. \text{take-bit } LENGTH('a) \ a \text{ mod take-bit } LENGTH('a) \ b \rangle$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *unat-div-distrib*:
 $\langle \text{unat } (v \text{ div } w) = \text{unat } v \text{ div unat } w \rangle$
 $\langle \text{proof} \rangle$

lemma *unat-mod-distrib*:
 $\langle \text{unat } (v \text{ mod } w) = \text{unat } v \text{ mod unat } w \rangle$
 $\langle \text{proof} \rangle$

instance *word* :: $(len) \text{ semiring-parity}$
 $\langle \text{proof} \rangle$

lemma *word-bit-induct* [case-names zero even odd]:
 $\langle P \ a \rangle$ **if** *word-zero*: $\langle P \ 0 \rangle$
and *word-even*: $\langle \bigwedge a. P \ a \implies 0 < a \implies a < 2^\wedge (LENGTH('a) - \text{Suc } 0) \implies$
 $P \ (2 * a) \rangle$
and *word-odd*: $\langle \bigwedge a. P \ a \implies a < 2^\wedge (LENGTH('a) - \text{Suc } 0) \implies P \ (1 + 2$
 $* \ a) \rangle$
for P **and** $a :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *bit-word-half-eq*:
 $\langle (\text{of-bool } b + a * 2) \text{ div } 2 = a \rangle$
if $\langle a < 2^\wedge (LENGTH('a) - \text{Suc } 0) \rangle$
for $a :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

lemma *even-mult-exp-div-word-iff*:
 $\langle \text{even } (a * 2^\wedge m \text{ div } 2^\wedge n) \longleftrightarrow \neg ($
 $m \leq n \wedge$
 $n < LENGTH('a) \wedge \text{odd } (a \text{ div } 2^\wedge (n - m))) \rangle$ **for** $a :: 'a::len \text{ word}$
 $\langle \text{proof} \rangle$

instantiation *word* :: $(len) \text{ semiring-bits}$
begin

lift-definition *bit-word* :: $\langle 'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$
is $\langle \lambda k \ n. \ n < LENGTH('a) \wedge \text{bit } k \ n \rangle$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *bit-word-eqI*:

⟨ $a = b$ ⟩ **if** ⟨ $\bigwedge n. n < \text{LENGTH}('a) \implies \text{bit } a \ n \longleftrightarrow \text{bit } b \ n$ ⟩
for $a \ b :: \langle 'a::\text{len word} \rangle$
 ⟨*proof*⟩

lemma *bit-imp-le-length*: ⟨ $n < \text{LENGTH}('a)$ ⟩ **if** ⟨ $\text{bit } w \ n$ ⟩ **for** $w :: \langle 'a::\text{len word} \rangle$
 ⟨*proof*⟩

lemma *not-bit-length* [*simp*]:

⟨ $\neg \text{bit } w \ \text{LENGTH}('a)$ ⟩ **for** $w :: \langle 'a::\text{len word} \rangle$
 ⟨*proof*⟩

lemma *finite-bit-word* [*simp*]:

⟨*finite* { $n. \text{bit } w \ n$ }⟩
for $w :: \langle 'a::\text{len word} \rangle$
 ⟨*proof*⟩

lemma *bit-numeral-word-iff* [*simp*]:

⟨ $\text{bit } (\text{numeral } w :: 'a::\text{len word}) \ n$ ⟩
 ⟷ $n < \text{LENGTH}('a) \wedge \text{bit } (\text{numeral } w :: \text{int}) \ n$ ⟩
 ⟨*proof*⟩

lemma *bit-neg-numeral-word-iff* [*simp*]:

⟨ $\text{bit } (- \text{numeral } w :: 'a::\text{len word}) \ n$ ⟩
 ⟷ $n < \text{LENGTH}('a) \wedge \text{bit } (- \text{numeral } w :: \text{int}) \ n$ ⟩
 ⟨*proof*⟩

instantiation $\text{word} :: (\text{len}) \text{ ring-bit-operations}$
begin

lift-definition *not-word* :: ⟨ $'a \text{ word} \Rightarrow 'a \text{ word}$ ⟩

is *not*
 ⟨*proof*⟩

lift-definition *and-word* :: ⟨ $'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}$ ⟩

is *and*
 ⟨*proof*⟩

lift-definition *or-word* :: ⟨ $'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}$ ⟩

is *or*
 ⟨*proof*⟩

lift-definition *xor-word* :: ⟨ $'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}$ ⟩

is *xor*
 ⟨*proof*⟩

lift-definition *mask-word* :: ⟨ $\text{nat} \Rightarrow 'a \text{ word}$ ⟩

```

is mask
  ⟨proof⟩

lift-definition set-bit-word :: ⟨nat ⇒ 'a word ⇒ 'a word⟩
is set-bit
  ⟨proof⟩

lift-definition unset-bit-word :: ⟨nat ⇒ 'a word ⇒ 'a word⟩
is unset-bit
  ⟨proof⟩

lift-definition flip-bit-word :: ⟨nat ⇒ 'a word ⇒ 'a word⟩
is flip-bit
  ⟨proof⟩

lift-definition push-bit-word :: ⟨nat ⇒ 'a word ⇒ 'a word⟩
is push-bit
  ⟨proof⟩

lift-definition drop-bit-word :: ⟨nat ⇒ 'a word ⇒ 'a word⟩
is ⟨λn. drop-bit n ∘ take-bit LENGTH('a)⟩
  ⟨proof⟩

lift-definition take-bit-word :: ⟨nat ⇒ 'a word ⇒ 'a word⟩
is ⟨λn. take-bit (min LENGTH('a) n)⟩
  ⟨proof⟩

context
  includes bit-operations-syntax
begin

instance ⟨proof⟩

end

end

lemma [code]:
  ⟨push-bit n w = w * 2 ^ n⟩ for w :: ⟨'a::len word⟩
  ⟨proof⟩

lemma [code]:
  ⟨Word.the-int (drop-bit n w) = drop-bit n (Word.the-int w)⟩
  ⟨proof⟩

lemma [code]:
  ⟨Word.the-int (take-bit n w) = (if n < LENGTH('a::len) then take-bit n (Word.the-int
w) else Word.the-int w)⟩
  for w :: ⟨'a::len word⟩

```


⟨proof⟩

lemma [code-abbrev]:
 ⟨push-bit n 1 = (2 :: 'a::len word) \wedge n ⟩
 ⟨proof⟩

context
includes *bit-operations-syntax*
begin

lemma [code]:
 ⟨NOT w = Word.of-int (NOT (Word.the-int w))⟩
for w :: ⟨'a::len word⟩
 ⟨proof⟩

lemma [code]:
 ⟨Word.the-int (v AND w) = Word.the-int v AND Word.the-int w ⟩
 ⟨proof⟩

lemma [code]:
 ⟨Word.the-int (v OR w) = Word.the-int v OR Word.the-int w ⟩
 ⟨proof⟩

lemma [code]:
 ⟨Word.the-int (v XOR w) = Word.the-int v XOR Word.the-int w ⟩
 ⟨proof⟩

lemma [code]:
 ⟨Word.the-int (mask n :: 'a::len word) = mask (min LENGTH('a) n)⟩
 ⟨proof⟩

lemma [code]:
 ⟨set-bit n w = w OR push-bit n 1⟩ **for** w :: ⟨'a::len word⟩
 ⟨proof⟩

lemma [code]:
 ⟨unset-bit n w = w AND NOT (push-bit n 1)⟩ **for** w :: ⟨'a::len word⟩
 ⟨proof⟩

lemma [code]:
 ⟨flip-bit n w = w XOR push-bit n 1⟩ **for** w :: ⟨'a::len word⟩
 ⟨proof⟩

context
includes *lifting-syntax*
begin

lemma *set-bit-word-transfer* [transfer-rule]:
 ⟨((=) ==> pcr-word ==> pcr-word) set-bit set-bit⟩

⟨proof⟩

lemma *unset-bit-word-transfer* [transfer-rule]:

⟨((=) == => pcr-word == => pcr-word) unset-bit unset-bit⟩
 ⟨proof⟩

lemma *flip-bit-word-transfer* [transfer-rule]:

⟨((=) == => pcr-word == => pcr-word) flip-bit flip-bit⟩
 ⟨proof⟩

lemma *signed-take-bit-word-transfer* [transfer-rule]:

⟨((=) == => pcr-word == => pcr-word)
 (λn k. signed-take-bit n (take-bit LENGTH('a::len) k))
 (signed-take-bit :: nat => 'a word => 'a word)⟩
 ⟨proof⟩

end

end

107.6 Conversions including casts

107.6.1 Generic unsigned conversion

context *semiring-bits*

begin

lemma *bit-unsigned-iff* [bit-simps]:

⟨bit (unsigned w) n ⟷ possible-bit TYPE('a) n ∧ bit w n⟩
for w :: ⟨'b::len word⟩
 ⟨proof⟩

end

lemma *possible-bit-word[simp]*:

⟨possible-bit TYPE(('a :: len) word) m ⟷ m < LENGTH('a)⟩
 ⟨proof⟩

context *semiring-bit-operations*

begin

lemma *unsigned-minus-1-eq-mask*:

⟨unsigned (− 1 :: 'b::len word) = mask LENGTH('b)⟩
 ⟨proof⟩

lemma *unsigned-push-bit-eq*:

⟨unsigned (push-bit n w) = take-bit LENGTH('b) (push-bit n (unsigned w))⟩
for w :: ⟨'b::len word⟩
 ⟨proof⟩

lemma *unsigned-take-bit-eq*:

$\langle \text{unsigned } (\text{take-bit } n \ w) = \text{take-bit } n \ (\text{unsigned } w) \rangle$

for $w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

end

context *linordered-euclidean-semiring-bit-operations*

begin

lemma *unsigned-drop-bit-eq*:

$\langle \text{unsigned } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{take-bit } \text{LENGTH}('b) \ (\text{unsigned } w)) \rangle$

for $w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

end

lemma *ucast-drop-bit-eq*:

$\langle \text{ucast } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{ucast } w :: 'b::\text{len word}) \rangle$

if $\langle \text{LENGTH}('a) \leq \text{LENGTH}('b) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$

$\langle \text{proof} \rangle$

context *semiring-bit-operations*

begin

context

includes *bit-operations-syntax*

begin

lemma *unsigned-and-eq*:

$\langle \text{unsigned } (v \ \text{AND} \ w) = \text{unsigned } v \ \text{AND} \ \text{unsigned } w \rangle$

for $v \ w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

lemma *unsigned-or-eq*:

$\langle \text{unsigned } (v \ \text{OR} \ w) = \text{unsigned } v \ \text{OR} \ \text{unsigned } w \rangle$

for $v \ w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

lemma *unsigned-xor-eq*:

$\langle \text{unsigned } (v \ \text{XOR} \ w) = \text{unsigned } v \ \text{XOR} \ \text{unsigned } w \rangle$

for $v \ w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

end

end

context *ring-bit-operations*

begin

context

includes *bit-operations-syntax*

begin

lemma *unsigned-not-eq*:

$\langle \text{unsigned } (\text{NOT } w) = \text{take-bit LENGTH('b)} (\text{NOT } (\text{unsigned } w)) \rangle$

for $w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

end

end

context *linordered-euclidean-semiring*

begin

lemma *unsigned-greater-eq* [*simp*]:

$\langle 0 \leq \text{unsigned } w \rangle$ **for** $w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

lemma *unsigned-less* [*simp*]:

$\langle \text{unsigned } w < 2^{\text{LENGTH('b)}} \rangle$ **for** $w :: \langle 'b::\text{len word} \rangle$

$\langle \text{proof} \rangle$

end

context *linordered-semidom*

begin

lemma *word-less-eq-iff-unsigned*:

$a \leq b \longleftrightarrow \text{unsigned } a \leq \text{unsigned } b$

$\langle \text{proof} \rangle$

lemma *word-less-iff-unsigned*:

$a < b \longleftrightarrow \text{unsigned } a < \text{unsigned } b$

$\langle \text{proof} \rangle$

end

107.6.2 Generic signed conversion

context *ring-bit-operations*

begin

lemma *bit-signed-iff* [*bit-simps*]:

$\langle \text{bit } (\text{signed } w) \text{ } n \longleftrightarrow \text{possible-bit TYPE('a)} \text{ } n \wedge \text{bit } w \text{ } (\text{min } (\text{LENGTH('b)} - \text{Suc } 0) \text{ } n) \rangle$

for $w :: \langle 'b :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-push-bit-eq*:

$\langle \text{signed } (\text{push-bit } n \ w) = \text{signed-take-bit } (\text{LENGTH}('b) - \text{Suc } 0) (\text{push-bit } n$
 $(\text{signed } w :: 'a)) \rangle$
for $w :: \langle 'b :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-eq*:

$\langle \text{signed } (\text{take-bit } n \ w) = (\text{if } n < \text{LENGTH}('b) \text{ then take-bit } n (\text{signed } w) \text{ else}$
 $\text{signed } w) \rangle$
for $w :: \langle 'b :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

context

includes *bit-operations-syntax*

begin

lemma *signed-not-eq*:

$\langle \text{signed } (\text{NOT } w) = \text{signed-take-bit } \text{LENGTH}('b) (\text{NOT } (\text{signed } w)) \rangle$
for $w :: \langle 'b :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-and-eq*:

$\langle \text{signed } (v \ \text{AND } w) = \text{signed } v \ \text{AND } \text{signed } w \rangle$
for $v \ w :: \langle 'b :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-or-eq*:

$\langle \text{signed } (v \ \text{OR } w) = \text{signed } v \ \text{OR } \text{signed } w \rangle$
for $v \ w :: \langle 'b :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-xor-eq*:

$\langle \text{signed } (v \ \text{XOR } w) = \text{signed } v \ \text{XOR } \text{signed } w \rangle$
for $v \ w :: \langle 'b :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

end

end

107.6.3 More

lemma *sint-greater-eq*:

$\langle - (2 \wedge (\text{LENGTH}('a) - \text{Suc } 0)) \leq \text{sint } w \rangle$ **for** $w :: \langle 'a :: \text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *sint-less*:

$\langle \text{sint } w < 2^{\wedge}(\text{LENGTH}('a) - \text{Suc } 0) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-div-distrib*:

$\langle \text{uint } (v \text{ div } w) = \text{uint } v \text{ div uint } w \rangle$
 $\langle \text{proof} \rangle$

lemma *unat-drop-bit-eq*:

$\langle \text{unat } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{unat } w) \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-mod-distrib*:

$\langle \text{uint } (v \text{ mod } w) = \text{uint } v \text{ mod uint } w \rangle$
 $\langle \text{proof} \rangle$

context *semiring-bit-operations*

begin

lemma *unsigned-ucast-eq*:

$\langle \text{unsigned } (\text{ucast } w :: 'c::\text{len word}) = \text{take-bit } \text{LENGTH}('c) \ (\text{unsigned } w) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-bit-operations*

begin

lemma *signed-ucast-eq*:

$\langle \text{signed } (\text{ucast } w :: 'c::\text{len word}) = \text{signed-take-bit } (\text{LENGTH}('c) - \text{Suc } 0) \ (\text{signed } w) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-scast-eq*:

$\langle \text{signed } (\text{scast } w :: 'c::\text{len word}) = \text{signed-take-bit } (\text{LENGTH}('c) - \text{Suc } 0) \ (\text{signed } w) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

end

lemma *uint-nonnegative*: $0 \leq \text{uint } w$

$\langle \text{proof} \rangle$

lemma *uint-bounded*: $\text{uint } w < 2^{\wedge} \text{LENGTH}('a)$

for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *uint-idem*: $\text{uint } w \bmod 2^{\text{LENGTH}('a)} = \text{uint } w$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-uint-eqI*: $\text{uint } a = \text{uint } b \implies a = b$
 $\langle \text{proof} \rangle$

lemma *word-uint-eq-iff*: $a = b \iff \text{uint } a = \text{uint } b$
 $\langle \text{proof} \rangle$

lemma *uint-word-of-int-eq*:
 $\langle \text{uint } (\text{word-of-int } k :: 'a::\text{len word}) = \text{take-bit } \text{LENGTH}('a) \ k \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-word-of-int*: $\text{uint } (\text{word-of-int } k :: 'a::\text{len word}) = k \bmod 2^{\text{LENGTH}('a)}$
 $\langle \text{proof} \rangle$

lemma *word-of-int-uint*: $\text{word-of-int } (\text{uint } w) = w$
 $\langle \text{proof} \rangle$

lemma *word-div-def* [code]:
 $a \text{ div } b = \text{word-of-int } (\text{uint } a \text{ div } \text{uint } b)$
 $\langle \text{proof} \rangle$

lemma *word-mod-def* [code]:
 $a \bmod b = \text{word-of-int } (\text{uint } a \bmod \text{uint } b)$
 $\langle \text{proof} \rangle$

lemma *split-word-all*: $(\bigwedge x::'a::\text{len word}. \text{PROP } P \ x) \equiv (\bigwedge x. \text{PROP } P \ (\text{word-of-int } x))$
 $\langle \text{proof} \rangle$

lemma *sint-uint*:
 $\langle \text{sint } w = \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \ (\text{uint } w) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *unat-eq-nat-uint*:
 $\langle \text{unat } w = \text{nat } (\text{uint } w) \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-eq*:
 $\langle \text{ucast } w = \text{word-of-int } (\text{uint } w) \rangle$
 $\langle \text{proof} \rangle$

lemma *scast-eq*:
 $\langle \text{scast } w = \text{word-of-int } (\text{sint } w) \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-0-eq*:

$\langle \text{uint } 0 = 0 \rangle$

$\langle \text{proof} \rangle$

lemma *uint-1-eq*:

$\langle \text{uint } 1 = 1 \rangle$

$\langle \text{proof} \rangle$

lemma *word-m1-wi*: $- 1 = \text{word-of-int } (- 1)$

$\langle \text{proof} \rangle$

lemma *uint-0-iff*: $\text{uint } x = 0 \longleftrightarrow x = 0$

$\langle \text{proof} \rangle$

lemma *unat-0-iff*: $\text{unat } x = 0 \longleftrightarrow x = 0$

$\langle \text{proof} \rangle$

lemma *unat-0*: $\text{unat } 0 = 0$

$\langle \text{proof} \rangle$

lemma *unat-gt-0*: $0 < \text{unat } x \longleftrightarrow x \neq 0$

$\langle \text{proof} \rangle$

lemma *ucast-0*: $\text{ucast } 0 = 0$

$\langle \text{proof} \rangle$

lemma *sint-0*: $\text{sint } 0 = 0$

$\langle \text{proof} \rangle$

lemma *scast-0*: $\text{scast } 0 = 0$

$\langle \text{proof} \rangle$

lemma *sint-n1*: $\text{sint } (- 1) = - 1$

$\langle \text{proof} \rangle$

lemma *scast-n1*: $\text{scast } (- 1) = - 1$

$\langle \text{proof} \rangle$

lemma *uint-1*: $\text{uint } (1 :: 'a :: \text{len word}) = 1$

$\langle \text{proof} \rangle$

lemma *unat-1*: $\text{unat } (1 :: 'a :: \text{len word}) = 1$

$\langle \text{proof} \rangle$

lemma *ucast-1*: $\text{ucast } (1 :: 'a :: \text{len word}) = 1$

$\langle \text{proof} \rangle$

instantiation *word* :: $(\text{len}) \text{ size}$

begin

lift-definition *size-word* :: $\langle 'a \text{ word} \Rightarrow \text{nat} \rangle$
is $\langle \lambda-. \text{LENGTH}('a) \rangle \langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *word-size* [code]:
 $\langle \text{size } w = \text{LENGTH}('a) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-size-gt-0* [iff]: $0 < \text{size } w$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemmas *lens-gt-0* = *word-size-gt-0* *len-gt-0*

lemma *lens-not-0* [iff]:
 $\langle \text{size } w \neq 0 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lift-definition *source-size* :: $\langle ('a::\text{len word} \Rightarrow 'b) \Rightarrow \text{nat} \rangle$
is $\langle \lambda-. \text{LENGTH}('a) \rangle \langle \text{proof} \rangle$

lift-definition *target-size* :: $\langle ('a \Rightarrow 'b::\text{len word}) \Rightarrow \text{nat} \rangle$
is $\langle \lambda-. \text{LENGTH}('b) \rangle \langle \text{proof} \rangle$

lift-definition *is-up* :: $\langle ('a::\text{len word} \Rightarrow 'b::\text{len word}) \Rightarrow \text{bool} \rangle$
is $\langle \lambda-. \text{LENGTH}('a) \leq \text{LENGTH}('b) \rangle \langle \text{proof} \rangle$

lift-definition *is-down* :: $\langle ('a::\text{len word} \Rightarrow 'b::\text{len word}) \Rightarrow \text{bool} \rangle$
is $\langle \lambda-. \text{LENGTH}('a) \geq \text{LENGTH}('b) \rangle \langle \text{proof} \rangle$

lemma *is-up-eq*:
 $\langle \text{is-up } f \longleftrightarrow \text{source-size } f \leq \text{target-size } f \rangle$
for $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *is-down-eq*:
 $\langle \text{is-down } f \longleftrightarrow \text{target-size } f \leq \text{source-size } f \rangle$
for $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lift-definition *word-int-case* :: $\langle (\text{int} \Rightarrow 'b) \Rightarrow 'a::\text{len word} \Rightarrow 'b \rangle$
is $\langle \lambda f. f \circ \text{take-bit LENGTH}('a) \rangle \langle \text{proof} \rangle$

lemma *word-int-case-eq-uint* [code]:

$\langle \text{word-int-case } f \ w = f \ (\text{uint } w) \rangle$
 $\langle \text{proof} \rangle$

translations

$\text{case } x \text{ of } XCONST \text{ of-int } y \Rightarrow b \Rightarrow CONST \text{ word-int-case } (\lambda y. \ b) \ x$
 $\text{case } x \text{ of } (XCONST \text{ of-int } :: 'a) \ y \Rightarrow b \rightarrow CONST \text{ word-int-case } (\lambda y. \ b) \ x$

107.7 Arithmetic operations

lemma *div-word-self*:

$\langle w \ \text{div} \ w = 1 \rangle \text{ if } \langle w \neq 0 \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *mod-word-self* [simp]:

$\langle w \ \text{mod} \ w = 0 \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *div-word-less*:

$\langle w \ \text{div} \ v = 0 \rangle \text{ if } \langle w < v \rangle \text{ for } w \ v :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *mod-word-less*:

$\langle w \ \text{mod} \ v = w \rangle \text{ if } \langle w < v \rangle \text{ for } w \ v :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *div-word-one* [simp]:

$\langle 1 \ \text{div} \ w = \text{of-bool } (w = 1) \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *mod-word-one* [simp]:

$\langle 1 \ \text{mod} \ w = 1 - w * \text{of-bool } (w = 1) \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *div-word-by-minus-1-eq* [simp]:

$\langle w \ \text{div} \ -1 = \text{of-bool } (w = -1) \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *mod-word-by-minus-1-eq* [simp]:

$\langle w \ \text{mod} \ -1 = w * \text{of-bool } (w < -1) \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

Legacy theorems:

lemma *word-add-def* [code]:

$a + b = \text{word-of-int } (\text{uint } a + \text{uint } b)$
 $\langle \text{proof} \rangle$

lemma *word-sub-wi* [code]:

$a - b = \text{word-of-int } (\text{uint } a - \text{uint } b)$
 $\langle \text{proof} \rangle$

lemma *word-mult-def* [code]:
 $a * b = \text{word-of-int } (\text{uint } a * \text{uint } b)$
 ⟨proof⟩

lemma *word-minus-def* [code]:
 $- a = \text{word-of-int } (- \text{uint } a)$
 ⟨proof⟩

lemma *word-0-wi*:
 $0 = \text{word-of-int } 0$
 ⟨proof⟩

lemma *word-1-wi*:
 $1 = \text{word-of-int } 1$
 ⟨proof⟩

lift-definition *word-succ* :: $'a::\text{len word} \Rightarrow 'a \text{ word is } \lambda x. x + 1$
 ⟨proof⟩

lift-definition *word-pred* :: $'a::\text{len word} \Rightarrow 'a \text{ word is } \lambda x. x - 1$
 ⟨proof⟩

lemma *word-succ-alt* [code]:
 $\text{word-succ } a = \text{word-of-int } (\text{uint } a + 1)$
 ⟨proof⟩

lemma *word-pred-alt* [code]:
 $\text{word-pred } a = \text{word-of-int } (\text{uint } a - 1)$
 ⟨proof⟩

lemmas *word-arith-wis* =
word-add-def word-sub-wi word-mult-def
word-minus-def word-succ-alt word-pred-alt
word-0-wi word-1-wi

lemma *wi-homs*:
shows *wi-hom-add*: $\text{word-of-int } a + \text{word-of-int } b = \text{word-of-int } (a + b)$
and *wi-hom-sub*: $\text{word-of-int } a - \text{word-of-int } b = \text{word-of-int } (a - b)$
and *wi-hom-mult*: $\text{word-of-int } a * \text{word-of-int } b = \text{word-of-int } (a * b)$
and *wi-hom-neg*: $- \text{word-of-int } a = \text{word-of-int } (- a)$
and *wi-hom-succ*: $\text{word-succ } (\text{word-of-int } a) = \text{word-of-int } (a + 1)$
and *wi-hom-pred*: $\text{word-pred } (\text{word-of-int } a) = \text{word-of-int } (a - 1)$
 ⟨proof⟩

lemmas *wi-hom-syms* = *wi-homs* [symmetric]

lemmas *word-of-int-homs* = *wi-homs word-0-wi word-1-wi*

lemmas *word-of-int-hom-syms* = *word-of-int-homs* [symmetric]

lemma *double-eq-zero-iff*:
 $\langle 2 * a = 0 \longleftrightarrow a = 0 \vee a = 2 \wedge (LENGTH('a) - Suc\ 0) \rangle$
for $a :: 'a::len\ word$
 $\langle proof \rangle$

107.8 Ordering

instance *word* :: (len) *wellorder*
 $\langle proof \rangle$

lemma *word-m1-ge* [*simp*]:
 $word-pred\ 0 \geq y$
 $\langle proof \rangle$

lemma *word-less-alt*:
 $a < b \longleftrightarrow uint\ a < uint\ b$
 $\langle proof \rangle$

lemma *word-zero-le* [*simp*]:
 $0 \leq y$ **for** $y :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-n1-ge* [*simp*]:
 $y \leq -\ 1$ **for** $y :: 'a::len\ word$
 $\langle proof \rangle$

lemmas *word-not-simps* [*simp*] =
 $word-zero-le\ [THEN\ leD]\ word-m1-ge\ [THEN\ leD]\ word-n1-ge\ [THEN\ leD]$

lemma *word-gt-0*:
 $0 < y \longleftrightarrow 0 \neq y$
for $y :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-gt-0-no* [*simp*]:
 $\langle (0 :: 'a::len\ word) < numeral\ y \longleftrightarrow (0 :: 'a::len\ word) \neq numeral\ y \rangle$
 $\langle proof \rangle$

lemma *word-le-nat-alt*:
 $a \leq b \longleftrightarrow unat\ a \leq unat\ b$
 $\langle proof \rangle$

lemma *word-less-nat-alt*:
 $a < b \longleftrightarrow unat\ a < unat\ b$
 $\langle proof \rangle$

lemmas *unat-mono* = *word-less-nat-alt* [*THEN* *iffD1*]

lemma *wi-less*:

$(\text{word-of-int } n < (\text{word-of-int } m :: 'a::\text{len word})) =$
 $(n \bmod 2 \wedge \text{LENGTH}('a) < m \bmod 2 \wedge \text{LENGTH}('a))$
 $\langle \text{proof} \rangle$

lemma *wi-le*:

$(\text{word-of-int } n \leq (\text{word-of-int } m :: 'a::\text{len word})) =$
 $(n \bmod 2 \wedge \text{LENGTH}('a) \leq m \bmod 2 \wedge \text{LENGTH}('a))$
 $\langle \text{proof} \rangle$

lift-definition *word-sle* :: $\langle 'a::\text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$

is $\langle \lambda k l. \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) k \leq \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) l \rangle$
 $\langle \text{proof} \rangle$

lift-definition *word-sless* :: $\langle 'a::\text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$

is $\langle \lambda k l. \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) k < \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) l \rangle$
 $\langle \text{proof} \rangle$

notation

word-sle $\langle '(\leq s') \rangle$ **and**
word-sle $\langle (\langle \text{notation} = \langle \text{infix } \leq s \rangle \rangle - / \leq s -) \rangle$ [51, 51] 50) **and**
word-sless $\langle '(< s') \rangle$ **and**
word-sless $\langle (\langle \text{notation} = \langle \text{infix } < s \rangle \rangle - / < s -) \rangle$ [51, 51] 50)

notation (*input*)

word-sle $\langle (\langle \text{notation} = \langle \text{infix } \leq s \rangle \rangle - / \leq s -) \rangle$ [51, 51] 50)

lemma *word-sle-eq* [code]:

$\langle a \leq s b \longleftrightarrow \text{sint } a \leq \text{sint } b \rangle$
 $\langle \text{proof} \rangle$

lemma *word-sless-alt* [code]:

$a < s b \longleftrightarrow \text{sint } a < \text{sint } b$
 $\langle \text{proof} \rangle$

lemma *signed-ordering*: $\langle \text{ordering word-sle word-sless} \rangle$

$\langle \text{proof} \rangle$

lemma *signed-linorder*: $\langle \text{class.linorder word-sle word-sless} \rangle$

$\langle \text{proof} \rangle$

interpretation *signed*: *linorder word-sle word-sless*

$\langle \text{proof} \rangle$

lemma *word-sless-eq*:

$\langle x < s y \longleftrightarrow x \leq s y \wedge x \neq y \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-1-sless-0* [*simp*]:

$\langle -1 <_s 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-0-sless-minus-1* [*simp*]:

$\langle \neg 0 <_s -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *minus-1-sless-eq-0* [*simp*]:

$\langle -1 \leq_s 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *not-0-sless-eq-minus-1* [*simp*]:

$\langle \neg 0 \leq_s -1 \rangle$
 $\langle \text{proof} \rangle$

107.9 Bit-wise operations

context

includes *bit-operations-syntax*

begin

lemma *take-bit-word-eq-self*:

$\langle \text{take-bit } n \ w = w \rangle$ **if** $\langle \text{LENGTH}(a) \leq n \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-length-eq* [*simp*]:

$\langle \text{take-bit } \text{LENGTH}(a) \ w = w \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-of-int-iff*:

$\langle \text{bit } (\text{word-of-int } k :: 'a::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}(a) \wedge \text{bit } k \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-uint-iff*:

$\langle \text{bit } (\text{uint } w) \ n \longleftrightarrow n < \text{LENGTH}(a) \wedge \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-sint-iff*:

$\langle \text{bit } (\text{sint } w) \ n \longleftrightarrow n \geq \text{LENGTH}(a) \wedge \text{bit } w \ (\text{LENGTH}(a) - 1) \vee \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-ucast-iff*:

$\langle \text{bit } (\text{ucast } w :: 'b::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}(a) \wedge n < \text{LENGTH}(b) \wedge \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$

⟨proof⟩

lemma *bit-word-scast-iff*:

⟨bit (scast w :: 'b::len word) n \longleftrightarrow
 $n < \text{LENGTH}('b) \wedge (\text{bit } w \ n \vee \text{LENGTH}('a) \leq n \wedge \text{bit } w \ (\text{LENGTH}('a) - \text{Suc } 0))$ ⟩
for w :: ⟨'a::len word⟩
 ⟨proof⟩

lemma *bit-word-iff-drop-bit-and* [code]:

⟨bit a n \longleftrightarrow drop-bit n a AND 1 = 1⟩ **for** a :: ⟨'a::len word⟩
 ⟨proof⟩

lemma

word-not-def: NOT (a::'a::len word) = word-of-int (NOT (uint a))
and word-and-def: (a::'a word) AND b = word-of-int (uint a AND uint b)
and word-or-def: (a::'a word) OR b = word-of-int (uint a OR uint b)
and word-xor-def: (a::'a word) XOR b = word-of-int (uint a XOR uint b)
 ⟨proof⟩

definition *even-word* :: ⟨'a::len word \Rightarrow bool⟩

where [code-abbrev]: ⟨even-word = even⟩

lemma *even-word-iff* [code]:

⟨even-word a \longleftrightarrow a AND 1 = 0⟩
 ⟨proof⟩

lemma *map-bit-range-eq-if-take-bit-eq*:

⟨map (bit k) [0.. n] = map (bit l) [0.. n]⟩
if ⟨take-bit n k = take-bit n l⟩ **for** k l :: int
 ⟨proof⟩

lemma

take-bit-word-Bit0-eq [simp]: ⟨take-bit (numeral n) (numeral (num.Bit0 m)) ::
 'a::len word⟩
 = 2 * take-bit (pred-numeral n) (numeral m)⟩ (**is** ?P)
and take-bit-word-Bit1-eq [simp]: ⟨take-bit (numeral n) (numeral (num.Bit1 m))
 :: 'a::len word⟩
 = 1 + 2 * take-bit (pred-numeral n) (numeral m)⟩ (**is** ?Q)
and take-bit-word-minus-Bit0-eq [simp]: ⟨take-bit (numeral n) (− numeral (num.Bit0
 m)) :: 'a::len word⟩
 = 2 * take-bit (pred-numeral n) (− numeral m)⟩ (**is** ?R)
and take-bit-word-minus-Bit1-eq [simp]: ⟨take-bit (numeral n) (− numeral (num.Bit1
 m)) :: 'a::len word⟩
 = 1 + 2 * take-bit (pred-numeral n) (− numeral (Num.inc m))⟩ (**is** ?S)
 ⟨proof⟩

107.10 More shift operations

lift-definition *signed-drop-bit* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \lambda n. \text{drop-bit } n \circ \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-signed-drop-bit-iff* [bit-simps]:
 $\langle \text{bit } (\text{signed-drop-bit } m \ w) \ n \longleftrightarrow \text{bit } w \ (\text{if } \text{LENGTH}('a) - m \leq n \wedge n < \text{LENGTH}('a) \text{ then } \text{LENGTH}('a) - 1 \text{ else } m + n) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma [code]:
 $\langle \text{Word.the-int } (\text{signed-drop-bit } n \ w) = \text{take-bit } \text{LENGTH}('a) \ (\text{drop-bit } n \ (\text{Word.the-signed-int } w)) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-of-0* [simp]:
 $\langle \text{signed-drop-bit } n \ 0 = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-of-minus-1* [simp]:
 $\langle \text{signed-drop-bit } n \ (-1) = -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-signed-drop-bit* [simp]:
 $\langle \text{signed-drop-bit } m \ (\text{signed-drop-bit } n \ w) = \text{signed-drop-bit } (m + n) \ w \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-0* [simp]:
 $\langle \text{signed-drop-bit } 0 \ w = w \rangle$
 $\langle \text{proof} \rangle$

lemma *sint-signed-drop-bit-eq*:
 $\langle \text{sint } (\text{signed-drop-bit } n \ w) = \text{drop-bit } n \ (\text{sint } w) \rangle$
 $\langle \text{proof} \rangle$

107.11 Single-bit operations

lemma *set-bit-eq-idem-iff*:
 $\langle \text{set-bit } n \ w = w \longleftrightarrow \text{bit } w \ n \vee n \geq \text{LENGTH}('a) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *unset-bit-eq-idem-iff*:
 $\langle \text{unset-bit } n \ w = w \longleftrightarrow \text{bit } w \ n \longrightarrow n \geq \text{LENGTH}('a) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *flip-bit-eq-idem-iff*:
 $\langle \text{flip-bit } n \ w = w \longleftrightarrow n \geq \text{LENGTH}('a) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

107.12 Rotation

lift-definition *word-rotr* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \lambda n \ k. \text{concat-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a))$
 $(\text{drop-bit } (n \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (n \bmod \text{LENGTH}('a)) \ k) \rangle$
 $\langle \text{proof} \rangle$

lift-definition *word-rotl* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \lambda n \ k. \text{concat-bit } (n \bmod \text{LENGTH}('a))$
 $(\text{drop-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) \ k) \rangle$
 $\langle \text{proof} \rangle$

lift-definition *word-roti* :: $\langle \text{int} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \lambda r \ k. \text{concat-bit } (\text{LENGTH}('a) - \text{nat } (r \bmod \text{int } \text{LENGTH}('a)))$
 $(\text{drop-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) \ k) \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rotl-eq-word-rotr* [code]:
 $\langle \text{word-rotl } n = (\text{word-rotr } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) :: 'a::\text{len word}$
 $\Rightarrow 'a \ \text{word}) \rangle$
 $\langle \text{proof} \rangle$

lemma *word-roti-eq-word-rotr-word-rotl* [code]:
 $\langle \text{word-roti } i \ w =$
 $(\text{if } i \geq 0 \text{ then } \text{word-rotr } (\text{nat } i) \ w \text{ else } \text{word-rotl } (\text{nat } (- \ i)) \ w) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-rotr-iff* [bit-simps]:
 $\langle \text{bit } (\text{word-rotr } m \ w) \ n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } w \ ((n + m) \bmod \text{LENGTH}('a)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-rotl-iff* [bit-simps]:
 $\langle \text{bit } (\text{word-rotl } m \ w) \ n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } w \ ((n + (\text{LENGTH}('a) - m \bmod \text{LENGTH}('a))) \bmod$
 $\text{LENGTH}('a)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-word-roti-iff* [*bit-simps*]:

⟨*bit* (*word-roti* *k w*) *n* \longleftrightarrow
 $n < \text{LENGTH}('a) \wedge \text{bit } w \text{ (nat ((int } n + k) \text{ mod int LENGTH('a)))}$ ⟩
for *w* :: ⟨*a*::len word⟩
 ⟨*proof*⟩

lemma *uint-word-rotr-eq*:

⟨*uint* (*word-rotr* *n w*) = *concat-bit* (*LENGTH*('a) − *n* mod *LENGTH*('a))

 (*drop-bit* (*n* mod *LENGTH*('a)) (*uint w*))

 (*uint* (*take-bit* (*n* mod *LENGTH*('a)) *w*))⟩

for *w* :: ⟨*a*::len word⟩
 ⟨*proof*⟩

lemma [*code*]:

⟨*Word.the-int* (*word-rotr* *n w*) = *concat-bit* (*LENGTH*('a) − *n* mod *LENGTH*('a))

 (*drop-bit* (*n* mod *LENGTH*('a)) (*Word.the-int w*))

 (*Word.the-int* (*take-bit* (*n* mod *LENGTH*('a)) *w*))⟩

for *w* :: ⟨*a*::len word⟩
 ⟨*proof*⟩

107.13 Split and cat operations

lift-definition *word-cat* :: ⟨*a*::len word \Rightarrow *b*::len word \Rightarrow *c*::len word⟩

is ⟨ $\lambda k l. \text{concat-bit LENGTH}('b) l (\text{take-bit LENGTH}('a) k)$ ⟩

⟨*proof*⟩

lemma *word-cat-eq*:

⟨(*word-cat* *v w* :: *c*::len word) = *push-bit* *LENGTH*('b) (*ucast v*) + *ucast w*⟩

for *v* :: ⟨*a*::len word⟩ **and** *w* :: ⟨*b*::len word⟩
 ⟨*proof*⟩

lemma *word-cat-eq'* [*code*]:

⟨*word-cat* *a b* = *word-of-int* (*concat-bit* *LENGTH*('b) (*uint b*) (*uint a*))⟩

for *a* :: ⟨*a*::len word⟩ **and** *b* :: ⟨*b*::len word⟩
 ⟨*proof*⟩

lemma *bit-word-cat-iff* [*bit-simps*]:

⟨*bit* (*word-cat* *v w* :: *c*::len word) *n* \longleftrightarrow $n < \text{LENGTH}('c) \wedge (\text{if } n < \text{LENGTH}('b)$

$\text{then bit } w \text{ } n \text{ else bit } v \text{ (} n - \text{LENGTH}('b) \text{))}$ ⟩

for *v* :: ⟨*a*::len word⟩ **and** *w* :: ⟨*b*::len word⟩
 ⟨*proof*⟩

definition *word-split* :: ⟨*a*::len word \Rightarrow *b*::len word \times *c*::len word⟩

where *word-split w* =

 (*ucast* (*drop-bit* *LENGTH*('c) *w*) :: *b*::len word, *ucast w* :: *c*::len word)⟩

definition *word-rcat* :: ⟨*a*::len word list \Rightarrow *b*::len word⟩

where *word-rcat* = *word-of-int* \circ *horner-sum uint* ($2 \wedge \text{LENGTH}('a)$) \circ *rev*⟩

107.14 More on conversions**lemma** *int-word-sint*:
$$\langle \text{sint} (\text{word-of-int } x :: 'a::\text{len word}) = (x + 2^{\wedge} (\text{LENGTH}('a) - 1)) \bmod 2^{\wedge} \text{LENGTH}('a) - 2^{\wedge} (\text{LENGTH}('a) - 1) \rangle$$

<proof>

lemma *sint-sbintrunc'*: $\text{sint} (\text{word-of-int bin} :: 'a \text{ word}) = \text{signed-take-bit} (\text{LENGTH}('a::\text{len}) - 1) \text{ bin}$ *<proof>***lemma** *uint-sint*: $\text{uint } w = \text{take-bit LENGTH}('a) (\text{sint } w)$ **for** $w :: 'a::\text{len word}$ *<proof>***lemma** *bintr-uint*: $\text{LENGTH}('a) \leq n \implies \text{take-bit } n (\text{uint } w) = \text{uint } w$ **for** $w :: 'a::\text{len word}$ *<proof>***lemma** *wi-bintr*: $\text{LENGTH}('a::\text{len}) \leq n \implies$ $\text{word-of-int} (\text{take-bit } n w) = (\text{word-of-int } w :: 'a \text{ word})$ *<proof>***lemma** *word-numeral-alt*: $\text{numeral } b = \text{word-of-int} (\text{numeral } b)$ *<proof>***declare** *word-numeral-alt* [*symmetric, code-abbrev*]**lemma** *word-neg-numeral-alt*: $-\text{numeral } b = \text{word-of-int} (-\text{numeral } b)$ *<proof>***declare** *word-neg-numeral-alt* [*symmetric, code-abbrev*]**lemma** *uint-bintrunc* [*simp*]: $\text{uint} (\text{numeral bin} :: 'a \text{ word}) = \text{take-bit LENGTH}('a::\text{len}) (\text{numeral bin})$ *<proof>***lemma** *uint-bintrunc-neg* [*simp*]: $\text{uint} (-\text{numeral bin} :: 'a \text{ word}) = \text{take-bit LENGTH}('a::\text{len}) (-\text{numeral bin})$ *<proof>***lemma** *sint-sbintrunc* [*simp*]: $\text{sint} (\text{numeral bin} :: 'a \text{ word}) = \text{signed-take-bit} (\text{LENGTH}('a::\text{len}) - 1) (\text{numeral bin})$ *<proof>***lemma** *sint-sbintrunc-neg* [*simp*]: $\text{sint} (-\text{numeral bin} :: 'a \text{ word}) = \text{signed-take-bit} (\text{LENGTH}('a::\text{len}) - 1) (-\text{numeral bin})$

$\langle \text{proof} \rangle$

lemma *unat-bintrunc* [simp]:

$\text{unat} (\text{numeral bin} :: 'a::\text{len word}) = \text{take-bit LENGTH}('a) (\text{numeral bin})$

$\langle \text{proof} \rangle$

lemma *unat-bintrunc-neg* [simp]:

$\text{unat} (- \text{numeral bin} :: 'a::\text{len word}) = \text{nat} (\text{take-bit LENGTH}('a) (- \text{numeral bin}))$

$\langle \text{proof} \rangle$

lemma *size-0-eq*: $\text{size } w = 0 \implies v = w$

for $v \ w :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-ge-0* [iff]: $0 \leq \text{uint } x$

$\langle \text{proof} \rangle$

lemma *uint-lt2p* [iff]: $\text{uint } x < 2^{\text{LENGTH}('a)}$

for $x :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *sint-ge*: $-(2^{\text{LENGTH}('a)} - 1) \leq \text{sint } x$

for $x :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *sint-lt*: $\text{sint } x < 2^{\text{LENGTH}('a)} - 1$

for $x :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-m2p-neg*: $\text{uint } x - 2^{\text{LENGTH}('a)} < 0$

for $x :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-m2p-not-non-neg*: $\neg 0 \leq \text{uint } x - 2^{\text{LENGTH}('a)}$

for $x :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *lt2p-lem*: $\text{LENGTH}('a) \leq n \implies \text{uint } w < 2^n$

for $w :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-le-0-iff* [simp]: $\text{uint } x \leq 0 \iff \text{uint } x = 0$

$\langle \text{proof} \rangle$

lemma *uint-nat*: $\text{uint } w = \text{int} (\text{unat } w)$

$\langle \text{proof} \rangle$

lemma *uint-numeral*: $\text{uint} (\text{numeral } b :: 'a::\text{len word}) = \text{numeral } b \bmod 2^{\text{LENGTH}('a)}$

$LENGTH('a)$
 $\langle proof \rangle$

lemma *uint-neg-numeral*: $uint \ (- \ numeral \ b :: 'a::len \ word) = - \ numeral \ b \ mod \ 2^{\wedge} LENGTH('a)$
 $\langle proof \rangle$

lemma *unat-numeral*: $unat \ (numeral \ b :: 'a::len \ word) = numeral \ b \ mod \ 2^{\wedge} LENGTH('a)$
 $\langle proof \rangle$

lemma *sint-numeral*:
 $sint \ (numeral \ b :: 'a::len \ word) =$
 $(numeral \ b + 2^{\wedge}(LENGTH('a) - 1)) \ mod \ 2^{\wedge} LENGTH('a) - 2^{\wedge}(LENGTH('a)$
 $- 1)$
 $\langle proof \rangle$

lemma *word-of-int-0* [*simp, code-post*]: $word-of-int \ 0 = 0$
 $\langle proof \rangle$

lemma *word-of-int-1* [*simp, code-post*]: $word-of-int \ 1 = 1$
 $\langle proof \rangle$

lemma *word-of-int-neg-1* [*simp*]: $word-of-int \ (- \ 1) = - \ 1$
 $\langle proof \rangle$

lemma *word-of-int-numeral* [*simp*]: $(word-of-int \ (numeral \ bin) :: 'a::len \ word) =$
 $numeral \ bin$
 $\langle proof \rangle$

lemma *word-int-case-wi*:
 $word-int-case \ f \ (word-of-int \ i :: 'b \ word) = f \ (i \ mod \ 2^{\wedge} LENGTH('b::len))$
 $\langle proof \rangle$

lemma *word-int-split*:
 $P \ (word-int-case \ f \ x) =$
 $(\forall i. \ x = (word-of-int \ i :: 'b::len \ word) \wedge 0 \leq i \wedge i < 2^{\wedge} LENGTH('b) \longrightarrow P$
 $(f \ i))$
 $\langle proof \rangle$

lemma *word-int-split-asm*:
 $P \ (word-int-case \ f \ x) =$
 $(\exists n. \ x = (word-of-int \ n :: 'b::len \ word) \wedge 0 \leq n \wedge n < 2^{\wedge} LENGTH('b::len)$
 $\wedge \neg P \ (f \ n))$
 $\langle proof \rangle$

lemma *uint-range-size*: $0 \leq uint \ w \wedge uint \ w < 2^{\wedge} size \ w$
 $\langle proof \rangle$

lemma *sint-range-size*: $-(2^{\wedge}(\text{size } w - \text{Suc } 0)) \leq \text{sint } w \wedge \text{sint } w < 2^{\wedge}(\text{size } w - \text{Suc } 0)$
 ⟨proof⟩

lemma *sint-above-size*: $2^{\wedge}(\text{size } w - 1) \leq x \implies \text{sint } w < x$
for $w :: 'a::\text{len word}$
 ⟨proof⟩

lemma *sint-below-size*: $x \leq -(2^{\wedge}(\text{size } w - 1)) \implies x \leq \text{sint } w$
for $w :: 'a::\text{len word}$
 ⟨proof⟩

lemma *word-unat-eq-iff*:
 $\langle v = w \longleftrightarrow \text{unat } v = \text{unat } w \rangle$
for $v w :: \langle 'a::\text{len word} \rangle$
 ⟨proof⟩

107.15 Testing bits

lemma *bin-nth-uint-imp*: $\text{bit } (\text{uint } w) \ n \implies n < \text{LENGTH}('a)$
for $w :: 'a::\text{len word}$
 ⟨proof⟩

lemma *bin-nth-sint*:
 $\text{LENGTH}('a) \leq n \implies$
 $\text{bit } (\text{sint } w) \ n = \text{bit } (\text{sint } w) \ (\text{LENGTH}('a) - 1)$
for $w :: 'a::\text{len word}$
 ⟨proof⟩

lemma *num-of-bintr'*:
 $\text{take-bit } (\text{LENGTH}('a::\text{len})) \ (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$
 ⟨proof⟩

lemma *num-of-sbintr'*:
 $\text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - 1) \ (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$
 ⟨proof⟩

lemma *num-abs-bintr*:
 $(\text{numeral } x :: 'a \text{ word}) =$
 $\text{word-of-int } (\text{take-bit } (\text{LENGTH}('a::\text{len})) \ (\text{numeral } x))$
 ⟨proof⟩

lemma *num-abs-sbintr*:
 $(\text{numeral } x :: 'a \text{ word}) =$
 $\text{word-of-int } (\text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - 1) \ (\text{numeral } x))$
 ⟨proof⟩

cast – note, no arg for new length, as it’s determined by type of result,

thus in $\text{cast } w = w$, the type means cast to length of w !

lemma *bit-ucast-iff*:

$\langle \text{bit } (\text{ucast } a :: 'a::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}('a::\text{len}) \wedge \text{bit } a \ n \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-id* [*simp*]: $\text{ucast } w = w$

$\langle \text{proof} \rangle$

lemma *scast-id* [*simp*]: $\text{scast } w = w$

$\langle \text{proof} \rangle$

lemma *ucast-mask-eq*:

$\langle \text{ucast } (\text{mask } n :: 'b \text{ word}) = \text{mask } (\text{min } \text{LENGTH}('b::\text{len}) \ n) \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-bintr* [*simp*]:

$\text{ucast } (\text{numeral } w :: 'a::\text{len word}) =$
 $\text{word-of-int } (\text{take-bit } (\text{LENGTH}('a)) \ (\text{numeral } w))$
 $\langle \text{proof} \rangle$

lemma *scast-sbintr* [*simp*]:

$\text{scast } (\text{numeral } w :: 'a::\text{len word}) =$
 $\text{word-of-int } (\text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \ (\text{numeral } w))$
 $\langle \text{proof} \rangle$

lemma *source-size*: $\text{source-size } (c :: 'a::\text{len word} \Rightarrow -) = \text{LENGTH}('a)$

$\langle \text{proof} \rangle$

lemma *target-size*: $\text{target-size } (c :: - \Rightarrow 'b::\text{len word}) = \text{LENGTH}('b)$

$\langle \text{proof} \rangle$

lemma *is-down*: $\text{is-down } c \longleftrightarrow \text{LENGTH}('b) \leq \text{LENGTH}('a)$

for $c :: 'a::\text{len word} \Rightarrow 'b::\text{len word}$

$\langle \text{proof} \rangle$

lemma *is-up*: $\text{is-up } c \longleftrightarrow \text{LENGTH}('a) \leq \text{LENGTH}('b)$

for $c :: 'a::\text{len word} \Rightarrow 'b::\text{len word}$

$\langle \text{proof} \rangle$

lemma *is-up-down*:

$\langle \text{is-up } c \longleftrightarrow \text{is-down } d \rangle$
for $c :: 'a::\text{len word} \Rightarrow 'b::\text{len word}$
and $d :: 'b::\text{len word} \Rightarrow 'a::\text{len word}$
 $\langle \text{proof} \rangle$

context

fixes *dummy-types* :: $'a::\text{len} \times 'b::\text{len}$

begin

private abbreviation (*input*) $UCAST :: \langle 'a::len\ word \Rightarrow 'b::len\ word \rangle$
where $\langle UCAST == ucast \rangle$

private abbreviation (*input*) $SCAST :: \langle 'a::len\ word \Rightarrow 'b::len\ word \rangle$
where $\langle SCAST == scast \rangle$

lemma *down-cast-same*:
 $\langle UCAST = scast \rangle$ **if** $\langle is_down\ UCAST \rangle$
 $\langle proof \rangle$

lemma *sint-up-scast*:
 $\langle sint\ (SCAST\ w) = sint\ w \rangle$ **if** $\langle is_up\ SCAST \rangle$
 $\langle proof \rangle$

lemma *uint-up-ucast*:
 $\langle uint\ (UCAST\ w) = uint\ w \rangle$ **if** $\langle is_up\ UCAST \rangle$
 $\langle proof \rangle$

lemma *ucast-up-ucast*:
 $\langle ucast\ (UCAST\ w) = ucast\ w \rangle$ **if** $\langle is_up\ UCAST \rangle$
 $\langle proof \rangle$

lemma *ucast-up-ucast-id*:
 $\langle ucast\ (UCAST\ w) = w \rangle$ **if** $\langle is_up\ UCAST \rangle$
 $\langle proof \rangle$

lemma *scast-up-scast*:
 $\langle scast\ (SCAST\ w) = scast\ w \rangle$ **if** $\langle is_up\ SCAST \rangle$
 $\langle proof \rangle$

lemma *scast-up-scast-id*:
 $\langle scast\ (SCAST\ w) = w \rangle$ **if** $\langle is_up\ SCAST \rangle$
 $\langle proof \rangle$

lemma *isduu*:
 $\langle is_up\ UCAST \rangle$ **if** $\langle is_down\ d \rangle$
for $d :: \langle 'b\ word \Rightarrow 'a\ word \rangle$
 $\langle proof \rangle$

lemma *isdus*:
 $\langle is_up\ SCAST \rangle$ **if** $\langle is_down\ d \rangle$
for $d :: \langle 'b\ word \Rightarrow 'a\ word \rangle$
 $\langle proof \rangle$

lemmas *ucast-down-ucast-id* = *isduu* [THEN *ucast-up-ucast-id*]

lemmas *scast-down-scast-id* = *isdus* [THEN *scast-up-scast-id*]

lemma *up-ucast-surj*:

$\langle \text{surj } (\text{ucast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle \text{ if } \langle \text{is-up } \text{UCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *up-scast-surj*:

$\langle \text{surj } (\text{scast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle \text{ if } \langle \text{is-up } \text{SCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *down-ucast-inj*:

$\langle \text{inj-on } \text{UCAST } A \rangle \text{ if } \langle \text{is-down } (\text{ucast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$
 $\langle \text{proof} \rangle$

lemma *down-scast-inj*:

$\langle \text{inj-on } \text{SCAST } A \rangle \text{ if } \langle \text{is-down } (\text{scast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-down-wi*:

$\langle \text{UCAST } (\text{word-of-int } x) = \text{word-of-int } x \rangle \text{ if } \langle \text{is-down } \text{UCAST} \rangle$
 $\langle \text{proof} \rangle$

lemma *ucast-down-no*:

$\langle \text{UCAST } (\text{numeral bin}) = \text{numeral bin} \rangle \text{ if } \langle \text{is-down } \text{UCAST} \rangle$
 $\langle \text{proof} \rangle$

end

lemmas *word-log-defs* = *word-and-def word-or-def word-xor-def word-not-def*

lemma *bit-last-iff*:

$\langle \text{bit } w (\text{LENGTH}('a) - \text{Suc } 0) \longleftrightarrow \text{sint } w < 0 \rangle \text{ (is } \langle ?P \longleftrightarrow ?Q \rangle)$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *drop-bit-eq-zero-iff-not-bit-last*:

$\langle \text{drop-bit } (\text{LENGTH}('a) - \text{Suc } 0) \ w = 0 \longleftrightarrow \neg \text{bit } w (\text{LENGTH}('a) - \text{Suc } 0) \rangle$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *unat-div*:

$\langle \text{unat } (x \text{ div } y) = \text{unat } x \text{ div unat } y \rangle$
 $\langle \text{proof} \rangle$

lemma *unat-mod*:

$\langle \text{unat } (x \text{ mod } y) = \text{unat } x \text{ mod unat } y \rangle$
 $\langle \text{proof} \rangle$

107.16 Word Arithmetic

lemmas *less-eq-word-numeral-numeral* [simp] =

word-le-def [of $\langle \text{numeral } a \rangle \langle \text{numeral } b \rangle$, simplified *wint-bintrunc wint-bintrunc-neg*

```

unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-numeral-numeral [simp] =
  word-less-def [of ⟨numeral a⟩ ⟨numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-eq-word-minus-numeral-numeral [simp] =
  word-le-def [of ⟨- numeral a⟩ ⟨numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-minus-numeral-numeral [simp] =
  word-less-def [of ⟨- numeral a⟩ ⟨numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-eq-word-numeral-minus-numeral [simp] =
  word-le-def [of ⟨numeral a⟩ ⟨- numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-numeral-minus-numeral [simp] =
  word-less-def [of ⟨numeral a⟩ ⟨- numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-eq-word-minus-numeral-minus-numeral [simp] =
  word-le-def [of ⟨- numeral a⟩ ⟨- numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-minus-numeral-minus-numeral [simp] =
  word-less-def [of ⟨- numeral a⟩ ⟨- numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-numeral-minus-1 [simp] =
  word-less-def [of ⟨numeral a⟩ ⟨- 1⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-minus-numeral-minus-1 [simp] =
  word-less-def [of ⟨- numeral a⟩ ⟨- 1⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b

lemmas sless-eq-word-numeral-numeral [simp] =
  word-sle-eq [of ⟨numeral a⟩ ⟨numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-word-numeral-numeral [simp] =
  word-sless-alt [of ⟨numeral a⟩ ⟨numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-eq-word-minus-numeral-numeral [simp] =
  word-sle-eq [of ⟨- numeral a⟩ ⟨numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-word-minus-numeral-numeral [simp] =

```

```

word-sless-alt [of <- numeral a> <numeral b>, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-eq-word-numeral-minus-numeral [simp] =
  word-sle-eq [of <numeral a> <- numeral b>, simplified sint-sbintrunc sint-sbintrunc-neg]
    for a b
lemmas sless-word-numeral-minus-numeral [simp] =
  word-sless-alt [of <numeral a> <- numeral b>, simplified sint-sbintrunc sint-sbintrunc-neg]
    for a b
lemmas sless-eq-word-minus-numeral-minus-numeral [simp] =
  word-sle-eq [of <- numeral a> <- numeral b>, simplified sint-sbintrunc sint-sbintrunc-neg]
    for a b
lemmas sless-word-minus-numeral-minus-numeral [simp] =
  word-sless-alt [of <- numeral a> <- numeral b>, simplified sint-sbintrunc sint-sbintrunc-neg]
    for a b

lemmas div-word-numeral-numeral [simp] =
  word-div-def [of <numeral a> <numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b
lemmas div-word-minus-numeral-numeral [simp] =
  word-div-def [of <- numeral a> <numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b
lemmas div-word-numeral-minus-numeral [simp] =
  word-div-def [of <numeral a> <- numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b
lemmas div-word-minus-numeral-minus-numeral [simp] =
  word-div-def [of <- numeral a> <- numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b
lemmas div-word-minus-1-numeral [simp] =
  word-div-def [of <- 1> <numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b
lemmas div-word-minus-1-minus-numeral [simp] =
  word-div-def [of <- 1> <- numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b

lemmas mod-word-numeral-numeral [simp] =
  word-mod-def [of <numeral a> <numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b
lemmas mod-word-minus-numeral-numeral [simp] =
  word-mod-def [of <- numeral a> <numeral b>, simplified uint-bintrunc uint-bintrunc-neg
    unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
    for a b
lemmas mod-word-numeral-minus-numeral [simp] =

```

word-mod-def [of $\langle \text{numeral } a \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for $a\ b$
lemmas *mod-word-minus-numeral-minus-numeral* [simp] =
word-mod-def [of $\langle - \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for $a\ b$
lemmas *mod-word-minus-1-numeral* [simp] =
word-mod-def [of $\langle - 1 \rangle \langle \text{numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for $a\ b$
lemmas *mod-word-minus-1-minus-numeral* [simp] =
word-mod-def [of $\langle - 1 \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for $a\ b$

lemma *signed-drop-bit-of-1* [simp]:
 $\langle \text{signed-drop-bit } n\ (1 :: 'a::\text{len word}) = \text{of_bool } (\text{LENGTH}('a) = 1 \vee n = 0) \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-word-beyond-length-eq*:
 $\langle \text{take-bit } n\ w = w \rangle \text{ if } \langle \text{LENGTH}('a) \leq n \rangle \text{ for } w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *word-div-no* [simp] = *word-div-def* [of *numeral a numeral b*] **for** $a\ b$
lemmas *word-mod-no* [simp] = *word-mod-def* [of *numeral a numeral b*] **for** $a\ b$
lemmas *word-less-no* [simp] = *word-less-def* [of *numeral a numeral b*] **for** $a\ b$
lemmas *word-le-no* [simp] = *word-le-def* [of *numeral a numeral b*] **for** $a\ b$
lemmas *word-sless-no* [simp] = *word-sless-eq* [of *numeral a numeral b*] **for** $a\ b$
lemmas *word-sle-no* [simp] = *word-sle-eq* [of *numeral a numeral b*] **for** $a\ b$

lemma *size-0-same'*: $\text{size } w = 0 \implies w = v$
for $v\ w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *size-0-same* = *size-0-same'* [unfolded *word-size*]

lemmas *unat-eq-0* = *unat-0-iff*
lemmas *unat-eq-zero* = *unat-0-iff*

lemma *mask-1*: $\text{mask } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *mask-Suc-0*: $\text{mask } (\text{Suc } 0) = 1$
 $\langle \text{proof} \rangle$

lemma *bin-last-bintrunc*: $\text{odd } (\text{take-bit } l\ n) \longleftrightarrow l > 0 \wedge \text{odd } n$
 $\langle \text{proof} \rangle$

lemma *push-bit-word-beyond* [simp]:
 $\langle \text{push-bit } n \ w = 0 \rangle \text{ if } \langle \text{LENGTH}('a) \leq n \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *drop-bit-word-beyond* [simp]:
 $\langle \text{drop-bit } n \ w = 0 \rangle \text{ if } \langle \text{LENGTH}('a) \leq n \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *signed-drop-bit-beyond*:
 $\langle \text{signed-drop-bit } n \ w = (\text{if bit } w \ (\text{LENGTH}('a) - \text{Suc } 0) \text{ then } - 1 \text{ else } 0) \rangle$
 $\text{if } \langle \text{LENGTH}('a) \leq n \rangle \text{ for } w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *take-bit-numeral-minus-numeral-word* [simp]:
 $\langle \text{take-bit } (\text{numeral } m) \ (- \text{numeral } n :: 'a::\text{len word}) =$
 $(\text{case take-bit-num } (\text{numeral } m) \ n \text{ of None } \Rightarrow 0 \mid \text{Some } q \Rightarrow \text{take-bit } (\text{numeral } m) \ (2 \wedge \text{numeral } m - \text{numeral } q)) \rangle \text{ (is } \langle ?lhs = ?rhs \rangle)$
 $\langle \text{proof} \rangle$

lemma *of-nat-inverse*:
 $\langle \text{word-of-nat } r = a \implies r < 2 \wedge \text{LENGTH}('a) \implies \text{unat } a = r \rangle$
 $\text{for } a :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

107.17 Transferring goals from words to ints

lemma *word-ths*:
shows *word-succ-p1*: $\text{word-succ } a = a + 1$
and *word-pred-m1*: $\text{word-pred } a = a - 1$
and *word-pred-succ*: $\text{word-pred } (\text{word-succ } a) = a$
and *word-succ-pred*: $\text{word-succ } (\text{word-pred } a) = a$
and *word-mult-succ*: $\text{word-succ } a * b = b + a * b$
 $\langle \text{proof} \rangle$

lemma *uint-cong*: $x = y \implies \text{uint } x = \text{uint } y$
 $\langle \text{proof} \rangle$

lemma *uint-word-ariths*:
fixes $a \ b :: 'a::\text{len word}$
shows $\text{uint } (a + b) = (\text{uint } a + \text{uint } b) \bmod 2 \wedge \text{LENGTH}('a::\text{len})$
and $\text{uint } (a - b) = (\text{uint } a - \text{uint } b) \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (a * b) = \text{uint } a * \text{uint } b \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (- a) = - \text{uint } a \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (\text{word-succ } a) = (\text{uint } a + 1) \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (\text{word-pred } a) = (\text{uint } a - 1) \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (0 :: 'a \text{ word}) = 0 \bmod 2 \wedge \text{LENGTH}('a)$
and $\text{uint } (1 :: 'a \text{ word}) = 1 \bmod 2 \wedge \text{LENGTH}('a)$
 $\langle \text{proof} \rangle$

lemma *uint-word-arith-bintrs*:

fixes $a\ b :: 'a::len\ word$

shows $uint\ (a + b) = take-bit\ (LENGTH('a))\ (uint\ a + uint\ b)$

and $uint\ (a - b) = take-bit\ (LENGTH('a))\ (uint\ a - uint\ b)$

and $uint\ (a * b) = take-bit\ (LENGTH('a))\ (uint\ a * uint\ b)$

and $uint\ (- a) = take-bit\ (LENGTH('a))\ (- uint\ a)$

and $uint\ (word-succ\ a) = take-bit\ (LENGTH('a))\ (uint\ a + 1)$

and $uint\ (word-pred\ a) = take-bit\ (LENGTH('a))\ (uint\ a - 1)$

and $uint\ (0 :: 'a\ word) = take-bit\ (LENGTH('a))\ 0$

and $uint\ (1 :: 'a\ word) = take-bit\ (LENGTH('a))\ 1$

<proof>

context

fixes $a\ b :: 'a::len\ word$

begin

lemma *sint-word-add*: $sint\ (a + b) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a + sint\ b)$

<proof>

lemma *sint-word-diff*: $sint\ (a - b) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a - sint\ b)$

<proof>

lemma *sint-word-mult*: $sint\ (a * b) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a * sint\ b)$

<proof>

lemma *sint-word-minus*: $sint\ (- a) = signed-take-bit\ (LENGTH('a) - 1)\ (- sint\ a)$

<proof>

lemma *sint-word-succ*: $sint\ (word-succ\ a) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a + 1)$

<proof>

lemma *sint-word-pred*: $sint\ (word-pred\ a) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a - 1)$

<proof>

lemma *sint-word-01*:

$sint\ (0 :: 'a\ word) = signed-take-bit\ (LENGTH('a) - 1)\ 0$

$sint\ (1 :: 'a\ word) = signed-take-bit\ (LENGTH('a) - 1)\ 1$

<proof>

end

lemmas *sint-word-ariths* =

*sint-word-add sint-word-diff sint-word-mult sint-word-minus
sint-word-succ sint-word-pred sint-word-01*

lemma *word-pred-0-n1*: *word-pred 0 = word-of-int (- 1)*
⟨proof⟩

lemma *succ-pred-no* [*simp*]:
word-succ (numeral w) = numeral w + 1
word-pred (numeral w) = numeral w - 1
word-succ (- numeral w) = - numeral w + 1
word-pred (- numeral w) = - numeral w - 1
⟨proof⟩

lemma *word-sp-01* [*simp*]:
word-succ (- 1) = 0 ∧ word-succ 0 = 1 ∧ word-pred 0 = - 1 ∧ word-pred 1 =
0
⟨proof⟩

lemma *word-of-int-Ex*: $\exists y. x = \text{word-of-int } y$
⟨proof⟩

107.18 Order on fixed-length words

lift-definition *udvd* :: $\langle 'a::\text{len word} \Rightarrow 'a::\text{len word} \Rightarrow \text{bool} \rangle$ (**infixl** $\langle \text{udvd} \rangle$ 50)
is $\langle \lambda k l. \text{take-bit LENGTH('a)} k \text{ dvd take-bit LENGTH('a)} l \rangle$ *⟨proof⟩*

lemma *udvd-iff-dvd*:
 $\langle x \text{ udvd } y \iff \text{unat } x \text{ dvd unat } y \rangle$
⟨proof⟩

lemma *udvd-iff-dvd-int*:
 $\langle v \text{ udvd } w \iff \text{uint } v \text{ dvd uint } w \rangle$
⟨proof⟩

lemma *udvdI* [*intro*]:
 $\langle v \text{ udvd } w \rangle$ **if** $\langle \text{unat } w = \text{unat } v * \text{unat } u \rangle$
⟨proof⟩

lemma *udvdE* [*elim*]:
fixes $v w :: \langle 'a::\text{len word} \rangle$
assumes $\langle v \text{ udvd } w \rangle$
obtains $u :: \langle 'a \text{ word} \rangle$ **where** $\langle \text{unat } w = \text{unat } v * \text{unat } u \rangle$
⟨proof⟩

lemma *udvd-imp-mod-eq-0*:
 $\langle w \bmod v = 0 \rangle$ **if** $\langle v \text{ udvd } w \rangle$
⟨proof⟩

lemma *mod-eq-0-imp-udvd* [*intro?*]:
 $\langle v \text{ udvd } w \rangle$ **if** $\langle w \bmod v = 0 \rangle$

$\langle \text{proof} \rangle$

lemma *udvd-imp-dvd*:

$\langle v \text{ dvd } w \rangle$ **if** $\langle v \text{ udvd } w \rangle$ **for** $v \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *exp-dvd-iff-exp-udvd*:

$\langle 2 \wedge^n \text{ dvd } w \longleftrightarrow 2 \wedge^n \text{ udvd } w \rangle$ **for** $v \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *udvd-nat-alt*:

$\langle a \text{ udvd } b \longleftrightarrow (\exists n. \text{unat } b = n * \text{unat } a) \rangle$
 $\langle \text{proof} \rangle$

lemma *udvd-unfold-int*:

$\langle a \text{ udvd } b \longleftrightarrow (\exists n \geq 0. \text{uint } b = n * \text{uint } a) \rangle$
 $\langle \text{proof} \rangle$

lemma *unat-minus-one*:

$\langle \text{unat } (w - 1) = \text{unat } w - 1 \rangle$ **if** $\langle w \neq 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *measure-unat*: $p \neq 0 \implies \text{unat } (p - 1) < \text{unat } p$

$\langle \text{proof} \rangle$

lemmas *uint-add-ge0* [simp] = *add-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

lemmas *uint-mult-ge0* [simp] = *mult-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

lemma *uint-sub-lt2p* [simp]: $\text{uint } x - \text{uint } y < 2 \wedge \text{LENGTH}('a)$

for $x \ y :: 'a::\text{len word}$ **and** $y :: 'b::\text{len word}$

$\langle \text{proof} \rangle$

107.19 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:

$(\text{uint } x + \text{uint } y < 2 \wedge \text{LENGTH}('a)) =$

$(\text{uint } (x + y) = \text{uint } x + \text{uint } y)$

for $x \ y :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-mult-lem*:

$(\text{uint } x * \text{uint } y < 2 \wedge \text{LENGTH}('a)) =$

$(\text{uint } (x * y) = \text{uint } x * \text{uint } y)$

for $x \ y :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-sub-lem*: $\text{uint } x \geq \text{uint } y \longleftrightarrow \text{uint } (x - y) = \text{uint } x - \text{uint } y$

$\langle \text{proof} \rangle$

lemma *uint-add-le*: $\text{uint } (x + y) \leq \text{uint } x + \text{uint } y$
 $\langle \text{proof} \rangle$

lemma *uint-sub-ge*: $\text{uint } (x - y) \geq \text{uint } x - \text{uint } y$
 $\langle \text{proof} \rangle$

lemma *int-mod-ge*: $\langle a \leq a \bmod n \rangle$ **if** $\langle a < n \rangle$ $\langle 0 < n \rangle$
for $a \ n :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *mod-add-if-z*:
 $\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$
 $(x + y) \bmod z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$
for $x \ y \ z :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *uint-plus-if'*:
 $\text{uint } (a + b) =$
 $(\text{if } \text{uint } a + \text{uint } b < 2^{\wedge \text{LENGTH('a)}} \text{ then } \text{uint } a + \text{uint } b$
 $\text{else } \text{uint } a + \text{uint } b - 2^{\wedge \text{LENGTH('a)}})$
for $a \ b :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *mod-sub-if-z*:
 $\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$
 $(x - y) \bmod z = (\text{if } y \leq x \text{ then } x - y \text{ else } x - y + z)$
for $x \ y \ z :: \text{int}$
 $\langle \text{proof} \rangle$

lemma *uint-sub-if'*:
 $\text{uint } (a - b) =$
 $(\text{if } \text{uint } b \leq \text{uint } a \text{ then } \text{uint } a - \text{uint } b$
 $\text{else } \text{uint } a - \text{uint } b + 2^{\wedge \text{LENGTH('a)}})$
for $a \ b :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-of-int-inverse*:
 $\text{word-of-int } r = a \implies 0 \leq r \implies r < 2^{\wedge \text{LENGTH('a)}} \implies \text{uint } a = r$
for $a :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *unat-split*: $P (\text{unat } x) \longleftrightarrow (\forall n. \text{of-nat } n = x \wedge n < 2^{\wedge \text{LENGTH('a)}} \longrightarrow P \ n)$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *unat-split-asm*: $P (\text{unat } x) \longleftrightarrow (\exists n. \text{of-nat } n = x \wedge n < 2^{\wedge \text{LENGTH('a)}} \wedge \neg P \ n)$

for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *un-ui-le*:
 $\langle unat\ a \leq unat\ b \longleftrightarrow uint\ a \leq uint\ b \rangle$
 $\langle proof \rangle$

lemma *unat-plus-if'*:
 $\langle unat\ (a + b) =$
 $(if\ unat\ a + unat\ b < 2^{\wedge}LENGTH('a)$
 $then\ unat\ a + unat\ b$
 $else\ unat\ a + unat\ b - 2^{\wedge}LENGTH('a)) \rangle$ **for** $a\ b :: 'a::len\ word$
 $\langle proof \rangle$

lemma *unat-sub-if-size*:
 $unat\ (x - y) =$
 $(if\ unat\ y \leq unat\ x$
 $then\ unat\ x - unat\ y$
 $else\ unat\ x + 2^{\wedge}size\ x - unat\ y)$
 $\langle proof \rangle$

lemmas *unat-sub-if' = unat-sub-if-size [unfolded word-size]*

lemma *uint-split*:
 $P\ (uint\ x) = (\forall i. word-of-int\ i = x \wedge 0 \leq i \wedge i < 2^{\wedge}LENGTH('a) \longrightarrow P\ i)$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

lemma *uint-split-asm*:
 $P\ (uint\ x) = (\nexists i. word-of-int\ i = x \wedge 0 \leq i \wedge i < 2^{\wedge}LENGTH('a) \wedge \neg P\ i)$
for $x :: 'a::len\ word$
 $\langle proof \rangle$

107.20 Some proof tool support

lemma *power-False-cong*: $False \Longrightarrow a^{\wedge} b = c^{\wedge} d$
 $\langle proof \rangle$

lemmas *unat-splits = unat-split unat-split-asm*

lemmas *unat-arith-simps =*
 $word-le-nat-alt\ word-less-nat-alt$
 $word-unat-eq-iff$
 $unat-sub-if'\ unat-plus-if'\ unat-div\ unat-mod$

lemmas *uint-splits = uint-split uint-split-asm*

lemmas *uint-arith-simps =*
 $word-le-def\ word-less-alt$

word-uint-eq-iff
uint-sub-if' uint-plus-if'

— *unat-arith-tac*: tactic to reduce word arithmetic to *nat*, try to solve via *arith*
 $\langle ML \rangle$

107.21 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*: $x \leq x + y \longleftrightarrow \text{uint } x + \text{uint } y < 2^{\text{size } x}$
 for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *no-olen-add* = *no-plus-overflow-uint-size* [*unfolded word-size*]

lemma *no-ulen-sub*: $x \geq x - y \longleftrightarrow \text{uint } y \leq \text{uint } x$
 for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *no-olen-add'*: $x \leq y + x \longleftrightarrow \text{uint } y + \text{uint } x < 2^{\text{LENGTH}('a)}$
 for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *olen-add-equiv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*]]

lemmas *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]
lemmas *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]
lemmas *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*]
lemmas *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]
lemmas *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]
lemmas *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

lemma *word-less-sub1*: $x \neq 0 \implies 1 < x \longleftrightarrow 0 < x - 1$
 for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-le-sub1*: $x \neq 0 \implies 1 \leq x \longleftrightarrow 0 \leq x - 1$
 for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *sub-wrap-lt*: $x < x - z \longleftrightarrow x < z$
 for $x \ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *sub-wrap*: $x \leq x - z \longleftrightarrow z = 0 \vee x < z$
 for $x \ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *plus-minus-not-NULL-ab*: $x \leq ab - c \implies c \leq ab \implies c \neq 0 \implies x + c \neq 0$

for $x \text{ } ab \text{ } c :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *plus-minus-no-overflow-ab*: $x \leq ab - c \implies c \leq ab \implies x \leq x + c$
for $x \text{ } ab \text{ } c :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *le-minus'*: $a + c \leq b \implies a \leq a + c \implies c \leq b - a$
for $a \text{ } b \text{ } c :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *le-plus'*: $a \leq b \implies c \leq b - a \implies a + c \leq b$
for $a \text{ } b \text{ } c :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemmas *le-plus = le-plus'* [rotated]

lemmas *le-minus = leD* [THEN *thin-rl*, THEN *le-minus'*]

lemma *word-plus-mono-right*: $y \leq z \implies x \leq x + z \implies x + y \leq x + z$
for $x \text{ } y \text{ } z :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *word-less-minus-cancel*: $y - x < z - x \implies x \leq z \implies y < z$
for $x \text{ } y \text{ } z :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *word-less-minus-mono-left*: $y < z \implies x \leq y \implies y - x < z - x$
for $x \text{ } y \text{ } z :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *word-less-minus-mono*: $a < c \implies d < b \implies a - b < a \implies c - d < c$
 $\implies a - b < c - d$
for $a \text{ } b \text{ } c \text{ } d :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *word-le-minus-cancel*: $y - x \leq z - x \implies x \leq z \implies y \leq z$
for $x \text{ } y \text{ } z :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *word-le-minus-mono-left*: $y \leq z \implies x \leq y \implies y - x \leq z - x$
for $x \text{ } y \text{ } z :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *word-le-minus-mono*:
 $a \leq c \implies d \leq b \implies a - b \leq a \implies c - d \leq c \implies a - b \leq c - d$
for $a \text{ } b \text{ } c \text{ } d :: 'a::len \text{ } word$
 $\langle proof \rangle$

lemma *plus-le-left-cancel-wrap*: $x + y' < x \implies x + y < x \implies x + y' < x + y$
 $\longleftrightarrow y' < y$

for $x\ y\ y' :: 'a::len\ word$
 $\langle proof \rangle$

lemma *plus-le-left-cancel-nowrap*: $x \leq x + y' \implies x \leq x + y \implies x + y' < x + y$
 $\longleftrightarrow y' < y$

for $x\ y\ y' :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-plus-mono-right2*: $a \leq a + b \implies c \leq b \implies a \leq a + c$

for $a\ b\ c :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-less-add-right*: $x < y - z \implies z \leq y \implies x + z < y$

for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-less-sub-right*: $x < y + z \implies y \leq x \implies x - y < z$

for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-le-plus-either*: $x \leq y \vee x \leq z \implies y \leq y + z \implies x \leq y + z$

for $x\ y\ z :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-less-nowrapI*: $x < z - k \implies k \leq z \implies 0 < k \implies x < x + k$

for $x\ z\ k :: 'a::len\ word$
 $\langle proof \rangle$

lemma *inc-le*: $i < m \implies i + 1 \leq m$

for $i\ m :: 'a::len\ word$
 $\langle proof \rangle$

lemma *less-imp-less-eq-dec*:

$\langle v \leq w - 1 \rangle$ **if** $\langle v < w \rangle$ **for** $v\ w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *inc-less-eq-triv-imp*:

$\langle w = -1 \rangle$ **if** $\langle w + 1 \leq w \rangle$ **for** $w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *less-eq-dec-triv-imp*:

$\langle w = 0 \rangle$ **if** $\langle w \leq w - 1 \rangle$ **for** $w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *inc-less-eq-iff*:

$\langle v + 1 \leq w \longleftrightarrow v = -1 \vee v < w \rangle$ **for** $v\ w :: \langle 'a::len\ word \rangle$
 $\langle proof \rangle$

lemma *less-eq-dec-iff*:

$\langle v \leq w - 1 \iff w = 0 \vee v < w \rangle$ **for** $v\ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *inc-i*: $1 \leq i \implies i < m \implies 1 \leq i + 1 \wedge i + 1 \leq m$

for $i\ m :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *dec-less-imp-less-eq*:

$\langle v \leq w \rangle$ **if** $\langle v - 1 < w \rangle$ **for** $v\ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *less-inc-imp-less-eq*:

$\langle v \leq w \rangle$ **if** $\langle v < w + 1 \rangle$ **for** $v\ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *less-eq-dec-self-iff-eq*:

$\langle w \leq w - 1 \iff w = 0 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *inc-less-eq-self-iff-eq*:

$\langle w + 1 \leq w \iff w = -1 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *udvd-incr-lem*:

$\llbracket up < uq; up = ua + n * \text{uint } K; uq = ua + n' * \text{uint } K \rrbracket$
 $\implies up + \text{uint } K \leq uq$
 $\langle \text{proof} \rangle$

lemma *udvd-incr'*:

$p < q \implies \text{uint } p = ua + n * \text{uint } K \implies$
 $\text{uint } q = ua + n' * \text{uint } K \implies p + K \leq q$
 $\langle \text{proof} \rangle$

lemma *udvd-decr'*:

assumes $p < q$ $\text{uint } p = ua + n * \text{uint } K$ $\text{uint } q = ua + n' * \text{uint } K$
shows $\text{uint } q = ua + n' * \text{uint } K \implies p \leq q - K$
 $\langle \text{proof} \rangle$

lemmas *udvd-incr-lem0* = *udvd-incr-lem* [**where** $ua=0$, *unfolded add-0-left*]

lemmas *udvd-incr0* = *udvd-incr'* [**where** $ua=0$, *unfolded add-0-left*]

lemmas *udvd-decr0* = *udvd-decr'* [**where** $ua=0$, *unfolded add-0-left*]

lemma *udvd-minus-le'*: $xy < k \implies z \text{ udvd } xy \implies z \text{ udvd } k \implies xy \leq k - z$

$\langle \text{proof} \rangle$

lemma *udvd-incr2-K*:

$p < a + s \implies a \leq a + s \implies K \text{ udvd } s \implies K \text{ udvd } p - a \implies a \leq p \implies$

$0 < K \implies p \leq p + K \wedge p + K \leq a + s$
 $\langle \text{proof} \rangle$

107.22 Arithmetic type class instantiations

lemmas *word-le-0-iff* [simp] =
word-zero-le [THEN *leD*, THEN *antisym-conv1*]

lemma *word-of-int-nat*: $0 \leq x \implies \text{word-of-int } x = \text{of-nat } (\text{nat } x)$
 $\langle \text{proof} \rangle$

note that *iszero-def* is only for class *comm-semiring-1-cancel*, which requires word length ≥ 1 , ie $'a::\text{len word}$

lemma *iszero-word-no* [simp]:
iszero (numeral bin :: $'a::\text{len word}$) =
iszero (take-bit *LENGTH*('a) (numeral bin :: int))
 $\langle \text{proof} \rangle$

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [simp] =
eq-numeral-iff-iszero [where $'a = 'a::\text{len word}$]

lemma *word-less-eq-imp-half-less-eq*:
 $\langle v \text{ div } 2 \leq w \text{ div } 2 \rangle \text{ if } \langle v \leq w \rangle \text{ for } v \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-half-less-imp-less-eq*:
 $\langle v \leq w \rangle \text{ if } \langle v \text{ div } 2 < w \text{ div } 2 \rangle \text{ for } v \ w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

107.23 Word and nat

lemma *word-nchotomy*: $\forall w :: 'a::\text{len word}. \exists n. w = \text{of-nat } n \wedge n < 2^{\text{LENGTH}('a)}$
 $\langle \text{proof} \rangle$

lemma *of-nat-eq*: $\text{of-nat } n = w \longleftrightarrow (\exists q. n = \text{unat } w + q * 2^{\text{LENGTH}('a)})$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *of-nat-eq-size*: $\text{of-nat } n = w \longleftrightarrow (\exists q. n = \text{unat } w + q * 2^{\text{size } w})$
 $\langle \text{proof} \rangle$

lemma *of-nat-0*: $\text{of-nat } m = (0 :: 'a::\text{len word}) \longleftrightarrow (\exists q. m = q * 2^{\text{LENGTH}('a)})$
 $\langle \text{proof} \rangle$

lemma *of-nat-2p* [simp]: $\text{of-nat } (2^{\text{LENGTH}('a)}) = (0 :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemma *of-nat-gt-0*: $\text{of-nat } k \neq 0 \implies 0 < k$

$\langle \text{proof} \rangle$

lemma *of-nat-neq-0*: $0 < k \implies k < 2^{\wedge} \text{LENGTH}('a::\text{len}) \implies \text{of-nat } k \neq (0 :: 'a \text{ word})$
 $\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-add*: $\text{of-nat } a + \text{of-nat } b = \text{of-nat } (a + b)$
 $\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-mult*: $\text{of-nat } a * \text{of-nat } b = (\text{of-nat } (a * b) :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-Suc*: $\text{word-succ } (\text{of-nat } a) = \text{of-nat } (\text{Suc } a)$
 $\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-0*: $(0 :: 'a::\text{len word}) = \text{of-nat } 0$
 $\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-1*: $(1 :: 'a::\text{len word}) = \text{of-nat } (\text{Suc } 0)$
 $\langle \text{proof} \rangle$

lemmas *Abs-fnat-homs* =
Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc
Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*: $a + b = \text{of-nat } (\text{unat } a + \text{unat } b)$
 $\langle \text{proof} \rangle$

lemma *word-arith-nat-mult*: $a * b = \text{of-nat } (\text{unat } a * \text{unat } b)$
 $\langle \text{proof} \rangle$

lemma *word-arith-nat-Suc*: $\text{word-succ } a = \text{of-nat } (\text{Suc } (\text{unat } a))$
 $\langle \text{proof} \rangle$

lemma *word-arith-nat-div*: $a \text{ div } b = \text{of-nat } (\text{unat } a \text{ div } \text{unat } b)$
 $\langle \text{proof} \rangle$

lemma *word-arith-nat-mod*: $a \text{ mod } b = \text{of-nat } (\text{unat } a \text{ mod } \text{unat } b)$
 $\langle \text{proof} \rangle$

lemmas *word-arith-nat-defs* =
word-arith-nat-add word-arith-nat-mult
word-arith-nat-Suc Abs-fnat-hom-0
Abs-fnat-hom-1 word-arith-nat-div
word-arith-nat-mod

lemma *unat-of-nat*:
 $\langle \text{unat } (\text{word-of-nat } x :: 'a::\text{len word}) = x \text{ mod } 2^{\wedge} \text{LENGTH}('a) \rangle$
 $\langle \text{proof} \rangle$

lemma *unat-cong*: $x = y \implies \text{unat } x = \text{unat } y$
 ⟨*proof*⟩

lemmas *unat-word-ariths* = *word-arith-nat-defs*
 [THEN *trans* [OF *unat-cong unat-of-nat*]]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
 [unfolded *linorder-not-less* [symmetric] *Not-eq-iff*]

lemma *unat-add-lem*:
 $\text{unat } x + \text{unat } y < 2^{\wedge \text{LENGTH('a)}} \longleftrightarrow \text{unat } (x + y) = \text{unat } x + \text{unat } y$
 for $x \ y :: 'a::\text{len word}$
 ⟨*proof*⟩

lemma *unat-mult-lem*:
 $\text{unat } x * \text{unat } y < 2^{\wedge \text{LENGTH('a)}} \longleftrightarrow \text{unat } (x * y) = \text{unat } x * \text{unat } y$
 for $x \ y :: 'a::\text{len word}$
 ⟨*proof*⟩

lemma *le-no-overflow*: $x \leq b \implies a \leq a + b \implies x \leq a + b$
 for $a \ b \ x :: 'a::\text{len word}$
 ⟨*proof*⟩

lemma *uint-div*:
 $\langle \text{uint } (x \text{ div } y) = \text{uint } x \text{ div uint } y \rangle$
 ⟨*proof*⟩

lemma *uint-mod*:
 $\langle \text{uint } (x \text{ mod } y) = \text{uint } x \text{ mod uint } y \rangle$
 ⟨*proof*⟩

lemma *no-plus-overflow-unat-size*: $x \leq x + y \longleftrightarrow \text{unat } x + \text{unat } y < 2^{\wedge \text{size } x}$
 for $x \ y :: 'a::\text{len word}$
 ⟨*proof*⟩

lemmas *no-olen-add-nat* =
no-plus-overflow-unat-size [unfolded *word-size*]

lemmas *unat-plus-simple* =
trans [OF *no-olen-add-nat unat-add-lem*]

lemma *word-div-mult*: $\llbracket 0 < y; \text{unat } x * \text{unat } y < 2^{\wedge \text{LENGTH('a)}} \rrbracket \implies x * y \text{ div } y = x$
 for $x \ y :: 'a::\text{len word}$
 ⟨*proof*⟩

lemma *div-lt'*: $i \leq k \text{ div } x \implies \text{unat } i * \text{unat } x < 2^{\wedge \text{LENGTH('a)}}$
 for $i \ k \ x :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemmas $\text{div-lt}'' = \text{order-less-imp-le} \ [\text{THEN } \text{div-lt}]$

lemma div-lt-mult : $\llbracket i < k \text{ div } x; 0 < x \rrbracket \implies i * x < k$
for $i \ k \ x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma div-le-mult : $\llbracket i \leq k \text{ div } x; 0 < x \rrbracket \implies i * x \leq k$
for $i \ k \ x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma $\text{div-lt-uint}'$: $i \leq k \text{ div } x \implies \text{uint } i * \text{uint } x < 2^{\wedge} \text{LENGTH}('a)$
for $i \ k \ x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas $\text{div-lt-uint}'' = \text{order-less-imp-le} \ [\text{THEN } \text{div-lt-uint}]$

lemma $\text{word-le-exists}'$: $x \leq y \implies \exists z. y = x + z \wedge \text{uint } x + \text{uint } z < 2^{\wedge} \text{LENGTH}('a)$
for $x \ y \ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas $\text{plus-minus-not-NULL} = \text{order-less-imp-le} \ [\text{THEN } \text{plus-minus-not-NULL-ab}]$

lemmas $\text{plus-minus-no-overflow} =$
 $\text{order-less-imp-le} \ [\text{THEN } \text{plus-minus-no-overflow-ab}]$

lemmas $\text{mcs} = \text{word-less-minus-cancel} \ \text{word-less-minus-mono-left}$
 $\text{word-le-minus-cancel} \ \text{word-le-minus-mono-left}$

lemmas $\text{word-l-diffs} = \text{mcs} \ [\text{where } y = w + x, \text{ unfolded add-diff-cancel}]$ **for** $w \ x$
lemmas $\text{word-diff-ls} = \text{mcs} \ [\text{where } z = w + x, \text{ unfolded add-diff-cancel}]$ **for** $w \ x$
lemmas $\text{word-plus-mcs} = \text{word-diff-ls} \ [\text{where } y = v + x, \text{ unfolded add-diff-cancel}]$
for $v \ x$

lemma le-unat-uoi :
 $\langle y \leq \text{unat } z \implies \text{unat} (\text{word-of-nat } y :: 'a \text{ word}) = y \rangle$
for $z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas $\text{thd} = \text{times-div-less-eq-dividend}$

lemmas $\text{uno-simps} \ [\text{THEN } \text{le-unat-uoi}] = \text{mod-le-divisor} \ \text{div-le-dividend}$

lemma $\text{word-mod-div-equality}$: $(n \text{ div } b) * b + (n \text{ mod } b) = n$
for $n \ b :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-div-mult-le*: $a \text{ div } b * b \leq a$
for $a \ b :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-mod-less-divisor*: $0 < n \implies m \text{ mod } n < n$
for $m \ n :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-of-int-power-hom*: $\text{word-of-int } a \wedge n = (\text{word-of-int } (a \wedge n) :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemma *word-arith-power-alt*: $a \wedge n = (\text{word-of-int } (\text{uint } a \wedge n) :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemma *unatSuc*: $1 + n \neq 0 \implies \text{unat } (1 + n) = \text{Suc } (\text{unat } n)$
for $n :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

107.24 Cardinality, finiteness of set of words

lemma *inj-on-word-of-int*: $\langle \text{inj-on } (\text{word-of-int} :: \text{int} \Rightarrow 'a \text{ word}) \{0..<2^{\text{LENGTH}('a::\text{len})}\} \rangle$
 $\langle \text{proof} \rangle$

lemma *range-uint*: $\langle \text{range } (\text{uint} :: 'a \text{ word} \Rightarrow \text{int}) = \{0..<2^{\text{LENGTH}('a::\text{len})}\} \rangle$
 $\langle \text{proof} \rangle$

lemma *UNIV-eq*: $\langle (\text{UNIV} :: 'a \text{ word set}) = \text{word-of-int } ' \{0..<2^{\text{LENGTH}('a::\text{len})}\} \rangle$
 $\langle \text{proof} \rangle$

lemma *card-word*: $\text{CARD}('a \text{ word}) = 2^{\text{LENGTH}('a::\text{len})}$
 $\langle \text{proof} \rangle$

lemma *card-word-size*: $\text{CARD}('a \text{ word}) = 2^{\text{size } x}$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

end

instance *word* :: $(\text{len}) \text{ finite}$
 $\langle \text{proof} \rangle$

107.25 Bitwise Operations on Words

context
includes *bit-operations-syntax*
begin

lemma *word-wi-log-defs*:
 $\text{NOT } (\text{word-of-int } a) = \text{word-of-int } (\text{NOT } a)$

$\text{word-of-int } a \text{ AND } \text{word-of-int } b = \text{word-of-int } (a \text{ AND } b)$
 $\text{word-of-int } a \text{ OR } \text{word-of-int } b = \text{word-of-int } (a \text{ OR } b)$
 $\text{word-of-int } a \text{ XOR } \text{word-of-int } b = \text{word-of-int } (a \text{ XOR } b)$
 $\langle \text{proof} \rangle$

lemma *word-no-log-defs* [simp]:

$\text{NOT } (\text{numeral } a) = \text{word-of-int } (\text{NOT } (\text{numeral } a))$
 $\text{NOT } (- \text{numeral } a) = \text{word-of-int } (\text{NOT } (- \text{numeral } a))$
 $\text{numeral } a \text{ AND } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ AND } \text{numeral } b)$
 $\text{numeral } a \text{ AND } - \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ AND } - \text{numeral } b)$
 $- \text{numeral } a \text{ AND } \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ AND } \text{numeral } b)$
 $- \text{numeral } a \text{ AND } - \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ AND } - \text{numeral } b)$
 $\text{numeral } a \text{ OR } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ OR } \text{numeral } b)$
 $\text{numeral } a \text{ OR } - \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ OR } - \text{numeral } b)$
 $- \text{numeral } a \text{ OR } \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ OR } \text{numeral } b)$
 $- \text{numeral } a \text{ OR } - \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ OR } - \text{numeral } b)$
 $\text{numeral } a \text{ XOR } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ XOR } \text{numeral } b)$
 $\text{numeral } a \text{ XOR } - \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ XOR } - \text{numeral } b)$
 $- \text{numeral } a \text{ XOR } \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ XOR } \text{numeral } b)$
 $- \text{numeral } a \text{ XOR } - \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ XOR } - \text{numeral } b)$
 $\langle \text{proof} \rangle$

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [simp]:

$\text{NOT } (1::'a::\text{len word}) = -2$
 $1 \text{ AND } \text{numeral } b = \text{word-of-int } (1 \text{ AND } \text{numeral } b)$
 $1 \text{ AND } - \text{numeral } b = \text{word-of-int } (1 \text{ AND } - \text{numeral } b)$
 $\text{numeral } a \text{ AND } 1 = \text{word-of-int } (\text{numeral } a \text{ AND } 1)$
 $- \text{numeral } a \text{ AND } 1 = \text{word-of-int } (- \text{numeral } a \text{ AND } 1)$
 $1 \text{ OR } \text{numeral } b = \text{word-of-int } (1 \text{ OR } \text{numeral } b)$
 $1 \text{ OR } - \text{numeral } b = \text{word-of-int } (1 \text{ OR } - \text{numeral } b)$
 $\text{numeral } a \text{ OR } 1 = \text{word-of-int } (\text{numeral } a \text{ OR } 1)$
 $- \text{numeral } a \text{ OR } 1 = \text{word-of-int } (- \text{numeral } a \text{ OR } 1)$
 $1 \text{ XOR } \text{numeral } b = \text{word-of-int } (1 \text{ XOR } \text{numeral } b)$
 $1 \text{ XOR } - \text{numeral } b = \text{word-of-int } (1 \text{ XOR } - \text{numeral } b)$
 $\text{numeral } a \text{ XOR } 1 = \text{word-of-int } (\text{numeral } a \text{ XOR } 1)$
 $- \text{numeral } a \text{ XOR } 1 = \text{word-of-int } (- \text{numeral } a \text{ XOR } 1)$
 $\langle \text{proof} \rangle$

Special cases for when one of the arguments equals -1.

lemma *word-bitwise-m1-simps* [simp]:

$\text{NOT } (-1::'a::\text{len word}) = 0$
 $(-1::'a::\text{len word}) \text{ AND } x = x$
 $x \text{ AND } (-1::'a::\text{len word}) = x$
 $(-1::'a::\text{len word}) \text{ OR } x = -1$
 $x \text{ OR } (-1::'a::\text{len word}) = -1$
 $(-1::'a::\text{len word}) \text{ XOR } x = \text{NOT } x$
 $x \text{ XOR } (-1::'a::\text{len word}) = \text{NOT } x$
 $\langle \text{proof} \rangle$

lemma *word-of-int-not-numeral-eq* [simp]:

$\langle \text{word-of-int } (\text{NOT } (\text{numeral bin})) :: 'a::\text{len word} = - \text{numeral bin} - 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-and*:

$\langle \text{uint } (x \text{ AND } y) = \text{uint } x \text{ AND } \text{uint } y \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-or*:

$\langle \text{uint } (x \text{ OR } y) = \text{uint } x \text{ OR } \text{uint } y \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-xor*:

$\langle \text{uint } (x \text{ XOR } y) = \text{uint } x \text{ XOR } \text{uint } y \rangle$
 $\langle \text{proof} \rangle$

lemmas *bwsimps* =

wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:

$(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-bw-comms*:

$x \text{ AND } y = y \text{ AND } x$
 $x \text{ OR } y = y \text{ OR } x$
 $x \text{ XOR } y = y \text{ XOR } x$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-bw-lcs*:

$y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-log-esimps*:

$x \text{ AND } 0 = 0$
 $x \text{ AND } -1 = x$
 $x \text{ OR } 0 = x$
 $x \text{ OR } -1 = -1$
 $x \text{ XOR } 0 = x$
 $x \text{ XOR } -1 = \text{NOT } x$
 $0 \text{ AND } x = 0$

```

-1 AND x = x
0 OR x = x
-1 OR x = -1
0 XOR x = x
-1 XOR x = NOT x
for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-not-dist*:

```

NOT (x OR y) = NOT x AND NOT y
NOT (x AND y) = NOT x OR NOT y
for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-bw-same*:

```

x AND x = x
x OR x = x
x XOR x = 0
for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-ao-absorbs* [simp]:

```

x AND (y OR x) = x
x OR y AND x = x
x AND (x OR y) = x
y AND x OR x = x
(y OR x) AND x = x
x OR x AND y = x
(x OR y) AND x = x
x AND y OR x = x
for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-not-not* [simp]: $\text{NOT } (\text{NOT } x) = x$

```

for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-ao-dist*: $(x \text{ OR } y) \text{ AND } z = x \text{ AND } z \text{ OR } y \text{ AND } z$

```

for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-oa-dist*: $x \text{ AND } y \text{ OR } z = (x \text{ OR } z) \text{ AND } (y \text{ OR } z)$

```

for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-add-not* [simp]: $x + \text{NOT } x = -1$

```

for x :: 'a::len word
  ⟨proof⟩

```

lemma *word-plus-and-or* [*simp*]: $(x \text{ AND } y) + (x \text{ OR } y) = x + y$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *leoa*: $w = x \text{ OR } y \implies y = w \text{ AND } y$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *leao*: $w' = x' \text{ AND } y' \implies x' = x' \text{ OR } w'$
for $x' :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-ao-equiv*: $w = w \text{ OR } w' \longleftrightarrow w' = w \text{ AND } w'$
for $w w' :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *le-word-or2*: $x \leq x \text{ OR } y$
for $x y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemmas *le-word-or1* = *xtrans*(3) [*OF word-bw-comms* (2) *le-word-or2*]
lemmas *word-and-le1* = *xtrans*(3) [*OF word-ao-absorbs* (4) [*symmetric*] *le-word-or2*]
lemmas *word-and-le2* = *xtrans*(3) [*OF word-ao-absorbs* (8) [*symmetric*] *le-word-or2*]

lemma *bit-horner-sum-bit-word-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{horner-sum of-bool } (2 :: 'a::\text{len word}) \text{ } bs) \text{ } n$
 $\longleftrightarrow n < \min \text{LENGTH}('a) (\text{length } bs) \wedge bs ! n \rangle$
 $\langle \text{proof} \rangle$

definition *word-reverse* :: $\langle 'a::\text{len word} \Rightarrow 'a \text{ word} \rangle$
where $\langle \text{word-reverse } w = \text{horner-sum of-bool } 2 (\text{rev } (\text{map } (\text{bit } w) [0..<\text{LENGTH}('a)])) \rangle$

lemma *bit-word-reverse-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{word-reverse } w) \text{ } n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } w (\text{LENGTH}('a) - \text{Suc } n) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rev-rev* [*simp*] : $\text{word-reverse } (\text{word-reverse } w) = w$
 $\langle \text{proof} \rangle$

lemma *word-rev-gal*: $\text{word-reverse } w = u \implies \text{word-reverse } u = w$
 $\langle \text{proof} \rangle$

lemma *word-rev-gal'*: $u = \text{word-reverse } w \implies w = \text{word-reverse } u$
 $\langle \text{proof} \rangle$

lemma *word-eq-reverseI*:
 $\langle v = w \rangle \text{ if } \langle \text{word-reverse } v = \text{word-reverse } w \rangle$
 $\langle \text{proof} \rangle$

lemma *uint-2p*: $(0 :: 'a :: \text{len word}) < 2 \wedge n \implies \text{uint } (2 \wedge n :: 'a :: \text{len word}) = 2 \wedge n$
 ⟨proof⟩

lemma *word-of-int-2p*: $(\text{word-of-int } (2 \wedge n) :: 'a :: \text{len word}) = 2 \wedge n$
 ⟨proof⟩

107.25.1 shift functions in terms of lists of bools

lemma *drop-bit-word-numeral [simp]*:
 ⟨drop-bit (numeral n) (numeral k) =
 (word-of-int (drop-bit (numeral n) (take-bit LENGTH('a) (numeral k))) :: 'a :: len
 word)⟩
 ⟨proof⟩

lemma *drop-bit-word-Suc-numeral [simp]*:
 ⟨drop-bit (Suc n) (numeral k) =
 (word-of-int (drop-bit (Suc n) (take-bit LENGTH('a) (numeral k))) :: 'a :: len
 word)⟩
 ⟨proof⟩

lemma *drop-bit-word-minus-numeral [simp]*:
 ⟨drop-bit (numeral n) (− numeral k) =
 (word-of-int (drop-bit (numeral n) (take-bit LENGTH('a) (− numeral k))) ::
 'a :: len word)⟩
 ⟨proof⟩

lemma *drop-bit-word-Suc-minus-numeral [simp]*:
 ⟨drop-bit (Suc n) (− numeral k) =
 (word-of-int (drop-bit (Suc n) (take-bit LENGTH('a) (− numeral k))) :: 'a :: len
 word)⟩
 ⟨proof⟩

lemma *signed-drop-bit-word-numeral [simp]*:
 ⟨signed-drop-bit (numeral n) (numeral k) =
 (word-of-int (drop-bit (numeral n) (signed-take-bit (LENGTH('a) − 1) (numeral
 k))) :: 'a :: len word)⟩
 ⟨proof⟩

lemma *signed-drop-bit-word-Suc-numeral [simp]*:
 ⟨signed-drop-bit (Suc n) (numeral k) =
 (word-of-int (drop-bit (Suc n) (signed-take-bit (LENGTH('a) − 1) (numeral
 k))) :: 'a :: len word)⟩
 ⟨proof⟩

lemma *signed-drop-bit-word-minus-numeral [simp]*:
 ⟨signed-drop-bit (numeral n) (− numeral k) =
 (word-of-int (drop-bit (numeral n) (signed-take-bit (LENGTH('a) − 1) (−
 numeral k))) :: 'a :: len word)⟩

⟨proof⟩

lemma *signed-drop-bit-word-Suc-minus-numeral* [simp]:

⟨signed-drop-bit (Suc n) (− numeral k) =
 (word-of-int (drop-bit (Suc n) (signed-take-bit (LENGTH('a) − 1) (− numeral
 k))) :: 'a::len word)⟩
 ⟨proof⟩

lemma *take-bit-word-numeral* [simp]:

⟨take-bit (numeral n) (numeral k) =
 (word-of-int (take-bit (min LENGTH('a) (numeral n)) (numeral k)) :: 'a::len
 word)⟩
 ⟨proof⟩

lemma *take-bit-word-Suc-numeral* [simp]:

⟨take-bit (Suc n) (numeral k) =
 (word-of-int (take-bit (min LENGTH('a) (Suc n)) (numeral k)) :: 'a::len word)⟩
 ⟨proof⟩

lemma *take-bit-word-minus-numeral* [simp]:

⟨take-bit (numeral n) (− numeral k) =
 (word-of-int (take-bit (min LENGTH('a) (numeral n)) (− numeral k)) :: 'a::len
 word)⟩
 ⟨proof⟩

lemma *take-bit-word-Suc-minus-numeral* [simp]:

⟨take-bit (Suc n) (− numeral k) =
 (word-of-int (take-bit (min LENGTH('a) (Suc n)) (− numeral k)) :: 'a::len
 word)⟩
 ⟨proof⟩

lemma *signed-take-bit-word-numeral* [simp]:

⟨signed-take-bit (numeral n) (numeral k) =
 (word-of-int (signed-take-bit (numeral n) (take-bit LENGTH('a) (numeral k)))
 :: 'a::len word)⟩
 ⟨proof⟩

lemma *signed-take-bit-word-Suc-numeral* [simp]:

⟨signed-take-bit (Suc n) (numeral k) =
 (word-of-int (signed-take-bit (Suc n) (take-bit LENGTH('a) (numeral k))) ::
 'a::len word)⟩
 ⟨proof⟩

lemma *signed-take-bit-word-minus-numeral* [simp]:

⟨signed-take-bit (numeral n) (− numeral k) =
 (word-of-int (signed-take-bit (numeral n) (take-bit LENGTH('a) (− numeral
 k))) :: 'a::len word)⟩
 ⟨proof⟩

lemma *signed-take-bit-word-Suc-minus-numeral* [simp]:

⟨signed-take-bit (Suc n) (− numeral k) =
 (word-of-int (signed-take-bit (Suc n) (take-bit LENGTH('a) (− numeral k))) ::
 'a::len word)⟩
 ⟨proof⟩

lemma *False-map2-or*: $\llbracket \text{set } xs \subseteq \{\text{False}\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\vee) \text{ } xs$
 $ys = ys$
 ⟨proof⟩

lemma *align-lem-or*:

assumes $\text{length } xs = n + m$ $\text{length } ys = n + m$
and $\text{drop } m \text{ } xs = \text{replicate } n \text{ False take } m \text{ } ys = \text{replicate } m \text{ False}$
shows $\text{map2 } (\vee) \text{ } xs \text{ } ys = \text{take } m \text{ } xs @ \text{drop } m \text{ } ys$
 ⟨proof⟩

lemma *False-map2-and*: $\llbracket \text{set } xs \subseteq \{\text{False}\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\wedge)$
 $xs \text{ } ys = xs$
 ⟨proof⟩

lemma *align-lem-and*:

assumes $\text{length } xs = n + m$ $\text{length } ys = n + m$
and $\text{drop } m \text{ } xs = \text{replicate } n \text{ False take } m \text{ } ys = \text{replicate } m \text{ False}$
shows $\text{map2 } (\wedge) \text{ } xs \text{ } ys = \text{replicate } (n + m) \text{ False}$
 ⟨proof⟩

107.25.2 Mask

lemma *minus-1-eq-mask*:

⟨− 1 = (mask LENGTH('a) :: 'a::len word)⟩
 ⟨proof⟩

lemma *mask-eq-decr-exp*:

⟨mask n = 2 ^ n − (1 :: 'a::len word)⟩
 ⟨proof⟩

lemma *mask-Suc-rec*:

⟨mask (Suc n) = 2 * mask n + (1 :: 'a::len word)⟩
 ⟨proof⟩

context

begin

qualified lemma *bit-mask-iff* [bit-simps]:

⟨bit (mask m :: 'a::len word) n \longleftrightarrow n < min LENGTH('a) m⟩
 ⟨proof⟩

end

lemma *mask-bin*: $\text{mask } n = \text{word-of-int } (\text{take-bit } n \text{ } (- 1))$
 ⟨proof⟩

lemma *and-mask-bintr*: $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \text{ } (\text{uint } w))$
 ⟨proof⟩

lemma *and-mask-wi*: $\text{word-of-int } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \text{ } i)$
 ⟨proof⟩

lemma *and-mask-wi'*:
 $\text{word-of-int } i \text{ AND } \text{mask } n = (\text{word-of-int } (\text{take-bit } (\text{min } \text{LENGTH}('a) \text{ } n) \text{ } i) :: 'a::\text{len word})$
 ⟨proof⟩

lemma *and-mask-no*: $\text{numeral } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \text{ } (\text{numeral } i))$
 ⟨proof⟩

lemma *and-mask-mod-2p*: $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{uint } w \text{ mod } 2^{\wedge} n)$
 ⟨proof⟩

lemma *uint-mask-eq*:
 $\langle \text{uint } (\text{mask } n :: 'a::\text{len word}) = \text{mask } (\text{min } \text{LENGTH}('a) \text{ } n) \rangle$
 ⟨proof⟩

lemma *and-mask-lt-2p*: $\text{uint } (w \text{ AND } \text{mask } n) < 2^{\wedge} n$
 ⟨proof⟩

lemma *mask-eq-iff*: $w \text{ AND } \text{mask } n = w \longleftrightarrow \text{uint } w < 2^{\wedge} n$
 ⟨proof⟩

lemma *and-mask-dvd*: $2^{\wedge} n \text{ dvd } \text{uint } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$
 ⟨proof⟩

lemma *and-mask-dvd-nat*: $2^{\wedge} n \text{ dvd } \text{unat } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$
 ⟨proof⟩

lemma *word-2p-lem*: $n < \text{size } w \implies w < 2^{\wedge} n = (\text{uint } w < 2^{\wedge} n)$
for $w :: 'a::\text{len word}$
 ⟨proof⟩

lemma *less-mask-eq*:
fixes $x :: 'a::\text{len word}$
assumes $x < 2^{\wedge} n$ **shows** $x \text{ AND } \text{mask } n = x$
 ⟨proof⟩

lemmas *mask-eq-iff-w2p* = *trans* [*OF* *mask-eq-iff* *word-2p-lem* [*symmetric*]]

lemmas *and-mask-less'* = *iffD2* [*OF* *word-2p-lem* *and-mask-lt-2p*, *simplified word-size*]

lemma *and-mask-less-size*: $n < \text{size } x \implies x \text{ AND mask } n < 2^n$
for $x :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-mod-2p-is-mask* [*OF refl*]: $c = 2^n \implies c > 0 \implies x \bmod c = x \text{ AND mask } n$
for $c \ x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *mask-egs*:

$(a \text{ AND mask } n) + b \text{ AND mask } n = a + b \text{ AND mask } n$
 $a + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$
 $(a \text{ AND mask } n) - b \text{ AND mask } n = a - b \text{ AND mask } n$
 $a - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$
 $a * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$
 $(b \text{ AND mask } n) * a \text{ AND mask } n = b * a \text{ AND mask } n$
 $(a \text{ AND mask } n) + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$
 $(a \text{ AND mask } n) - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$
 $(a \text{ AND mask } n) * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$
 $-(a \text{ AND mask } n) \text{ AND mask } n = -a \text{ AND mask } n$
 $\text{word-succ } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-succ } a \text{ AND mask } n$
 $\text{word-pred } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-pred } a \text{ AND mask } n$
 $\langle \text{proof} \rangle$

lemma *mask-power-eg*: $(x \text{ AND mask } n)^k \text{ AND mask } n = x^k \text{ AND mask } n$
for $x :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *mask-full* [*simp*]: $\text{mask LENGTH}('a) = (-1 :: 'a::\text{len word})$
 $\langle \text{proof} \rangle$

107.25.3 Slices

definition *slice1* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$

where $\langle \text{slice1 } n \ w = (\text{if } n < \text{LENGTH}('a)$
 $\text{then } \text{ucast } (\text{drop-bit } (\text{LENGTH}('a) - n) \ w)$
 $\text{else } \text{push-bit } (n - \text{LENGTH}('a)) \ (\text{ucast } w)) \rangle$

lemma *bit-slice1-iff* [*bit-simps*]:

$\langle \text{bit } (\text{slice1 } m \ w :: 'b::\text{len word}) \ n \longleftrightarrow m - \text{LENGTH}('a) \leq n \wedge n < \min$
 $\text{LENGTH}('b) \ m$
 $\wedge \text{bit } w \ (n + (\text{LENGTH}('a) - m) - (m - \text{LENGTH}('a))) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

definition *slice* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$

where $\langle \text{slice } n = \text{slice1 } (\text{LENGTH}('a) - n) \rangle$

lemma *bit-slice-iff* [*bit-simps*]:

⟨*bit* (*slice* *m w* :: '*b*::*len word*) $n \longleftrightarrow n < \min \text{LENGTH}('b) (\text{LENGTH}('a) - m) \wedge \text{bit } w (n + \text{LENGTH}('a) - (\text{LENGTH}('a) - m))$ ⟩
for *w* :: '*a*::*len word*⟩
 ⟨*proof*⟩

lemma *slice1-0* [*simp*] : *slice1* *n 0* = 0

⟨*proof*⟩

lemma *slice-0* [*simp*] : *slice* *n 0* = 0

⟨*proof*⟩

lemma *ucast-slice1*: *ucast w* = *slice1* (*size w*) *w*

⟨*proof*⟩

lemma *ucast-slice*: *ucast w* = *slice* 0 *w*

⟨*proof*⟩

lemma *slice-id*: *slice* 0 *t* = *t*

⟨*proof*⟩

lemma *rev-slice1*:

⟨*slice1* *n* (*word-reverse w* :: '*b*::*len word*) = *word-reverse* (*slice1* *k w* :: '*a*::*len word*)⟩
if $\langle n + k = \text{LENGTH}('a) + \text{LENGTH}('b) \rangle$
 ⟨*proof*⟩

lemma *rev-slice*:

$n + k + \text{LENGTH}('a::\text{len}) = \text{LENGTH}('b::\text{len}) \implies$
 $\text{slice } n (\text{word-reverse } (w::'b \text{ word})) = \text{word-reverse } (\text{slice } k w :: 'a \text{ word})$
 ⟨*proof*⟩

107.25.4 Revcast

definition *revcast* :: '*a*::*len word* \Rightarrow '*b*::*len word*⟩

where $\langle \text{revcast} = \text{slice1 } \text{LENGTH}('b) \rangle$

lemma *bit-revcast-iff* [*bit-simps*]:

⟨*bit* (*revcast w* :: '*b*::*len word*) $n \longleftrightarrow \text{LENGTH}('b) - \text{LENGTH}('a) \leq n \wedge n < \text{LENGTH}('b)$ ⟩
 $\wedge \text{bit } w (n + (\text{LENGTH}('a) - \text{LENGTH}('b)) - (\text{LENGTH}('b) - \text{LENGTH}('a))) \rangle$
for *w* :: '*a*::*len word*⟩
 ⟨*proof*⟩

lemma *revcast-slice1* [*OF refl*]: *rc* = *revcast w* $\implies \text{slice1 } (\text{size } rc) w = rc$

⟨*proof*⟩

lemma *revcast-rev-ucast* [*OF refl refl refl*]:

$cs = [rc, uc] \implies rc = \text{revcast } (\text{word-reverse } w) \implies uc = \text{ucast } w \implies$

$rc = \text{word-reverse } uc$
 $\langle \text{proof} \rangle$

lemma *revcast-ucast*: $\text{revcast } w = \text{word-reverse } (\text{ucast } (\text{word-reverse } w))$
 $\langle \text{proof} \rangle$

lemma *ucast-revcast*: $\text{ucast } w = \text{word-reverse } (\text{revcast } (\text{word-reverse } w))$
 $\langle \text{proof} \rangle$

lemma *ucast-rev-revcast*: $\text{ucast } (\text{word-reverse } w) = \text{word-reverse } (\text{revcast } w)$
 $\langle \text{proof} \rangle$

linking revcast and cast via shift

lemmas *wsst-TYs* = *source-size target-size word-size*

lemmas *sym-notr* =
 $\text{not-iff } [\text{THEN iffD2}, \text{ THEN not-sym}, \text{ THEN not-iff } [\text{THEN iffD1}]]$

107.26 Split and cat

lemmas *word-split-bin'* = *word-split-def*

lemmas *word-cat-bin'* = *word-cat-eq*

— this odd result is analogous to *ucast-id*, result to the length given by the result type

lemma *word-cat-id*: $\text{word-cat } a \ b = b$
 $\langle \text{proof} \rangle$

lemma *word-cat-split-alt*: $\llbracket \text{size } w \leq \text{size } u + \text{size } v; \text{ word-split } w = (u, v) \rrbracket \implies \text{word-cat } u \ v = w$
 $\langle \text{proof} \rangle$

lemmas *word-cat-split-size* = *sym* [THEN [2] *word-cat-split-alt* [symmetric]]

107.26.1 Split and slice

lemma *split-slices*:
assumes $\text{word-split } w = (u, v)$
shows $u = \text{slice } (\text{size } v) \ w \wedge v = \text{slice } 0 \ w$
 $\langle \text{proof} \rangle$

lemma *slice-cat1* [OF *refl*]:
 $\llbracket \text{wc} = \text{word-cat } a \ b; \text{ size } a + \text{size } b \leq \text{size } \text{wc} \rrbracket \implies \text{slice } (\text{size } b) \ \text{wc} = a$
 $\langle \text{proof} \rangle$

lemmas *slice-cat2* = *trans* [OF *slice-id word-cat-id*]

lemma *cat-slices*:

$\llbracket a = \text{slice } n \ c; b = \text{slice } 0 \ c; n = \text{size } b; \text{size } c \leq \text{size } a + \text{size } b \rrbracket \implies \text{word-cat } a \ b = c$
 ⟨proof⟩

lemma *word-split-cat-alt*:

assumes $w = \text{word-cat } u \ v$ **and** *size*: $\text{size } u + \text{size } v \leq \text{size } w$

shows $\text{word-split } w = (u, v)$

⟨proof⟩

lemma *horner-sum-uint-exp-Cons-eq*:

$\langle \text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \ (w \# \text{ws}) =$

$\text{concat-bit LENGTH}('a) \ (\text{uint } w) \ (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \ \text{ws}) \rangle$

for $\text{ws} :: \langle 'a :: \text{len word list} \rangle$

⟨proof⟩

lemma *bit-horner-sum-uint-exp-iff*:

$\langle \text{bit } (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \ \text{ws}) \ n \longleftrightarrow$

$n \text{ div LENGTH}('a) < \text{length } \text{ws} \wedge \text{bit } (\text{ws} ! (n \text{ div LENGTH}('a))) \ (n \text{ mod LENGTH}('a)) \rangle$

for $\text{ws} :: \langle 'a :: \text{len word list} \rangle$

⟨proof⟩

107.27 Rotation

lemma *word-rotr-word-rotr-eq*: $\langle \text{word-rotr } m \ (\text{word-rotr } n \ w) = \text{word-rotr } (m + n) \ w \rangle$

⟨proof⟩

lemma *word-rot-lem*: $\llbracket l + k = d + k \text{ mod } l; n < l \rrbracket \implies ((d + n) \text{ mod } l) = n$ **for** $l :: \text{nat}$

⟨proof⟩

lemma *word-rot-rl [simp]*: $\langle \text{word-rotl } k \ (\text{word-rotr } k \ v) = v \rangle$

⟨proof⟩

lemma *word-rot-lr [simp]*: $\langle \text{word-rotr } k \ (\text{word-rotl } k \ v) = v \rangle$

⟨proof⟩

lemma *word-rot-gal*:

$\langle \text{word-rotr } n \ v = w \longleftrightarrow \text{word-rotl } n \ w = v \rangle$

⟨proof⟩

lemma *word-rot-gal'*:

$\langle w = \text{word-rotr } n \ v \longleftrightarrow v = \text{word-rotl } n \ w \rangle$

⟨proof⟩

lemma *word-reverse-word-rotl*:

$\langle \text{word-reverse } (\text{word-rotl } n \ w) = \text{word-rotr } n \ (\text{word-reverse } w) \rangle$ (**is** $\langle ?lhs = ?rhs \rangle$)

⟨proof⟩

lemma *word-reverse-word-rotr*:

$\langle \text{word-reverse } (\text{word-rotr } n \ w) = \text{word-rotl } n \ (\text{word-reverse } w) \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rotl-rev*:

$\langle \text{word-rotl } n \ w = \text{word-reverse } (\text{word-rotr } n \ (\text{word-reverse } w)) \rangle$
 $\langle \text{proof} \rangle$

lemma *word-rotr-rev*:

$\langle \text{word-rotr } n \ w = \text{word-reverse } (\text{word-rotl } n \ (\text{word-reverse } w)) \rangle$
 $\langle \text{proof} \rangle$

lemma *word-roti-0 [simp]*: $\text{word-roti } 0 \ w = w$

$\langle \text{proof} \rangle$

lemma *word-roti-add*: $\text{word-roti } (m + n) \ w = \text{word-roti } m \ (\text{word-roti } n \ w)$

$\langle \text{proof} \rangle$

lemma *word-roti-conv-mod'*:

$\text{word-roti } n \ w = \text{word-roti } (n \bmod \text{int } (\text{size } w)) \ w$
 $\langle \text{proof} \rangle$

lemmas *word-roti-conv-mod = word-roti-conv-mod'* [unfolded word-size]

end

107.27.1 "Word rotation commutes with bit-wise operations

locale *word-rotate*

begin

context

includes *bit-operations-syntax*

begin

lemma *word-rot-logs*:

$\text{word-rotl } n \ (\text{NOT } v) = \text{NOT } (\text{word-rotl } n \ v)$
 $\text{word-rotr } n \ (\text{NOT } v) = \text{NOT } (\text{word-rotr } n \ v)$
 $\text{word-rotl } n \ (x \ \text{AND } y) = \text{word-rotl } n \ x \ \text{AND } \text{word-rotl } n \ y$
 $\text{word-rotr } n \ (x \ \text{AND } y) = \text{word-rotr } n \ x \ \text{AND } \text{word-rotr } n \ y$
 $\text{word-rotl } n \ (x \ \text{OR } y) = \text{word-rotl } n \ x \ \text{OR } \text{word-rotl } n \ y$
 $\text{word-rotr } n \ (x \ \text{OR } y) = \text{word-rotr } n \ x \ \text{OR } \text{word-rotr } n \ y$
 $\text{word-rotl } n \ (x \ \text{XOR } y) = \text{word-rotl } n \ x \ \text{XOR } \text{word-rotl } n \ y$
 $\text{word-rotr } n \ (x \ \text{XOR } y) = \text{word-rotr } n \ x \ \text{XOR } \text{word-rotr } n \ y$
 $\langle \text{proof} \rangle$

end

end

lemmas *word-rot-logs* = *word-rotate.word-rot-logs*

lemma *word-rotx-0* [*simp*] : *word-rotl i 0 = 0* \wedge *word-rotl i 0 = 0*
 ⟨*proof*⟩

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*
 ⟨*proof*⟩

declare *word-roti-eq-word-rotl-word-rotl* [*simp*]

107.28 Maximum machine word

context

includes *bit-operations-syntax*

begin

lemma *word-int-cases*:
fixes *x* :: '*a*::len word
obtains *n* **where** *x* = *word-of-int n* **and** $0 \leq n$ **and** $n < 2^{\text{LENGTH}('a)}$
 ⟨*proof*⟩

lemma *word-nat-cases* [*cases type: word*]:
fixes *x* :: '*a*::len word
obtains *n* **where** *x* = *of-nat n* **and** $n < 2^{\text{LENGTH}('a)}$
 ⟨*proof*⟩

lemma *max-word-max* [*intro!*]:
 $\langle n \leq -1 \rangle$ **for** *n* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-of-int-2p-len*: *word-of-int* ($2^{\text{LENGTH}('a)}$) = (*0*::'*a*::len word)
 ⟨*proof*⟩

lemma *word-pow-0*: ($2^{\text{LENGTH}('a)}$) \wedge *LENGTH*('a) = 0
 ⟨*proof*⟩

lemma *max-word-wrap*:
 $\langle x + 1 = 0 \implies x = -1 \rangle$ **for** *x* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-and-max*:
 $\langle x \text{ AND } -1 = x \rangle$ **for** *x* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-or-max*:
 $\langle x \text{ OR } -1 = -1 \rangle$ **for** *x* :: '*a*::len word
 ⟨*proof*⟩

lemma *word-ao-dist2*: $x \text{ AND } (y \text{ OR } z) = x \text{ AND } y \text{ OR } x \text{ AND } z$
for $x \ y \ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-oa-dist2*: $x \text{ OR } y \text{ AND } z = (x \text{ OR } y) \text{ AND } (x \text{ OR } z)$
for $x \ y \ z :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-and-not* [simp]: $x \text{ AND NOT } x = 0$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-or-not* [simp]:
 $\langle x \text{ OR NOT } x = -1 \rangle$ **for** $x :: \langle 'a::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-xor-and-or*: $x \text{ XOR } y = x \text{ AND NOT } y \text{ OR NOT } x \text{ AND } y$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *uint-lt-0* [simp]: $\text{uint } x < 0 = \text{False}$
 $\langle \text{proof} \rangle$

lemma *word-less-1* [simp]: $x < 1 \longleftrightarrow x = 0$
for $x :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *uint-plus-if-size*:
 $\text{uint } (x + y) =$
 $(\text{if } \text{uint } x + \text{uint } y < 2^{\text{size } x}$
 $\text{then } \text{uint } x + \text{uint } y$
 $\text{else } \text{uint } x + \text{uint } y - 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemma *unat-plus-if-size*:
 $\text{unat } (x + y) =$
 $(\text{if } \text{unat } x + \text{unat } y < 2^{\text{size } x}$
 $\text{then } \text{unat } x + \text{unat } y$
 $\text{else } \text{unat } x + \text{unat } y - 2^{\text{size } x})$
for $x \ y :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *word-neq-0-conv*: $w \neq 0 \longleftrightarrow 0 < w$
for $w :: 'a::\text{len word}$
 $\langle \text{proof} \rangle$

lemma *max-lt*: $\text{unat } (\max a \ b \ \text{div } c) = \text{unat } (\max a \ b) \ \text{div } \text{unat } c$
for $c :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *uint-sub-if-size*:

$\text{uint } (x - y) =$
 $(\text{if } \text{uint } y \leq \text{uint } x$
 $\text{then } \text{uint } x - \text{uint } y$
 $\text{else } \text{uint } x - \text{uint } y + 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemma *unat-sub*:

$\langle \text{unat } (a - b) = \text{unat } a - \text{unat } b \rangle$
if $\langle b \leq a \rangle$
 $\langle \text{proof} \rangle$

lemmas *word-less-sub1-numberof* [simp] = *word-less-sub1* [of numeral w] **for** w

lemmas *word-le-sub1-numberof* [simp] = *word-le-sub1* [of numeral w] **for** w

lemma *word-of-int-minus*: $\text{word-of-int } (2^{\text{LENGTH('a)}} - i) = (\text{word-of-int } (-i))::'a::\text{len word}$

$\langle \text{proof} \rangle$

lemma *word-of-int-inj*:

$\langle (\text{word-of-int } x :: 'a::\text{len word}) = \text{word-of-int } y \longleftrightarrow x = y \rangle$
if $\langle 0 \leq x \wedge x < 2^{\text{LENGTH('a)}} \rangle \langle 0 \leq y \wedge y < 2^{\text{LENGTH('a)}} \rangle$
 $\langle \text{proof} \rangle$

lemma *word-le-less-eq*: $x \leq y \longleftrightarrow x = y \vee x < y$

for $x \ y :: 'z::\text{len word}$

$\langle \text{proof} \rangle$

lemma *mod-plus-cong*:

fixes $b \ b' :: \text{int}$
assumes $1: b = b'$
 and $2: x \text{ mod } b' = x' \text{ mod } b'$
 and $3: y \text{ mod } b' = y' \text{ mod } b'$
 and $4: x' + y' = z'$
shows $(x + y) \text{ mod } b = z' \text{ mod } b'$
 $\langle \text{proof} \rangle$

lemma *mod-minus-cong*:

fixes $b \ b' :: \text{int}$
assumes $b = b'$
 and $x \text{ mod } b' = x' \text{ mod } b'$
 and $y \text{ mod } b' = y' \text{ mod } b'$
 and $x' - y' = z'$
shows $(x - y) \text{ mod } b = z' \text{ mod } b'$
 $\langle \text{proof} \rangle$

lemma *word-induct-less* [case-names zero less]:

$\langle P\ m \rangle$ **if** zero: $\langle P\ 0 \rangle$ **and** less: $\langle \bigwedge n. n < m \implies P\ n \implies P\ (1 + n) \rangle$
for $m :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-induct*: $P\ 0 \implies (\bigwedge n. P\ n \implies P\ (1 + n)) \implies P\ m$
for $P :: 'a::len\ word \Rightarrow bool$
 $\langle proof \rangle$

lemma *word-induct2* [case-names zero suc, induct type]: $P\ 0 \implies (\bigwedge n. 1 + n \neq 0 \implies P\ n \implies P\ (1 + n)) \implies P\ n$
for $P :: 'b::len\ word \Rightarrow bool$
 $\langle proof \rangle$

107.29 Recursion combinator for words

definition *word-rec* :: $'a \Rightarrow ('b::len\ word \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b\ word \Rightarrow 'a$
where *word-rec* forZero forSuc $n = rec\ nat\ forZero\ (forSuc \circ of\ nat)\ (unat\ n)$

lemma *word-rec-0* [simp]: $word\ rec\ z\ s\ 0 = z$
 $\langle proof \rangle$

lemma *word-rec-Suc* [simp]: $1 + n \neq 0 \implies word\ rec\ z\ s\ (1 + n) = s\ n\ (word\ rec\ z\ s\ n)$
for $n :: 'a::len\ word$
 $\langle proof \rangle$

lemma *word-rec-Pred*: $n \neq 0 \implies word\ rec\ z\ s\ n = s\ (n - 1)\ (word\ rec\ z\ s\ (n - 1))$
 $\langle proof \rangle$

lemma *word-rec-in*: $f\ (word\ rec\ z\ (\lambda\ -. f)\ n) = word\ rec\ (f\ z)\ (\lambda\ -. f)\ n$
 $\langle proof \rangle$

lemma *word-rec-in2*: $f\ n\ (word\ rec\ z\ f\ n) = word\ rec\ (f\ 0\ z)\ (f \circ (+)\ 1)\ n$
 $\langle proof \rangle$

lemma *word-rec-twice*:
 $m \leq n \implies word\ rec\ z\ f\ n = word\ rec\ (word\ rec\ z\ f\ (n - m))\ (f \circ (+)\ (n - m))\ m$
 $\langle proof \rangle$

lemma *word-rec-id*: $word\ rec\ z\ (\lambda\ -. id)\ n = z$
 $\langle proof \rangle$

lemma *word-rec-id-eq*: $(\bigwedge m. m < n \implies f\ m = id) \implies word\ rec\ z\ f\ n = z$
 $\langle proof \rangle$

lemma *word-rec-max*:
assumes $\forall m \geq n. m \neq -1 \longrightarrow f\ m = id$

shows $\text{word-rec } z \ f \ (-\ 1) = \text{word-rec } z \ f \ n$
 $\langle \text{proof} \rangle$

end

107.30 Some more naive computations rules

lemma *drop-bit-of-minus-1-eq* [simp]:
 $\langle \text{drop-bit } n \ (-\ 1 :: 'a::\text{len word}) = \text{mask } (\text{LENGTH}('a) - n) \rangle$
 $\langle \text{proof} \rangle$

context
includes *bit-operations-syntax*
begin

lemma *word-cat-eq-push-bit-or*:
 $\langle \text{word-cat } v \ w = (\text{push-bit } \text{LENGTH}('b) \ (\text{ucast } v) \ \text{OR } \text{ucast } w :: 'c::\text{len word}) \rangle$
for $v :: \langle 'a::\text{len word} \rangle$ **and** $w :: \langle 'b::\text{len word} \rangle$
 $\langle \text{proof} \rangle$

end

context *semiring-bit-operations*
begin

lemma *of-nat-take-bit-numeral-eq* [simp]:
 $\langle \text{of-nat } (\text{take-bit } m \ (\text{numeral } n)) = \text{take-bit } m \ (\text{numeral } n) \rangle$
 $\langle \text{proof} \rangle$

end

context *ring-bit-operations*
begin

lemma *signed-take-bit-of-int*:
 $\langle \text{signed-take-bit } n \ (\text{of-int } k) = \text{of-int } (\text{signed-take-bit } n \ k) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-signed-take-bit*:
 $\langle \text{of-int } (\text{signed-take-bit } n \ k) = \text{signed-take-bit } n \ (\text{of-int } k) \rangle$
 $\langle \text{proof} \rangle$

lemma *of-int-take-bit-minus-numeral-eq* [simp]:
 $\langle \text{of-int } (\text{take-bit } m \ (\text{numeral } n)) = \text{take-bit } m \ (\text{numeral } n) \rangle$
 $\langle \text{of-int } (\text{take-bit } m \ (-\ \text{numeral } n)) = \text{take-bit } m \ (-\ \text{numeral } n) \rangle$
 $\langle \text{proof} \rangle$

end

context

includes *bit-operations-syntax*

begin

lemma *concat-bit-numeral-of-one-1* [simp]:

⟨*concat-bit* (numeral *m*) 1 *l* = 1 OR *push-bit* (numeral *m*) *l*⟩
 ⟨*proof*⟩

lemma *concat-bit-of-one-2* [simp]:

⟨*concat-bit* *n k* 1 = *set-bit n* (*take-bit n k*)⟩
 ⟨*proof*⟩

lemma *concat-bit-numeral-of-minus-one-1* [simp]:

⟨*concat-bit* (numeral *m*) (− 1) *l* = *push-bit* (numeral *m*) *l* OR *mask* (numeral *m*)⟩
 ⟨*proof*⟩

lemma *concat-bit-numeral-of-minus-one-2* [simp]:

⟨*concat-bit* (numeral *m*) *k* (− 1) = *take-bit* (numeral *m*) *k* OR NOT (*mask* (numeral *m*))⟩
 ⟨*proof*⟩

lemma *concat-bit-numeral* [simp]:

⟨*concat-bit* (numeral *m*) (numeral *n*) (numeral *q*) = *take-bit* (numeral *m*) (numeral *n*) OR *push-bit* (numeral *m*) (numeral *q*)⟩
 ⟨*concat-bit* (numeral *m*) (− numeral *n*) (numeral *q*) = *take-bit* (numeral *m*) (− numeral *n*) OR *push-bit* (numeral *m*) (numeral *q*)⟩
 ⟨*concat-bit* (numeral *m*) (numeral *n*) (− numeral *q*) = *take-bit* (numeral *m*) (numeral *n*) OR *push-bit* (numeral *m*) (− numeral *q*)⟩
 ⟨*concat-bit* (numeral *m*) (− numeral *n*) (− numeral *q*) = *take-bit* (numeral *m*) (− numeral *n*) OR *push-bit* (numeral *m*) (− numeral *q*)⟩
 ⟨*proof*⟩

end

lemma *word-cat-0-left* [simp]:

⟨*word-cat* 0 *w* = *ucast w*⟩
 ⟨*proof*⟩

107.31 Executable intervals

instance *word* :: (len) ⟨{*interval-top*, *interval-bot*}⟩
 ⟨*proof*⟩

107.32 Tool support

⟨*ML*⟩

end

108 The Field of Integers mod 2

```
theory Z2
imports Main
begin
```

Note that in most cases *bool* is appropriate when a binary type is needed; the type provided here, for historical reasons named *bit*, is only needed if proper field operations are required.

```
typedef bit = ⟨UNIV :: bool set⟩ ⟨proof⟩
```

```
instantiation bit :: zero-neg-one
begin
```

```
definition zero-bit :: bit
  where ⟨0 = Abs-bit False⟩
```

```
definition one-bit :: bit
  where ⟨1 = Abs-bit True⟩
```

```
instance
  ⟨proof⟩
```

```
end
```

```
free-constructors case-bit for ⟨0::bit⟩ | ⟨1::bit⟩
  ⟨proof⟩
```

```
lemma bit-not-zero-iff [simp]:
  ⟨a ≠ 0 ⟷ a = 1⟩ for a :: bit
  ⟨proof⟩
```

```
lemma bit-not-one-iff [simp]:
  ⟨a ≠ 1 ⟷ a = 0⟩ for a :: bit
  ⟨proof⟩
```

```
instantiation bit :: semidom-modulo
begin
```

```
definition plus-bit :: ⟨bit ⇒ bit ⇒ bit⟩
  where ⟨a + b = Abs-bit (Rep-bit a ≠ Rep-bit b)⟩
```

```
definition minus-bit :: ⟨bit ⇒ bit ⇒ bit⟩
  where [simp]: ⟨minus-bit = plus⟩
```

```
definition times-bit :: ⟨bit ⇒ bit ⇒ bit⟩
  where ⟨a * b = Abs-bit (Rep-bit a ∧ Rep-bit b)⟩
```

```
definition divide-bit :: ⟨bit ⇒ bit ⇒ bit⟩
```

```

where [simp]:  $\langle \text{divide-bit} = \text{times} \rangle$ 

definition modulo-bit ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$ 
  where  $\langle a \bmod b = \text{Abs-bit } (\text{Rep-bit } a \wedge \neg \text{Rep-bit } b) \rangle$ 

instance
   $\langle \text{proof} \rangle$ 

end

lemma bit-2-eq-0 [simp]:
   $\langle 2 = (0::\text{bit}) \rangle$ 
   $\langle \text{proof} \rangle$ 

instance bit :: semiring-parity
   $\langle \text{proof} \rangle$ 

lemma Abs-bit-eq-of-bool [code-abbrev]:
   $\langle \text{Abs-bit} = \text{of-bool} \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma Rep-bit-eq-odd:
   $\langle \text{Rep-bit} = \text{odd} \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma Rep-bit-iff-odd [code-abbrev]:
   $\langle \text{Rep-bit } b \longleftrightarrow \text{odd } b \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma Not-Rep-bit-iff-even [code-abbrev]:
   $\langle \neg \text{Rep-bit } b \longleftrightarrow \text{even } b \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma Not-Not-Rep-bit [code-unfold]:
   $\langle \neg \neg \text{Rep-bit } b \longleftrightarrow \text{Rep-bit } b \rangle$ 
   $\langle \text{proof} \rangle$ 

code-datatype  $\langle 0::\text{bit} \rangle \langle 1::\text{bit} \rangle$ 

lemma Abs-bit-code [code]:
   $\langle \text{Abs-bit } \text{False} = 0 \rangle$ 
   $\langle \text{Abs-bit } \text{True} = 1 \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma Rep-bit-code [code]:
   $\langle \text{Rep-bit } 0 \longleftrightarrow \text{False} \rangle$ 
   $\langle \text{Rep-bit } 1 \longleftrightarrow \text{True} \rangle$ 
   $\langle \text{proof} \rangle$ 

```


context *zero-neq-one*
begin

abbreviation *of-bit* :: $\langle \text{bit} \Rightarrow 'a \rangle$
where $\langle \text{of-bit } b \equiv \text{of-bool } (\text{odd } b) \rangle$

end

context
begin

qualified lemma *bit-eq-iff*:
 $\langle a = b \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b) \rangle$ **for** $a \ b :: \text{bit}$
 $\langle \text{proof} \rangle$

end

lemma *modulo-bit-unfold* [*simp*, *code*]:
 $\langle a \bmod b = \text{of-bool } (\text{odd } a \wedge \text{even } b) \rangle$ **for** $a \ b :: \text{bit}$
 $\langle \text{proof} \rangle$

lemma *power-bit-unfold* [*simp*]:
 $\langle a \hat{\ } n = \text{of-bool } (\text{odd } a \vee n = 0) \rangle$ **for** $a :: \text{bit}$
 $\langle \text{proof} \rangle$

instantiation *bit* :: *field*
begin

definition *uminus-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{uminus-bit} = \text{id} \rangle$

definition *inverse-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{inverse-bit} = \text{id} \rangle$

instance
 $\langle \text{proof} \rangle$

end

instantiation *bit* :: *semiring-bits*
begin

definition *bit-bit* :: $\langle \text{bit} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$
where [*simp*]: $\langle \text{bit-bit } b \ n \longleftrightarrow \text{odd } b \wedge n = 0 \rangle$

instance
 $\langle \text{proof} \rangle$

end

instantiation *bit* :: *ring-bit-operations*
begin

context
includes *bit-operations-syntax*
begin

definition *not-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{NOT } b = \text{of_bool } (\text{even } b) \rangle$ **for** $b :: \text{bit}$

definition *and-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \text{ AND } c = \text{of_bool } (\text{odd } b \wedge \text{odd } c) \rangle$ **for** $b \ c :: \text{bit}$

definition *or-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \text{ OR } c = \text{of_bool } (\text{odd } b \vee \text{odd } c) \rangle$ **for** $b \ c :: \text{bit}$

definition *xor-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \text{ XOR } c = \text{of_bool } (\text{odd } b \neq \text{odd } c) \rangle$ **for** $b \ c :: \text{bit}$

definition *mask-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{mask } n = (\text{of_bool } (n > 0)) :: \text{bit} \rangle$

definition *set-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{set-bit } n \ b = \text{of_bool } (n = 0 \vee \text{odd } b) \rangle$ **for** $b :: \text{bit}$

definition *unset-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{unset-bit } n \ b = \text{of_bool } (n > 0 \wedge \text{odd } b) \rangle$ **for** $b :: \text{bit}$

definition *flip-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{flip-bit } n \ b = \text{of_bool } ((n = 0) \neq \text{odd } b) \rangle$ **for** $b :: \text{bit}$

definition *push-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{push-bit } n \ b = \text{of_bool } (\text{odd } b \wedge n = 0) \rangle$ **for** $b :: \text{bit}$

definition *drop-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{drop-bit } n \ b = \text{of_bool } (\text{odd } b \wedge n = 0) \rangle$ **for** $b :: \text{bit}$

definition *take-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{take-bit } n \ b = \text{of_bool } (\text{odd } b \wedge n > 0) \rangle$ **for** $b :: \text{bit}$

end

instance
 $\langle \text{proof} \rangle$

end

lemma *add-bit-eq-xor* [*simp*, *code*]:

$\langle (+) = (\text{Bit-Operations.xor} :: \text{bit} \Rightarrow -) \rangle$
 $\langle \text{proof} \rangle$

lemma *mult-bit-eq-and* [*simp*, *code*]:
 $\langle (*) = (\text{Bit-Operations.and} :: \text{bit} \Rightarrow -) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-even* [*simp*]:
 $\langle \text{numeral } (\text{Num.Bit0 } n) = (0 :: \text{bit}) \rangle$
 $\langle \text{proof} \rangle$

lemma *bit-numeral-odd* [*simp*]:
 $\langle \text{numeral } (\text{Num.Bit1 } n) = (1 :: \text{bit}) \rangle$
 $\langle \text{proof} \rangle$

end

109 Pointwise order on product types

theory *Product-Order*
imports *Product-Plus*
begin

109.1 Pointwise ordering

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
 $x \leq y \longleftrightarrow \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$

definition
 $(x :: 'a \times 'b) < y \longleftrightarrow x \leq y \wedge \neg y \leq x$

instance $\langle \text{proof} \rangle$

end

lemma *fst-mono*: $x \leq y \Longrightarrow \text{fst } x \leq \text{fst } y$
 $\langle \text{proof} \rangle$

lemma *snd-mono*: $x \leq y \Longrightarrow \text{snd } x \leq \text{snd } y$
 $\langle \text{proof} \rangle$

lemma *Pair-mono*: $x \leq x' \Longrightarrow y \leq y' \Longrightarrow (x, y) \leq (x', y')$
 $\langle \text{proof} \rangle$

lemma *Pair-le* [*simp*]: $(a, b) \leq (c, d) \longleftrightarrow a \leq c \wedge b \leq d$

$\langle proof \rangle$

lemma *atLeastAtMost-prod-eq*: $\{a..b\} = \{fst\ a..fst\ b\} \times \{snd\ a..snd\ b\}$
 $\langle proof \rangle$

instance *prod* :: (*preorder*, *preorder*) *preorder*
 $\langle proof \rangle$

instance *prod* :: (*order*, *order*) *order*
 $\langle proof \rangle$

109.2 Binary infimum and supremum

instantiation *prod* :: (*inf*, *inf*) *inf*
begin

definition *inf* *x y* = (*inf* (*fst* *x*) (*fst* *y*), *inf* (*snd* *x*) (*snd* *y*))

lemma *inf-Pair-Pair* [*simp*]: *inf* (*a*, *b*) (*c*, *d*) = (*inf* *a c*, *inf* *b d*)
 $\langle proof \rangle$

lemma *fst-inf* [*simp*]: *fst* (*inf* *x y*) = *inf* (*fst* *x*) (*fst* *y*)
 $\langle proof \rangle$

lemma *snd-inf* [*simp*]: *snd* (*inf* *x y*) = *inf* (*snd* *x*) (*snd* *y*)
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

instance *prod* :: (*semilattice-inf*, *semilattice-inf*) *semilattice-inf*
 $\langle proof \rangle$

instantiation *prod* :: (*sup*, *sup*) *sup*
begin

definition
sup *x y* = (*sup* (*fst* *x*) (*fst* *y*), *sup* (*snd* *x*) (*snd* *y*))

lemma *sup-Pair-Pair* [*simp*]: *sup* (*a*, *b*) (*c*, *d*) = (*sup* *a c*, *sup* *b d*)
 $\langle proof \rangle$

lemma *fst-sup* [*simp*]: *fst* (*sup* *x y*) = *sup* (*fst* *x*) (*fst* *y*)
 $\langle proof \rangle$

lemma *snd-sup* [*simp*]: *snd* (*sup* *x y*) = *sup* (*snd* *x*) (*snd* *y*)
 $\langle proof \rangle$

```

instance  $\langle proof \rangle$ 

end

instance  $prod :: (semilattice-sup, semilattice-sup) semilattice-sup$ 
   $\langle proof \rangle$ 

instance  $prod :: (lattice, lattice) lattice \langle proof \rangle$ 

instance  $prod :: (distrib-lattice, distrib-lattice) distrib-lattice$ 
   $\langle proof \rangle$ 

```

109.3 Top and bottom elements

```

instantiation  $prod :: (top, top) top$ 
begin

definition
   $top = (top, top)$ 

instance  $\langle proof \rangle$ 

end

lemma  $fst-top [simp]: fst\ top = top$ 
   $\langle proof \rangle$ 

lemma  $snd-top [simp]: snd\ top = top$ 
   $\langle proof \rangle$ 

lemma  $Pair-top-top: (top, top) = top$ 
   $\langle proof \rangle$ 

instance  $prod :: (order-top, order-top) order-top$ 
   $\langle proof \rangle$ 

instantiation  $prod :: (bot, bot) bot$ 
begin

definition
   $bot = (bot, bot)$ 

instance  $\langle proof \rangle$ 

end

lemma  $fst-bot [simp]: fst\ bot = bot$ 
   $\langle proof \rangle$ 

```

lemma *snd-bot* [*simp*]: *snd bot = bot*
 $\langle proof \rangle$

lemma *Pair-bot-bot*: $(bot, bot) = bot$
 $\langle proof \rangle$

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
 $\langle proof \rangle$

instance *prod* :: (*bounded-lattice*, *bounded-lattice*) *bounded-lattice* $\langle proof \rangle$

instance *prod* :: (*boolean-algebra*, *boolean-algebra*) *boolean-algebra*
 $\langle proof \rangle$

109.4 Complete lattice operations

instantiation *prod* :: (*Inf*, *Inf*) *Inf*
begin

definition *Inf A* = $(INF\ x \in A. fst\ x, INF\ x \in A. snd\ x)$

instance $\langle proof \rangle$

end

instantiation *prod* :: (*Sup*, *Sup*) *Sup*
begin

definition *Sup A* = $(SUP\ x \in A. fst\ x, SUP\ x \in A. snd\ x)$

instance $\langle proof \rangle$

end

instance *prod* :: (*conditionally-complete-lattice*, *conditionally-complete-lattice*)
conditionally-complete-lattice
 $\langle proof \rangle$

instance *prod* :: (*complete-lattice*, *complete-lattice*) *complete-lattice*
 $\langle proof \rangle$

lemma *fst-Inf*: $fst\ (Inf\ A) = (INF\ x \in A. fst\ x)$
 $\langle proof \rangle$

lemma *fst-INF*: $fst\ (INF\ x \in A. f\ x) = (INF\ x \in A. fst\ (f\ x))$
 $\langle proof \rangle$

lemma *fst-Sup*: $fst\ (Sup\ A) = (SUP\ x \in A. fst\ x)$

<proof>

lemma *fst-SUP*: $\text{fst } (\text{SUP } x \in A. f x) = (\text{SUP } x \in A. \text{fst } (f x))$
<proof>

lemma *snd-Inf*: $\text{snd } (\text{Inf } A) = (\text{INF } x \in A. \text{snd } x)$
<proof>

lemma *snd-INF*: $\text{snd } (\text{INF } x \in A. f x) = (\text{INF } x \in A. \text{snd } (f x))$
<proof>

lemma *snd-Sup*: $\text{snd } (\text{Sup } A) = (\text{SUP } x \in A. \text{snd } x)$
<proof>

lemma *snd-SUP*: $\text{snd } (\text{SUP } x \in A. f x) = (\text{SUP } x \in A. \text{snd } (f x))$
<proof>

lemma *INF-Pair*: $(\text{INF } x \in A. (f x, g x)) = (\text{INF } x \in A. f x, \text{INF } x \in A. g x)$
<proof>

lemma *SUP-Pair*: $(\text{SUP } x \in A. (f x, g x)) = (\text{SUP } x \in A. f x, \text{SUP } x \in A. g x)$
<proof>

Alternative formulations for set infima and suprema over the product of two complete lattices:

lemma *INF-prod-alt-def*:
 $\text{Inf } (f ' A) = (\text{Inf } ((\text{fst} \circ f) ' A), \text{Inf } ((\text{snd} \circ f) ' A))$
<proof>

lemma *SUP-prod-alt-def*:
 $\text{Sup } (f ' A) = (\text{Sup } ((\text{fst} \circ f) ' A), \text{Sup } ((\text{snd} \circ f) ' A))$
<proof>

109.5 Complete distributive lattices

instance *prod* :: (complete-distrib-lattice, complete-distrib-lattice) complete-distrib-lattice

<proof>

109.6 Bekic’s Theorem

Simultaneous fixed points over pairs can be written in terms of separate fixed points. Transliterated from HOLCF.Fix by Peter Gammie

lemma *lfp-prod*:
fixes $F :: 'a :: \text{complete-lattice} \times 'b :: \text{complete-lattice} \Rightarrow 'a \times 'b$
assumes *mono F*
shows $\text{lfp } F = (\text{lfp } (\lambda x. \text{fst } (F (x, \text{lfp } (\lambda y. \text{snd } (F (x, y)))))),$
 $\quad (\text{lfp } (\lambda y. \text{snd } (F (\text{lfp } (\lambda x. \text{fst } (F (x, \text{lfp } (\lambda y. \text{snd } (F (x, y)))))), y))))$
(is $\text{lfp } F = (?x, ?y)$ **)**

⟨proof⟩

lemma *gfp-prod*:

fixes $F :: 'a::complete-lattice \times 'b::complete-lattice \Rightarrow 'a \times 'b$

assumes *mono F*

shows $\text{gfp } F = (\text{gfp } (\lambda x. \text{fst } (F (x, \text{gfp } (\lambda y. \text{snd } (F (x, y)))))),$
 $(\text{gfp } (\lambda y. \text{snd } (F (\text{gfp } (\lambda x. \text{fst } (F (x, \text{gfp } (\lambda y. \text{snd } (F (x, y))))), y))))))$

(**is** $\text{gfp } F = (?x, ?y)$)

⟨proof⟩

end

110 Finite Lattices

theory *Finite-Lattice*

imports *Product-Order*

begin

110.1 Finite Complete Lattices

A non-empty finite lattice is a complete lattice. Since types are never empty in Isabelle/HOL, a type of classes *finite* and *lattice* should also have class *complete-lattice*. A type class is defined that extends classes *finite* and *lattice* with the operators *bot*, *top*, *Inf*, and *Sup*, along with assumptions that define these operators in terms of the ones of classes *finite* and *lattice*. The resulting class is a subclass of *complete-lattice*.

class *finite-lattice-complete* = *finite* + *lattice* + *bot* + *top* + *Inf* + *Sup* +

assumes *bot-def*: $\text{bot} = \text{Inf-fin UNIV}$

assumes *top-def*: $\text{top} = \text{Sup-fin UNIV}$

assumes *Inf-def*: $\text{Inf } A = \text{Finite-Set.fold inf top } A$

assumes *Sup-def*: $\text{Sup } A = \text{Finite-Set.fold sup bot } A$

The definitional assumptions on the operators *bot* and *top* of class *finite-lattice-complete* ensure that they yield bottom and top.

lemma *finite-lattice-complete-bot-least*: $(\text{bot}::'a::\text{finite-lattice-complete}) \leq x$

⟨proof⟩

instance *finite-lattice-complete* \subseteq *order-bot*

⟨proof⟩

lemma *finite-lattice-complete-top-greatest*: $(\text{top}::'a::\text{finite-lattice-complete}) \geq x$

⟨proof⟩

instance *finite-lattice-complete* \subseteq *order-top*

⟨proof⟩

instance *finite-lattice-complete* \subseteq *bounded-lattice* ⟨proof⟩

The definitional assumptions on the operators *Inf* and *Sup* of class *finite-lattice-complete* ensure that they yield infimum and supremum.

lemma *finite-lattice-complete-Inf-empty*: $\text{Inf } \{\} = (\text{top} :: 'a::\text{finite-lattice-complete})$
 $\langle \text{proof} \rangle$

lemma *finite-lattice-complete-Sup-empty*: $\text{Sup } \{\} = (\text{bot} :: 'a::\text{finite-lattice-complete})$
 $\langle \text{proof} \rangle$

lemma *finite-lattice-complete-Inf-insert*:
fixes $A :: 'a::\text{finite-lattice-complete}$ *set*
shows $\text{Inf } (\text{insert } x \ A) = \text{inf } x \ (\text{Inf } A)$
 $\langle \text{proof} \rangle$

lemma *finite-lattice-complete-Sup-insert*:
fixes $A :: 'a::\text{finite-lattice-complete}$ *set*
shows $\text{Sup } (\text{insert } x \ A) = \text{sup } x \ (\text{Sup } A)$
 $\langle \text{proof} \rangle$

lemma *finite-lattice-complete-Inf-lower*:
 $(x :: 'a::\text{finite-lattice-complete}) \in A \implies \text{Inf } A \leq x$
 $\langle \text{proof} \rangle$

lemma *finite-lattice-complete-Inf-greatest*:
 $\forall x :: 'a::\text{finite-lattice-complete} \in A. z \leq x \implies z \leq \text{Inf } A$
 $\langle \text{proof} \rangle$

lemma *finite-lattice-complete-Sup-upper*:
 $(x :: 'a::\text{finite-lattice-complete}) \in A \implies \text{Sup } A \geq x$
 $\langle \text{proof} \rangle$

lemma *finite-lattice-complete-Sup-least*:
 $\forall x :: 'a::\text{finite-lattice-complete} \in A. z \geq x \implies z \geq \text{Sup } A$
 $\langle \text{proof} \rangle$

instance *finite-lattice-complete* \subseteq *complete-lattice*
 $\langle \text{proof} \rangle$

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
 $(\text{bot} :: ('a::\text{finite-lattice-complete} \times 'b::\text{finite-lattice-complete})) =$
 $\text{Inf-fn } \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *finite-top-prod*:
 $(\text{top} :: ('a::\text{finite-lattice-complete} \times 'b::\text{finite-lattice-complete})) =$
 $\text{Sup-fn } \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *finite-Inf-prod*:

Inf ($A :: ('a::\text{finite-lattice-complete} \times 'b::\text{finite-lattice-complete}) \text{ set}$) =
Finite-Set.fold inf top A
 ⟨proof⟩

lemma *finite-Sup-prod*:
Sup ($A :: ('a::\text{finite-lattice-complete} \times 'b::\text{finite-lattice-complete}) \text{ set}$) =
Finite-Set.fold sup bot A
 ⟨proof⟩

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
 ⟨proof⟩

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

lemma *finite-bot-fun*: (*bot* :: ($'a::\text{finite} \Rightarrow 'b::\text{finite-lattice-complete}$)) = *Inf-fin UNIV*
 ⟨proof⟩

lemma *finite-top-fun*: (*top* :: ($'a::\text{finite} \Rightarrow 'b::\text{finite-lattice-complete}$)) = *Sup-fin UNIV*
 ⟨proof⟩

lemma *finite-Inf-fun*:
Inf ($A :: ('a::\text{finite} \Rightarrow 'b::\text{finite-lattice-complete}) \text{ set}$) =
Finite-Set.fold inf top A
 ⟨proof⟩

lemma *finite-Sup-fun*:
Sup ($A :: ('a::\text{finite} \Rightarrow 'b::\text{finite-lattice-complete}) \text{ set}$) =
Finite-Set.fold sup bot A
 ⟨proof⟩

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*
 ⟨proof⟩

110.2 Finite Distributive Lattices

A finite distributive lattice is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

class *finite-distrib-lattice-complete* =
distrib-lattice + *finite-lattice-complete*

lemma *finite-distrib-lattice-complete-sup-Inf*:
sup ($x :: 'a::\text{finite-distrib-lattice-complete}$) (*Inf* A) = (*INF* $y \in A.$ *sup* $x y$)
 ⟨proof⟩

lemma *finite-distrib-lattice-complete-inf-Sup*:
inf ($x :: 'a::\text{finite-distrib-lattice-complete}$) (*Sup* A) = (*SUP* $y \in A.$ *inf* $x y$)
 ⟨proof⟩

```

context finite-distrib-lattice-complete
begin
subclass finite-distrib-lattice
   $\langle proof \rangle$ 
end

```

```

instance finite-distrib-lattice-complete  $\subseteq$  complete-distrib-lattice  $\langle proof \rangle$ 

```

The product of two finite distributive lattices is already a finite distributive lattice.

```

instance prod ::
  (finite-distrib-lattice-complete, finite-distrib-lattice-complete)
  finite-distrib-lattice-complete
   $\langle proof \rangle$ 

```

Functions with a finite domain and with a finite distributive lattice as codomain already form a finite distributive lattice.

```

instance fun ::
  (finite, finite-distrib-lattice-complete) finite-distrib-lattice-complete
   $\langle proof \rangle$ 

```

110.3 Linear Orders

A linear order is a distributive lattice. A type class is defined that extends class *linorder* with the operators *inf* and *sup*, along with assumptions that define these operators in terms of the ones of class *linorder*. The resulting class is a subclass of *distrib-lattice*.

```

class linorder-lattice = linorder + inf + sup +
  assumes inf-def: inf x y = (if x  $\leq$  y then x else y)
  assumes sup-def: sup x y = (if x  $\geq$  y then x else y)

```

The definitional assumptions on the operators *inf* and *sup* of class *linorder-lattice* ensure that they yield infimum and supremum and that they distribute over each other.

```

lemma linorder-lattice-inf-le1: inf (x::'a::linorder-lattice) y  $\leq$  x
   $\langle proof \rangle$ 

```

```

lemma linorder-lattice-inf-le2: inf (x::'a::linorder-lattice) y  $\leq$  y
   $\langle proof \rangle$ 

```

```

lemma linorder-lattice-inf-greatest:
  (x::'a::linorder-lattice)  $\leq$  y  $\implies$  x  $\leq$  z  $\implies$  x  $\leq$  inf y z
   $\langle proof \rangle$ 

```

```

lemma linorder-lattice-sup-ge1: sup (x::'a::linorder-lattice) y  $\geq$  x
   $\langle proof \rangle$ 

```

```

lemma linorder-lattice-sup-ge2: sup (x::'a::linorder-lattice) y  $\geq$  y

```

$\langle proof \rangle$

lemma *linorder-lattice-sup-least*:

$(x::'a::linorder-lattice) \geq y \implies x \geq z \implies x \geq \sup y z$

$\langle proof \rangle$

lemma *linorder-lattice-sup-inf-distrib1*:

$\sup (x::'a::linorder-lattice) (\inf y z) = \inf (\sup x y) (\sup x z)$

$\langle proof \rangle$

instance *linorder-lattice* \subseteq *distrib-lattice*

$\langle proof \rangle$

110.4 Finite Linear Orders

A (non-empty) finite linear order is a complete linear order.

class *finite-linorder-complete* = *linorder-lattice* + *finite-lattice-complete*

instance *finite-linorder-complete* \subseteq *complete-linorder* $\langle proof \rangle$

A (non-empty) finite linear order is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

instance *finite-linorder-complete* \subseteq *finite-distrib-lattice-complete* $\langle proof \rangle$

end

111 Lexicographic order on lists

theory *List-Lexorder*

imports *Main*

begin

instantiation *list* :: (*ord*) *ord*

begin

definition

list-less-def: $xs < ys \longleftrightarrow (xs, ys) \in \text{lexord } \{(u, v). u < v\}$

definition

list-le-def: $(xs :: \text{list}) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

instance $\langle proof \rangle$

end

instance *list* :: (*order*) *order*

$\langle proof \rangle$

instance *list* :: (*linorder*) *linorder*
 ⟨*proof*⟩

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition (*inf* :: 'a *list* \Rightarrow -) = *min*

definition (*sup* :: 'a *list* \Rightarrow -) = *max*

instance
 ⟨*proof*⟩

end

lemma *not-less-Nil* [*simp*]: $\neg x < []$
 ⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: $[] < a \# x$
 ⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$
 ⟨*proof*⟩

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
 ⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
 ⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
 ⟨*proof*⟩

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
 ⟨*proof*⟩

end

lemma *less-list-code* [*code*]:
 $xs < ([] :: 'a :: \{equal, order\} list) \longleftrightarrow False$
 $[] < (x :: 'a :: \{equal, order\}) \# xs \longleftrightarrow True$
 $(x :: 'a :: \{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
 ⟨*proof*⟩

```

lemma less-eq-list-code [code]:
   $x \# xs \leq ([::'a::\{\text{equal}, \text{order}\} \text{ list}) \longleftrightarrow \text{False}$ 
   $[] \leq (xs::'a::\{\text{equal}, \text{order}\} \text{ list}) \longleftrightarrow \text{True}$ 
   $(x::'a::\{\text{equal}, \text{order}\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$ 
   $\langle \text{proof} \rangle$ 

end

```

112 Lexicographic order on lists

This version prioritises length and can yield wellorderings

```

theory List-Lenlexorder
imports Main
begin

```

```

instantiation list :: (ord) ord
begin

```

```

definition
  list-less-def:  $xs < ys \longleftrightarrow (xs, ys) \in \text{lenlex } \{(u, v). u < v\}$ 

```

```

definition
  list-le-def:  $(xs :: - \text{ list}) \leq ys \longleftrightarrow xs < ys \vee xs = ys$ 

```

```

instance  $\langle \text{proof} \rangle$ 

```

```

end

```

```

instance list :: (order) order
   $\langle \text{proof} \rangle$ 

```

```

instance list :: (linorder) linorder
   $\langle \text{proof} \rangle$ 

```

```

instance list :: (wellorder) wellorder
   $\langle \text{proof} \rangle$ 

```

```

instantiation list :: (linorder) distrib-lattice
begin

```

```

definition (inf :: 'a list  $\Rightarrow$  -) = min

```

```

definition (sup :: 'a list  $\Rightarrow$  -) = max

```

```

instance
   $\langle \text{proof} \rangle$ 

```

end

lemma *not-less-Nil* [*simp*]: $\neg x < []$
 ⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: $[] < a \# x$
 ⟨*proof*⟩

lemma *Cons-less-Cons*: $a \# x < b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x < y)$
 ⟨*proof*⟩

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
 ⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
 ⟨*proof*⟩

lemma *Cons-le-Cons*: $a \# x \leq b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x \leq y)$
 ⟨*proof*⟩

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
 ⟨*proof*⟩

end

end

113 Prefix order on lists as order class instance

theory *Prefix-Order*
imports *Sublist*
begin

instantiation *list* :: (*type*) *order*
begin

definition $xs \leq ys \equiv \text{prefix } xs \text{ } ys$ **for** $xs \text{ } ys :: 'a \text{ list}$

definition $xs < ys \equiv xs \leq ys \wedge \neg (ys \leq xs)$ **for** $xs \text{ } ys :: 'a \text{ list}$

instance
 ⟨*proof*⟩

end

lemma *less-list-def'*: $xs < ys \longleftrightarrow \text{strict-prefix } xs \ ys$ **for** $xs \ ys :: 'a \ \text{list}$
 ⟨*proof*⟩

lemmas *prefixI* [intro?] = *prefixI* [folded less-eq-list-def]
lemmas *prefixE* [elim?] = *prefixE* [folded less-eq-list-def]
lemmas *strict-prefixI'* [intro?] = *strict-prefixI'* [folded less-list-def]
lemmas *strict-prefixE'* [elim?] = *strict-prefixE'* [folded less-list-def]
lemmas *strict-prefixI* [intro?] = *strict-prefixI* [folded less-list-def]
lemmas *strict-prefixE* [elim?] = *strict-prefixE* [folded less-list-def]
lemmas *Nil-prefix* [iff] = *Nil-prefix* [folded less-eq-list-def]
lemmas *prefix-Nil* [simp] = *prefix-Nil* [folded less-eq-list-def]
lemmas *prefix-snoc* [simp] = *prefix-snoc* [folded less-eq-list-def]
lemmas *Cons-prefix-Cons* [simp] = *Cons-prefix-Cons* [folded less-eq-list-def]
lemmas *same-prefix-prefix* [simp] = *same-prefix-prefix* [folded less-eq-list-def]
lemmas *same-prefix-nil* [iff] = *same-prefix-nil* [folded less-eq-list-def]
lemmas *prefix-prefix* [simp] = *prefix-prefix* [folded less-eq-list-def]
lemmas *prefix-Cons* = *prefix-Cons* [folded less-eq-list-def]
lemmas *prefix-length-le* = *prefix-length-le* [folded less-eq-list-def]
lemmas *strict-prefix-simps* [simp, code] = *strict-prefix-simps* [folded less-list-def]
lemmas *not-prefix-induct* [consumes 1, case-names Nil Neq Eq] =
not-prefix-induct [folded less-eq-list-def]

end

114 Lexicographic order on product types

theory *Product-Lexorder*

imports *Main*

begin

instantiation *prod* :: (ord, ord) ord

begin

definition

$$x \leq y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$$

definition

$$x < y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x < \text{snd } y$$

instance ⟨*proof*⟩

end

lemma *less-eq-prod-simp* [simp, code]:

$$(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$$

⟨*proof*⟩

lemma *less-prod-simp* [*simp*, *code*]:
 $(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$
 $\langle proof \rangle$

A stronger version for partial orders.

lemma *less-prod-def'*:
fixes $x\ y :: 'a::order \times 'b::ord$
shows $x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x < snd\ y$
 $\langle proof \rangle$

instance *prod* :: (*preorder*, *preorder*) *preorder*
 $\langle proof \rangle$

instance *prod* :: (*order*, *order*) *order*
 $\langle proof \rangle$

instance *prod* :: (*linorder*, *linorder*) *linorder*
 $\langle proof \rangle$

instantiation *prod* :: (*linorder*, *linorder*) *distrib-lattice*
begin

definition
 $(inf :: 'a \times 'b \Rightarrow - \Rightarrow -) = min$

definition
 $(sup :: 'a \times 'b \Rightarrow - \Rightarrow -) = max$

instance
 $\langle proof \rangle$

end

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
 $bot = (bot, bot)$

instance $\langle proof \rangle$

end

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
 $\langle proof \rangle$

instantiation *prod* :: (*top*, *top*) *top*
begin

definition

$top = (top, top)$

instance $\langle proof \rangle$

end

instance $prod :: (order-top, order-top) \rightarrow order-top$
 $\langle proof \rangle$

instance $prod :: (wellorder, wellorder) \rightarrow wellorder$
 $\langle proof \rangle$

Legacy lemma bindings

lemmas $prod-le-def = less-eq-prod-def$

lemmas $prod-less-def = less-prod-def$

lemmas $prod-less-eq = less-prod-def'$

end

115 Subsequence Ordering

theory *Subseq-Order*

imports *Sublist*

begin

This theory defines subsequence ordering on lists. A list ys is a subsequence of a list xs , iff one obtains ys by erasing some elements from xs .

115.1 Definitions and basic lemmas

instantiation $list :: (type) \rightarrow ord$

begin

definition *less-eq-list*

where $\langle xs \leq ys \longleftrightarrow subseq\ xs\ ys \rangle$ **for** $xs\ ys :: \langle 'a\ list \rangle$

definition *less-list*

where $\langle xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs \rangle$ **for** $xs\ ys :: \langle 'a\ list \rangle$

instance $\langle proof \rangle$

end

instance $list :: (type) \rightarrow order$

$\langle proof \rangle$

lemmas $less-eq-list-induct$ [consumes 1, case-names empty drop take] =
 $list-emb.induct$ [of (=), folded less-eq-list-def]

lemma *less-eq-list-empty* [code]:

$\langle [] \leq xs \longleftrightarrow \text{True} \rangle$
 $\langle \text{proof} \rangle$

lemma *less-eq-list-below-empty* [code]:

$\langle x \# xs \leq [] \longleftrightarrow \text{False} \rangle$
 $\langle \text{proof} \rangle$

lemma *le-list-Cons2-iff* [simp, code]:

$\langle x \# xs \leq y \# ys \longleftrightarrow (\text{if } x = y \text{ then } xs \leq ys \text{ else } x \# xs \leq ys) \rangle$
 $\langle \text{proof} \rangle$

lemma *less-list-empty* [simp]:

$\langle [] < xs \longleftrightarrow xs \neq [] \rangle$
 $\langle \text{proof} \rangle$

lemma *less-list-empty-Cons* [code]:

$\langle [] < x \# xs \longleftrightarrow \text{True} \rangle$
 $\langle \text{proof} \rangle$

lemma *less-list-below-empty* [simp, code]:

$\langle xs < [] \longleftrightarrow \text{False} \rangle$
 $\langle \text{proof} \rangle$

lemma *less-list-Cons2-iff* [code]:

$\langle x \# xs < y \# ys \longleftrightarrow (\text{if } x = y \text{ then } xs < ys \text{ else } x \# xs \leq ys) \rangle$
 $\langle \text{proof} \rangle$

lemmas *less-eq-list-drop* = *list-emb.list-emb-Cons* [of (=), folded *less-eq-list-def*]

lemmas *le-list-map* = *subseq-map* [folded *less-eq-list-def*]

lemmas *le-list-filter* = *subseq-filter* [folded *less-eq-list-def*]

lemmas *le-list-length* = *list-emb-length* [of (=), folded *less-eq-list-def*]

lemma *less-list-length*: $xs < ys \implies \text{length } xs < \text{length } ys$

$\langle \text{proof} \rangle$

lemma *less-list-drop*: $xs < ys \implies xs < x \# ys$

$\langle \text{proof} \rangle$

lemma *less-list-take-iff*: $x \# xs < x \# ys \longleftrightarrow xs < ys$

$\langle \text{proof} \rangle$

lemma *less-list-drop-many*: $xs < ys \implies xs < zs @ ys$

$\langle \text{proof} \rangle$

lemma *less-list-take-many-iff*: $zs @ xs < zs @ ys \longleftrightarrow xs < ys$

$\langle \text{proof} \rangle$

lemma *less-list-rev-take*: $xs @ zs < ys @ zs \longleftrightarrow xs < ys$
<proof>

end

116 Records based on BNF/datatype machinery

theory *Datatype-Records*
imports *Main*
keywords *datatype-record* :: *thy-defn*
begin

This theory provides an alternative, stripped-down implementation of records based on the machinery of the **datatype** package.

It supports:

- similar declaration syntax as records
- record creation and update syntax (using `[...]` brackets)
- regular datatype features (e.g. dead type variables etc.)
- “after-the-fact” registration of single-free-constructor types as records

Caveats:

- there is no compatibility layer; importing this theory will disrupt existing syntax
- extensible records are not supported

nonterminal
ident **and**
field-type **and**
field-types **and**
field **and**
fields **and**
field-update **and**
field-updates

open-bundle *datatype-record-syntax*
begin

unbundle *no record-syntax*

syntax
-constify :: *id* => *ident* (*<->*)
-constify :: *longid* => *ident* (*<->*)

```

  -datatype-field      :: ident => 'a => field          (⟨(⟨indent=2 notation=⟨infix
field value⟩⟩- =/ -)⟩)
                        :: field => fields              (⟨-⟩)
  -datatype-fields     :: field => fields => fields      (⟨-,/ -⟩)
  -datatype-record     :: fields => 'a                  (⟨(⟨indent=3 notation=⟨mixfix
datatype record value⟩⟩(| - |^))⟩)
  -datatype-field-update :: ident => 'a => field-update   (⟨(⟨indent=2 nota-
tion=⟨infix field update⟩⟩- :=/ -)⟩)
                        :: field-update => field-updates (⟨-⟩)
  -datatype-field-updates :: field-update => field-updates => field-updates (⟨-,/ -⟩)
  -datatype-record-update :: 'a => field-updates => 'b    (⟨(⟨open-block nota-
tion=⟨mixfix datatype record update⟩⟩-/(3(| - |^)))⟩ [900, 0] 900)

```

syntax (ASCII)

```

  -datatype-record     :: fields => 'a                  (⟨(⟨indent=3 notation=⟨mixfix
datatype record value⟩⟩(| - |^))⟩)
  -datatype-record-update :: 'a => field-updates => 'b    (⟨(⟨open-block nota-
tion=⟨mixfix datatype record update⟩⟩-/(3(| - |^)))⟩ [900, 0] 900)

```

end

named-theorems datatype-record-update

⟨ML⟩

end

117 Implementation of mappings with Association Lists

theory AList-Mapping
imports AList Mapping
begin

lift-definition Mapping :: ('a × 'b) list ⇒ ('a, 'b) mapping **is** map-of ⟨proof⟩

code-datatype Mapping

lemma lookup-Mapping [simp, code]: Mapping.lookup (Mapping xs) = map-of xs
 ⟨proof⟩

lemma keys-Mapping [simp, code]: Mapping.keys (Mapping xs) = set (map fst xs)
 ⟨proof⟩

lemma empty-Mapping [code]: Mapping.empty = Mapping []
 ⟨proof⟩

lemma *is-empty-Mapping* [code]: $\text{Mapping.is-empty } (\text{Mapping } xs) \longleftrightarrow \text{List.null } xs$
 ⟨proof⟩

lemma *update-Mapping* [code]: $\text{Mapping.update } k \ v \ (\text{Mapping } xs) = \text{Mapping } (\text{AList.update } k \ v \ xs)$
 ⟨proof⟩

lemma *delete-Mapping* [code]: $\text{Mapping.delete } k \ (\text{Mapping } xs) = \text{Mapping } (\text{AList.delete } k \ xs)$
 ⟨proof⟩

lemma *ordered-keys-Mapping* [code]:
 $\text{Mapping.ordered-keys } (\text{Mapping } xs) = \text{sort } (\text{remdups } (\text{map fst } xs))$
 ⟨proof⟩

lemma *entries-Mapping* [code]:
 $\text{Mapping.entries } (\text{Mapping } xs) = \text{set } (\text{AList.clearjunk } xs)$
 ⟨proof⟩

lemma *ordered-entries-Mapping* [code]:
 $\text{Mapping.ordered-entries } (\text{Mapping } xs) = \text{sort-key fst } (\text{AList.clearjunk } xs)$
 ⟨proof⟩

lemma *fold-Mapping* [code]:
 $\text{Mapping.fold } f \ (\text{Mapping } xs) \ a = \text{List.fold } (\text{case-prod } f) \ (\text{sort-key fst } (\text{AList.clearjunk } xs)) \ a$
 ⟨proof⟩

lemma *size-Mapping* [code]: $\text{Mapping.size } (\text{Mapping } xs) = \text{length } (\text{remdups } (\text{map fst } xs))$
 ⟨proof⟩

lemma *tabulate-Mapping* [code]: $\text{Mapping.tabulate } ks \ f = \text{Mapping } (\text{map } (\lambda k. (k, f \ k)) \ ks)$
 ⟨proof⟩

lemma *bulkload-Mapping* [code]:
 $\text{Mapping.bulkload } vs = \text{Mapping } (\text{map } (\lambda n. (n, \text{vs } ! \ n)) \ [0..<\text{length } vs])$
 ⟨proof⟩

lemma *equal-Mapping* [code]:
 $\text{HOL.equal } (\text{Mapping } xs) \ (\text{Mapping } ys) \longleftrightarrow$
 (let $ks = \text{map fst } xs$; $ls = \text{map fst } ys$
 in $(\forall l \in \text{set } ls. l \in \text{set } ks) \wedge (\forall k \in \text{set } ks. k \in \text{set } ls \wedge \text{map-of } xs \ k = \text{map-of } ys \ k)$)
 ⟨proof⟩

lemma *map-values-Mapping* [code]:
 $\text{Mapping.map-values } f \ (\text{Mapping } xs) = \text{Mapping } (\text{map } (\lambda (x,y). (x, f \ x \ y)) \ xs)$

for $f :: 'c \Rightarrow 'a \Rightarrow 'b$ **and** $xs :: ('c \times 'a)$ list
 ⟨proof⟩

lemma *combine-with-key-code* [code]:
 $Mapping.combine-with-key\ f\ (Mapping\ xs)\ (Mapping\ ys) =$
 $Mapping.tabulate\ (remdups\ (map\ fst\ xs\ @\ map\ fst\ ys))$
 $(\lambda x. the\ (combine-options\ (f\ x)\ (map-of\ xs\ x)\ (map-of\ ys\ x)))$
 ⟨proof⟩

lemma *combine-code* [code]:
 $Mapping.combine\ f\ (Mapping\ xs)\ (Mapping\ ys) =$
 $Mapping.tabulate\ (remdups\ (map\ fst\ xs\ @\ map\ fst\ ys))$
 $(\lambda x. the\ (combine-options\ f\ (map-of\ xs\ x)\ (map-of\ ys\ x)))$
 ⟨proof⟩

lemma *map-of-filter-distinct*:
assumes *distinct* ($map\ fst\ xs$)
shows $map-of\ (filter\ P\ xs)\ x =$
 $(case\ map-of\ xs\ x\ of$
 $None \Rightarrow None$
 $| Some\ y \Rightarrow if\ P\ (x,y)\ then\ Some\ y\ else\ None)$
 ⟨proof⟩

lemma *filter-Mapping* [code]:
 $Mapping.filter\ P\ (Mapping\ xs) = Mapping\ (filter\ (\lambda(k,v). P\ k\ v)\ (AList.clearjunk\ xs))$
 ⟨proof⟩

lemma [code nbe]: $HOL.equal\ (x :: ('a, 'b)\ mapping)\ x \longleftrightarrow True$
 ⟨proof⟩

end

theory *Code-Abstract-Char*

imports

Main

HOL-Library.Char-ord

begin

definition *Chr* :: $\langle integer \Rightarrow char \rangle$
where [simp]: $\langle Chr = char-of \rangle$

lemma *char-of-integer-of-char* [code abstype]:
 $\langle Chr\ (integer-of-char\ c) = c \rangle$
 ⟨proof⟩

lemma *char-of-integer-code* [code]:
 $\langle integer-of-char\ (char-of-integer\ k) = (if\ 0 \leq k \wedge k < 256\ then\ k\ else\ k\ mod$

256)›
 ⟨proof⟩

lemma *of-char-code* [code]:
 ⟨of-char c = of-nat (nat-of-integer (integer-of-char c))›
 ⟨proof⟩

definition *byte* :: ⟨bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ integer⟩
where [simp]: ⟨byte b0 b1 b2 b3 b4 b5 b6 b7 = horner-sum of-bool 2 [b0, b1, b2, b3, b4, b5, b6, b7]›

lemma *byte-code* [code]:
 ⟨byte b0 b1 b2 b3 b4 b5 b6 b7 = (
 let
 s0 = if b0 then 1 else 0;
 s1 = if b1 then s0 + 2 else s0;
 s2 = if b2 then s1 + 4 else s1;
 s3 = if b3 then s2 + 8 else s2;
 s4 = if b4 then s3 + 16 else s3;
 s5 = if b5 then s4 + 32 else s4;
 s6 = if b6 then s5 + 64 else s5;
 s7 = if b7 then s6 + 128 else s6
 in s7)›
 ⟨proof⟩

lemma *Char-code* [code]:
 ⟨integer-of-char (Char b0 b1 b2 b3 b4 b5 b6 b7) = byte b0 b1 b2 b3 b4 b5 b6 b7›
 ⟨proof⟩

lemma *digit-0-code* [code]:
 ⟨digit0 c ⟷ bit (integer-of-char c) 0›
 ⟨proof⟩

lemma *digit-1-code* [code]:
 ⟨digit1 c ⟷ bit (integer-of-char c) 1›
 ⟨proof⟩

lemma *digit-2-code* [code]:
 ⟨digit2 c ⟷ bit (integer-of-char c) 2›
 ⟨proof⟩

lemma *digit-3-code* [code]:
 ⟨digit3 c ⟷ bit (integer-of-char c) 3›
 ⟨proof⟩

lemma *digit-4-code* [code]:
 ⟨digit4 c ⟷ bit (integer-of-char c) 4›
 ⟨proof⟩

lemma *digit-5-code* [code]:
 $\langle \text{digit5 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c) \ 5 \rangle$
 $\langle \text{proof} \rangle$

lemma *digit-6-code* [code]:
 $\langle \text{digit6 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c) \ 6 \rangle$
 $\langle \text{proof} \rangle$

lemma *digit-7-code* [code]:
 $\langle \text{digit7 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c) \ 7 \rangle$
 $\langle \text{proof} \rangle$

lemma *case-char-code* [code]:
 $\langle \text{case-char } f \ c = f \ (\text{digit0 } c) \ (\text{digit1 } c) \ (\text{digit2 } c) \ (\text{digit3 } c) \ (\text{digit4 } c) \ (\text{digit5 } c) \ (\text{digit6 } c) \ (\text{digit7 } c) \rangle$
 $\langle \text{proof} \rangle$

lemma *rec-char-code* [code]:
 $\langle \text{rec-char } f \ c = f \ (\text{digit0 } c) \ (\text{digit1 } c) \ (\text{digit2 } c) \ (\text{digit3 } c) \ (\text{digit4 } c) \ (\text{digit5 } c) \ (\text{digit6 } c) \ (\text{digit7 } c) \rangle$
 $\langle \text{proof} \rangle$

lemma *char-of-code* [code]:
 $\langle \text{integer-of-char } (\text{char-of } a) =$
 $\text{byte } (\text{bit } a \ 0) \ (\text{bit } a \ 1) \ (\text{bit } a \ 2) \ (\text{bit } a \ 3) \ (\text{bit } a \ 4) \ (\text{bit } a \ 5) \ (\text{bit } a \ 6) \ (\text{bit } a \ 7) \rangle$
 $\langle \text{proof} \rangle$

lemma *ascii-of-code* [code]:
 $\langle \text{integer-of-char } (\text{String.ascii-of } c) = (\text{let } k = \text{integer-of-char } c \text{ in if } k < 128 \text{ then } k \text{ else } k - 128) \rangle$
 $\langle \text{proof} \rangle$

lemma *equal-char-code* [code]:
 $\langle \text{HOL.equal } c \ d \longleftrightarrow \text{integer-of-char } c = \text{integer-of-char } d \rangle$
 $\langle \text{proof} \rangle$

lemma *less-eq-char-code* [code]:
 $\langle c \leq d \longleftrightarrow \text{integer-of-char } c \leq \text{integer-of-char } d \rangle \text{ (is } \langle ?P \longleftrightarrow ?Q \rangle)$
 $\langle \text{proof} \rangle$

lemma *less-char-code* [code]:
 $\langle c < d \longleftrightarrow \text{integer-of-char } c < \text{integer-of-char } d \rangle \text{ (is } \langle ?P \longleftrightarrow ?Q \rangle)$
 $\langle \text{proof} \rangle$

lemma *absdef-simps*:
 $\langle \text{horner-sum of-bool } 2 \ [] = (0 :: \text{integer}) \rangle$
 $\langle \text{horner-sum of-bool } 2 \ (\text{False} \ \# \ bs) = (0 :: \text{integer}) \longleftrightarrow \text{horner-sum of-bool } 2 \ bs = (0 :: \text{integer}) \rangle$

```

  ⟨horner-sum of-bool 2 (True # bs) = (1 :: integer) ⟷ horner-sum of-bool 2 bs
  = (0 :: integer)⟩
  ⟨horner-sum of-bool 2 (False # bs) = (numeral (Num.Bit0 n) :: integer) ⟷
  horner-sum of-bool 2 bs = (numeral n :: integer)⟩
  ⟨horner-sum of-bool 2 (True # bs) = (numeral (Num.Bit1 n) :: integer) ⟷
  horner-sum of-bool 2 bs = (numeral n :: integer)⟩
  ⟨proof⟩

⟨ML⟩

```

code-identifier

code-module *Code-Abstract-Char* \rightarrow

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

end

118 Avoidance of pattern matching on natural numbers

theory *Code-Abstract-Nat*

imports *Main*

begin

When natural numbers are implemented in another than the conventional inductive *0/Suc* representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

118.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code, code-unfold*]:
 $\text{case-nat} = (\lambda f\ g\ n. \text{if } n = 0 \text{ then } f \text{ else } g\ (n - 1))$
 ⟨*proof*⟩

118.2 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

lemma *Suc-if-eq*:
 assumes $\bigwedge n. f\ (\text{Suc } n) \equiv h\ n$
 assumes $f\ 0 \equiv g$
 shows $f\ n \equiv \text{if } n = 0 \text{ then } g \text{ else } h\ (n - 1)$

⟨proof⟩

The rule above is built into a preprocessor that is plugged into the code generator.

⟨ML⟩

118.3 Candidates which need special treatment

lemma *drop-bit-int-code* [code]:
 ⟨drop-bit n $k = k \text{ div } 2 \wedge n$ ⟩ **for** $k :: \text{int}$
 ⟨proof⟩

lemma *take-bit-num-code* [code]:
 ⟨take-bit-num n $\text{Num.One} =$
 (case n of $0 \Rightarrow \text{None} \mid \text{Suc } n \Rightarrow \text{Some Num.One}$)⟩
 ⟨take-bit-num n ($\text{Num.Bit0 } m$) =
 (case n of $0 \Rightarrow \text{None} \mid \text{Suc } n \Rightarrow (\text{case take-bit-num } n \text{ } m \text{ of } \text{None} \Rightarrow \text{None} \mid$
 $\text{Some } q \Rightarrow \text{Some } (\text{Num.Bit0 } q))$)⟩
 ⟨take-bit-num n ($\text{Num.Bit1 } m$) =
 (case n of $0 \Rightarrow \text{None} \mid \text{Suc } n \Rightarrow \text{Some } (\text{case take-bit-num } n \text{ } m \text{ of } \text{None} \Rightarrow$
 $\text{Num.One} \mid \text{Some } q \Rightarrow \text{Num.Bit1 } q)$)⟩
 ⟨proof⟩

end

119 Implementation of natural numbers as binary numerals

theory *Code-Binary-Nat*
imports *Code-Abstract-Nat*
begin

When generating code for functions on natural numbers, the canonical representation using 0 and Suc is unsuitable for computations involving large numbers. This theory refines the representation of natural numbers for code generation to use binary numerals, which do not grow linear in size but logarithmic.

119.1 Representation

code-datatype $0 :: \text{nat nat-of-num}$

lemma [code]:
 $\text{num-of-nat } 0 = \text{Num.One}$
 $\text{num-of-nat } (\text{nat-of-num } k) = k$
 ⟨proof⟩

lemma [code]:

$(1::nat) = \text{Numeral1}$
 $\langle \text{proof} \rangle$

lemma $[\text{code-abbrev}]$: $\text{Numeral1} = (1::nat)$
 $\langle \text{proof} \rangle$

lemma $[\text{code}]$:
 $\text{Suc } n = n + 1$
 $\langle \text{proof} \rangle$

119.2 Basic arithmetic

context
begin

lemma $\text{plus-nat-code } [\text{code}]$:
 $0 + n = (n::nat)$
 $m + 0 = (m::nat)$
 $\text{nat-of-num } k + \text{nat-of-num } l = \text{nat-of-num } (k + l)$
 $\langle \text{proof} \rangle$

Bounded subtraction needs some auxiliary

qualified definition $\text{dup} :: nat \Rightarrow nat$ **where**
 $\text{dup } n = n + n$

lemma $\text{dup-code } [\text{code}]$:
 $\text{dup } 0 = 0$
 $\text{dup } (\text{nat-of-num } k) = \text{nat-of-num } (\text{Num.Bit0 } k)$
 $\langle \text{proof} \rangle$ **definition** $\text{sub} :: num \Rightarrow num \Rightarrow nat \text{ option}$ **where**
 $\text{sub } k \ l = (\text{if } k \geq l \text{ then } \text{Some } (\text{numeral } k - \text{numeral } l) \text{ else } \text{None})$

lemma $\text{sub-code } [\text{code}]$:
 $\text{sub } \text{Num.One } \text{Num.One} = \text{Some } 0$
 $\text{sub } (\text{Num.Bit0 } m) \ \text{Num.One} = \text{Some } (\text{nat-of-num } (\text{Num.BitM } m))$
 $\text{sub } (\text{Num.Bit1 } m) \ \text{Num.One} = \text{Some } (\text{nat-of-num } (\text{Num.Bit0 } m))$
 $\text{sub } \text{Num.One } (\text{Num.Bit0 } n) = \text{None}$
 $\text{sub } \text{Num.One } (\text{Num.Bit1 } n) = \text{None}$
 $\text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit0 } n) = \text{map-option } \text{dup } (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit1 } n) = \text{map-option } \text{dup } (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit0 } n) = \text{map-option } (\lambda q. \text{dup } q + 1) (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit1 } n) = (\text{case } \text{sub } m \ n \text{ of } \text{None} \Rightarrow \text{None}$
 $\quad | \text{Some } q \Rightarrow \text{if } q = 0 \text{ then } \text{None} \text{ else } \text{Some } (\text{dup } q - 1))$
 $\langle \text{proof} \rangle$

lemma $\text{minus-nat-code } [\text{code}]$:
 $0 - n = (0::nat)$
 $m - 0 = (m::nat)$
 $\text{nat-of-num } k - \text{nat-of-num } l = (\text{case } \text{sub } k \ l \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } j \Rightarrow j)$
 $\langle \text{proof} \rangle$

lemma *times-nat-code* [code]:

$0 * n = (0::nat)$
 $m * 0 = (0::nat)$
 $nat-of-num\ k * nat-of-num\ l = nat-of-num\ (k * l)$
 ⟨proof⟩

lemma *equal-nat-code* [code]:

$HOL.equal\ 0\ (0::nat) \longleftrightarrow True$
 $HOL.equal\ 0\ (nat-of-num\ l) \longleftrightarrow False$
 $HOL.equal\ (nat-of-num\ k)\ 0 \longleftrightarrow False$
 $HOL.equal\ (nat-of-num\ k)\ (nat-of-num\ l) \longleftrightarrow HOL.equal\ k\ l$
 ⟨proof⟩

lemma *equal-nat-refl* [code nbe]:

$HOL.equal\ (n::nat)\ n \longleftrightarrow True$
 ⟨proof⟩

lemma *less-eq-nat-code* [code]:

$0 \leq (n::nat) \longleftrightarrow True$
 $nat-of-num\ k \leq 0 \longleftrightarrow False$
 $nat-of-num\ k \leq nat-of-num\ l \longleftrightarrow k \leq l$
 ⟨proof⟩

lemma *less-nat-code* [code]:

$(m::nat) < 0 \longleftrightarrow False$
 $0 < nat-of-num\ l \longleftrightarrow True$
 $nat-of-num\ k < nat-of-num\ l \longleftrightarrow k < l$
 ⟨proof⟩

lemma *divmod-nat-code* [code]:

$Euclidean-Rings.divmod-nat\ 0\ n = (0, 0)$
 $Euclidean-Rings.divmod-nat\ m\ 0 = (0, m)$
 $Euclidean-Rings.divmod-nat\ (nat-of-num\ k)\ (nat-of-num\ l) = divmod\ k\ l$
 ⟨proof⟩

end

119.3 Conversions

lemma *of-nat-code* [code]:

$of-nat\ 0 = 0$
 $of-nat\ (nat-of-num\ k) = numeral\ k$
 ⟨proof⟩

code-identifier

code-module *Code-Binary-Nat* \rightarrow
 (SML) *Arith* **and** (OCaml) *Arith* **and** (Haskell) *Arith*

end

120 Code generation of prolog programs

```
theory Code-Prolog
imports Main
keywords values-prolog :: diag
begin
```

$\langle ML \rangle$

121 Setup for Numerals

$\langle ML \rangle$

end

122 Implementation of integer numbers by target-language integers

```
theory Code-Target-Int
imports Main
begin
```

```
code-datatype int-of-integer
```

```
context
includes integer.lifting
begin
```

```
lemma [code]:
  integer-of-int (int-of-integer k) = k
   $\langle proof \rangle$ 
```

```
lemma [code]:
  Int.Pos = int-of-integer  $\circ$  integer-of-num
   $\langle proof \rangle$ 
```

```
lemma [code]:
  Int.Neg = int-of-integer  $\circ$  uminus  $\circ$  integer-of-num
   $\langle proof \rangle$ 
```

```
lemma [code-abbrev]:
  int-of-integer (numeral k) = Int.Pos k
   $\langle proof \rangle$ 
```

```
lemma [code-abbrev]:
```

int-of-integer ($- \text{numeral } k$) = *Int.Neg* k
 $\langle \text{proof} \rangle$

context
begin

qualified definition *positive* :: *num* \Rightarrow *int*
where [*simp*]: *positive* = *numeral*

qualified definition *negative* :: *num* \Rightarrow *int*
where [*simp*]: *negative* = *uminus* \circ *numeral*

lemma [*code-computation-unfold*]:
numeral = *positive*
Int.Pos = *positive*
Int.Neg = *negative*
 $\langle \text{proof} \rangle$

end

lemma [*code, symmetric, code-post*]:
 $0 = \text{int-of-integer } 0$
 $\langle \text{proof} \rangle$

lemma [*code, symmetric, code-post*]:
 $1 = \text{int-of-integer } 1$
 $\langle \text{proof} \rangle$

lemma [*code-post*]:
 $\text{int-of-integer } (-\ 1) = -\ 1$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $k + l = \text{int-of-integer } (\text{of-int } k + \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $- k = \text{int-of-integer } (-\ \text{of-int } k)$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $k - l = \text{int-of-integer } (\text{of-int } k - \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\text{Int.dup } k = \text{int-of-integer } (\text{Code-Numeral.dup } (\text{of-int } k))$
 $\langle \text{proof} \rangle$

declare [[*code drop*: *Int.sub*]]

lemma [code]:

$k * l = \text{int-of-integer } (\text{of-int } k * \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$k \text{ div } l = \text{int-of-integer } (\text{of-int } k \text{ div } \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$k \text{ mod } l = \text{int-of-integer } (\text{of-int } k \text{ mod } \text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$\text{divmod } m \ n = \text{map-prod int-of-integer int-of-integer } (\text{divmod } m \ n)$
 $\langle \text{proof} \rangle$

lemma [code]:

$\text{HOL.equal } k \ l = \text{HOL.equal } (\text{of-int } k :: \text{integer}) \ (\text{of-int } l)$
 $\langle \text{proof} \rangle$

lemma [code]:

$k \leq l \longleftrightarrow (\text{of-int } k :: \text{integer}) \leq \text{of-int } l$
 $\langle \text{proof} \rangle$

lemma [code]:

$k < l \longleftrightarrow (\text{of-int } k :: \text{integer}) < \text{of-int } l$
 $\langle \text{proof} \rangle$

lemma gcd-int-of-integer [code]:

$\text{gcd } (\text{int-of-integer } x) \ (\text{int-of-integer } y) = \text{int-of-integer } (\text{gcd } x \ y)$
 $\langle \text{proof} \rangle$

lemma lcm-int-of-integer [code]:

$\text{lcm } (\text{int-of-integer } x) \ (\text{int-of-integer } y) = \text{int-of-integer } (\text{lcm } x \ y)$
 $\langle \text{proof} \rangle$

end

lemma (in ring-1) of-int-code-if:

$\text{of-int } k = (\text{if } k = 0 \text{ then } 0$
 $\quad \text{else if } k < 0 \text{ then } - \text{of-int } (- k)$
 $\quad \text{else let}$
 $\quad \quad l = 2 * \text{of-int } (k \text{ div } 2);$
 $\quad \quad j = k \text{ mod } 2$
 $\quad \text{in if } j = 0 \text{ then } l \text{ else } l + 1)$
 $\langle \text{proof} \rangle$

declare of-int-code-if [code]

lemma [code]:
 $\text{nat} = \text{nat-of-integer} \circ \text{of-int}$
including *integer.lifting* ⟨proof⟩

definition *char-of-int* :: $\text{int} \Rightarrow \text{char}$
where [code-abbrev]: *char-of-int* = *char-of*

definition *int-of-char* :: $\text{char} \Rightarrow \text{int}$
where [code-abbrev]: *int-of-char* = *of-char*

lemma [code]:
 $\text{char-of-int} = \text{char-of-integer} \circ \text{integer-of-int}$
including *integer.lifting* ⟨proof⟩

lemma [code]:
 $\text{int-of-char} = \text{int-of-integer} \circ \text{integer-of-char}$
including *integer.lifting* ⟨proof⟩

context
includes *integer.lifting* **and** *bit-operations-syntax*
begin

lemma [code]:
 $\langle \text{bit } (\text{int-of-integer } k) \ n \longleftrightarrow \text{bit } k \ n \rangle$
 ⟨proof⟩

lemma [code]:
 $\langle \text{NOT } (\text{int-of-integer } k) = \text{int-of-integer } (\text{NOT } k) \rangle$
 ⟨proof⟩

lemma [code]:
 $\langle \text{int-of-integer } k \ \text{AND} \ \text{int-of-integer } l = \text{int-of-integer } (k \ \text{AND} \ l) \rangle$
 ⟨proof⟩

lemma [code]:
 $\langle \text{int-of-integer } k \ \text{OR} \ \text{int-of-integer } l = \text{int-of-integer } (k \ \text{OR} \ l) \rangle$
 ⟨proof⟩

lemma [code]:
 $\langle \text{int-of-integer } k \ \text{XOR} \ \text{int-of-integer } l = \text{int-of-integer } (k \ \text{XOR} \ l) \rangle$
 ⟨proof⟩

lemma [code]:
 $\langle \text{push-bit } n \ (\text{int-of-integer } k) = \text{int-of-integer } (\text{push-bit } n \ k) \rangle$
 ⟨proof⟩

lemma [code]:
 $\langle \text{drop-bit } n \ (\text{int-of-integer } k) = \text{int-of-integer } (\text{drop-bit } n \ k) \rangle$

⟨*proof*⟩

lemma [*code*]:

⟨*take-bit* *n* (*int-of-integer* *k*) = *int-of-integer* (*take-bit* *n* *k*)⟩

⟨*proof*⟩

lemma [*code*]:

⟨*mask* *n* = *int-of-integer* (*mask* *n*)⟩

⟨*proof*⟩

lemma [*code*]:

⟨*set-bit* *n* (*int-of-integer* *k*) = *int-of-integer* (*set-bit* *n* *k*)⟩

⟨*proof*⟩

lemma [*code*]:

⟨*unset-bit* *n* (*int-of-integer* *k*) = *int-of-integer* (*unset-bit* *n* *k*)⟩

⟨*proof*⟩

lemma [*code*]:

⟨*flip-bit* *n* (*int-of-integer* *k*) = *int-of-integer* (*flip-bit* *n* *k*)⟩

⟨*proof*⟩

end

code-identifier

code-module *Code-Target-Int* \rightarrow

(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

theory *Code-Real-Approx-By-Float*

imports *Complex-Main* *Code-Target-Int*

begin

WARNING! This theory implements mathematical reals by machine reals (floats). This is inconsistent. See the proof of False at the end of the theory, where an equality on mathematical reals is (incorrectly) disproved by mapping it to machine reals.

The **value** command cannot display real results yet.

The only legitimate use of this theory is as a tool for code generation purposes.

context

begin

qualified definition *real-of-integer* :: *integer* \Rightarrow *real*

where [*code-abbrev*]: *real-of-integer* = *of-int* \circ *int-of-integer*

end

code-datatype *Code-Real-Approx-By-Float.real-of-integer* $\langle (/) :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \rangle$

lemma [*code-unfold del*]: $\text{numeral } k \equiv \text{real-of-rat } (\text{numeral } k)$
 $\langle \text{proof} \rangle$

lemma [*code-unfold del*]: $-\text{numeral } k \equiv \text{real-of-rat } (-\text{numeral } k)$
 $\langle \text{proof} \rangle$

context
begin

qualified definition *real-of-int* :: $\langle \text{int} \Rightarrow \text{real} \rangle$
where [*code-abbrev*]: $\langle \text{real-of-int} = \text{of-int} \rangle$

lemma [*code*]: $\text{real-of-int} = \text{Code-Real-Approx-By-Float.real-of-integer} \circ \text{integer-of-int}$
 $\langle \text{proof} \rangle$ **definition** *exp-real* :: $\langle \text{real} \Rightarrow \text{real} \rangle$
where [*code-abbrev, code del*]: $\langle \text{exp-real} = \text{exp} \rangle$

qualified definition *sin-real* :: $\langle \text{real} \Rightarrow \text{real} \rangle$
where [*code-abbrev, code del*]: $\langle \text{sin-real} = \text{sin} \rangle$

qualified definition *cos-real* :: $\langle \text{real} \Rightarrow \text{real} \rangle$
where [*code-abbrev, code del*]: $\langle \text{cos-real} = \text{cos} \rangle$

qualified definition *tan-real* :: $\langle \text{real} \Rightarrow \text{real} \rangle$
where [*code-abbrev, code del*]: $\langle \text{tan-real} = \text{tan} \rangle$

end

lemma [*code*]: $\langle r - s = r + (-s) \rangle$ **for** $r\ s :: \text{real}$
 $\langle \text{proof} \rangle$

lemma [*code*]: $\langle \text{inverse } r = 1 / r \rangle$ **for** $r :: \text{real}$
 $\langle \text{proof} \rangle$

lemma [*code*]: $\langle \text{Ratreal } r = (\text{let } (p, q) = \text{quotient-of } r \text{ in } \text{real-of-int } p / \text{real-of-int } q) \rangle$
 $\langle \text{proof} \rangle$

declare [[*code drop*:
 $\langle \text{HOL.equal} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool} \rangle$
 $\langle (\leq) :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool} \rangle$
 $\langle (<) :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool} \rangle$
 $\langle (+) :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \rangle$
 $\langle \text{uminus} :: \text{real} \Rightarrow \text{real} \rangle$
 $\langle (*) :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \rangle$
sqrt

```

    <ln :: real ⇒ real>
    pi
    arcsin
    arccos
    arctan]]

```

code-reserved (*SML*) *Real*

code-printing

```

    type-constructor real ↪
      (SML) real
      and (OCaml) float
      and (Haskell) Prelude.Double
| constant 0 :: real ↪
      (SML) 0.0
      and (OCaml) 0.0
      and (Haskell) 0.0
| constant 1 :: real ↪
      (SML) 1.0
      and (OCaml) 1.0
      and (Haskell) 1.0
| constant HOL.equal :: real ⇒ real ⇒ bool ↪
      (SML) Real.== ((-), (-))
      and (OCaml) Pervasives.(=)
      and (Haskell) infix 4 ==
| class-instance real :: HOL.equal => (Haskell) –
| constant (≤) :: real ⇒ real ⇒ bool ↪
      (SML) Real.<= ((-), (-))
      and (OCaml) Pervasives.(<=)
      and (Haskell) infix 4 <=
| constant (<) :: real ⇒ real ⇒ bool ↪
      (SML) Real.< ((-), (-))
      and (OCaml) Pervasives.<
      and (Haskell) infix 4 <
| constant (+) :: real ⇒ real ⇒ real ↪
      (SML) Real.+ ((-), (-))
      and (OCaml) Pervasives.(+. )
      and (Haskell) infixl 6 +
| constant (*) :: real ⇒ real ⇒ real ↪
      (SML) Real.* ((-), (-))
      and (Haskell) infixl 7 *
| constant uminus :: real ⇒ real ↪
      (SML) Real.~
      and (OCaml) Pervasives.( ~-. )
      and (Haskell) negate
| constant (–) :: real ⇒ real ⇒ real ↪
      (SML) Real.- ((-), (-))
      and (OCaml) Pervasives.( -. )
      and (Haskell) infixl 6 –

```

```

| constant (/) :: real ⇒ real ⇒ real →
  (SML) Real.'/ ((-), (-))
  and (OCaml) Pervasives.( '/. )
  and (Haskell) infixl 7 /
| constant sqrt :: real ⇒ real →
  (SML) Math.sqrt
  and (OCaml) Pervasives.sqrt
  and (Haskell) Prelude.sqrt
| constant Code-Real-Approx-By-Float.exp-real →
  (SML) Math.exp
  and (OCaml) Pervasives.exp
  and (Haskell) Prelude.exp
| constant ln →
  (SML) Math.ln
  and (OCaml) Pervasives.ln
  and (Haskell) Prelude.log
| constant Code-Real-Approx-By-Float.sin-real →
  (SML) Math.sin
  and (OCaml) Pervasives.sin
  and (Haskell) Prelude.sin
| constant Code-Real-Approx-By-Float.cos-real →
  (SML) Math.cos
  and (OCaml) Pervasives.cos
  and (Haskell) Prelude.cos
| constant Code-Real-Approx-By-Float.tan-real →
  (SML) Math.tan
  and (OCaml) Pervasives.tan
  and (Haskell) Prelude.tan
| constant pi →
  (SML) Math.pi

  and (Haskell) Prelude.pi
| constant arcsin →
  (SML) Math.asin
  and (OCaml) Pervasives.asin
  and (Haskell) Prelude.asin
| constant arccos →
  (SML) Math.scos
  and (OCaml) Pervasives.acos
  and (Haskell) Prelude.acos
| constant arctan →
  (SML) Math.atan
  and (OCaml) Pervasives.atan
  and (Haskell) Prelude.atan
| constant Code-Real-Approx-By-Float.real-of-integer →
  (SML) Real.fromInt
  and (OCaml) Pervasives.float/ (Big'-int.to'-int (-))
  and (Haskell) Prelude.fromIntegral (-)

```

```

notepad
begin
  ⟨proof⟩
end

end

```

123 Implementation of natural numbers by target-language integers

```

theory Code-Target-Nat
imports Code-Abstract-Nat
begin

```

123.1 Implementation for *nat*

```

context
includes integer.lifting
begin

```

```

lift-definition Nat :: integer ⇒ nat
is nat
⟨proof⟩

```

```

lemma [code-post]:
  Nat 0 = 0
  Nat 1 = 1
  Nat (numeral k) = numeral k
⟨proof⟩

```

```

lemma [code-abbrev]:
  integer-of-nat = of-nat
⟨proof⟩

```

```

lemma [code-unfold]:
  Int.nat (int-of-integer k) = nat-of-integer k
⟨proof⟩

```

```

lemma [code abstype]:
  Code-Target-Nat.Nat (integer-of-nat n) = n
⟨proof⟩

```

```

lemma [code abstract]:
  integer-of-nat (nat-of-integer k) = max 0 k
⟨proof⟩

```

```

lemma [code-abbrev]:
  nat-of-integer (numeral k) = nat-of-num k

```

```

  ⟨proof⟩

context
begin

qualified definition natural :: num ⇒ nat
  where [simp]: natural = nat-of-num

lemma [code-computation-unfold]:
  numeral = natural
  nat-of-num = natural
  ⟨proof⟩

end

lemma [code abstract]:
  integer-of-nat (nat-of-num n) = integer-of-num n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat 0 = 0
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat 1 = 1
  ⟨proof⟩

lemma [code]:
  Suc n = n + 1
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m + n) = of-nat m + of-nat n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m − n) = max 0 (of-nat m − of-nat n)
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m * n) = of-nat m * of-nat n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m div n) = of-nat m div of-nat n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m mod n) = of-nat m mod of-nat n

```

$\langle \text{proof} \rangle$

context

includes *integer.lifting*

begin

lemma *divmod-nat-code* [code]:

Euclidean-Rings.divmod-nat m n = (
 let $k = \text{integer-of-nat } m$; $l = \text{integer-of-nat } n$
 in *map-prod nat-of-integer nat-of-integer*
 (*if* $k = 0$ *then* $(0, 0)$
 else *if* $l = 0$ *then* $(0, k)$ *else*
Code-Numeral.divmod-abs k l)

$\langle \text{proof} \rangle$

end

lemma [code]:

divmod m n = *map-prod nat-of-integer nat-of-integer* (*divmod* m n)

$\langle \text{proof} \rangle$

lemma [code]:

HOL.equal m n = *HOL.equal* (*of-nat* $m :: \text{integer}$) (*of-nat* n)

$\langle \text{proof} \rangle$

lemma [code]:

$m \leq n \longleftrightarrow (\text{of-nat } m :: \text{integer}) \leq \text{of-nat } n$

$\langle \text{proof} \rangle$

lemma [code]:

$m < n \longleftrightarrow (\text{of-nat } m :: \text{integer}) < \text{of-nat } n$

$\langle \text{proof} \rangle$

lemma *num-of-nat-code* [code]:

num-of-nat = *num-of-integer* \circ *of-nat*

$\langle \text{proof} \rangle$

end

lemma (*in semiring-1*) *of-nat-code-if*:

of-nat n = (*if* $n = 0$ *then* 0
 else *let*
 (m, q) = *Euclidean-Rings.divmod-nat* n 2 ;
 $m' = 2 * \text{of-nat } m$
in if $q = 0$ *then* m' *else* $m' + 1$)

$\langle \text{proof} \rangle$

declare *of-nat-code-if* [code]

definition *int-of-nat* :: *nat* \Rightarrow *int* **where**
 [code-abbrev]: *int-of-nat* = *of-nat*

lemma [code]:
int-of-nat *n* = *int-of-integer* (*of-nat* *n*)
 ⟨proof⟩

lemma [code abstract]:
integer-of-nat (*nat* *k*) = *max* 0 (*integer-of-int* *k*)
including *integer.lifting* ⟨proof⟩

definition *char-of-nat* :: *nat* \Rightarrow *char*
where [code-abbrev]: *char-of-nat* = *char-of*

definition *nat-of-char* :: *char* \Rightarrow *nat*
where [code-abbrev]: *nat-of-char* = *of-char*

lemma [code]:
char-of-nat = *char-of-integer* \circ *integer-of-nat*
including *integer.lifting* ⟨proof⟩

lemma [code abstract]:
integer-of-nat (*nat-of-char* *c*) = *integer-of-char* *c*
 ⟨proof⟩

lemma *term-of-nat-code* [code]:
 — Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such
 that reconstructed terms can be fed back to the code generator
term-of-class.term-of *n* =
Code-Evaluation.App
 (*Code-Evaluation.Const* (*STR* "Code-Numeral.nat-of-integer")
 (*typerep.Typerep* (*STR* "fun")
 [*typerep.Typerep* (*STR* "Code-Numeral.integer") [],
typerep.Typerep (*STR* "Nat.nat") []]))
 (*term-of-class.term-of* (*integer-of-nat* *n*))
 ⟨proof⟩

lemma *nat-of-integer-code-post* [code-post]:
nat-of-integer 0 = 0
nat-of-integer 1 = 1
nat-of-integer (*numeral* *k*) = *numeral* *k*
including *integer.lifting* ⟨proof⟩

context
includes *integer.lifting* **and** *bit-operations-syntax*
begin

lemma [code]:
 ⟨*bit* *m* *n* \longleftrightarrow *bit* (*integer-of-nat* *m*) *n*⟩

$\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (m \text{ AND } n) = \text{integer-of-nat } m \text{ AND } \text{integer-of-nat } n \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (m \text{ OR } n) = \text{integer-of-nat } m \text{ OR } \text{integer-of-nat } n \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (m \text{ XOR } n) = \text{integer-of-nat } m \text{ XOR } \text{integer-of-nat } n \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (\text{push-bit } n \ m) = \text{push-bit } n \ (\text{integer-of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (\text{drop-bit } n \ m) = \text{drop-bit } n \ (\text{integer-of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (\text{take-bit } n \ m) = \text{take-bit } n \ (\text{integer-of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (\text{mask } n) = \text{mask } n \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (\text{set-bit } n \ m) = \text{set-bit } n \ (\text{integer-of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (\text{unset-bit } n \ m) = \text{unset-bit } n \ (\text{integer-of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{integer-of-nat } (\text{flip-bit } n \ m) = \text{flip-bit } n \ (\text{integer-of-nat } m) \rangle$
 $\langle \text{proof} \rangle$

end

code-identifier

code-module *Code-Target-Nat* \hookrightarrow
 (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

124 Implementation of natural and integer numbers by target-language integers

```

theory Code-Target-Numeral
imports Code-Target-Nat Code-Target-Int
begin

end

```

125 Preprocessor setup for floats implemented by target language numerals

```

theory Code-Target-Numeral-Float
imports Float Code-Target-Numeral
begin

lemma numeral-float-computation-unfold [code-computation-unfold]:
   $\langle \text{numeral } k = \text{Float } (\text{int-of-integer } (\text{Code-Numeral.positive } k)) \ 0 \rangle$ 
   $\langle - \text{numeral } k = \text{Float } (\text{int-of-integer } (\text{Code-Numeral.negative } k)) \ 0 \rangle$ 
   $\langle \text{proof} \rangle$ 

end

```

```

theory Complex-Order
  imports Complex-Main
begin

```

```

instantiation complex :: order begin

```

```

definition  $\langle x < y \longleftrightarrow \text{Re } x < \text{Re } y \wedge \text{Im } x = \text{Im } y \rangle$ 

```

```

definition  $\langle x \leq y \longleftrightarrow \text{Re } x \leq \text{Re } y \wedge \text{Im } x = \text{Im } y \rangle$ 

```

```

instance
   $\langle \text{proof} \rangle$ 
end

```

```

lemma nonnegative-complex-is-real:  $\langle (x::\text{complex}) \geq 0 \implies x \in \mathbb{R} \rangle$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma complex-is-real-iff-compare0:  $\langle (x::\text{complex}) \in \mathbb{R} \longleftrightarrow x \leq 0 \vee x \geq 0 \rangle$ 
   $\langle \text{proof} \rangle$ 

```

```

instance complex :: ordered-comm-ring
   $\langle \text{proof} \rangle$ 

```

```

instance complex :: ordered-real-vector
   $\langle \text{proof} \rangle$ 

```

```
instance complex :: ordered-cancel-comm-semiring
  ⟨proof⟩
```

```
end
```

126 Abstract type of association lists with unique keys

```
theory DAList
imports AList
begin
```

This was based on some existing fragments in the AFP-Collection framework.

126.1 Preliminaries

```
lemma distinct-map-fst-filter:
  distinct (map fst xs)  $\implies$  distinct (map fst (List.filter P xs))
  ⟨proof⟩
```

126.2 Type (*'key*, *'value*) *alist*

```
typedef ('key, 'value) alist = {xs :: ('key  $\times$  'value) list. (distinct  $\circ$  map fst) xs}
morphisms impl-of Alist
  ⟨proof⟩
```

```
setup-lifting type-definition-alist
```

```
lemma alist-ext: impl-of xs = impl-of ys  $\implies$  xs = ys
  ⟨proof⟩
```

```
lemma alist-eq-iff: xs = ys  $\longleftrightarrow$  impl-of xs = impl-of ys
  ⟨proof⟩
```

```
lemma impl-of-distinct [simp, intro]: distinct (map fst (impl-of xs))
  ⟨proof⟩
```

```
lemma impl-of-Alist:
  ⟨impl-of (Alist xs) = xs⟩ if ⟨distinct (map fst xs)⟩
  ⟨proof⟩
```

```
lemma Alist-impl-of [code abstype]: Alist (impl-of xs) = xs
  ⟨proof⟩
```

126.3 Primitive operations

lift-definition *lookup* :: ('key, 'value) alist \Rightarrow 'key \Rightarrow 'value option **is** map-of
 ⟨proof⟩

lift-definition *empty* :: ('key, 'value) alist **is** []
 ⟨proof⟩

lift-definition *update* :: 'key \Rightarrow 'value \Rightarrow ('key, 'value) alist \Rightarrow ('key, 'value) alist
is AList.update
 ⟨proof⟩

lift-definition *delete* :: 'key \Rightarrow ('key, 'value) alist \Rightarrow ('key, 'value) alist
is AList.delete
 ⟨proof⟩

lift-definition *map-entry* ::
 'key \Rightarrow ('value \Rightarrow 'value) \Rightarrow ('key, 'value) alist \Rightarrow ('key, 'value) alist
is AList.map-entry
 ⟨proof⟩

lift-definition *filter* :: ('key \times 'value \Rightarrow bool) \Rightarrow ('key, 'value) alist \Rightarrow ('key, 'value)
 alist
is List.filter
 ⟨proof⟩

lift-definition *map-default* ::
 'key \Rightarrow 'value \Rightarrow ('value \Rightarrow 'value) \Rightarrow ('key, 'value) alist \Rightarrow ('key, 'value) alist
is AList.map-default
 ⟨proof⟩

126.4 Abstract operation properties

lemma *lookup-empty* [simp]: lookup empty k = None
 ⟨proof⟩

lemma *lookup-update*:
 lookup (update k1 v xs) k2 = (if k1 = k2 then Some v else lookup xs k2)
 ⟨proof⟩

lemma *lookup-update-eq* [simp]:
 k1 = k2 \implies lookup (update k1 v xs) k2 = Some v
 ⟨proof⟩

lemma *lookup-update-neq* [simp]:
 k1 \neq k2 \implies lookup (update k1 v xs) k2 = lookup xs k2
 ⟨proof⟩

lemma *update-update-eq* [simp]:

$k1 = k2 \implies \text{update } k2 \ v2 \ (\text{update } k1 \ v1 \ xs) = \text{update } k2 \ v2 \ xs$
 $\langle \text{proof} \rangle$

lemma *lookup-delete* [simp]: $\text{lookup } (\text{delete } k \ al) = (\text{lookup } al)(k := \text{None})$
 $\langle \text{proof} \rangle$

126.5 Further operations

126.5.1 Equality

instantiation *alist* :: (equal, equal) equal
begin

definition *HOL.equal* ($xs :: ('a, 'b) \text{alist}$) $ys == \text{impl-of } xs = \text{impl-of } ys$

instance
 $\langle \text{proof} \rangle$

end

126.5.2 Size

instantiation *alist* :: (type, type) size
begin

definition *size* ($al :: ('a, 'b) \text{alist}$) = $\text{length } (\text{impl-of } al)$

instance $\langle \text{proof} \rangle$

end

126.6 Quickcheck generators

context
includes *state-combinator-syntax* **and** *term-syntax*
begin

definition
 $\text{valterm-empty} :: ('key :: \text{typerep}, 'value :: \text{typerep}) \text{alist} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term})$
where $\text{valterm-empty} = \text{Code-Evaluation.valtermify empty}$

definition
 $\text{valterm-update} :: 'key :: \text{typerep} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow$
 $'value :: \text{typerep} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow$
 $('key, 'value) \text{alist} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow$
 $('key, 'value) \text{alist} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term})$ **where**
 $[\text{code-unfold}]: \text{valterm-update } k \ v \ a = \text{Code-Evaluation.valtermify update } \{\cdot\} \ k \ \{\cdot\}$
 $v \ \{\cdot\} \ a$

fun *random-aux-alist*

where

```

  random-aux-alist i j =
    (if i = 0 then Pair valterm-empty
     else Quickcheck-Random.collapse
      (Random.select-weight
       [(i, Quickcheck-Random.random j  $\circ \rightarrow$  ( $\lambda k.$  Quickcheck-Random.random j
 $\circ \rightarrow$ 
         ( $\lambda v.$  random-aux-alist (i - 1) j  $\circ \rightarrow$  ( $\lambda a.$  Pair (valterm-update k v a))))),
        (1, Pair valterm-empty)]))

```

end

instantiation alist :: (random, random) random
begin

definition random-alist

where

random-alist i = random-aux-alist i i

instance $\langle \text{proof} \rangle$

end

instantiation alist :: (exhaustive, exhaustive) exhaustive
begin

fun exhaustive-alist ::

(($'a, 'b$) alist \Rightarrow (bool \times term list) option) \Rightarrow natural \Rightarrow (bool \times term list) option

where

```

  exhaustive-alist f i =
    (if i = 0 then None
     else
      case f empty of
        Some ts  $\Rightarrow$  Some ts
      | None  $\Rightarrow$ 
        exhaustive-alist
          ( $\lambda a.$  Quickcheck-Exhaustive.exhaustive
           ( $\lambda k.$  Quickcheck-Exhaustive.exhaustive ( $\lambda v.$  f (update k v a)) (i - 1))
          (i - 1))
          (i - 1))

```

instance $\langle \text{proof} \rangle$

end

instantiation alist :: (full-exhaustive, full-exhaustive) full-exhaustive
begin

fun full-exhaustive-alist ::

```

((('a, 'b) alist × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ natural ⇒
 (bool × term list) option)
where
  full-exhaustive-alist f i =
    (if i = 0 then None
     else
      case f valterm-empty of
        Some ts ⇒ Some ts
      | None ⇒
        full-exhaustive-alist
          (λa.
            Quickcheck-Exhaustive.full-exhaustive
              (λk. Quickcheck-Exhaustive.full-exhaustive (λv. f (valterm-update k v
a)) (i - 1))
              (i - 1))
            (i - 1))
instance ⟨proof⟩
end

```

127 alist is a BNF

```

lift-bnf (dead 'k, set: 'v) alist [wits: [] :: ('k × 'v) list] for map: map rel: rel
  ⟨proof⟩

hide-const valterm-empty valterm-update random-aux-alist

hide-fact (open) lookup-def empty-def update-def delete-def map-entry-def filter-def
  map-default-def
hide-const (open) impl-of lookup empty update delete map-entry filter map-default
  map set rel

end

```

128 Multisets partially implemented by association lists

```

theory DAList-Multiset
imports Multiset DAList
begin

```

Raw operations on lists

```

definition join-raw ::
  ('key ⇒ 'val × 'val ⇒ 'val) ⇒
  ('key × 'val) list ⇒ ('key × 'val) list ⇒ ('key × 'val) list
where join-raw f xs ys = foldr (λ(k, v). map-default k v (λv'. f k (v', v))) ys xs

```


lemma *join-row-Nil* [simp]: *join-row f xs [] = xs*
 ⟨proof⟩

lemma *join-row-Cons* [simp]:
join-row f xs ((k, v) # ys) = map-default k v (λv'. f k (v', v)) (join-row f xs ys)
 ⟨proof⟩

lemma *map-of-join-row*:
assumes *distinct (map fst ys)*
shows *map-of (join-row f xs ys) x =*
 (*case map-of xs x of*
 None ⇒ map-of ys x
 | *Some v ⇒ (case map-of ys x of None ⇒ Some v | Some v' ⇒ Some (f x (v,*
v')))))
 ⟨proof⟩

lemma *distinct-join-row*:
assumes *distinct (map fst xs)*
shows *distinct (map fst (join-row f xs ys))*
 ⟨proof⟩

definition *subtract-entries-row xs ys = foldr (λ(k, v). AList.map-entry k (λv'. v' - v)) ys xs*

lemma *map-of-subtract-entries-row*:
assumes *distinct (map fst ys)*
shows *map-of (subtract-entries-row xs ys) x =*
 (*case map-of xs x of*
 None ⇒ None
 | *Some v ⇒ (case map-of ys x of None ⇒ Some v | Some v' ⇒ Some (v - v'))*)
 ⟨proof⟩

lemma *distinct-subtract-entries-row*:
assumes *distinct (map fst xs)*
shows *distinct (map fst (subtract-entries-row xs ys))*
 ⟨proof⟩

Operations on alists with distinct keys

lift-definition *join :: ('a ⇒ 'b × 'b ⇒ 'b) ⇒ ('a, 'b) alist ⇒ ('a, 'b) alist ⇒ ('a, 'b) alist*
is *join-row*
 ⟨proof⟩

lift-definition *subtract-entries :: ('a, ('b :: minus)) alist ⇒ ('a, 'b) alist ⇒ ('a, 'b) alist*
is *subtract-entries-row*
 ⟨proof⟩

Implementing multisets by means of association lists

definition *count-of :: ('a × nat) list ⇒ 'a ⇒ nat*

where $\text{count-of } xs \ x = (\text{case map-of } xs \ x \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } n \Rightarrow n)$

lemma *count-of-multiset*: $\text{finite } \{x. 0 < \text{count-of } xs \ x\}$
 $\langle \text{proof} \rangle$

lemma *count-simps* [simp]:
 $\text{count-of } [] = (\lambda-. 0)$
 $\text{count-of } ((x, n) \# xs) = (\lambda y. \text{if } x = y \text{ then } n \text{ else } \text{count-of } xs \ y)$
 $\langle \text{proof} \rangle$

lemma *count-of-empty*: $x \notin \text{fst } \text{'set } xs \Longrightarrow \text{count-of } xs \ x = 0$
 $\langle \text{proof} \rangle$

lemma *count-of-filter*: $\text{count-of } (\text{List.filter } (P \circ \text{fst}) \ xs) \ x = (\text{if } P \ x \text{ then } \text{count-of } xs \ x \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *count-of-map-default* [simp]:
 $\text{count-of } (\text{map-default } x \ b \ (\lambda x. x + b) \ xs) \ y =$
 $(\text{if } x = y \text{ then } \text{count-of } xs \ x + b \text{ else } \text{count-of } xs \ y)$
 $\langle \text{proof} \rangle$

lemma *count-of-join-row*:
 $\text{distinct } (\text{map fst } ys) \Longrightarrow$
 $\text{count-of } xs \ x + \text{count-of } ys \ x = \text{count-of } (\text{join-row } (\lambda x \ (x, y). x + y) \ xs \ ys) \ x$
 $\langle \text{proof} \rangle$

lemma *count-of-subtract-entries-row*:
 $\text{distinct } (\text{map fst } ys) \Longrightarrow$
 $\text{count-of } xs \ x - \text{count-of } ys \ x = \text{count-of } (\text{subtract-entries-row } xs \ ys) \ x$
 $\langle \text{proof} \rangle$

Code equations for multiset operations

definition *Bag* :: $('a, \text{nat}) \text{ alist} \Rightarrow 'a \text{ multiset}$
where $\text{Bag } xs = \text{Abs-multiset } (\text{count-of } (\text{DAList.impl-of } xs))$

code-datatype *Bag*

lemma *count-Bag* [simp, code]: $\text{count } (\text{Bag } xs) = \text{count-of } (\text{DAList.impl-of } xs)$
 $\langle \text{proof} \rangle$

lemma *Bag-eq*:
 $\langle \text{Bag } ms = (\sum (a, n) \leftarrow \text{alist.impl-of } ms. \text{replicate-mset } n \ a) \rangle$
for $ms :: \langle ('a, \text{nat}) \text{ alist} \rangle$
 $\langle \text{proof} \rangle$

lemma *Mempty-Bag* [code]: $\{\#\} = \text{Bag } (\text{DAList.empty})$
 $\langle \text{proof} \rangle$

lift-definition *is-empty-Bag-impl* :: (*'a*, *nat*) *alist* \Rightarrow *bool* **is**
 $\lambda xs. \text{list-all } (\lambda x. \text{snd } x = 0) \text{ } xs \langle \text{proof} \rangle$

lemma *is-empty-Bag* [code]: *Multiset.is-empty* (*Bag xs*) \longleftrightarrow *is-empty-Bag-impl xs*
 $\langle \text{proof} \rangle$

lemma *union-Bag* [code]: *Bag xs* + *Bag ys* = *Bag* (*join* ($\lambda x \text{ } (n1, n2). n1 + n2$) *xs ys*)
 $\langle \text{proof} \rangle$

lemma *add-mset-Bag* [code]: *add-mset x* (*Bag xs*) =
Bag (*join* ($\lambda x \text{ } (n1, n2). n1 + n2$) (*DAList.update x 1 DAList.empty*) *xs*)
 $\langle \text{proof} \rangle$

lemma *minus-Bag* [code]: *Bag xs* − *Bag ys* = *Bag* (*subtract-entries xs ys*)
 $\langle \text{proof} \rangle$

lemma *filter-Bag* [code]: *filter-mset P* (*Bag xs*) = *Bag* (*DAList.filter* (*P* \circ *fst*) *xs*)
 $\langle \text{proof} \rangle$

lemma *mset-eq* [code]: *HOL.equal* (*m1* :: *'a* :: *equal multiset*) *m2* \longleftrightarrow *m1* $\subseteq\#$ *m2* \wedge
m2 $\subseteq\#$ *m1*
 $\langle \text{proof} \rangle$

By default the code for $<$ is $(xs < ys) = (xs \leq ys \wedge xs \neq ys)$. With equality implemented by \leq , this leads to three calls of \leq . Here is a more efficient version:

lemma *mset-less*[code]: *xs* $\subset\#$ (*ys* :: *'a multiset*) \longleftrightarrow *xs* $\subseteq\#$ *ys* $\wedge \neg ys \subseteq\# xs$
 $\langle \text{proof} \rangle$

lemma *mset-less-eq-Bag0*:
Bag xs $\subseteq\# A \longleftrightarrow (\forall (x, n) \in \text{set } (DAList.impl-of xs). \text{count-of } (DAList.impl-of xs) x \leq \text{count } A x)$
(is ?lhs \longleftrightarrow ?rhs)
 $\langle \text{proof} \rangle$

lemma *mset-less-eq-Bag* [code]:
Bag xs $\subseteq\# (A :: 'a \text{ multiset}) \longleftrightarrow (\forall (x, n) \in \text{set } (DAList.impl-of xs). n \leq \text{count } A x)$
 $\langle \text{proof} \rangle$

declare *inter-mset-def* [code]
declare *union-mset-def* [code]
declare *mset.simps* [code]

fun *fold-impl* :: (*'a* \Rightarrow *nat* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow *'b* \Rightarrow (*'a* \times *nat*) *list* \Rightarrow *'b*
where

$$\text{fold-impl } fn \ e \ ((a,n) \# \ ms) = (\text{fold-impl } fn \ ((fn \ a \ n) \ e) \ ms)$$

$$| \text{fold-impl } fn \ e \ [] = e$$

context
begin

qualified definition $fold :: ('a \Rightarrow nat \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow ('a, nat) \text{ alist} \Rightarrow 'b$
where $fold \ f \ e \ al = \text{fold-impl } f \ e \ (DAList.\text{impl-of } al)$

end

context *comp-fun-commute*
begin

lemma *DAList-Multiset-fold*:
assumes $fn: \bigwedge a \ n \ x. \ fn \ a \ n \ x = (f \ a \ \frown n) \ x$
shows $\text{fold-mset } f \ e \ (Bag \ al) = DAList-Multiset.fold \ fn \ e \ al$
 $\langle proof \rangle$

end

context
begin

private lift-definition $single-alist-entry :: 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ alist} \text{ is } \lambda a \ b. [(a, b)]$
 $\langle proof \rangle$

lemma *image-mset-Bag* [code]:
 $\text{image-mset } f \ (Bag \ ms) =$
 $DAList-Multiset.fold \ (\lambda a \ n \ m. \ Bag \ (single-alist-entry \ (f \ a) \ n) + m) \ \{\#\} \ ms$
 $\langle proof \rangle$

end

— we cannot use $\lambda a \ n. (+) \ (a * n)$ for folding, since $(*)$ is not defined in *comm-monoid-add*

lemma *sum-mset-Bag* [code]: $\text{sum-mset } (Bag \ ms) = DAList-Multiset.fold \ (\lambda a \ n. ((+) \ a) \ \frown n) \ 0 \ ms$
 $\langle proof \rangle$

lemma *prod-mset-Bag* [code]: $\text{prod-mset } (Bag \ ms) = DAList-Multiset.fold \ (\lambda a \ n. ((*)) \ a) \ \frown n) \ 1 \ ms$
 $\langle proof \rangle$

lemma *size-fold*: $\text{size } A = \text{fold-mset } (\lambda -. \text{Suc}) \ 0 \ A \ (\text{is } - = \text{fold-mset } ?f \ -)$
 $\langle proof \rangle$

lemma *size-Bag* [code]: $\text{size } (Bag \ ms) = DAList-Multiset.fold \ (\lambda a \ n. (+) \ n) \ 0 \ ms$
 $\langle proof \rangle$

lemma *set-mset-fold*: *set-mset A = fold-mset insert {} A (is - = fold-mset ?f - -)*
 ⟨*proof*⟩

lemma *set-mset-Bag*[*code*]:
set-mset (Bag ms) = DAList-Multiset.fold (λa n. (if n = 0 then (λm. m) else insert a)) {} ms
 ⟨*proof*⟩

lemma *sorted-list-of-multiset-Bag* [*code*]:
 ⟨*sorted-list-of-multiset (Bag ms) = concat (map (λ(a, n). replicate n a) (sort-key fst (DAList.impl-of ms)))*⟩ (is ⟨*?lhs = ?rhs*⟩)
 ⟨*proof*⟩

instantiation *multiset* :: (*exhaustive*) *exhaustive*
begin

definition *exhaustive-multiset* ::
 (*'a multiset ⇒ (bool × term list) option*) ⇒ *natural ⇒ (bool × term list) option*
where *exhaustive-multiset f i = Quickcheck-Exhaustive.exhaustive (λxs. f (Bag xs)) i*

instance ⟨*proof*⟩

end

end

129 Implementation of Red-Black Trees

theory *RBT-Impl*
imports *Main*
begin

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

129.1 Datatype of RB trees

datatype *color* = *R* | *B*
datatype (*'a, 'b*) *rbt* = *Empty* | *Branch color ('a, 'b) rbt 'a 'b ('a, 'b) rbt*

lemma *rbt-cases*:
obtains (*Empty*) *t = Empty*
 | (*Red*) *l k v r* **where** *t = Branch R l k v r*
 | (*Black*) *l k v r* **where** *t = Branch B l k v r*
 ⟨*proof*⟩

129.2 Tree properties

129.2.1 Content of a tree

primrec *entries* :: ('a, 'b) rbt \Rightarrow ('a \times 'b) list

where

entries Empty = []

| *entries* (Branch - l k v r) = *entries* l @ (k,v) # *entries* r

abbreviation (input) *entry-in-tree* :: 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow bool

where

entry-in-tree k v t \equiv (k, v) \in set (*entries* t)

definition *keys* :: ('a, 'b) rbt \Rightarrow 'a list **where**

keys t = map fst (*entries* t)

lemma *keys-simps* [simp]:

keys Empty = []

keys (Branch c l k v r) = *keys* l @ k # *keys* r

\langle proof \rangle

lemma *entry-in-tree-keys*:

assumes (k, v) \in set (*entries* t)

shows k \in set (*keys* t)

\langle proof \rangle

lemma *keys-entries*:

k \in set (*keys* t) \longleftrightarrow (\exists v. (k, v) \in set (*entries* t))

\langle proof \rangle

lemma *non-empty-rbt-keys*:

t \neq rbt.Empty \implies *keys* t \neq []

\langle proof \rangle

129.2.2 Search tree properties

context ord begin

definition *rbt-less* :: 'a \Rightarrow ('a, 'b) rbt \Rightarrow bool

where

rbt-less-prop: *rbt-less* k t \longleftrightarrow (\forall x \in set (*keys* t). x < k)

abbreviation *rbt-less-symbol* (**infix** <|«> 50)

where t |« x \equiv *rbt-less* x t

definition *rbt-greater* :: 'a \Rightarrow ('a, 'b) rbt \Rightarrow bool (**infix** <«|> 50)

where

rbt-greater-prop: *rbt-greater* k t = (\forall x \in set (*keys* t). k < x)

lemma *rbt-less-simps* [simp]:

$Empty \mid \ll k = True$
 $Branch\ c\ lt\ kt\ v\ rt \mid \ll k \longleftrightarrow kt < k \wedge lt \mid \ll k \wedge rt \mid \ll k$
 $\langle proof \rangle$

lemma *rbt-greater-simps* [simp]:

$k \ll Empty = True$
 $k \ll (Branch\ c\ lt\ kt\ v\ rt) \longleftrightarrow k < kt \wedge k \ll lt \wedge k \ll rt$
 $\langle proof \rangle$

lemmas *rbt-ord-props* = *rbt-less-prop* *rbt-greater-prop*

lemmas *rbt-greater-nit* = *rbt-greater-prop* *entry-in-tree-keys*

lemmas *rbt-less-nit* = *rbt-less-prop* *entry-in-tree-keys*

lemma (*in order*)

shows *rbt-less-eq-trans*: $l \mid \ll u \Longrightarrow u \leq v \Longrightarrow l \mid \ll v$
and *rbt-less-trans*: $t \mid \ll x \Longrightarrow x < y \Longrightarrow t \mid \ll y$
and *rbt-greater-eq-trans*: $u \leq v \Longrightarrow v \ll r \Longrightarrow u \ll r$
and *rbt-greater-trans*: $x < y \Longrightarrow y \ll t \Longrightarrow x \ll t$
 $\langle proof \rangle$

primrec *rbt-sorted* :: (*'a*, *'b*) *rbt* \Rightarrow *bool*

where

$rbt\text{-}sorted\ Empty = True$
 $\mid rbt\text{-}sorted\ (Branch\ c\ l\ k\ v\ r) = (l \mid \ll k \wedge k \ll r \wedge rbt\text{-}sorted\ l \wedge rbt\text{-}sorted\ r)$

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:

$rbt\text{-}sorted\ t \Longrightarrow List.sorted\ (map\ fst\ (entries\ t))$
 $\langle proof \rangle$

lemma *distinct-entries*:

$rbt\text{-}sorted\ t \Longrightarrow distinct\ (map\ fst\ (entries\ t))$
 $\langle proof \rangle$

lemma *distinct-keys*:

$rbt\text{-}sorted\ t \Longrightarrow distinct\ (keys\ t)$
 $\langle proof \rangle$

129.2.3 Tree lookup

primrec (*in ord*) *rbt-lookup* :: (*'a*, *'b*) *rbt* \Rightarrow *'a* \rightarrow *'b*

where

$rbt\text{-}lookup\ Empty\ k = None$
 $\mid rbt\text{-}lookup\ (Branch\ -\ l\ x\ y\ r)\ k =$
 $(if\ k < x\ then\ rbt\text{-}lookup\ l\ k\ else\ if\ x < k\ then\ rbt\text{-}lookup\ r\ k\ else\ Some\ y)$

lemma *rbt-lookup-keys*: $\text{rbt-sorted } t \implies \text{dom } (\text{rbt-lookup } t) = \text{set } (\text{keys } t)$
 $\langle \text{proof} \rangle$

lemma *dom-rbt-lookup-Branch*:
 $\text{rbt-sorted } (\text{Branch } c \ t1 \ k \ v \ t2) \implies$
 $\text{dom } (\text{rbt-lookup } (\text{Branch } c \ t1 \ k \ v \ t2))$
 $= \text{Set.insert } k \ (\text{dom } (\text{rbt-lookup } t1) \cup \text{dom } (\text{rbt-lookup } t2))$
 $\langle \text{proof} \rangle$

lemma *finite-dom-rbt-lookup* [*simp*, *intro!*]: $\text{finite } (\text{dom } (\text{rbt-lookup } t))$
 $\langle \text{proof} \rangle$

end

context *ord* **begin**

lemma *rbt-lookup-rbt-less*[*simp*]: $t \mid\ll k \implies \text{rbt-lookup } t \ k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-greater*[*simp*]: $k \mid\ll t \implies \text{rbt-lookup } t \ k = \text{None}$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-Empty*: $\text{rbt-lookup } \text{Empty} = \text{Map.empty}$
 $\langle \text{proof} \rangle$

end

context *linorder* **begin**

lemma *map-of-entries*:
 $\text{rbt-sorted } t \implies \text{map-of } (\text{entries } t) = \text{rbt-lookup } t$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-in-tree*: $\text{rbt-sorted } t \implies \text{rbt-lookup } t \ k = \text{Some } v \longleftrightarrow (k, v) \in \text{set } (\text{entries } t)$
 $\langle \text{proof} \rangle$

lemma *set-entries-inject*:
assumes *rbt-sorted*: $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$
shows $\text{set } (\text{entries } t1) = \text{set } (\text{entries } t2) \longleftrightarrow \text{entries } t1 = \text{entries } t2$
 $\langle \text{proof} \rangle$

lemma *entries-eqI*:
assumes *rbt-sorted*: $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$
assumes *rbt-lookup*: $\text{rbt-lookup } t1 = \text{rbt-lookup } t2$
shows $\text{entries } t1 = \text{entries } t2$
 $\langle \text{proof} \rangle$

lemma *entries-rbt-lookup*:
assumes *rbt-sorted t1 rbt-sorted t2*
shows *entries t1 = entries t2 \longleftrightarrow rbt-lookup t1 = rbt-lookup t2*
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-from-in-tree*:
assumes *rbt-sorted t1 rbt-sorted t2*
and $\bigwedge v. (k, v) \in \text{set } (\text{entries } t1) \longleftrightarrow (k, v) \in \text{set } (\text{entries } t2)$
shows *rbt-lookup t1 k = rbt-lookup t2 k*
 $\langle \text{proof} \rangle$

end

129.2.4 Red-black properties

primrec *color-of* :: $('a, 'b) \text{rbt} \Rightarrow \text{color}$
where
color-of Empty = B
 $| \text{color-of } (\text{Branch } c \text{ - - -}) = c$

primrec *bheight* :: $('a, 'b) \text{rbt} \Rightarrow \text{nat}$
where
bheight Empty = 0
 $| \text{bheight } (\text{Branch } c \text{ lt k v rt}) = (\text{if } c = B \text{ then } \text{Suc } (\text{bheight } lt) \text{ else } \text{bheight } lt)$

primrec *inv1* :: $('a, 'b) \text{rbt} \Rightarrow \text{bool}$
where
inv1 Empty = True
 $| \text{inv1 } (\text{Branch } c \text{ lt k v rt}) \longleftrightarrow \text{inv1 } lt \wedge \text{inv1 } rt \wedge (c = B \vee \text{color-of } lt = B \wedge \text{color-of } rt = B)$

primrec *inv1l* :: $('a, 'b) \text{rbt} \Rightarrow \text{bool}$ — Weaker version
where
inv1l Empty = True
 $| \text{inv1l } (\text{Branch } c \text{ l k v r}) = (\text{inv1 } l \wedge \text{inv1 } r)$
lemma *[simp]: inv1 t \implies inv1l t* $\langle \text{proof} \rangle$

primrec *inv2* :: $('a, 'b) \text{rbt} \Rightarrow \text{bool}$
where
inv2 Empty = True
 $| \text{inv2 } (\text{Branch } c \text{ lt k v rt}) = (\text{inv2 } lt \wedge \text{inv2 } rt \wedge \text{bheight } lt = \text{bheight } rt)$

context *ord* **begin**

definition *is-rbt* :: $('a, 'b) \text{rbt} \Rightarrow \text{bool}$ **where**
is-rbt t \longleftrightarrow inv1 t \wedge inv2 t \wedge color-of t = B \wedge rbt-sorted t

lemma *is-rbt-rbt-sorted [simp]:*
is-rbt t \implies rbt-sorted t $\langle \text{proof} \rangle$

theorem *Empty-is-rbt* [simp]:
is-rbt Empty <proof>

end

129.3 Insertion

The function definitions are based on the book by Okasaki.

fun

balance :: ('a,'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a,'b) rbt \Rightarrow ('a,'b) rbt

where

balance (Branch R a w x b) s t (Branch R c y z d) = Branch R (Branch B a w x b) s t (Branch B c y z d) |

balance (Branch R (Branch R a w x b) s t c) y z d = Branch R (Branch B a w x b) s t (Branch B c y z d) |

balance (Branch R a w x (Branch R b s t c)) y z d = Branch R (Branch B a w x b) s t (Branch B c y z d) |

balance a w x (Branch R b s t (Branch R c y z d)) = Branch R (Branch B a w x b) s t (Branch B c y z d) |

balance a w x (Branch R (Branch R b s t c) y z d) = Branch R (Branch B a w x b) s t (Branch B c y z d) |

balance a s t b = Branch B a s t b

lemma *balance-inv1*: $\llbracket \text{inv1 } l; \text{ inv1 } r \rrbracket \Longrightarrow \text{inv1 } (\text{balance } l \text{ k } v \text{ r})$
 <proof>

lemma *balance-bheight*: $\text{bheight } l = \text{bheight } r \Longrightarrow \text{bheight } (\text{balance } l \text{ k } v \text{ r}) = \text{Suc } (\text{bheight } l)$
 <proof>

lemma *balance-inv2*:

assumes *inv2* l *inv2* r $\text{bheight } l = \text{bheight } r$

shows *inv2* (balance l k v r)

<proof>

context *ord* **begin**

lemma *balance-rbt-greater*[simp]: $(v \ll \text{balance } a \text{ k } x \text{ b}) = (v \ll a \wedge v \ll b \wedge v < k)$
 <proof>

lemma *balance-rbt-less*[simp]: $(\text{balance } a \text{ k } x \text{ b} \ll v) = (a \ll v \wedge b \ll v \wedge k < v)$
 <proof>

end

lemma (in *linorder*) *balance-rbt-sorted*:
 fixes k :: 'a

else if $k > x$ then Branch $R \ l \ x \ y \ (rbt-ins \ f \ k \ v \ r)$
else Branch $R \ l \ x \ (f \ k \ y \ v) \ r)$

lemma *ins-inv1-inv2:*

assumes *inv1 t inv2 t*

shows *inv2 (rbt-ins f k x t) bheight (rbt-ins f k x t) = bheight t*

color-of t = B \implies inv1 (rbt-ins f k x t) inv1l (rbt-ins f k x t)

<proof>

end

context *linorder* **begin**

lemma *ins-rbt-greater[simp]:* $(v \ll rbt-ins \ f \ (k :: 'a) \ x \ t) = (v \ll t \wedge k > v)$
<proof>

lemma *ins-rbt-less[simp]:* $(rbt-ins \ f \ k \ x \ t \mid\ll v) = (t \mid\ll v \wedge k < v)$
<proof>

lemma *ins-rbt-sorted[simp]:* $rbt-sorted \ t \implies rbt-sorted \ (rbt-ins \ f \ k \ x \ t)$
<proof>

lemma *keys-ins:* $set \ (keys \ (rbt-ins \ f \ k \ v \ t)) = \{ k \} \cup set \ (keys \ t)$
<proof>

lemma *rbt-lookup-ins:*

fixes *k :: 'a*

assumes *rbt-sorted t*

shows *rbt-lookup (rbt-ins f k v t) x = ((rbt-lookup t)(k |-> case rbt-lookup t k*

of None \Rightarrow v

| Some w \Rightarrow f k w v)) x

<proof>

end

context *ord* **begin**

definition *rbt-insert-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \ rbt \Rightarrow ('a, 'b) \ rbt$

where *rbt-insert-with-key f k v t = paint B (rbt-ins f k v t)*

definition *rbt-insertw-def:* $rbt-insert-with \ f = rbt-insert-with-key \ (\lambda-. \ f)$

definition *rbt-insert* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b) \ rbt \Rightarrow ('a, 'b) \ rbt$ **where**
rbt-insert = rbt-insert-with-key ($\lambda-$ - nv. nv)

end

context *linorder* **begin**

lemma *rbt-insertwk-rbt-sorted:* $rbt-sorted \ t \implies rbt-sorted \ (rbt-insert-with-key \ f \ (k$

$:: 'a) x t)$
 $\langle proof \rangle$

theorem *rbt-insertwk-is-rbt*:
assumes *inv*: *is-rbt* *t*
shows *is-rbt* (*rbt-insert-with-key* *f k x t*)
 $\langle proof \rangle$

lemma *rbt-lookup-rbt-insertwk*:
assumes *rbt-sorted* *t*
shows *rbt-lookup* (*rbt-insert-with-key* *f k v t*) *x* = ((*rbt-lookup* *t*)(*k* | \rightarrow case
rbt-lookup *t k* of None \Rightarrow *v*
 $|$ Some *w* \Rightarrow *f k w v*)) *x*
 $\langle proof \rangle$

lemma *rbt-insertw-rbt-sorted*: *rbt-sorted* *t* \Longrightarrow *rbt-sorted* (*rbt-insert-with* *f k v t*)
 $\langle proof \rangle$
theorem *rbt-insertw-is-rbt*: *is-rbt* *t* \Longrightarrow *is-rbt* (*rbt-insert-with* *f k v t*)
 $\langle proof \rangle$

lemma *rbt-lookup-rbt-insertw*:
is-rbt *t* \Longrightarrow
rbt-lookup (*rbt-insert-with* *f k v t*) =
(*rbt-lookup* *t*)(*k* \mapsto (if *k* \in dom (*rbt-lookup* *t*) then *f* (the (*rbt-lookup* *t k*)) *v*
else *v*))
 $\langle proof \rangle$

lemma *rbt-insert-rbt-sorted*: *rbt-sorted* *t* \Longrightarrow *rbt-sorted* (*rbt-insert* *k v t*)
 $\langle proof \rangle$
theorem *rbt-insert-is-rbt* [*simp*]: *is-rbt* *t* \Longrightarrow *is-rbt* (*rbt-insert* *k v t*)
 $\langle proof \rangle$

lemma *rbt-lookup-rbt-insert*: *is-rbt* *t* \Longrightarrow *rbt-lookup* (*rbt-insert* *k v t*) = (*rbt-lookup*
t)(*k* \mapsto *v*)
 $\langle proof \rangle$

end

129.4 Deletion

lemma *bheight-paintR* [*simp*]: *color-of* *t* = *B* \Longrightarrow *bheight* (*paint* *R t*) = *bheight* *t*
 $- 1$
 $\langle proof \rangle$

The function definitions are based on the Haskell code by Stefan Kahrs
at <http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html>.

fun
balance-left $:: ('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$
where

$\text{balance-left } (\text{Branch } R \ a \ k \ x \ b) \ s \ y \ c = \text{Branch } R \ (\text{Branch } B \ a \ k \ x \ b) \ s \ y \ c \mid$
 $\text{balance-left } \text{bl } k \ x \ (\text{Branch } B \ a \ s \ y \ b) = \text{balance } \text{bl } k \ x \ (\text{Branch } R \ a \ s \ y \ b) \mid$
 $\text{balance-left } \text{bl } k \ x \ (\text{Branch } R \ (\text{Branch } B \ a \ s \ y \ b) \ t \ z \ c) = \text{Branch } R \ (\text{Branch } B \ \text{bl}$
 $k \ x \ a) \ s \ y \ (\text{balance } b \ t \ z \ (\text{paint } R \ c)) \mid$
 $\text{balance-left } t \ k \ x \ s = \text{Empty}$

lemma *balance-left-inv2-with-inv1*:

assumes $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt + 1 = \text{bheight } rt \ \text{inv1 } rt$

shows $\text{bheight } (\text{balance-left } lt \ k \ v \ rt) = \text{bheight } lt + 1$

and $\text{inv2 } (\text{balance-left } lt \ k \ v \ rt)$

<proof>

lemma *balance-left-inv2-app*:

assumes $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt + 1 = \text{bheight } rt \ \text{color-of } rt = B$

shows $\text{inv2 } (\text{balance-left } lt \ k \ v \ rt)$

$\text{bheight } (\text{balance-left } lt \ k \ v \ rt) = \text{bheight } rt$

<proof>

lemma *balance-left-inv1*: $\llbracket \text{inv1 } a; \text{inv1 } b; \text{color-of } b = B \rrbracket \implies \text{inv1 } (\text{balance-left } a \ k \ x \ b)$

<proof>

lemma *balance-left-inv1l*: $\llbracket \text{inv1l } lt; \text{inv1 } rt \rrbracket \implies \text{inv1l } (\text{balance-left } lt \ k \ x \ rt)$

<proof>

lemma (in *linorder*) *balance-left-rbt-sorted*:

$\llbracket \text{rbt-sorted } l; \text{rbt-sorted } r; \text{rbt-less } k \ l; k \ll r \rrbracket \implies \text{rbt-sorted } (\text{balance-left } l \ k \ v \ r)$

<proof>

context *order* **begin**

lemma *balance-left-rbt-greater*:

fixes $k :: 'a$

assumes $k \ll a \ k \ll b \ k < x$

shows $k \ll \text{balance-left } a \ x \ t \ b$

<proof>

lemma *balance-left-rbt-less*:

fixes $k :: 'a$

assumes $a \ll k \ b \ll k \ x < k$

shows $\text{balance-left } a \ x \ t \ b \ll k$

<proof>

end

lemma *balance-left-in-tree*:

assumes $\text{inv1l } l \ \text{inv1 } r \ \text{bheight } l + 1 = \text{bheight } r$

shows $\text{entry-in-tree } k \ v \ (\text{balance-left } l \ a \ b \ r) = (\text{entry-in-tree } k \ v \ l \vee k = a \wedge v$

$= b \vee \text{entry-in-tree } k \ v \ r)$
 $\langle \text{proof} \rangle$

fun

$\text{balance-right} :: ('a, 'b) \text{ rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$

where

$\text{balance-right } a \ k \ x \ (\text{Branch } R \ b \ s \ y \ c) = \text{Branch } R \ a \ k \ x \ (\text{Branch } B \ b \ s \ y \ c) \mid$
 $\text{balance-right } (\text{Branch } B \ a \ k \ x \ b) \ s \ y \ bl = \text{balance } (\text{Branch } R \ a \ k \ x \ b) \ s \ y \ bl \mid$
 $\text{balance-right } (\text{Branch } R \ a \ k \ x \ (\text{Branch } B \ b \ s \ y \ c)) \ t \ z \ bl = \text{Branch } R \ (\text{balance}$
 $(\text{paint } R \ a) \ k \ x \ b) \ s \ y \ (\text{Branch } B \ c \ t \ z \ bl) \mid$
 $\text{balance-right } t \ k \ x \ s = \text{Empty}$

lemma *balance-right-inv2-with-inv1*:

assumes $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt = \text{bheight } rt + 1 \ \text{inv1 } lt$

shows $\text{inv2 } (\text{balance-right } lt \ k \ v \ rt) \wedge \text{bheight } (\text{balance-right } lt \ k \ v \ rt) = \text{bheight}$
 lt
 $\langle \text{proof} \rangle$

lemma *balance-right-inv1*: $\llbracket \text{inv1 } a; \text{inv1l } b; \text{color-of } a = B \rrbracket \Longrightarrow \text{inv1 } (\text{balance-right}$
 $a \ k \ x \ b)$
 $\langle \text{proof} \rangle$

lemma *balance-right-inv1l*: $\llbracket \text{inv1 } lt; \text{inv1l } rt \rrbracket \Longrightarrow \text{inv1l } (\text{balance-right } lt \ k \ x \ rt)$
 $\langle \text{proof} \rangle$

lemma (in *linorder*) *balance-right-rbt-sorted*:

$\llbracket \text{rbt-sorted } l; \text{rbt-sorted } r; \text{rbt-less } k \ l; k \ll r \rrbracket \Longrightarrow \text{rbt-sorted } (\text{balance-right } l \ k \ v$
 $r)$
 $\langle \text{proof} \rangle$

context *order* **begin**

lemma *balance-right-rbt-greater*:

fixes $k :: 'a$

assumes $k \ll a \ k \ll b \ k < x$

shows $k \ll \text{balance-right } a \ x \ t \ b$
 $\langle \text{proof} \rangle$

lemma *balance-right-rbt-less*:

fixes $k :: 'a$

assumes $a \ll k \ b \ll k \ x < k$

shows $\text{balance-right } a \ x \ t \ b \ll k$
 $\langle \text{proof} \rangle$

end

lemma *balance-right-in-tree*:

assumes $\text{inv1 } l \ \text{inv1l } r \ \text{bheight } l = \text{bheight } r + 1 \ \text{inv2 } l \ \text{inv2 } r$

shows $\text{entry-in-tree } x \ y \ (\text{balance-right } l \ k \ v \ r) = (\text{entry-in-tree } x \ y \ l \vee x = k \wedge$

$y = v \vee \text{entry-in-tree } x \ y \ r)$
 $\langle \text{proof} \rangle$

fun

$\text{combine} :: ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$

where

$\text{combine } \text{Empty } x = x$
 $| \text{combine } x \ \text{Empty} = x$
 $| \text{combine } (\text{Branch } R \ a \ k \ x \ b) \ (\text{Branch } R \ c \ s \ y \ d) = (\text{case } (\text{combine } b \ c) \ \text{of}$
 $\quad \text{Branch } R \ b2 \ t \ z \ c2 \Rightarrow (\text{Branch } R \ (\text{Branch } R \ a \ k \ x$
 $\quad b2) \ t \ z \ (\text{Branch } R \ c2 \ s \ y \ d)) \ |$
 $\quad bc \Rightarrow \text{Branch } R \ a \ k \ x \ (\text{Branch } R \ bc \ s \ y \ d))$
 $| \text{combine } (\text{Branch } B \ a \ k \ x \ b) \ (\text{Branch } B \ c \ s \ y \ d) = (\text{case } (\text{combine } b \ c) \ \text{of}$
 $\quad \text{Branch } R \ b2 \ t \ z \ c2 \Rightarrow \text{Branch } R \ (\text{Branch } B \ a \ k \ x \ b2)$
 $\quad t \ z \ (\text{Branch } B \ c2 \ s \ y \ d)) \ |$
 $\quad bc \Rightarrow \text{balance-left } a \ k \ x \ (\text{Branch } B \ bc \ s \ y \ d))$
 $| \text{combine } a \ (\text{Branch } R \ b \ k \ x \ c) = \text{Branch } R \ (\text{combine } a \ b) \ k \ x \ c$
 $| \text{combine } (\text{Branch } R \ a \ k \ x \ b) \ c = \text{Branch } R \ a \ k \ x \ (\text{combine } b \ c)$

lemma *combine-inv2*:

assumes $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt = \text{bheight } rt$

shows $\text{bheight } (\text{combine } lt \ rt) = \text{bheight } lt \ \text{inv2 } (\text{combine } lt \ rt)$

$\langle \text{proof} \rangle$

lemma *combine-inv1*:

assumes $\text{inv1 } lt \ \text{inv1 } rt$

shows $\text{color-of } lt = B \Longrightarrow \text{color-of } rt = B \Longrightarrow \text{inv1 } (\text{combine } lt \ rt)$

$\text{inv1l } (\text{combine } lt \ rt)$

$\langle \text{proof} \rangle$

context *linorder* **begin**

lemma *combine-rbt-greater[simp]*:

fixes $k :: 'a$

assumes $k \ll l \ k \ll r$

shows $k \ll \text{combine } l \ r$

$\langle \text{proof} \rangle$

lemma *combine-rbt-less[simp]*:

fixes $k :: 'a$

assumes $l \ll k \ r \ll k$

shows $\text{combine } l \ r \ll k$

$\langle \text{proof} \rangle$

lemma *combine-rbt-sorted*:

fixes $k :: 'a$

assumes $\text{rbt-sorted } l \ \text{rbt-sorted } r \ l \ll k \ k \ll r$

shows $\text{rbt-sorted } (\text{combine } l \ r)$

$\langle \text{proof} \rangle$

end

lemma *combine-in-tree*:

assumes $inv2\ l\ inv2\ r\ bheight\ l = bheight\ r\ inv1\ l\ inv1\ r$

shows $entry-in-tree\ k\ v\ (combine\ l\ r) = (entry-in-tree\ k\ v\ l \vee entry-in-tree\ k\ v\ r)$

<proof>

context *ord* **begin**

fun

$rbt-del-from-left :: 'a \Rightarrow ('a, 'b)\ rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$ **and**
 $rbt-del-from-right :: 'a \Rightarrow ('a, 'b)\ rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$ **and**
 $rbt-del :: 'a \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

where

$rbt-del\ x\ Empty = Empty \mid$

$rbt-del\ x\ (Branch\ c\ a\ y\ s\ b) =$

$(if\ x < y\ then\ rbt-del-from-left\ x\ a\ y\ s\ b$

$\ else\ (if\ x > y\ then\ rbt-del-from-right\ x\ a\ y\ s\ b\ else\ combine\ a\ b)) \mid$

$rbt-del-from-left\ x\ (Branch\ B\ lt\ z\ v\ rt)\ y\ s\ b = balance-left\ (rbt-del\ x\ (Branch\ B$
 $lt\ z\ v\ rt))\ y\ s\ b \mid$

$rbt-del-from-left\ x\ a\ y\ s\ b = Branch\ R\ (rbt-del\ x\ a)\ y\ s\ b \mid$

$rbt-del-from-right\ x\ a\ y\ s\ (Branch\ B\ lt\ z\ v\ rt) = balance-right\ a\ y\ s\ (rbt-del\ x$
 $(Branch\ B\ lt\ z\ v\ rt)) \mid$

$rbt-del-from-right\ x\ a\ y\ s\ b = Branch\ R\ a\ y\ s\ (rbt-del\ x\ b)$

end

context *linorder* **begin**

lemma

assumes $inv2\ lt\ inv1\ lt$

shows

$\llbracket inv2\ rt; bheight\ lt = bheight\ rt; inv1\ rt \rrbracket \implies$

$inv2\ (rbt-del-from-left\ x\ lt\ k\ v\ rt) \wedge$

$bheight\ (rbt-del-from-left\ x\ lt\ k\ v\ rt) = bheight\ lt \wedge$

$(color-of\ lt = B \wedge color-of\ rt = B \wedge inv1\ (rbt-del-from-left\ x\ lt\ k\ v\ rt) \vee$

$(color-of\ lt \neq B \vee color-of\ rt \neq B) \wedge inv1l\ (rbt-del-from-left\ x\ lt\ k\ v\ rt))$

and $\llbracket inv2\ rt; bheight\ lt = bheight\ rt; inv1\ rt \rrbracket \implies$

$inv2\ (rbt-del-from-right\ x\ lt\ k\ v\ rt) \wedge$

$bheight\ (rbt-del-from-right\ x\ lt\ k\ v\ rt) = bheight\ lt \wedge$

$(color-of\ lt = B \wedge color-of\ rt = B \wedge inv1\ (rbt-del-from-right\ x\ lt\ k\ v\ rt) \vee$

$(color-of\ lt \neq B \vee color-of\ rt \neq B) \wedge inv1l\ (rbt-del-from-right\ x\ lt\ k\ v\ rt))$

and $rbt-del-inv1-inv2: inv2\ (rbt-del\ x\ lt) \wedge (color-of\ lt = R \wedge bheight\ (rbt-del\ x$
 $lt) = bheight\ lt \wedge inv1\ (rbt-del\ x\ lt)$

$\vee color-of\ lt = B \wedge bheight\ (rbt-del\ x\ lt) = bheight\ lt - 1 \wedge inv1l\ (rbt-del\ x\ lt))$

<proof>

lemma

$\text{rbt-del-from-left-rbt-less: } \llbracket lt \mid \ll v; rt \mid \ll v; k < v \rrbracket \implies \text{rbt-del-from-left } x \text{ } lt \text{ } k \text{ } y \text{ } rt \mid \ll v$
and $\text{rbt-del-from-right-rbt-less: } \llbracket lt \mid \ll v; rt \mid \ll v; k < v \rrbracket \implies \text{rbt-del-from-right } x \text{ } lt \text{ } k \text{ } y \text{ } rt \mid \ll v$
and $\text{rbt-del-rbt-less: } lt \mid \ll v \implies \text{rbt-del } x \text{ } lt \mid \ll v$
 <proof>

lemma $\text{rbt-del-from-left-rbt-greater: } \llbracket v \ll \mid lt; v \ll \mid rt; k > v \rrbracket \implies v \ll \mid \text{rbt-del-from-left } x \text{ } lt \text{ } k \text{ } y \text{ } rt$
and $\text{rbt-del-from-right-rbt-greater: } \llbracket v \ll \mid lt; v \ll \mid rt; k > v \rrbracket \implies v \ll \mid \text{rbt-del-from-right } x \text{ } lt \text{ } k \text{ } y \text{ } rt$
and $\text{rbt-del-rbt-greater: } v \ll \mid lt \implies v \ll \mid \text{rbt-del } x \text{ } lt$
 <proof>

lemma $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll k; k \ll \mid rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-left } x \text{ } lt \text{ } k \text{ } y \text{ } rt)$
and $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll k; k \ll \mid rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-right } x \text{ } lt \text{ } k \text{ } y \text{ } rt)$
and $\text{rbt-del-rbt-sorted: } \text{rbt-sorted } lt \implies \text{rbt-sorted } (\text{rbt-del } x \text{ } lt)$
 <proof>

lemma $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll kt; kt \ll \mid rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; x < kt \rrbracket \implies \text{entry-in-tree } k \text{ } v (\text{rbt-del-from-left } x \text{ } lt \text{ } kt \text{ } y \text{ } rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v (\text{Branch } c \text{ } lt \text{ } kt \text{ } y \text{ } rt)))$
and $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll kt; kt \ll \mid rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; x > kt \rrbracket \implies \text{entry-in-tree } k \text{ } v (\text{rbt-del-from-right } x \text{ } lt \text{ } kt \text{ } y \text{ } rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v (\text{Branch } c \text{ } lt \text{ } kt \text{ } y \text{ } rt)))$
and $\text{rbt-del-in-tree: } \llbracket \text{rbt-sorted } t; \text{inv1 } t; \text{inv2 } t \rrbracket \implies \text{entry-in-tree } k \text{ } v (\text{rbt-del } x \text{ } t) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } t))$
 <proof>

definition (in ord) *rbt-delete* **where**
 $\text{rbt-delete } k \text{ } t = \text{paint } B (\text{rbt-del } k \text{ } t)$

theorem *rbt-delete-is-rbt* [simp]: **assumes** *is-rbt* *t* **shows** *is-rbt* (*rbt-delete* *k* *t*)
 <proof>

lemma *rbt-delete-in-tree*:
assumes *is-rbt* *t*
shows $\text{entry-in-tree } k \text{ } v (\text{rbt-delete } x \text{ } t) = (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } t)$
 <proof>

lemma *rbt-lookup-rbt-delete*:
assumes *is-rbt*: *is-rbt* *t*
shows $\text{rbt-lookup } (\text{rbt-delete } k \text{ } t) = (\text{rbt-lookup } t) \mid (-\{k\})$
 <proof>

end

129.5 Modifying existing entries

context *ord* **begin**

primrec

rbt-map-entry :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt

where

rbt-map-entry *k f Empty* = *Empty*

| *rbt-map-entry* *k f (Branch c lt x v rt)* =
 (if *k* < *x* then *Branch c (rbt-map-entry k f lt) x v rt*
 else if *k* > *x* then *Branch c lt x v (rbt-map-entry k f rt)*)
 else *Branch c lt x (f v) rt*)

lemma *rbt-map-entry-color-of*: *color-of (rbt-map-entry k f t) = color-of t* *<proof>*

lemma *rbt-map-entry-inv1*: *inv1 (rbt-map-entry k f t) = inv1 t* *<proof>*

lemma *rbt-map-entry-inv2*: *inv2 (rbt-map-entry k f t) = inv2 t bheight (rbt-map-entry k f t) = bheight t* *<proof>*

lemma *rbt-map-entry-rbt-greater*: *rbt-greater a (rbt-map-entry k f t) = rbt-greater a t* *<proof>*

lemma *rbt-map-entry-rbt-less*: *rbt-less a (rbt-map-entry k f t) = rbt-less a t* *<proof>*

lemma *rbt-map-entry-rbt-sorted*: *rbt-sorted (rbt-map-entry k f t) = rbt-sorted t* *<proof>*

theorem *rbt-map-entry-is-rbt* [*simp*]: *is-rbt (rbt-map-entry k f t) = is-rbt t* *<proof>*

end

theorem (**in** *linorder*) *rbt-lookup-rbt-map-entry*:

rbt-lookup (rbt-map-entry k f t) = (rbt-lookup t)(k := map-option f (rbt-lookup t k))
<proof>

129.6 Mapping all entries

primrec

map :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'c) rbt

where

map f Empty = *Empty*

| *map f (Branch c lt k v rt)* = *Branch c (map f lt) k (f k v) (map f rt)*

lemma *map-entries* [*simp*]: *entries (map f t) = List.map ($\lambda(k, v). (k, f k v)$) (entries t)*
<proof>

lemma *map-keys* [*simp*]: *keys (map f t) = keys t* *<proof>*

lemma *map-color-of*: *color-of (map f t) = color-of t* *<proof>*

lemma *map-inv1*: *inv1 (map f t) = inv1 t* *<proof>*

lemma *map-inv2*: *inv2 (map f t) = inv2 t bheight (map f t) = bheight t* *<proof>*

context *ord* **begin**

lemma *map-rbt-greater*: *rbt-greater* *k* (*map* *f* *t*) = *rbt-greater* *k* *t* *<proof>*

lemma *map-rbt-less*: *rbt-less* *k* (*map* *f* *t*) = *rbt-less* *k* *t* *<proof>*

lemma *map-rbt-sorted*: *rbt-sorted* (*map* *f* *t*) = *rbt-sorted* *t* *<proof>*

theorem *map-is-rbt* [*simp*]: *is-rbt* (*map* *f* *t*) = *is-rbt* *t*
<proof>

end

theorem (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup* (*map* *f* *t*) *x* = *map-option* (*f* *x*)
(*rbt-lookup* *t* *x*)
<proof>

hide-const (**open**) *map*

129.7 Folding over entries

definition *fold* :: (*'a* \Rightarrow *'b* \Rightarrow *'c* \Rightarrow *'c*) \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow *'c* \Rightarrow *'c* **where**
fold *f* *t* = *List.fold* (*case-prod* *f*) (*entries* *t*)

lemma *fold-simps* [*simp*]:
fold *f* *Empty* = *id*
fold *f* (*Branch* *c* *lt* *k* *v* *rt*) = *fold* *f* *rt* \circ *f* *k* *v* \circ *fold* *f* *lt*
<proof>

lemma *fold-code* [*code*]:
fold *f* *Empty* *x* = *x*
fold *f* (*Branch* *c* *lt* *k* *v* *rt*) *x* = *fold* *f* *rt* (*f* *k* *v* (*fold* *f* *lt* *x*))
<proof>

fun *foldi* :: (*'c* \Rightarrow *bool*) \Rightarrow (*'a* \Rightarrow *'b* \Rightarrow *'c* \Rightarrow *'c*) \Rightarrow (*'a* :: *linorder*, *'b*) *rbt* \Rightarrow *'c* \Rightarrow *'c*

where

foldi *c* *f* *Empty* *s* = *s* |
foldi *c* *f* (*Branch* *col* *l* *k* *v* *r*) *s* = (
 if (*c* *s*) *then*
 let *s'* = *foldi* *c* *f* *l* *s* *in*
 if (*c* *s'*) *then*
 foldi *c* *f* *r* (*f* *k* *v* *s'*)
 else *s'*
 else
 s
)

129.8 Bulkloading a tree

definition (**in** *ord*) *rbt-bulkload* :: (*'a* \times *'b*) *list* \Rightarrow (*'a*, *'b*) *rbt* **where**
rbt-bulkload *xs* = *foldr* ($\lambda(k, v). \text{rbt-insert } k \ v$) *xs* *Empty*

context *linorder* **begin**

lemma *rbt-bulkload-is-rbt* [*simp*, *intro*]:
 is-rbt (*rbt-bulkload xs*)
 ⟨*proof*⟩

lemma *rbt-lookup-rbt-bulkload*:
 rbt-lookup (*rbt-bulkload xs*) = *map-of xs*
 ⟨*proof*⟩

end

129.9 Building a RBT from a sorted list

These functions have been adapted from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *rbtreeify-f* :: *nat* \Rightarrow (*'a* \times *'b*) *list* \Rightarrow (*'a*, *'b*) *rbt* \times (*'a* \times *'b*) *list*
and *rbtreeify-g* :: *nat* \Rightarrow (*'a* \times *'b*) *list* \Rightarrow (*'a*, *'b*) *rbt* \times (*'a* \times *'b*) *list*
where

rbtreeify-f *n kvs* =
 (*if* *n* = 0 *then* (*Empty*, *kvs*)
 else if *n* = 1 *then*
 case kvs of (*k*, *v*) # *kvs'* \Rightarrow (*Branch R Empty k v Empty*, *kvs'*)
 else if (*n mod* 2 = 0) *then*
 case *rbtreeify-f* (*n div* 2) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow
 apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div* 2) *kvs'*)
 else case *rbtreeify-f* (*n div* 2) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow
 apfst (*Branch B t1 k v*) (*rbtreeify-f* (*n div* 2) *kvs'*))

| *rbtreeify-g* *n kvs* =
 (*if* *n* = 0 \vee *n* = 1 *then* (*Empty*, *kvs*)
 else if *n mod* 2 = 0 *then*
 case *rbtreeify-g* (*n div* 2) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow
 apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div* 2) *kvs'*)
 else case *rbtreeify-f* (*n div* 2) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow
 apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div* 2) *kvs'*))

definition *rbtreeify* :: (*'a* \times *'b*) *list* \Rightarrow (*'a*, *'b*) *rbt*
where *rbtreeify kvs* = *fst* (*rbtreeify-g* (*Suc* (*length kvs*)) *kvs*)

declare *rbtreeify-f.simps* [*simp del*] *rbtreeify-g.simps* [*simp del*]

lemma *rbtreeify-f-code* [*code*]:
 rbtreeify-f *n kvs* =
 (*if* *n* = 0 *then* (*Empty*, *kvs*)
 else if *n* = 1 *then*
 case kvs of (*k*, *v*) # *kvs'* \Rightarrow
 (*Branch R Empty k v Empty*, *kvs'*)
 else let (*n'*, *r*) = *Euclidean-Rings.divmod-nat n 2* *in*

if $r = 0$ then
 case $\text{rbtreeify-f } n' \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$
 $\text{apfst } (\text{Branch } B \ t1 \ k \ v) \ (\text{rbtreeify-g } n' \ \text{kvs}')$
 else case $\text{rbtreeify-f } n' \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$
 $\text{apfst } (\text{Branch } B \ t1 \ k \ v) \ (\text{rbtreeify-f } n' \ \text{kvs}')$
 $\langle \text{proof} \rangle$

lemma *rbtreeify-g-code* [code]:

$\text{rbtreeify-g } n \text{ kvs} =$
 (if $n = 0 \vee n = 1$ then $(\text{Empty}, \text{kvs})$
 else let $(n', r) = \text{Euclidean-Rings.divmod-nat } n \ 2$ in
 if $r = 0$ then
 case $\text{rbtreeify-g } n' \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$
 $\text{apfst } (\text{Branch } B \ t1 \ k \ v) \ (\text{rbtreeify-g } n' \ \text{kvs}')$
 else case $\text{rbtreeify-f } n' \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$
 $\text{apfst } (\text{Branch } B \ t1 \ k \ v) \ (\text{rbtreeify-g } n' \ \text{kvs}')$
 $\langle \text{proof} \rangle$

lemma *Suc-double-half*: $\text{Suc } (2 * n) \text{ div } 2 = n$

$\langle \text{proof} \rangle$

lemma *div2-plus-div2*: $n \text{ div } 2 + n \text{ div } 2 = (n :: \text{nat}) - n \text{ mod } 2$

$\langle \text{proof} \rangle$

lemma *rbtreeify-f-rec-aux-lemma*:

$\llbracket k - n \text{ div } 2 = \text{Suc } k'; n \leq k; n \text{ mod } 2 = \text{Suc } 0 \rrbracket$
 $\implies k' - n \text{ div } 2 = k - n$
 $\langle \text{proof} \rangle$

lemma *rbtreeify-f-simps*:

$\text{rbtreeify-f } 0 \text{ kvs} = (\text{Empty}, \text{kvs})$
 $\text{rbtreeify-f } (\text{Suc } 0) ((k, v) \# \text{kvs}) =$
 $(\text{Branch } R \ \text{Empty } k \ v \ \text{Empty}, \text{kvs})$
 $0 < n \implies \text{rbtreeify-f } (2 * n) \text{ kvs} =$
 $(\text{case } \text{rbtreeify-f } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$
 $\text{apfst } (\text{Branch } B \ t1 \ k \ v) \ (\text{rbtreeify-g } n \ \text{kvs}'))$
 $0 < n \implies \text{rbtreeify-f } (\text{Suc } (2 * n)) \text{ kvs} =$
 $(\text{case } \text{rbtreeify-f } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$
 $\text{apfst } (\text{Branch } B \ t1 \ k \ v) \ (\text{rbtreeify-f } n \ \text{kvs}'))$
 $\langle \text{proof} \rangle$

lemma *rbtreeify-g-simps*:

$\text{rbtreeify-g } 0 \text{ kvs} = (\text{Empty}, \text{kvs})$
 $\text{rbtreeify-g } (\text{Suc } 0) \text{ kvs} = (\text{Empty}, \text{kvs})$
 $0 < n \implies \text{rbtreeify-g } (2 * n) \text{ kvs} =$
 $(\text{case } \text{rbtreeify-g } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$
 $\text{apfst } (\text{Branch } B \ t1 \ k \ v) \ (\text{rbtreeify-g } n \ \text{kvs}'))$
 $0 < n \implies \text{rbtreeify-g } (\text{Suc } (2 * n)) \text{ kvs} =$
 $(\text{case } \text{rbtreeify-f } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \Rightarrow$

apfst (*Branch B t1 k v*) (*rbtreeify-g n kvs'*)
 ⟨*proof*⟩

declare *rbtreeify-f-simps*[*simp*] *rbtreeify-g-simps*[*simp*]

lemma *length-rbtreeify-f*: $n \leq \text{length } kvs$
 $\implies \text{length } (\text{snd } (\text{rbtreeify-f } n \ kvs)) = \text{length } kvs - n$
and *length-rbtreeify-g*: $\llbracket 0 < n; n \leq \text{Suc } (\text{length } kvs) \rrbracket$
 $\implies \text{length } (\text{snd } (\text{rbtreeify-g } n \ kvs)) = \text{Suc } (\text{length } kvs) - n$
 ⟨*proof*⟩

lemma *rbtreeify-induct* [*consumes 1*, *case-names f-0 f-1 f-even f-odd g-0 g-1 g-even g-odd*]:

fixes *P Q*

defines *f0* == $(\bigwedge kvs. P \ 0 \ kvs)$

and *f1* == $(\bigwedge k \ v \ kvs. P \ (\text{Suc } 0) \ ((k, v) \# \ kvs))$

and *feven* ==

$(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$
 $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$
 $\implies P \ (2 * n) \ kvs)$

and *fodd* ==

$(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$
 $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{length } kvs'; P \ n \ kvs' \rrbracket$
 $\implies P \ (\text{Suc } (2 * n)) \ kvs)$

and *g0* == $(\bigwedge kvs. Q \ 0 \ kvs)$

and *g1* == $(\bigwedge kvs. Q \ (\text{Suc } 0) \ kvs)$

and *geven* ==

$(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{Suc } (\text{length } kvs); Q \ n \ kvs;$
 $\text{rbtreeify-g } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$
 $\implies Q \ (2 * n) \ kvs)$

and *godd* ==

$(\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$
 $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$
 $\implies Q \ (\text{Suc } (2 * n)) \ kvs)$

shows $\llbracket n \leq \text{length } kvs;$

PROP f0; *PROP f1*; *PROP feven*; *PROP fodd*;
PROP g0; *PROP g1*; *PROP geven*; *PROP godd* \rrbracket
 $\implies P \ n \ kvs$

and $\llbracket n \leq \text{Suc } (\text{length } kvs);$

PROP f0; *PROP f1*; *PROP feven*; *PROP fodd*;
PROP g0; *PROP g1*; *PROP geven*; *PROP godd* \rrbracket
 $\implies Q \ n \ kvs$

⟨*proof*⟩

lemma *inv1-rbtreeify-f*: $n \leq \text{length } kvs$

$\implies \text{inv1 } (\text{fst } (\text{rbtreeify-f } n \ kvs))$

and *inv1-rbtreeify-g*: $n \leq \text{Suc } (\text{length } kvs)$

$\implies \text{inv1 } (\text{fst } (\text{rbtreeify-g } n \ kvs))$

⟨*proof*⟩

fun *plog2* :: *nat* \Rightarrow *nat*
where *plog2* *n* = (if *n* \leq 1 then 0 else *plog2* (*n* div 2) + 1)

declare *plog2.simps* [*simp del*]

lemma *plog2-simps* [*simp*]:
plog2 0 = 0 *plog2* (Suc 0) = 0
0 < *n* \implies *plog2* (2 * *n*) = 1 + *plog2* *n*
0 < *n* \implies *plog2* (Suc (2 * *n*)) = 1 + *plog2* *n*
⟨*proof*⟩

lemma *bheight-rbtreeify-f*: *n* \leq *length kvs*
 \implies *bheight* (fst (rbtreeify-f *n kvs*)) = *plog2* *n*
and *bheight-rbtreeify-g*: *n* \leq Suc (*length kvs*)
 \implies *bheight* (fst (rbtreeify-g *n kvs*)) = *plog2* *n*
⟨*proof*⟩

lemma *bheight-rbtreeify-f-eq-plog2I*:
 \llbracket rbtreeify-f *n kvs* = (*t*, *kvs'*); *n* \leq *length kvs* \rrbracket
 \implies *bheight* *t* = *plog2* *n*
⟨*proof*⟩

lemma *bheight-rbtreeify-g-eq-plog2I*:
 \llbracket rbtreeify-g *n kvs* = (*t*, *kvs'*); *n* \leq Suc (*length kvs*) \rrbracket
 \implies *bheight* *t* = *plog2* *n*
⟨*proof*⟩

hide-const (open) *plog2*

lemma *inv2-rbtreeify-f*: *n* \leq *length kvs*
 \implies *inv2* (fst (rbtreeify-f *n kvs*))
and *inv2-rbtreeify-g*: *n* \leq Suc (*length kvs*)
 \implies *inv2* (fst (rbtreeify-g *n kvs*))
⟨*proof*⟩

lemma *n* \leq *length kvs* \implies True
and *color-of-rbtreeify-g*:
 \llbracket *n* \leq Suc (*length kvs*); 0 < *n* \rrbracket
 \implies *color-of* (fst (rbtreeify-g *n kvs*)) = *B*
⟨*proof*⟩

lemma *entries-rbtreeify-f-append*:
n \leq *length kvs*
 \implies *entries* (fst (rbtreeify-f *n kvs*)) @ *snd* (rbtreeify-f *n kvs*) = *kvs*
and *entries-rbtreeify-g-append*:
n \leq Suc (*length kvs*)
 \implies *entries* (fst (rbtreeify-g *n kvs*)) @ *snd* (rbtreeify-g *n kvs*) = *kvs*
⟨*proof*⟩

lemma *length-entries-rbtreeify-f*:

$$n \leq \text{length } kvs \implies \text{length } (\text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs))) = n$$

and *length-entries-rbtreeify-g*:

$$n \leq \text{Suc } (\text{length } kvs) \implies \text{length } (\text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs))) = n - 1$$

<proof>

lemma *rbtreeify-f-conv-drop*:

$$n \leq \text{length } kvs \implies \text{snd } (\text{rbtreeify-f } n \text{ } kvs) = \text{drop } n \text{ } kvs$$

<proof>

lemma *rbtreeify-g-conv-drop*:

$$n \leq \text{Suc } (\text{length } kvs) \implies \text{snd } (\text{rbtreeify-g } n \text{ } kvs) = \text{drop } (n - 1) \text{ } kvs$$

<proof>

lemma *entries-rbtreeify-f [simp]*:

$$n \leq \text{length } kvs \implies \text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) = \text{take } n \text{ } kvs$$

<proof>

lemma *entries-rbtreeify-g [simp]*:

$$n \leq \text{Suc } (\text{length } kvs) \implies$$

$$\text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = \text{take } (n - 1) \text{ } kvs$$

<proof>

lemma *keys-rbtreeify-f [simp]*: $n \leq \text{length } kvs$

$$\implies \text{keys } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) = \text{take } n \text{ } (\text{map } \text{fst } kvs)$$

<proof>

lemma *keys-rbtreeify-g [simp]*: $n \leq \text{Suc } (\text{length } kvs)$

$$\implies \text{keys } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = \text{take } (n - 1) \text{ } (\text{map } \text{fst } kvs)$$

<proof>

lemma *rbtreeify-fD*:

$$\llbracket \text{rbtreeify-f } n \text{ } kvs = (t, kvs'); n \leq \text{length } kvs \rrbracket$$

$$\implies \text{entries } t = \text{take } n \text{ } kvs \wedge kvs' = \text{drop } n \text{ } kvs$$

<proof>

lemma *rbtreeify-gD*:

$$\llbracket \text{rbtreeify-g } n \text{ } kvs = (t, kvs'); n \leq \text{Suc } (\text{length } kvs) \rrbracket$$

$$\implies \text{entries } t = \text{take } (n - 1) \text{ } kvs \wedge kvs' = \text{drop } (n - 1) \text{ } kvs$$

<proof>

lemma *entries-rbtreeify [simp]*: $\text{entries } (\text{rbtreeify } kvs) = kvs$

<proof>

context *linorder* **begin**

lemma *rbt-sorted-rbtreeify-f*:

$$\llbracket n \leq \text{length } kvs; \text{sorted } (\text{map } \text{fst } kvs); \text{distinct } (\text{map } \text{fst } kvs) \rrbracket$$

```

    ⇒ rbt-sorted (fst (rbtreeify-f n kvs))
  and rbt-sorted-rbtreeify-g:
    [ n ≤ Suc (length kvs); sorted (map fst kvs); distinct (map fst kvs) ]
    ⇒ rbt-sorted (fst (rbtreeify-g n kvs))
  ⟨proof⟩

```

lemma *rbt-sorted-rbtreeify*:

```

  [ sorted (map fst kvs); distinct (map fst kvs) ] ⇒ rbt-sorted (rbtreeify kvs)
  ⟨proof⟩

```

lemma *is-rbt-rbtreeify*:

```

  [ sorted (map fst kvs); distinct (map fst kvs) ]
  ⇒ is-rbt (rbtreeify kvs)
  ⟨proof⟩

```

lemma *rbt-lookup-rbtreeify*:

```

  [ sorted (map fst kvs); distinct (map fst kvs) ] ⇒
    rbt-lookup (rbtreeify kvs) = map-of kvs
  ⟨proof⟩

```

end

Functions to compare the height of two rbt trees, taken from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

```

fun skip-red :: ('a, 'b) rbt ⇒ ('a, 'b) rbt
where
  skip-red (Branch color.R l k v r) = l
  | skip-red t = t

```

definition *skip-black* :: ('a, 'b) rbt ⇒ ('a, 'b) rbt

where

```

  skip-black t = (let t' = skip-red t in case t' of Branch color.B l k v r ⇒ l | - ⇒ t')

```

datatype *compare* = *LT* | *GT* | *EQ*

partial-function (*tailrec*) *compare-height* :: ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ *compare*

where

```

  compare-height sx s t tx =
    (case (skip-red sx, skip-red s, skip-red t, skip-red tx) of
      (Branch - sx' - - -, Branch - s' - - -, Branch - t' - - -, Branch - tx' - - -) ⇒
        compare-height (skip-black sx') s' t' (skip-black tx')
    | (-, rbt.Empty, -, Branch - - - -) ⇒ LT
    | (Branch - - - - -, -, rbt.Empty, -) ⇒ GT
    | (Branch - sx' - - -, Branch - s' - - -, Branch - t' - - -, rbt.Empty) ⇒
        compare-height (skip-black sx') s' t' rbt.Empty
    | (rbt.Empty, Branch - s' - - -, Branch - t' - - -, Branch - tx' - - -) ⇒
        compare-height rbt.Empty s' t' (skip-black tx')

```

| - $\Rightarrow EQ$)

declare *compare-height.simps* [code]

hide-type (open) *compare*

hide-const (open)

compare-height skip-black skip-red LT GT EQ case-compare rec-compare

Abs-compare Rep-compare

hide-fact (open)

Abs-compare-cases Abs-compare-induct Abs-compare-inject Abs-compare-inverse

Rep-compare Rep-compare-cases Rep-compare-induct Rep-compare-inject Rep-compare-inverse

compare.simps compare.exhaust compare.induct compare.rec compare.simps

compare.size compare.case-cong compare.case-cong-weak compare.case

compare.nchotomy compare.split compare.split-asm compare.eq.refl compare.eq.simps

equal-compare-def

skip-red.simps skip-red.cases skip-red.induct

skip-black-def

compare-height.simps

129.10 union and intersection of sorted associative lists

context *ord* **begin**

function *sunion-with* :: ($'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$) $\Rightarrow ('a \times 'b)$ list $\Rightarrow ('a \times 'b)$ list $\Rightarrow ('a \times 'b)$ list

where

sunion-with *f* ((*k*, *v*) # *as*) ((*k'*, *v'*) # *bs*) =

(if *k* > *k'* then (*k'*, *v'*) # *sunion-with* *f* ((*k*, *v*) # *as*) *bs*

else if *k* < *k'* then (*k*, *v*) # *sunion-with* *f* *as* ((*k'*, *v'*) # *bs*)

else (*k*, *f k v v'*) # *sunion-with* *f* *as* *bs*)

| *sunion-with* *f* [] *bs* = *bs*

| *sunion-with* *f* *as* [] = *as*

⟨proof⟩

termination ⟨proof⟩

function *sinter-with* :: ($'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$) $\Rightarrow ('a \times 'b)$ list $\Rightarrow ('a \times 'b)$ list $\Rightarrow ('a \times 'b)$ list

where

sinter-with *f* ((*k*, *v*) # *as*) ((*k'*, *v'*) # *bs*) =

(if *k* > *k'* then *sinter-with* *f* ((*k*, *v*) # *as*) *bs*

else if *k* < *k'* then *sinter-with* *f* *as* ((*k'*, *v'*) # *bs*)

else (*k*, *f k v v'*) # *sinter-with* *f* *as* *bs*)

| *sinter-with* *f* [] - = []

| *sinter-with* *f* - [] = []

⟨proof⟩

termination ⟨proof⟩

end

declare *ord.sunion-with.simps* [code] *ord.sinter-with.simps*[code]

context *linorder* **begin**

lemma *set-fst-sunion-with*:

$\text{set } (\text{map fst } (\text{sunion-with } f \text{ } xs \text{ } ys)) = \text{set } (\text{map fst } xs) \cup \text{set } (\text{map fst } ys)$
 $\langle \text{proof} \rangle$

lemma *sorted-sunion-with* [simp]:

$\llbracket \text{sorted } (\text{map fst } xs); \text{sorted } (\text{map fst } ys) \rrbracket$
 $\implies \text{sorted } (\text{map fst } (\text{sunion-with } f \text{ } xs \text{ } ys))$
 $\langle \text{proof} \rangle$

lemma *distinct-sunion-with* [simp]:

$\llbracket \text{distinct } (\text{map fst } xs); \text{distinct } (\text{map fst } ys); \text{sorted } (\text{map fst } xs); \text{sorted } (\text{map fst } ys) \rrbracket$
 $\implies \text{distinct } (\text{map fst } (\text{sunion-with } f \text{ } xs \text{ } ys))$
 $\langle \text{proof} \rangle$

lemma *map-of-sunion-with*:

$\llbracket \text{sorted } (\text{map fst } xs); \text{sorted } (\text{map fst } ys) \rrbracket$
 $\implies \text{map-of } (\text{sunion-with } f \text{ } xs \text{ } ys) \text{ } k =$
 $(\text{case map-of } xs \text{ } k \text{ of None } \Rightarrow \text{map-of } ys \text{ } k$
 $| \text{Some } v \Rightarrow \text{case map-of } ys \text{ } k \text{ of None } \Rightarrow \text{Some } v$
 $| \text{Some } w \Rightarrow \text{Some } (f \text{ } k \text{ } v \text{ } w))$
 $\langle \text{proof} \rangle$

lemma *set-fst-sinter-with* [simp]:

$\llbracket \text{sorted } (\text{map fst } xs); \text{sorted } (\text{map fst } ys) \rrbracket$
 $\implies \text{set } (\text{map fst } (\text{sinter-with } f \text{ } xs \text{ } ys)) = \text{set } (\text{map fst } xs) \cap \text{set } (\text{map fst } ys)$
 $\langle \text{proof} \rangle$

lemma *set-fst-sinter-with-subset1*:

$\text{set } (\text{map fst } (\text{sinter-with } f \text{ } xs \text{ } ys)) \subseteq \text{set } (\text{map fst } xs)$
 $\langle \text{proof} \rangle$

lemma *set-fst-sinter-with-subset2*:

$\text{set } (\text{map fst } (\text{sinter-with } f \text{ } xs \text{ } ys)) \subseteq \text{set } (\text{map fst } ys)$
 $\langle \text{proof} \rangle$

lemma *sorted-sinter-with* [simp]:

$\llbracket \text{sorted } (\text{map fst } xs); \text{sorted } (\text{map fst } ys) \rrbracket$
 $\implies \text{sorted } (\text{map fst } (\text{sinter-with } f \text{ } xs \text{ } ys))$
 $\langle \text{proof} \rangle$

lemma *distinct-sinter-with* [simp]:

$\llbracket \text{distinct } (\text{map fst } xs); \text{distinct } (\text{map fst } ys) \rrbracket$
 $\implies \text{distinct } (\text{map fst } (\text{sinter-with } f \text{ } xs \text{ } ys))$
 $\langle \text{proof} \rangle$

lemma *map-of-sinter-with*:

[[sorted (map fst xs); sorted (map fst ys)]]
 \impl map-of (sinter-with f xs ys) k =
 (case map-of xs k of None \impl None | Some v \impl map-option (f k v) (map-of ys k))
 <proof>

end

lemma *distinct-map-of-rev*: distinct (map fst xs) \impl map-of (rev xs) = map-of xs
 <proof>

lemma *map-map-filter*:

map f (List.map-filter g xs) = List.map-filter (map-option f \circ g) xs
 <proof>

lemma *map-filter-map-option-const*:

List.map-filter ($\lambda x. \text{map-option } (\lambda y. f x) (g (f x))$) xs = filter ($\lambda x. g x \neq \text{None}$)
 (map f xs)
 <proof>

lemma *set-map-filter*: set (List.map-filter P xs) = the ‘ (P ‘ set xs - {None})
 <proof>

definition *is-rbt-empty* :: ('a, 'b) rbt \Rightarrow bool **where**

is-rbt-empty t \longleftrightarrow (case t of RBT-Impl.Empty \Rightarrow True | - \Rightarrow False)

lemma *is-rbt-empty-prop[simp]*: is-rbt-empty t \longleftrightarrow t = RBT-Impl.Empty

<proof>

definition *small-rbt* :: ('a, 'b) rbt \Rightarrow bool **where**

small-rbt t \longleftrightarrow bheight t < 4

definition *flip-rbt* :: ('a, 'b) rbt \Rightarrow ('a, 'b) rbt \Rightarrow bool **where**

flip-rbt t1 t2 \longleftrightarrow bheight t2 < bheight t1

abbreviation (*input*) *MR* **where** MR l a b r \equiv Branch RBT-Impl.R l a b r

abbreviation (*input*) *MB* **where** MB l a b r \equiv Branch RBT-Impl.B l a b r

fun *rbt-baliL* :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**

rbt-baliL (MR (MR t1 a b t2) a' b' t3) a'' b'' t4 = MR (MB t1 a b t2) a' b' (MB t3 a'' b'' t4)
 | rbt-baliL (MR t1 a b (MR t2 a' b' t3)) a'' b'' t4 = MR (MB t1 a b t2) a' b' (MB t3 a'' b'' t4)
 | rbt-baliL t1 a b t2 = MB t1 a b t2

fun *rbt-baliR* :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**
rbt-baliR t1 a b (MR t2 a' b' (MR t3 a'' b'' t4)) = MR (MB t1 a b t2) a' b' (MB t3 a'' b'' t4)
| *rbt-baliR* t1 a b (MR (MR t2 a' b' t3) a'' b'' t4) = MR (MB t1 a b t2) a' b' (MB t3 a'' b'' t4)
| *rbt-baliR* t1 a b t2 = MB t1 a b t2

fun *rbt-baldL* :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**
rbt-baldL (MR t1 a b t2) a' b' t3 = MR (MB t1 a b t2) a' b' t3
| *rbt-baldL* t1 a b (MB t2 a' b' t3) = *rbt-baliR* t1 a b (MR t2 a' b' t3)
| *rbt-baldL* t1 a b (MR (MB t2 a' b' t3) a'' b'' t4) =
MR (MB t1 a b t2) a' b' (*rbt-baliR* t3 a'' b'' (paint RBT-Impl.R t4))
| *rbt-baldL* t1 a b t2 = MR t1 a b t2

fun *rbt-baldR* :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**
rbt-baldR t1 a b (MR t2 a' b' t3) = MR t1 a b (MB t2 a' b' t3)
| *rbt-baldR* (MB t1 a b t2) a' b' t3 = *rbt-baliL* (MR t1 a b t2) a' b' t3
| *rbt-baldR* (MR t1 a b (MB t2 a' b' t3)) a'' b'' t4 =
MR (*rbt-baliL* (paint RBT-Impl.R t1) a b t2) a' b' (MB t3 a'' b'' t4)
| *rbt-baldR* t1 a b t2 = MR t1 a b t2

fun *rbt-app* :: ('a, 'b) rbt \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**
rbt-app RBT-Impl.Empty t = t
| *rbt-app* t RBT-Impl.Empty = t
| *rbt-app* (MR t1 a b t2) (MR t3 a'' b'' t4) = (case *rbt-app* t2 t3 of
MR u2 a' b' u3 \Rightarrow (MR (MR t1 a b u2) a' b' (MR u3 a'' b'' t4))
| t23 \Rightarrow MR t1 a b (MR t23 a'' b'' t4))
| *rbt-app* (MB t1 a b t2) (MB t3 a'' b'' t4) = (case *rbt-app* t2 t3 of
MR u2 a' b' u3 \Rightarrow MR (MB t1 a b u2) a' b' (MB u3 a'' b'' t4)
| t23 \Rightarrow *rbt-baldL* t1 a b (MB t23 a'' b'' t4))
| *rbt-app* t1 (MR t2 a b t3) = MR (*rbt-app* t1 t2) a b t3
| *rbt-app* (MR t1 a b t2) t3 = MR t1 a b (*rbt-app* t2 t3)

fun *rbt-joinL* :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**
rbt-joinL l a b r = (if bheight l \geq bheight r then MR l a b r
else case r of MB l' a' b' r' \Rightarrow *rbt-baliL* (*rbt-joinL* l a b l') a' b' r'
| MR l' a' b' r' \Rightarrow MR (*rbt-joinL* l a b l') a' b' r')

declare *rbt-joinL.simps*[simp del]

fun *rbt-joinR* :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**
rbt-joinR l a b r = (if bheight l \leq bheight r then MR l a b r
else case l of MB l' a' b' r' \Rightarrow *rbt-baliR* l' a' b' (*rbt-joinR* r' a b r)
| MR l' a' b' r' \Rightarrow MR l' a' b' (*rbt-joinR* r' a b r))

declare *rbt-joinR.simps*[simp del]

definition *rbt-join* :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **where**
rbt-join l a b r =

$(\text{let } bhl = \text{bheight } l; bhr = \text{bheight } r$
 $\text{in if } bhl > bhr$
 $\text{then paint RBT-Impl.B (rbt-joinR } l \ a \ b \ r)$
 $\text{else if } bhl < bhr$
 $\text{then paint RBT-Impl.B (rbt-joinL } l \ a \ b \ r)$
 $\text{else MB } l \ a \ b \ r)$

lemma *size-paint[simp]*: $\text{size (paint } c \ t) = \text{size } t$
 $\langle \text{proof} \rangle$

lemma *size-baliL[simp]*: $\text{size (rbt-baliL } t1 \ a \ b \ t2) = \text{Suc (size } t1 + \text{size } t2)$
 $\langle \text{proof} \rangle$

lemma *size-baliR[simp]*: $\text{size (rbt-baliR } t1 \ a \ b \ t2) = \text{Suc (size } t1 + \text{size } t2)$
 $\langle \text{proof} \rangle$

lemma *size-baldL[simp]*: $\text{size (rbt-baldL } t1 \ a \ b \ t2) = \text{Suc (size } t1 + \text{size } t2)$
 $\langle \text{proof} \rangle$

lemma *size-baldR[simp]*: $\text{size (rbt-baldR } t1 \ a \ b \ t2) = \text{Suc (size } t1 + \text{size } t2)$
 $\langle \text{proof} \rangle$

lemma *size-rbt-app[simp]*: $\text{size (rbt-app } t1 \ t2) = \text{size } t1 + \text{size } t2$
 $\langle \text{proof} \rangle$

lemma *size-rbt-joinL[simp]*: $\text{size (rbt-joinL } t1 \ a \ b \ t2) = \text{Suc (size } t1 + \text{size } t2)$
 $\langle \text{proof} \rangle$

lemma *size-rbt-joinR[simp]*: $\text{size (rbt-joinR } t1 \ a \ b \ t2) = \text{Suc (size } t1 + \text{size } t2)$
 $\langle \text{proof} \rangle$

lemma *size-rbt-join[simp]*: $\text{size (rbt-join } t1 \ a \ b \ t2) = \text{Suc (size } t1 + \text{size } t2)$
 $\langle \text{proof} \rangle$

definition *inv-12* $t \longleftrightarrow \text{inv1 } t \wedge \text{inv2 } t$

lemma *rbt-Node*: $\text{inv-12 (RBT-Impl.Branch } c \ l \ a \ b \ r) \Longrightarrow \text{inv-12 } l \wedge \text{inv-12 } r$
 $\langle \text{proof} \rangle$

lemma *paint2*: $\text{paint } c2 \ (\text{paint } c1 \ t) = \text{paint } c2 \ t$
 $\langle \text{proof} \rangle$

lemma *inv1-rbt-baliL*: $\text{inv1 } l \Longrightarrow \text{inv1 } r \Longrightarrow \text{inv1 (rbt-baliL } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *inv1-rbt-baliR*: $\text{inv1 } l \Longrightarrow \text{inv1 } r \Longrightarrow \text{inv1 (rbt-baliR } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *rbt-bheight-rbt-baliL*: $\text{bheight } l = \text{bheight } r \Longrightarrow \text{bheight (rbt-baliL } l \ a \ b \ r)$

$$= \text{Suc } (\text{bheight } l) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{rbt-bheight-rbt-baliR}: \text{bheight } l = \text{bheight } r \implies \text{bheight } (\text{rbt-baliR } l \ a \ b \ r) \\ = \text{Suc } (\text{bheight } l) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv2-rbt-baliL}: \text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r \implies \text{inv2} \\ (\text{rbt-baliL } l \ a \ b \ r) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv2-rbt-baliR}: \text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r \implies \text{inv2} \\ (\text{rbt-baliR } l \ a \ b \ r) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv-rbt-baliR}: \text{inv2 } l \implies \text{inv2 } r \implies \text{inv1 } l \implies \text{inv1 } r \implies \text{bheight } l = \\ \text{bheight } r \implies \\ \text{inv1 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{inv2 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baliR } l \ a \ b \ r) = \\ \text{Suc } (\text{bheight } l) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv-rbt-baliL}: \text{inv2 } l \implies \text{inv2 } r \implies \text{inv1 } l \implies \text{inv1 } r \implies \text{bheight } l = \\ \text{bheight } r \implies \\ \text{inv1 } (\text{rbt-baliL } l \ a \ b \ r) \wedge \text{inv2 } (\text{rbt-baliL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baliL } l \ a \ b \ r) = \\ \text{Suc } (\text{bheight } l) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv2-rbt-baldL-inv1}: \text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l + 1 = \text{bheight } r \implies \\ \text{inv1 } r \implies \\ \text{inv2 } (\text{rbt-baldL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldL } l \ a \ b \ r) = \text{bheight } r \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv2-rbt-baldL-B}: \text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l + 1 = \text{bheight } r \implies \\ \text{color-of } r = \text{RBT-Impl.B} \implies \\ \text{inv2 } (\text{rbt-baldL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldL } l \ a \ b \ r) = \text{bheight } r \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv1-rbt-baldL}: \text{inv1 } l \implies \text{inv1 } r \implies \text{color-of } r = \text{RBT-Impl.B} \implies \text{inv1} \\ (\text{rbt-baldL } l \ a \ b \ r) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv1lI}: \text{inv1 } t \implies \text{inv1l } t \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{neq-Black[simp]}: (c \neq \text{RBT-Impl.B}) = (c = \text{RBT-Impl.R}) \\ \langle \text{proof} \rangle$$

$$\textbf{lemma } \textit{inv1l-rbt-baldL}: \text{inv1l } l \implies \text{inv1 } r \implies \text{inv1l } (\text{rbt-baldL } l \ a \ b \ r) \\ \langle \text{proof} \rangle$$

lemma *inv2-rbt-baldR-inv1*: $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r + 1 \implies inv1\ l \implies$
 $inv2\ (rbt-baldR\ l\ a\ b\ r) \wedge bheight\ (rbt-baldR\ l\ a\ b\ r) = bheight\ l$
 ⟨proof⟩

lemma *inv1-rbt-baldR*: $inv1\ l \implies inv1l\ r \implies color-of\ l = RBT-Impl.B \implies inv1$
 $(rbt-baldR\ l\ a\ b\ r)$
 ⟨proof⟩

lemma *inv1l-rbt-baldR*: $inv1\ l \implies inv1l\ r \implies inv1l\ (rbt-baldR\ l\ a\ b\ r)$
 ⟨proof⟩

lemma *inv2-rbt-app*: $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r \implies$
 $inv2\ (rbt-app\ l\ r) \wedge bheight\ (rbt-app\ l\ r) = bheight\ l$
 ⟨proof⟩

lemma *inv1-rbt-app*: $inv1\ l \implies inv1\ r \implies (color-of\ l = RBT-Impl.B \wedge$
 $color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-app\ l\ r)) \wedge inv1l\ (rbt-app\ l\ r)$
 ⟨proof⟩

lemma *inv-rbt-baldL*: $inv2\ l \implies inv2\ r \implies bheight\ l + 1 = bheight\ r \implies inv1l\ l$
 $\implies inv1\ r \implies$
 $inv2\ (rbt-baldL\ l\ a\ b\ r) \wedge bheight\ (rbt-baldL\ l\ a\ b\ r) = bheight\ r \wedge$
 $inv1l\ (rbt-baldL\ l\ a\ b\ r) \wedge (color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-baldL\ l\ a\ b$
 $r))$
 ⟨proof⟩

lemma *inv-rbt-baldR*: $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r + 1 \implies inv1\ l$
 $\implies inv1l\ r \implies$
 $inv2\ (rbt-baldR\ l\ a\ b\ r) \wedge bheight\ (rbt-baldR\ l\ a\ b\ r) = bheight\ l \wedge$
 $inv1l\ (rbt-baldR\ l\ a\ b\ r) \wedge (color-of\ l = RBT-Impl.B \longrightarrow inv1\ (rbt-baldR\ l\ a\ b$
 $r))$
 ⟨proof⟩

lemma *inv-rbt-app*: $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r \implies inv1\ l \implies$
 $inv1\ r \implies$
 $inv2\ (rbt-app\ l\ r) \wedge bheight\ (rbt-app\ l\ r) = bheight\ l \wedge$
 $inv1l\ (rbt-app\ l\ r) \wedge (color-of\ l = RBT-Impl.B \wedge color-of\ r = RBT-Impl.B \longrightarrow$
 $inv1\ (rbt-app\ l\ r))$
 ⟨proof⟩

lemma *inv1l-rbt-joinL*: $inv1\ l \implies inv1\ r \implies bheight\ l \leq bheight\ r \implies$
 $inv1l\ (rbt-joinL\ l\ a\ b\ r) \wedge$
 $(bheight\ l \neq bheight\ r \wedge color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-joinL\ l\ a\ b\ r))$
 ⟨proof⟩

lemma *inv1l-rbt-joinR*: $inv1\ l \implies inv2\ l \implies inv1\ r \implies inv2\ r \implies bheight\ l \geq$
 $bheight\ r \implies$

$inv1l \ (rbt-joinR \ l \ a \ b \ r) \wedge$
 $(bheight \ l \neq bheight \ r \wedge color-of \ l = RBT-Impl.B \longrightarrow inv1 \ (rbt-joinR \ l \ a \ b \ r))$
 $\langle proof \rangle$

lemma $bheight-rbt-joinL$: $inv2 \ l \Longrightarrow inv2 \ r \Longrightarrow bheight \ l \leq bheight \ r \Longrightarrow$
 $bheight \ (rbt-joinL \ l \ a \ b \ r) = bheight \ r$
 $\langle proof \rangle$

lemma $inv2-rbt-joinL$: $inv2 \ l \Longrightarrow inv2 \ r \Longrightarrow bheight \ l \leq bheight \ r \Longrightarrow inv2$
 $(rbt-joinL \ l \ a \ b \ r)$
 $\langle proof \rangle$

lemma $bheight-rbt-joinR$: $inv2 \ l \Longrightarrow inv2 \ r \Longrightarrow bheight \ l \geq bheight \ r \Longrightarrow$
 $bheight \ (rbt-joinR \ l \ a \ b \ r) = bheight \ l$
 $\langle proof \rangle$

lemma $inv2-rbt-joinR$: $inv2 \ l \Longrightarrow inv2 \ r \Longrightarrow bheight \ l \geq bheight \ r \Longrightarrow inv2$
 $(rbt-joinR \ l \ a \ b \ r)$
 $\langle proof \rangle$

lemma $keys-paint[simp]$: $RBT-Impl.keys \ (paint \ c \ t) = RBT-Impl.keys \ t$
 $\langle proof \rangle$

lemma $keys-rbt-baliL$: $RBT-Impl.keys \ (rbt-baliL \ l \ a \ b \ r) = RBT-Impl.keys \ l \ @ \ a$
 $\# \ RBT-Impl.keys \ r$
 $\langle proof \rangle$

lemma $keys-rbt-baliR$: $RBT-Impl.keys \ (rbt-baliR \ l \ a \ b \ r) = RBT-Impl.keys \ l \ @ \ a$
 $\# \ RBT-Impl.keys \ r$
 $\langle proof \rangle$

lemma $keys-rbt-baldL$: $RBT-Impl.keys \ (rbt-baldL \ l \ a \ b \ r) = RBT-Impl.keys \ l \ @ \ a$
 $\# \ RBT-Impl.keys \ r$
 $\langle proof \rangle$

lemma $keys-rbt-baldR$: $RBT-Impl.keys \ (rbt-baldR \ l \ a \ b \ r) = RBT-Impl.keys \ l \ @ \ a$
 $\# \ RBT-Impl.keys \ r$
 $\langle proof \rangle$

lemma $keys-rbt-app$: $RBT-Impl.keys \ (rbt-app \ l \ r) = RBT-Impl.keys \ l \ @ \ RBT-Impl.keys \ r$
 $\langle proof \rangle$

lemma $keys-rbt-joinL$: $bheight \ l \leq bheight \ r \Longrightarrow$
 $RBT-Impl.keys \ (rbt-joinL \ l \ a \ b \ r) = RBT-Impl.keys \ l \ @ \ a \ \# \ RBT-Impl.keys \ r$
 $\langle proof \rangle$

lemma $keys-rbt-joinR$: $RBT-Impl.keys \ (rbt-joinR \ l \ a \ b \ r) = RBT-Impl.keys \ l \ @ \ a$
 $\# \ RBT-Impl.keys \ r$

<proof>

lemma *rbt-set-rbt-baliL*: $\text{set } (RBT-Impl.keys \ (rbt-baliL \ l \ a \ b \ r)) =$
 $\text{set } (RBT-Impl.keys \ l) \cup \{a\} \cup \text{set } (RBT-Impl.keys \ r)$
<proof>

lemma *set-rbt-joinL*: $\text{set } (RBT-Impl.keys \ (rbt-joinL \ l \ a \ b \ r)) =$
 $\text{set } (RBT-Impl.keys \ l) \cup \{a\} \cup \text{set } (RBT-Impl.keys \ r)$
<proof>

lemma *rbt-set-rbt-baliR*: $\text{set } (RBT-Impl.keys \ (rbt-baliR \ l \ a \ b \ r)) =$
 $\text{set } (RBT-Impl.keys \ l) \cup \{a\} \cup \text{set } (RBT-Impl.keys \ r)$
<proof>

lemma *set-rbt-joinR*: $\text{set } (RBT-Impl.keys \ (rbt-joinR \ l \ a \ b \ r)) =$
 $\text{set } (RBT-Impl.keys \ l) \cup \{a\} \cup \text{set } (RBT-Impl.keys \ r)$
<proof>

lemma *set-keys-paint*: $\text{set } (RBT-Impl.keys \ (\text{paint } c \ t)) = \text{set } (RBT-Impl.keys \ t)$
<proof>

lemma *set-rbt-join*: $\text{set } (RBT-Impl.keys \ (rbt-join \ l \ a \ b \ r)) =$
 $\text{set } (RBT-Impl.keys \ l) \cup \{a\} \cup \text{set } (RBT-Impl.keys \ r)$
<proof>

lemma *inv-rbt-join*: $\text{inv-12 } l \implies \text{inv-12 } r \implies \text{inv-12 } (rbt-join \ l \ a \ b \ r)$
<proof>

fun *rbt-recolor* :: $('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **where**
 $\text{rbt-recolor } (Branch \ RBT-Impl.R \ t1 \ k \ v \ t2) =$
 $(\text{if } \text{color-of } t1 = RBT-Impl.B \wedge \text{color-of } t2 = RBT-Impl.B \text{ then } Branch$
 $RBT-Impl.B \ t1 \ k \ v \ t2$
 $\text{else } Branch \ RBT-Impl.R \ t1 \ k \ v \ t2)$
 $| \text{ rbt-recolor } t = t$

lemma *rbt-recolor*: $\text{inv-12 } t \implies \text{inv-12 } (rbt-recolor \ t)$
<proof>

fun *rbt-split-min* :: $('a, 'b) \text{ rbt} \Rightarrow 'a \times 'b \times ('a, 'b) \text{ rbt}$ **where**
 $\text{rbt-split-min } RBT-Impl.Empty = \text{undefined}$
 $| \text{ rbt-split-min } (RBT-Impl.Branch \ - \ l \ a \ b \ r) =$
 $(\text{if } \text{is-rbt-empty } l \text{ then } (a, b, r) \text{ else let } (a', b', l') = \text{rbt-split-min } l \text{ in } (a', b', \text{rbt-join}$
 $l' \ a \ b \ r))$

lemma *rbt-split-min-set*:
 $\text{rbt-split-min } t = (a, b, t') \implies t \neq RBT-Impl.Empty \implies$
 $a \in \text{set } (RBT-Impl.keys \ t) \wedge \text{set } (RBT-Impl.keys \ t) = \{a\} \cup \text{set } (RBT-Impl.keys \ t')$
<proof>

lemma *rbt-split-min-inv*: $\text{rbt-split-min } t = (a, b, t') \implies \text{inv-12 } t \implies t \neq \text{RBT-Impl.Empty} \implies \text{inv-12 } t'$
 ⟨proof⟩

definition *rbt-join2* :: $('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**
 $\text{rbt-join2 } l \ r = (\text{if is-rbt-empty } r \text{ then } l \text{ else let } (a, b, r') = \text{rbt-split-min } r \text{ in rbt-join } l \ a \ b \ r')$

lemma *set-rbt-join2[simp]*: $\text{set } (\text{RBT-Impl.keys } (\text{rbt-join2 } l \ r)) = \text{set } (\text{RBT-Impl.keys } l) \cup \text{set } (\text{RBT-Impl.keys } r)$
 ⟨proof⟩

lemma *inv-rbt-join2*: $\text{inv-12 } l \implies \text{inv-12 } r \implies \text{inv-12 } (\text{rbt-join2 } l \ r)$
 ⟨proof⟩

context *ord* **begin**

fun *rbt-split* :: $('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow ('a, 'b) \text{rbt} \times 'b \text{ option} \times ('a, 'b) \text{rbt}$ **where**
 $\text{rbt-split } \text{RBT-Impl.Empty } k = (\text{RBT-Impl.Empty}, \text{None}, \text{RBT-Impl.Empty})$
 $| \text{rbt-split } (\text{RBT-Impl.Branch } - \ l \ a \ b \ r) \ x =$
 $(\text{if } x < a \text{ then } (\text{case } \text{rbt-split } l \ x \text{ of } (l1, \beta, l2) \Rightarrow (l1, \beta, \text{rbt-join } l2 \ a \ b \ r))$
 $\text{else if } a < x \text{ then } (\text{case } \text{rbt-split } r \ x \text{ of } (r1, \beta, r2) \Rightarrow (\text{rbt-join } l \ a \ b \ r1, \beta, r2))$
 $\text{else } (l, \text{Some } b, r))$

lemma *rbt-split*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{inv-12 } t \implies \text{inv-12 } l \wedge \text{inv-12 } r$
 ⟨proof⟩

lemma *rbt-split-size*: $(l2, \beta, r2) = \text{rbt-split } t2 \ a \implies \text{size } l2 + \text{size } r2 \leq \text{size } t2$
 ⟨proof⟩

function *rbt-union-rec* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**
 $\text{rbt-union-rec } f \ t1 \ t2 = (\text{let } (f, t2, t1) =$
 $\text{if flip-rbt } t2 \ t1 \text{ then } (\lambda k \ v \ v'. f \ k \ v' \ v, t1, t2) \text{ else } (f, t2, t1) \text{ in}$
 $\text{if small-rbt } t2 \text{ then } \text{RBT-Impl.fold } (\text{rbt-insert-with-key } f) \ t2 \ t1$
 $\text{else } (\text{case } t1 \text{ of } \text{RBT-Impl.Empty} \Rightarrow t2$
 $| \text{RBT-Impl.Branch } - \ l1 \ a \ b \ r1 \Rightarrow$
 $\text{case } \text{rbt-split } t2 \ a \text{ of } (l2, \beta, r2) \Rightarrow$
 $\text{rbt-join } (\text{rbt-union-rec } f \ l1 \ l2) \ a \ (\text{case } \beta \text{ of } \text{None} \Rightarrow b \mid \text{Some } b' \Rightarrow f \ a \ b$
 $b') \ (\text{rbt-union-rec } f \ r1 \ r2)))$
 ⟨proof⟩

termination
 ⟨proof⟩

declare *rbt-union-rec.simps*[*simp del*]

function *rbt-union-swap-rec* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow \text{bool} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**

$\text{rbt-union-swap-rec } f \ \gamma \ t1 \ t2 = (\text{let } (\gamma, t2, t1) =$
 $\text{if flip-rbt } t2 \ t1 \text{ then } (\neg\gamma, t1, t2) \text{ else } (\gamma, t2, t1);$
 $f' = (\text{if } \gamma \text{ then } (\lambda k \ v \ v'. f \ k \ v' \ v) \text{ else } f) \text{ in}$
 $\text{if small-rbt } t2 \text{ then } \text{RBT-Impl.fold } (\text{rbt-insert-with-key } f') \ t2 \ t1$
 $\text{else } (\text{case } t1 \text{ of } \text{RBT-Impl.Empty} \Rightarrow t2$
 $\mid \text{RBT-Impl.Branch } - \ l1 \ a \ b \ r1 \Rightarrow$
 $\text{case rbt-split } t2 \ a \text{ of } (l2, \beta, r2) \Rightarrow$
 $\text{rbt-join } (\text{rbt-union-swap-rec } f \ \gamma \ l1 \ l2) \ a \ (\text{case } \beta \text{ of } \text{None} \Rightarrow b \mid \text{Some } b' \Rightarrow$
 $f' \ a \ b \ b') \ (\text{rbt-union-swap-rec } f \ \gamma \ r1 \ r2)))$
 $\langle \text{proof} \rangle$
termination
 $\langle \text{proof} \rangle$

declare $\text{rbt-union-swap-rec.simps}[\text{simp del}]$

lemma $\text{rbt-union-swap-rec: rbt-union-swap-rec } f \ \gamma \ t1 \ t2 =$
 $\text{rbt-union-rec } (\text{if } \gamma \text{ then } (\lambda k \ v \ v'. f \ k \ v' \ v) \text{ else } f) \ t1 \ t2$
 $\langle \text{proof} \rangle$

lemma $\text{rbt-fold-rbt-insert:}$
assumes $\text{inv-12 } t2$
shows $\text{inv-12 } (\text{RBT-Impl.fold } (\text{rbt-insert-with-key } f) \ t1 \ t2)$
 $\langle \text{proof} \rangle$

lemma $\text{rbt-union-rec: inv-12 } t1 \Longrightarrow \text{inv-12 } t2 \Longrightarrow \text{inv-12 } (\text{rbt-union-rec } f \ t1 \ t2)$
 $\langle \text{proof} \rangle$

definition $\text{map-filter-inter } f \ t1 \ t2 = \text{List.map-filter } (\lambda(k, v).$
 $\text{case rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{None}$
 $\mid \text{Some } v' \Rightarrow \text{Some } (k, f \ k \ v' \ v)) \ (\text{RBT-Impl.entries } t2)$

function $\text{rbt-inter-rec} :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a,$
 $'b) \text{rbt}$ **where**
 $\text{rbt-inter-rec } f \ t1 \ t2 = (\text{let } (f, t2, t1) =$
 $\text{if flip-rbt } t2 \ t1 \text{ then } (\lambda k \ v \ v'. f \ k \ v' \ v, t1, t2) \text{ else } (f, t2, t1) \text{ in}$
 $\text{if small-rbt } t2 \text{ then } \text{rbtreeify } (\text{map-filter-inter } f \ t1 \ t2)$
 $\text{else case } t1 \text{ of } \text{RBT-Impl.Empty} \Rightarrow \text{RBT-Impl.Empty}$
 $\mid \text{RBT-Impl.Branch } - \ l1 \ a \ b \ r1 \Rightarrow$
 $\text{case rbt-split } t2 \ a \text{ of } (l2, \beta, r2) \Rightarrow \text{let } l' = \text{rbt-inter-rec } f \ l1 \ l2; r' = \text{rbt-inter-rec}$
 $f \ r1 \ r2 \text{ in}$
 $(\text{case } \beta \text{ of } \text{None} \Rightarrow \text{rbt-join2 } l' \ r' \mid \text{Some } b' \Rightarrow \text{rbt-join } l' \ a \ (f \ a \ b \ b') \ r'))$
 $\langle \text{proof} \rangle$
termination
 $\langle \text{proof} \rangle$

declare $\text{rbt-inter-rec.simps}[\text{simp del}]$

function $\text{rbt-inter-swap-rec} :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow \text{bool} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a,$
 $'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**

```

rbt-inter-swap-rec f γ t1 t2 = (let (γ, t2, t1) =
  if flip-rbt t2 t1 then (¬γ, t1, t2) else (γ, t2, t1);
  f' = (if γ then (λk v v'. f k v' v) else f) in
  if small-rbt t2 then rbtreeify (map-filter-inter f' t1 t2)
  else case t1 of RBT-Impl.Empty ⇒ RBT-Impl.Empty
  | RBT-Impl.Branch - l1 a b r1 ⇒
    case rbt-split t2 a of (l2, β, r2) ⇒ let l' = rbt-inter-swap-rec f γ l1 l2; r' =
rbt-inter-swap-rec f γ r1 r2 in
  (case β of None ⇒ rbt-join2 l' r' | Some b' ⇒ rbt-join l' a (f' a b b') r'))
⟨proof⟩
termination
⟨proof⟩

```

declare *rbt-inter-swap-rec.simps*[simp del]

lemma *rbt-inter-swap-rec*: *rbt-inter-swap-rec* f γ t1 t2 =
rbt-inter-rec (if γ then (λk v v'. f k v' v) else f) t1 t2
 ⟨proof⟩

lemma *rbt-rbtreeify*[simp]: *inv-12* (rbtreeify kvs)
 ⟨proof⟩

lemma *rbt-inter-rec*: *inv-12* t1 ⇒ *inv-12* t2 ⇒ *inv-12* (rbt-inter-rec f t1 t2)
 ⟨proof⟩

definition *filter-minus* t1 t2 = filter (λ(k, -). rbt-lookup t2 k = None) (RBT-Impl.entries t1)

fun *rbt-minus-rec* :: ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt **where**
rbt-minus-rec t1 t2 = (if small-rbt t2 then RBT-Impl.fold (λk - t. rbt-delete k t)
 t2 t1
 else if small-rbt t1 then rbtreeify (filter-minus t1 t2)
 else case t2 of RBT-Impl.Empty ⇒ t1
 | RBT-Impl.Branch - l2 a b r2 ⇒
 case rbt-split t1 a of (l1, -, r1) ⇒ rbt-join2 (rbt-minus-rec l1 l2) (rbt-minus-rec
 r1 r2))

declare *rbt-minus-rec.simps*[simp del]

end

context *linorder* **begin**

lemma *rbt-sorted-entries-right-unique*:
 ⌈ (k, v) ∈ set (entries t); (k, v') ∈ set (entries t);
rbt-sorted t ⌋ ⇒ v = v'
 ⟨proof⟩

lemma *rbt-sorted-fold-rbt-insertwk*:

$\text{rbt-sorted } t \implies \text{rbt-sorted } (\text{List.fold } (\lambda(k, v). \text{rbt-insert-with-key } f \ k \ v) \ xs \ t)$
 $\langle \text{proof} \rangle$

lemma *is-rbt-fold-rbt-insertwk*:
assumes *is-rbt t1*
shows *is-rbt (fold (rbt-insert-with-key f) t2 t1)*
 $\langle \text{proof} \rangle$

lemma *rbt-delete: inv-12 t \implies inv-12 (rbt-delete x t)*
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-delete: rbt-sorted t \implies rbt-sorted (rbt-delete x t)*
 $\langle \text{proof} \rangle$

lemma *rbt-fold-rbt-delete*:
assumes *inv-12 t2*
shows *inv-12 (RBT-Impl.fold ($\lambda k - t. \text{rbt-delete } k \ t$) t1 t2)*
 $\langle \text{proof} \rangle$

lemma *rbt-minus-rec: inv-12 t1 \implies inv-12 t2 \implies inv-12 (rbt-minus-rec t1 t2)*
 $\langle \text{proof} \rangle$

end

context *linorder* **begin**

lemma *rbt-sorted-rbt-baliL: rbt-sorted l \implies rbt-sorted r \implies l $| \ll a \implies a \ll |$ r \implies rbt-sorted (rbt-baliL l a b r)*
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-baliL: rbt-sorted l \implies rbt-sorted r \implies l $| \ll a \implies a \ll |$ r \implies rbt-lookup (rbt-baliL l a b r) k = (if k < a then rbt-lookup l k else if k = a then Some b else rbt-lookup r k)*
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-rbt-baliR: rbt-sorted l \implies rbt-sorted r \implies l $| \ll a \implies a \ll |$ r \implies rbt-sorted (rbt-baliR l a b r)*
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-baliR: rbt-sorted l \implies rbt-sorted r \implies l $| \ll a \implies a \ll |$ r \implies rbt-lookup (rbt-baliR l a b r) k = (if k < a then rbt-lookup l k else if k = a then Some b else rbt-lookup r k)*
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-rbt-joinL: rbt-sorted (RBT-Impl.Branch c l a b r) \implies bheight l \leq bheight r \implies rbt-sorted (rbt-joinL l a b r)*
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-joinL*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 $\text{rbt-lookup } (\text{rbt-joinL } l \ a \ b \ r) \ k =$
 $(\text{if } k < a \text{ then } \text{rbt-lookup } l \ k \text{ else if } k = a \text{ then } \text{Some } b \text{ else } \text{rbt-lookup } r \ k)$
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-rbt-joinR*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 $\text{rbt-sorted } (\text{rbt-joinR } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-joinR*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 \implies
 $\text{rbt-lookup } (\text{rbt-joinR } l \ a \ b \ r) \ k =$
 $(\text{if } k < a \text{ then } \text{rbt-lookup } l \ k \text{ else if } k = a \text{ then } \text{Some } b \text{ else } \text{rbt-lookup } r \ k)$
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-paint*: $\text{rbt-sorted } (\text{paint } c \ t) = \text{rbt-sorted } t$
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-rbt-join*: $\text{rbt-sorted } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \implies$
 $\text{rbt-sorted } (\text{rbt-join } l \ a \ b \ r)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-join*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 $\text{rbt-lookup } (\text{rbt-join } l \ a \ b \ r) \ k =$
 $(\text{if } k < a \text{ then } \text{rbt-lookup } l \ k \text{ else if } k = a \text{ then } \text{Some } b \text{ else } \text{rbt-lookup } r \ k)$
 $\langle \text{proof} \rangle$

lemma *rbt-split-min-rbt-sorted*: $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq$
 $\text{RBT-Impl.Empty} \implies$
 $\text{rbt-sorted } t' \wedge (\forall x \in \text{set } (\text{RBT-Impl.keys } t'). \ a < x)$
 $\langle \text{proof} \rangle$

lemma *rbt-split-min-rbt-lookup*: $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq$
 $\text{RBT-Impl.Empty} \implies$
 $\text{rbt-lookup } t \ k = (\text{if } k < a \text{ then } \text{None} \text{ else if } k = a \text{ then } \text{Some } b \text{ else } \text{rbt-lookup } t' \ k)$
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-rbt-join2*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies$
 $\forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies \text{rbt-sorted}$
 $(\text{rbt-join2 } l \ r)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-rbt-join2*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies$
 $\forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies$
 $\text{rbt-lookup } (\text{rbt-join2 } l \ r) \ k = (\text{case } \text{rbt-lookup } l \ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } r \ k \mid \text{Some } v \Rightarrow \text{Some } v)$
 $\langle \text{proof} \rangle$

lemma *rbt-split-props*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies$
 $\text{set } (\text{RBT-Impl.keys } l) = \{a \in \text{set } (\text{RBT-Impl.keys } t). \ a < x\} \wedge$
 $\text{set } (\text{RBT-Impl.keys } r) = \{a \in \text{set } (\text{RBT-Impl.keys } t). \ x < a\} \wedge$
 $\text{rbt-sorted } l \wedge \text{rbt-sorted } r$
 $\langle \text{proof} \rangle$

lemma *rbt-split-lookup*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies$
 $\text{rbt-lookup } t \ k = (\text{if } k < x \text{ then } \text{rbt-lookup } l \ k \text{ else if } k = x \text{ then } \beta \text{ else } \text{rbt-lookup } r \ k)$
 $\langle \text{proof} \rangle$

lemma *rbt-sorted-fold-insertwk*: $\text{rbt-sorted } t \implies$
 $\text{rbt-sorted } (\text{RBT-Impl.fold } (\text{rbt-insert-with-key } f) \ t' \ t)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-iff-keys*:
 $\text{rbt-sorted } t \implies \text{set } (\text{RBT-Impl.keys } t) = \{k. \ \exists v. \ \text{rbt-lookup } t \ k = \text{Some } v\}$
 $\text{rbt-sorted } t \implies \text{rbt-lookup } t \ k = \text{None} \longleftrightarrow k \notin \text{set } (\text{RBT-Impl.keys } t)$
 $\text{rbt-sorted } t \implies (\exists v. \ \text{rbt-lookup } t \ k = \text{Some } v) \longleftrightarrow k \in \text{set } (\text{RBT-Impl.keys } t)$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-fold-rbt-insertwk*:
assumes $t1: \text{rbt-sorted } t1$ **and** $t2: \text{rbt-sorted } t2$
shows $\text{rbt-lookup } (\text{fold } (\text{rbt-insert-with-key } f) \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } t2 \ k$
 $\mid \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{Some } v$
 $\mid \text{Some } w \Rightarrow \text{Some } (f \ k \ w \ v))$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-union-rec*: $\text{rbt-sorted } t1 \implies \text{rbt-sorted } t2 \implies$
 $\text{rbt-sorted } (\text{rbt-union-rec } f \ t1 \ t2) \wedge \text{rbt-lookup } (\text{rbt-union-rec } f \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } t2 \ k$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{Some } v$
 $\mid \text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w)))$
 $\langle \text{proof} \rangle$

lemma *rbtreeify-map-filter-inter*:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$
assumes $\text{rbt-sorted } t2$
shows $\text{rbt-sorted } (\text{rbtreeify } (\text{map-filter-inter } f \ t1 \ t2))$
 $\text{rbt-lookup } (\text{rbtreeify } (\text{map-filter-inter } f \ t1 \ t2)) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{None}$
 $\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w)))$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-inter-rec*: $\text{rbt-sorted } t1 \implies \text{rbt-sorted } t2 \implies$
 $\text{rbt-sorted } (\text{rbt-inter-rec } f \ t1 \ t2) \wedge \text{rbt-lookup } (\text{rbt-inter-rec } f \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{None}$

| $\text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \text{ } k \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } w \Rightarrow \text{Some } (f \text{ } k \text{ } w))$
 $\langle \text{proof} \rangle$

lemma *rbt-lookup-delete*:

assumes *inv-12 t rbt-sorted t*

shows $\text{rbt-lookup } (\text{rbt-delete } x \text{ } t) \text{ } k = (\text{if } x = k \text{ then } \text{None} \text{ else } \text{rbt-lookup } t \text{ } k)$

$\langle \text{proof} \rangle$

lemma *fold-rbt-delete*:

assumes *inv-12 t1 rbt-sorted t1 rbt-sorted t2*

shows $\text{inv-12 } (\text{RBT-Impl.fold } (\lambda k \text{ } t. \text{rbt-delete } k \text{ } t) \text{ } t2 \text{ } t1) \wedge$

$\text{rbt-sorted } (\text{RBT-Impl.fold } (\lambda k \text{ } t. \text{rbt-delete } k \text{ } t) \text{ } t2 \text{ } t1) \wedge$

$\text{rbt-lookup } (\text{RBT-Impl.fold } (\lambda k \text{ } t. \text{rbt-delete } k \text{ } t) \text{ } t2 \text{ } t1) \text{ } k =$

$(\text{case } \text{rbt-lookup } t1 \text{ } k \text{ of } \text{None} \Rightarrow \text{None}$

$\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \text{ } k \text{ of } \text{None} \Rightarrow \text{Some } v \mid - \Rightarrow \text{None}))$

$\langle \text{proof} \rangle$

lemma *rbtreeify-filter-minus*:

assumes *rbt-sorted t1*

shows $\text{rbt-sorted } (\text{rbtreeify } (\text{filter-minus } t1 \text{ } t2)) \wedge$

$\text{rbt-lookup } (\text{rbtreeify } (\text{filter-minus } t1 \text{ } t2)) \text{ } k =$

$(\text{case } \text{rbt-lookup } t1 \text{ } k \text{ of } \text{None} \Rightarrow \text{None}$

$\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \text{ } k \text{ of } \text{None} \Rightarrow \text{Some } v \mid - \Rightarrow \text{None}))$

$\langle \text{proof} \rangle$

lemma *rbt-lookup-minus-rec*: $\text{inv-12 } t1 \Longrightarrow \text{rbt-sorted } t1 \Longrightarrow \text{rbt-sorted } t2 \Longrightarrow$

$\text{rbt-sorted } (\text{rbt-minus-rec } t1 \text{ } t2) \wedge \text{rbt-lookup } (\text{rbt-minus-rec } t1 \text{ } t2) \text{ } k =$

$(\text{case } \text{rbt-lookup } t1 \text{ } k \text{ of } \text{None} \Rightarrow \text{None}$

$\mid \text{Some } v \Rightarrow (\text{case } \text{rbt-lookup } t2 \text{ } k \text{ of } \text{None} \Rightarrow \text{Some } v \mid - \Rightarrow \text{None}))$

$\langle \text{proof} \rangle$

end

context *ord* **begin**

definition *rbt-union-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

$\Rightarrow ('a, 'b) \text{rbt}$

where

$\text{rbt-union-with-key } f \text{ } t1 \text{ } t2 = \text{paint } B \text{ } (\text{rbt-union-swap-rec } f \text{ } \text{False } t1 \text{ } t2)$

definition *rbt-union-with* **where**

$\text{rbt-union-with } f = \text{rbt-union-with-key } (\lambda \text{ } -. f)$

definition *rbt-union* **where**

$\text{rbt-union} = \text{rbt-union-with-key } (\% - \text{ } rv. \text{ } rv)$

definition *rbt-inter-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

$\Rightarrow ('a, 'b) \text{rbt}$

where

$\text{rbt-inter-with-key } f \ t1 \ t2 = \text{paint } B \ (\text{rbt-inter-swap-rec } f \ \text{False } t1 \ t2)$

definition *rbt-inter-with* **where**

$\text{rbt-inter-with } f = \text{rbt-inter-with-key } (\lambda -. f)$

definition *rbt-inter* **where**

$\text{rbt-inter} = \text{rbt-inter-with-key } (\lambda - \text{ rv. rv})$

definition *rbt-minus* **where**

$\text{rbt-minus } t1 \ t2 = \text{paint } B \ (\text{rbt-minus-rec } t1 \ t2)$

end

context *linorder* **begin**

lemma *is-rbt-rbt-unionwk* [simp]:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-union-with-key } f \ t1 \ t2)$
 <proof>

lemma *rbt-lookup-rbt-unionwk*:

$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$
 $\implies \text{rbt-lookup } (\text{rbt-union-with-key } f \ t1 \ t2) \ k =$
 (case $\text{rbt-lookup } t1 \ k$ of $\text{None} \Rightarrow \text{rbt-lookup } t2 \ k$
 | $\text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \ k$ of $\text{None} \Rightarrow \text{Some } v$
 | $\text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w)$)
 <proof>

lemma *rbt-unionw-is-rbt*: $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \implies \text{is-rbt } (\text{rbt-union-with } f \ lt \ rt)$

<proof>

lemma *rbt-union-is-rbt*: $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \implies \text{is-rbt } (\text{rbt-union } lt \ rt)$

<proof>

lemma *rbt-lookup-rbt-union*:

$\llbracket \text{rbt-sorted } s; \text{rbt-sorted } t \rrbracket \implies$
 $\text{rbt-lookup } (\text{rbt-union } s \ t) = \text{rbt-lookup } s \ ++ \ \text{rbt-lookup } t$
 <proof>

lemma *rbt-interwk-is-rbt* [simp]:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter-with-key } f \ t1 \ t2)$
 <proof>

lemma *rbt-interw-is-rbt*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter-with } f \ t1 \ t2)$
 <proof>

lemma *rbt-inter-is-rbt*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter } t1 \ t2)$
 <proof>

lemma *rbt-lookup-rbt-interwk*:

$$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$$

$$\implies \text{rbt-lookup } (\text{rbt-inter-with-key } f \ t1 \ t2) \ k =$$

$$(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{None}$$

$$| \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{None}$$

$$| \text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w))$$

$$\langle \text{proof} \rangle$$

lemma *rbt-lookup-rbt-inter*:

$$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$$

$$\implies \text{rbt-lookup } (\text{rbt-inter } t1 \ t2) = \text{rbt-lookup } t2 \mid ' \text{dom } (\text{rbt-lookup } t1)$$

$$\langle \text{proof} \rangle$$

lemma *rbt-minus-is-rbt*:

$$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-minus } t1 \ t2)$$

$$\langle \text{proof} \rangle$$

lemma *rbt-lookup-rbt-minus*:

$$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket$$

$$\implies \text{rbt-lookup } (\text{rbt-minus } t1 \ t2) = \text{rbt-lookup } t1 \mid ' (- \text{dom } (\text{rbt-lookup } t2))$$

$$\langle \text{proof} \rangle$$

end

129.11 Code generator setup

lemmas [code] =
ord.rbt-less-prop
ord.rbt-greater-prop
ord.rbt-sorted.simps
ord.rbt-lookup.simps
ord.is-rbt-def
ord.rbt-ins.simps
ord.rbt-insert-with-key-def
ord.rbt-insertw-def
ord.rbt-insert-def
ord.rbt-del-from-left.simps
ord.rbt-del-from-right.simps
ord.rbt-del.simps
ord.rbt-delete-def
ord.rbt-split.simps
ord.rbt-union-swap-rec.simps
ord.map-filter-inter-def
ord.rbt-inter-swap-rec.simps
ord.filter-minus-def
ord.rbt-minus-rec.simps
ord.rbt-union-with-key-def
ord.rbt-union-with-def

`ord.rbt-union-def`
`ord.rbt-inter-with-key-def`
`ord.rbt-inter-with-def`
`ord.rbt-inter-def`
`ord.rbt-minus-def`
`ord.rbt-map-entry.simps`
`ord.rbt-bulkload-def`

More efficient implementations for *entries* and *keys*

definition *gen-entries* ::

$((\text{'a} \times \text{'b}) \times (\text{'a}, \text{'b}) \text{ rbt}) \text{ list} \Rightarrow (\text{'a}, \text{'b}) \text{ rbt} \Rightarrow (\text{'a} \times \text{'b}) \text{ list}$

where

$\text{gen-entries kvs } t = \text{entries } t @ \text{concat } (\text{map } (\lambda(kv, t). kv \# \text{entries } t) \text{ kvs})$

lemma *gen-entries-simps* [*simp*, *code*]:

$\text{gen-entries } [] \text{ Empty} = []$

$\text{gen-entries } ((kv, t) \# \text{kvs}) \text{ Empty} = kv \# \text{gen-entries kvs } t$

$\text{gen-entries kvs } (\text{Branch } c \text{ l } k \text{ v } r) = \text{gen-entries } (((k, v), r) \# \text{kvs}) \text{ l}$

$\langle \text{proof} \rangle$

lemma *entries-code* [*code*]:

$\text{entries} = \text{gen-entries } []$

$\langle \text{proof} \rangle$

definition *gen-keys* :: $(\text{'a} \times (\text{'a}, \text{'b}) \text{ rbt}) \text{ list} \Rightarrow (\text{'a}, \text{'b}) \text{ rbt} \Rightarrow \text{'a} \text{ list}$

where $\text{gen-keys kts } t = \text{RBT-Impl.keys } t @ \text{concat } (\text{List.map } (\lambda(k, t). k \# \text{keys } t) \text{ kts})$

lemma *gen-keys-simps* [*simp*, *code*]:

$\text{gen-keys } [] \text{ Empty} = []$

$\text{gen-keys } ((k, t) \# \text{kts}) \text{ Empty} = k \# \text{gen-keys kts } t$

$\text{gen-keys kts } (\text{Branch } c \text{ l } k \text{ v } r) = \text{gen-keys } ((k, r) \# \text{kts}) \text{ l}$

$\langle \text{proof} \rangle$

lemma *keys-code* [*code*]:

$\text{keys} = \text{gen-keys } []$

$\langle \text{proof} \rangle$

Restore original type constraints for constants

$\langle \text{ML} \rangle$

hide-const (**open**) *MR MB R B Empty entries keys fold gen-keys gen-entries*

end

130 Abstract type of RBT trees

theory *RBT*

imports *Main RBT-Impl*

begin

130.1 Type definition

typedef (overloaded) ('a, 'b) rbt = {t :: ('a::linorder, 'b) RBT-Impl.rbt. is-rbt t}

morphisms impl-of RBT
 ⟨proof⟩

lemma rbt-eq-iff:
 $t1 = t2 \longleftrightarrow \text{impl-of } t1 = \text{impl-of } t2$
 ⟨proof⟩

lemma rbt-eqI:
 $\text{impl-of } t1 = \text{impl-of } t2 \implies t1 = t2$
 ⟨proof⟩

lemma is-rbt-impl-of [simp, intro]:
 $\text{is-rbt } (\text{impl-of } t)$
 ⟨proof⟩

lemma RBT-impl-of [simp, code abstype]:
 $\text{RBT } (\text{impl-of } t) = t$
 ⟨proof⟩

130.2 Primitive operations

setup-lifting type-definition-rbt

lift-definition lookup :: ('a::linorder, 'b) rbt \Rightarrow 'a \rightarrow 'b **is** rbt-lookup ⟨proof⟩

lift-definition empty :: ('a::linorder, 'b) rbt **is** RBT-Impl.Empty
 ⟨proof⟩

lift-definition insert :: 'a::linorder \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **is** rbt-insert
 ⟨proof⟩

lift-definition delete :: 'a::linorder \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt **is** rbt-delete
 ⟨proof⟩

lift-definition entries :: ('a::linorder, 'b) rbt \Rightarrow ('a \times 'b) list **is** RBT-Impl.entries
 ⟨proof⟩

lift-definition keys :: ('a::linorder, 'b) rbt \Rightarrow 'a list **is** RBT-Impl.keys ⟨proof⟩

lift-definition bulkload :: ('a::linorder \times 'b) list \Rightarrow ('a, 'b) rbt **is** rbt-bulkload
 ⟨proof⟩

lift-definition *map-entry* :: $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$
is *rbt-map-entry*
 $\langle \text{proof} \rangle$

lift-definition *map* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'c) \text{rbt}$ **is**
RBT-Impl.map
 $\langle \text{proof} \rangle$

lift-definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow 'c \Rightarrow 'c$ **is**
RBT-Impl.fold $\langle \text{proof} \rangle$

lift-definition *union* :: $('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **is**
rbt-union
 $\langle \text{proof} \rangle$

lift-definition *foldi* :: $('c \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a :: \text{linorder}, 'b)$
 $\text{rbt} \Rightarrow 'c \Rightarrow 'c$
is *RBT-Impl.foldi* $\langle \text{proof} \rangle$

lift-definition *combine-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow$
 $('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$
is *RBT-Impl.rbt-union-with-key* $\langle \text{proof} \rangle$

lift-definition *combine* :: $('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow$
 $('a, 'b) \text{rbt}$
is *RBT-Impl.rbt-union-with* $\langle \text{proof} \rangle$

130.3 Derived operations

definition *is-empty* :: $('a::\text{linorder}, 'b) \text{rbt} \Rightarrow \text{bool}$ **where**
 $[\text{code}]: \text{is-empty } t = (\text{case impl-of } t \text{ of } \text{RBT-Impl.Empty} \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

definition *filter* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a::\text{linorder}, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**
 $[\text{code}]: \text{filter } P \ t = \text{fold } (\lambda k \ v \ t. \text{if } P \ k \ v \ \text{then } \text{insert } k \ v \ t \ \text{else } t) \ t \ \text{empty}$

130.4 Abstract lookup properties

lemma *lookup-RBT*:
 $\text{is-rbt } t \Longrightarrow \text{lookup } (\text{RBT } t) = \text{rbt-lookup } t$
 $\langle \text{proof} \rangle$

lemma *lookup-impl-of*:
 $\text{rbt-lookup } (\text{impl-of } t) = \text{lookup } t$
 $\langle \text{proof} \rangle$

lemma *entries-impl-of*:
 $\text{RBT-Impl.entries } (\text{impl-of } t) = \text{entries } t$
 $\langle \text{proof} \rangle$

lemma *keys-impl-of*:

$$RBT-Impl.keys\ (impl\text{-}of\ t) = keys\ t$$

<proof>

lemma *lookup-keys*:

$$dom\ (lookup\ t) = set\ (keys\ t)$$

<proof>

lemma *lookup-empty* [*simp*]:

$$lookup\ empty = Map.empty$$

<proof>

lemma *lookup-insert* [*simp*]:

$$lookup\ (insert\ k\ v\ t) = (lookup\ t)(k \mapsto v)$$

<proof>

lemma *lookup-delete* [*simp*]:

$$lookup\ (delete\ k\ t) = (lookup\ t)(k := None)$$

<proof>

lemma *map-of-entries* [*simp*]:

$$map\text{-}of\ (entries\ t) = lookup\ t$$

<proof>

lemma *entries-lookup*:

$$entries\ t1 = entries\ t2 \longleftrightarrow lookup\ t1 = lookup\ t2$$

<proof>

lemma *lookup-bulkload* [*simp*]:

$$lookup\ (bulkload\ xs) = map\text{-}of\ xs$$

<proof>

lemma *lookup-map-entry* [*simp*]:

$$lookup\ (map\text{-}entry\ k\ f\ t) = (lookup\ t)(k := map\text{-}option\ f\ (lookup\ t\ k))$$

<proof>

lemma *lookup-map* [*simp*]:

$$lookup\ (map\ f\ t)\ k = map\text{-}option\ (f\ k)\ (lookup\ t\ k)$$

<proof>

lemma *lookup-combine-with-key* [*simp*]:

$$lookup\ (combine\text{-}with\text{-}key\ f\ t1\ t2)\ k = combine\text{-}options\ (f\ k)\ (lookup\ t1\ k)\ (lookup\ t2\ k)$$

<proof>

lemma *combine-altdef*: $combine\ f\ t1\ t2 = combine\text{-}with\text{-}key\ (\lambda\cdot. f)\ t1\ t2$

<proof>

lemma *lookup-combine* [*simp*]:

lookup (combine f t1 t2) k = combine-options f (lookup t1 k) (lookup t2 k)
⟨proof⟩

lemma *fold-fold*:
fold f t = List.fold (case-prod f) (entries t)
⟨proof⟩

lemma *impl-of-empty*:
impl-of empty = RBT-Impl.Empty
⟨proof⟩

lemma *is-empty-empty [simp]*:
is-empty t \longleftrightarrow t = empty
⟨proof⟩

lemma *RBT-lookup-empty [simp]*:
rbt-lookup t = Map.empty \longleftrightarrow t = RBT-Impl.Empty
⟨proof⟩

lemma *lookup-empty-empty [simp]*:
lookup t = Map.empty \longleftrightarrow t = empty
⟨proof⟩

lemma *keys-empty-eq [simp]*:
⟨keys empty = []⟩
⟨proof⟩

lemma *sorted-keys [iff]*:
sorted (keys t)
⟨proof⟩

lemma *distinct-keys [iff]*:
distinct (keys t)
⟨proof⟩

lemma *finite-dom-lookup [simp, intro!]*: *finite (dom (lookup t))*
⟨proof⟩

lemma *lookup-union*: *lookup (union s t) = lookup s ++ lookup t*
⟨proof⟩

lemma *lookup-in-tree*: *(lookup t k = Some v) = ((k, v) ∈ set (entries t))*
⟨proof⟩

lemma *keys-entries*: *(k ∈ set (keys t)) = (∃ v. (k, v) ∈ set (entries t))*
⟨proof⟩

lemma *fold-def-alt*:
fold f t = List.fold (case-prod f) (entries t)

<proof>

lemma *distinct-entries*: *distinct (List.map fst (entries t))*
<proof>

lemma *sorted-entries*: *sorted (List.map fst (entries t))*
<proof>

lemma *non-empty-keys*: *t ≠ empty ⇒ keys t ≠ []*
<proof>

lemma *keys-def-alt*:
keys t = List.map fst (entries t)
<proof>

context
begin

private lemma *lookup-filter-aux*:
assumes *distinct (List.map fst xs)*
shows *lookup (List.fold (λ(k, v) t. if P k v then insert k v t else t) xs t) k =*
(case map-of xs k of
None ⇒ lookup t k
| Some v ⇒ if P k v then Some v else lookup t k)
<proof>

lemma *lookup-filter*:
lookup (filter P t) k =
(case lookup t k of None ⇒ None | Some v ⇒ if P k v then Some v else None)
<proof>

end

130.5 Quickcheck generators

quickcheck-generator *rbt predicate: is-rbt constructors: empty, insert*

130.6 Hide implementation details

lifting-update *rbt.lifting*

lifting-forget *rbt.lifting*

hide-const (**open**) *impl-of empty lookup keys entries bulkload delete map fold*
union insert map-entry foldi

is-empty filter

hide-fact (**open**) *empty-def lookup-def keys-def entries-def bulkload-def delete-def*
map-def fold-def

union-def insert-def map-entry-def foldi-def is-empty-def filter-def

end

[illegible]

131.1 Data type and invariant

A value t of this type is a valid red-black tree if it satisfies the invariant $is_rbt\ t$. The abstract type $(\text{'}k, \text{'}v)\ RBT.rbt$ always obeys this invariant, and for this reason you should only use this in our application. Going back to $(\text{'}k, \text{'}v)\ RBT_Impl.rbt$ may be necessary in proofs if not yet proven properties about the operations must be established.

 $RBT.lookup$

131.2 Operations

 $RBT.empty$ $RBT.insert$ *RBT.delete**RBT.entries* $RBT.bulkload$

Builds a tree from a key-value list.

RBT.map-entry

Maps a single entry in a tree.

RBT.map

Maps all values in a tree. $O(n)$

RBT.fold

Folds over all entries in a tree. $O(n)$

131.3 Invariant preservation

<i>is-rbt</i> <i>rbt.Empty</i>	(<i>Empty-is-rbt</i>)
<i>is-rbt</i> ? <i>t</i> \implies <i>is-rbt</i> (<i>rbt-insert</i> ? <i>k</i> ? <i>v</i> ? <i>t</i>)	(<i>rbt-insert-is-rbt</i>)
<i>is-rbt</i> ? <i>t</i> \implies <i>is-rbt</i> (<i>rbt-delete</i> ? <i>k</i> ? <i>t</i>)	(<i>delete-is-rbt</i>)
<i>is-rbt</i> (<i>rbt-bulkload</i> ? <i>xs</i>)	(<i>bulkload-is-rbt</i>)
<i>is-rbt</i> (<i>rbt-map-entry</i> ? <i>k</i> ? <i>f</i> ? <i>t</i>) = <i>is-rbt</i> ? <i>t</i>	(<i>map-entry-is-rbt</i>)
<i>is-rbt</i> (<i>RBT-Impl.map</i> ? <i>f</i> ? <i>t</i>) = <i>is-rbt</i> ? <i>t</i>	(<i>map-is-rbt</i>)
$\llbracket \text{is-rbt } ?lt; \text{is-rbt } ?rt \rrbracket \implies \text{is-rbt } (\text{rbt-union } ?lt \text{ } ?rt)$	(<i>union-is-rbt</i>)

131.4 Map Semantics

lookup-empty

Mapping.lookup Mapping.empty ?*k* = *None*

lookup-insert

RBT.lookup (*RBT.insert* ?*k* ?*v* ?*t*) = (*RBT.lookup* ?*t*)(?*k* \mapsto ?*v*)

lookup-delete

Mapping.lookup (*Mapping.delete* ?*k* ?*m*) ?*k* = *None*

lookup-bulkload

RBT.lookup (*RBT.bulkload* ?*xs*) = *map-of* ?*xs*

lookup-map

RBT.lookup (*RBT.map* ?*f* ?*t*) ?*k* = *map-option* (?*f* ?*k*) (*RBT.lookup* ?*t* ?*k*)

end

132 Implementation of sets using RBT trees

```
theory RBT-Set
imports RBT Product-Lexorder
begin
```

133 Definition of code datatype constructors

```
definition Set :: ('a::linorder, unit) rbt  $\Rightarrow$  'a set
  where Set t = {x . RBT.lookup t x = Some ()}
```

```
definition Coset :: ('a::linorder, unit) rbt  $\Rightarrow$  'a set
  where [simp]: Coset t = - Set t
```

134 Lemmas

134.1 Auxiliary lemmas

```
lemma [simp]: x  $\neq$  Some ()  $\longleftrightarrow$  x = None
<proof>
```

```
lemma Set-set-keys: Set x = dom (RBT.lookup x)
<proof>
```

```
lemma finite-Set [simp, intro!]: finite (Set x)
<proof>
```

```
lemma set-keys: Set t = set(RBT.keys t)
<proof>
```

134.2 fold and filter

```
lemma finite-fold-rbt-fold-eq:
  assumes comp-fun-commute f
  shows Finite-Set.fold f A (set (RBT.entries t)) = RBT.fold (curry f) t A
<proof>
```

```
definition fold-keys :: ('a :: linorder  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, -) rbt  $\Rightarrow$  'b  $\Rightarrow$  'b
  where [code-unfold]: fold-keys f t A = RBT.fold ( $\lambda$ k - t. f k t) t A
```

```
lemma fold-keys-def-alt:
  fold-keys f t s = List.fold f (RBT.keys t) s
<proof>
```

```
lemma finite-fold-fold-keys:
  assumes comp-fun-commute f
  shows Finite-Set.fold f A (Set t) = fold-keys f t A
<proof>
```

definition *rbt-filter* :: (*'a* :: *linorder* \Rightarrow *bool*) \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow *'a set* **where**
rbt-filter *P t* = *RBT.fold* (λk - *A'*. if *P k* then *Set.insert k A'* else *A'*) *t* {}

lemma *Set-filter-rbt-filter*:
Set.filter P (Set t) = *rbt-filter P t*
 <proof>

134.3 foldi and Ball

lemma *Ball-False*: *RBT-Impl.fold* (λk *v s*. *s* \wedge *P k*) *t False* = *False*
 <proof>

lemma *rbt-foldi-fold-conj*:
RBT-Impl.foldi (λs . *s* = *True*) (λk *v s*. *s* \wedge *P k*) *t val* = *RBT-Impl.fold* (λk *v s*.
s \wedge *P k*) *t val*
 <proof>

lemma *foldi-fold-conj*: *RBT.foldi* (λs . *s* = *True*) (λk *v s*. *s* \wedge *P k*) *t val* = *fold-keys*
 (λk *s*. *s* \wedge *P k*) *t val*
 <proof> **including** *rbt.lifting* <proof>

134.4 foldi and Bex

lemma *Bex-True*: *RBT-Impl.fold* (λk *v s*. *s* \vee *P k*) *t True* = *True*
 <proof>

lemma *rbt-foldi-fold-disj*:
RBT-Impl.foldi (λs . *s* = *False*) (λk *v s*. *s* \vee *P k*) *t val* = *RBT-Impl.fold* (λk *v s*.
s \vee *P k*) *t val*
 <proof>

lemma *foldi-fold-disj*: *RBT.foldi* (λs . *s* = *False*) (λk *v s*. *s* \vee *P k*) *t val* = *fold-keys*
 (λk *s*. *s* \vee *P k*) *t val*
 <proof> **including** *rbt.lifting* <proof>

134.5 folding over non empty trees and selecting the minimal and maximal element

134.5.1 concrete

The concrete part is here because it's probably not general enough to be moved to *RBT-Impl*

definition *rbt-fold1-keys* :: (*'a* \Rightarrow *'a* \Rightarrow *'a*) \Rightarrow (*'a*::*linorder*, *'b*) *RBT-Impl.rbt* \Rightarrow *'a*
where *rbt-fold1-keys f t* = *List.fold f* (*tl*(*RBT-Impl.keys t*)) (*hd*(*RBT-Impl.keys t*))

minimum definition *rbt-min* :: (*'a*::*linorder*, *unit*) *RBT-Impl.rbt* \Rightarrow *'a*
where *rbt-min t* = *rbt-fold1-keys min t*

lemma *key-le-right*: $\text{rbt-sorted } (\text{Branch } c \text{ lt } k \text{ v } rt) \implies (\bigwedge x. x \in \text{set } (\text{RBT-Impl.keys } rt) \implies k \leq x)$
 <proof>

lemma *left-le-key*: $\text{rbt-sorted } (\text{Branch } c \text{ lt } k \text{ v } rt) \implies (\bigwedge x. x \in \text{set } (\text{RBT-Impl.keys } lt) \implies x \leq k)$
 <proof>

lemma *fold-min-triv*:
 fixes $k :: - :: \text{linorder}$
 shows $(\forall x \in \text{set } xs. k \leq x) \implies \text{List.fold min } xs \text{ } k = k$
 <proof>

lemma *rbt-min-simps*:
 $\text{is-rbt } (\text{Branch } c \text{ RBT-Impl.Empty } k \text{ v } rt) \implies \text{rbt-min } (\text{Branch } c \text{ RBT-Impl.Empty } k \text{ v } rt) = k$
 <proof>

fun *rbt-min-opt* **where**
 $\text{rbt-min-opt } (\text{Branch } c \text{ RBT-Impl.Empty } k \text{ v } rt) = k \mid$
 $\text{rbt-min-opt } (\text{Branch } c \text{ (Branch } lc \text{ llc } lk \text{ lv } lrt) \text{ } k \text{ v } rt) = \text{rbt-min-opt } (\text{Branch } lc \text{ llc } lk \text{ lv } lrt)$

lemma *rbt-min-opt-Branch*:
 $t1 \neq \text{rbt.Empty} \implies \text{rbt-min-opt } (\text{Branch } c \text{ } t1 \text{ } k \text{ } () \text{ } t2) = \text{rbt-min-opt } t1$
 <proof>

lemma *rbt-min-opt-induct* [case-names empty left-empty left-non-empty]:
 fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
 assumes $P \text{ rbt.Empty}$
 assumes $\bigwedge \text{color } t1 \text{ } a \text{ } b \text{ } t2. P \text{ } t1 \implies P \text{ } t2 \implies t1 = \text{rbt.Empty} \implies P \text{ (Branch color } t1 \text{ } a \text{ } b \text{ } t2)$
 assumes $\bigwedge \text{color } t1 \text{ } a \text{ } b \text{ } t2. P \text{ } t1 \implies P \text{ } t2 \implies t1 \neq \text{rbt.Empty} \implies P \text{ (Branch color } t1 \text{ } a \text{ } b \text{ } t2)$
 shows $P \text{ } t$
 <proof>

lemma *rbt-min-opt-in-set*:
 fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
 assumes $t \neq \text{rbt.Empty}$
 shows $\text{rbt-min-opt } t \in \text{set } (\text{RBT-Impl.keys } t)$
 <proof>

lemma *rbt-min-opt-is-min*:
 fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
 assumes $\text{rbt-sorted } t$
 assumes $t \neq \text{rbt.Empty}$
 shows $\bigwedge y. y \in \text{set } (\text{RBT-Impl.keys } t) \implies y \geq \text{rbt-min-opt } t$

<proof>

lemma *rbt-min-eq-rbt-min-opt*:
assumes $t \neq \text{RBT-Impl.Empty}$
assumes *is-rbt* t
shows $\text{rbt-min } t = \text{rbt-min-opt } t$
<proof>

maximum definition $\text{rbt-max} :: ('a :: \text{linorder}, \text{unit}) \text{RBT-Impl.rbt} \Rightarrow 'a$
where $\text{rbt-max } t = \text{rbt-fold1-keys max } t$

lemma *fold-max-triv*:
fixes $k :: - :: \text{linorder}$
shows $(\forall x \in \text{set } xs. x \leq k) \implies \text{List.fold max } xs \ k = k$
<proof>

lemma *fold-max-rev-eq*:
fixes $xs :: ('a :: \text{linorder}) \text{list}$
assumes $xs \neq []$
shows $\text{List.fold max } (tl \ xs) \ (hd \ xs) = \text{List.fold max } (tl \ (\text{rev } xs)) \ (hd \ (\text{rev } xs))$
<proof>

lemma *rbt-max-simps*:
assumes *is-rbt* $(\text{Branch } c \ lt \ k \ v \ \text{RBT-Impl.Empty})$
shows $\text{rbt-max } (\text{Branch } c \ lt \ k \ v \ \text{RBT-Impl.Empty}) = k$
<proof>

fun *rbt-max-opt where*
 $\text{rbt-max-opt } (\text{Branch } c \ lt \ k \ v \ \text{RBT-Impl.Empty}) = k \mid$
 $\text{rbt-max-opt } (\text{Branch } c \ lt \ k \ v \ (\text{Branch } rc \ rlc \ rk \ rv \ rrt)) = \text{rbt-max-opt } (\text{Branch } rc$
 $rlc \ rk \ rv \ rrt)$

lemma *rbt-max-opt-Branch*:
 $t2 \neq \text{rbt.Empty} \implies \text{rbt-max-opt } (\text{Branch } c \ t1 \ k \ () \ t2) = \text{rbt-max-opt } t2$
<proof>

lemma *rbt-max-opt-induct* [*case-names empty right-empty right-non-empty*]:
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{RBT-Impl.rbt}$
assumes $P \ \text{rbt.Empty}$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t2 = \text{rbt.Empty} \implies P \ (\text{Branch}$
 $\text{color } t1 \ a \ b \ t2)$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t2 \neq \text{rbt.Empty} \implies P \ (\text{Branch}$
 $\text{color } t1 \ a \ b \ t2)$
shows $P \ t$
<proof>

lemma *rbt-max-opt-in-set*:
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{RBT-Impl.rbt}$
assumes $t \neq \text{rbt.Empty}$

shows $\text{rbt-max-opt } t \in \text{set } (\text{RBT-Impl.keys } t)$
 $\langle \text{proof} \rangle$

lemma $\text{rbt-max-opt-is-max}$:
fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes $\text{rbt-sorted } t$
assumes $t \neq \text{rbt.Empty}$
shows $\bigwedge y. y \in \text{set } (\text{RBT-Impl.keys } t) \implies y \leq \text{rbt-max-opt } t$
 $\langle \text{proof} \rangle$

lemma $\text{rbt-max-eq-rbt-max-opt}$:
assumes $t \neq \text{RBT-Impl.Empty}$
assumes $\text{is-rbt } t$
shows $\text{rbt-max } t = \text{rbt-max-opt } t$
 $\langle \text{proof} \rangle$

134.5.2 abstract

context includes rbt.lifting **begin**

lift-definition $\text{fold1-keys} :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a :: \text{linorder}, 'b) \text{ rbt} \Rightarrow 'a$
is $\text{rbt-fold1-keys} \langle \text{proof} \rangle$

lemma $\text{fold1-keys-def-alt}$:
 $\text{fold1-keys } f \ t = \text{List.fold } f \ (\text{tl } (\text{RBT.keys } t)) \ (\text{hd } (\text{RBT.keys } t))$
 $\langle \text{proof} \rangle$

lemma $\text{finite-fold1-fold1-keys}$:
assumes $\text{semilattice } f$
assumes $\neg \text{RBT.is-empty } t$
shows $\text{semilattice-set.F } f \ (\text{Set } t) = \text{fold1-keys } f \ t$
 $\langle \text{proof} \rangle$

minimum lift-definition $\text{r-min} :: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a$ **is** rbt-min
 $\langle \text{proof} \rangle$

lift-definition $\text{r-min-opt} :: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a$ **is** $\text{rbt-min-opt} \langle \text{proof} \rangle$

lemma r-min-alt-def : $\text{r-min } t = \text{fold1-keys } \text{min } t$
 $\langle \text{proof} \rangle$

lemma $\text{r-min-eq-r-min-opt}$:
assumes $\neg (\text{RBT.is-empty } t)$
shows $\text{r-min } t = \text{r-min-opt } t$
 $\langle \text{proof} \rangle$

lemma $\text{fold-keys-min-top-eq}$:
fixes $t :: ('a :: \{\text{linorder}, \text{bounded-lattice-top}\}, \text{unit}) \text{ rbt}$
assumes $\neg (\text{RBT.is-empty } t)$
shows $\text{fold-keys } \text{min } t \ \text{top} = \text{fold1-keys } \text{min } t$

<proof>

maximum lift-definition $r\text{-max} :: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a \text{ is } r\text{bt-max}$
<proof>

lift-definition $r\text{-max-opt} :: ('a :: \text{linorder}, \text{unit}) \text{ rbt} \Rightarrow 'a \text{ is } r\text{bt-max-opt}$ *<proof>*

lemma $r\text{-max-alt-def}$: $r\text{-max } t = \text{fold1-keys max } t$
<proof>

lemma $r\text{-max-eq-r-max-opt}$:
assumes $\neg (\text{RBT.is-empty } t)$
shows $r\text{-max } t = r\text{-max-opt } t$
<proof>

lemma $\text{fold-keys-max-bot-eq}$:
fixes $t :: ('a :: \{\text{linorder}, \text{bounded-lattice-bot}\}, \text{unit}) \text{ rbt}$
assumes $\neg (\text{RBT.is-empty } t)$
shows $\text{fold-keys max } t \text{ bot} = \text{fold1-keys max } t$
<proof>

end

135 Code equations

code-datatype Set Coset

declare $\text{list.set}[code]$

lemma $\text{empty-Set } [code]$:
 $\text{Set.empty} = \text{Set RBT.empty}$
<proof>

lemma $\text{UNIV-Coset } [code]$:
 $\text{UNIV} = \text{Coset RBT.empty}$
<proof>

lemma $\text{is-empty-Set } [code]$:
 $\text{Set.is-empty } (\text{Set } t) = \text{RBT.is-empty } t$
<proof>

lemma $\text{compl-code } [code]$:
 – $\text{Set } xs = \text{Coset } xs$
 – $\text{Coset } xs = \text{Set } xs$
<proof>

lemma $\text{member-code } [code]$:
 $x \in (\text{Set } t) = (\text{RBT.lookup } t \ x = \text{Some } ())$
 $x \in (\text{Coset } t) = (\text{RBT.lookup } t \ x = \text{None})$

$\langle proof \rangle$

lemma *insert-code* [code]:

$Set.insert\ x\ (Set\ t) = Set\ (RBT.insert\ x\ ()\ t)$

$Set.insert\ x\ (Coset\ t) = Coset\ (RBT.delete\ x\ t)$

$\langle proof \rangle$

lemma *remove-code* [code]:

$Set.remove\ x\ (Set\ t) = Set\ (RBT.delete\ x\ t)$

$Set.remove\ x\ (Coset\ t) = Coset\ (RBT.insert\ x\ ()\ t)$

$\langle proof \rangle$

lemma *inter-Set* [code]:

$A \cap Set\ t = rbt-filter\ (\lambda k. k \in A)\ t$

$\langle proof \rangle$

lemma *union-Set-Set* [code]:

$Set\ t1 \cup Set\ t2 = Set\ (RBT.union\ t1\ t2)$

$\langle proof \rangle$

lemma *union-Set* [code]:

$Set\ t \cup A = fold-keys\ Set.insert\ t\ A$

$\langle proof \rangle$

lemma *minus-Set* [code]:

$A - Set\ t = fold-keys\ Set.remove\ t\ A$

$\langle proof \rangle$

lemma *inter-Coset-Coset* [code]:

$Coset\ t1 \cap Coset\ t2 = Coset\ (RBT.union\ t1\ t2)$

$\langle proof \rangle$

lemma *inter-Coset* [code]:

$A \cap Coset\ t = fold-keys\ Set.remove\ t\ A$

$\langle proof \rangle$

lemma *union-Coset* [code]:

$Coset\ t \cup A = -\ rbt-filter\ (\lambda k. k \notin A)\ t$

$\langle proof \rangle$

lemma *minus-Coset* [code]:

$A - Coset\ t = rbt-filter\ (\lambda k. k \in A)\ t$

$\langle proof \rangle$

lemma *filter-Set* [code]:

$Set.filter\ P\ (Set\ t) = rbt-filter\ P\ t$

$\langle proof \rangle$

lemma *image-Set* [code]:

$image\ f\ (Set\ t) = fold-keys\ (\lambda k\ A.\ Set.insert\ (f\ k)\ A)\ t\ \{\}$
 $\langle proof \rangle$

lemma *Ball-Set* [code]:

$Ball\ (Set\ t)\ P \longleftrightarrow RBT.foldi\ (\lambda s.\ s = True)\ (\lambda k\ v\ s.\ s \wedge P\ k)\ t\ True$
 $\langle proof \rangle$

lemma *Bex-Set* [code]:

$Bex\ (Set\ t)\ P \longleftrightarrow RBT.foldi\ (\lambda s.\ s = False)\ (\lambda k\ v\ s.\ s \vee P\ k)\ t\ False$
 $\langle proof \rangle$

lemma *subset-code* [code]:

$Set\ t \leq B \longleftrightarrow (\forall x \in Set\ t.\ x \in B)$
 $A \leq Coset\ t \longleftrightarrow (\forall y \in Set\ t.\ y \notin A)$
 $\langle proof \rangle$

lemma *subset-Coset-empty-Set-empty* [code]:

$Coset\ t1 \leq Set\ t2 \longleftrightarrow (case\ (RBT.impl-of\ t1,\ RBT.impl-of\ t2)\ of$
 $(rbt.Empty,\ rbt.Empty) \Rightarrow False \mid$
 $(-, -) \Rightarrow Code.abort\ (STR\ "non-empty-trees")\ (\lambda -. Coset\ t1 \leq Set\ t2))$
 $\langle proof \rangle$

A frequent case – avoid intermediate sets

lemma [code-unfold]:

$Set\ t1 \subseteq Set\ t2 \longleftrightarrow RBT.foldi\ (\lambda s.\ s = True)\ (\lambda k\ v\ s.\ s \wedge k \in Set\ t2)\ t1\ True$
 $\langle proof \rangle$

lemma *card-Set* [code]:

$card\ (Set\ t) = fold-keys\ (\lambda -. n.\ n + 1)\ t\ 0$
 $\langle proof \rangle$

lemma *sum-Set* [code]:

$sum\ f\ (Set\ xs) = fold-keys\ (plus \circ f)\ xs\ 0$
 $\langle proof \rangle$

lemma *prod-Set* [code]:

$prod\ f\ (Set\ xs) = fold-keys\ (times \circ f)\ xs\ 1$
 $\langle proof \rangle$

lemma *the-elem-set* [code]:

fixes $t :: ('a :: linorder,\ unit)\ rbt$
shows $the-elem\ (Set\ t) = (case\ RBT.impl-of\ t\ of$
 $(Branch\ RBT-Impl.B\ RBT-Impl.Empty\ x\ ()\ RBT-Impl.Empty) \Rightarrow x$
 $\mid - \Rightarrow Code.abort\ (STR\ "not-a-singleton-tree")\ (\lambda -. the-elem\ (Set\ t)))$
 $\langle proof \rangle$

lemma *Pow-Set* [code]: $Pow\ (Set\ t) = fold-keys\ (\lambda x\ A.\ A \cup Set.insert\ x\ 'A)\ t\ \{\{\}$
 $\{\{\}\}$
 $\langle proof \rangle$

lemma *product-Set* [code]:

Product-Type.product (*Set t1*) (*Set t2*) =
fold-keys ($\lambda x A. \text{fold-keys } (\lambda y. \text{Set.insert } (x, y)) \text{ t2 } A$) *t1* {}
 ⟨proof⟩

lemma *Id-on-Set* [code]: *Id-on* (*Set t*) = *fold-keys* ($\lambda x. \text{Set.insert } (x, x)$) *t* {}
 ⟨proof⟩

lemma *Image-Set* [code]:

(*Set t*) “*S* = *fold-keys* ($\lambda(x,y) A. \text{if } x \in S \text{ then Set.insert } y A \text{ else } A$) *t* {}
 ⟨proof⟩

lemma *tranc1-set-ntranc1* [code]:

tranc1 (*Set t*) = *ntranc1* (*card* (*Set t*) − 1) (*Set t*)
 ⟨proof⟩

lemma *relcomp-Set*[code]:

(*Set t1*) *O* (*Set t2*) = *fold-keys*
 ($\lambda(x,y) A. \text{fold-keys } (\lambda(w,z) A'. \text{if } y = w \text{ then Set.insert } (x,z) A' \text{ else } A') \text{ t2 } A$)
t1 {}
 ⟨proof⟩

lemma *wf-set*: *wf* (*Set t*) = *acyclic* (*Set t*)
 ⟨proof⟩

lemma *wf-code-set*[code]: *wf-code* (*Set t*) = *acyclic* (*Set t*)
 ⟨proof⟩

lemma *Min-fin-set-fold* [code]:

Min (*Set t*) =
 (if *RBT.is-empty t*
 then *Code.abort* (*STR* “not-non-empty-tree”) ($\lambda-. \text{Min } (\text{Set } t)$)
 else *r-min-opt t*)
 ⟨proof⟩

lemma *Inf-fin-set-fold* [code]:

Inf-fin (*Set t*) = *Min* (*Set t*)
 ⟨proof⟩

lemma *Inf-Set-fold*:

fixes *t* :: (*a* :: {*linorder*, *complete-lattice*}, *unit*) *rbt*
shows *Inf* (*Set t*) = (if *RBT.is-empty t* then *top* else *r-min-opt t*)
 ⟨proof⟩

lemma *Max-fin-set-fold* [code]:

Max (*Set t*) =
 (if *RBT.is-empty t*
 then *Code.abort* (*STR* “not-non-empty-tree”) ($\lambda-. \text{Max } (\text{Set } t)$)

else r-max-opt t)
 ⟨*proof*⟩

lemma *Sup-fin-set-fold* [code]:
 Sup-fin (Set t) = Max (Set t)
 ⟨*proof*⟩

lemma *Sup-Set-fold*:
 fixes *t :: ('a :: {linorder, complete-lattice}, unit) rbt*
 shows *Sup (Set t) = (if RBT.is-empty t then bot else r-max-opt t)*
 ⟨*proof*⟩

context
begin

qualified definition *Inf' :: 'a :: {linorder, complete-lattice} set ⇒ 'a*
 where [code-abbrev]: *Inf' = Inf*

lemma *Inf'-Set-fold* [code]:
 Inf' (Set t) = (if RBT.is-empty t then top else r-min-opt t)
 ⟨*proof*⟩ **definition** *Sup' :: 'a :: {linorder, complete-lattice} set ⇒ 'a*
 where [code-abbrev]: *Sup' = Sup*

lemma *Sup'-Set-fold* [code]:
 Sup' (Set t) = (if RBT.is-empty t then bot else r-max-opt t)
 ⟨*proof*⟩

end

lemma [code]:
 Gcd_{fin} (Set t) = fold-keys gcd t (0::'a::{semiring-gcd, linorder})
 ⟨*proof*⟩

lemma [code]:
 Gcd (Set t) = (Gcd_{fin} (Set t) :: nat)
 ⟨*proof*⟩

lemma [code]:
 Gcd (Set t) = (Gcd_{fin} (Set t) :: int)
 ⟨*proof*⟩

lemma [code]:
 Lcm_{fin} (Set t) = fold-keys lcm t (1::'a::{semiring-gcd, linorder})
 ⟨*proof*⟩

lemma [code]:
 Lcm (Set t) = (Lcm_{fin} (Set t) :: nat)
 ⟨*proof*⟩

```

lemma [code]:
   $Lcm (Set\ t) = (Lcm_{fin} (Set\ t) :: int)$ 
   $\langle proof \rangle$ 

lemma sorted-list-set [code]:  $sorted\text{-}list\text{-}of\text{-}set (Set\ t) = RBT.keys\ t$ 
   $\langle proof \rangle$ 

lemma Least-code [code]:
   $\langle Lattices\text{-}Big.Least (Set\ t) = (if\ RBT.is\ empty\ t\ then\ Lattices\text{-}Big.Least\ abort\ \{\}\$ 
   $else\ Min (Set\ t)) \rangle$ 
   $\langle proof \rangle$ 

lemma Greatest-code [code]:
   $\langle Lattices\text{-}Big.Greatest (Set\ t) = (if\ RBT.is\ empty\ t\ then\ Lattices\text{-}Big.Greatest\ abort\$ 
   $\{\}\ else\ Max (Set\ t)) \rangle$ 
   $\langle proof \rangle$ 

lemma [code]:
   $\langle Option.these\ A = the\ 'Set.filter\ (Not \circ Option.is\ none)\ A \rangle$ 
   $\langle proof \rangle$ 

lemma [code]:
   $\langle Option.image\ filter\ f\ A = Option.these\ (image\ f\ A) \rangle$ 
   $\langle proof \rangle$ 

lemma [code]:
   $\langle Set.can\ select\ P\ A = is\ singleton\ (Set.filter\ P\ A) \rangle$ 
   $\langle proof \rangle$ 

declare [[code drop:
   $\langle Inf :: - \Rightarrow 'a\ set \rangle$ 
   $\langle Sup :: - \Rightarrow 'a\ set \rangle$ 
   $\langle Inf :: - \Rightarrow 'a\ Predicate.pred \rangle$ 
   $\langle Sup :: - \Rightarrow 'a\ Predicate.pred \rangle$ 
   $pred\text{-}of\text{-}set$ 
   $Wellfounded.acc$ 
]]

hide-const (open)  $RBT\text{-}Set.Set\ RBT\text{-}Set.Coset$ 

end

theory Time-Manual
imports HOL-Library.Time-Commands
begin

```

136 Introduction

This manual describes the framework for the automatic definition of step-counting ‘running-time’ functions from HOL functions. The principles of the translation are described in Section 1.5, Running Time, of the book Functional Data Structures and Algorithms. A Proof Assistant Approach. <https://fdsa-book.net> To load the framework import *HOL-Library.Time-Commands*. The framework was implemented by Jonas Stahl.

As a first simple example consider *len*, which we define here returning an *int* (to distinguish it from the time functions returning *nat*):

```
fun len :: 'a list  $\Rightarrow$  int where
  len [] = 0 |
  len (x#xs) = 1 + len xs
```

time-fun len

Command *time-fun* defines a new function *T-len* of type *'a list \Rightarrow nat*, the time function for *len* that counts the number of computation steps. The definition is printed by *time-fun*: *fun T-len :: 'a list \Rightarrow nat where T-len [] = 1 | T-len (x # xs) = T-len xs + 1* The details of this translation are described in the book referenced above. This manual is about the use of the time framework.

Command *time-fun f* retrieves the definition of *f* and defines a corresponding step-counting running-time function *T-f*. For all auxiliary functions used by *f* (excluding constructors and predefined functions (see below)), running time functions must already have been defined. Example:

```
fun aux :: 'a  $\Rightarrow$  'a where
  aux x = x
```

time-fun aux

```
fun main :: bool  $\Rightarrow$  bool where
  main x = aux x
```

time-fun main

For functions defined by *definition*, there is a corresponding *time-definition* command. Example:

```
definition gdef :: 'a  $\Rightarrow$  'a where gdef x = x
```

time-definition gdef

thm T-gdef.simps

Note that *T-gdef* is defined via *fun*, which means that the defining equation is not named *T-gdef-def* but *T-gdef.simps* and is a simp-rule.

The time functions for many standard functions (in particular on lists) are already defined in theory *HOL–Library.Time-Functions* and basic upper bounds are proved.

137 Termination

If the definition of a recursive function requires a manual termination proof, use *time-function* accompanied by a *termination* command.

```
function sum-to :: int ⇒ int ⇒ int where
  sum-to i j = (if j ≤ i then 0 else i + sum-to (i+1) j)
  ⟨proof⟩
termination
  ⟨proof⟩

time-function sum-to
termination
  ⟨proof⟩
```

138 Partial Functions

Partial functions can also be ‘timed’.

```
partial-function (tailrec) positive :: int ⇒ bool where
  positive i = (if i = 1 then True else positive (i−1))
```

```
time-partial-function positive
```

The difference is that *T-positive* has return type *nat option* because *positive* may not terminate.

Timing a function defined with *partial-function* (*option*) is trickier and we do not go into it here.

139 Higher-Order Functions

A large subclass of higher-order functions are supported, covering *map*, *filter* and other standard functions. For example,

```
time-fun map
```

defines a time function *T-map* :: (*'a* ⇒ *nat*) ⇒ *'a list* ⇒ *nat*. The first argument (called *T-f* below) is the time function for the first argument *f* of *map*. We ignore the definition of *T-map* because the output of *time-fun map* suggests that you should add these lemmas

```
lemma T-map-simps [simp,code]:
  T-map T-f [] = 1
  T-map T-f (x # xs) = T-f x + T-map T-f xs + 1
```

<proof>

which are what you would expect as defining equations. You can click on the suggestion to have it copied into your theory. Afterwards, you can work with *T-map* as if it were defined via those equations.

In general, things are a bit more complicated, which is why *T-map* is defined the way it is. Consider

```
fun foldl :: ('b ⇒ 'a ⇒ 'b) ⇒ 'b ⇒ 'a list ⇒ 'b where
foldl f a [] = a |
foldl f a (x # xs) = foldl f (f a x) xs
```

time-fun foldl

This definition is generated:

```
fun T-foldl :: ('b ⇒ 'a ⇒ 'b) × ('b ⇒ 'a ⇒ nat) ⇒ 'b ⇒ 'a list ⇒ nat
where T-foldl (f, T-f) a [] = 1 | T-foldl (f, T-f) a (x # xs) = T-f a x +
T-foldl f (f a x) xs + 1
```

The meaning of the pair $(f, T-f)$ is obvious. The difference to *T-map* is that *T-foldl* needs not just *T-f* (like *T-map*) but also *f*. Function *T-map* does not need *f*: in the recursion equation $\text{map } f (x \# xs) = f x \# \text{map } f xs$ the result of subterm $f x$ is irrelevant for the computation of *T-map* because the running time of $(\#)$ is constant. This is in contrast to *foldl*, whose running time may depend on its second argument.

All higher-order functions are translated like *foldl*, but if the first element in $(f, T-f)$ is unused, a simplified definition is derived. This is the case for *T-map*.

In case you wonder how it is ensured that *T-foldl* is always passed a corresponding pair of a function and its timing function: this is the responsibility of the time framework when translating functions that use *foldl*. Example:

```
definition inc :: int ⇒ 'a ⇒ int where inc i x = i+1
definition len2 xs = foldl inc 0 xs
time-definition inc
time-definition len2
```

In the defining equation $T\text{-len2 } xs = T\text{-foldl } (inc, T\text{-inc}) 0 xs$ we find the correct pair $(inc, T\text{-inc})$.

139.1 Limitations

Partial application and lambda-abstraction are currently not supported. They need to be replaced by additional function definitions, if possible. For example,

```
definition fHO :: bool list ⇒ bool list where fHO = map (λx. x ∧ x)
```

is not acceptable (i.e. *time-definition* *fHO* fails), but can be replaced with

definition $\text{double} :: \text{int} \Rightarrow \text{int}$ **where** $\langle \text{double } i = 2 * i \rangle$

definition $\text{fHO}' :: \text{int list} \Rightarrow \text{int list}$ **where** $\langle \text{fHO}' xs = \text{map double } xs \rangle$

time-definition double

time-definition fHO'

That is why in the definition of *len2* above we could not just write *foldl* $(\lambda i x. i+1) 0 xs$.

140 Predefined Functions

The time framework requires executable functions. However, many basic types and functions are not defined via *datatype* and *fun* but in an abstract mathematical fashion and are not executable, i.e. the time framework does not apply (or gives the ‘wrong’ result).

In order to model actual hardware that executes these predefined functions in constant time, there is a command for axiomatically declaring that some function takes 0 time. (This is how we model constant time, to simplify the resulting time expressions. This does not change the asymptotic running time of user-defined functions using the predefined functions because 1 is added for every user-defined function call.) Theory *HOL-Library.Time-Commands* declares a number of predefined functions as 0-time functions. This includes $(+)$, $-$, $(*)$, $/$, *div*, *min*, *max*, $<$, \leq , \neg , \wedge , \vee and $=$ and can be extended with the command *time-fun-0*. This feature has to be used with care:

- Many of these functions are polymorphic and reside in type classes. The constant-time assumption is justified only for those types where the hardware offers suitable support, e.g. numeric types. The argument size is implicitly bounded, too.
- The constant-time assumption for $(=)$ is justified for recursive data types such as lists and trees as long as the comparison is of the form $t = c$ where c is a constant term, for example $xs = []$.

Users of the time framework need to ensure that 0-time functions are used only within these bounds.

141 Locales

If we want to apply the time framework to a function g defined within a locale, we need to add additional locale parameters $T\text{-}f :: \tau \Rightarrow \text{nat}$ for every locale parameter $f :: \tau \Rightarrow \tau'$ used in the definition of g .

In the following example we do not only parameterize the locale with $T\text{-}f$ but also assume a property of $T\text{-}f$. As a result we can prove a property of $T\text{-}g$ inside the locale:

lemma *T-map-sum*: $T\text{-map } T\text{-f } xs = \text{sum-list } (\text{map } T\text{-f } xs) + \text{length } xs + 1$
 $\langle \text{proof} \rangle$

locale *LT* =
fixes $f :: 'a \Rightarrow 'a$
and $T\text{-f} :: 'a \Rightarrow \text{nat}$
assumes $T\text{-f}: T\text{-f } x \leq 1$
begin

definition *g* **where** $g\ xs = \text{map } f\ xs$

time-definition *g*

lemma *sum-list-map-T-f-ub*: $\text{sum-list } (\text{map } T\text{-f } xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *T-g-ub*: $T\text{-g } xs \leq 2 * \text{length } xs + 1$
 $\langle \text{proof} \rangle$

end

Of course now you need to prove $T\text{-f } x \leq 1$ for every interpretation of the locale. A more flexible approach is not to constrain $T\text{-f}$ inside the locale. It may then be difficult to derive a generic time bound for $T\text{-g}$ inside the locale (in the above example it would not be difficult). If that is the case, one may also derive a bound for $T\text{-g}$ conditional on some specific bound for $T\text{-f}$. Or one can derive the bound for $T\text{-g}$ after a specific interpretation with a specific $T\text{-f}$. For a larger realistic example of the latter approach see theory *HOL-Data-Structures.Time-Locale-Example*.

142 Fine Points

Time functions for mutually recursive functions f, g, \dots : *time-fun f g ...*

If you want to generate time functions not from the defining equations of a function but from lemmas proved as equations, you can provide those lemmas explicitly. Example:

fun $f0 :: \text{nat} \Rightarrow \text{nat}$ **where**
 $f0\ 0 = 0 \mid$
 $f0\ (\text{Suc } n) = f0\ n$

lemma *f0-eq*: $f0\ n = 0$
 $\langle \text{proof} \rangle$

time-fun *f0* **equations** *f0-eq*

The T -prefix can be changed by modifying the *time-prefix* attribute. Example:

declare $[[time-prefix = t-]]$

The time framework is not verified (which is why the framework always prints out what it defines). There is no underlying formal model. This remains future work.

end

$\langle ML \rangle$

theory *Suc-Notation*

imports *Main*

begin

Nested *Suc* terms of depth $2 \leq n \leq 9$ are abbreviated with new notations Suc^n :

abbreviation $(input) Suc2$ **where** $Suc2\ n \equiv Suc\ (Suc\ n)$

abbreviation $(input) Suc3$ **where** $Suc3\ n \equiv Suc\ (Suc2\ n)$

abbreviation $(input) Suc4$ **where** $Suc4\ n \equiv Suc\ (Suc3\ n)$

abbreviation $(input) Suc5$ **where** $Suc5\ n \equiv Suc\ (Suc4\ n)$

abbreviation $(input) Suc6$ **where** $Suc6\ n \equiv Suc\ (Suc5\ n)$

abbreviation $(input) Suc7$ **where** $Suc7\ n \equiv Suc\ (Suc6\ n)$

abbreviation $(input) Suc8$ **where** $Suc8\ n \equiv Suc\ (Suc7\ n)$

abbreviation $(input) Suc9$ **where** $Suc9\ n \equiv Suc\ (Suc8\ n)$

notation $Suc2\ (Suc^2)$

notation $Suc3\ (Suc^3)$

notation $Suc4\ (Suc^4)$

notation $Suc5\ (Suc^5)$

notation $Suc6\ (Suc^6)$

notation $Suc7\ (Suc^7)$

notation $Suc8\ (Suc^8)$

notation $Suc9\ (Suc^9)$

Beyond 9, the syntax Suc^n kicks in:

syntax

$-Suc-tower :: num-token \Rightarrow nat \Rightarrow nat\ (Suc^-)$

$\langle ML \rangle$

end

theory *Predicate-Compile-Alternative-Defs*

imports *Main*

begin

143 Common constants

declare *HOL.if-bool-eq-disj*[*code-pred-inline*]

declare *bool-diff-def*[*code-pred-inline*]

declare *inf-bool-def*[*abs-def*, *code-pred-inline*]

declare *less-bool-def*[*abs-def*, *code-pred-inline*]

declare *le-bool-def*[*abs-def*, *code-pred-inline*]

lemma *min-bool-eq* [*code-pred-inline*]: $(\min :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}) == (\wedge)$
 $\langle \text{proof} \rangle$

lemma [*code-pred-inline*]:
 $((A :: \text{bool}) \neq (B :: \text{bool})) = ((A \wedge \neg B) \vee (B \wedge \neg A))$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

144 Pairs

$\langle ML \rangle$

145 Filters

$\langle ML \rangle$

146 Bounded quantifiers

declare *Ball-def*[*code-pred-inline*]

declare *Bex-def*[*code-pred-inline*]

147 Operations on Predicates

lemma *Diff*[*code-pred-inline*]:
 $(A - B) = (\%x. A\ x \wedge \neg B\ x)$
 $\langle \text{proof} \rangle$

lemma *subset-eq*[*code-pred-inline*]:
 $(P :: 'a \Rightarrow \text{bool}) < (Q :: 'a \Rightarrow \text{bool}) \equiv ((\exists x. Q\ x \wedge (\neg P\ x)) \wedge (\forall x. P\ x \longrightarrow Q\ x))$
 $\langle \text{proof} \rangle$

lemma *set-equality*[*code-pred-inline*]:
 $A = B \longleftrightarrow (\forall x. A\ x \longrightarrow B\ x) \wedge (\forall x. B\ x \longrightarrow A\ x)$
 $\langle \text{proof} \rangle$

148 Setup for Numerals

$\langle ML \rangle$

149 Arithmetic operations

149.1 Arithmetic on naturals and integers

definition *plus-eq-nat* :: *nat* => *nat* => *nat* => *bool*

where

plus-eq-nat *x y z* = (*x* + *y* = *z*)

definition *minus-eq-nat* :: *nat* => *nat* => *nat* => *bool*

where

minus-eq-nat *x y z* = (*x* − *y* = *z*)

definition *plus-eq-int* :: *int* => *int* => *int* => *bool*

where

plus-eq-int *x y z* = (*x* + *y* = *z*)

definition *minus-eq-int* :: *int* => *int* => *int* => *bool*

where

minus-eq-int *x y z* = (*x* − *y* = *z*)

definition *subtract*

where

[*code-unfold*]: *subtract* *x y* = *y* − *x*

$\langle ML \rangle$

149.2 Inductive definitions for ordering on naturals

inductive *less-nat*

where

less-nat 0 (*Suc y*)

| *less-nat* *x y* ==> *less-nat* (*Suc x*) (*Suc y*)

lemma *less-nat*[*code-pred-inline*]:

x < *y* = *less-nat* *x y*

$\langle proof \rangle$

inductive *less-eq-nat*

where

less-eq-nat 0 *y*

| *less-eq-nat* *x y* ==> *less-eq-nat* (*Suc x*) (*Suc y*)

lemma [*code-pred-inline*]:

x <= *y* = *less-eq-nat* *x y*

$\langle proof \rangle$

150 Alternative list definitions

150.1 Alternative rules for *length*

definition *size-list'* :: 'a list => nat
where *size-list'* = *size*

lemma *size-list'-simps*:
 $\text{size-list}' [] = 0$
 $\text{size-list}' (x \# xs) = \text{Suc} (\text{size-list}' xs)$
 <proof>

declare *size-list'-simps*[*code-pred-def*]
declare *size-list'-def*[*symmetric*, *code-pred-inline*]

150.2 Alternative rules for *list-all2*

lemma *list-all2-NilI* [*code-pred-intro*]: *list-all2* *P* [] []
 <proof>

lemma *list-all2-ConsI* [*code-pred-intro*]: *list-all2* *P* *xs* *ys* ==> *P* *x* *y* ==> *list-all2*
P (*x* # *xs*) (*y* # *ys*)
 <proof>

code-pred [*skip-proof*] *list-all2*
 <proof>

150.3 Alternative rules for membership in lists

lemma *in-set-member* [*code-pred-inline*]:
 $x \in \text{set } xs \longleftrightarrow \text{List.member } xs \ x$
 <proof>

lemma *member-intros* [*code-pred-intro*]:
 $\text{List.member } (x \# xs) \ x$
 $\text{List.member } xs \ x \implies \text{List.member } (y \# xs) \ x$
 <proof>

code-pred *List.member*
 <proof>

code-identifier constant *member-i-i*
 $\rightarrow (\text{SML}) \ \text{List.member-i-i}$
and (*OCaml*) *List.member-i-i*
and (*Haskell*) *List.member-i-i*
and (*Scala*) *List.member-i-i*

code-identifier constant *member-i-o*
 $\rightarrow (\text{SML}) \ \text{List.member-i-o}$
and (*OCaml*) *List.member-i-o*


```

and (Haskell) List.member-i-o
and (Scala) List.member-i-o

```

151 Setup for String.literal

⟨ML⟩

152 Simplification rules for optimisation

```

lemma [code-pred-simp]:  $\neg \text{False} == \text{True}$ 
  ⟨proof⟩

```

```

lemma [code-pred-simp]:  $\neg \text{True} == \text{False}$ 
  ⟨proof⟩

```

```

lemma less-nat-k-0 [code-pred-simp]: less-nat k 0 == False
  ⟨proof⟩

```

end

153 A Prototype of Quickcheck based on the Predicate Compiler

```

theory Predicate-Compile-Quickcheck
  imports Predicate-Compile-Alternative-Defs
begin

```

⟨ML⟩

end

154 TFL: recursive function definitions

```

theory Old-Recdef
imports Main
keywords
  recdef :: thy-defn and
  permissive congs hints
begin

```

154.1 Lemmas for TFL

```

lemma tfl-wf-induct:  $\forall R. \text{wf } R \longrightarrow$ 
   $(\forall P. (\forall x. (\forall y. (y,x) \in R \longrightarrow P y) \longrightarrow P x) \longrightarrow (\forall x. P x))$ 
  ⟨proof⟩

```

```

lemma tfl-cut-def: cut f r x  $\equiv (\lambda y. \text{if } (y,x) \in r \text{ then } f y \text{ else undefined})$ 

```

$\langle proof \rangle$

lemma *tfl-cut-apply*: $\forall f R. (x, a) \in R \longrightarrow (cut\ f\ R\ a)(x) = f(x)$
 $\langle proof \rangle$

lemma *tfl-wfrec*:
 $\forall M R f. (f = wfrec\ R\ M) \longrightarrow wf\ R \longrightarrow (\forall x. f\ x = M\ (cut\ f\ R\ x)\ x)$
 $\langle proof \rangle$

lemma *tfl-eq-True*: $(x = True) \longrightarrow x$
 $\langle proof \rangle$

lemma *tfl-rev-eq-mp*: $(x = y) \longrightarrow y \longrightarrow x$
 $\langle proof \rangle$

lemma *tfl-simp-thm*: $(x \longrightarrow y) \longrightarrow (x = x') \longrightarrow (x' \longrightarrow y)$
 $\langle proof \rangle$

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
 $\langle proof \rangle$

lemma *tfl-imp-trans*: $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$
 $\langle proof \rangle$

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
 $\langle proof \rangle$

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$
 $\langle proof \rangle$

lemma *tfl-exE*: $\exists x. P\ x \Longrightarrow \forall x. P\ x \longrightarrow Q \Longrightarrow Q$
 $\langle proof \rangle$

$\langle ML \rangle$

154.2 Rule setup

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong *let-cong* *image-cong* *INF-cong* *SUP-cong* *bex-cong* *ball-cong* *imp-cong*
map-cong *filter-cong* *takeWhile-cong* *dropWhile-cong* *foldl-cong* *foldr-cong*

lemmas [*recdef-wf*] =

```

wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-measures
wf-pred-nat
wf-same-fst
wf-on-bot

```

```
end
```

155 Program extraction from proofs involving datatypes and inductive predicates

```

theory Realizers
imports Main
begin

```

```

⟨ML⟩

```

```
end
```

156 Refute

```

theory Refute
imports Main
keywords
  refute :: diag and
  refute-params :: thy-decl
begin

```

```

⟨ML⟩

```

```

refute-params
[itself = 1,
 minsize = 1,
 maxsize = 8,
 maxvars = 10000,
 maxtime = 60,
 satsolver = auto,
 no-assms = false]

```

```

(* ----- *)
(* REFUTE *)
(* *)
(* We use a SAT solver to search for a (finite) model that refutes a given *)

```

```

(* HOL formula. *)
(* ----- *)

(* ----- *)
(* NOTE *)
(* *)
(* I strongly recommend that you install a stand-alone SAT solver if you *)
(* want to use 'refute'. For details see 'HOL/Tools/sat_solver.ML'. If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME' *)
(* in 'etc/settings'. *)
(* ----- *)

(* ----- *)
(* USAGE *)
(* *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples. The supported *)
(* parameters are explained below. *)
(* ----- *)

(* ----- *)
(* CURRENT LIMITATIONS *)
(* *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and *)
(* inductively defined sets. Constants are unfolded automatically, and sort *)
(* axioms are added as well. Other, user-asserted axioms however are *)
(* ignored. Inductive datatypes and recursive functions are supported, but *)
(* may lead to spurious countermodels. *)
(* *)
(* The (space) complexity of the algorithm is non-elementary. *)
(* *)
(* Schematic type variables are not supported. *)
(* ----- *)

(* ----- *)
(* PARAMETERS *)
(* *)
(* The following global parameters are currently supported (and required, *)
(* except for "expect"): *)
(* *)
(* Name          Type    Description *)
(* *)
(* "minsize"      int     Only search for models with size at least *)
(*                  'minsize'. *)
(* *)
(* "maxsize"      int     If >0, only search for models with size at most *)
(*                  'maxsize'. *)
(* *)
(* "maxvars"      int     If >0, use at most 'maxvars' boolean variables *)
(*                  when transforming the term into a propositional *)

```

```

(*)          formula.                                     *)
(*) "maxtime"  int    If >0, terminate after at most 'maxtime' seconds. *)
(*)          This value is ignored under some ML compilers. *)
(*) "satsolver" string Name of the SAT solver to be used. *)
(*) "no_assms" bool   If "true", assumptions in structured proofs are *)
(*)          not considered. *)
(*) "expect"   string Expected result ("genuine", "potential", "none", or *)
(*)          "unknown"). *)
(*) *)
(*) The size of particular types can be specified in the form type=size *)
(*) (where 'type' is a string, and 'size' is an int).  Examples: *)
(*) "'a'=1 *)
(*) "List.list"=2 *)
(*) ----- *)

(*) ----- *)
(*) FILES *)
(*) *)
(*) HOL/Tools/prop_logic.ML      Propositional logic *)
(*) HOL/Tools/sat_solver.ML      SAT solvers *)
(*) HOL/Tools/refute.ML          Translation HOL -> propositional logic and *)
(*)                               Boolean assignment -> HOL model *)
(*) HOL/Refute.thy               This file: loads the ML files, basic setup, *)
(*)                               documentation *)
(*) HOL/SAT.thy                  Sets default parameters *)
(*) HOL/ex/Refute_Examples.thy   Examples *)
(*) ----- *)

end

```

References

- [1] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009, 2010.
- [2] D. Leijen. Division and modulus for computer scientists. 2001.
- [3] A. Lochbihler and P. Stoop. Lazy algebraic types in Isabelle/HOL. In *Isabelle Workshop 2018*, 2018.
- [4] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.