

Software Foundations: SLF

August 4, 2026

Contents

1	Library <code>SLF.LibAxioms</code>	2
1.1	Functional extensionality	2
1.2	Propositional extensionality	2
1.3	Indefinite description	3
2	Library <code>SLF.LibTactics</code>	4
2.1	Fixing Stdlib	5
2.2	Tools for Programming with Ltac	5
2.2.1	Identity Continuation	5
2.2.2	Untyped Arguments for Tactics	5
2.2.3	Optional Arguments for Tactics	5
2.2.4	Wildcard Arguments for Tactics	6
2.2.5	Position Markers	6
2.2.6	List of Arguments for Tactics	7
2.2.7	Databases of Lemmas	9
2.2.8	On-the-Fly Removal of Hypotheses	10
2.2.9	Numbers as Arguments	11
2.2.10	Testing Tactics	12
2.2.11	Testing evars and non-evars	13
2.2.12	Check No Evar in Goal	13
2.2.13	Helper Function for Introducing Evars	13
2.2.14	Tagging of Hypotheses	14
2.2.15	More Tagging of Hypotheses	14
2.2.16	Deconstructing Terms	14
2.2.17	Action at Occurence and Action Not at Occurence	15
2.2.18	An Alias for <code>eq</code>	15
2.3	Check as a tactic instead of a top-level command, with possibility to turn off	16
2.4	Common Tactics for Simplifying Goals Like <code>intuition</code>	16
2.5	Backward and Forward Chaining	17
2.5.1	Application	17
2.5.2	Assertions	19
2.5.3	Instantiation and Forward-Chaining	21
2.5.4	Experimental Tactics for Application	31

2.5.5	Adding Assumptions	32
2.5.6	Application of Tautologies	32
2.5.7	Application Modulo Equalities	32
2.5.8	Absurd Goals	35
2.6	Introduction and Generalization	37
2.6.1	Introduction	37
2.6.2	Introduction using $\Rightarrow$ and $=\gg$	39
2.6.3	Generalization	41
2.6.4	Naming	42
2.7	Rewriting	45
2.7.1	Renaming	48
2.7.2	Unfolding	49
2.7.3	Simplification	51
2.7.4	Reduction	52
2.7.5	Substitution	52
2.7.6	Tactics to Work with Proof Irrelevance	53
2.7.7	Proving Equalities	53
2.8	Inversion	55
2.8.1	Basic Inversion	55
2.8.2	Inversion with Substitution	56
2.8.3	Inversion with Pattern Matching on Head of Hypothesis	60
2.8.4	Injection with Substitution	60
2.8.5	Inversion and Injection with Substitution –rough implementation	61
2.8.6	Case Analysis	62
2.9	Induction	65
2.10	Coinduction	67
2.11	Decidable Equality	68
2.12	Equivalence	68
2.13	N-ary Conjunctions and Disjunctions	69
2.14	Tactics to Prove Typeclass Instances	74
2.15	Tactics to Invoke Automation	75
2.15.1	Definitions for Parsing Compatibility	75
2.15.2	<i>hint</i> to Add Hints Local to a Lemma	75
2.15.3	<i>jauto</i> , a New Automation Tactic	75
2.15.4	Definitions of Automation Tactics	76
2.15.5	Parsing for Light Automation	78
2.15.6	Parsing for Strong Automation	87
2.16	Tactics to Sort Out the Proof Context	96
2.16.1	Hiding Hypotheses	96
2.16.2	Sorting Hypotheses	98
2.16.3	Clearing Hypotheses	98
2.17	Tactics for Development Purposes	100

2.17.1	Skipping Subgoals	100
2.18	Compatibility with standard library	102
2.19	Experimental features	103
2.19.1	'let Bindings (EXPERIMENTAL).	103
3	Library SLF.LibEqual	107
3.1	Definition of equality	107
3.1.1	Definition of Leibnitz' equality	107
3.1.2	Partial application of Leibnitz' equality	107
3.1.3	Typeclass to exploit extensionality	108
3.1.4	Tactic to exploit extensionality	108
3.2	Properties of equality	108
3.2.1	Equality as an equivalence relation	109
3.2.2	Properties of disequality	110
3.2.3	Symmetrized induction principles	110
3.3	Functional extensionality	110
3.3.1	Dependent functional extensionality	110
3.3.2	Non-dependent functional extensionality	112
3.3.3	Eta-conversion	113
3.4	Predicate extensionality	113
3.4.1	Dependent predicates	113
3.4.2	Non-dependent predicate extensionality	115
3.5	Equality of function and predicate applications	115
3.5.1	A same function applied to equal arguments yield equal result	115
3.5.2	Equal functions applied to same arguments return equal results	116
3.5.3	Equal predicates applied to same arguments return equivalent results	117
3.6	Proof Irrelevance	118
3.6.1	Proof of the proof-irrelevance result	118
3.6.2	Consequences of proof irrelevance	119
3.6.3	Injectivity of equality on dependent pairs	120
3.7	Dependent equality	122
3.8	John Major's equality	123
4	Library SLF.LibLogic	125
4.1	Strong existentials	125
4.2	Inhabited types	125
4.2.1	Typeclass for describing inhabited types	125
4.2.2	Tactics taking into account	126
4.2.3	Arbitrary values for inhabited types	126
4.2.4	Lemmas about inhabited types	126
4.3	Non-constructive conditionals	126
4.3.1	Excluded middle	126
4.3.2	Strong excluded middle	127

4.3.3	If-then-else on propositions	127
4.3.4	Consequences	128
4.3.5	Tactic for proving tautologies by bruteforce case-analysis	129
4.4	Properties of logical combinators	130
4.4.1	Simplification of conjunction and disjunction	130
4.4.2	Distribution of negation on basic operators	131
4.4.3	Double negation and contrapose	132
4.4.4	Distribution of negation on quantifiers	133
4.4.5	Propositions equal to True or False	134
4.4.6	Disequal propositions	135
4.4.7	Peirce rule and similar	136
4.4.8	Properties of logical equivalence	136
4.5	Tactics	138
4.5.1	Tactic for simplifying expressions	138
4.5.2	Tactic <i>tests: P</i> for classical disjunction on <i>P</i>	139
4.5.3	Tactic <i>case_classic</i>	140
4.5.4	Tactic <i>absurds</i>	141
4.6	Predicate combinators, comparison and compatibility	141
4.6.1	Definition of predicate combinators	141
4.6.2	Properties of combinators	142
4.6.3	Order on predicates	142
4.7	Existentials	142
4.8	Properties of unique existentials	142
4.9	Conjunctions	144
4.9.1	Changing the order of branches	144
4.9.2	Parallel strengthening of a conjunction	144
4.9.3	Projections of lemmas concluding on a conjunction	145
5	Library SLF.LibOperation	148
5.1	Definitions	148
5.1.1	Commutativity, associativity	148
5.1.2	Distributivity	148
5.1.3	Neutral and absorbant	149
5.1.4	Inverses	150
5.1.5	Morphism and automorphism	150
5.1.6	Injectivity	150
5.2	Lemmas	150
6	Library SLF.LibBool	152
6.1	Boolean type	152
6.1.1	Definition	152
6.1.2	Inhabited	152
6.1.3	Extensionality	152

6.2	Boolean Operations	152
6.2.1	Comparison	152
6.3	Boolean Decision Procedure : tactic working by exponential case	154
6.3.1	Tactic <i>tautob</i>	154
6.4	Properties of boolean operators	154
6.4.1	Properties of <i>eqb</i>	154
6.4.2	Properties of <i>and</i>	155
6.4.3	Properties of <i>or</i>	155
6.4.4	Properties of <i>neg</i>	156
6.4.5	Properties of <i>if then else</i>	156
6.4.6	Properties of <i>impl</i> and <i>xor</i>	157
6.4.7	Opacity	157
6.5	Tactics	157
6.5.1	Tactic <i>rew_neg_neg</i>	157
6.5.2	Tactic <i>rew_bool</i>	158
7	Library <i>SLF.LibReflect</i>	160
7.1	Reflection between booleans and propositions	160
7.1.1	Translation from booleans into propositions	160
7.1.2	Translation from propositions into booleans	161
7.1.3	Extensionality for boolean equality, stated using <i>istrue</i>	162
7.1.4	Specification of boolean equality	162
7.2	Rewriting rules	163
7.2.1	Rewriting rules for distributing <i>istrue</i>	163
7.2.2	Rewriting rules for distributing <i>isTrue</i>	164
7.2.3	Lemmas for testing booleans	165
7.2.4	Lemmas for normalizing <i>b = true</i> and <i>b = false</i> terms	165
7.3	Tactics	166
7.3.1	Tactics <i>rew_istrue</i> to distribute <i>istrue</i>	166
7.3.2	Tactics <i>rew_isTrue</i> to distribute <i>isTrue</i>	166
7.3.3	Tactics useful for program verification, when reasoning about	167
7.3.4	Tactics extended for reflection	168
8	Library <i>SLF.LibProd</i>	170
8.1	Product type	170
8.1.1	Definition	170
8.1.2	Inhabited	170
8.2	Structure	170
8.3	Operations	171
8.3.1	Definition of projections	171
8.3.2	Notation for projections	171
8.3.3	Currying	172
8.3.4	Uncurrying	172

8.3.5	Uncurrying for relations	173
8.3.6	Inverse projections for relations	173
9	Library <code>SLF.LibSum</code>	176
9.1	Sum type	176
9.1.1	Definition	176
9.1.2	Inhabited	176
9.2	Operations	177
9.2.1	Testing the branch of the sum	177
9.2.2	Stripping of the branch tag	177
9.2.3	Lifting functions over sum types	178
10	Library <code>SLF.LibRelation</code>	179
10.1	Type of binary relations	179
10.1.1	Type of endorelation, i.e. homogeneous binary relations	179
10.1.2	Inhabited	179
10.1.3	Extensionality	179
10.2	Properties of relations	180
10.2.1	Reflexivity	180
10.2.2	Irreflexivity	180
10.2.3	Symmetry	181
10.2.4	Asymmetry	181
10.2.5	Antisymmetry	182
10.2.6	Antisymmetry with respect to an equivalence relation	182
10.2.7	Transitivity	182
10.2.8	Equivalence relation	184
10.2.9	Inclusion	184
10.2.10	Totality	184
10.2.11	Trichotomy	185
10.2.12	Definedness	186
10.2.13	Functionality	186
10.2.14	Criteria for equality	187
10.2.15	Properties of the equality relation	187
10.2.16	Properties of the equality relation	188
10.2.17	Properties of the equivalence relation	188
10.3	Basic constructions	188
10.3.1	The empty relation	188
10.3.2	Union of two relations	189
10.3.3	Intersection of two relations	190
10.3.4	Complement of a relation	190
10.3.5	Inverse of a relation	191
10.3.6	<code>preimage</code>	193
10.3.7	<code>rel_seq</code>	193

10.3.8	Turning a function into a relation.	193
10.4	Products	193
10.4.1	Pointwise product	193
10.4.2	Lexicographical product	194
10.5	Closures	197
10.5.1	Reflexive closure	197
10.5.2	Symmetric closure	200
10.5.3	Reflexive-symmetric closure	202
10.5.4	Transitive closure (R^+), defined as $R \circ R^*$	204
10.5.5	Reflexive-transitive closure (R^*)	208
10.5.6	Symmetric-transitive closure	212
10.5.7	Reflexive-symmetric-transitive closure	214
10.5.8	Relationship between closures	216
10.5.9	Iterated closures	219
10.5.10	Other lemmas – TODO	220
10.5.11	Mixed transitivity between closures	222
10.5.12	Irreflexive restriction of a relation	223
10.5.13	Inclusion of a function in a relation	224
10.5.14	Inclusion of a relation in a function	225
11	Library <code>SLF.LibOrder</code>	226
11.1	Order relation upto an equivalence relation	228
11.2	Definition of order operators	234
11.2.1	Classes and notation for comparison operators	234
11.2.2	The operators <code>ge</code> , <code>lt</code> and <code>gt</code> are deduced from <code>le</code>	235
11.3	Classes for comparison properties	236
11.3.1	Definition of classes	236
11.4	Boolean comparison	247
11.5	Order on relations and on predicates	247
12	Library <code>SLF.LibNat</code>	249
12.1	Nat type	249
12.1.1	Definition	249
12.1.2	Inhabited	249
12.2	Order on natural numbers	250
12.2.1	Definition	250
12.2.2	Translating typeclass instances to Peano relations	250
12.2.3	The <code>nat_maths</code> database is used for registering automation	251
12.2.4	Total order instance	251
12.3	Induction	252
12.4	Simplification lemmas	252
12.4.1	Trivial monoid simplifications	252
12.4.2	Simplification tactic	253

12.5	Executable functions	254
13	Library SLF.LibEpsilon	255
13.1	Definition and specification of Hilbert's epsilon operator	255
13.1.1	Definition of epsilon	255
13.1.2	Lemmas about epsilon	256
13.1.3	(Private) tactic <i>epsilon_find</i>	256
13.1.4	Tactics <i>epsilon_name</i>	257
13.1.5	Tactics to work with <i>epsilon</i>	257
13.2	Construction of a function from a relation, using <i>epsilon</i>	258
14	Library SLF.LibInt	260
14.1	Parsing of integers and operations	260
14.1.1	Notation for type and operation	260
14.1.2	Inhabited type	260
14.1.3	Coercion from nat	260
14.1.4	Order relation	261
14.2	Conversion to natural numbers, for tactic programming	261
14.3	Decision procedure	261
14.3.1	Translation from typeclass order to ZArith, for using <i>lia</i>	262
14.3.2	Hypothesis selection	262
14.3.3	Normalization of arithmetic expressions	264
14.3.4	Setting up the goal for <i>lia</i>	264
14.3.5	Main driver for the set up process to goal <i>lia</i>	265
14.3.6	<i>math</i> tactic restricted to arithmetic goals	265
14.3.7	Elimination of multiplication, to call <i>lia</i>	265
14.3.8	Hint externs for calling <i>math</i> in the hint base <i>maths</i>	266
14.4	Rewriting on arithmetic expressions	268
14.4.1	Rewriting equalities provable by the <i>math</i> tactic	268
14.4.2	Addition and subtraction	268
14.4.3	Simplification tactic	269
14.5	Conversions of operations from nat to <i>int</i> and back	270
14.5.1	Lifting of comparisons from nat to <i>int</i>	270
14.5.2	Lifting of inequalities from nat to <i>int</i>	271
14.5.3	Lifting of operations from nat to <i>int</i>	272
14.5.4	Properties of comparison	272
14.5.5	Absolute function in nat	272
14.5.6	<i>abs</i> distribute on constants and operators	273
14.5.7	Tactic <i>rew_abs_nonneg</i> to normalize expressions involving <i>abs</i>	275
14.5.8	Positive part of an integer. Returns 0 on negative values.	275
14.5.9	<i>to_nat</i> distribute on constants and operators	276
14.5.10	Tactic <i>rew_to_nat_nonneg</i> to normalize expressions involving <i>to_nat</i>	277

15 Library SLF.LibMonoid	278
15.1	278
15.2	278
15.3 Monoids	278
15.4 Structures	278
15.5 Examples	279
15.6 Properties	279
15.7 Derived Properties	279
16 Library SLF.LibContainer	281
16.1 Operators	281
16.1.1 Definitions	281
16.1.2 Notation	282
16.1.3 $\forall x \text{ in } E, P \ x$ notation	282
16.1.4 $\exists x \text{ in } E \text{ st } P \ x$ notation	283
16.1.5 Foreach	283
16.1.6 index for natural numbers	283
16.1.7 Derivable	284
16.1.8 Properties	284
16.1.9 Derived Properties	289
17 Library SLF.LibOption	301
17.1 Option type	301
17.1.1 Definition	301
17.1.2 Inhabited	301
17.2 Operations	301
17.2.1 is_some	301
17.2.2 unsome_default	302
17.2.3 unsome	302
17.2.4 map	302
17.2.5 map_on	303
17.2.6 apply	303
17.2.7 apply_on	303
18 Library SLF.LibWf	304
18.1 Compatibility	304
18.1.1 Tactics	304
18.1.2 Definition	304
18.1.3 Properties	305
18.1.4 Measure on pairs	305
18.2 Construction of well-founded relations	306
18.2.1 Empty relation	306
18.2.2 Inclusion	306

18.2.3	<i>Peano.lt</i> on nat	306
18.2.4	lt on nat	307
18.2.5	The relation “greater than” on the set of	307
18.2.6	The relation “less than” on the set of	307
18.2.7	The relation “greater than” on the set of	308
18.3	Inverse projections	309
18.4	Lexicographical product	311
18.5	Symmetric product	312
18.6	Well-foundedness of a function image	313
18.7	Transitive closure	313
19	Library SLF.LibList	315
19.1	Inhabited	315
19.2	Normalization tactics	316
19.2.1	<i>rew_list</i> for basic list properties	316
19.2.2	<i>rew_listx</i> for all other operations on lists	316
19.3	Properties of operations	317
19.3.1	Definitions of Fold-right and App	317
19.3.2	App	317
19.3.3	FoldRight	319
19.3.4	FoldLeft	319
19.3.5	Length	320
19.4	Inversion lemmas for structural composition	321
19.4.1	nth_default as a partial function with a default	333
19.4.2	nth as a partial function	335
19.4.3	Rev	338
19.4.4	Inversion for rev	341
19.5	Make	341
19.6	Update as a function	343
19.6.1	Map	346
19.6.2	Concat	350
19.6.3	Filter	352
19.6.4	Remove	354
19.6.5	Combine	360
19.6.6	Split	362
19.6.7	Take	364
19.6.8	Drop	366
19.6.9	Take and drop decomposition of a list	368
19.6.10	TakeDropLast	370
19.7	Nat seq	387
19.8	Fold	388

20 Library SLF.LibListExec	394
20.0.1 <code>concat</code>	396
20.0.2 <code>filter</code>	396
20.0.3 <code>remove</code>	397
21 Library SLF.LibListZ	400
21.1 Tactics	400
21.1.1 <code>rew_listp</code> for operations with preconditions	400
21.1.2 <code>rew_index</code> for operations on indices	401
21.1.3 <code>rew_array</code> for case-analysis free rewriting	401
21.1.4 <code>rew_array</code> for case-analysis aware rewriting	402
21.1.5 <code>index_prove</code> for automating proofs on indices	402
21.2 List operations using indices in $\mathbb{Z}$	403
21.3 Length, with result as <i>int</i>	403
21.4 Inversion lemmas for structural composition	404
21.5 <code>index</code> , with length as <i>int</i> , as typeclass	405
21.6 <code>read</code> , with length as <i>int</i> , as typeclass	406
21.7 Equality between two lists from equality of length and	408
21.8 <code>update</code> , with index as <i>int</i> , as typeclass	409
21.9 <code>make</code> , with length as <i>int</i>	413
21.10 <code>LibList.rev</code> interactions with <code>LibListZ</code> operations	415
21.11 <code>LibList.map</code> : <code>map</code> interactions with <code>LibListZ</code> operations	416
21.12 <code>LibList.filter</code> interactions with <code>LibListZ</code> operations	417
21.13 <code>LibList.remove</code> interactions with <code>LibListZ</code> operations	418
21.13.1 Take, with an <i>int</i> as the number of elements	418
21.13.2 Drop, with an <i>int</i> as the number of elements.	421
21.13.3 Take and drop decomposition of a list	423
21.13.4 <code>sum</code> , for lists of int values	425
21.13.5 <code>count</code> , returning an int	426
21.14 <code>card</code> , with result as <code>nat</code> , as typeclass	429
21.15 Hints for <code>rew_array</code> and <code>rew_array_nocase</code>	429
21.16 <code>binds</code> , with length as <i>int</i> , as typeclass	429
22 Library SLF.LibMin	430
23 Library SLF.LibSet	434
23.1 Construction of sets as predicates	434
23.1.1 Basic definitions	434
23.1.2 Notations to help the typechecker	435
23.1.3 Inhabited	436
23.1.4 Notation through typeclasses	436
23.1.5 Notations for building sets	437
23.2 Properties of sets	437

23.3	Tactics for proving set equalities and set inclusions	446
23.4	More properties	448
23.4.1	Structural properties	453
23.5	MORE	453
23.5.1	TEMPORARY Foreach	453
24	Library SLF.LibChoice	456
24.1	Functional choice	456
24.1.1	Functional choice	456
24.1.2	Dependent functional choice	456
24.1.3	Guarded functional choice	457
24.1.4	Omniscient functional choice	457
24.2	Functional unique choice	457
24.2.1	Functional unique choice	458
24.2.2	Dependent functional unique choice	458
24.2.3	Guarded functional unique choice	458
24.2.4	Omniscient functional unique choice	459
24.3	Relational choice	459
24.3.1	Relational choice	459
24.3.2	Guarded relational choice	459
24.3.3	Omniscient relation choice	460
25	Library SLF.LibUnit	461
25.1	Unit type	461
25.1.1	Definition	461
25.1.2	Inhabited	461
25.2	Properties	461
25.2.1	Uniqueness	461
26	Library SLF.LibFun	462
26.1		462
26.2		462
26.2.1	Indentity function	462
26.2.2	Function update	464
26.2.3	Function image	464
26.2.4	Function preimage	466
26.2.5	Function iteration	466
27	Library SLF.LibString	468
27.1	Inhabited	468

28 Library SLF.LibMultiset	469
28.1	469
28.2	469
28.3 Construction of sets as predicates	469
28.3.1 Basic definitions	469
28.3.2 Notation through typeclasses	470
28.4 Properties of multisets	471
28.4.1 Structural properties	471
28.5 Additional predicates	473
28.5.1 Foreach	473
28.6 Tactics	474
28.6.1 Tactics to prove equalities on unions	474
28.6.2 Tactics to prove membership	479
28.6.3 Tactics to invert a membership hypothesis	481
29 Library SLF.LibCore	484
30 Library SLF.LibSepTLCbuffer	485
30.1 LibSepTLCbuffer: Appendix - Temporary Additions to TLC	485
31 Library SLF.LibSepFmap	486
31.1 LibSepFmap: Appendix - Finite Maps	486
31.2 Representation of Finite Maps	486
31.2.1 Representation of Potentially-Infinite Maps as Partial Functions	486
31.2.2 Representation of Finite Maps as Dependent Pairs	489
31.2.3 Predicates on Finite Maps	491
31.3 Properties of Operations on Finite Maps	492
31.3.1 Equality	492
31.3.2 Domain	492
31.3.3 Disjoint and Domain	493
31.3.4 Disjointness	494
31.3.5 Union	496
31.3.6 Compatibility	498
31.3.7 Read	500
31.3.8 Update	501
31.3.9 Removal	502
31.4 Tactics for Finite Maps	503
31.4.1 Tactic <code>disjoint</code> for proving disjointness results	503
31.4.2 Tactic <code>rew_map</code> for Normalizing Expressions	504
31.4.3 Tactic <code>fmap_eq</code> for Proving Equalities	504
31.5 Existence of Fresh Locations	507
31.5.1 Consecutive Locations	507
31.5.2 Existence of Fresh Locations	508

31.5.3	Existence of a Smallest Fresh Locations	510
31.5.4	Existence of Nonempty Heaps	510
32	Library SLF.LibSepVar	511
32.1	LibSepVar: Appendix - Program Variables	511
32.2	Representation of Program Variables	511
32.2.1	Representation of Variables	511
32.2.2	Tactic <i>case_var</i>	512
32.3	Representation of List of Variables	512
32.3.1	Definition of Distinct Variables	512
32.3.2	Generation of <i>n</i> Distinct Variables	512
32.4	Notation for Program Variables	514
32.4.1	Program Variables Expressed Using Definitions	514
32.4.2	Program Variables Expressed Using Notation, with a Quote Symbol	517
32.5	Optional Material	520
32.5.1	Bonuse: the Tactic <i>var_neq</i>	520
33	Library SLF.LibSepSimpl	521
33.1	LibSepSimpl: Appendix - Simplification Tactic for Entailments	521
33.2	A Functor for Separation Logic Entailment	521
33.3	Assumptions of the functor	522
33.3.1	Operators	522
33.3.2	Notation	522
33.3.3	Properties Assumed by the Functor	523
33.4	Body of the Functor	525
33.4.1	Properties of <i>himpl</i>	526
33.4.2	Properties of <i>qimpl</i>	526
33.4.3	Properties of <i>hstar</i>	527
33.4.4	Representation Predicates	527
33.4.5	<i>rew_heap</i> Tactic to Normalize Expressions with <i>hstar</i>	528
33.4.6	Auxiliary Tactics Used by <i>xpull</i> and <i>xsimpl</i>	528
33.4.7	Tactics <i>xsimpl</i> and <i>xpull</i> for Heap Entailments	529
33.5	Demos	554
34	Library SLF.LibSepMinimal	563
34.1	LibSepMinimal: Appendix - Minimalistic Soundness Proof	563
34.2	Source Language	563
34.2.1	Syntax	563
34.2.2	Semantics	565
34.2.3	Definition of Omni-Big-Step Semantics	565
34.2.4	Automation for Heap Equality and Heap Disjointness	566
34.3	Heap Predicates and Entailment	567
34.3.1	Extensionality Axioms	567

34.3.2	Core Heap Predicates	567
34.3.3	Entailment	568
34.3.4	Properties of <code>hstar</code>	569
34.3.5	Properties of <code>hpure</code>	571
34.3.6	Properties of <code>hexists</code>	571
34.3.7	Properties of <code>hforall</code>	572
34.3.8	Properties of <code>hsingle</code>	572
34.3.9	Basic Tactics for Simplifying Entailments	572
34.4	Separation Logic	573
34.4.1	Definition of Separation Logic triples	573
34.4.2	Key Properties of Omni-Big-Step Semantics	573
34.4.3	Structural Rules	574
34.4.4	Reasoning Rules for Terms	575
34.4.5	Specification of Primitive Operations	575
34.5	Bonus: Example Proof	577
35	Library <code>SLF.LibSepReference</code>	579
35.1	LibSepReference: Appendix - The Full Construction	579
35.2	Imports	579
35.2.1	Extensionality Axioms	579
35.2.2	Variables	580
35.2.3	Finite Maps	580
35.3	Source Language	580
35.3.1	Syntax	580
35.3.2	Rocq Tweaks	581
35.3.3	Values and Substitutions	582
35.3.4	Small-Step Semantics	583
35.3.5	Omni-Big-Step Semantics of primitive operations	585
35.3.6	Omni-Big-Step Semantics	586
35.4	Heap Predicates	590
35.4.1	Hprop and Entailment	590
35.4.2	Definition of Heap Predicates	590
35.4.3	Basic Properties of Separation Logic Operators	592
35.4.4	Properties of <code>haffine</code>	603
35.5	Reasoning Rules	605
35.5.1	Definition of Separation Logic Triples.	605
35.5.2	Soundness of Triples w.r.t. the Small-Step Semantics	605
35.5.3	Structural Properties of the <code>eval</code> Predicate	611
35.5.4	Structural Rules	612
35.5.5	Rules for Terms	613
35.5.6	Rules for Heap-Manipulating Primitive Operations	615
35.5.7	Rules for Other Primitive Operations	616
35.6	WP Generator	621

35.6.1	Definition of Context as List of Bindings	622
35.6.2	Multi-Substitution	624
35.6.3	Definition of <code>wpgen</code>	626
35.7	Practical Proofs	631
35.7.1	Lemmas for Tactics to Manipulate <code>wpgen</code> Formulae	631
35.7.2	Tactics to Manipulate <code>wpgen</code> Formulae	634
35.7.3	Additional Tooling for <code>xapp</code>	637
35.7.4	Notations for Triples and <code>wpgen</code>	638
35.7.5	Notation for Concrete Terms	640
35.7.6	Scopes, Coercions and Notations for Concrete Programs	645
35.8	Additional Separation Logic Operators	645
35.8.1	Disjunction: Definition and Properties of <code>hor</code>	645
35.8.2	Conjunction: Definition and Properties of <code>hand</code>	646
35.9	Generalization to N-ary Functions	647
35.9.1	Smart Constructors for N-ary Functions and Applications	647
35.9.2	Evaluation Rules for N-ary Constructs	648
35.9.3	Tooling for N-ary Functions and Applications	650
35.9.4	Implementation of the Tactic <code>xwp</code>	653
35.9.5	Specification of Array Operations	655
35.9.6	Specification of Record Operations	655
35.10	Demo Programs	660
36	Library SLF.Preface	665
36.1	Preface: Introduction to the Course	665
36.2	Welcome	665
36.3	Separation Logic	665
36.4	Separation Logic in a Proof Assistant	666
36.5	Multiple Reading Depths	667
36.6	Chapters Overview	668
36.7	Other Distributed Files	669
36.8	Practicalities	670
36.8.1	System Requirements	670
36.8.2	Feedback Welcome	670
36.8.3	Exercises	670
36.8.4	Recommended Citation Format	670
36.9	Thanks	670
37	Library SLF.Basic	671
37.1	Basic: Basic Proofs in Separation Logic	671
37.2	A First Taste	671
37.2.1	Parsing of Programs	671
37.2.2	Specification of the Increment Function	672
37.2.3	Verification of the Increment Function	673

37.2.4	A Function with a Return Value	674
37.3	Separation Logic Operators	676
37.3.1	Increment of Two References	676
37.3.2	Aliased Arguments	677
37.3.3	A Function that Takes Two References and Increments One	678
37.3.4	Transfer from one Reference to Another	679
37.3.5	Specification of Allocation	680
37.3.6	Allocation of a Reference with Greater Contents	681
37.3.7	The Power of the Frame Rule with Respect to Allocation	682
37.3.8	Deallocation in Separation Logic	682
37.3.9	Combined Reading and Freeing of a Reference	684
37.3.10	Nondeterminism: Specifying Random Output Values	685
37.4	Recursive Functions	685
37.4.1	Axiomatization of the Mathematical Factorial Function	685
37.4.2	A Partial Recursive Function, Without State	686
37.4.3	A Recursive Function with State	688
37.4.4	Trying to Prove Incorrect Specifications	689
37.4.5	A Recursive Function Involving two References	691
37.4.6	Computing Factorial using State	692
37.5	Summary	693
37.6	Optional Material	694
37.7	Historical Notes	696
38	Library SLF.Repr	697
38.1	Repr: Representation Predicates	697
38.2	First Pass	697
38.2.1	The List Representation Predicate	698
38.2.2	Alternative Characterizations of MList	699
38.2.3	In-place Concatenation of Two Mutable Lists	700
38.2.4	Smart Constructors for Linked Lists	701
38.2.5	Copy Function for Lists	703
38.2.6	Length Function for Lists	705
38.2.7	Alternative Length Function for Lists	705
38.2.8	Free Function for Lists	706
38.2.9	In-Place Reversal Function for Lists	707
38.3	More Details	708
38.3.1	A Stack Represented as a List and a Size	708
38.3.2	Formalization of the Tree Representation Predicate MTree	711
38.3.3	Alternative Characterization of MTree	712
38.3.4	Additional Tooling for MTree	713
38.3.5	Deep-Copy of a Tree	714
38.3.6	Computing the Sum of the Items in a Tree	715
38.3.7	Verification of a Counter Function with Local State	716

38.4	Optional Material	719
38.4.1	Specification of a Higher-Order Repeat Operator	719
38.4.2	Specification of an Iterator on Mutable Lists	720
38.4.3	Computing the Length of a Mutable List using an Iterator	721
38.4.4	Factorial Function in Continuation-Passing Style	722
38.4.5	In-Place Concatenation Function in Continuation-Passing Style	723
38.4.6	Historical Notes	725
39	Library <i>SLF.Hprop</i>	726
39.1	Hprop: Heap Predicates	726
39.2	First Pass	726
39.2.1	Description of Memory States	727
39.2.2	Heap Predicates	728
39.2.3	Extensionality for Heap Predicates	729
39.2.4	Fundamental Properties of Operators	730
39.2.5	Postconditions: Type, Syntax, and Extensionality	730
39.2.6	Introduction and Inversion Lemmas for Basic Operators	731
39.2.7	Proofs of Fundamental Properties	732
39.3	More Details	735
39.3.1	Alternative Definitions for Heap Predicates	735
39.4	Optional Material	736
39.4.1	More on the Definition of the Existential Quantifier	736
39.4.2	Formulation of the Extensionality Axioms	737
39.4.3	Historical Notes	737
40	Library <i>SLF.Himpl</i>	739
40.1	Himpl: Heap Entailment	739
40.2	First Pass	739
40.2.1	Definition of Entailment	740
40.2.2	Entailment for Postconditions	741
40.2.3	Frame Rule for Entailment	741
40.2.4	Rules for Proving Entailments	742
40.2.5	Extracting Information from Heap Predicates	743
40.2.6	The <i>xsimpl</i> Tactic	744
40.2.7	The <i>xchange</i> Tactic	749
40.2.8	Identifying True and False Entailments	749
40.3	More Details	752
40.3.1	Proving Entailments by Hand	752
40.3.2	Inside the <i>xchange</i> Tactic	753
40.4	Optional Material	753
40.4.1	Proofs of Rules for Entailment	753
40.4.2	Historical Notes	754

41 Library SLF.Triples	755
41.1 Triples: Structural Reasoning Rules	755
41.2 First Pass	755
41.2.1 Formalized Syntax of the Programming Language	756
41.2.2 Formalized Semantics of the Programming Language	760
41.2.3 Definition of Triples	766
41.2.4 Structural Rules	766
41.2.5 Derived Structural Rules	767
41.3 More Details	768
41.3.1 Proof of the Consequence Rule	768
41.3.2 Proof of the Frame Rule	769
41.3.3 Proof of the Extraction Rules	770
41.3.4 Rule for Naming Heaps	771
41.4 Optional Material	771
41.4.1 An Alternative Structural Rule for Existentials	771
41.4.2 Small-Step Semantics	772
41.4.3 Equivalence Between Small-Step and Omni-Big-Step	774
41.4.4 Small-Step-Based Triples	779
41.4.5 Separation Logic Triples for Quasi-Deterministic Semantics	782
41.4.6 Historical Notes	784
42 Library SLF.Rules	785
42.1 Rules: Reasoning Rules for Term Constructs	785
42.2 First Pass	785
42.2.1 Rules for Term Constructors	786
42.2.2 Primitive Arithmetic Operations	790
42.2.3 Primitive Heap-Manipulating Operations	792
42.2.4 Program Verification using the Reasoning Rules	795
42.3 More Details	797
42.3.1 Triple for Terms with Same Semantics	797
42.3.2 The Combined Let-Frame Rule	799
42.4 Optional Material	800
42.4.1 Alternative Specification Style for Pure Preconditions	800
42.4.2 Alternative Specification Style for Result Values	801
42.4.3 Historical Notes	802
43 Library SLF.Wand	804
43.1 Wand: The Magic Wand and Other Operators	804
43.2 First Pass	804
43.2.1 Definition of the Operator <code>hforall</code>	805
43.2.2 Definition of the Operator <code>hor</code>	806
43.2.3 Definition of the Operator <code>hand</code>	809
43.2.4 Definition of the Magic Wand Operator: <code>hwand</code>	810

43.2.5	Definition of the Magic Wand Operator for Postconditions: <code>qwand</code>	815
43.2.6	Simplifications of Magic Wands using <code>xsimpl</code>	817
43.2.7	Summary of All the Separation Logic Operators	818
43.2.8	The Ramified Frame Rule	819
43.3	More Details	821
43.3.1	Benefits of the Ramified Frame Rule	821
43.3.2	Some Tempting, But False, Properties of the Magic Wand	822
43.4	Optional Material	823
43.4.1	Equivalence Between Alternative Definitions of the Magic Wand	823
43.4.2	Equivalence Results for the Magic Wand for Postconditions	825
43.4.3	Historical Notes	827
44	Library <code>SLF.WPsem</code>	828
44.1	WPsem: Semantics of Weakest Preconditions	828
44.2	First Pass	828
44.2.1	Definition of Weakest Preconditions	829
44.2.2	Structural Rules in Weakest-Precondition Style	831
44.2.3	Reasoning Rules for Terms in Weakest-Precondition Style	834
44.3	More Details	836
44.3.1	Combined Structural Rule	836
44.4	Optional Material	838
44.4.1	Weakest Preconditions Derived from Triples, a First Route	838
44.4.2	Weakest Preconditions Derived From Triples, a Second Route	838
44.4.3	Equivalent Characterizations of <code>wp</code>	839
44.4.4	Texan Triples	840
44.4.5	Historical Notes	843
45	Library <code>SLF.WPgen</code>	844
45.1	WPgen: Weakest Precondition Generator	844
45.2	First Pass	844
45.2.1	The Basic Structure of <code>wpgen</code>	845
45.2.2	Step 1: <code>wpgen</code> as a Recursive Function over Terms	846
45.2.3	Step 2: <code>wpgen</code> Reformulated to Eagerly Match on the Term	846
45.2.4	Step 3: <code>wpgen</code> Reformulated with Auxiliary Definitions	847
45.2.5	Step 4: <code>wpgen</code> Augmented with Support for Structural Rules	847
45.3	More Details	848
45.3.1	Definition of <code>wpgen</code> for Each Term Construct	848
45.3.2	Computing with <code>wpgen</code>	852
45.3.3	Optimizing the Readability of <code>wpgen</code> 's Output	857
45.3.4	Extension of <code>wpgen</code> to Handle Structural Rules	860
45.3.5	Implementation of X-tactics	863
45.3.6	Reasoning about Local Functions	868
45.3.7	Proof Structure and Proof Automation	873

45.4	Optional Material	874
45.4.1	Additional Tactics <i>xconseq</i> and <i>xframe</i>	874
45.4.2	Evaluation of <i>wpgen</i> Recursively inside Local Functions	875
45.4.3	Historical Notes	882
46	Library SLF.WPsound	884
46.1	WPsound: Soundness of the Weakest Precondition Generator	884
46.2	More Details	884
46.2.1	Definition of the Predicate <i>formula_sound</i>	884
46.2.2	Soundness for the Case of Sequences	885
46.2.3	Soundness for the Other Term Constructs	886
46.2.4	Soundness of the Predicate Transformer <i>mkstruct</i>	888
46.2.5	Lemmas Capturing Properties of Iterated Substitutions	889
46.2.6	Main Induction for the Soundness Proof of <i>wpgen</i>	889
46.2.7	Statement of Soundness of <i>wpgen</i> for Closed Terms	890
46.3	Optional Material	891
46.3.1	Proofs of Properties of Iterated Substitution	891
47	Library SLF.Affine	896
47.1	Affine: Affine Separation Logic	896
47.2	First Pass	896
47.2.1	Motivation for the Discard Rule	897
47.2.2	Statement of The Discard Rule	898
47.2.3	Fine-grained Control of Collectable Predicates	899
47.2.4	The Affine-Top Heap Predicate: $\backslash GC$	902
47.2.5	Example Instantiations of <i>heap_affine</i>	904
47.2.6	Definition of a Partially Affine Separation Logic	906
47.3	More Details	910
47.3.1	Discard Rules in WP Style	910
47.3.2	Revised Definition of <i>mkstruct</i>	913
47.3.3	The Tactic <i>xaffine</i> and the Behavior of <i>xsimpl</i> on $\backslash GC$	914
47.3.4	Tactics for Applying the Discard Rules	915
47.4	Optional Material	917
47.4.1	Alternative Statement for Distribution of <i>haffine</i> on Quantifiers	917
47.4.2	Deriving the Rule GC-Post from the Rule GC-Pre	917
47.4.3	Historical Notes	918
48	Library SLF.Arrays	919
48.1	Arrays: Reasoning about Arrays	919
48.2	First Pass	919
48.2.1	Large-Footprint Specifications for Array Operations	920
48.2.2	Small-Footprint Specifications for Array Operations	922
48.2.3	Heap Predicate for Array Segments	923

48.2.4	Derived Segment-Based Specifications for Array Operations	925
48.2.5	Verification of the Safety of QuickSort	925
48.3	More Details	927
48.3.1	Formalization of Sorted Lists	927
48.3.2	Formalization of Arrays of Integer Values	930
48.3.3	Functional Correctness of Quicksort	931
48.3.4	Realization of <code>hheader</code> and <code>hcell</code>	932
48.3.5	Realization of <code>hseg</code> and <code>harray</code>	933
48.3.6	Focus Lemmas for Array Segments	935
48.4	Optional Material	937
48.4.1	Semantics of Pointer Arithmetic	937
48.4.2	Semantics of Low-Level Block Allocation	938
48.4.3	Low-Level Implementation of Arrays	939
48.4.4	Verification of Low-Level Operations for Arrays	940
48.5	Appendix	943
48.5.1	Verification of the Pivot Function	943
49	Library <code>SLF.Records</code>	949
49.1	Records: Reasoning about Records	949
49.2	First Pass	949
49.2.1	Representation of Individual Record Fields	950
49.2.2	Representation of Sets of Fields	950
49.2.3	Representation of Records	951
49.2.4	Reasoning about Reads from Record Fields	953
49.2.5	Reasoning about Writes To Record Fields	954
49.2.6	Reasoning about Record Allocation with Initializers	955
49.3	Optional Material	956
49.3.1	Implementation of Record Accesses using Pointer Arithmetic	956
49.3.2	Specification of Record Accesses w.r.t. <code>hfields</code>	957
49.3.3	Specification of Record Accesses w.r.t. <code>hrecord</code>	958
49.3.4	Implementation of Record Allocation without Initialization	959
49.3.5	Implementation of Record Allocation with Initialization	962
49.3.6	Extending <i>xapp</i> to Support Record Access Operations	963
50	Library <code>SLF.Postscript</code>	965
50.1	Postscript: Conclusion and Perspectives	965
50.2	Beyond the Scope of This Course	965
50.3	Tools Leveraging Separation Logic	965
50.4	Related Courses	966
50.5	Acknowledgments	967

51 Library SLF.Bib	968
51.1 Bib: Bibliography	968
51.2 Resources cited in this volume	968

Chapter 1

Library `SLF.LibAxioms`

`Set Implicit Arguments.`

This file is used to extend Rocq with standard axioms from classical, non-constructive higher-order logic. (Such axioms are provided by default in other theorem provers like Isabelle/HOL or HOL4.)

Three axioms taken are: functional extensionality, and propositional extensionality and indefinite description.

All other common axioms are derivable, including: the excluded middle, the strong excluded middle, propositional degeneracy, proof irrelevance, injectivity of equality on dependent pairs, predicate extensionality, definite description, and all the versions of the axiom of choice.

1.1 Functional extensionality

Two functions that yield equal results on equal arguments are equal.

```
Axiom fun_ext_dep : ∀ (A : Type) (B : A → Type) (f g : ∀ x, B x),  
  (∀ x, f x = g x) →  
  f = g.
```

1.2 Propositional extensionality

Two propositions that are equivalent can be considered to be equal.

```
Axiom prop_ext : ∀ (P Q : Prop),  
  (P ↔ Q) →  
  P = Q.
```

1.3 Indefinite description

Proofs of existence can be reified from `Prop` to `Type`. The lemma below is a concise statement for $(\exists x, P\ x) \rightarrow \{ x : A \mid P\ x \}$.

```
Axiom indefinite_description :  $\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}),$   
  ex  $P \rightarrow$   
  sig  $P$ .
```

Chapter 2

Library `SLF.LibTactics`

This file contains a set of tactics that extends the set of builtin tactics provided with the standard distribution of Rocq. It intends to overcome a number of limitations of the standard set of tactics, and thereby to help user to write shorter and more robust scripts.

Hopefully, Rocq tactics will be improved as time goes by, and this file should ultimately be useless. In the meanwhile, serious Rocq users will probably find it very useful.

The main features offered are:

- More convenient syntax for naming hypotheses, with tactics for introduction and inversion that take as input only the name of hypotheses of type `Prop`, rather than the name of all variables.
- Tactics providing true support for manipulating N-ary conjunctions, disjunctions and existentials, hiding the fact that the underlying implementation is based on binary propositions.
- Convenient support for automation: tactics followed with the symbol “~” or “*” will call automation on the generated subgoals. The symbol “~” stands for `auto` and “*” for `intuition eauto`. These bindings can be customized.
- Forward-chaining tactics are provided to instantiate lemmas either with variable or hypotheses or a mix of both.
- A more powerful implementation of `apply` is provided (it is based on `refine` and thus behaves better with respect to conversion).
- An improved inversion tactic which substitutes equalities on variables generated by the standard inversion mechanism. Moreover, it supports

the elimination of dependently typed equalities (requires axiom `K`, which is a weak form of Proof Irrelevance).

- Tactics for saving time when writing proofs, with tactics to asserts hypotheses or sub-goals, and improved tactics for clearing, renaming, and sorting hypotheses.

External credits:

- thanks to Xavier Leroy for providing the idea of tactic *forward*
- thanks to Georges Gonthier for the implementation trick in *rappl*

```
Set Implicit Arguments.
```

```
From Stdlib Require Import List.
```

```
Declare Scope ltac_scope.
```

2.1 Fixing Stdlib

```
#[global]
Remove Hints Bool.trans_eq_bool : core.
```

2.2 Tools for Programming with Ltac

2.2.1 Identity Continuation

```
Ltac idcont tt :=
  idtac.
```

2.2.2 Untyped Arguments for Tactics

Any Rocq value can be boxed into the type **Boxer**. This is useful to use Rocq computations for implementing tactics.

```
Inductive Boxer : Type :=
  | boxer : ∀ (A:Type), A → Boxer.
```

2.2.3 Optional Arguments for Tactics

ltac_no_arg is a constant that can be used to simulate optional arguments in tactic definitions. Use *mytactic* *ltac_no_arg* on the tactic invocation, and use `match arg with ltac_no_arg ⇒ ..` or `match type of arg with ltac_No_arg ⇒ ..` to test whether an argument was provided.

```
Inductive ltac_No_arg : Set :=
  | ltac_no_arg : ltac_No_arg.
```

2.2.4 Wildcard Arguments for Tactics

`ltac_wild` is a constant that can be used to simulate wildcard arguments in tactic definitions. Notation is `--`.

```
Inductive ltac_Wild : Set :=
| ltac_wild : ltac_Wild.
```

Notation `"'--'"` := `ltac_wild` : *ltac_scope*.

`ltac_wilds` is another constant that is typically used to simulate a sequence of *N* wildcards, with *N* chosen appropriately depending on the context. Notation is `---`.

```
Inductive ltac_Wilds : Set :=
| ltac_wilds : ltac_Wilds.
```

Notation `"'---'"` := `ltac_wilds` : *ltac_scope*.

Open Scope *ltac_scope*.

2.2.5 Position Markers

`ltac_Mark` and `ltac_mark` are dummy definitions used as sentinel by tactics, to mark a certain position in the context or in the goal.

```
Inductive ltac_Mark : Type :=
| ltac_mark : ltac_Mark.
```

gen_until_mark repeats `generalize` on hypotheses from the context, starting from the bottom and stopping as soon as reaching an hypothesis of type *Mark*. It fails if *Mark* does not appear in the context.

```
Ltac gen_until_mark :=
  match goal with H: ?T ⊢ _ ⇒
  match T with
  | ltac_Mark ⇒ clear H
  | _ ⇒ generalize H; clear H; gen_until_mark
  end end.
```

gen_until_mark_with_processing F is similar to *gen_until_mark* except that it calls *F* on each hypothesis immediately before generalizing it. This is useful for processing the hypotheses.

```
Ltac gen_until_mark_with_processing cont :=
  match goal with H: ?T ⊢ _ ⇒
  match T with
  | ltac_Mark ⇒ clear H
  | _ ⇒ cont H; generalize H; clear H;
        gen_until_mark_with_processing cont
  end end.
```

intro_until_mark repeats *intro* until reaching an hypothesis of type *Mark*. It throws away the hypothesis *Mark*. It fails if *Mark* does not appear as an hypothesis in the goal.

```
Ltac intro_until_mark :=
  match goal with
  | ⊢ (ltac_Mark → _) ⇒ intros _
  | _ ⇒ intro; intro_until_mark
  end.
```

2.2.6 List of Arguments for Tactics

A datatype of type `list Boxer` is used to manipulate list of Rocq values in `ltac`. Notation is `» v1 v2 ... vN` for building a list containing the values `v1` through `vN`.

```
Notation "»" :=
  (@nil Boxer)
  (at level 0)
  : ltac_scope.
```

```
Notation "»' v1" :=
  ((boxer v1)::nil)
  (at level 0, v1 at level 0)
  : ltac_scope.
```

```
Notation "»' v1 v2" :=
  ((boxer v1)::(boxer v2)::nil)
  (at level 0, v1 at level 0, v2 at level 0)
  : ltac_scope.
```

```
Notation "»' v1 v2 v3" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0)
  : ltac_scope.
```

```
Notation "»' v1 v2 v3 v4" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0)
  : ltac_scope.
```

```
Notation "»' v1 v2 v3 v4 v5" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0)
  : ltac_scope.
```

```
Notation "»' v1 v2 v3 v4 v5 v6" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
```

```

    v4 at level 0, v5 at level 0, v6 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0, v9 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0, v9 at level 0, v10 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
   ::(boxer v11)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0, v9 at level 0, v10 at level 0, v11 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
   ::(boxer v11)::(boxer v12)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,

```

```

    v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
    v8 at level 0, v9 at level 0, v10 at level 0, v11 at level 0,
    v12 at level 0)
  : ltac_scope.
Notation "'»' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
   ::(boxer v11)::(boxer v12)::(boxer v13)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0, v9 at level 0, v10 at level 0, v11 at level 0,
   v12 at level 0, v13 at level 0)
  : ltac_scope.

```

The tactic *list_boxer_of* inputs a term *E* and returns a term of type “list boxer”, according to the following rules:

- if *E* is already of type “list Boxer”, then it returns *E*;
- otherwise, it returns the list (boxer *E*)::nil.

```

Ltac list_boxer_of E :=
  match type of E with
  | List.list Boxer => constr:(E)
  | _ => constr:((boxer E)::nil)
  end.

```

2.2.7 Databases of Lemmas

Use the hint facility to implement a database mapping terms to terms. To declare a new database, use a definition: *Definition mydatabase := True.*

Then, to map *mykey* to *myvalue*, write the hint: *Hint Extern 1 (Register mydatabase mykey) => Provide myvalue.*

Finally, to query the value associated with a key, run the tactic *ltac_database_get mydatabase mykey*. This will leave at the head of the goal the term *myvalue*. It can then be named and exploited using *intro*.

```

Inductive Ltac_database_token : Prop := ltac_database_token.

```

```

Definition ltac_database (D:Boxer) (T:Boxer) (A:Boxer) := Ltac_database_token.

```

```

Notation "'Register' D T" := (ltac_database (boxer D) (boxer T) _)
  (at level 69, D at level 0, T at level 0).

```

```

Lemma ltac_database_provide : ∀ (A:Boxer) (D:Boxer) (T:Boxer),
  ltac_database D T A.

```

```

Proof using. split. Qed.

```



```

Ltac Provide T := apply (@ltac_database_provide (boxer T)).
Ltac ltac_database_get D T :=
  let A := fresh "TEMP" in evar (A:Boxer);
  let H := fresh "TEMP" in
  assert (H : ltac_database (boxer D) (boxer T) A);
  [ subst A; auto
  | subst A; match type of H with ltac_database _ _ (boxer ?L) =>
    generalize L end; clear H ].

```

2.2.8 On-the-Fly Removal of Hypotheses

In a list of arguments » *H1 H2 .. HN* passed to a tactic such as *lets* or *applies* or *forwards* or *specializes*, the term *rm*, an identity function, can be placed in front of the name of an hypothesis to be deleted.

Definition *rm* (*A*:Type) (*X*:*A*) := *X*.

rm_term *E* removes one hypothesis that admits the same type as *E*.

```

Ltac rm_term E :=
  let T := type of E in
  match goal with H: T ⊢ _ => try clear H end.

```

rm_inside *E* calls *rm_term* *Ei* for any subterm of the form *rm Ei* found in *E*

```

Ltac rm_inside E :=
  let go E := rm_inside E in
  match E with
  | rm ?X => rm_term X
  | ?X1 ?X2 =>
    go X1; go X2
  | ?X1 ?X2 ?X3 =>
    go X1; go X2; go X3
  | ?X1 ?X2 ?X3 ?X4 =>
    go X1; go X2; go X3; go X4
  | ?X1 ?X2 ?X3 ?X4 ?X5 =>
    go X1; go X2; go X3; go X4; go X5
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 =>
    go X1; go X2; go X3; go X4; go X5; go X6
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 =>
    go X1; go X2; go X3; go X4; go X5; go X6; go X7
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 =>
    go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ?X9 =>
    go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8; go X9
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ?X9 ?X10 =>

```

```

    go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8; go X9; go X10
  | _ => idtac
end.

```

For faster performance, one may deactivate *rm_inside* by replacing the body of this definition with *idtac*.

```

Ltac fast_rm_inside E :=
  rm_inside E.

```

2.2.9 Numbers as Arguments

When tactic takes a natural number as argument, it may be parsed either as a natural number or as a relative number. In order for tactics to convert their arguments into natural numbers, we provide a conversion tactic.

Note: the tactic *number_to_nat* is extended in *LibInt* to take into account the *Z* type.

From *Stdlib* Require *BinNums BinInt*.

```

Definition ltac_int_to_nat (x:BinInt.Z) : nat :=
  match x with
  | BinInt.Z0 => 0%nat
  | BinInt.Zpos p => BinPos.nat_of_P p
  | BinInt.Zneg p => 0%nat
  end.

```

```

Ltac number_to_nat N :=
  match type of N with
  | nat => constr:(N)
  | BinInt.Z => let N' := constr:(ltac_int_to_nat N) in eval compute in N'
  end.

```

ltac_pattern E at K is the same as *pattern E at K* except that *K* is a Rocq number (nat or Z) rather than a Ltac integer. Syntax *ltac_pattern E as K in H* is also available.

```

Tactic Notation "ltac_pattern" constr(E) "at" constr(K) :=
  match number_to_nat K with
  | 1 => pattern E at 1
  | 2 => pattern E at 2
  | 3 => pattern E at 3
  | 4 => pattern E at 4
  | 5 => pattern E at 5
  | 6 => pattern E at 6
  | 7 => pattern E at 7
  | 8 => pattern E at 8
  | _ => fail "ltac_pattern: arity not supported"
  end.

```

Tactic Notation "ltac_pattern" $\text{constr}(E)$ "at" $\text{constr}(K)$ "in" $\text{hyp}(H)$:=
 match $\text{number\_to\_nat } K$ with
 | 1 $\Rightarrow$ pattern E at 1 in H
 | 2 $\Rightarrow$ pattern E at 2 in H
 | 3 $\Rightarrow$ pattern E at 3 in H
 | 4 $\Rightarrow$ pattern E at 4 in H
 | 5 $\Rightarrow$ pattern E at 5 in H
 | 6 $\Rightarrow$ pattern E at 6 in H
 | 7 $\Rightarrow$ pattern E at 7 in H
 | 8 $\Rightarrow$ pattern E at 8 in H
 | _ $\Rightarrow$ fail "ltac_pattern: arity not supported"
 end.

$\text{ltac\_set } (x := E) \text{ at } K$ is the same as $\text{set } (x := E) \text{ at } K$ except that K is a Rocq number (nat or Z) rather than a Ltac integer.

Tactic Notation "ltac_set" "(" $\text{ident}(X)$ " := " $\text{constr}(E)$ ")" "at" $\text{constr}(K)$:=
 match $\text{number\_to\_nat } K$ with
 | 1%nat $\Rightarrow$ set $(X := E)$ at 1
 | 2%nat $\Rightarrow$ set $(X := E)$ at 2
 | 3%nat $\Rightarrow$ set $(X := E)$ at 3
 | 4%nat $\Rightarrow$ set $(X := E)$ at 4
 | 5%nat $\Rightarrow$ set $(X := E)$ at 5
 | 6%nat $\Rightarrow$ set $(X := E)$ at 6
 | 7%nat $\Rightarrow$ set $(X := E)$ at 7
 | 8%nat $\Rightarrow$ set $(X := E)$ at 8
 | 9%nat $\Rightarrow$ set $(X := E)$ at 9
 | 10%nat $\Rightarrow$ set $(X := E)$ at 10
 | 11%nat $\Rightarrow$ set $(X := E)$ at 11
 | 12%nat $\Rightarrow$ set $(X := E)$ at 12
 | 13%nat $\Rightarrow$ set $(X := E)$ at 13
 | _ $\Rightarrow$ fail "ltac_set: arity not supported"
 end.

2.2.10 Testing Tactics

$\text{show } \text{tac}$ executes a tactic tac that produces a result, and then display its result.

Tactic Notation "show" $\text{tactic}(\text{tac})$:=
 let $R := \text{tac}$ in pose R .

$\text{dup } N$ produces N copies of the current goal. It is useful for building examples on which to illustrate behaviour of tactics. dup is short for $\text{dup } 2$.

Lemma $\text{dup\_lemma} : \forall P, P \rightarrow P \rightarrow P$.

Proof using. auto. Qed.

```

Ltac dup_tactic N :=
  match number_to_nat N with
  | 0 => idtac
  | S 0 => idtac
  | S ?N' => apply dup_lemma; [ | dup_tactic N' ]
  end.

Tactic Notation "dup" constr(N) :=
  dup_tactic N.

Tactic Notation "dup" :=
  dup 2.

```

2.2.11 Testing evars and non-evars

is_not_evar E succeeds only if E is not an evar; it fails otherwise. It thus implements the negation of *is_evar*

```

Ltac is_not_evar E :=
  first [ is_evar E; fail 1
        | idtac ].

is_evar_as_bool E evaluates to true if  $E$  is an evar and to false otherwise.

```

```

Ltac is_evar_as_bool E :=
  constr:(ltac:(first
    [ is_evar E; exact true
    | exact false ])).

has_no_evar E succeeds if  $E$  contains no evars.

```

```

Ltac has_no_evar E :=
  first [ has_evar E; fail 1 | idtac ].

```

2.2.12 Check No Evar in Goal

```

Ltac check_noevar M :=
  first [ has_evar M; fail 2 | idtac ].

Ltac check_noevar_hyp H :=
  let T := type of H in check_noevar T.

Ltac check_noevar_goal :=
  match goal with ⊢ ?G => check_noevar G end.

```

2.2.13 Helper Function for Introducing Evvars

with_evar T (**fun** $M \Rightarrow tac$) creates a new evar that can be used in the tactic tac under the name M .

```

Ltac with_evar_base T cont :=
  let x := fresh "TEMP" in evar (x:T); cont x; subst x.
Tactic Notation "with_evar" constr(T) tactic(cont) :=
  with_evar_base T cont.

```

2.2.14 Tagging of Hypotheses

get_last_hyp tt is a function that returns the last hypothesis at the bottom of the context. It is useful to obtain the default name associated with the hypothesis, e.g. `intro; let H := get_last_hyp tt in let H' := fresh "P" H in ...`

```

Ltac get_last_hyp tt :=
  match goal with H: _ ⊢ _ ⇒ constr:(H) end.

```

2.2.15 More Tagging of Hypotheses

ltac_tag_subst is a specific marker for hypotheses which is used to tag hypotheses that are equalities to be substituted.

Definition *ltac_tag_subst* (*A*:Type) (*x*:*A*) := *x*.

ltac_to_generalize is a specific marker for hypotheses to be generalized.

Definition *ltac_to_generalize* (*A*:Type) (*x*:*A*) := *x*.

```

Ltac gen_to_generalize :=
  repeat match goal with
    H: ltac_to_generalize _ ⊢ _ ⇒ generalize H; clear H end.

```

```

Ltac mark_to_generalize H :=
  let T := type of H in
  change T with (ltac_to_generalize T) in H.

```

2.2.16 Deconstructing Terms

get_head *E* is a tactic that returns the head constant of the term *E*, ie, when applied to a term of the form *P* *x*₁ ... *x*_{*N*} it returns *P*. If *E* is not an application, it returns *E*. Warning: the tactic seems to loop in some cases when the goal is a product and one uses the result of this function.

```

Ltac get_head E :=
  match E with
  | ?E' ?x ⇒ get_head E'
  | _ ⇒ constr:(E)
  end.

```

get_fun_arg *E* is a tactic that decomposes an application term *E*, ie, when applied to a term of the form *X*₁ ... *X*_{*N*} it returns a pair made of *X*₁ .. *X*_(*N*-1) and *X*_{*N*}.

```

Ltac get_fun_arg E :=
  match E with
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X ⇒ constr:((X1 X2 X3 X4 X5 X6 X7, X))
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X ⇒ constr:((X1 X2 X3 X4 X5 X6, X))
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X ⇒ constr:((X1 X2 X3 X4 X5, X))
  | ?X1 ?X2 ?X3 ?X4 ?X ⇒ constr:((X1 X2 X3 X4, X))
  | ?X1 ?X2 ?X3 ?X ⇒ constr:((X1 X2 X3, X))
  | ?X1 ?X2 ?X ⇒ constr:((X1 X2, X))
  | ?X1 ?X ⇒ constr:((X1, X))
  end.

```

2.2.17 Action at Occurrence and Action Not at Occurrence

ltac_action_at K of E do Tac isolates the K -th occurrence of E in the goal, setting it in the form $P E$ for some named pattern P , then calls tactic Tac , and finally unfolds P . Syntax *ltac_action_at* K of E in H do Tac is also available.

```

Tactic Notation "ltac_action_at" constr(K) "of" constr(E) "do" tactic(Tac) :=
  let p := fresh "TEMP" in ltac_pattern E at K;
  match goal with ⊢ ?P _ ⇒ set (p:=P) end;
  Tac; unfold p; clear p.

```

```

Tactic Notation "ltac_action_at" constr(K) "of" constr(E) "in" hyp(H) "do" tactic(Tac)
:=
  let p := fresh "TEMP" in ltac_pattern E at K in H;
  match type of H with ?P _ ⇒ set (p:=P) in H end;
  Tac; unfold p in H; clear p.

```

protects E do Tac temporarily assigns a name to the expression E so that the execution of tactic Tac will not modify E . This is useful for instance to restrict the action of *simpl*.

```

Tactic Notation "protects" constr(E) "do" tactic(Tac) :=

```

```

  let x := fresh "TEMP" in let H := fresh "TEMP" in
  set (X := E) in *; assert (H : X = E) by reflexivity;
  clearbody X; Tac; subst x.

```

```

Tactic Notation "protects" constr(E) "do" tactic(Tac) "/" :=
  protects E do Tac.

```

2.2.18 An Alias for eq

eq' is an alias for *eq* to be used for equalities in inductive definitions, so that they don't get mixed with equalities generated by *inversion*.

Definition *eq'* := @*eq*.

```

#[global]

```

Hint Unfold eq' : *core*.

Notation "x '=' y" := (@eq' - x y)
(at level 70, y at *next* level).

2.3 Check as a tactic instead of a top-level command, with possibility to turn off

check E is like *Check E*, but works as a tactic. It can be turned off using *Ltac check_enabled* ::= *constr:(false)*.

Ltac check_enabled := *constr:(true)*.

Tactic Notation "check" *uconstr(E)* :=
 lazymatch *check_enabled* with
 | false ⇒ idtac
 | true ⇒
 first [generalize *E*; match goal with ⊢ ?*T* → _ ⇒ idtac *E*; idtac ":" *T*
end; intros _
 | idtac *E*; fail 1 "does not typecheck"]
end.

2.4 Common Tactics for Simplifying Goals Like *intuition*

Ltac jauto_set_hyps :=
 repeat match goal with *H*: ?*T* ⊢ _ ⇒
 match *T* with
 | _ ∧ _ ⇒ destruct *H*
 | iff _ _ ⇒ destruct *H*
 | ∃ *a*, _ ⇒ destruct *H*
 | _ ⇒ generalize *H*; clear *H*
 end
end.

Ltac jauto_set_goal :=
 repeat match goal with
 | ⊢ ∃ *a*, _ ⇒ *esplit*
 | ⊢ _ ∧ _ ⇒ *split*
 | ⊢ iff _ _ ⇒ *split*
 end.

Ltac jauto_set :=
 intros; *jauto_set_hyps*;
 intros; *jauto_set_goal*;

unfold not in $*$.

2.5 Backward and Forward Chaining

2.5.1 Application

*rappl*y is a tactic similar to *eappl*y except that it is based on the *refine* tactics, and thus is strictly more powerful (at least in theory :). In short, it is able to perform on-the-fly conversions when required for arguments to match, and it is able to instantiate existentials when required.

Tactic Notation "rapply" $\text{constr}(t) :=$

```
first
| eexact (@t)
| refine (@t)
| refine (@t _)
| refine (@t _ _)
| refine (@t _ _ _)
| refine (@t _ _ _ _)
| refine (@t _ _ _ _ _)
| refine (@t _ _ _ _ _ _)
| refine (@t _ _ _ _ _ _ _)
| refine (@t _ _ _ _ _ _ _ _)
| refine (@t _ _ _ _ _ _ _ _ _)
| refine (@t _ _ _ _ _ _ _ _ _ _)
| refine (@t _ _ _ _ _ _ _ _ _ _)
```

No-typeclass refine apply, TEMPORARY for Coq < 8.11.

$$\text{Ltac } nrapply\ H :=$$

```
first
[ notypeclasses refine (H)
| notypeclasses refine (H _)
| notypeclasses refine (H _ _)
| notypeclasses refine (H _ _ _)
| notypeclasses refine (H _ _ _ _)
| notypeclasses refine (H _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _)
```



```

| notypeclasses refine (H _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _)
| notypeclasses refine (H _ _ _ _ _ _ _) ].

```

The tactics *applies_N T*, where *N* is a natural number, provides a more efficient way of using *applies T*. It avoids trying out all possible arities, by specifying explicitly the arity of function *T*.

```

Tactic Notation "rapply_0" constr(t) :=
  refine (@t).
Tactic Notation "rapply_1" constr(t) :=
  refine (@t _).
Tactic Notation "rapply_2" constr(t) :=
  refine (@t _ _).
Tactic Notation "rapply_3" constr(t) :=
  refine (@t _ _ _).
Tactic Notation "rapply_4" constr(t) :=
  refine (@t _ _ _ _).
Tactic Notation "rapply_5" constr(t) :=
  refine (@t _ _ _ _ _).
Tactic Notation "rapply_6" constr(t) :=
  refine (@t _ _ _ _ _ _).
Tactic Notation "rapply_7" constr(t) :=
  refine (@t _ _ _ _ _ _ _).
Tactic Notation "rapply_8" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _).
Tactic Notation "rapply_9" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _ _).
Tactic Notation "rapply_10" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _ _ _).

```

lets_base H E adds an hypothesis *H* : *T* to the context, where *T* is the type of term *E*. If *H* is an introduction pattern, it will destruct *H* according to the pattern.

Ltac *lets_base I E* := generalize *E*; intros *I*.

applies_to H E transform the type of hypothesis *H* by replacing it by the result of the application of the term *E* to *H*. Intuitively, it is equivalent to *lets H*: (*E H*).

```

Tactic Notation "applies_to" hyp(H) constr(E) :=
  let H' := fresh "TEMP" in rename H into H';
  (first [ lets_base H (E H')
          | lets_base H (E _ H')
        ]

```

```

| lets_base H (E _ _ H')
| lets_base H (E _ _ _ H')
| lets_base H (E _ _ _ _ H')
| lets_base H (E _ _ _ _ _ H')
| lets_base H (E _ _ _ _ _ _ H')
| lets_base H (E _ _ _ _ _ _ _ H')
| lets_base H (E _ _ _ _ _ _ _ _ H') ]
); clear H'.

applies_to H1, ..., HN E applies E to several hypotheses
Tactic Notation "applies_to" hyp(H1) ", " hyp(H2) constr(E) :=
  applies_to H1 E; applies_to H2 E.
Tactic Notation "applies_to" hyp(H1) ", " hyp(H2) ", " hyp(H3) constr(E) :=
  applies_to H1 E; applies_to H2 E; applies_to H3 E.
Tactic Notation "applies_to" hyp(H1) ", " hyp(H2) ", " hyp(H3) ", " hyp(H4) constr(E)
:=
  applies_to H1 E; applies_to H2 E; applies_to H3 E; applies_to H4 E.
constructors calls constructor or econstructor.
Tactic Notation "constructors" :=
  first [ constructor | econstructor ]; unfold eq'.

```

2.5.2 Assertions

asserts *H*: *T* is another syntax for **assert** (*H* : *T*), which also works with introduction patterns.

```

Tactic Notation "asserts" simple_intropattern(I) ":" constr(T) :=
  let H := fresh "TEMP" in assert (H : T);
  [ | generalize H; clear H; intros I ].
asserts H1 .. HN: T is a shorthand for asserts \[H1 \[H2 \[..HN\]\]\]: T].
Tactic Notation "asserts" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(T) :=
  asserts [I1 I2]: T.
Tactic Notation "asserts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  asserts [I1 [I2 I3]]: T.
Tactic Notation "asserts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) ":" constr(T) :=
  asserts [I1 [I2 [I3 I4]]]: T.
Tactic Notation "asserts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)

```

```

    simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
      asserts [I1 [I2 [I3 [I4 I5]]]]: T.
Tactic Notation "asserts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) ":" constr(T) :=
    asserts [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.

    asserts: T is asserts H: T with H being chosen automatically.

Tactic Notation "asserts" ":" constr(T) :=
  let H := fresh "TEMP" in asserts H : T.

    cuts H: T is the same as asserts H: T except that the two subgoals generated are
    swapped: the subgoal T comes second. Note that contrary to cut, it introduces the hypoth-
    esis.

Tactic Notation "cuts" simple_intropattern(I) ":" constr(T) :=
  cut (T); [ intros I | idtac ].

    cuts: T is cuts H: T with H being chosen automatically.

Tactic Notation "cuts" ":" constr(T) :=
  let H := fresh "TEMP" in cuts H: T.

    cuts H1 .. HN: T is a shorthand for cuts \[H1 \[H2 \[.. HN\]\]\]: T].

Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(T) :=
  cuts [I1 I2]: T.
Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  cuts [I1 [I2 I3]]: T.
Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) ":" constr(T) :=
  cuts [I1 [I2 [I3 I4]]]: T.
Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
  cuts [I1 [I2 [I3 [I4 I5]]]]: T.
Tactic Notation "cuts" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3)
  simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) ":" constr(T) :=
  cuts [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.

```

2.5.3 Instantiation and Forward-Chaining

The instantiation tactics are used to instantiate a lemma E (whose type is a product) on some arguments. The type of E is made of implications and universal quantifications, e.g. $\forall x, P\ x \rightarrow \forall y\ z, Q\ x\ y\ z \rightarrow R\ z$.

The first possibility is to provide arguments in order: first x , then a proof of $P\ x$, then y etc... In this mode, called “Args”, all the arguments are to be provided. If a wildcard is provided (written $--$), then an existential variable will be introduced in place of the argument.

It is very convenient to give some arguments the lemma should be instantiated on, and let the tactic find out automatically where underscores should be insterted. Underscore arguments $--$ are interpret as follows: an underscore means that we want to skip the argument that has the same type as the next real argument provided (real means not an underscore). If there is no real argument after underscore, then the underscore is used for the first possible argument.

The general syntax is $tactic\ (\gg E1 \dots EN)$ where $tactic$ is the name of the tactic (possibly with some arguments) and Ei are the arguments. Moreover, some tactics accept the syntax $tactic\ E1 \dots EN$ as short for $tactic\ (\gg E1 \dots EN)$ for values of N up to 5.

Finally, if the argument EN given is a triple-underscore $---$, then it is equivalent to providing a list of wildcards, with the appropriate number of wildcards. This means that all the remaining arguments of the lemma will be instantiated. Definitions in the conclusion are not unfolded in this case.

```
Ltac app_assert t P cont :=
  let H := fresh "TEMP" in
  assert (H : P); [ | cont(t H); clear H ].

Ltac app_evar t A cont :=
  let x := fresh "TEMP" in
  evar (x:A);
  let t' := constr:(t x) in
  let t'' := (eval unfold x in t') in
  subst x; cont t''.

Ltac app_arg t P v cont :=
  let H := fresh "TEMP" in
  assert (H : P); [ apply v | cont(t H); try clear H ].

Ltac build_app_all t final :=
  let rec go t :=
    match type of t with
    | ?P → ?Q ⇒ app_assert t P go
    | ∀ _ : ?A, _ ⇒ app_evar t A go
    | _ ⇒ final t
  end in
  go t.
```

```

Ltac boxerlist_next_type vs :=
  match vs with
  | nil ⇒ constr:(ltac_wild)
  | (boxer ltac_wild)::?vs' ⇒ boxerlist_next_type vs'
  | (boxer ltac_wilds)::_ ⇒ constr:(ltac_wild)
  | (@boxer ?T _)::_ ⇒ constr:(T)
  end.

Ltac build_app_hnts t vs final :=
  let rec go t vs :=
    match vs with
    | nil ⇒ first [ final t | fail 1 ]
    | (boxer ltac_wilds)::_ ⇒ first [ build_app_alls t final | fail 1 ]
    | (boxer ?v)::?vs' ⇒
      let cont t' := go t' vs in
      let cont' t' := go t' vs' in
      let T := type of t in
      let T := eval hnf in T in
      match v with
      | ltac_wild ⇒
        first [ let U := boxerlist_next_type vs' in
          match U with
          | ltac_wild ⇒
            match T with
            | ?P → ?Q ⇒ first [ app_assert t P cont' | fail 3 ]
            | ∀ _:?A, _ ⇒ first [ app_evar t A cont' | fail 3 ]
            end
          | _ ⇒
            match T with
            | U → ?Q ⇒ first [ app_assert t U cont' | fail 3 ]
            | ∀ _:U, _ ⇒ first [ app_evar t U cont' | fail 3 ]
            | ?P → ?Q ⇒ first [ app_assert t P cont | fail 3 ]
            | ∀ _:?A, _ ⇒ first [ app_evar t A cont | fail 3 ]
            end
          end
        | fail 2 ]
      | _ ⇒
        match T with
        | ?P → ?Q ⇒ first [ app_arg t P v cont'
          | app_assert t P cont
          | fail 3 ]
        | ∀ _:Type, _ ⇒
          match type of v with

```

```

      | Type ⇒ first [ cont' (t v)
                      | app_evar t Type cont
                      | fail 3 ]
      | _ ⇒ first [ app_evar t Type cont
                  | fail 3 ]
    end
  | ∀ _ : ?A, _ ⇒
    let V := type of v in
    match type of V with
    | Prop ⇒ first [ app_evar t A cont
                  | fail 3 ]
    | _ ⇒ first [ cont' (t v)
                | app_evar t A cont
                | fail 3 ]
    end
  end
end in
go t vs.

```

newer version : support for typeclasses

```

Ltac app_typeclass t cont :=
  let t' := constr:(t _) in
  cont t'.

```

```

Ltac build_app_alls t final ::=
  let rec go t :=
    match type of t with
    | ?P → ?Q ⇒ app_assert t P go
    | ∀ _ : ?A, _ ⇒
      first [ app_evar t A go
            | app_typeclass t go
            | fail 3 ]
    | _ ⇒ final t
  end in
  go t.

```

```

Ltac build_app_hnts t vs final ::=
  let rec go t vs :=
    match vs with
    | nil ⇒ first [ final t | fail 1 ]
    | (boxer ltac_wilds)::_ ⇒ first [ build_app_alls t final | fail 1 ]
    | (boxer ?v)::?vs' ⇒
      let cont t' := go t' vs in

```

```

let cont' t' := go t' vs' in
let T := type of t in
let T := eval hnf in T in
match v with
| ltac_wild =>
  first [ let U := boxerlist_next_type vs' in
    match U with
    | ltac_wild =>
      match T with
      | ?P → ?Q => first [ app_assert t P cont' | fail 3 ]
      | ∀ _:?A, _ => first [ app_typeclass t cont'
                           | app_evar t A cont'
                           | fail 3 ]
      end
    | _ =>
      match T with
      | U → ?Q => first [ app_assert t U cont' | fail 3 ]
      | ∀ _:U, _ => first
        [ app_typeclass t cont'
        | app_evar t U cont'
        | fail 3 ]
      | ?P → ?Q => first [ app_assert t P cont | fail 3 ]
      | ∀ _:?A, _ => first
        [ app_typeclass t cont
        | app_evar t A cont
        | fail 3 ]
      end
    end
  | fail 2 ]
| _ =>
  match T with
  | ?P → ?Q => first [ app_arg t P v cont'
                      | app_assert t P cont
                      | fail 3 ]
  | ∀ _:Type, _ =>
    match type of v with
    | Type => first [ cont' (t v)
                    | app_evar t Type cont
                    | fail 3 ]
    | _ => first [ app_evar t Type cont
                  | fail 3 ]
    end
  end

```

```

|  $\forall \_ : ?A, \_ \Rightarrow$ 
  let  $V :=$  type of  $v$  in
  match type of  $V$  with
  | Prop  $\Rightarrow$  first [  $app\_typeclass\ t\ cont$ 
                       |  $app\_eval\ t\ A\ cont$ 
                       | fail 3 ]
  |  $\_ \Rightarrow$  first [  $cont' (t\ v)$ 
                   |  $app\_typeclass\ t\ cont$ 
                   |  $app\_eval\ t\ A\ cont$ 
                   | fail 3 ]
  end
end
end
end in
go  $t\ vs.$ 

Ltac build_app args final :=
  first [
    match args with (@boxer ? $T$  ? $t$ ) :: ? $vs \Rightarrow$ 
      let  $t :=$  constr:( $t:T$ ) in
      build_app_hnts  $t\ vs\ final$ ;
      fast_rm_inside args
    end
  | fail 1 "Instantiation fails for:" args].

Ltac unfold_head_until_product  $T :=$ 
  eval hnf in  $T$ .

Ltac args_unfold_head_if_not_product args :=
  match args with (@boxer ? $T$  ? $t$ ) :: ? $vs \Rightarrow$ 
    let  $T' :=$  unfold_head_until_product  $T$  in
    constr:(@boxer  $T'\ t$ ) ::  $vs$ )
  end.

Ltac args_unfold_head_if_not_product_but_params args :=
  match args with
  | (boxer ? $t$ ) :: (boxer ? $v$ ) :: ? $vs \Rightarrow$ 
    args_unfold_head_if_not_product args
  |  $\_ \Rightarrow$  constr:(args)
  end.

```

lets H : ($\gg E0\ E1\ \dots\ EN$) will instantiate lemma $E0$ on the arguments Ei (which may be wildcards $\_$), and name H the resulting term. H may be an introduction pattern, or a sequence of introduction patterns $I1\ I2\ \dots\ IN$, or empty. Syntax *lets* H : $E0\ E1\ \dots\ EN$ is also available. If the last argument EN is $\_ \_ \_$ (triple-underscore), then all arguments of H will be instantiated.


```

Ltac lets_build I Ei :=
  let args := list_boxer_of Ei in
  let args := args_unfold_head_if_not_product_but_params args in

  build_app args ltac:(fun R => lets_base I R).

Tactic Notation "lets" simple_intropattern(I) ":" constr(E) :=
  lets_build I E.

Tactic Notation "lets" ":" constr(E) :=
  let H := fresh in lets H: E.

Tactic Notation "lets" ":" constr(E0)
  constr(A1) :=
  lets: (» E0 A1).

Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) :=
  lets: (» E0 A1 A2).

Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets: (» E0 A1 A2 A3).

Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: (» E0 A1 A2 A3 A4).

Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: (» E0 A1 A2 A3 A4 A5).

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  ":" constr(E) :=
  lets [I1 I2]: E.

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) ":" constr(E) :=
  lets [I1 [I2 I3]]: E.

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) ":" constr(E) :=
  lets [I1 [I2 [I3 I4]]]: E.

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  ":" constr(E) :=
  lets [I1 [I2 [I3 [I4 I5]]]]: E.

Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  lets I: (» E0 A1).

Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=

```

```

    lets I: (» E0 A1 A2).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) :=
    lets I: (» E0 A1 A2 A3).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) :=
    lets I: (» E0 A1 A2 A3 A4).
Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    lets I: (» E0 A1 A2 A3 A4 A5).

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
    constr(A1) :=
    lets [I1 I2]: E0 A1.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
    constr(A1) constr(A2) :=
    lets [I1 I2]: E0 A1 A2.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) :=
    lets [I1 I2]: E0 A1 A2 A3.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) :=
    lets [I1 I2]: E0 A1 A2 A3 A4.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    lets [I1 I2]: E0 A1 A2 A3 A4 A5.

```

forwards H: (» *E0 E1 .. EN*) is short for *forwards H*: (» *E0 E1 .. EN ___*). The arguments *Ei* can be wildcards *__* (except *E0*). *H* may be an introduction pattern, or a sequence of introduction pattern, or empty. Syntax *forwards H*: *E0 E1 .. EN* is also available.

```

Ltac forwards_build_app_arg Ei :=
    let args := list_boxer_of Ei in
    let args := (eval simpl in (args ++ ((boxer ___)::nil))) in
    let args := args_unfold_head_if_not_product args in
    args.

Ltac forwards_then Ei cont :=
    let args := forwards_build_app_arg Ei in
    let args := args_unfold_head_if_not_product_but_params args in
    build_app args cont.

Tactic Notation "forwards" simple_intropattern(I) ":" constr(Ei) :=
    let args := forwards_build_app_arg Ei in
    lets I: args.

```

```

Tactic Notation "forwards" ":" constr(E) :=
  let H := fresh in forwards H: E.
Tactic Notation "forwards" ":" constr(E0)
  constr(A1) :=
  forwards: (» E0 A1).
Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) :=
  forwards: (» E0 A1 A2).
Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  forwards: (» E0 A1 A2 A3).
Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: (» E0 A1 A2 A3 A4).
Tactic Notation "forwards" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards: (» E0 A1 A2 A3 A4 A5).

Tactic Notation "forwards" simple_intropattern(I1) simple_intropattern(I2)
  ":" constr(E) :=
  forwards [I1 I2]: E.
Tactic Notation "forwards" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) ":" constr(E) :=
  forwards [I1 [I2 I3]]: E.
Tactic Notation "forwards" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) ":" constr(E) :=
  forwards [I1 [I2 [I3 I4]]]: E.
Tactic Notation "forwards" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  ":" constr(E) :=
  forwards [I1 [I2 [I3 [I4 I5]]]]: E.

Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  forwards I: (» E0 A1).
Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
  forwards I: (» E0 A1 A2).
Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  forwards I: (» E0 A1 A2 A3).
Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards I: (» E0 A1 A2 A3 A4).

```

Tactic Notation "forwards" *simple_intropattern*(*I*) ":" *constr*(*E0*)
constr(*A1*) *constr*(*A2*) *constr*(*A3*) *constr*(*A4*) *constr*(*A5*) :=
forwards *I*: (» *E0* *A1* *A2* *A3* *A4* *A5*).

forwards_nounfold *I*: *E* is like *forwards* *I*: *E* but does not unfold the head constant of *E* if there is no visible quantification or hypothesis in *E*. It is meant to be used mainly by tactics.

Tactic Notation "forwards_nounfold" *simple_intropattern*(*I*) ":" *constr*(*Ei*) :=
 let *args* := *list_boxer_of* *Ei* in
 let *args* := (eval simpl in (*args* ++ ((boxer ___)::nil))) in
 build_app *args* ltac:(fun *R* ⇒ *lets_base* *I* *R*).

forwards_nounfold_then *E* ltac:(fun *K* ⇒ ..) is like *forwards*: *E* but it provides the resulting term to a continuation, under the name *K*.

Ltac *forwards_nounfold_then* *Ei* *cont* :=
 let *args* := *list_boxer_of* *Ei* in
 let *args* := (eval simpl in (*args* ++ ((boxer ___)::nil))) in
 build_app *args* *cont*.

applies (» *E0* *E1* .. *EN*) instantiates lemma *E0* on the arguments *Ei* (which may be wildcards *--*), and apply the resulting term to the current goal, using the tactic *applies* defined earlier on. *applies* *E0* *E1* *E2* .. *EN* is also available.

Ltac *applies_build* *Ei* :=
 let *args* := *list_boxer_of* *Ei* in
 let *args* := *args_unfold_head_if_not_product_but_params* *args* in
 build_app *args* ltac:(fun *R* ⇒
 first [apply *R* | eapply *R* | rapply *R*]).

Ltac *applies_base* *E* :=
 match type of *E* with
 | list Boxer ⇒ *applies_build* *E*
 | _ ⇒ first [eapply *E* | rapply *E* | *applies_build* *E*]
 end; fast_rm_inside *E*.

Tactic Notation "applies" *constr*(*E*) :=
applies_base *E*.

Tactic Notation "applies" *constr*(*E0*) *constr*(*A1*) :=
applies (» *E0* *A1*).

Tactic Notation "applies" *constr*(*E0*) *constr*(*A1*) *constr*(*A2*) :=
applies (» *E0* *A1* *A2*).

Tactic Notation "applies" *constr*(*E0*) *constr*(*A1*) *constr*(*A2*) *constr*(*A3*) :=
applies (» *E0* *A1* *A2* *A3*).

Tactic Notation "applies" *constr*(*E0*) *constr*(*A1*) *constr*(*A2*) *constr*(*A3*) *constr*(*A4*)
:=
applies (» *E0* *A1* *A2* *A3* *A4*).

Tactic Notation "applies" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*)
constr(*A5*) :=

applies (» *E0 A1 A2 A3 A4 A5*).

fapplies (» *E0 E1 .. EN*) instantiates lemma *E0* on the arguments *Ei* and on the argument
--- meaning that all evvars should be explicitly instantiated, and apply the resulting term to
the current goal. *fapplies E0 E1 E2 .. EN* is also available.

Ltac *fapplies_build Ei* :=

```
let args := list_boxer_of Ei in
let args := (eval simpl in (args ++ ((boxer ___)::nil))) in
let args := args_unfold_head_if_not_product_but_params args in
build_app args ltac:(fun R => apply R).
```

Tactic Notation "fapplies" constr(*E0*) :=

```
match type of E0 with
| list Boxer => fapplies_build E0
| _ => fapplies_build (» E0)
end.
```

Tactic Notation "fapplies" constr(*E0*) constr(*A1*) :=

fapplies (» *E0 A1*).

Tactic Notation "fapplies" constr(*E0*) constr(*A1*) constr(*A2*) :=

fapplies (» *E0 A1 A2*).

Tactic Notation "fapplies" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) :=

fapplies (» *E0 A1 A2 A3*).

Tactic Notation "fapplies" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*)
:=

fapplies (» *E0 A1 A2 A3 A4*).

Tactic Notation "fapplies" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*)
constr(*A5*) :=

fapplies (» *E0 A1 A2 A3 A4 A5*).

specializes H (» *E1 E2 .. EN*) will instantiate hypothesis *H* on the arguments *Ei* (which
may be wildcards *--*). If the last argument *EN* is *---* (triple-underscore), then all arguments
of *H* get instantiated.

Ltac *specializes_build H Ei* :=

```
let H' := fresh "TEMP" in rename H into H';
let args := list_boxer_of Ei in
let args := constr:((boxer H')::args) in
let args := args_unfold_head_if_not_product args in
build_app args ltac:(fun R => lets H: R);
clear H'.
```

Ltac *specializes_base H Ei* :=

specializes_build H Ei; *fast_rm_inside Ei*.

Tactic Notation "specializes" *hyp(H)* :=

```

specializes_base H (---).
Tactic Notation "specializes" hyp(H) constr(A) :=
  specializes_base H A.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H (» A1 A2).
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H (» A1 A2 A3).
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  specializes H (» A1 A2 A3 A4).
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  specializes H (» A1 A2 A3 A4 A5).

```

specializes_vars *H* is equivalent to *specializes* *H* *-- .. --* with as many double underscore as the number of dependent arguments visible from the type of *H*. Note that no unfolding is currently being performed (this behavior might change in the future). The current implementation is restricted to the case where *H* is an existing hypothesis – Note: this could be generalized.

```

Ltac specializes_var_base H :=
  match type of H with
  | ?P → ?Q ⇒ fail 1
  | ∀ _:-, _ ⇒ specializes H --
  end.
Ltac specializes_vars_base H :=
  repeat (specializes_var_base H).
Tactic Notation "specializes_var" hyp(H) :=
  specializes_var_base H.
Tactic Notation "specializes_vars" hyp(H) :=
  specializes_vars_base H.

```

2.5.4 Experimental Tactics for Application

fapply is a version of *apply* based on *forwards*.

```

Tactic Notation "fapply" constr(E) :=
  let H := fresh "TEMP" in forwards H: E;
  first [ apply H | eapply H | rapply H | hnf; apply H
         | hnf; eapply H | applies H ].

```

sapply stands for “super apply”. It tries *apply*, *eapply*, *applies* and *fapply*, and also tries to head-normalize the goal first.

```

Tactic Notation "sapply" constr(H) :=

```

```

first [ apply H | eapply H | rapply H | applies H
      | hnf; apply H | hnf; eapply H | hnf; applies H
      | fapply H ].

```

2.5.5 Adding Assumptions

lets_simpl *H*: *E* is the same as *lets* *H*: *E* excepts that it calls *simpl* on the hypothesis *H*.
lets_simpl: *E* is also provided.

```

Tactic Notation "lets_simpl" ident(H) ":" constr(E) :=
  lets H: E; try simpl in H.

```

```

Tactic Notation "lets_simpl" ":" constr(T) :=
  let H := fresh "TEMP" in lets_simpl H: T.

```

lets_hnf *H*: *E* is the same as *lets* *H*: *E* excepts that it calls *hnf* to set the definition in head normal form. *lets_hnf*: *E* is also provided.

```

Tactic Notation "lets_hnf" ident(H) ":" constr(E) :=
  lets H: E; hnf in H.

```

```

Tactic Notation "lets_hnf" ":" constr(T) :=
  let H := fresh "TEMP" in lets_hnf H: T.

```

puts *X*: *E* is a synonymous for *pose* (*X* := *E*). Alternative syntax is *puts*: *E*.

```

Tactic Notation "puts" ident(X) ":" constr(E) :=
  pose (X := E).

```

```

Tactic Notation "puts" ":" constr(E) :=
  let X := fresh "X" in pose (X := E).

```

2.5.6 Application of Tautologies

logic *E*, where *E* is a fact, is equivalent to *assert* *H*:*E*; [*tauto* | *eapply* *H*; *clear* *H*]. It is useful for instance to prove a conjunction *A* ∧ *B* by showing first *A* and then *A* → *B*, through the command *logic* (*foral* *A* *B*, *A* → (*A* → *B*) → *A* ∧ *B*)

```

Ltac logic_base E cont :=
  assert (H:E); [ cont tt | eapply H; clear H ].

```

```

Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ => tauto).

```

2.5.7 Application Modulo Equalities

The tactic *equates* replaces a goal of the form *P* *x* *y* *z* with a goal of the form *P* *x* ?*a* *z* and a subgoal ?*a* = *y*. The introduction of the evar ?*a* makes it possible to apply lemmas that would not apply to the original goal, for example a lemma of the form ∀ *n* *m*, *P* *n* *n* *m*, because *x* and *y* might be equal but not convertible.

Usage is *equates i1 ... ik*, where the indices are the positions of the arguments to be replaced by evars, counting from the right-hand side. If 0 is given as argument, then the entire goal is replaced by an evar.

Section *equatesLemma*.

Variables (*A0 A1* : Type).

Variables (*A2* : $\forall (x1 : A1), \text{Type}$).

Variables (*A3* : $\forall (x1 : A1) (x2 : A2 x1), \text{Type}$).

Variables (*A4* : $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), \text{Type}$).

Variables (*A5* : $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), \text{Type}$).

Variables (*A6* : $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), \text{Type}$).

Lemma *equates_0* : $\forall (P Q : \text{Prop})$,

$P \rightarrow P = Q \rightarrow Q$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_1* :

$\forall (P : A0 \rightarrow \text{Prop}) x1 y1$,

$P y1 \rightarrow x1 = y1 \rightarrow P x1$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_2* :

$\forall y1 (P : A0 \rightarrow \forall (x1 : A1), \text{Prop}) x1 x2$,

$P y1 x2 \rightarrow x1 = y1 \rightarrow P x1 x2$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_3* :

$\forall y1 (P : A0 \rightarrow \forall (x1 : A1) (x2 : A2 x1), \text{Prop}) x1 x2 x3$,

$P y1 x2 x3 \rightarrow x1 = y1 \rightarrow P x1 x2 x3$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_4* :

$\forall y1 (P : A0 \rightarrow \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), \text{Prop}) x1 x2 x3 x4$,

$P y1 x2 x3 x4 \rightarrow x1 = y1 \rightarrow P x1 x2 x3 x4$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_5* :

$\forall y1 (P : A0 \rightarrow \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), \text{Prop}) x1 x2 x3 x4 x5$,

$P y1 x2 x3 x4 x5 \rightarrow x1 = y1 \rightarrow P x1 x2 x3 x4 x5$.

Proof using. intros. subst. auto. Qed.

Lemma *equates_6* :

$\forall y1 (P : A0 \rightarrow \forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), \text{Prop})$

$x1 x2 x3 x4 x5 x6$,

$P y1 x2 x3 x4 x5 x6 \rightarrow x1 = y1 \rightarrow P x1 x2 x3 x4 x5 x6$.

Proof using. intros. subst. auto. Qed.

End *equatesLemma*.

Ltac *equates_lemma n* :=


```

match number_to_nat n with
| 0 ⇒ constr:(equates_0)
| 1 ⇒ constr:(equates_1)
| 2 ⇒ constr:(equates_2)
| 3 ⇒ constr:(equates_3)
| 4 ⇒ constr:(equates_4)
| 5 ⇒ constr:(equates_5)
| 6 ⇒ constr:(equates_6)
| _ ⇒ fail 100 "equates tactic only support up to arity 6"
end.

Ltac equates_one n :=
  let L := equates_lemma n in
  eapply L.

Ltac equates_several E cont :=
  let all_pos := match type of E with
  | List.list Boxer ⇒ constr:(E)
  | _ ⇒ constr:((boxer E)::nil)
  end in
  let rec go pos :=
    match pos with
    | nil ⇒ cont tt
    | (boxer ?n)::?pos' ⇒ equates_one n; [ go pos' | ]
    end in
  go all_pos.

Tactic Notation "equates" constr(E) :=
  equates_several E ltac:(fun _ ⇒ idtac).
Tactic Notation "equates" constr(n1) constr(n2) :=
  equates (» n1 n2).
Tactic Notation "equates" constr(n1) constr(n2) constr(n3) :=
  equates (» n1 n2 n3).
Tactic Notation "equates" constr(n1) constr(n2) constr(n3) constr(n4) :=
  equates (» n1 n2 n3 n4).

applies_eq H i1 .. iK is the same as equates i1 .. iK followed by applies H on the first
subgoal.
DEPRECATED: use applies_eq H instead.

Tactic Notation "applies_eq" constr(H) constr(E) :=
  equates_several E ltac:(fun _ ⇒ sapply H).
Tactic Notation "applies_eq" constr(H) constr(n1) constr(n2) :=
  applies_eq H (» n1 n2).
Tactic Notation "applies_eq" constr(H) constr(n1) constr(n2) constr(n3) :=
  applies_eq H (» n1 n2 n3).

```

Tactic Notation "applys_eq" constr(*H*) constr(*n1*) constr(*n2*) constr(*n3*) constr(*n4*)
:=

applys_eq *H* (» *n1* *n2* *n3* *n4*).

applys_eq *H* helps proving a goal of the form $P \ x1 \dots xN$ from an hypothesis *H* that concludes $P \ y1 \dots yN$, where the arguments *xi* and *yi* may or may not be convertible. Equalities are produced for all arguments that don't unify.

The tactic invokes *equates* on all arguments, then calls *applys* *K*, and attempts *reflexivity* on the side equalities.

Lemma *applys_eq_init* : $\forall (P \ Q:\text{Prop})$,

$P = Q \rightarrow$

$Q \rightarrow$

P.

Proof using. intros. subst. auto. Qed.

Lemma *applys_eq_step_dep* : $\forall B (P \ Q: (\forall A, A \rightarrow B)) (T:\text{Type})$,

$P = Q \rightarrow$

$P \ T = Q \ T$.

Proof using. intros. subst. auto. Qed.

Lemma *applys_eq_step* : $\forall A \ B (P \ Q:A \rightarrow B) \ x \ y$,

$P = Q \rightarrow$

$x = y \rightarrow$

$P \ x = Q \ y$.

Proof using. intros. subst. auto. Qed.

Ltac *applys_eq_loop* *tt* :=

match goal with

| $\vdash ?P \ ?x \Rightarrow$

first [eapply *applys_eq_step*; [*applys_eq_loop* *tt* |]

| eapply *applys_eq_step_dep*; *applys_eq_loop* *tt*]

| $\_ \Rightarrow$ *reflexivity*

end.

Ltac *applys_eq_core* *H* :=

eapply *applys_eq_init*;

[*applys_eq_loop* *tt* | *applys* *H*];

try *reflexivity*.

Tactic Notation "applys_eq" constr(*H*) :=

applys_eq_core *H*.

2.5.8 Absurd Goals

false_goal replaces any goal by the goal **False**. Contrary to the tactic *false* (below), it does not try to do anything else

Tactic Notation "false_goal" :=
exfalse.

false_post is the underlying tactic used to prove goals of the form **False**. In the default implementation, it proves the goal if the context contains **False** or an hypothesis of the form $C \ x1 \dots xN = D \ y1 \dots yM$, or if the *congruence* tactic finds a proof of $x \neq x$ for some x .

Ltac *false_post* :=
 solve [*assumption* | *discriminate* | *congruence*].
false replaces any goal by the goal **False**, and calls *false_post*

Tactic Notation "false" :=
false_goal; try *false_post*.

tryfalse tries to solve a goal by contradiction, and leaves the goal unchanged if it cannot solve it. It is equivalent to *try solve* \[*false* \].

Tactic Notation "tryfalse" :=
 try solve [*false*].

false E tries to exploit lemma E to prove the goal false. *false E1 .. EN* is equivalent to *false* ($\gg E1 \dots EN$), which tries to apply *applys* ($\gg E1 \dots EN$) and if it does not work then tries *forwards H*: ($\gg E1 \dots EN$) followed with *false*

Ltac *false_then E cont* :=
false_goal; first
 [*applys E*
 | *forwards_then E ltac*:(fun $M \Rightarrow$
 pose M ; *jauto_set_hyps*; *intros*; *false*)];
cont tt.

Tactic Notation "false" constr(E) :=
false_then E ltac:(fun $_ \Rightarrow$ *idtac*).

Tactic Notation "false" constr(E) constr($E1$) :=
false ($\gg E \ E1$).

Tactic Notation "false" constr(E) constr($E1$) constr($E2$) :=
false ($\gg E \ E1 \ E2$).

Tactic Notation "false" constr(E) constr($E1$) constr($E2$) constr($E3$) :=
false ($\gg E \ E1 \ E2 \ E3$).

Tactic Notation "false" constr(E) constr($E1$) constr($E2$) constr($E3$) constr($E4$) :=
false ($\gg E \ E1 \ E2 \ E3 \ E4$).

false_invert H proves a goal if it absurd after calling *inversion H* and *false*

Ltac *false_invert_for H* :=
 let $M :=$ fresh "TEMP" in pose ($M := H$); *inversion H*; *false*.

Tactic Notation "false_invert" constr(H) :=
 try solve [*false_invert_for H* | *false*].

false_invert proves any goal provided there is at least one hypothesis *H* in the context (or as a universally quantified hypothesis visible at the head of the goal) that can be proved absurd by calling *inversion H*.

```
Ltac false_invert_iter :=
  match goal with H:_ ⊢ _ ⇒
    first [ solve [ inversion H; false ]
           | clear H; false_invert_iter ] end.
```

```
Tactic Notation "false_invert" :=
  intros; solve [ false_invert_iter | false ].
```

tryfalse_invert H and *tryfalse_invert* are like the above but leave the goal unchanged if they don't solve it.

```
Tactic Notation "tryfalse_invert" constr(H) :=
  try (false_invert H).
```

```
Tactic Notation "tryfalse_invert" :=
  try false_invert.
```

false_neq_self_hyp proves any goal if the context contains an hypothesis of the form *E* ≠ *E*. It is a restricted and optimized version of *false*. It is intended to be used by other tactics only.

```
Ltac false_neq_self_hyp :=
  match goal with H: ?x ≠ ?x ⊢ _ ⇒
    false_goal; apply H; reflexivity end.
```

2.6 Introduction and Generalization

2.6.1 Introduction

introv is used to name only non-dependent hypothesis.

- If *introv* is called on a goal of the form $\forall x, H$, it should introduce all the variables quantified with a $\forall$ at the head of the goal, but it does not introduce hypotheses that precede an arrow constructor, like in $P \rightarrow Q$.
- If *introv* is called on a goal that is not of the form $\forall x, H$ nor $P \rightarrow Q$, the tactic unfolds definitions until the goal takes the form $\forall x, H$ or $P \rightarrow Q$. If unfolding definitions does not produces a goal of this form, then the tactic *introv* does nothing at all.

```
Ltac introv_rec :=
  match goal with
  | ⊢ ?P → ?Q ⇒ idtac
  | ⊢ ∀ _, _ ⇒ intro; introv_rec
```

```

|  $\vdash \_ \Rightarrow$  idtac
end.

Ltac introv_noarg :=
  match goal with
  |  $\vdash ?P \rightarrow ?Q \Rightarrow$  idtac
  |  $\vdash \forall \_, \_ \Rightarrow$  introv_rec
  |  $\vdash ?G \Rightarrow$  hnf;
    match goal with
    |  $\vdash ?P \rightarrow ?Q \Rightarrow$  idtac
    |  $\vdash \forall \_, \_ \Rightarrow$  introv_rec
    end
  |  $\vdash \_ \Rightarrow$  idtac
  end.

Ltac introv_noarg_not_optimized :=
  intro; match goal with  $H:\vdash \_ \Rightarrow$  revert  $H$  end; introv_rec.

Ltac introv_arg  $H$  :=
  hnf; match goal with
  |  $\vdash ?P \rightarrow ?Q \Rightarrow$  intros  $H$ 
  |  $\vdash \forall \_, \_ \Rightarrow$  intro; introv_arg  $H$ 
  end.

Tactic Notation "introv" :=
  introv_noarg.

Tactic Notation "introv" simple_intropattern( $I1$ ) :=
  introv_arg  $I1$ .

Tactic Notation "introv" simple_intropattern( $I1$ ) simple_intropattern( $I2$ ) :=
  introv  $I1$ ; introv  $I2$ .

Tactic Notation "introv" simple_intropattern( $I1$ ) simple_intropattern( $I2$ )
  simple_intropattern( $I3$ ) :=
  introv  $I1$ ; introv  $I2$   $I3$ .

Tactic Notation "introv" simple_intropattern( $I1$ ) simple_intropattern( $I2$ )
  simple_intropattern( $I3$ ) simple_intropattern( $I4$ ) :=
  introv  $I1$ ; introv  $I2$   $I3$   $I4$ .

Tactic Notation "introv" simple_intropattern( $I1$ ) simple_intropattern( $I2$ )
  simple_intropattern( $I3$ ) simple_intropattern( $I4$ ) simple_intropattern( $I5$ ) :=
  introv  $I1$ ; introv  $I2$   $I3$   $I4$   $I5$ .

Tactic Notation "introv" simple_intropattern( $I1$ ) simple_intropattern( $I2$ )
  simple_intropattern( $I3$ ) simple_intropattern( $I4$ ) simple_intropattern( $I5$ )
  simple_intropattern( $I6$ ) :=
  introv  $I1$ ; introv  $I2$   $I3$   $I4$   $I5$   $I6$ .

Tactic Notation "introv" simple_intropattern( $I1$ ) simple_intropattern( $I2$ )
  simple_intropattern( $I3$ ) simple_intropattern( $I4$ ) simple_intropattern( $I5$ )

```

```
simple_intropattern(I6) simple_intropattern(I7) :=
  introv I1; introv I2 I3 I4 I5 I6 I7.
```

```
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8) :=
  introv I1; introv I2 I3 I4 I5 I6 I7 I8.
```

```
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) :=
  introv I1; introv I2 I3 I4 I5 I6 I7 I8 I9.
```

```
Tactic Notation "introv" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) simple_intropattern(I10) :=
  introv I1; introv I2 I3 I4 I5 I6 I7 I8 I9 I10.
```

intros_all repeats *intro* as long as possible. Contrary to *intros*, it unfolds any definition on the way. Remark that it also unfolds the definition of negation, so applying *intros_all* to a goal of the form $\forall x, P\ x \rightarrow \neg Q$ will introduce *x* and *P x* and *Q*, and will leave **False** in the goal.

```
Tactic Notation "intros_all" :=
  repeat intro.
```

intros_hnf introduces an hypothesis and sets in head normal form

```
Tactic Notation "intro_hnf" :=
  intro; match goal with H: _  $\vdash$  _  $\Rightarrow$  hnf in H end.
```

2.6.2 Introduction using $\Rightarrow$ and $=\gg$

```
Ltac ltac_intros_post := idtac.
```

```
Tactic Notation "=>" :=
  intros.
```

```
Tactic Notation "=>" simple_intropattern(I1) :=
  intros I1.
```

```
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2) :=
  intros I1 I2.
```

```
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) :=
  intros I1 I2 I3.
```

```
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) :=
  intros I1 I2 I3 I4.
```

```

Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5) :=
  intros I1 I2 I3 I4 I5.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) :=
  intros I1 I2 I3 I4 I5 I6.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) :=
  intros I1 I2 I3 I4 I5 I6 I7.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8) :=
  intros I1 I2 I3 I4 I5 I6 I7 I8.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) :=
  intros I1 I2 I3 I4 I5 I6 I7 I8 I9.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
  simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
  simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
  simple_intropattern(I9) simple_intropattern(I10) :=
  intros I1 I2 I3 I4 I5 I6 I7 I8 I9 I10.

Ltac intro_nondeps_aux_special_intro G :=
  fail.

Ltac intro_nondeps_aux is_already_hnf :=
  match goal with
  | ⊢ (?P → ?Q) ⇒ idtac
  | ⊢ ?G → _ ⇒ intro_nondeps_aux_special_intro G;
    intros; intro_nondeps_aux true
  | ⊢ (∀ _,_) ⇒ intros ?; intro_nondeps_aux true
  | ⊢ _ ⇒
    match is_already_hnf with
    | true ⇒ idtac
    | false ⇒ hnf; intro_nondeps_aux true
    end
  end.

Ltac intro_nondeps tt := intro_nondeps_aux false.

Tactic Notation "=>>" :=

```

```

    intro_nondeps tt.
Tactic Notation "=>" simple_intropattern(I1) :=
=>; intros I1.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2) :=
=>; intros I1 I2.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) :=
=>; intros I1 I2 I3.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) :=
=>; intros I1 I2 I3 I4.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5) :=
=>; intros I1 I2 I3 I4 I5.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) :=
=>; intros I1 I2 I3 I4 I5 I6.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) simple_intropattern(I7) :=
=>; intros I1 I2 I3 I4 I5 I6 I7.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8) :=
=>; intros I1 I2 I3 I4 I5 I6 I7 I8.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
simple_intropattern(I9) :=
=>; intros I1 I2 I3 I4 I5 I6 I7 I8 I9.
Tactic Notation "=>" simple_intropattern(I1) simple_intropattern(I2)
simple_intropattern(I3) simple_intropattern(I4) simple_intropattern(I5)
simple_intropattern(I6) simple_intropattern(I7) simple_intropattern(I8)
simple_intropattern(I9) simple_intropattern(I10) :=
=>; intros I1 I2 I3 I4 I5 I6 I7 I8 I9 I10.

```

2.6.3 Generalization

gen $X1 \dots XN$ is a shorthand for calling `generalize dependent` successively on variables $XN \dots X1$. Note that the variables are generalized in reverse order, following the convention of the `generalize` tactic: it means that $X1$ will be the first quantified variable in the resulting goal.

Tactic Notation "gen" *ident*($X1$) :=
 generalize dependent $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) :=
 gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) :=
 gen $X3$; gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) *ident*($X4$) :=
 gen $X4$; gen $X3$; gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) *ident*($X4$) *ident*($X5$) :=
 gen $X5$; gen $X4$; gen $X3$; gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) *ident*($X4$) *ident*($X5$)
 ident($X6$) :=
 gen $X6$; gen $X5$; gen $X4$; gen $X3$; gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) *ident*($X4$) *ident*($X5$)
 ident($X6$) *ident*($X7$) :=
 gen $X7$; gen $X6$; gen $X5$; gen $X4$; gen $X3$; gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) *ident*($X4$) *ident*($X5$)
 ident($X6$) *ident*($X7$) *ident*($X8$) :=
 gen $X8$; gen $X7$; gen $X6$; gen $X5$; gen $X4$; gen $X3$; gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) *ident*($X4$) *ident*($X5$)
 ident($X6$) *ident*($X7$) *ident*($X8$) *ident*($X9$) :=
 gen $X9$; gen $X8$; gen $X7$; gen $X6$; gen $X5$; gen $X4$; gen $X3$; gen $X2$; gen $X1$.
 Tactic Notation "gen" *ident*($X1$) *ident*($X2$) *ident*($X3$) *ident*($X4$) *ident*($X5$)
 ident($X6$) *ident*($X7$) *ident*($X8$) *ident*($X9$) *ident*($X10$) :=
 gen $X10$; gen $X9$; gen $X8$; gen $X7$; gen $X6$; gen $X5$; gen $X4$; gen $X3$; gen $X2$; gen $X1$.

generalizes X is a shorthand for calling *generalize* X ; *clear* X . It is weaker than tactic
gen X since it does not support dependencies. It is mainly intended for writing tactics.
 Tactic Notation "generalizes" *hyp*(X) :=
 generalize X ; clear X .
 Tactic Notation "generalizes" *hyp*($X1$) *hyp*($X2$) :=
 generalizes $X1$; generalizes $X2$.
 Tactic Notation "generalizes" *hyp*($X1$) *hyp*($X2$) *hyp*($X3$) :=
 generalizes $X1$ $X2$; generalizes $X3$.
 Tactic Notation "generalizes" *hyp*($X1$) *hyp*($X2$) *hyp*($X3$) *hyp*($X4$) :=
 generalizes $X1$ $X2$ $X3$; generalizes $X4$.

2.6.4 Naming

sets X : E is the same as *set* ($X := E$) *in* *, that is, it replaces all occurrences of E by a
 fresh meta-variable X whose definition is E .

Tactic Notation "sets" *ident*(X) ":" *constr*(E) :=
 set ($X := E$) *in* *.

def_to_eq $E\ X\ H$ applies when $X := E$ is a local definition. It adds an assumption $H: X = E$ and then clears the definition of X . *def_to_eq_sym* is similar except that it generates the equality $H: E = X$.

```
Ltac def_to_eq X HX E :=
  assert (HX : X = E) by reflexivity; clearbody X.
```

```
Ltac def_to_eq_sym X HX E :=
  assert (HX : E = X) by reflexivity; clearbody X.
```

set_eq $X\ H: E$ generates the equality $H: X = E$, for a fresh name X , and replaces E by X in the current goal. Syntaxes *set_eq* $X: E$ and *set_eq* $: E$ are also available. Similarly, *set_eq* $\leftarrow X\ H: E$ generates the equality $H: E = X$.

sets_eq $X\ HX: E$ does the same but replaces E by X everywhere in the goal. *sets_eq* $X\ HX: E$ in H replaces in H . *set_eq* $X\ HX: E$ in $\vdash$ performs no substitution at all.

```
Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) :=
  set (X := E); def_to_eq X HX E.
```

```
Tactic Notation "set_eq" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in set_eq X HX: E.
```

```
Tactic Notation "set_eq" ":" constr(E) :=
  let X := fresh "X" in set_eq X: E.
```

```
Tactic Notation "set_eq" "<-" ident(X) ident(HX) ":" constr(E) :=
  set (X := E); def_to_eq_sym X HX E.
```

```
Tactic Notation "set_eq" "<-" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in set_eq ← X HX: E.
```

```
Tactic Notation "set_eq" "<-" ":" constr(E) :=
  let X := fresh "X" in set_eq ← X: E.
```

```
Tactic Notation "sets_eq" ident(X) ident(HX) ":" constr(E) :=
  set (X := E) in *; def_to_eq X HX E.
```

```
Tactic Notation "sets_eq" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in sets_eq X HX: E.
```

```
Tactic Notation "sets_eq" ":" constr(E) :=
  let X := fresh "X" in sets_eq X: E.
```

```
Tactic Notation "sets_eq" "<-" ident(X) ident(HX) ":" constr(E) :=
  set (X := E) in *; def_to_eq_sym X HX E.
```

```
Tactic Notation "sets_eq" "<-" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in sets_eq ← X HX: E.
```

```
Tactic Notation "sets_eq" "<-" ":" constr(E) :=
  let X := fresh "X" in sets_eq ← X: E.
```

```
Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) "in" hyp(H) :=
  set (X := E) in H; def_to_eq X HX E.
```

```
Tactic Notation "set_eq" ident(X) ":" constr(E) "in" hyp(H) :=
  let HX := fresh "EQ" X in set_eq X HX: E in H.
```

```
Tactic Notation "set_eq" ":" constr(E) "in" hyp(H) :=
```

```

let  $X$  := fresh "X" in set_eq  $X$ :  $E$  in  $H$ .
Tactic Notation "set_eq" "<-" ident( $X$ ) ident( $HX$ ) ":" constr( $E$ ) "in" hyp( $H$ ) :=
  set ( $X$  :=  $E$ ) in  $H$ ; def_to_eq_sym  $X$   $HX$   $E$ .
Tactic Notation "set_eq" "<-" ident( $X$ ) ":" constr( $E$ ) "in" hyp( $H$ ) :=
  let  $HX$  := fresh "EQ"  $X$  in set_eq  $X$   $HX$ :  $E$  in  $H$ .
Tactic Notation "set_eq" "<-" ":" constr( $E$ ) "in" hyp( $H$ ) :=
  let  $X$  := fresh "X" in set_eq  $X$ :  $E$  in  $H$ .
Tactic Notation "set_eq" ident( $X$ ) ident( $HX$ ) ":" constr( $E$ ) "in" "|-" :=
  set ( $X$  :=  $E$ ) in |-; def_to_eq_sym  $X$   $HX$   $E$ .
Tactic Notation "set_eq" ident( $X$ ) ":" constr( $E$ ) "in" "|-" :=
  let  $HX$  := fresh "EQ"  $X$  in set_eq  $X$   $HX$ :  $E$  in |-.
Tactic Notation "set_eq" ":" constr( $E$ ) "in" "|-" :=
  let  $X$  := fresh "X" in set_eq  $X$ :  $E$  in |-.
Tactic Notation "set_eq" "<-" ident( $X$ ) ident( $HX$ ) ":" constr( $E$ ) "in" "|-" :=
  set ( $X$  :=  $E$ ) in |-; def_to_eq_sym  $X$   $HX$   $E$ .
Tactic Notation "set_eq" "<-" ident( $X$ ) ":" constr( $E$ ) "in" "|-" :=
  let  $HX$  := fresh "EQ"  $X$  in set_eq  $X$   $HX$ :  $E$  in |-.
Tactic Notation "set_eq" "<-" ":" constr( $E$ ) "in" "|-" :=
  let  $X$  := fresh "X" in set_eq  $X$ :  $E$  in |-.

```

gen_eq X : E is a tactic whose purpose is to introduce equalities so as to work around the limitation of the *induction* tactic which typically loses information. *gen_eq* E as X replaces all occurrences of term E with a fresh variable X and the equality $X = E$ as extra hypothesis to the current conclusion. In other words a conclusion C will be turned into $(X = E) \rightarrow C$. *gen_eq*: E and *gen_eq*: E as X are also accepted.

```

Tactic Notation "gen_eq" ident( $X$ ) ":" constr( $E$ ) :=
  let  $EQ$  := fresh "EQ"  $X$  in sets_eq  $X$   $EQ$ :  $E$ ; revert  $EQ$ .
Tactic Notation "gen_eq" ":" constr( $E$ ) :=
  let  $X$  := fresh "X" in gen_eq  $X$ :  $E$ .
Tactic Notation "gen_eq" ":" constr( $E$ ) "as" ident( $X$ ) :=
  gen_eq  $X$ :  $E$ .
Tactic Notation "gen_eq" ident( $X1$ ) ":" constr( $E1$ ) ","
  ident( $X2$ ) ":" constr( $E2$ ) :=
  gen_eq  $X2$ :  $E2$ ; gen_eq  $X1$ :  $E1$ .
Tactic Notation "gen_eq" ident( $X1$ ) ":" constr( $E1$ ) ","
  ident( $X2$ ) ":" constr( $E2$ ) "," ident( $X3$ ) ":" constr( $E3$ ) :=
  gen_eq  $X3$ :  $E3$ ; gen_eq  $X2$ :  $E2$ ; gen_eq  $X1$ :  $E1$ .

```

sets_let X finds the first let-expression in the goal and names its body X . *sets_eq_let* X is similar, except that it generates an explicit equality. Tactics *sets_let* X in H and *sets_eq_let* X in H allow specifying a particular hypothesis (by default, the first one that contains a *let* is considered).

Known limitation: it does not seem possible to support naming of multiple let-in constructs inside a term, from ltac.

```
Ltac sets_let_base tac :=
  match goal with
  | ⊢ context[let _ := ?E in _] ⇒ tac E; cbv zeta
  | H: context[let _ := ?E in _] ⊢ _ ⇒ tac E; cbv zeta in H
  end.

Ltac sets_let_in_base H tac :=
  match type of H with context[let _ := ?E in _] ⇒
    tac E; cbv zeta in H end.

Tactic Notation "sets_let" ident(X) :=
  sets_let_base ltac:(fun E ⇒ sets X: E).
Tactic Notation "sets_let" ident(X) "in" hyp(H) :=
  sets_let_in_base H ltac:(fun E ⇒ sets X: E).
Tactic Notation "sets_eq_let" ident(X) :=
  sets_let_base ltac:(fun E ⇒ sets_eq X: E).
Tactic Notation "sets_eq_let" ident(X) "in" hyp(H) :=
  sets_let_in_base H ltac:(fun E ⇒ sets_eq X: E).
```

2.7 Rewriting

rewrites *E* is similar to *rewrite* except that it supports the *rm* directives to clear hypotheses on the fly, and that it supports a list of arguments in the form *rewrites* (» *E1 E2 E3*) to indicate that *forwards* should be invoked first before *rewrites* is called.

```
Ltac rewrites_base E cont :=
  match type of E with
  | List.list Boxer ⇒ forwards_then E cont
  | _ ⇒ cont E; fast_rm_inside E
  end.

Tactic Notation "rewrites" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M ).
Tactic Notation "rewrites" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in H).
Tactic Notation "rewrites" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in *).
Tactic Notation "rewrites" "<-" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M ).
Tactic Notation "rewrites" "<-" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in H).
Tactic Notation "rewrites" "<-" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in *).
```

erewrites *E* is similar to *rewrite* except that it supports the *rm* directives to clear hypotheses on the fly, and that it supports a list of arguments in the form *rewrites* (*» E1 E2 E3*) to indicate that *forwards* should be invoked first before *rewrites* is called.

```
Tactic Notation "erewrites" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M ).
Tactic Notation "erewrites" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in H ).
Tactic Notation "erewrites" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in *).
Tactic Notation "erewrites" "<-" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M ).
Tactic Notation "erewrites" "<-" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in H ).
Tactic Notation "erewrites" "<-" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite ← M in *).
```

rewrite_all *E* iterates version of *rewrite* *E* as long as possible. Warning: this tactic can easily get into an infinite loop. Syntax for rewriting from right to left and/or into an hypothesis is similar to the one of *rewrite*.

```
Tactic Notation "rewrite_all" constr(E) :=
  repeat rewrite E.
Tactic Notation "rewrite_all" "<-" constr(E) :=
  repeat rewrite ← E.
Tactic Notation "rewrite_all" constr(E) "in" ident(H) :=
  repeat rewrite E in H.
Tactic Notation "rewrite_all" "<-" constr(E) "in" ident(H) :=
  repeat rewrite ← E in H.
Tactic Notation "rewrite_all" constr(E) "in" "*" :=
  repeat rewrite E in *.
Tactic Notation "rewrite_all" "<-" constr(E) "in" "*" :=
  repeat rewrite ← E in *.
```

asserts_rewrite *E* asserts that an equality *E* holds (generating a corresponding subgoal) and rewrite it straight away in the current goal. It avoids giving a name to the equality and later clearing it. Syntax for rewriting from right to left and/or into an hypothesis is similar to the one of *rewrite*. Note: the tactic *replaces* plays a similar role.

```
Ltac asserts_rewrite_tactic E action :=
  let EQ := fresh "TEMP" in (assert (EQ : E);
  [ idtac | action EQ; clear EQ ]).
Tactic Notation "asserts_rewrite" constr(E) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ).
Tactic Notation "asserts_rewrite" "<-" constr(E) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite ← EQ).
```

```

Tactic Notation "asserts_rewrite" constr(E) "in" hyp(H) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ in H).
Tactic Notation "asserts_rewrite" "<-" constr(E) "in" hyp(H) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite ← EQ in H).
Tactic Notation "asserts_rewrite" constr(E) "in" "*" :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ in *).
Tactic Notation "asserts_rewrite" "<-" constr(E) "in" "*" :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite ← EQ in *).

```

cuts_rewrite E is the same as *asserts_rewrite E* except that subgoals are permuted.

```

Ltac cuts_rewrite_tactic E action :=
  let EQ := fresh "TEMP" in (cuts EQ: E;
  [ action EQ; clear EQ | idtac ]).
Tactic Notation "cuts_rewrite" constr(E) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ).
Tactic Notation "cuts_rewrite" "<-" constr(E) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite ← EQ).
Tactic Notation "cuts_rewrite" constr(E) "in" hyp(H) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ in H).
Tactic Notation "cuts_rewrite" "<-" constr(E) "in" hyp(H) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite ← EQ in H).

```

rewrite_except H EQ rewrites equality *EQ* everywhere but in hypothesis *H*. Mainly useful for other tactics.

```

Ltac rewrite_except H EQ :=
  let K := fresh "TEMP" in let T := type of H in
  set (K := T) in H;
  rewrite EQ in *; unfold K in H; clear K.

```

rewrites E at K applies when *E* is of the form *T1 = T2* rewrites the equality *E* at the *K*-th occurrence of *T1* in the current goal. Syntaxes *rewrites* ← *E at K* and *rewrites E at K in H* are also available.

```

Tactic Notation "rewrites" constr(E) "at" constr(K) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T1 do (rewrites E) end.
Tactic Notation "rewrites" "<-" constr(E) "at" constr(K) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T2 do (rewrites ← E) end.
Tactic Notation "rewrites" constr(E) "at" constr(K) "in" hyp(H) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T1 in H do (rewrites E in H) end.
Tactic Notation "rewrites" "<-" constr(E) "at" constr(K) "in" hyp(H) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T2 in H do (rewrites ← E in H) end.

```

replaces E with F is the same as *replace* E with F except that the equality $E = F$ is generated as first subgoal. Syntax *replaces* E with F in H is also available. Note that contrary to *replace*, *replaces* does not try to solve the equality by *assumption*. Note: *replaces* E with F is similar to *asserts_rewrite* ($E = F$).

Tactic Notation "replaces" *constr*(E) "with" *constr*(F) :=
 let $T := \text{fresh}$ "TEMP" in assert ($T: E = F$); [| *replace* E with F ; clear T].

Tactic Notation "replaces" *constr*(E) "with" *constr*(F) "in" *hyp*(H) :=
 let $T := \text{fresh}$ "TEMP" in assert ($T: E = F$); [| *replace* E with F in H ; clear T].

replaces E at K with F replaces the K -th occurrence of E with F in the current goal. Syntax *replaces* E at K with F in H is also available.

Tactic Notation "replaces" *constr*(E) "at" *constr*(K) "with" *constr*(F) :=
 let $T := \text{fresh}$ "TEMP" in assert ($T: E = F$); [| *rewrites* T at K ; clear T].

Tactic Notation "replaces" *constr*(E) "at" *constr*(K) "with" *constr*(F) "in" *hyp*(H) :=
 let $T := \text{fresh}$ "TEMP" in assert ($T: E = F$); [| *rewrites* T at K in H ; clear T].

changes is like *change* except that it does not silently fail to perform its task. (Note that, *changes* is implemented using *rewrite*, meaning that it might perform additional beta-reductions compared with the original *change* tactic.

Tactic Notation "changes" *constr*($E1$) "with" *constr*($E2$) "in" *hyp*(H) :=
asserts_rewrite ($E1 = E2$) in H ; [reflexivity |].

Tactic Notation "changes" *constr*($E1$) "with" *constr*($E2$) :=
asserts_rewrite ($E1 = E2$); [reflexivity |].

Tactic Notation "changes" *constr*($E1$) "with" *constr*($E2$) "in" "*" :=
asserts_rewrite ($E1 = E2$) in *; [reflexivity |].

2.7.1 Renaming

renames $X1$ to $Y1$, ..., XN to YN is a shorthand for a sequence of renaming operations *rename* Xi into Yi .

Tactic Notation "renames" *ident*($X1$) "to" *ident*($Y1$) :=
rename $X1$ into $Y1$.

Tactic Notation "renames" *ident*($X1$) "to" *ident*($Y1$) ", "
ident($X2$) "to" *ident*($Y2$) :=
renames $X1$ to $Y1$; *renames* $X2$ to $Y2$.

Tactic Notation "renames" *ident*($X1$) "to" *ident*($Y1$) ", "
ident($X2$) "to" *ident*($Y2$) ", "*ident*($X3$) "to" *ident*($Y3$) :=
renames $X1$ to $Y1$; *renames* $X2$ to $Y2$, $X3$ to $Y3$.

Tactic Notation "renames" *ident*($X1$) "to" *ident*($Y1$) ", "
ident($X2$) "to" *ident*($Y2$) ", "*ident*($X3$) "to" *ident*($Y3$) ", "


```

ident(X4) "to" ident(Y4) :=
  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4.
Tactic Notation "renames" ident(X1) "to" ident(Y1) ", "
ident(X2) "to" ident(Y2) ", " ident(X3) "to" ident(Y3) ", "
ident(X4) "to" ident(Y4) ", " ident(X5) "to" ident(Y5) :=
  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4, X5 to Y5.
Tactic Notation "renames" ident(X1) "to" ident(Y1) ", "
ident(X2) "to" ident(Y2) ", " ident(X3) "to" ident(Y3) ", "
ident(X4) "to" ident(Y4) ", " ident(X5) "to" ident(Y5) ", "
ident(X6) "to" ident(Y6) :=
  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4, X5 to Y5, X6 to Y6.

```

2.7.2 Unfolding

unfolds unfolds the head definition in the goal, i.e. if the goal has form $P \ x1 \dots xN$ then it calls `unfold P`. If the goal is an equality, it tries to unfold the head constant on the left-hand side, and otherwise tries on the right-hand side. If the goal is a product, it calls `intros` first. warning: this tactic is overridden in LibReflect.

```

Ltac apply_to_head_of E cont :=
  let go E :=
    let P := get_head E in cont P in
    match E with
    |  $\forall \_,\_ \Rightarrow$  intros; apply_to_head_of E cont
    |  $?A = ?B \Rightarrow$  first [ go A | go B ]
    |  $?A \Rightarrow$  go A
    end.
Ltac unfolds_base :=
  match goal with  $\vdash ?G \Rightarrow$ 
    apply_to_head_of G ltac:(fun P  $\Rightarrow$  unfold P) end.
Tactic Notation "unfolds" :=
  unfolds_base.

```

unfolds in H unfolds the head definition of hypothesis H , i.e. if H has type $P \ x1 \dots xN$ then it calls `unfold P in H`.

```

Ltac unfolds_in_base H :=
  match type of H with  $?G \Rightarrow$ 
    apply_to_head_of G ltac:(fun P  $\Rightarrow$  unfold P in H) end.

```

```

Tactic Notation "unfolds" "in" hyp(H) :=
  unfolds_in_base H.

```

unfolds in H1,H2,...,HN allows unfolding the head constant in several hypotheses at once.

Tactic Notation "unfolds" "in" *hyp*(*H1*) *hyp*(*H2*) :=
unfolds in *H1*; *unfolds* in *H2*.

Tactic Notation "unfolds" "in" *hyp*(*H1*) *hyp*(*H2*) *hyp*(*H3*) :=
unfolds in *H1*; *unfolds* in *H2* *H3*.

Tactic Notation "unfolds" "in" *hyp*(*H1*) *hyp*(*H2*) *hyp*(*H3*) *hyp*(*H4*) :=
unfolds in *H1*; *unfolds* in *H2* *H3* *H4*.

Tactic Notation "unfolds" "in" *hyp*(*H1*) *hyp*(*H2*) *hyp*(*H3*) *hyp*(*H4*) *hyp*(*H5*) :=
unfolds in *H1*; *unfolds* in *H2* *H3* *H4* *H5*.

unfolds P1,...,PN is a shortcut for *unfold P1,...,PN* in *.

Tactic Notation "unfolds" *constr*(*F1*) :=
unfold F1 in *.

Tactic Notation "unfolds" *constr*(*F1*) ", " *constr*(*F2*) :=
unfold F1,F2 in *.

Tactic Notation "unfolds" *constr*(*F1*) ", " *constr*(*F2*)
", " *constr*(*F3*) :=
unfold F1,F2,F3 in *.

Tactic Notation "unfolds" *constr*(*F1*) ", " *constr*(*F2*)
", " *constr*(*F3*) ", " *constr*(*F4*) :=
unfold F1,F2,F3,F4 in *.

Tactic Notation "unfolds" *constr*(*F1*) ", " *constr*(*F2*)
", " *constr*(*F3*) ", " *constr*(*F4*) ", " *constr*(*F5*) :=
unfold F1,F2,F3,F4,F5 in *.

Tactic Notation "unfolds" *constr*(*F1*) ", " *constr*(*F2*)
", " *constr*(*F3*) ", " *constr*(*F4*) ", " *constr*(*F5*)
", " *constr*(*F6*) ", " *constr*(*F7*) :=
unfold F1,F2,F3,F4,F5,F6,F7 in *.

Tactic Notation "unfolds" *constr*(*F1*) ", " *constr*(*F2*)
", " *constr*(*F3*) ", " *constr*(*F4*) ", " *constr*(*F5*)
", " *constr*(*F6*) ", " *constr*(*F7*) ", " *constr*(*F8*) :=
unfold F1,F2,F3,F4,F5,F6,F7,F8 in *.

folds P1,...,PN is a shortcut for *fold P1* in *; ..; *fold PN* in *.

Tactic Notation "folds" *constr*(*H*) :=
fold H in *.

Tactic Notation "folds" *constr*(*H1*) ", " *constr*(*H2*) :=
folds H1; *folds H2*.

Tactic Notation "folds" *constr*(*H1*) ", " *constr*(*H2*) ", " *constr*(*H3*) :=
folds H1; *folds H2*; *folds H3*.

Tactic Notation "folds" *constr*(*H1*) ", " *constr*(*H2*) ", " *constr*(*H3*)

```

", " constr(H4) :=
  folds H1; folds H2; folds H3; folds H4.
Tactic Notation "folds" constr(H1) ", " constr(H2) ", " constr(H3)
", " constr(H4) ", " constr(H5) :=
  folds H1; folds H2; folds H3; folds H4; folds H5.

```

2.7.3 Simplification

simpls is a shortcut for `simpl in *`.

```

Tactic Notation "simpls" :=
  simpl in *.

```

simpls P1,...,PN is a shortcut for `simpl P1 in *; ..; simpl PN in *`.

```

Tactic Notation "simpls" constr(F1) :=
  simpl F1 in *.

```

```

Tactic Notation "simpls" constr(F1) ", " constr(F2) :=
  simpls F1; simpls F2.

```

```

Tactic Notation "simpls" constr(F1) ", " constr(F2)
", " constr(F3) :=
  simpls F1; simpls F2; simpls F3.

```

```

Tactic Notation "simpls" constr(F1) ", " constr(F2)
", " constr(F3) ", " constr(F4) :=
  simpls F1; simpls F2; simpls F3; simpls F4.

```

unsimpl E replaces all occurrence of *X* by *E*, where *X* is the result which the tactic `simpl` would give when applied to *E*. It is useful to undo what `simpl` has simplified too far.

```

Tactic Notation "unsimpl" constr(E) :=
  let F := (eval simpl in E) in change F with E.

```

unsimpl E in H is similar to *unsimpl E* but it applies inside a particular hypothesis *H*.

```

Tactic Notation "unsimpl" constr(E) "in" hyp(H) :=
  let F := (eval simpl in E) in change F with E in H.

```

*unsimpl E in ** applies *unsimpl E* everywhere possible. *unsimpls E* is a synonymous.

```

Tactic Notation "unsimpl" constr(E) "in" "*" :=
  let F := (eval simpl in E) in change F with E in *.

```

```

Tactic Notation "unsimpls" constr(E) :=
  unsimpl E in *.

```

nosimpl t protects the Rocq term `t` against some forms of simplification. See Gonthier's work for details on this trick.

```

Notation "'nosimpl' t" := (match tt with tt => t end)
(at level 10).

```

2.7.4 Reduction

Tactic Notation "hnfs" := hnf in *.

2.7.5 Substitution

subst does the same as *subst*, except that it does not fail when there are circular equalities in the context.

```
Tactic Notation "subst" :=
  repeat (match goal with H: ?x = ?y ⊢ _ ⇒
    first [ subst x | subst y ] end).
```

Implementation of *subst below*, which allows to call *subst* on all the hypotheses that lie beyond a given position in the proof context.

```
Ltac subst_below limit :=
  match goal with H: ?T ⊢ _ ⇒
  match T with
  | limit ⇒ idtac
  | ?x = ?y ⇒
    first [ subst x; subst_below limit
          | subst y; subst_below limit
          | generalizes H; subst_below limit; intro ]
  end end.
```

subst below body E applies *subst* on all equalities that appear in the context below the first hypothesis whose body is *E*. If there is no such hypothesis in the context, it is equivalent to *subst*. For instance, if *H* is an hypothesis, then *subst below H* will substitute equalities below hypothesis *H*.

```
Tactic Notation "subst" "below" "body" constr(M) :=
  subst_below M.
```

subst below H applies *subst* on all equalities that appear in the context below the hypothesis named *H*. Note that the current implementation is technically incorrect since it will confuse different hypotheses with the same body.

```
Tactic Notation "subst" "below" hyp(H) :=
  match type of H with ?M ⇒ subst_below body M end.
```

subst_hyp H substitutes the equality contained in the first hypothesis from the context.

```
Ltac intro_subst_hyp := fail.
```

subst_hyp H substitutes the equality contained in *H*.

```
Ltac subst_hyp_base H :=
  match type of H with
  | ( _ , _ , _ , _ , _ ) = ( _ , _ , _ , _ , _ ) ⇒ injection H; clear H; do 4 intro_subst_hyp
```

```

| ( _,_,_,_ ) = ( _,_,_,_ ) => injection H; clear H; do 4 intro_subst_hyp
| ( _,_,_ ) = ( _,_,_ ) => injection H; clear H; do 3 intro_subst_hyp
| ( _,_ ) = ( _,_ ) => injection H; clear H; do 2 intro_subst_hyp
| ?x = ?x => clear H
| ?x = ?y => first [ subst x | subst y ]
end.

```

Tactic Notation "subst_hyp" hyp(*H*) := *subst_hyp_base H*.

Ltac *intro_subst_hyp* :=

```
let H := fresh "TEMP" in intros H; subst_hyp H.
```

intro_subst is a shorthand for *intro H*; *subst_hyp H*: it introduces and substitutes the equality at the head of the current goal.

Tactic Notation "intro_subst" :=

```
let H := fresh "TEMP" in intros H; subst_hyp H.
```

subst_local substitutes all local definition from the context

Ltac *subst_local* :=

```
repeat match goal with H := _ ⊢ _ => subst H end.
```

subst_eq E takes an equality *x = t* and replace *x* with *t* everywhere in the goal

Ltac *subst_eq_base E* :=

```
let H := fresh "TEMP" in lets H: E; subst_hyp H.
```

Tactic Notation "subst_eq" constr(*E*) :=

```
subst_eq_base E.
```

2.7.6 Tactics to Work with Proof Irrelevance

From *Stdlib.Logic* Require Import *ProofIrrelevance*.

pi_rewrite E replaces *E* of type **Prop** with a fresh unification variable, and is thus a practical way to exploit proof irrelevance, without writing explicitly *rewrite (proof_irrelevance E E')*. Particularly useful when *E'* is a big expression.

Ltac *pi_rewrite_base E rewrite_tac* :=

```
let E' := fresh "TEMP" in let T := type of E in evar (E':T);
rewrite_tac (@proof_irrelevance _ E E'); subst E'.
```

Tactic Notation "pi_rewrite" constr(*E*) :=

```
pi_rewrite_base E ltac:(fun X => rewrite X).
```

Tactic Notation "pi_rewrite" constr(*E*) "in" hyp(*H*) :=

```
pi_rewrite_base E ltac:(fun X => rewrite X in H).
```

2.7.7 Proving Equalities

The tactic *fequal* enhances Rocq's tactic *f_equal*, which does not simplify equalities between tuples, nor between dependent pairs of the form *exist _ _* or *existT _ _*. For support of

dependent pairs, the file `LibEqual` must be imported.

Subgoals solvable by `reflexivity` are automatically discharged. See also the variant `fequals`, which discharges more subgoals.

Note: only `args_eq_2` is actually useful for the implementation of `fequal`, if we rely on Rocq's `f_equal` tactic for other arities. We provide these lemmas to show the pattern of lemmas to exploit for implementing `fequal` independently of `f_equal`.

Section `FuncEq`.

Variables $(A1\ A2\ A3\ A4\ A5\ A6\ A7\ B : \text{Type})$.

Lemma `args_eq_1` : $\forall (f:A1 \rightarrow B)\ x1\ y1,$

$x1 = y1 \rightarrow$
 $f\ x1 = f\ y1.$

Proof using. `intros. subst. auto. Qed.`

Lemma `args_eq_2` : $\forall (f:A1 \rightarrow A2 \rightarrow B)\ x1\ y1\ x2\ y2,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow$
 $f\ x1\ x2 = f\ y1\ y2.$

Proof using. `intros. subst. auto. Qed.`

Lemma `args_eq_3` : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow B)\ x1\ y1\ x2\ y2\ x3\ y3,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow$
 $f\ x1\ x2\ x3 = f\ y1\ y2\ y3.$

Proof using. `intros. subst. auto. Qed.`

Lemma `args_eq_4` : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow B)\ x1\ y1\ x2\ y2\ x3\ y3\ x4\ y4,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow$
 $f\ x1\ x2\ x3\ x4 = f\ y1\ y2\ y3\ y4.$

Proof using. `intros. subst. auto. Qed.`

Lemma `args_eq_5` : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow B)\ x1\ y1\ x2\ y2\ x3\ y3\ x4\ y4\ x5\ y5,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow x5 = y5 \rightarrow$
 $f\ x1\ x2\ x3\ x4\ x5 = f\ y1\ y2\ y3\ y4\ y5.$

Proof using. `intros. subst. auto. Qed.`

Lemma `args_eq_6` : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow A6 \rightarrow B)\ x1\ y1\ x2\ y2\ x3\ y3\ x4\ y4\ x5\ y5\ x6\ y6,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow x5 = y5 \rightarrow x6 = y6 \rightarrow$
 $f\ x1\ x2\ x3\ x4\ x5\ x6 = f\ y1\ y2\ y3\ y4\ y5\ y6.$

Proof using. `intros. subst. auto. Qed.`

Lemma `args_eq_7` : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow A6 \rightarrow A7 \rightarrow B)\ x1\ y1\ x2\ y2\ x3\ y3\ x4\ y4\ x5\ y5\ x6\ y6\ x7\ y7,$

$x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow x5 = y5 \rightarrow x6 = y6 \rightarrow x7 = y7 \rightarrow$
 $f\ x1\ x2\ x3\ x4\ x5\ x6\ x7 = f\ y1\ y2\ y3\ y4\ y5\ y6\ y7.$

Proof using. `intros. subst. auto. Qed.`

End `FuncEq`.

Ltac `fequal_post` :=

```
try solve [ reflexivity ].
```

fequal_support_for_exist, implemented in `LibEqual`, is meant to simplify goals of the form `exist _ _ = exist _ _` and `existT _ _ = existT _ _`, by exploiting proof irrelevance.

```
Ltac fequal_support_for_exist tt :=
  fail.
```

For a n-ary tuple, *fequal*, unlike `f_equal` enforces a recursive call on the (n-1)-ary tuple associated with the right component.

```
Ltac fequal_base :=
  match goal with
  | ⊢ ( _ , _ , _ ) = ( _ , _ , _ ) ⇒ apply args_eq-2; [ fequal_base | ]
  | ⊢ _ ⇒ first
      [ fequal_support_for_exist tt
      | apply args_eq-1
      | apply args_eq-2
      | apply args_eq-3
      | apply args_eq-4
      | apply args_eq-5
      | apply args_eq-6
      | apply args_eq-7
      | f_equal ]
  end.
```

```
Tactic Notation "fequal" :=
  fequal_base; fequal_post.
```

fequals is the same as *fequal* except that it tries and solve all trivial subgoals, using `reflexivity` and `congruence` (as well as the proof-irrelevance principle). *fequals* applies to goals of the form `f x1 .. xN = f y1 .. yN` and produces some subgoals of the form `xi = yi`.

```
Ltac fequals_post :=
  try solve [ reflexivity | congruence | apply proof_irrelevance ].
```

```
Tactic Notation "fequals" :=
  fequal; fequals_post.
```

fequals_rec calls *fequals* recursively. It is equivalent to `repeat (progress fequals)`.

```
Tactic Notation "fequals_rec" :=
  repeat (progress fequals).
```

2.8 Inversion

2.8.1 Basic Inversion

invert keep H is same to *inversion H* except that it puts all the facts obtained in the goal. The keyword *keep* means that the hypothesis *H* should not be removed.

Tactic Notation "invert" "keep" *hyp(H)* :=
 pose ltac_mark; *inversion H*; *gen_until_mark*.

invert keep H as X1 .. XN is the same as *inversion H as ...* except that only hypotheses which are not variable need to be named explicitly, in a similar fashion as *introv* is used to name only hypotheses.

Tactic Notation "invert" "keep" *hyp(H)* "as" *simple_intropattern(I1)* :=
invert keep H; *introv I1*.

Tactic Notation "invert" "keep" *hyp(H)* "as" *simple_intropattern(I1)*
simple_intropattern(I2) :=
invert keep H; *introv I1 I2*.

Tactic Notation "invert" "keep" *hyp(H)* "as" *simple_intropattern(I1)*
simple_intropattern(I2) simple_intropattern(I3) :=
invert keep H; *introv I1 I2 I3*.

invert H is same to *inversion H* except that it puts all the facts obtained in the goal and clears hypothesis *H*. In other words, it is equivalent to *invert keep H*; *clear H*.

Tactic Notation "invert" *hyp(H)* :=
invert keep H; *clear H*.

invert H as X1 .. XN is the same as *invert keep H as X1 .. XN* but it also clears hypothesis *H*.

Tactic Notation "invert_tactic" *hyp(H)* *tactic(tac)* :=
 let *H'* := fresh "TEMP" in *rename H into H'*; *tac H'*; *clear H'*.

Tactic Notation "invert" *hyp(H)* "as" *simple_intropattern(I1)* :=
invert_tactic H (fun *H* => *invert keep H as I1*).

Tactic Notation "invert" *hyp(H)* "as" *simple_intropattern(I1)*
simple_intropattern(I2) :=
invert_tactic H (fun *H* => *invert keep H as I1 I2*).

Tactic Notation "invert" *hyp(H)* "as" *simple_intropattern(I1)*
simple_intropattern(I2) simple_intropattern(I3) :=
invert_tactic H (fun *H* => *invert keep H as I1 I2 I3*).

2.8.2 Inversion with Substitution

Our inversion tactics is able to get rid of dependent equalities generated by *inversion*, using proof irrelevance.

Axiom *inj_pair2* :
 $\forall (U : \text{Type}) (P : U \rightarrow \text{Type}) (p : U) (x y : P p),$
 $\text{existT } P p x = \text{existT } P p y \rightarrow x = y.$

Ltac *inverts_tactic H i1 i2 i3 i4 i5 i6* :=

```

let rec go i1 i2 i3 i4 i5 i6 :=
  match goal with
  | ⊢ (ltac_Mark → _) ⇒ intros _
  | ⊢ (?x = ?y → _) ⇒ let H := fresh "TEMP" in intro H;
                        first [ subst x | subst y ];
                        go i1 i2 i3 i4 i5 i6
  | ⊢ (existT ?P ?p ?x = existT ?P ?p ?y → _) ⇒
      let H := fresh "TEMP" in intro H;
      generalize (@inj_pair2 _ P p x y H);
      clear H; go i1 i2 i3 i4 i5 i6
  | ⊢ (?P → ?Q) ⇒ i1; go i2 i3 i4 i5 i6 ltac:(intro)
  | ⊢ (∀ _, _) ⇒ intro; go i1 i2 i3 i4 i5 i6
  end in
generalize ltac_mark; invert keep H; go i1 i2 i3 i4 i5 i6;
unfold eq' in *.

```

inverts keep H is same to *invert keep H* except that it applies **subst** to all the equalities generated by the inversion.

```

Tactic Notation "inverts" "keep" hyp(H) :=
  inverts_tactic H ltac:(intro) ltac:(intro) ltac:(intro)
  ltac:(intro) ltac:(intro) ltac:(intro).

```

inverts keep H as X1 .. XN is the same as *invert keep H as X1 .. XN* except that it applies **subst** to all the equalities generated by the inversion

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1) :=
  inverts_tactic H ltac:(intros I1)
  ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2)
  ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)
  ltac:(intro) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)
  ltac:(intros I4) ltac:(intro) ltac:(intro).

```

```

Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)

```



```

    ltac:(intros I4) ltac:(intros I5) ltac:(intro).
Tactic Notation "inverts" "keep" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5) simple_intropattern(I6) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros I3)
    ltac:(intros I4) ltac:(intros I5) ltac:(intros I6).

  inverts H is same to inverts keep H except that it clears hypothesis H.

```

```

Tactic Notation "inverts" hyp(H) :=
  inverts keep H; try clear H.

```

inverts H as X1 .. XN is the same as *inverts keep H as X1 .. XN* but it also clears the hypothesis *H*.

```

Tactic Notation "inverts_tactic" hyp(H) tactic(tac) :=
  let H' := fresh "TEMP" in rename H into H'; tac H'; clear H'.
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1) :=
  invert_tactic H (fun H => inverts keep H as I1).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) :=
  invert_tactic H (fun H => inverts keep H as I1 I2).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3 I4).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3 I4 I5).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5) simple_intropattern(I6) :=
  invert_tactic H (fun H => inverts keep H as I1 I2 I3 I4 I5 I6).

```

inverts H as performs an inversion on hypothesis *H*, substitutes generated equalities, and put in the goal the other freshly-created hypotheses, for the user to name explicitly. *inverts keep H as* is the same except that it does not clear *H*. Note: maybe reimplement *inverts* above using this one

```

Ltac inverts_as_tactic H :=
  let rec go tt :=
    match goal with
    | ⊢ (ltac_Mark → _) => intros _
    | ⊢ (?x = ?y → _) => let H := fresh "TEMP" in intro H;

```

```

      first [ subst x | subst y ];
      go tt
| ⊢ (existT ?P ?p ?x = existT ?P ?p ?y → _) ⇒
  let H := fresh "TEMP" in intro H;
  generalize (@inj_pair2 - P p x y H);
  clear H; go tt
| ⊢ (∀ -, -) ⇒
  intro; let H := get_last_hyp tt in mark_to_generalize H; go tt
end in
pose ltac_mark; inversion H;
generalize ltac_mark; gen_until_mark;
go tt; gen_to_generalize; unfolds ltac_to_generalize;
unfold eq' in *.

Tactic Notation "inverts" "keep" hyp(H) "as" :=
  inverts_as_tactic H.

Tactic Notation "inverts" hyp(H) "as" :=
  inverts_as_tactic H; clear H.

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5) simple_intropattern(I6) simple_intropattern(I7) :=
  inverts H as; introv I1 I2 I3 I4 I5 I6 I7.

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5) simple_intropattern(I6) simple_intropattern(I7)
  simple_intropattern(I8) :=
  inverts H as; introv I1 I2 I3 I4 I5 I6 I7 I8.

  lets_inverts E as I1 .. IN is intuitively equivalent to inverts E, with the difference that
  it applies to any expression and not just to the name of an hypothesis.

Ltac lets_inverts_base E cont :=
  let H := fresh "TEMP" in lets H: E; try cont H.

Tactic Notation "lets_inverts" constr(E) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1)
  simple_intropattern(I2) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1)
  simple_intropattern(I2) simple_intropattern(I3) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2 I3).

Tactic Notation "lets_inverts" constr(E) "as" simple_intropattern(I1)

```

```
simple_intropattern(I2) simple_intropattern(I3) simple_intropattern(I4) :=
  lets_inverts_base E ltac:(fun H => inverts H as I1 I2 I3 I4).
```

2.8.3 Inversion with Pattern Matching on Head of Hypothesis

invert_post is invoked after *false_inverts* and *invert_if_head*. LATER: could be also called after *invert*.

```
Ltac invert_post tt :=
  try discriminate.
```

inverts_post is invoked after *false_inverts* and *inverts_if_head*. LATER: could be also called after *inverts*.

```
Ltac inverts_post tt :=
  try discriminate.
```

false_inverts is like *false_invert* but with substitutions.

```
Ltac false_inverts_iter :=
  match goal with H: _ ⊢ _ =>
    first [ solve [ inversion H; try subst; false; inverts_post tt ]
          | clear H; false_inverts_iter ] end.
```

```
Tactic Notation "false_inverts" :=
  intros; solve [ false_inverts_iter | false ].
```

invert_if_head E calls *invert* on an assumption whose head symbol is *E*. It then calls *invert_post tt* to attempt discarding absurd goals

```
Ltac invert_if_head E :=
  match goal with H: ?T ⊢ _ =>
    match get_head T with E =>
      invert H; invert_post tt
    end
  end.
```

inverts_if_head E calls *inverts* an assumption whose head symbol is *E*. It then calls *inverts_post tt* to attempt discarding absurd goals

```
Ltac inverts_if_head E :=
  match goal with H: ?T ⊢ _ =>
    match get_head T with E =>
      inverts H; inverts_post tt
    end
  end.
```

2.8.4 Injection with Substitution

Underlying implementation of *injects*

```

Ltac injects_tactic H :=
  let rec go _ :=
    match goal with
    | ⊢ (ltac_Mark → _) ⇒ intros _
    | ⊢ (?x = ?y → _) ⇒ let H := fresh "TEMP" in intro H;
                          first [ subst x | subst y | idtac ];
                          go tt
    end in
  generalize ltac_mark; injection H; go tt.

```

injects keep H takes an hypothesis *H* of the form $C\ a1 \dots aN = C\ b1 \dots bN$ and substitute all equalities $ai = bi$ that have been generated.

Tactic Notation "injects" "keep" *hyp*(*H*) :=
injects_tactic *H*.

injects H is similar to *injects keep H* but clears the hypothesis *H*.

Tactic Notation "injects" *hyp*(*H*) :=
injects_tactic *H*; clear *H*.

inject H as X1 .. XN is the same as *injection* followed by *intros X1 .. XN*

Tactic Notation "inject" *hyp*(*H*) :=
injection *H*.

Tactic Notation "inject" *hyp*(*H*) "as" *ident*(*X1*) :=
injection *H*; intros *X1*.

Tactic Notation "inject" *hyp*(*H*) "as" *ident*(*X1*) *ident*(*X2*) :=
injection *H*; intros *X1* *X2*.

Tactic Notation "inject" *hyp*(*H*) "as" *ident*(*X1*) *ident*(*X2*) *ident*(*X3*) :=
injection *H*; intros *X1* *X2* *X3*.

Tactic Notation "inject" *hyp*(*H*) "as" *ident*(*X1*) *ident*(*X2*) *ident*(*X3*)
ident(*X4*) :=
injection *H*; intros *X1* *X2* *X3* *X4*.

Tactic Notation "inject" *hyp*(*H*) "as" *ident*(*X1*) *ident*(*X2*) *ident*(*X3*)
ident(*X4*) *ident*(*X5*) :=
injection *H*; intros *X1* *X2* *X3* *X4* *X5*.

2.8.5 Inversion and Injection with Substitution –rough implementation

The tactics *inversions* and *injections* provided in this section are similar to *inverts* and *injects* except that they perform substitution on all equalities from the context and not only the ones freshly generated. The counterpart is that they have simpler implementations.

DEPRECATED: these tactics should no longer be used.

inversions keep H is the same as *inversions H* but it does not clear hypothesis *H*.

Tactic Notation "inversions" "keep" *hyp*(*H*) :=
 inversion *H*; subst.

inversions H is a shortcut for *inversion H* followed by *subst* and *clear H*. It is a rough implementation of *inverts keep H* which behave badly when the proof context already contains equalities. It is provided in case the better implementation turns out to be too slow.

Tactic Notation "inversions" *hyp*(*H*) :=
 inversion *H*; subst; try clear *H*.

injections keep H is the same as *injection H* followed by *intros* and *subst*. It is a rough implementation of *injects keep H* which behave badly when the proof context already contains equalities, or when the goal starts with a forall or an implication.

Tactic Notation "injections" "keep" *hyp*(*H*) :=
 injection *H*; intros; subst.

injections H is the same as *injection H* followed by *clear H* and *intros* and *subst*. It is a rough implementation of *injects keep H* which behave badly when the proof context already contains equalities, or when the goal starts with a forall or an implication.

Tactic Notation "injections" "keep" *hyp*(*H*) :=
 injection *H*; clear *H*; intros; subst.

2.8.6 Case Analysis

cases is similar to *case_eq E* except that it generates the equality in the context and not in the goal, and generates the equality the other way round. The syntax *cases E as H* allows specifying the name *H* of that hypothesis.

Tactic Notation "cases" *constr*(*E*) "as" *ident*(*H*) :=
 let *X* := fresh "TEMP" in
 set (*X* := *E*) in *; *def_to_eq_sym X H E*;
 destruct *X*.

Tactic Notation "cases" *constr*(*E*) :=
 let *H* := fresh "Eq" in *cases E as H*.

case_if_post H is to be defined later as a tactic to clean up hypothesis *H* and the goal. By defaults, it looks for obvious contradictions. Currently, this tactic is extended in LibReflect to clean up boolean propositions.

Ltac *case_if_post H* :=
tryfalse.

case_if looks for a pattern of the form *if ?B then ?E1 else ?E2* in the goal, and perform a case analysis on *B* by calling *destruct B*. Subgoals containing a contradiction are discarded. *case_if* looks in the goal first, and otherwise in the first hypothesis that contains an *if* statement. *case_if in H* can be used to specify which hypothesis to consider.

Syntaxes *case-if* as *Eq* and *case-if* in *H* as *Eq* allows to name the hypothesis coming from the case analysis.

```

Ltac case_if_on_tactic_core E Eq :=
  match type of E with
  | {_-}{_-} => destruct E as [Eq | Eq]
  | _ => let X := fresh "TEMP" in
        sets_eq ← X Eq: E;
        destruct X
  end.

Ltac case_if_on_tactic E Eq :=
  case_if_on_tactic_core E Eq; case_if_post Eq.

Tactic Notation "case-if-on" constr(E) "as" simple_intropattern(Eq) :=
  case_if_on_tactic E Eq.

Tactic Notation "case-if" "as" simple_intropattern(Eq) :=
  match goal with
  | ⊢ context [if ?B then _ else _] => case_if_on B as Eq
  | K: context [if ?B then _ else _] ⊢ _ => case_if_on B as Eq
  end.

Tactic Notation "case-if" "in" hyp(H) "as" simple_intropattern(Eq) :=
  match type of H with context [if ?B then _ else _] =>
    case_if_on B as Eq end.

Tactic Notation "case-if" :=
  let Eq := fresh "C" in case-if as Eq.

Tactic Notation "case-if" "in" hyp(H) :=
  let Eq := fresh "C" in case-if in H as Eq.

cases-if is similar to case-if with two main differences: if it creates an equality of the
form x = y and then substitutes it in the goal

Ltac cases_if_on_tactic_core E Eq :=
  match type of E with
  | {_-}{_-} => destruct E as [Eq|Eq]; try subst_hyp Eq
  | _ => let X := fresh "TEMP" in
        sets_eq ← X Eq: E;
        destruct X
  end.

Ltac cases_if_on_tactic E Eq :=
  cases_if_on_tactic_core E Eq; tryfalse; case_if_post Eq.

Tactic Notation "cases-if-on" constr(E) "as" simple_intropattern(Eq) :=
  cases_if_on_tactic E Eq.

Tactic Notation "cases-if" "as" simple_intropattern(Eq) :=

```

```

match goal with
| ⊢ context [if ?B then _ else _] ⇒ cases_if_on B as Eq
| K: context [if ?B then _ else _] ⊢ _ ⇒ cases_if_on B as Eq
end.

Tactic Notation "cases_if" "in" hyp(H) "as" simple_intropattern(Eq) :=
  match type of H with context [if ?B then _ else _] ⇒
    cases_if_on B as Eq end.

Tactic Notation "cases_if" :=
  let Eq := fresh "C" in cases_if as Eq.

Tactic Notation "cases_if" "in" hyp(H) :=
  let Eq := fresh "C" in cases_if in H as Eq.

  case_ifs is like repeat case_if

Ltac case_ifs_core :=
  repeat case_if.

Tactic Notation "case_ifs" :=
  case_ifs_core.

  destruct_if looks for a pattern of the form if ?B then ?E1 else ?E2 in the goal, and
  perform a case analysis on B by calling destruct B. It looks in the goal first, and otherwise
  in the first hypothesis that contains an if statement.

Ltac destruct_if_post := tryfalse.

Tactic Notation "destruct_if"
  "as" simple_intropattern(Eq1) simple_intropattern(Eq2) :=
  match goal with
  | ⊢ context [if ?B then _ else _] ⇒ destruct B as [Eq1|Eq2]
  | K: context [if ?B then _ else _] ⊢ _ ⇒ destruct B as [Eq1|Eq2]
  end;
  destruct_if_post.

Tactic Notation "destruct_if" "in" hyp(H)
  "as" simple_intropattern(Eq1) simple_intropattern(Eq2) :=
  match type of H with context [if ?B then _ else _] ⇒
    destruct B as [Eq1|Eq2] end;
  destruct_if_post.

Tactic Notation "destruct_if" "as" simple_intropattern(Eq) :=
  destruct_if as Eq Eq.

Tactic Notation "destruct_if" "in" hyp(H) "as" simple_intropattern(Eq) :=
  destruct_if in H as Eq Eq.

Tactic Notation "destruct_if" :=
  let Eq := fresh "C" in destruct_if as Eq Eq.

Tactic Notation "destruct_if" "in" hyp(H) :=

```

let $Eq := \text{fresh } "C" \text{ in } \text{destruct\_if} \text{ in } H \text{ as } Eq \text{ } Eq.$

cases' is provided for compatibility with *remember*.

$\text{cases}' E$ is similar to $\text{case\_eq } E$ except that it generates the equality in the context and not in the goal. The syntax $\text{cases}' E \text{ as } H$ allows specifying the name H of that hypothesis.

Tactic Notation "cases'" $\text{constr}(E)$ "as" $\text{ident}(H)$:=
 let $X := \text{fresh } "TEMP" \text{ in}$
 set $(X := E)$ in *; $\text{def\_to\_eq } X \ H \ E;$
 destruct X .

Tactic Notation "cases'" $\text{constr}(E)$:=
 let $x := \text{fresh } "Eq" \text{ in } \text{cases}' E \text{ as } H.$

$\text{cases\_if}'$ is similar to cases_if except that it generates the symmetric equality.

Ltac $\text{cases\_if\_on}' E \ Eq :=$
 match type of E with
 | $\{ \_ \} + \{ \_ \} \Rightarrow \text{destruct } E \text{ as } [Eq|Eq]; \text{try } \text{subst\_hyp } Eq$
 | $\_ \Rightarrow \text{let } X := \text{fresh } "TEMP" \text{ in}$
 $\text{sets\_eq } X \ Eq: E;$
 destruct X
 end; $\text{case\_if\_post } Eq.$

Tactic Notation "cases_if'" "as" $\text{simple\_intropattern}(Eq)$:=
 match goal with
 | $\vdash \text{context } [\text{if } ?B \text{ then } \_ \text{ else } \_] \Rightarrow \text{cases\_if\_on}' B \ Eq$
 | $K: \text{context } [\text{if } ?B \text{ then } \_ \text{ else } \_] \vdash \_ \Rightarrow \text{cases\_if\_on}' B \ Eq$
 end.

Tactic Notation "cases_if'" :=
 let $Eq := \text{fresh } "C" \text{ in } \text{cases\_if}' \text{ as } Eq.$

2.9 Induction

$\text{inductions } E$ is a shorthand for dependent induction E . $\text{inductions } E \text{ gen } X1 \ .. \ XN$ is a shorthand for dependent induction E generalizing $X1 \ .. \ XN$.

From *Stdlib.Program* Require Import Equality.

Ltac $\text{inductions\_post} :=$
 unfold eq' in *.

Tactic Notation "inductions" $\text{ident}(E)$:=
 dependent induction E ; $\text{inductions\_post}.$

Tactic Notation "inductions" $\text{ident}(E)$ "gen" $\text{ident}(X1)$:=
 dependent induction E generalizing $X1$; $\text{inductions\_post}.$

Tactic Notation "inductions" $\text{ident}(E)$ "gen" $\text{ident}(X1) \ \text{ident}(X2)$:=
 dependent induction E generalizing $X1 \ X2$; $\text{inductions\_post}.$


```

Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) :=
  dependent induction E generalizing X1 X2 X3; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) :=
  dependent induction E generalizing X1 X2 X3 X4; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) :=
  dependent induction E generalizing X1 X2 X3 X4 X5; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) ident(X6) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) ident(X6) ident(X7) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6 X7; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
  ident(X3) ident(X4) ident(X5) ident(X6) ident(X7) ident(X8) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6 X7 X8; inductions_post.

```

induction_wf IH: E X is used to apply the well-founded induction principle, for a given well-founded relation. It applies to a goal *PX* where *PX* is a proposition on *X*. First, it sets up the goal in the form `(fun a => P a) X`, using `pattern X`, and then it applies the well-founded induction principle instantiated on *E*.

Here *E* may be either:

- a proof of `wf R` for *R* of type $A \rightarrow A \rightarrow \text{Prop}$
- a binary relation of type $A \rightarrow A \rightarrow \text{Prop}$
- a measure of type $A \rightarrow \text{nat}$ // only when `LibWf` is used.

```

Ltac induction_wf_process_wf_hyp tt :=
  idtac.
Ltac induction_wf_process_measure E :=
  fail.
Ltac induction_wf_core_then E X cont :=
  let clearX tt :=
    first [ clear X | fail 3 "the variable on which the induction is done appears in the
hypotheses" ] in
  pattern X;
  first [ eapply (@well_founded_ind _ E)
        | eapply (@well_founded_ind _ (E _))
        | eapply (@well_founded_ind _ (E _ _))

```

```

      | eapply (@well_founded_ind _ (E _ _))
      | induction_wf_process_measure E
      | applies well_founded_ind E ];
clearX tt;
first [ induction_wf_process_wf_hyp tt
      | intros X; cont tt ].

Ltac induction_wf_core IH E X :=
  induction_wf_core_then E X ltac:(fun _ => intros IH).

Tactic Notation "induction_wf" ident(IH) ":" constr(E) ident(X) :=
  induction_wf_core IH E X.

```

Induction on the height of a derivation: the helper tactic *induct_height* helps proving the equivalence of the auxiliary judgment that includes a counter for the maximal height (see LibTacticsDemos for an example)

From *Stdlib.Arith* Require Import Compare_dec.

From *Stdlib.micromega* Require Import Lia.

Lemma induct_height_max2 : $\forall n1\ n2 : \mathbf{nat}$,
 $\exists n, n1 < n \wedge n2 < n$.

Proof using.

```

  intros. destruct (lt_dec n1 n2).
   $\exists$  (S n2). lia.
   $\exists$  (S n1). lia.

```

Qed.

```

Ltac induct_height_step x :=
  match goal with
  | H:  $\exists \_, \_ \vdash \_ \Rightarrow$ 
    let n := fresh "n" in let y := fresh "x" in
    destruct H as [n?];
    forwards (y&?&?): induct_height_max2 n x;
    induct_height_step y
  | _ =>  $\exists$  (S x); eauto
  end.

```

Ltac induct_height := induct_height_step O.

2.10 Coinduction

Tactic *cofixs IH* is like *cofix IH* except that the coinduction hypothesis is tagged in the form *IH*: **COIND** *P* instead of being just *IH*: *P*. This helps other tactics clearing the coinduction hypothesis using *clear_coind*

Definition **COIND** (*P*:Prop) := *P*.

Tactic Notation "cofixs" *ident*(*IH*) :=
 cofix *IH*;
 match type of *IH* with ?*P* ⇒ change *P* with (COIND *P*) in *IH* end.

Tactic *clear_coind* clears all the coinduction hypotheses, assuming that they have been tagged

Ltac *clear_coind* :=
 repeat match goal with *H*: COIND _ ⊢ _ ⇒ clear *H* end.

Tactic *abstracts tac* is like *abstract tac* except that it clears the coinduction hypotheses so that the productivity check will be happy. For example, one can use *abstracts lia* to obtain the same behavior as *lia* but with an auxiliary lemma being generated.

Tactic Notation "abstracts" *tactic*(*tac*) :=
clear_coind; *tac*.

2.11 Decidable Equality

decides_equality is the same as *decide equality* excepts that it is able to unfold definitions at head of the current goal.

Ltac *decides_equality_tactic* :=
 first [*decide equality* | progress(*unfolds*); *decides_equality_tactic*].

Tactic Notation "decides_equality" :=
decides_equality_tactic.

2.12 Equivalence

iff *H* can be used to prove an equivalence $P \leftrightarrow Q$ and name *H* the hypothesis obtained in each case. The syntaxes *iff* and *iff H1 H2* are also available to specify zero or two names. The tactic *iff* ← *H* swaps the two subgoals, i.e. produces (Q -> P) as first subgoal.

Lemma *iff_intro_swap* : ∀ (*P Q* : Prop),
 (*Q* → *P*) → (*P* → *Q*) → (*P* ↔ *Q*).

Proof using. intuition. Qed.

Tactic Notation "iff" *simple_intropattern*(*H1*) *simple_intropattern*(*H2*) :=
 split; [intros *H1* | intros *H2*].

Tactic Notation "iff" *simple_intropattern*(*H*) :=
iff *H* *H*.

Tactic Notation "iff" :=
 let *H* := fresh "H" in *iff* *H*.

Tactic Notation "iff" "<-" *simple_intropattern*(*H1*) *simple_intropattern*(*H2*) :=
 apply *iff_intro_swap*; [intros *H1* | intros *H2*].

Tactic Notation "iff" "<-" *simple_intropattern*(*H*) :=

```

    iff ← H H.
Tactic Notation "iff" "<-" :=
  let H := fresh "H" in iff ← H.

```

2.13 N-ary Conjunctions and Disjunctions

Underlying implementation of *splits*.

```

Ltac splits_tactic N :=
  match N with
  | 0 ⇒ fail
  | S 0 ⇒ idtac
  | S ?N' ⇒ split; [| splits_tactic N' |]
  end.

Ltac unfold_goal_until_conjunction :=
  match goal with
  | ⊢ _ ∧ _ ⇒ idtac
  | _ ⇒ progress(unfolds); unfold_goal_until_conjunction
  end.

Ltac get_term_conjunction_arity T :=
  match T with
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(8)
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(7)
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(6)
  | _ ∧ _ ∧ _ ∧ _ ⇒ constr:(5)
  | _ ∧ _ ∧ _ ⇒ constr:(4)
  | _ ∧ _ ∧ _ ⇒ constr:(3)
  | _ ∧ _ ⇒ constr:(2)
  | _ → ?T' ⇒ get_term_conjunction_arity T'
  | _ ⇒ let P := get_head T in
    let T' := eval unfold P in T in
    match T' with
    | T ⇒ fail 1
    | _ ⇒ get_term_conjunction_arity T'
    end

  end.

Ltac get_goal_conjunction_arity :=
  match goal with ⊢ ?T ⇒ get_term_conjunction_arity T end.

```

splits applies to a goal of the form $(T1 \wedge \dots \wedge TN)$ and destruct it into N subgoals $T1 \dots TN$. If the goal is not a conjunction, then it unfolds the head definition.

```
Tactic Notation "splits" :=
  unfold_goal_until_conjunction;
  let  $N$  := get_goal_conjunction_arity in
  splits_tactic  $N$ .
```

splits N is similar to *splits*, except that it will unfold as many definitions as necessary to obtain an N -ary conjunction.

```
Tactic Notation "splits" constr( $N$ ) :=
  let  $N$  := number_to_nat  $N$  in
  splits_tactic  $N$ .
```

Underlying implementation of *destructs*.

```
Ltac destructs_conjunction_tactic  $N$   $T$  :=
  match  $N$  with
  | 2  $\Rightarrow$  destruct  $T$  as [?]
  | 3  $\Rightarrow$  destruct  $T$  as [?] [?]
  | 4  $\Rightarrow$  destruct  $T$  as [?] [?] [?]
  | 5  $\Rightarrow$  destruct  $T$  as [?] [?] [?] [?]
  | 6  $\Rightarrow$  destruct  $T$  as [?] [?] [?] [?] [?]
  | 7  $\Rightarrow$  destruct  $T$  as [?] [?] [?] [?] [?] [?]
  end.
```

destructs T allows destructing a term T which is a N -ary conjunction. It is equivalent to *destruct* T as ($H1$.. HN), except that it does not require to manually specify N different names.

```
Tactic Notation "destructs" constr( $T$ ) :=
  let  $TT$  := type of  $T$  in
  let  $N$  := get_term_conjunction_arity  $TT$  in
  destructs_conjunction_tactic  $N$   $T$ .
```

destructs N T is equivalent to *destruct* T as ($H1$.. HN), except that it does not require to manually specify N different names. Remark that it is not restricted to N -ary conjunctions.

```
Tactic Notation "destructs" constr( $N$ ) constr( $T$ ) :=
  let  $N$  := number_to_nat  $N$  in
  destructs_conjunction_tactic  $N$   $T$ .
```

Underlying implementation of *branch*.

```
Ltac branch_tactic  $K$   $N$  :=
  match constr:(( $K$ , $N$ )) with
  | (_,0)  $\Rightarrow$  fail 1
  | (0,_)  $\Rightarrow$  fail 1
  | (1,1)  $\Rightarrow$  idtac
  | (1,_)  $\Rightarrow$  left
```

```
| (S ?K', S ?N') => right; branch_tactic K' N'
end.
```

```
Ltac unfold_goal_until_disjunction :=
  match goal with
  | ⊢ _ ∨ _ => idtac
  | _ => progress(unfolds); unfold_goal_until_disjunction
  end.
```

```
Ltac get_term_disjunction_arity T :=
  match T with
  | _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ => constr:(8)
  | _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ => constr:(7)
  | _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ => constr:(6)
  | _ ∨ _ ∨ _ ∨ _ ∨ _ => constr:(5)
  | _ ∨ _ ∨ _ ∨ _ => constr:(4)
  | _ ∨ _ ∨ _ => constr:(3)
  | _ ∨ _ => constr:(2)
  | _ → ?T' => get_term_disjunction_arity T'
  | _ => let P := get_head T in
    let T' := eval unfold P in T in
    match T' with
    | T => fail 1
    | _ => get_term_disjunction_arity T'
    end
  end.
```

```
Ltac get_goal_disjunction_arity :=
  match goal with ⊢ ?T => get_term_disjunction_arity T end.
```

branch N applies to a goal of the form $P1 \vee \dots \vee PK \vee \dots \vee PN$ and leaves the goal PK . It is only able to unfold the head definition (if there is one), but for more complex unfolding one should use the tactic *branch K of N*.

```
Tactic Notation "branch" constr(K) :=
  let K := number_to_nat K in
  unfold_goal_until_disjunction;
  let N := get_goal_disjunction_arity in
  branch_tactic K N.
```

branch K of N is similar to *branch K* except that the arity of the disjunction N is given manually, and so this version of the tactic is able to unfold definitions. In other words, it applies to a goal of the form $P1 \vee \dots \vee PK \vee \dots \vee PN$ and leaves the goal PK .

```
Tactic Notation "branch" constr(K) "of" constr(N) :=
  let N := number_to_nat N in
  let K := number_to_nat K in
  branch_tactic K N.
```

Underlying implementation of *branches*.

```
Ltac destructs_disjunction_tactic N T :=
  match N with
  | 2 => destruct T as [? | ?]
  | 3 => destruct T as [? | [? | ?]]
  | 4 => destruct T as [? | [? | [? | ?]]]
  | 5 => destruct T as [? | [? | [? | [? | ?]]]]
  end.
```

branches T allows destructing a term T which is a N -ary disjunction. It is equivalent to `destruct  $T$  as [  $H1$  | .. |  $HN$  ]`, and produces N subgoals corresponding to the N possible cases.

```
Tactic Notation "branches" constr(T) :=
  let TT := type of T in
  let N := get_term_disjunction_arity TT in
  destructs_disjunction_tactic N T.
```

branches N T is the same as *branches* T except that the arity is forced to N . This version is useful to unfold definitions on the fly.

```
Tactic Notation "branches" constr(N) constr(T) :=
  let N := number_to_nat N in
  destructs_disjunction_tactic N T.
```

branches automatically finds a hypothesis h that is a disjunction and destructs it.

```
Tactic Notation "branches" :=
  match goal with h: _  $\vee$  _  $\vdash$  _ => branches h end.
```

$\exists T1 \dots TN$ is a shorthand for $\exists T1; \dots; \exists TN$. It is intended to prove goals of the form `exist  $X1$  ..  $XN$ ,  $P$` . If an argument provided is `--` (double underscore), then an `evvar` is introduced. $\exists T1 \dots TN \text{ ---}$ is equivalent to $\exists T1 \dots TN \text{ --- --- ---}$ with as many `--` as possible.

```
Tactic Notation "exists_original" constr(T1) :=
   $\exists T1$ .
```

```
Tactic Notation "exists" constr(T1) :=
  match T1 with
  | ltac_wild => esplit
  | ltac_wilds => repeat esplit
  | _ =>  $\exists T1$ 
  end.
```

```
Tactic Notation "exists" constr(T1) constr(T2) :=
   $\exists T1; \exists T2$ .
```

```
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) :=
   $\exists T1; \exists T2; \exists T3$ .
```

```
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4) :=
```

```

  ∃ T1; ∃ T2; ∃ T3; ∃ T4.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
  constr(T5) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
  constr(T5) constr(T6) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5; ∃ T6.

For compatibility with Rocq syntax, ∃ T1, .., TN is also provided.
Tactic Notation "exists" constr(T1) "," constr(T2) :=
  ∃ T1 T2.
Tactic Notation "exists" constr(T1) "," constr(T2) "," constr(T3) :=
  ∃ T1 T2 T3.
Tactic Notation "exists" constr(T1) "," constr(T2) "," constr(T3) "," constr(T4) :=
  ∃ T1 T2 T3 T4.
Tactic Notation "exists" constr(T1) "," constr(T2) "," constr(T3) "," constr(T4) ","
  constr(T5) :=
  ∃ T1 T2 T3 T4 T5.
Tactic Notation "exists" constr(T1) "," constr(T2) "," constr(T3) "," constr(T4) ","
  constr(T5) "," constr(T6) :=
  ∃ T1 T2 T3 T4 T5 T6.

```

The tactic $\exists$, without arguments, repeats *esplit* as many times as there are visible existentials at the head of the goal. In there are zero existentials visible, the tactic *hnf* is called once. If there are still zero existentials visible, the tactic fails.

To obtain a tactic that supports arbitrary number of existentials including the possibility of having zero existentials, use *try* $\exists$.

```

Ltac exists_noarg_positive tt :=
  match goal with
  | ⊢ ∃ _, _ ⇒ esplit; try exists_noarg_positive tt
  end.

Ltac exists_noarg tt :=
  match goal with
  | ⊢ ∃ _, _ ⇒ exists_noarg_positive tt
  | _ ⇒ hnf; exists_noarg_positive tt
  end.

Tactic Notation "exists" :=
  exists_noarg tt.

```

```

Definition def_with_exists (n:nat) : Prop :=
  ∃ x, x = n.

```

The tactic *exists_nounfold* is similar to $\exists$, except that it does nothing if there are no existentials visible in the goal. This tactic may be useful for programming other tactics in a robust way.


```
Tactic Notation "exists_nounfold" :=
  try exists_noarg_positive tt.
```

unpack or *unpack H* destructs conjunctions and existentials in all or one hypothesis.

```
Ltac unpack_core :=
  repeat match goal with
  | H: _ ∧ _ ⊢ _ ⇒ destruct H
  | H: ∃ (varname: _), _ ⊢ _ ⇒

      let name := fresh varname in
      destruct H as [name ?]
  end.
```

```
Ltac unpack_hypothesis H :=
  try match type of H with
  | _ ∧ _ ⇒
      let h1 := fresh "TEMP" in
      let h2 := fresh "TEMP" in
      destruct H as [ h1 h2 ];
      unpack_hypothesis h1;
      unpack_hypothesis h2
  | ∃ (varname: _), _ ⇒

      let name := fresh varname in
      let body := fresh "TEMP" in
      destruct H as [name body];
      unpack_hypothesis body
  end.
```

```
Tactic Notation "unpack" :=
  unpack_core.
Tactic Notation "unpack" constr(H) :=
  unpack_hypothesis H.
```

2.14 Tactics to Prove Typeclass Instances

typeclass is an automation tactic specialized for finding typeclass instances.

```
Tactic Notation "typeclass" :=
  let go _ := eauto with typeclass_instances in
  solve [ go tt | constructor; go tt ].
```

solve_typeclass is a simpler version of *typeclass*, to use in hint tactics for resolving instances

```
Tactic Notation "solve_typeclass" :=
```

```
solve [ eauto with typeclass_instances ].
```

2.15 Tactics to Invoke Automation

2.15.1 Definitions for Parsing Compatibility

```
Tactic Notation "f_equal" :=  
  f_equal.  
Tactic Notation "constructor" :=  
  constructor.  
Tactic Notation "simple" :=  
  simpl.  
Tactic Notation "split" :=  
  split.  
Tactic Notation "right" :=  
  right.  
Tactic Notation "left" :=  
  left.
```

2.15.2 *hint* to Add Hints Local to a Lemma

hint E adds *E* as an hypothesis so that automation can use it. Syntax *hint E1,...,EN* is available

```
Tactic Notation "hint" constr(E) :=  
  let H := fresh "Hint" in lets H: E.  
Tactic Notation "hint" constr(E1) ", " constr(E2) :=  
  hint E1; hint E2.  
Tactic Notation "hint" constr(E1) ", " constr(E2) ", " constr(E3) :=  
  hint E1; hint E2; hint(E3).  
Tactic Notation "hint" constr(E1) ", " constr(E2) ", " constr(E3) ", " constr(E4) :=  
  hint E1; hint E2; hint(E3); hint(E4).
```

2.15.3 *jauto*, a New Automation Tactic

jauto is better at `intuition eauto` because it can open existentials from the context. In the same time, *jauto* can be faster than `intuition eauto` because it does not destruct disjunctions from the context. The strategy of *jauto* can be summarized as follows:

- open all the existentials and conjunctions from the context
- call `esplit` and `split` on the existentials and conjunctions in the goal

- call eauto.

Tactic Notation "jauto" :=
try solve [*jauto_set*; eauto].

Tactic Notation "jauto_fast" :=
try solve [auto | eauto | *jauto*].

iauto is a shorthand for intuition eauto

Tactic Notation "iauto" := try solve [intuition eauto].

2.15.4 Definitions of Automation Tactics

The two following tactics defined the default behaviour of “light automation” and “strong automation”. These tactics may be redefined at any time using the syntax `Ltac .. ::= ...`

auto_tilde is the tactic which will be called each time a symbol $\neg$ is used after a tactic.

Ltac *auto_tilde_default* := auto.

Ltac *auto_tilde* := *auto_tilde_default*.

auto_star is the tactic which will be called each time a symbol $\times$ is used after a tactic.

Ltac *auto_star_default* := try solve [auto | eauto | intuition eauto].

Ltac *auto_star* := *auto_star_default*.

autos $\neg$ is a notation for tactic *auto_tilde*. It may be followed by lemmas (or proofs terms) which auto will be able to use for solving the goal.

autos is an alias for *autos* $\neg$

Tactic Notation "autos" :=
auto_tilde.

Tactic Notation "autos" "~" :=
auto_tilde.

Tactic Notation "autos" "~" constr(*E1*) :=
lets: *E1*; *auto_tilde*.

Tactic Notation "autos" "~" constr(*E1*) constr(*E2*) :=
lets: *E1*; *autos* $\neg$ *E2*.

Tactic Notation "autos" "~" constr(*E1*) constr(*E2*) constr(*E3*) :=
lets: *E1*; *autos* $\neg$ *E2* *E3*.

Tactic Notation "autos" "~" constr(*E1*) constr(*E2*) constr(*E3*) constr(*E4*) :=
lets: *E1*; *autos* $\neg$ *E2* *E3* *E4*.

Tactic Notation "autos" "~" constr(*E1*) constr(*E2*) constr(*E3*) constr(*E4*)
constr(*E5*):=
lets: *E1*; *autos* $\neg$ *E2* *E3* *E4* *E5*.

Tactic Notation "autos" :=
auto_tilde.

Tactic Notation "autos" "~" :=

```

    auto_tilde.
Tactic Notation "autos" "~" constr(E1) :=
    lets: E1; auto_tilde.
Tactic Notation "autos" "~" constr(E1) ", " constr(E2) :=
    lets: E1; autos  $\neg$  E2.
Tactic Notation "autos" "~" constr(E1) ", " constr(E2) ", " constr(E3) :=
    lets: E1; autos  $\neg$  E2 E3.
Tactic Notation "autos" "~" constr(E1) ", " constr(E2) ", " constr(E3) ", " constr(E4)
:=
    lets: E1; autos  $\neg$  E2 E3 E4.
Tactic Notation "autos" "~" constr(E1) ", " constr(E2) ", " constr(E3) ", " constr(E4)
", "
constr(E5):=
    lets: E1; autos  $\neg$  E2 E3 E4 E5.

    autos $\times$  is a notation for tactic auto_star. It may be followed by lemmas (or proofs terms)
    which auto will be able to use for solving the goal.

Tactic Notation "autos" "*" :=
    auto_star.
Tactic Notation "autos" "*" constr(E1) :=
    lets: E1; auto_star.
Tactic Notation "autos" "*" constr(E1) constr(E2) :=
    lets: E1; autos $\times$  E2.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) :=
    lets: E1; autos $\times$  E2 E3.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) constr(E4) :=
    lets: E1; autos $\times$  E2 E3 E4.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) constr(E4)
constr(E5):=
    lets: E1; autos $\times$  E2 E3 E4 E5.

Tactic Notation "autos" "*" :=
    auto_star.
Tactic Notation "autos" "*" constr(E1) :=
    lets: E1; auto_star.
Tactic Notation "autos" "*" constr(E1) ", " constr(E2) :=
    lets: E1; autos $\times$  E2.
Tactic Notation "autos" "*" constr(E1) ", " constr(E2) ", " constr(E3) :=
    lets: E1; autos $\times$  E2 E3.
Tactic Notation "autos" "*" constr(E1) ", " constr(E2) ", " constr(E3) ", " constr(E4)
:=
    lets: E1; autos $\times$  E2 E3 E4.
Tactic Notation "autos" "*" constr(E1) ", " constr(E2) ", " constr(E3) ", " constr(E4)

```

```

", "
  constr(E5):=
    lets: E1; autos × E2 E3 E4 E5.

  auto_false is a version of auto able to spot some contradictions. There is an ad-hoc
  support for goals in  $\leftrightarrow$ : split is called first. auto_false  $\neg$  and auto_false  $\times$  are also available.

Ltac auto_false_base cont :=
  try solve [
    intros_all; try match goal with  $\vdash \_ \leftrightarrow \_ \Rightarrow$  split end;
    solve [ cont tt | intros_all; false; cont tt ] ].

Tactic Notation "auto_false" :=
  auto_false_base ltac:(fun tt  $\Rightarrow$  auto).
Tactic Notation "auto_false" "~" :=
  auto_false_base ltac:(fun tt  $\Rightarrow$  auto_tilde).
Tactic Notation "auto_false" "*" :=
  auto_false_base ltac:(fun tt  $\Rightarrow$  auto_star).
Tactic Notation "dauto" :=
  dintuition eauto.

```

2.15.5 Parsing for Light Automation

Any tactic followed by the symbol $\neg$ will have *auto_tilde* called on all of its subgoals. Three exceptions:

- *cuts* and *asserts* only call *auto* on their first subgoal,
- *apply* $\neg$ relies on *sapply* rather than *apply*,
- *tryfalse* $\neg$ is defined as *tryfalse* by *auto_tilde*.

Some builtin tactics are not defined using tactic notations and thus cannot be extended, e.g., *simpl* and *unfold*. For these, notation such as *simpl* $\neg$ will not be available.

```

Tactic Notation "equates" "~" constr(E) :=
  equates E; auto_tilde.
Tactic Notation "equates" "~" constr(n1) constr(n2) :=
  equates n1 n2; auto_tilde.
Tactic Notation "equates" "~" constr(n1) constr(n2) constr(n3) :=
  equates n1 n2 n3; auto_tilde.
Tactic Notation "equates" "~" constr(n1) constr(n2) constr(n3) constr(n4) :=
  equates n1 n2 n3 n4; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) :=
  applies_eq H; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(E) :=

```

```

    applies_eq H E; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(n1) constr(n2) :=
    applies_eq H n1 n2; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) :=
    applies_eq H n1 n2 n3; auto_tilde.
Tactic Notation "applies_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) constr(n4)
:=
    applies_eq H n1 n2 n3 n4; auto_tilde.
Tactic Notation "apply" "~" constr(H) :=
    sapply H; auto_tilde.
Tactic Notation "destruct" "~" constr(H) :=
    destruct H; auto_tilde.
Tactic Notation "destruct" "~" constr(H) "as" simple_intropattern(I) :=
    destruct H as I; auto_tilde.
Tactic Notation "f_equal" "~" :=
    f_equal; auto_tilde.
Tactic Notation "induction" "~" constr(H) :=
    induction H; auto_tilde.
Tactic Notation "inversion" "~" constr(H) :=
    inversion H; auto_tilde.
Tactic Notation "split" "~" :=
    split; auto_tilde.
Tactic Notation "subst" "~" :=
    subst; auto_tilde.
Tactic Notation "right" "~" :=
    right; auto_tilde.
Tactic Notation "left" "~" :=
    left; auto_tilde.
Tactic Notation "constructor" "~" :=
    constructor; auto_tilde.
Tactic Notation "constructors" "~" :=
    constructors; auto_tilde.
Tactic Notation "false" "~" :=
    false; auto_tilde.
Tactic Notation "false" "~" constr(E) :=
    false_then E ltac:(fun _ => auto_tilde.).
Tactic Notation "false" "~" constr(E0) constr(E1) :=
    false → (» E0 E1).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) :=
    false → (» E0 E1 E2).
Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) constr(E3) :=
    false → (» E0 E1 E2 E3).

```

```

Tactic Notation "false" "~" constr(E0) constr(E1) constr(E2) constr(E3) constr(E4)
:=
  false¬ (> E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "~" :=
  try solve [ false¬ ].
Tactic Notation "asserts" "~" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_tilde | idtac ].
Tactic Notation "asserts" "~" ":" constr(E) :=
  let H := fresh "H" in asserts¬ H: E.
Tactic Notation "cuts" "~" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_tilde | idtac ].
Tactic Notation "cuts" "~" ":" constr(E) :=
  cuts: E; [ auto_tilde | idtac ].
Tactic Notation "lets" "~" simple_intropattern(I) ":" constr(E) :=
  lets I: E; auto_tilde.
Tactic Notation "lets" "~" simple_intropattern(I) ":" constr(E0)
  constr(A1) :=
  lets I: E0 A1; auto_tilde.
Tactic Notation "lets" "~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) :=
  lets I: E0 A1 A2; auto_tilde.
Tactic Notation "lets" "~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets I: E0 A1 A2 A3; auto_tilde.
Tactic Notation "lets" "~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets I: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "lets" "~" simple_intropattern(I) ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets I: E0 A1 A2 A3 A4 A5; auto_tilde.
Tactic Notation "lets" "~" ":" constr(E) :=
  lets: E; auto_tilde.
Tactic Notation "lets" "~" ":" constr(E0)
  constr(A1) :=
  lets: E0 A1; auto_tilde.
Tactic Notation "lets" "~" ":" constr(E0)
  constr(A1) constr(A2) :=
  lets: E0 A1 A2; auto_tilde.
Tactic Notation "lets" "~" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets: E0 A1 A2 A3; auto_tilde.
Tactic Notation "lets" "~" ":" constr(E0)

```

```

    constr(A1) constr(A2) constr(A3) constr(A4) :=
      lets: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "lets" "~" ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
      lets: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E) :=
  forwards I: E; auto_tilde.
Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E0)
    constr(A1) :=
      forwards I: E0 A1; auto_tilde.
Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) :=
      forwards I: E0 A1 A2; auto_tilde.
Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) :=
      forwards I: E0 A1 A2 A3; auto_tilde.
Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) :=
      forwards I: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "forwards" "~" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
      forwards I: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "forwards" "~" ":" constr(E) :=
  forwards: E; auto_tilde.
Tactic Notation "forwards" "~" ":" constr(E0)
    constr(A1) :=
      forwards: E0 A1; auto_tilde.
Tactic Notation "forwards" "~" ":" constr(E0)
    constr(A1) constr(A2) :=
      forwards: E0 A1 A2; auto_tilde.
Tactic Notation "forwards" "~" ":" constr(E0)
    constr(A1) constr(A2) constr(A3) :=
      forwards: E0 A1 A2 A3; auto_tilde.
Tactic Notation "forwards" "~" ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) :=
      forwards: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "forwards" "~" ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
      forwards: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "applys" "~" constr(H) :=
  supply H; auto_tilde. Tactic Notation "applys" "~" constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.

```



```

Tactic Notation "applies" "~" constr(E0) constr(A1) :=
  applies E0 A1; auto_tilde.
Tactic Notation "applies" "~" constr(E0) constr(A1) constr(A2) :=
  applies E0 A1 A2; auto_tilde.
Tactic Notation "applies" "~" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applies E0 A1 A2 A3; auto_tilde.
Tactic Notation "applies" "~" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  applies E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "applies" "~" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  applies E0 A1 A2 A3 A4 A5; auto_tilde.
Tactic Notation "specializes" "~" hyp(H) :=
  specializes H; auto_tilde.
Tactic Notation "specializes" "~" hyp(H) constr(A1) :=
  specializes H A1; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H A1 A2 A3; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  specializes H A1 A2 A3 A4; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_tilde.
Tactic Notation "fapply" "~" constr(E) :=
  fapply E; auto_tilde.
Tactic Notation "sapply" "~" constr(E) :=
  sapply E; auto_tilde.
Tactic Notation "logic" "~" constr(E) :=
  logic_base E ltac:(fun _ => auto_tilde).
Tactic Notation "intros_all" "~" :=
  intros_all; auto_tilde.
Tactic Notation "unfolds" "~" :=
  unfolds; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) :=
  unfolds F1; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) "," constr(F2) :=
  unfolds F1, F2; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) "," constr(F2) "," constr(F3) :=

```

```

unfolds F1, F2, F3; auto_tilde.
Tactic Notation "unfolds" "~" constr(F1) "," constr(F2) "," constr(F3) ","
constr(F4) :=
unfolds F1, F2, F3, F4; auto_tilde.
Tactic Notation "simple" "~" :=
simpl; auto_tilde.
Tactic Notation "simple" "~" "in" hyp(H) :=
simpl in H; auto_tilde.
Tactic Notation "simpls" "~" :=
simpls; auto_tilde.
Tactic Notation "hnfs" "~" :=
hnfs; auto_tilde.
Tactic Notation "hnfs" "~" "in" hyp(H) :=
hnf in H; auto_tilde.
Tactic Notation "subst" "~" :=
subst; auto_tilde.
Tactic Notation "intro_hyp" "~" hyp(H) :=
subst_hyp H; auto_tilde.
Tactic Notation "intro_subst" "~" :=
intro_subst; auto_tilde.
Tactic Notation "subst_eq" "~" constr(E) :=
subst_eq E; auto_tilde.
Tactic Notation "rewrite" "~" constr(E) :=
rewrite E; auto_tilde.
Tactic Notation "rewrite" "~" "<-" constr(E) :=
rewrite ← E; auto_tilde.
Tactic Notation "rewrite" "~" constr(E) "in" hyp(H) :=
rewrite E in H; auto_tilde.
Tactic Notation "rewrite" "~" "<-" constr(E) "in" hyp(H) :=
rewrite ← E in H; auto_tilde.
Tactic Notation "rewrites" "~" constr(E) :=
rewrites E; auto_tilde.
Tactic Notation "rewrites" "~" constr(E) "in" hyp(H) :=
rewrites E in H; auto_tilde.
Tactic Notation "rewrites" "~" constr(E) "in" "*" :=
rewrites E in *; auto_tilde.
Tactic Notation "rewrites" "~" "<-" constr(E) :=
rewrites ← E; auto_tilde.
Tactic Notation "rewrites" "~" "<-" constr(E) "in" hyp(H) :=
rewrites ← E in H; auto_tilde.
Tactic Notation "rewrites" "~" "<-" constr(E) "in" "*" :=
rewrites ← E in *; auto_tilde.

```

Tactic Notation "rewrite_all" "~" constr(*E*) :=
 rewrite_all E; auto_tilde.

Tactic Notation "rewrite_all" "~" "<-" constr(*E*) :=
 rewrite_all ← E; auto_tilde.

Tactic Notation "rewrite_all" "~" constr(*E*) "in" ident(*H*) :=
 rewrite_all E in H; auto_tilde.

Tactic Notation "rewrite_all" "~" "<-" constr(*E*) "in" ident(*H*) :=
 rewrite_all ← E in H; auto_tilde.

Tactic Notation "rewrite_all" "~" constr(*E*) "in" "*" :=
 *rewrite_all E in *; auto_tilde.*

Tactic Notation "rewrite_all" "~" "<-" constr(*E*) "in" "*" :=
 *rewrite_all ← E in *; auto_tilde.*

Tactic Notation "asserts_rewrite" "~" constr(*E*) :=
 asserts_rewrite E; auto_tilde.

Tactic Notation "asserts_rewrite" "~" "<-" constr(*E*) :=
 asserts_rewrite ← E; auto_tilde.

Tactic Notation "asserts_rewrite" "~" constr(*E*) "in" hyp(*H*) :=
 asserts_rewrite E in H; auto_tilde.

Tactic Notation "asserts_rewrite" "~" "<-" constr(*E*) "in" hyp(*H*) :=
 asserts_rewrite ← E in H; auto_tilde.

Tactic Notation "asserts_rewrite" "~" constr(*E*) "in" "*" :=
 *asserts_rewrite E in *; auto_tilde.*

Tactic Notation "asserts_rewrite" "~" "<-" constr(*E*) "in" "*" :=
 *asserts_rewrite ← E in *; auto_tilde.*

Tactic Notation "cuts_rewrite" "~" constr(*E*) :=
 cuts_rewrite E; auto_tilde.

Tactic Notation "cuts_rewrite" "~" "<-" constr(*E*) :=
 cuts_rewrite ← E; auto_tilde.

Tactic Notation "cuts_rewrite" "~" constr(*E*) "in" hyp(*H*) :=
 cuts_rewrite E in H; auto_tilde.

Tactic Notation "cuts_rewrite" "~" "<-" constr(*E*) "in" hyp(*H*) :=
 cuts_rewrite ← E in H; auto_tilde.

Tactic Notation "erewrite" "~" constr(*E*) :=
 erewrite E; auto_tilde.

Tactic Notation "erewrites" "~" constr(*E*) :=
 erewrites E; auto_tilde.

Tactic Notation "fequal" "~" :=
 fequal; auto_tilde.

Tactic Notation "fequals" "~" :=
 fequals; auto_tilde.

Tactic Notation "pi_rewrite" "~" constr(*E*) :=

```

    pi_rewrite E; auto_tilde.
Tactic Notation "pi_rewrite" "~" constr(E) "in" hyp(H) :=
    pi_rewrite E in H; auto_tilde.
Tactic Notation "invert" "~" hyp(H) :=
    invert H; auto_tilde.
Tactic Notation "inverts" "~" hyp(H) :=
    inverts H; auto_tilde.
Tactic Notation "inverts" "~" hyp(E) "as" :=
    inverts E as; auto_tilde.
Tactic Notation "injects" "~" hyp(H) :=
    injects H; auto_tilde.
Tactic Notation "inversions" "~" hyp(H) :=
    inversions H; auto_tilde.
Tactic Notation "cases" "~" constr(E) "as" ident(H) :=
    cases E as H; auto_tilde.
Tactic Notation "cases" "~" constr(E) :=
    cases E; auto_tilde.
Tactic Notation "case_if" "~" :=
    case_if; auto_tilde.
Tactic Notation "case_ifs" "~" :=
    case_ifs; auto_tilde.
Tactic Notation "case_if" "~" "in" hyp(H) :=
    case_if in H; auto_tilde.
Tactic Notation "cases_if" "~" :=
    cases_if; auto_tilde.
Tactic Notation "cases_if" "~" "in" hyp(H) :=
    cases_if in H; auto_tilde.
Tactic Notation "destruct_if" "~" :=
    destruct_if; auto_tilde.
Tactic Notation "destruct_if" "~" "in" hyp(H) :=
    destruct_if in H; auto_tilde.
Tactic Notation "cases'" "~" constr(E) "as" ident(H) :=
    cases' E as H; auto_tilde.
Tactic Notation "cases'" "~" constr(E) :=
    cases' E; auto_tilde.
Tactic Notation "cases_if'" "~" "as" ident(H) :=
    cases_if' as H; auto_tilde.
Tactic Notation "cases_if'" "~" :=
    cases_if'; auto_tilde.
Tactic Notation "decides_equality" "~" :=
    decides_equality; auto_tilde.

```

Tactic Notation "iff" "~" :=
 iff; *auto_tilde*.
 Tactic Notation "iff" "~" *simple_intropattern*(*I*) :=
 iff I; *auto_tilde*.
 Tactic Notation "splits" "~" :=
 splits; *auto_tilde*.
 Tactic Notation "splits" "~" *constr*(*N*) :=
 splits N; *auto_tilde*.
 Tactic Notation "destructs" "~" *constr*(*T*) :=
 destructs T; *auto_tilde*.
 Tactic Notation "destructs" "~" *constr*(*N*) *constr*(*T*) :=
 destructs N T; *auto_tilde*.
 Tactic Notation "branch" "~" *constr*(*N*) :=
 branch N; *auto_tilde*.
 Tactic Notation "branch" "~" *constr*(*K*) "of" *constr*(*N*) :=
 branch K of N; *auto_tilde*.
 Tactic Notation "branches" "~" :=
 branches; *auto_tilde*.
 Tactic Notation "branches" "~" *constr*(*T*) :=
 branches T; *auto_tilde*.
 Tactic Notation "branches" "~" *constr*(*N*) *constr*(*T*) :=
 branches N T; *auto_tilde*.
 Tactic Notation "exists" "~" :=
 $\exists$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) :=
 $\exists T1$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) *constr*(*T2*) :=
 $\exists T1 T2$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) *constr*(*T2*) *constr*(*T3*) :=
 $\exists T1 T2 T3$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) *constr*(*T2*) *constr*(*T3*) *constr*(*T4*) :=
 $\exists T1 T2 T3 T4$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) *constr*(*T2*) *constr*(*T3*) *constr*(*T4*)
 constr(*T5*) :=
 $\exists T1 T2 T3 T4 T5$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) *constr*(*T2*) *constr*(*T3*) *constr*(*T4*)
 constr(*T5*) *constr*(*T6*) :=
 $\exists T1 T2 T3 T4 T5 T6$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) "," *constr*(*T2*) :=
 $\exists T1 T2$; *auto_tilde*.
 Tactic Notation "exists" "~" *constr*(*T1*) "," *constr*(*T2*) "," *constr*(*T3*) :=

```

    ∃ T1 T2 T3; auto_tilde.
Tactic Notation "exists" "~" constr(T1) "," constr(T2) "," constr(T3) ","
constr(T4) :=
    ∃ T1 T2 T3 T4; auto_tilde.
Tactic Notation "exists" "~" constr(T1) "," constr(T2) "," constr(T3) ","
constr(T4) "," constr(T5) :=
    ∃ T1 T2 T3 T4 T5; auto_tilde.
Tactic Notation "exists" "~" constr(T1) "," constr(T2) "," constr(T3) ","
constr(T4) "," constr(T5) "," constr(T6) :=
    ∃ T1 T2 T3 T4 T5 T6; auto_tilde.

```

2.15.6 Parsing for Strong Automation

Any tactic followed by the symbol $\times$ will have `auto $\times$` called on all of its subgoals. The exceptions to these rules are the same as for light automation.

Exception: use `subs $\times$` instead of `subst $\times$` if you import the library *Equivalence* from *Stdlib*.

```

Tactic Notation "equates" "*" constr(E) :=
    equates E; auto_star.
Tactic Notation "equates" "*" constr(n1) constr(n2) :=
    equates n1 n2; auto_star.
Tactic Notation "equates" "*" constr(n1) constr(n2) constr(n3) :=
    equates n1 n2 n3; auto_star.
Tactic Notation "equates" "*" constr(n1) constr(n2) constr(n3) constr(n4) :=
    equates n1 n2 n3 n4; auto_star.
Tactic Notation "applies_eq" "*" constr(H) :=
    applies_eq H; auto_star.
Tactic Notation "applies_eq" "*" constr(H) constr(E) :=
    applies_eq H E; auto_star.
Tactic Notation "applies_eq" "*" constr(H) constr(n1) constr(n2) :=
    applies_eq H n1 n2; auto_star.
Tactic Notation "applies_eq" "*" constr(H) constr(n1) constr(n2) constr(n3) :=
    applies_eq H n1 n2 n3; auto_star.
Tactic Notation "applies_eq" "*" constr(H) constr(n1) constr(n2) constr(n3) constr(n4)
:=
    applies_eq H n1 n2 n3 n4; auto_star.
Tactic Notation "apply" "*" constr(H) :=
    sapply H; auto_star.
Tactic Notation "destruct" "*" constr(H) :=
    destruct H; auto_star.
Tactic Notation "destruct" "*" constr(H) "as" simple_intropattern(I) :=
    destruct H as I; auto_star.

```

```

Tactic Notation "f_equal" "*" :=
  f_equal; auto_star.
Tactic Notation "induction" "*" constr(H) :=
  induction H; auto_star.
Tactic Notation "inversion" "*" constr(H) :=
  inversion H; auto_star.
Tactic Notation "split" "*" :=
  split; auto_star.
Tactic Notation "subs" "*" :=
  subst; auto_star.
Tactic Notation "subst" "*" :=
  subst; auto_star.
Tactic Notation "right" "*" :=
  right; auto_star.
Tactic Notation "left" "*" :=
  left; auto_star.
Tactic Notation "constructor" "*" :=
  constructor; auto_star.
Tactic Notation "constructors" "*" :=
  constructors; auto_star.
Tactic Notation "false" "*" :=
  false; auto_star.
Tactic Notation "false" "*" constr(E) :=
  false_then E ltac:(fun _ => auto_star).
Tactic Notation "false" "*" constr(E0) constr(E1) :=
  false × (» E0 E1).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) :=
  false × (» E0 E1 E2).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) constr(E3) :=
  false × (» E0 E1 E2 E3).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) constr(E3) constr(E4)
:=
  false × (» E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "*" :=
  try solve [ false × ].
Tactic Notation "asserts" "*" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_star | idtac ].
Tactic Notation "asserts" "*" ":" constr(E) :=
  let H := fresh "H" in asserts × H: E.
Tactic Notation "cuts" "*" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_star | idtac ].
Tactic Notation "cuts" "*" ":" constr(E) :=

```

```

    cuts: E; [ auto_star | idtac ].
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E) :=
    lets I: E; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
    constr(A1) :=
    lets I: E0 A1; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) :=
    lets I: E0 A1 A2; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) :=
    lets I: E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) :=
    lets I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    lets I: E0 A1 A2 A3 A4 A5; auto_star.
Tactic Notation "lets" "*" ":" constr(E) :=
    lets E; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
    constr(A1) :=
    lets E0 A1; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
    constr(A1) constr(A2) :=
    lets E0 A1 A2; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
    constr(A1) constr(A2) constr(A3) :=
    lets E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) :=
    lets E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
    constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    lets E0 A1 A2 A3 A4 A5; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E) :=
    forwards I: E; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
    constr(A1) :=
    forwards I: E0 A1; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
    constr(A1) constr(A2) :=

```



```

forwards I: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
constr(A1) constr(A2) constr(A3) :=
forwards I: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
constr(A1) constr(A2) constr(A3) constr(A4) :=
forwards I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E0)
constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
forwards I: E0 A1 A2 A3 A4 A5; auto_star.
Tactic Notation "forwards" "*" ":" constr(E) :=
forwards: E; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) :=
forwards: E0 A1; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) :=
forwards: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) constr(A3) :=
forwards: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) constr(A3) constr(A4) :=
forwards: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
forwards: E0 A1 A2 A3 A4 A5; auto_star.
Tactic Notation "applys" "*" constr(H) :=
supply H; auto_star. Tactic Notation "applys" "*" constr(E0) constr(A1) :=
applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) :=
applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) :=
applys E0 A1 A2; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) :=
applys E0 A1 A2 A3; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
applys E0 A1 A2 A3 A4; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
applys E0 A1 A2 A3 A4 A5; auto_star.

```

```

Tactic Notation "specializes" "*" hyp(H) :=
  specializes H; auto_star.
Tactic Notation "specializes" "~" hyp(H) constr(A1) :=
  specializes H A1; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H A1 A2 A3; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  specializes H A1 A2 A3 A4; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_star.
Tactic Notation "fapply" "*" constr(E) :=
  fapply E; auto_star.
Tactic Notation "sapply" "*" constr(E) :=
  sapply E; auto_star.
Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ => auto_star).
Tactic Notation "intros_all" "*" :=
  intros_all; auto_star.
Tactic Notation "unfolds" "*" :=
  unfolds; auto_star.
Tactic Notation "unfolds" "*" constr(F1) :=
  unfolds F1; auto_star.
Tactic Notation "unfolds" "*" constr(F1) "," constr(F2) :=
  unfolds F1, F2; auto_star.
Tactic Notation "unfolds" "*" constr(F1) "," constr(F2) "," constr(F3) :=
  unfolds F1, F2, F3; auto_star.
Tactic Notation "unfolds" "*" constr(F1) "," constr(F2) "," constr(F3) ","
constr(F4) :=
  unfolds F1, F2, F3, F4; auto_star.
Tactic Notation "simple" "*" :=
  simpl; auto_star.
Tactic Notation "simple" "*" "in" hyp(H) :=
  simpl in H; auto_star.
Tactic Notation "simpls" "*" :=
  simpls; auto_star.
Tactic Notation "hnfs" "*" :=
  hnfs; auto_star.

```

```

Tactic Notation "hnfs" "*" "in" hyp(H) :=
  hnf in H; auto_star.
Tactic Notation "subst" "*" :=
  subst; auto_star.
Tactic Notation "intro_hyp" "*" hyp(H) :=
  subst_hyp H; auto_star.
Tactic Notation "intro_subst" "*" :=
  intro_subst; auto_star.
Tactic Notation "subst_eq" "*" constr(E) :=
  subst_eq E; auto_star.
Tactic Notation "rewrite" "*" constr(E) :=
  rewrite E; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) :=
  rewrite ← E; auto_star.
Tactic Notation "rewrite" "*" constr(E) "in" hyp(H) :=
  rewrite E in H; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  rewrite ← E in H; auto_star.
Tactic Notation "rewrites" "*" constr(E) :=
  rewrites E; auto_star.
Tactic Notation "rewrites" "*" constr(E) "in" hyp(H) :=
  rewrites E in H; auto_star.
Tactic Notation "rewrites" "*" constr(E) "in" "*" :=
  rewrites E in *; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) :=
  rewrites ← E; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" hyp(H) :=
  rewrites ← E in H; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" "*" :=
  rewrites ← E in *; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) :=
  rewrite_all E; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) :=
  rewrite_all ← E; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) "in" ident(H) :=
  rewrite_all E in H; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" ident(H) :=
  rewrite_all ← E in H; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) "in" "*" :=
  rewrite_all E in *; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" "*" :=
  rewrite_all ← E in *; auto_star.

```

Tactic Notation "asserts_rewrite" "*" constr(*E*) :=
 asserts_rewrite E; auto_star.
 Tactic Notation "asserts_rewrite" "*" "<-" constr(*E*) :=
 asserts_rewrite ← E; auto_star.
 Tactic Notation "asserts_rewrite" "*" constr(*E*) "in" *hyp*(*H*) :=
 asserts_rewrite E; auto_star.
 Tactic Notation "asserts_rewrite" "*" "<-" constr(*E*) "in" *hyp*(*H*) :=
 asserts_rewrite ← E; auto_star.
 Tactic Notation "asserts_rewrite" "*" constr(*E*) "in" "*" :=
 *asserts_rewrite E in *; auto_tilde*.
 Tactic Notation "asserts_rewrite" "*" "<-" constr(*E*) "in" "*" :=
 *asserts_rewrite ← E in *; auto_tilde*.
 Tactic Notation "cuts_rewrite" "*" constr(*E*) :=
 cuts_rewrite E; auto_star.
 Tactic Notation "cuts_rewrite" "*" "<-" constr(*E*) :=
 cuts_rewrite ← E; auto_star.
 Tactic Notation "cuts_rewrite" "*" constr(*E*) "in" *hyp*(*H*) :=
 cuts_rewrite E in H; auto_star.
 Tactic Notation "cuts_rewrite" "*" "<-" constr(*E*) "in" *hyp*(*H*) :=
 cuts_rewrite ← E in H; auto_star.
 Tactic Notation "erewrite" "*" constr(*E*) :=
 erewrite E; auto_star.
 Tactic Notation "erewrites" "*" constr(*E*) :=
 erewrites E; auto_star.
 Tactic Notation "fequal" "*" :=
 fequal; auto_star.
 Tactic Notation "fequals" "*" :=
 fequals; auto_star.
 Tactic Notation "pi_rewrite" "*" constr(*E*) :=
 pi_rewrite E; auto_star.
 Tactic Notation "pi_rewrite" "*" constr(*E*) "in" *hyp*(*H*) :=
 pi_rewrite E in H; auto_star.
 Tactic Notation "invert" "*" *hyp*(*H*) :=
 invert H; auto_star.
 Tactic Notation "inverts" "*" *hyp*(*H*) :=
 inverts H; auto_star.
 Tactic Notation "inverts" "*" *hyp*(*E*) "as" :=
 inverts E as; auto_star.
 Tactic Notation "injects" "*" *hyp*(*H*) :=
 injects H; auto_star.
 Tactic Notation "inversions" "*" *hyp*(*H*) :=

```

inversions H; auto_star.
Tactic Notation "cases" "*" constr(E) "as" ident(H) :=
  cases E as H; auto_star.
Tactic Notation "cases" "*" constr(E) :=
  cases E; auto_star.
Tactic Notation "case_if" "*" :=
  case_if; auto_star.
Tactic Notation "case_ifs" "*" :=
  case_ifs; auto_star.
Tactic Notation "case_if" "*" "in" hyp(H) :=
  case_if in H; auto_star.
Tactic Notation "cases_if" "*" :=
  cases_if; auto_star.
Tactic Notation "cases_if" "*" "in" hyp(H) :=
  cases_if in H; auto_star.
Tactic Notation "destruct_if" "*" :=
  destruct_if; auto_star.
Tactic Notation "destruct_if" "*" "in" hyp(H) :=
  destruct_if in H; auto_star.
Tactic Notation "cases'" "*" constr(E) "as" ident(H) :=
  cases' E as H; auto_star.
Tactic Notation "cases'" "*" constr(E) :=
  cases' E; auto_star.
Tactic Notation "cases_if'" "*" "as" ident(H) :=
  cases_if' as H; auto_star.
Tactic Notation "cases_if'" "*" :=
  cases_if'; auto_star.
Tactic Notation "decides_equality" "*" :=
  decides_equality; auto_star.
Tactic Notation "iff" "*" :=
  iff; auto_star.
Tactic Notation "iff" "*" simple_intropattern(I) :=
  iff I; auto_star.
Tactic Notation "splits" "*" :=
  splits; auto_star.
Tactic Notation "splits" "*" constr(N) :=
  splits N; auto_star.
Tactic Notation "destructs" "*" constr(T) :=
  destructs T; auto_star.
Tactic Notation "destructs" "*" constr(N) constr(T) :=
  destructs N T; auto_star.

```

Tactic Notation "branch" "*" constr(N) :=
 branch N ; *auto_star*.
 Tactic Notation "branch" "*" constr(K) "of" constr(N) :=
 branch K of N ; *auto_star*.
 Tactic Notation "branches" "*" constr(T) :=
 branches T ; *auto_star*.
 Tactic Notation "branches" "*" constr(N) constr(T) :=
 branches N T ; *auto_star*.
 Tactic Notation "exists" "*" :=
 $\exists$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) :=
 $\exists$ $T1$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) constr($T2$) :=
 $\exists$ $T1$ $T2$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) constr($T2$) constr($T3$) :=
 $\exists$ $T1$ $T2$ $T3$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) constr($T2$) constr($T3$) constr($T4$) :=
 $\exists$ $T1$ $T2$ $T3$ $T4$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) constr($T2$) constr($T3$) constr($T4$)
 constr($T5$) :=
 $\exists$ $T1$ $T2$ $T3$ $T4$ $T5$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) constr($T2$) constr($T3$) constr($T4$)
 constr($T5$) constr($T6$) :=
 $\exists$ $T1$ $T2$ $T3$ $T4$ $T5$ $T6$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) "," constr($T2$) :=
 $\exists$ $T1$ $T2$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) "," constr($T2$) "," constr($T3$) :=
 $\exists$ $T1$ $T2$ $T3$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) "," constr($T2$) "," constr($T3$) ","
 constr($T4$) :=
 $\exists$ $T1$ $T2$ $T3$ $T4$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) "," constr($T2$) "," constr($T3$) ","
 constr($T4$) "," constr($T5$) :=
 $\exists$ $T1$ $T2$ $T3$ $T4$ $T5$; *auto_star*.
 Tactic Notation "exists" "*" constr($T1$) "," constr($T2$) "," constr($T3$) ","
 constr($T4$) "," constr($T5$) "," constr($T6$) :=
 $\exists$ $T1$ $T2$ $T3$ $T4$ $T5$ $T6$; *auto_star*.

2.16 Tactics to Sort Out the Proof Context

2.16.1 Hiding Hypotheses

Definition `ltac_something` (P :Type) (e : P) := e .

Notation "'Something'" :=
(`@ltac_something - _`).

Lemma `ltac_something_eq` : $\forall (e$:Type),
 $e = (@ltac\_something - e)$.

Proof using. auto. Qed.

Lemma `ltac_something_hide` : $\forall (e$:Type),
 $e \rightarrow (@ltac\_something - e)$.

Proof using. auto. Qed.

Lemma `ltac_something_show` : $\forall (e$:Type),
 $(@ltac\_something - e) \rightarrow e$.

Proof using. auto. Qed.

hide_def x and *show_def* x can be used to hide/show the body of the definition x .

Tactic Notation "hide_def" *hyp*(x) :=
let x' := `constr:( $x$ )` in
let T := `eval unfold  $x$  in  $x'$` in
change T with (`@ltac_something -  $T$`) in x .

Tactic Notation "show_def" *hyp*(x) :=
let x' := `constr:( $x$ )` in
let U := `eval unfold  $x$  in  $x'$` in
match U with `@ltac_something - ? $T$` $\Rightarrow$
change U with T in x end.

show_def unfolds *Something* in the goal

Tactic Notation "show_def" :=
unfold `ltac_something`.

Tactic Notation "show_def" "in" *hyp*(H) :=
unfold `ltac_something` in H .

Tactic Notation "show_def" "in" "*" :=
unfold `ltac_something` in *.

hide_defs and *show_defs* applies to all definitions

Tactic Notation "hide_defs" :=
repeat match goal with H := $?T \vdash - \Rightarrow$
match T with
| `@ltac_something - _` $\Rightarrow$ fail 1
| _ $\Rightarrow$ change T with (`@ltac_something -  $T$`) in H

```

    end
end.

```

```

Tactic Notation "show_defs" :=
  repeat match goal with H := (@ltac_something _ ?T) ⊢ _ ⇒
    change (@ltac_something _ T) with T in H end.

```

hide_hyp H replaces the type of *H* with the notation *Something* and *show_hyp H* reveals the type of the hypothesis. Note that the hidden type of *H* remains convertible the real type of *H*.

```

Tactic Notation "show_hyp" hyp(H) :=
  apply ltac_something_show in H.

```

```

Tactic Notation "hide_hyp" hyp(H) :=
  apply ltac_something_hide in H.

```

hide_hyps and *show_hyps* can be used to hide/show all hypotheses of type *Prop*.

```

Tactic Notation "show_hyps" :=
  repeat match goal with
    H: @ltac_something _ _ ⊢ _ ⇒ show_hyp H end.

```

```

Tactic Notation "hide_hyps" :=
  repeat match goal with H: ?T ⊢ _ ⇒
    match type of T with
    | Prop ⇒
      match T with
      | @ltac_something _ _ ⇒ fail 2
      | _ ⇒ hide_hyp H
      end
    | _ ⇒ fail 1
    end
  end.

```

hide H and *show H* automatically select between *hide_hyp* or *hide_def*, and *show_hyp* or *show_def*. Similarly *hide_all* and *show_all* apply to all.

```

Tactic Notation "hide" hyp(H) :=
  first [hide_def H | hide_hyp H].

```

```

Tactic Notation "show" hyp(H) :=
  first [show_def H | show_hyp H].

```

```

Tactic Notation "hide_all" :=
  hide_hyps; hide_defs.

```

```

Tactic Notation "show_all" :=
  unfold ltac_something in *.

```

hide_term E can be used to hide a term from the goal. *show_term* or *show_term E* can be used to reveal it. *hide_term E in H* can be used to specify an hypothesis.


```

Tactic Notation "hide_term" constr(E) :=
  change E with (@ltac_something _ E).
Tactic Notation "show_term" constr(E) :=
  change (@ltac_something _ E) with E.
Tactic Notation "show_term" :=
  unfold ltac_something.

Tactic Notation "hide_term" constr(E) "in" hyp(H) :=
  change E with (@ltac_something _ E) in H.
Tactic Notation "show_term" constr(E) "in" hyp(H) :=
  change (@ltac_something _ E) with E in H.
Tactic Notation "show_term" "in" hyp(H) :=
  unfold ltac_something in H.

```

show_unfold R unfolds the definition of *R* and reveals the hidden definition of *R*. –
 todo:test, and implement using unfold simply

```

Tactic Notation "show_unfold" constr(R1) :=
  unfold R1; show_def.
Tactic Notation "show_unfold" constr(R1) ", " constr(R2) :=
  unfold R1, R2; show_def.

```

2.16.2 Sorting Hypotheses

sort sorts out hypotheses from the context by moving all the propositions (hypotheses of type Prop) to the bottom of the context.

```

Ltac sort_tactic :=
  try match goal with H: ?T ⊢ _ ⇒
  match type of T with Prop ⇒
    generalizes H; (try sort_tactic); intro
  end end.

Tactic Notation "sort" :=
  sort_tactic.

```

2.16.3 Clearing Hypotheses

clears X1 ... XN is a variation on *clear* which clears the variables *X1..XN* as well as all the hypotheses which depend on them. Contrary to *clear*, it never fails.

```

Tactic Notation "clears" ident(X1) :=
  let rec doit _ :=
  match goal with
  | H:context[X1] ⊢ _ ⇒ clear H; try (doit tt)
  | _ ⇒ clear X1
  end in doit tt.

```

```

Tactic Notation "clears" ident(X1) ident(X2) :=
  clears X1; clears X2.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) :=
  clears X1; clears X2; clears X3.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4) :=
  clears X1; clears X2; clears X3; clears X4.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
  ident(X5) :=
  clears X1; clears X2; clears X3; clears X4; clears X5.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
  ident(X5) ident(X6) :=
  clears X1; clears X2; clears X3; clears X4; clears X5; clears X6.

```

clears (without any argument) clears all the unused variables from the context. In other words, it removes any variable which is not a proposition (i.e. not of type Prop) and which does not appear in another hypothesis nor in the goal.

```

Ltac clears_tactic :=
  match goal with H: ?T ⊢ _ ⇒
  match type of T with
  | Prop ⇒ generalizes H; (try clears_tactic); intro
  | ?TT ⇒ clear H; (try clears_tactic)
  | ?TT ⇒ generalizes H; (try clears_tactic); intro
  end end.

```

```

Tactic Notation "clears" :=
  clears_tactic.

```

clears_all clears all the hypotheses from the context that can be cleared. It leaves only the hypotheses that are mentioned in the goal.

```

Ltac clears_or_generalizes_all_core :=
  repeat match goal with H: _ ⊢ _ ⇒
    first [ clear H | generalizes H ] end.

```

```

Tactic Notation "clears_all" :=
  generalize ltac_mark;
  clears_or_generalizes_all_core;
  intro_until_mark.

```

clears_but H1 H2 .. HN clears all hypotheses except the one that are mentioned and those that cannot be cleared.

```

Ltac clears_but_core cont :=
  generalize ltac_mark;
  cont tt;
  clears_or_generalizes_all_core;
  intro_until_mark.

```

```

Tactic Notation "clears_but" :=
  clears_but_core ltac:(fun _ => idtac).
Tactic Notation "clears_but" ident(H1) :=
  clears_but_core ltac:(fun _ => gen H1).
Tactic Notation "clears_but" ident(H1) ident(H2) :=
  clears_but_core ltac:(fun _ => gen H1 H2).
Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) :=
  clears_but_core ltac:(fun _ => gen H1 H2 H3).
Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) ident(H4) :=
  clears_but_core ltac:(fun _ => gen H1 H2 H3 H4).
Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) ident(H4) ident(H5) :=
  clears_but_core ltac:(fun _ => gen H1 H2 H3 H4 H5).

```

Lemma demo_clears_all_and_clears_but :

$\forall x\ y:\text{nat}, y < 2 \rightarrow x = x \rightarrow x \geq 2 \rightarrow x < 3 \rightarrow \text{True}.$

Proof using.

```

  introv M1 M2 M3. dup 6.
  clears_all. auto.
  clears_but M3. auto.
  clears_but y. auto.
  clears_but x. auto.
  clears_but M2 M3. auto.
  clears_but x y. auto.

```

Qed.

clears_last clears the last hypothesis in the context. *clears_last N* clears the last *N* hypotheses in the context.

```

Tactic Notation "clears_last" :=
  match goal with H: ?T ⊢ _ => clear H end.

```

```

Ltac clears_last_base N :=
  match number_to_nat N with
  | 0 => idtac
  | S ?p => clears_last; clears_last_base p
  end.

```

```

Tactic Notation "clears_last" constr(N) :=
  clears_last_base N.

```

2.17 Tactics for Development Purposes

2.17.1 Skipping Subgoals

The *skip* tactic can be used at any time to admit the current goal. Unlike *admit*, it does not require ending the proof with *Admitted* instead of *Qed*. It thus saves the pain of renaming

`Qed` into *Admitted* and vice-versa all the time.

The implementation of *skip* relies on an axiom **False**. To obtain a safe development, it suffices to replace **False** with **True** in the statement of that axiom.

Note that it is still necessary to instantiate all the existential variables introduced by other tactics in order for `Qed` to be accepted.

To obtain a safe development, change to *skip_axiom* : **True** `Axiom skip_axiom` : **False**.

```
Ltac skip_with_axiom :=  
  exfalse; apply skip_axiom.
```

```
Tactic Notation "skip" :=  
  skip_with_axiom.
```

To use traditional *admit* instead of *skip* in the tactics defined below, uncomment the following definition, to bind *skip* to *admit*.

demo is like *admit* but it documents the fact that admit is intended

```
Tactic Notation "demo" :=  
  skip.
```

admits *H* : *T* adds an assumption named *H* of type *T* to the current context, blindly assuming that it is true. *admit* : *T* is another possible syntax. Note that *H* may be an intro pattern.

```
Tactic Notation "admits" simple_intropattern(I) ":" constr(T) :=  
  asserts I : T; [ skip | ].
```

```
Tactic Notation "admits" ":" constr(T) :=  
  let H := fresh "TEMP" in admits H : T.
```

```
Tactic Notation "admits" "~" ":" constr(T) :=  
  admits : T; auto_tilde.
```

```
Tactic Notation "admits" "*" ":" constr(T) :=  
  admits : T; auto_star.
```

admit_cuts *T* simply replaces the current goal with *T*.

```
Tactic Notation "admit_cuts" constr(T) :=  
  cuts : T; [ skip | ].
```

admit_goal *H* applies to any goal. It simply assumes the current goal to be true. The assumption is named “H”. It is useful to set up proof by induction or coinduction. Syntax *admit_goal* is also accepted.

```
Tactic Notation "admit_goal" ident(H) :=  
  match goal with  $\vdash ?G \Rightarrow$  admits H : G end.
```

```
Tactic Notation "admit_goal" :=  
  let IH := fresh "IH" in admit_goal IH.
```

admit_rewrite *T* can be applied when *T* is an equality. It blindly assumes this equality to be true, and rewrite it in the goal.

Tactic Notation "admit_rewrite" constr(*T*) :=
 let *M* := fresh "TEMP" in admits *M*: *T*; rewrite *M*; clear *M*.
admit_rewrite T in H is similar as *admit_rewrite*, except that it rewrites in hypothesis *H*.

Tactic Notation "admit_rewrite" constr(*T*) "in" *hyp*(*H*) :=
 let *M* := fresh "TEMP" in admits *M*: *T*; rewrite *M* in *H*; clear *M*.
admit_rewrites_all T is similar as *admit_rewrite*, except that it rewrites everywhere (goal and all hypotheses).

Tactic Notation "admit_rewrite_all" constr(*T*) :=
 let *M* := fresh "TEMP" in admits *M*: *T*; rewrite_all *M*; clear *M*.
forwards_nounfold_admit_sides_then E ltac:(fun K ⇒ ..) is like *forwards*: *E* but it provides the resulting term to a continuation, under the name *K*, and it admits any side-condition produced by the instantiation of *E*, using the *skip* tactic.

Inductive **ltac_goal_to_discard** := ltac_goal_to_discard_intro.

Ltac *forwards_nounfold_admit_sides_then S cont* :=
 let *MARK* := fresh "TEMP" in
 generalize ltac_goal_to_discard_intro;
 intro *MARK*;
forwards_nounfold_then S ltac:(fun *K* ⇒
 clear *MARK*;
 cont *K*);
 match goal with
 | *MARK*: **ltac_goal_to_discard** ⊢ _ ⇒ *skip*
 | _ ⇒ idtac
 end.

2.18 Compatibility with standard library

The module *Program* contains definitions that conflict with the current module. If you import *Program*, either directly or indirectly (e.g. through *Setoid* or *ZArith*), you will need to import the compability definitions through the top-level command: **Import** **LIBTACTICSCOMPATIBILITY**.

Module **LIBTACTICSCOMPATIBILITY**.
 Tactic Notation "apply" "*" constr(*H*) :=
 sapply *H*; auto_star.
 Tactic Notation "subst" "*" :=
 subst; auto_star.
End **LIBTACTICSCOMPATIBILITY**.
Open Scope *nat_scope*.

2.19 Experimental features

2.19.1 'let Bindings (EXPERIMENTAL).

The syntax `'let x := v in e` has the same meaning as `let x := v in e` except that the binding is implemented using a beta-redex that is not reduced automatically by `simpl`. The `'let` construct therefore makes it possible to simplify or push to the context let-bindings one by one.

Definition of `'let`

Declare Scope *let_scope*.

Definition `let_binding (A B:Type) (v:A) (K:A→B) := K v`.

Notation "'let' x ':'= v 'in' e" := (`let_binding v (fun x => e)`)
(at level 69, *x name*, right associativity,
format "[v' ':' 'let' x ':'= v 'in' ']' '/' '[' e ']' ']'")
: *let_scope*.

Notation "'let' x ':' A ':'= v 'in' e" := (`let_binding (v:A) (fun x:A => e)`)
(at level 69, *x name*, right associativity,
format "[v' ':' 'let' x ':' A ':'= v 'in' ']' '/' '[' e ']' ']'")
: *let_scope*.

Global Open Scope *let_scope*.

Lemma `let_binding_unfold : ∀ (A B:Type) (v:A) (K:A→B),
let_binding v K = K v`.

Proof using. `reflexivity`. Qed.

Ltac *let_get_fresh_binding_name K* :=
match *K* with (fun *x* => _) => let *y* := fresh *x* in *y* end.

let_simpl finds the first occurrence of a `'let` binding and substitutes it.

Tactic Notation "let_simpl" "in" *hyp(H)* :=
match type of *H* with context [`let_binding ?v ?K`] =>
changes (`let_binding v K`) with (*K v*) in *H*
end.

Tactic Notation "let_simpl" :=
match goal with
| ⊢ context [`let_binding ?v ?K`] =>
changes (`let_binding v K`) with (*K v*)
| *H*: context [`let_binding ?v ?K`] ⊢ _ =>
let_simpl in *H*
end.

Tactic Notation "let_simpl" *constr(v)* "in" *hyp(H)* :=
repeat match type of *H* with context [`let_binding v ?K`] =>

changes (let_binding *v K*) with (*K v*) in *H*
end.

Tactic Notation "let_simpl" constr(*v*) :=
repeat match goal with
| $\vdash$ context [let_binding *v ?K*] $\Rightarrow$
 changes (let_binding *v K*) with (*K v*)
| *H*: context [let_binding *v ?K*] $\vdash$ _ $\Rightarrow$
 let_simpl v in *H*
end.

let_name finds the first occurrence of a 'let binding and moves this binding to the proof context.

Tactic Notation "let_name" "in" *hyp(H)* :=
match type of *H* with context [let_binding ?*v ?K*] $\Rightarrow$
 let *x* := *let_get_fresh_binding_name K* in
 set_eq x: v in *H*;
 let_simpl in *H*
end.

Tactic Notation "let_name" "in" *hyp(H)* "as" *ident(x)* :=
match type of *H* with context [let_binding ?*v ?K*] $\Rightarrow$
 set_eq x: v in *H*;
 let_simpl in *H*
end.

Tactic Notation "let_name" :=
match goal with
| $\vdash$ context [let_binding ?*v ?K*] $\Rightarrow$
 let *x* := *let_get_fresh_binding_name K* in
 set_eq x: v;
 let_simpl
| *H*: context [let_binding ?*v ?K*] $\vdash$ _ $\Rightarrow$
 let_name in *H*
end.

Tactic Notation "let_name" "as" *ident(x)* :=
match goal with
| $\vdash$ context [let_binding ?*v ?K*] $\Rightarrow$
 set_eq x: v;
 let_simpl
| *H*: context [let_binding ?*v ?K*] $\vdash$ _ $\Rightarrow$
 let_name in *H* as *x*
end.

let_name_all finds the first occurrence of a 'let binding, moves this binding to the proof context, and further simplify all the other 'let bindings that are binding the same value.

(See LibFixDemos for a practical motivation.)

```
Tactic Notation "let_name_all" "in" hyp(H) :=
  match type of H with context [ let_binding ?v ?K ] =>
    let x := let_get_fresh_binding_name K in
    set_eq x: v in H;
    let_simpl x in H
end.
```

```
Tactic Notation "let_name_all" "in" hyp(H) "as" ident(x) :=
  match type of H with context [ let_binding ?v ?K ] =>
    set_eq x: v in H;
    let_simpl x in H
end.
```

```
Tactic Notation "let_name_all" :=
  match goal with
  | ⊢ context [ let_binding ?v ?K ] =>
    let x := let_get_fresh_binding_name K in
    set_eq x: v;
    let_simpl x
  | H: context [ let_binding ?v ?K ] ⊢ _ =>
    let_name_all in H
end.
```

```
Tactic Notation "let_name_all" "as" ident(x) :=
  match goal with
  | ⊢ context [ let_binding ?v ?K ] =>
    set_eq x: v;
    let_simpl x
  | H: context [ let_binding ?v ?K ] ⊢ _ =>
    let_name_all in H as x
end.
```

Generalize all propositions into the goal. Naive implementation:

Ltac generalize_all_prop := repeat match goal with H: ?T |- _ => match type of T with Prop => generalizes H end end.

The real implementation is careful to not generalize **ltac_Mark**, even though it is of type **Prop**. TODO: investigate whether it would be sufficient to put **ltac_Mark** in **Type** to obtain the desired behavior.

```
Ltac generalize_all_prop :=
  repeat match goal with H: ?T ⊢ _ =>
    try match T with ltac_Mark => fail 2 end;
    match type of T with Prop =>
      generalizes H
    end end.
```


Work around for inefficiency bug of `autorewrite in *`. Usage, e.g.: `Tactic Notation "rew_list" "in" "*" := autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_list).`

```
Ltac autorewrite_in_star_patch cont :=  
  generalize ltac_mark;  
  generalize_all_prop;  
  cont tt;  
  intro_until_mark.
```

Chapter 3

Library SLF.LibEqual

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibAxioms.
Generalizable Variables A.
```

3.1 Definition of equality

3.1.1 Definition of Leibnitz' equality

```
Arguments eq {A}.
```

3.1.2 Partial application of Leibnitz' equality

$= x$ is a unary predicate which holds of values equal to x . It simply denotes the partial application of equality. $= x :> A$ allows to specify the type.

```
Notation "'=' x :> A" := (fun y => y = x :> A)
  (at level 71, x at next level).
```

```
Notation "'=' x" := (fun y => y = x)
  (at level 71).
```

$\neq x$ is a unary predicate which holds of values disequal to x . It simply denotes the partial application of disequality. $\neq x :> A$ allows to specify the type.

```
Notation "'<>' x :> A" := (fun y => y ≠ x :> A)
  (at level 71, x at next level).
```

```
Notation "'<>' x" := (fun y => y ≠ x)
  (at level 71).
```

3.1.3 Typeclass to exploit extensionality

The property `EXTENSIONALITY A` captures the fact that the type `A` features an extensional equality, in the sense that to prove the equality between two values of type `A` it suffices to prove that those two values are related by some binary relation.

```
Class Extensionality (A:Type) := Extensionality_make {  
  extensionality_hyp : A → A → Prop;  
  extensionality : ∀ (x y : A), extensionality_hyp x y → x = y }.
```

Arguments extensionality [A].

Arguments Extensionality_make [A] [extensionality_hyp].

Instance for propositional extensionality

Global Instance extensionality_prop : Extensionality Prop.

Proof using. intros. apply (Extensionality_make prop_ext). Defined.

3.1.4 Tactic to exploit extensionality

```
Ltac extends_reveal_eq tt :=  
  match goal with  
  | ⊢ _ = _ ⇒ idtac  
  | _ ⇒ first [ intro; extends_reveal_eq tt  
               | fail 2 "extends needs hnf to reveal an equality" ]  
  end.
```

```
Ltac extends_core tt :=  
  extends_reveal_eq tt;  
  applies extensionality;  
  simpl extensionality_hyp.
```

```
Tactic Notation "extends" :=  
  extends_core tt.
```

```
Tactic Notation "extends" "~" :=  
  extends; auto_tilde.
```

```
Tactic Notation "extends" "*" :=  
  extends; auto_star.
```

3.2 Properties of equality

This section contains a reformulation of the lemmas provided by the standard library concerning equality.

3.2.1 Equality as an equivalence relation

See also sectin `Eq` from `LibRelation` for reformulation of theses results using high-level definitions.

Section `EqualityProp`.

Variables $(A : \text{Type})$.

Implicit Types $x\ y\ z : A$.

Reflexivity is captured by the constructor `eq_refl`.

Symmetry

Lemma `eq_sym` : $\forall\ x\ y,$

$x = y \rightarrow$

$y = x$.

Proof using. `introv H. destruct  $\neg$  H. Qed.`

Transitivity

Lemma `eq_trans_ll` : $\forall\ y\ x\ z,$

$x = y \rightarrow$

$y = z \rightarrow$

$x = z$.

Proof using. `introv H1 H2. destruct  $\neg$  H2. Qed.`

Definition `eq_trans` := `eq_trans_ll`.

Lemma `eq_trans_lr` : $\forall\ y\ x\ z,$

$x = y \rightarrow$

$z = y \rightarrow$

$x = z$.

Proof using. `introv H1 H2. destruct  $\neg$  H2. Qed.`

Lemma `eq_trans_rl` : $\forall\ y\ x\ z,$

$y = x \rightarrow$

$y = z \rightarrow$

$x = z$.

Proof using. `introv H1 H2. destruct  $\neg$  H2. Qed.`

Lemma `eq_trans_rr` : $\forall\ y\ x\ z,$

$y = x \rightarrow$

$z = y \rightarrow$

$x = z$.

Proof using. `introv H1 H2. destruct  $\neg$  H2. Qed.`

End `EqualityProp`.

Arguments `eq_trans_ll` $[A]$.

Arguments `eq_trans_lr` $[A]$.

Arguments `eq_trans_rl` $[A]$.

Arguments `eq_trans_rr` $[A]$.

Arguments eq_trans [A].

3.2.2 Properties of disequality

Section DisequalityProp.

Variables (A : Type).

Implicit Types x y : A.

Symmetry

Lemma neq_sym : $\forall x y,$

$x \neq y \rightarrow$

$y \neq x.$

Proof using. *intros H K. destruct $\neg K$. Qed.*

End DisequalityProp.

3.2.3 Symmetrized induction principles

Section EqInductionSym.

Variables (A : Type) (x : A).

Definition eq_ind_r : $\forall (P:A \rightarrow \text{Prop}),$

$P x \rightarrow \forall y, y = x \rightarrow P y.$

Proof using. intros. subst $\times$. Qed.

Definition eq_rec_r : $\forall (P:A \rightarrow \text{Set}),$

$P x \rightarrow \forall y, y = x \rightarrow P y.$

Proof using. intros. subst $\times$. Qed.

Definition eq_rect_r : $\forall (P:A \rightarrow \text{Type}),$

$P x \rightarrow \forall y, y = x \rightarrow P y.$

Proof using. intros. subst $\times$. Qed.

End EqInductionSym.

3.3 Functional extensionality

3.3.1 Dependent functional extensionality

Section FuncExtDep.

Variables (A1 : Type).

Variables (A2 : $\forall (x1 : A1), \text{Type}$).

Variables (A3 : $\forall (x1 : A1) (x2 : A2 x1), \text{Type}$).

Variables (A4 : $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), \text{Type}$).

Variables (A5 : $\forall (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), \text{Type}$).

Variables ($A6 : \forall (x1 : A1) (x2 : A2\ x1) (x3 : A3\ x2) (x4 : A4\ x3) (x5 : A5\ x4), \text{Type}$).
 Lemma fun_ext_1 : $\forall (f\ g : \forall (x1:A1), A2\ x1),$
 $(\forall x1, f\ x1 = g\ x1) \rightarrow$
 $f = g.$
 Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.
 Lemma fun_ext_2 : $\forall (f\ g : \forall (x1:A1) (x2:A2\ x1), A3\ x2),$
 $(\forall x1\ x2, f\ x1\ x2 = g\ x1\ x2) \rightarrow$
 $f = g.$
 Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.
 Lemma fun_ext_3 : $\forall (f\ g : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2), A4\ x3),$
 $(\forall x1\ x2\ x3, f\ x1\ x2\ x3 = g\ x1\ x2\ x3) \rightarrow$
 $f = g.$
 Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.
 Lemma fun_ext_4 : $\forall (f\ g : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3), A5\ x4),$
 $(\forall x1\ x2\ x3\ x4, f\ x1\ x2\ x3\ x4 = g\ x1\ x2\ x3\ x4) \rightarrow$
 $f = g.$
 Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.
 Lemma fun_ext_5 : $\forall (f\ g : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4), A6\ x5),$
 $(\forall x1\ x2\ x3\ x4\ x5, f\ x1\ x2\ x3\ x4\ x5 = g\ x1\ x2\ x3\ x4\ x5) \rightarrow$
 $f = g.$
 Proof using. repeat (intros; apply fun_ext_dep). auto. Qed.
 Global Instance Extensionality_fun_1 :
 Extensionality $(\forall (x1:A1), A2\ x1).$
 Proof using. intros. apply (Extensionality_make fun_ext_1). Defined.
 Global Instance Extensionality_fun_2 :
 Extensionality $(\forall (x1:A1) (x2:A2\ x1), A3\ x2).$
 Proof using. intros. apply (Extensionality_make fun_ext_2). Defined.
 Global Instance Extensionality_fun_3 :
 Extensionality $(\forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2), A4\ x3).$
 Proof using. intros. apply (Extensionality_make fun_ext_3). Defined.
 Global Instance Extensionality_fun_4 :
 Extensionality $(\forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3), A5\ x4).$
 Proof using. intros. apply (Extensionality_make fun_ext_4). Defined.
 Global Instance Extensionality_fun_5 :
 Extensionality $(\forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4), A6\ x5).$
 Proof using. intros. apply (Extensionality_make fun_ext_5). Defined.
 Lemma fun_eta_dep_1 : $\forall (f : \forall (x1:A1), A2\ x1),$

```

    (fun x1 => f x1) = f.
Proof using. intros. apply fun_ext_1. Qed.
Lemma fun_eta_dep_2 : ∀ (f : ∀ (x1:A1) (x2:A2 x1), A3 x2),
    (fun x1 x2 => f x1 x2) = f.
Proof using. intros. apply fun_ext_2. Qed.
Lemma fun_eta_dep_3 : ∀ (f : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2), A4 x3),
    (fun x1 x2 x3 => f x1 x2 x3) = f.
Proof using. intros. apply fun_ext_3. Qed.
Lemma fun_eta_dep_4 : ∀ (f : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2) (x4:A4 x3), A5 x4),
    (fun x1 x2 x3 x4 => f x1 x2 x3 x4) = f.
Proof using. intros. apply fun_ext_4. Qed.
Lemma fun_eta_dep_5 : ∀ (f : ∀ (x1:A1) (x2:A2 x1) (x3:A3 x2) (x4:A4 x3) (x5:A5 x4),
    A6 x5),
    (fun x1 x2 x3 x4 x5 => f x1 x2 x3 x4 x5) = f.
Proof using. intros. apply fun_ext_5. Qed.
End FuncExtDep.

```

3.3.2 Non-dependent functional extensionality

```

Lemma fun_ext_nondep_1 : ∀ A1 B (f g : A1 → B),
    (∀ x1, f x1 = g x1) →
    f = g.
Proof using. intros. apply fun_ext_1. Qed.
Lemma fun_ext_nondep_2 : ∀ A1 A2 B (f g : A1 → A2 → B),
    (∀ x1 x2, f x1 x2 = g x1 x2) →
    f = g.
Proof using. intros. apply fun_ext_2. Qed.
Lemma fun_ext_nondep_3 : ∀ A1 A2 A3 B (f g : A1 → A2 → A3 → B),
    (∀ x1 x2 x3, f x1 x2 x3 = g x1 x2 x3) →
    f = g.
Proof using. intros. apply fun_ext_3. Qed.
Lemma fun_ext_nondep_4 : ∀ A1 A2 A3 A4 B (f g : A1 → A2 → A3 → A4 → B),
    (∀ x1 x2 x3 x4, f x1 x2 x3 x4 = g x1 x2 x3 x4) →
    f = g.
Proof using. intros. apply fun_ext_4. Qed.
Lemma fun_ext_nondep_5 : ∀ A1 A2 A3 A4 A5 B (f g : A1 → A2 → A3 → A4 → A5
    → B),
    (∀ x1 x2 x3 x4 x5, f x1 x2 x3 x4 x5 = g x1 x2 x3 x4 x5) →
    f = g.
Proof using. intros. apply fun_ext_5. Qed.

```

3.3.3 Eta-conversion

```

Lemma fun_eta_1 : ∀ A1 B (f : A1 → B),
  (fun x1 ⇒ f x1) = f.
Proof using. intros. apply fun_ext_1. Qed.

Lemma fun_eta_2 : ∀ A1 A2 B (f : A1 → A2 → B),
  (fun x1 x2 ⇒ f x1 x2) = f.
Proof using. intros. apply fun_ext_2. Qed.

Lemma fun_eta_3 : ∀ A1 A2 A3 B (f : A1 → A2 → A3 → B),
  (fun x1 x2 x3 ⇒ f x1 x2 x3) = f.
Proof using. intros. apply fun_ext_3. Qed.

Lemma fun_eta_4 : ∀ A1 A2 A3 A4 B (f : A1 → A2 → A3 → A4 → B),
  (fun x1 x2 x3 x4 ⇒ f x1 x2 x3 x4) = f.
Proof using. intros. apply fun_ext_4. Qed.

Lemma fun_eta_5 : ∀ A1 A2 A3 A4 A5 B (f : A1 → A2 → A3 → A4 → A5 → B),
  (fun x1 x2 x3 x4 x5 ⇒ f x1 x2 x3 x4 x5) = f.
Proof using. intros. apply fun_ext_4. Qed.

#[global]
Hint Rewrite fun_eta_1 fun_eta_2 fun_eta_3 fun_eta_4 fun_eta_5 : rew_eta.

```

3.4 Predicate extensionality

3.4.1 Dependend predicates

```

Section PropExt.
Variables (A1 : Type).
Variables (A2 : ∀ (x1 : A1), Type).
Variables (A3 : ∀ (x1 : A1) (x2 : A2 x1), Type).
Variables (A4 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), Type).
Variables (A5 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), Type).
Variables (A6 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), Type).

Lemma pred_ext_1 : ∀ (P Q : ∀ (x1:A1), Prop),
  (∀ x1, P x1 ↔ Q x1) →
  P = Q.
Proof using. repeat (intros; apply fun_ext_dep). intros. apply fun_ext. Qed.

Lemma pred_ext_2 : ∀ (P Q : ∀ (x1:A1) (x2:A2 x1), Prop),
  (∀ x1 x2, P x1 x2 ↔ Q x1 x2) →
  P = Q.
Proof using. repeat (intros; apply fun_ext_dep). intros. apply fun_ext. Qed.

```


Lemma pred_ext_3 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2), \text{Prop}),$
 $(\forall x1\ x2\ x3, P\ x1\ x2\ x3 \leftrightarrow Q\ x1\ x2\ x3) \rightarrow$
 $P = Q.$
 Proof using. repeat (intros; apply fun_ext_dep). intros. apply $\neg$ prop_ext. Qed.

Lemma pred_ext_4 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3), \text{Prop}),$
 $(\forall x1\ x2\ x3\ x4, P\ x1\ x2\ x3\ x4 \leftrightarrow Q\ x1\ x2\ x3\ x4) \rightarrow$
 $P = Q.$
 Proof using. repeat (intros; apply fun_ext_dep). intros. apply $\neg$ prop_ext. Qed.

Lemma pred_ext_5 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4), \text{Prop}),$
 $(\forall x1\ x2\ x3\ x4\ x5, P\ x1\ x2\ x3\ x4\ x5 \leftrightarrow Q\ x1\ x2\ x3\ x4\ x5) \rightarrow$
 $P = Q.$
 Proof using. repeat (intros; apply fun_ext_dep). intros. apply $\neg$ prop_ext. Qed.

Lemma pred_ext_6 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5), \text{Prop}),$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6, P\ x1\ x2\ x3\ x4\ x5\ x6 \leftrightarrow Q\ x1\ x2\ x3\ x4\ x5\ x6) \rightarrow$
 $P = Q.$
 Proof using. repeat (intros; apply fun_ext_dep). intros. apply $\neg$ prop_ext. Qed.

Global Instance Extensionality_pred_1 :
Extensionality $(\forall (x1:A1), \text{Prop}).$
 Proof using. intros. apply (Extensionality_make pred_ext_1). Defined.

Global Instance Extensionality_pred_2 :
Extensionality $(\forall (x1:A1) (x2:A2\ x1), \text{Prop}).$
 Proof using. intros. apply (Extensionality_make pred_ext_2). Defined.

Global Instance Extensionality_pred_3 :
Extensionality $(\forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2), \text{Prop}).$
 Proof using. intros. apply (Extensionality_make pred_ext_3). Defined.

Global Instance Extensionality_pred_4 :
Extensionality $(\forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2) (x4:A4\ x3), \text{Prop}).$
 Proof using. intros. apply (Extensionality_make pred_ext_4). Defined.

Global Instance Extensionality_pred_5 :
Extensionality $(\forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4), \text{Prop}).$
 Proof using. intros. apply (Extensionality_make pred_ext_5). Defined.

Global Instance Extensionality_pred_6 :
Extensionality $(\forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5), \text{Prop}).$
 Proof using. intros. apply (Extensionality_make pred_ext_6). Defined.

End PropExt.

3.4.2 Non-dependend predicate extensionality

Lemma pred_ext_nondep_1 :

$\forall A1 (P Q : A1 \rightarrow \text{Prop}),$
 $(\forall x1, P x1 \leftrightarrow Q x1) \rightarrow$
 $P = Q.$

Proof using. intros. apply $\neg$ pred_ext_1. Qed.

Lemma pred_ext_nondep_2 :

$\forall A1 A2 (P Q : A1 \rightarrow A2 \rightarrow \text{Prop}),$
 $(\forall x1 x2, P x1 x2 \leftrightarrow Q x1 x2) \rightarrow$
 $P = Q.$

Proof using. intros. apply $\neg$ pred_ext_2. Qed.

Lemma pred_ext_nondep_3 :

$\forall A1 A2 A3 (P Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow \text{Prop}),$
 $(\forall x1 x2 x3, P x1 x2 x3 \leftrightarrow Q x1 x2 x3) \rightarrow$
 $P = Q.$

Proof using. intros. apply $\neg$ pred_ext_3. Qed.

Lemma pred_ext_nondep_4 :

$\forall A1 A2 A3 A4 (P Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow \text{Prop}),$
 $(\forall x1 x2 x3 x4, P x1 x2 x3 x4 \leftrightarrow Q x1 x2 x3 x4) \rightarrow$
 $P = Q.$

Proof using. intros. apply $\neg$ pred_ext_4. Qed.

Lemma pred_ext_nondep_5 :

$\forall A1 A2 A3 A4 A5 (P Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow \text{Prop}),$
 $(\forall x1 x2 x3 x4 x5, P x1 x2 x3 x4 x5 \leftrightarrow Q x1 x2 x3 x4 x5) \rightarrow$
 $P = Q.$

Proof using. intros. apply $\neg$ pred_ext_5. Qed.

Lemma pred_ext_nondep_6 :

$\forall A1 A2 A3 A4 A5 A6 (P Q : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow A6 \rightarrow \text{Prop}),$
 $(\forall x1 x2 x3 x4 x5 x6, P x1 x2 x3 x4 x5 x6 \leftrightarrow Q x1 x2 x3 x4 x5 x6) \rightarrow$
 $P = Q.$

Proof using. intros. apply $\neg$ pred_ext_6. Qed.

3.5 Equality of function and predicate applications

3.5.1 A same function applied to equal arguments yield equal result

Section ArgsEq.

Variables $(A1 A2 A3 A4 A5 B : \text{Type}).$

Lemma args_eq_1 : $\forall (f:A1 \rightarrow B) \ x1 \ y1,$

$x1 = y1 \rightarrow$
 $f \ x1 = f \ y1.$

Proof using. intros. subst $\neg$. Qed.

Lemma args_eq_2 : $\forall (f:A1 \rightarrow A2 \rightarrow B) \ x1 \ y1 \ x2 \ y2,$

$x1 = y1 \rightarrow$
 $x2 = y2 \rightarrow$
 $f \ x1 \ x2 = f \ y1 \ y2.$

Proof using. intros. subst $\neg$. Qed.

Lemma args_eq_3 : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow B) \ x1 \ y1 \ x2 \ y2 \ x3 \ y3,$

$x1 = y1 \rightarrow$
 $x2 = y2 \rightarrow$
 $x3 = y3 \rightarrow$
 $f \ x1 \ x2 \ x3 = f \ y1 \ y2 \ y3.$

Proof using. intros. subst $\neg$. Qed.

Lemma args_eq_4 : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow B) \ x1 \ y1 \ x2 \ y2 \ x3 \ y3 \ x4 \ y4,$

$x1 = y1 \rightarrow$
 $x2 = y2 \rightarrow$
 $x3 = y3 \rightarrow$
 $x4 = y4 \rightarrow$
 $f \ x1 \ x2 \ x3 \ x4 = f \ y1 \ y2 \ y3 \ y4.$

Proof using. intros. subst $\neg$. Qed.

Lemma args_eq_5 : $\forall (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow B) \ x1 \ y1 \ x2 \ y2 \ x3 \ y3 \ x4 \ y4 \ x5 \ y5,$

$x1 = y1 \rightarrow$
 $x2 = y2 \rightarrow$
 $x3 = y3 \rightarrow$
 $x4 = y4 \rightarrow$
 $x5 = y5 \rightarrow$
 $f \ x1 \ x2 \ x3 \ x4 \ x5 = f \ y1 \ y2 \ y3 \ y4 \ y5.$

Proof using. intros. subst $\neg$. Qed.

End ArgsEq.

3.5.2 Equal functions applied to same arguments return equal results

These results are exploited by tactic *fequals* (see LibTactics); however the lemmas remain useful for forward-reasoning.

Section FuncEq.

Variables (*A1 A2 A3 A4 A5 B*:Type).

Variables (*x1:A1*) (*x2:A2*) (*x3:A3*) (*x4:A4*) (*x5:A5*).

```

Lemma fun_eq_1 :  $\forall f\ g,$ 
   $f = g \rightarrow$ 
   $f\ x1 = g\ x1 :> B.$ 
Proof using. intros. subst $\neg$ . Qed.

Lemma fun_eq_2 :  $\forall f\ g,$ 
   $f = g \rightarrow$ 
   $f\ x1\ x2 = g\ x1\ x2 :> B.$ 
Proof using. intros. subst $\neg$ . Qed.

Lemma fun_eq_3 :  $\forall f\ g,$ 
   $f = g \rightarrow$ 
   $f\ x1\ x2\ x3 = g\ x1\ x2\ x3 :> B.$ 
Proof using. intros. subst $\neg$ . Qed.

Lemma fun_eq_4 :  $\forall f\ g,$ 
   $f = g \rightarrow$ 
   $f\ x1\ x2\ x3\ x4 = g\ x1\ x2\ x3\ x4 :> B.$ 
Proof using. intros. subst $\neg$ . Qed.

Lemma fun_eq_5 :  $\forall f\ g,$ 
   $f = g \rightarrow$ 
   $f\ x1\ x2\ x3\ x4\ x5 = g\ x1\ x2\ x3\ x4\ x5 :> B.$ 
Proof using. intros. subst $\neg$ . Qed.

End FuncEq.

```

3.5.3 Equal predicates applied to same arguments return equivalent results

These results are exploited by tactic *fequals* (see LibTactics); however the lemmas remain useful for forward-reasoning.

```

Section PredEq.
Variables ( $A1\ A2\ A3\ A4\ A5\ B$ :Type).
Variables ( $x1$ : $A1$ ) ( $x2$ : $A2$ ) ( $x3$ : $A3$ ) ( $x4$ : $A4$ ) ( $x5$ : $A5$ ).

Lemma pred_eq_1 :  $\forall P\ Q,$ 
   $P = Q \rightarrow$ 
   $P\ x1 \leftrightarrow Q\ x1.$ 
Proof using. intros. subst $\times$ . Qed.

Lemma pred_eq_2 :  $\forall P\ Q,$ 
   $P = Q \rightarrow$ 
   $P\ x1\ x2 \leftrightarrow Q\ x1\ x2.$ 
Proof using. intros. subst $\times$ . Qed.

Lemma pred_eq_3 :  $\forall P\ Q,$ 
   $P = Q \rightarrow$ 

```

```

     $P\ x1\ x2\ x3 \leftrightarrow Q\ x1\ x2\ x3.$ 
Proof using. intros. subst×. Qed.

Lemma pred_eq_4 :  $\forall\ P\ Q,$ 
     $P = Q \rightarrow$ 
     $P\ x1\ x2\ x3\ x4 \leftrightarrow Q\ x1\ x2\ x3\ x4.$ 
Proof using. intros. subst×. Qed.

Lemma pred_eq_5 :  $\forall\ P\ Q,$ 
     $P = Q \rightarrow$ 
     $P\ x1\ x2\ x3\ x4\ x5 \leftrightarrow Q\ x1\ x2\ x3\ x4\ x5.$ 
Proof using. intros. subst×. Qed.

End PredEq.

```

3.6 Proof Irrelevance

The proof irrelevance lemma states that two proofs of a same proposition are always equal.
 $\forall\ (P : \text{Prop})\ (p\ q : P),\ p = q$ This result is a consequence of propositional extensionality.

3.6.1 Proof of the proof-irrelevance result

```

Module PIFROMEXT.

Implicit Types  $P : \text{Prop}.$ 

```

First, we prove that on an inhabited proposition type, there exists a fixpoint combinator.

```

Lemma prop_eq_self_impl_when_true :  $\forall\ P,$ 
     $P \rightarrow$ 
     $P = (P \rightarrow P).$ 
Proof using. intros. apply× prop_ext. Qed.

Record has_fixpoint ( $P : \text{Prop}$ ) :  $\text{Prop} := \text{has\_fixpoint\_make}$ 
{ has_fixpoint_F :  $(P \rightarrow P) \rightarrow P;$ 
  has_fixpoint_fix :  $\forall\ f,\ \text{has\_fixpoint\_F}\ f = f\ (\text{has\_fixpoint\_F}\ f)\ }.$ 

Lemma prop_has_fixpoint_when_true :  $\forall\ P,$ 
     $P \rightarrow$ 
    has_fixpoint  $P.$ 
Proof using.
  intros  $P\ a.$  set ( $P' := P$ ).
  set ( $g1 := \text{id} : P' \rightarrow P$ ). set ( $g2 := \text{id} : P \rightarrow P'$ ).
  asserts¬ Fix: ( $\forall\ x,\ g1\ (g2\ x) = x$ ).
  clearbody  $g1\ g2.$  gen  $g1\ g2.$ 
  rewrite (prop_eq_self_impl_when_true  $a$ ).
  subst  $P'.$  intros.

```

```

set (Y := fun f => (fun x => f (g1 x x)) (g2 (fun x => f (g1 x x)))).
applys (has_fixpoint_make Y). { intros f. unfold Y at 1. rewrite¬ Fix. }
Qed.

```

We exploit the fixpoint combinator on the negation function, applied to the following special proposition type (isomorphic to booleans, but living in Prop).

```

Inductive boolP : Prop :=
| trueP : boolP
| falseP : boolP.

```

Lemma trueP_eq_falseP : trueP = falseP.

Proof using.

```

lets (Y&Yfix): (@prop_has_fixpoint_when_true boolP trueP).
set (neg := fun b => match b with | trueP => falseP | falseP => trueP end).
lets F: ((rm Yfix) neg).
set (b := Y neg).
asserts¬ E: (b = Y neg).
destruct b.
{ change (trueP = neg trueP) in ⊢ ×. rewrite E. rewrite¬ ← F. }
{ change (neg falseP = falseP) in ⊢ ×. rewrite E. rewrite¬ ← F. }

```

Qed.

We now have two distinct constructors `trueP` and `falseP`, which we can distinguish in the logic using `match` for any goal concluding on a proposition; and, at the same time, these two constructors are provably equal. We can exploit these properties to prove that two proofs of a same theorem are equal.

Lemma proof_irrelevance :

```

∀ (P : Prop) (p q : P), p = q.

```

Proof using.

```

intros P p q.
set (f := fun b => match b with | trueP => p | falseP => q end).
change p with (f trueP).
change q with (f falseP).
rewrite¬ trueP_eq_falseP.

```

Qed.

End PIFROMEXT.

Lemma proof_irrelevance : ∀ (P : Prop) (p q : P), p = q.

Proof using. exact PIfromExt.proof_irrelevance. Qed.

3.6.2 Consequences of proof irrelevance

Uniqueness of identity proofs

Lemma identity_proofs_unique :

$\forall (A : \text{Type}) (x\ y : A) (p\ q : x = y),$
 $p = q.$

Proof using. intros. apply proof_irrelevance. Qed.

Uniqueness of reflexive identity proofs (special case)

Lemma reflexive_identity_proofs_unique :

$\forall (A : \text{Type}) (x : A) (p : x = x),$
 $p = \text{refl\_equal } x.$

Proof using. intros. *applies* identity_proofs_unique. Qed.

Invariance by substitution of reflexive equality proofs

Lemma eq_rect_refl_eq :

$\forall (A : \text{Type}) (p : A) (Q : A \rightarrow \text{Type}) (x : Q\ p) (h : p = p),$
 $\text{eq\_rect } p\ Q\ x\ p\ h = x.$

Proof using. intros. rewrite $\neg$ (reflexive_identity_proofs_unique h). Qed.

Streicher's axiom K

Lemma streicher_K :

$\forall (A : \text{Type}) (x : A) (P : x = x \rightarrow \text{Prop}),$
 $P (\text{refl\_equal } x) \rightarrow$
 $\forall (p : x = x), P\ p.$

Proof using. intros. rewrite $\neg$ (reflexive_identity_proofs_unique p). Qed.

Coercion for rewriting a type equality in a term

Definition rewrite_type (A B:Type) (E:A=B) (V:A) : B :=
 $\text{eq\_rect } A (\text{fun } T \Rightarrow T) V\ B\ E.$

Lemma rewrite_type_self : $\forall A (E:A=A) (V:A),$
 $\text{rewrite\_type } E\ V = V.$

Proof using. intros. unfold rewrite_type. rewrite $\times$ eq_rect_refl_eq. Qed.

3.6.3 Injectivity of equality on dependent pairs

This section establishes that $\text{existT } P\ p\ x = \text{existT } P\ p\ y$ implies that x is equal to y . It indirectly results from the proof irrelevance property.

Definition of dependent equality, with non-dependent return type

Inductive **eq_dep_nd** (A : Type) (P : A $\rightarrow$ Type)
 $(p : A) (x : P\ p) (q : A) (y : P\ q) : \text{Prop} :=$
 $| \text{eq\_dep\_nd\_intro} : \forall (h : q = p),$
 $x = \text{eq\_rect } q\ P\ y\ p\ h \rightarrow \text{eq\_dep\_nd } P\ p\ x\ q\ y.$

Arguments **eq_dep_nd** [A] [P] [p] x [q] y.

Arguments eq_dep_nd_intro [A] [P] [p] [x] [q] [y].

Reflexivity of **eq_dep_nd**

Lemma eq_dep_nd_refl : $\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x : P\ p),$

eq_dep_nd $x\ x$.

Proof using. `intros. apply (eq_dep_nd_intro (refl_equal p)). auto. Qed.`

Injectivity of **eq_dep_nd**

Lemma `eq_dep_nd_same_inv` :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x\ y : P\ p),$
eq_dep_nd $x\ y \rightarrow$
 $x = y.$

Proof using. `intros H. inversions H. rewrite  $\neg$  eq_rect_refl_eq. Qed.`

Equality on dependent pairs implies **eq_dep_nd**

Lemma `eq_existT_inv` :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p\ q : A) (x : P\ p) (y : P\ q),$
 $\text{existT } P\ p\ x = \text{existT } P\ q\ y \rightarrow$
eq_dep_nd $x\ y.$

Proof using. `intros E. dependent rewrite E. simpl. apply eq_dep_nd_refl. Qed.`

Injectivity of equality on dependent pairs

Lemma `eq_existT_same_inv` :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x\ y : P\ p),$
 $\text{existT } P\ p\ x = \text{existT } P\ p\ y \rightarrow$
 $x = y.$

Proof using. `intros. apply eq_dep_nd_same_inv. apply  $\neg$  eq_existT_inv. Qed.`

Reformulated as an equality

Lemma `eq_existT_same_eq` :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Type}) (p : A) (x\ y : P\ p),$
 $(\text{existT } P\ p\ x = \text{existT } P\ p\ y) = (x = y).$

Proof using.

`extens. iff M.`

`{ apply eq_dep_nd_same_inv. apply  $\neg$  eq_existT_inv. }`
`{ subst  $\times$ . }`

`Qed.`

This is another consequence of proof irrelevance

Scheme `eq_indd` := Induction for **eq** Sort Prop.

Lemma `exist_eq_exist` : $\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}) (x\ y : A) (p : P\ x) (q : P\ y),$
 $x = y \rightarrow$
 $\text{exist } P\ x\ p = \text{exist } P\ y\ q.$

Proof using.

`intros. rewrite (proof_irrelevance q (eq_rect x P p y H)). subst  $\times$ .`

`Qed.`

Lemma `existT_eq_existT` : $\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}) (x\ y : A) (p : P\ x) (q : P\ y),$
 $x = y \rightarrow$


```

    existT P x p = existT P y q.
Proof using.
  intros. rewrite (proof_irrelevance q (eq_rect x P p y H)). subst×.
Qed.
Ltac fequal_support_for_exist tt ::=
  first [ apply exist_eq_exist | apply existT_eq_existT ].

```

3.7 Dependent equality

In this section, we prove that **eq_dep** x y implies $x = y$.

Definition of **eq_dep** (copied from the Prelude)

```

Inductive eq_dep (A : Type) (P : A → Type) (p : A) (x : P p)
  : ∀ q, P q → Prop :=
| eq_dep_refl : eq_dep P p x p x.

```

Arguments **eq_dep** [A] [P] [p] x [q].

Symmetry of **eq_dep**

```

Lemma eq_dep_sym : ∀ (A : Type) (P : A → Type)
  (p q : A) (x : P p) (y : P q),
  eq_dep x y →
  eq_dep y x.

```

Proof using. *intros E. destruct E. constructor. Qed.*

Transitivity of **eq_dep**

```

Lemma eq_dep_trans : ∀ (A : Type) (P : A → Type)
  (p q r : A) (y : P q) (x : P p) (z : P r),
  eq_dep x y →
  eq_dep y z →
  eq_dep x z.

```

Proof using. *intros E F. destruct¬ E. Qed.*

Proof of equivalence between **eq_dep_nd** and **eq_dep**

Scheme eq_induction := Induction for **eq** Sort Prop.

```

Lemma eq_dep_of_eq_dep_nd :
  ∀ (A : Type) (P : A → Type) (p q : A) (x : P p) (y : P q),
  eq_dep_nd x y →
  eq_dep x y.

```

Proof using.

intros E. destruct E as (h,H).

destruct h using eq_induction. subst¬. constructor.

Qed.

Lemma eq_dep_nd_of_eq_dep :

```

  ∀ (A : Type) (P : A → Type) (p q : A) (x : P p) (y : P q),
  eq_dep x y →
  eq_dep_nd x y.
Proof using. introv H. destruct H. apply (eq_dep_nd_intro (refl_equal p)); auto. Qed.

```

Injectivity of dependent equality

```

Lemma eq_dep_same_inv :
  ∀ (A : Type) (P : A → Type) (p : A) (x y : P p),
  eq_dep x y →
  x = y.
Proof using.
  introv R. inversion R. apply eq_dep_nd_same_inv. apply¬ eq_dep_nd_of_eq_dep.
Qed.

```

Equality on dependent pairs implies dependent equality

```

Lemma eq_dep_of_eq_existT :
  ∀ (A : Type) (P : A → Type) (p q : A) (x : P p) (y : P q),
  existT P p x = existT P q y →
  eq_dep x y.
Proof using. introv E. dependent rewrite E. simple¬. constructor. Qed.

```

3.8 John Major's equality

From *Stdlib* Require Import JMeq.

The module above defines John Major's equality:

```

Inductive JMeq (A : Type) (x : A) : forall B : Type, B -> Prop := | JMeq_refl : JMeq
x x.

```

In this section, we prove that $\text{JMeq } x \ y$ implies $x = y$ when x and y have the same type.

Symmetry, transitivity of JMeq

```

Lemma JMeq_sym : ∀ (A B : Type) (x : A) (y : B),
  JMeq x y →
  JMeq y x.

```

Proof using. *introv E. destruct¬ E. Qed.*

```

Lemma JMeq_trans : ∀ (A B C : Type) (y : B) (x : A) (z : C),
  JMeq x y →
  JMeq y z →
  JMeq x z.

```

Proof using. *introv E F. destruct¬ E. Qed.*

Local Hint Immediate *JMeq_sym*.

Relation between JMeq and `eq_dep`

```

Lemma eq_dep_of_JMeq : ∀ (A B : Type) (x : A) (y : B),

```

```

JMeq  $x\ y \rightarrow$ 
  @eq_dep Type (fun  $T \Rightarrow T$ )  $A\ x\ B\ y$ .
Proof using. intros  $E$ . destruct  $E$ . constructor. Qed.

Lemma JMeq_of_eq_dep :  $\forall (A\ B : \text{Type}) (x : A) (y : B)$ ,
  @eq_dep Type (fun  $T \Rightarrow T$ )  $A\ x\ B\ y \rightarrow$ 
    JMeq  $x\ y$ .
Proof using. intros  $E$ . destruct  $\neg E$ . Qed.

Injectivity of JMeq

Lemma JMeq_same_inv :  $\forall (A : \text{Type}) (x\ y : A)$ ,
  JMeq  $x\ y \rightarrow$ 
     $x = y$ .
Proof using.
  intros  $E$ . apply (@eq_dep_same_inv Type (fun  $T \Rightarrow T$ )).
  apply  $\neg$  eq_dep_of_JMeq.
Qed.

```

Chapter 4

Library SLF.LibLogic

```
Set Implicit Arguments.
From SLF Require Import LibTactics.
From SLF Require Export LibAxioms LibEqual.
Generalizable Variables A B P.
```

4.1 Strong existentials

Type **sig** is defined in LibLogicCore
Projections

```
Definition sig_val (A : Type) (P : A → Prop) (e : sig P) : A :=
  match e with exist _ a _ ⇒ a end.

Definition sig_proof (A : Type) (P : A → Prop) (e : sig P) : P (sig_val e) :=
  match e with exist _ _ b ⇒ b end.
```

4.2 Inhabited types

4.2.1 Typeclass for describing inhabited types

The proposition **Inhab** *A* captures the fact that the type *A* is inhabited (i.e., there exists at least one value of type *A*).

```
Class Inhab (A : Type) : Prop :=
  { Inhab_intro : (∃ (x : A), True) }.

#[global]
Hint Mode Inhab + : typeclass_instances.
```

4.2.2 Tactics taking into account

Extension to LibTactics' fast introduction tactic `=>` to handle specifically the case of the Inhabited typeclass.

```
Ltac intro_nondeps_aux_special_intro G ::=
  match G with
  | (Inhab _) => idtac
  end.
```

4.2.3 Arbitrary values for inhabited types

The value `arbitrary` can be used as a dummy value at any place where a value of an inhabited type is expected.

```
Definition arbitrary '{Inhab A} : A :=
  sig_val (@indefinite_description A - Inhab_intro).
  Extraction of arbitrary constants as a runtime error.
```

4.2.4 Lemmas about inhabited types

Proving a type to be inhabited

```
Lemma Inhab_of_val : ∀ (A:Type),
  A →
  Inhab A.
```

Proof using. `intros A x. constructor. ∃ x. auto. Qed.`

Arrows are inhabited if their codomain is inhabited.

```
#[global]
Instance Inhab_impl : ∀ A B {I:Inhab B},
  Inhab (A → B).
Proof using. intros. apply (Inhab_of_val (fun _ => arbitrary)). Qed.
```

4.3 Non-constructive conditionals

4.3.1 Excluded middle

Every proposition is either `True` or `False`.

```
Lemma classic : ∀ (P : Prop),
  P ∨ ¬ P.
Proof using.
  intros.
  set (B1 := fun b => b = true ∨ P).
```

```

set (B2 := fun b => b = false ∨ P).
asserts H1: (ex B1). ∃ true. left¬.
asserts H2: (ex B2). ∃ false. left¬.
sets i1: (indefinite_description H1).
sets i2: (indefinite_description H2).
destruct (sig_proof i1) as [HA]; [|auto].
destruct (sig_proof i2) as [HB]; [|auto].
right. intros HP. asserts EB: (B1 = B2).
  apply pred_ext_1. intros b. split; intros _; right; auto.
subst i1 i2. destruct EB.
rewrite (proof_irrelevance H2 H1) in HB. congruence.
Qed.

Definition prop_inv := classic.

```

4.3.2 Strong excluded middle

From classical logic and indefinite description, we can prove the strong (or informative) excluded middle, which allows Rocq definitions to make a case analysis on the truth value of any proposition.

```

Lemma classicT : ∀ (P : Prop),
  {P} + {¬ P}.
Proof using.
  intros. pose (select := fun (b:bool) => if b then P else ¬P).
  cuts (M,HP): { b:bool | select b }.
  destruct M. left¬. right¬.
  apply indefinite_description.
  destruct (prop_inv P). ∃¬ true. ∃¬ false.
Qed.

```

Simplification lemmas

```

Lemma classicT_l : ∀ (P : Prop) (H:P),
  classicT P = left _ H.
Proof using. intros. destruct (classicT P). fequals. false¬. Qed.

Lemma classicT_r : ∀ (P : Prop) (H:¬P),
  classicT P = right _ H.
Proof using. intros. destruct (classicT P). false¬. fequals. Qed.

```

4.3.3 If-then-else on propositions

The expression *If* P *then* $v1$ *else* $v2$ can be used to build a conditional that depends on a proposition P .

Notation "If' P 'then' $v1$ 'else' $v2$ " :=

(if (classicT P) then $v1$ else $v2$)
(at level 200, right associativity) : *type_scope*.

Section Ifthenelse.

Variables A : Type.

Implicit Types P : Prop.

Implicit Types $x\ y$: A .

Lemmas to simplify If-then-else statement

Lemma If_l : $\forall P\ x\ y$,
 $P \rightarrow$
 $(\text{If } P \text{ then } x \text{ else } y) = x$.
Proof using. intros. *case_if* ×. Qed.

Lemma If_r : $\forall P\ x\ y$,
 $\neg P \rightarrow$
 $(\text{If } P \text{ then } x \text{ else } y) = y$.
Proof using. intros. *case_if* ×. Qed.

A lemma to prove an equality between two If-then-else

Lemma If_eq : $\forall P\ P'\ x\ x'\ y\ y'$,
 $(P \leftrightarrow P') \rightarrow$
 $(P \rightarrow x = x') \rightarrow$
 $(\neg P \rightarrow y = y') \rightarrow$
 $(\text{If } P \text{ then } x \text{ else } y) = (\text{If } P' \text{ then } x' \text{ else } y')$.
Proof using. intros. do 2 *case_if*; *autos* ×. Qed.

A simpler version of the above lemma

Lemma If_eq_simple : $\forall P\ P'\ x\ x'\ y\ y'$,
 $(P \leftrightarrow P') \rightarrow$
 $(x = x') \rightarrow$
 $(y = y') \rightarrow$
 $(\text{If } P \text{ then } x \text{ else } y) = (\text{If } P' \text{ then } x' \text{ else } y')$.
Proof using. intros. subst. *asserts_rewrite* ($P = P'$). apply $\neg$ *prop_ext*. auto. Qed.
End Ifthenelse.

4.3.4 Consequences

Propositional extensionality stated using itself

Lemma eq_prop_eq_iff : $\forall P\ Q$,
 $(P = Q) = (P \leftrightarrow Q)$.
Proof using. intros. *extens*. *iff*. subst ×. apply $\neg$ *prop_ext*. Qed.

Propositional completeness (degeneracy)

Lemma prop_degeneracy : $\forall (P : \text{Prop})$,

$P = \text{True} \vee P = \text{False}.$

Proof using.

```
intros. destruct (prop_inv P).
  left. apply× prop_ext.
  right. apply× prop_ext.
```

Qed.

Independence of general premises

Lemma indep_general_premises :

```
∀ A ‘{Inhab A} (P : A → Prop) (Q : Prop),
  (Q → ∃ x, P x) →
  (∃ x, Q → P x).
```

Proof using.

```
introv I M. destruct (prop_inv Q).
destruct× (M H).
∃ (arbitrary (A:=A)). auto_false.
```

Qed.

Small drinker’s paradox

Lemma small_drinker_paradox : $\forall \text{ ‘}\{Inhab\ A\} (P : A \rightarrow \text{Prop}),$

```
∃ x, (∃ x, P x) → P x.
```

Proof using.

```
intros A I P. destruct (prop_inv (∃ x, P x)).
destruct H. ∃ x. auto.
∃ (arbitrary (A:=A)). auto_false.
```

Qed.

4.3.5 Tactic for proving tautologies by brute-force case-analysis

The tactic *tautop* $P1 \dots PN$ helps performing case analysis on the propositions/booleans that are given to it as parameters.

Ltac *tautop_case* $P :=$

```
  match type of P with
  | Prop ⇒ destruct (prop_degeneracy P)
  | bool ⇒ destruct P
  end.
```

Ltac *tautop_pre* $tt :=$

```
  intros;
  repeat rewrite eq_prop_eq_iff in *.
```

Ltac *tautop_post* $tt :=$

```
  subst;
  try solve [ intuition auto_false ].
```



```

Ltac tautop_core args :=
  tautop_pre tt;
  let rec aux Ps :=
    match Ps with
    | nil => idtac
    | cons (boxer ?P) ?Ps' => tautop_case P; aux Ps'
  end in
  aux args;
  tautop_post tt.

Ltac tautop_noargs tt :=
  let rec aux Ps :=
    match goal with
    | ⊢ ∀ ( _:Prop), _ =>
      let P := fresh "P" in intro P; aux (cons (boxer P) Ps)
    | _ => tautop_core Ps
  end
  in
  aux constr:(@nil Boxer).

Tactic Notation "tautop" :=
  tautop_noargs tt.

Tactic Notation "tautop" constr(P1) :=
  tautop_core (» P1).

Tactic Notation "tautop" constr(P1) constr(P2) :=
  tautop_core (» P1 P2).

Tactic Notation "tautop" constr(P1) constr(P2) constr(P3) :=
  tautop_core (» P1 P2 P3).

```

4.4 Properties of logical combinators

4.4.1 Simplification of conjunction and disjunction

Section SimplConjDisj.
 Implicit Types $P Q : \text{Prop}$.
 Lemma and_True_l_eq : $\forall P$,
 (**True** $\wedge P$) = P .
 Proof using. *tautop*. Qed.
 Lemma and_True_r_eq : $\forall P$,
 ($P \wedge$ **True**) = P .
 Proof using. *tautop*. Qed.
 Lemma and_False_l_eq : $\forall P$,

```

    (False ∧ P) = False.
Proof using. tautop. Qed.
Lemma and_False_r_eq : ∀ P,
    (P ∧ False) = False.
Proof using. tautop. Qed.
Lemma or_True_l_eq : ∀ P,
    (True ∨ P) = True.
Proof using. tautop. Qed.
Lemma or_True_r_eq : ∀ P,
    (P ∨ True) = True.
Proof using. tautop. Qed.
Lemma or_False_l_eq : ∀ P,
    (False ∨ P) = P.
Proof using. tautop. Qed.
Lemma or_False_r_eq : ∀ P,
    (P ∨ False) = P.
Proof using. tautop. Qed.
End SimplConjDisj.

```

4.4.2 Distribution of negation on basic operators

```

Section SimplNot.
Implicit Types P Q : Prop.
Lemma not_True_eq :
    (¬ True) = False.
Proof using. tautop. Qed.
Lemma not_False_eq :
    (¬ False) = True.
Proof using. tautop. Qed.
Lemma not_not_eq : ∀ P,
    (¬ (¬ P)) = P.
Proof using. tautop. Qed.
Lemma not_and_eq : ∀ P Q,
    (¬ (P ∧ Q)) = (¬ P ∨ ¬ Q).
Proof using. tautop. Qed.
Lemma not_or_eq : ∀ P Q,
    (¬ (P ∨ Q)) = (¬ P ∧ ¬ Q).
Proof using. tautop. Qed.
Lemma not_impl_eq : ∀ P Q,

```

$(\neg (P \rightarrow Q)) = (P \wedge \neg Q).$
 Proof using. *tautop*. Qed.

Lemma not_or_ots_eq : $\forall P Q,$
 $(\neg (\sim P \vee \neg Q)) = (P \wedge Q).$
 Proof using. *tautop*. Qed.

Lemma not_and_ots_eq : $\forall P Q,$
 $(\neg (\sim P \wedge \neg Q)) = (P \vee Q).$
 Proof using. *tautop*. Qed.
 End SimplNot.

4.4.3 Double negation and contrapose

Section DoubleNeg.
 Implicit Types $P Q : \text{Prop}.$

Double negation

Lemma not_not_inv : $\forall P,$
 $\neg \neg P \rightarrow$
 $P.$
 Proof using. *tautop*. Qed.

Lemma not_not : $\forall P,$
 $P \rightarrow$
 $\neg \neg P.$
 Proof using. *tautop*. Qed.

Contrapose

Lemma contrapose_eq : $\forall P Q,$
 $(\neg P \rightarrow \neg Q) = (Q \rightarrow P).$
 Proof using. *tautop*. Qed.

Lemma contrapose_inv : $\forall P Q,$
 $(\neg Q \rightarrow \neg P) \rightarrow$
 $(P \rightarrow Q).$
 Proof using. *tautop*. Qed.

Lemma contrapose : $\forall P Q,$
 $(Q \rightarrow P) \rightarrow$
 $(\neg P \rightarrow \neg Q).$
 Proof using. *tautop*. Qed.

Negation is injective

Lemma injective_not : $\forall P Q,$
 $(\neg P) = (\neg Q) \rightarrow$

$(P = Q)$.
 Proof using. *tautop*. Qed.
 End DoubleNeg.

4.4.4 Distribution of negation on quantifiers

Section SimplNotQuantifiers.

Variables $(A : \text{Type})$.

Implicit Types $P : A \rightarrow \text{Prop}$.

Three auxiliary facts (private lemmas)

Lemma exists_of_not_forall : $\forall P,$
 $\neg (\forall x, \neg P x) \rightarrow$
 $(\exists x, P x).$

Proof using. intros. apply $\times$ not_not_inv. Qed.

Lemma forall_of_not_exists : $\forall P,$
 $\neg (\exists x, \neg P x) \rightarrow$
 $(\forall x, P x).$

Proof using. intros. apply $\times$ not_not_inv. Qed.

Lemma not_not_pred_eq : $\forall A (P : A \rightarrow \text{Prop}),$
 $P = (\text{fun } x \Rightarrow \neg \neg (P x)).$

Proof using. intros. *extens*. intros. rewrite $\times$ not_not_eq. Qed.

Rewriting rules for quantifiers

Lemma not_forall_eq : $\forall P,$
 $(\neg (\forall x, P x)) = (\exists x, \neg P x).$

Proof using.

extens. *iff*.

$\{ \text{apply exists\_of\_not\_forall. rewrite } \neg (\text{not\_not\_pred\_eq } P) \text{ in } H. \}$
 $\{ \text{intros } M. \text{destruct } \neg H \text{ as } [x \ Cx]. \}$

Qed.

Lemma not_exists_eq : $\forall P,$
 $(\neg (\exists x, P x)) = (\forall x, \neg P x).$

Proof using.

intros. apply injective_not. rewrite not_forall_eq.
 rewrite not_not_eq. set $(P' := P)$ at 1.
 rewrite $\neg (\text{not\_not\_pred\_eq } P')$.

Qed.

Lemma not_forall_not_eq : $\forall P,$
 $(\neg (\forall x, \neg P x)) = (\exists x, P x).$

Proof using.

```

    intros. rewrite not_forall_eq.
    set (P' := P) at 2. rewrite ¬ (not_not_pred_eq P').
Qed.

Lemma not_exists_not_eq : ∀ P,
  (¬ (∃ x, ¬ P x)) = (∀ x, P x).
Proof using.
  intros. rewrite not_exists_eq.
  set (P' := P) at 2. rewrite ¬ (not_not_pred_eq P').
Qed.

End SimplNotQuantifiers.

```

4.4.5 Propositions equal to **True** or **False**

Section EqTrueFalse.

Implicit Types $P Q : \text{Prop}$.

Propositions equal to **True**

```

Lemma prop_eq_True_eq : ∀ P,
  (P = True) = P.

```

Proof using. *tautop*. Qed.

```

Lemma prop_eq_True : ∀ P,
  P →
  P = True.

```

Proof using. *tautop*. Qed.

```

Lemma prop_eq_True_inv : ∀ P,
  P = True →
  P.

```

Proof using. *tautop*. Qed.

Propositions equal to **False**

```

Lemma prop_eq_False_eq : ∀ P,
  (P = False) = ¬ P.

```

Proof using. *tautop*. Qed.

```

Lemma prop_eq_False : ∀ P,
  ¬ P →
  P = False.

```

Proof using. *tautop*. Qed.

```

Lemma prop_eq_False_inv : ∀ P,
  P = False →
  ¬ P.

```

Proof using. *tautop*. Qed.

End EqTrueFalse.

#[*global*]

Hint Resolve *prop_eq_True prop_eq_False*.

4.4.6 Disequal propositions

Section NeqProp.

Implicit Types $P Q : \text{Prop}$.

Propositions not **True** or not **False**

Lemma prop_neq_True_eq : $\forall P,$
 $(P \neq \mathbf{True}) = \neg P.$

Proof using. *tautop*. Qed.

Lemma prop_neq_False_eq : $\forall P,$
 $(P \neq \mathbf{False}) = P.$

Proof using. *tautop*. Qed.

Proving two propositions not equal

Lemma prop_neq_of_iff_l : $\forall P Q,$
 $(P \leftrightarrow \neg Q) \rightarrow$
 $P \neq Q.$

Proof using. *tautop*. Qed.

Lemma prop_neq_of_iff_r : $\forall P Q,$
 $(\neg P \leftrightarrow Q) \rightarrow$
 $P \neq Q.$

Proof using. *tautop*. Qed.

Lemma prop_neq_inv_iff_l : $\forall P Q,$
 $P \neq Q \rightarrow$
 $(P \leftrightarrow \neg Q).$

Proof using. *tautop*. Qed.

Lemma prop_neq_inv_iff_r : $\forall P Q,$
 $P \neq Q \rightarrow$
 $(\neg P \leftrightarrow Q).$

Proof using. *tautop*. Qed.

True is not False

Lemma True_neq_False : $\mathbf{True} \neq \mathbf{False}.$

Proof using. *tautop*. Qed.

End NeqProp.

#[*global*]

Hint Resolve *True_neq_False*.

4.4.7 Peirce rule and similar

Section ClassicOr.

Implicit Types $P Q : \text{Prop}$.

Peirce's result: proving a fact by assuming its negation

Lemma assume_not : $\forall P,$
 $(\neg P \rightarrow P) \rightarrow$
 $P.$

Proof using. *tautop*. Qed.

Proving a disjunction, assuming the negation of the other branch

Lemma or_classic_l : $\forall P Q,$
 $(\neg Q \rightarrow P) \rightarrow$
 $P \vee Q.$

Proof using. *tautop*. Qed.

Lemma or_classic_r : $\forall P Q,$
 $(\neg P \rightarrow Q) \rightarrow$
 $P \vee Q.$

Proof using. *tautop*. Qed.

Same, but in forward style

Lemma or_inv_classic_l : $\forall P Q,$
 $P \vee Q \rightarrow$
 $(P \vee (\neg P \wedge Q)).$

Proof using. *tautop*. Qed.

Lemma or_inv_classic_r : $\forall P Q,$
 $P \vee Q \rightarrow$
 $(Q \vee (P \wedge \neg Q)).$

Proof using. *tautop*. Qed.

End ClassicOr.

4.4.8 Properties of logical equivalence

Section IffProp.

Implicit Types $P Q R : \text{Prop}$.

Introduction

Lemma iff_eq_and : $\forall P Q : \text{Prop},$
 $(P \leftrightarrow Q) = ((P \rightarrow Q) \wedge (Q \rightarrow P)).$

Proof using. *tautop*. Qed.

Lemma iff_intro : $\forall P Q : \text{Prop},$
 $(P \rightarrow Q) \rightarrow$

$(Q \rightarrow P) \rightarrow$
 $(P \leftrightarrow Q).$

Proof using. *intros. rewrite× iff_eq_and. Qed.*

Reflexivity: *refl iff*

Lemma *iff_refl* : $\forall P,$
 $P \leftrightarrow P.$

Proof using. *tautop. Qed.*

Symmetry: *sym iff*

Lemma *iff_sym* : $\forall P Q,$
 $(P \leftrightarrow Q) \rightarrow$
 $(Q \leftrightarrow P).$

Proof using. *tautop. Qed.*

Transitivity: *trans iff*

Lemma *iff_trans* : $\forall P Q R,$
 $(P \leftrightarrow Q) \rightarrow$
 $(Q \leftrightarrow R) \rightarrow$
 $(P \leftrightarrow R).$

Proof using. *tautop. Qed.*

First projection

Lemma *iff_l* : $\forall P Q,$
 $(P \leftrightarrow Q) \rightarrow$
 $P \rightarrow$
 $Q.$

Proof using. *tautop. Qed.*

Second projection

Lemma *iff_r* : $\forall P Q,$
 $(P \leftrightarrow Q) \rightarrow$
 $Q \rightarrow$
 $P.$

Proof using. *tautop. Qed.*

Contrapose of the first projection

Lemma *iff_not_l* : $\forall P Q,$
 $(P \leftrightarrow Q) \rightarrow$
 $\neg P \rightarrow$
 $\neg Q.$

Proof using. *tautop. Qed.*

Contrapose of the second projection

Lemma *iff_not_r* : $\forall P Q,$

$$(P \leftrightarrow Q) \rightarrow \\ \neg Q \rightarrow \\ \neg P.$$

Proof using. *tautop*. Qed.

Negation can change side of an equivalence

Lemma *iff_not_swap_eq* : $\forall P Q,$
 $((\neg P) \leftrightarrow Q) = (P \leftrightarrow (\neg Q)).$

Proof using. *tautop*. Qed.

Negation can be cancelled on both sides

Lemma *iff_not_not_eq* : $\forall P Q,$
 $((\neg P) \leftrightarrow (\neg Q)) = (P \leftrightarrow Q).$

Proof using. *tautop*. Qed.

End IffProp.

4.5 Tactics

4.5.1 Tactic for simplifying expressions

The tactic *rew_logic* can be used to automatically simplify logical expressions. Syntax *rew_logic in H* and *rew_logic in ** are also available.

#[global]

Hint Rewrite

not_not_eq not_and_eq not_or_eq not_impl_eq not_True_eq not_False_eq
not_forall_eq not_forall_not_eq
not_exists_eq not_exists_not_eq not_impl_eq
prop_eq_True_eq prop_eq_False_eq eq_prop_eq_iff
and_True_l_eq and_True_r_eq and_False_l_eq and_False_r_eq
or_True_l_eq or_True_r_eq or_False_l_eq or_False_r_eq
not_False_eq not_True_eq
 : *rew_logic*.

Tactic Notation "rew_logic" :=

autorewrite with rew_logic.

Tactic Notation "rew_logic" "in" *hyp(H)* :=

autorewrite with rew_logic in H.

Tactic Notation "rew_logic" "in" "*" :=

autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_logic).

Tactic Notation "rew_logic" "~" :=

rew_logic; auto_tilde.

Tactic Notation "rew_logic" "~" "in" *hyp(H)* :=

rew_logic in H; auto_tilde.

```

Tactic Notation "rew_logic" "~" "in" "*" :=
  rew_logic in *; auto_tilde.

Tactic Notation "rew_logic" "*" :=
  rew_logic; auto_star.

Tactic Notation "rew_logic" "*" "in" hyp(H) :=
  rew_logic in H; auto_star.

Tactic Notation "rew_logic" "*" "in" "*" :=
  rew_logic in *; auto_star.

```

4.5.2 Tactic *tests*: P for classical disjunction on P .

The tactic *tests*: P can be used to tests whether the proposition P is true or not. If P is an equality, it is substituted. Use the tactic *tests_nosubst*: P to disable the automated substitution. Use the tactic *tests_basic*: P to moreover disable simplification of logical expressions. To name the resulting hypotheses use *tests I*: P , or *tests I1 I2*: P to assign different names to both sides. (In LibReflect, the tactic is extended so that P can be a boolean.)

```

Ltac tests_ssum_base E H1 H2 :=
  destruct E as [H1|H2].

Ltac tests_prop_base E H1 H2 :=
  tests_ssum_base (classicT E) H1 H2.

Ltac tests_dispatch E H1 H2 :=
  match type of E with
  | Prop => tests_prop_base E H1 H2
  | {_}+{_} => tests_ssum_base E H1 H2
  end.

Ltac tests_post H introstac :=
  tryfalse; rew_logic in H; revert H;
  introstac tt; tryfalse.

Ltac tests_post_subst H I :=
  tests_post H ltac:(fun _ => first [ intro_subst_hyp | intros I ]).

Ltac tests_post_nosubst H I :=
  tests_post H ltac:(fun _ => intros I).

Ltac tests_base E I1 I2 tests_post :=
  let H1 := fresh "TEMP" in
  let H2 := fresh "TEMP" in
  tests_dispatch E H1 H2;
  [ tests_post H1 I1
  | tests_post H2 I2 ].

```

```

Tactic Notation "tests" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E)
:=
  tests_base E I1 I2 ltac:(tests_post_subst).
Tactic Notation "tests" simple_intropattern(I) ":" constr(E) :=
  tests I I: E.
Tactic Notation "tests" ":" constr(E) :=
  let I := fresh "C" in tests I: E.
Tactic Notation "tests" "~" ":" constr(E) :=
  tests: E; auto_tilde.
Tactic Notation "tests" "*" ":" constr(E) :=
  tests: E; auto_star.

Tactic Notation "tests_nosubst" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E)
:=
  tests_base E I1 I2 ltac:(tests_post_nosubst).
Tactic Notation "tests_nosubst" simple_intropattern(I) ":" constr(E) :=
  tests_nosubst I I: E.
Tactic Notation "tests_nosubst" ":" constr(E) :=
  let I := fresh "C" in tests I: E.
Tactic Notation "tests_nosubst" "~" ":" constr(E) :=
  tests: E; auto_tilde.
Tactic Notation "tests_nosubst" "*" ":" constr(E) :=
  tests: E; auto_star.

Tactic Notation "tests_basic" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E)
:=
  tests_dispatch E I1 I2.
Tactic Notation "tests_basic" simple_intropattern(I1) ":" constr(E) :=
  tests_basic I1 I1: E.
Tactic Notation "tests_basic" ":" constr(E) :=
  let C := fresh "C" in tests_basic C: E.

```

4.5.3 Tactic *case_classic*

case_classic performs a case analysis on the first expression of the form *classicT* ?*E* that appears in the goal.

```

Tactic Notation "case_classic" "as" simple_intropattern(C) :=
  match goal with
  | ⊢ context [ classicT ?E ] ⇒ destruct (classicT E) as [C|C]
  | H: context [ classicT ?E ] ⊢ _ ⇒ destruct (classicT E) as [C|C]
  end; tryfalse.

Tactic Notation "case_classic" :=
  let C := fresh "C" in case_classic as C.

```

4.5.4 Tactic *absurds*

absurds applies to a goal G and replaces it with $(\sim G) \rightarrow \mathbf{False}$. The expression $\neg G$ is simplified using *rew_logic*.

absurds $;\Rightarrow H$ introduces H as the negation of the goal, and leaves $\mathbf{False}$ is the goal.

absurds_nosimpl is similar, but it does not perform simplifications.

```
Lemma absurds_lemma :  $\forall (G:\text{Prop}),$   
   $((\neg G) \rightarrow \mathbf{False}) \rightarrow$   
   $G$ .
```

Proof using. intros. *applies* $\neg$ not_not_inv. Qed.

```
Ltac absurds_nosimpl_core tt :=  
  applies absurds_lemma.
```

```
Tactic Notation "absurds_nosimpl" :=  
  absurds_nosimpl_core tt.
```

```
Ltac absurds_post tt :=  
  rew_logic.
```

```
Ltac absurds_core tt :=  
  applies absurds_lemma;  
  absurds_post tt.
```

```
Tactic Notation "absurds" :=  
  absurds_core tt.
```

4.6 Predicate combinators, comparison and compatibility

4.6.1 Definition of predicate combinators

```
Definition pred_true {A : Type} :=  
  fun (_:A)  $\Rightarrow \mathbf{True}$ .
```

```
Definition pred_false {A : Type} :=  
  fun (_:A)  $\Rightarrow \mathbf{False}$ .
```

```
Definition pred_not (A : Type) (P : A  $\rightarrow$  Prop) :=  
  fun x  $\Rightarrow \neg (P\ x)$ .
```

```
Definition pred_and (A : Type) (P Q : A  $\rightarrow$  Prop) :=  
  fun x  $\Rightarrow P\ x \wedge Q\ x$ .
```

```
Definition pred_or (A : Type) (P Q : A  $\rightarrow$  Prop) :=  
  fun x  $\Rightarrow P\ x \vee Q\ x$ .
```

```
Definition pred_impl (A : Type) (P Q : A  $\rightarrow$  Prop) :=
```

```

    fun x => P x → Q x.
#[global]
Hint Unfold pred_true pred_false.

```

4.6.2 Properties of combinators

```

Lemma pred_conj_forall_distrib : ∀ A (P Q : A → Prop),
  ((∀ x, P x) ∧ (∀ x, Q x)) = (∀ x, P x ∧ Q x).
Proof using. intros. apply prop_ext. iff H. autos×. split; intros x; apply× (H x).
Qed.

```

4.6.3 Order on predicates

```

Definition pred_incl (A : Type) (P Q : A → Prop) :=
  ∀ x, P x → Q x.

```

```

Lemma pred_eq_forall_impl : ∀ A (P Q : A → Prop),
  pred_incl P Q = (∀ x, P x → Q x).
Proof using. auto. Qed.

```

```

Lemma pred_incl_refl : ∀ A (P : A → Prop),
  pred_incl P P.
Proof using. unfolds¬ pred_incl. Qed.

```

```

Lemma pred_incl_trans : ∀ A (Q P R : A → Prop),
  pred_incl P Q →
  pred_incl Q R →
  pred_incl P R.
Proof using. unfolds¬ pred_incl. Qed.

```

```

Lemma pred_incl_antisym : ∀ A (P Q : A → Prop),
  pred_incl P Q →
  pred_incl Q P →
  P = Q.
Proof using. extens×. Qed.

```

See also [LibRelation](#) and [LibOrder](#) for higher-level statements of these results

4.7 Existentials

4.8 Properties of unique existentials

`unique_st P x` asserts that `x` is the unique element for which `P x` holds.

Definition unique_st ($A : \text{Type}$) ($P : A \rightarrow \text{Prop}$) ($x : A$) :=
 $P\ x \wedge \forall y, P\ y \rightarrow y = x.$

#[global]

Hint Unfold unique_st.

$\text{ex\_unique}\ P$ asserts that there exists a unique element for which $P\ x$ holds.

Definition ex_unique ($A : \text{Type}$) ($P : A \rightarrow \text{Prop}$) :=
 $\text{ex}\ (\text{unique\_st}\ P).$

$\exists! x, P$ is the notation for $\text{ex\_unique}\ P$.

Notation "'exists' ! x , P" := ($\text{ex\_unique}\ (\text{fun } x \Rightarrow P)$)
(at level 200, x name, right associativity,
format "'[' 'exists' ! ' / ' x , ' / ' P ']'") : type_scope.

Notation "'exists' ! x : A , P" :=
($\text{ex\_unique}\ (\text{fun } x:A \Rightarrow P)$)
(at level 200, x name, right associativity,
format "'[' 'exists' ! ' / ' x : A , ' / ' P ']'") : type_scope.

$\text{at\_most\_one}\ P$ asserts that there exists at most one element for which $P\ x$ holds. In other words, P has 0 or 1 inhabitant.

Definition at_most_one ($A : \text{Type}$) ($P : A \rightarrow \text{Prop}$) :=
 $\forall x\ y, P\ x \rightarrow P\ y \rightarrow x = y.$

Section UniqueProp.

Variables ($A : \text{Type}$).

Implicit Types ($P : A \rightarrow \text{Prop}$).

$\exists! x, P$ entails $\exists x, P$

Lemma ex_of_ex_unique : $\forall P,$
 $\text{ex\_unique}\ P \rightarrow$
 $\text{ex}\ P.$

Proof using. *introv* ($x \& H \& U$). *eauto*. Qed.

$\exists! x, P$ entails $\text{at\_most\_one}\ P$

Lemma at_most_one_of_ex_unique : $\forall P,$
 $\text{ex\_unique}\ P \rightarrow$
 $\text{at\_most\_one}\ P.$

Proof using.

introv ($a \& H \& U$) $Px\ Py$. *applies* $\neg$ *eq_trans* *a*. *rewrite* $\neg \leftarrow (U\ y).$

Qed.

$\exists x, P$ and $\text{at\_most\_one}\ P$ entail $\exists! x, P$

Lemma ex_unique_of_ex_at_most_one : $\forall P,$
 $\text{ex}\ P \rightarrow$
 $\text{at\_most\_one}\ P \rightarrow$

```

    ex_unique P.
Proof using. intros [x Px] H.  $\exists$  x. split  $\neg$ . Qed.
End UniqueProp.

```

4.9 Conjunctions

4.9.1 Changing the order of branches

```

Lemma conj_swap:  $\forall$  (P Q: Prop),
  P  $\rightarrow$ 
  Q  $\rightarrow$ 
  Q  $\wedge$  P.

```

Proof using. *autos* $\times$. Qed.

```

Lemma conj_dup_r :  $\forall$  P Q : Prop,
  Q  $\rightarrow$ 
  (Q  $\rightarrow$  P)  $\rightarrow$ 
  P  $\wedge$  Q.

```

Proof using. *autos* $\times$. Qed.

```

Lemma conj_dup_l :  $\forall$  P Q : Prop,
  P  $\rightarrow$ 
  (P  $\rightarrow$  Q)  $\rightarrow$ 
  P  $\wedge$  Q.

```

Proof using. *autos* $\times$. Qed.

4.9.2 Parallel strengthening of a conjunction

```

Lemma conj_strengthen_2 :  $\forall$  (Q1 Q2 P1 P2 : Prop),
  (Q1  $\wedge$  Q2)  $\rightarrow$ 
  (Q1  $\rightarrow$  P1)  $\rightarrow$ 
  (Q2  $\rightarrow$  P2)  $\rightarrow$ 
  (P1  $\wedge$  P2).

```

Proof using. *autos* $\times$. Qed.

```

Lemma conj_strengthen_3 :  $\forall$  (Q1 Q2 Q3 P1 P2 P3 : Prop),
  (Q1  $\wedge$  Q2  $\wedge$  Q3)  $\rightarrow$ 
  (Q1  $\rightarrow$  P1)  $\rightarrow$ 
  (Q2  $\rightarrow$  P2)  $\rightarrow$ 
  (Q3  $\rightarrow$  P3)  $\rightarrow$ 
  (P1  $\wedge$  P2  $\wedge$  P3).

```

Proof using. *autos* $\times$. Qed.

```

Lemma conj_strengthen_4 :  $\forall$  (Q1 Q2 Q3 Q4 P1 P2 P3 P4 : Prop),

```

```

(Q1 ∧ Q2 ∧ Q3 ∧ Q4) →
(Q1 → P1) →
(Q2 → P2) →
(Q3 → P3) →
(Q4 → P4) →
(P1 ∧ P2 ∧ P3 ∧ P4).

```

Proof using. *autos*×. Qed.

```

Lemma conj_strengthen_5 : ∀ (Q1 Q2 Q3 Q4 Q5 P1 P2 P3 P4 P5 : Prop),
  (Q1 ∧ Q2 ∧ Q3 ∧ Q4 ∧ Q5) →
  (Q1 → P1) →
  (Q2 → P2) →
  (Q3 → P3) →
  (Q4 → P4) →
  (Q5 → P5) →
  (P1 ∧ P2 ∧ P3 ∧ P4 ∧ P5).

```

Proof using. *autos*×. Qed.

4.9.3 Projections of lemmas concluding on a conjunction

Section ProjLemma.

Variables (A1 : Type).

Variables (A2 : ∀ (x1 : A1), Type).

Variables (A3 : ∀ (x1 : A1) (x2 : A2 x1), Type).

Variables (A4 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), Type).

Variables (A5 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), Type).

Variables (A6 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), Type).

Variables (A7 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5), Type).

Variables (A8 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6), Type).

Variables (A9 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6) (x8 : A8 x7), Type).

Variables (A10 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6) (x8 : A8 x7) (x9 : A9 x8), Type).

Variables (A11 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4) (x6 : A6 x5) (x7 : A7 x6) (x8 : A8 x7) (x9 : A9 x8) (x10 : A10 x9), Type).

```

Ltac prove_forall_conj_inv :=
  intros;
  match goal with H: context [_ ∧ _] ⊢ _ =>
    split; intros; apply H end.

```

```

Lemma forall_conj_inv_1 : ∀ (P Q : ∀ (x1:A1), Prop),
  (∀ x1, P x1 ∧ Q x1) →

```


$(\forall x1, P x1) \wedge$
 $(\forall x1, Q x1).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_2 : $\forall (P Q : \forall (x1:A1) (x2:A2 x1), \text{Prop}),$
 $(\forall x1 x2, P x1 x2 \wedge Q x1 x2) \rightarrow$
 $(\forall x1 x2, P x1 x2) \wedge$
 $(\forall x1 x2, Q x1 x2).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_3 : $\forall (P Q : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2), \text{Prop}),$
 $(\forall x1 x2 x3, P x1 x2 x3 \wedge Q x1 x2 x3) \rightarrow$
 $(\forall x1 x2 x3, P x1 x2 x3) \wedge$
 $(\forall x1 x2 x3, Q x1 x2 x3).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_4 : $\forall (P Q : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$
 $(x4:A4 x3), \text{Prop}),$
 $(\forall x1 x2 x3 x4, P x1 x2 x3 x4 \wedge Q x1 x2 x3 x4) \rightarrow$
 $(\forall x1 x2 x3 x4, P x1 x2 x3 x4) \wedge$
 $(\forall x1 x2 x3 x4, Q x1 x2 x3 x4).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_5 : $\forall (P Q : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$
 $(x4:A4 x3) (x5:A5 x4), \text{Prop}),$
 $(\forall x1 x2 x3 x4 x5, P x1 x2 x3 x4 x5 \wedge Q x1 x2 x3 x4 x5) \rightarrow$
 $(\forall x1 x2 x3 x4 x5, P x1 x2 x3 x4 x5) \wedge$
 $(\forall x1 x2 x3 x4 x5, Q x1 x2 x3 x4 x5).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_6 : $\forall (P Q : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$
 $(x4:A4 x3) (x5:A5 x4) (x6:A6 x5), \text{Prop}),$
 $(\forall x1 x2 x3 x4 x5 x6, P x1 x2 x3 x4 x5 x6 \wedge Q x1 x2 x3 x4 x5 x6) \rightarrow$
 $(\forall x1 x2 x3 x4 x5 x6, P x1 x2 x3 x4 x5 x6) \wedge$
 $(\forall x1 x2 x3 x4 x5 x6, Q x1 x2 x3 x4 x5 x6).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_7 : $\forall (P Q : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$
 $(x4:A4 x3) (x5:A5 x4) (x6:A6 x5) (x7:A7 x6), \text{Prop}),$
 $(\forall x1 x2 x3 x4 x5 x6 x7, P x1 x2 x3 x4 x5 x6 x7 \wedge Q x1 x2 x3 x4 x5 x6 x7) \rightarrow$
 $(\forall x1 x2 x3 x4 x5 x6 x7, P x1 x2 x3 x4 x5 x6 x7) \wedge$
 $(\forall x1 x2 x3 x4 x5 x6 x7, Q x1 x2 x3 x4 x5 x6 x7).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_8 : $\forall (P Q : \forall (x1:A1) (x2:A2 x1) (x3:A3 x2)$
 $(x4:A4 x3) (x5:A5 x4) (x6:A6 x5) (x7:A7 x6) (x8:A8 x7), \text{Prop}),$
 $(\forall x1 x2 x3 x4 x5 x6 x7 x8, P x1 x2 x3 x4 x5 x6 x7 x8 \wedge Q x1 x2 x3 x4 x5 x6 x7 x8)$
 $\rightarrow$

$(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_9 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6) (x8:A8\ x7) (x9:A9\ x8), \text{Prop}),$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9 \wedge Q\ x1\ x2\ x3\ x4\ x5\ x6$
 $x7\ x8\ x9) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_10 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6) (x8:A8\ x7) (x9:A9\ x8) (x10:A10\ x9),$
 Prop),
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10 \wedge Q\ x1\ x2\ x3$
 $x4\ x5\ x6\ x7\ x8\ x9\ x10) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10).$

Proof using. *prove_forall_conj_inv*. Qed.

Lemma forall_conj_inv_11 : $\forall (P\ Q : \forall (x1:A1) (x2:A2\ x1) (x3:A3\ x2)$
 $(x4:A4\ x3) (x5:A5\ x4) (x6:A6\ x5) (x7:A7\ x6) (x8:A8\ x7) (x9:A9\ x8) (x10:A10\ x9)$
 $(x11:A11\ x10), \text{Prop}),$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11 \wedge Q$
 $x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11) \rightarrow$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11, P\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11) \wedge$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11, Q\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11).$

Proof using. *prove_forall_conj_inv*. Qed.

End ProjLemma.

Arguments forall_conj_inv_1 [A1] [P] [Q].
 Arguments forall_conj_inv_2 [A1] [A2] [P] [Q].
 Arguments forall_conj_inv_3 [A1] [A2] [A3] [P] [Q].
 Arguments forall_conj_inv_4 [A1] [A2] [A3] [A4] [P] [Q].
 Arguments forall_conj_inv_5 [A1] [A2] [A3] [A4] [A5] [P] [Q].
 Arguments forall_conj_inv_6 [A1] [A2] [A3] [A4] [A5] [A6] [P] [Q].
 Arguments forall_conj_inv_7 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [P] [Q].
 Arguments forall_conj_inv_8 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [P] [Q].
 Arguments forall_conj_inv_9 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [A9] [P] [Q].
 Arguments forall_conj_inv_10 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [A9] [A10] [P] [Q].
 Arguments forall_conj_inv_11 [A1] [A2] [A3] [A4] [A5] [A6] [A7] [A8] [A9] [A10] [A11]
 [P] [Q].

Chapter 5

Library SLF.LibOperation

Set Implicit Arguments.
From *SLF* Require Import LibTactics.

5.1 Definitions

Section Definitions.
Variables ($A : \text{Type}$).
Implicit Types $f\ g : A \rightarrow A \rightarrow A$.
Implicit Types $i : A \rightarrow A$.

5.1.1 Commutativity, associativity

Commutativity

Definition $\text{comm } f := \forall x\ y,$
 $f\ x\ y = f\ y\ x.$

Associativity

Definition $\text{assoc } f := \forall x\ y\ z,$
 $f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z.$

Combined associativity commutativity

Definition $\text{comm\_assoc } f := \forall x\ y\ z,$
 $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z).$

5.1.2 Distributivity

Distributivity of unary operator

Definition $\text{distrib } i\ f := \forall x\ y,$

$$i (f x y) = f (i x) (i y).$$

comm distributivity of unary operator

Definition `distrib_comm` $i f := \forall x y,$

$$i (f x y) = f (i y) (i x).$$

Left distributivity

Definition `distrib_l` $f g := \forall x y z,$

$$f x (g y z) = g (f x y) (f x z).$$

Right distributivity

Definition `distrib_r` $f g := \forall x y z,$

$$f (g y z) x = g (f y x) (f z x).$$

5.1.3 Neutral and absorbant

Left Neutral

Definition `neutral_l` $f e := \forall x,$

$$f e x = x.$$

Right Neutral

Definition `neutral_r` $f e := \forall x,$

$$f x e = x.$$

Left Absorbant

Definition `absorb_l` $f a := \forall x,$

$$f a x = a.$$

Right Absorbant

Definition `absorb_r` $f a := \forall x,$

$$f x a = a.$$

Idempotence

Definition `idempotent` $i := \forall x,$

$$i (i x) = i x.$$

Idempotence

Definition `involutive` $i := \forall x,$

$$i (i x) = x.$$

Idempotence for binary operators

Definition `idempotent2` $f := \forall x,$

$$f x x = x.$$

Self Neutral

Definition `self_neutral` $f e x :=$

$$f x x = e.$$

5.1.4 Inverses

Left Inverse

Definition `inverse_for_l` $f\ e\ a\ b :=$
 $f\ a\ b = e.$

Right Inverse

Definition `inverse_for_r` $f\ e\ a\ b :=$
 $f\ b\ a = e.$

Left Inverse function – todo arguments in order f i e ?

Definition `inverse_l` $f\ e\ i := \forall\ x,$
 $f\ (i\ x)\ x = e.$

Right Inverse function

Definition `inverse_r` $f\ e\ i := \forall\ x,$
 $f\ x\ (i\ x) = e.$

Self Inverse

Definition `self_inverse` $i\ x :=$
 $i\ x = x.$

End Definitions.

5.1.5 Morphism and automorphism

Morphism

Definition `morphism` $(A\ B : \text{Type})\ (h : A \rightarrow B)\ (f : A \rightarrow A \rightarrow A)\ (g : B \rightarrow B \rightarrow B) :=$
 $\forall\ x\ y, h\ (f\ x\ y) = g\ (h\ x)\ (h\ y).$

Auto-morphism

Definition `automorphism` $A := @morphism\ A\ A.$
Arguments `automorphism` $[A].$

5.1.6 Injectivity

Definition `injective` $A\ B\ (f : A \rightarrow B) :=$
 $\forall\ x\ y, f\ x = f\ y \rightarrow x = y.$

5.2 Lemmas

Section `OpProperties`.

Variables $(A : \text{Type}).$

Implicit Types $h : A \rightarrow A.$

Implicit Types $f\ g : A \rightarrow A \rightarrow A$.

Lemma idempotent_inv : $\forall\ h\ x\ y,$
 idempotent $h \rightarrow$
 $y = h\ x \rightarrow$
 $h\ y = y$.

Proof using. intros. subst $\times$. Qed.

For comm operators, right-properties can be derived from corresponding left-properties

Lemma neutral_r_of_comm_neutral_l : $\forall\ f\ e,$
 comm $f \rightarrow$
 neutral_l $f\ e \rightarrow$
 neutral_r $f\ e$.

Proof using. introv $C\ N$. intros_all. rewrite $\times$ C . Qed.

Lemma inverse_r_of_comm_inverse_l : $\forall\ f\ e\ i,$
 comm $f \rightarrow$
 inverse_l $f\ e\ i \rightarrow$
 inverse_r $f\ e\ i$.

Proof using. introv $C\ I$. intros_all. rewrite $\times$ C . Qed.

Lemma distrib_r_of_comm_distrib_l : $\forall\ f\ g,$
 comm $f \rightarrow$
 distrib_l $f\ g \rightarrow$
 distrib_r $f\ g$.

Proof using.
 introv $C\ N$. intros_all. unfolds distrib_l.
 do 3 rewrite $\leftarrow (C\ x)$. auto.
Qed.

Lemma comm_assoc_of_comm_and_assoc : $\forall\ f,$
 comm $f \rightarrow$
 assoc $f \rightarrow$
 comm_assoc f .

Proof using.
 introv $C\ S$. intros_all. rewrite C .
 rewrite $\leftarrow S$. rewrite $\neg$ $(C\ x)$.
Qed.

End OpProperties.

Chapter 6

Library **SLF.LibBool**

```
Set Implicit Arguments.  
From SLF Require Import LibTactics LibLogic LibOperation.
```

6.1 Boolean type

6.1.1 Definition

From the Prelude:

```
Inductive bool : Type := | true : bool | false : bool.
```

6.1.2 Inhabited

```
#[global]  
Instance Inhab_bool : Inhab bool.  
Proof using. constructor. apply (Inhab_of_val true). Qed.
```

6.1.3 Extensionality

See **LibReflect** for extensionality of booleans.

6.2 Boolean Operations

6.2.1 Comparison

```
Definition eqb (x y:bool) : bool :=  
  match x, y with  
  | true, true => true
```

```

| false, false  $\Rightarrow$  true
| _, _  $\Rightarrow$  false
end.

```

Negation

```

Definition neg (x:bool) : bool :=
  match x with
  | true  $\Rightarrow$  false
  | false  $\Rightarrow$  true
  end.

```

Conjunction

```

Definition and (x y:bool) : bool :=
  match x, y with
  | true, true  $\Rightarrow$  true
  | _, _  $\Rightarrow$  false
  end.

```

Disjunction

```

Definition or (x y:bool) : bool :=
  match x, y with
  | false, false  $\Rightarrow$  false
  | _, _  $\Rightarrow$  true
  end.

```

Implication

```

Definition impl (x y:bool) : bool :=
  or x (neg y).

```

Exclusive or

```

Definition xor (x y:bool) : bool :=
  neg (eqb x y).

```

Notations

Declare Scope *Bool_scope*.

Bind Scope *Bool_scope* with *bool*.

Open Scope *Bool_scope*.

Notation " $\neg$ x" := (neg x)

(at level 35, right associativity) : *Bool_scope*.

Infix "&&" := and

(at level 40, left associativity) : *Bool_scope*.

Infix "||" := or

(at level 50, left associativity) : *Bool_scope*.

Notation " $\neg$ x" := (neg x)


```

    (at level 35, right associativity) : Bool_scope.
Infix "&&" := and
    (at level 40, left associativity) : Bool_scope.
Infix "||" := or
    (at level 50, left associativity) : Bool_scope.

```

6.3 Boolean Decision Procedure : tactic working by exponential case

analysis on all variables of type bool.

6.3.1 Tactic *tautob*

tautob introduces all variables that it can, performs a case analysis on all the boolean variables, then splits all subgoals and attempts resolution using intuition

```

Ltac tautob_post tt :=
  simpls; try split; intros; try discriminate;
  try solve [ intuition auto_false ].

```

```

Ltac tautob_core tt :=
  let rec aux tt :=
    (try intros_all); match goal with
    | b : bool ⊢ _ ⇒ destruct b; clear b; aux tt
    | _ ⇒ tautob_post tt
  end in
  aux tt.

```

```

Tactic Notation "tautob" :=
  tautob_core tt.

```

6.4 Properties of boolean operators

6.4.1 Properties of *eqb*

```

Lemma eqb_same : ∀ x,
  eqb x x = true.

```

Proof using. *tautob*. Qed.

```

Lemma eqb_true_l : neutral_l eqb true.

```

Proof using. *tautob*. Qed.

```

Lemma eqb_true_r : neutral_r eqb true.

```

Proof using. *tautob*. Qed.

Lemma eqb_false_l : $\forall x$,
eqb false x = neg x .
Proof using. *tautob*. Qed.

Lemma eqb_false_r : $\forall x$,
eqb x false = neg x .
Proof using. *tautob*. Qed.

Lemma eqb_comm : comm eqb.
Proof using. *tautob*. Qed.

6.4.2 Properties of and

Lemma and_same : idempotent2 and.
Proof using. *tautob*. Qed.

Lemma and_true_l : neutral_l and true.
Proof using. *tautob*. Qed.

Lemma and_true_r : neutral_r and true.
Proof using. *tautob*. Qed.

Lemma and_false_l : absorb_l and false.
Proof using. *tautob*. Qed.

Lemma and_false_r : absorb_r and false.
Proof using. *tautob*. Qed.

Lemma and_comm : comm and.
Proof using. *tautob*. Qed.

Lemma and_assoc : assoc and.
Proof using. *tautob*. Qed.

Lemma and_or_l : distrib_r and or.
Proof using. *tautob*. Qed.

Lemma and_or_r : distrib_l and or.
Proof using. *tautob*. Qed.

6.4.3 Properties of or

Lemma or_same : idempotent2 or.
Proof using. *tautob*. Qed.

Lemma or_false_l : neutral_l or false.
Proof using. *tautob*. Qed.

Lemma or_false_r : neutral_r or false.
Proof using. *tautob*. Qed.

Lemma `or_true_l` : `absorb_l` or true.
 Proof using. *tautob*. Qed.
 Lemma `or_true_r` : `absorb_r` or true.
 Proof using. *tautob*. Qed.
 Lemma `or_comm` : `comm` or.
 Proof using. *tautob*. Qed.
 Lemma `or_assoc` : `assoc` or.
 Proof using. *tautob*. Qed.
 Lemma `or_and_l` : `distrib_r` or and.
 Proof using. *tautob*. Qed.
 Lemma `or_and_r` : `distrib_l` or and.
 Proof using. *tautob*. Qed.

6.4.4 Properties of `neg`

Lemma `neg_false` : ! `false` = true.
 Proof using. `auto`. Qed.
 Lemma `neg_true` : ! `true` = false.
 Proof using. `auto`. Qed.
 Lemma `neg_and` : automorphism `neg` and or.
 Proof using. *tautob*. Qed.
 Lemma `neg_or` : automorphism `neg` or and.
 Proof using. *tautob*. Qed.
 Lemma `neg_neg` : involutive `neg`.
 Proof using. *tautob*. Qed.

6.4.5 Properties of `if then else`

Section `PropertiesIf`.
 Implicit Types *x y z* : `bool`.
 Lemma `if_true` : $\forall x y,$
 $(\text{if true then } x \text{ else } y) = x.$
 Proof using. `auto`. Qed.
 Lemma `if_false` : $\forall x y,$
 $(\text{if false then } x \text{ else } y) = y.$
 Proof using. `auto`. Qed.
 Lemma `if_then_else_same` : $\forall x y,$
 $(\text{if } x \text{ then } y \text{ else } y) = y.$

```

Proof using. tautob. Qed.
Lemma if_then_true_else_false :  $\forall x$ ,
  (if x then true else false) = x.
Proof using. tautob. Qed.
Lemma if_then_false_else_true :  $\forall x$ ,
  (if x then false else true) = !x.
Proof using. tautob. Qed.
Lemma if_then_true :  $\forall x\ y$ ,
  (if x then true else y) = x || y.
Proof using. tautob. Qed.
Lemma if_then_false :  $\forall x\ y$ ,
  (if x then false else y) = (!x) && y.
Proof using. tautob. Qed.
Lemma if_else_false :  $\forall x\ y$ ,
  (if x then y else false) = x && y.
Proof using. tautob. Qed.
Lemma if_else_true :  $\forall x\ y$ ,
  (if x then y else true) = (!x) || y.
Proof using. tautob. Qed.
End PropertiesIf.

```

6.4.6 Properties of **impl** and **xor**

We do not provide lemmas for **impl** and **xor** because these functions can be easily expressed in terms of the other operators.

6.4.7 Opacity

Opaque eqb neg and or.

6.5 Tactics

6.5.1 Tactic *rew_neg_neg*

rew_neg_neg is a tactic that simplifies all double negations of booleans, i.e. replaces !!*b* with *b*.

```

#[global]
Hint Rewrite neg_neg : rew_neg_neg.
Tactic Notation "rew_neg_neg" :=

```

```

    autorewrite with rew_neg_neg.
Tactic Notation "rew_neg_neg" "~" :=
    rew_neg_neg; auto_tilde.
Tactic Notation "rew_neg_neg" "*" :=
    rew_neg_neg; auto_star.
Tactic Notation "rew_neg_neg" "in" hyp(H) :=
    autorewrite with rew_neg_neg in H.
Tactic Notation "rew_neg_neg" "~" "in" hyp(H) :=
    rew_neg_neg in H; auto_tilde.
Tactic Notation "rew_neg_neg" "*" "in" hyp(H) :=
    rew_neg_neg in H; auto_star.
Tactic Notation "rew_neg_neg" "in" "*" :=
    autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_neg_neg).
Tactic Notation "rew_neg_neg" "~" "in" "*" :=
    rew_neg_neg in *; auto_tilde.
Tactic Notation "rew_neg_neg" "*" "in" "*" :=
    rew_neg_neg in *; auto_star.

```

6.5.2 Tactic *rew_bool*

rew_bool simplifies boolean expressions, using rewriting lemmas in the database *rew_bool* defined below.

#[*global*]

Hint Rewrite

```

    eqb_same eqb_true_l eqb_true_r eqb_false_l eqb_false_r
    neg_false neg_true neg_neg neg_and neg_or
    and_true_l and_true_r and_false_l and_false_r
    or_false_l or_false_r or_true_l or_true_r
    if_true if_false if_then_else_same
    if_then_true_else_false if_then_false_else_true
    if_then_true if_else_false
    if_then_false if_else_true
    : rew_bool.

```

```

Tactic Notation "rew_bool" :=
    autorewrite with rew_bool.
Tactic Notation "rew_bool" "~" :=
    rew_bool; auto_tilde.
Tactic Notation "rew_bool" "*" :=
    rew_bool; auto_star.
Tactic Notation "rew_bool" "in" hyp(H) :=
    autorewrite with rew_bool in H.
Tactic Notation "rew_bool" "~" "in" hyp(H) :=

```

```

    rew_bool in H; auto_tilde.
Tactic Notation "rew_bool" "*" "in" hyp(H) :=
    rew_bool in H; auto_star.
Tactic Notation "rew_bool" "in" "*" :=
    autorewrite_in_star_patch ltac:(fun tt ⇒ autorewrite with rew_bool).
Tactic Notation "rew_bool" "~" "in" "*" :=
    rew_bool in *; auto_tilde.
Tactic Notation "rew_bool" "*" "in" "*" :=
    rew_bool in *; auto_star.

```

Chapter 7

Library SLF.LibReflect

```
Set Implicit Arguments.
From SLF Require Import LibTactics.
From SLF Require Export LibBool LibLogic.

Implicit Type P : Prop.
Implicit Type b : bool.
```

7.1 Reflection between booleans and propositions

- `istrue b` produces a proposition that is **True** if and only if the boolean `b` is equal to `true`.
 - `isTrue P` produces a boolean expression that is `true` if and only if the proposition `P` is equal to **True**.

7.1.1 Translation from booleans into propositions

Any boolean `b` can be viewed as a proposition through the relation `b = true`.

```
Coercion istrue (b : bool) : Prop := (b = true).
```

Specification

```
Lemma istrue_eq_eq_true : ∀ b,
  istrue b = (b = true).
```

Proof using. reflexivity. Qed.

```
Lemma istrue_true_eq :
  istrue true = True.
```

Proof using. rewrite istrue_eq_eq_true. *extends*×. Qed.

```
Lemma istrue_false_eq :
```

```

    istrue false = False.
Proof using. rewrite istrue_eq_eq_true. extens. iff; auto_false. Qed.
Global Opaque istrue.

    Proving the goals true and  $\neg$  false
Lemma istrue_true : istrue true. Proof using. reflexivity. Qed.
Lemma not_istrue_false :  $\neg$  (istrue false). Proof using. rewrite istrue_false_eq. intuition.
Qed.

    Equivalence of false and False
Lemma false_of_False :
    False  $\rightarrow$ 
    false.
Proof using. intros K. false. Qed.
Lemma False_of_false :
    false  $\rightarrow$ 
    False.
Proof using. intros K. rewrite  $\neg$  istrue_false_eq in K. Qed.

    Hints for proving false and False
#[global]
Hint Resolve istrue_true not_istrue_false.
#[global]
Hint Extern 1 (istrue false)  $\Rightarrow$ 
    apply false_of_False.
#[global]
Hint Extern 1 (False)  $\Rightarrow$  match goal with
| H: istrue false  $\vdash$  _  $\Rightarrow$  apply (not_istrue_false H) end.

```

7.1.2 Translation from propositions into booleans

The expression `isTrue P` evaluates to `true` if and only if the proposition *P* is **True**.

```

Definition isTrue (P : Prop) : bool :=
    If P then true else false.

```

Specification

```

Lemma isTrue_eq_if :  $\forall$  P,
    isTrue P = If P then true else false.
Proof using. reflexivity. Qed.

Lemma isTrue_True :
    isTrue True = true.
Proof using. unfolds. case_if; auto_false  $\neg$ . Qed.

```



```

Lemma isTrue_False :
  isTrue False = false.
Proof using. unfolds. case-if; auto_false¬. Qed.

Global Opaque isTrue.

Lemmas

Lemma isTrue_eq_true : ∀ P,
  P →
  isTrue P = true.
Proof using. intros. rewrite isTrue_eq_if. case-if×. Qed.

Lemma isTrue_eq_false : ∀ P,
  ¬ P →
  isTrue P = false.
Proof using. intros. rewrite isTrue_eq_if. case-if×. Qed.

```

7.1.3 Extensionality for boolean equality, stated using **istrue**

```

Lemma bool_ext : ∀ b1 b2,
  (b1 ↔ b2) →
  b1 = b2.
Proof using.
  destruct b1; destruct b2; intros; auto_false.
  destruct H. false H; auto.
  destruct H. false H0; auto.
Qed.

Lemma bool_ext_eq : ∀ b1 b2,
  (b1 = b2) = (b1 ↔ b2).
Proof using.
  intros. extens. iff M. { subst×. } { applies× bool_ext. }
Qed.

#[global]
Instance Extensionality_bool : Extensionality bool.
Proof using. apply (Extensionality_make bool_ext). Defined.

```

7.1.4 Specification of boolean equality

`isTrue_pred p P` asserts that `p` is a boolean function that implements the predicate *P*, in the sense that `p x` returns `true` iff *P* `x` holds.

```

Definition isTrue_pred A (p:A→bool) (P:A→Prop) : Prop :=
  ∀ x, p x = isTrue (P x).

```

`is_beq f` asserts that `f` is a boolean comparison function that implements logical equality:
`f x y` returns `true` iff `x = y`.

Definition `is_beq A (f:A→A→bool) :=`
 $\forall x\ y, f\ x\ y = \text{isTrue}\ (x = y).$

7.2 Rewriting rules

7.2.1 Rewriting rules for distributing `istrue`

Lemma `istrue_isTrue_eq` : $\forall P,$
 $\text{istrue}\ (\text{isTrue}\ P) = P.$

Proof using. *extens. rewrite isTrue_eq_if. case_if; auto_false*×. Qed.

Lemma `istrue_neg_eq` : $\forall b,$
 $\text{istrue}\ (!b) = \neg (\text{istrue}\ b).$

Proof using. *extens. tautob.* Qed.

Lemma `istrue_and_eq` : $\forall b1\ b2,$
 $\text{istrue}\ (b1 \ \&\&\ b2) = (\text{istrue}\ b1 \wedge \text{istrue}\ b2).$

Proof using. *extens. tautob.* Qed.

Lemma `istrue_or_eq` : $\forall b1\ b2,$
 $\text{istrue}\ (b1 \ ||\ b2) = (\text{istrue}\ b1 \vee \text{istrue}\ b2).$

Proof using. *extens. tautob.* Qed.

Corollary

Lemma `istrue_neg_isTrue` : $\forall P,$
 $\text{istrue}\ (!\ \text{isTrue}\ P) = \neg P.$

Proof using. *intros. rewrite istrue_neg_eq. rewrite¬ istrue_isTrue_eq.* Qed.

`istrue` and conditionals

Lemma `If_istrue` : $\forall b\ A\ (x\ y : A),$
 $(\text{If}\ \text{istrue}\ b\ \text{then}\ x\ \text{else}\ y)$
 $= (\text{if}\ b\ \text{then}\ x\ \text{else}\ y).$

Proof using. *intros. case_if as C; case_if as D; auto.* Qed.

Lemma `istrue_If_eq` : $\forall P\ b1\ b2,$
 $\text{istrue}\ (\text{If}\ P\ \text{then}\ b1\ \text{else}\ b2)$
 $= (\text{If}\ P\ \text{then}\ \text{istrue}\ b1\ \text{else}\ \text{istrue}\ b2).$

Proof using. *extens. case_if*×. Qed.

Lemma `istrue_if_eq` : $\forall b1\ b2\ b3,$
 $\text{istrue}\ (\text{if}\ b1\ \text{then}\ b2\ \text{else}\ b3)$
 $= (\text{If}\ \text{istrue}\ b1\ \text{then}\ \text{istrue}\ b2\ \text{else}\ \text{istrue}\ b3).$

Proof using. *intros. do 2 case_if; auto.* Qed.

7.2.2 Rewriting rules for distributing `isTrue`

Lemma `isTrue_istrue` : $\forall b,$

`isTrue (istrue b) = b.`

Proof using. *extens*. `rewrite`× `istrue_isTrue_eq`. Qed.

Lemma `isTrue_not` : $\forall P,$

`isTrue ( $\neg P$ ) = ! isTrue P.`

Proof using. *extens*. `do 2 rewrite isTrue_eq_if`. `do 2 case_if; auto_false`×. Qed.

Lemma `isTrue_and` : $\forall P1\ P2,$

`isTrue ( $P1 \wedge P2$ ) = (isTrue P1 && isTrue P2).`

Proof using. *extens*. `do 3 rewrite isTrue_eq_if`. `do 3 case_if; auto_false`×. Qed.

Lemma `isTrue_or` : $\forall P1\ P2,$

`isTrue ( $P1 \vee P2$ ) = (isTrue P1 || isTrue P2).`

Proof using. *extens*. `do 3 rewrite isTrue_eq_if`. `do 3 case_if; auto_false`×. Qed.

Corollary

Lemma `isTrue_not_istrue` : $\forall b,$

`isTrue ( $\neg$  istrue b) = !b.`

Proof using. `intros`. `rewrite isTrue_not`. `rewrite`¬ `isTrue_istrue`. Qed.

Simplification of equalities involving `isTrue`

Section `IsTrueEqualities`.

Ltac *prove_isTrue_lemma* :=

`intros; try extens; try iff; rewrite isTrue_eq_if in *; case_if; auto_false`×.

Lemma `true_eq_isTrue_eq` : $\forall P,$

`(true = isTrue P) = P.`

Proof using. *prove_isTrue_lemma*. Qed.

Lemma `isTrue_eq_true_eq` : $\forall P,$

`(isTrue P = true) = P.`

Proof using. *prove_isTrue_lemma*. Qed.

Lemma `false_eq_isTrue_eq` : $\forall P,$

`(false = isTrue P) =  $\neg P$ .`

Proof using. *prove_isTrue_lemma*. Qed.

Lemma `isTrue_eq_false_eq` : $\forall P,$

`(isTrue P = false) =  $\neg P$ .`

Proof using. *prove_isTrue_lemma*. Qed.

Lemma `isTrue_eq_isTrue_eq` : $\forall P1\ P2,$

`(isTrue P1 = isTrue P2) = ( $P1 \leftrightarrow P2$ ).`

Proof using.

`intros. extens. iff; repeat rewrite isTrue_eq_if in *;`
`repeat case_if; auto_false`×.

Qed.

End IsTrueEqualities.

isTrue and conditionals

Lemma if_isTrue : $\forall P A (x y : A)$,
 (if isTrue P then x else y)
 = (If P then x else y).

Proof using.

 intros. *case_if* as C ; *case_if* as D ; auto.
 { rewrite $\times$ isTrue_eq_true_eq in C . }
 { rewrite $\times$ isTrue_eq_false_eq in C . }

Qed.

Lemma isTrue_If : $\forall P1 P2 P3$,
 isTrue (If $P1$ then $P2$ else $P3$)
 = If $P1$ then isTrue $P2$ else isTrue $P3$.

Proof using. *extens. case_if* $\times$. Qed.

Lemma isTrue_If_eq_if_isTrue : $\forall P1 P2 P3$,
 isTrue (If $P1$ then $P2$ else $P3$)
 = (if isTrue $P1$ then isTrue $P2$ else isTrue $P3$).

Proof using. intros. rewrite if_isTrue. rewrite $\neg$ isTrue_If. Qed.

7.2.3 Lemmas for testing booleans

Lemma bool_inv_or : $\forall b$,
 $b \vee !b$.

Proof using. *tautob*. Qed.

Lemma bool_inv_or_eq : $\forall b$,
 $b = \text{true} \vee b = \text{false}$.

Proof using. *tautob*. Qed.

Lemma xor_inv_or : $\forall b1 b2$,
 xor $b1 b2 \rightarrow$
 ($b1 = \text{true} \wedge b2 = \text{false}$)
 $\vee (b1 = \text{false} \wedge b2 = \text{true})$.

Proof using. *tautob*; *auto_false* $\times$. Qed.

Arguments xor_inv_or [$b1$] [$b2$].

7.2.4 Lemmas for normalizing $b = \text{true}$ and $b = \text{false}$ terms

Lemma bool_eq_true_eq : $\forall b$,
 ($b = \text{true}$) = isTrue b .

Proof using. *extens. tautob*. Qed.

```

Lemma bool_eq_false_eq :  $\forall b$ ,
  ( $b = \text{false}$ ) = istrue ( $!b$ ).
Proof using. extens. tautob. Qed.

Lemma true_eq_bool_eq :  $\forall b$ ,
  ( $\text{true} = b$ ) = istrue  $b$ .
Proof using. extens. tautob. Qed.

Lemma false_eq_bool_eq :  $\forall b$ ,
  ( $\text{false} = b$ ) = istrue ( $!b$ ).
Proof using. extens. tautob. Qed.

```

7.3 Tactics

7.3.1 Tactics *rew_istrue* to distribute *istrue*

rew_istrue distributes *istrue*. It is useful to replace all boolean operators with corresponding logical operators.

```

#[global]
Hint Rewrite istrue_true_eq istrue_false_eq istrue_isTrue_eq
  istrue_neg_eq istrue_and_eq istrue_or_eq
  If_istrue istrue_If_eq istrue_if_eq: rew_istrue.

Tactic Notation "rew_istrue" :=
  autorewrite with rew_istrue.

Tactic Notation "rew_istrue" "in" hyp(H) :=
  autorewrite with rew_istrue in  $H$ .

Tactic Notation "rew_istrue" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun  $tt \Rightarrow$  autorewrite with rew_istrue).

Tactic Notation "rew_istrue" "~" :=
  rew_istrue; auto_tilde.

Tactic Notation "rew_istrue" "~" "in" hyp(H) :=
  rew_istrue in  $H$ ; auto_tilde.

Tactic Notation "rew_istrue" "~" "in" "*" :=
  rew_istrue in *; auto_tilde.

Tactic Notation "rew_istrue" "*" :=
  rew_istrue; auto_star.

Tactic Notation "rew_istrue" "*" "in" hyp(H) :=
  rew_istrue in  $H$ ; auto_star.

Tactic Notation "rew_istrue" "*" "in" "*" :=
  rew_istrue in *; auto_star.

```

7.3.2 Tactics *rew_isTrue* to distribute *isTrue*

rew_isTrue distributes *isTrue*. This tactic is probably much less useful than *rew_istrue*, since logical operators are often simpler to work with.

```
#[global]
```

```
Hint Rewrite isTrue_True isTrue_False isTrue_istrue
  isTrue_not isTrue_and isTrue_or
  if_isTrue isTrue_If : rew_isTrue.
```

```
Tactic Notation "rew_isTrue" :=
  autorewrite with rew_isTrue.
```

```
Tactic Notation "rew_isTrue" "in" hyp(H) :=
  autorewrite with rew_isTrue in H.
```

```
Tactic Notation "rew_isTrue" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_isTrue).
```

```
Tactic Notation "rew_isTrue" "~" :=
  rew_isTrue; auto_tilde.
```

```
Tactic Notation "rew_isTrue" "~" "in" hyp(H) :=
  rew_isTrue in H; auto_tilde.
```

```
Tactic Notation "rew_isTrue" "~" "in" "*" :=
  rew_isTrue in *; auto_tilde.
```

```
Tactic Notation "rew_isTrue" "*" :=
  rew_isTrue; auto_star.
```

```
Tactic Notation "rew_isTrue" "*" "in" hyp(H) :=
  rew_isTrue in H; auto_star.
```

```
Tactic Notation "rew_isTrue" "*" "in" "*" :=
  rew_isTrue in *; auto_star.
```

7.3.3 Tactics useful for program verification, when reasoning about

the result of if-statements over boolean expressions, i.e. an expression of the form *b* = .. or .. = *b*, which produces hypotheses of the form *true* = .. and *false* = .. or symmetric. It is used as post-treatment for tactic *case_if*.

```
#[global]
```

```
Hint Rewrite
```

```
  true_eq_isTrue_eq isTrue_eq_true_eq
  false_eq_isTrue_eq isTrue_eq_false_eq
  isTrue_eq_isTrue_eq
  not_not_eq
  istrue_true_eq istrue_false_eq istrue_isTrue_eq
  istrue_neg_eq istrue_and_eq istrue_or_eq
  bool_eq_true_eq bool_eq_false_eq true_eq_bool_eq false_eq_bool_eq
  : rew_bool_eq.
```

```
Tactic Notation "rew_bool_eq" :=
```

```

    autorewrite with rew_bool_eq.
Tactic Notation "rew_bool_eq" "~" :=
    rew_bool_eq; auto_tilde.
Tactic Notation "rew_bool_eq" "*" :=
    rew_bool_eq; auto_star.

Tactic Notation "rew_bool_eq" "in" hyp(H) :=
    autorewrite with rew_bool_eq in H.
Tactic Notation "rew_bool_eq" "~" "in" hyp(H) :=
    rew_bool_eq in H; auto_tilde.
Tactic Notation "rew_bool_eq" "*" "in" hyp(H) :=
    rew_bool_eq in H; auto_star.

Tactic Notation "rew_bool_eq" "in" "*" :=
    autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_bool_eq).
Tactic Notation "rew_bool_eq" "~" "in" "*" :=
    rew_bool_eq in *; auto_tilde.
Tactic Notation "rew_bool_eq" "*" "in" "*" :=
    rew_bool_eq in *; auto_star.

```

7.3.4 Tactics extended for reflection

Extension of the tactic *case_if* to automatically performs simplification using *logics*.

For less aggressive introduction of *istrue*, consider rewriting without the lemmas: *bool_eq_true_eq* *bool_eq_false_eq* *true_eq_bool_eq* *false_eq_bool_eq*

```

Ltac case_if_post H ::=
    rew_bool_eq in H; tryfalse.

```

Extension of the tactic *test_dispatch* from *LibLogic.v*, so as to be able to call the tactic *tests* directly on boolean expressions

```

Ltac tests_bool_base E H1 H2 :=
    tests_prop_base (istrue E) H1 H2.

Ltac tests_dispatch E H1 H2 ::=
    match type of E with
    | bool => tests_bool_base E H1 H2
    | Prop => tests_prop_base E H1 H2
    | { _ } + { _ } => tests_ssum_base E H1 H2
    end.

```

Extension of the tactic *apply_to_head_of* (see *LibTactics*).

```

Ltac apply_to_head_of E cont ::=
    let go E := let P := get_head E in cont P in
    match E with
    | istrue ?A => go A

```

```
| istrue (neg ?A) ⇒ go A
| ?A = ?B ⇒ first [ go A | go B ]
| ?A ⇒ go A
end.
```


Chapter 8

Library SLF.LibProd

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibLogic LibReflect.
Generalizable Variables A B.
```

8.1 Product type

8.1.1 Definition

From the Prelude:

```
Inductive prod A B : Type := | pair : A -> B -> prod A B.
Hint Constructors prod : core.
Add Printing Let prod. Notation "x * y" := (prod x y) : type_scope. Notation "( x , y ,
.. , z )" := (pair .. (pair x y) .. z) : core_scope.
Definition fst A B (p:A*B) : A := match p with (x,y) => x end.
Definition snd A B (p:A*B) : B := match p with (x,y) => y end.
Remark: to follow conventions pair should be renamed to prod_intro.
```

8.1.2 Inhabited

```
Global Instance Inhab_prod :  $\forall$  ' { Inhab A, Inhab B }, Inhab (A  $\times$  B).
Proof using. intros. apply (Inhab_of_val (arbitrary, arbitrary)). Qed.
```

8.2 Structure

Decomposition as projection

```
Lemma prod2_eq_tuple_proj :  $\forall$  A1 A2 (x:A1  $\times$  A2),
  x = (fst x, snd x).
```

Proof using. intros. destruct $\neg$ x . Qed.

Structural equality

Section Properties.

Variables ($A1$ $A2$ $A3$ $A4$: Type).

Lemma eq_prod2 : $\forall$ ($x1$ $y1:A1$) ($x2$ $y2:A2$),

$x1 = y1 \rightarrow$

$x2 = y2 \rightarrow$

$(x1, x2) = (y1, y2)$.

Proof using. intros. subst $\neg$. Qed.

Lemma eq_prod3 : $\forall$ ($x1$ $y1:A1$) ($x2$ $y2:A2$) ($x3$ $y3:A3$),

$x1 = y1 \rightarrow$

$x2 = y2 \rightarrow$

$x3 = y3 \rightarrow$

$(x1, x2, x3) = (y1, y2, y3)$.

Proof using. intros. subst $\neg$. Qed.

Lemma eq_prod4 : $\forall$ ($x1$ $y1:A1$) ($x2$ $y2:A2$) ($x3$ $y3:A3$) ($x4$ $y4:A4$),

$x1 = y1 \rightarrow$

$x2 = y2 \rightarrow$

$x3 = y3 \rightarrow$

$x4 = y4 \rightarrow$

$(x1, x2, x3, x4) = (y1, y2, y3, y4)$.

Proof using. intros. subst $\neg$. Qed.

End Properties.

#[*global*]

Hint Immediate eq_prod2 eq_prod3 eq_prod4.

8.3 Operations

8.3.1 Definition of projections

`fst` and `snd` are defined in the Prelude

Definition fst A B (p:A*B) : A := match p with (x,y) => x end.

Definition snd A B (p:A*B) : B := match p with (x,y) => y end.

Arguments fst { A } { B }.

Arguments snd { A } { B }.

8.3.2 Notation for projections

N-ary projections are defined as notations and not as definitions, which has appeared to be more flexible with respect to type inference. TODO: investigate the possibility of using

definitions.

```
Notation "'proj11' P" := P (at level 69, only parsing).
Notation "'proj21' P" := (proj1 P) (at level 69, only parsing).
Notation "'proj22' P" := (proj2 P) (at level 69, only parsing).
Notation "'proj31' P" := (proj1 P) (at level 69).
Notation "'proj32' P" := (proj1 (proj2 P)) (at level 69).
Notation "'proj33' P" := (proj2 (proj2 P)) (at level 69).
Notation "'proj41' P" := (proj1 P) (at level 69).
Notation "'proj42' P" := (proj1 (proj2 P)) (at level 69).
Notation "'proj43' P" := (proj1 (proj2 (proj2 P))) (at level 69).
Notation "'proj44' P" := (proj2 (proj2 (proj2 P))) (at level 69).
Notation "'proj51' P" := (proj1 P) (at level 69).
Notation "'proj52' P" := (proj1 (proj2 P)) (at level 69).
Notation "'proj53' P" := (proj1 (proj2 (proj2 P))) (at level 69).
Notation "'proj54' P" := (proj1 (proj2 (proj2 (proj2 P)))) (at level 69).
Notation "'proj55' P" := (proj2 (proj2 (proj2 (proj2 P)))) (at level 69).
```

8.3.3 Currying

Section Currying.

Variables ($A1\ A2\ A3\ A4\ A5\ B : \text{Type}$).

Definition curry1 $f : A1 \rightarrow B :=$

f .

Definition curry2 $f : A1 \rightarrow A2 \rightarrow B :=$

$\text{fun } x1\ x2 \Rightarrow f\ (x1, x2)$.

Definition curry3 $f : A1 \rightarrow A2 \rightarrow A3 \rightarrow B :=$

$\text{fun } x1\ x2\ x3 \Rightarrow f\ (x1, x2, x3)$.

Definition curry4 $f : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow B :=$

$\text{fun } x1\ x2\ x3\ x4 \Rightarrow f\ (x1, x2, x3, x4)$.

Definition curry5 $f : A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow B :=$

$\text{fun } x1\ x2\ x3\ x4\ x5 \Rightarrow f\ (x1, x2, x3, x4, x5)$.

End Currying.

8.3.4 Uncurrying

Section Uncurrying.

Variables ($A1\ A2\ A3\ A4\ A5\ B : \text{Type}$).

Definition uncurry1 $f : A1 \rightarrow B :=$

f .

Definition uncurry2 $f : A1 \times A2 \rightarrow B :=$

$\text{fun } p \Rightarrow \text{match } p \text{ with } (x1, x2) \Rightarrow$

```

  f x1 x2 end.
Definition uncurry3 f : A1 × A2 × A3 → B :=
  fun p ⇒ match p with (x1, x2, x3) ⇒
    f x1 x2 x3 end.
Definition uncurry4 f : A1 × A2 × A3 × A4 → B :=
  fun p ⇒ match p with (x1, x2, x3, x4) ⇒
    f x1 x2 x3 x4 end.
Definition uncurry5 f : A1 × A2 × A3 × A4 × A5 → B :=
  fun p ⇒ match p with (x1, x2, x3, x4, x5) ⇒
    f x1 x2 x3 x4 x5 end.
End Uncurrying.

```

8.3.5 Uncurrying for relations

uncurrypN turns a function of type $A1 \rightarrow A1 \rightarrow \dots \rightarrow AN \rightarrow AN \rightarrow B$ into a function of type $(A1^{*..*}AN) \rightarrow (A1^{*..*}AN) \rightarrow B$.

Section Uncurryp.

Variables (*A1 A2 A3 A4 B* : Type).

```

Definition uncurryp1 f : A1 → A1 → B :=
  f.

```

```

Definition uncurryp2 f : A1 × A2 → A1 × A2 → B :=
  fun p1 p2 ⇒ match p1, p2 with (x1, x2), (y1, y2) ⇒
    f x1 y1 x2 y2 end.

```

```

Definition uncurryp3 f : A1 × A2 × A3 → A1 × A2 × A3 → B :=
  fun p1 p2 ⇒ match p1, p2 with (x1, x2, x3), (y1, y2, y3) ⇒
    f x1 x2 x3 y1 y2 y3 end.

```

```

Definition uncurryp4 f : A1 × A2 × A3 × A4 → A1 × A2 × A3 × A4 → B :=
  fun p1 p2 ⇒ match p1, p2 with (x1, x2, x3, x4), (y1, y2, y3, y4) ⇒
    f x1 x2 x3 x4 y1 y2 y3 y4 end.

```

End Uncurryp.

Unfolding

```

Tactic Notation "unfold_uncurryp" :=
  unfold uncurryp1, uncurryp2, uncurryp3, uncurryp4.

```

```

Tactic Notation "unfolds_uncurryp" :=
  unfold uncurryp1, uncurryp2, uncurryp3, uncurryp4 in *.

```

8.3.6 Inverse projections for relations

unprojNK turns a function of type $AK \rightarrow AK \rightarrow B$ into a function of type $(A1^{*..*}AN) \rightarrow (A1^{*..*}AN) \rightarrow B$.

Section Unproj.

Variables ($A1\ A2\ A3\ A4\ A5\ B : \text{Type}$).

Definition unproj21 $f : A1 \times A2 \rightarrow A1 \times A2 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2), (y1, y2) \Rightarrow$
 $f\ x1\ y1$ end.

Definition unproj22 $f : A1 \times A2 \rightarrow A1 \times A2 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2), (y1, y2) \Rightarrow$
 $f\ x2\ y2$ end.

Definition unproj31 $f : A1 \times A2 \times A3 \rightarrow A1 \times A2 \times A3 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3), (y1, y2, y3) \Rightarrow$
 $f\ x1\ y1$ end.

Definition unproj32 $f : A1 \times A2 \times A3 \rightarrow A1 \times A2 \times A3 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3), (y1, y2, y3) \Rightarrow$
 $f\ x2\ y2$ end.

Definition unproj33 $f : A1 \times A2 \times A3 \rightarrow A1 \times A2 \times A3 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3), (y1, y2, y3) \Rightarrow$
 $f\ x3\ y3$ end.

Definition unproj41 $f : A1 \times A2 \times A3 \times A4 \rightarrow A1 \times A2 \times A3 \times A4 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4), (y1, y2, y3, y4) \Rightarrow$
 $f\ x1\ y1$ end.

Definition unproj42 $f : A1 \times A2 \times A3 \times A4 \rightarrow A1 \times A2 \times A3 \times A4 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4), (y1, y2, y3, y4) \Rightarrow$
 $f\ x2\ y2$ end.

Definition unproj43 $f : A1 \times A2 \times A3 \times A4 \rightarrow A1 \times A2 \times A3 \times A4 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4), (y1, y2, y3, y4) \Rightarrow$
 $f\ x3\ y3$ end.

Definition unproj44 $f : A1 \times A2 \times A3 \times A4 \rightarrow A1 \times A2 \times A3 \times A4 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4), (y1, y2, y3, y4) \Rightarrow$
 $f\ x4\ y4$ end.

Definition unproj51 $f : A1 \times A2 \times A3 \times A4 \times A5 \rightarrow A1 \times A2 \times A3 \times A4 \times A5 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4, x5), (y1, y2, y3, y4, y5) \Rightarrow$
 $f\ x1\ y1$ end.

Definition unproj52 $f : A1 \times A2 \times A3 \times A4 \times A5 \rightarrow A1 \times A2 \times A3 \times A4 \times A5 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4, x5), (y1, y2, y3, y4, y5) \Rightarrow$
 $f\ x2\ y2$ end.

Definition unproj53 $f : A1 \times A2 \times A3 \times A4 \times A5 \rightarrow A1 \times A2 \times A3 \times A4 \times A5 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4, x5), (y1, y2, y3, y4, y5) \Rightarrow$
 $f\ x3\ y3$ end.

Definition unproj54 $f : A1 \times A2 \times A3 \times A4 \times A5 \rightarrow A1 \times A2 \times A3 \times A4 \times A5 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4, x5), (y1, y2, y3, y4, y5) \Rightarrow$
 $f\ x4\ y4$ end.

Definition unproj55 $f : A1 \times A2 \times A3 \times A4 \times A5 \rightarrow A1 \times A2 \times A3 \times A4 \times A5 \rightarrow B :=$
fun $p1\ p2 \Rightarrow$ match $p1, p2$ with $(x1, x2, x3, x4, x5), (y1, y2, y3, y4, y5) \Rightarrow$

f x5 y5 end.

End Unproj.

Arguments unproj21 [*A1*] *A2* [*B*].
Arguments unproj22 *A1* [*A2*] [*B*].
Arguments unproj31 [*A1*] *A2* *A3* [*B*].
Arguments unproj32 *A1* [*A2*] *A3* [*B*].
Arguments unproj33 *A1* *A2* [*A3*] [*B*].
Arguments unproj41 [*A1*] *A2* *A3* *A4* [*B*].
Arguments unproj42 *A1* [*A2*] *A3* *A4* [*B*].
Arguments unproj43 *A1* *A2* [*A3*] *A4* [*B*].
Arguments unproj44 *A1* *A2* *A3* [*A4*] [*B*].
Arguments unproj51 [*A1*] *A2* *A3* *A4* *A5* [*B*].
Arguments unproj52 *A1* [*A2*] *A3* *A4* *A5* [*B*].
Arguments unproj53 *A1* *A2* [*A3*] *A4* *A5* [*B*].
Arguments unproj54 *A1* *A2* *A3* [*A4*] *A5* [*B*].
Arguments unproj55 *A1* *A2* *A3* *A4* [*A5*] [*B*].

Unfolding

Tactic Notation "unfold_unproj" :=
 unfold unproj21, unproj22, unproj31, unproj32, unproj33,
 unproj41, unproj42, unproj43, unproj44,
 unproj51.

Tactic Notation "unfolds_unproj" :=
 unfold unproj21, unproj22, unproj31, unproj32, unproj33,
 unproj41, unproj42, unproj43, unproj44,
 unproj51 in *.

Chapter 9

Library SLF.LibSum

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibLogic LibBool.
Generalizable Variables A B.
```

9.1 Sum type

9.1.1 Definition

From the Prelude:

Inductive sum A B : Type := | inl : A -> sum A B | inr : B -> sum A B.

Hint Constructors sum : core. Notation “x + y” := (sum x y) : type_scope.

Remark: ideally, constructors would be renamed to *sum_l* and *sum_r*; to follow conventions.

```
Arguments inl {A} {B}.
Arguments inr {A} {B}.
```

9.1.2 Inhabited

```
#[global]
Instance sum_inhab_l : ∀ ‘{Inhab A} B, Inhab (A + B).
Proof using. intros. apply (Inhab_of_val (inl arbitrary)). Qed.

#[global]
Instance sum_inhab_r : ∀ ‘{Inhab B} A, Inhab (A + B).
Proof using. intros. apply (Inhab_of_val (inr arbitrary)). Qed.

Definition Inhab_sum : ∀ ‘{Inhab A, Inhab B}, Inhab (A + B).
Proof using. typeclass. Qed.
```

9.2 Operations

9.2.1 Testing the branch of the sum

```
Definition is_inl (A B : Type) (x : A + B) : bool :=
  match x with
  | inl _ => true
  | inr _ => false
  end.
```

```
Definition is_inr (A B : Type) (x : A + B) : bool :=
  match x with
  | inl _ => false
  | inr _ => true
  end.
```

Section IsIn.

Variables (A B : Type).

Implicit Type x : A + B.

Lemma is_inl_neg_is_inr : $\forall x$,
is_inl x = ! (is_inr x).

Proof using. intros x. destruct $\neg$ x. Qed.

Lemma is_inr_neg_is_inl : $\forall x$,
is_inr x = ! (is_inl x).

Proof using. intros x. destruct $\neg$ x. Qed.

End IsIn.

9.2.2 Stripping of the branch tag

Section Get.

Context '{IA1:Inhab A1} '{IA2:Inhab A2}.

Implicit Types x : A1+A2.

```
Definition get21 x :=
  match x with
  | inl x1 => x1
  | inr x2 => arbitrary
  end.
```

```
Definition get22 x :=
  match x with
  | inl x1 => arbitrary
  | inr x2 => x2
  end.
```


End Get.

9.2.3 Lifting functions over sum types

Section Fget.

Context $\{A1:Type\} \{A2:Type\} \{IB1:Inhab\ B1\} \{IB2:Inhab\ B2\}$.

Implicit Types $f : A1+A2 \rightarrow B1+B2$.

Definition fun_get21 $f :=$
 fun $x \Rightarrow$ get21 (f (inl x)).

Definition fun_get22 $f :=$
 fun $x \Rightarrow$ get22 (f (inr x)).

End Fget.

Chapter 10

Library SLF.LibRelation

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibLogic LibBool LibLogic LibProd LibSum.
From SLF Require Export LibOperation.
```

10.1 Type of binary relations

10.1.1 Type of endorelation, i.e. homogeneous binary relations

```
Definition binary (A : Type) := A → A → Prop.
```

10.1.2 Inhabited

```
#[global]
Instance Inhab_binary : ∀ A, Inhab (binary A).
Proof using. intros. apply (Inhab_of_val (fun _ _ => True)). Qed.
```

10.1.3 Extensionality

```
Lemma binary_ext : ∀ A (R1 R2:binary A),
  (∀ x y, R1 x y ↔ R2 x y) →
  R1 = R2.
Proof using. extens×. Qed.

#[global]
Instance Extensionality_binary : ∀ A,
  Extensionality (binary A).
Proof using. intros. apply (Extensionality_make (@binary_ext A)). Defined.
```

10.2 Properties of relations

10.2.1 Reflexivity

Definition `refl A (R:binary A) :=`
 $\forall x, R\ x\ x.$

Section `Refl.`

Variables $(A : \text{Type}).$

Implicit Types $R : \text{binary } A.$

Lemma `refl_inv` : $\forall x\ y\ R,$
 `refl R` $\rightarrow$
 $x = y \rightarrow$
 $R\ x\ y.$

Proof using. `intros_all. subst`. `Qed.`

End `Refl.`

10.2.2 Irreflexivity

Definition `irrefl A (R:binary A) :=`
 $\forall x, \neg (R\ x\ x).$

Section `Irrefl.`

Variables $(A : \text{Type}).$

Implicit Types $R : \text{binary } A.$

Lemma `irrefl_inv` : $\forall x\ R,$
 `irrefl R` $\rightarrow$
 $R\ x\ x \rightarrow$
 False.

Proof using. `introv H P. apply` $\times$ `H. Qed.`

Lemma `irrefl_eq_forall_neq` : $\forall R,$
 `irrefl R` $= (\forall x\ y, R\ x\ y \rightarrow x \neq y).$

Proof using.

`unfold irrefl. extens. iff M.`
 $\{ \text{introv } H\ E. \text{subst} \times. \}$
 $\{ \text{autos} \times. \}$

`Qed.`

Lemma `irrefl_inv_neq` : $\forall x\ y\ R,$
 `irrefl R` $\rightarrow$
 $R\ x\ y \rightarrow$
 $x \neq y.$

Proof using. `introv H M. rewrite` $\times$ `irrefl_eq_forall_neq in H. Qed.`

End Irrefl.

10.2.3 Symmetry

Definition sym A (R :binary A) :=

$\forall x\ y, R\ x\ y \rightarrow R\ y\ x.$

Section Sym.

Variables (A : Type).

Implicit Types R : binary A .

Lemma sym_inv : $\forall x\ y\ R,$

sym $R \rightarrow$

$R\ x\ y \rightarrow$

$R\ y\ x.$

Proof using. *intros* $Sy\ R1$. *apply*× Sy . Qed.

Lemma sym_inv_eq : $\forall x\ y\ R,$

sym $R \rightarrow$

$R\ x\ y = R\ y\ x.$

Proof using. *unfold* sym. *extens*×. Qed.

Lemma sym_eq_forall_eq : $\forall R,$

sym $R = (\forall x\ y, R\ x\ y = R\ y\ x).$

Proof using.

unfold sym. *extens*. *iff* M .

{ *extens*×. }

{ *intros*. *rewrite*× M . }

Qed.

End Sym.

10.2.4 Asymmetry

Definition asym A (R :binary A) :=

$\forall x\ y, R\ x\ y \rightarrow \neg R\ y\ x.$

Section Asym.

Variables (A : Type).

Implicit Types R : binary A .

Lemma asym_eq_forall_false : $\forall R,$

asym $R = (\forall x\ y, R\ x\ y \rightarrow R\ y\ x \rightarrow \mathbf{False}).$

Proof using. *unfold* asym. *extens*×. Qed.

Lemma asym_inv : $\forall x\ y\ R,$

asym $R \rightarrow$

$R\ x\ y \rightarrow$

```

     $R\ y\ x \rightarrow$ 
    False.
Proof using. intros  $H\ M1\ M2$ . apply  $\times\ H$ . Qed.
End Asym.

```

10.2.5 Antisymmetry

```

Definition antisym  $A\ (R:\text{binary } A) :=$ 
   $\forall\ x\ y,\ R\ x\ y \rightarrow R\ y\ x \rightarrow x = y$ .
Section Antisym.
Variables ( $A : \text{Type}$ ).
Implicit Types  $R : \text{binary } A$ .
Lemma antisym_eq_forall_eq :  $\forall\ R,$ 
  antisym  $R = (\forall\ x\ y,\ R\ x\ y \rightarrow R\ y\ x \rightarrow x = y)$ .
Proof using. unfold antisym. extens  $\times$ . Qed.
Lemma antisym_inv :  $\forall\ x\ y\ R,$ 
  antisym  $R \rightarrow$ 
   $R\ x\ y \rightarrow$ 
   $R\ y\ x \rightarrow$ 
   $x \neq y \rightarrow$ 
  False.
Proof using. intros_all  $\times$ . Qed.
End Antisym.

```

10.2.6 Antisymmetry with respect to an equivalence relation

```

Definition antisym_wrt  $A\ (E:\text{binary } A)\ R :=$ 
   $\forall\ x\ y,\ R\ x\ y \rightarrow R\ y\ x \rightarrow E\ x\ y$ .

```

10.2.7 Transitivity

```

Definition trans  $A\ (R:\text{binary } A) :=$ 
   $\forall\ y\ x\ z,\ R\ x\ y \rightarrow R\ y\ z \rightarrow R\ x\ z$ .
Section Trans.
Variables ( $A : \text{Type}$ ).
Implicit Types  $R : \text{binary } A$ .
Lemma trans_eq_forall_impl :  $\forall\ R,$ 
  trans  $R = (\forall\ y\ x\ z,\ R\ x\ y \rightarrow R\ y\ z \rightarrow R\ x\ z)$ .
Proof using. unfold trans. extens  $\times$ . Qed.

```

```

Lemma trans_inv :  $\forall y x z R,$ 
  trans  $R \rightarrow$ 
   $R x y \rightarrow$ 
   $R y z \rightarrow$ 
   $R x z.$ 
Proof using. intros Tr R1 R2. apply $\times$  Tr. Qed.

Lemma trans_inv_swap :  $\forall y x z R,$ 
  trans  $R \rightarrow$ 
   $R y z \rightarrow$ 
   $R x y \rightarrow$ 
   $R x z.$ 
Proof using. intros Tr R1 R2. apply $\times$  Tr. Qed.

  trans + sym

Definition trans_sym_ll := trans_inv.

Lemma trans_sym_lr :  $\forall y x z R,$ 
  trans  $R \rightarrow$ 
  sym  $R \rightarrow$ 
   $R x y \rightarrow$ 
   $R z y \rightarrow$ 
   $R x z.$ 
Proof using. intros Tr Sy R1 R2. apply $\times$  Tr. Qed.

Lemma trans_sym_rr :  $\forall y x z R,$ 
  trans  $R \rightarrow$ 
  sym  $R \rightarrow$ 
   $R y x \rightarrow$ 
   $R z y \rightarrow$ 
   $R x z.$ 
Proof using. intros Tr Sy R1 R2. apply $\times$  Tr. Qed.

Lemma trans_sym_rl :  $\forall y x z R,$ 
  trans  $R \rightarrow$ 
  sym  $R \rightarrow$ 
   $R y x \rightarrow$ 
   $R y z \rightarrow$ 
   $R x z.$ 
Proof using. intros Tr Sy R1 R2. apply $\times$  Tr. Qed.

End Trans.

Arguments trans_inv [A] y [x] [z] [R].
Arguments trans_inv_swap [A] y [x] [z] [R].
Arguments trans_sym_rr [A] y [x] [z] [R].
Arguments trans_sym_lr [A] y [x] [z] [R].
Arguments trans_sym_rl [A] y [x] [z] [R].

```

10.2.8 Equivalence relation

```
Record equiv A (R:binary A) :=  
  { equiv_refl : refl R;  
    equiv_sym : sym R;  
    equiv_trans : trans R }.
```

Section Equiv.

Variables (A : Type).

Implicit Types R : binary A.

End Equiv.

10.2.9 Inclusion

```
Definition rel_incl A B (R1 R2:A→B→Prop) :=  
  ∀ x y, R1 x y → R2 x y.
```

Section Incl.

Variables (A B : Type).

Implicit Types R : A→B→Prop.

```
Lemma rel_incl_eq_forall_impl : ∀ R1 R2,  
  rel_incl R1 R2 = (∀ x y, R1 x y → R2 x y).
```

Proof using. auto. Qed.

```
Lemma refl_rel_incl :  
  refl (@rel_incl A B).
```

Proof using. *unfolds* refl, rel_incl. *autos*×. Qed.

```
Lemma refl_rel_incl' : ∀ R,  
  rel_incl R R.
```

Proof using. intros. *applies* refl_rel_incl. Qed.

Hint Resolve *refl_rel_incl* *refl_rel_incl'*.

```
Lemma antisym_rel_incl :  
  antisym (@rel_incl A B).
```

Proof using. *unfolds* rel_incl. *extends*×. Qed.

```
Lemma trans_rel_incl :  
  trans (@rel_incl A B).
```

Proof using. unfold trans, rel_incl. *autos*×. Qed.

End Incl.

10.2.10 Totality

```
Definition total A (R:binary A) :=
```

$\forall x y, R x y \vee R y x.$

Section Total.

Variables $(A : \text{Type}).$

Implicit Types $R : \text{binary } A.$

Lemma total_eq_forall_or : $\forall R,$
total $R = (\forall x y, R x y \vee R y x).$

Proof using. auto. Qed.

Lemma total_inv : $\forall x y R,$
total $R \rightarrow$
 $R x y \vee R y x.$

Proof using. introv $H.$ apply $\times H.$ Qed.

Lemma total_inv_not_l : $\forall x y R,$
total $R \rightarrow$
 $\neg R x y \rightarrow$
 $R y x.$

Proof using. introv $H N.$ destruct $\times (H x y).$ Qed.

Lemma total_inv_not_r : $\forall x y R,$
total $R \rightarrow$
 $\neg R y x \rightarrow$
 $R x y.$

Proof using. introv $H N.$ destruct $\times (H x y).$ Qed.

End Total.

10.2.11 Trichotomy

Inductive **trichotomy** $A (R:\text{binary } A) : \text{binary } A :=$
| trichotomy_left : $\forall x y,$
 $R x y \rightarrow$
 $x \neq y \rightarrow$
 $\neg R y x \rightarrow$
 trichotomy $R x y$
| trichotomy_eq : $\forall x,$
 $\neg R x x \rightarrow$
 trichotomy $R x x$
| trichotomy_right : $\forall x y,$
 $\neg R x y \rightarrow$
 $x \neq y \rightarrow$
 $R y x \rightarrow$
 trichotomy $R x y.$

Definition trichotomous $A (R:\text{binary } A) :=$
 $\forall x y, \text{trichotomy } R x y.$

10.2.12 Definedness

Definition defined $A\ B\ (R:A\rightarrow B\rightarrow\text{Prop}) :=$
 $\forall x, \exists y, R\ x\ y.$

Section Defined.

Variables $(A : \text{Type}).$

Implicit Types $R : \text{binary } A.$

Lemma total_eq_forall_exists : $\forall R,$
defined $R = (\forall x, \exists y, R\ x\ y).$

Proof using. auto. Qed.

Lemma defined_inv : $\forall x\ R,$
defined $R \rightarrow$
 $\exists y, R\ x\ y.$

Proof using. *introv* $H.$ *apply* $\times H.$ Qed.

Lemma defined_inv_not : $\forall x\ R,$
defined $R \rightarrow$
 $(\forall y, \neg R\ x\ y) \rightarrow$
False.

Proof using. *introv* $H\ N.$ *forwards* $\times (?&?): H.$ Qed.

End Defined.

10.2.13 Functionality

Definition functional $A\ B\ (R:A\rightarrow B\rightarrow\text{Prop}) :=$
 $\forall x\ y\ z, R\ x\ y \rightarrow R\ x\ z \rightarrow y = z.$

Section Functional.

Variables $(A : \text{Type}).$

Implicit Types $R : \text{binary } A.$

Lemma functional_eq_forall_eq : $\forall R,$
functional $R = (\forall x\ y\ z, R\ x\ y \rightarrow R\ x\ z \rightarrow y = z).$

Proof using. auto. Qed.

Lemma functional_inv : $\forall x\ z\ y\ R,$
functional $R \rightarrow$
 $R\ x\ y \rightarrow$
 $R\ x\ z \rightarrow$
 $y = z.$

Proof using. *introv* $H\ N1\ N2.$ *apply* $\times H.$ Qed.

End Functional.

10.2.14 Criteria for equality

Section Equality.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

If $R1$ is defined, $R2$ is functional, and $R1$ is a subset of $R2$, then $R1$ equals $R2$. In that case, $R1$ and $R2$ represent the graph of a total function.

Lemma `eq_of_incl_defined_functional` : $\forall R1 R2$,
 `rel_incl` $R1 R2 \rightarrow$
 `defined` $R1 \rightarrow$
 `functional` $R2 \rightarrow$
 $R1 = R2$.

Proof using.

`intros` $Hincl Hdef Hfun$. `extens`. `intros` $x y$. `iff` M .
 { `eauto`. }
 { `forwards` ($w' \& M1$): $Hdef x$.
 `forwards` $M2$: $Hincl M1$.
 `forwards`: $Hfun M M2$. `subst`×. }

Qed.

End Equality.

10.2.15 Properties of the equality relation

These results are not in `LibEqual` because the definitions from `LibRelation` are not yet available from that file.

Section Eq.

Variables ($A : \text{Type}$).

Lemma `refl_eq` :

`refl` (`@eq` A).

Proof using. `intros_all`; `subst`⊖. Qed.

Lemma `sym_eq` :

`sym` (`@eq` A).

Proof using. `intros_all`; `subst`⊖. Qed.

Lemma `trans_eq` :

`trans` (`@eq` A).

Proof using. `intros_all`; `subst`⊖. Qed.

Lemma `equiv_eq` :

`equiv` (`@eq` A).

Proof using. `intros`. `constructor`; `intros_all`; `subst`⊖. Qed.

End Eq.

10.2.16 Properties of the equality relation

These results are not in `LibLogic` because the definitions from `LibRelation` are not yet available from that file. See also `LibOrder` for a package of these properties.

Section `Pred_incl`.

Variables `(A : Type)`.

Lemma `refl_pred_incl` :

`refl (@pred_incl A)`.

Proof using. `unfold refl, pred_incl. autos ×. Qed.`

Lemma `antisym_pred_incl` :

`antisym (@pred_incl A)`.

Proof using. `unfold antisym, pred_incl. extens ×. Qed.`

Lemma `trans_pred_incl` :

`trans (@pred_incl A)`.

Proof using. `unfold trans, pred_incl. autos ×. Qed.`

End `Pred_incl`.

10.2.17 Properties of the equivalence relation

These results are not in `LibLogic` because the definitions from `LibRelation` are not yet available from that file. See also `LibOrder` for a package of these properties.

Section `Iff`.

Lemma `refl_iff` :

`refl iff`.

Proof using. `unfold refl, iff. autos ×. Qed.`

Lemma `antisym_iff` :

`antisym iff`.

Proof using. `unfold antisym, iff. extens ×. Qed.`

Lemma `trans_iff` :

`trans iff`.

Proof using. `unfold trans, iff. autos ×. Qed.`

End `Iff`.

10.3 Basic constructions

10.3.1 The empty relation

Definition `empty A : binary A :=`

`fun x y => False.`

```

Section Empty.
Variables (A : Type).
Implicit Types x y : A.
Lemma empty_eq :  $\forall x y$ ,
  empty x y = False.
Proof using. auto. Qed.
Lemma empty_inv :  $\forall x y$ ,
  empty x y  $\rightarrow$ 
False.
Proof using. auto. Qed.
Lemma functional_empty :
  functional (@empty A).
Proof using. unfolds  $\times$  empty, functional. Qed.
End Empty.

```

10.3.2 Union of two relations

```

Definition union A (R1 R2:binary A) : binary A :=
  fun x y  $\Rightarrow$  R1 x y  $\vee$  R2 x y.

```

```

Section Union.
Variables (A : Type).
Implicit Types R : binary A.
Lemma union_l :  $\forall R1 R2 x y$ ,
  R1 x y  $\rightarrow$ 
  union R1 R2 x y.
Proof using. unfold union. eauto. Qed.
Lemma union_r :  $\forall R1 R2 x y$ ,
  R2 x y  $\rightarrow$ 
  union R1 R2 x y.
Proof using. unfold union. eauto. Qed.
Lemma rel_incl_union_l :  $\forall R1 R2$ ,
  rel_incl R1 (union R1 R2).
Proof using. unfold rel_incl, union. eauto. Qed.
Lemma rel_incl_union_r :  $\forall R1 R2$ ,
  rel_incl R2 (union R1 R2).
Proof using. unfold rel_incl, union. eauto. Qed.
Lemma refl_union_l :  $\forall R1 R2$ ,
  refl R1  $\rightarrow$ 
  refl (union R1 R2).

```

```

Proof using. unfold refl, union. eauto. Qed.

Lemma refl_union_r :  $\forall R1 R2$ ,
  refl  $R2 \rightarrow$ 
  refl (union  $R1 R2$ ).
Proof using. unfold refl, union. eauto. Qed.

Lemma comm_union :
  comm (@union A).
Proof using. unfold union. extens×. Qed.

Lemma comm_union_args :  $\forall R1 R2 x y$ ,
  union  $R2 R1 x y \rightarrow$ 
  union  $R1 R2 x y$ .
Proof using. intros. rewrite¬ comm_union. Qed.

  Union is functional provided disjoint domains
  Lemma functional_union :  $\forall R1 R2$ ,
    functional  $R1 \rightarrow$ 
    functional  $R2 \rightarrow$ 
    ( $\forall x y z, R1 x y \rightarrow R2 x z \rightarrow \text{False}$ )  $\rightarrow$ 
    functional (union  $R1 R2$ ).
Proof using.
  intros. unfold union. intros  $x y z Hxy Hxz$ .
  destruct  $Hxy$ ; destruct  $Hxz$ ; auto_false×.
Qed.

Lemma covariant_union :  $\forall R1 R2 S1 S2$ ,
  rel_incl  $R1 S1 \rightarrow$ 
  rel_incl  $R2 S2 \rightarrow$ 
  rel_incl (union  $R1 R2$ ) (union  $S1 S2$ ).
Proof using. unfold rel_incl, union. autos×. Qed.

End Union.

```

10.3.3 Intersection of two relations

```

Definition inter A ( $R1 R2$ :binary A) : binary A :=
  fun x y  $\Rightarrow R1 x y \wedge R2 x y$ .

```

10.3.4 Complement of a relation

```

Definition compl A ( $R$ :binary A) : binary A :=
  fun x y  $\Rightarrow \neg R y x$ .

```

10.3.5 Inverse of a relation

Definition `inverse` A (R :binary A) : binary A :=
 fun x y $\Rightarrow R$ y x .

Section `Inverse`.

Variables (A : Type).

Implicit Types R : binary A .

Lemma `inverse_eq_fun` : $\forall R$,
 inverse R = (fun x y $\Rightarrow R$ y x).

Proof using. `auto`. `Qed`.

Lemma `inverse_eq` : $\forall R$ x y ,
 inverse R x y = R y x .

Proof using. `auto`. `Qed`.

Lemma `injective_inverse` :
 injective (@inverse A).

Proof using.

intros $R1$ $R2$ E . *extens*. intros x y .
 unfolds inverse. *rewrite* $\times$ (fun_eq_2 y x E).

`Qed`.

Lemma `inverse_sym` : $\forall R$,
 sym R $\rightarrow$
 inverse R = R .

Proof using. intros. *unfold* inverse. *extens* $\times$. `Qed`.

Lemma `inverse_inverse` : $\forall R$,
 inverse (inverse R) = R .

Proof using. *extens* $\times$. `Qed`.

Lemma `inverse_eq_l` : $\forall R2$ $R1$,
 $R1$ = inverse $R2$ $\rightarrow$
 inverse $R1$ = $R2$.

Proof using. intros. *apply* injective_inverse. *rewrite* $\neg$ inverse_inverse. `Qed`.

Lemma `inverse_eq_r` : $\forall R2$ $R1$,
 inverse $R1$ = $R2$ $\rightarrow$
 $R1$ = inverse $R2$.

Proof using. intros. *apply* injective_inverse. *rewrite* $\neg$ inverse_inverse. `Qed`.

Lemma `refl_inverse` : $\forall R$,
 refl R $\rightarrow$
 refl (inverse R).

Proof using. *intros_all*. *unfolds* inverse. `auto`. `Qed`.

Lemma `trans_inverse` : $\forall R$,
 trans R $\rightarrow$

```

    trans (inverse R).
Proof using. intros_all. unfolds inverse. eauto. Qed.
Lemma antisym_inverse :  $\forall R$ ,
  antisym  $R \rightarrow$ 
  antisym (inverse  $R$ ).
Proof using. intros_all. unfolds inverse. auto. Qed.
Lemma antisym_wrt_inverse :  $\forall E R$ ,
  antisym_wrt  $E R \rightarrow$ 
  antisym_wrt  $E$  (inverse  $R$ ).
Proof using. intros_all. unfolds inverse. auto. Qed.
Lemma asym_inverse :  $\forall R$ ,
  asym  $R \rightarrow$ 
  asym (inverse  $R$ ).
Proof using. intros_all. unfolds inverse. apply  $\times H$ . Qed.
Lemma total_inverse :  $\forall R$ ,
  total  $R \rightarrow$ 
  total (inverse  $R$ ).
Proof using. intros_all. unfolds inverse. auto. Qed.
Lemma trichotomous_inverse :  $\forall R$ ,
  trichotomous  $R \rightarrow$ 
  trichotomous (inverse  $R$ ).
Proof using.
  introv H. intros x y. destruct (H x y).
  apply  $\neg$  trichotomy_right.
  apply  $\neg$  trichotomy_eq.
  apply  $\neg$  trichotomy_left.
Qed.
Lemma inverse_equiv :  $\forall A (E:\text{binary } A)$ ,
  equiv  $E \rightarrow$ 
  equiv (inverse  $E$ ).
Proof using.
  introv Equi. unfold inverse. constructor; intros_all;
  dintuition eauto.
Qed.
Lemma inverse_union :  $\forall R1 R2$ ,
  inverse (union  $R1 R2$ ) = union (inverse  $R1$ ) (inverse  $R2$ ).
Proof using.
  unfold inverse, union. constructor; intros_all;
  dintuition eauto.
Qed.
End Inverse.

```

10.3.6 preimage

Preimage, a.k.a. inverse image

Definition `rel_preimage` ($A\ B:\text{Type}$) ($R:\text{binary } B$) ($f:A\rightarrow B$) : `binary A` :=
 `fun x y => R (f x) (f y)`.

10.3.7 rel_seq

Composition of two relations, usually written $R1; R2$.

Definition `rel_seq` ($A\ B\ C:\text{Type}$) ($R1:A\rightarrow B\rightarrow\text{Prop}$) ($R2:B\rightarrow C\rightarrow\text{Prop}$) : $A\rightarrow C\rightarrow\text{Prop}$:=
 `fun x z => ∃ y, R1 x y ∧ R2 y z`.

A relation R is functional if and only if `inverse R` composed with R is a subset of the diagonal relation `eq`.

Lemma `functional_eq_seq_inverse_incl_eq` : $\forall A\ (R:\text{binary } A)$,
 `functional R = rel_incl (rel_seq (inverse R) R) eq`.

Proof using.

`unfold functional, rel_incl, rel_seq, inverse. extens. iff M; jauto.`

Qed.

10.3.8 Turning a function into a relation.

Definition `rel_fun` $A\ B$ ($f:A\rightarrow B$) :=
 `fun x y => (y = f x)`.

Section `Rel_fun`.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

End `Rel_fun`.

10.4 Products

10.4.1 Pointwise product

Definition `prod2` ($A1\ A2:\text{Type}$)
 ($R1:\text{binary } A1$) ($R2:\text{binary } A2$)
 : `binary (A1×A2)` :=
 `fun p1 p2 : A1×A2 => match p1,p2 with (x1,x2),(y1,y2) =>`
 `R1 x1 y1 ∧ R2 x2 y2 end`.

Definition `prod3` ($A1\ A2\ A3:\text{Type}$)
 ($R1:\text{binary } A1$) ($R2:\text{binary } A2$) ($R3:\text{binary } A3$)


```

: binary (A1 × A2 × A3) :=
prod2 (prod2 R1 R2) R3.

```

```

Definition prod4 (A1 A2 A3 A4:Type)
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4)
: binary (A1 × A2 × A3 × A4) :=
prod2 (prod3 R1 R2 R3) R4.

```

Tactics

```

Tactic Notation "unfold_prod" :=
  unfold prod4, prod3, prod2.
Tactic Notation "unfolds_prod" :=
  unfold prod4, prod3, prod2 in *.

```

Equivalence

```

Lemma prod2_equiv : ∀ A1 A2 (E1:binary A1) (E2:binary A2),
  equiv E1 →
  equiv E2 →
  equiv (prod2 E1 E2).

```

Proof using.

```

  introv [R1 S1 T1] [R2 S2 T2]. constructor.
  { intros [x1 x2]. simple×. }
  { intros [x1 x2] [y1 y2]. simple×. }
  { intros [x1 x2] [y1 y2] [z1 z2]. simple×. }

```

Qed.

10.4.2 Lexicographical product

```

Definition lexico2 {A1 A2} (R1:binary A1) (R2:binary A2)
: binary (A1 × A2) :=
  fun p1 p2 : A1 × A2 ⇒ let (x1,x2) := p1 in let (y1,y2) := p2 in
  (R1 x1 y1) ∨ (x1 = y1) ∧ (R2 x2 y2).

```

```

Definition lexico3 {A1 A2 A3}
(R1:binary A1) (R2:binary A2) (R3:binary A3) : binary (A1 × A2 × A3) :=
  lexico2 (lexico2 R1 R2) R3.

```

```

Definition lexico4 {A1 A2 A3 A4}
(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4)
: binary (A1 × A2 × A3 × A4) :=
  lexico2 (lexico3 R1 R2 R3) R4.

```

Tactics

```

Tactic Notation "unfold_lexico" :=
  unfold lexico4, lexico3, lexico2.

```

Tactic Notation "unfolds_lexico" :=
 unfold lexico4, lexico3, lexico2 in *.

Elimination

Section Lexico.

Variables (*A1 A2 A3 A4*:Type).

Variables (*R1*:binary *A1*) (*R2*:binary *A2*) (*R3*:binary *A3*) (*R4*:binary *A4*).

Lemma lexico2_1 : $\forall x1\ x2\ y1\ y2,$
 $R1\ x1\ y1 \rightarrow$
 $lexico2\ R1\ R2\ (x1, x2)\ (y1, y2).$

Proof using. intros. left $\neg$. Qed.

Lemma lexico2_2 : $\forall x1\ x2\ y1\ y2,$
 $x1 = y1 \rightarrow$
 $R2\ x2\ y2 \rightarrow$
 $lexico2\ R1\ R2\ (x1, x2)\ (y1, y2).$

Proof using. intros. right $\neg$. Qed.

Lemma lexico3_1 : $\forall x1\ x2\ x3\ y1\ y2\ y3,$
 $R1\ x1\ y1 \rightarrow$
 $lexico3\ R1\ R2\ R3\ (x1, x2, x3)\ (y1, y2, y3).$

Proof using. intros. left. left $\neg$. Qed.

Lemma lexico3_2 : $\forall x1\ x2\ x3\ y1\ y2\ y3,$
 $x1 = y1 \rightarrow$
 $R2\ x2\ y2 \rightarrow$
 $lexico3\ R1\ R2\ R3\ (x1, x2, x3)\ (y1, y2, y3).$

Proof using. intro. left. right $\neg$. Qed.

Lemma lexico3_3 : $\forall x1\ x2\ x3\ y1\ y2\ y3,$
 $x1 = y1 \rightarrow$
 $x2 = y2 \rightarrow$
 $R3\ x3\ y3 \rightarrow$
 $lexico3\ R1\ R2\ R3\ (x1, x2, x3)\ (y1, y2, y3).$

Proof using. intros. right $\neg$. Qed.

Lemma lexico4_1 : $\forall x1\ x2\ x3\ x4\ y1\ y2\ y3\ y4,$
 $R1\ x1\ y1 \rightarrow$
 $lexico4\ R1\ R2\ R3\ R4\ (x1, x2, x3, x4)\ (y1, y2, y3, y4).$

Proof using. intros. left. left. left $\neg$. Qed.

Lemma lexico4_2 : $\forall x1\ x2\ x3\ x4\ y1\ y2\ y3\ y4,$
 $x1 = y1 \rightarrow$
 $R2\ x2\ y2 \rightarrow$
 $lexico4\ R1\ R2\ R3\ R4\ (x1, x2, x3, x4)\ (y1, y2, y3, y4).$

Proof using. intros. left. left. right $\neg$. Qed.

Lemma lexico4_3 : $\forall x1\ x2\ x3\ x4\ y1\ y2\ y3\ y4,$

$x1 = y1 \rightarrow$
 $x2 = y2 \rightarrow$
 $R3\ x3\ y3 \rightarrow$
 $\text{lexico4}\ R1\ R2\ R3\ R4\ (x1, x2, x3, x4)\ (y1, y2, y3, y4).$
 Proof using. intros. left. right $\neg$. Qed.

Lemma lexico4_4 : $\forall\ x1\ x2\ x3\ x4\ y1\ y2\ y3\ y4,$
 $x1 = y1 \rightarrow$
 $x2 = y2 \rightarrow$
 $x3 = y3 \rightarrow$
 $R4\ x4\ y4 \rightarrow$
 $\text{lexico4}\ R1\ R2\ R3\ R4\ (x1, x2, x3, x4)\ (y1, y2, y3, y4).$
 Proof using. intros. right $\neg$. Qed.

End Lexico.

Transitivity

Lemma trans_lexico2 : $\forall\ A1\ A2$
 $(R1:\text{binary}\ A1)\ (R2:\text{binary}\ A2),$
 $\text{trans}\ R1 \rightarrow$
 $\text{trans}\ R2 \rightarrow$
 $\text{trans}\ (\text{lexico2}\ R1\ R2).$

Proof using.
 $\text{intros}\ Tr1\ Tr2. \text{intros}\ [x1\ x2]\ [y1\ y2]\ [z1\ z2]\ Rxy\ Ryz.$
 $\text{simpls. destruct}\ Rxy \text{ as } [L1][Eq1\ L1];$
 $\text{destruct}\ Ryz \text{ as } [M2][Eq2\ M2]; \text{subst}\times.$
 Qed.

Lemma trans_lexico3 : $\forall\ A1\ A2\ A3$
 $(R1:\text{binary}\ A1)\ (R2:\text{binary}\ A2)\ (R3:\text{binary}\ A3),$
 $\text{trans}\ R1 \rightarrow$
 $\text{trans}\ R2 \rightarrow$
 $\text{trans}\ R3 \rightarrow$
 $\text{trans}\ (\text{lexico3}\ R1\ R2\ R3).$

Proof using.
 $\text{intros}\ Tr1\ Tr2\ Tr3. \text{applies}\neg\ \text{trans\_lexico2. applies}\neg\ \text{trans\_lexico2.}$
 Qed.

Lemma trans_lexico4 : $\forall\ A1\ A2\ A3\ A4$
 $(R1:\text{binary}\ A1)\ (R2:\text{binary}\ A2)\ (R3:\text{binary}\ A3)\ (R4:\text{binary}\ A4),$
 $\text{trans}\ R1 \rightarrow$
 $\text{trans}\ R2 \rightarrow$
 $\text{trans}\ R3 \rightarrow$
 $\text{trans}\ R4 \rightarrow$
 $\text{trans}\ (\text{lexico4}\ R1\ R2\ R3\ R4).$

Proof using.

intros Tr1 Tr2 Tr3. applies $\neg$ trans_lexico3. applies $\neg$ trans_lexico2.
 Qed.

Inclusion

Lemma `rel_incl_lexico2` : $\forall A1 A2$
 (*R1 R1'*:binary *A1*) (*R2 R2'*:binary *A2*),
`rel_incl R1 R1'` $\rightarrow$
`rel_incl R2 R2'` $\rightarrow$
`rel_incl (lexico2 R1 R2) (lexico2 R1' R2')`.

Proof using.

intros I1 I2. intros [x1 x2] [y1 y2] [H1|[H1 H2]].
 { `left  $\neg$ .` } { `subst. right  $\neg$ .` }

Qed.

10.5 Closures

10.5.1 Reflexive closure

Inductive `rclosure` *A* (*R*:binary *A*) : binary *A* :=
 | `rclosure_once` : $\forall x y,$
 R x y $\rightarrow$
 `rclosure R x y`
 | `rclosure_refl` : $\forall x,$
 `rclosure R x x`.

Section `Rclosure`.

Variables (*A* : Type).

Implicit Types *R* : binary *A*.

Hint Constructors `rclosure`.

Equivalent definition

Lemma `rclosure_eq_fun` : $\forall R,$
`rclosure R` = (`fun x y  $\Rightarrow$  R x y  $\vee$  x = y`).

Proof using. *extens. iff M; destruct M; subst $\times$.* Qed.

Lemma `rclosure_eq` : $\forall R x y,$
`rclosure R x y` = (*R x y* $\vee$ *x = y*).

Proof using. *extens. iff M; destruct M; subst $\times$.* Qed.

Lemma `rclosure_inv` : $\forall R x y,$
`rclosure R x y` $\rightarrow$
R x y $\vee$ *x = y*.

Proof using. *intros M; rewrite $\times$ rclosure_eq in M.* Qed.

Properties

Lemma refl_rclosure : $\forall R$,
 refl (**rclosure** R).
 Proof using. *unfolds* $\times$ refl. Qed.

Lemma sym_rclosure : $\forall R$,
 sym $R \rightarrow$
 sym (**rclosure** R).
 Proof using. *unfolds* sym. *introv* $M N$. *destruct* $\times$ N . Qed.

Lemma antisym_rclosure : $\forall R$,
 antisym $R \rightarrow$
 antisym (**rclosure** R).
 Proof using.
 unfolds antisym. *introv* $M N1 N2$.
 destruct $N1$; *destruct* $N2$; *subst* $\times$.
 Qed.

Lemma antisym_wrt_rclosure : $\forall E R$,
 antisym_wrt $E R \rightarrow$
 antisym_wrt (**rclosure** E) (**rclosure** R).
 Proof using.
 unfolds antisym_wrt. *introv* $M N1 N2$.
 destruct $N1$; *destruct* $N2$; *subst* $\times$.
 Qed.

Lemma trans_rclosure : $\forall R$,
 trans $R \rightarrow$
 trans (**rclosure** R).
 Proof using.
 unfolds trans. *introv* $H M1 M2$.
 destruct $M1$; *destruct* $M2$; *subst* $\times$.
 Qed.

Lemma total_rclosure : $\forall R$,
 total $R \rightarrow$
 total (**rclosure** R).
 Proof using.
 unfolds total. *introv* H . *intros* $x y$. *destruct* $\times$ ($H x y$).
 Qed.

Lemma rclosure_eq_of_refl : $\forall R$,
 refl $R \rightarrow$
 rclosure $R = R$.
 Proof using.
 unfolds refl. *introv* H . *extens*. *iff* M .
 { *destruct* M ; *subst* $\times$. } { *auto*. }
 Qed.

Lemma rclosure_inverse_eq : $\forall R$,
 $\text{rclosure} (\text{inverse } R) = \text{inverse} (\text{rclosure } R)$.
 Proof using. unfold inverse. extens. iff M; destruct× M. Qed.

Constructors

TODO: rename to rclosure_of_rclosure_step rclosure_of_step_rclosure trans_inv_rclosure_step
 trans_inv_step_rclosure

Lemma rclosure_trans_l : $\forall y x z R$,
 $\text{trans } R \rightarrow$
 $\text{rclosure } R x y \rightarrow$
 $R y z \rightarrow$
 $\text{rclosure } R x z$.

Proof using. introv T M N. rewrite rclosure_eq in *. destruct M; subst×. Qed.

Lemma rclosure_trans_r : $\forall y x z R$,
 $\text{trans } R \rightarrow$
 $R x y \rightarrow$
 $\text{rclosure } R y z \rightarrow$
 $\text{rclosure } R x z$.

Proof using. introv T M N. rewrite rclosure_eq in *. destruct N; subst×. Qed.

Lemma trans_rclosure_l : $\forall y x z R$,
 $\text{trans } R \rightarrow$
 $\text{rclosure } R x y \rightarrow$
 $R y z \rightarrow$
 $R x z$.

Proof using. introv T M H. rewrite rclosure_eq in *. destruct M; subst×. Qed.

Lemma trans_rclosure_r : $\forall y x z R$,
 $\text{trans } R \rightarrow$
 $R x y \rightarrow$
 $\text{rclosure } R y z \rightarrow$
 $R x z$.

Proof using. introv T M N. rewrite rclosure_eq in *. destruct N; subst×. Qed.

Negation

Lemma not_rclosure_inv : $\forall R x y$,
 $\neg \text{rclosure } R x y \rightarrow$
 $\neg R x y \wedge x \neq y$.

Proof using. introv M. rewrite× rclosure_eq in M. Qed.

Lemma not_rclosure_inv_rel : $\forall R x y$,
 $\neg \text{rclosure } R x y \rightarrow$
 $\neg R x y$.

Proof using. introv M. rewrite× rclosure_eq in M. Qed.

Lemma not_rclosure_inv_neq : $\forall R x y$,

```

    ¬ rclosure R x y →
    x ≠ y.
Proof using. intros M. rewrite× rclosure_eq in M. Qed.

Inclusions
Lemma rel_incl_rclosure : ∀ R,
  rel_incl R (rclosure R).
Proof using. unfolds× rel_incl. Qed.
Lemma rclosure_of_rel : ∀ R x y,
  R x y →
  rclosure R x y.
Proof using. intros. applies× rel_incl_rclosure. Qed.
Lemma covariant_rclosure : ∀ R1 R2,
  rel_incl R1 R2 →
  rel_incl (rclosure R1) (rclosure R2).
Proof using. intros H M. destruct× M. Qed.
Lemma rel_incl_rclosure_rclosure : ∀ R1 R2,
  rel_incl R1 (rclosure R2) →
  rel_incl (rclosure R1) (rclosure R2).
Proof using. intros H M. destruct× M. Qed.

End Rclosure.

#[global]
Hint Constructors rclosure : rclosure.

```

10.5.2 Symmetric closure

```

Inductive sclosure A (R:binary A) : binary A :=
| sclosure_once : ∀ x y,
  R x y →
  sclosure R x y
| sclosure_sym : ∀ x y,
  sclosure R y x →
  sclosure R x y.

```

Section Sclosure.

Variables (*A* : Type).

Implicit Types *R* : binary *A*.

Hint Constructors **sclosure**.

Equivalent definition

```

Lemma sclosure_eq : ∀ R x y,
  sclosure R x y = (R x y ∨ R y x).

```

Proof using.

extens. *iff* M .
{ *induction* $\times M$. }
{ *destruct* M ; *subst* $\times$. }

Qed.

Lemma *sclosure_inv* : $\forall R\ x\ y$,
sclosure $R\ x\ y \rightarrow$
 $R\ x\ y \vee R\ y\ x$.

Proof using. *intros* M ; *rewrite* $\times$ *sclosure_eq* in M . Qed.

Properties

Lemma *refl_sclosure* : $\forall R$,
refl $R \rightarrow$
refl (**sclosure** R).

Proof using. *unfolds* $\times$ *refl*. Qed.

Lemma *sym_sclosure* : $\forall R$,
sym (**sclosure** R).

Proof using. *unfolds* *sym*. *intros* N . *destruct* $\times N$. Qed.

Lemma *total_sclosure* : $\forall R$,
total $R \rightarrow$
total (**sclosure** R).

Proof using.

unfolds *total*. *intros* H . *intros* $x\ y$. *destruct* $\times (H\ x\ y)$.

Qed.

Lemma *sclosure_eq_of_sym* : $\forall R$,
sym $R \rightarrow$
sclosure $R = R$.

Proof using.

unfolds *sym*. *intros* H . *extens.* *intros*. *rewrite* *sclosure_eq*.
iff M . { *destruct* M ; *subst* $\times$. } { *auto*. }

Qed.

Lemma *sclosure_inverse_eq* : $\forall R$,
sclosure (*inverse* R) = **sclosure** R .

Proof using.

unfolds *inverse*. *extens.* *intros*. *do* 2 *rewrite* *sclosure_eq*. *autos* $\times$.

Qed.

Negation

Lemma *not_sclosure_inv* : $\forall R\ x\ y$,
 $\neg$ **sclosure** $R\ x\ y \rightarrow$
 $\neg R\ x\ y \wedge \neg R\ y\ x$.

Proof using. *intros* M . *rewrite* $\times$ *sclosure_eq* in M . Qed.

Lemma not_sclosure_inv_l : $\forall R\ x\ y,$
 $\neg \text{sclosure } R\ x\ y \rightarrow$
 $R\ x\ y \rightarrow$
False.

Proof using. *intros* M . *rewrite* $\times$ *sclosure_eq* in M . *Qed*.

Lemma not_sclosure_inv_r : $\forall R\ x\ y,$
 $\neg \text{sclosure } R\ x\ y \rightarrow$
 $R\ y\ x \rightarrow$
False.

Proof using. *intros* M . *rewrite* $\times$ *sclosure_eq* in M . *Qed*.

Inclusions

Lemma rel_incl_sclosure : $\forall R,$
 $\text{rel\_incl } R\ (\text{sclosure } R).$

Proof using. *unfolds* $\times$ *rel_incl*. *Qed*.

Lemma rel_incl_inverse_sclosure : $\forall R,$
 $\text{rel\_incl } (\text{inverse } R)\ (\text{sclosure } R).$

Proof using. *unfolds* $\times$ *rel_incl*, *inverse*. *Qed*.

Lemma covariant_sclosure : $\forall R1\ R2,$
 $\text{rel\_incl } R1\ R2 \rightarrow$
 $\text{rel\_incl } (\text{sclosure } R1)\ (\text{sclosure } R2).$

Proof using. *intros* $H\ M$. *induction* $\times$ M . *Qed*.

Lemma rel_incl_sclosure_sclosure : $\forall R1\ R2,$
 $\text{rel\_incl } R1\ (\text{sclosure } R2) \rightarrow$
 $\text{rel\_incl } (\text{sclosure } R1)\ (\text{sclosure } R2).$

Proof using. *intros* $H\ M$. *induction* $\times$ M . *Qed*.

End Sclosure.

#[global]

Hint Constructors **sclosure** : *sclosure*.

10.5.3 Reflexive-symmetric closure

Inductive **rsclosure** $A\ (R:\text{binary } A) : \text{binary } A :=$
 $| \text{rsclosure\_once} : \forall x\ y,$
 $\quad R\ x\ y \rightarrow$
 $\quad \text{rsclosure } R\ x\ y$
 $| \text{rsclosure\_refl} : \forall x,$
 $\quad \text{rsclosure } R\ x\ x$
 $| \text{rsclosure\_sym} : \forall x\ y,$
 $\quad \text{rsclosure } R\ x\ y \rightarrow$
 $\quad \text{rsclosure } R\ y\ x.$

Section Rsclosure.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

Hint Constructors **rsclosure**.

Equivalent definition

Lemma rsclosure_eq : $\forall R \ x \ y,$
rsclosure $R \ x \ y = (R \ x \ y \vee R \ y \ x \vee x = y).$

Proof using.

extens. iff M.

{ induction× $M.$ }

{ destruct M as $[M|[M|M]]$; subst×. }

Qed.

Lemma rsclosure_inv : $\forall R \ x \ y,$

rsclosure $R \ x \ y \rightarrow$

$R \ x \ y \vee R \ y \ x \vee x = y.$

Proof using. *introv M*; rewrite× rsclosure_eq in M . Qed.

Properties

Lemma refl_rsclosure : $\forall R,$

refl (**rsclosure** R).

Proof using. *unfolds*× refl. Qed.

Lemma sym_rsclosure : $\forall R,$

sym (**rsclosure** R).

Proof using. *unfolds* sym. *introv N*. destruct× N . Qed.

Lemma total_rsclosure : $\forall R,$

total $R \rightarrow$

total (**rsclosure** R).

Proof using.

unfolds total. *introv H*. intros $x \ y$. destruct× $(H \ x \ y).$

Qed.

Lemma rsclosure_inverse_eq : $\forall R,$

rsclosure (inverse R) = **rsclosure** R .

Proof using.

unfold inverse. *extens*. intros. do 2 rewrite rsclosure_eq. *autos*×.

Qed.

Negation

Lemma not_rsclosure_inv : $\forall R \ x \ y,$

$\neg$ **rsclosure** $R \ x \ y \rightarrow$

$\neg R \ x \ y \wedge \neg R \ y \ x \wedge \neg x = y.$

Proof using. *introv M*. rewrite× rsclosure_eq in M . Qed.

Lemma not_rsclosure_inv_l : $\forall R\ x\ y,$
 $\neg \text{rsclosure } R\ x\ y \rightarrow$
 $R\ x\ y \rightarrow$
False.

Proof using. *intros* M . *rewrite* $\times$ *rsclosure_eq* in M . Qed.

Lemma not_rsclosure_inv_r : $\forall R\ x\ y,$
 $\neg \text{rsclosure } R\ x\ y \rightarrow$
 $R\ y\ x \rightarrow$
False.

Proof using. *intros* M . *rewrite* $\times$ *rsclosure_eq* in M . Qed.

Lemma not_rsclosure_inv_neq : $\forall R\ x\ y,$
 $\neg \text{rsclosure } R\ x\ y \rightarrow$
 $x \neq y.$

Proof using. *intros* M . *rewrite* $\times$ *rsclosure_eq* in M . Qed.

Inclusions

Lemma rel_incl_rsclosure : $\forall R,$
 $\text{rel\_incl } R\ (\text{rsclosure } R).$

Proof using. *unfolds* $\times$ *rel_incl*. Qed.

Lemma rel_incl_inverse_rsclosure : $\forall R,$
 $\text{rel\_incl } (\text{inverse } R)\ (\text{rsclosure } R).$

Proof using. *unfolds* $\times$ *rel_incl*, *inverse*. Qed.

Lemma covariant_rsclosure : $\forall R1\ R2,$
 $\text{rel\_incl } R1\ R2 \rightarrow$
 $\text{rel\_incl } (\text{rsclosure } R1)\ (\text{rsclosure } R2).$

Proof using. *intros* $H\ M$. *induction* $\times$ M . Qed.

Lemma rel_incl_rsclosure_rsclosure : $\forall R1\ R2,$
 $\text{rel\_incl } R1\ (\text{rsclosure } R2) \rightarrow$
 $\text{rel\_incl } (\text{rsclosure } R1)\ (\text{rsclosure } R2).$

Proof using. *intros* $H\ M$. *induction* $\times$ M . Qed.

End Rsclosure.

#[global]

Hint Constructors **rsclosure** : *rsclosure*.

10.5.4 Transitive closure (R^+), defined as $R \setminus \circ R \times$

Inductive **tclosure** $A\ (R:\text{binary } A) : \text{binary } A :=$
 $| \text{tclosure\_once} : \forall x\ y,$
 $\quad R\ x\ y \rightarrow$
 $\quad \text{tclosure } R\ x\ y$
 $| \text{tclosure\_trans} : \forall y\ x\ z,$

```

tclosure  $R\ x\ y \rightarrow$ 
tclosure  $R\ y\ z \rightarrow$ 
tclosure  $R\ x\ z$ .

```

Section Tclosure.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

Hint Constructors **tclosure**.

Properties

```

Lemma refl_tclosure :  $\forall\ R,$ 
  refl  $R \rightarrow$ 
  refl (tclosure  $R$ ).

```

Proof using. *unfolds* $\times$ refl. Qed.

```

Lemma sym_tclosure :  $\forall\ R,$ 
  sym  $R \rightarrow$ 
  sym (tclosure  $R$ ).

```

Proof using. *unfolds* sym. *intros* $H\ N$. *induction* $\times\ N$. Qed.

```

Lemma trans_tclosure :  $\forall\ R,$ 
  trans (tclosure  $R$ ).

```

Proof using. *unfolds* $\times$ trans. Qed.

```

Lemma total_tclosure :  $\forall\ R,$ 
  total  $R \rightarrow$ 
  total (tclosure  $R$ ).

```

Proof using.

```

unfolds total. intros  $H$ . intros  $x\ y$ . destruct  $\times\ (H\ x\ y)$ .

```

Qed.

```

Lemma tclosure_eq_of_trans :  $\forall\ R,$ 
  trans  $R \rightarrow$ 
  tclosure  $R = R$ .

```

Proof using.

```

unfolds trans. intros  $H$ . extends. iff  $M$ .
{ induction  $M$ ; subst  $\times$ . }
{ auto. }

```

Qed.

```

Lemma tclosure_inverse_eq :  $\forall\ R,$ 
  tclosure (inverse  $R$ ) = inverse (tclosure  $R$ ).

```

Proof using.

```

unfolds inverse. extends. intros  $x\ y$ . iff  $M$ ; induction  $\times\ M$ .

```

Qed.

Constructors

```

Lemma tclosure_l :  $\forall\ R\ y\ x\ z,$ 

```

$R\ x\ y \rightarrow$
tclosure $R\ y\ z \rightarrow$
tclosure $R\ x\ z$.

Proof using. *autos*×. Qed.

Lemma **tclosure_r** : $\forall\ R\ y\ x\ z,$
tclosure $R\ x\ y \rightarrow$
 $R\ y\ z \rightarrow$
tclosure $R\ x\ z$.

Proof using. *autos*×. Qed.

Inclusions

Lemma **rel_incl_tclosure** : $\forall\ R,$
 rel_incl R (**tclosure** R).

Proof using. *unfolds*× rel_incl. Qed.

Lemma **covariant_tclosure** : $\forall\ A\ (R1\ R2 : \text{binary } A),$
 rel_incl $R1\ R2 \rightarrow$
 rel_incl (**tclosure** $R1$) (**tclosure** $R2$).

Proof using. *introv H M. induction*× M . Qed.

Lemma **rel_incl_tclosure_tclosure** : $\forall\ R1\ R2,$
 rel_incl $R1$ (**tclosure** $R2$) $\rightarrow$
 rel_incl (**tclosure** $R1$) (**tclosure** $R2$).

Proof using. *introv H M. induction*× M . Qed.

Induction principle with steps at head or tail

Section Ind.

Inductive **tclosure'l** $A\ (R : \text{binary } A) : \text{binary } A :=$
 | **tclosure'l_once** : $\forall\ x\ y,$
 $R\ x\ y \rightarrow$
 tclosure'l $R\ x\ y$
 | **tclosure'l_step** : $\forall\ y\ x\ z,$
 $R\ x\ y \rightarrow$
 tclosure'l $R\ y\ z \rightarrow$
 tclosure'l $R\ x\ z$.

Lemma **trans_tclosure'l** : $\forall\ R,$
 trans (**tclosure'l** R).

Proof using.

Hint Constructors **tclosure'l**.

intros R y x z M1. gen z. induction M1; introv M2; autos×.
 Qed.

Lemma **tclosure_eq_tclosure'l** : $\forall\ R,$
tclosure $R = \text{tclosure'l } R$.

Proof using.

```

    extens. intros x y. iff M.
    { induction× M. appls× trans_tclosure'l y. }
    { induction× M. }
Qed.

Lemma tclosure_ind_l : ∀ R (P : A → A → Prop),
  (∀ x y, R x y → P x y) →
  (∀ y x z, R x y → tclosure R y z → P y z → P x z) →
  (∀ x y, tclosure R x y → P x y).
Proof using.
  introv H1 H2 M. rewrite tclosure_eq_tclosure'l in *. induction× M.
Qed.

Inductive tclosure'r A (R:binary A) : binary A :=
| tclosure'r_once : ∀ x y,
  R x y →
  tclosure'r R x y
| tclosure'r_step : ∀ y x z,
  tclosure'r R x y →
  R y z →
  tclosure'r R x z.

Lemma trans_tclosure'r : ∀ R,
  trans (tclosure'r R).
Proof using.
  Hint Constructors tclosure'r.
  intros R y x z M1 M2. gen x. induction M2; introv M1; autos×.
Qed.

Lemma tclosure_eq_tclosure'r : ∀ R,
  tclosure R = tclosure'r R.
Proof using.
  extens. intros x y. iff M.
  { induction× M. appls× trans_tclosure'r y. }
  { induction× M. }
Qed.

Lemma tclosure_ind_r : ∀ R (P : A → A → Prop),
  (∀ x y, R x y → P x y) →
  (∀ y x z, tclosure R x y → P x y → R y z → P x z) →
  (∀ x y, tclosure R x y → P x y).
Proof using.
  introv H1 H2 M. rewrite tclosure_eq_tclosure'r in *. induction× M.
Qed.

End Ind.

```

Inversion principle with steps at head or tail

```

Lemma tclosure_inv_l :  $\forall R\ x\ z,$ 
  tclosure  $R\ x\ z \rightarrow$ 
     $(R\ x\ z) \vee (\exists y, R\ x\ y \wedge \text{tclosure } R\ y\ z).$ 
Proof using. intros  $R.$  applies  $\times$  tclosure_ind_l. Qed.

Lemma tclosure_inv_r :  $\forall R\ x\ z,$ 
  tclosure  $R\ x\ z \rightarrow$ 
     $(R\ x\ z) \vee (\exists y, \text{tclosure } R\ x\ y \wedge R\ y\ z).$ 
Proof using. intros  $R.$  applies  $\times$  tclosure_ind_r. Qed.

End Tclosure.

#[global]
Hint Resolve tclosure_once tclosure_l tclosure_r
  : rtclosure.

```

10.5.5 Reflexive-transitive closure (R^*)

```

Inductive rtclosure  $A\ (R:\text{binary } A) : \text{binary } A :=
| rtclosure_once :  $\forall x\ y,$ 
   $R\ x\ y \rightarrow$ 
  rtclosure  $R\ x\ y$ 
| rtclosure_refl :  $\forall x,$ 
  rtclosure  $R\ x\ x$ 
| rtclosure_trans :  $\forall y\ x\ z,$ 
  rtclosure  $R\ x\ y \rightarrow$ 
  rtclosure  $R\ y\ z \rightarrow$ 
  rtclosure  $R\ x\ z.$$ 
```

Section Rtclosure.
 Variables ($A : \text{Type}$).
 Implicit Types $R : \text{binary } A$.
 Hint Constructors **rtclosure**.

Properties

```

Lemma refl_rtclosure :  $\forall R,$ 
  refl (rtclosure  $R$ ).
Proof using. unfolds  $\times$  refl. Qed.

Lemma sym_rtclosure :  $\forall R,$ 
  sym  $R \rightarrow$ 
  sym (rtclosure  $R$ ).
Proof using. unfolds sym. introv  $M\ N.$  induction  $\times$   $N.$  Qed.

Lemma trans_rtclosure :  $\forall R,$ 
  trans (rtclosure  $R$ ).
Proof using. unfolds  $\times$  trans. Qed.

```

Lemma total_rtclosure : $\forall R$,
 total $R \rightarrow$
 total (**rtclosure** R).

Proof using.

unfolds total. *introv* H . *intros* $x y$. *destruct* $\times (H x y)$.

Qed.

Lemma tclosure_eq_of_refl_trans : $\forall R$,
 refl $R \rightarrow$
 trans $R \rightarrow$
rtclosure $R = R$.

Proof using.

unfolds refl, trans. *introv* $H1 H2$. *extens*. *iff* M .
 { *induction* M ; *subst* $\times$. }
 { *autos* $\times$. }

Qed.

Lemma rtclosure_inverse_eq : $\forall R$,
rtclosure (inverse R) = inverse (**rtclosure** R).

Proof using. *unfold* inverse. *extens*. *iff* M ; *induction* $\times M$. Qed.

Constructors

Lemma rtclosure_l : $\forall R y x z$,
 $R x y \rightarrow$
rtclosure $R y z \rightarrow$
rtclosure $R x z$.

Proof using. *autos* $\times$. Qed.

Lemma rtclosure_r : $\forall R y x z$,
rtclosure $R x y \rightarrow$
 $R y z \rightarrow$
rtclosure $R x z$.

Proof using. *autos* $\times$. Qed.

Lemma rtclosure_r' : $\forall R y x z$,
 $R y z \rightarrow$
rtclosure $R x y \rightarrow$
rtclosure $R x z$.

Proof using. *autos* $\times$. Qed.

Inclusion

Lemma rel_incl_rtclosure : $\forall R$,
 rel_incl R (**rtclosure** R).

Proof using. *unfolds* $\times$ rel_incl. Qed.

Lemma covariant_rtclosure : $\forall R1 R2$,
 rel_incl $R1 R2 \rightarrow$

rel_incl (rtclosure *R1*) (rtclosure *R2*).
 Proof using. *unfolds* rel_incl. *intros* *H M*. *induction*× *M*. Qed.

Lemma rel_incl_rtclosure_rtclosure : ∀ *R1 R2*,

rel_incl *R1* (rtclosure *R2*) →
 rel_incl (rtclosure *R1*) (rtclosure *R2*).

Proof using. *unfolds* rel_incl. *intros* *H M*. *induction*× *M*. Qed.

Lemma rel_incl_union_rtclosure : ∀ *R1 R2*,

rel_incl (union (rtclosure *R1*) (rtclosure *R2*))
 (rtclosure (union *R1 R2*)).

Proof using.

hint rel_incl_union_l, rel_incl_union_r. *intros* [*M*|*M*].
 { *applies*× covariant_rtclosure *R1*. }
 { *applies*× covariant_rtclosure *R2*. }

Qed.

Negation

Lemma not_rtclosure_inv_neq : ∀ *R x y*,

¬ rtclosure *R x y* →
x ≠ *y*.

Proof using. *intros* *M E*. *subst*. *induction*× *M*. Qed.

Induction principle with steps at head or tail

Section Ind.

Inductive rtclosure' *l A* (*R*:binary *A*) : binary *A* :=

| rtclosure'_l_refl : ∀ *x*,
 rtclosure' *l R x x*
 | rtclosure'_l_step : ∀ *y x z*,
R x y →
 rtclosure' *l R y z* →
 rtclosure' *l R x z*.

Lemma trans_rtclosure' *l* : ∀ *R*,

trans (rtclosure' *l R*).

Proof using.

Hint Constructors rtclosure' *l*.

intros *R y x z M1*. *gen* *z*. *induction* *M1*; *intros* *M2*; *autos*×.

Qed.

Lemma rtclosure_eq_rtclosure' *l* : ∀ *R*,

rtclosure *R* = rtclosure' *l R*.

Proof using.

extends. *intros* *x y*. *iff* *M*.
 { *induction*× *M*. *applies*× trans_rtclosure' *l y*. }
 { *induction*× *M*. }

Qed.

Lemma `rtclosure_ind_l` : $\forall R (P : A \rightarrow A \rightarrow \text{Prop}),$
 $(\forall x, P\ x\ x) \rightarrow$
 $(\forall y\ x\ z, R\ x\ y \rightarrow \text{rtclosure}\ R\ y\ z \rightarrow P\ y\ z \rightarrow P\ x\ z) \rightarrow$
 $(\forall x\ y, \text{rtclosure}\ R\ x\ y \rightarrow P\ x\ y).$

Proof using.

*intros H1 H2 M. rewrite rtclosure_eq_rtclosure' in *. induction× M.*

Qed.

Inductive `rtclosure'r` $A (R:\text{binary } A) : \text{binary } A :=$
 | `rtclosure'r_refl` : $\forall x,$
 `rtclosure'r` $R\ x\ x$
 | `rtclosure'r_step` : $\forall y\ x\ z,$
 `rtclosure'r` $R\ x\ y \rightarrow$
 $R\ y\ z \rightarrow$
 `rtclosure'r` $R\ x\ z.$

Lemma `trans_rtclosure'r` : $\forall R,$
 `trans` (`rtclosure'r` R).

Proof using.

Hint Constructors `rtclosure'r`.

intros R y x z M1 M2. gen x. induction M2; introv M1; autos×.

Qed.

Lemma `rtclosure_eq_rtclosure'r` : $\forall R,$
 `rtclosure` $R = \text{rtclosure'r}$ $R.$

Proof using.

extens. intros x y. iff M.
 { *induction× M. applys× trans_rtclosure'r y. }*
 { *induction× M. }*

Qed.

Lemma `rtclosure_ind_r` : $\forall R (P : A \rightarrow A \rightarrow \text{Prop}),$
 $(\forall x, P\ x\ x) \rightarrow$
 $(\forall y\ x\ z, \text{rtclosure}\ R\ x\ y \rightarrow P\ x\ y \rightarrow R\ y\ z \rightarrow P\ x\ z) \rightarrow$
 $(\forall x\ y, \text{rtclosure}\ R\ x\ y \rightarrow P\ x\ y).$

Proof using.

*intros H1 H2 M. rewrite rtclosure_eq_rtclosure'r in *. induction× M.*

Qed.

End Ind.

Inversion principle with steps at head or tail

Lemma `rtclosure_inv_l` : $\forall R\ x\ z,$
 `rtclosure` $R\ x\ z \rightarrow$
 $(x = z) \vee (\exists y, R\ x\ y \wedge \text{rtclosure}\ R\ y\ z).$

Proof using. `intros R. applys× rtclosure_ind_l. Qed.`

Lemma `rtclosure_inv_r` : $\forall R\ x\ z,$

`rtclosure R x z` $\rightarrow$

$(x = z) \vee (\exists y, \text{rtclosure } R\ x\ y \wedge R\ y\ z).$

Proof using. `intros R. applys× rtclosure_ind_r. Qed.`

End `Rtclosure`.

`#[global]`

Hint Resolve `rtclosure_refl rtclosure_once`

`rtclosure_l rtclosure_r'` : `rtclosure`.

10.5.6 Symmetric-transitive closure

Inductive `stclosure A (R:binary A) : binary A` :=

| `stclosure_once` : $\forall x\ y,$

$R\ x\ y \rightarrow$

`stclosure R x y`

| `stclosure_sym` : $\forall x\ y,$

`stclosure R x y` $\rightarrow$

`stclosure R y x`

| `stclosure_trans` : $\forall y\ x\ z,$

`stclosure R x y` $\rightarrow$

`stclosure R y z` $\rightarrow$

`stclosure R x z`.

Section `Stclosure`.

Variables (A : Type).

Implicit Types R : binary A .

Hint Constructors `stclosure`.

Properties

Lemma `refl_stclosure` : $\forall R,$

`refl R` $\rightarrow$

`refl (stclosure R)`.

Proof using. `unfolds× refl. Qed.`

Lemma `sym_stclosure` : $\forall R,$

`sym (stclosure R)`.

Proof using. `unfolds sym. introv N. induction× N. Qed.`

Lemma `trans_stclosure` : $\forall R,$

`trans (stclosure R)`.

Proof using. `unfolds× trans. Qed.`

Lemma `total_stclosure` : $\forall R,$

`total R` $\rightarrow$

total (**stclosure** R).

Proof using.

unfolds total. *introv* H . *intros* $x\ y$. *destruct* $\times$ ($H\ x\ y$).

Qed.

Lemma stclosure_eq_of_sym_trans : $\forall\ R$,

sym $R \rightarrow$

trans $R \rightarrow$

stclosure $R = R$.

Proof using.

unfolds sym, trans. *introv* $H1\ H2$. *extens*. *iff* M .

{ *induction* M ; *subst* $\times$. }

{ *autos* $\times$. }

Qed.

Lemma stclosure_inverse_eq : $\forall\ R$,

stclosure (inverse R) = inverse (**stclosure** R).

Proof using.

unfolds inverse. *extens*. *intros* $x\ y$. *iff* M ; *induction* $\times\ M$.

Qed.

Constructors

Lemma stclosure_l : $\forall\ R\ y\ x\ z$,

$R\ x\ y \rightarrow$

stclosure $R\ y\ z \rightarrow$

stclosure $R\ x\ z$.

Proof using. *autos* $\times$. Qed.

Lemma stclosure_r : $\forall\ R\ y\ x\ z$,

stclosure $R\ x\ y \rightarrow$

$R\ y\ z \rightarrow$

stclosure $R\ x\ z$.

Proof using. *autos* $\times$. Qed.

Inclusion

Lemma rel_incl_stclosure : $\forall\ R$,

rel_incl R (**stclosure** R).

Proof using. *unfolds* $\times$ rel_incl. Qed.

Lemma covariant_stclosure : $\forall\ R1\ R2$,

rel_incl $R1\ R2 \rightarrow$

rel_incl (**stclosure** $R1$) (**stclosure** $R2$).

Proof using. *introv* $H\ M$. *induction* $\times\ M$. Qed.

Lemma rel_incl_stclosure_stclosure : $\forall\ R1\ R2$,

rel_incl $R1$ (**stclosure** $R2$) $\rightarrow$

rel_incl (**stclosure** $R1$) (**stclosure** $R2$).

Proof using. *intros* H M . *induction* $\times$ M . Qed.

End Stclosure.

#[*global*]

Hint Constructors **stclosure** : *stclosure*.

10.5.7 Reflexive-symmetric-transitive closure

Inductive **rstclosure** A (R :binary A) : binary A :=

| **rstclosure_once** : $\forall x y$,
 $R x y \rightarrow$
 rstclosure $R x y$
| **rstclosure_refl** : $\forall x$,
 rstclosure $R x x$
| **rstclosure_sym** : $\forall x y$,
 rstclosure $R x y \rightarrow$
 rstclosure $R y x$
| **rstclosure_trans** : $\forall y x z$,
 rstclosure $R x y \rightarrow$
 rstclosure $R y z \rightarrow$
 rstclosure $R x z$.

Section Rstclosure.

Variables (A : Type).

Implicit Types R : binary A .

Hint Constructors **rstclosure**.

Properties

Lemma **refl_rstclosure** : $\forall R$,
 refl (**rstclosure** R).

Proof using. *unfolds* $\times$ **refl**. Qed.

Lemma **sym_rstclosure** : $\forall R$,
 sym (**rstclosure** R).

Proof using. *unfolds* $\times$ **sym**. Qed.

Lemma **trans_rstclosure** : $\forall R$,
 trans (**rstclosure** R).

Proof using. *unfolds* $\times$ **trans**. Qed.

Lemma **total_rstclosure** : $\forall R$,
 total $R \rightarrow$
 total (**rstclosure** R).

Proof using.

unfolds **total**. *intros* H . *intros* $x y$. *destruct* $\times$ ($H x y$).

Qed.

Lemma `rstclosure_eq_of_refl_sym_trans` : $\forall R$,
`refl` $R \rightarrow$
`sym` $R \rightarrow$
`trans` $R \rightarrow$
`rstclosure` $R = R$.

Proof using.

`unfolds` `refl`, `sym`, `trans`. `introv` H . `extens`. `iff` M .
`{ induction` M ; `subst` $\times$. `}`
`{ auto. }`

Qed.

Lemma `rstclosure_inverse_eq` : $\forall R$,
`rstclosure` (`inverse` R) = `rstclosure` R .

Proof using.

`unfolds` `inverse`. `extens`. `intros` x y . `iff` M ; `induction` $\times$ M .

Qed.

Constructors

Lemma `rstclosure_l` : $\forall R$ y x z ,
 R x $y \rightarrow$
`rstclosure` R y $z \rightarrow$
`rstclosure` R x z .

Proof using. `autos` $\times$. Qed.

Lemma `rstclosure_r` : $\forall R$ y x z ,
`rstclosure` R x $y \rightarrow$
 R y $z \rightarrow$
`rstclosure` R x z .

Proof using. `autos` $\times$. Qed.

Inclusion

Lemma `rel_incl_rstclosure` : $\forall R$,
`rel_incl` R (`rstclosure` R).

Proof using. `unfolds` $\times$ `rel_incl`. Qed.

Lemma `rel_incl_inverse_rstclosure` : $\forall R$,
`rel_incl` (`inverse` R) (`rstclosure` R).

Proof using. `unfolds` $\times$ `rel_incl`, `inverse`. Qed.

Lemma `covariant_rstclosure` : $\forall R1$ $R2$,
`rel_incl` $R1$ $R2 \rightarrow$
`rel_incl` (`rstclosure` $R1$) (`rstclosure` $R2$).

Proof using. `introv` H M . `induction` $\times$ M . Qed.

Lemma `rel_incl_rstclosure_rstclosure` : $\forall R1$ $R2$,
`rel_incl` $R1$ (`rstclosure` $R2$) $\rightarrow$
`rel_incl` (`rstclosure` $R1$) (`rstclosure` $R2$).

Proof using. *intros* H M . *induction* $\times$ M . *Qed*.

Lemma *rel_incl_union_rstclosure* : $\forall$ $R1$ $R2$,
 $\text{rel\_incl } (\text{union } (\text{rstclosure } R1) (\text{rstclosure } R2))$
 $(\text{rstclosure } (\text{union } R1 R2))$.

Proof using.
hint *rel_incl_union_l*, *rel_incl_union_r*. *intros* $[M|M]$.
 $\{ \text{applies} \times \text{covariant\_rstclosure } R1. \}$
 $\{ \text{applies} \times \text{covariant\_rstclosure } R2. \}$

Qed.

End *Rstclosure*.

#[global]
Hint Constructors *rstclosure* : *rstclosure*.

10.5.8 Relationship between closures

Section *ClosuresRel*.
Variables $(A : \text{Type})$.
Implicit Types $R : \text{binary } A$.
Hint Constructors *rtclosure* *rscclosure* *stcclosure* *rstclosure*.

rclosure to *rtclosure*

Lemma *rtclosure_of_rclosure* : $\forall$ R x y ,
 $\text{rclosure } R$ x $y \rightarrow$
 $\text{rtclosure } R$ x y .

Proof using. *intros*. *destruct* $\times$ H . *Qed*.

Lemma *rel_incl_rclosure_rtclosure* : $\forall$ R ,
 $\text{rel\_incl } (\text{rclosure } R) (\text{rtclosure } R)$.

Proof using. *intros*. *applies* $\times$ *rtclosure_of_rclosure*. *Qed*.

tclosure to *rtclosure*

Lemma *rtclosure_of_tclosure* : $\forall$ R x y ,
 $\text{tclosure } R$ x $y \rightarrow$
 $\text{rtclosure } R$ x y .

Proof using. *intros*. *induction* $\times$ H . *Qed*.

Lemma *rel_incl_tclosure_rtclosure* : $\forall$ R ,
 $\text{rel\_incl } (\text{tclosure } R) (\text{rtclosure } R)$.

Proof using. *intros*. *applies* $\times$ *rtclosure_of_tclosure*. *Qed*.

rclosure to *rscclosure*

Lemma *rscclosure_of_rclosure* : $\forall$ R x y ,
 $\text{rclosure } R$ x $y \rightarrow$
 $\text{rscclosure } R$ x y .

Proof using. intros. destruct× H . Qed.

Lemma rel_incl_rclosure_rsclosure : $\forall R$,
 rel_incl (**rclosure** R) (**rsclosure** R).

Proof using. intros. *applies*× rsclosure_of_rclosure. Qed.

sclosure to **rsclosure**

Lemma rsclosure_of_sclosure : $\forall R\ x\ y$,
sclosure $R\ x\ y \rightarrow$
rsclosure $R\ x\ y$.

Proof using. intros. induction× H . Qed.

Lemma rel_incl_sclosure_rsclosure : $\forall R$,
 rel_incl (**sclosure** R) (**rsclosure** R).

Proof using. intros. *applies*× rsclosure_of_sclosure. Qed.

sclosure to **stclosure**

Lemma stclosure_of_sclosure : $\forall R\ x\ y$,
sclosure $R\ x\ y \rightarrow$
stclosure $R\ x\ y$.

Proof using. intros. induction× H . Qed.

Lemma rel_incl_sclosure_stclosure : $\forall R$,
 rel_incl (**sclosure** R) (**stclosure** R).

Proof using. intros. *applies*× stclosure_of_sclosure. Qed.

tclosure to **stclosure**

Lemma stclosure_of_tclosure : $\forall R\ x\ y$,
tclosure $R\ x\ y \rightarrow$
stclosure $R\ x\ y$.

Proof using. intros. induction× H . Qed.

Lemma rel_incl_tclosure_stclosure : $\forall R$,
 rel_incl (**tclosure** R) (**stclosure** R).

Proof using. intros. *applies*× stclosure_of_tclosure. Qed.

rclosure to **rstclosure**

Lemma rstclosure_of_rclosure : $\forall R\ x\ y$,
rclosure $R\ x\ y \rightarrow$
rstclosure $R\ x\ y$.

Proof using. intros. destruct× H . Qed.

Lemma rel_incl_rclosure_rstclosure : $\forall R$,
 rel_incl (**rclosure** R) (**rstclosure** R).

Proof using. intros. *applies*× rstclosure_of_rclosure. Qed.

sclosure to **rstclosure**

Lemma rstclosure_of_sclosure : $\forall R\ x\ y$,


```

sclosure  $R\ x\ y \rightarrow$ 
rstclosure  $R\ x\ y$ .
Proof using. intros. induction×  $H$ . Qed.

Lemma rel_incl_sclosure_rstclosure :  $\forall\ R$ ,
  rel_incl (sclosure  $R$ ) (rstclosure  $R$ ).
Proof using. intros. applies× rstclosure_of_sclosure. Qed.

tclosure to tstclosure

Lemma rstclosure_of_tclosure :  $\forall\ R\ x\ y$ ,
tclosure  $R\ x\ y \rightarrow$ 
rstclosure  $R\ x\ y$ .
Proof using. intros. induction×  $H$ . Qed.

Lemma rel_incl_tclosure_rstclosure :  $\forall\ R$ ,
  rel_incl (tclosure  $R$ ) (rstclosure  $R$ ).
Proof using. intros. applies× rstclosure_of_tclosure. Qed.

rscclosure to rstclosure

Lemma rstclosure_of_rscclosure :  $\forall\ R\ x\ y$ ,
rscclosure  $R\ x\ y \rightarrow$ 
rstclosure  $R\ x\ y$ .
Proof using. intros. induction×  $H$ . Qed.

Lemma rel_incl_rscclosure_rstclosure :  $\forall\ R$ ,
  rel_incl (rscclosure  $R$ ) (rstclosure  $R$ ).
Proof using. intros. applies× rstclosure_of_rscclosure. Qed.

rtclosure to rstclosure

Lemma rstclosure_of_rtclosure :  $\forall\ R\ x\ y$ ,
rtclosure  $R\ x\ y \rightarrow$ 
rstclosure  $R\ x\ y$ .
Proof using. intros. induction×  $H$ . Qed.

Lemma rel_incl_rtclosure_rstclosure :  $\forall\ R$ ,
  rel_incl (rtclosure  $R$ ) (rstclosure  $R$ ).
Proof using. intros. applies× rstclosure_of_rtclosure. Qed.

stclosure to rstclosure

Lemma rstclosure_of_stclosure :  $\forall\ R\ x\ y$ ,
stclosure  $R\ x\ y \rightarrow$ 
rstclosure  $R\ x\ y$ .
Proof using. intros. induction×  $H$ . Qed.

Lemma rel_incl_stclosure_rstclosure :  $\forall\ R$ ,
  rel_incl (stclosure  $R$ ) (rstclosure  $R$ ).
Proof using. intros. applies× rstclosure_of_stclosure. Qed.

End ClosuresRel.

```

10.5.9 Iterated closures

Section IterClosures.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

Hint Constructors **rclosure sclosure tclosure**

rtclosure rsclosure stclosure rstclosure.

Hint Resolve *sym_tclosure sym_sclosure sym_rclosure*
sym_rtclosure sym_rsclosure sym_rtclosure sym_rstclosure.

Lemma *rclosure_sclosure_eq_rsclosure* : $\forall R$,
rclosure (**sclosure** R) = **rsclosure** R .

Proof using.

extens. intros $x y$. *iff* M .
{ destruct $\times M$. { *applies* $\times$ *rsclosure_of_sclosure*. } }
{ induction $\times M$. { *applies* $\times$ *sym_inv*. } }

Qed.

Lemma *sclosure_rclosure_eq_rsclosure* : $\forall R$,
sclosure (**rclosure** R) = **rsclosure** R .

Proof using.

extens. intros $x y$. *iff* M .
{ induction $\times M$. { *applies* $\times$ *rsclosure_of_rclosure*. } }
{ induction $\times M$. }

Qed.

Lemma *rclosure_tclosure_eq_rtclosure* : $\forall R$,
rclosure (**tclosure** R) = **rtclosure** R .

Proof using.

extens. intros $x y$. *iff* M .
{ induction $\times M$. { *applies* $\times$ *rtclosure_of_tclosure*. } }
{ induction $\times M$. { destruct *IHM1*; destruct *IHM2*; *autos* $\times$. } }

Qed.

Lemma *tclosure_rclosure_eq_rtclosure* : $\forall R$,
tclosure (**rclosure** R) = **rtclosure** R .

Proof using.

extens. intros $x y$. *iff* M .
{ induction $\times M$. { *applies* $\times$ *rtclosure_of_rclosure*. } }
{ induction $\times M$. }

Qed.

Lemma *tclosure_sclosure_eq_stclosure* : $\forall R$,
tclosure (**sclosure** R) = **stclosure** R .

Proof using.

extens. intros $x y$. *iff* M .

```

{ induction× M. { applies× stclosure_of_sclosure. } }
{ induction× M. { applies× sym_inv. } }

```

Qed.

Lemma rclosure_stclosure_eq_rstclosure : $\forall R$,
rclosure (**stclosure** *R*) = **rstclosure** *R*.

Proof using.

```

extends. intros x y. iff M.
{ induction× M. { applies× rstclosure_of_stclosure. } }
{ induction× M.
  { destruct IHM; autos×. }
  { destruct IHM1; destruct IHM2; autos×. } }

```

Qed.

Lemma stclosure_rclosure_eq_rstclosure : $\forall R$,
stclosure (**rclosure** *R*) = **rstclosure** *R*.

Proof using.

```

extends. intros x y. iff M.
{ induction× M. { applies× rstclosure_of_rclosure. } }
{ induction× M. }

```

Qed.

Lemma rtclosure_sclosure_eq_rstclosure : $\forall R$,
rtclosure (**sclosure** *R*) = **rstclosure** *R*.

Proof using.

```

extends. intros x y. iff M.
{ induction× M. { applies× rstclosure_of_sclosure. } }
{ induction× M. { applies× sym_inv. } }

```

Qed.

Lemma tclosure_rsclosure_eq_rstclosure : $\forall R$,
tclosure (**rsclosure** *R*) = **rstclosure** *R*.

Proof using.

```

extends. intros x y. iff M.
{ induction× M. { applies× rstclosure_of_rsclosure. } }
{ induction× M. { applies× sym_inv. } }

```

Qed.

End IterClosures.

10.5.10 Other lemmas – TODO

Section EquivClosures.

Variables (*A* : Type).

Implicit Types *R* : binary *A*.

Hint Constructors **rclosure** **sclosure** **tclosure** **rsclosure** **stclosure**.

Lemma `rscclosure_eq_union_rclosure_sclosure` : $\forall R$,
 $\text{rscclosure } R = \text{union } (\text{rclosure } R) (\text{sclosure } R).$

Proof using.

extens. intros $x y$. unfold union. *iff* M .
 $\{ \text{induction} \times M. \text{destruct} \times IHM \text{ as } [H|H]. \text{destruct} \times H. \}$
 $\{ \text{destruct} \times M \text{ as } [H|H]. \text{destruct} \times H. \text{applies} \times \text{rscclosure\_of\_sclosure}. \}$

Qed.

Lemma `rtcclosure_eq_union_rclosure_tclosure` : $\forall R$,
 $\text{rtcclosure } R = \text{union } (\text{rclosure } R) (\text{tclosure } R).$

Proof using.

extens. intros $x y$. unfold union. *iff* M .
 $\{ \text{induction} \times M. \text{destruct } IHM1 \text{ as } [H1|H1]; \text{destruct } IHM2 \text{ as } [H2|H2].$
 $\{ \text{destruct } H1; \text{destruct} \times H2. \}$
 $\{ \text{destruct} \times H1. \}$
 $\{ \text{destruct} \times H2. \}$
 $\{ \text{autos} \times. \} \}$
 $\{ \text{destruct} \times M.$
 $\{ \text{applies} \times \text{rtcclosure\_of\_rclosure}. \}$
 $\{ \text{applies} \times \text{rtcclosure\_of\_tclosure}. \} \}$

Qed.

Lemma `rtcclosure_inv_rclosure_or_tclosure` : $\forall R x y$,
 $\text{rtcclosure } R x y \rightarrow$
 $x = y \vee \text{tclosure } R x y.$

Proof using.

introv M . rewrite `rtcclosure_eq_union_rclosure_tclosure` in M .
 $\text{destruct } M \text{ as } [M|M]. \{ \text{destruct} \times M. \} \{ \text{auto}. \}$

Qed.

Lemma `stcclosure_eq_rstcclosure_of_refl` : $\forall R$,
 $\text{refl } R \rightarrow$
 $\text{stcclosure } R = \text{rstcclosure } R.$

Proof using.

introv H . *extens.* intros $x y$. *iff* M .
 $\{ \text{applies} \times \text{rstcclosure\_of\_stcclosure}. \}$
 $\{ \text{induction} \times M. \}$

Qed.

Lemma `rel_incl_tclosure_stcclosure_l` : $\forall R1 R2$,
 $\text{rel\_incl } R1 (\text{stcclosure } R2) \rightarrow$
 $\text{rel\_incl } (\text{tclosure } R1) (\text{stcclosure } R2).$

Proof using. *introv* $H M$. *induction* $\times M$. Qed.

End EquivClosures.

10.5.11 Mixed transitivity between closures

Section MixedClosures.

Variables ($A : \text{Type}$).

Implicit Types $R : \text{binary } A$.

Hint Constructors **tclosure**.

Lemma `tclosure_of_rtclosure_l` : $\forall R\ x\ y\ z,$

rtclosure $R\ x\ y \rightarrow$

$R\ y\ z \rightarrow$

tclosure $R\ x\ z$.

Proof using.

intros $H\ M$. *destruct* (`rtclosure_inv_rclosure_or_tclosure` H).

{ *subst* $\times$. }

{ *applies* $\times$ `tclosure_r`. }

Qed.

Lemma `tclosure_of_rtclosure_r` : $\forall R\ x\ y\ z,$

$R\ x\ y \rightarrow$

rtclosure $R\ y\ z \rightarrow$

tclosure $R\ x\ z$.

Proof using.

intros $M\ H$. *destruct* (`rtclosure_inv_rclosure_or_tclosure` H).

{ *subst* $\times$. }

{ *applies* $\times$ `tclosure_l`. }

Qed.

Lemma `tclosure_of_rtclosure_tclosure` : $\forall R\ y\ x\ z,$

rtclosure $R\ x\ y \rightarrow$

tclosure $R\ y\ z \rightarrow$

tclosure $R\ x\ z$.

Proof using.

intros $H\ M$. *destruct* (`rtclosure_inv_rclosure_or_tclosure` H); *subst* $\times$.

Qed.

Lemma `tclosure_of_tclosure_rtclosure` : $\forall R\ y\ x\ z,$

tclosure $R\ x\ y \rightarrow$

rtclosure $R\ y\ z \rightarrow$

tclosure $R\ x\ z$.

Proof using.

intros $M\ H$. *destruct* (`rtclosure_inv_rclosure_or_tclosure` H); *subst* $\times$.

Qed.

End MixedClosures.

10.5.12 Irreflexive restriction of a relation

Definition `strict` A (R :binary A) : binary A :=

`fun` x $y \Rightarrow R$ x $y \wedge x \neq y$.

Section `Strict`.

Variables (A : Type).

Implicit Types R : binary A .

Hint `Unfold strict`.

Lemma `strict_eq_fun` : $\forall R$,

`strict` $R = (\text{fun } x \ y \Rightarrow R \ x \ y \wedge x \neq y)$.

Proof using. `auto`. Qed.

Lemma `strict_eq` : $\forall R \ x \ y$,

`strict` $R \ x \ y = (R \ x \ y \wedge x \neq y)$.

Proof using. `auto`. Qed.

Lemma `inverse_strict` : $\forall R$,

`inverse` (`strict` R) = `strict` (`inverse` R).

Proof using. `intros`. `unfold inverse, strict`. *extens*×. Qed.

Lemma `trans_strict_l` : $\forall y \ x \ z \ R$,

`trans` $R \rightarrow$
`strict` $R \ x \ y \rightarrow$
 $R \ y \ z \rightarrow$
 $R \ x \ z$.

Proof using. *introv* T ($E \& H$) H' ; `subst`×. Qed.

Lemma `trans_strict_r` : $\forall y \ x \ z \ R$,

`trans` $R \rightarrow$
 $R \ x \ y \rightarrow$
`strict` $R \ y \ z \rightarrow$
 $R \ x \ z$.

Proof using. *introv* $T \ H$ ($E \& H'$); `subst`×. Qed.

Lemma `irrefl_strict` : $\forall R$,

`irrefl` (`strict` R).

Proof using. `unfold strict, irrefl`. `intros`. *rew_logic*×. Qed.

Lemma `antisym_strict` : $\forall R$,

`antisym` $R \rightarrow$
`antisym` (`strict` R).

Proof using. *unfolds*× `antisym, strict`. Qed.

Lemma `trans_strict` : $\forall R$,

`trans` $R \rightarrow$
`antisym` $R \rightarrow$
`trans` (`strict` R).

Proof using.

```
  introv  $T$   $S$ . unfold strict. introv [ $H1$   $H2$ ] [ $H3$   $H4$ ]. split.
  { apply×  $T$ . }
  { intros  $K$ . subst. apply  $H2$ . apply¬  $S$ . }
```

Qed.

Lemma strict_rclosure : $\forall R$,
 irrefl $R \rightarrow$
 strict (**rclosure** R) = R .

Proof using.

```
  unfold strict. extens. intros  $x$   $y$ . iff ( $K1 \& K2$ )  $K$ .
  { destruct×  $K1$ . }
  { split. { left×. } { apply× irrefl_inv_neq. } }
```

Qed.

Lemma rclosure_strict : $\forall R$,
 refl $R \rightarrow$
 rclosure (strict R) = R .

Proof using.

```
  Hint Constructors rclosure.
  unfold strict. extens. intros  $x$   $y$ . iff  $K$ .
  { destruct  $K$ ; subst×. }
  { tests: ( $x = y$ ); subst×. }
```

Qed.

End Strict.

10.5.13 Inclusion of a function in a relation

`fun_in_rel` f R asserts that input-output pairs of f are included in the relation R .

Definition `fun_in_rel` A B ($f:A \rightarrow B$) ($R:A \rightarrow B \rightarrow \text{Prop}$) :=
 $\forall x, R\ x\ (f\ x)$.

Section `Fun_in_rel`.

Variables (A B : Type).

Implicit Type $f : A \rightarrow B$.

Implicit Type $R : A \rightarrow B \rightarrow \text{Prop}$.

Lemma `defined_of_fun_in_rel` : $\forall f\ R$,
 `fun_in_rel` $f\ R \rightarrow$
 `defined` R .

Proof using. `unfolds`× `fun_in_rel`, `defined`. Qed.

The relation built from a function f is included in a relation R iff the function f is included in R

Lemma `rel_incl_rel_fun_eq_fun_in_rel` : $\forall f\ R$,

```

    rel_incl (rel_fun f) R = fun_in_rel f R.
Proof using.
  extends. unfold rel_fun, fun_in_rel. iff H; intros x; specializes H x.
  { applies × H. }
  { intros y Hy. subst ⊥. }
Qed.
End Fun_in_rel.

```

10.5.14 Inclusion of a relation in a function

`rel_in_fun R f` asserts that input-output pairs of R are input-output for f .

Definition `rel_in_fun A B (R:A→B→Prop) (f:A→B) :=`
 $\forall x\ y, R\ x\ y \rightarrow f\ x = y.$

Section `Rel_in_fun`.

Variables $(A\ B : \text{Type}).$

Implicit Type $f : A \rightarrow B.$

Implicit Type $R : A \rightarrow B \rightarrow \text{Prop}.$

Lemma `functional_of_rel_in_fun` : $\forall R\ f,$
 $\text{rel\_in\_fun } R\ f \rightarrow$
 $\text{functional } R.$

Proof using.

```

  unfold rel_in_fun, functional. introv M N1 N2.
  lets: M N1. lets: M N2. congruence.

```

Qed.

Lemma `rel_in_fun_of_fun_in_rel_functional` : $\forall f\ R,$
 $\text{fun\_in\_rel } f\ R \rightarrow$
 $\text{functional } R \rightarrow$
 $\text{rel\_in\_fun } R\ f.$

Proof using.

```

  introv h1 h2. intros a b H. forwards M: h1 a. forwards*: h2 H M.

```

Qed.

End `Rel_in_fun`.

Chapter 11

Library SLF.LibOrder

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibReflect LibOperation LibRelation.

Generalizable Variables *A*.

Definition

```
Record preorder A (R:binary A) : Prop := {  
  preorder_refl : refl R;  
  preorder_trans : trans R }.
```

Arguments preorder_trans [*A*] [*R*] [*p*] *y* [*x*] [*z*].

Transformations

```
Lemma preorder_inverse : ∀ A (R:binary A),  
  preorder R →  
  preorder (inverse R).
```

Proof using. *hint* trans_inverse. *intros* [*Re Tr*]. constructor $\neg$. Qed.

```
Lemma preorder_rclosure : ∀ A (R:binary A),  
  preorder R →  
  preorder (rclosure R).
```

Proof using. *hint* refl_rclosure, trans_rclosure. *intros* [*Re Tr*]. constructor $\neg$. Qed.

Definition of total preorder relations

```
Record total_preorder A (R:binary A) : Prop := {  
  total_preorder_trans : trans R;  
  total_preorder_total : total R }.
```

Arguments total_preorder_trans [*A*] [*R*] *t y* [*x*] [*z*].

Conversion to preorder

```
Lemma total_preorder_refl : ∀ A (le:binary A),  
  total_preorder le →
```

```

    refl le.
Proof using. introv [Tr To]. intros x. destruct¬ (To x x). Qed.
#[global]
Hint Resolve total_preorder_refl.

Coercion total_preorder_to_preorder A (R:binary A)
  (O:total_preorder R) : preorder R.
Proof using. lets [M _]: O. constructor¬. Qed.
#[global]
Hint Resolve total_preorder_to_preorder.

Transformations

Lemma total_preorder_inverse : ∀ A (R:binary A),
  total_preorder R →
  total_preorder (inverse R).
Proof using. hint trans_inverse, total_inverse. introv [Tr To]. constructor¬. Qed.

Lemma total_preorder_rclosure : ∀ A (R:binary A),
  total_preorder R →
  total_preorder (rclosure R).
Proof using. hint trans_rclosure, total_rclosure. introv [Re Tr]. constructor¬. Qed.

Properties

Lemma inverse_of_not : ∀ A (R:binary A) x y,
  total R →
  ¬ R x y →
  inverse R x y.
Proof using. introv T H. destruct (T x y); auto_false¬. Qed.

Lemma inverse_strict_of_not : ∀ A (R:binary A) x y,
  total R →
  ¬ R x y →
  inverse (strict R) x y.
Proof using.
  introv T H. destruct (T x y). auto_false¬.
  hnf. split¬. intro_subst¬.
Qed.

Definition

Record order A (R:binary A) : Prop := {
  order_refl : refl R;
  order_trans : trans R;
  order_antisym : antisym R }.

Arguments order_trans [A] [R] [o] y [x] [z].
Arguments order_antisym [A] [R] [o] [x] [y].

```

Conversion to preorder

```
Coercion order_to_preorder A (R:binary A)
  (O:order R) : preorder R.
Proof using. destruct× O. constructors×. Qed.
#[global]
Hint Resolve order_to_preorder.
```

Transformations

```
Lemma order_inverse : ∀ A (R:binary A),
  order R →
  order (inverse R).
Proof using.
  hint trans_inverse, antisym_inverse.
  introv [Re Tr An]. constructor¬.
Qed.
```

```
Lemma order_rclosure : ∀ A (R:binary A),
  order R →
  order (rclosure R).
Proof using.
  hint refl_rclosure, trans_rclosure, antisym_rclosure.
  introv [Re Tr An]. constructor¬.
Qed.
```

Properties

11.1 Order relation upto an equivalence relation

Note: this is used in LibFix

```
Record order_wrt (A:Type) (E:binary A) (R:binary A) : Prop := {
  order_wrt_refl : refl R;
  order_wrt_trans : trans R;
  order_wrt_antisym : antisym_wrt E R }.
Arguments order_wrt_trans [A] [E] [R] [o] y [x] [z].
Arguments order_wrt_antisym [A] [E] [R] [o] [x] [y].
```

Conversion to preorder

```
Coercion order_wrt_to_preorder A (E:binary A) (R:binary A)
  (O:order_wrt E R) : preorder R.
Proof using. destruct× O. constructors×. Qed.
#[global]
Hint Resolve order_wrt_to_preorder.
```

Transformations

Lemma `order_wrt_inverse` : $\forall A (E:\text{binary } A) (R:\text{binary } A),$
 `order_wrt` $E R \rightarrow$
 `order_wrt` $E (\text{inverse } R).$

Proof using.

$\text{hint trans\_inverse, antisym\_wrt\_inverse.}$

$\text{intros [Re Tr An]. constructor}\neg.$

Qed.

Lemma `order_wrt_rclosure` : $\forall A (E:\text{binary } A) (R:\text{binary } A),$
 `order_wrt` $E R \rightarrow$
 `order_wrt` $(\text{rclosure } E) (\text{rclosure } R).$

Proof using.

$\text{hint refl\_rclosure, trans\_rclosure, antisym\_wrt\_rclosure.}$

$\text{intros [Re Tr An]. constructor}\neg.$

Qed.

Properties

Definition

Record `total_order` $A (R:\text{binary } A) : \text{Prop} := \{$
 `total_order_order` : `order` R ;
 `total_order_total` : `total` R }.

Projections

Definition `total_order_refl` := `order_refl`.

Definition `total_order_trans` := `order_trans`.

Definition `total_order_antisym` := `order_antisym`.

Arguments `total_order_trans` $[A] [R] [o] y [x] [z].$

Arguments `total_order_antisym` $[A] [R] [o] [x] [y].$

Construction

Lemma `total_order_intro` : $\forall A (R:\text{binary } A),$
 `trans` $R \rightarrow$
 `antisym` $R \rightarrow$
 `total` $R \rightarrow$
 `total_order` $R.$

Proof using.

$\text{intros Tra Ant Tot. constructor}\neg. \text{constructor}\neg.$

$\text{intros\_all. destruct}\neg (Tot\ x\ x).$

Qed.

Conversion to order

Coercion `total_order_to_total_preorder` $A (R:\text{binary } A)$
 $(O:\text{total\_order } R) : \text{total\_preorder } R.$

Proof using. *destruct*× *O. constructors*×. *applies*× *order_trans*. Qed.

Definition *total_order_to_order* := *total_order_order*.

#[*global*]

Hint Resolve *total_order_to_order total_order_to_total_preorder*.

Transformations

Lemma *total_order_inverse* : $\forall A (R:\text{binary } A),$
 total_order *R* $\rightarrow$
 total_order (*inverse* *R*).

Proof using.

hint *total_inverse*, *order_inverse*.

introv [*Or To*]. *constructor*¬.

Qed.

Lemma *total_order_rclosure* : $\forall A (R:\text{binary } A),$
 total_order *R* $\rightarrow$
 total_order (*rclosure* *R*).

Proof using.

hint *total_rclosure*, *order_rclosure*.

introv [*Or To*]. *constructor*¬.

Qed.

Properties

Section *TotalOrderProp*.

Variables (*A*:Type) (*R* : binary *A*).

WARNING: notations here are not typeclass operators TODO: is this really what we want? perhaps it would be clearer to inline these notations.

Notation "'le'" := (*R*).

Notation "'ge'" := (*inverse* *R*).

Notation "'lt'" := (*strict* *R*).

Notation "'gt'" := (*inverse* *lt*).

Ltac *total_order_normalize* :=
 repeat rewrite *rclosure_eq_fun*;
 repeat rewrite *inverse_eq_fun*;
 repeat rewrite *strict_eq_fun*.

Lemma *total_order_le_is_rclosure_lt* : $\forall (To:\text{total\_order } R),$
 le = *rclosure* *lt*.

Proof using.

extens. *intros*. *total_order_normalize*. *iff* *M*.

tests~: (*x* = *y*).

destruct *M*. *autos*×. *subst*×. *dintuition* *eauto*.

Qed.

```

Lemma total_order_lt_is_strict_le :  $\forall$  (To:total_order R),
  lt = strict le.
Proof using.
  auto.
Qed.

Lemma total_order_ge_is_rclosure_gt :  $\forall$  (To:total_order R),
  ge = rclosure gt.
Proof using.
  extens. intros. total_order_normalize. iff M.
  tests~: (x = y).
  destruct M. autos×. subst×. dintuition eauto.
Qed.

Lemma total_order_gt_is_strict_ge :  $\forall$  (To:total_order R),
  gt = strict ge.
Proof using.
  extens. intros. total_order_normalize. iff M.
  tests~: (x = y).
  destruct M. autos×.
  destruct M. autos×.
Qed.

Lemma total_order_lt_or_eq_or_gt :  $\forall$  (To:total_order R) x y,
  lt x y  $\vee$  x = y  $\vee$  gt x y.
Proof using.
  introv H. intros. total_order_normalize. tests: (x = y).
  branch¬ 2.
  destruct (total_order_total H x y).
  branch¬ 1.
  branch¬ 3.
Qed.

Lemma total_order_lt_or_ge :  $\forall$  (To:total_order R) x y,
  lt x y  $\vee$  ge x y.
Proof using.
  intros. branches (total_order_lt_or_eq_or_gt To x y).
  left¬.
  right¬. subst. hnf. apply¬ total_order_refl.
  right¬. rewrite¬ total_order_ge_is_rclosure_gt.
  total_order_normalize. hnfs¬.
Qed.

Lemma total_order_le_or_gt :  $\forall$  (To:total_order R) x y,
  le x y  $\vee$  gt x y.
Proof using.

```

```

intros. branches¬ (total_order_lt_or_eq_or_gt To x y).
left¬. rewrite¬ total_order_le_is_rclosure_lt. total_order_normalize. hnfs¬.
left¬. subst. apply¬ total_order_refl.
Qed.
End TotalOrderProp.

```

Definition

```

Record strict_order A (R:binary A) : Prop := {
  strict_order_irrefl : irrefl R;
  strict_order_asym : asym R;
  strict_order_trans : trans R }.

```

```

Arguments strict_order_trans [A] [R] [s] y [x] [z].

```

Transformations

```

Lemma strict_order_inverse : ∀ A (R:binary A),
  strict_order R →
  strict_order (inverse R).

```

Proof using.

```

hint antisym_inverse, trans_inverse, asym_inverse.
introv [Ir As Tr]. constructor¬.

```

Qed.

```

Lemma strict_order_strict : ∀ A (R:binary A),
  order R →
  strict_order (strict R).

```

Proof using.

```

introv [Re As Tr]. unfold strict. constructor; intros_all; simpls.
destruct× H.
applies× antisym_inv x y.
split. applies× As. intros E. subst. applies× antisym_inv y z.

```

Qed.

```

Lemma order_rclosure_of_strict_order : ∀ A (R:binary A),
  strict_order R →
  order (rclosure R).

```

Proof using.

```

introv [Re As Tr]. rewrite rclosure_eq_fun. constructor; simpl.
intros_all¬.
introv [H1|E1] [H2|E2]; subst; auto.
  left. apply× trans_inv.
introv [H1|E1] [H2|E2]; try subst; auto.
  false. apply× As.

```

Qed.

Definition

```

Record strict_total_order A (R:binary A) : Prop := {
  strict_total_order_trans : trans R;
  strict_total_order_trichotomous : trichotomous R }.

Arguments strict_total_order_trans [A] [R] [s] y [x] [z].

Conversion to strict order and back

Lemma strict_total_order_irrefl : ∀ A (R:binary A),
  strict_total_order R →
  irrefl R.
Proof using. intros [Tr Tk]. intros x. lets: (Tk x x). inverts⊃ H. Qed.

Lemma strict_total_order_asym : ∀ A (R:binary A),
  strict_total_order R →
  asym R.
Proof using. intros [Tr Tk]. intros x y. lets: (Tk x y). inverts⊃ H. Qed.

Coercion strict_total_order_to_strict_order A (R:binary A)
  (O:strict_total_order R) : strict_order R.
Proof using.
  lets [M _]: O. constructor;
  autos⊃ strict_total_order_irrefl strict_total_order_asym.
Qed.

#[global]
Hint Resolve strict_total_order_to_strict_order.

Transformation

Lemma strict_total_order_inverse : ∀ A (R:binary A),
  strict_total_order R →
  strict_total_order (inverse R).
Proof using.
  intros [Tr Tk]. constructor. apply⊃ trans_inverse.
  apply⊃ trichotomous_inverse.
Qed.

From total order

Lemma strict_total_order_of_total_order : ∀ A (R:binary A),
  total_order R →
  strict_total_order (strict R).
Proof using.
  intros [[Re Tr As] To]. constructor.
  apply⊃ trans_strict.
  intros x y. tests: (x = y).
  subst. apply trichotomy_eq. unfolds× strict.
  unfold strict. destruct (To x y).
  apply× trichotomy_left.

```



```

    apply× trichotomy_right.
Qed.

```

11.2 Definition of order operators

11.2.1 Classes and notation for comparison operators

Operators

```

Class Le (A : Type) := { le : binary A }.
Class Ge (A : Type) := { ge : binary A }.
Class Lt (A : Type) := { lt : binary A }.
Class Gt (A : Type) := { gt : binary A }.

```

```

Global Opaque le lt ge gt.

```

Structures

```

Class Le_preorder '{Le A} : Prop :=
  { le_preorder : preorder le }.
Class Le_total_preorder '{Le A} : Prop :=
  { le_total_preorder : total_preorder le }.
Class Le_order '{Le A} : Prop :=
  { le_order : order le }.
Class Le_total_order '{Le A} : Prop :=
  { le_total_order : total_order le }.
Class Lt_strict_order '{Lt A} : Prop :=
  { lt_strict_order : strict_order lt }.
Class Lt_strict_total_order '{Lt A} : Prop :=
  { lt_strict_total_order : strict_total_order lt }.

```

Notation

```

Declare Scope comp_scope.
Notation "x <= y" := (le x y)
  (at level 70, no associativity) : comp_scope.
Notation "x >= y" := (ge x y)
  (at level 70, no associativity) : comp_scope.
Notation "x < y" := (lt x y)
  (at level 70, no associativity) : comp_scope.
Notation "x > y" := (gt x y)
  (at level 70, no associativity) : comp_scope.
Open Scope comp_scope.
Notation "x <= y <= z" := (x ≤ y ∧ y ≤ z)

```

```

    (at level 70, y at next level) : comp_scope.
Notation "x <= y < z" := (x ≤ y ∧ y < z)
    (at level 70, y at next level) : comp_scope.
Notation "x < y <= z" := (x < y ∧ y ≤ z)
    (at level 70, y at next level) : comp_scope.
Notation "x < y < z" := (x < y ∧ y < z)
    (at level 70, y at next level) : comp_scope.

```

11.2.2 The operators *ge*, *lt* and *gt* are deduced from *le*

```

#[global]
Instance ge_of_le : ∀ ' { Le A }, Ge A.
  constructor. apply (inverse le). Defined.
#[global]
Instance lt_of_le : ∀ ' { Le A }, Lt A.
  constructor. apply (strict le). Defined.
#[global]
Instance gt_of_le : ∀ ' { Le A }, Gt A.
  constructor. apply (inverse lt). Defined.

Lemma ge_is_inverse_le : ∀ ' { Le A }, ge = inverse le.
Proof using. extens×. Qed.

Lemma lt_is_strict_le : ∀ ' { Le A }, lt = strict le.
Proof using. extens×. Qed.

Lemma gt_is_inverse_lt : ∀ ' { Le A }, gt = inverse lt.
Proof using. extens×. Qed.

Lemma gt_is_inverse_strict_le : ∀ ' { Le A }, gt = inverse (strict le).
Proof using. extens. intros. rewrite gt_is_inverse_lt. rewrite× lt_is_strict_le. Qed.

Global Opaque ge_of_le lt_of_le gt_of_le.

  Local tactic rew_to_le

#[global]
Hint Rewrite @gt_is_inverse_strict_le @ge_is_inverse_le @lt_is_strict_le : rew_to_le.

Tactic Notation "rew_to_le" :=
  autorewrite with rew_to_le in *.

#[global]
Hint Rewrite @ge_is_inverse_le @gt_is_inverse_lt : rew_to_le_lt.

Tactic Notation "rew_to_le_lt" :=
  autorewrite with rew_to_le_lt in *.

Lemma gt_is_strict_inverse_le : ∀ ' { Le A },
  gt = strict (inverse le).

```

Proof using. intros. *rew_to_le*. apply inverse_strict. Qed.

Lemma le_is_rclosure_lt : $\forall \{Le\ A\},$
 refl le $\rightarrow$
 le = rclosure lt.

Proof using. intros. *rew_to_le*. rewrite $\neg$ rclosure_strict. Qed.

Lemma le_is_inverse_ge : $\forall \{Le\ A\},$
 le = inverse ge.

Proof using. intros. *rew_to_le*. rewrite $\neg$ inverse_inverse. Qed.

Lemma lt_is_inverse_gt : $\forall \{Le\ A\},$
 lt = inverse gt.

Proof using. intros. *rew_to_le*. rewrite $\neg$ inverse_inverse. Qed.

Lemma gt_is_strict_ge : $\forall \{Le\ A\},$
 gt = strict ge.

Proof using. intros. *rew_to_le*. apply inverse_strict. Qed.

Lemma ge_is_rclosure_gt : $\forall \{Le\ A\},$
 refl le $\rightarrow$
 ge = rclosure gt.

Proof using. intros. rewrite gt_is_strict_ge. rewrite $\neg$ rclosure_strict. Qed.

11.3 Classes for comparison properties

11.3.1 Definition of classes

symmetric structure

```

Class Ge_preorder '{Le A} : Prop :=
  { ge_preorder : preorder ge }.
Class Ge_total_preorder '{Le A} : Prop :=
  { ge_total_preorder : total_preorder ge }.
Class Ge_order '{Le A} : Prop :=
  { ge_order : order ge }.
Class Ge_total_order '{Le A} : Prop :=
  { ge_total_order : total_order le }.
Class Gt_strict_order '{Le A} : Prop :=
  { gt_strict_order : strict_order gt }.
Class Gt_strict_total_order '{Le A} : Prop :=
  { gt_strict_total_order : strict_total_order gt }.

properties of le
Class Le_refl '{Le A} :=
  { le_refl : refl le }.
Class Le_trans '{Le A} :=

```

```

    { le_trans : trans le }.
Class Le_antisym '{Le A} :=
    { le_antisym : antisym le }.
Class Le_total '{Le A} :=
    { le_total : total le }.

properties of ge

Class Ge_refl '{Ge A} :=
    { ge_refl : refl ge }.
Class Ge_trans '{Ge A} :=
    { ge_trans : trans ge }.
Class Ge_antisym '{Ge A} :=
    { ge_antisym : antisym ge }.
Class Ge_total '{Ge A} :=
    { ge_total : total ge }.

properties of lt

Class Lt_irrefl '{Lt A} :=
    { lt_irrefl : irrefl lt }.
Class Lt_trans '{Lt A} :=
    { lt_trans : trans lt }.

properties of gt

Class Gt_irrefl '{Gt A} :=
    { gt_irrefl : irrefl gt }.
Class Gt_trans '{Gt A} :=
    { gt_trans : trans gt }.

mixed transitivity results

Class Lt_le_trans '{Le A} :=
    { lt_le_trans :  $\forall y\ x\ z, x < y \rightarrow y \leq z \rightarrow x < z$  }.
Class Le_lt_trans '{Le A} :=
    { le_lt_trans :  $\forall y\ x\ z, x \leq y \rightarrow y < z \rightarrow x < z$  }.
Class Gt_ge_trans '{Le A} :=
    { gt_ge_trans :  $\forall y\ x\ z, x > y \rightarrow y \geq z \rightarrow x > z$  }.
Class Ge_gt_trans '{Le A} :=
    { ge_gt_trans :  $\forall y\ x\ z, x \geq y \rightarrow y > z \rightarrow x > z$  }.

Arguments lt_irrefl [A] [H] [Lt_irrefl].
Arguments le_trans {A} {H} {Le_trans} y [x] [z].
Arguments ge_trans {A} {H} {Ge_trans} y [x] [z].
Arguments lt_trans {A} {H} {Lt_trans} y [x] [z].
Arguments gt_trans {A} {H} {Gt_trans} y [x] [z].

conversion between operators

```

```

Class Ge_as_sle '{Le A} : Prop :=
  { ge_as_sle :  $\forall x y : A, (x \geq y) = (y \leq x)$  }.

Class Gt_as_slt '{Le A} : Prop :=
  { gt_as_slt :  $\forall x y : A, (x > y) = (y < x)$  }.

Class Ngt_as_sle '{Le A} : Prop :=
  { ngt_as_sle :  $\forall x y : A, (\neg x < y) = (y \leq x)$  }.

Class Nlt_as_ge '{Le A} : Prop :=
  { nlt_as_ge :  $\forall x y : A, (\neg x < y) = (x \geq y)$  }.

Class Ngt_as_le '{Le A} : Prop :=
  { ngt_as_le :  $\forall x y : A, (\neg x > y) = (x \leq y)$  }.

Class Nle_as_gt '{Le A} : Prop :=
  { nle_as_gt :  $\forall x y : A, (\neg x \leq y) = (x > y)$  }.

Class Nge_as_lt '{Le A} : Prop :=
  { nge_as_lt :  $\forall x y : A, (\neg x \geq y) = (x < y)$  }.

  inclusion between operators

Class Lt_to_le '{Le A} : Prop :=
  { lt_to_le :  $\forall x y : A, (x < y) \rightarrow (x \leq y)$  }.

Class Gt_to_ge '{Le A} : Prop :=
  { gt_to_ge :  $\forall x y : A, (x > y) \rightarrow (x \geq y)$  }.

Class Nle_to_sle '{Le A} : Prop :=
  { nle_to_sle :  $\forall x y : A, (\neg x \leq y) \rightarrow (y \leq x)$  }.

Class Nle_to_slt '{Le A} : Prop :=
  { nle_to_slt :  $\forall x y : A, (\neg x \leq y) \rightarrow (y < x)$  }.

  case analysis

Class Case_eq_lt_gt '{Le A} : Prop :=
  { case_eq_lt_gt :  $\forall x y : A, x = y \vee x < y \vee x > y$  }.

Class Case_eq_lt_slt '{Le A} : Prop :=
  { case_eq_lt_slt :  $\forall x y : A, x = y \vee x < y \vee y < x$  }.

Class Case_le_gt '{Le A} : Prop :=
  { case_le_gt :  $\forall x y : A, x \leq y \vee x > y$  }.

Class Case_le_slt '{Le A} : Prop :=
  { case_le_slt :  $\forall x y : A, x \leq y \vee y < x$  }.

Class Case_lt_ge '{Le A} : Prop :=
  { case_lt_ge :  $\forall x y : A, x < y \vee x \geq y$  }.

Class Case_lt_sle '{Le A} : Prop :=
  { case_lt_sle :  $\forall x y : A, x < y \vee y \leq x$  }.

  case analysis under one assumption

```

```

Class Neq_case_lt_gt '{Le A} : Prop :=
  { neq_case_lt_gt :  $\forall x y : A, x \neq y \rightarrow x < y \vee x > y$  }.

Class Neq_case_lt_slt '{Le A} : Prop :=
  { neq_case_lt_slt :  $\forall x y : A, x \neq y \rightarrow x < y \vee y < x$  }.

Class Le_case_eq_lt '{Le A} : Prop :=
  { le_case_eq_lt :  $\forall x y : A, x \leq y \rightarrow x = y \vee x < y$  }.

Class Ge_case_eq_gt '{Le A} : Prop :=
  { ge_case_eq_gt :  $\forall x y : A, x \geq y \rightarrow x = y \vee x > y$  }.

  case analysis under two assumptions

Class Le_neq_to_lt '{Le A} : Prop :=
  { le_neq_to_lt :  $\forall x y : A, x \leq y \rightarrow x \neq y \rightarrow x < y$  }.

Class Ge_neq_to_gt '{Le A} : Prop :=
  { ge_neq_to_gt :  $\forall x y : A, x \geq y \rightarrow x \neq y \rightarrow x > y$  }.

Class Nlt_nslt_to_eq '{Le A} : Prop :=
  { nlt_nslt_to_eq :  $\forall x y : A, \neg (lt\ x\ y) \rightarrow \neg (lt\ y\ x) \rightarrow x = y$  }.

  contradiction from case analysis

Class Lt_ge_false '{Le A} : Prop :=
  { lt_ge_false :  $\forall x y : A, x < y \rightarrow x \geq y \rightarrow \mathbf{False}$  }.

Class Lt_gt_false '{Le A} : Prop :=
  { lt_gt_false :  $\forall x y : A, x < y \rightarrow x > y \rightarrow \mathbf{False}$  }.

Class Lt_slt_false '{Le A} : Prop :=
  { lt_slt_false :  $\forall x y : A, x < y \rightarrow y < x \rightarrow \mathbf{False}$  }.

```

Section Instances.

Context '{Le A}.

Ltac *auto_star* ::= try solve [*dauto*].

derived structures

Global Instance Le_preorder_of_Le_order :

Le_order →

Le_preorder.

Proof using. constructor. intros. apply× order_to_preorder. Qed.

Global Instance Le_total_preorder_of_Le_total_order :

Le_total_order →

Le_total_preorder.

Proof using. constructor. intros. apply× total_order_to_total_preorder. Qed.

Global Instance Le_preorder_of_Total_preorder :

Le_total_preorder →

Le_preorder.

```

Proof using. constructor. intros. apply× total_preorder_to_preorder. Qed.

Global Instance Le_order_of_Le_total_order :
  Le_total_order →
  Le_order.
Proof using. constructor. intros. apply× total_order_to_order. Qed.

Global Instance Lt_strict_order_of_Lt_strict_total_order :
  Lt_strict_total_order →
  Lt_strict_order.
Proof using. constructor. intros. apply× strict_total_order_to_strict_order. Qed.

Global Instance Lt_strict_order_of_Le_order :
  Le_order →
  Lt_strict_order.
Proof using. constructor. intros. rew_to_le. apply× strict_order_strict. Qed.

Global Instance Lt_strict_total_order_of_Le_total_order :
  Le_total_order →
  Lt_strict_total_order.
Proof using. constructor. intros. rew_to_le. apply× strict_total_order_of_total_order.
Qed.

    symmetric structures

Global Instance Ge_preorder_of_Le_order :
  Le_order →
  Ge_preorder.
Proof using. constructor. rew_to_le. apply preorder_inverse. apply le_preorder. Qed.

Global Instance Ge_total_preorder_of_Le_total_order :
  Le_total_order →
  Ge_total_preorder.
Proof using. constructor. rew_to_le. apply total_preorder_inverse. apply le_total_preorder.
Qed.

Global Instance Ge_preorder_of_Total_preorder :
  Le_total_preorder →
  Ge_preorder.
Proof using. constructor. rew_to_le. apply preorder_inverse. apply le_preorder. Qed.

Global Instance Ge_order_of_Le_total_order :
  Le_total_order →
  Ge_order.
Proof using. constructor. rew_to_le. apply order_inverse. apply le_order. Qed.

Global Instance Gt_strict_order_of_Lt_strict_total_order :
  Lt_strict_total_order →
  Gt_strict_order.

```

Proof using. constructor. rewrite gt_is_inverse_lt. apply strict_order_inverse. apply lt_strict_order. Qed.

Global Instance Gt_strict_order_of_Le_order :

Le_order →
Gt_strict_order.

Proof using. constructor. rewrite gt_is_inverse_lt. apply strict_order_inverse. apply lt_strict_order. Qed.

Global Instance Gt_strict_total_order_of_Le_total_order :

Le_total_order →
Gt_strict_total_order.

Proof using. constructor. rewrite gt_is_inverse_lt. apply strict_total_order_inverse. apply lt_strict_total_order. Qed.

properties of le

Global Instance Le_refl_of_Le_preorder :

Le_preorder →
Le_refl.

Proof using. intros *[[Re Tr]]*. constructor $\neg$. Qed.

Global Instance Le_trans_of_Le_preorder :

Le_preorder →
Le_trans.

Proof using. intros *[[Re Tr]]*. constructor $\neg$. Qed.

Global Instance Le_antisym_of_Le_order :

Le_order →
Le_antisym.

Proof using. constructor. intros. apply $\times$ order_antisym. Qed.

Global Instance Le_total_of_Le_total_order :

Le_total_order →
Le_total.

Proof using. constructor. intros. apply $\times$ total_order_total. Qed.

properties of ge

Global Instance Ge_refl_of_Le_preorder :

Le_preorder →
Ge_refl.

Proof using. constructor. *rew_to_le*. apply refl_inverse. apply le_refl. Qed.

Global Instance Ge_trans_of_Le_preorder :

Le_preorder →
Ge_trans.

Proof using. constructor. *rew_to_le*. apply trans_inverse. apply le_trans. Qed.

Global Instance Ge_antisym_of_Le_order :


```

Le_order →
Ge_antisym.
Proof using. constructor. rew_to_le. apply antisym_inverse. apply le_antisym. Qed.

Global Instance Ge_total_of_Le_total_order :
Le_total_order →
Ge_total.
Proof using. constructor. rew_to_le. apply total_inverse. apply le_total. Qed.
  properties of lt

Global Instance Lt_irrefl_of_Le_order :
Le_order →
Lt_irrefl.
Proof using. constructor. apply strict_order_irrefl. apply lt_strict_order. Qed.

Global Instance Lt_trans_of_Le_order :
Le_order →
Lt_trans.
Proof using. constructor. apply strict_order_trans. apply lt_strict_order. Qed.
  properties of gt

Global Instance Gt_irrefl_of_Le_order :
Le_order →
Gt_irrefl.
Proof using. constructor. apply strict_order_irrefl. apply gt_strict_order. Qed.

Global Instance Gt_trans_of_Le_order :
Le_order →
Gt_trans.
Proof using. constructor. apply strict_order_trans. apply gt_strict_order. Qed.
  mixed transitivity results

Global Instance Lt_le_trans_of_Le_order :
Le_order →
Lt_le_trans.
Proof using.
  constructor. intros K L. rew_to_le. destruct K as [U V].
  split¬. apply× le_trans. intro_subst. apply V. apply× le_antisym.
  Qed.

Global Instance Le_lt_trans_of_Le_order :
Le_order →
Le_lt_trans.
Proof using.
  constructor. intros K L. rew_to_le. destruct L as [U V].
  split¬. apply× le_trans. intro_subst. apply V. apply× le_antisym.
  Qed.

```

```

Global Instance gt_ge_trans_of_Le_order :
  Le_order →
  Gt_ge_trans.
Proof using.
  constructor. introv K L. rew_to_le_lt. hnf in *. apply× le_lt_trans.
Qed.

Global Instance ge_gt_trans_of_Le_order :
  Le_order →
  Ge_gt_trans.
Proof using.
  constructor. introv K L. rew_to_le_lt. hnf in *. apply× lt_le_trans.
Qed.
  conversion between operators

Global Instance Ge_as_sle_of :
  Ge_as_sle.
Proof using. constructor. intros. rew_to_le. auto. Qed.

Global Instance Gt_as_slt_of :
  Gt_as_slt.
Proof using. constructor. intros. rew_to_le. auto. Qed.

Global Instance Ngt_as_sle_of_Le_total_order :
  Le_total_order →
  Ngt_as_sle.
Proof using.
  constructor. intros. rew_to_le. unfold strict. rew_logic. iff M.
  destruct M.
    forwards K:(inverse_strict_of_not (R:=le)); eauto.
    apply le_total. apply (proj1 K).
    subst. apply le_refl.
    apply or_classic_l. intros P Q. apply P. apply× le_antisym.
Qed.

Global Instance Nlt_as_ge_of_Le_total_order :
  Le_total_order →
  Nlt_as_ge.
Proof using. constructor. intros. rew_to_le_lt. unfold inverse. apply ngt_as_sle. Qed.

Global Instance Ngt_as_le_of_Le_total_order :
  Le_total_order →
  Ngt_as_le.
Proof using. constructor. intros. rew_to_le_lt. unfold inverse. apply ngt_as_sle. Qed.

Global Instance Nle_as_gt_of_Le_total_order :
  Le_total_order →
  Nle_as_gt.

```

Proof using.

```
constructor. intros. rew_to_le_lt. unfold inverse.  
rewrite ← ngt_as_sle. rewrite ¬ not_not_eq.
```

Qed.

Global Instance Nge_as_lt_of_Le_total_order :

```
Le_total_order →  
Nge_as_lt.
```

Proof using.

```
constructor. intros. rew_to_le_lt. unfold inverse.  
rewrite nle_as_gt. rewrite ¬ gt_is_inverse_lt.
```

Qed.

inclusion between operators

Global Instance Lt_to_le_of :

```
Lt_to_le.
```

Proof using. constructor. intros. *rew_to_le*. *unfolds*× strict. Qed.

Global Instance Gt_to_ge_of :

```
Gt_to_ge.
```

Proof using. constructor. intros. *rew_to_le*. *unfolds*× inverse, strict. Qed.

Global Instance Nle_to_sle_of_Le_total_order :

```
Le_total_order →  
Nle_to_sle.
```

Proof using.

```
constructor. introv K. rewrite nle_as_gt in K.  
rew_to_le. unfolds× inverse, strict.
```

Qed.

Global Instance Nle_to_slt_of_Le_total_order :

```
Le_total_order →  
Nle_to_slt.
```

Proof using.

```
constructor. introv K. rewrite nle_as_gt in K.  
rew_to_le. unfolds× inverse, strict.
```

Qed.

case analysis under no assumption

Global Instance Case_eq_lt_gt_of_Le_total_order :

```
Le_total_order →  
Case_eq_lt_gt.
```

Proof using.

```
introv K. constructor. intros.  
lets [(M1&M2)][M](M1&M2)]: (total_order_lt_or_eq_or_gt le_total_order x y).  
rewrite le_is_rclosure_lt in M1 by applies× total_order_refl. destruct× M1.
```

```

    autos×.
    rewrite le_is_rclosure_lt in M1 by applies× total_order_refl. destruct× M1.
Qed.

Global Instance Case_eq_lt_slt_of_Le_total_order :
  Le_total_order →
  Case_eq_lt_slt.
Proof using.
  constructor. intros. pattern lt at 2. rewrite lt_is_inverse_gt.
  apply case_eq_lt_gt.
Qed.

Global Instance Case_le_gt_of_Le_total_order :
  Le_total_order →
  Case_le_gt.
Proof using.
  constructor. intros.
  rewrite le_is_rclosure_lt by applies× total_order_refl. rewrite rclosure_eq.
  branches (total_order_lt_or_eq_or_gt le_total_order x y); eauto.
Qed.

Global Instance Case_eq_lt_ge_of_Le_total_order :
  Le_total_order →
  Case_lt_ge.
Proof using.
  constructor. intros.
  rewrite ge_is_rclosure_gt by applies× total_order_refl. rewrite rclosure_eq.
  branches (total_order_lt_or_eq_or_gt le_total_order x y); eauto.
Qed.

Global Instance Case_le_slt_of_Le_total_order :
  Le_total_order →
  Case_le_slt.
Proof using. constructor. intros. rewrite lt_is_inverse_gt. apply case_le_gt. Qed.

Global Instance Case_eq_lt_sle_of_Le_total_order :
  Le_total_order →
  Case_lt_sle.
Proof using. constructor. intros. rewrite le_is_inverse_ge. apply case_lt_ge. Qed.
  case analysis under one assumption

Global Instance Neq_case_lt_gt_of_Le_total_order :
  Le_total_order →
  Neq_case_lt_gt.
Proof using. constructor. intros. destruct× (case_eq_lt_gt x y). Qed.

Global Instance Neq_case_lt_slt_of_Le_total_order :

```

```

Le_total_order →
Neq_case_lt_sl.
Proof using. constructor. intros. destruct× (case_eq_lt_gt x y). Qed.
Global Instance Le_case_eq_lt_of_Le_total_order :
Le_total_order →
Le_case_eq_lt.
Proof using. constructor. intros. rew_to_le. unfold strict. tests*: (x = y). Qed.
Global Instance Ge_case_eq_gt_of_Le_total_order :
Le_total_order →
Ge_case_eq_gt.
Proof using. constructor. intros. rew_to_le. unfold inverse, strict. tests*: (x = y).
Qed.
    case analysis under two assumptions
Global Instance Le_neq_to_lt_of_Le_total_order :
Le_total_order →
Le_neq_to_lt.
Proof using. constructor. intros. rew_to_le. hnfs×. Qed.
Global Instance Ge_neq_to_gt_of_Le_total_order :
Le_total_order →
Ge_neq_to_gt.
Proof using. constructor. intros. rew_to_le. hnfs×. Qed.
Global Instance Nlt_nslt_to_eq_of_Le_total_order :
Le_total_order →
Nlt_nslt_to_eq.
Proof using. constructor. intros. branches× (case_eq_lt_gt x y). Qed.
    contradiction from case analysis
Global Instance Lt_ge_false_of_Le_total_order :
Le_total_order →
Lt_ge_false.
Proof using. constructor. introv H1 H2. rewrite¬ ← nlt_as_ge in H2. Qed.
Global Instance Lt_gt_false_of_Le_total_order :
Le_total_order →
Lt_gt_false.
Proof using.
    constructor. introv H1 H2. rewrite¬ ← nle_as_gt in H2.
    apply H2. apply× lt_to_le.
Qed.
Global Instance Lt_sl
```

Proof using.

```
constructor. intros H1 H2. rewrite ← gt_as_slc in H2.  
apply × lt_gt_false.
```

Qed.

End Instances.

Other lemmas needs arguments to be implicit?

11.4 Boolean comparison

Module BOOLEANCOMPARISON.

Open Scope *comp_scope*.

Additional notation for reflected boolean comparison. Use `Open Scope comp_scope_reflect` to use them.

Declare Scope *comp_scope_reflect*.

```
Notation "x "<=' y" := (isTrue (@le _ _ x y))  
(at level 70, no associativity) : comp_scope_reflect.
```

```
Notation "x ">=' y" := (isTrue (@ge _ _ x y))  
(at level 70, no associativity) : comp_scope_reflect.
```

```
Notation "x "<' y" := (isTrue (@lt _ _ x y))  
(at level 70, no associativity) : comp_scope_reflect.
```

```
Notation "x ">' y" := (isTrue (@gt _ _ x y))  
(at level 70, no associativity) : comp_scope_reflect.
```

End BOOLEANCOMPARISON.

11.5 Order on relations and on predicates

Lemma `order_rel_incl` : $\forall A B$,

order (@rel_incl *A B*).

Proof using.

```
hint refl_rel_incl, antisym_rel_incl, trans_rel_incl.  
constructors ×.
```

Qed.

Lemma `order_pred_incl` : $\forall A B$,

order (@rel_incl *A B*).

Proof using.

```
hint refl_rel_incl, antisym_rel_incl, trans_rel_incl.  
constructors ×.
```

Qed.

Definition min $\{Le\ A\} (n\ m:A) : A :=$

If $n \leq m$ then n else m .

Lemma min_l : $\forall \{Le\ A\} (n\ m:A),$

antisym le $\rightarrow$

$n \leq m \rightarrow$

min $n\ m = n$.

Proof using. *introv* $T\ M$. unfold min. *case_if* $\times$. Qed.

Lemma min_r : $\forall \{Le\ A\} (n\ m:A),$

antisym le $\rightarrow$

$m \leq n \rightarrow$

min $n\ m = m$.

Proof using. *introv* $T\ M$. unfold min. *case_if* $\times$. Qed.

Definition max $\{Le\ A\} (n\ m:A) : A :=$

If $n \leq m$ then m else n .

Lemma max_l : $\forall \{Le\ A\} (n\ m:A),$

antisym le $\rightarrow$

$n \geq m \rightarrow$

max $n\ m = n$.

Proof using. *introv* $T\ M$. unfold max. *case_if* $\times$. Qed.

Lemma max_r : $\forall \{Le\ A\} (n\ m:A),$

antisym le $\rightarrow$

$n \leq m \rightarrow$

max $n\ m = m$.

Proof using. *introv* $T\ M$. unfold max. *case_if* $\times$. Qed.

Chapter 12

Library SLF.LibNat

```
Set Implicit Arguments.
From Stdlib Require Export Arith Lia.
From SLF Require Import LibTactics LibReflect LibBool LibOperation LibRelation LibOrder.
From SLF Require Export LibOrder.
Global Close Scope positive_scope.
```

12.1 Nat type

12.1.1 Definition

From the Prelude:

```
Inductive nat : Set := | O : nat | S : nat -> nat.
```

Remark: ideally, constructors would be renamed to *zero* and *succ*, or *nat_zero* and *nat_succ*, with the notations *O* or *0%nat*, and *S n* or *succ n*. It is indeed problematic to prevent the use of single letter variables in pattern matching to the user who does not care about *nat*.

12.1.2 Inhabited

```
#[global]
Instance Inhab_nat : Inhab nat.
Proof using. intros. apply (Inhab_of_val 0). Qed.
```


12.2 Order on natural numbers

12.2.1 Definition

The typeclass instance of `le` on `nat` is defined to be the `le` relation on Peano numbers from Rocq’s standard library.

```
#[global]
Instance le_nat_inst : Le nat := Build_Le Peano.le.
```

12.2.2 Translating typeclass instances to Peano relations

These lemmas and tactics are useful to transform arithmetic goals into a form on which the `lia` decision procedure may apply.

```
Lemma le_peano : le = Peano.le.
```

```
Proof using. extens ×. Qed.
```

```
Global Opaque le_nat_inst.
```

```
Lemma lt_peano : lt = Peano.lt.
```

```
Proof using.
```

```
  extens. rew_to_le. rewrite le_peano.
```

```
  unfold strict. intros. lia.
```

```
Qed.
```

```
Lemma ge_peano : ge = Peano.ge.
```

```
Proof using.
```

```
  extens. rew_to_le. rewrite le_peano.
```

```
  unfold inverse. intros. lia.
```

```
Qed.
```

```
Lemma gt_peano : gt = Peano.gt.
```

```
Proof using.
```

```
  extens. rew_to_le. rewrite le_peano.
```

```
  unfold strict, inverse. intros. lia.
```

```
Qed.
```

```
#[global]
```

```
Hint Rewrite le_peano lt_peano ge_peano gt_peano : rew_nat_comp.
```

```
Ltac nat_comp_to_peano :=
```

```
  autorewrite with rew_nat_comp in *.
```

nat_math calls *lia* after basic pre-processing (*intros* and *split*) and after replacing comparison operators with the ones defined in `Peano` library.

```
Ltac nat_math_setup :=
```

```
  intros;
```

```
  try match goal with ⊢ _ ∧ _ ⇒ split end;
```

```

    try match goal with  $\vdash$  _ = _  $\Rightarrow$  Prop  $\Rightarrow$  apply prop_ext; iff end;
    nat_comp_to_peano.
Ltac nat_math :=
    nat_math_setup; lia.

```

12.2.3 The *nat_maths* database is used for registering automation on mathematical goals.

```

Ltac nat_math_hint := nat_math.
#[global]
Hint Extern 3 ( _ = _  $\Rightarrow$  nat )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( _  $\neq$  _  $\Rightarrow$  nat )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( istrue (isTrue ( _ = _  $\Rightarrow$  nat )))  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( istrue (isTrue ( _  $\neq$  _  $\Rightarrow$  nat )))  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( ( _  $\leq$  _ ) % nat )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( ( _  $\geq$  _ ) % nat )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( ( _ < _ ) % nat )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( ( _ > _ ) % nat )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( @le nat _ _ _ )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( @lt nat _ _ _ )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( @ge nat _ _ _ )  $\Rightarrow$  nat_math_hint : nat_maths.
#[global]
Hint Extern 3 ( @gt nat _ _ _ )  $\Rightarrow$  nat_math_hint : nat_maths.

```

12.2.4 Total order instance

```

#[global]
Instance nat_le_total_order : Le_total_order (A:=nat).
Proof using.
    constructor. constructor. constructor; unfolds.
    nat_math. nat_math. nat_math.

```

unfolds. intros. tests: (x ≤ y). left⊢. right. nat_math.
Qed.

12.3 Induction

Lemma *peano_induction* :

$\forall (P:\mathbf{nat} \rightarrow \mathbf{Prop}),$
 $(\forall n, (\forall m, m < n \rightarrow P\ m) \rightarrow P\ n) \rightarrow$
 $(\forall n, P\ n).$

Proof using.

intros H. cuts× K: (∀ n m, m < n → P m).
nat_comp_to_peano.
induction n; intros Le. inversion Le. apply H.
intros. apply IHn. nat_math.

Qed.

Lemma *measure_induction* :

$\forall (A:\mathbf{Type})\ (mu:A \rightarrow \mathbf{nat})\ (P:A \rightarrow \mathbf{Prop}),$
 $(\forall x, (\forall y, mu\ y < mu\ x \rightarrow P\ y) \rightarrow P\ x) \rightarrow$
 $(\forall x, P\ x).$

Proof using.

intros IH. intros x. gen_eq n: (mu x). gen x.
induction n using peano_induction. intros Eq. subst×.

Qed.

12.4 Simplification lemmas

12.4.1 Trivial monoid simplifications

Lemma *plus_zero_r* : $\forall n,$

$n + 0 = n.$

Proof using. *nat_math. Qed.*

Lemma *plus_zero_l* : $\forall n,$

$0 + n = n.$

Proof using. *nat_math. Qed.*

Lemma *minus_zero_r* : $\forall n,$

$n - 0 = n.$

Proof using. *nat_math. Qed.*

Lemma *plus_succ* : $\forall n1\ n2,$

$n1 + S\ n2 = S\ (n1 + n2).$

Proof using. *nat_math. Qed.*

```

Lemma minus_zero :  $\forall n,$ 
   $n - 0 = n.$ 
Proof using. nat_math. Qed.

Lemma succ_minus_succ :  $\forall n1\ n2,$ 
   $S\ n1 - S\ n2 = n1 - n2.$ 
Proof using. nat_math. Qed.

Lemma minus_same :  $\forall n,$ 
   $n - n = 0.$ 
Proof using. nat_math. Qed.

Lemma plus_minus_same :  $\forall n1\ n2,$ 
   $n1 + n2 - n1 = n2.$ 
Proof using. nat_math. Qed.

Lemma mult_zero_l :  $\forall n,$ 
   $0 \times n = 0.$ 
Proof using. nat_math. Qed.

Lemma mult_zero_r :  $\forall n,$ 
   $n \times 0 = 0.$ 
Proof using. nat_math. Qed.

Lemma mult_one_l :  $\forall n,$ 
   $1 \times n = n.$ 
Proof using. nat_math. Qed.

Lemma mult_one_r :  $\forall n,$ 
   $n \times 1 = n.$ 
Proof using. nat_math. Qed.

```

12.4.2 Simplification tactic

rew_nat performs some basic simplification on expressions involving natural numbers

#[global]

```

Hint Rewrite plus_zero_r plus_zero_l minus_zero_r
  plus_succ minus_zero succ_minus_succ minus_same plus_minus_same
  mult_zero_l mult_zero_r mult_one_l mult_one_r : rew_nat.

```

```

Tactic Notation "rew_nat" :=
  autorewrite with rew_nat.

```

```

Tactic Notation "rew_nat" "~" :=
  rew_nat; auto_tilde.

```

```

Tactic Notation "rew_nat" "*" :=
  rew_nat; auto_star.

```

```

Tactic Notation "rew_nat" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt  $\Rightarrow$  autorewrite with rew_nat).

```

```

Tactic Notation "rew_nat" "~" "in" "*" :=
  rew_nat in *; auto_tilde.
Tactic Notation "rew_nat" "*" "in" "*" :=
  rew_nat in *; auto_star.
Tactic Notation "rew_nat" "in" hyp(H) :=
  autorewrite with rew_nat in H.
Tactic Notation "rew_nat" "~" "in" hyp(H) :=
  rew_nat in H; auto_tilde.
Tactic Notation "rew_nat" "*" "in" hyp(H) :=
  rew_nat in H; auto_star.

```

12.5 Executable functions

```

Fixpoint beq (x y : nat) :=
  match x, y with
  | O, O => true
  | S x', S y' => beq x' y'
  | _, _ => false
  end.

```

```

Lemma beq_eq : ∀ n1 n2,
  beq n1 n2 = isTrue (n1 = n2).

```

Proof using.

```

  intros n1. induction n1; intros; destruct n2; simpl; rew_bool_eq; auto_false.
  rewrite IHn1. extens. rew_istrue. nat_math.

```

Qed.

Chapter 13

Library SLF.LibEpsilon

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibLogic LibRelation.
Generalizable Variables A B.
```

13.1 Definition and specification of Hilbert's epsilon operator

13.1.1 Definition of epsilon

`epsilon` P where P is a predicate over an inhabited type A , returns a value x of type A that satisfies P , if there exists one such value, else it returns an arbitrary value of type A .

```
Definition epsilon_def : ∀ A {IA:Inhab A} (P:A→Prop),
  { x : A | (∃ y, P y) → P x }.
```

Proof using.

```
  intros A IA P. destruct (classicT (∃ y, P y)) as [H|H].
  { apply indefinite_description. destruct H as [x H].
    ∃ x. intros -. apply H. }
  { ∃ (@arbitrary A IA). intros N. false H. apply N. }
```

Qed.

```
Definition epsilon A {IA: Inhab A} (P:A→Prop) : A :=
  sig_val (epsilon_def P).
```

```
Lemma pred_epsilon : ∀ A {IA:Inhab A} (P:A→Prop),
  (∃ x, P x) →
  P (epsilon P).
```

Proof using. intros. apply $\neg$ (sig_proof (epsilon_def P)). Qed.

Opaque epsilon.

```
Definition epsilon_def' A {IA:Inhab A} (P:A→Prop) :
```

```

{ x : A | (∃ y, P y) → P x } :=
match classicT (∃ y, P y) with
| left H ⇒
  indefinite_description
  (let (x,H0) := H in
   ex_intro (fun x0 ⇒ (∃ y, P y) → P x0)
             x
             (fun N ⇒ H0))
| right H ⇒
  exist (fun x ⇒ (∃ y, P y) → P x)
        arbitrary
        (fun N ⇒ False_ind (P arbitrary) (H N))
end.

```

13.1.2 Lemmas about epsilon

Lemma `pred_epsilon_weaken` : $\forall A \{IA:\text{Inhab } A\} (P \ Q : A \rightarrow \text{Prop}),$
 $(\exists x, P \ x) \rightarrow$
 $(\forall x, P \ x \rightarrow Q \ x) \rightarrow$
 $Q \ (\text{epsilon } P).$

Proof using. `introv E M. apply M. apply× pred_epsilon. Qed.`

Lemma `pred_epsilon_of_val` : $\forall A (x:A) (P:A \rightarrow \text{Prop}) \{IA:\text{Inhab } A\},$
 $P \ x \rightarrow$
 $P \ (\text{epsilon } P).$

Proof using. `intros. apply× pred_epsilon. Qed.`

Lemma `pred_epsilon_of_val_weaken` : $\forall A (x:A) (P \ Q:A \rightarrow \text{Prop}) \{IA:\text{Inhab } A\},$
 $P \ x \rightarrow$
 $(\forall x, P \ x \rightarrow Q \ x) \rightarrow$
 $Q \ (\text{epsilon } P).$

Proof using. `introv Px W. apply W. apply× pred_epsilon. Qed.`

Lemma `epsilon_eq` : $\forall A \{IA:\text{Inhab } A\} (P \ Q:A \rightarrow \text{Prop}),$
 $(\forall x, P \ x \leftrightarrow Q \ x) \rightarrow$
 $\text{epsilon } P = \text{epsilon } Q.$

Proof using. `introv H. fequals. extens×. Qed.`

13.1.3 (Private) tactic *epsilon_find*

epsilon_find cont locates an expression of the form `epsilon P` in the goal and invokes the continuation *cont* on *P*.

epsilon_find_in H cont is similar but looks for the expression only in the hypothesis named *H*.

```

Ltac epsilon_find cont :=
  match goal with
  |  $\vdash$  context [epsilon ?P]  $\Rightarrow$  cont P
  | H: context [epsilon ?P]  $\vdash$  _  $\Rightarrow$  cont P
  end.

Ltac epsilon_find_in H cont :=
  match type of H with context [epsilon ?P]  $\Rightarrow$  cont P end.

```

13.1.4 Tactics *epsilon_name*

epsilon_name *X* assigns a name *X* to an expression of the form *epsilon* *P* that appears in the goal or an hypothesis, by calling `set (X := epsilon P)`.

epsilon_name *X* in *H* assigns a name *X* to an expression of the form *epsilon* *P* that appears in hypothesis *H*.

```

Ltac epsilon_name_core X :=
  epsilon_find ltac:(fun P  $\Rightarrow$  sets X: (epsilon P)).

Ltac epsilon_name_in_core X H :=
  epsilon_find_in H ltac:(fun P  $\Rightarrow$  sets X: (epsilon P)).

Tactic Notation "epsilon_name" ident(X) :=
  epsilon_name_core X.

Tactic Notation "epsilon_name" ident(X) "in" hyp(H) :=
  epsilon_name_in_core X H.

```

13.1.5 Tactics to work with *epsilon*

epsilon *X* locates an expression of the form *epsilon* *P* in the goal, names *X* this expression (like *epsilon_name* *X*), then produces a subgoal $\exists x, P\ x$, and leaves at the head of the main goal an hypothesis *P* *X*.

epsilon *X* in *H* is similar, but looks for *epsilon* *P* only in hypothesis *H*.

```

Lemma pred_epsilon' :  $\forall A (P:A \rightarrow \text{Prop}) (IA:\text{Inhab } A),$ 
  ( $\exists x, P\ x$ )  $\rightarrow$ 
  P (epsilon P).

```

Proof using. intros. *applies* $\times$ pred_epsilon. Qed.

```

Ltac epsilon_cont X P :=
  let I := fresh "H" X in
  lets I: ( $\gg$  (@pred_epsilon' _ P) -- --);
  [ | sets X: (epsilon P); revert I ].

Ltac epsilon_core X :=
  epsilon_find ltac:(fun P  $\Rightarrow$  epsilon_cont X P).

Ltac epsilon_in_core X H :=

```



```

    epsilon_find_in H ltac:(fun P => epsilon_cont X P).
Tactic Notation "epsilon" ident(X) :=
    epsilon_core X.
Tactic Notation "epsilon" ident(X) "in" hyp(H) :=
    epsilon_in_core X H.
Tactic Notation "epsilon" "~" ident(X) :=
    epsilon X; auto_tilde.
Tactic Notation "epsilon" "~" ident(X) "in" hyp(H) :=
    epsilon X in H; auto_tilde.
Tactic Notation "epsilon" "*" ident(X) :=
    epsilon X; auto_star.
Tactic Notation "epsilon" "*" ident(X) "in" hyp(H) :=
    epsilon X in H; auto_star.

```

13.2 Construction of a function from a relation, using epsilon

```

Definition rel_to_fun A '{IB:Inhab B} (R:A→B→Prop) : A → B :=
    fun (a:A) => epsilon (fun b => R a b).

```

Section Rel_to_fun.

```
Context (A B : Type) {IB:Inhab B}.
```

```
Implicit Types R : A → B → Prop.
```

```

Lemma rel_rel_to_fun_of_exists : ∀ R a,
    (∃ b, R a b) →
    R a (rel_to_fun R a).

```

Proof using IB. introv [x H]. unfold rel_to_fun. epsilon× y. Qed.

```

Lemma rel_rel_to_fun_of_rel : ∀ R a b,
    R a b →
    R a (rel_to_fun R a).

```

Proof using IB. intros. applies× rel_rel_to_fun_of_exists. Qed.

```

Lemma rel_rel_to_fun_of_not_forall : ∀ R a,
    ¬ (∀ b, ¬ R a b) →
    R a (rel_to_fun R a).

```

Proof using IB.

```

    introv. rew_logic. intros [x H]. applies× rel_rel_to_fun_of_exists.
Qed.

```

```

Lemma rel_in_fun_rel_to_fun : ∀ R,
    functional R →

```

```

    rel_in_fun  $R$  (rel_to_fun  $R$ ).
Proof using IB. unfold rel_in_fun, rel_to_fun. intros  $M$   $H$ . epsilon  $\times$   $z$ . Qed.

```

Reformulation of above

```

Lemma rel_to_fun_eq_of_functional :  $\forall$   $x$   $y$   $R$ ,
  functional  $R \rightarrow$ 
   $R$   $x$   $y \rightarrow$ 
  rel_to_fun  $R$   $x = y$ .
Proof using IB. intros  $M$   $E$ . applies  $\times$  rel_in_fun_rel_to_fun. Qed.
End Rel_to_fun.

```

Chapter 14

Library SLF.LibInt

```
Set Implicit Arguments.
From Stdlib Require Export ZArith.
From SLF Require Import LibTactics LibLogic LibReflect LibRelation.
Export LIBTACTICSCOMPATIBILITY.
From SLF Require Export LibNat.
```

14.1 Parsing of integers and operations

14.1.1 Notation for type and operation

Define *int* as an alias for *Z*, the type of integers from Rocq's *stdlib*.

```
Declare Scope Int_scope.
Notation "'int'" := Z : Int_scope.
Delimit Scope Int_scope with I.
Local Open Scope Int_scope.
```

14.1.2 Inhabited type

```
#[global]
Instance Inhab_int : Inhab int.
Proof using. intros. apply (Inhab_of_val 0%Z). Qed.
```

14.1.3 Coercion from nat

Remark: we cannot simply use the coercion: Coercion *Z_of_nat* : *nat* $\rightarrow$ *Z*. because otherwise when we try to make the coercion opaque using: Opaque *Z_of_nat*. the *lia* fails to work. Thus, we introduce an alias, called *nat_to_Z* for *Z_of_nat*, and we register *nat_to_Z* as coercion.

```

Definition nat_to_Z := Z_of_nat.
Lemma nat_to_Z_eq_Z_of_nat : nat_to_Z = Z_of_nat.
Proof using. reflexivity. Qed.
Global Opaque nat_to_Z.
Coercion nat_to_Z : nat >-> Z.

```

14.1.4 Order relation

The comparison operators on integers are those from `LibOrder`, not the ones from Rocq's `ZArith`.

```

Open Scope Z_scope.
Open Scope comp_scope.

```

The typeclass `le` on type `int` is bound to `Zle`, from Rocq's standard library

```

#[global]
Instance le_int_inst : Le int := Build_Le Z.le.

```

14.2 Conversion to natural numbers, for tactic programming

These tactics are helpful to convert a number passed to a Ltac tactic into a `nat`, regardless of whether it is a `nat` or an `int`.

```

Definition ltac_int_to_nat (x:Z) : nat :=
  match x with
  | Z0 => 0%nat
  | Zpos p => nat_of_P p
  | Zneg p => 0%nat
  end.

Ltac number_to_nat N ::=
  match type of N with
  | nat => constr:(N)
  | int => let N' := constr:(ltac_int_to_nat N) in eval compute in N'
  | Z => let N' := constr:(ltac_int_to_nat N) in eval compute in N'

end.

```

14.3 Decision procedure

A lot of hacks to allow calling the `lia` tactic

14.3.1 Translation from typeclass order to ZArith, for using *lia*

```
Lemma le_zarith : le = Z.le.
Proof using. extends ×. Qed.

Global Opaque le_int_inst.

Lemma lt_zarith : lt = Z.lt.
Proof using.
  extends. rew_to_le. rewrite le_zarith.
  unfold strict. intros. lia.
Qed.

Lemma ge_zarith : ge = Z.ge.
Proof using.
  extends. rew_to_le. rewrite le_zarith.
  unfold inverse. intros. lia.
Qed.

Lemma gt_zarith : gt = Z.gt.
Proof using.
  extends. rew_to_le. rewrite le_zarith.
  unfold strict, inverse. intros. lia.
Qed.

#[global]
Hint Rewrite le_zarith lt_zarith ge_zarith gt_zarith : rew_int_comp.

Ltac int_comp_to_zarith :=
  autorewrite with rew_int_comp in *.
```

14.3.2 Hypothesis selection

is_additional_arith_type *T* allows for extending the behavior of *is_arith_type*.

```
Ltac is_additional_arith_type T :=
  constr:(false).

is_arity_type T returns a boolean indicating whether T is equal to nat or int

Ltac is_arith_type T :=
  match T with
  | nat ⇒ constr:(true)
  | int ⇒ constr:(true)
  | _ ⇒ is_additional_arith_type T
  end.

is_arity E returns a boolean indicating whether E is an arithmetic expression

Ltac is_arith E :=
```

```

match E with
| _ = _ :> ?T ⇒ is_arith_type T
| _ ≠ _ :> ?T ⇒ is_arith_type T
| ¬ ?E' ⇒ is_arith E'
| ?E' → False ⇒ is_arith E'
| @le ?T _ _ _ ⇒ is_arith_type T
| @ge ?T _ _ _ ⇒ is_arith_type T
| @lt ?T _ _ _ ⇒ is_arith_type T
| @gt ?T _ _ _ ⇒ is_arith_type T
| (_ ≤ _) % Z ⇒ constr:(true)
| (_ ≥ _) % Z ⇒ constr:(true)
| (_ < _) % Z ⇒ constr:(true)
| (_ > _) % Z ⇒ constr:(true)
| (_ ≤ _) % nat ⇒ constr:(true)
| (_ ≥ _) % nat ⇒ constr:(true)
| (_ < _) % nat ⇒ constr:(true)
| (_ > _) % nat ⇒ constr:(true)
| _ ⇒ constr:(false)
end.

```

arith_goal_or_false looks at the current goal and replaces it with **False** if it is not an arithmetic goal

```

Ltac arith_goal_or_false :=
  match goal with ⊢ ?E ⇒
    match is_arith E with
    | true ⇒ idtac
    | false ⇒ false
    end
  end.

```

generalize_arith generalizes all hypotheses which correspond to some arithmetic goal. It destructs conjunctions on the fly.

```

Lemma istrue_isTrue_forw : ∀ (P:Prop),
  istrue (isTrue P) →
  P.

```

Proof using. *intros* *H*. *rew_istrue*¬ in *H*. Qed.

```

Ltac generalize_arith :=
  repeat match goal with
  | H: istrue (isTrue _) ⊢ _ ⇒ generalize (@istrue_isTrue_forw _ H); clear H; intro
  | H: ?E1 ∧ ?E2 ⊢ _ ⇒ destruct H
  | H: ?E → False ⊢ _ ⇒
    match is_arith E with
    | true ⇒ change (E → False) with (¬ E) in H

```

```

| false => clear H
end
| H:?E ⊢ _ =>
  match is_arith E with
  | true => generalize H; clear H
  | false => clear H
  end
end.

```

14.3.3 Normalization of arithmetic expressions

Two lemmas to help lia out

Lemma `Z_of_nat_O` :

`Z_of_nat O = 0`.

Proof using. `reflexivity`. Qed.

Lemma `Z_of_nat_S` : $\forall n$,

`Z_of_nat (S n) = Z.succ (Z_of_nat n)`.

Proof using. `intros`. `rewrite` $\leftarrow$ `Zpos_P_of_succ_nat`. Qed.

Lemma `Z_of_nat_plus1` : $\forall n$,

`Z_of_nat (1 + n) = Z.succ (Z_of_nat n)`.

Proof using. `intros`. `rewrite` $\leftarrow$ `Z_of_nat_S`. `fequals` $\neg$. Qed.

`rew_maths` rewrite any lemma in the base `rew_maths`. The goal should not contain any `evr`, otherwise tactic might loop.

`#[global]`

Hint Rewrite `nat_to_Z_eq_Z_of_nat Z_of_nat_O Z_of_nat_S Z_of_nat_plus1` : `rew_maths`.

Ltac `rew_maths` :=

`autorewrite with rew_maths in *`.

14.3.4 Setting up the goal for *lia*

`math_setup_goal` does introduction, splits, and replace the goal by **False** if it is not arithmetic. If the goal is of the form $P1 = P2 :> \text{Prop}$, then the goal is changed to $P1 \leftrightarrow P2$.

Ltac `math_setup_goal_step tt` :=

```

  match goal with
  | ⊢ _ → _ => intro
  | ⊢ _ ↔ _ => iff
  | ⊢ ∀ _, _ => intro
  | ⊢ _ ∧ _ => split
  | ⊢ _ = _ :> Prop => apply prop_ext; iff

```

```

end.
Ltac math_setup_goal :=
  repeat (math_setup_goal_step tt);
  arith_goal_or_false.

```

math tactics performs several preprocessing step, selects all arithmetic hypotheses, and the call *lia*.

14.3.5 Main driver for the set up process to goal *lia*

```

Ltac check_noevar_goal ::=
  match goal with ⊢ ?G ⇒ first [ has_evar G; fail 1 | idtac ] end.
Ltac math_0 := idtac.
Ltac math_1 := intros; generalize_arith.
Ltac math_2 := check_noevar_goal.
Ltac math_3 := autorewrite with rew_maths rew_int_comp rew_nat_comp; intros.
Ltac math_4 := math_setup_goal.
Ltac math_5 := lia.

Ltac math_setup := math_0; math_1; math_2; math_3; math_4.
Ltac math_base := math_setup; math_5.

Tactic Notation "math" := math_base.

Tactic Notation "math" simple_intropattern(I) ":" constr(E) :=
  asserts I: E; [ math | ].

Tactic Notation "maths" constr(E) :=
  let H := fresh "H" in asserts H: E; [ math | ].

```

14.3.6 *math* tactic restricted to arithmetic goals

math_only calls *math* but only on goals which have an arithmetic form. Thus, contracty to *math*, it does not attempt to look for a contradiction in the hypotheses if the conclusion is not an arithmetic goal. Useful for efficiency.

```

Ltac math_only_if_arith_core tt :=
  match goal with ⊢ ?T ⇒
    match is_arith T with true ⇒ math end end.

Tactic Notation "math_only_if_arith" :=
  math_only_if_arith_core tt.

```

14.3.7 Elimination of multiplication, to call *lia*

Lemma `mult_2_eq_plus` : $\forall x, 2 \times x = x + x$.


```

Proof using. intros. ring. Qed.
Lemma mult_3_eq_plus :  $\forall x, 3 \times x = x + x + x$ .
Proof using. intros. ring. Qed.

```

14.3.8 Hint externs for calling math in the hint base *maths*

```

Ltac math_hint := math.
#[global]
Hint Extern 3 ( _ = _ :> nat ) => math_hint : maths.
#[global]
Hint Extern 3 ( _ = _ :> int ) => math_hint : maths.
#[global]
Hint Extern 3 ( _ ≠ _ :> nat ) => math_hint : maths.
#[global]
Hint Extern 3 ( _ ≠ _ :> int ) => math_hint : maths.
#[global]
Hint Extern 3 ( istrue (isTrue ( _ = _ :> nat ))) => math_hint : maths.
#[global]
Hint Extern 3 ( istrue (isTrue ( _ = _ :> int ))) => math_hint : maths.
#[global]
Hint Extern 3 ( istrue (isTrue ( _ ≠ _ :> nat ))) => math_hint : maths.
#[global]
Hint Extern 3 ( istrue (isTrue ( _ ≠ _ :> int ))) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ ≤ _ )%nat ) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ ≥ _ )%nat ) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ < _ )%nat ) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ > _ )%nat ) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ ≤ _ )%Z ) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ ≥ _ )%Z ) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ < _ )%Z ) => math_hint : maths.
#[global]
Hint Extern 3 ( ( _ > _ )%Z ) => math_hint : maths.
#[global]
Hint Extern 3 ( @le nat _ _ _ ) => math_hint : maths.
#[global]

```

```

Hint Extern 3 (@lt nat _ _ _) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@ge nat _ _ _) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@gt nat _ _ _) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@le int _ _ _) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@lt int _ _ _) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@ge int _ _ _) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@gt int _ _ _) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (¬ @le int _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (¬ @lt int _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (¬ @ge int _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (¬ @gt int _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (¬ @eq int _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (@le int _ _ _ → False) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@lt int _ _ _ → False) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@ge int _ _ _ → False) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@gt int _ _ _ → False) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (@eq int _ _ _ → False) ⇒ math_hint : maths.
#[global]
Hint Extern 3 (¬ @le nat _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (¬ @lt nat _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (¬ @ge nat _ _ _) ⇒ unfold not; math_hint : maths.
#[global]
Hint Extern 3 (¬ @gt nat _ _ _) ⇒ unfold not; math_hint : maths.
#[global]

```

```

Hint Extern 3 ( $\neg$  @eq nat _ _)  $\Rightarrow$  unfold not; math_hint : maths.
#[global]
Hint Extern 3 (@le nat _ _  $\rightarrow$  False)  $\Rightarrow$  math_hint : maths.
#[global]
Hint Extern 3 (@lt nat _ _  $\rightarrow$  False)  $\Rightarrow$  math_hint : maths.
#[global]
Hint Extern 3 (@ge nat _ _  $\rightarrow$  False)  $\Rightarrow$  math_hint : maths.
#[global]
Hint Extern 3 (@gt nat _ _  $\rightarrow$  False)  $\Rightarrow$  math_hint : maths.
#[global]
Hint Extern 3 (@eq nat _ _  $\rightarrow$  False)  $\Rightarrow$  math_hint : maths.

```

14.4 Rewriting on arithmetic expressions

14.4.1 Rewriting equalities provable by the *math* tactic

math_rewrite ($E = F$) replaces all occurrences of E with the expression F . It produces the equality $E = F$ as subgoal, and tries to solve it using the tactic *math*

```

Tactic Notation "math_rewrite" constr(E) :=
  asserts_rewrite E; [ try math | ].
Tactic Notation "math_rewrite" constr(E) "in" hyp(H) :=
  asserts_rewrite E in H; [ try math | ].
Tactic Notation "math_rewrite" constr(E) "in" "*" :=
  asserts_rewrite E in *; [ try math | ].
Tactic Notation "math_rewrite" "~" constr(E) :=
  math_rewrite E; auto_tilde.
Tactic Notation "math_rewrite" "~" constr(E) "in" hyp(H) :=
  math_rewrite E in H; auto_tilde.
Tactic Notation "math_rewrite" "~" constr(E) "in" "*" :=
  math_rewrite E in *; auto_tilde.
Tactic Notation "math_rewrite" "*" constr(E) :=
  math_rewrite E; auto_star.
Tactic Notation "math_rewrite" "*" constr(E) "in" hyp(H) :=
  math_rewrite E in H; auto_star.
Tactic Notation "math_rewrite" "*" constr(E) "in" "*" :=
  math_rewrite E in *; auto_star.

```

14.4.2 Addition and subtraction

```

Lemma plus_zero_r :  $\forall n,$ 
   $n + 0 = n.$ 

```

Proof using. *math*. Qed.

Lemma plus_zero_l : $\forall n,$
 $0 + n = n.$

Proof using. *math*. Qed.

Lemma minus_zero_r : $\forall n,$
 $n - 0 = n.$

Proof using. *math*. Qed.

Lemma minus_zero_l : $\forall n,$
 $0 - n = (-n).$

Proof using. *math*. Qed.

Lemma mult_zero_l : $\forall n,$
 $0 \times n = 0.$

Proof using. *math*. Qed.

Lemma mult_zero_r : $\forall n,$
 $n \times 0 = 0.$

Proof using. *math*. Qed.

Lemma mult_one_l : $\forall n,$
 $1 \times n = n.$

Proof using. *math*. Qed.

Lemma mult_one_r : $\forall n,$
 $n \times 1 = n.$

Proof using. *math*. Qed.

Lemma minus_self : $\forall n,$
 $n - n = 0.$

Proof using. *math*. Qed.

Lemma one_plus_minus_one_r : $\forall n,$
 $1 + (n - 1) = n.$

Proof using. *math*. Qed.

Lemma plus_one_minus_one_l : $\forall n,$
 $(n + 1) - 1 = n.$

Proof using. *math*. Qed.

Lemma one_plus_minus_one_l : $\forall n,$
 $(1 + n) - 1 = n.$

Proof using. *math*. Qed.

14.4.3 Simplification tactic

rew_int performs some basic simplification on expressions involving integers
`#[global]`

```

Hint Rewrite plus_zero_r plus_zero_l minus_zero_r minus_zero_l
  mult_zero_l mult_zero_r mult_one_l mult_one_r
  minus_self one_plus_minus_one_r plus_one_minus_one_l
  one_plus_minus_one_l : rew_int.

Tactic Notation "rew_int" :=
  autorewrite with rew_int.
Tactic Notation "rew_int" "~" :=
  rew_int; auto_tilde.
Tactic Notation "rew_int" "*" :=
  rew_int; auto_star.
Tactic Notation "rew_int" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_int).
Tactic Notation "rew_int" "~" "in" "*" :=
  rew_int in *; auto_tilde.
Tactic Notation "rew_int" "*" "in" "*" :=
  rew_int in *; auto_star.
Tactic Notation "rew_int" "in" hyp(H) :=
  autorewrite with rew_int in H.
Tactic Notation "rew_int" "~" "in" hyp(H) :=
  rew_int in H; auto_tilde.
Tactic Notation "rew_int" "*" "in" hyp(H) :=
  rew_int in H; auto_star.

```

14.5 Conversions of operations from **nat** to *int* and back

LATER: make proofs below no longer depend on Rocq's stdlib

14.5.1 Lifting of comparisons from **nat** to *int*

```

Lemma eq_nat_of_eq_int : ∀ (n m: nat),
  (n:int) = (m:int) →
  n = m :> nat.

```

Proof using. *math*. Qed.

```

Lemma neq_nat_of_neq_int : ∀ (n m: nat),
  (n:int) ≠ (m:int) →
  (n ≠ m)%nat.

```

Proof using. *math*. Qed.

```

Lemma eq_int_of_eq_nat : ∀ (n m: nat),
  n = m :> nat →
  (n:int) = (m:int).

```

Proof using. *math*. Qed.

Lemma neq_int_of_neq_nat : $\forall (n\ m:\mathbf{nat}),$
 $(n \neq m) \% \mathbf{nat} \rightarrow$
 $(n:\mathbf{int}) \neq (m:\mathbf{int}).$
 Proof using. *math*. Qed.

14.5.2 Lifting of inequalities from **nat** to *int*

Lemma le_nat_of_le_int : $\forall (n\ m:\mathbf{nat}),$
 $(n:\mathbf{int}) \leq (m:\mathbf{int}) \rightarrow$
 $(n \leq m).$

Proof using. *math*. Qed.

Lemma le_int_of_le_nat : $\forall (n\ m:\mathbf{nat}),$
 $(n \leq m) \rightarrow$
 $(n:\mathbf{int}) \leq (m:\mathbf{int}).$

Proof using. *math*. Qed.

Lemma lt_nat_of_lt_int : $\forall (n\ m:\mathbf{nat}),$
 $(n:\mathbf{int}) < (m:\mathbf{int}) \rightarrow$
 $(n < m).$

Proof using. *math*. Qed.

Lemma lt_int_of_lt_nat : $\forall (n\ m:\mathbf{nat}),$
 $(n < m) \rightarrow$
 $(n:\mathbf{int}) < (m:\mathbf{int}).$

Proof using. *math*. Qed.

Lemma ge_nat_of_ge_int : $\forall (n\ m:\mathbf{nat}),$
 $(n:\mathbf{int}) \geq (m:\mathbf{int}) \rightarrow$
 $(n \geq m).$

Proof using. *math*. Qed.

Lemma ge_int_of_ge_nat : $\forall (n\ m:\mathbf{nat}),$
 $(n \geq m) \rightarrow$
 $(n:\mathbf{int}) \geq (m:\mathbf{int}).$

Proof using. *math*. Qed.

Lemma gt_nat_of_gt_int : $\forall (n\ m:\mathbf{nat}),$
 $(n:\mathbf{int}) > (m:\mathbf{int}) \rightarrow$
 $(n > m).$

Proof using. *math*. Qed.

Lemma gt_int_of_gt_nat : $\forall (n\ m:\mathbf{nat}),$
 $(n > m) \rightarrow$
 $(n:\mathbf{int}) > (m:\mathbf{int}).$

Proof using. *math*. Qed.

14.5.3 Lifting of operations from **nat** to *int*

```
Lemma plus_nat_eq_plus_int : ∀ (n m: nat),  
  (n+m)%nat = (n:int) + (m:int) :> int.  
Proof using.  
  Transparent nat_to_Z.  
  intros. unfold nat_to_Z. applies Nat2Z.inj_add.  
Qed.
```

```
Lemma minus_nat_eq_minus_int : ∀ (n m: nat),  
  (n ≥ m)%nat →  
  (n-m)%nat = (n:int) - (m:int) :> int.  
Proof using.  
  Transparent nat_to_Z.  
  intros. unfold nat_to_Z. applies Nat2Z.inj_sub. math.  
Qed.
```

```
#[global]  
Hint Rewrite plus_nat_eq_plus_int : rew-maths.
```

14.5.4 Properties of comparison

```
Lemma antisym_le_int :  
  antisym (le (A:=int)).  
Proof using. intros x y L1 L2. math. Qed.
```

14.5.5 Absolute function in **nat**

```
Notation "'abs'" := Z.abs_nat (at level 0).  
Global Arguments Z.abs : simpl never.  
Global Arguments Z.abs_nat : simpl never.  
Lemma abs_nat : ∀ (n: nat),  
  abs n = n.  
Proof using. exact Zabs_nat_Z_of_nat. Qed.  
Lemma abs_nonneg : ∀ (x: int),  
  x ≥ 0 →  
  abs x = x :> int.  
Proof using.  
  intros. rewrite inj_Zabs_nat.  
  rewrite Z.abs_eq. math. math.  
Qed.  
Lemma abs_eq_nat_eq : ∀ (x: int) (y: nat),
```

```

 $x \geq 0 \rightarrow$ 
  (abs x =  $y :> \mathbf{nat}$ ) = ( $x = \mathbf{Z\_of\_nat} \ y :> \mathbf{int}$ ).
Proof using.
  introv M. extens. iff E.
  { subst. rewrite Zabs2Nat.id_abs, Z.abs_eq; math. }
  { subst. rewrite  $\neg$  Zabs2Nat.id. }
Qed.

Lemma lt_abs_abs :  $\forall (n \ m : \mathbf{int})$ ,
  ( $0 \leq n$ )  $\rightarrow$ 
  ( $n < m$ )  $\rightarrow$ 
  (abs n < abs m).
Proof using.
  intros. nat_comp_to_peano. apply Zabs_nat_lt. math.
Qed.

Lemma abs_to_int :  $\forall (n : \mathbf{int})$ ,
   $0 \leq n \rightarrow$ 
   $\mathbf{Z\_of\_nat} \ (\text{abs } n) = n$ .
Proof using. intros. rewrite  $\neg$  abs_nonneg. Qed.

Lemma abs_le_nat_le :  $\forall (x:\mathbf{int}) (y:\mathbf{nat})$ ,
  ( $0 \leq x$ )  $\rightarrow$ 
  (abs x  $\leq y$ ) = ( $x \leq y$ )%Z.
Proof.
  intros. extens. iff E.
  { rewrites  $\neg$  <-( $\gg$  abs_to_int x). math. }
  { rewrites  $\neg$  <-( $\gg$  abs_to_int x) in E. math. }
Qed.

Lemma abs_ge_nat_ge :  $\forall (x:\mathbf{int}) (y:\mathbf{nat})$ ,
  ( $0 \leq x$ )  $\rightarrow$ 
  (abs x  $\geq y$ ) = ( $x \geq y$ )%Z .
Proof.
  intros. extens. iff E.
  { rewrites  $\neg$  <-( $\gg$  abs_to_int x). math. }
  { rewrites  $\neg$  <-( $\gg$  abs_to_int x) in E. math. }
Qed.

TODO: many useful lemmas missing

```

14.5.6 abs distribute on constants and operators

Lemma abs_0 : abs 0 = 0% $\mathbf{nat} :> \mathbf{nat}$.

Proof using. reflexivity. Qed.

Lemma abs_1 : abs 1 = 1% $\mathbf{nat} :> \mathbf{nat}$.

Proof using. reflexivity. Qed.

Lemma abs_plus : $\forall (x\ y:\text{int})$,

$(x \geq 0) \rightarrow$

$(y \geq 0) \rightarrow$

$\text{abs } (x + y) = (\text{abs } x + \text{abs } y) \% \text{nat} :> \text{nat}.$

Proof using. intros. *applies* Zabs2Nat.inj_add; *math*. Qed.

Lemma abs_minus : $\forall (x\ y:\text{int})$,

$(x \geq y) \rightarrow$

$(y \geq 0) \rightarrow$

$\text{abs } (x - y) = (\text{abs } x - \text{abs } y) \% \text{nat} :> \text{nat}.$

Proof using. intros. *applies* Zabs2Nat.inj_sub; *math*. Qed.

Lemma abs_nat_plus_nonneg : $\forall (n:\text{nat}) (x:\text{int})$,

$x \geq 0 \rightarrow$

$\text{abs } (n + x) \% Z = (n + \text{abs } x) \% \text{nat}.$

Proof using.

introv N. *applies* eq_nat_of_eq_int.

rewrite plus_nat_eq_plus_int.

do 2 (rewrite abs_nonneg; [|*math*]). auto.

Qed.

Lemma abs_gt_minus_nat : $\forall (n:\text{nat}) (x:\text{int})$,

$(x \geq n) \% Z \rightarrow$

$\text{abs } (x - n) \% Z = (\text{abs } x - n) \% \text{nat}.$

Proof using.

introv N. *applies* eq_nat_of_eq_int.

rewrite minus_nat_eq_minus_int.

do 2 (rewrite abs_nonneg; [|*math*]). auto.

applies ge_nat_of_ge_int. rewrite abs_nonneg; *math*.

Qed.

Lemma succ_abs_eq_abs_one_plus : $\forall (x:\text{int})$,

$x \geq 0 \rightarrow$

$S (\text{abs } x) = \text{abs } (1 + x) :> \text{nat}.$

Proof using.

intros *x*. pattern *x*. *applies* (@measure_induction _ abs). clear *x*.

intros *x IH Pos*. rewrite $\leftarrow$ Zabs_nat_Zsucc. *fequals*. *math*. *math*.

Qed.

Lemma abs_eq_succ_abs_minus_one : $\forall (x:\text{int})$,

$x > 0 \rightarrow$

$\text{abs } x = S (\text{abs } (x - 1)) :> \text{nat}.$

Proof using.

intros. apply eq_nat_of_eq_int.

rewrite abs_nonneg; try *math*.

Qed.

14.5.7 Tactic *rew_abs_nonneg* to normalize expressions involving *abs*

issuing side-conditions that arguments are nonnegative

#[global]

Hint Rewrite *abs_nat abs_0 abs_1 abs_plus abs_nonneg* : *rew_abs_nonneg*.

Tactic Notation "rew_abs_nonneg" :=

autorewrite with *rew_abs_nonneg*.

Tactic Notation "rew_abs_nonneg" "~" :=

autorewrite with *rew_abs_nonneg*; try *math*; autos~.

14.5.8 Positive part of an integer. Returns 0 on negative values.

Notation "'to_nat'" := *Z.to_nat* (at level 0).

Lemma *to_nat_nat* : $\forall (n:\mathbf{nat})$,

to_nat *n* = *n*.

Proof using. exact *Nat2Z.id*. Qed.

Lemma *to_nat_nonneg* : $\forall (x:\mathbf{int})$,

$x \geq 0 \rightarrow$

to_nat *x* = *x* :> *int*.

Proof using. intros. apply~ *Z2Nat.id*. Qed.

Lemma *to_nat_neg* : $\forall (x:\mathbf{int})$,

$x \leq 0 \rightarrow$

to_nat *x* = 0%*nat*.

Proof using.

intros *x H*. destruct~ *x*.

assert (*Z.pos p* = 0) as $\rightarrow$. { *forwards*: *Z.le_0_pos p. math*. }

reflexivity.

Qed.

Lemma *to_nat_eq_nat_eq* : $\forall (x:\mathbf{int}) (y:\mathbf{nat})$,

$x \geq 0 \rightarrow$

(*to_nat* *x* = *y* :> *nat*) = (*x* = *Z.of_nat y* :> *int*).

Proof using.

introv M. extens. iff E.

{ subst. rewrite~ *Z2Nat.id*. }

{ subst. rewrite~ *Nat2Z.id*. }

Qed.

Lemma *lt_to_nat_to_nat* : $\forall (n\ m : \mathbf{int})$,

```

    (0 ≤ n) →
    (n < m) →
    (to_nat n < to_nat m).
Proof using.
  intros. nat_comp_to_peano.
  rewrite <! $Z$ abs2Nat.abs_nat_nonneg by math.
  apply! Zabs_nat_lt. math.
Qed.

Lemma to_nat_to_int : ∀ (n : int),
  0 ≤ n →
  Z_of_nat (to_nat n) = n.
Proof using. intros. rewrite! to_nat_nonneg. Qed.

Lemma to_nat_le_nat_le : ∀ (x:int) (y:nat),
  (0 ≤ x) →
  (to_nat x ≤ y) = (x ≤ y)%Z.
Proof.
  intros. extens. iff E.
  { rewrites! <!( $\gg$  to_nat_to_int x). math. }
  { rewrites! <!( $\gg$  to_nat_to_int x) in E. math. }
Qed.

Lemma to_nat_ge_nat_ge : ∀ (x:int) (y:nat),
  (0 ≤ x) →
  (to_nat x ≥ y) = (x ≥ y)%Z .
Proof.
  intros. extens. iff E.
  { rewrites! <!( $\gg$  to_nat_to_int x). math. }
  { rewrites! <!( $\gg$  to_nat_to_int x) in E. math. }
Qed.

```

14.5.9 to_nat distribute on constants and operators

```

Lemma to_nat_0 : to_nat 0 = 0%nat :> nat.
Proof using. reflexivity. Qed.

Lemma to_nat_1 : to_nat 1 = 1%nat :> nat.
Proof using. reflexivity. Qed.

Lemma to_nat_plus : ∀ (x y:int),
  (x ≥ 0) →
  (y ≥ 0) →
  to_nat (x + y) = (to_nat x + to_nat y)%nat :> nat.
Proof using. intros. apply! Z2Nat.inj_add. Qed.

Lemma to_nat_minus : ∀ (x y:int),

```

```

    (x ≥ y) →
    (y ≥ 0) →
    to_nat (x - y) = (to_nat x - to_nat y)%nat :> nat.
Proof using. intros. apply¬ Z2Nat.inj_sub. Qed.

```

14.5.10 Tactic *rew_to_nat_nonneg* to normalize expressions involving *to_nat*

issuing side-conditions that arguments are nonnegative

```

#[global]
Hint Rewrite to_nat_nat to_nat_0 to_nat_1 to_nat_plus to_nat_minus
    to_nat_nonneg : rew_to_nat_nonneg.
Tactic Notation "rew_to_nat_nonneg" :=
    autorewrite with rew_to_nat_nonneg.
Tactic Notation "rew_to_nat_nonneg" "~" :=
    autorewrite with rew_to_nat_nonneg; try math; autos¬.

```

Chapter 15

Library **SLF.LibMonoid**

DISCLAIMER: the current presentation of monoids uses typeclasses, but in fact it's not obvious that typeclasses are needed/useful here. Indeed, there is no overloading involved. Thus, the interface might change in the near future.

15.1

15.2

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibOperation.

Generalizable Variables *A B*.

15.3 Monoids

15.4 Structures

Monoid structure: binary operator and neutral element

```
Record monoid_op (A:Type) : Type := monoid_make {  
  monoid_oper : A → A → A;  
  monoid_neutral : A }.
```

Monoid properties Note that field names are suffixed by *_prop* because the corresponding properties are also available through typeclass instances.

```
Class Monoid A (m:monoid_op A) : Prop := Monoid_make {  
  monoid_assoc_prop : let (o,n) := m in assoc o;  
  monoid_neutral_l_prop : let (o,n) := m in neutral_l o n;
```

```

monoid_neutral_r_prop : let (o,n) := m in neutral_r o n }.

Commutative monoid

Class Comm_monoid A (m:monoid_op A) : Prop := Comm_monoid_make {
  comm_monoid_monoid : Monoid m;
  comm_monoid_comm : let (o,n) := m in comm o }.

```

15.5 Examples

Example:

Instance monoid_plus_zero: Monoid (monoid_make plus 0). Proof using. constructor;
repeat intro; lia. Qed.

15.6 Properties

```

Section MonoidProp.
Context {A:Type}.

Class Monoid_assoc {m:monoid_op A} := {
  monoid_assoc : assoc (monoid_oper m) }.

Class Monoid_neutral_l {m:monoid_op A} := {
  monoid_neutral_l : neutral_l (monoid_oper m) (monoid_neutral m) }.

Class Monoid_neutral_r {m:monoid_op A} := {
  monoid_neutral_r : neutral_r (monoid_oper m) (monoid_neutral m) }.

Class Monoid_comm {m:monoid_op A} := {
  monoid_comm : comm (monoid_oper m) }.

End MonoidProp.

```

15.7 Derived Properties

```

Section MonoidInst.
Variables (A:Type).
Implicit Types m : monoid_op A.

Global Instance Monoid_assoc_of_Monoid : ∀ m (M:Monoid m),
  Monoid_assoc (m:=m).
Proof using.
  introv M. constructor. destruct M as [U ? ?]. destruct m. simpl. apply U.
Qed.

Global Instance Monoid_neutral_l_of_Monoid : ∀ m (M:Monoid m),

```

Monoid_neutral_l ($m:=m$).

Proof using.

intros M . constructor. destruct M as [$? U$?]. destruct m . simpl. apply U .
Qed.

Global Instance Monoid_neutral_r_of_Monoid : $\forall m (M:\mathbf{Monoid} \ m)$,

Monoid_neutral_r ($m:=m$).

Proof using.

intros M . constructor. destruct M as [$? ? U$]. destruct m . simpl. apply U .
Qed.

Global Instance Monoid_of_Comm_monoid : $\forall m (M:\mathbf{Comm\_monoid} \ m)$,

Monoid m .

Proof using.

intros M . destruct M as [U ?]. destruct m . simpl. apply U .
Qed.

Global Instance Monoid_comm_of_Comm_Monoid : $\forall m (M:\mathbf{Comm\_monoid} \ m)$,

Monoid_comm ($m:=m$).

Proof using.

intros M . constructor. destruct M as [$? U$]. destruct m . simpl. apply U .
Qed.

End MonoidInst.

Chapter 16

Library SLF.LibContainer

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibLogic LibReflect
  LibRelation LibOperation LibInt LibMonoid.
Generalizable Variables A B K T.
```

16.1 Operators

16.1.1 Definitions

```
Class BagEmpty T := { empty : T }.
Class BagSingle A T := { single : A → T }.
Class BagSingleBind A B T := { single_bind : A → B → T }.
Class BagIn A T := { is_in : A → T → Prop }.
Class BagBinds A B T := { binds : T → A → B → Prop }.
Class BagRead A B T := { read : T → A → B }.
Class BagUpdate A B T := { update : T → A → B → T }.
Class BagUnion T := { union : T → T → T }.
Class BagInter T := { inter : T → T → T }.
Class BagIncl T := { incl : binary T }.
Class BagDisjoint T := { disjoint : binary T }.
Class BagRestrict T K := { restrict : T → K → T }.
Class BagRemove T K := { remove : T → K → T }.
Class BagFold I F T := { fold : monoid_op I → F → T → I }.
Class BagCard T := { card : T → nat }.
Class BagDom T K := { dom : T → K }.
Class BagImg T K := { img : T → K }.
Class BagIndex A T := { index : T → A → Prop }.

Definition notin '{BagIn A T} x m :=
```


$\neg (\text{is\_in } x \ m).$

16.1.2 Notation

Declare Scope *container_scope*.

Notation " $\{\}$ " := (empty)
: *container_scope*.

Notation " $\{ x \}$ " := (single x)
: *container_scope*.

Notation " $x \text{ 'in' } m$ " := (is_in $x \ m$)
(at level 39) : *container_scope*.

Notation " $x \text{ 'notin' } E$ " := (notin $x \ E$)
(at level 39) : *container_scope*.

Notation " $x \text{ ':=' } v$ " := (single_bind $x \ v$)
(at level 29) : *container_scope*.

Notation " $m \ [\ x \]$ " := (read $m \ x$)
(at level 7, *format* " $m \ [\ x \]$ ", left associativity).

Notation " $m \ [\ x \text{ ':=' } v \]$ " := (update $m \ x \ v$)
(at level 7, *format* " $m \ [\ x \text{ ':=' } v \]$ ", left associativity).

Notation " $m1 \text{ 'c' } m2$ " := (incl $m1 \ m2$)
(at level 38) : *container_scope*.

Notation " $m1 \text{ 'u' } m2$ " := (union $m1 \ m2$)
(at level 37, right associativity) : *container_scope*.

Notation " $m1 \text{ 'n' } m2$ " := (inter $m1 \ m2$)
(at level 36, right associativity) : *container_scope*.

Notation " $m1 \text{ '-' } m2$ " := (remove $m1 \ m2$)
(at level 35) : *container_scope*.

Notation " $m1 \text{ '|' } m2$ " := (restrict $m1 \ m2$)
(at level 34) : *container_scope*.

Notation " $m1 \text{ '#'} m2$ " := (disjoint $m1 \ m2$)
(at level 37, right associativity) : *container_scope*.

Open Scope *container_scope*.

Notation " $\{ x \}$ " := (single x) (*format* " $\{ x \}$ ")
: *container_scope*.

Notation " $M \setminus i$ " := ($M \setminus \{i\}$) (at level 35) : *container_scope*.

16.1.3 $\forall x \text{ in } E, P \ x$ notation

Notation " $\text{'forall_' } x \text{ 'in' } E \text{ ', ' } P$ " :=
($\forall x, x \text{ in } E \rightarrow P$)

(at level 200, x name) : *container_scope*.
 Notation "'forall_' x y 'in' E ',' P " :=
 ($\forall x\ y, x \text{ in } E \rightarrow y \text{ in } E \rightarrow P$)
 (at level 200, x name, y name) : *container_scope*.
 Notation "'forall_' x y z 'in' E ',' P " :=
 ($\forall x\ y\ z, x \text{ in } E \rightarrow y \text{ in } E \rightarrow z \text{ in } E \rightarrow P$)
 (at level 200, x name, y name, z name) : *container_scope*.

16.1.4 $\exists x \text{ in } E \text{ st } P\ x$ notation

Notation "'exists_' x 'in' E ',' P " :=
 ($\exists x, x \text{ in } E \wedge P$)
 (at level 200, x name) : *container_scope*.
 Notation "'exists_' x y 'in' E ',' P " :=
 ($\exists x, x \text{ in } E \wedge y \text{ in } E \wedge P$)
 (at level 200, x name, y name) : *container_scope*.
 Notation "'exists_' x y z 'in' E ',' P " :=
 ($\exists x, x \text{ in } E \wedge y \text{ in } E \wedge z \text{ in } E \wedge P$)
 (at level 200, x name, y name, z name) : *container_scope*.

16.1.5 Foreach

Definition foreach ' $\{BagIn\ A\ T\}$ ($P:A \rightarrow Prop$) ($E:T$) :=
 $\forall x, x \text{ in } E \rightarrow P\ x$.

16.1.6 index for natural numbers

Local Open Scope *Int_scope*.
 #[*global*]
 Instance int_index : **BagIndex** int int.
 Proof using. intros. constructor. exact (fun n (i :int) $\Rightarrow 0 \leq i < n$). Defined.
 Lemma int_index_eq : $\forall (n\ i : \text{int})$,
 index $n\ i = (0 \leq i < n)$.
 Proof using. auto. Qed.
 Global Opaque int_index.
 Lemma int_index_le : $\forall i\ n\ m : \text{int}$,
 index $n\ i \rightarrow n \leq m \rightarrow \text{index } m\ i$.
 Proof using. *introv*. do 2 rewrite @int_index_eq. *math*. Qed.
 Lemma int_index_prove : $\forall (n\ i : \text{int})$,
 $0 \leq i \rightarrow i < n \rightarrow \text{index } n\ i$.

Proof using. intros. rewrite $\neg$ int_index_eq. Qed.

Lemma int_index_succ : $\forall n\ i, n \geq 0 \rightarrow$
 index $(n + 1)\ i = (\text{index } n\ i \vee i = n)$.

Proof using.

introv P . do 2 rewrite int_index_eq. extens. iff H .
 apply or_classic_l. math.
 destruct H ; math.

Qed.

16.1.7 Derivable

Bag update can be defined as bag union with a singleton bag

#[global]

Instance bag_update_as_union_single : $\forall A\ B\ T$
 ‘{BagSingleBind $A\ B\ T$ } ‘{BagUnion T },
BagUpdate $A\ B\ T$.
 constructor. apply (fun $m\ k\ v \Rightarrow m \setminus u\ (k \setminus := v)$). Defined.

Global Opaque bag_update_as_union_single.

16.1.8 Properties

Section Properties.

Context { $A\ T$:Type}

{ BI : BagIn $A\ T$ } { BE : BagEmpty T } { BS : BagSingle $A\ T$ }
 { BN : BagInter T } { BU : BagUnion T } { BR : BagRemove $T\ T$ } { BC : BagCard T }
 { BL : BagIncl T } { BD : BagDisjoint T }.

In

Class In_empty_eq :=
 { in_empty_eq : $\forall x, x \setminus \text{in } \{\}$ = False }.

Class In_empty :=
 { in_empty : $\forall x, x \setminus \text{in } \{\} \rightarrow \text{False}$ }.

Class Notin_eq :=
 { notin_eq : $\forall x\ E, (x \setminus \text{notin } E) = \neg (x \setminus \text{in } E)$ }.

Class Notin_empty :=
 { notin_empty : $\forall x, x \setminus \text{notin } \{\}$ }.

Class In_single_eq :=
 { in_single_eq : $\forall x\ y, x \setminus \text{in } \{y\} = (x = y)$ }.

Class In_single :=
 { in_single : $\forall x\ y, x \setminus \text{in } \{y\} \rightarrow x = y$ }.

Class In_single_self :=

```

{ in_single_self :  $\forall x, x \in \{x\}$  }.
Class In_extens_eq :=
{ in_extens_eq :  $\forall E F, (\forall x, x \in E = x \in F) \rightarrow E = F$  }.
Class In_extens :=
{ in_extens :  $\forall E F, (\forall x, x \in E \leftrightarrow x \in F) \rightarrow E = F$  }.
Class Is_empty_eq :=
{ is_empty_eq :  $\forall E, (E = \{\}) = (\forall x, x \in E \rightarrow \text{False})$  }.
Class Is_empty_prove :=
{ is_empty_prove :  $\forall E, (\forall x, x \in E \rightarrow \text{False}) \rightarrow E = \{\}$  }.
Class Is_empty_inv :=
{ is_empty_inv :  $\forall x E, E = \{\} \rightarrow x \in E \rightarrow \text{False}$  }.
Class Is_nonempty_prove :=
{ is_nonempty_prove :  $\forall x E, x \in E \rightarrow E \neq \{\}$  }.
Class Is_single_eq :=
{ is_single_eq :  $\forall x E, (E = \{x\}) = (x \in E \wedge (\forall y, y \in E \rightarrow y = x))$  }.
Class Is_single_prove :=
{ is_single_prove :  $\forall x E, x \in E \rightarrow (\forall y, y \in E \rightarrow y = x) \rightarrow E = \{x\}$  }.
Class Is_single_inv :=
{ is_single_inv :  $\forall x y E, E = \{x\} \rightarrow y \in E \rightarrow y = x$  }.
Class Notin_single_eq :=
{ notin_single_eq :  $\forall x y, x \notin \{y\} = (x \neq y)$  }.
Class In_inter_eq :=
{ in_inter_eq :  $\forall x E F, x \in (E \cap F) = (x \in E \wedge x \in F)$  }.
Class In_inter :=
{ in_inter :  $\forall x E F, x \in E \rightarrow x \in F \rightarrow x \in (E \cap F)$  }.
Class In_inter_inv :=
{ in_inter_inv :  $\forall x E F, x \in (E \cap F) \rightarrow x \in E \wedge x \in F$  }.
Class Notin_inter_eq :=
{ notin_inter_eq :  $\forall x E F, x \notin (E \cap F) = (x \notin E \vee x \notin F)$  }.
Class Notin_inter_l :=
{ notin_inter_l :  $\forall x E F, x \notin E \rightarrow x \notin (E \cap F)$  }.
Class Notin_inter_r :=
{ notin_inter_r :  $\forall x E F, x \notin F \rightarrow x \notin (E \cap F)$  }.
Class Notin_inter_inv :=
{ notin_inter_inv :  $\forall x E F, x \notin (E \cap F) \rightarrow x \notin E \vee x \notin F$  }.
Class In_union_eq :=
{ in_union_eq :  $\forall x (E F : T), x \in (E \cup F) = (x \in E \vee x \in F)$  }.
Class In_union_l :=
{ in_union_l :  $\forall x E F, x \in E \rightarrow x \in (E \cup F)$  }.
Class In_union_r :=
{ in_union_r :  $\forall x E F, x \in F \rightarrow x \in (E \cup F)$  }.

```

```

Class In_union_inv :=
  { in_union_inv :  $\forall x (E F : T), x \in (E \cup F) \rightarrow (x \in E \vee x \in F)$  }.

Class Notin_union_eq :=
  { notin_union_eq :  $\forall x E F, x \notin (E \cup F) = (x \notin E \wedge x \notin F)$  }.

Class Notin_union :=
  { notin_union :  $\forall x E F, x \notin E \rightarrow x \notin F \rightarrow x \notin (E \cup F)$  }.

Class Notin_union_inv :=
  { notin_union_inv :  $\forall x E F, x \notin (E \cup F) \rightarrow x \notin E \wedge x \notin F$  }.

Class In_remove_eq :=
  { in_remove_eq :  $\forall x (E F : T), x \in (E \setminus F) = (x \in E \wedge x \notin F)$  }.

Class Remove_incl :=
  { remove_incl :  $\forall (E F : T), (E \setminus F) \subseteq E$  }.

Class Remove_disjoint :=
  { remove_disjoint :  $\forall (E F : T), F \# (E \setminus F)$  }.

Incl

Class Incl_in_eq :=
  { incl_in_eq :  $\forall E F, (E \subseteq F) = (\forall x, x \in E \rightarrow x \in F)$  }.

Class Incl_prove :=
  { incl_prove :  $\forall E F, (\forall x, x \in E \rightarrow x \in F) \rightarrow E \subseteq F$  }.

Class Incl_inv :=
  { incl_inv :  $\forall x E F, E \subseteq F \rightarrow x \in E \rightarrow x \in F$  }.

Class Incl_refl :=
  { incl_refl : refl incl }.

Class Incl_trans :=
  { incl_trans : trans incl }.

Class Incl_antisym :=
  { incl_antisym : antisym incl }.

Class Incl_order :=
  { incl_order : LibOrder.order incl }.

Class Empty_incl :=
  { empty_incl :  $\forall E, \{\} \subseteq E$  }.

Class Incl_empty :=
  { incl_empty :  $\forall E, (E \subseteq \{\}) = (E = \{\})$  }.

Class Incl_empty_inv :=
  { incl_empty_inv :  $\forall E, E \subseteq \{\} \rightarrow E = \{\}$  }.

Class Single_incl_r_eq :=
  { single_incl_r_eq :  $\forall x E, (\{x\} \subseteq E) = (x \in E)$  }.

Class Single_incl_r :=
  { single_incl_r :  $\forall x E, x \in E \rightarrow \{x\} \subseteq E$  }.

Class Single_incl_l_eq :=

```

```

    { single_incl_l_eq :  $\forall x E, (E \setminus c \setminus \{x\}) = (E = \setminus \{\} \vee E = \setminus \{x\})$  }.
Class Incl_union_l :=
  { incl_union_l :  $\forall E F G, E \setminus c F \rightarrow E \setminus c (F \setminus u G)$  }.
Class Incl_union_r :=
  { incl_union_r :  $\forall E F G, E \setminus c G \rightarrow E \setminus c (F \setminus u G)$  }.
Class Union_incl_eq :=
  { union_incl_eq :  $\forall E F G, ((E \setminus u F) \setminus c G) = (E \setminus c G \wedge F \setminus c G)$  }.
Class Union_incl :=
  { union_incl :  $\forall E F G, E \setminus c G \rightarrow F \setminus c G \rightarrow (E \setminus u F) \setminus c G$  }.
Class Union_incl_inv :=
  { union_incl_inv :  $\forall E F G, (E \setminus u F) \setminus c G \rightarrow E \setminus c G \wedge F \setminus c G$  }.
Class Union_incl_union :=
  { union_incl_union :  $\forall E1 F1 E2 F2, E1 \setminus c F1 \rightarrow E2 \setminus c F2 \rightarrow (E1 \setminus u E2) \setminus c (F1 \setminus u F2)$  }.
Class Union_incl_union_same_l :=
  { union_incl_union_same_l :  $\forall E F1 F2, F1 \setminus c F2 \rightarrow (E \setminus u F1) \setminus c (E \setminus u F2)$  }.
Class Union_incl_union_same_r :=
  { union_incl_union_same_r :  $\forall E F1 F2, F1 \setminus c F2 \rightarrow (F1 \setminus u E) \setminus c (F2 \setminus u E)$  }.
Class Incl_inter_eq :=
  { incl_inter_eq :  $\forall E F G, E \setminus c (F \setminus n G) = (E \setminus c F \wedge E \setminus c G)$  }.
Class Incl_inter :=
  { incl_inter :  $\forall E F G, E \setminus c F \rightarrow E \setminus c G \rightarrow E \setminus c (F \setminus n G)$  }.
Class Incl_inter_inv :=
  { incl_inter_inv :  $\forall E F G, E \setminus c (F \setminus n G) \rightarrow E \setminus c F \wedge E \setminus c G$  }.

Union

Class Union_assoc :=
  { union_assoc : assoc union }.
Class Union_comm :=
  { union_comm : comm union }.
Class Union_comm_assoc :=
  { union_comm_assoc : comm_assoc union }.
Class Union_empty_l :=
  { union_empty_l : neutral_l union empty }.
Class Union_empty_r :=
  { union_empty_r : neutral_r union empty }.
Class Union_empty_inv :=
  { union_empty_inv :  $\forall E F, E \setminus u F = \setminus \{\} \rightarrow E = \setminus \{\} \wedge F = \setminus \{\}$  }.
Class Union_self :=
  { union_self : idempotent2 union }.

Intersection

Class Inter_assoc :=

```

```

    { inter_assoc : assoc inter }.
Class Inter_comm :=
    { inter_comm : comm inter }.
Class Inter_comm_assoc :=
    { inter_comm_assoc : comm_assoc inter }.
Class Inter_empty_l :=
    { inter_empty_l : absorb_l inter empty }.
Class Inter_empty_r :=
    { inter_empty_r : absorb_r inter empty }.
Class Inter_self :=
    { inter_self : idempotent2 inter }.

Removal

Cardinal

Class Card_empty :=
    { card_empty : card \{\} = 0%nat }.
Class Card_single :=
    { card_single :  $\forall X$ , card \{X\} = 1%nat }.
Class Card_union :=
    { card_union :  $\forall E F$ , card (E \u F) = (card E + card F)%nat }.
Class Card_disjoin_union :=
    { card_disjoin_union :  $\forall E F$ , E \# F  $\rightarrow$  card (E \u F) = (card E + card F)%nat }.
Class Card_union_le :=
    { card_union_le :  $\forall E F$ , card (E \u F)  $\leq$  (card E + card F)%nat }.

Disjointness

Class Disjoint_sym :=
    { disjoint_sym : sym disjoint }.
Class Disjoint_eq :=
    { disjoint_eq :  $\forall E F$ , (E \# F) = ( $\forall x, x \text{ \textit{in} } E \rightarrow x \text{ \textit{in} } F \rightarrow \mathbf{False}$ ) }.
Class Disjoint_not_eq :=
    { disjoint_not_eq :  $\forall E F$ , ( $\neg$  (E \# F)) = ( $\exists x, x \text{ \textit{in} } E \wedge x \text{ \textit{in} } F$ ) }.
Class Disjoint_prove :=
    { disjoint_prove :  $\forall E F$ , ( $\forall x, x \text{ \textit{in} } E \rightarrow x \text{ \textit{in} } F \rightarrow \mathbf{False}$ )  $\rightarrow E \text{ \# } F$  }.
Class Disjoint_inv :=
    { disjoint_inv :  $\forall x E F$ , (E \# F)  $\rightarrow x \text{ \textit{in} } E \rightarrow x \text{ \textit{in} } F \rightarrow \mathbf{False}$  }.
Class Disjoint_single_l_eq :=
    { disjoint_single_l_eq :  $\forall x E$ , (x \# E) = x \notin E }.
Class Disjoint_single_r_eq :=
    { disjoint_single_r_eq :  $\forall x E$ , (E \# x) = x \notin E }.
Class Inter_disjoint :=
    { inter_disjoint :  $\forall E F$ , E \# F  $\rightarrow E \text{ \n } F = \{\}$  }.

```

End Properties.

Lemmas with premises and operators in the conclusion need additional Arguments

Arguments is_empty_inv {A} {T} {BI} {BE} {Is_empty_inv} [x] [E].
Arguments is_nonempty_prove {A} {T} {BI} {BE} {Is_nonempty_prove} [x] [E].
Arguments in_single A T {BI} {BS} {In_single} [x] [y].
Arguments is_single_inv {A} {T} {BI} {BS} {Is_single_inv} [x] y [E].
Arguments in_inter {A} {T} {BI} {BN} {In_inter} [x] [E] [F].
Arguments in_inter_inv {A} {T} {BI} {BN} {In_inter_inv} [x] [E] [F].
Arguments notin_inter_l {A} {T} {BI} {BN} {Notin_inter_l} [x] [E] [F].
Arguments notin_inter_r {A} {T} {BI} {BN} {Notin_inter_r} [x] [E] [F].
Arguments notin_inter_inv {A} {T} {BI} {BN} {Notin_inter_inv} [x] [E] [F].
Arguments in_union_l {A} {T} {BI} {BU} {In_union_l} [x] [E] [F].
Arguments in_union_r {A} {T} {BI} {BU} {In_union_r} [x] [E] [F].
Arguments in_union_inv {A} {T} {BI} {BU} {In_union_inv} [x] [E] [F].
Arguments notin_union {A} {T} {BI} {BU} {Notin_union} [x] [E] [F].
Arguments notin_union_inv {A} {T} {BI} {BU} {Notin_union_inv} [x] [E] [F].
Arguments incl_prove {A} {T} {BI} {BL} {Incl_prove} [E] [F].
Arguments incl_inv {A} {T} {BI} {BL} {Incl_inv} [x] [E] [F].
Arguments incl_trans {T} [BL] {Incl_trans} y [x] [z].
Arguments incl_empty_inv {T} {BE} {BL} {Incl_empty_inv} [E].
Arguments incl_union_l {T} {BU} {BL} {Incl_union_l} [E] [F] [G].
Arguments incl_union_r {T} {BU} {BL} {Incl_union_r} [E] [F] [G].
Arguments incl_inter {T} {BN} {BL} {Incl_inter} [E] [F] [G].
Arguments incl_inter_inv {T} {BN} {BL} {Incl_inter_inv} [E] [F] [G].
Arguments union_incl_union {T} {BU} {BL} {Union_incl_union} [E1] [F1] [E2] [F2].
Arguments union_incl_union_same_l {T} {BU} {BL} {Union_incl_union_same_l} [E] [F1] [F2].
Arguments union_incl_union_same_r {T} {BU} {BL} {Union_incl_union_same_r} [E] [F1] [F2].
Arguments union_empty_inv {T} {BE} {BU} {Union_empty_inv} [E] [F].
Arguments disjoint_sym {T} {BD} {Disjoint_sym}.
Arguments disjoint_prove {A} {T} {BI} {BD} {Disjoint_prove} [E] [F].
Arguments disjoint_inv {A} {T} {BI} {BD} {Disjoint_inv} [x] [E] [F].
Arguments disjoint_single_l_eq {A} {T} {BI} {BS} {BD} {Disjoint_single_l_eq}.
Arguments disjoint_single_r_eq {A} {T} {BI} {BS} {BD} {Disjoint_single_r_eq}.

16.1.9 Derived Properties

Section DerivedProperties.


```

Context {A T:Type}
  {BI: BagIn A T} {BE: BagEmpty T} {BS: BagSingle A T}
  {BN: BagInter T} {BU: BagUnion T} {BR: BagRemove T T} {BC: BagCard T}
  {BL: BagIncl T} {BD: BagDisjoint T}.

In

Global Instance in_empty_of_in_empty_eq :
  In_empty_eq →
  In_empty.
Proof using. constructor. introv I. rewrite¬ in_empty_eq in I. Qed.

Global Instance notin_eq_of_nothing :
  Notin_eq.
Proof using. constructor. intros. unfold notin. auto. Qed.

Global Instance notin_empty_of_in_empty_eq :
  In_empty_eq →
  Notin_empty.
Proof using. constructor. introv I. rewrite¬ in_empty_eq in I. Qed.

Global Instance in_single_of_in_single_eq :
  In_single_eq →
  In_single.
Proof using. constructor. introv I. rewrite¬ in_single_eq in I. Qed.

Global Instance in_single_self_of_in_single_eq :
  In_single_eq →
  In_single_self.
Proof using. constructor. intros. rewrite¬ in_single_eq. Qed.

Global Instance in_extens_eq_of_in_extens :
  In_extens →
  In_extens_eq.
Proof using. constructor. introv I. apply in_extens. intros. rewrite× I. Qed.

Global Instance is_empty_eq_of_in_empty_eq :
  In_extens →
  In_empty_eq →
  Is_empty_eq.
Proof using.
  constructor. intros. extens. iff M.
    subst. introv N. rewrite× in_empty_eq in N.
    apply in_extens. iff N. false× M. rewrite× in_empty_eq in N.
Qed.

Global Instance is_empty_prove_of_is_empty_eq :
  Is_empty_eq →
  Is_empty_prove.

```

Proof using. constructor. *intros I*. rewrite× is_empty_eq. Qed.

Global Instance is_empty_inv_of_is_empty_eq :
is_empty_eq →
is_empty_inv.

Proof using. constructor. *intros I1 I2*. rewrite× is_empty_eq in *I1*. Qed.

Global Instance is_nonempty_prove_of_is_empty_eq :
is_empty_eq →
is_nonempty_prove.

Proof using. constructor. *intros I1 I2*. rewrite is_empty_eq in *I2*. eauto. Qed.

Global Instance is_single_eq_of_in_single_eq :
In_extens →
In_single_eq →
is_single_eq.

Proof using.
 constructor. intros. *extens*. iff *M* (*M1*&*M2*).
 subst. split. rewrite× in_single_eq. *intros N*. rewrite× in_single_eq in *N*.
 apply in_extens. iff *N*.
 rewrite× (*M2 x0*). rewrite× in_single_eq.
 rewrite× in_single_eq in *N*. subst×.

Qed.

Global Instance is_single_prove_of_is_single_eq :
is_single_eq →
is_single_prove.

Proof using. constructor. *intros I*. rewrite× is_single_eq. Qed.

Global Instance is_single_inv_of_is_single_eq :
is_single_eq →
is_single_inv.

Proof using. constructor. *intros I1 I2*. rewrite× is_single_eq in *I1*. Qed.

Global Instance notin_single_eq_of_in_single_eq :
In_single_eq →
Notin_single_eq.

Proof using. constructor. intros. unfold notin. rewrite in_single_eq. eauto. Qed.

Global Instance in_inter_of_in_inter_eq :
In_inter_eq →
In_inter.

Proof using. constructor. *intros I1 I2*. rewrite× in_inter_eq. Qed.

Global Instance in_inter_inv_of_in_inter_eq :
In_inter_eq →
In_inter_inv.

Proof using. constructor. *intros I*. rewrite $\neg \leftarrow$ in_inter_eq. Qed.

```

Global Instance notin_inter_l_of_notin_inter_eq :
  Notin_inter_eq →
  Notin_inter_l.
Proof using. constructor. introv I. rewrite¬ notin_inter_eq. Qed.

Global Instance notin_inter_r_of_notin_inter_eq :
  Notin_inter_eq →
  Notin_inter_r.
Proof using. constructor. introv I. rewrite¬ notin_inter_eq. Qed.

Global Instance notin_inter_inv_of_notin_inter_eq :
  Notin_inter_eq →
  Notin_inter_inv.
Proof using. constructor. introv I. rewrite¬ notin_inter_eq in I. Qed.

Global Instance in_union_l_of_in_union_eq :
  In_union_eq →
  In_union_l.
Proof using. constructor. introv I. rewrite× in_union_eq. Qed.

Global Instance in_union_r_of_in_union_eq :
  In_union_eq →
  In_union_r.
Proof using. constructor. introv I. rewrite× in_union_eq. Qed.

Global Instance in_union_inv_of_in_union_eq :
  In_union_eq →
  In_union_inv.
Proof using. constructor. introv I. rewrite¬ @in_union_eq in I. Qed.

Global Instance notin_union_of_notin_union_eq :
  Notin_union_eq →
  Notin_union.
Proof using. constructor. introv I1 I2. rewrite¬ notin_union_eq. Qed.

Global Instance notin_union_inv_of_notin_union_eq :
  Notin_union_eq →
  Notin_union_inv.
Proof using. constructor. introv I. rewrite¬ notin_union_eq in I. Qed.

Incl

Global Instance incl_prove_of_in_eq :
  Incl_in_eq →
  Incl_prove.
Proof using. constructor. introv I. rewrite× incl_in_eq. Qed.

Global Instance incl_inv_of_in_eq :
  Incl_in_eq →

```

Incl_inv.

Proof using. constructor. *intros I1 I2. rewrite× incl_in_eq in I1. Qed.*

Global Instance incl_order_of_incl_in_eq :

In_extens →

Incl_in_eq →

Incl_order.

Proof using.

constructor. constructor.

intros *x*. rewrite× incl_in_eq.

intros *E F G I1 I2*. rewrite incl_in_eq. rewrite incl_in_eq in *I1,I2*. *autos×.*

intros *E F I1 I2*. rewrite incl_in_eq in *I1,I2*. apply× in_extens.

Qed.

Global Instance incl_refl_of_incl_order :

Incl_order →

Incl_refl.

Proof using. constructor. apply order_refl. apply incl_order. Qed.

Global Instance incl_trans_of_incl_order :

Incl_order →

Incl_trans.

Proof using. constructor. apply order_trans. apply incl_order. Qed.

Global Instance incl_antisym_of_incl_order :

Incl_order →

Incl_antisym.

Proof using. constructor. apply order_antisym. apply incl_order. Qed.

Global Instance empty_incl_inv_of_incl_in_eq_and_in_empty_eq :

Incl_in_eq →

In_empty_eq →

Empty_incl.

Proof using.

constructor. intros. rewrite incl_in_eq. *intros M.*

rewrite in_empty_eq in *M*. *false.*

Qed.

Global Instance incl_empty_of_in_empty_eq_and_incl_in_eq :

In_extens →

In_empty_eq →

Incl_in_eq →

Incl_empty.

Proof using.

constructor. intros. *extens.* rewrite incl_in_eq. *iff M.*

apply in_extens. *iff N. applies× M.* rewrite in_empty_eq in *N*. *false.*

subst. *intros N.* rewrite in_empty_eq in *N*. *false.*

Qed.

Global Instance incl_empty_inv_of_incl_empty :

Incl_empty →
Incl_empty_inv.

Proof using. constructor. *intros I*. rewrite $\neg$ incl_empty in *I*. Qed.

Global Instance single_incl_r_eq_of_in_single_eq_and_and_incl_in_eq :

In_extens →
In_single_eq →
Incl_in_eq →
Single_incl_r_eq.

Proof using.

constructor. intros. *extens*. rewrite incl_in_eq. *iff M*.
applies $\times$ *M*. rewrite $\neg$ in_single_eq.
intros N. rewrite in_single_eq in *N*. subst $\neg$.

Qed.

Global Instance single_incl_r_of_single_incl_r_eq:

Single_incl_r_eq →
Single_incl_r.

Proof using.

constructor. intros. rewrite single_incl_r_eq. assumption.

Qed.

Global Instance single_incl_l_eq_of_in_empty_eq_and_in_single_eq_and_and_incl_in_eq :

In_extens →
In_empty_eq →
In_single_eq →
Incl_in_eq →
Single_incl_l_eq.

Proof using.

constructor. intros. *extens*. rewrite incl_in_eq. *iff M*.
tests: (x \in E).
right. apply $\times$ is_single_prove. *intros N*. *forwards* $\neg$ *R: M y*.
rewrite $\times$ in_single_eq in *R*.
left. apply in_extens. *iff N*. *forwards* $\neg$ *R: M x0*.
rewrite $\times$ in_single_eq in *R*. subst. *false* $\times$.
rewrite in_empty_eq in *N*. *false*.
intros N. rewrite in_single_eq. destruct *M*.
subst. rewrite in_empty_eq in *N*. *false*.
subst. rewrite in_single_eq in *N*. auto.

Qed.

Global Instance union_incl_eq_of_in_union_eq_and_and_incl_in_eq :

In_extens →

In_union_eq →
Incl_in_eq →
Union_incl_eq.

Proof using.

```

constructor. intros. extends. repeat rewrite incl_in_eq. iff M (M1&M2).
  split. intros x N. specializes M x. rewrite× in_union_eq in M.
    intros x N. specializes M x. rewrite× in_union_eq in M.
      intros x N. specializes M1 x. specializes M2 x. rewrite× in_union_eq in N.

```

Qed.

Global Instance union_incl_union_of_union_incl_and_incl_union_l_and_incl_union_r :

Union_incl →
Incl_union_l →
Incl_union_r →
Union_incl_union.

Proof using.

```

constructor. intros. apply union_incl.
{ applies× incl_union_l. } { applies× incl_union_r. }

```

Qed.

Global Instance union_incl_union_same_l_of_union_incl_union_and_incl_refl :

Union_incl_union →
Incl_refl →
Union_incl_union_same_l.

Proof using.

```

constructor. intros. apply union_incl_union.
{ applies× incl_refl. } { auto. }

```

Qed.

Global Instance union_incl_union_same_r_of_union_incl_union_same_l_and_union_comm :

Union_incl_union_same_l →
Union_comm →
Union_incl_union_same_r.

Proof using.

```

constructor. intros. rewrite (union_comm F1). rewrite (union_comm F2).
  apply union_incl_union_same_l. auto.

```

Qed.

Global Instance incl_union_l_of_incl_in_eq_and_in_union_eq :

Incl_in_eq →
In_union_eq →
Incl_union_l.

Proof using.

```

constructor. introv I. rewrite incl_in_eq. rewrite incl_in_eq in I.

```

```

    intros N. rewrite× in_union_eq.
Qed.

Global Instance incl_union_r_of_incl_union_l :
  Incl_union_l →
  Union_comm →
  Incl_union_r.
Proof using. constructor. intros I. rewrite union_comm. apply× @incl_union_l. Qed.

Global Instance union_incl_of_union_incl_eq :
  Union_incl_eq →
  Union_incl_eq.
Proof using. constructor. intros_all. rewrite× union_incl_eq. Qed.

Global Instance union_incl_inv_of_union_incl_eq :
  Union_incl_eq →
  Union_incl_inv.
Proof using. constructor. intros I. rewrite union_incl_eq in I. destruct× I. Qed.

Global Instance incl_inter_eq_of_in_inter_eq_and_and_incl_in_eq :
  In_extens →
  In_inter_eq →
  Incl_in_eq →
  Incl_inter_eq.
Proof using.
  constructor. intros. extens. repeat rewrite incl_in_eq. iff M (M1&M2).
  split. intros x N. specializes M x. rewrite× in_inter_eq in M.
    intros x N. specializes M x. rewrite× in_inter_eq in M.
    intros x N. specializes M1 N. specializes M2 N. rewrite× in_inter_eq.
Qed.

Global Instance incl_inter_of_incl_inter_eq :
  Incl_inter_eq →
  Incl_inter.
Proof using. constructor. intros. rewrite× incl_inter_eq. Qed.

Global Instance incl_inter_inv_of_incl_inter_eq :
  Incl_inter_eq →
  Incl_inter_inv.
Proof using. constructor. intros N. rewrite× incl_inter_eq in N. Qed.

  Local tactic contain_by_in_double to prove inclusion

Hint Rewrite @in_union_eq @in_inter_eq
  @in_empty_eq @in_single_eq : rew_in_eq.

Ltac contain_by_in_double :=
  intros_all; apply in_extens; intros;
  autorewrite with rew_in_eq;

```

```

intuition (try solve [auto|eauto|auto_false|false]).
Union
Global Instance union_comm_form_in_union_eq :
  In_extens →
  In_union_eq →
  Union_comm.
Proof using. constructor. contain_by_in_double. Qed.
Global Instance union_assoc_form_in_union_eq :
  In_extens →
  In_union_eq →
  Union_assoc.
Proof using. constructor. contain_by_in_double. Qed.
Global Instance union_comm_assoc_of_union_comm_and_union_assoc :
  Union_comm →
  Union_assoc →
  Union_comm_assoc.
Proof using.
  constructor. intros_all. do 2 rewrite union_assoc.
  rewrite (union_comm _ x). auto.
Qed.
Global Instance union_empty_l_of_in_union_eq_and_in_empty_eq :
  In_extens →
  In_union_eq →
  In_empty_eq →
  Union_empty_l.
Proof using. constructor. contain_by_in_double. Qed.
Global Instance union_empty_r_of_union_empty_l :
  Union_empty_l → Union_comm → Union_empty_r.
Proof using. constructor. intros_all. rewrite union_comm. apply union_empty_l. Qed.
Global Instance union_empty_inv_of_in_union_eq :
  In_extens →
  In_empty_eq →
  In_union_eq →
  Union_empty_inv.
Proof using.
  constructor. introv N. split.
  apply in_extens. iff R. rewrite ← N. rewrite× in_union_eq. rewrite× in_empty_eq
in R.
  apply in_extens. iff R. rewrite ← N. rewrite× in_union_eq. rewrite× in_empty_eq
in R.
Qed.

```



```

Global Instance union_self_of_in_union_eq :
  In_extens →
  In_union_eq →
  Union_self.
Proof using. constructor. contain_by_in_double. Qed.
Global Instance notin_union_eq_of_in_union_eq :
  In_union_eq →
  Notin_union_eq.
Proof using. constructor. intros. unfold notin. rewrite @in_union_eq. extens×.
eauto. Qed.

Inter

Global Instance inter_comm_form_in_inter_eq :
  In_extens →
  In_inter_eq →
  Inter_comm.
Proof using. constructor. contain_by_in_double. Qed.
Global Instance inter_assoc_form_in_inter_eq :
  In_extens →
  In_inter_eq →
  Inter_assoc.
Proof using. constructor. contain_by_in_double. Qed.
Global Instance inter_comm_assoc_of_inter_comm_and_inter_assoc :
  Inter_comm →
  Inter_assoc →
  Inter_comm_assoc.
Proof using.
  constructor. intros_all. do 2 rewrite inter_assoc.
  rewrite (inter_comm - x). auto.
Qed.
Global Instance inter_empty_l_of_in_inter_eq_and_in_empty_eq :
  In_extens →
  In_inter_eq →
  In_empty_eq →
  Inter_empty_l.
Proof using. constructor. contain_by_in_double. Qed.
Global Instance inter_empty_r_of_inter_empty_l :
  Inter_empty_l →
  Inter_comm →
  Inter_empty_r.
Proof using. constructor. intros_all. rewrite inter_comm. apply inter_empty_l. Qed.
Global Instance inter_self_of_in_inter_eq :

```

```

In_extens →
In_inter_eq →
Inter_self.
Proof using. constructor. contain_by_in_double. Qed.

Remove

Global Instance remove_incl_of_in_remove_eq_and_incl_in_eq :
In_remove_eq →
Incl_in_eq →
Remove_incl.
Proof using.
  constructor. intros. rewrite incl_in_eq. introv N. rewrite× in_remove_eq in N.
Qed.

Global Instance remove_disjoint_of_in_remove_eq_and_disjoint_eq :
In_remove_eq →
Disjoint_eq →
Remove_disjoint.
Proof using.
  constructor. intros. rewrite disjoint_eq. introv N M.
  rewrite× in_remove_eq in M.
Qed.

Disjoint

Global Instance disjoint_not_eq_of_disjoint_eq :
Disjoint_eq →
Disjoint_not_eq.
Proof using.
  constructor. intros. rewrite disjoint_eq. extens. iff M.
  { rew_logic in M. destruct M as [x M]. rew_logic× in M. }
  { destruct M as [x M]. rew_logic. ∃ x. rew_logic×. }
Qed.

Global Instance disjoint_prove_of_disjoint_eq :
Disjoint_eq →
Disjoint_prove.
Proof using. constructor. intros. rewrite× disjoint_eq. Qed.

Global Instance disjoint_inv_of_disjoint_eq :
Disjoint_eq →
Disjoint_inv.
Proof using. constructor. introv I I1 I2. rewrite× disjoint_eq in I. Qed.

Global Instance disjoint_single_l_eq_of_disjoint_eq_and_in_single_eq :
Disjoint_eq →
In_single_eq →

```

Disjoint_single_l_eq.

Proof using.

```
constructor. intros. rewrite disjoint_eq. unfold notin. extends. iff M.  
  introv N. specializes M N. rewrite× in_single_eq. false.  
  introv N1 N2. rewrite in_single_eq in N1. subst. false.
```

Qed.

Global Instance disjoint_sym_of_disjoint_eq :

Disjoint_eq →

Disjoint_sym.

Proof using. constructor. intros *x y*. do 2 rewrite× disjoint_eq. Qed.

Global Instance disjoint_single_r_eq_of_disjoint_single_l :

Disjoint_single_l_eq →

Disjoint_sym →

Disjoint_single_r_eq.

Proof using.

```
constructor. intros_all.  
rewrites (» sym_inv_eq disjoint_sym). apply disjoint_single_l_eq.
```

Qed.

Global Instance inter_disjoint_of_disjoint_eq_and_in_inter_eq :

In_extens →

In_empty_eq →

Disjoint_eq →

In_inter_eq →

Inter_disjoint.

Proof using.

```
constructor. introv M. apply in_extens. rewrite disjoint_eq in M. iff N.  
  rewrite in_inter_eq in N. false×.  
  rewrite in_empty_eq in N. false.
```

Qed.

End DerivedProperties.

Chapter 17

Library SLF.LibOption

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibReflect.
Generalizable Variables A.
```

17.1 Option type

17.1.1 Definition

From the Prelude:

Inductive option A : Type := | Some : A -> option A | None : option A.

Arguments Some {A}.

Arguments None {A}.

17.1.2 Inhabited

#[*global*]

Instance Inhab_option : $\forall A$, Inhab (option A).

Proof using. intros. apply (Inhab_of_val None). Qed.

17.2 Operations

17.2.1 is_some

is_some o holds when o is of the form Some x

Definition is_some A (o:option A) :=

```
  match o with
  | None => false
```

```
| Some _ ⇒ true
end.
```

17.2.2 **unsome_default**

`unsome_default d o` returns the content of the option, and returns `d` in case the option is `None`.

```
Definition unsome_default A d (o:option A) :=
  match o with
  | None ⇒ d
  | Some x ⇒ x
  end.
```

17.2.3 **unsome**

`unsome o` returns the content of the option, and returns an arbitrary value in case the option is `None`.

```
Definition unsome {Inhab A} :=
  unsome_default (arbitrary (A:=A)).
```

17.2.4 **map**

`map f o` takes an option and returns an option, and maps the function `f` to the content of the option if it has one.

```
Definition map A B (f : A → B) (o : option A) : option B :=
  match o with
  | None ⇒ None
  | Some x ⇒ Some (f x)
  end.
```

```
Lemma map_eq_none_inv : ∀ A B (f : A → B) o,
  map f o = None →
  o = None.
```

Proof using. *intros H. destruct o; simpl; inverts × H. Qed.*

```
Lemma map_eq_some_inv : ∀ A B (f : A → B) o x,
  map f o = Some x →
  ∃ z, o = Some z ∧ x = f z.
```

Proof using. *intros H. destruct o; simpl; inverts × H. Qed.*

Arguments map_eq_none_inv [A] [B] [f] [o].

Arguments map_eq_some_inv [A] [B] [f] [o] [x].

17.2.5 map_on

Definition map_on $A B$ ($o : \text{option } A$) ($f : A \rightarrow B$) : $\text{option } B :=$
map f o .

Lemma map_on_eq_none_inv : $\forall A B (f : A \rightarrow B) o,$
map f $o = \text{None} \rightarrow$
 $o = \text{None}.$

Proof using. *intros H. destruct $\neg o$; try false. Qed.*

Lemma map_on_eq_some_inv : $\forall A B (f : A \rightarrow B) o x,$
map_on $o f = \text{Some } x \rightarrow$
 $\exists z, o = \text{Some } z \wedge x = f z.$

Proof using. *intros H. destruct o ; simpl; inverts $\times H$. Qed.*

Arguments map_eq_none_inv $[A] [B] [f] [o]$.

Arguments map_on_eq_some_inv $[A] [B] [f] [o] [x]$.

17.2.6 apply

apply f o optionnaly applies a function of type $A \rightarrow \text{option } B$

Definition apply $A B$ ($f : A \rightarrow \text{option } B$) ($o : \text{option } A$) : $\text{option } B :=$
match o with
| None $\Rightarrow$ None
| Some $x \Rightarrow f x$
end.

17.2.7 apply_on

Definition apply_on $A B$ ($o : \text{option } A$) ($f : A \rightarrow \text{option } B$) : $\text{option } B :=$
apply f o .

Lemma apply_on_eq_none_inv : $\forall A B (f : A \rightarrow \text{option } B) o,$
apply_on $o f = \text{None} \rightarrow$
($o = \text{None}$)
 $\vee (\exists x, o = \text{Some } x \wedge f x = \text{None}).$

Proof using. *intros H. destruct o ; simpl; inverts $\times H$. Qed.*

Lemma apply_on_eq_some_inv : $\forall A B (f : A \rightarrow \text{option } B) o x,$
apply_on $o f = \text{Some } x \rightarrow$
 $\exists z, o = \text{Some } z \wedge f z = \text{Some } x.$

Proof using. *intros H. destruct o ; simpl; inverts $\times H$. Qed.*

Arguments apply_on_eq_none_inv $[A] [B] [f] [o]$.

Arguments apply_on_eq_some_inv $[A] [B] [f] [o] [x]$.

Chapter 18

Library **SLF.LibWf**

```
Set Implicit Arguments.
From SLF Require Import LibTactics LibLogic
      LibProd LibSum LibRelation LibNat LibInt.
```

18.1 Compatibility

Rocq's stdlib Prelude defines:

Inductive Acc A (R:A->A->Prop) (x:A) : Prop := | Acc_intro : (forall (y:A), R y x -> Acc y) -> Acc x.

Definition well_founded A (R:A->A->Prop) := forall (x:A), Acc x.

TLC introduces **wf** as a shorter name for **well_founded**, both for conciseness and for tactics to specifically recognize this symbol.

Definition **wf** := well_founded.

18.1.1 Tactics

auto with wf attempts to unfold the names of the relations given as argument to **wf**.

#[global]

Hint Extern 1 (**wf** ?*R*) => progress (unfold *R*) : **wf**.

solve_wf is a shorthand for solving goals using **auto with wf**, aimed to prove goals of the form **wf** *R*.

Tactic Notation "solve_wf" :=
 solve [auto with wf].

18.1.2 Definition

measure *f* is a well-founded binary relation which relates *x* to *y* when *f* *x* < *f* *y*, at type **nat**.

```

Definition measure A (f:A→nat) : binary A :=
  fun x1 x2 => (f x1 < f x2).

```

18.1.3 Properties

Section Measure.

Variables (A : Type).

Implicit Type f : A → nat.

```

Lemma wf_measure : ∀ f,
  wf (measure f).

```

Proof using.

```

  intros f a. gen_eq n: (f a). gen a. pattern n.
  apply peano_induction. clear n. introv IH Eq.
  apply Acc_intro. introv H. unfolds in H.
  rewrite ← Eq in H. apply× IH.

```

Qed.

```

Lemma trans_measure : ∀ (f : A → nat),
  trans (measure f).

```

Proof using. intros. unfold measure, trans. intros. nat_math. Qed.

End Measure.

#[global]

Hint Resolve wf_measure : wf.

18.1.4 Measure on pairs

```

Definition measure2 A1 A2 (f : A1 → A2 → nat) : binary (A1 × A2) :=
  fun p1 p2 => let (x1,y1) := p1 in
    let (x2,y2) := p2 in
    (f x1 y1 < f x2 y2).

```

```

Lemma wf_measure2 : ∀ A1 A2 (f:A1→A2→nat),
  wf (measure2 f).

```

Proof using.

```

  intros A1 A2 f [x1 x2]. apply (@measure_induction _ (uncurry2 f)). clear x1 x2.
  intros [x1 x2] H. apply Acc_intro. intros [y1 y2] Lt. apply¬ H.

```

Qed.

#[global]

Hint Resolve wf_measure2 : wf.

```

Ltac induction_wf_process_wf_hyp tt ::=
  match goal with

```



```

| ⊢ wf _ ⇒ auto with wf
| ⊢ well_founded _ ⇒ change well_founded with wf; auto with wf
end.

Ltac induction_wf_process_measure E ::=
  applys well_founded_ind (wf_measure E).

```

18.2 Construction of well-founded relations

18.2.1 Empty relation

```

Lemma wf_empty : ∀ A,
  wf (@empty A).
Proof using. intros_all. constructor. introv H. false. Qed.

#[global]
Hint Resolve wf_empty : wf.

```

18.2.2 Inclusion

Well-foundedness preserved by inclusion

```

Lemma wf_of_rel_incl : ∀ A (R1 R2 : binary A),
  wf R1 →
  rel_incl R2 R1 →
  wf R2.
Proof using.
  introv W1 Inc. intros x.
  pattern x. apply (well_founded_ind W1). clear x.
  intros x IH. constructor. intros. apply IH. apply¬ Inc.
Qed.

```

18.2.3 *Peano.lt* on **nat**

The relation “less than” on natural numbers is well-founded.

```

Lemma wf_peano_lt : wf Peano.lt.
Proof using.
  intros x.
  induction x using peano_induction. apply¬ Acc_intro.
  intros. applys H. math.
Qed.

#[global]
Hint Resolve wf_peano_lt : wf.

```

18.2.4 lt on nat

The relation “less than” on natural numbers is well_founded.

Lemma wf_lt : @wf nat lt.

Proof using.

intros x.

induction x using peano_induction. apply Acc_intro.

Qed.

#[global]

Hint Resolve wf_lt : wf.

18.2.5 The relation “greater than” on the set of

natural number lower than a fixed upper bound.

Definition nat_upto (b:nat) :=

fun (n m:nat) => (n ≤ b)%nat ∧ (m < n)%nat.

Lemma nat_upto_eq : ∀ (b n m:nat),

nat_upto b n m = ((n ≤ b)%nat ∧ (m < n)%nat).

Proof using. auto. Qed.

Lemma wf_nat_upto : ∀ (b:nat),

wf (nat_upto b).

Proof using.

intros b n.

induction_wf IH: (wf_measure (fun n => (b-n)%nat)) n.

apply Acc_intro. introv [H1 H2]. apply IH.

hnf. nat_math.

Qed.

18.2.6 The relation “less than” on the set of

integers greater than a fixed lower bound.

Definition downto (b:Z) :=

fun (n m:Z) => (b ≤ n) ∧ (n < m).

Lemma downto_eq : ∀ (b n m:Z),

downto b n m = (b ≤ n ∧ n < m).

Proof using. auto. Qed.

Lemma downto_intro : ∀ (b n m:Z),

b ≤ n →

n < m →

downto b n m.

```

Proof using. split $\neg$ . Qed.
Lemma wf_downto :  $\forall (b:\mathbb{Z})$ ,
  wf (downto b).
Proof using.
  intros b n.
  induction_wf IH: (wf_measure (fun n  $\Rightarrow$  Z.abs_nat (n-b))) n.
  apply Acc_intro. introv [H1 H2]. apply IH.
  unfolds. applies lt_abs_abs; math.
Qed.
#[global]
Hint Resolve wf_downto : wf.
#[global]
Hint Unfold downto.
#[global]
Hint Extern 1 (downto _ _ )  $\Rightarrow$  math : maths.

```

18.2.7 The relation “greater than” on the set of

integers lower than a fixed upper bound.

```

Definition upto (b: $\mathbb{Z}$ ) :=
  fun (n m: $\mathbb{Z}$ )  $\Rightarrow$  (n  $\leq$  b)  $\wedge$  (m < n).
Lemma upto_eq :  $\forall b n m$ ,
  upto b n m = ((n  $\leq$  b)  $\wedge$  (m < n)).
Proof using. auto. Qed.
Lemma upto_intro :  $\forall b n m$ ,
  n  $\leq$  b  $\rightarrow$ 
  m < n  $\rightarrow$ 
  upto b n m.
Proof using. split $\neg$ . Qed.
Lemma wf_upto :  $\forall n$ ,
  wf (upto n).
Proof using.
  intros b n.
  induction_wf IH: (wf_measure (fun n  $\Rightarrow$  Z.abs_nat (b-n))) n.
  apply Acc_intro. introv [H1 H2]. apply IH.
  applies lt_abs_abs; math.
Qed.
#[global]
Hint Resolve wf_upto : wf.
#[global]
Hint Unfold upto.

```

```
#[global]
Hint Extern 1 (upto _ _ ) => math : maths.
```

18.3 Inverse projections

Section UnprojWf.

Variables (*A1 A2 A3 A4 A5* : Type).

Lemma wf_unproj21 : $\forall (R:\text{binary } A1),$
 wf *R* $\rightarrow$
 wf (unproj21 *A2* *R*).

Proof using.

```
intros R H [x1 x2]. gen x2.
induction_wf IH: H x1. constructor. intros [y1 y2]. auto.
```

Qed.

Lemma wf_unproj22 : $\forall (R:\text{binary } A2),$
 wf *R* $\rightarrow$
 wf (unproj22 *A1* *R*).

Proof using.

```
intros R H [x1 x2]. gen x1.
induction_wf IH: H x2. constructor. intros [y1 y2]. auto.
```

Qed.

Lemma wf_unproj31 : $\forall (R:\text{binary } A1),$
 wf *R* $\rightarrow$
 wf (unproj31 *A2* *A3* *R*).

Proof using.

```
intros R H [[x1 x2] x3]. gen x2 x3.
induction_wf IH: H x1. constructor. intros [[y1 y2] y3]. auto.
```

Qed.

Lemma wf_unproj32 : $\forall (R:\text{binary } A2),$
 wf *R* $\rightarrow$
 wf (unproj32 *A1* *A3* *R*).

Proof using.

```
intros R H [[x1 x2] x3]. gen x1 x3.
induction_wf IH: H x2. constructor. intros [[y1 y2] y3]. auto.
```

Qed.

Lemma wf_unproj33 : $\forall (R:\text{binary } A3),$
 wf *R* $\rightarrow$
 wf (unproj33 *A1* *A2* *R*).

Proof using.

```
intros R H [[x1 x2] x3]. gen x1 x2.
```

```

    induction_wf IH: H x3. constructor. intros [[y1 y2] y3]. auto.
Qed.
Lemma wf_unproj41 : ∀ (R:binary A1),
  wf R →
  wf (unproj41 A2 A3 A4 R).
Proof using.
  intros R H [[[x1 x2] x3] x4]. gen x2 x3 x4.
  induction_wf IH: H x1. constructor. intros [[[y1 y2] y3] y4]. auto.
Qed.
Lemma wf_unproj42 : ∀ (R:binary A2),
  wf R →
  wf (unproj42 A1 A3 A4 R).
Proof using.
  intros R H [[[x1 x2] x3] x4]. gen x1 x3 x4.
  induction_wf IH: H x2. constructor. intros [[[y1 y2] y3] y4]. auto.
Qed.
Lemma wf_unproj43 : ∀ (R:binary A3),
  wf R →
  wf (unproj43 A1 A2 A4 R).
Proof using.
  intros R H [[[x1 x2] x3] x4]. gen x1 x2 x4.
  induction_wf IH: H x3. constructor. intros [[[y1 y2] y3] y4]. auto.
Qed.
Lemma wf_unproj44 : ∀ (R:binary A4),
  wf R →
  wf (unproj44 A1 A2 A3 R).
Proof using.
  intros R H [[[x1 x2] x3] x4]. gen x1 x2 x3.
  induction_wf IH: H x4. constructor. intros [[[y1 y2] y3] y4]. auto.
Qed.
Lemma wf_unproj51 : ∀ (R:binary A1),
  wf R →
  wf (unproj51 A2 A3 A4 A5 R).
Proof using.
  intros R H [[[[x1 x2] x3] x4] x5]. gen x2 x3 x4 x5.
  induction_wf IH: H x1. constructor. intros [[[[y1 y2] y3] y4] y5]. auto.
Qed.
End UnprojWf.
#[global]
Hint Resolve
  wf_unproj21 wf_unproj22

```

```

wf_unproj31 wf_unproj32 wf_unproj33
wf_unproj41 wf_unproj42 wf_unproj43 wf_unproj44
wf_unproj51 : wf.

```

18.4 Lexicographical product

```

Lemma wf_lexico2 : ∀ A1 A2
  (R1:binary A1) (R2:binary A2),
  wf R1 →
  wf R2 →
  wf (lexico2 R1 R2).

```

Proof using.

```

  introv W1 W2. intros [x1 x2]. gen x2.
  induction_wf IH1: W1 x1. intros.
  induction_wf IH2: W2 x2. constructor. intros [y1 y2] H.
  simpls. destruct H as [H1|[H1 H2]].
  apply¬ IH1. rewrite H1. apply¬ IH2.

```

Qed.

```

Lemma wf_lexico3 : ∀ A1 A2 A3
  (R1:binary A1) (R2:binary A2) (R3:binary A3),
  wf R1 →
  wf R2 →
  wf R3 →
  wf (lexico3 R1 R2 R3).

```

Proof using.

```

  intros. apply¬ wf_lexico2. apply¬ wf_lexico2.

```

Qed.

```

Lemma wf_lexico4 : ∀ A1 A2 A3 A4
  (R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
  wf R1 →
  wf R2 →
  wf R3 →
  wf R4 →
  wf (lexico4 R1 R2 R3 R4).

```

Proof using.

```

  intros. apply¬ wf_lexico3. apply¬ wf_lexico2.

```

Qed.

#[global]

Hint Resolve wf_lexico2 wf_lexico3 wf_lexico4 : wf.

18.5 Symmetric product

Lemma wf_prod2_of_wf_1 : $\forall (A1\ A2:\text{Type})$

($R1:\text{binary } A1$) ($R2:\text{binary } A2$),
wf $R1 \rightarrow$
wf (prod2 $R1\ R2$).

Proof using.

introv $W1$. intros [$x1\ x2$].
gen $x2$. induction_wf IH : $W1\ x1$. intros.
constructor. intros [$y1\ y2$] [$E1\ E2$]. apply $\neg IH$.

Qed.

Lemma wf_prod2_of_wf_2 : $\forall (A1\ A2:\text{Type})$

($R1:\text{binary } A1$) ($R2:\text{binary } A2$),
wf $R2 \rightarrow$
wf (prod2 $R1\ R2$).

Proof using.

introv $W2$. intros [$x1\ x2$].
gen $x1$. induction_wf IH : $W2\ x2$. intros.
constructor. intros [$y1\ y2$] [$E1\ E2$]. apply $\neg IH$.

Qed.

Lemma wf_prod3_of_wf_1 : $\forall (A1\ A2\ A3:\text{Type})$

($R1:\text{binary } A1$) ($R2:\text{binary } A2$) ($R3:\text{binary } A3$),
wf $R1 \rightarrow$
wf (prod3 $R1\ R2\ R3$).

Proof using. intros. apply wf_prod2_of_wf_1. apply $\neg$ wf_prod2_of_wf_1. Qed.

Lemma wf_prod3_of_wf_2 : $\forall (A1\ A2\ A3:\text{Type})$

($R1:\text{binary } A1$) ($R2:\text{binary } A2$) ($R3:\text{binary } A3$),
wf $R2 \rightarrow$
wf (prod3 $R1\ R2\ R3$).

Proof using. intros. apply wf_prod2_of_wf_1. apply $\neg$ wf_prod2_of_wf_2. Qed.

Lemma wf_prod3_of_wf_3 : $\forall (A1\ A2\ A3:\text{Type})$

($R1:\text{binary } A1$) ($R2:\text{binary } A2$) ($R3:\text{binary } A3$),
wf $R3 \rightarrow$
wf (prod3 $R1\ R2\ R3$).

Proof using. intros. apply $\neg$ wf_prod2_of_wf_2. Qed.

Lemma wf_prod4_of_wf_1 : $\forall (A1\ A2\ A3\ A4:\text{Type})$

($R1:\text{binary } A1$) ($R2:\text{binary } A2$) ($R3:\text{binary } A3$) ($R4:\text{binary } A4$),
wf $R1 \rightarrow$
wf (prod4 $R1\ R2\ R3\ R4$).

Proof using. intros. apply wf_prod2_of_wf_1. apply $\neg$ wf_prod3_of_wf_1. Qed.

Lemma wf_prod4_of_wf_2 : $\forall (A1\ A2\ A3\ A4:\text{Type})$

```

(R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
  wf R2 →
  wf (prod4 R1 R2 R3 R4).
Proof using. intros. apply wf_prod2_of_wf_1. apply¬ wf_prod3_of_wf_2. Qed.

Lemma wf_prod4_of_wf_3 : ∀ (A1 A2 A3 A4:Type)
  (R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
  wf R3 →
  wf (prod4 R1 R2 R3 R4).
Proof using. intros. apply wf_prod2_of_wf_1. apply¬ wf_prod3_of_wf_3. Qed.

Lemma wf_prod4_of_wf_4 : ∀ (A1 A2 A3 A4:Type)
  (R1:binary A1) (R2:binary A2) (R3:binary A3) (R4:binary A4),
  wf R4 →
  wf (prod4 R1 R2 R3 R4).
Proof using. intros. apply¬ wf_prod2_of_wf_2. Qed.

#[global]
Hint Resolve
  wf_prod2_of_wf_1 wf_prod2_of_wf_2
  wf_prod3_of_wf_1 wf_prod3_of_wf_2 wf_prod3_of_wf_3
  wf_prod4_of_wf_1 wf_prod4_of_wf_2 wf_prod4_of_wf_3 wf_prod4_of_wf_4 : wf.

```

18.6 Well-foundedness of a function image

```

Lemma wf_rel_preimage : ∀ A B (R:binary B) (f:A→B),
  wf R →
  wf (rel_preimage R f).
Proof using.
  introv W. intros x. gen_eq a: (f x). gen x.
  induction_wf IH: W a. introv E. constructors.
  intros y Hy. subst a. hnf in Hy. applies× IH.
Qed.

#[global]
Hint Resolve wf_rel_preimage : wf.

```

18.7 Transitive closure

```

Lemma wf_tclosure : ∀ A (R:binary A),
  wf R →
  wf (tclosure R).
Proof using.

```



```

unfold wf, well_founded.
introv HAcc. intro a. specializes HAcc a. generalize dependent a.
induction 1 as [ a - IH ].
constructor. intros b Hba.
generalize a b Hba IH. clear a b Hba IH.
induction 1; eauto using Acc_inv.
Qed.

```

Chapter 19

Library SLF.LibList

```
Set Implicit Arguments.
Require Import Stdlib.Classes.Morphisms. From SLF Require Import LibTactics LibLogic
LibReflect LibOperation
  LibProd LibOption LibNat LibInt LibWf LibMonoid LibRelation.
Generalizable Variables A B.
Local Open Scope nat_scope.
Local Open Scope comp_scope.
Global Close Scope list_scope.

Arguments nil {A}.
Arguments cons {A}.

Declare Scope liblist_scope.
Open Scope liblist_scope.
Delimit Scope liblist_scope with mylist.
Bind Scope liblist_scope with list.

Infix "::<" := cons (at level 60, right associativity) : liblist_scope.

Module LIBLISTNOTATION.
Notation "[ ]" := nil (format "[ ]") : liblist_scope.
Notation "[ x ]" := (cons x nil) : liblist_scope.
Notation "[ x ; y ; .. ; z ]" := (cons x (cons y .. (cons z nil) ..)) : liblist_scope.
End LIBLISTNOTATION.
```

19.1 Inhabited

```
#[global]
Instance Inhab_list :  $\forall A$ , Inhab (list A).
Proof using. intros. apply (Inhab_of_val nil). Qed.
```

19.2 Normalization tactics

19.2.1 *rew_list* for basic list properties

Normalize

- ++
- length
- rev

```
Tactic Notation "rew_list" :=
  autorewrite with rew_list.
Tactic Notation "rew_list" "~" :=
  rew_list; auto_tilde.
Tactic Notation "rew_list" "*" :=
  rew_list; auto_star.
Tactic Notation "rew_list" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_list).
Tactic Notation "rew_list" "~" "in" "*" :=
  rew_list in *; auto_tilde.
Tactic Notation "rew_list" "*" "in" "*" :=
  rew_list in *; auto_star.
Tactic Notation "rew_list" "in" hyp(H) :=
  autorewrite with rew_list in H.
Tactic Notation "rew_list" "~" "in" hyp(H) :=
  rew_list in H; auto_tilde.
Tactic Notation "rew_list" "*" "in" hyp(H) :=
  rew_list in H; auto_star.
```

19.2.2 *rew_listx* for all other operations on lists

```
Tactic Notation "rew_listx" :=
  autorewrite with rew_listx.
Tactic Notation "rew_listx" "~" :=
  rew_listx; auto_tilde.
Tactic Notation "rew_listx" "*" :=
  rew_listx; auto_star.
Tactic Notation "rew_listx" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_listx).
Tactic Notation "rew_listx" "~" "in" "*" :=
  rew_listx in *; auto_tilde.
```

```

Tactic Notation "rew_listx" "*" "in" "*" :=
  rew_listx in *; auto_star.
Tactic Notation "rew_listx" "in" hyp(H) :=
  autorewrite with rew_listx in H.
Tactic Notation "rew_listx" "~" "in" hyp(H) :=
  rew_listx in H; auto_tilde.
Tactic Notation "rew_listx" "*" "in" hyp(H) :=
  rew_listx in H; auto_star.

```

19.3 Properties of operations

19.3.1 Definitions of Fold-right and App

```

Fixpoint fold_right A B (f:A→B→B) (i:B) (l:list A) :=
  match l with
  | nil ⇒ i
  | x::L' ⇒ f x (fold_right f i L')
  end.

```

```

Definition app A (l1 l2 : list A) :=
  fold_right (fun x (acc:list A) ⇒ x::acc) l2 l1.

```

`l1 ++ l2` concatenates two lists

```

Infix "++" := app (right associativity, at level 60) : liblist_scope.

```

`l & x` extends the list `l` with the value `x` at the right end

```

Notation "l & x" := (l ++ (x::nil))
  (at level 28, left associativity) : liblist_scope.

```

19.3.2 App

Section App.

Variables $A B$: Type.

Implicit Types x : A .

Implicit Types l : list A .

```

Lemma app_cons_l : ∀ x l1 l2,
  (x::l1) ++ l2 = x :: (l1++l2).

```

Proof using. auto. Qed.

```

Lemma app_nil_l : ∀ l,
  nil ++ l = l.

```

Proof using. auto. Qed.

```

Lemma app_nil_r :  $\forall l$ ,
   $l ++ \text{nil} = l$ .
Proof using. induction  $l$ . auto. rewrite app_cons_l. fequals $\neg$ . Qed.

Lemma app_assoc :  $\forall l1\ l2\ l3$ ,
   $(l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3)$ .
Proof using.
  intros. induction  $l1$ .
  { rewrite_all $\neg$  app_nil_l. }
  { rewrite_all $\neg$  app_cons_l. fequals $\neg$ . }
Qed.

Lemma app_cons_r :  $\forall x\ l1\ l2$ ,
   $l1 ++ (x :: l2) = (l1 \& x) ++ l2$ .
Proof using. intros. rewrite $\neg$  app_assoc. Qed.

Lemma app_cons_one_r :  $\forall x\ l$ ,
   $(x :: \text{nil}) ++ l = x :: l$ .
Proof using. auto. Qed.

Lemma app_cons_one_l :  $\forall x\ l$ ,
   $l ++ (x :: \text{nil}) = l \& x$ . Proof using. auto. Qed.

Lemma app_last_l :  $\forall x\ l1\ l2$ ,
   $(l1 \& x) ++ l2 = l1 ++ (x :: l2)$ .
Proof using. intros. rewrite $\neg$   $\leftarrow$  app_cons_r. Qed.

Lemma app_last_r :  $\forall x\ l1\ l2$ ,
   $l1 ++ (l2 \& x) = (l1 ++ l2) \& x$ .
Proof using. intros. rewrite $\neg$   $\leftarrow$  app_assoc. Qed.

Lemma last_nil :  $\forall x$ ,
   $\text{nil} \& x = x :: \text{nil}$ .
Proof using. auto. Qed.

Lemma last_cons :  $\forall x\ y\ l$ ,
   $(x :: l) \& y = x :: (l \& y)$ .
Proof using. auto. Qed.

Lemma last_one :  $\forall x\ y$ ,
   $(x :: \text{nil}) \& y = x :: y :: \text{nil}$ .
Proof using. auto. Qed.

Lemma last_app :  $\forall x\ l1\ l2$ ,
   $(l1 ++ l2) \& x = l1 ++ (l2 \& x)$ .
Proof using. intros. rewrite $\neg$  app_last_r. Qed.

End App.

Global Opaque app.
#[global]

```

Hint Rewrite app_cons_l app_nil_l app_nil_r app_assoc
 app_cons_one_r : *rew_list*.

#[*global*]

Hint Rewrite app_cons_l app_nil_l app_nil_r app_assoc
 app_cons_one_r : *rew_listx*.

#[*global*]

Hint Rewrite app_cons_l app_nil_l app_nil_r app_assoc
 app_cons_one_r : *rew_lists*.

19.3.3 FoldRight

Section FoldRight.

Variables $A\ B$: Type.

Implicit Types x : A .

Implicit Types l : **list** A .

Implicit Types $(f : A \rightarrow B \rightarrow B)$ $(i : B)$.

Lemma fold_right_nil : $\forall f\ i$,
 fold_right $f\ i$ nil = i .

Proof using. auto. Qed.

Lemma fold_right_cons : $\forall f\ i\ x\ l$,
 fold_right $f\ i$ $(x :: l)$ = $f\ x$ (fold_right $f\ i\ l$).

Proof using. auto. Qed.

Lemma fold_right_app : $\forall f\ i\ l1\ l2$,
 fold_right $f\ i$ $(l1 ++ l2)$ = fold_right f (fold_right $f\ i\ l2$) $l1$.

Proof using.

intros. induction $\neg l1$.

{ *rew_list*. do 2 rewrite fold_right_cons. *fequals* $\neg$. }

Qed.

Lemma fold_right_last : $\forall f\ i\ x\ l$,
 fold_right $f\ i$ $(l \& x)$ = fold_right f $(f\ x\ i)$ l .

Proof using. intros. rewrite $\neg$ fold_right_app. Qed.

End FoldRight.

Global Opaque fold_right.

#[*global*]

Hint Rewrite fold_right_nil fold_right_cons fold_right_last : *rew_listx*.

19.3.4 FoldLeft

Fixpoint fold_left $A\ B$ $(f : A \rightarrow B \rightarrow B)$ $(i : B)$ $(l : \text{list } A)$: $B :=$

```

match  $l$  with
| nil  $\Rightarrow i$ 
|  $x :: L' \Rightarrow \text{fold\_left } f (f\ x\ i)\ L'$ 
end.

```

Section FoldLeft.

Variables $A\ B : \text{Type}$.

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

Implicit Types $(f : A \rightarrow B \rightarrow B)\ (i : B)$.

Lemma fold_left_nil : $\forall f\ i$,

fold_left $f\ i\ \text{nil} = i$.

Proof using. auto. Qed.

Lemma fold_left_cons : $\forall f\ i\ x\ l$,

fold_left $f\ i\ (x :: l) = \text{fold\_left } f (f\ x\ i)\ l$.

Proof using. auto. Qed.

Lemma fold_left_app : $\forall f\ i\ l1\ l2$,

fold_left $f\ i\ (l1 ++ l2) = \text{fold\_left } f (\text{fold\_left } f\ i\ l1)\ l2$.

Proof using.

intros. *gen* i . induction $l1$.

{ intros. *rew_list*. do 2 rewrite fold_left_cons. rewrite $\neg IHl1$. }

Qed.

Lemma fold_left_last : $\forall f\ i\ x\ l$,

fold_left $f\ i\ (l \& x) = f\ x (\text{fold\_left } f\ i\ l)$.

Proof using. intros. rewrite $\neg$ fold_left_app. Qed.

End FoldLeft.

Global Opaque fold_left.

#[*global*]

Hint Rewrite fold_left_nil fold_left_cons

fold_left_last : *rew_list*.

19.3.5 Length

Definition length $A\ (l : \text{list } A) : \text{nat} :=$

fold_right (fun $x\ acc \Rightarrow 1 + acc$) 0 l .

Section Length.

Variables ($A : \text{Type}$).

Implicit Types $l : \text{list } A$.

Lemma length_nil :

length (@nil A) = 0.

```

Proof using. auto. Qed.
Lemma length_cons :  $\forall x\ l,$ 
  length  $(x :: l) = 1 + \text{length } l$ .
Proof using. auto. Qed.
Lemma length_app :  $\forall l1\ l2,$ 
  length  $(l1 ++ l2) = \text{length } l1 + \text{length } l2$ .
Proof using.
  intros. unfold length at 1. rewrite fold_right_app.
  fold (length  $l2$ ). induction  $l1$ ; rew_listx.
  { auto. }
  { rewrite length_cons. rewrite IHL1.math. }
Qed.
Lemma length_last :  $\forall x\ l,$ 
  length  $(l \& x) = 1 + \text{length } l$ .
Proof using.
  intros. rewrite length_app, length_cons, length_nil.
  simpl. math.
Qed.
End Length.
Global Opaque length.
#[global]
Hint Rewrite length_nil length_cons length_app
  length_last : rew_list.
#[global]
Hint Rewrite length_nil length_cons length_app
  length_last : rew_listx.

```

19.4 Inversion lemmas for structural composition

```

Section AppInversion.
Variables  $A$  : Type.
Implicit Types  $x$  :  $A$ .
Implicit Types  $l$  : list  $A$ .

Lemma length_zero_inv :  $\forall l,$ 
  length  $l = 0\%nat \rightarrow$ 
   $l = \text{nil}$ .
Proof using.
  intros  $l$ . destruct  $l$ ; rew_list; introv E. { auto. } { false. }
Qed.

```


Lemma `length_zero_eq_eq_nil` : $\forall l$,
 $(\text{length } l = 0) = (l = \text{nil})$.
 Proof using.
`extens. iff M. destruct l; rew_list; auto_false×. { subst×. }`
 Qed.

Lemma `length_one_inv` : $\forall l$,
 $\text{length } l = 1 \rightarrow$
 $\exists x, l = x :: \text{nil}$.
 Proof using.
`introv N. destruct l as [|x []]; rew_list in *; tryfalse. eauto.`
 Qed.

Lemma `length_neq_inv` : $\forall l1\ l2$,
 $\text{length } l1 \neq \text{length } l2 \rightarrow$
 $(l1 \neq l2)$.
 Proof using. `introv N E. subst×. Qed.`

Lemma `length_pos_inv_cons` : $\forall l$,
 $(\text{length } l > 0\%nat) \rightarrow$
 $\exists x\ l', l = x :: l'$.
 Proof using.
`introv E. destruct l; rew_list in *.`
`{ false. math. }`
`{ eauto. }`
 Qed.

Lemma `length_pos_inv_last` : $\forall l$,
 $(\text{length } l > 0\%nat) \rightarrow$
 $\exists x\ l', l = l' \& x$.
 Proof using.
`intros l. induction l; rew_list; introv H.`
`{ false. math. }`
`{ destruct l.`
`{  $\exists \neg a (@\text{nil } A)$ . }`
`{ destruct IHl as (x&l'&E).`
`{ rew_list in *. math. }`
`{  $\exists x (a :: l')$ . rewrite $\neg$  E. } } }`
 Qed.

Lemma `list_same_length_inv_nil` : $\forall l1\ l2$,
 $\text{length } l1 = \text{length } l2 \rightarrow$
 $l1 = \text{nil} \leftrightarrow l2 = \text{nil}$.
 Proof using. `intros. destruct l1; destruct l2; auto_false×. Qed.`

Lemma `cons_case` : $\forall l$,
 $l = \text{nil} \vee \exists x\ l', l = x :: l'$.

Proof using. `intros. destruct× l. Qed.`

Lemma `cons_eq_nil_inv` : $\forall x\ l,$
 $x :: l = \text{nil} \rightarrow$
False.

Proof using. `auto_false. Qed.`

Lemma `nil_eq_cons_inv` : $\forall x\ l,$
 $\text{nil} = x :: l \rightarrow$
False.

Proof using. `auto_false. Qed.`

Lemma `list_neq_nil_inv_cons` : $\forall l,$
 $l \neq \text{nil} \rightarrow$
 $\exists x\ q, l = x :: q.$

Proof using. `introv N. destruct× l. Qed.`

Lemma `cons_eq_cons_inv` : $\forall x1\ x2\ l1\ l2,$
 $x1 :: l1 = x2 :: l2 \rightarrow$
 $x1 = x2 \wedge l1 = l2.$

Proof using. `introv H. inverts× H. Qed.`

Lemma `app_eq_nil_inv` : $\forall l1\ l2,$
 $l1 ++ l2 = \text{nil} \rightarrow$
 $l1 = \text{nil} \wedge l2 = \text{nil}.$

Proof using. `intros. destruct l1; destruct l2; intros; tryfalse~; auto. Qed.`

Lemma `nil_eq_app_inv` : $\forall l1\ l2,$
 $\text{nil} = l1 ++ l2 \rightarrow$
 $l1 = \text{nil} \wedge l2 = \text{nil}.$

Proof using. `intros. symmetry in H. apply× app_eq_nil_inv. Qed.`

Lemma `app_not_empty_l` : $\forall l1\ l2,$
 $l1 \neq \text{nil} \rightarrow$
 $l1 ++ l2 \neq \text{nil}.$

Proof using. `introv NE K. apply NE. forwards*: app_eq_nil_inv K. Qed.`

Lemma `app_not_empty_r` : $\forall l1\ l2,$
 $l2 \neq \text{nil} \rightarrow$
 $l1 ++ l2 \neq \text{nil}.$

Proof using. `introv NE K. apply NE. forwards*: app_eq_nil_inv K. Qed.`

Lemma `app_l_eq_self_inv` : $\forall l1\ l2,$
 $l1 ++ l2 = l1 \rightarrow$
 $l2 = \text{nil}.$

Proof using.
`introv E. apply length_zero_inv.`
`lets: (args_eq_1 (@length A) E). rew_list in H. math.`
`Qed.`

Lemma self_eq_app_l_inv : $\forall l1\ l2,$
 $l1 = l1 ++ l2 \rightarrow$
 $l2 = \text{nil}.$

Proof using. intros. *applys*× app_l_eq_self_inv. Qed.

Lemma app_r_eq_self_inv : $\forall l1\ l2,$
 $l1 ++ l2 = l2 \rightarrow$
 $l1 = \text{nil}.$

Proof using.

intros E. apply length_zero_inv.
lets: (args_eq_1 (@length A) E). rew_list in H. math.

Qed.

Lemma self_eq_app_r_inv : $\forall l1\ l2,$
 $l2 = l1 ++ l2 \rightarrow$
 $l1 = \text{nil}.$

Proof using. intros. *applys*× app_r_eq_self_inv. Qed.

Lemma app_cancel_same_length_l : $\forall l1\ l2\ t1\ t2,$
 $l1 ++ l2 = t1 ++ t2 \rightarrow$
 $\text{length } l1 = \text{length } t1 \rightarrow$
 $l1 = t1 \wedge l2 = t2.$

Proof using.

*intros l1. induction l1; introv He Hl; rew_list in *.*
{ forwards× ->: length_zero_inv t1. }
*{ destruct t1; rew_list in *; try math.*
inverts He. forwards× (-&->): IHl1 H1. }

Qed.

Lemma app_cancel_same_length_r : $\forall l1\ l2\ t1\ t2,$
 $l1 ++ l2 = t1 ++ t2 \rightarrow$
 $\text{length } l2 = \text{length } t2 \rightarrow$
 $l1 = t1 \wedge l2 = t2.$

Proof using.

*intros l1. induction l1; introv He Hl; rew_list in *.*
{ gen l2. destruct t1; rew_list; intros.
{ autos×. }
{ destruct l2 as [|x l2']; inverts He. rew_list in Hl. math. } }
*{ destruct t1; rew_list in *.*
{ subst t2; rew_list in; try math. }*
{ inverts He. forwards× (-&->): IHl1 H1. } }

Qed.

Lemma app_cancel_l : $\forall l1\ l2\ l3,$
 $l1 ++ l2 = l1 ++ l3 \rightarrow$

$l2 = l3$.

Proof using.

intros E . *induction* $l1$; *rew_list* in *. *auto*. *inverts* $\times E$.

Qed.

Lemma *app_cancel_r* : $\forall l1\ l2\ l3$,

$l1 ++ l3 = l2 ++ l3 \rightarrow$

$l1 = l2$.

Proof using.

intros $l1$. *induction* $l1$; *intros* E ; *rew_list* in *.

{ *rewrites* $\neg (\gg \text{self\_eq\_app\_r\_inv } E)$. }

{ *destruct* $l2$; *rew_list* in *.

{ *rewrite* $\leftarrow \text{app\_cons\_l in } E$. *rewrites* $\neg (\gg \text{self\_eq\_app\_r\_inv } (\text{eq\_sym } E))$. }

{ *inverts* E . *fequals*. *applies* $\times \text{IH}l1$. } }

Qed.

Lemma *last_case* : $\forall l$,

$l = \text{nil} \vee \exists x\ l',\ l = l' \ \&\ x$.

Proof using.

intros. *destruct* l . { *left* $\times$. }

{ *right*. *forwards* $\times (x \& l' \& H)$: (*length_pos_inv_last* ($a :: l$)).

rew_list. *math*. }

Qed.

Lemma *last_eq_nil_inv* : $\forall a\ l$,

$l \ \&\ a = \text{nil} \rightarrow$

False.

Proof using. *intros* E . *induction* l ; *rew_list*; *false*. Qed.

Lemma *nil_eq_last_inv* : $\forall a\ l$,

$\text{nil} = l \ \&\ a \rightarrow$

False.

Proof using. *intros*. *apply* $\times \text{last\_eq\_nil\_inv}$. Qed.

Lemma *list_neq_nil_inv_last* : $\forall l$,

$l \neq \text{nil} \rightarrow$

$\exists x\ q,\ l = q \ \&\ x$.

Proof using. *intros* N . *destruct* $\times (\text{@last\_case } l)$. Qed.

Lemma *last_eq_last_inv* : $\forall x1\ x2\ l1\ l2$,

$l1 \ \&\ x1 = l2 \ \&\ x2 \rightarrow$

$l1 = l2 \wedge x1 = x2$.

Proof using.

intros H . *gen* $l2$. *induction* $l1$; *intros* E ; *rew_list* in E .

{ *destruct* $l2$; *rew_list* in E ; *inverts* E as E .

{ *auto*. } { *false* *nil_eq_last_inv* E . } }

{ *destruct* $l2$; *rew_list* in E .

```

{ inverts E as E. false last_eq_nil_inv E. }
{ inverts E. forwards × [? ?]: IHL1.
  split; congruence. } }

```

Qed.

```

Lemma nil_eq_middle_inv : ∀ x l1 l2,
  nil = l1 & x ++ l2 →

```

False.

Proof using. intros. destruct *l1*; *inverts* *H*. Qed.

```

Lemma cons_eq_middle_inv : ∀ x y l1 l2 l,
  x :: l = l1 & y ++ l2 →
  (l1 = nil ∧ x = y ∧ l = l2) ∨ (∃ l1', l1 = x::l1').

```

Proof using.

```

  intros. destruct l1; rew_list in H; inverts H. { left⊢. } { right×. }

```

Qed.

```

Lemma last_eq_middle_inv : ∀ x y l1 l2 l,
  l & x = l1 & y ++ l2 →
  (l = l1 ∧ x = y ∧ l2 = nil) ∨ (∃ l2', l2 = l2'&x).

```

Proof using.

```

  introv E. destruct (last_case l2) as [(z&t&K)].
  { subst. rew_list in *. lets (?&?): last_eq_last_inv E. left×. }
  { subst. repeat rewrite ← app_assoc in E. lets (?&?): last_eq_last_inv E.
    subst. right×. }

```

Qed.

```

Lemma list_middle_inv : ∀ n l,
  n < length l →
  ∃ l1 x l2, l = l1 ++ x::l2 ∧ length l1 = n.

```

Proof using.

```

  intros. gen l. induction n; introv N;
  destruct l as [(x l'); rew_list in *]; try solve [ false; math ].
  { ∃⊢ (@nil A) x l'. }
  { forwards (l1&x'&l2&E&M): IHn l'. math.
    ∃ (x::l1) x' l2. subst l'. rew_list⊢. }

```

Qed.

End AppInversion.

```

Arguments last_eq_nil_inv [A] [a] [l].
Arguments nil_eq_last_inv [A] [a] [l].
Arguments app_eq_nil_inv [A] [l1] [l2].
Arguments nil_eq_app_inv [A] [l1] [l2].
Arguments nil_eq_middle_inv [A] [x] [l1] [l2].
Arguments cons_eq_middle_inv [A] [x] [y] [l1] [l2] [l].

```

`mem x l` asserts that `x` belongs to `l`. Remark: it could be also defined as `Exists (=x) l`.

Inductive `mem A (x:A) : list A → Prop` :=

```
| mem_here : ∀ l,
  mem x (x::l)
| mem_next : ∀ y l,
  mem x l →
  mem x (y::l).
```

Section Mem.

Variables `(A : Type)`.

Implicit Types `x : A`.

Implicit Types `l : list A`.

Hint Constructors `mem`.

Induction

```
Lemma mem_induct : ∀ (x : A) (P : list A → Prop),
  (∀ l : list A, P (x :: l)) →
  (∀ (y : A) (l : list A), mem x l → x ≠ y → P l → P (y :: l)) →
  (∀ l : list A, mem x l → P l).
```

Proof using.

```
introv HH HN M. induction l.
inverts M.
tests: (x = a). auto. invert M; auto_false×.
```

Qed.

Rewriting

```
Lemma mem_nil_eq : ∀ x,
```

```
  mem x nil = False.
```

Proof using. intros. *extens*. *iff* H; *invert* H. Qed.

```
Lemma mem_cons_eq : ∀ x y l,
```

```
  mem x (y::l) = (x = y ∨ mem x l).
```

Proof using. intros. *extens*. *iff* H; *invert*× H. Qed.

```
Lemma mem_cons_eq_cases : ∀ x y l,
```

```
  mem x (y::l) = ((x = y) ∨ (x ≠ y ∧ mem x l)).
```

Proof using.

```
intros. extens. tests: (x = y).
{ autos×. }
{ iff H. { invert¬ H. } { destruct H as [l(?&?)]; subst×. } }
```

Qed.

```
Lemma mem_one_eq : ∀ x y,
```

```
  mem x (y::nil) = (x = y).
```

Proof using. intros. *extens*. *iff* H; *invert*¬ H. *false_invert*. Qed.

Lemma mem_app_eq : $\forall l1\ l2\ x$,

mem $x\ (l1 ++ l2) = (\text{mem } x\ l1 \vee \text{mem } x\ l2)$.

Proof using.

```

intros. extens. induction l1; rew_list.
{ split. { auto. } { introv [H|?]. inverts H. auto. } }
{ iff M.
  { inverts $\neg$  M. rewrite IHL1 in H0. destruct $\times$  H0. }
  { destruct M. inverts $\neg$  H. constructors. rewrite $\neg$  IHL1.
    constructors. rewrite $\neg$  IHL1. } }

```

Qed.

Lemma mem_last_eq : $\forall x\ y\ l$,

mem $x\ (l\&y) = (\text{mem } x\ l \vee x = y)$.

Proof using. intros. *rewrite* mem_app_eq. *rewrite* $\neg$ mem_one_eq. Qed.

Lemma mem_last_eq_cases : $\forall x\ y\ l$,

mem $x\ (l\&y) = ((x \neq y \wedge \text{mem } x\ l) \vee (x = y))$.

Proof using.

```

intros. extens. induction l; rew_list.
{ tests: (x = y). { autos $\times$ . }
  { iff M.
    { inverts $\neg$  M. }
    { destruct M as [(?&H)|]. { inverts $\neg$  H. } { subst $\times$ . } } } }
{ tests: (x = y). { autos $\times$ . }
  { iff M.
    { inverts M as M'; auto. rewrite IHL in M'. destruct $\times$  M'. }
    { destruct M as [(?&H)|].
      { inverts $\neg$  H. constructors. rewrite $\times$  IHL. }
      { constructors. rewrite $\neg$  IHL. } } } }

```

Qed.

Backward

Lemma mem_cons : $\forall l\ x\ y$,

$x = y \vee \text{mem } x\ l \rightarrow$

mem $x\ (y::l)$.

Proof using. intros. *rewrite* $\times$ mem_cons_eq. Qed.

Lemma mem_cons_l : $\forall l\ x$,

mem $x\ (x::l)$.

Proof using. intros. *rewrite* $\times$ mem_cons_eq. Qed.

Lemma mem_cons_r : $\forall l\ x\ y$,

mem $x\ l \rightarrow$

mem $x\ (y::l)$.

Proof using. intros. *rewrite* $\times$ mem_cons_eq. Qed.

Lemma mem_app : $\forall l1\ l2\ x$,

mem x $l1$ $\vee$ **mem** x $l2 \rightarrow$
mem x ($l1 ++ l2$).

Proof using. intros. rewrite $\neg$ mem_app_eq. Qed.

Lemma mem_app_l : $\forall l1\ l2\ x$,

mem x $l1 \rightarrow$
mem x ($l1 ++ l2$).

Proof using. intros. *applies* $\times$ mem_app. Qed.

Lemma mem_app_r : $\forall l1\ l2\ x$,

mem x $l2 \rightarrow$
mem x ($l1 ++ l2$).

Proof using. intros. *applies* $\times$ mem_app. Qed.

Lemma mem_last : $\forall l\ x\ y$,

mem x $l \vee x = y \rightarrow$
mem x ($l \& y$).

Proof using. intros. rewrite $\times$ mem_last_eq. Qed.

Lemma mem_last_r : $\forall l\ x$,

mem x ($l \& x$).

Proof using. intros. rewrite $\times$ mem_last_eq. Qed.

Lemma mem_last_l : $\forall l\ x\ y$,

mem x $l \rightarrow$
mem x ($l \& y$).

Proof using. intros. rewrite $\times$ mem_last_eq. Qed.

Inversion

Lemma mem_nil_inv : $\forall x$,

mem x nil $\rightarrow$
False.

Proof using. *introv* E . *inverts* E . Qed.

Lemma mem_cons_inv : $\forall l\ x\ y$,

mem x ($y::l$) $\rightarrow$
 $x = y \vee$ **mem** x l .

Proof using. *introv* E . rewrite $\times$ mem_cons_eq in E . Qed.

Lemma not_mem_cons_inv : $\forall x\ y\ l$,

$\neg$ **mem** x ($y::l$) $\rightarrow$
 $x \neq y \wedge \neg$ **mem** x l .

Proof using.

introv M . split.
{ *intro_subst*. *false* $\times$ M . }
{ intros N . *false* $\times$ M . }

Qed.

Lemma mem_cons_inv_cases : $\forall l\ x\ y$,


```

mem  $x$  ( $y::l$ )  $\rightarrow$ 
 $x = y \vee (x \neq y \wedge \text{mem } x \ l).$ 
Proof using. intros  $E$ . rewrite $\times$  mem_cons_eq_cases in  $E$ . Qed.

Lemma mem_app_inv :  $\forall x \ l1 \ l2,$ 
mem  $x$  ( $l1 ++ l2$ )  $\rightarrow$ 
mem  $x \ l1 \vee \text{mem } x \ l2.$ 
Proof using. intros  $E$ . rewrite $\neg$  mem_app_eq in  $E$ . Qed.

Lemma mem_last_inv :  $\forall l \ x \ y,$ 
mem  $x$  ( $l\&y$ )  $\rightarrow$ 
 $(x \neq y \wedge \text{mem } x \ l) \vee x = y.$ 
Proof using. intros  $E$ . rewrite $\times$  mem_last_eq_cases in  $E$ . Qed.

Lemma mem_inv_middle_first :  $\forall l \ x,$ 
mem  $x \ l \rightarrow$ 
 $\exists l1 \ l2, \ l = l1 ++ x :: l2 \wedge \neg \text{mem } x \ l1.$ 
Proof using.
intros  $M$ . induction  $M$ .
{  $\exists$  (@nil  $A$ )  $l$ . rewrite $\times$  mem_nil_eq. }
{ tests  $C$ :  $(x=y)$ .
  {  $\exists$  (@nil  $A$ )  $l$ . rewrite $\times$  mem_nil_eq. }
  { destruct  $IHM$  as  $(l1\&l2\&E\&N)$ .  $\exists (y::l1) \ l2$ .
    subst. rew_list. rewrite $\times$  mem_cons_eq. } }
Qed.

Lemma mem_inv_middle :  $\forall l \ x,$ 
mem  $x \ l \rightarrow$ 
 $\exists l1 \ l2, \ l = l1 ++ x :: l2.$ 
Proof using. intros  $E$ . forwards $\times$   $(\&\&\&\&\&?)$ : mem_inv_middle_first  $E$ . Qed.

Lemma eq_list_of_not_mem :  $\forall l,$ 
 $(\forall x, \neg \text{mem } x \ l) \rightarrow$ 
 $l = \text{nil}.$ 
Proof using. intros  $P$ . destruct $\neg$   $l$ . false  $P$ . applys mem_here. Qed.

End Mem.

#[global]
Hint Rewrite mem_nil_eq mem_cons_eq mem_app_eq mem_last_eq : rew_listx.

```

$\text{Nth } n \ L \ x$ asserts that the n -th element of the list L exists and is exactly x

```

Inductive Nth  $A : \text{nat} \rightarrow \text{list } A \rightarrow A \rightarrow \text{Prop} :=
| Nth_zero :  $\forall l \ x,$ 
  Nth 0 ( $x::l$ )  $x$ 
| Nth_succ :  $\forall y \ n \ l \ x,$ 
  Nth  $n \ l \ x \rightarrow$ 
  Nth (S  $n$ ) ( $y::l$ )  $x$ .$ 
```

Section Nth.

Variables ($A : \text{Type}$).

Implicit Types $l : \text{list } A$.

Implicit Types $x : A$.

Implicit Types $n : \text{nat}$.

Hint Constructors **mem** Nth.

Lemma Nth_cons_match : $\forall n \ l \ x \ y$,

```
Nth  $n \ (y :: l) \ x =$   
  match  $n$  with  
  |  $O \Rightarrow x = y$   
  |  $S \ n' \Rightarrow$  Nth  $(S \ n') \ (y :: l) \ x$   
  end.
```

Proof using.

```
intros. extens. destruct  $n$  as [| $n'$ ].  
{ iff  $M$ . inverts  $\neg M$ . subst  $\neg$ . }  
{ iff  $M$ . inverts  $\neg M$ . subst  $\neg$ . }
```

Qed.

LATER: add *nth_last* lemma

Lemma Nth_functional : $\forall n \ l \ x1 \ x2$,

```
Nth  $n \ l \ x1 \rightarrow$   
Nth  $n \ l \ x2 \rightarrow$   
 $x1 = x2$ .
```

Proof using. *introv* $H1$. induction $H1$; intro $H2$; *inverts* $\neg H2$. Qed.

Lemma Nth_mem : $\forall l \ x \ n$,

```
Nth  $n \ l \ x \rightarrow$   
mem  $x \ l$ .
```

Proof using. *introv* N . induction N ; *autos* $\times$. Qed.

Lemma mem_Nth : $\forall l \ x$,

```
mem  $x \ l \rightarrow$   
 $\exists n$ , Nth  $n \ l \ x$ .
```

Proof using.

```
introv  $H$ . induction  $l$ ; rew_listx in *.  
{ inverts  $H$ . }  
{ destruct  $H$ . { subst  $\times$ . } { forwards  $\times (n \&?)$ : IHL. } }
```

Qed.

Lemma Nth_inbound : $\forall n \ l \ x$,

```
Nth  $n \ l \ x \rightarrow$   
 $n < \text{length } l$ .
```

Proof using.

```
intros  $n$ . induction  $n$ ; introv  $H$ ; inverts  $H$ ; rew_list in *.  
{ math. }
```

```

    { forwards*: IHn. math. }
Qed.

Lemma Nth_inbound_inv : ∀ n l,
  n < length l →
  ∃ x, Nth n l x.
Proof using.
  induction n; introv N; destruct l as [|a l']; rew_list in N; try solve [math].
  { eauto. }
  { simpls. forwards (x&Hx): IHn l'. math. ∃ x. apply× Nth_succ. }
Qed.

Lemma Nth_app_l : ∀ n x l1 l2,
  Nth n l1 x →
  Nth n (l1 ++ l2) x.
Proof using. intros n. induction n; introv H; inverts H; rew_list×. Qed.

Lemma Nth_app_r : ∀ n x l1 l2,
  Nth n l2 x →
  Nth (n + length l1)%nat (l1 ++ l2) x.
Proof using.
  intros. gen n. induction l1; introv H; rew_list.
  { applys_eq¬ H. }
  { applys_eq× Nth_succ. }
Qed.

Lemma Nth_app_r' : ∀ n m x l1 l2,
  Nth m l2 x →
  n = (m + length l1)%nat →
  Nth n (l1 ++ l2) x.
Proof using. intros. subst. applys× Nth_app_r. Qed.

Lemma Nth_nil_inv : ∀ n x,
  Nth n nil x →
  False.
Proof using. introv H. inverts H. Qed.

Lemma Nth_inv_neq_nil : ∀ n l x,
  Nth n l x →
  l ≠ nil.
Proof using. introv H. inverts H; auto_false. Qed.

Lemma Nth_cons_inv : ∀ n x y q,
  Nth n (y::q) x →
  (n = 0 ∧ x = y)
  ∨ (∃ m, n = m+1 ∧ Nth m q x).
Proof using.
  introv H. inverts H. { left×. } { right. ∃ n0. splits¬. math. }

```

Qed.

Lemma Nth_app_inv : $\forall n x l1 l2$,
 Nth $n (l1 ++ l2) x \rightarrow$
 (Nth $n l1 x$)
 $\vee (\exists m, n = \text{length } l1 + m \wedge \text{Nth } m l2 x)$.

Proof using.

introv. gen n. induction l1; introv H; rew_list in H.
 { right. rew_list. $\exists \neg n$. }
 { inverts H.
 { left $\neg$. }
 { forwards $\times M$: IHl1. destruct M as [(m&?&?)].
 { left $\neg$. }
 { right $\times$. rew_list. $\exists m$. split $\neg$. math. } } }

Qed.

Lemma Nth_last_inv : $\forall n x y l$,
 Nth $n (l \& y) x \rightarrow$
 (Nth $n l x$)
 $\vee (n = \text{length } l \wedge y = x)$.

Proof using.

introv H. lets [(m&E&F)]: Nth_app_inv H.
 { left $\neg$. }
 { right. inverts F as G. { splits $\neg$. math. } { inverts G. } }

Qed.

End Nth.

19.4.1 nth_default as a partial function with a default

Fixpoint nth_default $A (d:A) (n:\text{nat}) (l:\text{list } A) \{ \text{struct } l \} : A :=$
 match l with
 | nil $\Rightarrow d$
 | $x :: l' \Rightarrow$
 match n with
 | 0 $\Rightarrow x$
 | S $n' \Rightarrow \text{nth\_default } d n' l'$
 end
 end.

Section NthDef.

Variables $(A:\text{Type})$.

Implicit Types $n : \text{nat}$.

Implicit Types $d x : A$.

Implicit Types $l : \text{list } A$.

Hint Constructors **Nth**.

Lemma nth_default_nil : $\forall n d$,
nth_default $d n$ nil = d .

Proof using. *intros*. destruct $\neg n$. Qed.

Lemma nth_default_zero : $\forall x l d$,
nth_default $d 0 (x :: l) = x$.

Proof using. *intros*. reflexivity. Qed.

Lemma nth_default_succ : $\forall n x l d$,
nth_default $d (S n) (x :: l) = \text{nth\_default } d n l$.

Proof using. *intros*. reflexivity. Qed.

Definition nth_default_cons := nth_default_succ.

Lemma nth_default_gt_length : $\forall l d n$,
 $n \geq \text{length } l \rightarrow$
nth_default $d n l = d$.

Proof using.
induction l as [|l']; *intros N; rew_list* in *.
{ auto. }
{ destruct n as [|n']. { false; math. }
 simpl. rewrite $\neg IHL$. math. }

Qed.

Lemma nth_default_of_Nth : $\forall n l x d$,
Nth $n l x \rightarrow$
nth_default $d n l = x$.

Proof using. *intros H*. induction $\neg H$. Qed.

Lemma Nth_of_nth_default : $\forall l d n x$,
nth_default $d n l = x \rightarrow$
 $n < \text{length } l \rightarrow$
Nth $n l x$.

Proof using.
intros l. induction l ; *rew_list; intros E N*.
{ false. math. }
{ destruct n ; *simpls*.
 { subst $\times$. }
 { *constructors*. *applies* $\times IHL$. math. } }

Qed.

End NthDef.

Arguments nth_default $[A]$: simpl *never*.

#[*global*]

Hint Rewrite nth_default_nil nth_default_zero nth_default_succ : *rew_listx*.

19.4.2 nth as a partial function

Definition nth ‘{IA:Inhab A} :=
nth_default (arbitrary (A:=A)).

Section NthFunc.

Context (A:Type) {IA: Inhab A}.

Implicit Types n : nat.

Implicit Types x : A.

Implicit Types l : list A.

Lemma nth_lt_length : $\forall x l$,
nth 0 (x::l) = x.

Proof using. intros. apply nth_default_zero. Qed.

Lemma nth_zero : $\forall x l$,
nth 0 (x::l) = x.

Proof using. intros. apply nth_default_zero. Qed.

Lemma nth_succ : $\forall n x l$,
nth (S n) (x::l) = nth n l.

Proof using. intros. apply nth_default_succ. Qed.

Definition nth_cons := nth_succ.

Lemma nth_cons_case : $\forall n x l$,
nth n (x::l) = (If n = 0 then x else nth (n-1) l).

Proof using.

intros. destruct n as [|n’].
{ case_if. rewrite $\neg$ nth_zero. }
{ case_if. rewrite $\neg$ nth_succ. fequals; math. }

Qed.

Lemma nth_pos : $\forall n x l$,
n > 0 $\rightarrow$
nth n (x::l) = nth (n-1) l.

Proof using.

intros. destruct n.
{ false. math. }
{ rewrite nth_succ. fequals. math. }

Qed.

Lemma nth_gt_length : $\forall l n$,
n $\geq$ length l $\rightarrow$
nth n l = arbitrary.

Proof using. introv N. applies $\neg$ nth_default_gt_length. Qed.

Lemma nth_of_Nth : $\forall n l x$,
Nth n l x $\rightarrow$

```

    nth n l = x.
Proof using. intros H. apply¬ nth_default_of_Nth. Qed.
Lemma Nth_of_nth :  $\forall l\ n\ x,$ 
    nth n l = x  $\rightarrow$ 
    n < length l  $\rightarrow$ 
    Nth n l x.
Proof using. intros. applies× Nth_of_nth_default. Qed.
Lemma Nth_nth :  $\forall l\ n,$ 
    n < length l  $\rightarrow$ 
    Nth n l (nth n l).
Proof using. intros. applies× Nth_of_nth. Qed.
Lemma mem_nth :  $\forall l\ x,$ 
    mem x l  $\rightarrow$ 
     $\exists n,$  nth n l = x.
Proof using.
    intros. forwards [n P]: mem_Nth H.  $\exists n.$  apply¬ nth_of_Nth.
Qed.
Lemma nth_mem :  $\forall n\ l\ x,$ 
    nth n l = x  $\rightarrow$ 
    n < length l  $\rightarrow$ 
    mem x l.
Proof using. intros E N. forwards¬ H: Nth_of_nth E. applies× Nth_mem H. Qed.
Lemma nth_app_l :  $\forall n\ l1\ l2,$ 
    n < length l1  $\rightarrow$ 
    nth n (l1 ++ l2) = nth n l1.
Proof using.
    intros N. applies nth_of_Nth. applies Nth_app_l. applies¬ Nth_nth.
Qed.
Lemma nth_app_r :  $\forall n\ l1\ l2,$ 
    n  $\geq$  length l1  $\rightarrow$ 
    nth n (l1 ++ l2) = nth (n - length l1) l2.
Proof using.
    unfold nth. intros n l1. gen n. induction l1; intros N; rew_list in *.
    { fequals; math. }
    { destruct n as [|n']. { false; math. }
      unfold nth_default; fold nth_default. simpl.
      rewrite IHl1; [|math].
      fequals; math. }
Qed.
Lemma nth_app :  $\forall n\ l1\ l2,$ 
    nth n (l1 ++ l2) = If (n < length l1) then (nth n l1) else (nth (n - length l1) l2).

```

Proof using.

```
  intros. case_if. { applys ¬ nth_app_l. } { applys nth_app_r; math. }  
Qed.
```

Lemma nth_middle : $\forall n \ l1 \ x \ l2$,
 $n = \text{length } l1 \rightarrow$
 $\text{nth } n \ (l1 ++ x :: l2) = x$.

Proof using.

```
  introv E. rewrite nth_app_r.  
  { math_rewrite (n - length l1 = 0). rewrite ¬ nth_zero. }  
  math.  
Qed.
```

Lemma nth_last_case : $\forall n \ x \ l$,
 $\text{nth } n \ (l \&x) = (\text{If } n = \text{length } l \text{ then } x \text{ else } \text{nth } n \ l)$.

Proof using.

```
  intros. case_if.  
  { rewrite nth_app_r; [|math|. math_rewrite (n-length l = 0).  
    rewrite ¬ nth_zero. }  
  { tests C': (n < length l).  
    { rewrite ¬ nth_app_l. }  
    { rewrite nth_gt_length; [|rew_list; math|. ]  
      rewrite nth_gt_length; [|math|. auto. } }  
Qed.
```

End NthFunc.

Section NthFuncAux.

Context (A :Type) { IA : Inhab A }.

Implicit Types n : nat.

Implicit Types x : A .

Implicit Types l : list A .

Lemma eq_of_extens : $\forall l1 \ l2$,
 $\text{length } l1 = \text{length } l2 \rightarrow$
 $(\forall n, n < \text{length } l1 \rightarrow \text{nth } n \ l1 = \text{nth } n \ l2) \rightarrow$
 $l1 = l2$.

Proof using IA .

```
  intros l1. induction l1; intros l2; destruct l2; simpl; introv HL HN;  
  try solve [ eauto | false ]. rew_list in *. f_equal.  
  { forwards M: (rm HN) 0. math. do 2 rewrite nth_zero in M. auto. }  
  { applys IHL1. { math. }  
    { intros n L. forwards M: (rm HN) (S n). math.  
      do 2 rewrite nth_cons in M. auto. } }  
Qed.
```

End NthFuncAux.

Arguments nth [A] {IA}.

Global Opaque nth.

#[global]

Hint Rewrite nth_zero nth_succ : *rew_listx*.

19.4.3 Rev

Definition rev A (l:list A) : list A :=
 fold_left (fun x acc => x :: acc) (@nil A) l.

Section Rev.

Variables (A : Type).

Implicit Types x : A.

Implicit Types l : list A.

Lemma rev_nil :

rev (@nil A) = nil.

Proof using. auto. Qed.

Lemma rev_app : ∀ l1 l2,

rev (l1 ++ l2) = rev l2 ++ rev l1.

Proof using.

intros. unfold rev. asserts K1: (∀ l accu,
 fold_left (fun x acc => x :: acc) accu l =
 fold_left (fun x acc => x :: acc) nil l ++ accu).
 { induction l; intros.
 { auto. }
 { *rew_listx*. rewrite IHL. rewrite (@IHL (a::nil)). *rew_list*¬. } }
 asserts K2: (∀ accu,
 fold_left (fun x acc => x :: acc) accu (l1 ++ l2) =
 fold_left (fun x acc => x :: acc) nil l2 ++
 fold_left (fun x acc => x :: acc) nil l1 ++ accu).
 { induction l1; intros.
 { *rew_list*. apply K1. }
 { *rew_listx*. rewrite IHL1. rewrite (@K1 l1 (a::nil)). *rew_list*¬. } }
 lets K3: (@K2 nil). rewrite app_nil_r in K3. auto.

Qed.

Lemma rev_cons : ∀ x l,

rev (x :: l) = rev l & x.

Proof using. intros. rewrite ← app_cons_one_r. rewrite¬ rev_app. Qed.

Lemma rev_one : ∀ x,

rev (x :: nil) = x :: nil.

Proof using. intros. rewrite¬ rev_cons. Qed.

```

Lemma rev_last : ∀ x l,
  rev (l & x) = x :: (rev l).
Proof using. intros. rewrite¬ rev_app. Qed.

Lemma rev_rev : ∀ l,
  rev (rev l) = l.
Proof using.
  intros. induction¬ l. { rewrite rev_cons, rev_last. fequals. }
Qed.

Lemma rev_inj : ∀ l1 l2,
  rev l1 = rev l2 →
  l1 = l2.
Proof using.
  introv E. forwards E': f_equal (@rev A) (rm E).
  do 2 rewrite¬ rev_rev in E'.
Qed.

Lemma mem_rev : ∀ l x,
  mem x l →
  mem x (rev l).
Proof using.
  introv H. induction H.
  { rewrite rev_cons. apply mem_last_r. }
  { rewrite rev_cons. apply¬ mem_last_l. }
Qed.

Lemma mem_rev_eq : ∀ l x,
  mem x (rev l) = mem x l.
Proof using.
  extens. iff M.
  { lets H: mem_rev M. rewrite¬ rev_rev in H. }
  { apply× mem_rev. }
Qed.

Lemma length_rev : ∀ l,
  length (rev l) = length l.
Proof using. intros. induction¬ l. { rewrite rev_cons. rew_list¬. } Qed.

Lemma fold_right_rev : ∀ B (f : A → B → B) i l,
  fold_right f i (rev l) = fold_left f i l.
Proof using.
  introv. gen i. induction¬ l.
  { introv. rewrite rev_cons. rew_listx¬. }
Qed.

Lemma Nth_rev : ∀ n x l,
  Nth n l x →

```

```

Nth (length  $l - 1 - n$ ) (rev  $l$ )  $x$ .
Proof using.
  intro. sets_eq  $N$ : (length  $l$ ). gen  $x$   $n$   $l$ .
  induction  $N$ ; intro  $E$   $I$ ; destruct  $l$ ; tryfalse.
  { inverts  $I$ . }
  { rewrite rev_cons. inverts  $I$  as  $I$ .
    { applys Nth_app_r' 0.
      { constructors. }
      { rewrite length_rev. rewrite length_cons in  $E$ . math. } }
    { rew_list in  $E$ . lets  $K$ : IHN  $I$ . { math. }
      apply Nth_app_l. applys_eq  $K$ . math. } }
Qed.

Lemma Nth_rev' :  $\forall$   $n$   $x$   $l$ ,
   $n < \text{length } l \rightarrow$ 
  Nth (length  $l - 1 - n$ )  $l$   $x \rightarrow$ 
  Nth  $n$  (rev  $l$ )  $x$ .
Proof using.
  intro  $L$   $M$ . applys_eq ( $\gg$  Nth_rev  $M$ ). math.
Qed.

Lemma Nth_rev_inv :  $\forall$   $n$   $x$   $l$ ,
  Nth  $n$  (rev  $l$ )  $x \rightarrow$ 
  Nth (length  $l - 1 - n$ )  $l$   $x$ .
Proof using.
  intro  $N$ . lets  $K$ : Nth_rev  $N$ . rewrite rev_rev, length_rev in  $K$ . auto.
Qed.

Lemma nth_rev :  $\forall$  ' $\{Inhab\ A\}$   $n$   $l$ ,
   $n < \text{length } l \rightarrow$ 
  nth  $n$  (rev  $l$ ) = nth (length  $l - 1 - n$ )  $l$ .
Proof using.
  intro  $L$ . applys nth_of_Nth. applys  $\neg$  Nth_rev'.
  applys  $\neg$  Nth_of_nth. math.
Qed.

End Rev.

Global Opaque rev.

#[global]
Hint Rewrite rev_nil rev_app rev_cons rev_last rev_rev length_rev : rew_list.
#[global]
Hint Rewrite rev_nil rev_app rev_cons rev_last rev_rev length_rev : rew_listx.

```

19.4.4 Inversion for rev

Section RevInversion.

Variables ($A : \text{Type}$).

Implicit Types $l : \text{list } A$.

Lemma rev_eq_nil_inv : $\forall l$,
 rev $l = \text{nil} \rightarrow$
 $l = \text{nil}$.

Proof using.

intros l . destruct l ; rew_list; intros.
 { auto. } { false $\times$ last_eq_nil_inv. }

Qed.

Lemma nil_eq_rev_inv : $\forall l$,
 $\text{nil} = \text{rev } l \rightarrow$
 $l = \text{nil}$.

Proof using. introv H . apply $\neg$ rev_eq_nil_inv. Qed.

Lemma app_rev_eq_nil_inv : $\forall l1\ l2$,
 $l1 ++ \text{rev } l2 = \text{nil} \rightarrow$
 $l1 = \text{nil} \wedge l2 = \text{nil}$.

Proof using.

intros. lets $H1\ H2$: app_eq_nil_inv H .
 applys_to $H2$ rev_eq_nil_inv. autos $\times$.

Qed.

Lemma nil_eq_app_rev_inv : $\forall l1\ l2$,
 $\text{nil} = l1 ++ \text{rev } l2 \rightarrow$
 $l1 = \text{nil} \wedge l2 = \text{nil}$.

Proof using. intros. apply $\times$ app_rev_eq_nil_inv. Qed.

End RevInversion.

Arguments rev_eq_nil_inv [A] [l].

Arguments nil_eq_rev_inv [A] [l].

Arguments app_rev_eq_nil_inv [A] [$l1$] [$l2$].

Arguments nil_eq_app_rev_inv [A] [$l1$] [$l2$].

19.5 Make

Fixpoint make A ($n : \text{nat}$) ($v : A$) : list A :=
 match n with
 | 0 $\Rightarrow$ nil
 | S n' $\Rightarrow v :: \text{make } n' v$
 end.

```

Section Make.
Context (A:Type) {IA:Inhab A}.
Implicit Types i n : nat.
Implicit Types v : A.
Implicit Types l : list A.

Lemma make_zero : ∀ v,
  make 0 v = nil.
Proof using. auto. Qed.

Lemma make_succ : ∀ n v,
  make (S n) v = v :: (make n v).
Proof using. auto. Qed.

Lemma make_pos : ∀ n v,
  n > 0 →
  make n v = v :: (make (n-1) v).
Proof using.
  introv E. destruct n.
  { math. }
  { rewrite make_succ. fequals_rec. math. }
Qed.

Lemma length_make : ∀ n v,
  length (make n v) = n.
Proof using.
  intros. induction n.
  { auto. }
  { rewrite make_succ. rewrite length_cons. math. }
Qed.

Lemma Nth_make : ∀ i n v,
  i < n →
  Nth i (make n v) v.
Proof using.
  Hint Constructors Nth.
  introv. gen n; induction i; introv E; destruct n; try solve [ false; math ].
  { constructors. }
  { rewrite make_succ. applies Nth_succ. applies IHi. math. }
Qed.

Lemma nth_make : ∀ i n v,
  i < n →
  nth i (make n v) = v.
Proof using. intros. applies nth_of_Nth. applies Nth_make. Qed.

End Make.

Global Opaque make.

```

```
#[global]
Hint Rewrite make_zero make_succ length_make : rew_listx.
```

19.6 Update as a function

```
Fixpoint update A (n:nat) (v:A) (l:list A) { struct l } : list A :=
  match l with
  | nil  $\Rightarrow$  nil
  | x::l'  $\Rightarrow$ 
    match n with
    | 0  $\Rightarrow$  v::l'
    | S n'  $\Rightarrow$  x::update n' v l'
    end
  end.
```

Section Update.

Context (*A*:Type) {*IA*: Inhab *A*}.

Implicit Types *n* : nat.

Implicit Types *x v* : *A*.

Implicit Types *l* : list *A*.

Lemma update_nil : $\forall$ *n v*,
 update *n v* nil = nil.

Proof using. auto. Qed.

Lemma update_cons_match : $\forall$ *n v x l*,
 update *n v* (*x*::*l*) =
 match *n* with
 | 0 $\Rightarrow$ *v*::*l*
 | S *n'* $\Rightarrow$ *x*::(update *n'* *v* *l*)
 end.

Proof using. auto. Qed.

Lemma update_zero : $\forall$ *v x l*,
 update 0 *v* (*x*::*l*) = *v*::*l*.

Proof using. auto. Qed.

Lemma update_succ : $\forall$ *n v x l*,
 update (S *n*) *v* (*x*::*l*) = *x*::(update *n v l*).

Proof using. auto. Qed.

Definition update_cons := update_succ.

Lemma update_cons_case : $\forall$ *n v x l*,
 update *n v* (*x*::*l*) = (If *n* = 0 then *v*::*l* else *x*::(update (*n*-1) *v* *l*)).

Proof using.

intros. destruct *n* as [|*n'*]; case_if.

```

    { aplys update_zero. }
    { rewrite update_cons. fequals_rec. math. }
Qed.

```

```

Lemma update_cons_pos :  $\forall n v x l,$ 
   $n > 0 \rightarrow$ 
   $\text{update } n v (x :: l) = x :: (\text{update } (n-1) v l).$ 

```

```

Proof using.
  intros N. rewrite update_cons_case. case-if.
  { false. math. }
  { auto. }
Qed.

```

```

Lemma update_app_l :  $\forall l1 l2 n v,$ 
   $n < \text{length } l1 \rightarrow$ 
   $\text{update } n v (l1 ++ l2) = \text{update } n v l1 ++ l2.$ 

```

```

Proof using IA.
  intros l1. induction l1 as [| x l1']; introv E; rew_list in *.
  { fequals. math. }
  { destruct n as [| n'].
    { do 2 rewrite update_zero. rew_list  $\neg$ . }
    { do 2 rewrite update_cons. rew_list  $\neg$ .
      fequals. erewrite  $\times$  IHL1'. math. } }

```

Qed.

```

Lemma update_app_r :  $\forall l1 l2 n v,$ 
   $n \geq \text{length } l1 \rightarrow$ 
   $\text{update } n v (l1 ++ l2) = l1 ++ \text{update } (n - \text{length } l1) v l2.$ 

```

```

Proof using IA.
  intros l1. induction l1 as [| x l1']; introv E; rew_list in *.
  { fequals. math. }
  { destruct n as [| n']. { false. math. }
    { rewrite update_cons. fequals. erewrite  $\times$  IHL1'. math. } }

```

Qed.

```

Lemma update_middle :  $\forall l1 l2 n x v,$ 
   $n = \text{length } l1 \rightarrow$ 
   $\text{update } n v (l1 ++ x :: l2) = l1 \& v ++ l2.$ 

```

```

Proof using IA.
  intros. subst. rew_list. rewrites update_app_r.
  { math_rewrite ( $\text{length } l1 - \text{length } l1 = 0$ ). rew_list  $\neg$ . }
  { math. }

```

Qed.

```

Lemma update_ge_length :  $\forall n v l,$ 
   $n \geq \text{length } l \rightarrow$ 

```

```

    update  $n$   $v$   $l$  =  $l$ .
Proof using.
  introv  $E$ . gen  $n$ . induction  $l$ ; rew_list; intros.
  { auto. }
  { rewrite update_cons_pos; [|math|. fequals. applies IHL. math. ] }
Qed.

Lemma length_update :  $\forall$   $n$   $v$   $l$ ,
  length (update  $n$   $v$   $l$ ) = length  $l$ .
Proof using.
  intros. gen  $n$ . induction  $l$ ; intros.
  { auto. }
  { destruct  $n$  as [| $n'$ ].
    { rewrite update_zero. rew_list $\neg$ . }
    { rewrite update_succ. rew_list. rewrite $\neg$  IHL. } } }
Qed.

Lemma nth_update_same :  $\forall$   $n$   $l$   $v$ ,
   $n$  < length  $l$   $\rightarrow$ 
  nth  $n$  (update  $n$   $v$   $l$ ) =  $v$ .
Proof using.
  intros  $n$   $l$ . gen  $n$ . induction  $l$ ; introv  $N$ ; rew_list in  $N$ .
  { false. math. }
  { destruct  $n$  as [| $n'$ ].
    { rewrite update_zero. rew_listx $\neg$ . }
    { rewrite update_cons. rew_listx. applies $\times$  IHL. math. } } }
Qed.

Lemma nth_update_neq :  $\forall$   $n$   $m$   $l$   $v$ ,
   $n \neq m$   $\rightarrow$ 
  nth  $n$  (update  $m$   $v$   $l$ ) = nth  $n$   $l$ .
Proof using.
  intros  $n$   $m$   $l$ . gen  $n$   $m$ . induction  $l$ ; introv  $N$ .
  { rewrite $\neg$  update_nil. }
  { destruct  $m$  as [| $m'$ ].
    { rewrite update_zero. do 2 (rewrite nth_pos; [|math|]). auto. }
    { rewrite update_succ. destruct  $n$  as [| $n'$ ].
      { rew_listx $\neg$ . }
      { rew_listx. applies $\neg$  IHL. } } } }
Qed.

Lemma update_nth_same :  $\forall$   $n$   $l$ ,
   $n$  < length  $l$   $\rightarrow$ 
  update  $n$  (nth  $n$   $l$ )  $l$  =  $l$ .
Proof using.

```



```

    introv E. gen n. induction l as [|x l']; intros.
    { rewrite length_nil in E. false. math. }
    { rewrite length_cons in E. rewrite update_cons_case.
      destruct n as [|n']; case_if.
      { subst. rewrite× nth_zero. }
      { fequals. rewrite nth_cons. math_rewrite (S n' - 1 = n')%nat.
        rewrite× IHI'. math. } }
Qed.

End Update.

Global Opaque update.

#[global]
Hint Rewrite length_update update_nil update_zero update_succ : rew_listx.

```

19.6.1 Map

```

Definition map A B (f:A→B) (l:list A) : list B :=
  fold_right (fun x acc => (f x)::acc) (@nil B) l.

Section Map.
Variable (A B : Type).
Implicit Types x : A.
Implicit Types l : list A.
Implicit Types f : A → B.

Lemma map_nil : ∀ f,
  map f nil = nil.
Proof using. auto. Qed.

Lemma map_cons : ∀ f x l,
  map f (x::l) = f x :: map f l.
Proof using. auto. Qed.

Lemma map_app : ∀ f l1 l2,
  map f (l1 ++ l2) = map f l1 ++ map f l2.
Proof using.
  intros. unfold map.
  assert (∀ accu,
    fold_right (fun x acc => f x :: acc) accu (l1 ++ l2) =
    fold_right (fun x acc => f x :: acc) nil l1 ++
    fold_right (fun x acc => f x :: acc) nil l2 ++ accu).
  { induction l1; intros; simpl.
    { rew_list. gen accu.
      induction l2; intros.
      { auto. }
    }
  }

```

```

      { rew_listx. rewrite  $\neg$  IHL2. } }
      { rew_listx. fequals. } }
    specializes H (@nil B). rew_list  $\neg$  in H.
Qed.

Lemma map_last :  $\forall f x l$ ,
  map f (l & x) = map f l & f x.
Proof using. intros. rewrite  $\neg$  map_app. Qed.

Lemma map_rev :  $\forall f l$ ,
  map f (rev l) = rev (map f l).
Proof using.
  intros. induction  $\neg$  l.
  { rewrite map_cons. rew_list. rewrite map_last. rewrite  $\neg$  IHL. }
Qed.

Lemma length_map :  $\forall f l$ ,
  length (map f l) = length l.
Proof using.
  intros. induction  $\neg$  l.
  { rewrite map_cons. do 2 rewrite length_cons. auto. }
Qed.

Lemma map_make :  $\forall f n v$ ,
  map f (make n v) = make n (f v).
Proof using.
  intros. induction n as [|n'].
  { do 2 rewrite make_zero. auto. }
  { do 2 rewrite make_succ. rewrite map_cons. fequals. }
Qed.

Lemma map_update :  $\forall n f l x$ ,
  n < length l  $\rightarrow$ 
  map f (update n x l) = update n (f x) (map f l).
Proof using.
  introv I. gen n. induction l; intros; rew_list in *.
  { false; math. }
  { destruct n as [|n'].
    { rewrite  $\neg$  update_zero. }
    { rewrite map_cons. do 2 rewrite update_cons. rewrite map_cons.
      fequals. applies IHL. math. } }
Qed.

Lemma map_eq_nil_inv :  $\forall f l$ ,
  map f l = nil  $\rightarrow$ 
  l = nil.
Proof using.

```

intros E. destruct $\neg l$. rewrite map_cons in E. false.

Qed.

Lemma map_eq_cons_inv : $\forall f\ l1\ (x2:B)\ t2,$
 $\text{map } f\ l1 = x2 :: t2 \rightarrow$
 $\exists x1\ t1, l1 = x1 :: t1 \wedge x2 = f\ x1 \wedge t2 = \text{map } f\ t1.$

Proof using.

intros E. destruct l1; [false].
rewrite map_cons in E. inverts $\times$ E.

Qed.

Lemma map_eq_app_inv : $\forall f\ l\ r1\ r2,$
 $\text{map } f\ l = r1 ++ r2 \rightarrow$
 $\exists l1\ l2, l = l1 ++ l2 \wedge r1 = \text{map } f\ l1 \wedge r2 = \text{map } f\ l2.$

Proof using.

intros f l. induction l as [|x l']; intros E.
{ rewrite map_nil in E. lets (?&?): nil_eq_app_inv E. subst.
 $\exists \times$ (@nil A) (@nil A). }
{ rewrite map_cons in E. destruct r1 as [|y r1']; rew_list in E.
{ $\exists \times$ (@nil A) (x :: l'). }
{ invert E ;=> N1 N2. forwards (l1'&l2&E1&E2&E3): IHL' N2.
 $\exists$ (x :: l1') l2. subst $\times$. } }

Qed.

Lemma map_eq_middle_inv : $\forall f\ l\ r1\ y\ r2,$
 $\text{map } f\ l = r1 ++ y :: r2 \rightarrow$
 $\exists l1\ x\ l2, l = l1 ++ x :: l2 \wedge r1 = \text{map } f\ l1 \wedge y = f\ x \wedge r2 = \text{map } f\ l2.$

Proof using.

intros E. lets (l1&l2'&E1&E2&E3): map_eq_app_inv (rm E).
lets (x&l2&E4&E5&E6): map_eq_cons_inv (eq-sym E3).
 $\exists l1\ x\ l2$. subst $\times$.

Qed.

Lemma map_inj : $\forall f\ l1\ l2,$
 $(\forall x\ y, f\ x = f\ y \rightarrow x = y) \rightarrow$
 $\text{map } f\ l1 = \text{map } f\ l2 \rightarrow$
 $l1 = l2.$

Proof using.

intros f l1. induction l1; intros I E.
{ rewrite map_nil in E. forwards: map_eq_nil_inv. }*
{ destruct l2 as [|b l2]; tryfalse. do 2 rewrite map_cons in E.
inverts E. fequals $\times$. }

Qed.

Lemma map_congr : $\forall f1\ f2\ l,$
 $(\forall x, \text{mem } x\ l \rightarrow f1\ x = f2\ x) \rightarrow$

```

    map f1 l = map f2 l.
Proof using.
  introv H. induction l. { auto. }
  { do 2 rewrite map_cons. fequals.
    { applys H. applys mem_here. }
    { applys IHL. intros x Hx. applys H. applys× mem_next. } }
Qed.

End Map.

Global Opaque map.

#[global]
Hint Rewrite map_nil map_cons map_app map_last length_map : rew_listx.

Lemma map_id : ∀ A (l:list A),
  map id l = l.
Proof using. introv. induction¬ l. rew_listx. fequals¬. Qed.

Lemma map_id_ext : ∀ A (l:list A) (f:A→A),
  (∀ x, f x = x) →
  map f l = l.
Proof using. introv E. cuts_rewrite (f = id). apply map_id. extens¬. Qed.

Lemma mem_map : ∀ (A B:Type) (f:A→B) (l:list A) x,
  mem x l →
  mem (f x) (map f l).
Proof using. introv M. induction M; rew_listx; auto. Qed.

Lemma mem_map' : ∀ A B (l:list A) (f:A→B) (x:A) (y:B),
  mem x l →
  y = f x →
  mem y (map f l).
Proof using. intros. subst. applys× mem_map. Qed.

Lemma map_map : ∀ A B C (l:list A) (f:A→B) (g:B→C),
  map g (map f l) = map (fun x ⇒ g (f x)) l.
Proof using.
  intros. induction l as [|x l'].
  { auto. }
  { repeat rewrite map_cons. fequals. }
Qed.

Lemma Nth_map : ∀ (A B:Type) (f:A→B) (l:list A) n (x:A),
  Nth n l x →
  Nth n (map f l) (f x).
Proof using.
#[global]
Hint Constructors Nth.

```

```

    introv M. induction M; rew_listx; auto.
Qed.

Lemma Nth_map_inv : ∀ (A B:Type) (f:A→B) (l:list A) n (y:B),
  Nth n (map f l) y →
  ∃ x, y = f x ∧ Nth n l x.
Proof using.
  introv N. gen n. induction l; introv N.
  { inverts N. }
  { rewrite map_cons in N. inverts N as N.
    { autos×. }
    { lets (x&E&N'): IHL N. ∃× x. } }
Qed.

Lemma nth_map : ∀ ‘{IA:Inhab A} ‘{IB:Inhab B} (f:A→B) (l:list A) n,
  n < length l →
  nth n (map f l) = f (nth n l).
Proof using.
  introv N. applys nth_of_Nth. applys Nth_map. applys¬ Nth_of_nth.
Qed.

```

19.6.2 Concat

```

Definition concat A (m:list (list A)) : list A :=
  fold_right (@app A) (@nil A) m.

Section Concat.
Variables (A : Type).
Implicit Types x : A.
Implicit Types l : list A.
Implicit Types m : list (list A).

Lemma concat_nil :
  concat (@nil (list A)) = nil.
Proof using. auto. Qed.

Lemma concat_cons : ∀ l m,
  concat (l::m) = l ++ concat m.
Proof using. auto. Qed.

Lemma concat_one : ∀ l,
  concat (l::nil) = l.
Proof using.
  intros. rewrite concat_cons. rewrite concat_nil. rew_list¬.
Qed.

Lemma concat_app : ∀ m1 m2 : list (list A),

```

```

    concat (m1 ++ m2) = concat m1 ++ concat m2.
Proof using.
  intros m1. induction m1; intros.
  { rewrite concat_nil. rew_list¬. }
  { rew_list. do 2 rewrite concat_cons. rew_list. fequals. }
Qed.

Lemma concat_last : ∀ l m,
  concat (m & l) = concat m ++ l.
Proof using. intros. rewrite¬ concat_app. rewrite¬ concat_one. Qed.

Lemma mem_concat_eq : ∀ m x,
  mem x (concat m)
  = ∃ l, mem l m ∧ mem x l.
Proof using.
  Hint Constructors mem.
  introv. extens. induction m.
  { simpl. iff I. { inverts I. } { destruct I as (?&H&?). inverts H. } }
  { rewrite concat_cons. rewrite mem_app_eq. iff I (l&M&N).
    { destruct I as [I|I].
      { ∃¬ a. }
      { rewrite IHm in I. destruct× I as (l&?&?). } }
    { inverts× M. } }
Qed.

Lemma concat_eq_nil_mem_inv : ∀ l m,
  concat m = nil →
  mem l m →
  l = nil.
Proof using.
  introv E M. induction M as [|x].
  { rewrite concat_cons in E. forwards*: app_eq_nil_inv E. }
  { rewrite concat_cons in E. destruct x.
    { rew_list in E. applies¬ IHM. }
    { rew_list in E. false. } }
Qed.

End Concat.

Global Opaque concat.

#[global]
Hint Rewrite concat_nil concat_cons concat_app concat_last : rew_listx.

```

19.6.3 Filter

`filter` P l produces a list l' that is the sublist of l made exactly of the elements of l that satisfy P .

Definition `filter` A ($P:A \rightarrow \text{Prop}$) ($l:\text{list } A$) : `list` A :=
`fold_right` (`fun` x $acc \Rightarrow$ `If` P x `then` $x::acc$ `else` acc) (`@nil` A) l .

Section `Filter`.

Variables (A : `Type`).

Implicit Types x : A .

Implicit Types l : `list` A .

Implicit Types P : $A \rightarrow \text{Prop}$.

Hint Constructors `mem`.

Lemma `filter_nil` : $\forall P$,
`filter` P `nil` = `nil`.

Proof using. `auto`. Qed.

Lemma `filter_cons` : $\forall x$ l P ,
`filter` P ($x::l$) = (`If` P x `then` $x::$ `filter` P l `else` `filter` P l).

Proof using. `auto`. Qed.

Lemma `filter_app` : $\forall l1$ $l2$ P ,
`filter` P ($l1 ++ l2$) = `filter` P $l1$ ++ `filter` P $l2$.

Proof using. `intros`. `unfold filter`.

```
assert ( $\forall$   $acc$ ,
  fold_right (fun  $x$   $acc \Rightarrow$  If  $P$   $x$  then  $x::acc$  else  $acc$ )  $acc$  ( $l1 ++ l2$ ) =
  fold_right (fun  $x$   $acc \Rightarrow$  If  $P$   $x$  then  $x::acc$  else  $acc$ ) nil  $l1$  ++
  fold_right (fun  $x$   $acc \Rightarrow$  If  $P$   $x$  then  $x::acc$  else  $acc$ ) nil  $l2$  ++  $acc$ ).
{ induction  $l1$ ; intros.
  { rew-list. gen  $acc$ . induction  $l2$ ; intros.
    { auto. }
    { rew-listx. case-if. rew-list. fequals. rewrite  $IHl2$ . fequals. } }
    { rew-listx. case-if. rew-list. fequals. rewrite  $IHl1$ . fequals. } }
  specializes  $H$  (@nil  $A$ ). rewrite  $\neg$  app_nil_r in  $H$ .
```

Qed.

Lemma `filter_last` : $\forall x$ l P ,
`filter` P ($l \& x$) = `filter` P l ++ (`If` P x `then` $x::$ `nil` `else` `nil`).

Proof using. `intros`. `rewrite` $\neg$ `filter_app`. Qed.

Lemma `filter_rev` : $\forall l$ P ,
`filter` P (`rev` l) = `rev` (`filter` P l).

Proof using.

```
intros. induction  $l$ .
- rewrite !rev_nil, filter_nil. auto.
- rewrite rev_cons, filter_cons, filter_last,  $IHL$ .
```

case_if~; rew_listx¬.

Qed.

Lemma filter_eq_self_of_mem_implies_P : $\forall l P$,

$(\forall x, \text{mem } x l \rightarrow P x) \rightarrow$
 $\text{filter } P l = l.$

Proof using.

induction l; introv M.
{ *auto.* }
{ *rewrite filter_cons. case_if.*
 { *fequals. applies IHL. introv Mx. applies× M.* }
 { *false× M.* } }

Qed.

See also *filter_of_Forall*

Lemma mem_filter_eq : $\forall x P l$,

$\text{mem } x (\text{filter } P l) = (\text{mem } x l \wedge P x).$

Proof using.

intros. extens. induction l.
{ *rewrite filter_nil. iff M (M&?); inverts M.* }
{ *rewrite filter_cons. case_if; rew_listx; rewrite IHL.*
 { *iff [M|M] N; subst×.* }
 { *iff M ([N|N]&K); subst×.* } }

Qed.

Lemma mem_filter : $\forall x P l$,

$\text{mem } x l \rightarrow$
 $P x \rightarrow$
 $\text{mem } x (\text{filter } P l).$

Proof using. *intros. rewrite× mem_filter_eq. Qed.*

Lemma mem_filter_inv : $\forall x P l$,

$\text{mem } x (\text{filter } P l) \rightarrow$
 $\text{mem } x l \wedge P x.$

Proof using. *introv E. rewrite× mem_filter_eq in E. Qed.*

Lemma length_filter : $\forall l P$,

$\text{length } (\text{filter } P l) \leq \text{length } l.$

Proof using.

intros. induction l.
{ *rewrite filter_nil. math.* }
{ *rewrite filter_cons. case_if; rew_list; math.* } }

Qed.

Lemma filter_length_two_disjoint : $\forall (P Q : A \rightarrow \text{Prop}) l$,

$(\forall x, \text{mem } x l \rightarrow P x \rightarrow Q x \rightarrow \text{False}) \rightarrow$
 $\text{length } (\text{filter } P l) + \text{length } (\text{filter } Q l) \leq \text{length } l.$

Proof using.

```
  intro. induction l; intro H.
  { rew_list. nat_math. }
  { specializes IHL. { intros. applys × H x. }
    repeat rewrite filter_cons. do 2 case_if; rew_list.
    { false × H. } { nat_math. } { nat_math. } { nat_math. } }
```

Qed.

Lemma filter_length_partition : $\forall P l$,
length (filter (fun $x \Rightarrow P x$) l)
+ length (filter (fun $x \Rightarrow \neg P x$) l)
= length l .

Proof using.

```
  intro. induction l.
  { rew_list. nat_math. }
  { repeat rewrite filter_cons. repeat case_if; rew_list; math. }
```

Qed.

Lemma length_filter_eq_mem_ge_one : $\forall x l$,
mem $x l \rightarrow$
length (filter ($= x$) l) ≥ 1 .

Proof using.

```
  intro M. induction l.
  { inverts M. }
  { rewrite filter_cons. case_if.
    { rew_list. nat_math. }
    { inverts M. false. applys  $\neg$  IHL. } }
```

Qed.

End Filter.

Global Opaque filter.

#[global]

Hint Rewrite filter_nil filter_cons filter_app filter_last filter_rev
mem_filter_eq : rew_listx.

19.6.4 Remove

Definition remove $A (a:A) (l:\text{list } A) : \text{list } A :=$
filter ($\neq a$) l .

Section Remove.

Variables ($A : \text{Type}$).

Implicit Types $a x : A$.

Implicit Types $l : \text{list } A$.

Lemma remove_as_filter : $\forall a\ l,$
 remove $a\ l = \text{filter } (\neq a)\ l.$

Proof using. auto. Qed.

Lemma remove_not_mem : $\forall x\ l,$
 $\neg \text{mem } x\ l \rightarrow$
 remove $x\ l = l.$

Proof using.

intros. *applies* filter_eq_self_of_mem_implies_P.

intros $y\ My \rightarrow$. *false*.

Qed.

Lemma mem_remove_eq : $\forall x\ a\ l,$
 $\text{mem } x\ (\text{remove } a\ l) = (\text{mem } x\ l \wedge x \neq a).$

Proof using. intros. *applies* $\times$ mem_filter_eq. Qed.

Lemma mem_remove : $\forall a\ x\ l,$
 $\text{mem } x\ l \rightarrow$
 $x \neq a \rightarrow$
 $\text{mem } x\ (\text{remove } a\ l).$

Proof using. intros. rewrite $\times$ mem_remove_eq. Qed.

Lemma mem_remove_inv : $\forall x\ a\ l,$
 $\text{mem } x\ (\text{remove } a\ l) \rightarrow$
 $\text{mem } x\ l \wedge x \neq a.$

Proof using. *introv* E . rewrite $\times$ mem_remove_eq in E . Qed.

Lemma mem_remove_same_inv : $\forall x\ l,$
 $\text{mem } x\ (\text{remove } x\ l) \rightarrow$
False.

Proof using. *introv* E . *lets*: mem_remove_inv E . *false* $\times$. Qed.

Lemma length_remove : $\forall l\ a,$
 $\text{length } (\text{remove } a\ l) \leq \text{length } l.$

Proof using. intros. *applies* length_filter. Qed.

Lemma length_remove_mem : $\forall x\ l,$
 $\text{mem } x\ l \rightarrow$
 $\text{length } (\text{remove } x\ l) < \text{length } l.$

Proof using.

unfold remove. intros $x\ l$. induction l ; *rew_list* x ; *introv* M .

{ *false*. }

{ *case_if*; *rew_list*.

{ destruct M ; *tryfalse*. *forwards*~: *IHl*. *math*. }

{ *lets*: length_filter $l\ (\neq x)$. *math*. }

Qed.

End Remove.

Global Opaque remove.

`noduplicates` L asserts that L does not contain any duplicated item.

```
Inductive noduplicates  $A : \text{list } A \rightarrow \text{Prop} :=$   
| noduplicates_nil : noduplicates nil  
| noduplicates_cons :  $\forall x\ l,$   
   $\neg (\text{mem } x\ l) \rightarrow$   
  noduplicates  $l \rightarrow$   
  noduplicates  $(x::l)$ .
```

Section Noduplicates.

Variables $(A : \text{Type})$.

Implicit Types $l : \text{list } A$.

Hint Constructors **noduplicates**.

```
Lemma noduplicates_nil_eq :  
  noduplicates  $(@nil\ A) = \text{True}$ .
```

Proof using. *extens. iff* M ; *inverts* $\times M$. Qed.

```
Lemma noduplicates_cons_eq :  $\forall x\ l,$   
  noduplicates  $(x::l) = (\neg (\text{LibList.mem } x\ l) \wedge \text{noduplicates } l)$ .
```

Proof using. *extens. iff* M ; *inverts* $\times M$. Qed.

```
Lemma noduplicates_one :  $\forall (x:A),$   
  noduplicates  $(x::nil)$ .
```

Proof using.

```
  intros. applies noduplicates_cons. { intros  $M$ . false  $\times$  mem_nil_inv. }  
  applies noduplicates_nil.
```

Qed.

```
Lemma noduplicates_two :  $\forall (x\ y:A),$   
   $x \neq y \rightarrow$   
  noduplicates  $(x::y::nil)$ .
```

Proof using.

```
  intros. applies noduplicates_cons.  
  { intros  $M$ . rewrite mem_one_eq in  $M$ . false. }  
  applies noduplicates_one.
```

Qed.

```
Lemma noduplicates_app :  $\forall\ l1\ l2,$   
  noduplicates  $l1 \rightarrow$   
  noduplicates  $l2 \rightarrow$   
   $(\forall x, \text{mem } x\ l1 \rightarrow \text{mem } x\ l2 \rightarrow \text{False}) \rightarrow$   
  noduplicates  $(l1 ++ l2)$ .
```

Proof using.

Hint Constructors **mem**.

```
  intros  $l1$ . induction  $l1$ ; introv  $N1\ N2\ EQ$ ; rew_list.
```

```

{ auto. }
{ inverts N1 as N N1'. constructors.
  { rewrite mem_app_eq. rew_logic×. }
  { applies¬ IHI1. introv Mx1 Mx2. applies× EQ x. } }

```

Qed.

Lemma noduplicates_app_inv : $\forall l1\ l2$,
noduplicates ($l1 ++ l2$) $\rightarrow$
noduplicates $l1$
 $\wedge$ **noduplicates** $l2$
 $\wedge \neg (\exists x, \text{mem } x\ l1 \wedge \text{mem } x\ l2)$.

Proof using.

```

introv ND. splits.
{ induction l1.
  { constructors. }
  { rew_list in ND. inverts ND as ND1 ND2.
    rewrite mem_app_eq in ND1. rew_logic× in ND1. } }
{ induction l1; rew_list in ND.
  { auto. }
  { inverts¬ ND. } }
{ introv (x&I1&I2). induction I1; rew_list in ND.
  { inverts ND as ND1 ND2. false ND1. apply× mem_app. }
  { apply IHI1. inverts¬ ND. } }

```

Qed.

Lemma noduplicates_length_le : $\forall l1\ l2$,
noduplicates $l1 \rightarrow$
 $(\forall x, \text{mem } x\ l1 \rightarrow \text{mem } x\ l2) \rightarrow$
 $\text{length } l1 \leq \text{length } l2$.

Proof using.

```

Hint Constructors mem.
introv NL ML. gen l2. induction l1 as [|a l1']; intros; rew_list.
{ math. }
{ inverts NL as HM NL'. sets_eq l2': (remove a l2).
  forwards H: length_remove_mem a l2. { applies× ML. }
  rewrite ← EQL2' in H.
  forwards~: IHI1' l2'.
  { introv N. tests: (x = a). subst l2'. applies× mem_remove. }
  math. }

```

Qed.

Lemma noduplicates_length_eq : $\forall l1\ l2$,
noduplicates $l1 \rightarrow$
noduplicates $l2 \rightarrow$
 $(\forall x, \text{mem } x\ l1 \leftrightarrow \text{mem } x\ l2) \rightarrow$

length $l1$ = length $l2$.

Proof using.

introv $H1$ $H2$ EQ .

forwards~: noduplicates_length_le $l1$ $l2$. { intros. rewrite $\neg \leftarrow EQ$. }

forwards~: noduplicates_length_le $l2$ $l1$. { intros. rewrite $\neg EQ$. }

math.

Qed.

Lemma noduplicates_Nth_same : $\forall l$,

($\forall n1$ $n2$ x ,
Nth $n1$ l $x \rightarrow$
Nth $n2$ l $x \rightarrow$
 $n1 = n2$) $\rightarrow$

noduplicates l .

Proof using.

introv NL . induction l ; constructors.

{ introv I . lets ($n \& N$): mem_Nth (rm I).

forwards $\times$ Ab : NL Nth_zero Nth_succ. inverts Ab . }

{ apply IHL . introv $N1$ $N2$. forwards G : NL .

applies Nth_succ $N1$. applies Nth_succ $N2$. inverts $\neg G$. }

Qed.

Lemma noduplicates_Nth_same_inv : $\forall l$ $n1$ $n2$ x ,

noduplicates $l \rightarrow$

Nth $n1$ l $x \rightarrow$

Nth $n2$ l $x \rightarrow$

$n1 = n2$.

Proof using.

introv NL . gen $n1$ $n2$. induction NL ; introv $N1$ $N2$.

{ inverts $N1$. }

{ inverts $N1$ as $N1$; inverts $N2$ as $N2$; autos $\neg$.

{ apply Nth_mem in $N2$. false $\times$. }

{ apply Nth_mem in $N1$. false $\times$. } }

Qed.

Lemma noduplicates_filter : $\forall P$ l ,

noduplicates $l \rightarrow$

noduplicates (filter P l).

Proof using.

introv H . induction H ; rew_listx. { auto. }

{ case_if.

{ constructors $\times$. rew_listx $\times$. }

{ auto. } }

Qed.

Lemma noduplicates_remove : $\forall a\ l$,
noduplicates $l \rightarrow$
noduplicates (remove $a\ l$).

Proof using.

intros. rewrite remove_as_filter. *applies* $\times$ noduplicates_filter.

Qed.

Lemma noduplicates_rev : $\forall l$,
noduplicates $l \rightarrow$
noduplicates (rev l).

Proof using.

introv M. apply $\neg$ noduplicates_Nth_same. *introv N1 N2.*

lets N1': Nth_rev_inv N1. lets N2': Nth_rev_inv N2.

lets I1: Nth_inbound N1. lets I2: Nth_inbound N2.

rewrite length_rev in *.

lets: noduplicates_Nth_same_inv M N1' N2'. math.

Qed.

End Noduplicates.

#[*global*] Hint Rewrite noduplicates_nil_eq noduplicates_cons_eq : *rew_listx*.

remove_duplicates l produces a list l' that is the sublist of l obtained by keeping only the first occurrence of every item.

Fixpoint remove_duplicates $A\ (l:\text{list } A) : \text{list } A :=$
 match l with
 | nil $\Rightarrow$ nil
 | $x::l' \Rightarrow x :: (\text{remove } x\ (\text{remove\_duplicates } l'))$
 end.

Section Remove_duplicates.

Variables ($A : \text{Type}$).

Implicit Types $l : \text{list } A$.

Hint Constructors **mem noduplicates**.

Lemma remove_duplicates_spec : $\forall l\ l'$,

$l' = \text{remove\_duplicates } l \rightarrow$

noduplicates l'

$\wedge (\forall x, \text{mem } x\ l' \leftrightarrow \text{mem } x\ l)$

$\wedge \text{length } l' \leq \text{length } l$.

Proof using.

introv E.

asserts (R1&R2): (noduplicates $l' \wedge (\forall x, \text{mem } x\ l' \leftrightarrow \text{mem } x\ l)$).

{ gen $l'\ E$. induction l ; introv E ; simpls.

{ subst. splits $\times$. }

{ sets_eq l' : (remove_duplicates l). forwards $\neg [E'\ M]$: IHL. subst l' . splits.

```

    { constructors. applys¬ mem_remove_same_inv. applys¬ noduplicates_remove. }
    { intros x. lets (M1&M2): M x. iff N.
      { inverts N as R. auto. lets: mem_remove_inv R. constructors×. }
      { lets [E|(H1&H2)]: mem_cons_inv_cases N. { subst×. }
        constructors. applys× mem_remove. } } } }
    splits¬. applys¬ noduplicates_length_le. introv Hx. rewrite¬ ← R2.
Qed.

Lemma noduplicates_remove_duplicates : ∀ l,
  noduplicates (remove_duplicates l).
Proof using. introv. forwards*: remove_duplicates_spec l. Qed.

Lemma mem_remove_duplicates : ∀ x l,
  mem x (remove_duplicates l) = mem x l.
Proof using.
  introv. extens. repeat rewrite ← Mem_mem. apply¬ remove_duplicates_spec.
Qed.

Lemma length_remove_duplicates : ∀ l,
  length (remove_duplicates l) ≤ length l.
Proof using. introv. forwards*: remove_duplicates_spec l. Qed.

End Remove_duplicates.

#[global]
Hint Rewrite mem_remove_duplicates : rew_listx.

```

19.6.5 Combine

```

Fixpoint combine A B (r:list A) (s:list B) : list (A×B) :=
  match r, s with
  | nil, nil ⇒ nil
  | a::r', b::s' ⇒ (a,b)::(combine r' s')
  | _, _ ⇒ arbitrary
  end.

```

Section Combine.

Variable (A B : Type).

Implicit Types r : list A.

Implicit Types s : list B.

Lemma combine_nil :

combine (@nil A) (@nil B) = nil.

Proof using. auto. Qed.

Lemma combine_cons : ∀ x r y s,

combine (x::r) (y::s) = (x,y)::(combine r s).

Proof using. auto. Qed.

Lemma combine_app : $\forall r1\ r2\ s1\ s2,$
 length $r1$ = length $s1 \rightarrow$
 combine $(r1++r2)\ (s1++s2) = (\text{combine } r1\ s1)++(\text{combine } r2\ s2).$
 Proof using.
 intros $r1$. induction $r1$; intro E ; destruct $s1$; tryfalse.
 { auto. }
 { rew_list in *. do 2 rewrite combine_cons. rew_list . rewrite $\neg$ IHr1. }
 Qed.

Lemma combine_last : $\forall x\ r\ y\ s,$
 length r = length $s \rightarrow$
 combine $(r\&x)\ (s\&y) = (\text{combine } r\ s)\&(x,y).$
 Proof using. intro E . applies $\neg$ combine_app. Qed.

Lemma combine_rev : $\forall r\ s,$
 length r = length $s \rightarrow$
 combine $(\text{rev } r)\ (\text{rev } s) = \text{rev } (\text{combine } r\ s).$
 Proof using.
 intros r . induction r ; intro E ; destruct s ; tryfalse.
 { auto. }
 { rew_list in *. rewrite combine_last, combine_cons.
 { rewrite IHr. rew_list . math. }
 { rew_list . math. } }
 Qed.

Lemma length_combine : $\forall r\ s,$
 length r = length $s \rightarrow$
 length $(\text{combine } r\ s) = \text{length } r.$
 Proof using.
 intros r . induction r ; intro E ; destruct s ; tryfalse.
 { auto. }
 { rewrite combine_cons. rew_list . }
 Qed.

End Combine.

Lemma Nth_combine : $\forall A\ B\ n\ (r:\text{list } A)\ (s:\text{list } B)\ x\ y,$
 Nth $n\ r\ x \rightarrow$
 Nth $n\ s\ y \rightarrow$
 Nth $n\ (\text{combine } r\ s)\ (x,y).$

Proof using.
 intro $N1$. gen s . induction $N1$; intro $N2$; inverts $N2$.
 { constructors. }
 { rewrite combine_cons. constructors. }
 Qed.

Lemma nth_combine : $\forall \{IA:\text{Inhab } A\} \{IB:\text{Inhab } B\} n\ (r:\text{list } A)\ (s:\text{list } B),$


```

  n < length r →
  length r = length s →
  nth n (combine r s) = (nth n r, nth n s).

```

Proof using.

```

  introv N E. applys nth_of_Nth. applys × Nth_combine.
  { applys × Nth_nth. } { applys × Nth_nth. math. }

```

Qed.

Global Opaque combine.

#[global]

Hint Rewrite combine_nil combine_cons : rew_listx.

19.6.6 Split

```

Fixpoint split A B (l:list(A×B)) : (list A × list B) :=
  match l with
  | nil ⇒ (nil, nil)
  | (a,b)::l' ⇒ let (la,lb) := split l' in (a::la, b::lb)
  end.

```

Section Split.

Variable (A B : Type).

Implicit Types l : list (A×B).

Implicit Types x : A.

Implicit Types y : B.

Implicit Types r : list A.

Implicit Types s : list B.

Lemma split_nil :

```

  split (@nil (A×B)) = (nil, nil).

```

Proof using. auto. Qed.

Lemma split_cons_let : ∀ x y l,

```

  split ((x,y)::l) = let '(r,s) := split l in (x::r, y::s).

```

Proof using. auto. Qed.

Lemma split_cons : ∀ x y l r s,

```

  (r,s) = split l →
  split ((x,y)::l) = (x::r, y::s).

```

Proof using.

```

  introv H. rewrite split_cons_let. rewrite ← H.

```

Qed.

Lemma split_app : ∀ l1 l2 r1 r2 s1 s2,

```

  (r1,s1) = split l1 →
  (r2,s2) = split l2 →

```

`split (l1++l2) = (r1++r2, s1++s2).`

Proof using.

```
intros l1. induction l1 as [[x y] l1']; introv H1 H2.
{ rewrite split_nil in H1. inverts¬ H1. }
{ rewrite split_cons_let in H1. destruct (split l1') as [r1' s1'].
  inverts H1. rew_list. rewrite split_cons_let.
  erewrite¬ (IHl1' l2). }
```

Qed.

Lemma `split_last` : $\forall x y l r s,$

$(r, s) = \text{split } l \rightarrow$
 $\text{split } (l \& (x, y)) = (r \& x, s \& y).$

Proof using. *introv H. erewrite split_app; fequals*. Qed.

Lemma `split_length` : $\forall l r s,$

$(r, s) = \text{split } l \rightarrow$
 $\text{length } r = \text{length } l \wedge \text{length } s = \text{length } l.$

Proof using.

```
intros l. induction l as [[x y] l']; introv E.
{ rewrite split_nil in E. inverts¬ E. }
{ rewrite split_cons_let in E. destruct (split l') as [r' s'].
  inverts E. rew_list. forwards¬ (?&?): IHl'. }
```

Qed.

Lemma `split_length_l` : $\forall l r s,$

$(r, s) = \text{split } l \rightarrow$
 $\text{length } r = \text{length } l.$

Proof using. *introv E. forwards**: `split_length E`. Qed.

Lemma `split_length_r` : $\forall l r s,$

$(r, s) = \text{split } l \rightarrow$
 $\text{length } s = \text{length } l.$

Proof using. *introv E. forwards**: `split_length E`. Qed.

Lemma `Nth_split` : $\forall n l x y r s,$

Nth $n l (x, y) \rightarrow$
 $(r, s) = \text{split } l \rightarrow$
Nth $n r x \wedge \text{Nth } n s y.$

Proof using.

Hint Constructors **Nth**.

```
introv N E. gen_eq p: (x, y). gen r s x y.
induction N as [l' [x' y']|[x' y'] n' l' [x'' y'']]; intros.
{ inverts EQp. rewrite split_cons_let in E.
  destruct (split l') as [r' s']. inverts× E. }
{ inverts EQp. rewrite split_cons_let in E.
  destruct (split l') as [r' s'].
```

$\text{forwards}^*: \text{IHN}. \text{inverts} \times E. \}$
 Qed.

Lemma nth_split : $\forall \{IA:\text{Inhab } A\} \{IB:\text{Inhab } B\} n \, l \, (r:\text{list } A) \, (s:\text{list } B),$
 $(r, s) = \text{split } l \rightarrow$
 $n < \text{length } l \rightarrow$
 $\text{nth } n \, l = (\text{nth } n \, r, \text{nth } n \, s).$

Proof using.

$\text{introv } E \, N. \text{applies } \text{nth\_of\_Nth}. \text{lets } ([x \, y] \& M): \text{Nth\_inbound\_inv } N.$
 $\text{forwards } (F1 \& F2): \text{Nth\_split } M \, E.$
 $\text{rewrite } (\text{nth\_of\_Nth } F1). \text{rewrite} \neg (\text{nth\_of\_Nth } F2).$

Qed.

Lemma nth_split_l : $\forall \{IA:\text{Inhab } A\} \{IB:\text{Inhab } B\} n \, l \, (r:\text{list } A) \, (s:\text{list } B),$
 $(r, s) = \text{split } l \rightarrow$
 $n < \text{length } l \rightarrow$
 $\text{nth } n \, r = \text{fst } (\text{nth } n \, l).$

Proof using. $\text{introv } E \, N. \text{rewrites} \neg (\gg \text{nth\_split } E \, N).$ Qed.

Lemma nth_split_r : $\forall \{IA:\text{Inhab } A\} \{IB:\text{Inhab } B\} n \, l \, (r:\text{list } A) \, (s:\text{list } B),$
 $(r, s) = \text{split } l \rightarrow$
 $n < \text{length } l \rightarrow$
 $\text{nth } n \, s = \text{snd } (\text{nth } n \, l).$

Proof using. $\text{introv } E \, N. \text{rewrites} \neg (\gg \text{nth\_split } E \, N).$ Qed.

End Split.

Global Opaque split.

$\#[\text{global}]$

Hint Rewrite split_nil : $\text{rew\_listx}.$

19.6.7 Take

Fixpoint take $A \, (n:\text{nat}) \, (l:\text{list } A) : \text{list } A :=$
 $\text{match } n \text{ with}$
 $| 0 \Rightarrow \text{nil}$
 $| S \, n' \Rightarrow \text{match } l \text{ with}$
 $| \text{nil} \Rightarrow \text{nil}$
 $| a :: l' \Rightarrow a :: (\text{take } n' \, l')$
 end
 $\text{end}.$

Section Take.

Variables $(A : \text{Type}).$

Implicit Types $n : \text{nat}.$

Implicit Types $x : A.$

```

Implicit Types  $l : \text{list } A$ .

Lemma take_nil :  $\forall n$ ,
  take  $n$  (@nil  $A$ ) = nil.
Proof using. intros. destruct  $n$ ; auto. Qed.

Lemma take_zero :  $\forall l$ ,
  take 0  $l$  = nil.
Proof using. auto. Qed.

Lemma take_succ :  $\forall x \ l \ n$ ,
  take (S  $n$ ) ( $x :: l$ ) =  $x ::$  (take  $n \ l$ ).
Proof using. auto. Qed.

Definition take_cons := take_succ.

Lemma take_cons_pos :  $\forall x \ l \ n$ ,
  ( $n > 0$ )  $\rightarrow$ 
  take  $n$  ( $x :: l$ ) =  $x ::$  (take ( $n-1$ )  $l$ ).
Proof using.
  introv  $H$ . destruct  $n$ . false; math.
  rewrite take_cons. fequals-rec. math.
Qed.

Lemma take_ge :  $\forall n \ l$ ,
  ( $n \geq \text{length } l$ )  $\rightarrow$ 
  take  $n \ l$  =  $l$ .
Proof using.
  induction  $n$ ; destruct  $l$ ; introv  $H$ ; rew_list in *; auto.
  { false. math. }
  { rewrite take_cons. fequals. applies IHn. math. }
Qed.

Lemma take_is_prefix :  $\forall n \ l$ ,
   $\exists q, l = \text{take } n \ l ++ q$ .
Proof using.
  intros  $n \ l$ . gen  $n$ . induction  $l$ ; intros.
  {  $\exists$  (@nil  $A$ ). rewrite  $\neg$  take_nil. }
  { destruct  $n$  as [ $n'$ ].
    { rewrite take_zero.  $\exists \neg (a :: l)$ . }
    { rewrite take_succ. forwards ( $q' \& E$ ): IHL  $n'$ .  $\exists q'$ .
      rew_list. congruence. } }
Qed.

Lemma take_app_l :  $\forall n \ l \ l'$ ,
  ( $n \leq \text{length } l$ )  $\rightarrow$ 
  take  $n$  ( $l ++ l'$ ) = take  $n \ l$ .
Proof using.

```

```

induction n; destruct l; introv H; rew_list in *; auto.
{ false. math. }
{ do 2 rewrite take_cons. fequals. applies IHn. math. }
Qed.

Lemma take_app_r :  $\forall n\ l\ l'$ ,
  ( $n \geq \text{length } l$ )  $\rightarrow$ 
  take  $n\ (l ++ l') = l ++ \text{take } (n - \text{length } l)\ l'$ .
Proof using.
  intros. gen n. induction l; introv H.
  { rewrite length_nil in *. do 2 rewrite app_nil_l.
    fequals. math. }
  { rewrite length_cons in *. destruct n as [|n'].
    { false. math. }
    { rew_list. rewrite take_cons. fequals. applies IHL. math. } }
Qed.

Lemma take_prefix_length :  $\forall l\ l'$ ,
  take (length l) (l ++ l') = l.
Proof using.
  intros. rewrite take_app_r; [|math].
  math_rewrite ( $\forall a, a - a = 0$ ).
  rewrite take_zero. rew_list $\neg$ .
Qed.

Lemma take_full_length :  $\forall l$ ,
  take (length l) l = l.
Proof using.
  intros. lets H: (@take_prefix_length l nil). rew_list $\neg$  in H.
Qed.

End Take.

Global Opaque take.

#[global]
Hint Rewrite take_nil take_zero take_succ : rew_listx.

```

19.6.8 Drop

```

Fixpoint drop A (n:nat) (l:list A) : list A :=
  match n with
  | 0  $\Rightarrow$  l
  | S n'  $\Rightarrow$  match l with
  | nil  $\Rightarrow$  nil
  | a::l'  $\Rightarrow$  drop n' l'

```

```

    end
  end.

Section Drop.
Variables (A : Type).
Implicit Types n : nat.
Implicit Types x : A.
Implicit Types l : list A.

Lemma drop_nil : ∀ n,
  drop n (@nil A) = nil.
Proof using. intros. destruct n; auto. Qed.

Lemma drop_zero : ∀ l,
  drop 0 l = l.
Proof using. auto. Qed.

Lemma drop_succ : ∀ x l n,
  drop (S n) (x::l) = (drop n l).
Proof using. auto. Qed.

Definition drop_cons := drop_succ.

Lemma drop_cons_pos : ∀ x l n,
  (n > 0) →
  drop n (x::l) = drop (n-1) l.
Proof using.
  introv H. destruct n. false; math.
  rewrite drop_cons. fequals_rec. math.
Qed.

Lemma drop_is_suffix : ∀ n l,
  ∃ q, l = q ++ drop n l.
Proof using.
  intros n l. gen n. induction l; intros.
  { ∃ (@nil A). rewrite ⊢ drop_nil. }
  { destruct n as [|n'].
    { rewrite drop_zero. ∃ ⊢ (@nil A). }
    { rewrite drop_succ. forwards (q'&E): IHL n'. ∃ (a::q').
      rew_list. congruence. } }
Qed.

Lemma drop_app_l : ∀ n l l',
  (n ≤ length l) →
  drop n (l ++ l') = drop n l ++ l'.
Proof using.
  induction n; destruct l; introv H; rew_list in *; auto.
  { false. math. }

```

```

    { do 2 rewrite drop_cons. fequals. applys IHn. math. }
Qed.

```

```

Lemma drop_app_r : ∀ n l l',
  (n ≥ length l) →
  drop n (l ++ l') = drop (n - length l) l'.

```

Proof using.

```

  induction n; destruct l; introv H; rew_list in *; auto.
  { false. math. }
  { rewrite drop_cons. rewrite IHn. fequals. math. }

```

Qed.

```

Lemma drop_app_length : ∀ l l',
  drop (length l) (l ++ l') = l'.

```

Proof using.

```

  intros. rewrite drop_app_r; [|math].
  math_rewrite (∀ a, a - a = 0).
  rewrite drop_zero. rew_list¬.

```

Qed.

```

Lemma drop_at_length : ∀ l,
  drop (length l) l = nil.

```

Proof using.

```

  intros. lets H: (@drop_app_length l nil). rew_list¬ in H.

```

Qed.

```

Lemma drop_drop : ∀ n m l,
  drop n (drop m l) = drop (n+m) l.

```

Proof using.

```

  intros n. induction m; simpl; intros.
  { rew_nat×. }
  { destruct l as [|x l'].
    { repeat rewrite drop_nil. auto. }
    { rewrite drop_cons_pos; try math. rew_nat×. } }

```

Qed.

End Drop.

Global Opaque drop.

#[global]

Hint Rewrite drop_nil drop_zero drop_succ : rew_listx.

19.6.9 Take and drop decomposition of a list

Section TakeAndDrop.

Variables (A : Type).

Implicit Types $x : A$.

Implicit Types $l : \text{list } A$.

Lemma take_app_drop_spec : $\forall n l f r$,

$f = \text{take } n l \rightarrow$

$r = \text{drop } n l \rightarrow$

$n \leq \text{length } l \rightarrow$

$l = f ++ r$

$\wedge \text{length } f = n$

$\wedge \text{length } r = \text{length } l - n$.

Proof using.

intros n . induction n ; intro $F R L$.

{ subst. rew_listx. splits. math. }

{ destruct l ; rew_listx in L .

{ rew_listx in L . false. math. }

{ forwards. (F' & R' & L'): ($\gg$ IHn l (take $n l$) r). { math. }

subst f . rew_listx. splits. { fequals. } { math. } { math. } }

Qed.

Lemma take_spec : $\forall n l$,

$n \leq \text{length } l \rightarrow$

$\exists l', \text{length } (\text{take } n l) = n$

$\wedge l = (\text{take } n l) ++ l'$.

Proof using. intro E . forwards $\times (E1 \& E2 \& E3)$: take_app_drop_spec. Qed.

Lemma length_take : $\forall n l$,

$n \leq \text{length } l \rightarrow$

$\text{length } (\text{take } n l) = n$.

Proof using. intro E . forwards. (l' & N & M): take_spec $n l$. Qed.

Lemma drop_spec : $\forall n l$,

$n \leq \text{length } l \rightarrow$

$\exists l', \text{length } l' = n$

$\wedge l = l' ++ (\text{drop } n l)$.

Proof using. intro E . forwards $\times (E1 \& E2 \& E3)$: take_app_drop_spec. Qed.

Lemma length_drop : $\forall n l$,

$n \leq \text{length } l \rightarrow$

$\text{length } (\text{drop } n l) = \text{length } l - n$.

Proof using.

intro E . forwards. (l' & N & M): drop_spec $n l$.

pattern l at 2. rewrite M . rew_list. math.

Qed.

Lemma list_eq_take_app_drop : $\forall n l$,

$n \leq \text{length } l \rightarrow$

$\text{take } n l ++ \text{drop } n l = l$.

Proof using. *intros H. forwards**: take_app_drop_spec *n l*. Qed.

End TakeAndDrop.

Arguments take_app_drop_spec [A].

Arguments take_spec [A].

Arguments drop_spec [A].

19.6.10 TakeDropLast

take_drop_last *l* returns a pair (*q,x*) such that *l* = *q* & *x*

```
Fixpoint take_drop_last '{IA:Inhab A} (l:list A) : (list A)*A :=
  match l with
  | nil => arbitrary
  | x::l' =>
    match l' with
    | nil => (nil,x)
    | _ => let (t,y) := take_drop_last l' in
           (x::t, y)
    end
  end.
```

Section TakeDropLast.

Context (A:Type) {IA:Inhab A}.

Implicit Types *x* : A.

Implicit Types *l* : list A.

```
Lemma take_drop_last_cons : ∀ (x:A) (l: list A),
  take_drop_last (x::l) =
    match l with
    | nil => (nil, x)
    | _::- => let (t, y) := take_drop_last l in (x :: t, y)
    end.
```

Proof using. auto. Qed.

```
Lemma take_drop_last_spec : ∀ (x:A) (l l': list A),
  (l',x) = take_drop_last l →
  l ≠ nil →
  l = l' & x.
```

Proof using.

```
  induction l as [|a t]; intros E N. false.
  rewrite take_drop_last_cons in E.
  destruct (take_drop_last t) as [u r].
  { destruct t; inverts E. rewrite× app_nil_l.
    rew_list. fequals. applys IHt; auto_false×. }
```

Qed.

Lemma take_drop_last_length : $\forall l l' x,$
 $(l', x) = \text{take\_drop\_last } l \rightarrow$
 $l \neq \text{nil} \rightarrow$
 $\text{length } l' = \text{length } l - 1.$

Proof using.

intros E N. forwards: take_drop_last_spec *E N.*
subst *l*. rewrite length_last. *math.*

Qed.

End TakeDropLast.

Global Opaque take_drop_last.

Arguments take_drop_last [*A*] {*IA*}.

Arguments take_drop_last_spec [*A*] {*IA*}.

Arguments take_drop_last_length [*A*] {*IA*}.

Forall *P* *l* asserts that all the elements in the list *l* satisfy the predicate *P*.

Inductive Forall *A* (*P*:*A*→Prop) : list *A* → Prop :=

| Forall_nil :
Forall *P* nil
| Forall_cons : $\forall l x,$
 $P x \rightarrow$
Forall *P* *l* →
Forall *P* (*x*::*l*).

Section ForallProp.

Variables *A* : Type.

Implicit Types *l* : list *A*.

Implicit Types *P* : *A*→Prop.

Hint Constructors Forall.

Lemma Forall_nil_eq : $\forall P,$

Forall *P* nil = **True**.

Proof using. auto. Qed.

Lemma Forall_cons_eq : $\forall P l x,$

Forall *P* (*x*::*l*) = (*P* *x* ∧ **Forall** *P* *l*).

Proof using.

intros. extens. iff M. { inverts × M. } { constructors ×. }

Qed.

Lemma Forall_one_eq : $\forall P x,$

Forall *P* (*x*::nil) = (*P* *x*).

Proof using.

intros. extens. rewrite Forall_cons_eq. rewrite × Forall_nil_eq.

Qed.

Lemma Forall_app_eq : $\forall P\ l1\ l2,$
Forall $P\ (l1\ ++\ l2) = (\text{Forall } P\ l1 \wedge \text{Forall } P\ l2).$
Proof using.
intros. *extens*. induction $l1$; *rew_list*.
{ *autos*×. }
{ *iff* $M\ (M1\ \&\ M2).$ { *inverts*× $M.$ } { *inverts*× $M1.$ } }
Qed.

Lemma Forall_last_eq : $\forall P\ l\ x,$
Forall $P\ (l\ \&\ x) = (\text{Forall } P\ l \wedge P\ x).$
Proof using.
intros. *extens*. induction l ; *rew_list*.
{ *iff* $M\ (M1\ \&\ M2).$ { *inverts*× $M.$ } { *autos*×. } }
{ *iff* $M\ (M1\ \&\ M2).$ { *inverts*× $M.$ } { *inverts*× $M1.$ } }
Qed.

Lemma Forall_app : $\forall P\ l1\ l2,$
Forall $P\ l1 \rightarrow$
Forall $P\ l2 \rightarrow$
Forall $P\ (l1\ ++\ l2).$
Proof using. intros. *rewrite*× Forall_app_eq. Qed.

Lemma Forall_last : $\forall P\ l\ x,$
Forall $P\ l \rightarrow$
 $P\ x \rightarrow$
Forall $P\ (l\ \&\ x).$
Proof using. intros. *rewrite*× Forall_last_eq. Qed.

Lemma Forall_cons_inv : $\forall P\ l\ x,$
Forall $P\ (x::l) \rightarrow$
 $P\ x \wedge \text{Forall } P\ l.$
Proof using. *introv* H . *rewrite*× Forall_cons_eq in H . Qed.

Lemma Forall_cons_inv_head : $\forall P\ l\ x,$
Forall $P\ (x::l) \rightarrow$
 $P\ x.$
Proof using. *introv* H . *rewrite*× Forall_cons_eq in H . Qed.

Lemma Forall_cons_inv_tail : $\forall P\ l\ x,$
Forall $P\ (x::l) \rightarrow$
Forall $P\ l.$
Proof using. *introv* H . *rewrite*× Forall_cons_eq in H . Qed.

Lemma Forall_app_inv : $\forall P\ l1\ l2,$
Forall $P\ (l1\ ++\ l2) \rightarrow$
Forall $P\ l1 \wedge \text{Forall } P\ l2.$
Proof using. *introv* H . *rewrite*× Forall_app_eq in H . Qed.

Lemma Forall_last_inv : $\forall P\ l\ x$,

Forall $P\ (l\ \&\ x) \rightarrow$

Forall $P\ l \wedge P\ x$.

Proof using. *intros* H . *rewrite*× Forall_last_eq in H . Qed.

Lemma Forall_make : $\forall n\ v\ P$,

$P\ v \rightarrow$

Forall $P\ (\text{make } n\ v)$.

Proof using.

intros N . *induction* n as $[[n']$.

{ *rewrite* make_zero. *applies*× Forall_nil. }

{ *rewrite* make_succ. *applies*× Forall_cons. }

Qed.

Lemma Forall_eq_forall_mem : $\forall P\ l$,

Forall $P\ l = (\forall x, \text{mem } x\ l \rightarrow P\ x)$.

Proof using.

extens. intros. induction l ; *iff* I .

{ *intros* IN . *inverts* IN . }

{ *auto*. }

{ *intros* IN . *rew_listx* in IN . *inverts* I . *destruct* IN ; *subst*×. }

{ *constructors*. }

{ *apply* I . *rew_listx*×. }

{ *apply*¬ IHL . *intros* IN . *apply*¬ I . *rew_listx*×. } }

Qed.

Lemma Forall_mem_inv : $\forall P\ l\ x$,

Forall $P\ l \rightarrow$

mem $x\ l \rightarrow$

$P\ x$.

Proof using. *intros* $F\ I$. *rewrite* Forall_eq_forall_mem in F . *apply*¬ F . Qed.

Lemma Forall_Nth_inv : $\forall P\ n\ l\ x$,

Forall $P\ l \rightarrow$

Nth $n\ l\ x \rightarrow$

$P\ x$.

Proof using. *intros* $F\ N$. *applies*× Forall_mem_inv F . *applies*× Nth_mem. Qed.

Lemma Forall_nth_inv : $\forall \{IA:\text{Inhab } A\}\ P\ n\ l$,

Forall $P\ l \rightarrow$

$n < \text{length } l \rightarrow$

$P\ (\text{nth } n\ l)$.

Proof using. *intros* $F\ N$. *applies* Forall_Nth_inv $n\ F$. *applies*¬ Nth_nth. Qed.

Lemma Forall_rev : $\forall P\ l$,

Forall $P\ l \rightarrow$

Forall $P\ (\text{rev } l)$.

Proof using.

```
  introv E. induction l.
  { auto. }
  { rew_list. rewrite Forall_last_eq. inverts × E. }
```

Qed.

Lemma Forall_rev_eq : $\forall P l$,
Forall P (rev l) = **Forall** $P l$.

Proof using.

```
  intros. extens. iff M.
  { rewrite ← rev_rev. applies → Forall_rev. }
  { applies → Forall_rev. }
```

Qed.

Lemma Forall_map_eq : $\forall (B:Type) (P:B \rightarrow Prop) l (f:A \rightarrow B)$,
Forall P (map $f l$) = **Forall** (fun $x \Rightarrow P (f x)$) l .

Proof using.

```
  intros. extens. induction l; intros; rew_listx.
  { autos ×. }
  { iff M; inverts × M. }
```

Qed.

Lemma Forall_take : $\forall P n l$,
Forall $P l \rightarrow$
Forall P (take $n l$).

Proof using.

```
  introv E. forwards (q&K): take_is_prefix n l.
  rewrite K in E. rewrite × Forall_app_eq in E.
```

Qed.

Lemma Forall_drop : $\forall P n l$,
Forall $P l \rightarrow$
Forall P (drop $n l$).

Proof using.

```
  introv E. forwards (q&K): drop_is_suffix n l.
  rewrite K in E. rewrite × Forall_app_eq in E.
```

Qed.

Lemma Forall_pred_incl : $\forall P Q l$,
Forall $P l \rightarrow$
pred_incl $P Q \rightarrow$
Forall $Q l$.

Proof using. introv. induction l; introv H L; inverts × H. Qed.

Lemma filter_eq_self_of_Forall : $\forall l P$,
Forall $P l \rightarrow$
filter $P l = l$.

Proof using.

intros M . *rewrite* Forall_eq_forall_mem in M .

applies $\times$ filter_eq_self_of_mem_implies_P.

Qed.

Lemma Forall_filter_same : $\forall P l$,

Forall P (filter $P l$).

Proof using.

intros. *induction* l .

{ *rewrite* filter_nil. *constructors* $\neg$. }

{ *rewrite* filter_cons. *cases_if* $\neg$. }

Qed.

Lemma Forall_filter_pred_incl : $\forall P Q l$,

pred_incl $P Q \rightarrow$

Forall Q (filter $P l$).

Proof using.

intros E . *applies* $\neg$ Forall_pred_incl P . *applies* Forall_filter_same.

Qed.

End ForallProp.

#[*global*]

Hint Rewrite Forall_nil_eq Forall_cons_eq Forall_app_eq Forall_last_eq

Forall_rev_eq : *rew-listx*.

Forall2 $P L1 L2$ asserts that the lists $L1$ and $L2$ have the same length and that elements at corresponding indices are related by the binary relation P .

Inductive **Forall2** $A B (P:A \rightarrow B \rightarrow \text{Prop}) : \text{list } A \rightarrow \text{list } B \rightarrow \text{Prop} :=$

| Forall2_nil :

Forall2 P nil nil

| Forall2_cons : $\forall s r x y$,

$P x y \rightarrow$

Forall2 $P s r \rightarrow$

Forall2 $P (x::s) (y::r)$.

Section Forall2.

Variables $A B : \text{Type}$.

Implicit Types $x : A$.

Implicit Types $y : B$.

Implicit Types $r : \text{list } A$.

Implicit Types $s : \text{list } B$.

Implicit Types $P : A \rightarrow B \rightarrow \text{Prop}$.

Hint Constructors **Forall2**.

Lemma Forall2_inv_length : $\forall P r s$,

Forall2 $P r s \rightarrow$

$\text{length } r = \text{length } s.$
 Proof using. *intros H. induction H; rew_list; math. Qed.*

Lemma Forall2_app : $\forall P \ r1 \ s1 \ r2 \ s2,$
 Forall2 $P \ r1 \ s1 \rightarrow$
 Forall2 $P \ r2 \ s2 \rightarrow$
 Forall2 $P \ (r1 ++ r2) \ (s1 ++ s2).$
 Proof using. *intros H H'. induction H; rew_list. Qed.*

Lemma Forall2_last : $\forall P \ r \ s \ x \ y,$
 Forall2 $P \ r \ s \rightarrow$
 $P \ x \ y \rightarrow$
 Forall2 $P \ (r \& x) \ (s \& y).$
 Proof using. *intros. apply Forall2_app. Qed.*

Lemma Forall2_nil_eq : $\forall P,$
 Forall2 $P \ \text{nil} \ \text{nil} = \text{True}.$
 Proof using. *intros. extens. iff M. { inverts $\times M.$ } { auto. } Qed.*

Lemma Forall2_cons_eq : $\forall P \ x \ y \ r \ s,$
 Forall2 $P \ (x::r) \ (y::s) = (P \ x \ y \wedge \text{Forall2 } P \ r \ s).$
 Proof using. *intros. extens. iff M (M1&M2). { inverts $\times M.$ } { auto. } Qed.*

Lemma Forall2_one_eq : $\forall P \ x \ y,$
 Forall2 $P \ (x::\text{nil}) \ (y::\text{nil}) = P \ x \ y.$
 Proof using. *intros. extens. rewrite Forall2_cons_eq. rewrite $\times$ Forall2_nil_eq. Qed.*

Lemma Forall2_app_eq : $\forall r1 \ r2 \ s1 \ s2 \ P,$
 $\text{length } r1 = \text{length } s1 \rightarrow$
 Forall2 $P \ (r1 ++ r2) \ (s1 ++ s2) = (\text{Forall2 } P \ r1 \ s1 \wedge \text{Forall2 } P \ r2 \ s2).$
 Proof using.
 intros r1. induction r1; intros E;
 *destruct s1; tryfalse; rew_list in *; extens.*
 { autos $\times.$ }
 { iff M (M1&M2).
 { inverts M as N1 N2. rewrite $\times$ IHr1 in N2. }
 { inverts M1. constructors. rewrite $\times$ IHr1. } }
 Qed.

Lemma Forall2_last_eq : $\forall P \ r \ s \ x \ y,$
 Forall2 $P \ (r \& x) \ (s \& y) = (\text{Forall2 } P \ r \ s \wedge P \ x \ y).$
 Proof using.
 intros. extens. iff M (M1&M2).
 { lets E: Forall2_inv_length M. rew_list in E.
 rewrite Forall2_app_eq in M. destruct M as (M1&M2). inverts $\times M2.$ }
 { applies Forall2_last. }
 Qed.

Lemma Forall2_nil_l_inv : $\forall P s$,

Forall2 P nil $s \rightarrow$
 $s = \text{nil}$.

Proof using. *intros E. inverts* $\times$ *E. Qed.*

Lemma Forall2_nil_r_inv : $\forall P r$,

Forall2 P r nil $\rightarrow$
 $r = \text{nil}$.

Proof using. *intros E. inverts* $\times$ *E. Qed.*

Lemma Forall2_cons_inv : $\forall P r s x y$,

Forall2 $P (x::r) (y::s) \rightarrow$
 $P x y \wedge \text{Forall2 } P r s$.

Proof using. *intros E. rewrite* $\neg$ *Forall2_cons_eq in E. Qed.*

Lemma Forall2_cons_l_inv : $\forall P r1 s x$,

Forall2 $P (x::r1) s \rightarrow$
 $\exists y s1, s = y::s1 \wedge P x y \wedge \text{Forall2 } P r1 s1$.

Proof using. *intros E. destruct s as [|y s1]; inverts E. autos* $\times$ *. Qed.*

Lemma Forall2_cons_r_inv : $\forall P r s1 y$,

Forall2 $P r (y::s1) \rightarrow$
 $\exists x r1, r = x::r1 \wedge P x y \wedge \text{Forall2 } P r1 s1$.

Proof using. *intros E. destruct r as [|x r1]; inverts E. autos* $\times$ *. Qed.*

Lemma Forall2_app_inv : $\forall P r1 s1 r2 s2$,

Forall2 $P (r1 ++ r2) (s1 ++ s2) \rightarrow$
 $\text{length } r1 = \text{length } s1 \rightarrow$
Forall2 $P r1 s1 \wedge \text{Forall2 } P r2 s2$.

Proof using. *intros M E. rewrite* $\neg$ *Forall2_app_eq in M. Qed.*

Lemma Forall2_last_inv : $\forall P r s x y$,

Forall2 $P (r \& x) (s \& y) \rightarrow$
Forall2 $P r s \wedge P x y$.

Proof using. *intros E. rewrite* $\neg$ *Forall2_last_eq in E. Qed.*

Lemma Forall2_last_l_inv : $\forall P r1 s x$,

Forall2 $P (r1 \& x) s \rightarrow$
 $\exists s1 y, s = s1 \& y \wedge P x y \wedge \text{Forall2 } P r1 s1$.

Proof using.

intros M. forwards (y&s1&E): list_neq_nil_inv_last s.
{ intro_subst. inverts M. applies $\times$ *nil_eq_last_inv. }*
subst s. rewrite $\times$ *Forall2_last_eq in M.*

Qed.

Lemma Forall2_last_r_inv : $\forall P r s1 y$,

Forall2 $P r (s1 \& y) \rightarrow$
 $\exists r1 x, r = r1 \& x \wedge P x y \wedge \text{Forall2 } P r1 s1$.

Proof using.

```
introv M. forwards (x&r1&E): list_neq_nil_inv_last r.  
{ intro_subst. inverts M. applies× nil_eq_last_inv. }  
subst r. rewrite× Forall2_last_eq in M.
```

Qed.

Lemma Forall2_swap : $\forall P r s$,

Forall2 $P r s \rightarrow$

Forall2 (fun $b a \Rightarrow P a b$) $s r$.

Proof using. *introv F. induction $\neg F$. Qed.*

Lemma Forall2_swap_inv : $\forall P r s$,

Forall2 (fun $b a \Rightarrow P a b$) $s r \rightarrow$

Forall2 $P r s$.

Proof using. *introv F. induction $\neg F$. Qed.*

Lemma Forall2_swap_eq : $\forall P r s$,

Forall2 $P r s = \mathbf{Forall2}$ (fun $b a \Rightarrow P a b$) $s r$.

Proof using.

```
intros. extens. iff M.  
{ applies× Forall2_swap. }  
{ applies× Forall2_swap_inv. }
```

Qed.

Lemma Forall2_rel_le : $\forall P Q r s$,

Forall2 $P r s \rightarrow$

rel_incl $P Q \rightarrow$

Forall2 $Q r s$.

Proof using.

introv F W. unfolds rel_incl, pred_incl. induction F; constructors $\neg$.

Qed.

Lemma Forall2_rev : $\forall P r s$,

Forall2 $P r s \rightarrow$

Forall2 P (rev r) (rev s).

Proof using.

Hint Resolve Forall2_last.

```
intros P r. induction r; introv M; inverts M; rew_listx  $\neg$ .
```

Qed.

Lemma Forall2_rev_eq : $\forall P r s$,

Forall2 $P r s = \mathbf{Forall2}$ P (rev r) (rev s).

Proof using.

```
intros. extens. iff M.  
{ applies  $\neg$  Forall2_rev. }  
{ rewrite  $\leftarrow$  (rev_rev  $r$ ). rewrite  $\leftarrow$  (rev_rev  $s$ ). applies  $\neg$  Forall2_rev. }
```

Qed.

Lemma Forall2_take : $\forall P\ n\ r\ s,$
 Forall2 $P\ r\ s \rightarrow$
 Forall2 $P\ (\text{take } n\ r)\ (\text{take } n\ s).$
 Proof using. intros $P\ n.$ induction $n;$ introv $H;$ inverts $H;$ rew_listx $\neg$. Qed.

Lemma Forall2_drop : $\forall P\ n\ r\ s,$
 Forall2 $P\ r\ s \rightarrow$
 Forall2 $P\ (\text{drop } n\ r)\ (\text{drop } n\ s).$
 Proof using. intros $P\ n.$ induction $n;$ introv $H;$ inverts $H;$ rew_listx $\neg$. Qed.

Lemma Forall2_map_l : $\forall f\ P\ l,$
 $(\forall y, P\ (f\ y)\ y) \rightarrow$
 Forall2 $P\ (\text{map } f\ l)\ l.$
 Proof using. introv $I.$ induction $l;$ rew_listx $\neg$. Qed.

Lemma Forall2_map_r : $\forall f\ P\ l,$
 $(\forall x, P\ x\ (f\ x)) \rightarrow$
 Forall2 $P\ l\ (\text{map } f\ l).$
 Proof using. introv $I.$ induction $l;$ rew_listx $\neg$. Qed.

Lemma Forall2_Nth_inv : $\forall P\ r\ s\ n\ x\ y,$
 Forall2 $P\ r\ s \rightarrow$
 Nth $n\ r\ x \rightarrow$
 Nth $n\ s\ y \rightarrow$
 $P\ x\ y.$
 Proof using.
 introv $F\ N1\ N2.$ gen $n.$
 induction $\neg\ F;$ introv $N1\ N2;$ inverts $N1;$ inverts $\times\ N2.$
 Qed.

Lemma Forall2_nth_inv : $\forall \{IA:\text{Inhab } A\} \{IB:\text{Inhab } B\} P\ r\ s\ n,$
 Forall2 $P\ r\ s \rightarrow$
 $n < \text{length } r \rightarrow$
 $P\ (\text{nth } n\ r)\ (\text{nth } n\ s).$
 Proof using.
 introv $F\ N.$ forwards: Forall2_inv_length $F.$
 applies Forall2_Nth_inv $n\ F;$ applies Nth_nth; math.
 Qed.

Lemma Forall2_of_forall_nth : $\forall \{IA:\text{Inhab } A\} \{IB:\text{Inhab } B\} P\ r\ s,$
 $(\forall n, n < \text{length } r \rightarrow P\ (\text{nth } n\ r)\ (\text{nth } n\ s)) \rightarrow$
 $\text{length } r = \text{length } s \rightarrow$
 Forall2 $P\ r\ s.$
 Proof using.
 introv $H\ E.$ gen $s.$ induction $r;$ intros; destruct $s;$ tryfalse.
 { auto. }
 { rew_list in $E.$ constructors. }

```

{ appls H 0. rew_list. math. }
{ appls  $\neg$  IHr. introv N. appls H (S n). rew_list. math. } }

```

Qed.

End Forall2.

Forall3 is similar to **Forall2** except that it relates three lists.

```

Inductive Forall3 A B C (P : A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  Prop)
: list A  $\rightarrow$  list B  $\rightarrow$  list C  $\rightarrow$  Prop :=
| Forall3_nil :
  Forall3 P nil nil nil
| Forall3_cons :  $\forall$  l1 l2 l3 x1 x2 x3,
  P x1 x2 x3  $\rightarrow$ 
  Forall3 P l1 l2 l3  $\rightarrow$ 
  Forall3 P (x1::l1) (x2::l2) (x3::l3).

```

Exists *P l* asserts that there exists a value in the list *l* that satisfied the predicate *P*.

```

Inductive Exists A (P:A $\rightarrow$ Prop) : list A  $\rightarrow$  Prop :=
| Exists_head :  $\forall$  l x,
  P x  $\rightarrow$ 
  Exists P (x::l)
| Exists_tail :  $\forall$  l x,
  Exists P l  $\rightarrow$ 
  Exists P (x::l).

```

Section Exists.

Variables *A* : Type.

Implicit Types *l* : list *A*.

Implicit Types *P* : *A* $\rightarrow$ Prop.

Hint Constructors **Exists**.

Lemma Exists_nil_eq : $\forall$ *P*,

Exists *P* nil = **False**.

Proof using. intros. *extens*. iff *M*. { *invert* $\times$ *M*. } { *false*. } Qed.

Lemma Exists_cons_eq : $\forall$ *P l x*,

Exists *P* (*x*::*l*) = (*P* *x* $\vee$ **Exists** *P* *l*).

Proof using. intros. *extens*. iff *M*. { *inverts* $\times$ *M*. } { *destruct* $\times$ *M*. } Qed.

Lemma Exists_one_eq : $\forall$ *P x*,

Exists *P* (*x*::nil) = *P* *x*.

Proof using. intros. *extens*. rewrite Exists_cons_eq. rewrite $\times$ Exists_nil_eq. Qed.

Lemma Exists_app_eq : $\forall$ *P l1 l2*,

Exists *P* (*l1*++*l2*) = (**Exists** *P* *l1* $\vee$ **Exists** *P* *l2*).

Proof using.

```

intros. extens. induction l1; rew_list.
{ iff M [M|M]. { autos×. } { inverts¬ M. } { autos×. } }
{ iff M [M|M].
  { inverts× M. }
  { inverts M. { auto. } { applies× Exists_tail. } }
  { applies× Exists_tail. } }
Qed.

Lemma Exists_last_eq : ∀ P l x,
  Exists P (l&x) = (P x ∨ Exists P l).
Proof using.
  intros. extens. rewrite Exists_app_eq. rewrite Exists_cons_eq.
  rewrite× Exists_nil_eq.
Qed.

Lemma Exists_app_l : ∀ P l1 l2,
  Exists P l1 →
  Exists P (l1++l2).
Proof using. intros. rewrite× Exists_app_eq. Qed.

Lemma Exists_app_r : ∀ P l1 l2,
  Exists P l2 →
  Exists P (l1++l2).
Proof using. intros. rewrite× Exists_app_eq. Qed.

Lemma Exists_last : ∀ P l x,
  Exists P l →
  P x →
  Exists P (l&x).
Proof using. intros. rewrite× Exists_last_eq. Qed.

Lemma Exists_nil_inv : ∀ P,
  Exists P nil →
  False.
Proof using. introv H. rewrite× Exists_nil_eq in H. Qed.

Lemma Exists_cons_inv : ∀ P l x,
  Exists P (x::l) →
  P x ∨ Exists P l.
Proof using. introv H. rewrite× Exists_cons_eq in H. Qed.

Lemma Exists_one_inv : ∀ P x,
  Exists P (x::nil) →
  P x.
Proof using. introv H. rewrite× Exists_one_eq in H. Qed.

Lemma Exists_app_inv : ∀ P l1 l2,
  Exists P (l1++l2) →

```

Exists $P\ l1 \vee \text{Exists } P\ l2$.

Proof using. *intros* H . *rewrite*× *Exists_app_eq* in H . Qed.

Lemma *Exists_last_inv* : $\forall P\ l\ x$,

Exists $P\ (l\&x) \rightarrow$
 $P\ x \vee \text{Exists } P\ l$.

Proof using. *intros* H . *rewrite*× *Exists_last_eq* in H . Qed.

Lemma *Exists_eq_exists_mem* : $\forall P\ l$,

Exists $P\ l = (\exists x, \text{mem } x\ l \wedge P\ x)$.

Proof using.

Hint Constructors **mem**.

intros. *extens*. *induction* l as $[[y\ l']$.

{ *iff* $M\ (x\&M\&H)$. { *inverts* M . } { *inverts* M . } }
{ *iff* $M\ (x\&M\&H)$.
{ *inverts* M as N .
{ $\exists x\ y$. }
{ *forwards* $(x\&M\&H)$: (*proj1* IHL') N . $\exists x$. } }
{ *destruct* (*mem_cons_inv* M) as $[N|N]$.
{ *subst*×. }
{ *applies*× *Exists_tail*. } } }

Qed.

Lemma *mem_Exists* : $\forall P\ l\ x$,

mem $x\ l \rightarrow$
 $P\ x \rightarrow$
Exists $P\ l$.

Proof using. *intros* $M\ H$. *rewrite*× *Exists_eq_exists_mem*. Qed.

Lemma *mem_Exists_eq* : $\forall P\ l\ x$,

mem $x\ l = \text{Exists } (=x)\ l$.

Proof using.

intros. *extens*. *rewrite*× *Exists_eq_exists_mem*. *iff* $M\ (y\&?\&?)$; *subst*×.

Qed.

Lemma *Nth_Exists* : $\forall P\ n\ l\ x$,

Nth $n\ l\ x \rightarrow$
 $P\ x \rightarrow$
Exists $P\ l$.

Proof using. *intros* $M\ H$. *applies*× *mem_Exists*. *applies*× *Nth_mem*. Qed.

Lemma *nth_Exists* : $\forall \{IA:\text{Inhab } A\} P\ n\ l$,

$P\ (\text{nth } n\ l) \rightarrow$
 $n < \text{length } l \rightarrow$
Exists $P\ l$.

Proof using. *intros* $H\ N$. *applies*× *Nth_Exists* n . *applies*× $\neg$ *Nth_nth*. Qed.

Lemma *Exists_rev* : $\forall P\ l$,

```

Exists  $P\ l \rightarrow$ 
Exists  $P\ (\text{rev } l)$ .
Proof using.
  intros  $E$ . induction  $l$ .
  { auto. }
  { rew_list. rewrite Exists_last_eq. inverts  $\times E$ . }
Qed.

Lemma Exists_rev_eq :  $\forall\ P\ l$ ,
  Exists  $P\ (\text{rev } l) =$  Exists  $P\ l$ .
Proof using.
  intros. extens. iff  $M$ .
  { rewrite  $\leftarrow$  rev_rev. applies  $\neg$  Exists_rev. }
  { applies  $\neg$  Exists_rev. }
Qed.

Lemma Exists_pred_incl :  $\forall\ P\ Q\ l$ ,
  Exists  $P\ l \rightarrow$ 
  pred_incl  $P\ Q \rightarrow$ 
  Exists  $Q\ l$ .
Proof using. intros. induction  $l$ ; intros  $H\ L$ ; inverts  $\times H$ . Qed.

Lemma Exists_filter_pred_incl :  $\forall\ P\ Q\ l$ ,
  Exists  $P\ l \rightarrow$ 
  pred_incl  $P\ Q \rightarrow$ 
  Exists  $P\ (\text{filter } Q\ l)$ .
Proof using.
  intros  $M\ N$ . induction  $M$ ; rew_listx.
  { forwards*:  $N$ . case_if  $\times$ . }
  { case_if  $\neg$ . }
Qed.

Lemma Exists_filter_same :  $\forall\ P\ l$ ,
  Exists  $P\ l \rightarrow$ 
  Exists  $P\ (\text{filter } P\ l)$ .
Proof using. intros  $M$ . applies  $\times$  Exists_filter_pred_incl. applies pred_incl_refl. Qed.

Lemma Exists_take_inv :  $\forall\ P\ n\ l$ ,
  Exists  $P\ (\text{take } n\ l) \rightarrow$ 
  Exists  $P\ l$ .
Proof using.
  intros  $E$ . forwards  $(q\ \&\ K)$ : take_is_prefix  $n\ l$ .
  rewrite  $K$ . rewrite  $\times$  Exists_app_eq.
Qed.

Lemma Exists_drop_inv :  $\forall\ P\ n\ l$ ,
  Exists  $P\ (\text{drop } n\ l) \rightarrow$ 

```

Exists $P\ l$.

Proof using.

intros E . *forwards* $(q\&K)$: *drop_is_suffix* $n\ l$.

rewrite K . *rewrite* $\times$ *Exists_app_eq*.

Qed.

Lemma *Exists_inv_middle_first* : $\forall\ P\ l$,

Exists $P\ l \rightarrow$

$\exists\ l1\ x\ l2$,

$l = l1 ++ x :: l2$

$\wedge$ **Forall** $(\text{fun } x \Rightarrow \neg P\ x)\ l1$

$\wedge P\ x$.

Proof using.

intros E . *induction* E .

$\exists\ (@nil\ A)\ x\ l$. *splits* $\neg$. *constructors* $\neg$.

lets $(l1\&x'\&l2\&E1\&F\&HP)$: $(\text{rm } IHE)$. *tests* Px : $(P\ x)$.

$\exists\ (@nil\ A)\ x\ l$. *splits* $\neg$. *constructors* $\neg$.

subst. $\exists\ (x :: l1)\ x'\ l2$. *splits* $\neg$. *constructors* $\neg$.

Qed.

End *Exists*.

#[global]

Hint *Rewrite* *Exists_nil_eq* *Exists_cons_eq* *Exists_one_eq* *Exists_app_eq* *Exists_last_eq* : *rew_listx*.

Exists2 $P\ L1\ L2$ asserts that there exists an index n such that the n -th element of $L1$ and the n -th element of $L2$ are related by the binary relation P .

Inductive **Exists2** $A1\ A2\ (P : A1 \rightarrow A2 \rightarrow \text{Prop})$

: $\text{list } A1 \rightarrow \text{list } A2 \rightarrow \text{Prop} :=$

| *Exists2_here* : $\forall\ l1\ l2\ x1\ x2$,

$P\ x1\ x2 \rightarrow$

Exists2 $P\ (x1 :: l1)\ (x2 :: l2)$

| *Exists2_next* : $\forall\ l1\ l2\ x1\ x2$,

Exists2 $P\ l1\ l2 \rightarrow$

Exists2 $P\ (x1 :: l1)\ (x2 :: l2)$.

count $P\ l$ returns the number of elements of l satisfying the predicate P .

Definition *count* $A\ (P : A \rightarrow \text{Prop})\ (l : \text{list } A) : \text{nat} :=$

length (*filter* $P\ l$).

Section *Count*.

Variables $A : \text{Type}$.

Implicit Types $l : \text{list } A$.

Implicit Types $P : A \rightarrow \text{Prop}$.

Lemma count_eq_length_filter : $\forall P l$,
 count $P l$ = length (filter $P l$).

Proof using. auto. Qed.

Lemma count_nil : $\forall P$,
 count P nil = 0.

Proof using. auto. Qed.

Lemma count_one : $\forall P x$,
 count $P (x::\text{nil})$ = If $P x$ then 1 else 0.

Proof using. intros. unfold count. *rew_listx*. *case_if* $\neg$. Qed.

Lemma count_app : $\forall P l1 l2$,
 count $P (l1++l2)$ = count $P l1$ + count $P l2$.

Proof using. intros. unfold count. *rew_listx* $\neg$. Qed.

Lemma count_cons : $\forall P l x$,
 count $P (x::l)$ = count $P l$ + If $P x$ then 1 else 0.

Proof using.

intros. rewrite $\leftarrow$ app_cons_one_r, count_app, count_one. *math*.
 Qed.

Lemma count_last : $\forall P l x$,
 count $P (l\&x)$ = count $P l$ + If $P x$ then 1 else 0.

Proof using. intros. rewrite count_app, count_one. *math*. Qed.

Lemma count_rev : $\forall P l$,
 count $P (\text{rev } l)$ = count $P l$.

Proof using. intros. unfold count. *rew_listx* $\neg$. Qed.

End Count.

#[*global*]

Hint Rewrite count_nil count_cons count_one count_app count_rev : *rew_listx*.

Section Count2.

Variables A : Type.

Implicit Types l : list A .

Implicit Types P : $A \rightarrow \text{Prop}$.

Lemma count_le_length : $\forall P l$,
 count $P l \leq \text{length } l$.

Proof using. intros. apply length_filter. Qed.

Lemma Forall_eq_count_eq_length : $\forall P l$,
Forall $P l$ = (count $P l$ = length l).

Proof using.

intros. induction l ; *rew_listx*.
 { *extens* $\times$. }


```

{ rewrite IHL. extens. iff (M1&M2) M; case-if.
  { math. }
  { split¬. math. }
  { false. forwards~: count_le_length P l. math. } }

```

Qed.

```

Lemma count_Forall : ∀ P l,
  Forall P l →
  count P l = length l.

```

Proof using. *introv*. rewrite¬ Forall_eq_count_eq_length. Qed.

```

Lemma Forall_of_count_eq_length : ∀ P l,
  count P l = length l →
  Forall P l.

```

Proof using. *introv*. rewrite¬ Forall_eq_count_eq_length. Qed.

```

Lemma Forall_not_eq_count_eq_zero : ∀ P l,
  Forall (fun x ⇒ ¬ P x) l = (count P l = 0).

```

Proof using.

```

intros. induction l; rew_listx.
{ extens×. }
{ rewrite IHL. extens. iff (M1&M2) M; case-if.
  { math. }
  { false. math. }
  { split¬. math. } }

```

Qed.

```

Lemma Forall_not_of_count_eq_zero : ∀ P l,
  count P l = 0 →
  Forall (fun x ⇒ ¬ P x) l.

```

Proof using. *introv*. rewrite¬ Forall_not_eq_count_eq_zero. Qed.

```

Lemma count_eq_zero_of_Forall_not : ∀ P l,
  Forall (fun x ⇒ ¬ P x) l →
  count P l = 0.

```

Proof using. *introv*. rewrite¬ Forall_not_eq_count_eq_zero. Qed.

```

Lemma Exists_eq_count_pos : ∀ P l,
  Exists P l = (count P l > 0).

```

Proof using.

```

intros. induction l; rew_listx.
{ extens. iff M; false; math. }
{ rewrite IHL. extens. iff [M|M] M; case-if; try math.
  { left¬. }
  { right. math. } }

```

Qed.

```

Lemma Exists_of_count_pos :  $\forall P l,$ 
  count  $P l > 0 \rightarrow$ 
  Exists  $P l$ .
Proof using. intros. rewrite  $\neg$  Exists_eq_count_pos. Qed.

Lemma count_pos_of_Exists :  $\forall P l,$ 
  Exists  $P l \rightarrow$ 
  count  $P l > 0$ .
Proof using. intros. rewrite  $\neg$  Exists_eq_count_pos. Qed.

Lemma count_pos_eq_exists_mem :  $\forall P l,$ 
  (count  $P l > 0$ ) = ( $\exists x, \text{mem } x l \wedge P x$ ).
Proof using.
  intros. rewrite  $\leftarrow$  Exists_eq_count_pos, Exists_eq_exists_mem. auto.
Qed.

Lemma count_pos_of_mem :  $\forall x P l,$ 
  mem  $x l \rightarrow$ 
   $P x \rightarrow$ 
  count  $P l > 0$ .
Proof using. intros. rewrite  $\times$  count_pos_eq_exists_mem. Qed.

Lemma exists_mem_of_count_pos :  $\forall P l,$ 
  count  $P l > 0 \rightarrow$ 
   $\exists x, \text{mem } x l \wedge P x$ .
Proof using. intros. rewrite  $\times$  count_pos_eq_exists_mem. Qed.

End Count2.

Global Opaque count.

```

19.7 Nat seq

`nat_seq i n` generates a list of variables $x_1; x_2; \dots; x_n$ with $x_1 = i$ and $x_n = i + n - 1$. Such lists are useful for generic programming.

```

Fixpoint nat_seq (start:nat) (nb:nat) :=
  match nb with
  | 0  $\Rightarrow$  nil
  | S nb'  $\Rightarrow$  start :: nat_seq (S start) nb'
  end.

```

```

Lemma nat_seq_zero :  $\forall o,$ 
  nat_seq  $o 0 = \text{nil}$ .

```

Proof using. `auto`. Qed.

```

Lemma nat_seq_succ :  $\forall o k,$ 
  nat_seq  $o (S k) = o :: (\text{nat\_seq } (S o) k)$ .

```

```

Proof using. auto. Qed.

Lemma nat_seq_succ' : ∀ o k,
  nat_seq o (1+k) = o :: (nat_seq (o+1) k).
Proof using. intros. simpl. fequals_rec. math. Qed.

Lemma length_nat_seq : ∀ start nb,
  length (nat_seq start nb) = nb.
Proof using.
  intros. gen start. induction nb; simpl; intros.
  { auto. } { rew_list. rewrite ¬ IHnb. }
Qed.

Global Opaque nat_seq.

#[global]
Hint Rewrite nat_seq_zero nat_seq_succ : rew_listx.

```

19.8 Fold

WARNING: EXPERIMENTAL SECTION

```

Definition fold A B (m:monoid_op B) (f:A→B) (L:list A) : B :=
  fold_right (fun x acc => monoid_oper m (f x) acc) (monoid_neutral m) L.

```

Section Fold.

Variables (A B:Type).

Implicit Types m : monoid_op B.

Implicit Types l : list A.

Implicit Types f g : A→B.

```

Lemma fold_nil : ∀ m f,
  fold m f nil = monoid_neutral m.

```

Proof using. auto. Qed.

```

Lemma fold_cons : ∀ m f x l,
  fold m f (x::l) = monoid_oper m (f x) (fold m f l).

```

Proof using. auto. Qed.

```

Lemma fold_one : ∀ m f x,
  Monoid m →
  fold m f (x::nil) = f x.

```

Proof using.

```

  intros. rewrite fold_cons, fold_nil. rewrite ¬ monoid_neutral_r.
Qed.

```

```

Lemma fold_app : ∀ m f l1 l2,
  Monoid m →
  fold m f (l1 ++ l2) = monoid_oper m (fold m f l1) (fold m f l2).

```

Proof using.

```
unfold fold. intros. rewrite fold_right_app. gen l2.
induction l1; intros.
repeat rewrite fold_right_nil. rewrite  $\neg$  monoid_neutral_l.
repeat rewrite fold_right_cons. rewrite  $\leftarrow$  monoid_assoc. fequals.
```

Qed.

Lemma fold_last : $\forall m f x l$,

```
Monoid m  $\rightarrow$ 
fold m f (l & x) = monoid_oper m (fold m f l) (f x).
```

Proof using.

```
intros. rewrite  $\neg$  fold_app. rewrite fold_cons.
rewrite fold_nil. rewrite monoid_neutral_r. auto.
```

Qed.

Lemma fold_congruence : $\forall m f g l$,

```
( $\forall x$ , mem x l  $\rightarrow f x = g x$ )  $\rightarrow$ 
fold m f l = fold m g l.
```

Proof using.

```
Hint Constructors mem.
intros. unfold fold.
induction l as [| x l']; intros; simpl.
{ eauto. }
{ rew_listx. fequals. applies  $\neg$  H. eauto. }
```

Qed.

Lemma fold_equiv_step : $\forall m f l a$,

```
Comm_monoid m  $\rightarrow$ 
noduplicates l  $\rightarrow$ 
mem a l  $\rightarrow$ 
 $\exists l'$ ,
  fold m f l = fold m f (a :: l')
 $\wedge (\forall x$ , mem x l  $\leftrightarrow$  mem x (a :: l'))
 $\wedge$  noduplicates (a :: l').
```

Proof using. *introv Hm*. induction *l* as [| *b t*]; *introv DL La*. *inverts La*.

tests: (*a* = *b*).

$\exists t$. *splits* $\times$.

inverts La. *false*. *inverts DL* as *DLb DT*. *forwards* $\neg$ (*L'* & *EL'* & *EQ* & *DL'*): *IHt*.

$\exists (b :: L')$. *splits*.

do 3 rewrite fold_cons. rewrite *EL'*.

rewrite fold_cons. do 2 rewrite monoid_assoc.

rewrite $\neg$ (monoid_comm (*f b*)).

intros *x*. *specializes EQ x*. rewrite mem_cons_eq in *EQ*.

do 3 rewrite mem_cons_eq. *autos* $\times$.

inverts DL'. *constructors*.

```

      introv N. inverts N. false. false.
      constructors¬. introv N. applies DLb. rewrite EQ. constructors¬.

Qed.

Lemma fold_equiv : ∀ m f l1 l2,
  Comm_monoid m →
  noduplicates l1 →
  noduplicates l2 →
  (∀ x, mem x l1 ↔ mem x l2) →
  fold m f l1 = fold m f l2.
Proof using.   intros m f l1. induction l1; introv HM D1 D2 EQ.
  cuts_rewrite (l2 = nil). rewrite¬ fold_nil.
    destruct l2; auto. forwards¬ M: (proj2 (EQ a)). inverts M.
  inverts D1. asserts L2a: (mem a l2). rewrite¬ ← EQ.
    forwards× (l2' & V2' & EQ' & D2'): fold_equiv_step f L2a.
    rewrite V2'. do 2 rewrite fold_cons.
  inverts D2'.
  rewrite¬ (IHL1 l2'). intros.
  tests: (x = a).
    iff; auto_false×.
  asserts_rewrite (mem x l1 = mem x (a :: l1)).
    extens. iff¬ M. inverts¬ M. false.
  asserts_rewrite (mem x l2' = mem x (a :: l2')).
    extens. iff¬ M. inverts¬ M. false.
  rewrite EQ. rewrite× EQ'.

Qed.

End Fold.

Lemma fold_pointwise : ∀ B m (leB : B → B → Prop),
  Monoid m →
  refl leB →
  Proper (leB ++> leB ++> leB) (monoid_oper m) →
  ∀ A (l : list A),
  ∀ (f f' : A → B),
  (∀ x, mem x l → leB (f x) (f' x)) →
  leB (fold m f l) (fold m f' l).
Proof using. #[global]
  Hint Constructors mem.
  introv HM HR HP. induction l; introv HL.
  do 2 rewrite fold_nil. applies HR.
  do 2 rewrite fold_cons. apply HP. applies¬ HL. applies¬ IHL.

Qed.

Global Opaque fold.

```

```

#[global]
Hint Rewrite fold_nil fold_cons fold_app : rew_listx.

Section ListSub.
Variable (A:Type).

    Sub-list well-founded order

Inductive list_sub : list A → list A → Prop :=
| list_sub_cons : ∀ x l,
    list_sub l (x::l)
| list_sub_tail : ∀ x l1 l2,
    list_sub l1 l2 →
    list_sub l1 (x::l2).

Hint Constructors list_sub.

Lemma list_sub_wf : LibWf.wf list_sub.
Proof using.
    intros l. induction l; apply Acc_intro; introv H.
    { inverts¬ H. }
    { inverts¬ H. applies¬ IHL. }
Qed.

End ListSub.

Arguments list_sub [A].
#[global]
Hint Constructors list_sub.
#[global]
Hint Resolve list_sub_wf : wf.

    Induction on all but last item

Lemma list_ind_last : ∀ (A : Type) (P : list A → Prop),
    P nil →
    (∀ (a : A) (l : list A), P l → P (l & a)) →
    ∀ l : list A, P l.
Proof using.
    introv H1 H2. intros. induction_wf IH: (wf_measure (@length A)) l.
    lets [E|(x&l'&E)|]: (last_case l); subst. auto.
    unfolds measure. rewrite length_last in IH. auto with maths.
Qed.

Lemma list2_ind : ∀ A B (P:list A→list B→Prop) l1 l2,
    length l1 = length l2 →
    P nil nil →
    (∀ x1 xs1 x2 xs2,
        length xs1 = length xs2 → P xs1 xs2 → P (x1::xs1) (x2::xs2)) →

```

P l1 l2.

Proof using.

```
introv E M1 M2. gen l2. induction l1 as [|x1 l1']; intros;  
  destruct l2 as [|x2 l2']; try solve [false; math]; auto.
```

Qed.

```
Tactic Notation "list2_ind" constr(l1) constr(l2) :=  
  pattern l2; pattern l1;  
  match goal with ⊢ (fun a ⇒ (fun b ⇒ @?P a b) _) _ ⇒
```

```
  let X := fresh "P" in set (X := P); applies list2_ind X; unfold X; try clear X  
end.
```

```
Tactic Notation "list2_ind" "~" constr(l1) constr(l2) :=  
  list2_ind l1 l2; auto_tilde.
```

```
Tactic Notation "list2_ind" "*" constr(l1) constr(l2) :=  
  list2_ind l1 l2; auto_star.
```

```
Tactic Notation "list2_ind" constr(E) :=  
  match type of E with length ?l1 = length ?l2 ⇒  
    list2_ind l1 l2; [ apply E | | ] end.
```

```
Lemma list2_ind_last : ∀ A B (P:list A→list B→Prop) l1 l2,  
  length l1 = length l2 →  
  P nil nil →  
  (∀ x1 xs1 x2 xs2,  
    length xs1 = length xs2 → P xs1 xs2 → P (xs1&x1) (xs2&x2)) →  
  P l1 l2.
```

Proof using.

```
introv E M1 M2. gen l2. induction l1 using list_ind_last;  
  [| rename a into x1, l1 into l1']; intros;  
  destruct (last_case l2) as [(x2&l2'&E2)]; subst; rew_list in *;  
  try solve [false; math]; auto.
```

Qed.

```
Tactic Notation "list2_ind_last" constr(l1) constr(l2) :=  
  pattern l2; pattern l1;  
  match goal with ⊢ (fun a ⇒ (fun b ⇒ @?P a b) _) _ ⇒
```

```
  let X := fresh "P" in set (X := P); applies list2_ind_last X; unfold X; try clear X  
end.
```

```
Tactic Notation "list2_ind_last" "~" constr(l1) constr(l2) :=  
  list2_ind_last l1 l2; auto_tilde.
```

```
Tactic Notation "list2_ind_last" "*" constr(l1) constr(l2) :=  
  list2_ind_last l1 l2; auto_star.
```

```
Tactic Notation "list2_ind_last" constr(E) :=  
  match type of E with length ?l1 = length ?l2 =>  
    list2_ind_last l1 l2; [ apply E | | ] end.
```


Chapter 20

Library SLF.LibListExec

```
Set Implicit Arguments.
Generalizable Variables A B.
From SLF Require Import LibTactics LibReflect.
From SLF Require Export LibNat LibList.
```

This module shadows the function from LibList. Hence, it is not meant to be “imported”. Instead, just use `Import REWLISTEXEC`.

```
Module Import REWLISTEXEC.

Tactic Notation "rew_list_exec" :=
  autorewrite with rew_list_exec.
Tactic Notation "rew_list_exec" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_list_exec).
Tactic Notation "rew_list_exec" "in" hyp(H) :=
  autorewrite with rew_list_exec in H.

End REWLISTEXEC.
```

```
Definition is_nil A (l:list A) : bool :=
  match l with
  | nil => true
  | _ => false
  end.
```

```
Lemma is_nil_eq :  $\forall A (l:\text{list } A),$ 
  is_nil l = isTrue (l = nil).
```

Proof using. intros. destruct l; simpl; *rew_bool_eq*; *auto_false*. Qed.

```
Definition is_not_nil A (l:list A) : bool :=
  match l with
  | nil => false
  | _ => true
  end.
```

```

Lemma is_not_nil_eq :  $\forall A (l : \text{list } A),$ 
  is_not_nil  $l = \text{isTrue } (l \neq \text{nil})$ .
Proof.
  intros. destruct  $l$ ; simpl; rew_bool_eq; auto_false.
Qed.

#[global] Hint Rewrite is_nil_eq is_not_nil_eq : rew_list_exec.

Definition length :  $\forall A, \text{list } A \rightarrow \text{nat} :=$ 
  @List.length.

Lemma length_eq :
  length = LibList.length.
Proof using. extens  $\Rightarrow A l$ . induction  $l$ ; simpl; rew_list; auto. Qed.

#[global] Hint Rewrite length_eq : rew_list_exec.

Definition app :  $\forall A, \text{list } A \rightarrow \text{list } A \rightarrow \text{list } A :=$ 
  @List.app.

Lemma app_eq :
  app = LibList.app.
Proof using.
  extens  $\Rightarrow A L1 L2$ . induction  $L1$ ; simpl; rew_list; congruence.
Qed.

#[global] Hint Rewrite app_eq : rew_list_exec.

Definition rev :  $\forall A, \text{list } A \rightarrow \text{list } A :=$ 
  @List.rev.

Lemma rev_eq :  $\forall A,$ 
  @List.rev  $A = @\text{LibList.rev } A$ .
Proof using.
  extens  $\Rightarrow L$ . induction  $L$ ; simpl; rew_list. { auto. }
  { rewrite IHL. rewrite  $\leftarrow$  app_eq. unfold app. fequals. }
Qed.

#[global] Hint Rewrite rev_eq : rew_list_exec.

Definition map :  $\forall A B, (A \rightarrow B) \rightarrow \text{list } A \rightarrow \text{list } B :=$ 
  @List.map.

Lemma map_eq :
  map = LibList.map.
Proof using.
  extens  $\Rightarrow A B f L$ . induction  $L$ ; simpl; rew_listx; congruence.
Qed.

#[global] Hint Rewrite map_eq : rew_list_exec.

```

```

Fixpoint mem A (cmp:A→A→bool) (x:A) (l:list A) : bool :=
  match l with
  | nil ⇒ false
  | y::l' ⇒ cmp x y || mem cmp x l'
  end.

```

```

Lemma mem_eq : ∀ A (cmp:A→A→bool) x l,
  is_beq cmp →
  mem cmp x l = isTrue (LibList.mem x l).

```

Proof using.

```

  introv M. induction l as [|y l']; simpl; rew_listx; rew_isTrue; fequals.

```

Qed.

```

#[global] Hint Rewrite mem_eq : rew_list_exec.

```

20.0.1 concat

```

Definition concat A (m:list (list A)) : list A :=
  fold_right (@app A) (@nil A) m.

```

Transparent LibList.concat.

```

Lemma concat_eq :
  concat = LibList.concat.

```

Proof using.

```

  unfold concat. rew_list_exec. unfold LibList.concat. rew_list_exec. auto.

```

Qed.

```

#[global] Hint Rewrite concat_eq : rew_list_exec.

```

20.0.2 filter

```

Definition filter A (p:A→bool) (l:list A) : list A :=
  fold_right (fun x acc ⇒ if p x then x::acc else acc) (@nil A) l.

```

```

Lemma filter_eq : ∀ A (p:A→bool) (P:A→Prop) l,
  isTrue_pred p P →
  filter p l = LibList.filter P l.

```

Proof using.

```

  Local Transparent LibList.filter.
  introv E. unfold filter, LibList.filter. rew_list_exec.
  fequals. extens. intros. rewrite E. rewrite× if_isTrue.

```

Qed.

```

#[global] Hint Rewrite filter_eq : rew_list_exec.

```

20.0.3 remove

Definition remove A ($cmp:A \rightarrow A \rightarrow \mathbf{bool}$) ($a:A$) ($l:\mathbf{list}\ A$) : $\mathbf{list}\ A :=$
 filter (fun $x \Rightarrow \text{neg } (cmp\ a\ x)$) l .

Lemma remove_eq : $\forall\ A\ (cmp:A \rightarrow A \rightarrow \mathbf{bool})\ (a:A)\ (l:\mathbf{list}\ A),$
 is_beq $cmp \rightarrow$
 remove $cmp\ a\ l = \mathbf{LibList.remove}\ a\ l$.

Proof using.

Local Transparent LibList.remove.

introv E . unfold remove, LibList.remove.

rewrites $\times$ ($\gg$ filter_eq ($\neq a$)).

{ intros x . rewrite E . rew_isTrue. fequals. rew_bool_eq $\times$. }

Qed.

#[global] Hint Rewrite remove_eq : rew_list_exec.

Fixpoint noduplicates A ($cmp:A \rightarrow A \rightarrow \mathbf{bool}$) ($l:\mathbf{list}\ A$) : $\mathbf{bool} :=$
 match l with
 | nil $\Rightarrow$ true
 | $x::l' \Rightarrow \text{neg } (\text{mem } cmp\ x\ l') \ \&\& \text{noduplicates } cmp\ l'$
 end.

Lemma noduplicates_eq : $\forall\ A\ (cmp:A \rightarrow A \rightarrow \mathbf{bool})\ (l:\mathbf{list}\ A),$
 is_beq $cmp \rightarrow$
 noduplicates $cmp\ l = \text{isTrue } (\mathbf{LibList.noduplicates}\ l)$.

Proof using.

introv E . extens. induction l as [| $x\ l'$]; simpl; rew_listx; rew_bool_eq in *.

{ autos $\times$. }

{ rewrite $\times$ mem_eq. rew_bool_eq. iff ($M1\ \&\ M2$); splits $\times$. }

Qed.

#[global] Hint Rewrite noduplicates_eq : rew_list_exec.

Fixpoint remove_duplicates A ($cmp:A \rightarrow A \rightarrow \mathbf{bool}$) ($l:\mathbf{list}\ A$) : $\mathbf{list}\ A :=$
 match l with
 | nil $\Rightarrow$ nil
 | $x::l' \Rightarrow x :: (\text{remove } cmp\ x\ (\text{remove\_duplicates } cmp\ l'))$
 end.

Lemma remove_duplicates_eq : $\forall\ A\ (cmp:A \rightarrow A \rightarrow \mathbf{bool})\ (l:\mathbf{list}\ A),$
 is_beq $cmp \rightarrow$
 remove_duplicates $cmp\ l = \mathbf{LibList.remove\_duplicates}\ l$.

Proof using.

Local Transparent LibList.remove_duplicates.

introv E . induction l as [| $x\ l'$]; simpl; fequals.

rewrite $\times$ remove_eq. congruence.

Qed.

#[global] Hint Rewrite remove_duplicates_eq : *rew_list_exec*.

```
Fixpoint Forall A (p:A→bool) (l:list A) : bool :=
  match l with
  | nil ⇒ true
  | x::l' ⇒ p x && Forall p l'
  end.
```

Lemma Forall_eq : $\forall A (p:A \rightarrow \text{bool}) (P:A \rightarrow \text{Prop}) l$,
isTrue_pred p P $\rightarrow$
Forall p l = isTrue (LibList.Forall P l).

Proof using.

```
  introv E. extens. induction l as [|x l']; simpl; rew_listx in *; rew_bool_eq in *.
  { iff ×. }
  { rewrite E. rew_bool_eq. iff ×. }
```

Qed.

#[global] Hint Rewrite Forall_eq : *rew_list_exec*.

```
Fixpoint Exists A (p:A→bool) (l:list A) : bool :=
  match l with
  | nil ⇒ false
  | x::l' ⇒ p x || Exists p l'
  end.
```

Lemma Exists_eq : $\forall A (p:A \rightarrow \text{bool}) (P:A \rightarrow \text{Prop}) l$,
isTrue_pred p P $\rightarrow$
Exists p l = isTrue (LibList.Exists P l).

Proof using.

```
  introv E. extens. induction l as [|x l']; simpl; rew_listx in *; rew_bool_eq in *.
  { iff ×. }
  { rewrite E. rew_bool_eq. iff ×. }
```

Qed.

#[global] Hint Rewrite Exists_eq : *rew_list_exec*.

```
Fixpoint count A (p:A→bool) (l:list A) : nat :=
  match l with
  | nil ⇒ 0%nat
  | x::l' ⇒ if p x then S (count p l') else count p l'
  end.
```

Lemma count_eq : $\forall A (p:A \rightarrow \text{bool}) (P:A \rightarrow \text{Prop}) l$,
isTrue_pred p P $\rightarrow$
count p l = LibList.count P l.

Proof using.

```

    introv E. induction l as [|x l']; simpl; rew_listx.
    { auto. }
    { rewrite IHl'. rewrite E. do 2 case_if; try nat_math. }
Qed.

```

```

#[global] Hint Rewrite count_eq : rew_list_exec.

```

```

Definition combine :  $\forall A B$ , list A  $\rightarrow$  list B  $\rightarrow$  list (A  $\times$  B) :=
  @List.combine.

```

```

Lemma combine_eq :  $\forall A B$  (L1:list A) (L2:list B),
  LibList.length L1 = LibList.length L2  $\rightarrow$ 
  combine L1 L2 = LibList.combine L1 L2.

```

```

Proof using.   introv E. gen L2.
  induction L1 as [|x1 L1']; intros; destruct L2 as [|x2 L2']; tryfalse.
  { auto. }
  { rew_list in E. rew_listx. simpl. fequals $\neg$ . }
Qed.

```

```

#[global] Hint Rewrite combine_eq : rew_list_exec.

```

LibList.fold_right computes well. Here, we prove that it matches Rocq's standard library *fold_right*.

```

Lemma fold_right_eq :  $\forall A B$ ,
  @List.fold_right B A = @LibList.fold_right A B.

```

```

Proof using. extens. intros a b l. induction l; simpl; rew_listx; fequals. Qed.

```

```

#[global] Hint Rewrite fold_right_eq : rew_list_exec.

```

Chapter 21

Library SLF.LibListZ

```
Set Implicit Arguments.
Generalizable Variables A B.
From SLF Require Import LibTactics LibLogic LibOperation LibReflect
  LibProd LibNat LibInt LibOption LibWf.
From SLF Require Export LibList LibNat.
From SLF Require Import LibInt.
From SLF Require Export LibContainer.

Open Scope Int_scope.
Local Open Scope comp_scope.

Ltac auto_tilde ::= eauto with maths.
```

21.1 Tactics

21.1.1 *rew_listp* for operations with preconditions

```
Tactic Notation "rew_listp" :=
  autorewrite with rew_listp.
Tactic Notation "rew_listp" "~" :=
  rew_listp; auto_tilde.
Tactic Notation "rew_listp" "*" :=
  rew_listp; auto_star.
Tactic Notation "rew_listp" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_listp).
Tactic Notation "rew_listp" "~" "in" "*" :=
  rew_listp in *; auto_tilde.
Tactic Notation "rew_listp" "*" "in" "*" :=
  rew_listp in *; auto_star.
Tactic Notation "rew_listp" "in" hyp(H) :=
```

```

    autorewrite with rew_listp in H.
Tactic Notation "rew_listp" "~" "in" hyp(H) :=
    rew_listp in H; auto_tilde.
Tactic Notation "rew_listp" "*" "in" hyp(H) :=
    rew_listp in H; auto_star.

```

21.1.2 *rew_index* for operations on indices

```

Tactic Notation "rew_index" :=
    autorewrite with rew_index.
Tactic Notation "rew_index" "in" hyp(H) :=
    autorewrite with rew_index in H.
Tactic Notation "rew_index" "in" "*" :=
    autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_index).
Tactic Notation "rew_index" "~" :=
    rew_index; auto_tilde.
Tactic Notation "rew_index" "*" :=
    rew_index; auto_star.
Tactic Notation "rew_index" "~" "in" hyp(H) :=
    rew_index in H; auto_tilde.
Tactic Notation "rew_index" "*" "in" hyp(H) :=
    rew_index in H; auto_star.
Tactic Notation "rew_index" "~" "in" "*" :=
    rew_index in *; auto_tilde.
Tactic Notation "rew_index" "*" "in" "*" :=
    rew_index in *; auto_star.

```

21.1.3 *rew_array* for case-analysis free rewriting

```

Tactic Notation "rew_array_nocase" :=
    autorewrite with rew_array_nocase.
Tactic Notation "rew_array_nocase" "in" hyp(H) :=
    autorewrite with rew_array_nocase in H.
Tactic Notation "rew_array_nocase" "in" "*" :=
    autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_array_nocase).
Tactic Notation "rew_array_nocase" "~" :=
    rew_array_nocase; auto_tilde.
Tactic Notation "rew_array_nocase" "*" :=
    rew_array_nocase; auto_star.
Tactic Notation "rew_array_nocase" "~" "in" hyp(H) :=
    rew_array_nocase in H; auto_tilde.
Tactic Notation "rew_array_nocase" "*" "in" hyp(H) :=

```



```

    rew_array_nocase in H; auto_star.
Tactic Notation "rew_array_nocase" "~" "in" "*" :=
    rew_array_nocase in *; auto_tilde.
Tactic Notation "rew_array_nocase" "*" "in" "*" :=
    rew_array_nocase in *; auto_star.

```

21.1.4 *rew_array* for case-analysis aware rewriting

```

Tactic Notation "rew_array" :=
    autorewrite with rew_array.
Tactic Notation "rew_array" "in" hyp(H) :=
    autorewrite with rew_array in H.
Tactic Notation "rew_array" "in" "*" :=
    autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_array).
Tactic Notation "rew_array" "~" :=
    rew_array; auto_tilde.
Tactic Notation "rew_array" "*" :=
    rew_array; auto_star.
Tactic Notation "rew_array" "~" "in" hyp(H) :=
    rew_array in H; auto_tilde.
Tactic Notation "rew_array" "*" "in" hyp(H) :=
    rew_array in H; auto_star.
Tactic Notation "rew_array" "~" "in" "*" :=
    rew_array in *; auto_tilde.
Tactic Notation "rew_array" "*" "in" "*" :=
    rew_array in *; auto_star.
#[global] Hint Rewrite int_index_eq : rew_array.

```

21.1.5 *index_prove* for automating proofs on indices

```

Ltac index_prove tt :=
    rew_index; repeat case_if; rew_list in *; math.

Module INDEXHINTS.
    #[global]
    Hint Extern 1 (index _ _) => index_prove tt.
End INDEXHINTS.

```

21.2 List operations using indices in $\mathbb{Z}$

21.3 Length, with result as *int*

Defined using a coercion from **nat** to *int*

Definition length *A* (*l*:list *A*) : int :=
 LibList.length *l*.

Section Length.

Variables *A* : Type.

Implicit Types *x* : *A*.

Implicit Types *l* : list *A*.

Implicit Types *i* : int.

Lemma length_eq : $\forall l$,
 length *l* = LibList.length *l*.

Proof using. auto. Qed.

Lemma abs_length : $\forall i\ l$,
 i = length *l* $\rightarrow$
 abs *i* = LibList.length *l*.

Proof using.

intros E. unfold length in *E*.

applies eq_nat_of_eq_int.

rewrite abs_nonneg; *math*.

Qed.

Lemma length_nonneg : $\forall (l : \text{list } A)$,
 $0 \leq \text{length } l$.

Proof using. intros. rewrite length_eq. *math*. Qed.

Lemma length_nil :
 length (@nil *A*) = 0.

Proof using. auto. Qed.

Lemma length_cons : $\forall x\ l$,
 length (*x* :: *l*) = 1 + length *l*.

Proof using. intros. unfold length. *rew_list* $\neg$. Qed.

Lemma length_one : $\forall x$,
 length (*x* :: nil) = 1.

Proof using. reflexivity. Qed.

Lemma length_app : $\forall l1\ l2$,
 length (*l1* ++ *l2*) = length *l1* + length *l2*.

Proof using. intros. unfold length. *rew_list* $\neg$. Qed.

Lemma length_last : $\forall x\ l$,

```

length (l & x) = 1 + length l.
Proof using. intros. unfold length. rew_list⊢. Qed.
End Length.

Lemma Forall2_inv_length : ∀ A B (P:A→B→Prop) r s,
  Forall2 P r s →
  length r = length s.
Proof using. introv E. unfold length. lets*: Forall2_inv_length E. Qed.
#[global] Hint Rewrite length_nil length_cons length_app
length_last : rew_list.
#[global] Hint Rewrite length_nil length_cons length_app
length_last : rew_listx.

Automation for math, to unfold length
#[global] Hint Rewrite length_eq : rew_maths.
#[global] Hint Rewrite length_cons : rew_index.
#[global] Hint Rewrite length_app : rew_index.

Demo of automation with maths
Goal ∀ A (l : list A), 0 ≤ length l.
Proof using. intros. math. Qed.

Goal ∀ (l : list Z) (s n : int),
  s ≤ n →
  s = length l →
  n ≥ 0.
Proof using. intros. math. Qed.

```

21.4 Inversion lemmas for structural composition

```

Section AppInversion.
Variables A : Type.
Implicit Types x : A.
Implicit Types l : list A.

Lemma length_zero_inv : ∀ l,
  length l = 0 →
  l = nil.
Proof using. intros. unfolds length. applies⊢ LibList.length_zero_inv. Qed.

Lemma length_zero_eq_eq_nil : ∀ l,
  (length l = 0) = (l = nil).
Proof using.
  intros. unfolds length. rewrite ←LibList.length_zero_eq_eq_nil. math.

```

Qed.

Lemma length_neq_inv : $\forall l1\ l2,$
 $\text{length } l1 \neq \text{length } l2 \rightarrow$
 $(l1 \neq l2).$

Proof using. *intros N E. subst*×. Qed.

Lemma length_pos_inv_cons : $\forall l,$
 $(\text{length } l > 0) \rightarrow$
 $\exists x\ l',\ l = x :: l'.$

Proof using.
intros. unfolds length. applies¬ LibList.length_pos_inv_cons.
Qed.

Lemma length_pos_inv_last : $\forall l,$
 $(\text{length } l > 0) \rightarrow$
 $\exists x\ l',\ l = l' \& x.$

Proof using.
intros. unfolds length. applies¬ LibList.length_pos_inv_last.
Qed.

End AppInversion.

21.5 index, with length as *int*, as typeclass

Definition index_impl *A* (*l*:list *A*) (*i*:int) : Prop :=
index (LibList.length *l* : int) *i*.

#[*global*]

Instance index_inst : $\forall A,$ **BagIndex** int (list *A*).
Proof using. *constructor. rapply* (@index_impl *A*). Defined.

Global Opaque index_inst.

Section Index.

Variables (*A* : Type).

Implicit Types *l* : list *A*.

Implicit Types *n i* : int.

Lemma index_eq_inbound : $\forall l\ i,$
 $\text{index } l\ i = (0 \leq i < \text{length } l).$

Proof using. *auto.* Qed.

Lemma index_of_inbound : $\forall l\ i,$
 $0 \leq i < \text{length } l \rightarrow$
 $\text{index } l\ i.$

Proof using. *intros. rewrite*¬ index_eq_inbound. Qed.

Lemma index_eq_index_length : $\forall l\ i,$

```

    index l i = index (length l) i.
Proof using. auto. Qed.
Lemma index_of_index_length : ∀ l i,
    index (length l) i →
    index l i.
Proof using. intros H. rewrite× index_eq_index_length. Qed.
Hint Rewrite index_eq_index_length int_index_eq : rew_index.
Lemma index_app_l : ∀ l l' i,
    index l i →
    index (l++l') i.
Proof using. intros. rew_index. math. Qed.
Lemma index_app_r : ∀ l l' i,
    index l' (i - length l) →
    i ≥ length l →
    index (l++l') i.
Proof using. intros. rew_index. math. Qed.
Lemma index_app_eq : ∀ l l' i,
    index (l++l') i = (If i < length l then index l i else index l' (i - length l)).
Proof using. intros. extens. rew_index. iff; case_if; math. Qed.

    Reformulation of above, helpful for automation
Lemma index_of_index_length' : ∀ l n i,
    index n i →
    n = length l →
    index l i.
Proof using. intros. subst. rewrite¬ index_eq_index_length. Qed.
End Index.
#[global] Hint Rewrite index_eq_index_length int_index_eq index_app_eq : rew_index.

```

21.6 read, with length as *int*, as typeclass

```

Definition read_impl '{Inhab A} (l:list A) (i:int) : A :=
    If i < 0 then arbitrary else nth (abs i) l.
#[global]
Instance read_inst : ∀ '{Inhab A}, BagRead int A (list A).
Proof using. constructor. rappl (@read_impl A H). Defined.

Section Read.
Context (A : Type) (IA:Inhab A).
Implicit Types x : A.

```

```

Implicit Types  $l : \text{list } A$ .
Implicit Types  $n \ i : \text{int}$ .

Lemma read_cons_case :  $\forall \ l \ i \ v$ ,
   $(v :: l)[i] = (\text{If } i = 0 \text{ then } v \text{ else } l[i-1])$ .
Proof using IA.
  introv. simpl. unfold read_impl. case_if.
  { case_if. math. case_if  $\neg$ . math. }
  { case_if  $\neg$ .
    { subst. rewrite abs_0. rew_listx  $\neg$ . }
    { rewrite  $\neg$  abs_eq_succ_abs_minus_one. rew_listx.
      case_if. math. auto. } }
Qed.

Lemma read_cons_pos :  $\forall \ x \ l \ i$ ,
   $i > 0 \rightarrow$ 
   $(x :: l)[i] = l[i-1]$ .
Proof using IA. introv P. rewrite read_cons_case. case_if; try math. auto. Qed.

Lemma read_zero :  $\forall \ x \ l$ ,
   $(x :: l)[0] = x$ .
Proof using IA.
  intros. rewrite read_cons_case. case_if. auto.
Qed.

Lemma read_succ :  $\forall \ x \ l \ i$ ,
   $0 \leq i < \text{length } l \rightarrow$ 
   $(x :: l)[i+1] = l[i]$ .
Proof using IA.
  introv M. rewrite read_cons_case. case_if. { math. }
  fequals. math.
Qed.

Lemma read_last_case :  $\forall \ l \ i \ v$ ,
   $(l \ \& \ v)[i] = (\text{If } i = \text{length } l \text{ then } v \text{ else } l[i])$ .
Proof using IA.
  introv. simpl. unfold read_impl. case_if.
  { case_if ~; math. }
  { rewrite nth_last_case. rewrite  $\neg$  abs_eq_nat_eq. }
Qed.

Lemma read_middle :  $\forall \ i \ l1 \ l2 \ x$ ,
   $i = \text{length } l1 \rightarrow$ 
   $(l1 ++ x :: l2)[i] = x$ .
Proof using IA.
  introv M. rewrite length_eq in M. unfold read, read_inst, read_impl.
  case_if. { false; math. }

```

```

    rewrite  $\neg$  nth_middle.
Qed.
Lemma read_app :  $\forall i\ l1\ l2$ ,
   $(l1 ++ l2)[i] = (\text{If } i < \text{length } l1 \text{ then } l1[i] \text{ else } l2[i - \text{length } l1])$ .
Proof using IA.
  intros. rewrite length_eq. unfold read, read_inst, read_impl. case_if.
  { case_if. { auto. } { false; math. } }
  case_if as C'.
  { applies nth_app_l. applies lt_nat_of_lt_int. rewrite abs_nonneg; math. }
  case_if. { false; math. }
  rewrite abs_gt_minus_nat; [|math|. applies nth_app_r.
  { applies ge_nat_of_ge_int. rewrite abs_nonneg; math. }
Qed.

```

21.7 Equality between two lists from equality of length and

extensional equality of reads.

```

Lemma eq_of_extens_range :  $\forall l1\ l2$ ,
  length  $l1 = \text{length } l2 \rightarrow$ 
   $(\forall i, 0 \leq i < \text{length } l1 \rightarrow l1[i] = l2[i]) \rightarrow$ 
   $l1 = l2$ .
Proof using IA.
  introv HL HR. do 2 rewrite length_eq in HL.
  unfold read, read_inst, read_impl in HR.
  applies  $\neg$  LibList.eq_of_extens  $l1\ l2$ .
  { intros  $n\ L$ . forwards  $M$ : (rm HR) (nat_to_Z  $n$ ). math.
    case_if. false; math. rewrite  $\neg$  abs_nat in  $M$ . }
Qed.

```

```

Lemma eq_of_extens :  $\forall l1\ l2$ ,
  length  $l1 = \text{length } l2 \rightarrow$ 
   $(\forall i, \text{index } l1\ i \rightarrow l1[i] = l2[i]) \rightarrow$ 
   $l1 = l2$ .

```

Proof using IA. intros. applies $\neg$ eq_of_extens_range. Qed.

mem

```

Lemma mem_read :  $\forall l\ i$ ,
  index  $l\ i \rightarrow$ 
  mem  $l[i]\ l$ .

```

Proof using IA.

```

  intros  $l$ . induction  $l$ ; introv  $Hi$ .

```

```

{ rew_index in Hi. rew_list × in Hi. math. }
{ rewrite read_cons_case. case_if ×. applys mem_cons_r.
  applys IHL. rew_index in *. math. }
Qed.

End Read.

Lemma mem_inv_read : ∀ A (IA:Inhab A) (l:list A) x,
  mem x l →
  ∃ i, index l i ∧ x = l[i].
Proof using.
  intros. gen x. induction l; introv Hx; inversion Hx; subst.
  { ∃ 0. splits ×. rewrite × read_zero. }
  { lets × (i,(Hi&->)): (IHL x).      rew_index in *. ∃ (i+1). split ×. rewrite ×
read_succ. }
Qed.

Lemma Forall2_inv_read : ∀ A (IA:Inhab A) B (IB:Inhab B) P (xs:list A) (ys:list B) i,
  Forall2 P xs ys →
  index xs i →
  P xs[i] ys[i].
Proof using.
  induction xs; introv HF Hi; rew_index in Hi; rew_list in Hi.
  { math. }
  { lets (x&xs'&M1&M2&M3): Forall2_cons_l_inv HF.      subst. do 2 rewrite read_cons_case.
case_if. { auto. } { applys × IHxs. } }
Qed.

Global Opaque read_inst.

#[global] Hint Rewrite read_zero : rew_list.
#[global] Hint Rewrite read_zero : rew_listx.

```

21.8 update, with index as *int*, as typeclass

```

Definition update_impl A (l:list A) (i:int) (v:A) : list A :=
  If i < 0 then l else LibList.update (abs i) v l.

#[global]
Instance update_inst : ∀ A, BagUpdate int A (list A).
Proof using. constructor. rapply (@update_impl A). Defined.

Section Update.
Transparent index_inst read_inst update_inst.
Context (A : Type) '{IA:Inhab A}.
Implicit Types x v w : A.
Implicit Types l : list A.

```



```

Implicit Types  $i\ j : \text{int}$ .
Lemma length_update :  $\forall\ l\ i\ v$ ,
  length ( $l[i:=v]$ ) = length  $l$ .
Proof using.
  intros. unfold update_inst, update_impl, length, update. simpl.
  case-if. math. rewrite  $\neg$  length_update.
Qed.

Lemma index_update_eq :  $\forall\ l\ i\ j\ v$ ,
  index ( $l[j:=v]$ )  $i$  = index  $l\ i$ .
Proof using. intros. rewrite index_eq_index_length in *. rewrite  $\neg$  length_update.
Qed.

Lemma index_update :  $\forall\ l\ i\ j\ v$ ,
  index  $l\ i \rightarrow$ 
  index ( $l[j:=v]$ )  $i$ .
Proof using. intros. rewrite  $\neg$  index_update_eq. Qed.

Lemma read_update_case :  $\forall\ l\ i\ j\ v$ ,
  index  $l\ j \rightarrow$ 
   $l[i:=v][j] = (\text{If } i = j \text{ then } v \text{ else } l[j])$ .
Proof using.
  introv. unfold index_inst, index_impl, update_inst, update_impl, update,
    read_inst, read_impl. simpl. introv  $N$ . rewrite int_index_eq in  $N$ .
  case-if. math.
  case-if. case-if. auto. case-if.
  subst. rewrite  $\neg$  nth_update_same.
  rewrite  $\neg$  nth_update_neq.
Qed.

Lemma read_update_same :  $\forall\ l\ i\ v$ ,
  index  $l\ i \rightarrow$ 
  ( $l[i:=v]$ ) [ $i$ ] =  $v$ .
Proof using. introv  $N$ . rewrite  $\neg$  read_update_case. case-if  $\neg$ . Qed.

Lemma read_update_neq :  $\forall\ l\ i\ j\ v$ ,
  index  $l\ j \rightarrow$ 
  ( $i \neq j$ )  $\rightarrow$ 
  ( $l[i:=v]$ ) [ $j$ ] =  $l[j]$ .
Proof using. introv  $N$ . rewrite  $\neg$  read_update_case. case-if; auto_false  $\neg$ . Qed.

End Update.

Section UpdateNoInhab.
Transparent index_inst read_inst update_inst.
Context ( $A : \text{Type}$ ).
Implicit Types  $x\ v\ w : A$ .

```

```

Implicit Types  $l : \text{list } A$ .
Implicit Types  $i\ j : \text{int}$ .

Lemma update_zero :  $\forall v\ x\ l$ ,
   $(x :: l)[0 := v] = v :: l$ .
Proof using.
  intros. unfold update, update_inst, update_impl.
  case_if. false; math. rewrite  $\neg$  update_zero.
Qed.

Lemma update_cons_pos :  $\forall n\ v\ x\ l$ ,
   $n > 0 \rightarrow$ 
   $(x :: l)[n := v] = x :: (l[(n-1) := v])$ .
Proof using.
  introv N. unfold update, update_inst, update_impl.
  do 2 (case_if; try solve [ false; math ]).
  rewrite  $\neg$  update_cons_pos.
  { fequals_rec. rewrite  $\leftarrow$  abs_gt_minus_nat. fequals. math. }
Qed.

Lemma update_cons_case :  $\forall x\ l\ i\ z$ ,
  index  $(x :: l)\ i \rightarrow$ 
   $(x :: l)[i := z] = \text{If } i = 0 \text{ then } z :: l \text{ else } x :: (l[i-1 := z])$ .
Proof using.
  introv H. case_if.
  { subst. rewrite  $\times$  update_zero. }
  { rewrite  $\times$  update_cons_pos. rew_index in H. math. }
Qed.

Lemma update_same :  $\forall (IA : \text{Inhab } A)\ l\ i$ ,
  index  $l\ i \rightarrow$ 
   $l[i := l[i]] = l$ .
Proof using.
  intros IA l. induction l; intros i Hi.
  { rew_index in Hi. rew_list in Hi. false. math. }
  { rew_index in *. rewrite  $\times$  update_cons_case. case_if.
    { fequals. subst. rewrite  $\times$  read_zero. }
    { fequals. rewrite  $\times$  read_cons_pos. math. } }
Qed.

Lemma update_update_same :  $\forall l\ i\ v\ w$ ,
  index  $l\ i \rightarrow$ 
   $l[i := v][i := w] = l[i := w]$ .
Proof using.
  intros. asserts IA2: (Inhab A). typeclass.
  eapply eq_of_extens; repeat rewrite length_update. { auto. }

```

```

intros  $k$   $Hk$ . repeat rewrite index_update_eq in  $Hk$ .
repeat rewrite read_update_case by eauto using index_update.
case_if  $\neg$ .

```

Qed.

```

Lemma update_update_neq :  $\forall l\ i\ j\ v\ w$ ,
  index  $l\ i \rightarrow$ 
  index  $l\ j \rightarrow$ 
   $i \neq j \rightarrow$ 
   $l[i:=v][j:=w] = l[j:=w][i:=v]$ .

```

Proof using.

```

intros. asserts  $IA$ : (Inhab  $A$ ). typeclass.
applies eq_of_extens; repeat rewrite length_update. { auto. }
intros  $k$   $Hk$ . repeat rewrite index_update_eq in  $Hk$ .
repeat rewrite read_update_case by eauto using index_update.
repeat case_if  $\neg$ .

```

Qed.

```

Lemma update_app_l :  $\forall l1\ i\ l2\ v$ ,
   $0 \leq i < \text{length } l1 \rightarrow$ 
   $(l1 ++ l2)[i:=v] = (l1[i:=v]) ++ l2$ .

```

Proof using.

```

introv  $N$ . asserts  $IA$ : (Inhab  $A$ ). typeclass.
unfold LibContainer.update, update_inst, update_impl.
rewrite length_eq in  $N$ . case_if  $\neg$ . rewrite  $\neg$  update_app_l.

```

Qed.

```

Lemma update_app_r :  $\forall l2\ j\ l1\ i\ ij\ v$ ,
   $i = \text{length } l1 \rightarrow$ 
   $0 \leq j \rightarrow$ 
   $ij = i + j \rightarrow$ 
   $(l1 ++ l2)[ij:=v] = l1 ++ (l2[j:=v])$ .

```

Proof using.

```

intros. asserts  $IA$ : (Inhab  $A$ ). typeclass. subst  $ij$ .
unfold LibContainer.update, update_inst, update_impl.
do 2 (case_if; [ math | ]).
rewrite Zabs2Nat.inj_add; try math. subst  $i$ .
rewrite length_eq. rewrite abs_nat.
rewrite  $\neg$  update_app_r. fequals_rec. math.

```

Qed.

```

Lemma update_middle :  $\forall i\ l1\ l2\ v\ w$ ,
   $i = \text{length } l1 \rightarrow$ 
   $(l1 ++ w :: l2)[i := v] = l1 \& v ++ l2$ .

```

Proof using.

```

    introv E. rewrites¬ (» update_app_r 0 i).
    rewrite update_zero. rew_list¬.
Qed.
Lemma Forall_update : ∀ P l i x,
  Forall P l →
  P x →
  index l i →
  Forall P (l[i:=x]).
Proof using.
  introv. gen i. induction l; introv HL Hx Hi; rew_listx.
  { rew_index in Hi. rew_list in *. false. math. }
  { rew_index in *. rew_listx × in HL. destruct HL.
    rewrite × update_cons_case. case_if*; rew_listx ×. }
Qed.
End UpdateNoInhab.
#[global] Hint Rewrite update_same length_update : rew_listx.
#[global] Hint Rewrite index_update_eq : rew_array.
#[global] Hint Rewrite index_update_eq : rew_index.
Global Opaque update_inst.

```

21.9 make, with length as *int*

Definition make *A* (*n*:int) (*v*:*A*) : list *A* :=
 If *n* < 0 then arbitrary else make (abs *n*) *v*.

Section Make.

Transparent index_inst read_inst.

Context (*A* : Type) {*IA*:Inhab *A*}.

Implicit Types *x v* : *A*.

Implicit Types *l* : list *A*.

Implicit Types *n i* : int.

Lemma length_make : ∀ *n v*,

n ≥ 0 →

length (make *n v*) = *n*.

Proof using.

introv *N*. unfold make. case_if. math.

unfold length. rewrite LibList.length_make.

rewrite¬ abs_nonneg.

Qed.

Lemma index_make : ∀ *n i v*,

index *n i* →

```

    index (make  $n$   $v$ )  $i$ .
Proof using.
  intros  $H$ . rewrite index_eq_index_length.
  rewrite int_index_eq in  $H$ .
  rewrite  $\neg$  length_make.
Qed.

Lemma index_make_eq :  $\forall$   $n$   $i$   $v$ ,
   $n \geq 0 \rightarrow$ 
  index (make  $n$   $v$ )  $i$  = index  $n$   $i$ .
Proof using.
  intros. rewrite index_eq_index_length.
  rewrite  $\times$  length_make.
Qed.

Lemma read_make :  $\forall$   $i$   $n$   $v$ ,
  index  $n$   $i \rightarrow$ 
  (make  $n$   $v$ ) [ $i$ ] =  $v$ .
Proof using.
  intros  $N$ . rewrite int_index_eq in  $N$ .
  unfold make, read_inst, read_impl.
  case_if. math. simpl. case_if. math.
  applies nth_make. forwards: lt_abs_abs  $i$   $n$ ; try math.
Qed.

Lemma make_zero :  $\forall$   $v$ ,
  make 0  $v$  = nil.
Proof using.
  intros. unfold make. case_if. { false; math. }
  asserts_rewrite (abs 0 = 0%nat).
  { applies eq_nat_of_eq_int. rewrite abs_nonneg; math. }
  auto.
Qed.

End Make.

Section MakeNoInhab.
Transparent index_inst read_inst.
Context ( $A$  : Type).
Implicit Types  $x$   $v$  :  $A$ .
Implicit Types  $l$  : list  $A$ .
Implicit Types  $n$   $i$  : int.

Lemma make_succ_l :  $\forall$   $n$   $v$ ,
   $n \geq 0 \rightarrow$ 
  make ( $n+1$ )  $v$  =  $v ::$  make  $n$   $v$ .
Proof using.

```

```

  introv N. unfold make. case_if; case_if; try solve [false;math].
  rewrite ← LibList.make_succ. fequal.
  { rewrite succ_abs_eq_abs_one_plus. fequal. math. math. }
Qed.

```

```

Lemma make_succ_r : ∀ n v,
  n ≥ 0 →
  make (n+1) v = make n v & v.

```

Proof using.

```

  intros. asserts IA: (Inhab A). applys Inhab_of_val v.
  applys eq_of_extens_range.
  { rewrite length_make. rew_list. rewrite length_make.
    math. math. math. }
  { intros i Ei. rewrite length_make in Ei; [| math ].
    rewrite read_make; [| rewrite int_index_eq; math ].
    rewrite read_app. rewrite length_make; [|math|.
    case_if as C.
    { rewrite read_make. auto. rewrite int_index_eq; math. }
    { math_rewrite (i-n = 0). rewrite ¬ read_zero. } }

```

Qed.

```

Lemma make_eq_cons_make_pred : ∀ n v,
  0 < n →
  make n v = v :: (make (n-1) v).

```

Proof using.

```

  intros. math_rewrite (n = (n-1)+1). rewrite make_succ_l.
  fequals_rec. math. math.

```

Qed.

End MakeNoInhab.

```

#[global] Hint Rewrite length_make read_make : rew_listp.
#[global] Hint Rewrite index_make_eq : rew_array.
#[global] Hint Rewrite length_make index_make_eq : rew_index.

```

Global Opaque make.

21.10 *LibList.rev* interactions with **LibListZ** operations

Section Rev.

Variables (*A* : Type).

Implicit Types *x* : *A*.

Implicit Types *l* : **list** *A*.

```

Lemma length_rev : ∀ l,
  length (rev l) = length l.

```

```

Proof using. intros. unfold length. rew_list¬. Qed.

Lemma index_rev_eq :  $\forall l i,$ 
  index (rev l) i = index l i.
Proof using.
  extens. intros.
  now rewrite index_eq_index_length, length_rev, index_eq_index_length.
Qed.

End Rev.

#[global] Hint Rewrite length_rev : rew_list.
#[global] Hint Rewrite length_rev : rew_listx.
#[global] Hint Rewrite index_rev_eq : rew_index.

```

21.11 *LibList.map*: map interactions with LibListZ operations

```

Section Map.
Transparent index_inst read_inst update_inst.
Implicit Types n i : int.

Lemma length_map :  $\forall A B (l:\text{list } A) (f:A \rightarrow B),$ 
  length (map f l) = length l.
Proof using. intros. unfold length. rewrite¬ length_map. Qed.

Lemma index_map_eq :  $\forall A B (l:\text{list } A) i (f:A \rightarrow B),$ 
  index (map f l) i = index l i.
Proof using. intros. rewrite index_eq_inbound in *. rewrite¬ length_map. Qed.

Lemma index_map :  $\forall A B (l:\text{list } A) i (f:A \rightarrow B),$ 
  index l i  $\rightarrow$  index (map f l) i.
Proof using. intros. rewrite¬ index_map_eq. Qed.

Lemma map_make :  $\forall A B (f:A \rightarrow B) (n:\text{int}) (v:A),$ 
   $n \geq 0 \rightarrow$ 
  map f (make n v) = make n (f v).
Proof using.
  Transparent make.
  intros. unfold make. case_if. { false; math. }
  applys map_make.
Qed.

Lemma map_update :  $\forall A B (l:\text{list } A) (i:\text{int}) (x:A) (f:A \rightarrow B),$ 
  index l i  $\rightarrow$ 
  map f (l[i := x]) = (map f l) [i := f x].
Proof using.

```

```

    introv H. rewrite index_eq_inbound in H.
    unfold update_inst, update_impl, update. simpl.
    case-if. { false; math. }
    { applys LibList.map_update.
      { applys lt_nat_of_lt_int. rewrite abs_nonneg; math. } }
Qed.

Lemma read_map :  $\forall \{IA:Inhab\ A\} \{IB:Inhab\ B\} (l:list\ A) (i:int) (f:A \rightarrow B),$ 
  index  $l\ i \rightarrow$ 
  (map  $f\ l$ ) [ $i$ ] =  $f\ (l[i])$ .
Proof using.
  introv H. rewrite index_eq_inbound in H.
  unfold read_inst, read_impl. simpl. case-if.
  { false; math. }
  { rewrite nth_map. auto.
    applys lt_nat_of_lt_int. rewrite abs_nonneg; math. }
Qed.

End Map.

#[global] Hint Rewrite length_map index_map_eq : rew_listx.
#[global] Hint Rewrite length_map index_map_eq : rew_index.

```

21.12 *LibList.filter* interactions with **LibListZ** operations

```

Section Filter.
Variables (A : Type).
Implicit Types x : A.
Implicit Types l : list A.
Implicit Types P : A  $\rightarrow$  Prop.

Lemma length_filter :  $\forall\ l\ P,$ 
  length (filter  $P\ l$ )  $\leq$  length  $l$ .
Proof using.
  intros. unfolds length. forwards~: LibList.length_filter  $l\ P$ .
Qed.

Lemma filter_length_two_disjoint :  $\forall\ (P\ Q : A \rightarrow Prop)\ l,$ 
  ( $\forall\ x, \text{mem } x\ l \rightarrow P\ x \rightarrow Q\ x \rightarrow \text{False}$ )  $\rightarrow$ 
  length (filter  $P\ l$ ) + length (filter  $Q\ l$ )  $\leq$  length  $l$ .
Proof using.
  intros. unfolds length.
  forwards~: LibList.filter_length_two_disjoint.
Qed.

Lemma filter_length_partition :  $\forall\ P\ l,$ 

```



```

    length (filter (fun x => P x) l)
+ length (filter (fun x => ¬ P x) l)
≤ length l.
Proof using.
  intros. unfolds length. forwards~: LibList.filter_length_partition P l.
Qed.
Lemma length_filter_eq_mem_ge_one : ∀ x l,
  mem x l →
  length (filter (= x) l) ≥ 1.
Proof using.
  intros. unfolds length. forwards~: LibList.length_filter_eq_mem_ge_one.
Qed.
End Filter.

```

21.13 *LibList.remove* interactions with **LibListZ** operations

```

Section Remove.
Variables (A : Type).
Implicit Types a x : A.
Implicit Types l : list A.
Lemma length_remove : ∀ l a,
  length (LibList.remove a l) ≤ length l.
Proof using. intros. rewrite remove_as_filter. applies length_filter. Qed.
Lemma length_remove_mem : ∀ x l,
  mem x l →
  length (LibList.remove x l) < length l.
Proof using.
  intros. unfolds length. forwards~: LibList.length_remove_mem.
Qed.
End Remove.

```

21.13.1 Take, with an *int* as the number of elements

```

Definition take A (n:int) (l:list A) : list A :=
  LibList.take (to_nat n) l.
Section Take.
Variables (A : Type).
Implicit Types n : int.

```

```

Implicit Types  $x : A$ .
Implicit Types  $l : \text{list } A$ .
Lemma take_nil :  $\forall n$ ,
  take  $n$  (@nil  $A$ ) = nil.
Proof using. intros. unfold take. apply LibList.take_nil. Qed.
Lemma take_zero :  $\forall l$ ,
  take 0  $l$  = nil.
Proof using. auto. Qed.
Lemma take_succ :  $\forall x \ l \ n$ ,
   $0 \leq n \rightarrow$ 
  take  $(n+1)$  ( $x :: l$ ) =  $x ::$  (take  $n$   $l$ ).
Proof using.
  intros. unfold take. rew_to_nat_nonneg $\neg$ .
  rewrite Nat.add_1_r. apply LibList.take_succ.
Qed.
Definition take_cons := take_succ.
Lemma take_cons_pos :  $\forall x \ l \ n$ ,
   $(n > 0) \rightarrow$ 
  take  $n$  ( $x :: l$ ) =  $x ::$  (take  $(n-1)$   $l$ ).
Proof using.
  intros. unfold take. rew_to_nat_nonneg $\neg$ .
  rewrite $\neg$  LibList.take_cons_pos.
Qed.
Lemma take_cons_case :  $\forall x \ l \ n$ ,
   $n \geq 0 \rightarrow$ 
  take  $n$  ( $x :: l$ ) = If  $n = 0$  then nil else  $x ::$  (take  $(n-1)$   $l$ ).
Proof using.
  introv Hn. case_if.
  { subst. applies take_zero. }
  { applies $\times$  take_cons_pos. math. }
Qed.
Lemma take_pos_last :  $\forall (IA : \text{Inhab } A) \ l \ i$ ,
  index  $l$   $(i-1) \rightarrow$ 
  take  $i$   $l$  = take  $(i-1)$   $l$  &  $l[i-1]$ .
Proof using.
  introv Hi. gen i. induction l; intros; rew_index in Hi; rew_list in Hi.
  { math. }
  { rewrite take_cons_pos; try math.
    rewrite take_cons_case; try math. case_if as C.
    { rew_list. rewrite C. rewrite take_zero. rewrite $\times$  read_zero. }
    { rew_list. fequals. rewrite IHL; [|rew_index; math]. }
```

```

      fequals. rewrite × read_cons_pos. math. } }
Qed.

Lemma take_neg : ∀ n l,
  n ≤ 0 →
  take n l = nil.
Proof using. intros. unfold take. rewrite ¬ to_nat_neg. Qed.

Lemma take_ge : ∀ n l,
  (n ≥ length l) →
  take n l = l.
Proof using.
  intros. unfold take, length in *. applies ¬ LibList.take_ge.
Qed.

Lemma take_is_prefix : ∀ n l,
  ∃ q, l = take n l ++ q.
Proof using.
  intros. unfold take. applies ¬ LibList.take_is_prefix.
Qed.

Lemma take_app_l : ∀ n l l',
  (n ≤ length l) →
  take n (l ++ l') = take n l.
Proof using.
  intros. tests: (0 ≤ n).
  { unfold take, length in *.
    applies ¬ LibList.take_app_l. }
  { rewrite !take_neg; auto; math. }
Qed.

Lemma take_app_r : ∀ n l l',
  (n ≥ length l) →
  take n (l ++ l') = l ++ take (n - length l) l'.
Proof using.
  intros. unfold take, length in *. rew_to_nat_nonneg ¬.
  applies ¬ LibList.take_app_r.
Qed.

Lemma take_prefix_length : ∀ l l',
  take (length l) (l ++ l') = l.
Proof using.
  intros. unfold take, length. rew_to_nat_nonneg ¬.
  applies ¬ LibList.take_prefix_length.
Qed.

Lemma take_full_length : ∀ l,
  take (length l) l = l.

```

Proof using.

```
intros. unfold take, length. rew_to_nat_nonneg⊢.  
apply LibList.take_full_length.
```

Qed.

End Take.

```
#[global] Hint Rewrite take_nil take_zero take_succ : rew_listx.
```

21.13.2 Drop, with an *int* as the number of elements.

Definition drop A (n :int) (l :list A) : list A :=
LibList.drop (to_nat n) l .

Section Drop.

Variables (A : Type).

Implicit Types n : int.

Implicit Types x : A .

Implicit Types l : list A .

Lemma drop_nil : $\forall n$,
drop n (@nil A) = nil.

Proof using. intros. unfold drop. apply LibList.drop_nil. Qed.

Lemma drop_zero : $\forall l$,
drop 0 l = l .

Proof using. auto. Qed.

Lemma drop_succ : $\forall x\ l\ n$,
 $0 \leq n \rightarrow$
drop ($n+1$) ($x::l$) = (drop $n\ l$).

Proof using.

```
intros. unfold drop. rew_to_nat_nonneg⊢.  
rewrite Nat.add_1_r. apply LibList.drop_succ.
```

Qed.

Definition drop_cons := drop_succ.

Lemma drop_neg : $\forall n\ l$,
 $n \leq 0 \rightarrow$
drop $n\ l$ = l .

Proof using. intros. unfold drop. rewrite⊢ to_nat_neg. Qed.

Lemma drop_cons_pos : $\forall x\ l\ n$,
 $(n > 0) \rightarrow$
drop n ($x::l$) = drop ($n-1$) l .

Proof using.

```
intros. unfold drop. rew_to_nat_nonneg⊢.
```

```

    apply¬ LibList.drop_cons_pos.
Qed.

Lemma drop_is_suffix : ∀ n l,
  ∃ q, l = q ++ drop n l.
Proof using.
  intros. unfold drop. apply LibList.drop_is_suffix.
Qed.

Lemma drop_app_l : ∀ n l l',
  (n ≤ length l) →
  drop n (l ++ l') = drop n l ++ l'.
Proof using.
  intros. tests: (0 ≤ n).
  { unfold drop, length in *.
    apply LibList.drop_app_l. rewrites¬ to_nat_le_nat_le. }
  { rewrite !drop_neg; auto; math. }
Qed.

Lemma drop_app_r : ∀ n l l',
  (n ≥ length l) →
  drop n (l ++ l') = drop (n - length l) l'.
Proof using.
  intros. unfold drop, length in *. rewrites¬ to_nat_nonneg¬.
  apply LibList.drop_app_r. rewrites¬ to_nat_ge_nat_ge.
Qed.

Lemma drop_app_length : ∀ l l',
  drop (length l) (l ++ l') = l'.
Proof using.
  intros. unfold drop, length. rewrites¬ to_nat_nonneg¬.
  apply LibList.drop_app_length.
Qed.

Lemma drop_at_length : ∀ l,
  drop (length l) l = nil.
Proof using.
  intros. unfold drop, length. rewrites¬ to_nat_nonneg¬.
  apply LibList.drop_at_length.
Qed.

Lemma drop_drop : ∀ l n m,
  n ≥ 0 →
  m ≥ 0 →
  drop n (drop m l) = drop (n+m) l.
Proof using.
  introv Hn Hm.

```

```

rewrite× ← (to_nat_nonneg Hn).
rewrite× ← (to_nat_nonneg Hm).
unfold drop.
applies_eq LibList.drop_drop. fequals. math.
Qed.

End Drop.

#[global] Hint Rewrite drop_at_length drop_nil drop_zero drop_succ : rew_listx.
#[global] Hint Rewrite drop_drop : rew_listp.
#[global] Hint Rewrite drop_nil drop_zero drop_succ : rew_listx.

```

21.13.3 Take and drop decomposition of a list

Section TakeAndDrop.

Variables (*A* : Type).

Implicit Types *x* : *A*.

Implicit Types *l* : list *A*.

Lemma take_app_drop_spec : $\forall n\ l\ f\ r,$

```

f = take n l →
r = drop n l →
( $0 \leq n \leq \text{length } l \rightarrow$ 
   $l = f ++ r$ 
   $\wedge \text{length } f = n$ 
   $\wedge \text{length } r = \text{length } l - n$ )  $\wedge$ 
( $n \leq 0 \rightarrow$ 
   $f = \text{nil} \wedge r = l$ ).

```

Proof using.

```

intros ? ?. split.
{ intros (? & Hn). unfold take, drop, length in *.
  forwards¬ (? & ? & Hlenrest): @LibList.take_app_drop_spec (to_nat n) l f r. }
{ intros. rewrites¬ take_neg in *. rewrites¬ drop_neg in *. }

```

Qed.

Lemma take_spec : $\forall n\ l,$

```

 $0 \leq n \leq \text{length } l \rightarrow$ 
 $\exists l', \text{length } (\text{take } n\ l) = n$ 
 $\wedge l = (\text{take } n\ l) ++ l'$ .

```

Proof using. *intros E. forwards*× (*E1*&_): take_app_drop_spec. *forwards**: *E1*. Qed.

Lemma length_take_nonneg : $\forall n\ l,$

```

 $0 \leq n \leq \text{length } l \rightarrow$ 
length (take n l) = n.

```

Proof using. *intros E. forwards*¬ (*l*'&*N*&*M*): take_spec *n l*. Qed.

Lemma length_take : $\forall n\ l,$
 $n \leq \text{length } l \rightarrow$
 $\text{length } (\text{take } n\ l) = \text{Z.max } 0\ n.$

Proof using.

intros. *tests*: $(0 \leq n).$
 $\{ \text{rewrite} \neg \text{length\_take\_nonneg.} \}$
 $\{ \text{rewrite} \neg \text{take\_neg. rewrite} \neg \text{Z.max\_l.} \}$

Qed.

Lemma read_take : $\forall (IA:\text{Inhab } A)\ l\ s\ i,$
 $s \leq \text{length } l \rightarrow$
 $\text{index } s\ i \rightarrow$
 $(\text{take } s\ l)[i] = l[i].$

Proof using. *introv Hs Hi. rew_index in **.

lets $(l' \& Hl \& Hl')$: $\text{take\_spec } s\ l. \{ \text{math.} \}$
 $\text{rewrite } Hl' \text{ at } 2. \text{rewrite read\_app. case\_if} \times.$
 $\{ \text{false. rewrite length\_take\_nonneg in } C; \text{try math.} \}$

Qed.

Lemma drop_spec : $\forall n\ l,$
 $0 \leq n \leq \text{length } l \rightarrow$
 $\exists l', \text{length } l' = n$
 $\wedge l = l' ++ (\text{drop } n\ l).$

Proof using. *introv E. forwards* $\times (E1 \& \_): \text{take\_app\_drop\_spec. forwards}^*: E1.$ Qed.

Lemma read_drop : $\forall (IA:\text{Inhab } A)\ l\ (s\ i:\text{int}),$
 $0 \leq s \leq \text{length } l \rightarrow$
 $\text{index } (\text{length } l - s)\ i \rightarrow$
 $(\text{drop } s\ l)[i] = l[s+i].$

Proof using.

*introv Hs Hi. rew_index in **.
lets $(l' \& Hl \& Hl')$: $\text{drop\_spec } s\ l. \{ \text{math.} \}$
 $\text{rewrite } Hl' \text{ at } 2. \text{rewrite read\_app. case\_if} \times.$
 $\{ \text{false. math.} \} \{ \text{fequals. math.} \}$

Qed.

Lemma length_drop_nonneg : $\forall n\ l,$
 $0 \leq n \leq \text{length } l \rightarrow$
 $\text{length } (\text{drop } n\ l) = \text{length } l - n.$

Proof using.

introv E. forwards $\neg (l' \& N \& M): \text{drop\_spec } n\ l.$
 $\text{pattern } l \text{ at } 2. \text{rewrite } M. \text{rew\_list. math.}$

Qed.

Lemma length_drop : $\forall n\ l,$
 $n \leq \text{length } l \rightarrow$

```

length (drop  $n$   $l$ ) = Z.min (length  $l$ ) (length  $l$  -  $n$ ).
Proof using.
  intros. tests: ( $0 \leq n$ ).
  { rewrite  $\neg$  length_drop_nonneg. }
  { rewrite  $\neg$  drop_neg. }
Qed.

Lemma list_eq_take_app_drop :  $\forall n\ l$ ,
   $n \leq \text{length } l \rightarrow$ 
  take  $n\ l$  ++ drop  $n\ l$  =  $l$ .
Proof using.
  introv  $H$ . tests: ( $0 \leq n$ ).
  { forwards*: take_app_drop_spec  $n\ l$ . }
  { rewrite  $\neg$  take_neg. rewrite  $\neg$  drop_neg. }
Qed.

Lemma mem_drop_inv :  $\forall l\ i\ x$ ,
  mem  $x$  (drop  $i\ l$ )  $\rightarrow$ 
   $0 \leq i \leq \text{length } l \rightarrow$ 
  mem  $x\ l$ .
Proof using.
  introv  $Hx\ Hi$ . rewrites  $\times \leftarrow (\gg \text{list\_eq\_take\_app\_drop } i)$ .
  apply  $\times$  mem_app_r.
Qed.

End TakeAndDrop.

Arguments take_app_drop_spec [ $A$ ].
Arguments take_spec [ $A$ ].
Arguments drop_spec [ $A$ ].

#[global] Hint Rewrite read_take read_drop length_take length_drop : rew_listp.

```

21.13.4 sum, for lists of int values

```

Definition sum ( $l$ :list int) : int :=
  fold_right Z.add 0  $l$ .

Lemma sum_nil :
  sum nil = 0.
Proof using. rew_listx  $\times$ . Qed.

Lemma sum_cons :  $\forall (l$ :list int) ( $x$ :int),
  sum ( $x::l$ ) =  $x$  + sum  $l$ .
Proof using. rew_listx  $\times$ . Qed.

#[global] Hint Rewrite sum_nil sum_cons : rew_listx.

Lemma sum_app :  $\forall (l1\ l2$ :list int),

```


$\text{sum } (l1 ++ l2) = \text{sum } l1 + \text{sum } l2.$
 Proof using. induction $l1$; intros $l2$; unfold sum; *rew_listx* ×.
 rewrite × *IHL1*. unfold sum. *math*.
 Qed.

#[*global*] Hint Rewrite sum_nil sum_cons sum_app : *rew_listx*.

Lemma length_concat_eq_sum : $\forall A (l:\text{list } (A)),$
 $\text{length } (\text{concat } l) = \text{sum } (\text{map } (\text{length } (A:=A)) l).$
 Proof using. intros. induction l ; *rew_listx*; *math*. Qed.

21.13.5 count, returning an int

Definition count $A (P: A \rightarrow \text{Prop}) (l: \text{list } A): \text{int} :=$
 $\text{abs } (\text{LibList.count } P l).$

Section Count.

Variables $A : \text{Type}.$

Implicit Types $l : \text{list } A.$

Implicit Types $P : A \rightarrow \text{Prop}.$

Ltac *gowith* $L :=$
 let $H := \text{fresh in}$
 $\text{lets } H: L;$
 repeat (let $x := \text{fresh in intro } x$; *specializes* $H x$);
 unfold count in *; repeat *rewrites* $\neg \text{abs\_nonneg}$ in *;
 try solve [
 first [*rewrites* $\neg H$ in * | *forwards* $\sim: H$];
 repeat *case_if* $\neg$
].

Lemma count_nil : $\forall P,$
 $\text{count } P \text{ nil} = 0.$

Proof using. auto. Qed.

Lemma count_one : $\forall P x,$
 $\text{count } P (x::\text{nil}) = \text{If } P x \text{ then } 1 \text{ else } 0.$

Proof using. *gowith* LibList.count_one. Qed.

Lemma count_cons : $\forall P l x,$
 $\text{count } P (x::l) = \text{count } P l + \text{If } P x \text{ then } 1 \text{ else } 0.$

Proof using. *gowith* LibList.count_cons. Qed.

Lemma count_app : $\forall P l1 l2,$
 $\text{count } P (l1 ++ l2) = \text{count } P l1 + \text{count } P l2.$

Proof using. *gowith* LibList.count_app. Qed.

Lemma count_last : $\forall P l x,$

$\text{count } P (l \& x) = \text{count } P l + \text{If } P x \text{ then } 1 \text{ else } 0.$
 Proof using. *gowith* LibList.count_last. Qed.

Lemma count_rev : $\forall P l,$
 $\text{count } P (\text{rev } l) = \text{count } P l.$
 Proof using. *gowith* LibList.count_rev. Qed.

Lemma count_eq_length_filter : $\forall P l,$
 $\text{count } P l = \text{length } (\text{filter } P l).$
 Proof using. *gowith* LibList.count_eq_length_filter. Qed.

Lemma count_nonneg : $\forall P l,$
 $0 \leq \text{count } P l.$
 Proof using. intros. unfold count. *math*. Qed.

Lemma count_le_length : $\forall P l,$
 $\text{count } P l \leq \text{length } l.$
 Proof using. *gowith* LibList.count_le_length. Qed.

Lemma Forall_eq_count_eq_length : $\forall P l,$
Forall $P l = (\text{count } P l = \text{length } l).$
 Proof using.
 intros. rewrite LibList.Forall_eq_count_eq_length, length_eq.
 unfold count. rewrite abs_nonneg; *math*.
 Qed.

Lemma count_Forall : $\forall P l,$
Forall $P l \rightarrow$
 $\text{count } P l = \text{length } l.$
 Proof using. *introv*. rewrite $\neg$ Forall_eq_count_eq_length. Qed.

Lemma Forall_of_count_eq_length : $\forall P l,$
 $\text{count } P l = \text{length } l \rightarrow$
Forall $P l.$
 Proof using. *introv*. rewrite $\neg$ Forall_eq_count_eq_length. Qed.

Lemma Forall_not_eq_count_eq_zero : $\forall P l,$
Forall $(\text{fun } x \Rightarrow \neg P x) l = (\text{count } P l = 0).$
 Proof using.
 intros. rewrite LibList.Forall_not_eq_count_eq_zero.
 unfold count. rewrite abs_nonneg; *math*.
 Qed.

Lemma Forall_not_of_count_eq_zero : $\forall P l,$
 $\text{count } P l = 0 \rightarrow$
Forall $(\text{fun } x \Rightarrow \neg P x) l.$
 Proof using. *introv*. rewrite $\neg$ Forall_not_eq_count_eq_zero. Qed.

Lemma count_eq_zero_of_Forall_not : $\forall P l,$

```

Forall (fun  $x \Rightarrow \neg P\ x$ )  $l \rightarrow$ 
  count  $P\ l = 0$ .
Proof using. intros. rewrite  $\neg$  Forall_not_eq_count_eq_zero. Qed.

Lemma Exists_eq_count_pos :  $\forall P\ l$ ,
  Exists  $P\ l = (\text{count } P\ l > 0)$ .
Proof using.
  intros. rewrite LibList.Exists_eq_count_pos.
  unfold count. rewrite abs_nonneg; math.
Qed.

Lemma Exists_of_count_pos :  $\forall P\ l$ ,
  count  $P\ l > 0 \rightarrow$ 
  Exists  $P\ l$ .
Proof using. intros. rewrite  $\neg$  Exists_eq_count_pos. Qed.

Lemma count_pos_of_Exists :  $\forall P\ l$ ,
  Exists  $P\ l \rightarrow$ 
  count  $P\ l > 0$ .
Proof using. intros. rewrite  $\neg$  Exists_eq_count_pos. Qed.

Lemma count_pos_eq_exists_mem :  $\forall P\ l$ ,
  (count  $P\ l > 0$ ) = ( $\exists x$ , mem  $x\ l \wedge P\ x$ ).
Proof using.
  intros. rewrite  $\leftarrow$  LibList.count_pos_eq_exists_mem.
  unfold count. rewrite abs_nonneg; math.
Qed.

Lemma count_pos_of_mem :  $\forall x\ P\ l$ ,
  mem  $x\ l \rightarrow$ 
   $P\ x \rightarrow$ 
  count  $P\ l > 0$ .
Proof using. intros. rewrite  $\times$  count_pos_eq_exists_mem. Qed.

Lemma exists_mem_of_count_pos :  $\forall P\ l$ ,
  count  $P\ l > 0 \rightarrow$ 
   $\exists x$ , mem  $x\ l \wedge P\ x$ .
Proof using. intros. rewrite  $\times$  count_pos_eq_exists_mem. Qed.

End Count.

Opaque count.

#[global] Hint Rewrite count_nil count_cons count_app count_last count_rev
  : rew_listx.

```

21.14 `card`, with result as `nat`, as typeclass

Note that `card` produces a `nat`, whereas `length` produces an `int`. Currently, in practice, we use `LibList.length` rather than `card`.

```
Definition card_impl A (l:list A) : nat :=  
  LibList.length l.
```

```
#[global]
```

```
Instance card_inst : ∀ A, BagCard (list A).
```

```
Proof using. constructor. rapply (@card_impl A). Defined.
```

```
Global Opaque card_inst.
```

21.15 Hints for `rew_array` and `rew_array_nocase`

`rew_array_nocase` is a light normalization tactic for array

```
#[global] Hint Rewrite @read_make @length_make @length_update @read_update_same  
  : rew_array_nocase.
```

`rew_array` is a normalization tactic for array, which introduces case analyses for all read-on-update operations.

```
#[global] Hint Rewrite @read_make @length_make @length_update @read_update_same  
  @read_update_case @read_cons_case @read_last_case : rew_array.
```

```
Ltac auto_tilde ::= auto_tilde_default.
```

21.16 `binds`, with length as `int`, as typeclass

Chapter 22

Library SLF.LibMin

Set Implicit Arguments.
From *SLF* Require Import LibTactics LibLogic LibReflect LibOperation
LibRelation LibOrder LibEpsilon.
Generalizable Variables *A*.

Definition lower_bound *A* (*le*:binary *A*) (*P*:*A*→Prop) (*x*:*A*) :=
 $\forall y, P\ y \rightarrow le\ x\ y$.

Definition min_element *A* (*le*:binary *A*) (*P*:*A*→Prop) (*x*:*A*) :=
 $P\ x \wedge lower\_bound\ le\ P\ x$.

Definition mmin ‘{*Inhab A*} (*le*:binary *A*) (*P*:*A*→Prop) :=
 epsilon (min_element *le* *P*).

Definition upper_bound *A* (*le*:binary *A*) (*P*:*A*→Prop) (*x*:*A*) :=
 $\forall y, P\ y \rightarrow le\ y\ x$.

Definition max_element *A* (*le*:binary *A*) (*P*:*A*→Prop) (*x*:*A*) :=
 $P\ x \wedge upper\_bound\ le\ P\ x$.

Definition mmax ‘{*Inhab A*} (*le*:binary *A*) (*P*:*A*→Prop) :=
 epsilon (max_element *le* *P*).

Definition lub *A* (*le*:binary *A*) (*P*:*A*→Prop) (*x*:*A*) :=
 min_element *le* (upper_bound *le* *P*) *x*.

Definition glb *A* (*le*:binary *A*) (*P*:*A*→Prop) (*x*:*A*) :=
 max_element *le* (lower_bound *le* *P*) *x*.

Lemma upper_bound_inverse : $\forall A\ (le:binary\ A),$
 upper_bound *le* = lower_bound (inverse *le*).

Proof using.

extends. intros P x . *unfolds* lower_bound, upper_bound. *iff* ×.

Qed.

Lemma max_element_inverse : $\forall A (le:binary\ A) (P:A \rightarrow Prop) (x:A)$,
max_element le P x = min_element (inverse le) P x .

Proof using.

extends. unfold max_element, min_element. *rewrite* × upper_bound_inverse.

Qed.

Lemma mmax_inverse : $\forall \{Inhab\ A\} (le:binary\ A) (P:A \rightarrow Prop)$,
mmax le P = mmin (inverse le) P .

Proof using.

intros. *applies* epsilon_eq. intros x . *rewrite* × max_element_inverse.

Qed.

Definition bounded_has_minimal $A (le:binary\ A) :=$

$\forall P$,
ex $P \rightarrow$
ex (lower_bound le P) $\rightarrow$
ex (min_element le P).

Lemma mmin_spec : $\forall \{Inhab\ A\} (le:binary\ A) (P:A \rightarrow Prop) m$,
 m = mmin le $P \rightarrow$
ex $P \rightarrow$
ex (lower_bound le P) $\rightarrow$
bounded_has_minimal $le \rightarrow$
min_element le P m .

Proof using.

intros. subst. unfold mmin. *epsilon* × m .

Qed.

From *SLF* Require Import LibNat.

Lemma increment_lower_bound_nat : $\forall (P : nat \rightarrow Prop) x$,
lower_bound le P $x \rightarrow$
 $\neg P$ $x \rightarrow$
lower_bound le P $(x + 1) \% nat$.

Proof using.

introv hlo ?. intros y ?.
destruct (eq_nat_dec x y).
{ subst. tauto. }
{ *forwards*: hlo ; eauto. *nat_math*. }

Qed.

Lemma bounded_has_minimal_nat :

@bounded_has_minimal **nat** le.

Proof using.

intros P [$y ?$].

cut (

$\forall (k\ x : \mathbf{nat}), (y + 1 - x)\%nat = k \rightarrow$

lower_bound le $P\ x \rightarrow$

$\exists z, \text{min\_element le } P\ z$

). { intros ? [$x ?$]. eauto. }

induction k ; introv ? hlo .

{ false. forwards: hlo ; eauto. nat_math. }

destruct (prop_inv ($P\ x$)).

{ $\exists x$. split; eauto. }

{ eapply ($IHk\ (x + 1)\%nat$). nat_math. eauto using increment_lower_bound_nat. }

Qed.

#[global]

Hint Resolve bounded_has_minimal_nat : bounded_has_minimal.

Lemma admits_lower_bound_nat : $\forall (P : \mathbf{nat} \rightarrow \text{Prop}),$

ex (lower_bound le P).

Proof using.

$\exists 0\%nat$. unfold lower_bound. nat_math.

Qed.

#[global]

Hint Resolve admits_lower_bound_nat : admits_lower_bound.

Lemma bounded_has_maximal_nat :

@bounded_has_minimal **nat** (inverse le).

Proof using.

intros P [$y ?$] [$x\ h$].

assert ($y \leq x$).

{ forwards: h . eauto. eauto. }

assert (self_inverse: $\forall i, i \leq x \rightarrow (x - (x - i))\%nat = i$).

intros. nat_math.

forwards [z [? hz]]: (@bounded_has_minimal_nat (fun $i \Rightarrow P\ (x - i)\%nat$)).

{ $\exists (x - y)\%nat$. rewrite self_inverse by eauto. eauto. }

{ eauto using admits_lower_bound_nat. }

$\exists (x - z)\%nat$.

clear dependent y .

split; [assumption |].

```

intros y ?.
assert (y ≤ x).
  { forwards: h. eauto. eauto. }
forwards: hz (x - y)%nat.
  { rewrite self_inverse by eauto. eauto. }
unfold inverse. nat_math.
Qed.

#[global]
Hint Resolve bounded_has_maximal_nat : bounded_has_minimal.

Lemma mmin_spec_nat:
  ∀ (P: nat → Prop) m,
  m = mmin le P →
  ex P →
  P m ∧ (∀ x, P x → m ≤ x).
Proof using.
  introv E Q. applys (@mmin_spec _ _ _ P m E Q).
  applys admits_lower_bound_nat.
  applys bounded_has_minimal_nat.
Qed.

Definition MMin ‘{Inhab A} ‘{Le A} := mmin le.

Definition MMax ‘{Inhab A} ‘{Le A} := mmax le.

```


Chapter 23

Library SLF.LibSet

```
Set Implicit Arguments.
Generalizable Variables A B.
Require Import Stdlib.Classes.Morphisms. From SLF Require Import LibTactics LibLogic
LibReflect LibList
  LibOperation LibMonoid LibInt LibNat
  LibEpsilon LibRelation LibMin.
From SLF Require Export LibContainer.
```

23.1 Construction of sets as predicates

23.1.1 Basic definitions

```
Definition set (A : Type) := A → Prop.

Section Operations.
Variables (A B : Type).
Implicit Types x : A.
Implicit Types E F G : set A.

Definition set_st (P:A→Prop) : set A :=
  P.

Definition empty_impl : set A :=
  (fun _ ⇒ False).

Definition full_impl : set A :=
  (fun _ ⇒ True).

Definition single_impl x :=
  (= x).

Definition in_impl x E :=
```

$E\ x.$

Definition compl_impl : set $A \rightarrow$ set $A :=$
@pred_not A .

Definition union_impl : set $A \rightarrow$ set $A \rightarrow$ set $A :=$
@pred_or A .

Definition inter_impl : set $A \rightarrow$ set $A \rightarrow$ set $A :=$
@pred_and A .

Definition remove_impl : set $A \rightarrow$ set $A \rightarrow$ set $A :=$
fun $E\ F\ x \Rightarrow E\ x \wedge \neg F\ x$.

Definition incl_impl : set $A \rightarrow$ set $A \rightarrow$ Prop :=
@pred_incl A .

Definition disjoint_impl : set $A \rightarrow$ set $A \rightarrow$ Prop :=
fun $E\ F : set\ A \Rightarrow$ inter_impl $E\ F =$ empty_impl.

Definition list_repr_impl (E :set A) (l :list A) :=
noduplicates $l \wedge \forall x, \text{mem } x\ l \leftrightarrow E\ x$.

Definition to_list (E :set A) :=
epsilon (list_repr_impl E).

Definition to_set (xs : list A) : set $A :=$
set_st (fun $x \Rightarrow \text{mem } x\ xs$).

Definition list_covers_impl (E :set A) $L :=$
 $\forall x, E\ x \rightarrow \text{mem } x\ L$.

Definition finite (E :set A) :=
 $\exists L, \text{list\_covers\_impl } E\ L$.

Definition card_impl (E :set A) : nat :=
mmin le (fun $n \Rightarrow \exists L, \text{list\_covers\_impl } E\ L \wedge n = \text{length } L$).

Definition fold_impl (m :monoid_op B) (f : $A \rightarrow B$) (E :set A) :=
LibList.fold_right (fun $x\ acc \Rightarrow \text{monoid\_oper } m\ (f\ x)\ acc$)
(monoid_neutral m) (to_list E).

End Operations.

23.1.2 Notations to help the typechecker

Notation " $x \backslash \text{indom } E$ " := ($x \backslash \text{in } (\text{dom } E : \text{set } \_)$)
(at level 39) : *container_scope*.

Notation " $x \backslash \text{notindom } E$ " := ($x \backslash \text{notin } ((\text{dom } E) : \text{set } \_)$)
(at level 39) : *container_scope*.

23.1.3 Inhabited

```
#[global]
Instance Inhab_set :  $\forall A$ , Inhab (set  $A$ ).
Proof using. intros. apply (Inhab_of_val (@empty_impl  $A$ )). Qed.
```

23.1.4 Notation through typeclasses

```
Lemma in_inst :  $\forall A$ , BagIn  $A$  (set  $A$ ).
Proof using. constructor. exact (@in_impl  $A$ ). Defined.

#[global]
Hint Extern 1 (BagIn _ (set _))  $\Rightarrow$  apply in_inst : typeclass_instances.

#[global]
Instance empty_inst :  $\forall A$ , BagEmpty (set  $A$ ).
  constructor. apply (@empty_impl  $A$ ). Defined.

#[global]
Instance single_inst :  $\forall A$ , BagSingle  $A$  (set  $A$ ).
  constructor. rapply (@single_impl  $A$ ). Defined.

#[global]
Instance union_inst :  $\forall A$ , BagUnion (set  $A$ ).
  constructor. rapply (@union_impl  $A$ ). Defined.

#[global]
Instance inter_inst :  $\forall A$ , BagInter (set  $A$ ).
  constructor. rapply (@inter_impl  $A$ ). Defined.

#[global]
Instance remove_inst :  $\forall A$ , BagRemove (set  $A$ ) (set  $A$ ).
  constructor. rapply (@remove_impl  $A$ ). Defined.

#[global]
Instance incl_inst :  $\forall A$ , BagIncl (set  $A$ ).
  constructor. rapply (@incl_impl  $A$ ). Defined.

#[global]
Instance disjoint_inst :  $\forall A$ , BagDisjoint (set  $A$ ).
  constructor. rapply (@disjoint_impl  $A$ ). Defined.

#[global]
Instance fold_inst :  $\forall A B$ , BagFold  $B$  ( $A \rightarrow B$ ) (set  $A$ ).
  constructor. rapply (@fold_impl  $A B$ ). Defined.

#[global]
Instance card_inst :  $\forall A$ , BagCard (set  $A$ ).
  constructor. rapply (@card_impl  $A$ ). Defined.
```

Global Opaque set finite in_inst empty_inst single_inst union_inst inter_inst
remove_inst incl_inst disjoint_inst card_inst fold_inst.

list_repr E L asserts that elements of E are exactly the elements from the list L .

Definition list_repr A (E :set A) (L :list A) :=
noduplicates L $\wedge$ ($\forall x$, mem x $L \leftrightarrow x \in E$).

list_covers E L asserts that all elements of E all belong to the list L .

Definition list_covers A (E :set A) (L :list A) :=
 $\forall x$, $x \in E \rightarrow$ mem x L .

23.1.5 Notations for building sets

DISCLAIMER: these definitions are experimental, they'll probably change

Declare Scope *set_scope*.

Notation " $\set\{ x \mid P \}$ " := (@set_st _ (fun $x \Rightarrow P$))
(at level 0, x name, P at level 200) : *set_scope*.

Notation " $\set\{ x : A \mid P \}$ " := (@set_st A (fun $x \Rightarrow P$))
(at level 0, x name, P at level 200) : *set_scope*.

Notation " $\set\{ x \in E \mid P \}$ " := (@set_st _ (fun $x \Rightarrow x \in E \wedge P$))
(at level 0, x name, P at level 200) : *set_scope*.

Notation " $\set\{ = e \mid x \in E \}$ " :=
(@set_st _ (fun $a \Rightarrow$ exists_ $x \in E$, $a = e$))
(at level 0, x name, E at level 200) : *set_scope*.

Notation " $\set\{ = e \mid x \in E, y \in F \}$ " :=
(@set_st _ (fun $a \Rightarrow$ exists_ $x \in E$, exists_ $y \in F$, $a = e$))
(at level 0, x name, F at level 200) : *set_scope*.

Notation " $\set\{ = e \mid x y \in E \}$ " :=
(@set_st _ (fun $a \Rightarrow$ exists_ $x y \in E$, $a = e$))
(at level 0, x name, y name, E at level 200) : *set_scope*.

23.2 Properties of sets

Section Instances.

Variables (A :Type).

Implicit Types E F : set A .

Transparent set finite empty_inst single_inst single_impl in_inst
incl_inst inter_inst union_inst card_inst fold_inst remove_inst
disjoint_inst.

Hint Constructors mem.

Local tactic to help unfolding all intermediate definitions

```

Ltac set_unf := unfold finite,
  card_inst, card_impl, card,
  to_list,
  disjoint_impl, disjoint_inst, disjoint,
  incl_inst, incl_impl,
  empty_inst, empty_impl, empty,
  single_inst, single_impl, single,
  in_inst, in_impl, is_in,
  incl_inst, incl_impl, incl,
  compl_impl, pred_not,
  inter_inst, inter_impl, inter, pred_and,
  union_inst, union_impl, union, pred_or,
  remove_inst, remove_impl, remove,
  fold_inst, fold_impl, fold in *.

```

```

Lemma disjoint_eq_inter_empty :  $\forall E F,$ 
   $(E \setminus \# F) = (E \setminus_n F = \{\})$ .

```

Proof using. auto. Qed.

```

Lemma in_set_st_eq :  $\forall (P:A \rightarrow \text{Prop}) x,$ 
   $x \setminus \text{in set\_st } P = P x$ .

```

Proof using. intros. apply $\times$ *prop_ext*. Qed.

```

Lemma set_ext_eq :  $\forall E F,$ 
   $(E = F) = (\forall (x:A), x \setminus \text{in } E \leftrightarrow x \setminus \text{in } F)$ .

```

Proof using.

```

  intros. apply prop_ext. iff H. subst  $\times$ . extens  $\times$ .

```

Qed.

```

Lemma set_ext :  $\forall E F,$ 
   $(\forall (x:A), x \setminus \text{in } E \leftrightarrow x \setminus \text{in } F) \rightarrow$ 
   $E = F$ .

```

Proof using. intros. rewrite $\neg$ set_ext_eq. Qed.

```

Lemma set_st_eq :  $\forall A (P Q : A \rightarrow \text{Prop}),$ 
   $(\forall (x:A), P x \leftrightarrow Q x) \rightarrow$ 
  set_st  $P = \text{set\_st } Q$ .

```

Proof using. intros. *asserts_rewrite* $\neg (P = Q)$. *extens* $\neg$. Qed.

```

Global Instance in_extens_inst : In_extens (A:=A) (T:=set A).

```

Proof using. constructor. intros. rewrite $\times$ set_ext_eq. Qed.

```

Global Instance in_empty_eq_inst : In_empty_eq (A:=A) (T:=set A).

```

Proof using. constructor. intros. apply $\times$ *prop_ext*. Qed.

```

Global Instance in_single_eq_inst : In_single_eq (A:=A) (T:=set A).

```

Proof using. constructor. intros. apply $\times$ *prop_ext*. Qed.

```

Global Instance in_union_eq_inst : In_union_eq (A:=A) (T:=set A).
Proof using. constructor. intros. set_unf. simpl. apply× prop_ext. Qed.

Global Instance in_inter_eq_inst : In_inter_eq (A:=A) (T:=set A).
Proof using. constructor. intros. set_unf. apply× prop_ext. Qed.

Global Instance in_remove_eq_inst : In_remove_eq (A:=A) (T:=set A).
Proof using. constructor. intros. set_unf. appls× prop_ext. Qed.

Global Instance incl_in_eq_inst : Incl_in_eq (A:=A) (T:=set A).
Proof using. constructor. intros. set_unf. autos×. Qed.

Global Instance disjoint_eq_inst : Disjoint_eq (T:=set A).
Proof using.
  constructor. intros. rewrite disjoint_eq_inter_empty.
  set_unf. appls prop_ext. iff M.
  intros x. rewrite× ← (@fun_eq_1 _ _ x _ M).
  extens×.
Qed.

Lemma eq_union_single_remove_one : ∀ E x,
  x \in E →
  E = \{x\} \u (E \- x).
Proof using.
  introv H. set_unf. extens. intros y. iff M.
  simpls. tests*: (y = x).
  destruct M. subst×. autos×.
Qed.

Lemma set_remove_one_add_same : ∀ E x,
  x \notin E →
  E = (E \u \{x\}) \- x.
Proof using.
  introv Hx. set_unf. extens. iff.
  { split. eauto. intro. subst×. }
  { tauto. }
Qed.

Lemma list_covers_of_list_repr : ∀ E L,
  list_repr E L →
  list_covers E L.
Proof using. introv (ND&EQ). introv Hx. rewrite¬ EQ. Qed.

Lemma list_repr_disjoint_union : ∀ E F LE LF,
  E \# F →
  list_repr E LE →
  list_repr F LF →

```

```

list_repr (E \u F) (LE ++ LF).
Proof using.
  introv D (HE&QE) (HF&QF). split.
  applys¬ noduplicates_app.
  intros x ? ?. applys× @disjoint_inv x.
  typeclass. rewrite¬ ← QE. rewrite¬ ← QF.
  intros x. rewrite mem_app_eq. rewrite in_union_eq.
  rewrite ← QE. rewrite× ← QF.
Qed.

Lemma noduplicates_of_list_repr : ∀ E xs,
  list_repr E xs →
  noduplicates xs.
Proof using. unfold list_repr. tauto. Qed.

Lemma ex_list_repr_impl_of_ex_list_covers_impl : ∀ E,
  ex (list_covers_impl E) →
  ex (list_repr_impl E).
Proof using.
  introv (L&M). sets_eq L1 EQL1: (remove_duplicates L).
  forwards¬ (HN&HM&_): remove_duplicates_spec EQL1.
  sets L2: (filter (fun x ⇒ x \in E) L1).
  ∃ L2. split.
  applys× noduplicates_filter.
  intros x. specializes M x. rewrite ← HM in M. set_unf. iff N.
  subst L2. forwards*: mem_filter_inv N.
  applys× mem_filter.
Qed.

Lemma list_repr_to_list_of_finite : ∀ E,
  finite E →
  list_repr E (to_list E).
Proof using.
  introv FE. unfolds to_list, finite, list_repr_impl.
  epsilon¬ L'.
  applys¬ ex_list_repr_impl_of_ex_list_covers_impl.
Qed.

Lemma eq_to_list_inv : ∀ E L,
  L = to_list E →
  finite E →
  list_repr E L.
Proof.
  introv EQ HE. unfolds. subst. forwards× (?&?): list_repr_to_list_of_finite HE.
  Qed.

```

Lemma finite_eq_in_iff_mem_to_list : $\forall E$,
 finite $E = (\forall x, x \in E \leftrightarrow \text{mem } x (\text{to\_list } E))$.

Proof.

intros. *applies prop_ext. iff M.*
 { *forwards* $\times (N1 \& N2)$: eq_to_list_inv E . intros x . *specializes N2 x. autos* $\times$. }
 { $\exists (\text{to\_list } E)$. intros x Ex . *rewrite* $\neg \leftarrow M$. }

Qed.

Lemma to_list_empty :
 to_list ($\{\}$:set A) = nil.

Proof using.

set_unf. epsilon l.
 { $\exists (@\text{nil } A)$. *split.* { *constructor.* } { intros. *rewrite* $\times$ mem_nil_eq. } }
 intros Hl . *inverts Hl. simpls. destruct* $\neg l$. *false. rewrite* $\leftarrow H0$. *simple* $\neg$.

Qed.

Lemma to_list_single : $\forall (x:A)$,
 to_list ($\{x\}$) = $x::\text{nil}$.

Proof using.

intros. *unfold to_list. epsilon l.*
 { $\exists (x::\text{nil})$. *split.*
 { *applies noduplicates_one.* }
 { *unfold single_inst, single_impl. simple* $\neg$.
 intros. *rewrite* $\times$ mem_one_eq. } }
introv Hl. unfolds single_inst, single_impl. simpls $\neg$.
inverts Hl as H H0. destruct (H0 x). specializes $\neg H2$.
destruct l.
 { *inverts H2.* }
 { *tests E: (x = a).*
 { *fequals. destruct l. auto. forwards* $\sim$: (*proj1* ($H0 a0$)).
subst. inverts H as M1 M2. false $\times M1$. }
 { *inverts H2. false. forwards* $\sim$: (*proj1* ($H0 a$)). *false.* } } }

Qed.

Lemma finite_of_list_covers : $\forall (E:\text{set } A) L$,
 list_covers $E L \rightarrow$
 finite E .

Proof using. *introv H. $\exists \times L$. Qed.*

Lemma finite_of_list_repr : $\forall (E:\text{set } A) L$,
 list_repr $E L \rightarrow$
 finite E .

Proof using. *introv (ND&EQ). $\exists \neg L$. introv Hx. rewrite* $\neg EQ$. Qed.

Lemma finite_of_ex_list_covers : $\forall (E:\text{set } A)$,


```

    ex (list_covers E) →
      finite E.
Proof using. introv (L&H). applies× finite_of_list_covers. Qed.

Definition finite_inv_list_covers_and_card : ∀ (E:set A),
  finite E →
  ∃ L, list_covers E L ∧ card E = length L.
Proof.
  introv (L&H). sets m: (card E).
  forwards× (R&P): mmin_spec_nat m.
Qed.

Lemma finite_inv_list_covers : ∀ (E:set A),
  finite E →
  ∃ L, list_covers E L.
Proof using. introv (L&HN). ∃ L. intros. applies× HN. Qed.

Lemma finite_empty :
  finite (\{\} : set A).
Proof using.
  intros. apply finite_of_ex_list_covers. set_unf.
  ∃ (@nil A). introv M. inverts M.
Qed.

Lemma finite_single : ∀ (a : A),
  finite \{a\}.
Proof using.
  intros. apply finite_of_ex_list_covers. set_unf.
  ∃ (a::nil). introv M. hnf in M. subst×.
Qed.

Lemma finite_union : ∀ E F,
  finite E →
  finite F →
  finite (E \u F).
Proof using.
  introv H1 H2. apply finite_of_ex_list_covers.
  lets (L1&E1): finite_inv_list_covers H1.
  lets (L2&E2): finite_inv_list_covers H2.
  ∃ (L1++L2). unfolds list_covers.
  introv M.
  rewrite @in_union_eq in M; try typeclass.
  rewrite× mem_app_eq.
Qed.

Lemma finite_union_inv : ∀ E F,

```

finite ($E \setminus u F$) $\rightarrow$
 finite $E \wedge$ finite F .

Proof using.

intros H. lets (L&HL): finite_inv_list_covers H. split.
{ apply finite_of_ex_list_covers. $\exists$ L.
unfolds list_covers. intros x M. applies HL. apply in_union_l. auto. }
{ apply finite_of_ex_list_covers. $\exists$ L.
unfolds list_covers. intros x M. applies HL. apply in_union_r. auto. }

Qed.

Lemma finite_union_eq : $\forall E F$,
 finite ($E \setminus u F$) = (finite $E \wedge$ finite F).

Proof using.

extends. iff. applies $\times$ finite_union_inv. applies $\times$ finite_union.

Qed.

Lemma finite_inter : $\forall E F$,
 finite $E \vee$ finite $F \rightarrow$
 finite ($E \setminus n F$).

Proof using.

intros H. apply finite_of_ex_list_covers. destruct H.
lets (L&EQ): finite_inv_list_covers H. $\exists$ L. unfold list_covers. set_unf. autos $\times$.
lets (L&EQ): finite_inv_list_covers H. $\exists$ L. unfold list_covers. set_unf. autos $\times$.

Qed.

Lemma finite_inter_l : $\forall E F$,
 finite $E \rightarrow$
 finite ($E \setminus n F$).

Proof using. intros. applies $\times$ finite_inter. Qed.

Lemma finite_inter_r : $\forall E F$,
 finite $F \rightarrow$
 finite ($E \setminus n F$).

Proof using. intros. applies $\times$ finite_inter. Qed.

Lemma finite_incl : $\forall E F$,
 $E \setminus c F \rightarrow$
 finite $F \rightarrow$
 finite E .

Proof using.

intros HI HF. apply finite_of_ex_list_covers.
lets (L&EQ): finite_inv_list_covers HF. unfold list_covers.
set_unf. $\exists \times$ L. intros Ex. applies EQ. applies $\neg$ HI.

Qed.

Lemma finite_remove : $\forall E F$,
 finite $E \rightarrow$

```

    finite ( $E \setminus F$ ).
Proof using.
    introv HE. apply finite_of_ex_list_covers.
    lets (L&EQ): finite_inv_list_covers HE. unfold list_covers. set_unf.  $\exists \times L$ .
Qed.

Section Finite_remove_inv.
Local Opaque remove_inst single_inst.

Lemma finite_remove_inv :  $\forall E F$ ,
    finite ( $E \setminus F$ )  $\rightarrow$ 
    finite  $F \rightarrow$ 
    finite  $E$ .
Proof using.
    introv H1 H2. lets (L1&R1): finite_inv_list_covers H1.
    lets (L2&R2): finite_inv_list_covers H2.
    applies finite_of_list_covers (L1 ++ L2).
    intros y Hy. rewrite  $\neg$  mem_app_eq. tests: ( $y \in F$ ).
    autos  $\neg$ .
    forwards  $\neg M: R1 y$ . rewrite  $\neg$  @in_remove_eq. typeclass.
Qed.

End Finite_remove_inv.

Lemma finite_remove_one_inv :  $\forall E x$ ,
    finite ( $E \setminus x$ )  $\rightarrow$ 
    finite  $E$ .
Proof using.
    introv H. applies finite_remove_inv H. applies finite_single.
Qed.

Lemma list_repr_nil:
    list_repr  $\setminus \{ \}$  (@nil  $A$ ).
Proof using.
    rewrite  $\leftarrow$  to_list_empty.
    eapply eq_to_list_inv; eauto using finite_empty.
Qed.

Definition list_covers_inv_card :  $\forall (E:\text{set } A) L$ ,
    list_covers  $E L \rightarrow$ 
    ( $\text{card } E \leq \text{length } L$ )%nat.
Proof using.
    introv H. sets m: ( $\text{card } E$ ). set_unf.
    forwards  $\times (R\&P)$ : mmin_spec_nat m.
    simpls. applies P.  $\exists L$ . splits  $\neg$ .
Qed.

```

Definition `finite_inv_list_repr_and_card` : $\forall (E:\text{set } A),$

`finite E` $\rightarrow$

$\exists L, \text{list\_repr } E \ L \wedge \text{card } E = \text{length } L.$

Proof.

introv H. forwards (L1&HL1&EL1): finite_inv_list_covers_and_card H.

sets L2: (remove_duplicates L1).

forwards (ND&EQ&LE): remove_duplicates_spec L1 L2.

sets L3: (filter (fun x $\Rightarrow$ x \in E) L2).

asserts: (length L3 $\leq$ length L2)%nat. applies length_filter.

asserts R3: (list_repr E L3).

split.

applies $\neg$ noduplicates_filter.

intros x. iff M.

unfold L3 in M. lets $\neg$ (&?): mem_filter_inv M.

applies $\neg$ mem_filter. rewrite EQ.

forwards C3: list_covers_of_list_repr R3.

$\exists L3. splits \times$.

forwards: list_covers_inv_card C3. math.

Qed.

Lemma `list_repr_inv_card` : $\forall (E:\text{set } A) (L:\text{list } A),$

`list_repr E L` $\rightarrow$

`card E` = `length L`.

Proof using.

introv HR. lets (ND&EQ): HR.

forwards (L'&(ND'&HR')&EQ'): finite_inv_list_repr_and_card E.

applies $\times$ finite_of_list_repr.

unfold card. simpl. rewrite EQ'.

applies $\neg$ noduplicates_length_eq.

intros x. rewrite EQ. rewrite $\times$ HR'.

Qed.

Definition `finite_inv_card_ge` : $\forall (E:\text{set } A) \ n,$

`finite E` $\rightarrow$

$(\forall L, \text{list\_covers } E \ L \rightarrow (\text{length } L \geq n)\%nat) \rightarrow$

$(\text{card } E \geq n)\%nat.$

Proof using.

introv H. sets m: (card E).

forwards $\times$ (R&P): mmin_spec_nat m.

lets (L&EL): finite_inv_list_covers H. $\exists \neg$ (length L) L.

simpls. introv HL. destruct R as (L&CR&ER).

forwards~: HL L. math.

Qed.

Definition `list_covers_inv_card_eq` : $\forall (E:\text{set } A) \ L,$

```

list_covers E L →
(∀ L', list_covers E L' → (length L' ≥ length L)%nat) →
card E = length L.
Proof using.
  intros HC HG.
  forwards~: list_covers_inv_card HC.
  forwards~: finite_inv_card_ge HG.
  applies× finite_of_list_covers.
  math.
Qed.

Lemma card_eq_length_to_list : ∀ (E:set A),
  finite E →
  card E = length (to_list E).
Proof using.
  intros FE. applies list_repr_inv_card. applies¬ eq_to_list_inv.
Qed.

Global Instance card_empty_inst : Card_empty (T:=set A).
Proof using.
  constructor. rewrite card_eq_length_to_list.
  lets E: to_list_empty. set_unf. rewrite E. rew_list¬.
  applies finite_empty.
Qed.

Global Instance card_single_inst : Card_single (T:=set A).
Proof using.
  constructor. intros a. rewrite card_eq_length_to_list.
  lets E: to_list_single a. set_unf. rewrite E. rew_list¬.
  applies finite_single.
Qed.

End Instances.

#[global]
Hint Resolve finite_empty finite_single finite_union
  finite_inter finite_incl finite_remove : finite.

```

23.3 Tactics for proving set equalities and set inclusions

The tactic *set_prove* aims at proving set equality by testing double inclusion using a boolean tautology decision procedure.

Section Autorewrite.

Variables (*A* : Type).

Implicit Types *x y* : *A*.

```

Implicit Types  $E F : \text{set } A$ .
Lemma set_in_empty_eq :  $\forall x,$ 
   $x \in (\{\}: \text{set } A) = \text{False}$ .
Proof using. apply in_empty_eq. Qed.

Lemma set_in_single_eq :  $\forall x y,$ 
   $x \in (\{y\}: \text{set } A) = (x = y)$ .
Proof using. apply in_single_eq. Qed.

Lemma set_in_inter_eq :  $\forall x E F,$ 
   $x \in (E \cap F) = (x \in E \wedge x \in F)$ .
Proof using. apply in_inter_eq. Qed.

Lemma set_in_union_eq :  $\forall x E F,$ 
   $x \in (E \cup F) = (x \in E \vee x \in F)$ .
Proof using. apply in_union_eq. Qed.

Lemma set_in_remove_eq :  $\forall x E F,$ 
   $x \in (E \setminus F) = (x \in E \wedge \neg x \in F)$ .
Proof using. apply in_remove_eq. Qed.

Lemma set_in_extens_eq :  $\forall E F,$ 
   $(E = F) = (\forall x, x \in E \leftrightarrow x \in F)$ .
Proof using.
  extens. iff M.
  subst $\times$ .
  applies @in_extens_eq. typeclass. intros. extens $\times$ .
Qed.

Lemma set_incl_in_eq :  $\forall E F,$ 
   $(E \subseteq F) = (\forall x, x \in E \rightarrow x \in F)$ .
Proof using. apply incl_in_eq. Qed.

Lemma set_disjoint_eq :  $\forall E F,$ 
   $(E \# F) = (\forall x, x \in E \rightarrow x \in F \rightarrow \text{False})$ .
Proof using. apply disjoint_eq. Qed.

End Autorewrite.

#[global]
Hint Rewrite in_set_st_eq set_in_empty_eq set_in_single_eq
  set_in_inter_eq set_in_union_eq set_in_remove_eq set_in_extens_eq
  set_incl_in_eq set_disjoint_eq : rew_set.

Ltac rew_set_tactic tt :=
  autorewrite_in_star_patch ltac:(fun tt  $\Rightarrow$  autorewrite with rew_set).

Ltac set_specialize_hyps A x :=
  repeat match goal with  $H: \forall \_ : ?A, \_ \vdash \_ \Rightarrow$ 
    specializes H x

```

```

end.

Ltac set_specialize_classic x :=
  repeat match goal with E: set _ ⊢ _ ⇒
    match goal with
    | H: x \in E ∨ ¬ x \in E ⊢ _ ⇒ fail 1
    | _ ⇒ lets: (prop_inv (x \in E))
    end
  end.

Ltac set_specialize_use_classic :=
  match goal with ⊢ ∀ _ :? A, _ ⇒
    let x := fresh "x" in intros x;
    set_specialize_hyps A x;
    match use_classic with
    | true ⇒ set_specialize_classic x
    | false ⇒ idtac
    end
  end.

Ltac set_prove_setup_use_classic :=
  intros;
  rew_set_tactic tt;
  try set_specialize_use_classic;
  rew_set_tactic tt.

Ltac set_prove_conclude :=
  solve [ intros; subst; tauto ].

Ltac set_prove :=
  set_prove_setup false; set_prove_conclude.

Ltac set_prove_classic :=
  set_prove_setup true; set_prove_conclude.

```

23.4 More properties

```

Lemma card_le_of_incl : ∀ A (E F:set A),
  finite F →
  E \c F →
  (card E ≤ card F)%nat.
Proof using.
  introv FF CF. lets FE: finite_incl CF FF.
  lets (LF&RF&QF): finite_inv_list_covers_and_card FF.
  rewrite QF. applies list_covers_inv_card. introv Ex.
  applies RF. applies × @incl_inv. typeclass.

```

Qed.

Lemma card_union_le : $\forall A (E F:\text{set } A),$
 finite $E \rightarrow$
 finite $F \rightarrow$
 card $(E \setminus \cup F) \leq (\text{card } E + \text{card } F)\%nat.$

Proof using.

intros FE FF.
lets (LE&RE&QE): finite_inv_list_covers_and_card FE.
lets (LF&RF&QF): finite_inv_list_covers_and_card FF.
lets H: list_covers_inv_card (E \setminus \cup F) (LE++LF) ---.
unfolds list_covers. intros. apply mem_app.
rewrite in_union_eq in H. autos.
rew_list in H. math.

Qed.

Lemma card_disjoint_union : $\forall A (E F:\text{set } A),$
 finite $E \rightarrow$
 finite $F \rightarrow$
 $E \setminus \# F \rightarrow$
 card $(E \setminus \cup F) = (\text{card } E + \text{card } F)\%nat.$

Proof using.

intros FE FF EF.
forwards: card_union_le FE FF.
cuts: (card (E \setminus \cup F) ≥ (card E + card F)%nat). math. clear H.
forwards (L&LC&LL): finite_inv_list_covers_and_card (E \setminus \cup F). applies $\neg$ finite_union.
rewrite LL. clear LL.
sets PE: (fun x $\Rightarrow$ x \in E). sets LE: (filter PE L).
sets PF: (fun x $\Rightarrow$ x \in F). sets LF: (filter PF L).
forwards: list_covers_inv_card E LE.
unfold LE, PE. introv Ex. forwards: LC x. set_prove. applies $\neg$ mem_filter.
forwards: list_covers_inv_card F LF.
unfold LF, PF. introv Fx. forwards: LC x. set_prove. applies $\neg$ mem_filter.
forwards LEF: filter_length_two_disjoint PE PF L.
introv _ HEx HFx. unfolds PE, PF. applies $\times$ @disjoint_inv. typeclass.
subst LE LF. math.

Qed.

Lemma card_inter_le_l : $\forall A (E F:\text{set } A),$
 finite $E \rightarrow$
 card $(E \setminus \cap F) \leq \text{card } E.$

Proof using.

intros. applies $\neg$ card_le_of_incl. set_prove.

Qed.

Lemma card_inter_le_r : $\forall A (E F : \text{set } A),$
 finite $F \rightarrow$
 $\text{card } (E \setminus F) \leq \text{card } F.$

Proof using.

intros. rewrite inter_comm. apply $\neg$ card_inter_le_l.

Qed.

Lemma card_ge_one : $\forall A (E : \text{set } A) x,$
 $x \in E \rightarrow$
 finite $E \rightarrow$
 $1 \leq \text{card } E.$

Proof using.

intros.
 rewrite $\leftarrow$ (card_single x).
 $\text{applies } \neg$ card_le_of_incl.
 $\text{set\_prove}.$

Qed.

Lemma card_disjoint_union_single : $\forall A (E : \text{set } A) x,$
 finite $E \rightarrow$
 $x \notin E \rightarrow$
 $(\text{card } (E \cup \{x\}) = \text{card } E + 1).$

Proof using.

intros.
 replace 1 with $(\text{card } \{x\})$ by eauto using @card_single, @card_single_inst.
 $\text{applies } \neg$ card_disjoint_union. applies finite_single.
 rewrite disjoint_single_req. auto.

Qed.

Lemma card_diff_single : $\forall A (E : \text{set } A) x,$
 finite $E \rightarrow$
 $x \in E \rightarrow$
 $(\text{card } (E \setminus x) = \text{card } E - 1).$

Proof using.

intros.
 assert ($h1 : (E \setminus x) \cup \{x\} = E$).
 { rewrite union_comm. erewrite eq_union_single_remove_one by eauto. reflexivity.
 }
 $\text{forwards } h2 : \text{card\_disjoint\_union\_single } (E \setminus x) x.$
 { eauto with finite. }
 { unfold notin. rewrite set_in_remove_eq.
 $\text{rew\_logic}.$ right.
 eapply in_single_self. }
 rewrite $h1$ in $h2$.

math.

Qed.

Lemma fold_eq_fold_to_list : $\forall A B (m:\text{monoid\_op } B) (f:A \rightarrow B) (E: \text{set } A),$
fold $m f E = \text{LibList.fold } m f (\text{to\_list } E).$

Proof using. reflexivity. Qed.

Lemma fold_eq_fold_list_repr : $\forall A B (m:\text{monoid\_op } B) (f:A \rightarrow B) (E: \text{set } A) L,$
Comm_monoid $m \rightarrow$
list_repr $E L \rightarrow$
fold $m f E = \text{LibList.fold } m f L.$

Proof using.

intros HM EL. rewrite fold_eq_fold_to_list.
forwards $\neg (N \& EQ2)$: eq_to_list_inv E. applies $\times$ finite_of_list_repr.
destruct EL as (ND & EQ1).
applies $\neg$ LibList.fold_equiv. intros. rewrite EQ2. rewrite $\times$ EQ1.

Qed.

Lemma fold_induction :

$\forall A B (m : \text{monoid\_op } B) (f : A \rightarrow B) (P : B \rightarrow \text{Prop}) (E : \text{set } A),$
Comm_monoid $m \rightarrow$
 $P (\text{monoid\_neutral } m) \rightarrow$
 $(\forall x a, a \in E \rightarrow P x \rightarrow P (\text{monoid\_oper } m (f a) x)) \rightarrow$
finite $E \rightarrow$
 $P (\text{fold } m f E).$

Proof using. *intros Hm Hbase Hstep Hfinite.*

assert $(\forall xs, \text{LibList.Forall } (\text{fun } x \Rightarrow x \in E) xs \rightarrow P (\text{LibList.fold } m f xs)).$
{ induction xs; rew_listx $\times$. }
forwards: list_repr_to_list_of_finite Hfinite.
erewrite fold_eq_fold_list_repr by eauto.
applies H. sets L: (to_list E). destruct H0 as ($\_ \& M$).
asserts M' : $(\forall x : A, \text{mem } x L \rightarrow x \in E). \{ \text{intros. applies } \times M. \}$ clear M.
{ induction L. { constructor. } { constructor.
{ applies $\times M'$. } { applies IHL. introv Lx. applies $\times M'$. } } }

Qed.

Lemma fold_congruence : $\forall A B (m:\text{monoid\_op } B) (f g:A \rightarrow B) (E:\text{set } A),$
Comm_monoid $m \rightarrow$
finite $E \rightarrow$
 $(\forall x, x \in E \rightarrow f x = g x) \rightarrow$
fold $m f E = \text{fold } m g E.$

Proof using. *intros Hm HE h. do 2 rewrite fold_eq_fold_to_list.*

eapply LibList.fold_congruence. intros.

eapply h. rewrite finite_eq_in_iff_mem_to_list in HE. rewrite $\times$ HE.

Qed.

Lemma fold_empty : $\forall A B (m:\text{monoid\_op } B) (f:A \rightarrow B)$,
 fold $m f (\backslash\{\}: \text{set } A) = \text{monoid\_neutral } m$.
 Proof using.
 intros. rewrite fold_eq_fold_to_list.
 rewrite to_list_empty. rewrite $\neg$ LibList.fold_nil.
 Qed.

Lemma fold_single : $\forall A B (m:\text{monoid\_op } B) (f:A \rightarrow B) (x:A)$,
Monoid $m \rightarrow$
 fold $m f \backslash\{x\} = f x$.
 Proof using.
 intros. rewrite fold_eq_fold_to_list.
 rewrite to_list_single. rewrite $\neg$ fold_cons.
 rewrite fold_nil. rewrite $\neg$ monoid_neutral_r.
 Qed.

Lemma fold_union : $\forall A B (m:\text{monoid\_op } B) (f:A \rightarrow B) (E F:\text{set } A)$,
Comm_monoid $m \rightarrow$
 finite $E \rightarrow$
 finite $F \rightarrow$
 $E \backslash\# F \rightarrow$
 fold $m f (E \backslash\cup F) = \text{monoid\_oper } m (\text{fold } m f E) (\text{fold } m f F)$.
 Proof using.
 introv HM HE HF HD.
 rewrites ($\gg$ fold_eq_fold_to_list E).
 rewrites ($\gg$ fold_eq_fold_to_list F).
 hint list_repr_to_list_of_finite.
 forwards $\neg$ HR: list_repr_disjoint_union HD.
 rewrites $\neg$ ($\gg$ fold_eq_fold_list_repr HR).
 rewrite $\neg$ LibList.fold_app. typeclass.
 Qed.

Lemma fold_isolate : $\forall A (E:\text{set } A) x$,
 finite $E \rightarrow$
 $x \backslash\text{in } E \rightarrow$
 $\forall B (m : \text{monoid\_op } B)$,
Comm_monoid $m \rightarrow$
 $\forall (f : A \rightarrow B)$,
 fold $m f E = \text{monoid\_oper } m (f x) (\text{fold } m f (E \backslash - \backslash\{x\}))$.
 Proof using. intros.
 rewrite (eq_union_single_remove_one $E x$) at 1 by eauto.
 erewrite $\leftarrow$ (fold_single $f x$) by typeclass.
 epose (remove_disjoint).
 eapply fold_union; eauto with finite.
 Qed.

23.4.1 Structural properties

Rewriting tactics *rew_set*

#[global]

Hint Rewrite *in_set_st_eq* : *rew_set*.

Tactic Notation "rew_set" :=
 autorewrite with *rew_set*.

Tactic Notation "rew_set" "in" *hyp*(*H*) :=
 autorewrite with *rew_set* in *H*.

Tactic Notation "rew_set" "in" "*" :=
 autorewrite with *rew_set* in *.

23.5 MORE

23.5.1 TEMPORARY Foreach

TODO: these lemmas should be instead derived as typeclasses in a generic way, in LibContainer.

TODO: add a paragraphe of the definition: $\text{foreach } P \ E = (\text{forall } x, x \in E \rightarrow P \ x)$

Section ForeachProp.

Variables (*A* : Type).

Implicit Types *P Q* : *A* → Prop.

Implicit Types *E F* : set *A*.

Lemma foreach_empty : $\forall P,$
 @foreach *A* (set *A*) _ *P* \{\}.

Proof using. *intros_all*. *false*. Qed.

Lemma foreach_single : $\forall P \ X,$
 P X →
 @foreach *A* (set *A*) _ *P* (\{ *X* \}).

Proof using. *intros_all*. rewrite *in_single_eq* in *H0*. subst×. Qed.

Lemma foreach_union : $\forall P \ E \ F,$
 foreach *P E* →
 foreach *P F* →
 foreach *P* (*E* \u *F*).

Proof using. *intros_all*. destruct¬ (in_union_inv *H1*). Qed.

Lemma foreach_union_inv : $\forall P \ E \ F,$
 foreach *P* (*E* \u *F*) →
 foreach *P E* ∧ foreach *P F*.

Proof using.

intros H. split; *intros K*.

```

    apply H.rewrite⊢ @in_union_eq. typeclass.
    apply H.rewrite⊢ @in_union_eq. typeclass.
Qed.

Lemma foreach_union_eq : ∀ P E F,
  foreach P (E ∪ F) = (foreach P E ∧ foreach P F).
Proof using.
  intros. extens. iff.
  apply⊢ foreach_union_inv.
  intuition eauto using foreach_union.
Qed.

Lemma foreach_single_eq : ∀ P X,
  foreach P (\{X\}:set A) = P X.
Proof using.
  intros. extens. iff.
  apply H. apply in_single_self.
  apply⊢ foreach_single.
Qed.

Lemma foreach_of_pred_incl: ∀ P Q E,
  foreach P E →
  pred_incl P Q →
  foreach Q E.
Proof using. introv H L K. apply⊢ L. Qed.

Lemma foreach_remove_of_foreach_all : ∀ P E F,
  foreach P E →
  foreach P (E \- F).
Proof using. introv M H. applies M. rewrite in_remove_eq in H. autos×. Qed.

Lemma foreach_remove : ∀ P E F,
  (∀ x, x \in E → x \notin F → P x) →
  foreach P (E \- F).
Proof using. introv M Px. rewrite in_remove_eq in Px. applies× M. Qed.

Lemma notin_of_foreach_not : ∀ P x E,
  foreach P E →
  ⊢ P x →
  x \notin E.
Proof using. introv M N I. applies N. applies⊢ M. Qed.

End ForeachProp.

#[global]
Hint Resolve foreach_empty foreach_single foreach_union.
#[global]
Hint Rewrite foreach_union_eq foreach_single_eq : rew_foreach.

```

```

Tactic Notation "rew_foreach" :=
  autorewrite with rew_foreach.
Tactic Notation "rew_foreach" "in" hyp(H) :=
  autorewrite with rew_foreach in H.
Tactic Notation "rew_foreach" "in" "*" :=
  autorewrite_in_star_patch ltac:(fun tt => autorewrite with rew_foreach).
Tactic Notation "rew_foreach" "~" :=
  rew_foreach; auto_tilde.
Tactic Notation "rew_foreach" "*" :=
  rew_foreach; auto_star.
Tactic Notation "rew_foreach" "~" "in" constr(H) :=
  rew_foreach in H; auto_tilde.
Tactic Notation "rew_foreach" "*" "in" constr(H) :=
  rew_foreach in H; auto_star.
Tactic Notation "rew_foreach" "~" "in" "*" :=
  rew_foreach in *; auto_tilde.
Tactic Notation "rew_foreach" "*" "in" "*" :=
  rew_foreach in *; auto_star.

```

Chapter 24

Library SLF.LibChoice

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibEpsilon LibRelation.

Generalizable Variables *A B*.

This file includes several versions of the axiom of choice. This “axiom” is actually proved in terms of indefinite description. Remark: the choice results contained in this file are not very useful in practice since it is usually more convenient to use the epsilon operator directly.

24.1 Functional choice

24.1.1 Functional choice

This result can be used to build a function from a relation that maps every input to at least one output.

```
Lemma functional_choice :  $\forall A B (R:A \rightarrow B \rightarrow \text{Prop}),$   
  ( $\forall x, \exists y, R x y$ )  $\rightarrow$   
  ( $\exists f, \forall x, R x (f x)$ ).
```

Proof using.

```
  intros.  $\exists$  (fun x  $\Rightarrow$  sig_val (indefinite_description (H x))).  
  intro x. apply (sig_proof (indefinite_description (H x))).  
Qed.
```

24.1.2 Dependent functional choice

It is a generalization of functional choice to dependent functions.

Scheme and_indd := Induction for **and** Sort Prop.

Scheme eq_indd := Induction for **eq** Sort Prop.

Lemma dependent_functional_choice :

```

  ∀ A (B:A→Type) (R:∀ x, B x → Prop),
  (∀ x, ∃ y, R x y) →
  (∃ f, ∀ x, R x (f x)).

```

Proof using.

```

  introv H.
  pose (B' := { x:A & B x }).
  pose (R' := fun (x:A) (y:B') => projT1 y = x ∧ R (projT1 y) (projT2 y)).
  destruct (functional_choice R') as (f,Hf).
  intros x. destruct (H x) as (y,Hy).
  ∃ (existT (fun x => B x) x y). split⊢.
  sets proj1-transparent: (fun P Q (p:P∧Q) => let (a,b) := p in a).
  ∃ (fun x => eq_rect _ _ (projT2 (f x)) _ (proj1-transparent _ _ (Hf x))).
  intros x. destruct (Hf x) as (Heq,HR) using and_indd.
  destruct (f x). simpls. destruct Heq using eq_indd. apply HR.

```

Qed.

Arguments dependent_functional_choice [A] [B].

24.1.3 Guarded functional choice

Similar to functional choice, except that it targets partial functions

```

Lemma guarded_functional_choice : ∀ A '{Inhab B} (P : A→Prop) (R : A→B→Prop),
  (∀ x, P x → ∃ y, R x y) →
  (∃ f, ∀ x, P x → R x (f x)).

```

Proof using.

```

  intros. apply (functional_choice (fun x y => P x → R x y)).
  intros. applies× indep_general_premises.

```

Qed.

24.1.4 Omniscient functional choice

Similar to functional choice except that the proof of functionality of the relation is given after the fact, for each argument.

```

Lemma omniscient_functional_choice : ∀ A '{Inhab B} (R : A→B→Prop),
  ∃ f, ∀ x, (∃ y, R x y) → R x (f x).

```

Proof using. intros. apply⊢ guarded_functional_choice. Qed.

24.2 Functional unique choice

This section provides a similar set of results excepts that it is specialized for the case where each argument has a unique image.

24.2.1 Functional unique choice

Lemma `functional_unique_choice` : $\forall A B (R:A \rightarrow B \rightarrow \text{Prop}),$

$(\forall x, \exists! y, R x y) \rightarrow$

$(\exists! f, \forall x, R x (f x)).$

Proof using.

`intros. destruct (functional_choice R) as [f Hf].`

`intros. apply (ex_of_ex_unique (H x)).`

`$\exists f$ . split. auto.`

`intros g Hg. apply fun_ext_1. intros y.`

`apply $\neg$  (at_most_one_of_ex_unique (H y)).`

Qed.

24.2.2 Dependent functional unique choice

Theorem `dependent_functional_unique_choice` :

$\forall (A:\text{Type}) (B:A \rightarrow \text{Type}) (R:\forall (x:A), B x \rightarrow \text{Prop}),$

$(\forall (x:A), \exists! y : B x, R x y) \rightarrow$

$(\exists! f : (\forall (x:A), B x), \forall (x:A), R x (f x)).$

Proof using.

`intros. destruct (dependent_functional_choice R) as [f Hf].`

`intros. apply (ex_of_ex_unique (H x)).`

`$\exists f$ . split. auto.`

`intros g Hg. extens. intros y.`

`apply $\neg$  (at_most_one_of_ex_unique (H y)).`

Qed.

Arguments `dependent_functional_unique_choice` [A] [B].

24.2.3 Guarded functional unique choice

Lemma `guarded_functional_unique_choice` :

$\forall A \{Inhab B\} (P : A \rightarrow \text{Prop}) (R : A \rightarrow B \rightarrow \text{Prop}),$

$(\forall x, P x \rightarrow \exists! y, R x y) \rightarrow$

$(\exists f, \forall x, P x \rightarrow R x (f x)).$

Proof using.

`introv I M. apply (functional_choice (fun x y  $\Rightarrow$  P x  $\rightarrow$  R x y)).`

`intros. apply $\times$  indep_general_premises.`

`introv H. destruct $\times$  (M - H) as (y&Hy&-).`

Qed.

24.2.4 Omniscient functional unique choice

Lemma `omniscient_functional_unique_choice` :

```

  ∀ A {Inhab B} (R : A → B → Prop),
  ∃ f, ∀ x, (∃! y, R x y) → R x (f x).

```

Proof using.

```

  intros. destruct (omniscient_functional_choice R) as [f F].
  ∃ f. introv (y & Hy & Uy). autos×.

```

Qed.

24.3 Relational choice

Relational choice can be used to extract from a relation a subrelation that describes a function, by mapping every argument to a unique image.

24.3.1 Relational choice

Lemma `rel_choice` : $\forall A B (R : A \rightarrow B \rightarrow \text{Prop}),$

```

  (∀ x, ∃ y, R x y) →
  (∃ R', rel_incl R' R
   ∧ ∀ x, ∃! y, R' x y).

```

Proof using.

```

  introv H. destruct⊢ (functional_choice R) as [f Hf].
  ∃ (fun x y ⇒ f x = y). split.
  introv E. simpls. subst⊢.
  intros x. ∃⊢ (f x).

```

Qed.

24.3.2 Guarded relational choice

Lemma `guarded_rel_choice` : $\forall A B (P : A \rightarrow \text{Prop}) (R : A \rightarrow B \rightarrow \text{Prop}),$

```

  (∀ x, P x → ∃ y, R x y) →
  (∃ R', rel_incl R' R
   ∧ ∀ x, P x → ∃! y, R' x y).

```

Proof using.

```

  intros. destruct (rel_choice (fun (x: sig P) (y: B) ⇒ R (sig_val x) y))
  as (R', (HR' R, M)).
  intros (x, HPx). destruct (H _ HPx) as (y, HRxy). ∃⊢ y.
  set (R'' := fun (x: A) (y: B) ⇒ ∃ (H : P x), R' (exist P x H) y).
  ∃ R''. split.
  intros x y (HPx, HR'xy). apply (HR' R _ _ HR'xy).

```

```

intros  $x$   $HPx$ . destruct ( $M$  (exist  $P$   $x$   $HPx$ )) as ( $y$ , ( $HR'xy$ ,  $Uniq$ )).
   $\exists$   $y$ . split.
   $\exists \neg HPx$ .
  intros  $y'$  ( $H'Px$ ,  $HR'xy'$ ). apply  $Uniq$ .
  rewrite  $\neg$  (proof_irrelevance  $HPx$   $H'Px$ ).
Qed.

```

24.3.3 Omniscient relation choice

Lemma omniscient_rel_choice : $\forall A B (R : A \rightarrow B \rightarrow \text{Prop})$,
 $\exists R'$, rel_incl $R' R$
 $\wedge \forall x, (\exists y, R x y) \rightarrow (\exists! y, R' x y)$.
Proof using. intros. apply $\neg$ guarded_rel_choice. Qed.

Chapter 25

Library **SLF.LibUnit**

```
Set Implicit Arguments.  
From SLF Require Import LibTactics LibLogic LibReflect.
```

25.1 Unit type

25.1.1 Definition

From the Prelude.

```
Inductive unit : Type := | tt : unit.
```

```
Add Printing If bool. Delimit Scope bool_scope with bool.
```

25.1.2 Inhabited

```
#[global]  
Instance Inhab_unit : Inhab unit.  
Proof using. intros. apply (Inhab_of_val tt). Qed.
```

25.2 Properties

25.2.1 Uniqueness

```
Lemma unit_eq :  $\forall$  (tt1 tt2 : unit),  
  tt1 = tt2.  
Proof using. intros. destruct tt1. destruct  $\neg$  tt2. Qed.
```

Chapter 26

Library SLF.LibFun

DEPRECATED? \o notation for composition is used in LibFixDemos

26.1

26.2

Set Implicit Arguments.

From *SLF* Require Import LibTactics LibLogic LibContainer LibSet.

From *SLF* Require LibList.

Generalizable Variables *A*.

26.2.1 Indentity function

Definition id {*A*} (*x* : *A*) :=

x.

Definition const {*A B*} (*v* : *B*) : *A* → *B* :=

fun _ ⇒ *v*.

Definition const1 :=

@const.

Definition const2 {*A1 A2 B*} (*v*:*B*) : *A1* → *A2* → *B* :=

fun _ _ ⇒ *v*.

Definition const3 {*A1 A2 A3 B*} (*v*:*B*) : *A1* → *A2* → *A3* → *B* :=

fun _ _ _ ⇒ *v*.

Definition const4 {*A1 A2 A3 A4 B*} (*v*:*B*) : *A1* → *A2* → *A3* → *A4* → *B* :=

fun _ _ _ _ ⇒ *v*.

Definition const5 {*A1 A2 A3 A4 A5 B*} (*v*:*B*) : *A1* → *A2* → *A3* → *A4* → *A5* → *B* :=

fun _ _ _ _ _ $\Rightarrow v$.

Definition apply {A B} (f : A $\rightarrow$ B) (x : A) :=
f x.

Definition apply_to (A : Type) (x : A) (B : Type) (f : A $\rightarrow$ B) :=
f x.

Definition compose {A B C} (g : B $\rightarrow$ C) (f : A $\rightarrow$ B) :=
fun x $\Rightarrow$ g (f x).

Declare Scope fun_scope.

Notation "f1 \o f2" := (compose f1 f2)
(at level 49, right associativity) : fun_scope.

Section Combinators.

Open Scope fun_scope.

Variables (A B C D : Type).

Lemma compose_id_l : $\forall$ (f : A $\rightarrow$ B),
id \o f = f.

Proof using. intros. apply $\neg$ fun_ext_1. Qed.

Lemma compose_id_r : $\forall$ (f : A $\rightarrow$ B),
f \o id = f.

Proof using. intros. apply $\neg$ fun_ext_1. Qed.

Lemma compose_assoc : $\forall$ (f : C $\rightarrow$ D) (g : B $\rightarrow$ C) (h : A $\rightarrow$ B),
(f \o g) \o h = f \o (g \o h).

Proof using. intros. apply $\neg$ fun_ext_1. Qed.

Lemma compose_eq_l : $\forall$ (f : B $\rightarrow$ C) (g1 g2 : A $\rightarrow$ B),
g1 = g2 $\rightarrow$
f \o g1 = f \o g2.

Proof using. intros. subst $\neg$. Qed.

Lemma compose_eq_r : $\forall$ (f : A $\rightarrow$ B) (g1 g2 : B $\rightarrow$ C),
g1 = g2 $\rightarrow$
g1 \o f = g2 \o f.

Proof using. intros. subst $\neg$. Qed.

Composition of *LibList.map* behaves well. Import LIBLIST.

Lemma list_map_compose : $\forall$ A B C (f : A $\rightarrow$ B) (g : B $\rightarrow$ C) l,
LibList.map g (LibList.map f l) = LibList.map (g \o f) l.

Proof using.

introv. induction l.
reflexivity.
rew_listx. fequals $\neg$.

Qed.

End Combinators.

Tactic for simplifying function compositions

```
#[global]
Hint Rewrite compose_id_l compose_id_r compose_assoc : rew_compose.
Tactic Notation "rew_compose" :=
  autorewrite with rew_compose.
Tactic Notation "rew_compose" "in" "*" :=
  autorewrite with rew_compose in *.
Tactic Notation "rew_compose" "in" hyp(H) :=
  autorewrite with rew_compose in H.
```

26.2.2 Function update

fupdate *f a b x* is like *f* except that it returns *b* for input *a*

Definition fupdate *A B (f : A → B) (a : A) (b : B) : A → B :=*
 fun *x* ⇒ If (*x* = *a*) then *b* else *f x*.

Lemma fupdate_eq : ∀ *A B (f:A→B) a b x*,
 fupdate *f a b x* = If (*x* = *a*) then *b* else *f x*.

Proof using. auto. Qed.

Lemma fupdate_same : ∀ *A B (f:A→B) a b*,
 fupdate *f a b a* = *b*.

Proof using. intros. unfold fupdate. *case-if* ×. Qed.

Lemma fupdate_neq : ∀ *A B (f:A→B) a b x*,
 x ≠ *a* →
 fupdate *f a b x* = *f x*.

Proof using. intros. unfold fupdate. *case-if* ×. Qed.

26.2.3 Function image

Section FunctionImage.

Open Scope *set_scope*.

Import LIBLIST.

Definition image *A B (f : A → B) (E : set A) : set B :=*
 \set{ *y* | exists_ *x* \in *E*, *y* = *f x* }.

Lemma in_image_prove_eq : ∀ *A B x (f : A → B) (E : set A)*,
 x \in *E* → *f x* \in image *f E*.

Proof using. *introv N*. unfold image. *rew_set*. ∃ × *x*. Qed.

Lemma in_image_prove : ∀ *A B x y (f : A → B) (E : set A)*,
 x \in *E* → *y* = *f x* → *y* \in image *f E*.

```

Proof using. intros. subst. appls× in_image_prove_eq. Qed.

Lemma in_image_inv : ∀ A B y (f : A → B) (E : set A),
  y \in image f E → ∃ x, x \in E ∧ y = f x.
Proof using. introv N. unfolds image. rew_set in N. auto. Qed.

Lemma finite_image : ∀ A B (f : A → B) (E : set A),
  finite E →
  finite (image f E).
Proof using.
  introv M. lets (L&H): finite_inv_list_covers M.
  appls finite_of_list_covers (LibList.map f L). introv N.
  lets (y&Hy&Ey): in_image_inv (rm N). subst x. appls× mem_map.
Qed.

Lemma image_covariant : ∀ A B (f : A → B) (E F : set A),
  E \c F →
  image f E \c image f F.
Proof using.
  introv. do 2 rewrite incl_in_eq. introv M N.
  lets (y&Hy&Ey): in_image_inv (rm N). appls× in_image_prove.
Qed.

Lemma image_union : ∀ A B (f : A → B) (E F : set A),
  image f (E \u F) = image f E \u image f F.
Proof using.
  Hint Resolve in_image_prove.
  introv. apply in_extens. intros x. iff N.
  lets (y&Hy&Ey): in_image_inv (rm N). rewrite in_union_eq in Hy.
  rewrite in_union_eq. destruct× Hy.
  rewrite in_union_eq in N. destruct N as [N|N].
  lets (y&Hy&Ey): in_image_inv (rm N). appls× in_image_prove.
  rewrite in_union_eq. eauto.
  lets (y&Hy&Ey): in_image_inv (rm N). appls× in_image_prove.
  rewrite in_union_eq. eauto.
Qed.

Lemma image_singleton : ∀ A B (f : A → B) (x : A),
  image f \{x} = \{f x}.
Proof using.
  intros. apply in_extens. intros z. rewrite in_single_eq. iff N.
  lets (y&Hy&Ey): in_image_inv (rm N). rewrite in_single_eq in Hy. subst¬.
  appls× in_image_prove. rewrite¬ @in_single_eq. typeclass.
Qed.

End FunctionImage.

#[global]

```


Hint Resolve *finite_image* : *finite*.

26.2.4 Function preimage

Section FunctionPreimage.

Open Scope *set_scope*.

Definition preimage *A B* (*f* : *A* → *B*) (*E* : set *B*) : set *A* :=
 \set{ *x* | exists_ *y* \in *E*, *y* = *f x* }.

End FunctionPreimage.

26.2.5 Function iteration

Fixpoint applyn *A n* (*f* : *A* → *A*) *x* :=
 match *n* with
 | 0 ⇒ *x*
 | S *n'* ⇒ *f* (applyn *n'* *f x*)
 end.

Lemma applyn_fix : ∀ *A n f* (*x* : *A*),
 applyn (S *n*) *f x* = applyn *n f* (*f x*).

Proof using. *intros. induction n. simpls. rewrite IHn. Qed.*

Lemma applyn_comp : ∀ *A n m f* (*x* : *A*),
 applyn *n f* (applyn *m f x*) = applyn (*n + m*) *f x*.

Proof using.

intros. gen m; induction n; intros; simpls.
rewrite IHn.

Qed.

Lemma applyn_nested : ∀ *A n m f* (*x* : *A*),
 applyn *n* (applyn *m f*) *x* = applyn (*n × m*) *f x*.

Proof using.

intros. gen m. induction n; intros; simpls.
rewrite IHn. rewrite applyn_comp.

Qed.

Lemma applyn_altern : ∀ *A B* (*f* : *A* → *B*) (*g* : *B* → *A*) *x n*,
 applyn *n* (fun *x* ⇒ *f* (*g x*)) (*f x*) =
 f (applyn *n* (fun *x* ⇒ *g* (*f x*)) *x*).

Proof using. *intros. gen x. induction n. intros. repeat rewrite applyn_fix. autos.*
Qed.

Lemma applyn_ind : ∀ *A* (*P* : *A* → Prop) (*f* : *A* → *A*) *x n*,
 (∀ *x*, *P x* → *P* (*f x*)) →
 P x →

P (applyn n f x).

Proof using. *intros* I . induction n ; *intros* Hx ; *autos*×. Qed.

Chapter 27

Library SLF.LibString

```
Set Implicit Arguments.  
From SLF Require Import LibTactics LibReflect.  
From Stdlib Require Export String.
```

27.1 Inhabited

```
#[global]  
Instance Inhab_string : Inhab string.  
Proof using. apply (Inhab_of_val EmptyString). Qed.
```

Chapter 28

Library SLF.LibMultiset

DISCLAIMER: this file is under construction and still contains a couple admits.

28.1

28.2

```
Set Implicit Arguments.
Generalizable Variables A B.
From SLF Require Import LibTactics LibLogic LibReflect
  LibRelation LibList LibInt LibNat LibOperation
  LibEpsilon LibSet LibMonoid.
From SLF Require Export LibContainer.
Open Scope nat_scope.
```

28.3 Construction of sets as predicates

28.3.1 Basic definitions

```
Definition multiset (A : Type) := A → nat.
Section Operations.
Variables (A B : Type).
Implicit Types x : A.
Implicit Types E F G : multiset A.
Definition empty_impl : multiset A := fun _ => 0.
Definition single_impl x := fun y => If x = y then 1 else 0.
```

Definition in_impl $x \ E := E \ x > 0$.

Definition union_impl $E \ F := \text{fun } x \Rightarrow (E \ x + F \ x) \% \text{nat}$.

Definition incl_impl $E \ F := \forall \ x, E \ x \leq F \ x$.

Definition dom_impl $E := \text{set\_st } (\text{fun } x \Rightarrow E \ x > 0)$.

Definition list_repr_impl $(E:\text{multiset } A) \ (l:\text{list } (A \times \text{nat})) :=$
 $\text{noduplicates } (\text{LibList.map } (@\text{fst } \_) \ l)$
 $\wedge \forall \ n \ x, \text{mem } (x, n) \ l \leftrightarrow (n = E \ x \wedge n > 0)$.

Definition to_list_impl $(E:\text{multiset } A) := \text{epsilon } (\text{list\_repr\_impl } E)$.

Definition fold_impl $(m:\text{monoid\_op } B) \ (f:A \rightarrow \text{nat} \rightarrow B) \ (E:\text{multiset } A) :=$
 $\text{LibList.fold\_right } (\text{fun } p \ acc \Rightarrow \text{let } (x, n) := p : A \times \text{nat} \text{ in monoid\_oper } m \ (f \ x \ n) \ acc)$
 $(\text{monoid\_neutral } m) \ (\text{to\_list\_impl } E)$.

End Operations.

Definition card_impl $A \ (E:\text{multiset } A) :=$
 $\text{fold\_impl } (\text{monoid\_make plus } 0) \ (\text{fun } \_ \ n \Rightarrow n) \ E$.

28.3.2 Notation through typeclasses

Lemma in_inst : $\forall \ A, \text{BagIn } A \ (\text{multiset } A)$.

Proof using. constructor. exact (@in_impl A). Defined.

#[global]

Hint Extern 1 (**BagIn** $\_$ (multiset $\_$)) $\Rightarrow$ apply in_inst
 $: \text{typeclass\_instances}$.

#[global]

Instance empty_inst : $\forall \ A, \text{BagEmpty } (\text{multiset } A)$.
 $\text{constructor. rapply } (@\text{empty\_impl } A)$. Defined.

#[global]

Instance single_inst : $\forall \ A, \text{BagSingle } A \ (\text{multiset } A)$.
 $\text{constructor. rapply } (@\text{single\_impl } A)$. Defined.

#[global]

Instance union_inst : $\forall \ A, \text{BagUnion } (\text{multiset } A)$.
 $\text{constructor. rapply } (@\text{union\_impl } A)$. Defined.

#[global]

Instance incl_inst : $\forall \ A, \text{BagIncl } (\text{multiset } A)$.
 $\text{constructor. rapply } (@\text{incl\_impl } A)$. Defined.

#[global]

Instance fold_inst : $\forall \ A \ B, \text{BagFold } B \ (A \rightarrow \text{nat} \rightarrow B) \ (\text{multiset } A)$.
 $\text{constructor. rapply } (@\text{fold\_impl } A \ B)$. Defined.

#[global]

```

Instance card_inst :  $\forall A$ , BagCard (multiset  $A$ ).
  constructor. rappl (@card_impl  $A$ ). Defined.

Global Opaque multiset empty_inst single_inst in_inst
  union_inst incl_inst fold_inst card_inst.

#[global]
Instance Inhab_multiset :  $\forall A$ , Inhab (multiset  $A$ ).
Proof using. intros. apply (Inhab_of_val (@empty_impl  $A$ )). Qed.

```

28.4 Properties of multisets

28.4.1 Structural properties

```

Section Instances.
Variables ( $A$ :Type).
Hint Extern 1 False  $\Rightarrow$  math.
Hint Extern 1 ( $\_ > \_$ )  $\Rightarrow$  solve [ math | false ].
Hint Extern 1 ( $\_ = \_$ )  $\Rightarrow$  math.

Transparent multiset empty_inst single_inst in_inst
  union_inst incl_inst fold_inst card_inst.

Global Instance in_empty_eq_inst : In_empty_eq ( $A:=A$ ) ( $T:=$ multiset  $A$ ).
Proof using.
  constructor. intros. extens. simpl.
  unfold empty_impl, in_impl. autos $\times$ .
Qed.

Global Instance in_single_eq_inst : In_single_eq ( $A:=A$ ) ( $T:=$ multiset  $A$ ).
Proof using.
  constructor. intros. extens. simpl.
  unfold single_impl, in_impl. case_if $\times$ .
Qed.

Global Instance in_union_eq_inst : In_union_eq ( $A:=A$ ) ( $T:=$ multiset  $A$ ).
Proof using.
  constructor. intros. extens. simpl.
  unfold single_impl, union_impl, in_impl.
  iff. tests: ( $E\ x = 0$ ). right $\neg$ . left $\neg$ . destruct  $H$ ; math.
Qed.

Global Instance incl_refl_inst : Incl_refl ( $T:=$ multiset  $A$ ).
Proof using.
  constructor. intros  $x$ . unfold incl_inst, incl_impl, incl. intros. math.
Qed.

```

Global Instance incl_inv_inst : **Incl_inv** (A:=A) (T:=multiset A).

Proof using.

```
constructor. intros x. introv M N.  
unfold incl_inst, incl_impl, incl, in_inst, in_impl, is_in in *.  
specializes M x. math.
```

Qed.

Global Instance union_empty_inv_inst : **Union_empty_inv** (T:=multiset A).

Proof using.

```
constructor. introv.  
unfold union_inst, union_impl, union, empty_impl,  
empty_inst, empty, empty_impl, multiset. introv N.  
split; extens¬.  
intros x. lets: fun_eq_1 x N. math.  
intros x. lets: fun_eq_1 x N. math.
```

Qed.

Global Instance union_empty_l_eq_inst : **Union_empty_l** (T:=multiset A).

Proof using.

```
constructor. intros_all. simpl.  
unfold union_impl, empty_impl, multiset. simpl. extens¬.
```

Qed.

Global Instance union_comm_eq_inst : **Union_comm** (T:=multiset A).

Proof using.

```
constructor. intros_all. simpl.  
unfold union_impl, multiset. simpl. extens¬.
```

Qed.

Global Instance union_assoc_eq_inst : **Union_assoc** (T:=multiset A).

Proof using.

```
constructor. intros_all. simpl.  
unfold union_impl, multiset. simpl. extens¬.
```

Qed.

Global Instance empty_incl_inst : **Empty_incl** (T:=multiset A).

Proof using.

```
constructor. intros_all. simpl.  
unfold empty_impl, multiset. math.
```

Qed.

Global Instance union_incl_union_inst : **Union_incl_union** (T:=multiset A).

Proof using.

```
constructor.  
unfold incl_inst, incl_impl, union_inst, union_impl, multiset. simpl.  
introv H1 H2. intros x. specializes H1 x. specializes H2 x. math.
```

Qed.

End Instances.

28.5 Additional predicates

28.5.1 Foreach

TODO: foreach properties should be shared in LibContainer

Section ForeachProp.

Variables ($A : \text{Type}$).

Implicit Types $P Q : A \rightarrow \text{Prop}$.

Implicit Types $E F : \text{multiset } A$.

Lemma foreach_empty : $\forall P$,

 @foreach A (multiset A) $\neg P \setminus \{\}$.

Proof using. *intros_all*. rewrite in_empty_eq in H . *false*. Qed.

Lemma foreach_single : $\forall P X$,

$P X \rightarrow$

 @foreach A (multiset A) $\neg P (\setminus \{ X \})$.

Proof using. *intros_all*. rewrite in_single_eq in $H0$. *subst*×. Qed.

Lemma foreach_union : $\forall P E F$,

 foreach $P E \rightarrow$ foreach $P F \rightarrow$ foreach $P (E \setminus u F)$.

Proof using. *intros_all*. *destruct*¬ (in_union_inv $H1$). Qed.

Hint Resolve *foreach_empty foreach_single foreach_union*.

Lemma foreach_union_inv : $\forall P E F$,

 foreach $P (E \setminus u F) \rightarrow$

 foreach $P E \wedge$ foreach $P F$.

Proof using.

introv H. *split*; *introv K*.

apply H. *rewrite*¬ @in_union_eq. *typeclass*.

apply H. *rewrite*¬ @in_union_eq. *typeclass*.

Qed.

Lemma foreach_union_eq : $\forall P E F$,

 foreach $P (E \setminus u F) = (\text{foreach } P E \wedge \text{foreach } P F)$.

Proof using.

intros. *extens*. *iff*.

apply¬ foreach_union_inv. *apply*× foreach_union.

Qed.

Lemma foreach_single_eq : $\forall P X$,

 foreach $P (\setminus \{ X \} : \text{multiset } A) = P X$.

Proof using.

intros. *extens*. *iff*.


```

    apply H. apply in_single_self.
    apply  $\neg$  foreach_single.
Qed.

Lemma foreach_weaken :  $\forall P Q E$ ,
  foreach P E  $\rightarrow$ 
  pred_incl P Q  $\rightarrow$ 
  foreach Q E.
Proof using. intros H L K. apply  $\neg$  L. Qed.
End ForeachProp.

```

28.6 Tactics

EXPERIMENTAL tactics – TODO: add documentation

28.6.1 Tactics to prove equalities on unions

```

Lemma for_multiset_union_assoc :  $\forall A$ , assoc (union (T:=multiset A)).
Proof using. intros. apply union_assoc. Qed.

Lemma for_multiset_union_comm :  $\forall A$ , comm (union (T:=multiset A)).
Proof using. intros. apply union_comm. Qed.

Lemma for_multiset_union_empty_l :  $\forall A (E:\text{multiset } A)$ ,  $\setminus\{\}$   $\setminus$ u E = E.
Proof using. intros. apply union_empty_l. Qed.

Lemma for_multiset_union_empty_r :  $\forall A (E:\text{multiset } A)$ , E  $\setminus$ u  $\setminus\{\}$  = E.
Proof using. intros. apply union_empty_r. Qed.

Lemma for_multiset_empty_incl :  $\forall A (E:\text{multiset } A)$ ,  $\setminus\{\}$   $\setminus$ c E.
Proof using. intros. apply empty_incl. Qed.

#[global]
Hint Rewrite  $\leftarrow$  for_multiset_union_assoc : rew_permut_simpl.
#[global]
Hint Rewrite for_multiset_union_empty_l for_multiset_union_empty_r : rew_permut_simpl.
Ltac rew_permut_simpl :=
  autorewrite with rew_permut_simpl.
Ltac rews_permut_simpl :=
  autorewrite with rew_permut_simpl in *.

Section PermutationTactic.
Context (A:Type).
Implicit Types l : multiset A.

Lemma permut_get_l :  $\forall l1 l2$ ,
  (l1  $\setminus$ u l2) = (l1  $\setminus$ u l2).

```

Proof using. intros. auto. Qed.

Lemma permut_get_2 : $\forall l1\ l2\ l3,$
 $(l1 \setminus u\ l2 \setminus u\ l3) = (l2 \setminus u\ l1 \setminus u\ l3).$

Proof using. intros. apply union_comm_assoc. Qed.

Lemma permut_get_3 : $\forall l1\ l2\ l3\ l4,$
 $(l1 \setminus u\ l2 \setminus u\ l3 \setminus u\ l4) = (l2 \setminus u\ l3 \setminus u\ l1 \setminus u\ l4).$

Proof using.
 intros. do 2 rewrite (union_assoc l2). apply permut_get_2.
 Qed.

Lemma permut_get_4 : $\forall l1\ l2\ l3\ l4\ l5,$
 $(l1 \setminus u\ l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5)$
 $= (l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l1 \setminus u\ l5).$

Proof using.
 intros. do 2 rewrite (union_assoc l2). apply permut_get_3.
 Qed.

Lemma permut_get_5 : $\forall l1\ l2\ l3\ l4\ l5\ l6,$
 $(l1 \setminus u\ l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l6)$
 $= (l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l1 \setminus u\ l6).$

Proof using.
 intros. do 2 rewrite (union_assoc l2). apply permut_get_4.
 Qed.

Lemma permut_get_6 : $\forall l1\ l2\ l3\ l4\ l5\ l6\ l7,$
 $(l1 \setminus u\ l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l6 \setminus u\ l7)$
 $= (l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l6 \setminus u\ l1 \setminus u\ l7).$

Proof using.
 intros. do 2 rewrite (union_assoc l2). apply permut_get_5.
 Qed.

Lemma permut_get_7 : $\forall l1\ l2\ l3\ l4\ l5\ l6\ l7\ l8,$
 $(l1 \setminus u\ l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l6 \setminus u\ l7 \setminus u\ l8)$
 $= (l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l6 \setminus u\ l7 \setminus u\ l1 \setminus u\ l8).$

Proof using.
 intros. do 2 rewrite (union_assoc l2). apply permut_get_6.
 Qed.

Lemma permut_get_8 : $\forall l1\ l2\ l3\ l4\ l5\ l6\ l7\ l8\ l9,$
 $(l1 \setminus u\ l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l6 \setminus u\ l7 \setminus u\ l8 \setminus u\ l9)$
 $= (l2 \setminus u\ l3 \setminus u\ l4 \setminus u\ l5 \setminus u\ l6 \setminus u\ l7 \setminus u\ l8 \setminus u\ l1 \setminus u\ l9).$

Proof using.
 intros. do 2 rewrite (union_assoc l2). apply permut_get_7.
 Qed.

Lemma permut_tactic_setup : $\forall C\ l1\ l2,$

$C (\backslash\{\} \backslash u \textit{ l1} \backslash u \backslash\{\}) (\textit{ l2} \backslash u \backslash\{\}) \rightarrow C \textit{ l1} \textit{ l2}$.
Proof using. intros. *rews_permut_simpl*. eauto. Qed.

Lemma *permut_tactic_keep* : $\forall C \textit{ l1} \textit{ l2} \textit{ l3} \textit{ l4}$,

$C ((\textit{ l1} \backslash u \textit{ l2}) \backslash u \textit{ l3}) \textit{ l4} \rightarrow$

$C (\textit{ l1} \backslash u (\textit{ l2} \backslash u \textit{ l3})) \textit{ l4}$.

Proof using. intros. *rews_permut_simpl*. eauto. Qed.

Lemma *permut_tactic_trans* : $\forall C \textit{ l1} \textit{ l2} \textit{ l3}$,

$\textit{ l3} = \textit{ l2} \rightarrow C \textit{ l1} \textit{ l3} \rightarrow C \textit{ l1} \textit{ l2}$.

Proof using. intros. subst $\neg$. Qed.

End PermutationTactic.

permut_lemma_get *n* returns the lemma *permut_get_n* for the given value of *n*

Ltac *permut_lemma_get* *n* :=

match *number_to_nat* *n* with

| 1%*nat* $\Rightarrow$ constr:(*permut_get_1*)

| 2%*nat* $\Rightarrow$ constr:(*permut_get_2*)

| 3%*nat* $\Rightarrow$ constr:(*permut_get_3*)

| 4%*nat* $\Rightarrow$ constr:(*permut_get_4*)

| 5%*nat* $\Rightarrow$ constr:(*permut_get_5*)

| 6%*nat* $\Rightarrow$ constr:(*permut_get_6*)

| 7%*nat* $\Rightarrow$ constr:(*permut_get_7*)

| 8%*nat* $\Rightarrow$ constr:(*permut_get_8*)

end.

permut_prepare applies to a goal of the form **permut** *l l'* and sets *l* and *l'* in the form *l1* $\backslash u \textit{ l2} \backslash u \dots \backslash u \backslash\{\}$, (some of the lists *li* are put in the form *x*:: $\backslash\{\}$).

Ltac *permut_simpl_prepare* :=

rew_permut_simpl;

apply *permut_tactic_setup*;

repeat rewrite $\leftarrow$ *union_assoc*.

Ltac *permut_index_of* *l lcontainer* :=

match constr:(*lcontainer*) with

| *l* $\backslash u$ _ $\Rightarrow$ constr:(1)

| _ $\backslash u$ *l* $\backslash u$ _ $\Rightarrow$ constr:(2)

| _ $\backslash u$ _ $\backslash u$ *l* $\backslash u$ _ $\Rightarrow$ constr:(3)

| _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ *l* $\backslash u$ _ $\Rightarrow$ constr:(4)

| _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ *l* $\backslash u$ _ $\Rightarrow$ constr:(5)

| _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ *l* $\backslash u$ _ $\Rightarrow$ constr:(6)

| _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ *l* $\backslash u$ _ $\Rightarrow$ constr:(7)

| _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ _ $\backslash u$ *l* $\backslash u$ _ $\Rightarrow$ constr:(8)

| _ $\Rightarrow$ constr:(0)

end.

permut_simplify simplifies a goal of the form **permut** *l l'* where *l* and *l'* are lists built with concatenation and consing

```
Ltac permut_simpl_once_for permut_tactic_simpl :=
```

```
  let go l0 l' :=
    match l0 with
    | _ \u \{} ⇒ fail 1
    | _ \u (?l \u ?lr) ⇒
      match permut_index_of l l' with
      | 0 ⇒ apply permut_tactic_keep
      | ?n ⇒ let F := permut_lemma_get n in
        eapply permut_tactic_trans;
        [ eapply F; try typeclass
          | apply permut_tactic_simpl ]
      end
    end in
  match goal with
  | ⊢ _ ?x ?y ⇒ go x y
  | ⊢ _ _ ?x ?y ⇒ go x y
  | ⊢ _ _ _ ?x ?y ⇒ go x y
  | ⊢ _ _ _ _ ?x ?y ⇒ go x y
  end.
```

```
Ltac permut_conclude :=
```

```
  first [ apply refl_equal
          | apply for_multiset_empty_incl ].
```

```
Ltac permut_simpl_for permut_tactic_simpl :=
```

```
  permut_simpl_prepare;
  repeat permut_simpl_once_for permut_tactic_simpl;
  rew_permut_simpl;
  try permut_conclude.
```

Lemma permut_tactic_simpl_eq : $\forall A (l1\ l2\ l3\ l4:\text{multiset } A),$

$(l1 \setminus u\ l3) = l4 \rightarrow$
 $(l1 \setminus u (l2 \setminus u\ l3)) = (l2 \setminus u\ l4).$

Proof using. intros. subst. apply permut_get_2. Qed.

Lemma permut_tactic_simpl_incl : $\forall A (l1\ l2\ l3\ l4:\text{multiset } A),$

$(l1 \setminus u\ l3) \setminus c\ l4 \rightarrow$
 $(l1 \setminus u (l2 \setminus u\ l3)) \setminus c (l2 \setminus u\ l4).$

Proof using.

```
  introv H. rewrites (union_comm l2 l3). rewrite union_assoc.
  sets l0: (l1 \u l3). rewrite union_comm. apply union_incl_union.
  { apply incl_refl. } { auto. }
```

Qed.

```

Ltac get_premut_tactic_simpl tt :=
  match goal with
  | ⊢ ?x = ?y ⇒ constr:(permut_tactic_simpl_eq)
  | ⊢ ?x \c ?y ⇒ constr:(permut_tactic_simpl_incl)
  end.

Ltac permut_simpl_once :=
  let L := get_premut_tactic_simpl tt in permut_simpl_once_for L.

Ltac permut_simpl :=
  let L := get_premut_tactic_simpl tt in permut_simpl_for L.

Tactic Notation "multiset_eq" :=
  check_noever_goal; permut_simpl.

Section DemoSetUnion.
Variables (A:Type).
Implicit Types l : multiset A.

Lemma demo_multiset_union_permut_simpl_1 :
  ∀ l1 l2 l3 : multiset A,
  (l1 \u l2 \u l3) = (l3 \u l2 \u l1).
Proof using.
  intros.
  permut_simpl_prepare.
  permut_simpl_once.
  permut_simpl_once.
  permut_simpl_once.
  rew_permut_simpl.
  permut_conclude.
Qed.

Lemma demo_multiset_union_permut_simpl_2 :
  ∀
  (x:A) l1 l2 l3 l4,
  (l1 \u \{x} \u l3 \u l2) \c (l1 \u l2 \u l4 \u (\{x} \u l3)).
Proof using.
  intros.
  permut_simpl_prepare.
  permut_simpl_once.
  permut_simpl_once.
  permut_simpl_once.
  permut_simpl_once.
  rew_permut_simpl.
  permut_conclude.
Qed.

Lemma demo_multiset_union_permut_simpl_3 : ∀ (x y:A) l1 l1' l2 l3 l4,

```

$$l1 \setminus c \ l4 \setminus u \ l1' \rightarrow$$

$$(l1 \setminus u (\setminus \{x\} \setminus u \ l2) \setminus u \setminus \{x\} \setminus u (\setminus \{y\} \setminus u \ l3)) \setminus c$$

$$(\setminus \{y\} \setminus u \setminus \{x\} \setminus u (l1' \setminus u \ l2 \setminus u \ l4) \setminus u (\setminus \{x\} \setminus u \ l3)).$$

Proof using.

```

intros. dup.
permut_simpl_prepare.
permut_simpl_once.
permut_simpl_once.
permut_simpl_once.
permut_simpl_once.
permut_simpl_once.
permut_simpl_once.
try permut_simpl_once.
rew_permut_simpl.
equates 1. apply H.
permut_simpl_prepare.
permut_simpl_once.
rew_permut_simpl.
permut_conclude.
permut_simpl. applys_eq H. permut_simpl.

```

Qed.

End DemoSetUnion.

28.6.2 Tactics to prove membership

Section InUnionGet.

Variables (A:Type).

Implicit Types l : multiset A.

Lemma in_union_get_1 : $\forall x \ l1 \ l2,$
 $x \setminus in \ l1 \rightarrow x \setminus in (l1 \setminus u \ l2).$

Proof using. intros. apply in_union_l. auto. Qed.

Lemma in_union_get_2 : $\forall x \ l1 \ l2 \ l3,$
 $x \setminus in \ l2 \rightarrow x \setminus in (l1 \setminus u \ l2 \setminus u \ l3).$

Proof using. intros. apply in_union_r. apply $\neg$ in_union_get_1. Qed.

Lemma in_union_get_3 : $\forall x \ l1 \ l2 \ l3 \ l4,$
 $x \setminus in \ l3 \rightarrow x \setminus in (l1 \setminus u \ l2 \setminus u \ l3 \setminus u \ l4).$

Proof using. intros. apply in_union_r. apply $\neg$ in_union_get_2. Qed.

Lemma in_union_get_4 : $\forall x \ l1 \ l2 \ l3 \ l4 \ l5,$
 $x \setminus in \ l4 \rightarrow x \setminus in (l1 \setminus u \ l2 \setminus u \ l3 \setminus u \ l4 \setminus u \ l5).$

Proof using. intros. apply in_union_r. apply $\neg$ in_union_get_3. Qed.

Lemma in_union_get_5 : $\forall x \ l1 \ l2 \ l3 \ l4 \ l5 \ l6,$

```

 $x \in l5 \rightarrow x \in (l1 \cup l2 \cup l3 \cup l4 \cup l5 \cup l6).$ 
Proof using. intros. apply in_union_r. apply  $\neg$  in_union_get_4. Qed.
End InUnionGet.

Arguments in_union_get_1 [A] [x] [l1] [l2].
Arguments in_union_get_2 [A] [x] [l1] [l2] [l3].
Arguments in_union_get_3 [A] [x] [l1] [l2] [l3].
Arguments in_union_get_4 [A] [x] [l1] [l2] [l3] [l4].
Arguments in_union_get_5 [A] [x] [l1] [l2] [l3] [l4] [l5].

 $\text{in\_union\_get}$  solves a goal of the form  $x \in F1 \cup \dots \cup FN$  when there exists an
hypothesis of the form  $x \in Fi$  for some  $i$ .

Ltac  $\text{in\_union\_get} :=$ 
  match goal with  $H: ?x \in ?A \vdash ?x \in ?B \Rightarrow$ 
  match  $B$  with context [A]  $\Rightarrow$ 
  let  $go\ tt :=$  first
    [ apply (in_union_get_1  $H$ )
    | apply (in_union_get_2  $H$ )
    | apply (in_union_get_3  $H$ )
    | apply (in_union_get_4  $H$ )
    | apply (in_union_get_5  $H$ ) ] in
  first [  $go\ tt$ 
    | rewrite  $\leftarrow$  (for_multiset_union_empty_r  $B$ );
      repeat rewrite  $\leftarrow$  for_multiset_union_assoc;
       $go\ tt$  ]
  end end.

Section InUnionExtract.
Variables ( $A$ :Type).
Implicit Types  $l$  : multiset  $A$ .
Lemma in_union_extract_1 :  $\forall x\ l1,$ 
   $x \in (\{x\} \cup l1).$ 
Proof using. intros. apply in_union_get_1. apply in_single_self. Qed.
Lemma in_union_extract_2 :  $\forall x\ l1\ l2,$ 
   $x \in (l1 \cup \{x\} \cup l2).$ 
Proof using. intros. apply in_union_get_2. apply in_single_self. Qed.
Lemma in_union_extract_3 :  $\forall x\ l1\ l2\ l3,$ 
   $x \in (l1 \cup l2 \cup \{x\} \cup l3).$ 
Proof using. intros. apply in_union_get_3. apply in_single_self. Qed.
Lemma in_union_extract_4 :  $\forall x\ l1\ l2\ l3\ l4,$ 
   $x \in (l1 \cup l2 \cup l3 \cup \{x\} \cup l4).$ 
Proof using. intros. apply in_union_get_4. apply in_single_self. Qed.
Lemma in_union_extract_5 :  $\forall x\ l1\ l2\ l3\ l4\ l5,$ 

```

```

   $x \in (l1 \cup l2 \cup l3 \cup l4 \cup \{x\} \cup l5)$ .
Proof using. intros. apply in_union_get_5. apply in_single_self. Qed.
End InUnionExtract.

in_union_extract solves a goal of the form  $x \in F1 \cup \dots \cup \{x\} \cup \dots \cup FN$ .
Ltac in_union_extract :=
  match goal with  $\vdash ?x \in ?A \Rightarrow$ 
  match  $A$  with context [ $\{x\}$ ]  $\Rightarrow$ 
  let go tt := first
    [ apply (in_union_extract_1)
    | apply (in_union_extract_2)
    | apply (in_union_extract_3)
    | apply (in_union_extract_4)
    | apply (in_union_extract_5) ] in
  first [ go tt
    | rewrite  $\leftarrow$  (for_multiset_union_empty_r  $A$ );
      repeat rewrite  $\leftarrow$  for_multiset_union_assoc;
      go tt ]
  end end.

Tactic Notation "multiset_in" :=
  first [ in_union_extract | in_union_get ].

```

28.6.3 Tactics to invert a membership hypothesis

```

Section InversionsTactic.
Context ( $A$ :Type).
Implicit Types  $l$  : multiset  $A$ .
Implicit Types  $x$  :  $A$ .

Lemma empty_eq_single_inv_1 :  $\forall x \ l1 \ l2$ ,
   $l1 = l2 \rightarrow x \notin l1 \rightarrow x \in l2 \rightarrow \text{False}$ .
Proof using. intros. subst $\times$ . Qed.

Lemma empty_eq_single_inv_2 :  $\forall x \ l1 \ l2$ ,
   $l1 = l2 \rightarrow x \notin l2 \rightarrow x \in l1 \rightarrow \text{False}$ .
Proof using. intros. subst $\times$ . Qed.

Lemma notin_empty :  $\forall x$ ,
   $x \notin (\{\}: \text{multiset } A)$ .
Proof using. intros. unfold notin. rewrite in_empty_eq. auto. Qed.

End InversionsTactic.
#[global]
Hint Resolve notin_empty.

Ltac in_union_meta :=

```



```

match goal with
|  $\vdash \_ \setminus \text{in } \{ \_ \} \Rightarrow$  apply in_single_self
|  $\vdash \_ \setminus \text{in } \{ \_ \} \setminus \text{u } \_ \Rightarrow$  apply in_union_l; apply in_single_self
|  $\vdash \_ \setminus \text{in } \_ \setminus \text{u } \_ \Rightarrow$  apply in_union_r; in_union_meta
end.

Ltac fseq_inv_core  $H :=$ 
  let go  $L :=$ 
    false  $L$ ; [ apply  $H$ 
      | try apply notin_empty
      | try in_union_meta ] in
  match type of  $H$  with
  |  $\{ \} = \_ \Rightarrow$  go empty_eq_single_inv_1
  |  $\_ = \{ \} \Rightarrow$  go empty_eq_single_inv_2
  |  $\_ = \_ \Rightarrow$  go empty_eq_single_inv_1
  end.

multiset_inv  $H$  solves a goal by contraction if there exists an hypothesis of the form
 $\{ \} = E1 \setminus \text{u } \dots \setminus \text{u } \{ x \} \setminus \text{u } \dots \setminus \text{u } EN$  (or its symmetric). multiset_inv speculates on the
hypothesis  $H$  to be used.

Tactic Notation "multiset_inv" constr( $H$ ) :=
  fseq_inv_core  $H$ .

Tactic Notation "multiset_inv" :=
  match goal with
  |  $\vdash \{ \} \neq \_ \Rightarrow$  let  $H :=$  fresh in intro  $H$ ; multiset_inv  $H$ 
  |  $\vdash \_ \neq \{ \} \Rightarrow$  let  $H :=$  fresh in intro  $H$ ; multiset_inv  $H$ 
  |  $H : \{ \} = \_ \vdash \_ \Rightarrow$  multiset_inv  $H$ 
  |  $H : \_ = \{ \} \vdash \_ \Rightarrow$  multiset_inv  $H$ 
  end.

Section InUnionInv.
Variables ( $A$ :Type).
Implicit Types  $l$  : multiset  $A$ .

Lemma in_empty_inv :  $\forall x$ ,
   $x \setminus \text{in } (\{ \} : \text{multiset } A) \rightarrow \text{False}$ .
Proof using. introv. apply notin_empty. Qed.

Lemma in_single_inv :  $\forall x y : A$ ,
   $x \setminus \text{in } (\{ y \} : \text{multiset } A) \rightarrow x = y$ .
Proof using. intros. rewrite @in_single_eq in  $H$ . auto. typeclass. Qed.

Lemma in_union_inv :  $\forall x l1 l2$ ,
   $x \setminus \text{in } (l1 \setminus \text{u } l2) \rightarrow x \setminus \text{in } l1 \vee x \setminus \text{in } l2$ .
Proof using. introv  $H$ . rewrite @in_union_eq in  $H$ . auto. typeclass. Qed.

End InUnionInv.

```

```

Arguments in_single_inv [A] [x] [y].
Arguments in_union_inv [A] [x] [l1] [l2].

Ltac multiset_in_inv_base H M :=
  match type of H with
  | _ \in \{\} => false; apply (@in_empty_inv _ _ H)
  | _ \in \{-\} =>
    generalize (in_single_inv H); try clear H; try intro_subst_hyp
  | _ \in _ \u _ =>
    let H' := fresh "TEMP" in
    destruct (in_union_inv H) as [H'|H'];
    try clear H; multiset_in_inv_base H' M
  | _ => rename H into M
  end.

Tactic Notation "multiset_in" constr(H) "as" ident(M) :=
  multiset_in_inv_base H M.

Tactic Notation "multiset_in" constr(H) :=
  let M := fresh "H" in multiset_in H as M.

Lemma union_empty_inv_multiset :  $\forall A (l1\ l2:\text{multiset } A),$ 
   $l1 \setminus u\ l2 = \{\} \rightarrow l1 = \{\} \wedge l2 = \{\}.$ 
Proof using. intros. eapply union_empty_inv_inst. eauto. Qed.

Arguments union_empty_inv_multiset [A] [l1] [l2].

Ltac multiset_empty_core H :=
  match type of H with
  | _ \u _ = \{\} =>
    let H1 := fresh "M" in let H2 := fresh "M" in
    destruct (union_empty_inv_multiset H) as [H1 H2]; clear H;
    multiset_empty_core H1; multiset_empty_core H2
  | ?E = \{\} => try subst E; try solve [ false ]
  | \{\} = _ \u _ => symmetry in H; multiset_empty_core H
  | \{\} = ?E => symmetry in H; multiset_empty_core H
  end.

Tactic Notation "multiset_empty" "in" hyp(H) :=
  multiset_empty_core H.

Tactic Notation "multiset_empty" :=
  let H := fresh "M" in intro H; multiset_empty in H.

Tactic Notation "multiset_empty" constr(E) :=
  let H := fresh "M" in lets H:E; multiset_empty in H.

```

Chapter 29

Library SLF.LibCore

```
From SLF Require Export LibTactics LibLogic LibOperation LibReflect.  
From SLF Require Export LibUnit LibProd LibSum LibOption LibNat LibInt LibList.  
From SLF Require Export LibRelation LibOrder LibWf.  
Export LIBTACTICSCOMPATIBILITY.  
  
Open Scope Z_scope.  
Open Scope Int_scope.  
Open Scope comp_scope.
```

Chapter 30

Library **SLF.LibSepTLCbuffer**

30.1 LibSepTLCbuffer: Appendix - Temporary Additions to TLC

This file contains temporary definitions that will eventually get merged into the various files from the TLC library.

```
Set Implicit Arguments.  
From SLF Require Import LibTactics.  
From SLF Require Import LibInt.  
Generalizable Variables A B.  
  
Global Opaque Z.mul.  
Global Opaque Z.add.
```

Chapter 31

Library `SLF.LibSepFmap`

31.1 `LibSepFmap`: Appendix - Finite Maps

```
Set Implicit Arguments.  
From SLF Require Import LibCore.
```

31.2 Representation of Finite Maps

This file provides a representation of finite maps, which may be used to represent the memory state of a program.

- It implements basic operations such as creation of a singleton map, union of maps, read and update operations.
- It includes predicates to assert disjointness of two maps (predicate `disjoint`), and coherence of two maps on the intersection of their domain (predicate `agree`).
- It comes with a tactic for `fmap_eq` proving equalities modulo commutativity and associativity of map union.

The definition of the type `fmap` is slightly technical in that it involves a dependent pair to pack the partial function of type $A \rightarrow \text{option } B$ that represents the map, together with a proof of finiteness of the domain of this map. One useful lemma established in this file is the existence of fresh keys: for any finite map whose keys are natural numbers, there exists a natural number that does not already belong to the domain of that map.

31.2.1 Representation of Potentially-Infinite Maps as Partial Functions

Representation

Type of partial functions from A to B

Definition `map` ($A\ B : \text{Type}$) : $\text{Type} :=$
 $A \rightarrow \text{option } B$.

Operations

Disjoint union of two partial functions

Definition `map_union` ($A\ B : \text{Type}$) ($f1\ f2 : \text{map } A\ B$) : $\text{map } A\ B :=$
`fun` ($x:A$) $\Rightarrow$ `match` $f1\ x$ `with`
 $\quad | \text{Some } y \Rightarrow \text{Some } y$
 $\quad | \text{None} \Rightarrow f2\ x$
`end`.

Removal from a partial functions

Definition `map_remove` ($A\ B : \text{Type}$) ($f : \text{map } A\ B$) ($k:A$) : $\text{map } A\ B :=$
`fun` ($x:A$) $\Rightarrow$ `If` $x = k$ `then` `None` `else` $f\ x$.

Finite domain of a partial function

Definition `map_finite` ($A\ B : \text{Type}$) ($f : \text{map } A\ B$) :=
 $\exists (L : \text{list } A), \forall (x:A), f\ x \neq \text{None} \rightarrow \text{mem } x\ L$.

Disjointness of domain of two partial functions

Definition `map_disjoint` ($A\ B : \text{Type}$) ($f1\ f2 : \text{map } A\ B$) :=
 $\forall (x:A), f1\ x = \text{None} \vee f2\ x = \text{None}$.

Compatibility of two partial functions on the intersection of their domains (only for Separation Logic with RO-permissions)

Definition `map_agree` ($A\ B : \text{Type}$) ($f1\ f2 : \text{map } A\ B$) :=
 $\forall x\ v1\ v2,$
 $f1\ x = \text{Some } v1 \rightarrow$
 $f2\ x = \text{Some } v2 \rightarrow$
 $v1 = v2$.

Domain of a map (as a predicate)

Definition `map_indom` ($A\ B : \text{Type}$) ($f1 : \text{map } A\ B$) : ($A \rightarrow \text{Prop}$) :=
`fun` ($x:A$) $\Rightarrow f1\ x \neq \text{None}$.

Filter the bindings of a map

Definition `map_filter` $A\ B$ ($F:A \rightarrow B \rightarrow \text{Prop}$) ($f:\text{map } A\ B$) : $\text{map } A\ B :=$
`fun` ($x:A$) $\Rightarrow$ `match` $f\ x$ `with`
 $\quad | \text{None} \Rightarrow \text{None}$
 $\quad | \text{Some } y \Rightarrow \text{If } F\ x\ y \text{ then } \text{Some } y \text{ else } \text{None}$
`end`.

Map a function on the values of a map

Definition `map_map` $A\ B1\ B2$ ($F:A \rightarrow B1 \rightarrow B2$) ($f:\text{map } A\ B1$) : $\text{map } A\ B2 :=$

```

fun (x:A) ⇒ match f x with
| None ⇒ None
| Some y ⇒ Some (F x y)
end.

```

Properties

Section MapOps.

Variables ($A B : \text{Type}$).

Implicit Types $f : \text{map } A B$.

Disjointness and domains

Lemma map_disjoint_eq : $\forall f1 f2$,
 map_disjoint $f1 f2 = (\forall x, \text{map\_indom } f1 x \rightarrow \text{map\_indom } f2 x \rightarrow \text{False})$.

Proof using.

```

intros. unfold map_disjoint, map_indom. extens. iff M.
{ intros x. specializes M x. gen M. case_eq (f1 x); case_eq (f2 x); auto_false×. }
{ intros x. specializes M x. gen M. case_eq (f1 x); case_eq (f2 x); auto_false×.
  { intros. false. applys M; auto_false. } }

```

Qed.

Symmetry of disjointness

Lemma map_disjoint_sym :
 sym (@map_disjoint $A B$).

Proof using.

```

introv H. unfolds map_disjoint. intros z. specializes H z. intuition.

```

Qed.

Commutativity of disjoint union

Lemma map_union_comm : $\forall f1 f2$,
 map_disjoint $f1 f2 \rightarrow$
 map_union $f1 f2 = \text{map\_union } f2 f1$.

Proof using.

```

introv H. unfold map.
extens. intros x. unfolds map_disjoint, map_union.
specializes H x. cases (f1 x); cases (f2 x); auto. destruct H; false.

```

Qed.

Finiteness of union

Lemma map_union_finite : $\forall f1 f2$,
 map_finite $f1 \rightarrow$
 map_finite $f2 \rightarrow$
 map_finite (map_union $f1 f2$).

Proof using.

```

  introv [L1 F1] [L2 F2].  $\exists$  (L1 ++ L2). intros x M.
  specializes F1 x. specializes F2 x. unfold map_union in M.
  apply mem_app. destruct  $\neg$  (f1 x).
Qed.

```

Finiteness of removal

```

Definition map_remove_finite :  $\forall$  x f,
  map_finite f  $\rightarrow$ 
  map_finite (map_remove f x).

```

Proof using.

```

  introv [L F].  $\exists$  L. intros x' M.
  specializes F x'. unfold map_remove in M. case_if  $\neg$ .
Qed.

```

Finiteness of filter

```

Definition map_filter_finite :  $\forall$  (F:A $\rightarrow$ B $\rightarrow$ Prop) f,
  map_finite f  $\rightarrow$ 
  map_finite (map_filter F f).

```

Proof using.

```

  introv [L N].  $\exists$  L. intros x' M.
  specializes N x'. unfold map_filter in M.
  destruct (f x'); tryfalse. case_if. applies N; auto_false.
Qed.

```

Finiteness of map

```

Definition map_map_finite :  $\forall$  C (F:A $\rightarrow$ B $\rightarrow$ C) f,
  map_finite f  $\rightarrow$ 
  map_finite (map_map F f).

```

Proof using.

```

  introv [L N].  $\exists$  L. intros x' M.
  specializes N x'. unfold map_map in M.
  destruct (f x'); tryfalse. applies N; auto_false.
Qed.

```

End MapOps.

31.2.2 Representation of Finite Maps as Dependent Pairs

Representation

```

Inductive fmap (A B : Type) : Type := make {
  fmap_data :> map A B;
  fmap_finite : map_finite fmap_data }.

```

Arguments make [A] [B].

Operations

Declare Scope *fmap_scope*.

Domain of a fmap (as a predicate)

Definition *indom* (*A B* : Type) (*h* : *fmap A B*) : (*A* → Prop) :=
map_indom *h*.

Empty fmap

Program Definition *empty* (*A B* : Type) : *fmap A B* :=
make (fun *l* ⇒ None) _.
Next Obligation. ∃ (@nil *A*). *auto_false*. Qed.

Arguments *empty* {*A*} {*B*}.

Singleton fmap

Program Definition *single* *A B* (*x* : *A*) (*v* : *B*) : *fmap A B* :=
make (fun *x'* ⇒ If *x* = *x'* then Some *v* else None) _.
Next Obligation.
∃ (*x* :: nil). intros. *case_if*. subst⊢.
Qed.

Union of fmaps

Program Definition *union* *A B* (*h1 h2* : *fmap A B*) : *fmap A B* :=
make (map_union *h1 h2*) _.
Next Obligation. destruct *h1*. destruct *h2*. apply⊢ map_union_finite. Qed.
Notation "*h1* \+ *h2*" := (*union h1 h2*)
(at level 51, right associativity) : *fmap_scope*.
Open Scope *fmap_scope*.

Update of a fmap

Definition *update* *A B* (*h* : *fmap A B*) (*x* : *A*) (*v* : *B*) : *fmap A B* :=
union (single *x v*) *h*.

Read in a fmap

Definition *read* (*A B* : Type) {*IB* : Inhab *B*} (*h* : *fmap A B*) (*x* : *A*) : *B* :=
match fmap_data *h x* with
| Some *y* ⇒ *y*
| None ⇒ arbitrary
end.

Removal from a fmap

Program Definition *remove* *A B* (*h* : *fmap A B*) (*x* : *A*) : *fmap A B* :=
make (map_remove *h x*) _.
Next Obligation. destruct *h*. apply⊢ map_remove_finite. Qed.

Filter from a fmap

```
Program Definition filter A B (F:A→B→Prop) (h:fmap A B) : fmap A B :=  
  make (map_filter F h) _.
```

Next Obligation. destruct *h*. apply¬ map_filter_finite. Qed.

Map a function onto the keys of a fmap

```
Program Definition map_ A B1 B2 (F:A→B1→B2) (h:fmap A B1) : fmap A B2 :=  
  make (map_map F h) _.
```

Next Obligation. destruct *h*. apply¬ map_map_finite. Qed.

Inhabited type **fmap**

```
Global Instance Inhab_fmap A B : Inhab (fmap A B).
```

Proof using. intros. *applies* Inhab_of_val (@empty A B). Qed.

31.2.3 Predicates on Finite Maps

Compatible fmaps (only for Separation Logic with RO-permissions)

```
Definition agree A B (h1 h2:fmap A B) :=  
  map_agree h1 h2.
```

Disjoint fmaps

```
Definition disjoint A B (h1 h2 : fmap A B) : Prop :=  
  map_disjoint h1 h2.
```

Three disjoint fmaps (not needed for basic separation logic)

```
Definition disjoint_3 A B (h1 h2 h3 : fmap A B) :=  
  disjoint h1 h2  
  ∧ disjoint h2 h3  
  ∧ disjoint h1 h3.
```

Notation for disjointness

```
Module NOTATIONFORFMAPDISJOINT.
```

```
Notation "\# h1 h2" := (disjoint h1 h2)  
  (at level 40, h1 at level 0, h2 at level 0, no associativity) : fmap_scope.
```

```
Notation "\# h1 h2 h3" := (disjoint_3 h1 h2 h3)  
  (at level 40, h1 at level 0, h2 at level 0, h3 at level 0, no associativity)  
  : fmap_scope.
```

```
End NOTATIONFORFMAPDISJOINT.
```

```
Import NOTATIONFORFMAPDISJOINT.
```

31.3 Properties of Operations on Finite Maps

Section `FmapProp`.

Variables $(A\ B : \text{Type})$.

Implicit Types $f\ g\ h : \text{fmap}\ A\ B$.

Implicit Types $x : A$.

Implicit Types $v : B$.

31.3.1 Equality

Lemma `make_eq` : $\forall (f1\ f2 : \text{map}\ A\ B)\ F1\ F2,$
 $(\forall x, f1\ x = f2\ x) \rightarrow$
`make` $f1\ F1 = \text{make}\ f2\ F2$.

Proof using.

`intros H. asserts: (f1 = f2). { unfold map. extens $\neg$ . }`
`subst. fequals. Qed.`

Lemma `eq_inv_fmap_data_eq` : $\forall h1\ h2,$
 $h1 = h2 \rightarrow$
 $\forall x, \text{fmap\_data}\ h1\ x = \text{fmap\_data}\ h2\ x$.

Proof using. `intros. fequals. Qed.`

Lemma `fmap_extens` : $\forall h1\ h2,$
 $(\forall x, \text{fmap\_data}\ h1\ x = \text{fmap\_data}\ h2\ x) \rightarrow$
 $h1 = h2$.

Proof using. `intros [f1 F1] [f2 F2] M. simpls. applies $\neg$  make_eq. Qed.`

31.3.2 Domain

Lemma `indom_single_eq` : $\forall x\ y\ v,$
 $\text{indom}(\text{single}\ x\ v)\ y = (x = y)$.

Proof using.

`intros. unfold single, indom, map_indom. simpl. extens. case_if; auto_false.`
`Qed.`

Lemma `indom_single` : $\forall x\ v,$
 $\text{indom}(\text{single}\ x\ v)\ x$.

Proof using. `intros. rewrite $\times$  indom_single_eq. Qed.`

Lemma `indom_union_eq` : $\forall h1\ h2\ x,$
 $\text{indom}(\text{union}\ h1\ h2)\ x = (\text{indom}\ h1\ x \vee \text{indom}\ h2\ x)$.

Proof using.

`intros. extens. unfolds indom, union, map_indom, map_union. simpls.`
`case_eq (fmap_data h1 x); case_eq (fmap_data h2 x); auto_false $\times$ .`
`Qed.`

Lemma indom_union_l : $\forall h1\ h2\ x,$
 indom $h1\ x \rightarrow$
 indom (union $h1\ h2$) x .
 Proof using. intros. rewrite $\times$ indom_union_eq. Qed.
 Lemma indom_union_r : $\forall h1\ h2\ x,$
 indom $h2\ x \rightarrow$
 indom (union $h1\ h2$) x .
 Proof using. intros. rewrite $\times$ indom_union_eq. Qed.
 Lemma indom_update_eq : $\forall h\ x\ v\ y,$
 indom (update $h\ x\ v$) $y = (x = y \vee \text{indom } h\ y)$.
 Proof using.
 intros. unfold update. rewrite indom_union_eq, indom_single_eq. auto.
 Qed.
 Lemma indom_remove_eq : $\forall h\ x\ y,$
 indom (remove $h\ x$) $y = (x \neq y \wedge \text{indom } h\ y)$.
 Proof using.
 intros. *extens. unfolds* indom, remove, map_indom, map_remove. *simpls.*
 case_if; auto_false $\times$.
 Qed.

31.3.3 Disjoint and Domain

Lemma disjoint_eq : $\forall h1\ h2,$
 disjoint $h1\ h2 = (\forall x, \text{indom } h1\ x \rightarrow \text{indom } h2\ x \rightarrow \text{False})$.
 Proof using. intros $[f1\ F1]\ [f2\ F2]$. apply map_disjoint_eq. Qed.
 Lemma disjoint_eq' : $\forall h1\ h2,$
 disjoint $h1\ h2 = (\forall x, \text{indom } h1\ x \rightarrow \text{indom } h2\ x \rightarrow \text{False})$.
 Proof using.
 extens. iff $D\ E$.
 { *introv* $M1\ M2$. *destruct* ($D\ x$); *false* $\times$. }
 { *intros* x . *specializes* $E\ x$. *unfolds* indom, map_indom.
 applies not_not_inv. *intros* N . *rew_logic* in N . *false* $\times$. }
 Qed.
 Lemma disjoint_of_not_indom_both : $\forall h1\ h2,$
 $(\forall x, \text{indom } h1\ x \rightarrow \text{indom } h2\ x \rightarrow \text{False}) \rightarrow$
 disjoint $h1\ h2$.
 Proof using. *introv* M . rewrite $\neg$ disjoint_eq. Qed.
 Lemma disjoint_inv_not_indom_both : $\forall h1\ h2\ x,$
 disjoint $h1\ h2 \rightarrow$
 indom $h1\ x \rightarrow$
 indom $h2\ x \rightarrow$

False.

Proof using. *intros*. *rewrite*× *disjoint_eq*. Qed.

Lemma *disjoint_single_of_not_indom* : $\forall h\ x\ v,$
 $\neg \text{indom } h\ x \rightarrow$
 $\text{disjoint } (\text{single } x\ v)\ h.$

Proof using.

intros N. *unfolds* *disjoint*, *map_disjoint*. *unfolds* *single*, *indom*, *map_indom*.
simpl. *rew_logic* in *N*. *intros l'*. *case_if*; *subst*; *autos*×.

Qed.

Note that the reciprocal of the above lemma is a special instance of *disjoint_inv_not_indom_both*

31.3.4 Disjointness

Lemma *disjoint_sym* : $\forall h1\ h2,$
 $\backslash\# h1\ h2 \rightarrow$
 $\backslash\# h2\ h1.$

Proof using. *intros [f1 F1] [f2 F2]*. *apply* *map_disjoint_sym*. Qed.

Lemma *disjoint_comm* : $\forall h1\ h2,$
 $\backslash\# h1\ h2 = \backslash\# h2\ h1.$

Proof using. *lets*: *disjoint_sym*. *extens*×. Qed.

Lemma *disjoint_empty_l* : $\forall h,$
 $\backslash\# \text{empty } h.$

Proof using. *intros [f F] x*. *simple*¬. Qed.

Lemma *disjoint_empty_r* : $\forall h,$
 $\backslash\# h\ \text{empty}.$

Proof using. *intros [f F] x*. *simple*¬. Qed.

Hint Immediate *disjoint_sym*.

Hint Resolve *disjoint_empty_l* *disjoint_empty_r*.

Lemma *disjoint_union_eq_r* : $\forall h1\ h2\ h3,$
 $\backslash\# h1\ (h2\ \backslash+ h3) =$
 $(\backslash\# h1\ h2\ \wedge\ \backslash\# h1\ h3).$

Proof using.

intros [f1 F1] [f2 F2] [f3 F3].
unfolds *disjoint*, *union*. *simpls*.
unfolds *map_disjoint*, *map_union*. *extens*. *iff* *M* [*M1 M2*].
split; *intros x*; *specializes* *M* *x*;
destruct (*f2* *x*); *intuition*; *tryfalse*.
intros x. *specializes* *M1* *x*. *specializes* *M2* *x*.
destruct (*f2* *x*); *intuition*.

Qed.

Lemma disjoint_union_eq_l : $\forall h1\ h2\ h3,$
 $\backslash\# (h2\ \backslash+ h3)\ h1 =$
 $(\backslash\# h1\ h2 \wedge \backslash\# h1\ h3).$

Proof using.

 intros. rewrite disjoint_comm.
 apply disjoint_union_eq_r.

Qed.

Lemma disjoint_single_single : $\forall (x1\ x2:A)\ (v1\ v2:B),$
 $x1 \neq x2 \rightarrow$
 $\backslash\# (\text{single } x1\ v1)\ (\text{single } x2\ v2).$

Proof using.

 introv N. intros x. simpls. do 2 case_if; auto.

Qed.

Lemma disjoint_single_single_same_inv : $\forall (x:A)\ (v1\ v2:B),$
 $\backslash\# (\text{single } x\ v1)\ (\text{single } x\ v2) \rightarrow$
 False.

Proof using.

 introv D. specializes D x. simpls. case_if. destruct D; tryfalse.

Qed.

Lemma disjoint_3_unfold : $\forall h1\ h2\ h3,$
 $\backslash\# h1\ h2\ h3 = (\backslash\# h1\ h2 \wedge \backslash\# h2\ h3 \wedge \backslash\# h1\ h3).$

Proof using. auto. Qed.

Lemma disjoint_single_set : $\forall h\ l\ v1\ v2,$
 $\backslash\# (\text{single } l\ v1)\ h \rightarrow$
 $\backslash\# (\text{single } l\ v2)\ h.$

Proof using.

 introv M. unfolds disjoint, single, map_disjoint; simpls.
 intros l'. specializes M l'. case_if $\neg$. destruct M; auto_false.

Qed.

Lemma disjoint_update_l : $\forall h1\ h2\ x\ v,$
 $\text{disjoint } h1\ h2 \rightarrow$
 $\text{indom } h1\ x \rightarrow$
 $\text{disjoint } (\text{update } h1\ x\ v)\ h2.$

Proof using.

 introv HD Hx. rewrite disjoint_eq in *. intros y Hy1 Hy2.
 rewrite indom_update_eq in Hy1. destruct Hy1.
 { subst. false $\times$. }
 { applies $\times$ HD y. }

Qed.

Lemma disjoint_update_not_r : $\forall h1\ h2\ x\ v,$

disjoint $h1\ h2 \rightarrow$
 $\neg \text{indom } h2\ x \rightarrow$
disjoint (update $h1\ x\ v$) $h2$.

Proof using.

```
introv HD Hx. rewrite disjoint_eq in *. intros y Hy1 Hy2.
rewrite indom_update_eq in Hy1. destruct Hy1.
{ subst. false×. }
{ applies× HD y. }
```

Qed.

Lemma disjoint_remove_l : $\forall\ h1\ h2\ x$,
disjoint $h1\ h2 \rightarrow$
disjoint (remove $h1\ x$) $h2$.

Proof using.

```
introv M. rewrite disjoint_eq in *. intros y Hy1 Hy2.
rewrite× indom_remove_eq in Hy1.
```

Qed.

31.3.5 Union

Lemma union_self : $\forall\ h$,
 $h \setminus + h = h$.

Proof using.

```
intros [f F]. apply¬ make_eq. simpl. intros x.
unfold map_union. cases¬ (f x).
```

Qed.

Lemma union_empty_l : $\forall\ h$,
empty $\setminus + h = h$.

Proof using.

```
intros [f F]. unfold union, map_union, empty. simpl.
apply¬ make_eq.
```

Qed.

Lemma union_empty_r : $\forall\ h$,
 $h \setminus + \text{empty} = h$.

Proof using.

```
intros [f F]. unfold union, map_union, empty. simpl.
apply make_eq. intros x. destruct¬ (f x).
```

Qed.

Lemma union_eq_empty_inv_l : $\forall\ h1\ h2$,
 $h1 \setminus + h2 = \text{empty} \rightarrow$
 $h1 = \text{empty}$.

Proof using.

```
intros (f1&F1) (f2&F2) M. inverts M as M.
```

```

applies make_eq. intros l.
unfolds map_union.
lets: fun_eq_1 l M.
cases (f1 l); auto_false.

```

Qed.

```

Lemma union_eq_empty_inv_r :  $\forall h1\ h2,$ 
   $h1 \setminus + h2 = \text{empty} \rightarrow$ 
   $h2 = \text{empty}.$ 

```

Proof using.

```

intros (f1 & F1) (f2 & F2) M. inverts M as M.
applies make_eq. intros l.
unfolds map_union.
lets: fun_eq_1 l M.
cases (f1 l); auto_false.

```

Qed.

```

Lemma union_comm_of_disjoint :  $\forall h1\ h2,$ 
   $\setminus \# h1\ h2 \rightarrow$ 
   $h1 \setminus + h2 = h2 \setminus + h1.$ 

```

Proof using.

```

intros [f1 F1] [f2 F2] H. simpls. apply make_eq. simpl.
intros. rewrite  $\neg$  map_union_comm.

```

Qed.

```

Lemma union_comm_of_agree :  $\forall h1\ h2,$ 
  agree h1 h2  $\rightarrow$ 
   $h1 \setminus + h2 = h2 \setminus + h1.$ 

```

Proof using.

```

intros [f1 F1] [f2 F2] H. simpls. apply make_eq. simpl.
intros l. specializes H l. unfolds map_union. simpls.
cases (f1 l); cases (f2 l); auto. fequals. applies  $\times$  H.

```

Qed.

```

Lemma union_assoc :  $\forall h1\ h2\ h3,$ 
   $(h1 \setminus + h2) \setminus + h3 = h1 \setminus + (h2 \setminus + h3).$ 

```

Proof using.

```

intros [f1 F1] [f2 F2] [f3 F3]. unfolds union. simpls.
apply make_eq. intros x. unfold map_union. destruct  $\neg$  (f1 x).

```

Qed.

```

Lemma union_eq_inv_of_disjoint :  $\forall h2\ h1\ h1',$ 
   $\setminus \# h1\ h2 \rightarrow$ 
   $\setminus \# h1'\ h2 \rightarrow$ 
   $h1 \setminus + h2 = h1' \setminus + h2 \rightarrow$ 
   $h1 = h1'.$ 

```


Proof using.

```
intros [f2 F2] [f1' F1'] [f1 F1] D D' E.  
applies make_eq. intros x. specializes D x. specializes D' x.  
lets E': eq_inv_fmap_data_eq (rm E) x. simpls.  
unfolds map_union.  
cases (f1' x); cases (f1 x);  
  destruct D; try congruence;  
  destruct D'; try congruence.
```

Qed.

31.3.6 Compatibility

Lemma agree_refl : $\forall h$,
 agree h h .

Proof using.

```
intros h. introv M1 M2. congruence.
```

Qed.

Lemma agree_sym : $\forall f1\ f2$,
 agree $f1\ f2 \rightarrow$
 agree $f2\ f1$.

Proof using.

```
introv M. intros l v1 v2 E1 E2.  
specializes M l E1.
```

Qed.

Lemma agree_of_disjoint : $\forall h1\ h2$,
 disjoint $h1\ h2 \rightarrow$
 agree $h1\ h2$.

Proof using.

```
introv HD. intros l v1 v2 M1 M2. destruct (HD l); false.
```

Qed.

Lemma agree_empty_l : $\forall h$,
 agree empty h .

Proof using. intros $h\ l\ v1\ v2\ E1\ E2$. simpls. false. Qed.

Lemma agree_empty_r : $\forall h$,
 agree h empty.

Proof using.

```
hint agree_sym, agree_empty_l. eauto.
```

Qed.

Lemma agree_union_l : $\forall f1\ f2\ f3$,
 agree $f1\ f3 \rightarrow$
 agree $f2\ f3 \rightarrow$

```

    agree (f1 \+ f2) f3.
Proof using.
  introv M1 M2. intros l v1 v2 E1 E2.
  specializes M1 l. specializes M2 l.
  simpls. unfolds map_union. cases (fmap_data f1 l).
  { inverts E1. applies× M1. }
  { applies× M2. }

```

Qed.

```

Lemma agree_union_r : ∀ f1 f2 f3,
  agree f1 f2 →
  agree f1 f3 →
  agree f1 (f2 \+ f3).

```

Proof using.

```

  hint agree_sym, agree_union_l. eauto.

```

Qed.

```

Lemma agree_union_lr : ∀ f1 g1 f2 g2,
  agree g1 g2 →
  \# f1 f2 (g1 \+ g2) →
  agree (f1 \+ g1) (f2 \+ g2).

```

Proof using.

```

  introv M1 (M2a&M2b&M2c).
  rewrite disjoint_union_eq_r in *.
  applies agree_union_l; applies agree_union_r;
  try solve [ applies× agree_of_disjoint ].
  auto.

```

Qed.

```

Lemma agree_union_ll_inv : ∀ f1 f2 f3,
  agree (f1 \+ f2) f3 →
  agree f1 f3.

```

Proof using.

```

  introv M. intros l v1 v2 E1 E2.
  specializes M l. simpls. unfolds map_union.
  rewrite E1 in M. applies× M.

```

Qed.

```

Lemma agree_union_rl_inv : ∀ f1 f2 f3,
  agree f1 (f2 \+ f3) →
  agree f1 f2.

```

Proof using.

```

  hint agree_union_ll_inv, agree_sym. eauto.

```

Qed.

```

Lemma agree_union_lr_inv_agree_agree : ∀ f1 f2 f3,

```

```

agree (f1 \+ f2) f3 →
agree f1 f2 →
agree f2 f3.
Proof using.
  introv M D. rewrite¬ (@union_comm_of_agree f1 f2) in M.
  applies× agree_union_ll_inv.
Qed.

Lemma agree_union_rr_inv_agree : ∀ f1 f2 f3,
  agree f1 (f2 \+ f3) →
  agree f2 f3 →
  agree f1 f3.
Proof using.
  hint agree_union_lr_inv_agree_agree, agree_sym. eauto.
Qed.

Lemma agree_union_l_inv : ∀ f1 f2 f3,
  agree (f1 \+ f2) f3 →
  agree f1 f2 →
  agree f1 f3
  ∧ agree f2 f3.
Proof using.
  introv M1 M2. split.
  { applies× agree_union_ll_inv. }
  { applies× agree_union_lr_inv_agree_agree. }
Qed.

Lemma agree_union_r_inv : ∀ f1 f2 f3,
  agree f1 (f2 \+ f3) →
  agree f2 f3 →
  agree f1 f2
  ∧ agree f1 f3.
Proof using.
  hint agree_sym.
  intros. forwards¬ (M1&M2): agree_union_l_inv f2 f3 f1.
Qed.

```

31.3.7 Read

```

Lemma read_single : ∀ {IB:Inhab B} x v,
  read (single x v) x = v.
Proof using.
  intros. unfold read, single. simpl. case-if¬.
Qed.

```

Lemma read_union_l : $\forall \{IB:\text{Inhab } B\} \ h1 \ h2 \ x,$
 $\text{indom } h1 \ x \rightarrow$
 $\text{read } (\text{union } h1 \ h2) \ x = \text{read } h1 \ x.$
Proof using.
 $\text{intros. unfold read, union, map\_union. simpl.}$
 $\text{case\_eq (fmap\_data } h1 \ x); \text{auto\_false.}$
Qed.

Lemma read_union_r : $\forall \{IB:\text{Inhab } B\} \ h1 \ h2 \ x,$
 $\neg \text{indom } h1 \ x \rightarrow$
 $\text{read } (\text{union } h1 \ h2) \ x = \text{read } h2 \ x.$
Proof using.
 $\text{intros. unfold read, union, map\_union. simpl.}$
 $\text{case\_eq (fmap\_data } h1 \ x).$
 $\{ \text{intros } v \ Hv. \text{false } H. \text{auto\_false.} \}$
 $\{ \text{auto\_false.} \}$
Qed.

31.3.8 Update

Lemma update_eq_union_single : $\forall \ h \ x \ v,$
 $\text{update } h \ x \ v = \text{union } (\text{single } x \ v) \ h.$
Proof using. auto. Qed.

Lemma update_empty : $\forall \ x \ v,$
 $\text{update empty } x \ v = \text{single } x \ v.$
Proof using.
 $\text{intros. rewrite update\_eq\_union\_single. rewrite } \times \text{ union\_empty\_r.}$
Qed.

Lemma update_single : $\forall \ x \ v \ w,$
 $\text{update } (\text{single } x \ v) \ x \ w = \text{single } x \ w.$
Proof using.
 $\text{intros. rewrite update\_eq\_union\_single.}$
 $\text{applys make\_eq. intros } y.$
 $\text{unfold map\_union, single. simpl. case\_if } \neg.$
Qed.

Lemma update_union_l : $\forall \ h1 \ h2 \ x \ v,$
 $\text{indom } h1 \ x \rightarrow$
 $\text{update } (\text{union } h1 \ h2) \ x \ v = \text{union } (\text{update } h1 \ x \ v) \ h2.$
Proof using.
 $\text{intros. do 2 rewrite update\_eq\_union\_single.}$
 $\text{applys make\_eq. intros } y.$
 $\text{unfold map\_union, union, map\_union. simpl. case\_if } \neg.$

Qed.

Lemma update_union_r : $\forall h1\ h2\ x\ v,$
 $\neg \text{indom } h1\ x \rightarrow$
 $\text{update } (\text{union } h1\ h2)\ x\ v = \text{union } h1\ (\text{update } h2\ x\ v).$

Proof using.

```
introv M. asserts IB: (Inhab B). { applies Inhab_of_val v. }  
do 2 rewrite update_eq_union_single.  
applies make_eq. intros y.  
unfold map_union, union, map_union. simpl. case-if  $\neg$ .  
{ subst. case_eq (fmap_data h1 y); auto_false.  
  { intros w Hw. false M. auto_false. } }
```

Qed.

Lemma update_union_not_r : $\forall h1\ h2\ x\ v,$
 $\neg \text{indom } h2\ x \rightarrow$
 $\text{update } (\text{union } h1\ h2)\ x\ v = \text{union } (\text{update } h1\ x\ v)\ h2.$

Proof using.

```
intros. do 2 rewrite update_eq_union_single.  
applies make_eq. intros y.  
unfold map_union, union, map_union. simpl. case-if  $\neg$ .
```

Qed.

Lemma update_union_not_l : $\forall h1\ h2\ x\ v,$
 $\neg \text{indom } h1\ x \rightarrow$
 $\text{update } (\text{union } h1\ h2)\ x\ v = \text{union } h1\ (\text{update } h2\ x\ v).$

Proof using.

```
introv M. asserts IB: (Inhab B). { applies Inhab_of_val v. }  
do 2 rewrite update_eq_union_single.  
applies make_eq. intros y.  
unfold map_union, union, map_union. simpl. case-if  $\neg$ .  
{ subst. case_eq (fmap_data h1 y); auto_false.  
  { intros w Hw. false M. auto_false. } }
```

Qed.

31.3.9 Removal

Lemma remove_single : $\forall x\ v,$
 $\text{remove } (\text{single } x\ v)\ x = \text{empty}.$

Proof using.

```
intros. applies fmap_extens. intros y.  
unfold remove, map_remove, single, empty. simpl.  
case-if  $\times$ . case-if  $\times$ .
```

Qed.

```

Lemma remove_disjoint_union_l :  $\forall$   $h1$   $h2$   $x$ ,
  indom  $h1$   $x$   $\rightarrow$ 
  disjoint  $h1$   $h2$   $\rightarrow$ 
  remove (union  $h1$   $h2$ )  $x$  = union (remove  $h1$   $x$ )  $h2$ .

```

Proof using.

```

intros  $M$   $D$ . applys fmap_extens. intros  $y$ .
rewrite disjoint_eq in  $D$ . specializes  $D$   $M$ . asserts  $\times D'$ : ( $\neg$  indom  $h2$   $x$ ).
unfold remove, map_remove, union, map_union, single. simpls. case_if.
{ destruct  $h1$  as [ $f1$   $F1$ ]. unfolds indom, map_indom. simpls. subst.
  rew_logic  $\neg$  in  $D'$ . }
{ auto. }

```

Qed.

```

Lemma remove_union_single_l :  $\forall$   $h$   $x$   $v$ ,
   $\neg$  indom  $h$   $x$   $\rightarrow$ 
  remove (union (single  $x$   $v$ )  $h$ )  $x$  =  $h$ .

```

Proof using.

```

intros  $M$ . applys fmap_extens. intros  $y$ .
unfold remove, map_remove, union, map_union, single. simpls. case_if.
{ destruct  $h$  as [ $f$   $F$ ]. unfolds indom, map_indom. simpls. subst. rew_logic  $\neg$  in  $M$ . }
{ case_if  $\neg$ . }

```

Qed.

End FmapProp.

Fixing arguments

```

Arguments union_assoc [ $A$ ] [ $B$ ].
Arguments union_comm_of_disjoint [ $A$ ] [ $B$ ].
Arguments union_comm_of_agree [ $A$ ] [ $B$ ].

```

31.4 Tactics for Finite Maps

31.4.1 Tactic **disjoint** for proving disjointness results

disjoint proves goals of the form $\backslash\#$ $h1$ $h2$ and $\backslash\#$ $h1$ $h2$ $h3$ by expanding all hypotheses into binary forms $\backslash\#$ $h1$ $h2$ and then exploiting symmetry and disjointness with **empty**.

Module Export DISJOINTHINTS.

#[export]

Hint Resolve *disjoint_sym disjoint_empty_l disjoint_empty_r*.

End DISJOINTHINTS.

#[global]

Hint Rewrite

disjoint_union_eq_l

```

disjoint_union_eq_r
disjoint_3_unfold : rew_disjoint.

Tactic Notation "rew_disjoint" :=
  autorewrite with rew_disjoint in *.
Tactic Notation "rew_disjoint" "*" :=
  rew_disjoint; auto_star.

Ltac fmap_disjoint_pre tt :=
  subst; rew_disjoint; jauto_set.

Tactic Notation "fmap_disjoint" :=
  solve [ fmap_disjoint_pre tt; auto ].

Lemma disjoint_demo :  $\forall A B (h1\ h2\ h3\ h4\ h5 : \text{fmap } A\ B),$ 
   $h1 = h2 \setminus + h3 \rightarrow$ 
   $\setminus \# h2\ h3 \rightarrow$ 
   $\setminus \# h1\ h4\ h5 \rightarrow$ 
   $\setminus \# h3\ h2\ h5 \wedge \setminus \# h4\ h5$ .

Proof using.
  intros. dup 2.
  { subst. rew_disjoint. jauto_set. auto. auto. auto. auto. }
  { fmap_disjoint. }

Qed.

```

31.4.2 Tactic *rew_map* for Normalizing Expressions

```

#[global]
Hint Rewrite
  union_assoc
  union_empty_l
  union_empty_r : rew_fmap rew_fmap_for_fmap_eq.

Tactic Notation "rew_fmap" :=
  autorewrite with rew_fmap in *.

Tactic Notation "rew_fmap" "~" :=
  rew_fmap; auto_tilde.

Tactic Notation "rew_fmap" "*" :=
  rew_fmap; auto_star.

```

31.4.3 Tactic *fmap_eq* for Proving Equalities

```

Section StateEq.
Variables (A B : Type).
Implicit Types h : fmap A B.

```

eq proves equalities between unions of fmaps, of the form $h1 \setminus + h2 \setminus + h3 \setminus + h4 = h1' \setminus + h2' \setminus + h3' \setminus + h4'$. It attempts to discharge the disjointness side-conditions. Disclaimer: it cancels heaps at depth up to 4, but no more.

```
Lemma union_eq_cancel_1 : ∀ h1 h2 h2',
  h2 = h2' →
  h1 \+ h2 = h1 \+ h2'.
```

Proof using. intros. subst. auto. Qed.

```
Lemma union_eq_cancel_1' : ∀ h1,
  h1 = h1.
```

Proof using. intros. auto. Qed.

```
Lemma union_eq_cancel_2 : ∀ h1 h1' h2 h2',
  \# h1 h1' →
  h2 = h1' \+ h2' →
  h1 \+ h2 = h1' \+ h1 \+ h2'.
```

Proof using.

```
  intros. subst. rewrite ← union_assoc.
  rewrite (union_comm_of_disjoint h1 h1').
  rewrite ← union_assoc. auto.
```

Qed.

```
Lemma union_eq_cancel_2' : ∀ h1 h1' h2,
  \# h1 h1' →
  h2 = h1' →
  h1 \+ h2 = h1' \+ h1.
```

Proof using.

```
  intros. subst. apply ← union_comm_of_disjoint.
```

Qed.

```
Lemma union_eq_cancel_3 : ∀ h1 h1' h2 h2' h3',
  \# h1 (h1' \+ h2') →
  h2 = h1' \+ h2' \+ h3' →
  h1 \+ h2 = h1' \+ h2' \+ h1 \+ h3'.
```

Proof using.

```
  intros. subst.
  rewrite ← (union_assoc h1' h2' h3').
  rewrite ← (union_assoc h1' h2' (h1 \+ h3')).
  apply ← union_eq_cancel_2.
```

Qed.

```
Lemma union_eq_cancel_3' : ∀ h1 h1' h2 h2',
  \# h1 (h1' \+ h2') →
  h2 = h1' \+ h2' →
  h1 \+ h2 = h1' \+ h2' \+ h1.
```

Proof using.


```

intros. subst.
rewrite ← (union_assoc h1' h2' h1).
apply¬ union_eq_cancel_2'.
Qed.

Lemma union_eq_cancel_4 : ∀ h1 h1' h2 h2' h3' h4',
  \# h1 ((h1' \+ h2') \+ h3') →
  h2 = h1' \+ h2' \+ h3' \+ h4' →
  h1 \+ h2 = h1' \+ h2' \+ h3' \+ h1 \+ h4'.
Proof using.
  intros. subst.
  rewrite ← (union_assoc h1' h2' (h3' \+ h4')).
  rewrite ← (union_assoc h1' h2' (h3' \+ h1 \+ h4')).
  apply¬ union_eq_cancel_3.
Qed.

Lemma union_eq_cancel_4' : ∀ h1 h1' h2 h2' h3',
  \# h1 (h1' \+ h2' \+ h3') →
  h2 = h1' \+ h2' \+ h3' →
  h1 \+ h2 = h1' \+ h2' \+ h3' \+ h1.
Proof using.
  intros. subst.
  rewrite ← (union_assoc h2' h3' h1).
  apply¬ union_eq_cancel_3'.
Qed.

End StateEq.

Ltac fmap_eq_step tt :=
  match goal with | ⊢ ?G ⇒ match G with
  | ?h1 = ?h1 ⇒ apply union_eq_cancel_1'
  | ?h1 \+ ?h2 = ?h1 \+ ?h2' ⇒ apply union_eq_cancel_1
  | ?h1 \+ ?h2 = ?h1' \+ ?h1 ⇒ apply union_eq_cancel_2'
  | ?h1 \+ ?h2 = ?h1' \+ ?h1 \+ ?h2' ⇒ apply union_eq_cancel_2
  | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h1 ⇒ apply union_eq_cancel_3'
  | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h1 \+ ?h3' ⇒ apply union_eq_cancel_3
  | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h3' \+ ?h1 ⇒ apply union_eq_cancel_4'
  | ?h1 \+ ?h2 = ?h1' \+ ?h2' \+ ?h3' \+ ?h1 \+ ?h4' ⇒ apply union_eq_cancel_4
  end end.

Tactic Notation "fmap_eq" :=
  subst;
  autorewrite with rew_fmap_for_fmap_eq;
  repeat (fmap_eq_step tt);
  try match goal with
  | ⊢ \# _ _ ⇒ fmap_disjoint

```

```

| ⊢ \# _ _ _ ⇒ fmap_disjoint
end.

Tactic Notation "fmap_eq" "~" :=
  fmap_eq; auto_tilde.

Tactic Notation "fmap_eq" "*" :=
  fmap_eq; auto_star.

Lemma fmap_eq_demo : ∀ A B (h1 h2 h3 h4 h5 : fmap A B),
  \# h1 h2 h3 →
  \# (h1 \+ h2 \+ h3) h4 h5 →
  h1 = h2 \+ h3 →
  h4 \+ h1 \+ h5 = h2 \+ h5 \+ h4 \+ h3.
Proof using.
  intros. dup 2.
  { subst. rew_fmap.
    fmap_eq_step tt. fmap_disjoint.
    fmap_eq_step tt.
    fmap_eq_step tt. fmap_disjoint. auto. }
  { fmap_eq. }
Qed.

```

31.5 Existence of Fresh Locations

31.5.1 Consecutive Locations

The notion of “consecutive locations” is useful for reasoning about records and arrays.

```

Fixpoint conseq (B : Type) (vs : list B) (l : nat) : fmap nat B :=
  match vs with
  | nil ⇒ empty
  | v :: vs' ⇒ (single l v) \+ (conseq vs' (S l))
  end.

```

```

Lemma conseq_nil : ∀ B (l : nat),
  conseq (@nil B) l = empty.

```

Proof using. auto. Qed.

```

Lemma conseq_cons : ∀ B (l : nat) (v : B) (vs : list B),
  conseq (v :: vs) l = (single l v) \+ (conseq vs (S l)).

```

Proof using. auto. Qed.

```

Lemma conseq_cons' : ∀ B (l : nat) (v : B) (vs : list B),
  conseq (v :: vs) l = (single l v) \+ (conseq vs (l+1)).

```

Proof using. intros. math_rewrite (l+1 = S l)%nat. applys conseq_cons. Qed.

Global Opaque conseq.

```
#[global]
Hint Rewrite conseq_nil conseq_cons : rew_listx.
```

31.5.2 Existence of Fresh Locations

```
Definition loc_fresh_gen (L : list nat) :=
  (1 + fold_right plus 0 L)%nat.
```

```
Lemma loc_fresh_ind : ∀ l L,
  mem l L →
  (l < loc_fresh_gen L)%nat.
```

```
Proof using.
  intros l L. unfold loc_fresh_gen.
  induction L; introv M; inverts M; rew_listx.
  { math. }
  { forwards~: IHL. math. }
```

Qed.

```
Lemma loc_fresh_nat_ge : ∀ (L:list nat),
  ∃ (l:nat), ∀ (i:nat), ¬ mem (l+i)%nat L.
```

```
Proof using.
  intros L. ∃ (loc_fresh_gen L). intros i M.
  lets: loc_fresh_ind M. math.
```

Qed.

For any finite list of locations (implemented as **list nat**), there exists one location not in that list.

```
Lemma loc_fresh_nat : ∀ (L:list nat),
  ∃ (l:nat), ¬ mem l L.
```

```
Proof using.
  intros L. forwards (l&P): loc_fresh_nat_ge L.
  ∃ l. intros M. applies (P 0%nat). applies_eq M. math.
```

Qed.

Section FmapFresh.

Variables (*B* : Type).

Implicit Types *h* : **fmap nat B**.

For any heap, there exists one (non-null) location not already in the domain of that heap.

```
Lemma exists_fresh : ∀ h,
  ∃ l, ¬ indom h l ∧ l ≠ null.
```

```
Proof using.
  intros null (m&(L&M)). unfold indom, map_indom. simpl.
  lets (l&F): (loc_fresh_nat (null::L)).
```

```

  ∃ l. split.
  { simpl. intros l'. forwards ⊢ E: M l. }
  { intro_subst. applies ⊢ F. }
Qed.

```

For any heap h , there exists one (non-null) location l such that the singleton map that binds l to a value v is disjoint from h .

```

Lemma single_fresh : ∀ null h v,
  ∃ l, \# (single l v) h ∧ l ≠ null.

```

Proof using.

```

  intros. forwards (l&F&N): exists_fresh null h.
  ∃ l. split ⊢. applies × disjoint_single_of_not_indom.
Qed.

```

```

Lemma conseq_fresh : ∀ null h vs,
  ∃ l, \# (conseq vs l) h ∧ l ≠ null.

```

Proof using.

```

  intros null (m&(L&M)) vs.
  unfold disjoint, map_disjoint. simpl.
  lets (l&F): (loc_fresh_nat_ge (null::L)).
  ∃ l. split.
  { intros l'. gen l. induction vs as [|v vs']; intros.
    { simple ⊢. }
    { rewrite conseq_cons. destruct (IHvs' (S l)%nat) as [E|?].
      { intros i N. applies F (S i). applies_eq N. math. }
      { simpl. unfold map_union. case_if ⊢.
        { subst. right. applies not_not_inv. intros H. applies F 0%nat.
          constructor. math_rewrite (l'+0 = l')%nat. applies ⊢ M. } }
      { auto. } } }
  { intro_subst. applies ⊢ F 0%nat. rew_nat. auto. }
Qed.

```

```

Lemma disjoint_single_conseq : ∀ B l l' L (v:B),
  (l < l')%nat ∨ (l ≥ l'+length L)%nat →
  \# (single l v) (conseq L l').

```

Proof using.

```

  introv N. gen l'. induction L as [|L']; intros.
  { rewrite ⊢ conseq_nil. }
  { rew_list in N. rewrite conseq_cons. rew_disjoint. split.
    { applies disjoint_single_single. destruct N; math. }
    { applies IHL. destruct N. { left. math. } { right. math. } } }
Qed.

```

31.5.3 Existence of a Smallest Fresh Locations

The notion of smallest fresh location is useful for setting up deterministic semantics.

Definition `fresh` (*null:nat*) (*h:fmap nat B*) (*l:nat*) : `Prop` :=
 $\neg \text{indom } h \ l \wedge l \neq \text{null}.$

Definition `smallest_fresh` (*null:nat*) (*h:fmap nat B*) (*l:nat*) : `Prop` :=
`fresh null h l` $\wedge (\forall l', l' < l \rightarrow \neg \text{fresh } null \ h \ l').$

Lemma `exists_smallest_fresh` : $\forall \text{null } h,$
 $\exists l, \text{smallest\_fresh } null \ h \ l.$

Proof using.

```

intros.
cuts M: (∀ l0, fresh null h l0 →
        ∃ l, fresh null h l
        ∧ (∀ l', l' < l → ¬ fresh null h l')).
{ lets (l0&F&N): exists_fresh null h. applies M l0. split×. }
intros l0. induction_wf IH: wf_lt l0. intros F.
tests C: (∀ l', l' < l0 → ¬ fresh null h l').
{ ∃× l0. }
{ destruct C as (l0'&K). rew_logic in K. destruct K as (L&F').
  applies× IH l0'. }

```

Qed.

31.5.4 Existence of Nonempty Heaps

The existence of nonempty (nat-indexed) finite maps is useful for constructing counterexamples.

Lemma `exists_not_empty` : $\forall \{IB:\text{Inhab } B\},$
 $\exists (h:\text{fmap nat } B), h \neq \text{empty}.$

Proof using.

```

intros. sets l: 0%nat. sets h: (single l (arbitrary (A:=B))).
∃ h. intros N.
sets h': (empty:fmap nat B).
asserts M1: (indom h l).
{ applies indom_single. }
asserts M2: (¬ indom h' l).
{ unfolds @indom, map_indom, @empty. simpls. auto_false. }
rewrite N in M1. false×.

```

Qed.

End `FmapFresh`.

Chapter 32

Library SLF.LibSepVar

32.1 LibSepVar: Appendix - Program Variables

```
Set Implicit Arguments.
From SLF Require LibListExec.
From SLF Require Export LibString LibList LibCore.
From SLF Require Import LibSepFmap LibSepTLCbuffer.
Open Scope string_scope.
Generalizable Variables A.
```

32.2 Representation of Program Variables

This file contains definitions, lemmas, tactics and notations for manipulating program variables and list of program variables.

32.2.1 Representation of Variables

Variables are represented as strings

```
Definition var : Type := string.
```

The boolean function `var_eq s1 s2` compares two variables.

```
Definition var_eq (s1:var) (s2:var) : bool :=
  if String.string_dec s1 s2 then true else false.
```

The boolean function `var_eq s1 s2` returns `true` iff the equality `v1 = v2` holds.

```
Lemma var_eq_spec : ∀ s1 s2,
  var_eq s1 s2 = isTrue (s1 = s2).
```

Proof using.

```
  intros. unfold var_eq. case_if; rew_bool_eq¬.
Qed.
```

Global Opaque var.

32.2.2 Tactic *case_var*

The tactic *case_var* performs case analysis on expressions of the form `if var_eq x y then .. else ..` that appear in the goal.

```
Tactic Notation "case_var" :=  
  repeat rewrite var_eq_spec in *; repeat case_if.
```

```
Tactic Notation "case_var" "~" :=  
  case_var; auto_tilde.
```

```
Tactic Notation "case_var" "*" :=  
  case_var; auto_star.
```

32.3 Representation of List of Variables

32.3.1 Definition of Distinct Variables

`vars` is the type of a list of variables

Definition `vars : Type := list var`.

`var_fresh y xs` asserts that `y` does not belong to the list `xs`

Definition `var_fresh (y:var) (xs:vars) : Prop :=
 ¬ mem y xs`.

The following lemma asserts that if `x` is a variable in the list `xs`, and `y` is fresh from this list `xs`, then `y` is not equal to `x`.

Lemma `var_fresh_mem_inv : ∀ y x xs,
 var_fresh x xs →
 mem y xs →
 x ≠ y`.

Proof using. `intros H M N. unfolds var_fresh. subst×. Qed`.

32.3.2 Generation of `n` Distinct Variables

`nat_to_var n` converts `nat` values into distinct *name* values.

Definition `dummy_char :=
 Ascii.ascii_of_nat 0%nat`.

Fixpoint `nat_to_var (n:nat) : var :=
 match n with
 | 0 ⇒ String.EmptyString
 | S n' ⇒ String.String dummy_char (nat_to_var n')`

end.

Lemma injective_nat_to_var :
 injective nat_to_var.

Proof using.

```

  intros  $n$ . induction  $n$  as [| $n'$ ]; intros  $m$   $E$ ; destruct  $m$  as [| $m'$ ]; tryfalse.
  { auto. }
  { inverts  $E$ . fequals $\neg$ . }

```

Qed.

`var_seq i n` generates a list of variables $x_1; x_2; \dots; x_n$ with $x_1 = i$ and $x_n = i + n - 1$. The ability to start at a given offset is sometimes useful.

```

Fixpoint var_seq (start:nat) (nb:nat) : vars :=
  match nb with
  | 0  $\Rightarrow$  nil
  | S  $nb'$   $\Rightarrow$  (nat_to_var start) :: var_seq (S start)  $nb'$ 
  end.

```

The properties of `var_seq` are stated next. They assert that this function produce the expected number of variables, that the variables are pairwise distinct

Section Var_seq.

Implicit Types start nb : nat.

Lemma var_fresh_var_seq_lt : $\forall (x:\text{nat})$ start nb,
 ($x < \text{start}$)%nat $\rightarrow$
 var_fresh (nat_to_var x) (var_seq start nb).

Proof using.

```

  intros. unfold var_fresh. gen start. induction nb; simpl; introv N; rew_listx.
  { auto. }
  { simpl. case_var. rew_logic. split.
    { intros  $E$ . lets: injective_nat_to_var  $E$ . math. }
    { applies IHnb. math. } }

```

Qed.

Lemma var_fresh_var_seq_ge : $\forall (x:\text{nat})$ start nb,
 ($x \geq \text{start} + \text{nb}$)%nat $\rightarrow$
 var_fresh (nat_to_var x) (var_seq start nb).

Proof using.

```

  intros. unfold var_fresh. gen start. induction nb; simpl; introv N; rew_listx.
  { auto. }
  { simpl. case_var. rew_logic. split.
    { intros  $E$ . lets: injective_nat_to_var  $E$ . math. }
    { applies IHnb. math. } }

```

Qed.

Lemma noduplicates_var_seq : $\forall$ start nb,


```
LibList.noduplicates (var_seq start nb).
```

Proof using.

```
intros. gen start. induction nb; intros; simpl; rew_listx.
{ auto. }
{ split.
  { applys var_fresh_var_seq_lt. math. }
  { auto. } }
```

Qed.

Lemma length_var_seq : $\forall$ start nb,
length (var_seq start nb) = nb.

Proof using.

```
intros. gen start. induction nb; simpl; intros.
{ auto. } { rew_list. rewrite  $\neg$  IHnb. }
```

Qed.

End Var_seq.

32.4 Notation for Program Variables

When writing deeply-embedded programs, it is nice to write identifiers (program variables) without quotes. In the future, the “Custom Entry” mechanism might allow for this in a generic manner. In the meantime, we need one definition or one notation for each identifier

32.4.1 Program Variables Expressed Using Definitions

These definitions allowing to use the string notation "x" without any double quote or quotes. However, the scope of these definitions should be wrapped in a module to avoid clashes with local variables. E.g. `Module Export Foo. Import DEFINITIONSFORVARIABLES. (...) End Foo.`

Module DEFINITIONSFORVARIABLES.

```
Definition a := ("a":var).
Definition b := ("b":var).
Definition c := ("c":var).
Definition d := ("d":var).
Definition e := ("e":var).
Definition f := ("f":var).
Definition g := ("g":var).
Definition h := ("h":var).
Definition i := ("i":var).
Definition j := ("j":var).
Definition k := ("k":var).
Definition l := ("l":var).
```

```

Definition m := ("m":var).
Definition n := ("n":var).
Definition o := ("o":var).
Definition p := ("p":var).
Definition q := ("q":var).
Definition r := ("r":var).
Definition s := ("s":var).
Definition t := ("t":var).
Definition u := ("u":var).
Definition v := ("v":var).
Definition w := ("w":var).
Definition x := ("x":var).
Definition y := ("y":var).
Definition z := ("z":var).

Definition a1 := ("a1":var).
Definition b1 := ("b1":var).
Definition c1 := ("c1":var).
Definition d1 := ("d1":var).
Definition e1 := ("e1":var).
Definition f1 := ("f1":var).
Definition g1 := ("g1":var).
Definition h1 := ("h1":var).
Definition i1 := ("i1":var).
Definition j1 := ("j1":var).
Definition k1 := ("k1":var).
Definition l1 := ("l1":var).
Definition m1 := ("m1":var).
Definition n1 := ("n1":var).
Definition o1 := ("o1":var).
Definition p1 := ("p1":var).
Definition q1 := ("q1":var).
Definition r1 := ("r1":var).
Definition s1 := ("s1":var).
Definition t1 := ("t1":var).
Definition u1 := ("u1":var).
Definition v1 := ("v1":var).
Definition w1 := ("w1":var).
Definition x1 := ("x1":var).
Definition y1 := ("y1":var).
Definition z1 := ("z1":var).

Definition a2 := ("a2":var).
Definition b2 := ("b2":var).

```

```

Definition c2 := ("c2":var).
Definition d2 := ("d2":var).
Definition e2 := ("e2":var).
Definition f2 := ("f2":var).
Definition g2 := ("g2":var).
Definition h2 := ("h2":var).
Definition i2 := ("i2":var).
Definition j2 := ("j2":var).
Definition k2 := ("k2":var).
Definition l2 := ("l2":var).
Definition m2 := ("m2":var).
Definition n2 := ("n2":var).
Definition o2 := ("o2":var).
Definition p2 := ("p2":var).
Definition q2 := ("q2":var).
Definition r2 := ("r2":var).
Definition s2 := ("s2":var).
Definition t2 := ("t2":var).
Definition u2 := ("u2":var).
Definition v2 := ("v2":var).
Definition w2 := ("w2":var).
Definition x2 := ("x2":var).
Definition y2 := ("y2":var).
Definition z2 := ("z2":var).

Definition a3 := ("a3":var).
Definition b3 := ("b3":var).
Definition c3 := ("c3":var).
Definition d3 := ("d3":var).
Definition e3 := ("e3":var).
Definition f3 := ("f3":var).
Definition g3 := ("g3":var).
Definition h3 := ("h3":var).
Definition i3 := ("i3":var).
Definition j3 := ("j3":var).
Definition k3 := ("k3":var).
Definition l3 := ("l3":var).
Definition m3 := ("m3":var).
Definition n3 := ("n3":var).
Definition o3 := ("o3":var).
Definition p3 := ("p3":var).
Definition q3 := ("q3":var).
Definition r3 := ("r3":var).

```

```

Definition s3 := ("s3":var).
Definition t3 := ("t3":var).
Definition u3 := ("u3":var).
Definition v3 := ("v3":var).
Definition w3 := ("w3":var).
Definition x3 := ("x3":var).
Definition y3 := ("y3":var).
Definition z3 := ("z3":var).

```

Tactic to unfold these definitions. Useful to avoid the definitions to get in the way of `simpl`.

```

Ltac libsepvar_unfold :=
  unfold
  a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,
  a1, b1, c1, d1, e1, f1, g1, h1, i1, j1, k1, l1, m1, n1, o1, p1, q1, r1, s1, t1, u1, v1, w1, x1,
  y1, z1,
  a2, b2, c2, d2, e2, f2, g2, h2, i2, j2, k2, l2, m2, n2, o2, p2, q2, r2, s2, t2, u2, v2, w2, x2,
  y2, z2,
  a3, b3, c3, d3, e3, f3, g3, h3, i3, j3, k3, l3, m3, n3, o3, p3, q3, r3, s3, t3, u3, v3, w3, x3,
  y3, z3.
End DEFINITIONSFORVARIABLES.

```

32.4.2 Program Variables Expressed Using Notation, with a Quote Symbol

To avoid using the string notation `"x"` for referring to a variable called `x`, one can use the notation `'x`, available by importing the following module.

Declare Custom Entry `trm`.

Module NOTATIONFORVARIABLES.

Declare Scope *trm_scope*.

```

Notation "'a'" := ("a":var) (in custom trm at level 0) : trm_scope.
Notation "'b'" := ("b":var) (in custom trm at level 0) : trm_scope.
Notation "'c'" := ("c":var) (in custom trm at level 0) : trm_scope.
Notation "'d'" := ("d":var) (in custom trm at level 0) : trm_scope.
Notation "'e'" := ("e":var) (in custom trm at level 0) : trm_scope.
Notation "'f'" := ("f":var) (in custom trm at level 0) : trm_scope.
Notation "'g'" := ("g":var) (in custom trm at level 0) : trm_scope.
Notation "'h'" := ("h":var) (in custom trm at level 0) : trm_scope.
Notation "'i'" := ("i":var) (in custom trm at level 0) : trm_scope.
Notation "'j'" := ("j":var) (in custom trm at level 0) : trm_scope.
Notation "'k'" := ("k":var) (in custom trm at level 0) : trm_scope.

```

[illegible]

[illegible]

```

Notation "'r3'" := ("r3":var) (in custom trm at level 0) : trm_scope.
Notation "'s3'" := ("s3":var) (in custom trm at level 0) : trm_scope.
Notation "'t3'" := ("t3":var) (in custom trm at level 0) : trm_scope.
Notation "'u3'" := ("u3":var) (in custom trm at level 0) : trm_scope.
Notation "'v3'" := ("v3":var) (in custom trm at level 0) : trm_scope.
Notation "'w3'" := ("w3":var) (in custom trm at level 0) : trm_scope.
Notation "'x3'" := ("x3":var) (in custom trm at level 0) : trm_scope.
Notation "'y3'" := ("y3":var) (in custom trm at level 0) : trm_scope.
Notation "'z3'" := ("z3":var) (in custom trm at level 0) : trm_scope.
End NOTATIONFORVARIABLES.

```

32.5 Optional Material

32.5.1 Bonuse: the Tactic *var_neq*

This tactic is not used in the SLF volume.

The tactic *var_neq* helps prove calls of the form $x \neq y$, where x and y are two concrete program variables. This tactic is registered as hint for *auto*

```

Ltac var_neq :=
  match goal with  $\vdash ?x \neq ?y :>$  var  $\Rightarrow$ 
    solve [ let  $E$  := fresh in
      destruct (String.string_dec  $x$   $y$ ) as [ $E|E$ ];
      [ false | apply  $E$  ] ] end.

#[global]
Hint Extern 1 ( $?x \neq ?y$ )  $\Rightarrow$  var_neq.

```

Chapter 33

Library `SLF.LibSepSimpl`

33.1 LibSepSimpl: Appendix - Simplification Tactic for Entailments

```
Set Implicit Arguments.  
From SLF Require Export LibCore.  
From SLF Require Export LibSepTLCbuffer.
```

33.2 A Functor for Separation Logic Entailment

This file consists of a functor that provides a tactic for simplifying entailment relations. This tactic is somewhat generic in that it can be used for several variants of Separation Logic, hence the use of a functor to implement the tactic with respect to abstract definitions of heap predicates.

The file provides a number of lemmas that hold in any variant of Separation Logic satisfying the requirements of the functor. It also provides the following key tactics:

- *xsimpl* simplifies heap entailments.
- *xpull* is a restricted version of *xsimpl* that only act over the left-hand side of the entailment, leaving the right-hand side unmodified.
- *xchange* performs transitivity steps in entailments, it enables replacing a subset of the heap predicates on the right-hand side with another set of heap predicates entailed by the former.

Bonus: the tactic *rew_heap* normalizes heap predicate expressions; it is not used in the course.

33.3 Assumptions of the functor

Module Type `XSIMPLPARAMS`.

33.3.1 Operators

The notion of heap predicate and entailment must be provided.

Parameter `hprop` : Type.

Parameter `himpl` : `hprop` → `hprop` → Prop.

Definition `qimpl A (Q1 Q2:A→hprop) :=`
`∀ r, himpl (Q1 r) (Q2 r).`

The core operators of Separation Logic must be provided.

Parameter `hempty` : `hprop`.

Parameter `hstar` : `hprop` → `hprop` → `hprop`.

Parameter `hpure` : Prop → `hprop`.

Parameter `htop` : `hprop`.

Parameter `hgc` : `hprop`.

Parameter `hwand` : `hprop` → `hprop` → `hprop`.

Parameter `qwand` : ∀ `A`, (`A`→`hprop`) → (`A`→`hprop`) → `hprop`.

Parameter `hexists` : ∀ `A`, (`A`→`hprop`) → `hprop`.

Parameter `hforall` : ∀ `A`, (`A`→`hprop`) → `hprop`.

The predicate `haffine` must be provided. For a fully linear logic, use the always-false predicate. For a fully affine logic, use the always-true predicate.

Parameter `haffine` : `hprop` → Prop.

33.3.2 Notation

The following notation is used for stating the required properties.

Declare Scope `heap_scope`.

Notation "`H1 ==> H2`" := (`himpl H1 H2`)
(at level 55) : `heap_scope`.

Notation "`Q1 ===> Q2`" := (`qimpl Q1 Q2`)
(at level 55) : `heap_scope`.

Notation "`\|`" := (`hempty`)
(at level 0) : `heap_scope`.

Notation "`\[ P ]`" := (`hpure P`)

$(\text{at level } 0, \text{format } "[P]") : \text{heap\_scope}.$
 Notation " $\backslash \text{Top}$ " := $(\text{htop}) : \text{heap\_scope}.$
 Notation " $\backslash \text{GC}$ " := $(\text{hgc}) : \text{heap\_scope}.$
 Notation " $H1 \backslash^* H2$ " := $(\text{hstar } H1 \ H2)$
 $(\text{at level } 41, \text{right associativity}) : \text{heap\_scope}.$
 Notation " $Q \backslash^+ H$ " := $(\text{fun } x \Rightarrow \text{hstar } (Q \ x) \ H)$
 $(\text{at level } 40) : \text{heap\_scope}.$
 Notation " $H1 \backslash^* H2$ " := $(\text{hwand } H1 \ H2)$
 $(\text{at level } 43, \text{right associativity}) : \text{heap\_scope}.$
 Notation " $Q1 \backslash^* Q2$ " := $(\text{qwand } Q1 \ Q2)$
 $(\text{at level } 43) : \text{heap\_scope}.$
 Notation " $\backslash \text{exists}' \ x1 \ .. \ xn \ , \ H$ " :=
 $(\text{hexists } (\text{fun } x1 \Rightarrow .. (\text{hexists } (\text{fun } xn \Rightarrow H)) \ ..))$
 $(\text{at level } 39, \text{ } x1 \text{ binder, } H \text{ at level } 50, \text{right associativity,}$
 $\text{format } "[\backslash \text{exists}' \ / \ ' \ x1 \ .. \ xn \ , \ / \ ' \ H \ ']") : \text{heap\_scope}.$
 Notation " $\backslash \text{forall}' \ x1 \ .. \ xn \ , \ H$ " :=
 $(\text{hforall } (\text{fun } x1 \Rightarrow .. (\text{hforall } (\text{fun } xn \Rightarrow H)) \ ..))$
 $(\text{at level } 39, \text{ } x1 \text{ binder, } H \text{ at level } 50, \text{right associativity,}$
 $\text{format } "[\backslash \text{forall}' \ / \ ' \ x1 \ .. \ xn \ , \ / \ ' \ H \ ']") : \text{heap\_scope}.$
 Local Open Scope $\text{heap\_scope}.$

33.3.3 Properties Assumed by the Functor

The following properties must be satisfied.

Implicit Types $P : \text{Prop}.$

Implicit Types $H : \text{hprop}.$

Entailment must be an order.

Parameter $\text{himpl\_refl} : \forall H,$
 $H \implies H.$

Parameter $\text{himpl\_trans} : \forall H2 \ H1 \ H3,$
 $(H1 \implies H2) \rightarrow$
 $(H2 \implies H3) \rightarrow$
 $(H1 \implies H3).$

Parameter $\text{himpl\_antisym} : \forall H1 \ H2,$
 $(H1 \implies H2) \rightarrow$
 $(H2 \implies H1) \rightarrow$
 $(H1 = H2).$

The star and the empty heap predicate must form a commutative monoid.

Parameter *hstar_empty_l* : $\forall H,$
 $\backslash [] \backslash * H = H.$

Parameter *hstar_empty_r* : $\forall H,$
 $H \backslash * \backslash [] = H.$

Parameter *hstar_comm* : $\forall H1\ H2,$
 $H1 \backslash * H2 = H2 \backslash * H1.$

Parameter *hstar_assoc* : $\forall H1\ H2\ H3,$
 $(H1 \backslash * H2) \backslash * H3 = H1 \backslash * (H2 \backslash * H3).$

The frame property for entailment must hold.

Parameter *himpl_frame_lr* : $\forall H1\ H1'\ H2\ H2',$
 $H1 \implies H1' \rightarrow$
 $H2 \implies H2' \rightarrow$
 $(H1 \backslash * H2) \implies (H1' \backslash * H2').$

Parameter *himpl_hstar_trans_l* : $\forall H1\ H2\ H3\ H4,$
 $H1 \implies H2 \rightarrow$
 $H2 \backslash * H3 \implies H4 \rightarrow$
 $H1 \backslash * H3 \implies H4.$

Characterization of *hpure*

Parameter *himpl_empty_hpure* : $\forall P,$
 $P \rightarrow$
 $\backslash [] \implies \backslash [P].$

Parameter *himpl_hstar_hpure_l* : $\forall P\ H\ H',$
 $(P \rightarrow H \implies H') \rightarrow$
 $(\backslash [P] \backslash * H) \implies H'.$

Characterization of *hexists*

Parameter *himpl_hexists_l* : $\forall A\ H\ (J:A \rightarrow \text{hprop}),$
 $(\forall x, J\ x \implies H) \rightarrow$
 $(\text{hexists } J) \implies H.$

Parameter *himpl_hexists_r* : $\forall A\ (x:A)\ H\ J,$
 $(H \implies J\ x) \rightarrow$
 $H \implies (\text{hexists } J).$

Parameter *hstar_hexists* : $\forall A\ (J:A \rightarrow \text{hprop})\ H,$
 $(\text{hexists } J) \backslash * H = \text{hexists } (\text{fun } x \Rightarrow (J\ x) \backslash * H).$

Characterization of *hforall*

Parameter *himpl_hforall_r* : $\forall A\ (J:A \rightarrow \text{hprop})\ H,$
 $(\forall x, H \implies J\ x) \rightarrow$
 $H \implies (\text{hforall } J).$

Parameter *hstar_hforall* : $\forall H\ A\ (J:A \rightarrow \text{hprop}),$

```

    (hforall J) \* H ==> hforall (J \*+ H).

Characterization of hwand
Parameter hwand_equiv : ∀ H0 H1 H2,
  (H0 ==> H1 \-* H2) ↔ (H1 \* H0 ==> H2).
Parameter hwand_curry_eq : ∀ H1 H2 H3,
  (H1 \* H2) \-* H3 = H1 \-* (H2 \-* H3).
Parameter hwand_hempty_l : ∀ H,
  (\ [] \-* H) = H.

Characterization of qwand
Parameter qwand_equiv : ∀ H A (Q1 Q2:A→hprop),
  H ==> (Q1 \-* Q2) ↔ (Q1 \*+ H) ==> Q2.
Parameter hwand_cancel : ∀ H1 H2,
  H1 \* (H1 \-* H2) ==> H2.
Parameter qwand_specialize : ∀ A (x:A) (Q1 Q2:A→hprop),
  (Q1 \-* Q2) ==> (Q1 x \-* Q2 x).

Characterization of htop
Parameter himpl_htop_r : ∀ H,
  H ==> \Top.
Parameter hstar_htop_htop :
  \Top \* \Top = \Top.

Characterization of hgc
Parameter haffine_hempty :
  haffine \ [].
Parameter himpl_hgc_r : ∀ H,
  haffine H →
  H ==> \GC.
Parameter hstar_hgc_hgc :
  \GC \* \GC = \GC.
End XSIMPLPARAMS.

```

33.4 Body of the Functor

When all the assumptions of the functor are statisfied, all the lemmas stated below hold, and all the tactics defined below may be used.

```

Module XSIMPLSETUP (HP : XSIMPLPARAMS).
Import HP.
Local Open Scope heap_scope.

```

```

Implicit Types  $H : \text{hprop}$ .
Implicit Types  $P : \text{Prop}$ .
#[global]
Hint Resolve himpl_refl.

```

33.4.1 Properties of **himpl**

```

Lemma himpl_of_eq :  $\forall H1\ H2$ ,
   $H1 = H2 \rightarrow$ 
   $H1 \impl H2$ .
Proof. intros. subst. applies $\neg$  himpl_refl. Qed.

Lemma himpl_of_eq_sym :  $\forall H1\ H2$ ,
   $H1 = H2 \rightarrow$ 
   $H2 \impl H1$ .
Proof. intros. subst. applies $\neg$  himpl_refl. Qed.

Lemma himpl_hstar_trans_l :  $\forall H1\ H2\ H3\ H4$ ,
   $H1 \impl H2 \rightarrow$ 
   $H2 \setminus * H3 \impl H4 \rightarrow$ 
   $H1 \setminus * H3 \impl H4$ .
Proof using.
  introv M1 M2. applies himpl_trans M2. applies $\times$  himpl_frame_lr M1.
Qed.

```

33.4.2 Properties of **qimpl**

```

Lemma qimpl_refl :  $\forall A\ (Q:A \rightarrow \text{hprop})$ ,
   $Q \impl\impl Q$ .
Proof using. intros. hnfs $\times$ . Qed.
#[global] Hint Resolve qimpl_refl.

Lemma qimpl_trans :  $\forall A\ (Q2\ Q1\ Q3:A \rightarrow \text{hprop})$ ,
   $(Q1 \impl\impl Q2) \rightarrow$ 
   $(Q2 \impl\impl Q3) \rightarrow$ 
   $(Q1 \impl\impl Q3)$ .
Proof using. introv M1 M2. intros v. applies $\times$  himpl_trans. Qed.

Lemma qimpl_antisym :  $\forall A\ (Q1\ Q2:A \rightarrow \text{hprop})$ ,
   $(Q1 \impl\impl Q2) \rightarrow$ 
   $(Q2 \impl\impl Q1) \rightarrow$ 
   $(Q1 = Q2)$ .
Proof using. introv M1 M2. apply fun_ext_1. intros v. applies $\times$  himpl_antisym. Qed.

```

33.4.3 Properties of `hstar`

Lemma `hstar_comm_assoc` : $\forall H1\ H2\ H3,$
 $H1 \setminus * H2 \setminus * H3 = H2 \setminus * H1 \setminus * H3.$

Proof using.

`intros. rewrite ← hstar_assoc.`
`rewrite (@hstar_comm H1 H2). rewrite hstar_assoc.`

Qed.

33.4.4 Representation Predicates

The arrow notation typically takes the form $x \sim> R\ x$, to indicate that X is the logical value that describes the heap structure x , according to the representation predicate R . It is just a notation for the heap predicate $R\ X\ x$.

Definition `repr` ($A:Type$) ($S:A \rightarrow hprop$) ($x:A$) : $hprop$:=
 $S\ x.$

Notation " $x \sim> S$ " := (`repr S x`)
(at level 33, no associativity) : *heap_scope*.

Lemma `repr_eq` : $\forall (A:Type) (S:A \rightarrow hprop) (x:A),$
 $(x \sim> S) = (S\ x).$

Proof using. `auto. Qed.`

$x \sim> Id\ X$ holds when x is equal to X in the empty heap. `Id` is called the identity representation predicate.

Definition `Id` { $A:Type$ } ($X:A$) ($x:A$) :=
 $\setminus [X = x].$

`xrepr_clean` simplifies instances of $p \sim> (\text{fun } _ \Rightarrow \_)$ by unfolding the arrow, but only when the body does not captures `mklocally` bound variables. This tactic should normally not be used directly

Ltac `xrepr_clean_core tt` :=
`repeat match goal with ⊢ context C [?p ~> ?E] ⇒`
`match E with (fun _ ⇒ _) ⇒`
`let E' := eval cbv beta in (E p) in`
`let G' := context C [E'] in`
`let G := match goal with ⊢ ?G ⇒ G end in`
`change G with G' end end.`

Tactic Notation "`xrepr_clean`" :=
`xrepr_clean_core tt.`

Lemma `repr_id` : $\forall A (x\ X:A),$
 $(x \sim> Id\ X) = \setminus [X = x].$

Proof using. `intros. unfold Id. xrepr_clean. auto. Qed.`

33.4.5 *rew_heap* Tactic to Normalize Expressions with *hstar*

rew_heap removes empty heap predicates, and normalizes w.r.t. associativity of star.

rew_heap_assoc only normalizes w.r.t. associativity. (It should only be used internally for tactic implementation.)

Lemma *star_post_empty* : $\forall B (Q:B \rightarrow \text{hprop})$,

$Q \setminus * \setminus [] = Q$.

Proof using. *extens. intros. rewrite* $\times$ *hstar_empty_r. Qed.*

#[*global*]

Hint Rewrite *hstar_empty_l hstar_empty_r*

hstar_assoc star_post_empty hwand_empty_l : rew_heap.

Tactic Notation "rew_heap" :=

autorewrite with *rew_heap*.

Tactic Notation "rew_heap" "in" "*" :=

autorewrite with *rew_heap* in ***.

Tactic Notation "rew_heap" "in" *hyp*(*H*) :=

autorewrite with *rew_heap* in *H*.

Tactic Notation "rew_heap" "~" :=

rew_heap; auto_tilde.

Tactic Notation "rew_heap" "~" "in" "*" :=

rew_heap in ***; *auto_tilde.*

Tactic Notation "rew_heap" "~" "in" *hyp*(*H*) :=

rew_heap in *H*; *auto_tilde.*

Tactic Notation "rew_heap" "*" :=

rew_heap; auto_star.

Tactic Notation "rew_heap" "*" "in" "*" :=

rew_heap in ***; *auto_star.*

Tactic Notation "rew_heap" "*" "in" *hyp*(*H*) :=

rew_heap in *H*; *auto_star.*

#[*global*]

Hint Rewrite *hstar_assoc : rew_heap_assoc.*

Tactic Notation "rew_heap_assoc" :=

autorewrite with *rew_heap_assoc.*

33.4.6 Auxiliary Tactics Used by *xpull* and *xsimpl*

Ltac *remove_empty_heaps_from H* :=

match H with context [?*H1* $\setminus * \setminus []$] $\Rightarrow$

match is_evar_as_bool H1 with

| *false* $\Rightarrow$ *rewrite* (*@hstar_empty_r H1*)

| *true* $\Rightarrow$ *let X := fresh in*

```

      set (X := H1);
      rewrite (@hstar_hempty_r X);
      subst X
    end end.

Ltac remove_empty_heaps_haffine tt :=
  repeat match goal with ⊢ haffine ?H ⇒ remove_empty_heaps_from H end.

Ltac remove_empty_heaps_left tt :=
  repeat match goal with ⊢ ?H1 ==> _ ⇒ remove_empty_heaps_from H1 end.

Ltac remove_empty_heaps_right tt :=
  repeat match goal with ⊢ _ ==> ?H2 ⇒ remove_empty_heaps_from H2 end.

```

33.4.7 Tactics *xsimpl* and *xpull* for Heap Entailments

The implementation of the tactics is fairly involved. The high-level specification of the tactic appears in the last appendix of: http://www.chargueraud.org/research/2020/seq_seplogic/seq_seplogic.pdf.

xaffine placeholder

```

Ltac xaffine_core tt :=
  try solve [ assumption | apply haffine_hempty ].

Tactic Notation "xaffine" :=
  xaffine_core tt.

```

Hints for tactics such as *xsimpl*

```

Inductive Xsimpl_hint : list Boxer → Type :=
| xsimpl_hint : ∀ (L: list Boxer), Xsimpl_hint L.

Ltac xsimpl_hint_put L :=
  let H := fresh "Hint" in
  generalize (xsimpl_hint L); intros H.

Ltac xsimpl_hint_next cont :=
  match goal with H: Xsimpl_hint ((boxer ?x)::?L) ⊢ _ ⇒
  clear H; xsimpl_hint_put L; cont x end.

Ltac xsimpl_hint_remove tt :=
  match goal with H: Xsimpl_hint _ ⊢ _ ⇒ clear H end.

```

Lemmas *hstars_reorder...* to flip an iterated *hstar*.

hstars_flip tt applies to a goal of the form $H1 \setminus^* \dots \setminus^* Hn \setminus^* \setminus[] = ?H$ and instantiates H with $Hn \setminus^* \dots \setminus^* H1 \setminus^* \setminus[]$. If $n > 9$, the maximum arity supported, the tactic unifies H with the original LHS.

Lemma *hstars_flip_0* :

$$\setminus[] = \setminus[].$$

Proof using. *auto*. Qed.

Lemma *hstars_flip_1* : $\forall H1,$

$$H1 \setminus^* \setminus[] = H1 \setminus^* \setminus[].$$

Proof using. *auto*. Qed.

Lemma *hstars_flip_2* : $\forall H1 H2,$

$$H1 \setminus^* H2 \setminus^* \setminus[] = H2 \setminus^* H1 \setminus^* \setminus[].$$

Proof using. *intros. rew_heap. rewrite (hstar_comm H2). rew_heap*-. Qed.

Lemma *hstars_flip_3* : $\forall H1 H2 H3,$

$$H1 \setminus^* H2 \setminus^* H3 \setminus^* \setminus[] = H3 \setminus^* H2 \setminus^* H1 \setminus^* \setminus[].$$

Proof using. *intros. rewrite* $\leftarrow$ (*hstars_flip_2 H1*). *rew_heap. rewrite (hstar_comm H3). rew_heap*-. Qed.

Lemma *hstars_flip_4* : $\forall H1 H2 H3 H4,$

$$H1 \setminus^* H2 \setminus^* H3 \setminus^* H4 \setminus^* \setminus[] = H4 \setminus^* H3 \setminus^* H2 \setminus^* H1 \setminus^* \setminus[].$$

Proof using. *intros. rewrite* $\leftarrow$ (*hstars_flip_3 H1*). *rew_heap. rewrite (hstar_comm H4). rew_heap*-. Qed.

Lemma *hstars_flip_5* : $\forall H1 H2 H3 H4 H5,$

$$H1 \setminus^* H2 \setminus^* H3 \setminus^* H4 \setminus^* H5 \setminus^* \setminus[] = H5 \setminus^* H4 \setminus^* H3 \setminus^* H2 \setminus^* H1 \setminus^* \setminus[].$$

Proof using. *intros. rewrite* $\leftarrow$ (*hstars_flip_4 H1*). *rew_heap. rewrite (hstar_comm H5). rew_heap*-. Qed.

Lemma *hstars_flip_6* : $\forall H1 H2 H3 H4 H5 H6,$

$$\begin{aligned} & H1 \setminus^* H2 \setminus^* H3 \setminus^* H4 \setminus^* H5 \setminus^* H6 \setminus^* \setminus[] \\ &= H6 \setminus^* H5 \setminus^* H4 \setminus^* H3 \setminus^* H2 \setminus^* H1 \setminus^* \setminus[] \end{aligned}$$

Proof using. *intros. rewrite* $\leftarrow$ (*hstars_flip_5 H1*). *rew_heap. rewrite (hstar_comm H6). rew_heap*-. Qed.

Lemma *hstars_flip_7* : $\forall H1 H2 H3 H4 H5 H6 H7,$

$$\begin{aligned} & H1 \setminus^* H2 \setminus^* H3 \setminus^* H4 \setminus^* H5 \setminus^* H6 \setminus^* H7 \setminus^* \setminus[] \\ &= H7 \setminus^* H6 \setminus^* H5 \setminus^* H4 \setminus^* H3 \setminus^* H2 \setminus^* H1 \setminus^* \setminus[] \end{aligned}$$

Proof using. *intros. rewrite* $\leftarrow$ (*hstars_flip_6 H1*). *rew_heap. rewrite (hstar_comm H7). rew_heap*-. Qed.

Lemma *hstars_flip_8* : $\forall H1 H2 H3 H4 H5 H6 H7 H8,$

$$\begin{aligned} & H1 \setminus^* H2 \setminus^* H3 \setminus^* H4 \setminus^* H5 \setminus^* H6 \setminus^* H7 \setminus^* H8 \setminus^* \setminus[] \\ &= H8 \setminus^* H7 \setminus^* H6 \setminus^* H5 \setminus^* H4 \setminus^* H3 \setminus^* H2 \setminus^* H1 \setminus^* \setminus[] \end{aligned}$$

Proof using. *intros. rewrite* $\leftarrow$ (*hstars_flip_7 H1*). *rew_heap. rewrite (hstar_comm H8). rew_heap*-. Qed.

Lemma *hstars_flip_9* : $\forall H1 H2 H3 H4 H5 H6 H7 H8 H9,$

$H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H8 \setminus * H9 \setminus * \setminus []$
 $= H9 \setminus * H8 \setminus * H7 \setminus * H6 \setminus * H5 \setminus * H4 \setminus * H3 \setminus * H2 \setminus * H1 \setminus * \setminus []$.

Proof using. intros. rewrite $\leftarrow$ (hstars_flip_8 H1). rew_heap. rewrite (hstar_comm H9). rew_heap. Qed.

```
Ltac hstars_flip_lemma i :=
  match number_to_nat i with
  | 0%nat => constr:(hstars_flip_0)
  | 1%nat => constr:(hstars_flip_1)
  | 2%nat => constr:(hstars_flip_2)
  | 3%nat => constr:(hstars_flip_3)
  | 4%nat => constr:(hstars_flip_4)
  | 5%nat => constr:(hstars_flip_5)
  | 6%nat => constr:(hstars_flip_6)
  | 7%nat => constr:(hstars_flip_7)
  | 8%nat => constr:(hstars_flip_8)
  | 9%nat => constr:(hstars_flip_9)
  | _ => constr:(hstars_flip_1)
  end.
```

```
Ltac hstars_arity i Hs :=
  match Hs with
  | \[] => constr:(i)
  | ?H1 \* ?H2 => hstars_arity (S i) H2
  end.
```

```
Ltac hstars_flip_arity tt :=
  match goal with  $\vdash$  ?HL = ?HR => hstars_arity 0%nat HL end.
```

```
Ltac hstars_flip tt :=
  let i := hstars_flip_arity tt in
  let L := hstars_flip_lemma i in
  eapply L.
```

Lemmas *hstars_pick_...* to extract hyps in depth.

hstars_search Hs test applies to an expression *Hs* of the form either $H1 \setminus * \dots \setminus * Hn \setminus * \setminus []$ or $H1 \setminus * \dots \setminus * Hn$. It invokes the function *test i Hi* for each of the *Hi* in turn until the tactic succeeds. In the particular case of invoking *test n Hn* when there is no trailing $\setminus []$, the call is of the form *test* (hstars_last n) *Hn* where *hstars_last* is an identity tag.

Definition hstars_last (*A:Type*) (*X:A*) := *X*.

```
Ltac hstars_search Hs test :=
  let rec aux i Hs :=
    first | match Hs with ?H \* _ => test i H end
          | match Hs with _ \* ?Hs' => aux (S i) Hs' end
```

| match Hs with ? $H \Rightarrow test (hstars\_last\ i)\ H$ end | in
aux 1%nat Hs.

hstars_pick_lemma i returns one of the lemma below, which enables reordering in iterated stars, by extracting the i -th item to bring it to the front.

Lemma *hstars_pick_1* : $\forall\ H1\ H,$
 $H1 \setminus * H = H1 \setminus * H.$

Proof using. auto. Qed.

Lemma *hstars_pick_2* : $\forall\ H1\ H2\ H,$
 $H1 \setminus * H2 \setminus * H = H2 \setminus * H1 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H1).$ Qed.

Lemma *hstars_pick_3* : $\forall\ H1\ H2\ H3\ H,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H = H3 \setminus * H1 \setminus * H2 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H2).$ *applies* *hstars_pick_2.* Qed.

Lemma *hstars_pick_4* : $\forall\ H1\ H2\ H3\ H4\ H,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H = H4 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H3).$ *applies* *hstars_pick_3.* Qed.

Lemma *hstars_pick_5* : $\forall\ H1\ H2\ H3\ H4\ H5\ H,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H = H5 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H4).$ *applies* *hstars_pick_4.* Qed.

Lemma *hstars_pick_6* : $\forall\ H1\ H2\ H3\ H4\ H5\ H6\ H,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H$
 $= H6 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H5).$ *applies* *hstars_pick_5.* Qed.

Lemma *hstars_pick_7* : $\forall\ H1\ H2\ H3\ H4\ H5\ H6\ H7\ H,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H$
 $= H7 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H6).$ *applies* *hstars_pick_6.* Qed.

Lemma *hstars_pick_8* : $\forall\ H1\ H2\ H3\ H4\ H5\ H6\ H7\ H8\ H,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H8 \setminus * H$
 $= H8 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H7).$ *applies* *hstars_pick_7.* Qed.

Lemma *hstars_pick_9* : $\forall\ H1\ H2\ H3\ H4\ H5\ H6\ H7\ H8\ H9\ H,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H8 \setminus * H9 \setminus * H$
 $= H9 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H8 \setminus * H.$

Proof using. intros. rewrite $\neg (hstar\_comm\_assoc\ H8).$ *applies* *hstars_pick_8.* Qed.

Lemma *hstars_pick_last_1* : $\forall\ H1,$
 $H1 = H1.$

Proof using. auto. Qed.

Lemma *hstars_pick_last_2* : $\forall\ H1\ H2,$

$H1 \setminus * H2 = H2 \setminus * H1$.
 Proof using. intros. rewrite $\neg$ (hstar_comm). Qed.

Lemma hstars_pick_last_3 : $\forall H1 H2 H3$,
 $H1 \setminus * H2 \setminus * H3 = H3 \setminus * H1 \setminus * H2$.
 Proof using. intros. rewrite $\neg$ (hstar_comm H2). applys hstars_pick_2. Qed.

Lemma hstars_pick_last_4 : $\forall H1 H2 H3 H4$,
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 = H4 \setminus * H1 \setminus * H2 \setminus * H3$.
 Proof using. intros. rewrite $\neg$ (hstar_comm H3). applys hstars_pick_3. Qed.

Lemma hstars_pick_last_5 : $\forall H1 H2 H3 H4 H5$,
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 = H5 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4$.
 Proof using. intros. rewrite $\neg$ (hstar_comm H4). applys hstars_pick_4. Qed.

Lemma hstars_pick_last_6 : $\forall H1 H2 H3 H4 H5 H6$,
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6$
 $= H6 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5$.
 Proof using. intros. rewrite $\neg$ (hstar_comm H5). applys hstars_pick_5. Qed.

Lemma hstars_pick_last_7 : $\forall H1 H2 H3 H4 H5 H6 H7$,
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7$
 $= H7 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6$.
 Proof using. intros. rewrite $\neg$ (hstar_comm H6). applys hstars_pick_6. Qed.

Lemma hstars_pick_last_8 : $\forall H1 H2 H3 H4 H5 H6 H7 H8$,
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H8$
 $= H8 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7$.
 Proof using. intros. rewrite $\neg$ (hstar_comm H7). applys hstars_pick_7. Qed.

Lemma hstars_pick_last_9 : $\forall H1 H2 H3 H4 H5 H6 H7 H8 H9$,
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H8 \setminus * H9$
 $= H9 \setminus * H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \setminus * H6 \setminus * H7 \setminus * H8$.
 Proof using. intros. rewrite $\neg$ (hstar_comm H8). applys hstars_pick_8. Qed.

Ltac hstars_pick_lemma i :=
 let unsupported tt := fail 100 "hstars_pick supports only arity up to 9" in
 match i with
 | hstars_last ?j $\Rightarrow$ match number_to_nat j with
 | 1%nat $\Rightarrow$ constr:(hstars_pick_last_1)
 | 2%nat $\Rightarrow$ constr:(hstars_pick_last_2)
 | 3%nat $\Rightarrow$ constr:(hstars_pick_last_3)
 | 4%nat $\Rightarrow$ constr:(hstars_pick_last_4)
 | 5%nat $\Rightarrow$ constr:(hstars_pick_last_5)
 | 6%nat $\Rightarrow$ constr:(hstars_pick_last_6)
 | 7%nat $\Rightarrow$ constr:(hstars_pick_last_7)
 | 8%nat $\Rightarrow$ constr:(hstars_pick_last_8)
 | 9%nat $\Rightarrow$ constr:(hstars_pick_last_9)

```

| _ => unsupported tt
end
| ?j => match number_to_nat j with
| 1%nat => constr:(hstars_pick_1)
| 2%nat => constr:(hstars_pick_2)
| 3%nat => constr:(hstars_pick_3)
| 4%nat => constr:(hstars_pick_4)
| 5%nat => constr:(hstars_pick_5)
| 6%nat => constr:(hstars_pick_6)
| 7%nat => constr:(hstars_pick_7)
| 8%nat => constr:(hstars_pick_8)
| 9%nat => constr:(hstars_pick_9)
| _ => unsupported tt
end
end.

```

Documentation for the Tactic *xsimpl*

... doc for *xsimpl* to update

At the end, there remains a heap entailment with a simplified LHS and a simplified RHS, with items not cancelled out. At this point, if the goal is of the form $H ==> \backslash GC$ or $H ==> \backslash Top$ or $H ==> ?H'$ for some evar H' , then *xsimpl* solves the goal. Else, it leaves whatever remains.

For the cancellation part, *xsimpl* cancels out H from the LHS with H' from the RHS if either H' is syntactically equal to H , or if H and H' both have the form $x \sim > \dots$ for the same x . Note that, in case of a mismatch with $x \sim > R X$ on the LHS and $x \sim > R' X'$ on the RHS, *xsimpl* will produce a goal of the form $(x \sim > R X) = (x \sim > R' X')$ which will likely be unsolvable. It is the user's responsibility to perform the appropriate conversion steps prior to calling *xsimpl*.

Remark: the reason for the special treatment of $x \sim > \dots$ is that it is very useful to be able to automatically cancel out $x \sim > R X$ from the LHS with $x \sim > R ?Y$, for some evar $?Y$ which typically is introduced from an existential, e.g. $\backslash \exists Y, x \sim > R Y$.

Remark: importantly, *xsimpl* does not attempt any simplification on a representation predicate of the form $?x \sim > \dots$, when the $?x$ is an uninstantiated evar. Such situation may arise for example with the following RHS: $\backslash \exists p, (r \sim > Ref\ p) \backslash * (p \sim > Ref\ 3)$.

As a special feature, *xsimpl* may be provided optional arguments for instantiating the existentials (instead of introducing evars). These optional arguments need to be given in left-right order, and are used on a first-match basis: the head value is used if its type matches the type expected by the existential, else an evar is introduced for that existential.

Xsimpl (*Hla*, *Hlw*, *Hlt*) (*Hra*, *Hrg*, *Hrt*) is interpreted as the entailment $Hla \backslash * Hlw \backslash * Hlt ==> Hra \backslash * Hrg \backslash * Hrt$ where

- *Hla* denotes “cleaned up” items from the left hand side

- Hlw denotes the $H1 \setminus^* H2$ and $Q1 \setminus^* Q2$ items from the left hand side
- Hlt denotes the remaining items to process items from the left hand side
- Hra denotes “cleaned up” items from the right hand side
- Hrg denotes the $\setminus GC$ and $\setminus Top$ items from the right hand side
- Hrt denotes the remaining items to process from the right hand side

Note: we assume that all items consist of iterated hstars, and are always terminated by an empty heap.

Definition $Xsimpl\ (HL\ HR:hprop \times hprop \times hprop) :=$

```
let '(Hla, Hlw, Hlt) := HL in
let '(Hra, Hrg, Hrt) := HR in
Hla \* Hlw \* Hlt ==> Hra \* Hrg \* Hrt.
```

protect X is use to prevent $xsimpl$ from investigating inside X

Definition $protect\ (A:Type)\ (X:A) : A := X$.

Auxiliary lemmas to prove lemmas for $xsimpl$ implementation.

Lemma $Xsimpl\_trans\_l : \forall\ Hla1\ Hlw1\ Hlt1\ Hla2\ Hlw2\ Hlt2\ HR,$
 $Xsimpl\ (Hla2, Hlw2, Hlt2)\ HR \rightarrow$
 $Hla1 \setminus^* Hlw1 \setminus^* Hlt1 ==> Hla2 \setminus^* Hlw2 \setminus^* Hlt2 \rightarrow$
 $Xsimpl\ (Hla1, Hlw1, Hlt1)\ HR.$

Proof using.

```
introv M1 E. destruct HR as [[Hra Hrg] Hrt]. unfolds Xsimpl.
applies \times himpl\_trans M1.
```

Qed.

Lemma $Xsimpl\_trans\_r : \forall\ Hra1\ Hrg1\ Hrt1\ Hra2\ Hrg2\ Hrt2\ HL,$
 $Xsimpl\ HL\ (Hra2, Hrg2, Hrt2) \rightarrow$
 $Hra2 \setminus^* Hrg2 \setminus^* Hrt2 ==> Hra1 \setminus^* Hrg1 \setminus^* Hrt1 \rightarrow$
 $Xsimpl\ HL\ (Hra1, Hrg1, Hrt1).$

Proof using.

```
introv M1 E. destruct HL as [[Hla Hlw] Hlt]. unfolds Xsimpl.
applies \times himpl\_trans M1.
```

Qed.

Lemma $Xsimpl\_trans : \forall\ Hla1\ Hlw1\ Hlt1\ Hla2\ Hlw2\ Hlt2\ Hra1\ Hrg1\ Hrt1\ Hra2\ Hrg2\ Hrt2,$
 $Xsimpl\ (Hla2, Hlw2, Hlt2)\ (Hra2, Hrg2, Hrt2) \rightarrow$
 $(Hla2 \setminus^* Hlw2 \setminus^* Hlt2 ==> Hra2 \setminus^* Hrg2 \setminus^* Hrt2 \rightarrow$
 $Hla1 \setminus^* Hlw1 \setminus^* Hlt1 ==> Hra1 \setminus^* Hrg1 \setminus^* Hrt1) \rightarrow$
 $Xsimpl\ (Hla1, Hlw1, Hlt1)\ (Hra1, Hrg1, Hrt1).$

Proof using. $introv\ M1\ E.$ unfolds $Xsimpl$. eauto. Qed.

Basic cancellation tactic used to establish lemmas used by *xsimpl*

```
Lemma hstars_simpl_start : ∀ H1 H2,
  H1 \* \[] ==> \[] \* H2 \* \[] →
  H1 ==> H2.
```

Proof using. *intros* *M*. *rew_heap* $\neg$ in *. Qed.

```
Lemma hstars_simpl_keep : ∀ H1 Ha H Ht,
  H1 ==> (H \* Ha) \* Ht →
  H1 ==> Ha \* H \* Ht.
```

Proof using. *intros* *M*. *rew_heap* in *. *rewrite* $\neg$ *hstar_comm_assoc*. Qed.

```
Lemma hstars_simpl_cancel : ∀ H1 Ha H Ht,
  H1 ==> Ha \* Ht →
  H \* H1 ==> Ha \* H \* Ht.
```

Proof using. *intros* *M*. *rewrite* *hstar_comm_assoc*. *applies* $\neg$ *himpl_frame_lr*. Qed.

```
Lemma hstars_simpl_pick_lemma : ∀ H1 H1' H2,
  H1 = H1' →
  H1' ==> H2 →
  H1 ==> H2.
```

Proof using. *intros* *M*. *subst* $\neg$. Qed.

```
Ltac hstars_simpl_pick i :=
```

```
  let L := hstars_pick_lemma i in
  eapply hstars_simpl_pick_lemma; [ apply L | ].
```

```
Ltac hstars_simpl_start tt :=
  match goal with ⊢ ?H1 ==> ?H2 ⇒ idtac end;
  applies hstars_simpl_start;
  rew_heap_assoc.
```

```
Ltac hstars_simpl_step tt :=
  match goal with ⊢ ?H1 ==> ?Ha \* ?H \* ?H2 ⇒
  first [
    hstars_search H1 ltac:(fun i H' ⇒
      match H' with H ⇒ hstars_simpl_pick i end);
    apply hstars_simpl_cancel
  | apply hstars_simpl_keep ]
  end.
```

```
Ltac hstars_simpl_post tt :=
  rew_heap; try apply himpl_refl.
```

```
Ltac hstars_simpl_core tt :=
  hstars_simpl_start tt;
  repeat (hstars_simpl_step tt);
```

hstars_simpl_post tt.

Tactic Notation "hstars_simpl" :=
hstars_simpl_core tt.

Transition lemmas

Transition lemmas to start the process

Lemma *xpull_protect* : $\forall H1 H2,$
 $H1 \implies \text{protect } H2 \rightarrow$
 $H1 \implies H2.$

Proof using. *auto. Qed.*

Lemma *xsimpl_start* : $\forall H1 H2,$
 $\text{Xsimpl } (\backslash [], \backslash [], (H1 \backslash * \backslash [])) (\backslash [], \backslash [], (H2 \backslash * \backslash [])) \rightarrow$
 $H1 \implies H2.$

Proof using. *intros M. unfolds Xsimpl. rew_heap $\neg$ in *. Qed.*

Transition lemmas for LHS extraction operations

Ltac *xsimpl_l_start M* :=
intros M;
match goal with HR: hprop $\times$ hprop $\times$ hprop $\vdash$ _ $\Rightarrow$
destruct HR as [[Hra Hrg] Hrt]; unfolds Xsimpl end.

Ltac *xsimpl_l_start' M* :=
xsimpl_l_start M; applies himpl_trans (rm M); hstars_simpl.

Lemma *xsimpl_l_empty* : $\forall Hla Hlw Hlt HR,$
 $\text{Xsimpl } (Hla, Hlw, Hlt) HR \rightarrow$
 $\text{Xsimpl } (Hla, Hlw, (\backslash [] \backslash * Hlt)) HR.$

Proof using. *xsimpl_l_start' M. Qed.*

Lemma *xsimpl_l_hpure* : $\forall P Hla Hlw Hlt HR,$
 $(P \rightarrow \text{Xsimpl } (Hla, Hlw, Hlt) HR) \rightarrow$
 $\text{Xsimpl } (Hla, Hlw, (\backslash [P] \backslash * Hlt)) HR.$

Proof using.

xsimpl_l_start M. rewrite hstars_pick_3. applies $\times$ himpl_hstar_hpure_l.

Qed.

Lemma *xsimpl_l_hexists* : $\forall A (J:A \rightarrow \text{hprop}) Hla Hlw Hlt HR,$
 $(\forall x, \text{Xsimpl } (Hla, Hlw, (J x \backslash * Hlt)) HR) \rightarrow$
 $\text{Xsimpl } (Hla, Hlw, (\text{hexists } J \backslash * Hlt)) HR.$

Proof using.

xsimpl_l_start M. rewrite hstars_pick_3. rewrite hstar_hexists.

applies $\times$ himpl_hexists_l. intros. rewrite $\neg \leftarrow$ hstars_pick_3.

Qed.

Lemma *xsimpl_l_acc_wand* : $\forall H Hla Hlw Hlt HR,$

$\text{Xsimpl } (Hla, (H \setminus * Hlw), Hlt) HR \rightarrow$
 $\text{Xsimpl } (Hla, Hlw, (H \setminus * Hlt)) HR.$

Proof using. *xsimpl_l_start'* M. Qed.

Lemma *xsimpl_l_acc_other* : $\forall H Hla Hlw Hlt HR,$

$\text{Xsimpl } ((H \setminus * Hla), Hlw, Hlt) HR \rightarrow$
 $\text{Xsimpl } (Hla, Hlw, (H \setminus * Hlt)) HR.$

Proof using. *xsimpl_l_start'* M. Qed.

Transition lemmas for LHS cancellation operations *Hlt* is meant to be empty there

Lemma *xsimpl_l_cancel_hwand_hempty* : $\forall H2 Hla Hlw Hlt HR,$

$\text{Xsimpl } (Hla, Hlw, (H2 \setminus * Hlt)) HR \rightarrow$
 $\text{Xsimpl } (Hla, ((\setminus []) \setminus * H2) \setminus * Hlw), Hlt) HR.$

Proof using. *xsimpl_l_start'* M. Qed.

Lemma *xsimpl_l_cancel_hwand* : $\forall H1 H2 Hla Hlw Hlt HR,$

$\text{Xsimpl } (\setminus [], Hlw, (Hla \setminus * H2 \setminus * Hlt)) HR \rightarrow$
 $\text{Xsimpl } ((H1 \setminus * Hla), ((H1 \setminus * H2) \setminus * Hlw), Hlt) HR.$

Proof using. *xsimpl_l_start'* M. *applies* $\neg$ *hwand_cancel*. Qed.

Lemma *xsimpl_l_cancel_qwand* : $\forall A (x:A) (Q1 Q2:A \rightarrow \text{hprop}) Hla Hlw Hlt HR,$

$\text{Xsimpl } (\setminus [], Hlw, (Hla \setminus * Q2 x \setminus * Hlt)) HR \rightarrow$
 $\text{Xsimpl } ((Q1 x \setminus * Hla), ((Q1 \setminus * Q2) \setminus * Hlw), Hlt) HR.$

Proof using.

xsimpl_l_start' M. *rewrite* *hstar_comm*. *applies* *himpl_hstar_trans_l*.
applies *qwand_specialize x*. *rewrite* *hstar_comm*. *applies* *hwand_cancel*.

Qed.

Lemma *xsimpl_l_keep_wand* : $\forall H Hla Hlw Hlt HR,$

$\text{Xsimpl } ((H \setminus * Hla), Hlw, Hlt) HR \rightarrow$
 $\text{Xsimpl } (Hla, (H \setminus * Hlw), Hlt) HR.$

Proof using. *xsimpl_l_start'* M. Qed.

Lemma *xsimpl_l_hwand_reorder* : $\forall H1 H1' H2 Hla Hlw Hlt HR,$

$H1 = H1' \rightarrow$
 $\text{Xsimpl } (Hla, ((H1' \setminus * H2) \setminus * Hlw), Hlt) HR \rightarrow$
 $\text{Xsimpl } (Hla, ((H1 \setminus * H2) \setminus * Hlw), Hlt) HR.$

Proof using. *intros*. *subst* $\times$. Qed.

Lemma *xsimpl_l_cancel_hwand_hstar* : $\forall H1 H2 H3 Hla Hlw Hlt HR,$

$\text{Xsimpl } (Hla, Hlw, ((H2 \setminus * H3) \setminus * Hlt)) HR \rightarrow$
 $\text{Xsimpl } ((H1 \setminus * Hla), (((H1 \setminus * H2) \setminus * H3) \setminus * Hlw), Hlt) HR.$

Proof using.

xsimpl_l_start' M. *rewrite* *hwand_curry_eq*. *applies* *hwand_cancel*.

Qed.

Lemma *xsimpl_l_cancel_hwand_hstar_hempty* : $\forall H2 H3 Hla Hlw Hlt HR,$

$\text{Xsimpl } (Hla, Hlw, ((H2 \setminus * H3) \setminus * Hlt)) HR \rightarrow$

$\text{Xsimpl } (Hla, (((\backslash [] \backslash * H2) \backslash - * H3) \backslash * Hlw), Hlt) \text{ HR.}$
 Proof using. $\text{ximpl\_l\_start' } M. \text{ Qed.}$

Transition lemmas for RHS extraction operations

$\text{Ltac ximpl\_r\_start } M :=$
 $\text{ introv } M;$
 $\text{ match goal with } HL: \text{ hprop} \times \text{ hprop} \times \text{ hprop} \vdash \_ \Rightarrow$
 $\text{ destruct } HL \text{ as } [[Hla \ Hlw] \ Hlt]; \text{ unfolds } \text{Xsimpl} \text{ end.}$

$\text{Ltac ximpl\_r\_start' } M :=$
 $\text{ ximpl\_r\_start } M; \text{ applies himpl\_trans (rm } M); \text{ hstars\_simpl.}$

$\text{Lemma ximpl\_r\_hempty : } \forall \text{ Hra Hrg Hrt HL,}$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, Hrt) } \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, } (\backslash [] \backslash * \text{ Hrt})).$

Proof using. $\text{ximpl\_r\_start' } M. \text{ Qed.}$

$\text{Lemma ximpl\_r\_hwand\_same : } \forall \text{ H Hra Hrg Hrt HL,}$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, Hrt) } \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, } ((H \backslash - * H) \backslash * \text{ Hrt})).$

Proof using. $\text{ximpl\_r\_start' } M. \text{ rewrite hwand\_equiv. rew\_heap} \neg. \text{ Qed.}$

$\text{Lemma ximpl\_r\_hpure : } \forall \text{ P Hra Hrg Hrt HL,}$
 $\text{ P } \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, Hrt) } \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, } (\backslash [P] \backslash * \text{ Hrt})).$

Proof using.

$\text{ introv } HP. \text{ ximpl\_r\_start' } M. \text{ applies} \times \text{ himpl\_hempty\_hpure.}$

Qed.

$\text{Lemma ximpl\_r\_hexists : } \forall \text{ A (x:A) (J:A} \rightarrow \text{ hprop) Hra Hrg Hrt HL,}$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, } (J \text{ x } \backslash * \text{ Hrt})) \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, (hexists J } \backslash * \text{ Hrt)).}$

Proof using. $\text{ximpl\_r\_start' } M. \text{ applies} \times \text{ himpl\_hexists\_r. Qed.}$

$\text{Lemma ximpl\_r\_id : } \forall \text{ A (x X:A) Hra Hrg Hrt HL,}$
 $\text{ (X = x) } \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, Hrt) } \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, (x } \sim > \text{ Id X } \backslash * \text{ Hrt)).}$

Proof using.

$\text{ introv } \rightarrow. \text{ ximpl\_r\_start' } M. \text{ rewrite repr\_id.}$

$\text{ applies} \times \text{ himpl\_hempty\_hpure.}$

Qed.

$\text{Lemma ximpl\_r\_id\_unify : } \forall \text{ A (x:A) Hra Hrg Hrt HL,}$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, Hrt) } \rightarrow$
 $\text{ Xsimpl } HL \text{ (Hra, Hrg, (x } \sim > \text{ Id x } \backslash * \text{ Hrt)).}$

Proof using. $\text{ introv } M. \text{ applies} \neg \text{ ximpl\_r\_id. Qed.}$

Lemma xsimpl_r_keep : $\forall H \text{ Hra Hrg Hrt HL},$
 $\text{Xsimpl } HL ((H \setminus * \text{ Hra}), \text{ Hrg}, \text{ Hrt}) \rightarrow$
 $\text{Xsimpl } HL (\text{ Hra}, \text{ Hrg}, (H \setminus * \text{ Hrt})).$

Proof using. *xsimpl_r_start'* M. Qed.

Transition lemmas for $\setminus \text{Top}$ and $\setminus \text{GC}$ cancellation

Lemma xsimpl_r_hgc_or_htop : $\forall H \text{ Hra Hrg Hrt HL},$
 $\text{Xsimpl } HL (\text{ Hra}, (H \setminus * \text{ Hrg}), \text{ Hrt}) \rightarrow$
 $\text{Xsimpl } HL (\text{ Hra}, \text{ Hrg}, (H \setminus * \text{ Hrt})).$

Proof using. *xsimpl_r_start'* M. Qed.

Lemma xsimpl_r_htop_replace_hgc : $\forall \text{ Hra Hrg Hrt HL},$
 $\text{Xsimpl } HL (\text{ Hra}, (\setminus \text{Top} \setminus * \text{ Hrg}), \text{ Hrt}) \rightarrow$
 $\text{Xsimpl } HL (\text{ Hra}, (\setminus \text{GC} \setminus * \text{ Hrg}), (\setminus \text{Top} \setminus * \text{ Hrt})).$

Proof using. *xsimpl_r_start'* M. *applies* *himpl_hgc_r*. *xaffine*. Qed.

Lemma xsimpl_r_hgc_drop : $\forall \text{ Hra Hrg Hrt HL},$
 $\text{Xsimpl } HL (\text{ Hra}, \text{ Hrg}, \text{ Hrt}) \rightarrow$
 $\text{Xsimpl } HL (\text{ Hra}, \text{ Hrg}, (\setminus \text{GC} \setminus * \text{ Hrt})).$

Proof using. *xsimpl_r_start'* M. *applies* *himpl_hgc_r*. *xaffine*. Qed.

Lemma xsimpl_r_htop_drop : $\forall \text{ Hra Hrg Hrt HL},$
 $\text{Xsimpl } HL (\text{ Hra}, \text{ Hrg}, \text{ Hrt}) \rightarrow$
 $\text{Xsimpl } HL (\text{ Hra}, \text{ Hrg}, (\setminus \text{Top} \setminus * \text{ Hrt})).$

Proof using. *xsimpl_r_start'* M. *applies* *himpl_htop_r*. Qed.

Transition lemmas for LHS/RHS cancellation meant to be applied when Hlw and Hlt are empty

Ltac *xsimpl_lr_start* M :=
intros M; *unfolds* *Xsimpl*; *rew_heap* in *.

Ltac *xsimpl_lr_start'* M :=
xsimpl_lr_start M; *hstars_simpl*;
try (*applies* *himpl_trans* (*rm* M); *hstars_simpl*).

Lemma xsimpl_lr_cancel_same : $\forall H \text{ Hla Hlw Hlt Hra Hrg Hrt},$
 $\text{Xsimpl } (\text{ Hla}, \text{ Hlw}, \text{ Hlt}) (\text{ Hra}, \text{ Hrg}, \text{ Hrt}) \rightarrow$
 $\text{Xsimpl } ((H \setminus * \text{ Hla}), \text{ Hlw}, \text{ Hlt}) (\text{ Hra}, \text{ Hrg}, (H \setminus * \text{ Hrt})).$

Proof using. *xsimpl_lr_start'* M. Qed.

Lemma xsimpl_lr_cancel_htop : $\forall H \text{ Hla Hlw Hlt Hra Hrg Hrt},$
 $\text{Xsimpl } (\text{ Hla}, \text{ Hlw}, \text{ Hlt}) (\text{ Hra}, (\setminus \text{Top} \setminus * \text{ Hrg}), \text{ Hrt}) \rightarrow$
 $\text{Xsimpl } ((H \setminus * \text{ Hla}), \text{ Hlw}, \text{ Hlt}) (\text{ Hra}, (\setminus \text{Top} \setminus * \text{ Hrg}), \text{ Hrt}).$

Proof using.

xsimpl_lr_start M. *rewrite* (*hstar_comm_assoc* Hra) in *.
rewrite $\leftarrow$ *hstar_htop_htop*. *rew_heap*. *applies* *himpl_frame_lr* M.
applies *himpl_htop_r*.

Qed.

Lemma xsimpl_lr_cancel_hgc : $\forall Hla Hlw Hlt Hra Hrg Hrt,$
 $Xsimpl (Hla, Hlw, Hlt) (Hra, (\backslash GC \backslash * Hrg), Hrt) \rightarrow$
 $Xsimpl ((\backslash GC \backslash * Hla), Hlw, Hlt) (Hra, (\backslash GC \backslash * Hrg), Hrt).$

Proof using.

xsimpl_lr_start *M*. *rewrite* (*hstar_comm_assoc* *Hra*).
rewrite $\leftarrow$ *hstar_hgc_hgc* at 2. *rew_heap*.
applies $\neg$ *himpl_frame_lr*. *applies* *himpl_trans* *M*. *hstars_simpl*.

Qed.

Lemma xsimpl_lr_cancel_eq : $\forall H1 H2 Hla Hlw Hlt Hra Hrg Hrt,$
 $(H1 = H2) \rightarrow$
 $Xsimpl (Hla, Hlw, Hlt) (Hra, Hrg, Hrt) \rightarrow$
 $Xsimpl ((H1 \backslash * Hla), Hlw, Hlt) (Hra, Hrg, (H2 \backslash * Hrt)).$

Proof using. *intros* $\rightarrow$. *apply* $\neg$ *xsimpl_lr_cancel_same*. Qed.

Lemma xsimpl_lr_cancel_eq_repr : $\forall A p (E1 E2:A \rightarrow hprop) Hla Hlw Hlt Hra Hrg Hrt,$
 $E1 = E2 \rightarrow$
 $Xsimpl (Hla, Hlw, Hlt) (Hra, Hrg, Hrt) \rightarrow$
 $Xsimpl (((p \sim > E1) \backslash * Hla), Hlw, Hlt) (Hra, Hrg, ((p \sim > E2) \backslash * Hrt)).$

Proof using. *intros* *M*. *subst*. *apply* $\neg$ *xsimpl_lr_cancel_same*. Qed.

Lemma xsimpl_lr_hwand : $\forall H1 H2 Hla,$
 $Xsimpl (\backslash [], \backslash [], (H1 \backslash * Hla)) (\backslash [], \backslash [], H2 \backslash * \backslash []) \rightarrow$
 $Xsimpl (Hla, \backslash [], \backslash []) ((H1 \backslash - * H2) \backslash * \backslash [], \backslash [], \backslash []).$

Proof using.

xsimpl_lr_start' *M*. *rewrite* *hwand_equiv*.
applies *himpl_trans* (*rm* *M*). *hstars_simpl*.

Qed.

Lemma xsimpl_lr_hwand_hfalse : $\forall Hla H1,$
 $Xsimpl (Hla, \backslash [], \backslash []) ((\backslash [\mathbf{False}] \backslash - * H1) \backslash * \backslash [], \backslash [], \backslash []).$

Proof using.

intros. *generalize* **True**. *xsimpl_lr_start* *M*. *rewrite* *hwand_equiv*.
applies *himpl_hstar_hpure_l*. *auto_false*.

Qed.

Lemma xsimpl_lr_qwand : $\forall A (Q1 Q2:A \rightarrow hprop) Hla,$
 $(\forall x, Xsimpl (\backslash [], \backslash [], (Q1 x \backslash * Hla)) (\backslash [], \backslash [], Q2 x \backslash * \backslash [])) \rightarrow$
 $Xsimpl (Hla, \backslash [], \backslash []) ((Q1 \backslash - * Q2) \backslash * \backslash [], \backslash [], \backslash []).$

Proof using.

xsimpl_lr_start *M*. *rewrite* *qwand_equiv*. *intros* *x*.
specializes *M* *x*. *rew_heap* $\neg$ *in* *M*.

Qed.

Lemma xsimpl_lr_qwand_unit : $\forall (Q1 Q2:\mathbf{unit} \rightarrow hprop) Hla,$
 $Xsimpl (\backslash [], \backslash [], (Q1 \mathbf{tt} \backslash * Hla)) (\backslash [], \backslash [], (Q2 \mathbf{tt} \backslash * \backslash [])) \rightarrow$
 $Xsimpl (Hla, \backslash [], \backslash []) ((Q1 \backslash - * Q2) \backslash * \backslash [], \backslash [], \backslash []).$

Proof using. *intros M. applies xsimpl_lr_qwand. intros []. applies M. Qed.*

Lemma xsimpl_lr_hforall : $\forall A (J:A \rightarrow \text{hprop}) Hla,$
 $(\forall x, \text{Xsimpl} (\backslash [], \backslash [], Hla) (\backslash [], \backslash [], J x \backslash * \backslash [])) \rightarrow$
 $\text{Xsimpl} (Hla, \backslash [], \backslash []) ((\text{hforall } J) \backslash * \backslash [], \backslash [], \backslash []).$

Proof using.

xsimpl_lr_start M. applies himpl_hforall_r. intros x.
specializes M x. rew_heap $\neg$ in M.

Qed.

Lemma himpl_lr_refl : $\forall Hla,$
 $\text{Xsimpl} (Hla, \backslash [], \backslash []) (Hla, \backslash [], \backslash []).$

Proof using. *intros. unfolds Xsimpl. hstars_simpl. Qed.*

Lemma himpl_lr_qwand_unify : $\forall A (Q:A \rightarrow \text{hprop}) Hla,$
 $\text{Xsimpl} (Hla, \backslash [], \backslash []) ((Q \backslash * (Q \backslash * Hla)) \backslash * \backslash [], \backslash [], \backslash []).$

Proof using. *intros. unfolds Xsimpl. hstars_simpl. rewrite $\neg$ qwand_equiv. Qed.*

Lemma himpl_lr_htop : $\forall Hla Hrg,$
 $\text{Xsimpl} (\backslash [], \backslash [], \backslash []) (\backslash [], Hrg, \backslash []) \rightarrow$
 $\text{Xsimpl} (Hla, \backslash [], \backslash []) (\backslash [], (\backslash \text{Top} \backslash * Hrg), \backslash []).$

Proof using.

xsimpl_lr_start M. rewrite $\leftarrow$ (hstar_empty_l Hla).
applies himpl_hstar_trans_l M. hstars_simpl. apply himpl_htop_r.

Qed.

Lemma himpl_lr_hgc : $\forall Hla Hrg,$
 $\text{haffine } Hla \rightarrow$
 $\text{Xsimpl} (\backslash [], \backslash [], \backslash []) (\backslash [], Hrg, \backslash []) \rightarrow$
 $\text{Xsimpl} (Hla, \backslash [], \backslash []) (\backslash [], (\backslash \text{GC} \backslash * Hrg), \backslash []).$

Proof using.

intros N. xsimpl_lr_start M. rewrite $\leftarrow$ (hstar_empty_l Hla).
applies himpl_hstar_trans_l M. hstars_simpl. apply $\times$ himpl_hgc_r.

Qed.

Lemma xsimpl_lr_exit_nogc : $\forall Hla Hra,$
 $Hla ==> Hra \rightarrow$
 $\text{Xsimpl} (Hla, \backslash [], \backslash []) (Hra, \backslash [], \backslash []).$

Proof using. *intros M. unfolds Xsimpl. hstars_simpl. auto. Qed.*

Lemma xsimpl_lr_exit : $\forall Hla Hra Hrg,$
 $Hla ==> Hra \backslash * Hrg \rightarrow$
 $\text{Xsimpl} (Hla, \backslash [], \backslash []) (Hra, Hrg, \backslash []).$

Proof using. *intros M. unfolds Xsimpl. hstars_simpl. rewrite $\neg$ hstar_comm. Qed.*

Lemmas to flip accumulators back in place

Lemma xsimpl_flip_acc_l : $\forall Hla Hra Hla' Hrg,$
 $Hla = Hla' \rightarrow$

```

Xsimpl (Hla', \[], \[]) (Hra, Hrg, \[]) →
Xsimpl (Hla, \[], \[]) (Hra, Hrg, \[]).
Proof using. introv E1 M. subst×. Qed.

```

```

Lemma xsimpl_flip_acc_r : ∀ Hla Hra Hra' Hrg,
  Hra = Hra' →
  Xsimpl (Hla, \[], \[]) (Hra', Hrg, \[]) →
  Xsimpl (Hla, \[], \[]) (Hra, Hrg, \[]).
Proof using. introv E1 M. subst×. Qed.

```

```

Ltac xsimpl_flip_acc_l tt :=
  eapply xsimpl_flip_acc_l; [ hstars_flip tt | ].
Ltac xsimpl_flip_acc_r tt :=
  eapply xsimpl_flip_acc_r; [ hstars_flip tt | ].
Ltac xsimpl_flip_acc_lr tt :=
  xsimpl_flip_acc_l tt; xsimpl_flip_acc_r tt.

```

Lemmas to pick the hypothesis to cancel

xsimpl_pick *i* applies to a goal of the form $\text{Xsimpl } ((H1 \setminus^* \dots \setminus^* Hi \setminus^* \dots \setminus^* Hn), Hlw, Hlt) \text{ } HR$ and turns it into $\text{Xsimpl } ((Hi \setminus^* H1 \dots \setminus^* H\{i-1\} \setminus^* H\{i+1\} \setminus^* \dots \setminus^* Hn), Hlw, Hlt) \text{ } HR$.

```

Lemma xsimpl_pick_lemma : ∀ Hla1 Hla2 Hlw Hlt HR,
  Hla1 = Hla2 →
  Xsimpl (Hla2, Hlw, Hlt) HR →
  Xsimpl (Hla1, Hlw, Hlt) HR.
Proof using. introv M. subst⊃. Qed.

```

```

Ltac xsimpl_pick i :=
  let L := hstars_pick_lemma i in
  eapply xsimpl_pick_lemma; [ apply L | ].

```

xsimpl_pick_st *f* applies to a goal of the form $\text{Xsimpl } ((H1 \setminus^* \dots \setminus^* Hi \setminus^* \dots \setminus^* Hn), Hlw, Hlt) \text{ } HR$ and turns it into $\text{Xsimpl } ((Hi \setminus^* H1 \dots \setminus^* H\{i-1\} \setminus^* H\{i+1\} \setminus^* \dots \setminus^* Hn), Hlw, Hlt) \text{ } HR$ for the first *i* such that *f* *Hi* returns **true**.

```

Ltac xsimpl_pick_st f :=
  match goal with ⊢ Xsimpl (?Hla, ?Hlw, ?Hlt) ?HR ⇒
    hstars_search Hla ltac:(fun i H ⇒
      match f H with true ⇒ xsimpl_pick i end)
  end.

```

xsimpl_pick_syntactically *H* is a variant of the above that only checks for syntactic equality, not unifiability.

```

Ltac xsimpl_pick_syntactically H :=
  xsimpl_pick_st ltac:(fun H' ⇒

```

```
match  $H'$  with  $H \Rightarrow \text{constr}:(\text{true}) \text{ end}$ ).
```

xsimpl_pick_unifiable H applies to a goal of the form $\text{Xsimpl } (Hla, Hlw, Hlt) HR$, where Hla is of the form $H1 \setminus * .. \setminus * Hn \setminus * \setminus []$. It searches for H among the Hi . If it finds it, it moves this Hi to the front, just before $H1$. Else, it fails.

```
Ltac xsimpl_pick_unifiable  $H$  :=
  match goal with  $\vdash \text{Xsimpl } (?Hla, ?Hlw, ?Hlt) ?HR \Rightarrow$ 
    hstars_search  $Hla$  ltac:(fun  $i$   $H' \Rightarrow$ 
      unify  $H$   $H'$ ; xsimpl_pick  $i$ )
  end.
```

xsimpl_pick_same H is a choice for one of the above two, it is the default version used by *xsimpl*. Syntactic matching is faster but less expressive.

```
Ltac xsimpl_pick_same  $H$  :=
  xsimpl_pick_unifiable  $H$ .
```

xsimpl_pick_applied Q applies to a goal of the form $\text{Xsimpl } (Hla, Hlw, Hlt) HR$, where Hla is of the form $H1 \setminus * .. \setminus * Hn \setminus * \setminus []$. It searches for Q $?x$ among the Hi . If it finds it, it moves this Hi to the front, just before $H1$. Else, it fails.

```
Ltac xsimpl_pick_applied  $Q$  :=
  xsimpl_pick_st ltac:(fun  $H' \Rightarrow$ 
    match  $H'$  with  $Q \_ \Rightarrow \text{constr}:(\text{true}) \text{ end}$ ).
```

repr_get_predicate H applies to a H of the form $p \sim > R \_ \dots \_$ and it returns R .

```
Ltac repr_get_predicate  $H$  :=
  match  $H$  with  $?p \sim > ?E \Rightarrow \text{get\_head } E \text{ end}$ .
```

xsimpl_pick_repr H applies to a goal of the form $\text{Xsimpl } (Hla, Hlw, Hlt) HR$, where Hla is of the form $H1 \setminus * .. \setminus * Hn \setminus * \setminus []$, and where H is of the form $p \sim > R \_$ (same as *repr* p). It searches for $p \sim > R \_$ among the Hi . If it finds it, it moves this Hi to the front, just before $H1$. Else, it fails.

```
Ltac xsimpl_pick_repr  $H$  :=
  match  $H$  with  $?p \sim > ?E \Rightarrow$ 
    let  $R := \text{get\_head } E$  in
    xsimpl_pick_st ltac:(fun  $H' \Rightarrow$ 
      match  $H'$  with  $(p \sim > ?E') \Rightarrow$ 
        let  $R' := \text{get\_head } E'$  in
        match  $R'$  with  $R \Rightarrow \text{constr}:(\text{true}) \text{ end end}$ )
  end.
```

Tactic start and stop

Opaque Xsimpl .

```
Ltac xsimpl_handle_qimpl  $tt$  :=
```

```

match goal with
| ⊢ @qimpl _ _ ?Q2 ⇒ is_evar Q2; apply qimpl_refl
| ⊢ @qimpl unit ?Q1 ?Q2 ⇒ let t := fresh "_tt" in intros t; destruct t
| ⊢ @qimpl _ _ _ ⇒ let r := fresh "r" in intros r
| ⊢ himpl _ ?H2 ⇒ is_evar H2; apply himpl_refl
| ⊢ himpl _ _ ⇒ idtac
| ⊢ @eq hprop _ _ ⇒ applies himpl_antisym
| ⊢ @eq (_ → hprop) _ _ ⇒ applies fun_ext_1; applies himpl_antisym
| _ ⇒ fail 1 "not a goal for xsimpl/xpull"
end.

Ltac xsimpl_intro tt :=
  applies xsimpl_start.

Ltac xpull_start tt :=
  pose ltac_mark;
  intros;
  xsimpl_handle_qimpl tt;
  applies xpull_protect;
  xsimpl_intro tt.

Ltac xsimpl_start tt :=
  pose ltac_mark;
  intros;
  xsimpl_handle_qimpl tt;
  xsimpl_intro tt.

Ltac xsimpl_post_before_generalize tt :=
  idtac.

Ltac xsimpl_post_after_generalize tt :=
  idtac.

Ltac himpl_post_processing_for_hyp H :=
  idtac.

Ltac xsimpl_handle_false_subgoals tt :=
  tryfalse.

Ltac xsimpl_clean tt :=
  try remove_empty_heaps_right tt;
  try remove_empty_heaps_left tt;
  try xsimpl_hint_remove tt.

Ltac gen_until_mark_with_processing_and_cleaning cont :=
  match goal with H: ?T ⊢ _ ⇒
  match T with
  | ltac_Mark ⇒ clear H
  | _ ⇒ cont H;

```



```

    let  $T$  := type of  $H$  in
    generalize  $H$ ; clear  $H$ ;

    try (match goal with  $\vdash \_ \rightarrow \_ \Rightarrow$ 
        match type of  $T$  with
        | Prop  $\Rightarrow$  idtac
        |  $\_ \Rightarrow$  intros  $\_$ 
        end end);
    gen_until_mark_with_processing cont
end end.

Ltac xsimpl_generalize tt :=
  xsimpl_post_before_generalize tt;
  xsimpl_handle_false_subgoals tt;
  gen_until_mark_with_processing_and_cleaning
    ltac:(himpl_post_processing_for_hyp);
  xsimpl_post_after_generalize tt.

Ltac xsimpl_post tt :=
  xsimpl_clean tt;
  xsimpl_generalize tt.

Ltac xpull_post tt :=
  xsimpl_clean tt;
  unfold protect;
  xsimpl_generalize tt.

```

Auxiliary functions step

xsimpl_lr_cancel_eq_repr_post tt is meant to simplify goals of the form $E1 = E2$ that arises from cancelling $p \sim > E1$ with $p \sim > E2$ in the case where $E1$ and $E2$ share the same head symbol, that is, the same representation predicate R .

```

Ltac xsimpl_lr_cancel_eq_repr_post tt :=
  try fequal; try reflexivity.

```

xsimpl_r_hexists_apply tt is a tactic to apply *xsimpl_r_hexists* by exploiting a hint if one is available (see the hint section above) to specify the instantiation of the existential.

```

Ltac xsimpl_r_hexists_apply tt :=
  first [
    xsimpl_hint_next ltac:(fun x  $\Rightarrow$ 
      match  $x$  with
      |  $-- \Rightarrow$  nrapply xsimpl_r_hexists
      |  $\_ \Rightarrow$  apply (@xsimpl_r_hexists  $\_$   $x$ )
      end)
    | nrapply xsimpl_r_hexists ].

```

xsimpl_hook H can be customize to handle cancellation of specific kind of heap predicates (e.g., *hsingle*).

Ltac xsimpl_hook H := fail.

Tactic step

```
Ltac xsimpl_hwand_hstars_l tt :=
  match goal with ⊢ Xsimpl (?Hla, ((?H1s \-* ?H2) \* ?Hlw), \[]) ?HR =>
    hstars_search H1s ltac:(fun i H =>
      let L := hstars_pick_lemma i in
      eapply xsimpl_l_hwand_reorder;
      [ apply L
      | match H with
        | \[] => apply xsimpl_l_cancel_hwand_hstar_empty
        | _ => xsimpl_pick_same H; apply xsimpl_l_cancel_hwand_hstar
        end
      ])
  end.
```

Ltac xsimpl_step_l cancel_wands :=

```
let cancel_wands := match cancel_wands with tt => constr:(true) | ?x => x end in
match goal with ⊢ Xsimpl ?HL ?HR =>
  match HL with
  | (?Hla, ?Hlw, (?H \* ?Hlt)) =>
    match H with
    | \[] => eapply xsimpl_l_empty
    | \[?P] => eapply xsimpl_l_hpure; intro
    | ?H1 \* ?H2 => rewrite (@hstar_assoc H1 H2)
    | hexists ?J => eapply xsimpl_l_hexists; intro
    | ?H1 \-* ?H2 => eapply xsimpl_l_acc_wand
    | ?Q1 \-* ?Q2 => eapply xsimpl_l_acc_wand
    | _ => eapply xsimpl_l_acc_other
    end
  | (?Hla, ((?H1 \-* ?H2) \* ?Hlw), \[]) =>
    match H1 with
    | \[] => apply xsimpl_l_cancel_hwand_empty
    | (_ \* _) => xsimpl_hwand_hstars_l tt
    | _ =>
      match cancel_wands with
      | true => xsimpl_pick_same H1; eapply xsimpl_l_cancel_hwand
      | _ => eapply xsimpl_l_keep_wand
      end
    end
```

```

    end
  | (?Hla, ((?Q1 \-* ?Q2) \* ?Hlw), \[]) =>
    match cancel_wands with
    | true => xsimpl_pick_applied Q1; eapply xsimpl_l_cancel_qwand
    | _ => eapply xsimpl_l_keep_wand
    end
end end.

Ltac xsimpl_hgc_or_htop_cancel cancel_item cancel_lemma :=

  repeat (xsimpl_pick_same cancel_item; apply cancel_lemma).

Ltac xsimpl_hgc_or_htop_step tt :=
  match goal with ⊢ Xsimpl (?Hla, \[], \[]) (?Hra, ?Hrg, (?H \* ?Hrt)) =>
  match constr:((Hrg, H)) with
  | (\[], \GC) => applys xsimpl_r_hgc_or_htop;
    xsimpl_hgc_or_htop_cancel (\GC) xsimpl_lr_cancel_hgc
  | (\[], \Top) => applys xsimpl_r_hgc_or_htop;
    xsimpl_hgc_or_htop_cancel (\GC) xsimpl_lr_cancel_htop;
    xsimpl_hgc_or_htop_cancel (\Top) xsimpl_lr_cancel_htop
  | (\GC \* \[], \Top) => applys xsimpl_r_htop_replace_hgc;
    xsimpl_hgc_or_htop_cancel (\Top) xsimpl_lr_cancel_htop
  | (\GC \* \[], \GC) => applys xsimpl_r_hgc_drop
  | (\Top \* \[], \GC) => applys xsimpl_r_hgc_drop
  | (\Top \* \[], \Top) => applys xsimpl_r_htop_drop
  end end.

Ltac xsimpl_cancel_same H :=
  xsimpl_pick_same H; apply xsimpl_lr_cancel_same.

Ltac xsimpl_step_r tt :=
  match goal with ⊢ Xsimpl (?Hla, \[], \[]) (?Hra, ?Hrg, (?H \* ?Hrt)) =>
  match H with
  | ?H' => xsimpl_hook H
  | \[] => apply xsimpl_r_hempty
  | \[?P] => apply xsimpl_r_hpure
  | ?H1 \* ?H2 => rewrite (@hstar_assoc H1 H2)
  | ?H \-* ?H'eqH =>
    match H with
    | \[?P] => fail 1
    | _ =>
      match H'eqH with
      | H => eapply xsimpl_r_hwand_same
      end
    end
  end
end

```

```

    end
  | hexists ?J ⇒ xsimpl_r_hexists_apply tt
  | \GC ⇒ xsimpl_hgc_or_hstop_step tt
  | \Top ⇒ xsimpl_hgc_or_hstop_step tt
  | protect ?H' ⇒ eapply xsimpl_r_keep
  | protect ?Q' _ ⇒ eapply xsimpl_r_keep
  | ?H' ⇒ is_not_evar H; xsimpl_cancel_same H
  | ?p ~> _ ⇒ xsimpl_pick_repr H; eapply xsimpl_lr_cancel_eq_repr;
    [ xsimpl_lr_cancel_eq_repr_post tt | ]
  | ?x ~> Id ?X ⇒ has_no_evar x; eapply xsimpl_r_id
  | ?x ~> ?T_evar ?X_evar ⇒ has_no_evar x; is_evar T_evar; is_evar X_evar;
    eapply xsimpl_r_id_unify
  | _ ⇒ eapply xsimpl_r_keep
end end.

Ltac xsimpl_step_lr tt :=
  match goal with ⊢ Xsimpl (?Hla, \[], \[]) (?Hra, ?Hrg, \[]) ⇒
  match Hrg with
  | \[] ⇒
    match Hra with
    | ?H1 \* \[] ⇒
      match H1 with
      | ?Hra_evar ⇒ is_evar Hra_evar; rew_heap; eapply himpl_lr_refl
      | ?Q1 \-* ?Q2 ⇒ is_evar Q2; eapply himpl_lr_qwand_unify
      | \[False] \-* ?H2 ⇒ apply xsimpl_lr_hwand_hfalse
      | ?H1 \-* ?H2 ⇒ xsimpl_flip_acc_l tt; apply xsimpl_lr_hwand
      | ?Q1 \-* ?Q2 ⇒
        xsimpl_flip_acc_l tt;
        match H1 with
        | @qwand unit ?Q1' ?Q2' ⇒ eapply xsimpl_lr_qwand_unit
        | _ ⇒ eapply xsimpl_lr_qwand; intro
        end
      | hforall _ ⇒ xsimpl_flip_acc_l tt; apply xsimpl_lr_hforall; intro
      end
    | \[] ⇒ eapply himpl_lr_refl
    | _ ⇒ xsimpl_flip_acc_lr tt; eapply xsimpl_lr_exit_nogc
    end
  | (\Top \* _) ⇒ eapply himpl_lr_hstop
  | (\GC \* _) ⇒ eapply himpl_lr_hgc;
    [ try remove_empty_heaps_haffine tt; xaffine | ]
  | ?Hrg' ⇒ xsimpl_flip_acc_lr tt; eapply xsimpl_lr_exit
end end.

Ltac xsimpl_step cancel_wands :=

```

```

first [ xsimpl_step_l cancel_wands
      | xsimpl_step_r tt
      | xsimpl_step_lr tt ].

```

Tactic Notation

```

Ltac xpull_core tt :=
  xpull_start tt;
  repeat (xsimpl_step tt);
  xpull_post tt.

Tactic Notation "xpull" := xpull_core tt.
Tactic Notation "xpull" "~" := xpull; auto_tilde.
Tactic Notation "xpull" "*" := xpull; auto_star.

Ltac xsimpl_core_mode cancel_wands :=
  xsimpl_start tt;
  repeat (xsimpl_step cancel_wands);
  xsimpl_post tt.

Ltac xsimpl_core tt :=
  xsimpl_core_mode constr:(true).

Ltac xsimpl_no_cancel_wand tt :=
  xsimpl_core_mode constr:(false).

Tactic Notation "xsimpl" := xsimpl_core tt.
Tactic Notation "xsimpl" "~" := xsimpl; auto_tilde.
Tactic Notation "xsimpl" "*" := xsimpl; auto_star.

Tactic Notation "xsimpl" constr(L) :=
  match type of L with
  | list Boxer => xsimpl_hint_put L
  | _ => xsimpl_hint_put (boxer L :: nil)
  end; xsimpl.

Tactic Notation "xsimpl" constr(X1) constr(X2) :=
  xsimpl (> X1 X2).

Tactic Notation "xsimpl" constr(X1) constr(X2) constr(X3) :=
  xsimpl (> X1 X2 X3).
  Tactic Notation "xsimpl" constr(X1) constr(X2) constr(X3) constr(X4) :=
  xsimpl (> X1 X2 X3 X4).

Tactic Notation "xsimpl" constr(X1) constr(X2) constr(X3) constr(X4) constr(X5)
:=
  xsimpl (> X1 X2 X3 X4 X5).

Tactic Notation "xsimpl" constr(X1) constr(X2) constr(X3) constr(X4) constr(X5)
constr(X6) :=
  xsimpl (> X1 X2 X3 X4 X5 X6).

```

```

Tactic Notation "xsimpl" "~" constr(L) :=
  xsimpl L; auto_tilde.
Tactic Notation "xsimpl" "~" constr(X1) constr(X2) :=
  xsimpl X1 X2; auto_tilde.
Tactic Notation "xsimpl" "~" constr(X1) constr(X2) constr(X3) :=
  xsimpl X1 X2 X3; auto_tilde.
Tactic Notation "xsimpl" "~" constr(X1) constr(X2) constr(X3) constr(X4) :=
  xsimpl X1 X2 X3 X4; auto_tilde.
Tactic Notation "xsimpl" "~" constr(X1) constr(X2) constr(X3) constr(X4) constr(X5)
:=
  xsimpl X1 X2 X3 X4 X5; auto_tilde.
Tactic Notation "xsimpl" "~" constr(X1) constr(X2) constr(X3) constr(X4) constr(X5)
constr(X6) :=
  xsimpl X1 X2 X3 X4 X5 X6; auto_tilde.
Tactic Notation "xsimpl" "*" constr(L) :=
  xsimpl L; auto_star.
Tactic Notation "xsimpl" "*" constr(X1) constr(X2) :=
  xsimpl X1 X2; auto_star.
Tactic Notation "xsimpl" "*" constr(X1) constr(X2) constr(X3) :=
  xsimpl X1 X2 X3; auto_star.
Tactic Notation "xsimpl" "*" constr(X1) constr(X2) constr(X3) constr(X4) :=
  xsimpl X1 X2 X3 X4; auto_star.
Tactic Notation "xsimpl" "*" constr(X1) constr(X2) constr(X3) constr(X4) constr(X5)
:=
  xsimpl X1 X2 X3 X4 X5; auto_star.
Tactic Notation "xsimpl" "*" constr(X1) constr(X2) constr(X3) constr(X4) constr(X5)
constr(X6) :=
  xsimpl X1 X2 X3 X4 X5 X6; auto_star.

```

Tactic *xchange*

xchange performs rewriting on the LHS of an entailment. Essentially, it applies to a goal of the form $H1 \setminus^* H2 \implies H3$, and exploits an entailment $H1 \implies H1'$ to replace the goal with $H1' \setminus^* H2 \implies H3$.

The tactic is actually a bit more flexible in two respects:

- it does not force the LHS to be exactly of the form $H1 \setminus^* H2$
- it takes as argument any lemma, whose instantiation result in a heap entailment of the form $H1 \implies H1'$.

Concretely, the tactic is just a wrapper around an application of the lemma called `xchange_lemma`, which appears below.

xchanges combines a call to *xchange* with calls to *xsimpl* on the subgoals.

```
Lemma xchange_lemma : ∀ H1 H2 H3 H4,
  H1 ==> H2 →
  H3 ==> H1 \* (H2 \-* protect H4) →
  H3 ==> H4.
```

Proof using.

```
  introv M1 M2. applys himpl_trans (rm M2).
  applys himpl_hstar_trans_l (rm M1). applys hwand_cancel.
```

Qed.

```
Ltac xchange_apply L :=
  eapply xchange_lemma; [ eapply L | ].
```

```
Ltac xchange_build_entailment modifier K :=
  match modifier with
  | -- =>
    match type of K with
    | _ = _ => constr:(@himpl_of_eq _ _ K)
    | _ => constr:(K)
    end
  | _ => constr:(@modifier _ _ K)
  end.
```

```
Ltac xchange_perform L modifier cont :=
  forwards_nounfold_then L ltac:(fun K =>
    let X := fresh "TEMP" in
    set (X := K);
    let M := xchange_build_entailment modifier K in
    clear X;
    xchange_apply M;
    cont tt).
```

```
Ltac xchange_core L modifier cont :=
  pose ltac_mark;
  intros;
  match goal with
  | ⊢ _ ==> _ => idtac
  | ⊢ _ ==> _ => let x := fresh "r" in intros x
  end;
  xchange_perform L modifier cont;
  gen_until_mark.
```

Error reporting support for *xchange* (not for *xchanges*)

Definition *xchange_hidden* (*P*:Type) (*e*:*P*) := *e*.

Notation "'__XCHANGE_FAILED_TO_MATCH_PRECONDITION__'" :=

```

(@xchange_hidden _ _).
Ltac xchange_report_error tt :=
  match goal with ⊢ context [?H1 \-* protect ?H2] ⇒
    change (H1 \-* protect H2) with (@xchange_hidden _ (H1 \-* protect H2)) end.
Ltac xchange_xpull_cont tt :=
  xsimpl; first
  [ xchange_report_error tt
  | unfold protect; try solve [ apply himpl_refl ] ].
Ltac xchange_xpull_cont_basic tt :=
  xsimpl; unfold protect; try solve [ apply himpl_refl ].
Ltac xchange_xsimpl_cont tt :=
  unfold protect; xsimpl; try solve [ apply himpl_refl ].
Ltac xchange_nosimpl_base E modifier :=
  xchange_core E modifier ltac:(idcont).
Tactic Notation "xchange_nosimpl" constr(E) :=
  xchange_nosimpl_base E ___.
Tactic Notation "xchange_nosimpl" "->" constr(E) :=
  xchange_nosimpl_base E himpl_of_eq.
Tactic Notation "xchange_nosimpl" "<-" constr(E) :=
  xchange_nosimpl_base himpl_of_eq_sym.
Ltac xchange_base E modif :=
  xchange_core E modif ltac:(xchange_xpull_cont).
Tactic Notation "xchange" constr(E) :=
  xchange_base E ___.
Tactic Notation "xchange" "~" constr(E) :=
  xchange E; auto_tilde.
Tactic Notation "xchange" "*" constr(E) :=
  xchange E; auto_star.
Tactic Notation "xchange" "->" constr(E) :=
  xchange_base E himpl_of_eq.
Tactic Notation "xchange" "~" "->" constr(E) :=
  xchange → E; auto_tilde.
Tactic Notation "xchange" "*" "->" constr(E) :=
  xchange → E; auto_star.
Tactic Notation "xchange" "<-" constr(E) :=
  xchange_base E himpl_of_eq_sym.
Tactic Notation "xchange" "~" "<-" constr(E) :=
  xchange ← E; auto_tilde.
Tactic Notation "xchange" "*" "<-" constr(E) :=
  xchange ← E; auto_star.

```



```

Ltac xchanges_base E modif :=
  xchange_core E modif ltac:(xchange_xsimpl_cont).

Tactic Notation "xchanges" constr(E) :=
  xchanges_base E --.
Tactic Notation "xchanges" "~" constr(E) :=
  xchanges E; auto_tilde.
Tactic Notation "xchanges" "*" constr(E) :=
  xchanges E; auto_star.

Tactic Notation "xchanges" "->" constr(E) :=
  xchanges_base E himpl_of_eq.
Tactic Notation "xchanges" "~" "->" constr(E) :=
  xchanges → E; auto_tilde.
Tactic Notation "xchanges" "*" "->" constr(E) :=
  xchanges → E; auto_star.

Tactic Notation "xchanges" "<-" constr(E) :=
  xchanges_base E himpl_of_eq_sym.
Tactic Notation "xchanges" "~" "<-" constr(E) :=
  xchanges ← E; auto_tilde.
Tactic Notation "xchanges" "*" "<-" constr(E) :=
  xchanges ← E; auto_star.

Tactic Notation "xchange" constr(E1) "," constr(E2) :=
  xchange E1; try xchange E2.
Tactic Notation "xchange" constr(E1) "," constr(E2) "," constr(E3) :=
  xchange E1; try xchange E2; try xchange E3.
Tactic Notation "xchange" constr(E1) "," constr(E2) "," constr(E3) "," constr(E4)
:=
  xchange E1; try xchange E2; try xchange E3; try xchange E4.

```

33.5 Demos

rew_heap demos

Lemma *rew_heap_demo_with_evar* : $\forall H1\ H2\ H3,$
 $(\forall H, H1 \setminus * (H \setminus * H2) \setminus * \setminus [] = H3 \rightarrow \text{True}) \rightarrow \text{True}.$

Proof using.

```

intro M. dup 3.
{ eapply M.rewrite hstar_assoc. rewrite hstar_assoc. demo. }
{ eapply M.rew_heap_assoc. demo. }
{ eapply M.rew_heap. demo. }

```

Abort.

hstars demos

Lemma *hstars_flip_demo* : $\forall H1\ H2\ H3\ H4,$
 $(\forall H, H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * \square = H \rightarrow H = H \rightarrow \text{True}) \rightarrow \text{True}.$

Proof using.

intros M. eapply M. hstars_flip tt.

Abort.

Lemma *hstars_flip_demo_0* :

$(\forall H, \square = H \rightarrow H = H \rightarrow \text{True}) \rightarrow \text{True}.$

Proof using.

intros M. eapply M. hstars_flip tt.

Abort.

xsimpl_hint demos

Lemma *xsimpl_demo_hints* : $\exists n, n = 3.$

Proof using.

xsimpl_hint_put ($\gg 3$ true).

xsimpl_hint_next ltac:(fun x $\Rightarrow \exists x$).

xsimpl_hint_remove tt.

Abort.

hstars_pick demos

Lemma *demo_hstars_pick_1* : $\forall H1\ H2\ H3\ H4\ Hresult,$
 $(\forall H, H1 \setminus * H2 \setminus * H3 \setminus * H4 = H \rightarrow H = Hresult \rightarrow \text{True}) \rightarrow \text{True}.$

Proof using.

intros M. dup 2.

{ *eapply M. let L := hstars_pick_lemma 3 in eapply L. demo.* }

{ *eapply M. let L := hstars_pick_lemma (hstars_last 4) in eapply L. demo.* }

Qed.

hstars_simpl demos

Lemma *demo_hstars_simpl_1* : $\forall H1\ H2\ H3\ H4\ H5,$
 $H2 \Rightarrow H5 \rightarrow$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \Rightarrow H4 \setminus * H5 \setminus * H3 \setminus * H1.$

Proof using.

intros. dup.

{ *hstars_simpl_start tt.*

hstars_simpl_step tt.

hstars_simpl_step tt.

```

    hstars_simpl_step tt.
    hstars_simpl_step tt.
    hstars_simpl_post tt. auto. }
  { hstars_simpl. auto. }
Qed.

Lemma demo_hstars_simpl_2 : ∀ H1 H2 H3 H4 H5,
  (∀ H, H \* H2 \* H3 \* H4 ==> H4 \* H5 \* H3 \* H1 → True) → True.
Proof using.
  introv M. eapply M. hstars_simpl.
Abort.

```

*x*simpl_pick demos

```

Lemma xsimpl_pick_demo : ∀ (Q:bool→hprop) (P:Prop) H1 H2 H3 Hlw Hlt Hra Hrg Hrt,
  (∀ HX HY,
    Xsimpl ((H1 \* H2 \* H3 \* Q true \* (¬[P] \* HX) \* HY \* ¬[]), Hlw, Hlt)
      (Hra, Hrg, Hrt)
    → True) → True.
Proof using.
  introv M. applys (rm M).
  let L := hstars_pick_lemma 2%nat in set (X:=L).
  eapply xsimpl_pick_lemma. apply X.
  xsimpl_pick 2%nat.
  xsimpl_pick_same H3.
  xsimpl_pick_applied Q.
  xsimpl_pick_same H2.
  xsimpl_pick_unifiable H3.
  xsimpl_pick_unifiable ¬[True].
  xsimpl_pick_unifiable (¬[P] \* H1).
Abort.

```

*x*pull and *x*simpl demos

```

Tactic Notation "xpull0" := xpull_start tt.
Tactic Notation "xsimpl0" := xsimpl_start tt.
Tactic Notation "xsimpl1" := xsimpl_step tt.
Tactic Notation "xsimpl2" := xsimpl_post tt.
Tactic Notation "xsimpll" := xsimpl_step_l tt.
Tactic Notation "xsimplr" := xsimpl_step_r tt.
Tactic Notation "xsimpllr" := xsimpl_step_lr tt.

Declare Scope xsimpl_scope.

```

Notation "'HSIMPL' Hla Hlw Hlt =====> Hra Hrg Hrt" := (Xsimpl (Hla, Hlw, Hlt)
 (Hra, Hrg, Hrt))
 (at level 69, Hla, Hlw, Hlt, Hra, Hrg, Hrt at level 0,
 format "'[v' 'HSIMPL' '/' Hla '/' Hlw '/' Hlt '/' =====> '/' Hra '/' Hrg '/' Hrt ']'")
 : xsimpl_scope.

Local Open Scope xsimpl_scope.

Lemma xpull_demo : $\forall H1\ H2\ H3\ H,$
 $(H1 \setminus * \setminus [] \setminus * (H2 \setminus * \setminus \exists (y:\text{int})\ z\ (n:\text{nat}), \setminus [y = y + z + n]) \setminus * H3) \implies H.$

Proof using.

dup.
 { intros. xpull0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
 xsimpl1. xsimpl1. xsimpl2. demo. }
 { xpull. intros. demo. }

Abort.

Lemma xsimpl_demo_stars : $\forall H1\ H2\ H3\ H4\ H5,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \implies H4 \setminus * H3 \setminus * H5 \setminus * H2.$

Proof using.

dup 3.
 { xpull. demo. }
 { intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
 xsimpl1. xsimpl1. xsimpl1. xsimpl1. demo. }
 { intros. xsimpl. demo. }

Abort.

Lemma xsimpl_demo_keep_order : $\forall H1\ H2\ H3\ H4\ H5\ H6\ H7,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \implies H5 \setminus * H3 \setminus * H6 \setminus * H7.$

Proof using. intros. xsimpl. demo. Abort.

Lemma xsimpl_demo_stars_top : $\forall H1\ H2\ H3\ H4\ H5,$
 $H1 \setminus * H2 \setminus * H3 \setminus * H4 \setminus * H5 \implies H3 \setminus * H1 \setminus * H2 \setminus * \text{Top}.$

Proof using.

dup.
 { intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
 xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
 xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
 { intros. xsimpl. }

Abort.

Lemma xsimpl_demo_hint : $\forall H1\ (Q:\text{int} \rightarrow \text{hprop}),$
 $Q\ 4 \implies Q\ 3 \rightarrow$
 $H1 \setminus * Q\ 4 \implies \setminus \exists x, Q\ x \setminus * H1.$

Proof using.

introv W. dup.
 { intros. xsimpl_hint_put ($\gg 3$).

```

    xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl2. auto. }
  { xsimpl 3. auto. }
Qed.

Lemma xsimpl_demo_stars_gc :  $\forall H1 H2$ ,
  haffine  $H2 \rightarrow$ 
   $H1 \setminus * H2 \implies H1 \setminus * \setminus GC$ .
Proof using.
  dup.
  { intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. }
  { intros. xsimpl $\neg$ . }
Abort.

Lemma xsimpl_demo_evar_1 :  $\forall H1 H2$ ,
  ( $\forall H, H1 \setminus * H2 \implies H \rightarrow \mathbf{True}$ )  $\rightarrow \mathbf{True}$ .
Proof using. intros. eapply  $H$ . xsimpl. Abort.

Lemma xsimpl_demo_evar_2 :  $\forall H1$ ,
  ( $\forall H, H1 \implies H1 \setminus * H \rightarrow \mathbf{True}$ )  $\rightarrow \mathbf{True}$ .
Proof using.
  introv  $M$ . dup.
  { eapply  $M$ . xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
  { eapply  $M$ . xsimpl $\neg$ . }
Abort.

Lemma xsimpl_demo_htop_both_sides :  $\forall H1 H2$ ,
   $H1 \setminus * H2 \setminus * \setminus \mathbf{Top} \implies H1 \setminus * \setminus \mathbf{Top}$ .
Proof using.
  dup.
  { intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
  { intros. xsimpl $\neg$ . }
Abort.

Lemma xsimpl_demo_htop_multiple :  $\forall H1 H2$ ,
   $H1 \setminus * H2 \setminus * \setminus \mathbf{Top} \implies H1 \setminus * \setminus \mathbf{Top} \setminus * \setminus \mathbf{Top}$ .
Proof using. intros. xsimpl $\neg$ . Abort.

Lemma xsimpl_demo_hgc_multiple :  $\forall H1 H2$ ,
  haffine  $H2 \rightarrow$ 
   $H1 \setminus * H2 \setminus * \setminus GC \implies H1 \setminus * \setminus GC \setminus * \setminus GC$ .
Proof using. intros. xsimpl $\neg$ . Qed.

Lemma xsimpl_demo_hwand :  $\forall H1 H2 H3 H4$ ,
  ( $H1 \setminus - * (H2 \setminus - * H3)$ )  $\setminus * H1 \setminus * H4 \implies (H2 \setminus - * (H3 \setminus * H4))$ .
Proof using.

```

```

dup.
{ intros. xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
  xsimpl1. xsimpl1. xsimpl1.
  xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
  xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
  xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
{ intros. xsimpl¬. }
Qed.

Lemma xsimpl_demo_qwand : ∀ A (x:A) (Q1 Q2:A→hprop) H1,
  H1 \* (H1 \-* (Q1 \-* Q2)) \* (Q1 x) ==> (Q2 x).
Proof using. intros. xsimpl¬. Qed.

Lemma xsimpl_demo_hwand_r : ∀ H1 H2 H3,
  H1 \* H2 ==> H1 \* (H3 \-* (H2 \* H3)).
Proof using. intros. xsimpl¬. Qed.

Lemma xsimpl_demo_qwand_r : ∀ A (x:A) (Q1 Q2:A→hprop) H1 H2,
  H1 \* H2 ==> H1 \* (Q1 \-* (Q1 \*+ H2)).
Proof using. intros. xsimpl. Qed.

Lemma xsimpl_demo_hwand_multiple_1 : ∀ H1 H2 H3 H4 H5,
  H1 \-* H4 ==> H5 →
  (H2 \* ((H1 \* H2 \* H3) \-* H4)) \* H3 ==> H5.
Proof using. introv M. xsimpl. auto. Qed.

Lemma xsimpl_demo_hwand_multiple_2 : ∀ H1 H2 H3 H4 H5,
  (H1 \* H2 \* ((H1 \* H3) \-* (H4 \-* H5))) \* (H2 \-* H3) \* H4 ==> H5.
Proof using. intros. xsimpl. Qed.

Lemma xsimpl_demo_hwand_hempty : ∀ H1 H2 H3,
  (\[] \-* H1) \* H2 ==> H3.
Proof using. intros. xsimpl. Abort.

Lemma xsimpl_demo_hwand_hstar_hempty : ∀ H0 H1 H2 H3,
  ((H0 \* \[]) \-* \[] \-* H1) \* H2 ==> H3.
Proof using. intros. xsimpl. rew_heap. Abort.

Lemma xsimpl_demo_hwand_iter : ∀ H1 H2 H3 H4 H5,
  H1 \* H2 \* ((H1 \* H3) \-* (H4 \-* H5)) \* H4 ==> ((H2 \-* H3) \-* H5).
Proof using.
  intros. dup.
  { xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
    xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
  { xsimpl. }

```

Qed.

Lemma xsimpl_demo_repr_1 : $\forall p\ q\ (R:\text{int} \rightarrow \text{int} \rightarrow \text{hprop}),$
 $p \sim R\ 3 \wedge q \sim R\ 4 \implies \exists n\ m, p \sim R\ n \wedge q \sim R\ m.$

Proof using.

```
intros. dup.
{ xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
  xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
{ xsimpl¬. }
```

Qed.

Lemma xsimpl_demo_repr_2 : $\forall p\ (R\ R':\text{int} \rightarrow \text{int} \rightarrow \text{hprop}),$
 $R = R' \rightarrow$
 $p \sim R'\ 3 \implies \exists n, p \sim R\ n.$

Proof using. *introv E. xsimpl. subst R'. xsimpl. Qed.*

Lemma xsimpl_demo_repr_3 : $\forall p\ (R:\text{int} \rightarrow \text{int} \rightarrow \text{hprop}),$
let $R' := R$ in
 $p \sim R'\ 3 \implies \exists n, p \sim R\ n.$

Proof using.

```
intros. dup.
{ xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. }
{ xsimpl¬. }
```

Qed.

Lemma xsimpl_demo_repr_4 : $\forall p\ n\ m\ (R:\text{int} \rightarrow \text{int} \rightarrow \text{hprop}),$
 $n = m + 0 \rightarrow$
 $p \sim R\ n \implies p \sim R\ m.$

Proof using. *intros. xsimpl. math. Qed.*

Lemma xsimpl_demo_gc_0 : $\forall H1\ H2,$
 $H1 \implies H2 \wedge \text{GC} \wedge \text{GC}.$

Proof using. *intros. xsimpl. Abort.*

Lemma xsimpl_demo_gc_1 : $\forall H1\ H2,$
 $H1 \implies H2 \wedge \text{GC} \wedge \text{Top} \wedge \text{Top} \wedge \text{GC}.$

Proof using.

```
intros. dup.
{ xsimpl0. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1. xsimpl1.
  xsimpl1. xsimpl1. xsimpl1. xsimpl2. demo. }
{ xsimpl¬. demo. }
```

Abort.

Lemma xsimpl_demo_gc_2 : $\forall H1\ H2\ H3,$
 $H1 \wedge H2 \wedge \text{Top} \wedge \text{GC} \wedge \text{Top} \implies H3 \wedge \text{GC} \wedge \text{GC}.$

Proof using. *intros. xsimpl. Abort.*

Lemma xsimpl_demo_gc_3 : $\forall H1\ H2,$

$H1 \setminus * H2 \setminus * \setminus GC \setminus * \setminus GC \implies H2 \setminus * \setminus GC \setminus * \setminus GC \setminus * \setminus GC.$

Proof using. intros. *xsimpl*. *xAffine*. Abort.

Lemma *xsimpl_demo_gc_4* : $\forall H1 H2,$

$H1 \setminus * H2 \setminus * \setminus GC \implies H2 \setminus * \setminus GC \setminus * \setminus Top \setminus * \setminus Top \setminus * \setminus GC.$

Proof using. intros. *xsimpl*. Abort.

xchange demos

Lemma *xchange_demo_1* : $\forall H1 H2 H3 H4 H5 H6,$

$H1 \implies H2 \setminus * H3 \rightarrow$

$H1 \setminus * H4 \implies (H5 \setminus - * H6).$

Proof using.

intros M. dup 3.

{ *xchange_nosimpl M. xsimpl. demo.* }

{ *xchange M. xsimpl. demo.* }

{ *xchanges M. demo.* }

Qed.

Lemma *xchange_demo_2* : $\forall A (Q:A \rightarrow \text{hprop}) H1 H2 H3,$

$H1 \implies \setminus \exists x, Q x \setminus * (H2 \setminus - * H3) \rightarrow$

$H1 \setminus * H2 \implies \setminus \exists x, Q x \setminus * H3.$

Proof using.

intros M. dup 3.

{ *xchange_nosimpl M. xsimpl. unfold protect. xsimpl.* }

{ *xchange M. xsimpl.* }

{ *xchanges M.* }

Qed.

Lemma *xchange_demo_4* : $\forall A (Q1 Q2:A \rightarrow \text{hprop}) H,$

$Q1 \implies Q2 \rightarrow$

$Q1 \setminus *+ H \implies Q2 \setminus *+ H.$

Proof using. *intros M. xchanges M.* Qed.

Lemma *xsimpl_demo_hfalse* : $\forall H1 H2,$

$H1 \implies \setminus [\text{False}] \rightarrow$

$H1 \setminus * H2 \implies \setminus [\text{False}].$

Proof using.

intros M. dup.

{ *xchange_nosimpl M. xsimpl0. xsimpl1. xsimpl1. xsimpl1.*

xsimpl1. xsimpl1. xsimpl1. xsimpl1. }

{ *xchange M.* }

Qed.

Lemma *xchange_demo_hforall_1* :

$\forall (\text{hforall\_specialize} : \forall A (x:A) (J:A \rightarrow \text{hprop}), (\text{hforall } J) \implies (J x)),$


```

  ∀ (Q:int→hprop) H1,
    (∀ x, Q x) \* H1 ==> Q 2 \* \Top.
Proof using.
  intros. xchange (» hforall_specialize 2). xsimpl.
Qed.
End XSIMPLSETUP.

```

Chapter 34

Library `SLF.LibSepMinimal`

34.1 LibSepMinimal: Appendix - Minimalistic Soundness Proof

This file contains a stand-alone, minimalistic formalization of the soundness of Separation Logic reasoning rules with respect to an omni-big-step semantics. For a development grounded with respect to a small-step semantics, see the two approaches described in chapter [Triples](#).

34.2 Source Language

34.2.1 Syntax

```
Set Implicit Arguments.
From SLF Require Export LibString LibCore.
From SLF Require Export LibSepTLCbuffer LibSepFmap.
Module FMAP := LIBSEPFMAP.
```

Variables are defined as strings, `var_eq` denotes boolean comparison.

```
Definition var : Type := string.
```

```
Definition var_eq := String.string_dec.
```

Locations are defined as natural numbers.

```
Definition loc : Type := nat.
```

Primitive operations include memory operations, as well as addition and division to illustrate a total and a partial arithmetic operations.

```
Inductive prim : Type :=
| val_ref : prim
| val_get : prim
```

```

| val_set : prim
| val_free : prim
| val_add : prim
| val_div : prim
| val_rand : prim.

```

The grammar of closed values (assumed to contain no free variables) includes values of ground types, primitive operations, and closures.

```

Inductive val : Type :=
| val_unit : val
| val_bool : bool → val
| val_int : int → val
| val_loc : loc → val
| val_prim : prim → val
| val_fix : var → var → trm → val

```

The grammar of terms includes closed values, variables, functions, applications, let-binding and conditional.

```

with trm : Type :=
| trm_val : val → trm
| trm_var : var → trm
| trm_fix : var → var → trm → trm
| trm_app : trm → trm → trm
| trm_let : var → trm → trm → trm
| trm_if : trm → trm → trm → trm.

```

Coercions are used to improve conciseness in the statment of evaluation rules.

```

Coercion val_bool : bool >-> val.
Coercion val_int : Z >-> val.
Coercion val_loc : loc >-> val.
Coercion val_prim : prim >-> val.
Coercion trm_val : val >-> trm.
Coercion trm_var : var >-> trm.
Coercion trm_app : trm >-> Funclass.

```

The type of values is inhabited (useful for finite map operations).

```

Global Instance Inhab_val : Inhab val.
Proof. apply (Inhab_of_val val_unit). Qed.

```

A heap, a.k.a. heap, consists of a finite map from location to values. Finite maps are formalized in the file *LibSepFmap.v*. (We purposely do not use TLC's finite map library to avoid complications with typeclasses.)

```

Definition heap : Type := fmap loc val.

```

Implicit types associate meta-variable with types.

```

Implicit Types  $f$  : var.
Implicit Types  $b$  : bool.
Implicit Types  $p$  : loc.
Implicit Types  $n$  : int.
Implicit Types  $v\ w\ r$  : val.
Implicit Types  $t$  : trm.
Implicit Types  $s\ h$  : heap.

```

34.2.2 Semantics

The standard capture-avoiding substitution is written `subst y w t`.

```

Fixpoint subst (y:var) (w:val) (t:trm) : trm :=
  let aux t := subst y w t in
  let if_y_eq x t1 t2 := if var_eq x y then t1 else t2 in
  match t with
  | trm_val v  $\Rightarrow$  trm_val v
  | trm_var x  $\Rightarrow$  if_y_eq x (trm_val w) t
  | trm_fix f x t1  $\Rightarrow$  trm_fix f x (if_y_eq f t1 (if_y_eq x t1 (aux t1)))
  | trm_app t1 t2  $\Rightarrow$  trm_app (aux t1) (aux t2)
  | trm_let x t1 t2  $\Rightarrow$  trm_let x (aux t1) (if_y_eq x t2 (aux t2))
  | trm_if t0 t1 t2  $\Rightarrow$  trm_if (aux t0) (aux t1) (aux t2)
  end.

```

34.2.3 Definition of Omni-Big-Step Semantics

```

Implicit Types  $Q$  : val  $\rightarrow$  heap  $\rightarrow$  Prop.
Inductive eval : heap  $\rightarrow$  trm  $\rightarrow$  (val  $\rightarrow$  heap  $\rightarrow$  Prop)  $\rightarrow$  Prop :=
  | eval_val :  $\forall s\ v\ Q$ ,
     $Q\ v\ s \rightarrow$ 
    eval s (trm_val v) Q
  | eval_fix :  $\forall s\ f\ x\ t1\ Q$ ,
     $Q\ (\text{val\_fix } f\ x\ t1)\ s \rightarrow$ 
    eval s (trm_fix f x t1) Q
  | eval_app_fix :  $\forall s\ v1\ v2\ f\ x\ t1\ Q$ ,
     $v1 = \text{val\_fix } f\ x\ t1 \rightarrow$ 
    eval s (subst x v2 (subst f v1 t1)) Q  $\rightarrow$ 
    eval s (trm_app v1 v2) Q
  | eval_let :  $\forall Q1\ s\ x\ t1\ t2\ Q$ ,
    eval s t1 Q1  $\rightarrow$ 
    ( $\forall v1\ s2, Q1\ v1\ s2 \rightarrow$  eval s2 (subst x v1 t2) Q)  $\rightarrow$ 
    eval s (trm_let x t1 t2) Q
  | eval_if :  $\forall s\ (b:\text{bool})\ t1\ t2\ Q$ ,

```

```

    eval s (if b then t1 else t2) Q →
    eval s (trm_if (val_bool b) t1 t2) Q
| eval_add : ∀ s n1 n2 Q,
  Q (val_int (n1 + n2)) s →
  eval s (val_add (val_int n1) (val_int n2)) Q
| eval_div : ∀ s n1 n2 Q,
  n2 ≠ 0 →
  Q (val_int (Z.quot n1 n2)) s →
  eval s (val_div (val_int n1) (val_int n2)) Q
| eval_rand : ∀ s n Q,
  n > 0 →
  (∀ n1, 0 ≤ n1 < n → Q n1 s) →
  eval s (val_rand (val_int n)) Q
| eval_ref : ∀ s v Q,
  (∀ p, ¬ Fmap.indom s p →
    Q (val_loc p) (Fmap.update s p v)) →
  eval s (val_ref v) Q
| eval_get : ∀ s p Q,
  Fmap.indom s p →
  Q (Fmap.read s p) s →
  eval s (val_get (val_loc p)) Q
| eval_set : ∀ s p v Q,
  Fmap.indom s p →
  Q val_unit (Fmap.update s p v) →
  eval s (val_set (val_loc p) v) Q
| eval_free : ∀ s p Q,
  Fmap.indom s p →
  Q val_unit (Fmap.remove s p) →
  eval s (val_free (val_loc p)) Q.

```

34.2.4 Automation for Heap Equality and Heap Disjointness

For goals asserting equalities between heaps, i.e., of the form $h1 = h2$, we set up automation so that it performs some tidying: substitution, removal of empty heaps, normalization with respect to associativity.

```

#[global] Hint Rewrite union_assoc union_empty_l union_empty_r : fmap.
#[global] Hint Extern 1 (_ = _ :> heap) ⇒ subst; autorewrite with fmap.

```

For goals asserting disjointness between heaps, i.e., of the form $Fmap.disjoint\ h1\ h2$, we set up automation to perform simplifications: substitution, exploit distributivity of the disjointness predicate over unions of heaps, and exploit disjointness with empty heaps. The tactic *jauto_set* used here comes from the TLC library; essentially, it destructs conjunctions and existentials.

```

#[global] Hint Resolve Fmap.disjoint_empty_l Fmap.disjoint_empty_r.
#[global] Hint Rewrite disjoint_union_eq_l disjoint_union_eq_r : disjoint.
#[global] Hint Extern 1 (Fmap.disjoint - _) =>
  subst; autorewrite with rew_disjoint in *; jauto_set.

```

34.3 Heap Predicates and Entailment

34.3.1 Extensionality Axioms

Extensionality axioms are essential to assert equalities between heap predicates of type `hprop`, and between postconditions, of type `val → hprop`.

```

Axiom functional_extensionality : ∀ A B (f g:A→B),
  (∀ x, f x = g x) →
  f = g.
Axiom propositional_extensionality : ∀ (P Q:Prop),
  (P ↔ Q) →
  P = Q.

```

34.3.2 Core Heap Predicates

The type of heap predicates is named `hprop`.

Definition `hprop := heap → Prop`.

We bind a few more meta-variables.

```

Implicit Types P : Prop.
Implicit Types H : hprop.

```

Core heap predicates, and their associated notations:

- `\[]` denotes the empty heap predicate
- `\[P]` denotes a pure fact
- `p ~~> v` denotes a singleton heap
- `H1 \* H2` denotes the separating conjunction
- `Q1 \*+ H2` denotes the separating conjunction extending a postcondition
- `\∃ x, H` denotes an existential
- `\∀ x, H` denotes a universal.

```

Definition hempty : hprop :=
  fun h => (h = Fmap.empty).

Definition hsingle (p:loc) (v:val) : hprop :=
  fun h => (h = Fmap.single p v).

Definition hstar (H1 H2 : hprop) : hprop :=
  fun h => ∃ h1 h2, H1 h1
    ∧ H2 h2
    ∧ Fmap.disjoint h1 h2
    ∧ h = Fmap.union h1 h2.

Definition hexists A (J:A→hprop) : hprop :=
  fun h => ∃ x, J x h.

Definition hforall (A : Type) (J : A → hprop) : hprop :=
  fun h => ∀ x, J x h.

Definition hpure (P:Prop) : hprop :=
  hexists (fun (p:P) => hempty).

Declare Scope hprop_scope.

Notation "[]" := (hempty)
  (at level 0) : hprop_scope.

Notation "[ P ]" := (hpure P)
  (at level 0, format "[ P ]") : hprop_scope.

Notation "p '~>' v" := (hsingle p v) (at level 32) : hprop_scope.

Notation "H1 '*' H2" := (hstar H1 H2)
  (at level 41, right associativity) : hprop_scope.

Notation "Q '*+ H" := (fun x => hstar (Q x) H)
  (at level 40) : hprop_scope.

Notation "'\exists' x1 .. xn , H" :=
  (hexists (fun x1 => .. (hexists (fun xn => H)) ..))
  (at level 39, x1 binder, H at level 50, right associativity,
   format "'[\exists' '/' x1 .. xn , '/' H ]'") : hprop_scope.

Notation "'\forall' x1 .. xn , H" :=
  (hforall (fun x1 => .. (hforall (fun xn => H)) ..))
  (at level 39, x1 binder, H at level 50, right associativity,
   format "'[\forall' '/' x1 .. xn , '/' H ]'") : hprop_scope.

```

34.3.3 Entailment

```

Declare Scope hprop_scope.
Open Scope hprop_scope.

```

Entailment for heap predicates, written $H1 \implies H2$.

Definition `himpl` ($H1\ H2$:`hprop`) : `Prop` :=
 $\forall h, H1\ h \rightarrow H2\ h$.

Notation "`H1 ==> H2`" := (`himpl H1 H2`) (at level 55) : *hprop_scope*.

Entailment between postconditions, written $Q1 \implies Q2$

Definition `qimpl` A ($Q1\ Q2$: $A \rightarrow \text{hprop}$) : `Prop` :=
 $\forall (v:A), Q1\ v \implies Q2\ v$.

Notation "`Q1 ==> Q2`" := (`qimpl Q1 Q2`) (at level 55) : *hprop_scope*.

Entailment defines an order on heap predicates

Lemma `himpl_refl` : $\forall H,$
 $H \implies H$.

Proof. *intros M. auto. Qed.*

Lemma `himpl_trans` : $\forall H2\ H1\ H3,$
 $(H1 \implies H2) \rightarrow$
 $(H2 \implies H3) \rightarrow$
 $(H1 \implies H3)$.

Proof. *intros M1 M2. unfolds $\times$ himpl. Qed.*

Lemma `himpl_antisym` : $\forall H1\ H2,$
 $(H1 \implies H2) \rightarrow$
 $(H2 \implies H1) \rightarrow$
 $(H1 = H2)$.

Proof. *intros M1 M2. applies pred_ext_1. intros h. iff $\times$. Qed.*

Lemma `qimpl_refl` : $\forall Q,$
 $Q \implies Q$.

Proof. *intros Q v. applies himpl_refl. Qed.*

#[global] Hint Resolve himpl_refl qimpl_refl.

34.3.4 Properties of `hstar`

Lemma `hstar_intro` : $\forall H1\ H2\ h1\ h2,$
 $H1\ h1 \rightarrow$
 $H2\ h2 \rightarrow$
`Fmap.disjoint` $h1\ h2 \rightarrow$
 $(H1 \setminus * H2) (\text{Fmap.union } h1\ h2)$.

Proof. *intros. $\exists \times h1\ h2$. Qed.*

Lemma `hstar_comm` : $\forall H1\ H2,$
 $H1 \setminus * H2 = H2 \setminus * H1$.

Proof.

unfold hprop, hstar. intros H1 H2. applies himpl_antisym.


```

{ intros  $h$  ( $h1 \& h2 \& M1 \& M2 \& D \& U$ ).
  rewrite  $\times$  Fmap.union_comm_of_disjoint in  $U$ .  $\exists \times h2\ h1$ . }
{ intros  $h$  ( $h1 \& h2 \& M1 \& M2 \& D \& U$ ).
  rewrite  $\times$  Fmap.union_comm_of_disjoint in  $U$ .  $\exists \times h2\ h1$ . }

```

Qed.

Lemma `hstar_assoc` : $\forall\ H1\ H2\ H3$,
 $(H1 \setminus * H2) \setminus * H3 = H1 \setminus * (H2 \setminus * H3)$.

Proof.

```

intros  $H1\ H2\ H3$ . applies himpl_antisym; intros  $h$ .
{ intros ( $h' \& h3 \& (h1 \& h2 \& M3 \& M4 \& D' \& U')$ )  $\& M2 \& D \& U$ . subst  $h'$ .
   $\exists\ h1\ (h2 \setminus + h3)$ . splits  $\times$ . { applies  $\times$  hstar_intro. } }
{ intros ( $h1 \& h' \& M1 \& (h2 \& h3 \& M3 \& M4 \& D' \& U')$ )  $\& D \& U$ . subst  $h'$ .
   $\exists\ (h1 \setminus + h2)\ h3$ . splits  $\times$ . { applies  $\times$  hstar_intro. } }

```

Qed.

Lemma `hstar_hempty_l` : $\forall\ H$,
 $\setminus [] \setminus * H = H$.

Proof.

```

intros. applies himpl_antisym; intros  $h$ .
{ intros ( $h1 \& h2 \& M1 \& M2 \& D \& U$ ). hnf in  $M1$ . subst. rewrite  $\times$  Fmap.union_empty_l. }
{ intros  $M$ .  $\exists\ (@\text{Fmap.empty loc } \text{val})\ h$ . splits  $\times$ . { hnfs  $\times$ . } }

```

Qed.

Lemma `hstar_hexists` : $\forall\ A\ (J : A \rightarrow \text{hprop})\ H$,
 $(\text{hexists } J) \setminus * H = \text{hexists } (\text{fun } x \Rightarrow (J\ x) \setminus * H)$.

Proof.

```

intros. applies himpl_antisym; intros  $h$ .
{ intros ( $h1 \& h2 \& (x \& M1) \& M2 \& D \& U$ ).  $\exists \times x\ h1\ h2$ . }
{ intros ( $x \& (h1 \& h2 \& M1 \& M2 \& D \& U)$ ).  $\exists\ h1\ h2$ . splits  $\times$ . {  $\exists \times x$ . } }

```

Qed.

Lemma `hstar_hforall` : $\forall\ H\ A\ (J : A \rightarrow \text{hprop})$,
 $(\text{hforall } J) \setminus * H ==> \text{hforall } (J \setminus * + H)$.

Proof.

```

intros. intros  $h\ M$ . destruct  $M$  as ( $h1 \& h2 \& M1 \& M2 \& D \& U$ ). intros  $x$ .  $\exists \times h1\ h2$ .

```

Qed.

Lemma `himpl_frame_l` : $\forall\ H2\ H1\ H1'$,
 $H1 ==> H1' \rightarrow$
 $(H1 \setminus * H2) ==> (H1' \setminus * H2)$.

Proof. *introv* $W\ (h1 \& h2 \& ?)$. $\exists \times h1\ h2$. Qed.

Additional, symmetric results, useful for tactics

Lemma `hstar_hempty_r` : $\forall\ H$,
 $H \setminus * \setminus [] = H$.

Proof.

applies neutral_r_of_comm_neutral_l. *applies*× hstar_comm. *applies*× hstar_empty_l.

Qed.

Lemma *himpl_frame_r* : $\forall H1 H2 H2'$,

$H2 \implies H2' \rightarrow$

$(H1 \setminus * H2) \implies (H1 \setminus * H2')$.

Proof.

intros M. do 2 rewrite (@hstar_comm *H1*). *applies*× *himpl_frame_l*.

Qed.

34.3.5 Properties of *hpure*

Lemma *hstar_hpure_l* : $\forall P H h$,

$(\setminus [P] \setminus * H) h = (P \wedge H h)$.

Proof.

intros. apply *prop_ext*. unfold *hpure*.

rewrite *hstar_hexists*. rewrite× *hstar_empty_l*.

iff (*p&M*) (*p&M*). { *split*×. } { $\exists$ × *p*. }

Qed.

Lemma *himpl_hstar_hpure_r* : $\forall P H H'$,

$P \rightarrow$

$(H \implies H') \rightarrow$

$H \implies (\setminus [P] \setminus * H')$.

Proof. *intros HP W*. *intros h K*. rewrite× *hstar_hpure_l*. Qed.

Lemma *himpl_hstar_hpure_l* : $\forall P H H'$,

$(P \rightarrow H \implies H') \rightarrow$

$(\setminus [P] \setminus * H) \implies H'$.

Proof. *intros W Hh*. rewrite *hstar_hpure_l* in *Hh*. *applies*× *W*. Qed.

34.3.6 Properties of *hexists*

Lemma *himpl_hexists_l* : $\forall A H (J:A \rightarrow \text{hprop})$,

$(\forall x, J x \implies H) \rightarrow$

$(\text{hexists } J) \implies H$.

Proof. *intros W*. *intros h* (*x&Hh*). *applies*× *W*. Qed.

Lemma *himpl_hexists_r* : $\forall A (x:A) H J$,

$(H \implies J x) \rightarrow$

$H \implies (\text{hexists } J)$.

Proof. *intros W*. *intros h*. $\exists x$. apply× *W*. Qed.

Lemma *himpl_hexists* : $\forall A (J1 J2:A \rightarrow \text{hprop})$,

$J1 \implies J2 \rightarrow$

```

    hexists J1 ==> hexists J2.
Proof.
    introv W. applys himpl_hexists_l. intros x. applys himpl_hexists_r W.
Qed.

```

34.3.7 Properties of **hforall**

```

Lemma himpl_hforall_r : ∀ A (J:A→hprop) H,
  (∀ x, H ==> J x) →
  H ==> (hforall J).
Proof. introv M. intros h Hh x. apply× M. Qed.

Lemma himpl_hforall_l : ∀ A x (J:A→hprop) H,
  (J x ==> H) →
  (hforall J) ==> H.
Proof. introv M. intros h Hh. apply× M. Qed.

Lemma himpl_hforall : ∀ A (J1 J2:A→hprop),
  J1 ==> J2 →
  hforall J1 ==> hforall J2.
Proof.
    introv W. applys himpl_hforall_r. intros x. applys himpl_hforall_l W.
Qed.

```

34.3.8 Properties of **hsingle**

```

Lemma hstar_hsingle_same_loc : ∀ p v1 v2,
  (p ~~> v1) \* (p ~~> v2) ==> \[False].
Proof.
    intros. unfold hsingle. intros h (h1&h2&E1&E2&D&E). false.
    subst. applys× Fmap.disjoint_single_single_same_inv.
Qed.

```

34.3.9 Basic Tactics for Simplifying Entailments

xsimpl performs immediate simplifications on entailment relations.

#[*global*] Hint Rewrite hstar_assoc hstar_hempty_l hstar_hempty_r : *hstar*.

```

Tactic Notation "xsimpl" :=
  try solve [ apply qimpl_refl ];
  try match goal with ⊢ _ ==> _ ⇒ intros ? end;
  autorewrite with hstar; repeat match goal with
  | ⊢ ?H \* _ ==> ?H \* _ ⇒ apply himpl_frame_r
  | ⊢ _ \* ?H ==> _ \* ?H ⇒ apply himpl_frame_l

```

```

| ⊢ _ \* ?H ==> ?H \* _ ⇒ rewrite hstar_comm; apply himpl_frame_r
| ⊢ ?H \* _ ==> _ \* ?H ⇒ rewrite hstar_comm; apply himpl_frame_l
| ⊢ ?H ==> ?H ⇒ apply himpl_refl
| ⊢ ?H ==> ?H' ⇒ is_evar H'; apply himpl_refl end.

```

Tactic Notation "xsimpl" "*" := *xsimpl*; *auto_star*.

xchange helps rewriting in entailments.

```

Lemma xchange_lemma : ∀ H1 H1',
  H1 ==> H1' → ∀ H H' H2,
  H ==> H1 \* H2 →
  H1' \* H2 ==> H' →
  H ==> H'.

```

Proof.

```

  introv M1 M2 M3. applies himpl_trans M2. applies himpl_trans.
  { applies himpl_frame_l. applies M1. } { applies M3. }

```

Qed.

```

Tactic Notation "xchange" constr(M) :=
  forwards_nounfold_then M ltac:(fun K ⇒
    eapply xchange_lemma; [ eapply K | xsimpl | ]).

```

34.4 Separation Logic

34.4.1 Definition of Separation Logic triples

```

Definition triple (t:trm) (H:hprop) (Q:val→hprop) : Prop :=
  ∀ s, H s → eval s t Q.

```

34.4.2 Key Properties of Omni-Big-Step Semantics

```

Lemma eval_conseq : ∀ s t Q1 Q2,
  eval s t Q1 →
  Q1 ==> Q2 →
  eval s t Q2.

```

Proof using.

```

  introv M W.
  asserts W': (∀ v h, Q1 v h → Q2 v h). { auto. } clear W.
  induction M; try solve [ constructors × ].

```

Qed.

```

Lemma eval_frame : ∀ h1 h2 t Q,
  eval h1 t Q →
  Fmap.disjoint h1 h2 →

```

eval (Fmap.union *h1 h2*) *t* (*Q* *+ (= *h2*)).

Proof using.

```

intros M HD. gen h2. induction M; intros;
  try solve [ hint hstar_intro; constructors× ].
{ rename M into M1, H into M2, IHM into IH1, H0 into IH2.
  specializes IH1 HD. applies eval_let IH1. introv HK.
  lets (h1'&h2'&K1'&K2'&KD&KU): HK. subst. applies× IH2. }
{ rename H into M. applies eval_ref. intros p Hp.
  rewrite Fmap.indom_union_eq in Hp. rew_logic in Hp.
  destruct Hp as [Hp1 Hp2].
  rewrite× Fmap.update_union_not_r. applies hstar_intro.
  { applies× M.  { auto.  } { applies× Fmap.disjoint_update_not_r.  } }
{ applies eval_get. { rewrite× Fmap.indom_union_eq. 
  { rewrite× Fmap.read_union_l. applies× hstar_intro.  } } }
{ applies eval_set. { rewrite× Fmap.indom_union_eq. 
  { rewrite× Fmap.update_union_l. applies hstar_intro.
    { auto.  } { auto.  } { applies× Fmap.disjoint_update_l.  } } } }
{ applies eval_free. { rewrite× Fmap.indom_union_eq. 
  { rewrite× Fmap.remove_disjoint_union_l. applies hstar_intro.
    { auto.  } { auto.  } { applies× Fmap.disjoint_remove_l.  } } } }
  }

```

Qed.

34.4.3 Structural Rules

Lemma `triple_conseq` : $\forall t H' Q' H Q,$

```

triple t H' Q' →
H ==> H' →
Q' ==> Q →
triple t H Q.

```

Proof using. *unfolds triple. introv M MH MQ HF. applies× eval_conseq. Qed.*

Lemma `triple_frame` : $\forall t H Q H',$

```

triple t H Q →
triple t (H \* H') (Q \*+ H').

```

Proof.

```

intros M. intros h HF. lets (h1&h2&M1&M2&MD&MU): (rm HF).
  subst. specializes M M1. applies eval_conseq.
  { applies eval_frame M MD.  } { xsimpl. intros h' →. applies M2.  }

```

Qed.

Lemma `triple_hexists` : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q,$

```

( $\forall (x:A), \text{triple } t (J x) Q$ )  $\rightarrow$ 
triple t (hexists J) Q.

```

Proof using. *intros M. intros h (x&Hh). applies M Hh. Qed.*

Lemma triple_hpure : $\forall t (P:\text{Prop}) H Q,$
 $(P \rightarrow \text{triple } t H Q) \rightarrow$
 $\text{triple } t (\backslash[P] \backslash * H) Q.$

Proof using.

intros M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).
inverts HP. subst. rewrite Fmap.union_empty_l. applies $\neg$ M.

Qed.

34.4.4 Reasoning Rules for Terms

Lemma triple_val : $\forall v H Q,$
 $H \Rightarrow Q v \rightarrow$
 $\text{triple } (\text{trm\_val } v) H Q.$

Proof using. *intros M Hs. applies $\times$ eval_val. Qed.*

Lemma triple_fix : $\forall f x t1 H Q,$
 $H \Rightarrow Q (\text{val\_fix } f x t1) \rightarrow$
 $\text{triple } (\text{trm\_fix } f x t1) H Q.$

Proof using. *intros M Hs. applies $\times$ eval_fix. Qed.*

Lemma triple_if : $\forall (b:\text{bool}) t1 t2 H Q,$
 $\text{triple } (\text{if } b \text{ then } t1 \text{ else } t2) H Q \rightarrow$
 $\text{triple } (\text{trm\_if } b t1 t2) H Q.$

Proof using. *intros M Hs. applies $\times$ eval_if. Qed.*

Lemma triple_app_fix : $\forall v1 v2 f x t1 H Q,$
 $v1 = \text{val\_fix } f x t1 \rightarrow$
 $\text{triple } (\text{subst } x v2 (\text{subst } f v1 t1)) H Q \rightarrow$
 $\text{triple } (\text{trm\_app } v1 v2) H Q.$

Proof using. *intros E M Hs. applies $\times$ eval_app_fix. Qed.*

Lemma triple_let : $\forall x t1 t2 Q1 H Q,$
 $\text{triple } t1 H Q1 \rightarrow$
 $(\forall v1, \text{triple } (\text{subst } x v1 t2) (Q1 v1) Q) \rightarrow$
 $\text{triple } (\text{trm\_let } x t1 t2) H Q.$

Proof using. *intros M1 M2 Hs. applies $\times$ eval_let. Qed.*

34.4.5 Specification of Primitive Operations

Lemma triple_add : $\forall n1 n2,$
 $\text{triple } (\text{val\_add } n1 n2)$
 $\backslash []$
 $(\text{fun } r \Rightarrow \backslash[r = \text{val\_int } (n1 + n2)]).$

Proof using.

intros Hs. applies $\times$ eval_add. inverts Hs. $\exists \times$. hnfs $\times$.

Qed.

```
Lemma triple_div : ∀ n1 n2,  
  n2 ≠ 0 →  
  triple (val_div n1 n2)  
    \ []  
    (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Proof using.

```
  introv Hn2 Hs. applies× eval_div. inverts Hs. ∃×. hnfs×.
```

Qed.

```
Lemma triple_rand : ∀ n,  
  n > 0 →  
  triple (val_rand n)  
    \ []  
    (fun r ⇒ \[∃ n1, r = val_int n1 ∧ 0 ≤ n1 < n]).
```

Proof using.

```
  introv Hn2 Hs. applies× eval_rand. inverts Hs.  
  intros n1 Hn1. hnfs. ∃×. hnfs×.
```

Qed.

```
Lemma triple_ref : ∀ v,  
  triple (val_ref v)  
    \ []  
    (fun r ⇒ \∃ p, \[r = val_loc p] \* p ~~> v).
```

Proof using.

```
  intros. intros s1 K. applies eval_ref. intros p D.  
  inverts K. rewrite Fmap.update_empty. ∃ p.  
  rewrite hstar_hpure_l. split×. hnfs×.
```

Qed.

```
Lemma triple_get : ∀ v p,  
  triple (val_get p)  
    (p ~~> v)  
    (fun r ⇒ \[r = v] \* (p ~~> v)).
```

Proof using.

```
  intros. intros s K. inverts K. applies eval_get.  
  { applies× Fmap.indom_single. }  
  { rewrite hstar_hpure_l. split×. rewrite× Fmap.read_single. hnfs×. }
```

Qed.

```
Lemma triple_set : ∀ w p v,  
  triple (val_set (val_loc p) v)  
    (p ~~> w)  
    (fun r ⇒ (p ~~> v)).
```

Proof using.

```

intros. intros s1 K. inverts K. applies eval_set.
{ applies× Fmap.indom_single. }
{ rewrite Fmap.update_single. hnfs×. }
Qed.

```

Lemma `triple_free` : $\forall p\ v$,
`triple (val_free (val_loc p))`
 $(p \sim\sim> v)$
 $(\text{fun } r \Rightarrow \backslash []).$

Proof using.

```

intros. intros s1 K. inverts K. applies eval_free.
{ applies× Fmap.indom_single. }
{ rewrite× Fmap.remove_single. hnfs×. }
Qed.

```

34.5 Bonus: Example Proof

See chapter [Rules](#) for comments on this proof.

OCaml:

```
let incr p = let n = !p in let m = n+1 in p := m
```

Definition of the function *incr*, using low-level syntax.

Open Scope *string_scope*.

```

Definition incr : val :=
  val_fix "f" "p" (
    trm_let "n" (trm_app val_get (trm_var "p")) (
      trm_let "m" (trm_app (trm_app val_add
        (trm_var "n")) (val_int 1)) (
        trm_app (trm_app val_set (trm_var "p")) (trm_var "m")))).

```

Specification of the function *incr*.

```

Lemma triple_incr :  $\forall (p:\text{loc}) (n:\text{int})$ ,
  triple (trm_app incr p)
    (p  $\sim\sim> n$ )
    (fun _  $\Rightarrow p \sim\sim> (n+1)$ ).

```

Verification of *incr*, applying the reasoning rules by hand.

Proof using.

```

intros. applies triple_app_fix. { reflexivity. } simpl.
applies triple_let. { apply triple_get. }
intros n'. simpl. apply triple_hpure. intros  $\rightarrow$ .
applies triple_let. { applies triple_conseq.
  { applies triple_frame. applies triple_add. }
  { xsimpl. }

```



```

      { xsimpl. } }
intros m'. simpl. apply triple_hpure. intros →.
{ apply triple_set. }
Qed.

```

Chapter 35

Library `SLF.LibSepReference`

35.1 LibSepReference: Appendix - The Full Construction

This file provides an end-to-end construction of a weakest-precondition style characteristic formula generator (the function named `wpgen` in chapter `WPgen`), for a core programming language with programs assumed to be in A-normal form, including support for records with up to 3 fields.

```
Set Implicit Arguments.
From SLF Require Export LibCore.
From SLF Require Export LibSepTLCbuffer LibSepVar LibSepFmap.
From SLF Require LibSepSimpl.
Module FMAP := LIBSEPFMAP.
```

35.2 Imports

35.2.1 Extensionality Axioms

These standard extensionality axioms may also be found in the *LibAxiom.v* file associated with the TLC library.

```
Axiom functional_extensionality :  $\forall A B (f g:A \rightarrow B),$ 
   $(\forall x, f x = g x) \rightarrow$ 
   $f = g.$ 
Axiom propositional_extensionality :  $\forall (P Q:\text{Prop}),$ 
   $(P \leftrightarrow Q) \rightarrow$ 
   $P = Q.$ 
```

35.2.2 Variables

The file *LibSepVar.v*, whoses definitions are imported in the header to the present file, defines the type `var` as an alias for `string`. It also provides the boolean function `var_eq x y` to compare two variables, and the tactic `case_var` to perform case analysis on expressions of the form `if var_eq x y then .. else ..` that appear in the goal.

35.2.3 Finite Maps

The file *LibSepFmap.v*, which is bound by the short name `FMAP` in the header, provides a formalization of finite maps. These maps are used to represent heaps in the semantics. The library provides a tactic called *fmap_disjoint* to automate disjointness proofs, and a tactic called *fmap_eq* for proving equalities between heaps modulo associativity and commutativity. Without these two tactics, proofs involving finite maps would be much more tedious and fragile.

35.3 Source Language

35.3.1 Syntax

The grammar of primitive operations includes a number of operations.

Inductive `prim` : Type :=

```
| val_ref : prim
| val_get : prim
| val_set : prim
| val_free : prim
| val_neg : prim
| val_opp : prim
| val_eq : prim
| val_add : prim
| val_neq : prim
| val_sub : prim
| val_mul : prim
| val_div : prim
| val_mod : prim
| val_rand : prim
| val_le : prim
| val_lt : prim
| val_ge : prim
| val_gt : prim
| val_ptr_add : prim.
```

Locations are defined as natural numbers.

Definition `loc : Type := nat`.

The null location corresponds to address zero.

Definition `null : loc := 0%nat`.

The grammar of closed values includes basic values such as `int` and `bool`, but also locations, closures. It also includes two special values, `val_uninit` used in the formalization of arrays, and `val_error` used for stating semantics featuring error-propagation.

Inductive `val : Type :=`

```
| val_unit : val
| val_bool : bool → val
| val_int : int → val
| val_loc : loc → val
| val_prim : prim → val
| val_fun : var → trm → val
| val_fix : var → var → trm → val
| val_uninit : val
| val_error : val
```

The grammar of terms includes values, variables, functions, applications, sequence, let-bindings, and conditions. Sequences are redundant with let-bindings, but are useful in practice to avoid binding dummy names, and serve on numerous occasion as a warm-up before proving results on let-bindings.

with `trm : Type :=`

```
| trm_val : val → trm
| trm_var : var → trm
| trm_fun : var → trm → trm
| trm_fix : var → var → trm → trm
| trm_app : trm → trm → trm
| trm_seq : trm → trm → trm
| trm_let : var → trm → trm → trm
| trm_if : trm → trm → trm → trm.
```

A state, a.k.a. `heap`, consists of a finite map from location to values. Records and arrays are represented as sets of consecutive cells, preceded by a header field describing the length of the block.

Definition `heap : Type := fmap loc val`.

35.3.2 Rocq Tweaks

`h1 \u h2` is a notation for union of two heaps.

Notation `"h1 \u h2" := (Fmap.union h1 h2)`
(at level 37, right associativity).

Implicit types associated with meta-variables.

```
Implicit Types f : var.
Implicit Types b : bool.
Implicit Types p : loc.
Implicit Types n : int.
Implicit Types v w r vf vx : val.
Implicit Types t : trm.
Implicit Types h s : heap.
```

The types of values and terms are inhabited.

```
Global Instance Inhab_val : Inhab val.
Proof using. apply (Inhab_of_val val_unit). Qed.

Global Instance Inhab_trm : Inhab trm.
Proof using. apply (Inhab_of_val (trm_val val_unit)). Qed.
```

Coercions to improve conciseness in the statment of evaluation rules.

```
Coercion val_bool : bool >-> val.
Coercion val_int : Z >-> val.
Coercion val_loc : loc >-> val.
Coercion val_prim : prim >-> val.

Coercion trm_val : val >-> trm.
Coercion trm_var : var >-> trm.
Coercion trm_app : trm >-> Funclass.
```

35.3.3 Values and Substitutions

The predicate `trm_is_val t` asserts that `t` is a value.

```
Definition trm_is_val (t:trm) : Prop :=
  match t with trm_val v => True | _ => False end.
```

The standard capture-avoiding substitution, written `subst x v t`.

```
Fixpoint subst (y:var) (v':val) (t:trm) : trm :=
  let aux t := subst y v' t in
  let if_y_eq x t1 t2 := if var_eq x y then t1 else t2 in
  match t with
  | trm_val v => trm_val v
  | trm_var x => if_y_eq x (trm_val v') t
  | trm_fun x t1 => trm_fun x (if_y_eq x t1 (aux t1))
  | trm_fix f x t1 => trm_fix f x (if_y_eq f t1 (if_y_eq x t1 (aux t1)))
  | trm_app t1 t2 => trm_app (aux t1) (aux t2)
  | trm_seq t1 t2 => trm_seq (aux t1) (aux t2)
  | trm_let x t1 t2 => trm_let x (aux t1) (if_y_eq x t2 (aux t2))
  | trm_if t0 t1 t2 => trm_if (aux t0) (aux t1) (aux t2)
```

end.

35.3.4 Small-Step Semantics

The judgment **step** $s\ t\ s'\ t'$ asserts that the configuration (s,t) can take one reduction step towards the program configuration (s',t') .

Inductive **step** : heap $\rightarrow$ trm $\rightarrow$ heap $\rightarrow$ trm $\rightarrow$ Prop :=

```
| step_seq_ctx :  $\forall\ s1\ s2\ t1\ t1'\ t2,$   
  step  $s1\ t1\ s2\ t1' \rightarrow$   
  step  $s1\ (\text{trm\_seq } t1\ t2)\ s2\ (\text{trm\_seq } t1'\ t2)$   
| step_let_ctx :  $\forall\ s1\ s2\ x\ t1\ t1'\ t2,$   
  step  $s1\ t1\ s2\ t1' \rightarrow$   
  step  $s1\ (\text{trm\_let } x\ t1\ t2)\ s2\ (\text{trm\_let } x\ t1'\ t2)$   
| step_app_arg1 :  $\forall\ s1\ s2\ t1\ t1'\ t2,$   
  step  $s1\ t1\ s2\ t1' \rightarrow$   
  step  $s1\ (\text{trm\_app } t1\ t2)\ s2\ (\text{trm\_app } t1'\ t2)$   
| step_app_arg2 :  $\forall\ s1\ s2\ v1\ t2\ t2',$   
  step  $s1\ t2\ s2\ t2' \rightarrow$   
  step  $s1\ (\text{trm\_app } v1\ t2)\ s2\ (\text{trm\_app } v1\ t2')$   
  
| step_fun :  $\forall\ s\ x\ t1,$   
  step  $s\ (\text{trm\_fun } x\ t1)\ s\ (\text{val\_fun } x\ t1)$   
| step_fix :  $\forall\ s\ f\ x\ t1,$   
  step  $s\ (\text{trm\_fix } f\ x\ t1)\ s\ (\text{val\_fix } f\ x\ t1)$   
| step_app_fun :  $\forall\ s\ v1\ v2\ x\ t1,$   
   $v1 = \text{val\_fun } x\ t1 \rightarrow$   
  step  $s\ (\text{trm\_app } v1\ v2)\ s\ (\text{subst } x\ v2\ t1)$   
| step_app_fix :  $\forall\ s\ v1\ v2\ f\ x\ t1,$   
   $v1 = \text{val\_fix } f\ x\ t1 \rightarrow$   
  step  $s\ (\text{trm\_app } v1\ v2)\ s\ (\text{subst } x\ v2\ (\text{subst } f\ v1\ t1))$   
| step_if :  $\forall\ s\ b\ t1\ t2,$   
  step  $s\ (\text{trm\_if } (\text{val\_bool } b)\ t1\ t2)\ s\ (\text{if } b\ \text{then } t1\ \text{else } t2)$   
| step_seq :  $\forall\ s\ t2\ v1,$   
  step  $s\ (\text{trm\_seq } v1\ t2)\ s\ t2$   
| step_let :  $\forall\ s\ x\ t2\ v1,$   
  step  $s\ (\text{trm\_let } x\ v1\ t2)\ s\ (\text{subst } x\ v1\ t2)$   
  
| step_neg :  $\forall\ s\ b,$ 
```

```

    step s (val_neg (val_bool b)) s (val_bool (neg b))
| step_opp : ∀ s n,
    step s (val_opp (val_int n)) s (val_int (- n))
| step_rand : ∀ s n n1,
    0 ≤ n1 < n →
    step s (val_rand (val_int n)) s (val_int n1)

| step_eq : ∀ s v1 v2,
    step s (val_eq v1 v2) s (val_bool (isTrue (v1 = v2)))
| step_neq : ∀ s v1 v2,
    step s (val_neq v1 v2) s (val_bool (isTrue (v1 ≠ v2)))
| step_add : ∀ s n1 n2,
    step s (val_add (val_int n1) (val_int n2)) s (val_int (n1 + n2))
| step_sub : ∀ s n1 n2,
    step s (val_sub (val_int n1) (val_int n2)) s (val_int (n1 - n2))
| step_mul : ∀ s n1 n2,
    step s (val_mul (val_int n1) (val_int n2)) s (val_int (n1 × n2))
| step_div : ∀ s n1 n2,
    n2 ≠ 0 →
    step s (val_div (val_int n1) (val_int n2)) s (Z.quot n1 n2)
| step_mod : ∀ s n1 n2,
    n2 ≠ 0 →
    step s (val_mod (val_int n1) (val_int n2)) s (Z.rem n1 n2)
| step_le : ∀ s n1 n2,
    step s (val_le (val_int n1) (val_int n2)) s (val_bool (isTrue (n1 ≤ n2)))
| step_lt : ∀ s n1 n2,
    step s (val_lt (val_int n1) (val_int n2)) s (val_bool (isTrue (n1 < n2)))
| step_ge : ∀ s n1 n2,
    step s (val_ge (val_int n1) (val_int n2)) s (val_bool (isTrue (n1 ≥ n2)))
| step_gt : ∀ s n1 n2,
    step s (val_gt (val_int n1) (val_int n2)) s (val_bool (isTrue (n1 > n2)))
| step_ptr_add : ∀ s p1 p2 n,
    (p2:int) = p1 + n →
    step s (val_ptr_add (val_loc p1) (val_int n)) s (val_loc p2)

| step_ref : ∀ s v p,
    ¬ Fmap.indom s p →
    step s (val_ref v) (Fmap.update s p v) (val_loc p)
| step_get : ∀ s p,
    Fmap.indom s p →

```

```

    step  $s$  (val_get (val_loc  $p$ ))  $s$  (Fmap.read  $s$   $p$ )
| step_set :  $\forall s p v$ ,
    Fmap.indom  $s p \rightarrow$ 
    step  $s$  (val_set (val_loc  $p$ )  $v$ ) (Fmap.update  $s p v$ ) val_unit
| step_free :  $\forall s p$ ,
    Fmap.indom  $s p \rightarrow$ 
    step  $s$  (val_free (val_loc  $p$ )) (Fmap.remove  $s p$ ) val_unit.

```

The judgment **steps** $s t s' t'$ corresponds to the reflexive-transitive closure of **step**. Concretely, this judgment asserts that the configuration (s,t) can reduce in zero, one, or several steps to (s',t') .

```

Inductive steps : heap  $\rightarrow$  trm  $\rightarrow$  heap  $\rightarrow$  trm  $\rightarrow$  Prop :=
| steps_refl :  $\forall s t$ ,
    steps  $s t s t$ 
| steps_step :  $\forall s1 s2 s3 t1 t2 t3$ ,
    step  $s1 t1 s2 t2 \rightarrow$ 
    steps  $s2 t2 s3 t3 \rightarrow$ 
    steps  $s1 t1 s3 t3$ .

```

```

Lemma steps_of_step :  $\forall s1 s2 t1 t2$ ,
    step  $s1 t1 s2 t2 \rightarrow$ 
    steps  $s1 t1 s2 t2$ .

```

Proof using. *intros M. applys steps_step M. applys steps_refl. Qed.*

```

Lemma steps_trans :  $\forall s1 s2 s3 t1 t2 t3$ ,
    steps  $s1 t1 s2 t2 \rightarrow$ 
    steps  $s2 t2 s3 t3 \rightarrow$ 
    steps  $s1 t1 s3 t3$ .

```

Proof using. *intros M1. induction M1; intros M2. { auto. } { constructors $\times$. } Qed.*

The predicate **reducible** $s t$ asserts that the configuration (s,t) can take a step.

```

Definition reducible ( $s$ :heap) ( $t$ :trm) : Prop :=
     $\exists s' t'$ , step  $s t s' t'$ .

```

The predicate **notstuck** $s t$ asserts that t is a value or is reducible.

```

Definition notstuck ( $s$ :heap) ( $t$ :trm) : Prop :=
    trm_is_val  $t \vee$  reducible  $s t$ .

```

35.3.5 Omni-Big-Step Semantics of primitive operations

Evaluation rules for unary operations are captured by the predicate **evalunop** $op v1 P$, which asserts that $op v1$ evaluates to a value $v2$ satisfying P .

```

Inductive evalunop : prim  $\rightarrow$  val  $\rightarrow$  (val $\rightarrow$ Prop)  $\rightarrow$  Prop :=
| evalunop_neg :  $\forall b1$ ,
    evalunop val_neg (val_bool  $b1$ ) (= val_bool (neg  $b1$ ))

```



```

| evalunop_opp :  $\forall n1$ ,
  evalunop val_opp (val_int  $n1$ ) (= val_int (-  $n1$ ))
| evalunop_rand :  $\forall n$ ,
   $n > 0 \rightarrow$ 
  evalunop val_rand (val_int  $n$ ) (fun  $r \Rightarrow \exists n1, r = \text{val\_int } n1 \wedge 0 \leq n1 < n$ ).

```

Evaluation rules for binary operations are captured by the predicate *redupop op v1 v2 P*, which asserts that *op v1 v2* evaluates to a value *v3* satisfying *P*.

```

Inductive evalbinop : val  $\rightarrow$  val  $\rightarrow$  val  $\rightarrow$  (val $\rightarrow$ Prop)  $\rightarrow$  Prop :=
| evalbinop_eq :  $\forall v1 v2$ ,
  evalbinop val_eq  $v1 v2$  (= val_bool (isTrue ( $v1 = v2$ )))
| evalbinop_neq :  $\forall v1 v2$ ,
  evalbinop val_neq  $v1 v2$  (= val_bool (isTrue ( $v1 \neq v2$ )))
| evalbinop_add :  $\forall n1 n2$ ,
  evalbinop val_add (val_int  $n1$ ) (val_int  $n2$ ) (= val_int ( $n1 + n2$ ))
| evalbinop_sub :  $\forall n1 n2$ ,
  evalbinop val_sub (val_int  $n1$ ) (val_int  $n2$ ) (= val_int ( $n1 - n2$ ))
| evalbinop_mul :  $\forall n1 n2$ ,
  evalbinop val_mul (val_int  $n1$ ) (val_int  $n2$ ) (= val_int ( $n1 \times n2$ ))
| evalbinop_div :  $\forall n1 n2$ ,
   $n2 \neq 0 \rightarrow$ 
  evalbinop val_div (val_int  $n1$ ) (val_int  $n2$ ) (= val_int (Z.quot  $n1 n2$ ))
| evalbinop_mod :  $\forall n1 n2$ ,
   $n2 \neq 0 \rightarrow$ 
  evalbinop val_mod (val_int  $n1$ ) (val_int  $n2$ ) (= val_int (Z.rem  $n1 n2$ ))
| evalbinop_le :  $\forall n1 n2$ ,
  evalbinop val_le (val_int  $n1$ ) (val_int  $n2$ ) (= val_bool (isTrue ( $n1 \leq n2$ )))
| evalbinop_lt :  $\forall n1 n2$ ,
  evalbinop val_lt (val_int  $n1$ ) (val_int  $n2$ ) (= val_bool (isTrue ( $n1 < n2$ )))
| evalbinop_ge :  $\forall n1 n2$ ,
  evalbinop val_ge (val_int  $n1$ ) (val_int  $n2$ ) (= val_bool (isTrue ( $n1 \geq n2$ )))
| evalbinop_gt :  $\forall n1 n2$ ,
  evalbinop val_gt (val_int  $n1$ ) (val_int  $n2$ ) (= val_bool (isTrue ( $n1 > n2$ )))
| evalbinop_ptr_add :  $\forall p1 p2 n$ ,
  ( $p2:\text{int}$ ) =  $p1 + n \rightarrow$ 
  evalbinop val_ptr_add (val_loc  $p1$ ) (val_int  $n$ ) (= val_loc  $p2$ ).

```

35.3.6 Omni-Big-Step Semantics

Section Eval.

The predicate *purepost s P* converts a predicate *P:val $\rightarrow$ Prop* into a postcondition of type *val $\rightarrow$ heap $\rightarrow$ Prop* that holds in the state *s*.

Definition purepost ($s:\text{heap}$) ($P:\text{val} \rightarrow \text{Prop}$) : $\text{val} \rightarrow \text{heap} \rightarrow \text{Prop} :=$
 fun $v\ s' \Rightarrow P\ v \wedge s' = s$.

Definition purepostin ($s:\text{heap}$) ($P:\text{val} \rightarrow \text{Prop}$) ($Q:\text{val} \rightarrow \text{heap} \rightarrow \text{Prop}$) : $\text{Prop} :=$

$\forall v, P\ v \rightarrow Q\ v\ s$.

Omni-Big-step evaluation judgement, written $\text{eval}\ s\ t\ Q$.

Implicit Types $Q : \text{val} \rightarrow \text{heap} \rightarrow \text{Prop}$.

Inductive $\text{eval} : \text{heap} \rightarrow \text{trm} \rightarrow (\text{val} \rightarrow \text{heap} \rightarrow \text{Prop}) \rightarrow \text{Prop} :=$

| eval_val : $\forall s\ v\ Q,$
 $Q\ v\ s \rightarrow$
 $\text{eval}\ s\ (\text{trm\_val}\ v)\ Q$
 | eval_fun : $\forall s\ x\ t1\ Q,$
 $Q\ (\text{val\_fun}\ x\ t1)\ s \rightarrow$
 $\text{eval}\ s\ (\text{trm\_fun}\ x\ t1)\ Q$
 | eval_fix : $\forall s\ f\ x\ t1\ Q,$
 $Q\ (\text{val\_fix}\ f\ x\ t1)\ s \rightarrow$
 $\text{eval}\ s\ (\text{trm\_fix}\ f\ x\ t1)\ Q$
 | eval_app_arg1 : $\forall s1\ t1\ t2\ Q1\ Q,$
 $\neg \text{trm\_is\_val}\ t1 \rightarrow$
 $\text{eval}\ s1\ t1\ Q1 \rightarrow$
 $(\forall v1\ s2, Q1\ v1\ s2 \rightarrow \text{eval}\ s2\ (\text{trm\_app}\ v1\ t2)\ Q) \rightarrow$
 $\text{eval}\ s1\ (\text{trm\_app}\ t1\ t2)\ Q$
 | eval_app_arg2 : $\forall s1\ v1\ t2\ Q1\ Q,$
 $\neg \text{trm\_is\_val}\ t2 \rightarrow$
 $\text{eval}\ s1\ t2\ Q1 \rightarrow$
 $(\forall v2\ s2, Q1\ v2\ s2 \rightarrow \text{eval}\ s2\ (\text{trm\_app}\ v1\ v2)\ Q) \rightarrow$
 $\text{eval}\ s1\ (\text{trm\_app}\ v1\ t2)\ Q$
 | eval_app_fun : $\forall s1\ v1\ v2\ x\ t1\ Q,$
 $v1 = \text{val\_fun}\ x\ t1 \rightarrow$
 $\text{eval}\ s1\ (\text{subst}\ x\ v2\ t1)\ Q \rightarrow$
 $\text{eval}\ s1\ (\text{trm\_app}\ v1\ v2)\ Q$
 | eval_app_fix : $\forall s\ v1\ v2\ f\ x\ t1\ Q,$
 $v1 = \text{val\_fix}\ f\ x\ t1 \rightarrow$
 $\text{eval}\ s\ (\text{subst}\ x\ v2\ (\text{subst}\ f\ v1\ t1))\ Q \rightarrow$
 $\text{eval}\ s\ (\text{trm\_app}\ v1\ v2)\ Q$
 | eval_seq : $\forall Q1\ s\ t1\ t2\ Q,$
 $\text{eval}\ s\ t1\ Q1 \rightarrow$
 $(\forall v1\ s2, Q1\ v1\ s2 \rightarrow \text{eval}\ s2\ t2\ Q) \rightarrow$
 $\text{eval}\ s\ (\text{trm\_seq}\ t1\ t2)\ Q$
 | eval_let : $\forall Q1\ s\ x\ t1\ t2\ Q,$
 $\text{eval}\ s\ t1\ Q1 \rightarrow$

```

    (∀ v1 s2, Q1 v1 s2 → eval s2 (subst x v1 t2) Q) →
    eval s (trm_let x t1 t2) Q
| eval_if : ∀ s (b:bool) t1 t2 Q,
    eval s (if b then t1 else t2) Q →
    eval s (trm_if (val_bool b) t1 t2) Q
| eval_unop : ∀ op s v1 P Q,
    evalunop op v1 P →
    purepostin s P Q →
    eval s (op v1) Q
| eval_binop : ∀ op s v1 v2 P Q,
    evalbinop op v1 v2 P →
    purepostin s P Q →
    eval s (op v1 v2) Q
| eval_ref : ∀ s v Q,
    (∀ p, ¬ Fmap.indom s p →
     Q (val_loc p) (Fmap.update s p v)) →
    eval s (val_ref v) Q
| eval_get : ∀ s p Q,
    Fmap.indom s p →
    Q (Fmap.read s p) s →
    eval s (val_get (val_loc p)) Q
| eval_set : ∀ s p v Q,
    Fmap.indom s p →
    Q val_unit (Fmap.update s p v) →
    eval s (val_set (val_loc p) v) Q
| eval_free : ∀ s p Q,
    Fmap.indom s p →
    Q val_unit (Fmap.remove s p) →
    eval s (val_free (val_loc p)) Q.

```

Specialized rule for values, to instantiate the postcondition

Lemma eval_val_minimal : ∀ s v,
 eval s (trm_val v) (fun v' s' ⇒ (v' = v) ∧ (s' = s)).

Proof using. intros. *applies*× eval_val. Qed.

Specialized evaluation rules for selected operations, to avoid the indirection via `eval_unop` and `eval_binop` in the course.

Lemma eval_add : ∀ s n1 n2 Q,
 Q (val_int (n1 + n2)) s →
 eval s (val_add (val_int n1) (val_int n2)) Q.

Proof using.

```

    intros. applies× eval_binop.
    { applies× evalbinop_add. }

```

{ intros ? →. auto. }

Qed.

Lemma eval_div : $\forall s \ n1 \ n2 \ Q,$
 $n2 \neq 0 \rightarrow$
 $Q \text{ (val\_int (Z.quot } n1 \ n2)) \ s \rightarrow$
 eval $s \text{ (val\_div (val\_int } n1) \text{ (val\_int } n2))} \ Q.$

Proof using.

intros. *applies* × eval_binop.
{ *applies* × evalbinop_div. }
{ intros ? →. auto. }

Qed.

Lemma eval_rand : $\forall s \ n \ Q,$
 $n > 0 \rightarrow$
 $(\forall n1, 0 \leq n1 < n \rightarrow Q \ n1 \ s) \rightarrow$
 eval $s \text{ (val\_rand (val\_int } n))} \ Q.$

Proof using.

intros. *applies* × eval_unop.
{ *applies* × evalunop_rand. }
{ intros ? (?&->&?). auto. }

Qed.

Derived rule for reasoning about an applications, without need to perform a case analysis on whether the entities are already values.

Lemma eval_app_arg1' : $\forall s1 \ t1 \ t2 \ Q1 \ Q,$
 eval $s1 \ t1 \ Q1 \rightarrow$
 $(\forall v1 \ s2, Q1 \ v1 \ s2 \rightarrow \text{eval } s2 \text{ (trm\_app } v1 \ t2) \ Q) \rightarrow$
 eval $s1 \text{ (trm\_app } t1 \ t2) \ Q.$

Proof using.

introv $M1 \ M2.$ tests $C1$: (trm_is_val $t1$).
{ destruct $t1$; tryfalse. inverts $M1.$ *applies* × $M2.$ }
{ *applies* × eval_app_arg1. }

Qed.

Lemma eval_app_arg2' : $\forall s1 \ v1 \ t2 \ Q1 \ Q,$
 eval $s1 \ t2 \ Q1 \rightarrow$
 $(\forall v2 \ s2, \text{eval } s2 \text{ (trm\_app } v1 \ v2) \ Q) \rightarrow$
 eval $s1 \text{ (trm\_app } v1 \ t2) \ Q.$

Proof using.

introv $M1 \ M2.$ tests $C1$: (trm_is_val $t2$).
{ destruct $t2$; tryfalse. inverts $M1.$ *applies* × $M2.$ }
{ *applies* × eval_app_arg2. }

Qed.

eval_like $t1 \ t2$ asserts that $t2$ evaluates like $t1$. In particular, this relation hold whenever

$t2$ reduces in small-step to $t1$.

```
Definition eval_like (t1 t2:trm) : Prop :=
  ∀ s Q, eval s t1 Q → eval s t2 Q.
```

```
End Eval.
```

35.4 Heap Predicates

We next define heap predicates and establish their properties. All this material is wrapped in a module, allowing us to instantiate the functor from chapter `LibSepSimpl` that defines the tactic `xsimpl`. `Module SEPSIMPLARGS`.

35.4.1 Hprop and Entailment

```
Declare Scope hprop_scope.
```

```
Open Scope hprop_scope.
```

The type of heap predicates is named `hprop`.

```
Definition hprop := heap → Prop.
```

Implicit types for meta-variables.

```
Implicit Types P : Prop.
```

```
Implicit Types H : hprop.
```

```
Implicit Types Q : val → hprop.
```

Entailment for heap predicates, written $H1 \implies H2$. This entailment is linear.

```
Definition himpl (H1 H2:hprop) : Prop :=
  ∀ h, H1 h → H2 h.
```

```
Notation "H1 ==> H2" := (himpl H1 H2) (at level 55) : hprop_scope.
```

Entailment between postconditions, written $Q1 \implies Q2$.

```
Definition qimpl A (Q1 Q2:A → hprop) : Prop :=
  ∀ (v:A), Q1 v ==> Q2 v.
```

```
Notation "Q1 ==> Q2" := (qimpl Q1 Q2) (at level 55) : hprop_scope.
```

35.4.2 Definition of Heap Predicates

The core heap predicates are defined next, together with the associated notation:

- $\backslash[]$ denotes the empty heap predicate
- $\backslash[P]$ denotes a pure fact
- $\backslash\textit{Top}$ denotes the true heap predicate (affine)

- $p \rightsquigarrow v$ denotes a singleton heap
- $H1 \searrow^* H2$ denotes the separating conjunction
- $Q1 \searrow^+ H2$ denotes the separating conjunction extending a postcondition
- $\exists x, H$ denotes an existential quantifier
- $\forall x, H$ denotes a universal quantifier
- $H1 \searrow^* H2$ denotes a magic wand between heap predicates
- $Q1 \searrow^* Q2$ denotes a magic wand between postconditions.

Definition `hempty : hprop :=`
`fun h => (h = Fmap.empty).`

Definition `hsingle (p:loc) (v:val) : hprop :=`
`fun h => (h = Fmap.single p v).`

Definition `hstar (H1 H2 : hprop) : hprop :=`
`fun h =>  $\exists h1 h2, H1 h1$`

$$\wedge H2 h2$$

$$\wedge \text{Fmap.disjoint } h1 h2$$

$$\wedge h = \text{Fmap.union } h1 h2.$$

Definition `hexists A (J:A→hprop) : hprop :=`
`fun h =>  $\exists x, J x h$ .`

Definition `hforall (A : Type) (J : A → hprop) : hprop :=`
`fun h =>  $\forall x, J x h$ .`

Notation `"\[" := (hempty)`
`(at level 0) : hprop_scope.`

Notation `"p '~~>' v" := (hsingle p v) (at level 32) : hprop_scope.`

Notation `"H1 ' \searrow^* ' H2" := (hstar H1 H2)`
`(at level 41, right associativity) : hprop_scope.`

Notation `"'\exists' x1 .. xn , H" :=`
`(hexists (fun x1 => .. (hexists (fun xn => H)) ..))`
`(at level 39, x1 binder, H at level 50, right associativity,`
`format "'[' '\exists' '/' x1 .. xn , '/' H ']'") : hprop_scope.`

Notation `"'\forall' x1 .. xn , H" :=`
`(hforall (fun x1 => .. (hforall (fun xn => H)) ..))`
`(at level 39, x1 binder, H at level 50, right associativity,`
`format "'[' '\forall' '/' x1 .. xn , '/' H ']'") : hprop_scope.`

The remaining operators are defined in terms of the previous above, rather than directly as functions over heaps. Doing so reduces the amount of proofs, by allowing to better leverage the tactic *xsimpl*.

```
Definition hpure (P:Prop) : hprop :=
  \∃ (p:P), \[].
```

```
Definition htop : hprop :=
  \∃ (H:hprop), H.
```

```
Definition hwand (H1 H2 : hprop) : hprop :=
  \∃ H0, H0 \* hpure ((H1 \* H0) ==> H2).
```

```
Definition qwand A (Q1 Q2:A→hprop) : hprop :=
  \∀ x, hwand (Q1 x) (Q2 x).
```

```
Notation "\[ P ]" := (hpure P)
  (at level 0, format "\[ P ]" : hprop_scope.
```

```
Notation "\Top" := (htop) : hprop_scope.
```

```
Notation "Q \*+ H" := (fun x => hstar (Q x) H)
  (at level 40) : hprop_scope.
```

```
Notation "H1 \-* H2" := (hwand H1 H2)
  (at level 43, right associativity) : hprop_scope.
```

```
Notation "Q1 \-* Q2" := (qwand Q1 Q2)
  (at level 43) : hprop_scope.
```

35.4.3 Basic Properties of Separation Logic Operators

Tactic for Automation

We set up *auto* to process goals of the form *h1* = *h2* by calling *fmap_eq*, which proves equality modulo associativity and commutativity.

```
#[global] Hint Extern 1 (_ = _ :> heap) => fmap_eq.
```

We also set up *auto* to process goals of the form *Fmap.disjoint* *h1* *h2* by calling the tactic *fmap_disjoint*, which essentially normalizes all disjointness goals and hypotheses, split all conjunctions, and invokes proof search with a base of hints specified in *LibSepFmap.v*.

```
Import FMAP.DISJOINTHINTS.
```

```
Tactic Notation "fmap_disjoint_pre" :=
  subst; rew_disjoint; jauto_set.
```

```
#[global] Hint Extern 1 (Fmap.disjoint _ _) => fmap_disjoint_pre.
```

Properties of *himpl* and *qimpl*

Lemma *himpl_refl* : $\forall H,$

```

    H ==> H.
Proof using. intros M. auto. Qed.
#[global] Hint Resolve himpl_refl.
Lemma himpl_trans :  $\forall H2\ H1\ H3,$ 
  (H1 ==> H2)  $\rightarrow$ 
  (H2 ==> H3)  $\rightarrow$ 
  (H1 ==> H3).
Proof using. intros M1 M2. unfolds  $\times$  himpl. Qed.
Lemma himpl_trans_r :  $\forall H2\ H1\ H3,$ 
  H2 ==> H3  $\rightarrow$ 
  H1 ==> H2  $\rightarrow$ 
  H1 ==> H3.
Proof using. intros M1 M2. applies  $\times$  himpl_trans M2 M1. Qed.
Lemma himpl_antisym :  $\forall H1\ H2,$ 
  (H1 ==> H2)  $\rightarrow$ 
  (H2 ==> H1)  $\rightarrow$ 
  (H1 = H2).
Proof using. intros M1 M2. applies pred_ext_1. intros h. iff  $\times$ . Qed.
Lemma hprop_op_comm :  $\forall (op:hprop \rightarrow hprop \rightarrow hprop),$ 
  ( $\forall H1\ H2, op\ H1\ H2 ==> op\ H2\ H1$ )  $\rightarrow$ 
  ( $\forall H1\ H2, op\ H1\ H2 = op\ H2\ H1$ ).
Proof using. intros M. intros. applies himpl_antisym; applies M. Qed.
Lemma qimpl_refl :  $\forall A\ (Q:A \rightarrow hprop),$ 
  Q ==> Q.
Proof using. intros. unfolds  $\times$ . Qed.
#[global] Hint Resolve qimpl_refl.

```

Properties of **hempty**

```

Lemma hempty_intro :
  \[] Fmap.empty.
Proof using. unfolds  $\times$ . Qed.
Lemma hempty_inv :  $\forall h,$ 
  \[] h  $\rightarrow$ 
  h = Fmap.empty.
Proof using. auto. Qed.

```

Properties of **hstar**

```

Lemma hstar_intro :  $\forall H1\ H2\ h1\ h2,$ 

```


$H1 \ h1 \rightarrow$
 $H2 \ h2 \rightarrow$
 $\text{Fmap.disjoint } h1 \ h2 \rightarrow$
 $(H1 \setminus * H2) (\text{Fmap.union } h1 \ h2).$
 Proof using. *intros. $\exists \neg h1 \ h2$. Qed.*

Lemma *hstar_inv* : $\forall H1 \ H2 \ h,$
 $(H1 \setminus * H2) \ h \rightarrow$
 $\exists h1 \ h2, H1 \ h1 \wedge H2 \ h2 \wedge \text{Fmap.disjoint } h1 \ h2 \wedge h = \text{Fmap.union } h1 \ h2.$
 Proof using. *introv M. applies M. Qed.*

Lemma *hstar_comm* : $\forall H1 \ H2,$
 $H1 \setminus * H2 = H2 \setminus * H1.$
 Proof using.
applies hprop_op_comm. unfold hprop, hstar. intros H1 H2.
intros h (h1&h2&M1&M2&D&U). rewrite $\neg$ Fmap.union_comm_of_disjoint in U.
 $\exists \times h2 \ h1$.
 Qed.

Lemma *hstar_assoc* : $\forall H1 \ H2 \ H3,$
 $(H1 \setminus * H2) \setminus * H3 = H1 \setminus * (H2 \setminus * H3).$
 Proof using.
intros H1 H2 H3. applies himpl_antisym; intros h.
{ intros (h'&h3&(h1&h2&M3&M4&D'&U')&M2&D&U). subst h'.
 $\exists h1 \ (h2 \setminus + h3).$ splits $\neg$. { applies $\times$ hstar_intro. } }
{ intros (h1&h'&M1&(h2&h3&M3&M4&D'&U')&D&U). subst h'.
 $\exists (h1 \setminus + h2) \ h3.$ splits $\neg$. { applies $\times$ hstar_intro. } }
 Qed.

Lemma *hstar_hempty_l* : $\forall H,$
 $\setminus [] \setminus * H = H.$
 Proof using.
intros. applies himpl_antisym; intros h.
{ intros (h1&h2&M1&M2&D&U). forwards E: hempty_inv M1. subst.
rewrite $\neg$ Fmap.union_empty_l. }
{ intros M. $\exists$ (Fmap.empty:heap) h. splits $\neg$. { applies hempty_intro. } }
 Qed.

Lemma *hstar_hempty_r* : $\forall H,$
 $H \setminus * \setminus [] = H.$
 Proof using.
applies neutral_r_of_comm_neutral_l. applies $\neg$ hstar_comm. applies $\neg$ hstar_hempty_l.
 Qed.

Lemma *hstar_hexists* : $\forall A \ (J:A \rightarrow \text{hprop}) \ H,$
 $(\text{hexists } J) \setminus * H = \text{hexists } (\text{fun } x \Rightarrow (J \ x) \setminus * H).$
 Proof using.

```

    intros. applies himpl_antisym; intros h.
    { intros (h1&h2&(x&M1)&M2&D&U).  $\exists \neg x$  h1 h2. }
    { intros (x&(h1&h2&M1&M2&D&U)).  $\exists h1 h2$ . splits  $\neg$ . {  $\exists \neg x$ . } }
Qed.

Lemma hstar_hforall :  $\forall H A (J:A \rightarrow \text{hprop})$ ,
  (hforall J) \* H ==> hforall (J \*+ H).
Proof using.
  intros. intros h M. destruct M as (h1&h2&M1&M2&D&U). intros x.  $\exists \neg h1 h2$ .
Qed.

Lemma himpl_frame_l :  $\forall H2 H1 H1'$ ,
  H1 ==> H1'  $\rightarrow$ 
  (H1 \* H2) ==> (H1' \* H2).
Proof using. introv W (h1&h2&?).  $\exists \times h1 h2$ . Qed.

Lemma himpl_frame_r :  $\forall H1 H2 H2'$ ,
  H2 ==> H2'  $\rightarrow$ 
  (H1 \* H2) ==> (H1 \* H2').
Proof using.
  introv M. do 2 rewrite (@hstar_comm H1). applies  $\neg$  himpl_frame_l.
Qed.

Lemma himpl_frame_lr :  $\forall H1 H1' H2 H2'$ ,
  H1 ==> H1'  $\rightarrow$ 
  H2 ==> H2'  $\rightarrow$ 
  (H1 \* H2) ==> (H1' \* H2').
Proof using.
  introv M1 M2. applies himpl_trans. applies  $\neg$  himpl_frame_l M1. applies  $\neg$  himpl_frame_r.
Qed.

Lemma himpl_hstar_trans_l :  $\forall H1 H2 H3 H4$ ,
  H1 ==> H2  $\rightarrow$ 
  H2 \* H3 ==> H4  $\rightarrow$ 
  H1 \* H3 ==> H4.
Proof using.
  introv M1 M2. applies himpl_trans_r M2. applies himpl_frame_l M1.
Qed.

Lemma himpl_hstar_trans_r :  $\forall H1 H2 H3 H4$ ,
  H1 ==> H2  $\rightarrow$ 
  H3 \* H2 ==> H4  $\rightarrow$ 
  H3 \* H1 ==> H4.
Proof using.
  introv M1 M2. applies himpl_trans_r M2. applies himpl_frame_r M1.
Qed.

```

Properties of hpure

Lemma hpure_intro : $\forall P,$
 $P \rightarrow$

$\backslash[P] \text{ Fmap.empty}.$

Proof using. *introv* $M. \exists M. \text{unfolds} \times. \text{Qed}.$

Lemma hpure_inv : $\forall P h,$
 $\backslash[P] h \rightarrow$

$P \wedge h = \text{Fmap.empty}.$

Proof using. *introv* $(p \& M). \text{split} \neg. \text{Qed}.$

Lemma hstar_hpure_l : $\forall P H h,$
 $(\backslash[P] \backslash * H) h = (P \wedge H h).$

Proof using.

intros. apply prop_ext. unfold hpure.

rewrite hstar_hexists. rewrite $\times$ hstar_hempty_l.

iff $(p \& M) (p \& M). \{ \text{split} \neg. \} \{ \exists \neg p. \}$

Qed.

Lemma hstar_hpure_r : $\forall P H h,$
 $(H \backslash * \backslash[P]) h = (H h \wedge P).$

Proof using.

intros. rewrite hstar_comm. rewrite hstar_hpure_l. apply $\times$ prop_ext.

Qed.

Lemma himpl_hstar_hpure_r : $\forall P H H',$
 $P \rightarrow$

$(H ==> H') \rightarrow$

$H ==> (\backslash[P] \backslash * H').$

Proof using. *introv HP W. intros h K. rewrite $\times$ hstar_hpure_l. Qed.*

Lemma hpure_inv_hempty : $\forall P h,$
 $\backslash[P] h \rightarrow$

$P \wedge \backslash[] h.$

Proof using.

introv M. rewrite $\leftarrow$ hstar_hpure_l. rewrite $\neg$ hstar_hempty_r.

Qed.

Lemma hpure_intro_hempty : $\forall P h,$
 $\backslash[] h \rightarrow$

$P \rightarrow$

$\backslash[P] h.$

Proof using.

introv M N. rewrite $\leftarrow$ $(\text{hstar\_hempty\_l} \backslash[P]). \text{rewrite} \neg \text{hstar\_hpure\_r}.$

Qed.

Lemma himpl_hempty_hpure : $\forall P,$

$P \rightarrow \backslash [] \Rightarrow \backslash [P]$.
 Proof using. *introv HP. intros h Hh. applies*× *hpure_intro_hempty. Qed.*
 Lemma *himpl_hstar_hpure_l* : $\forall P H H'$,
 $(P \rightarrow H \Rightarrow H') \rightarrow (\backslash [P] \backslash * H) \Rightarrow H'$.
 Proof using.
introv W Hh. rewrite hstar_hpure_l in Hh. applies× *W. Qed.*
 Lemma *hempty_eq_hpure_true* :
 $\backslash [] = \backslash [\text{True}]$.
 Proof using.
applies *himpl_antisym; intros h M.*
 $\{ \text{applies} \times \text{hpure\_intro\_hempty.} \}$
 $\{ \text{forwards}^*: \text{hpure\_inv\_hempty } M. \}$
Qed.
 Lemma *hfalse_hstar_any* : $\forall H$,
 $\backslash [\text{False}] \backslash * H = \backslash [\text{False}]$.
 Proof using.
intros. applies *himpl_antisym; intros h; rewrite hstar_hpure_l; intros M.*
 $\{ \text{false} \times. \} \{ \text{lets: hpure\_inv\_hempty } M. \text{false} \times. \}$
Qed.

Properties of **hexists**

Lemma *hexists_intro* : $\forall A (x:A) (J:A \rightarrow \text{hprop}) h$,
 $J x h \rightarrow (\text{hexists } J) h$.
 Proof using. *intros. $\exists \neg x$. Qed.*
 Lemma *hexists_inv* : $\forall A (J:A \rightarrow \text{hprop}) h$,
 $(\text{hexists } J) h \rightarrow \exists x, J x h$.
 Proof using. *intros. destruct H as [x H]. $\exists \neg x$. Qed.*
 Lemma *himpl_hexists_l* : $\forall A H (J:A \rightarrow \text{hprop})$,
 $(\forall x, J x \Rightarrow H) \rightarrow (\text{hexists } J) \Rightarrow H$.
 Proof using. *introv W. intros h (x&Hh). applies*× *W. Qed.*
 Lemma *himpl_hexists_r* : $\forall A (x:A) H J$,
 $(H \Rightarrow J x) \rightarrow H \Rightarrow (\text{hexists } J)$.
 Proof using. *introv W. intros h. $\exists x$. apply*¬ *W. Qed.*

Lemma `himpl_hexists` : $\forall A (J1\ J2:A \rightarrow \text{hprop}),$
 $J1 \implies J2 \rightarrow$
 $\text{hexists } J1 \implies \text{hexists } J2.$
Proof using.
`introv W. applies himpl_hexists_l. intros x. applies himpl_hexists_r W.`
Qed.

Properties of `hforall`

Lemma `hforall_intro` : $\forall A (J:A \rightarrow \text{hprop})\ h,$
 $(\forall x, J\ x\ h) \rightarrow$
 $(\text{hforall } J)\ h.$
Proof using. `introv M. applies  $\times$  M. Qed.`
Lemma `hforall_inv` : $\forall A (J:A \rightarrow \text{hprop})\ h,$
 $(\text{hforall } J)\ h \rightarrow$
 $\forall x, J\ x\ h.$
Proof using. `introv M. applies  $\times$  M. Qed.`
Lemma `himpl_hforall_r` : $\forall A (J:A \rightarrow \text{hprop})\ H,$
 $(\forall x, H \implies J\ x) \rightarrow$
 $H \implies (\text{hforall } J).$
Proof using. `introv M. intros h Hh x. apply  $\neg$  M. Qed.`
Lemma `himpl_hforall_l` : $\forall A\ x (J:A \rightarrow \text{hprop})\ H,$
 $(J\ x \implies H) \rightarrow$
 $(\text{hforall } J) \implies H.$
Proof using. `introv M. intros h Hh. apply  $\neg$  M. Qed.`
Lemma `hforall_specialize` : $\forall A\ (x:A) (J:A \rightarrow \text{hprop}),$
 $(\text{hforall } J) \implies (J\ x).$
Proof using. `intros. applies  $\times$  himpl_hforall_l x. Qed.`
Lemma `himpl_hforall` : $\forall A (J1\ J2:A \rightarrow \text{hprop}),$
 $J1 \implies J2 \rightarrow$
 $\text{hforall } J1 \implies \text{hforall } J2.$
Proof using.
`introv W. applies himpl_hforall_r. intros x. applies himpl_hforall_l W.`
Qed.

Properties of `hwand`

Lemma `hwand_equiv` : $\forall H0\ H1\ H2,$
 $(H0 \implies H1 \ \backslash\text{-}\ * \ H2) \leftrightarrow (H1 \ \backslash\text{-}\ * \ H0 \implies H2).$
Proof using.
`unfold hwand. iff M.`

```

{ rewrite hstar_comm. applies himpl_hstar_trans_l (rm M).
  rewrite hstar_hexists. applies himpl_hexists_l. intros H.
  rewrite (hstar_comm H). rewrite hstar_assoc.
  rewrite (hstar_comm H H1). applies  $\neg$  himpl_hstar_hpure_l. }
{ applies himpl_hexists_r H0.
  rewrite  $\leftarrow$  (hstar_empty_r H0) at 1.
  applies himpl_frame_r. applies himpl_empty_hpure M. }

```

Qed.

Lemma himpl_hwand_r : $\forall H1 H2 H3$,

```

H2 \* H1 ==> H3  $\rightarrow$ 
H1 ==> (H2 \-* H3).

```

Proof using. *intros M*. rewrite $\neg$ hwand_equiv. Qed.

Lemma himpl_hwand_r_inv : $\forall H1 H2 H3$,

```

H1 ==> (H2 \-* H3)  $\rightarrow$ 
H2 \* H1 ==> H3.

```

Proof using. *intros M*. rewrite $\neg \leftarrow$ hwand_equiv. Qed.

Lemma hwand_cancel : $\forall H1 H2$,

```

H1 \* (H1 \-* H2) ==> H2.

```

Proof using. intros. *applies* himpl_hwand_r_inv. *applies* himpl_refl. Qed.

Arguments hwand_cancel : clear *implicit*s.

Lemma himpl_empty_hwand_same : $\forall H$,

```

\[] ==> (H \-* H).

```

Proof using. intros. apply himpl_hwand_r. rewrite $\neg$ hstar_empty_r. Qed.

Lemma hwand_empty_l : $\forall H$,

```

(\[] \-* H) = H.

```

Proof using.

```

intros. applies himpl_antisym.
{ rewrite  $\leftarrow$  hstar_empty_l at 1. applies hwand_cancel. }
{ rewrite hwand_equiv. rewrite  $\neg$  hstar_empty_l. }

```

Qed.

Lemma hwand_hpure_l : $\forall P H$,

```

P  $\rightarrow$ 
(\[P] \-* H) = H.

```

Proof using.

```

intros HP. applies himpl_antisym.
{ lets K: hwand_cancel \[P] H. applies himpl_trans_r K.
  applies  $\times$  himpl_hstar_hpure_r. }
{ rewrite hwand_equiv. applies  $\times$  himpl_hstar_hpure_l. }

```

Qed.

Lemma hwand_curry : $\forall H1 H2 H3$,

```

    (H1 \-* H2) \-* H3 ==> H1 \-* (H2 \-* H3).
Proof using.
  intros. apply himpl_hwand_r. apply himpl_hwand_r.
  rewrite ← hstar_assoc. rewrite (hstar_comm H1 H2).
  applies hwand_cancel.

```

Qed.

```

Lemma hwand_uncurry : ∀ H1 H2 H3,
  H1 \-* (H2 \-* H3) ==> (H1 \-* H2) \-* H3.

```

Proof using.

```

  intros. rewrite hwand_equiv. rewrite (hstar_comm H1 H2).
  rewrite hstar_assoc. applies himpl_hstar_trans_r.
  { applies hwand_cancel. } { applies hwand_cancel. }

```

Qed.

```

Lemma hwand_curry_eq : ∀ H1 H2 H3,
  (H1 \-* H2) \-* H3 = H1 \-* (H2 \-* H3).

```

Proof using.

```

  intros. applies himpl_antisym.
  { applies hwand_curry. }
  { applies hwand_uncurry. }

```

Qed.

```

Lemma hwand_inv : ∀ h1 h2 H1 H2,
  (H1 \-* H2) h2 →
  H1 h1 →
  Fmap.disjoint h1 h2 →
  H2 (h1 \u h2).

```

Proof using.

```

  introv M2 M1 D. unfolds hwand. lets (H0&M3): hexists_inv M2.
  lets (h0&h3&P1&P3&D'&U): hstar_inv M3. lets (P4&E3): hpure_inv P3.
  subst h2 h3. rewrite union_empty_r in *. applies P4. applies × hstar_intro.

```

Qed.

Properties of qwand

```

Lemma qwand_equiv : ∀ H A (Q1 Q2:A→hprop),
  H ==> (Q1 \-* Q2) ↔ (Q1 \*+ H) ==> Q2.

```

Proof using.

```

  unfold qwand. iff M.
  { intros x. rewrite hstar_comm. applies himpl_hstar_trans_l (rm M).
    applies himpl_trans. applies hstar_hforall. simpl.
    applies himpl_hforall_l x. rewrite hstar_comm. applies hwand_cancel. }
  { applies himpl_hforall_r. intros x. rewrite × hwand_equiv. }

```

Qed.

Lemma qwand_cancel : $\forall A (Q1\ Q2:A \rightarrow \text{hprop}),$
 $Q1 \setminus *+ (Q1 \setminus -*\ Q2) ==> Q2.$
 Proof using. intros. rewrite $\leftarrow$ qwand_equiv. *applies* qimpl_refl. Qed.
 Lemma himpl_qwand_r : $\forall A (Q1\ Q2:A \rightarrow \text{hprop})\ H,$
 $Q1 \setminus *+ H ==> Q2 \rightarrow$
 $H ==> (Q1 \setminus -*\ Q2).$
 Proof using. *introv* *M*. rewrite $\neg$ qwand_equiv. Qed.
 Arguments himpl_qwand_r [A].
 Lemma qwand_specialize : $\forall A (x:A) (Q1\ Q2:A \rightarrow \text{hprop}),$
 $(Q1 \setminus -*\ Q2) ==> (Q1\ x \setminus -*\ Q2\ x).$
 Proof using. intros. *applies* $\times$ himpl_hforall_l *x*. Qed.
 Arguments qwand_specialize [A].

Properties of htop

Lemma htop_intro : $\forall h,$
 $\setminus \text{Top}\ h.$
 Proof using. intros. $\exists \neg (=h)$. Qed.
 Lemma himpl_htop_r : $\forall H,$
 $H ==> \setminus \text{Top}.$
 Proof using. intros. intros *h Hh*. *applies* $\times$ htop_intro. Qed.
 Lemma htop_eq :
 $\setminus \text{Top} = (\setminus \exists\ H,\ H).$
 Proof using. auto. Qed.
 Lemma hstar_htop_htop :
 $\setminus \text{Top} \setminus * \setminus \text{Top} = \setminus \text{Top}.$
 Proof using.
applies himpl_antisym.
 $\{ \text{applies himpl\_htop\_r. } \}$
 $\{ \text{rewrite } \leftarrow \text{ hstar\_hempty\_r at 1. } \text{applies himpl\_frame\_r. } \text{applies himpl\_htop\_r. } \}$
 Qed.

Properties of hsingle

Lemma hsingle_intro : $\forall p\ v,$
 $(p \sim\!\sim\!> v) (\text{Fmap.single}\ p\ v).$
 Proof using. intros. *hnfs* $\times$. Qed.
 Lemma hsingle_inv : $\forall p\ v\ h,$
 $(p \sim\!\sim\!> v)\ h \rightarrow$
 $h = \text{Fmap.single}\ p\ v.$

Proof using. auto. Qed.

Lemma hstar_hsingle_same_loc : $\forall p\ w1\ w2,$
 $(p \rightsquigarrow w1) \setminus * (p \rightsquigarrow w2) \implies \setminus [\text{False}].$

Proof using.

intros. unfold hsingle. intros $h\ (h1 \& h2 \& E1 \& E2 \& D \& E).$ false.

subst. *applies* $\times$ Fmap.disjoint_single_single_same_inv.

Qed.

Arguments hstar_hsingle_same_loc : clear *implicit*s.

Definition and Properties of **haffine** and **hgc**

Definition haffine (H :hprop) :=

True.

Lemma haffine_hany : $\forall (H$:hprop),
haffine H .

Proof using. unfold haffine. auto. Qed.

Lemma haffine_hempty :
haffine $\setminus []$.

Proof using. *applies* haffine_hany. Qed.

Definition hgc :=

htop.

Notation " $\setminus \text{GC}$ " := (hgc) : *hprop_scope*.

Lemma haffine_hgc :
haffine $\setminus \text{GC}$.

Proof using. *applies* haffine_hany. Qed.

Lemma himpl_hgc_r : $\forall H,$
haffine $H \rightarrow$
 $H \implies \setminus \text{GC}.$

Proof using. *intro* M . *applies* himpl_htop_r. Qed.

Lemma hstar_hgc_hgc :
 $\setminus \text{GC} \setminus * \setminus \text{GC} = \setminus \text{GC}.$

Proof using. *applies* hstar_htop_htop. Qed.

Functor Instantiation to Obtain *xsimpl*

End SEPSIMPLARGS.

We are now ready to instantiate the functor that defines *xsimpl*. Demos of *xsimpl* are presented in chapter *Himpl.v*.

Module Export HS := LIBSEPSIMPL.XSIMPLSETUP(SEPSIMPLARGS).

Export SEPSIMPLARGS.

From now on, all operators have opaque definitions.

Global Opaque hempty hpure hstar hsingle hexists hforall
hwand qwand htop hgc haffine.

At this point, the tactic *xsimpl* is defined. There remains to customize the tactic so that it recognizes the predicate $p \sim\sim v$ in a special way when performing simplifications.

The tactic *xsimpl_hook_hsingle* $p \ v$ operates as part of *xsimpl*. The specification that follows makes sense only in the context of the presentation of the invariants of *xsimpl* described in *LibSepSimpl.v*. This tactic is invoked on goals of the form $Xsimpl \ (Hla, Hlw, Hlt) \ HR$, where *Hla* is of the form $H1 \ \backslash^* \dots \ \backslash^* \ Hn \ \backslash^* \ \backslash[]$. The tactic *xsimpl_hook_hsingle* $p \ v$ searches among the *Hi* for a heap predicate of the form $p \ \sim\sim v'$. If it finds one, it moves this *Hi* to the front, just before *H1*. Then, it cancels it out with the $p \ \sim\sim v$ that occurs in *HR*. Otherwise, the tactic fails.

```
Ltac xsimpl_hook_hsingle p :=
  xsimpl_pick_st ltac:(fun H' =>
    match H' with (hsingle p ?v') =>
      constr:(true) end);
  apply xsimpl_lr_cancel_eq;
  [ xsimpl_lr_cancel_eq_repr_post tt | ].
```

The tactic *xsimpl_hook* handles cancellation of heap predicates of the form $p \ \sim\sim v$. It forces their cancellation against heap predicates of the form $p \ \sim\sim w$, by asserting the equality $v = w$. Note: this tactic is later refined to also handle records.

```
Ltac xsimpl_hook H ::=
  match H with
  | hsingle ?p ?v => xsimpl_hook_hsingle p
  end.
```

One last hack is to improve the *math* tactic so that it is able to handle the *val_int* coercion in goals and hypotheses of the form $val\_int \ ?n = val\_int \ ?m$, and so that it is able to process the well-founded relations *downto* and *upto* for induction on integers.

```
Ltac math_0 ::=
  unfolds downto, upto;
  repeat match goal with
  | ⊢ val_int _ = val_int _ => fequal
  | H: val_int _ = val_int _ ⊢ _ => inverts H
  end.
```

35.4.4 Properties of *haffine*

The lemma *haffine_any* asserts that *haffine* *H* holds for any *H*. Nevertheless, let us state corollaries of *haffine_any* for specific heap predicates, to illustrate how the *xaffine* tactic works in chapter *Affine*.

```

Lemma haffine_hempty :
  haffine \ [].
Proof using. intros. applies haffine_hany. Qed.

Lemma haffine_hpure :  $\forall P$ ,
  haffine \ [P].
Proof using. intros. applies haffine_hany. Qed.

Lemma haffine_hstar :  $\forall H1\ H2$ ,
  haffine H1  $\rightarrow$ 
  haffine H2  $\rightarrow$ 
  haffine (H1 \* H2).
Proof using. intros. applies haffine_hany. Qed.

Lemma haffine_hexists :  $\forall A\ (J:A\rightarrow\text{hprop})$ ,
  ( $\forall x$ , haffine (J x))  $\rightarrow$ 
  haffine (\exists x, (J x)).
Proof using. intros. applies haffine_hany. Qed.

Lemma haffine_hforall :  $\forall A\ \{\text{Inhab } A\}\ (J:A\rightarrow\text{hprop})$ ,
  ( $\forall x$ , haffine (J x))  $\rightarrow$ 
  haffine (\forall x, (J x)).
Proof using. intros. applies haffine_hany. Qed.

Lemma haffine_hstar_hpure :  $\forall (P:\text{Prop})\ H$ ,
  (P  $\rightarrow$  haffine H)  $\rightarrow$ 
  haffine (\[P] \* H).
Proof using. intros. applies haffine_hany. Qed.

Lemma haffine_hgc :
  haffine \ GC.
Proof using. intros. applies haffine_hany. Qed.

```

Using these lemmas, we are able to configure the *xaffine* tactic.

```

Ltac xaffine_core tt ::=
  repeat match goal with  $\vdash$  haffine ?H  $\Rightarrow$ 
    match H with
    | (hempty)  $\Rightarrow$  apply haffine_hempty
    | (hpure _)  $\Rightarrow$  apply haffine_hpure
    | (hstar _ _)  $\Rightarrow$  apply haffine_hstar
    | (hexists _)  $\Rightarrow$  apply haffine_hexists
    | (hforall _)  $\Rightarrow$  apply haffine_hforall
    | (hgc)  $\Rightarrow$  apply haffine_hgc
    | _  $\Rightarrow$  eauto with haffine
    end
  end.

```

35.5 Reasoning Rules

Implicit Types $P : \text{Prop}$.
 Implicit Types $H : \text{hprop}$.
 Implicit Types $Q : \text{val} \rightarrow \text{hprop}$.

35.5.1 Definition of Separation Logic Triples.

The Separation Logic triple $\text{triple } t \ H \ Q$ is defined in terms of the omni-big-step semantics predicate $\text{eval } s \ t \ Q$.

Definition $\text{triple } (t:\text{trm}) \ (H:\text{hprop}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{Prop} :=$
 $\forall s, H \ s \rightarrow \text{eval } s \ t \ Q$.

We introduce a handy notation for postconditions of functions that return a pointer: $\text{funloc } p \Rightarrow H$ is short for $\text{fun } (r:\text{val}) \Rightarrow \backslash \exists (p:\text{loc}), \backslash [r = \text{val\_loc } p] \backslash * H$.

Notation $"'\text{funloc}' \ p \ '=>' \ H" :=$
 $(\text{fun } (r:\text{val}) \Rightarrow \backslash \exists \ p, \backslash [r = \text{val\_loc } p] \backslash * H)$
 $(\text{at level } 200, \text{ } p \ \text{name}, \text{ format } "'\text{funloc}' \ p \ '=>' \ H") : \text{hprop\_scope}.$

35.5.2 Soundness of Triples w.r.t. the Small-Step Semantics

If one considers the omni-big-step semantics to be the starting point of a development, then the definition of triple is obviously sound with respect to that semantics, as it concludes $\text{eval } s \ t \ Q$. If, however, one considers the small-step semantics to be the starting point, then to establish soundness it is required to prove that $\text{triple } t \ H \ Q$ ensures that all possible evaluations of t in a heap satisfy H are: (1) terminating, (2) always reaching a value, and (3) this value and the final state obtained satisfy the postcondition Q .

The judgment $\text{terminates } s \ t$ is defined inductively. The execution starting from (s, t) terminates if, for any possible step leads to a configuration that terminates. Note that a configuration that has reached a value cannot take a step, hence is considered terminating.

Inductive $\text{terminates} : \text{heap} \rightarrow \text{trm} \rightarrow \text{Prop} :=$
 $| \text{terminates\_step} : \forall s \ t,$
 $(\forall s' \ t', \text{steps } s \ t \ s' \ t' \rightarrow \text{terminates } s' \ t') \rightarrow$
 $\text{terminates } s \ t.$

The judgment $\text{safe } s \ t$ asserts that no execution may reach a stuck term. In other words, for any configuration (s', t') reachable from (s, t) , it is the case that the configuration (s', t') is either a value or is reducible.

Definition $\text{safe } (s:\text{heap}) \ (t:\text{trm}) : \text{Prop} :=$
 $\forall s' \ t', \text{steps } s \ t \ s' \ t' \rightarrow \text{notstuck } s' \ t'.$

The judgment $\text{correct } s \ t \ Q$ asserts that if the execution of (s, t) reaches a final configuration, then this final configuration satisfies Q .

Definition correct (s :heap) (t :trm) (Q :val $\rightarrow$ hprop) : Prop :=
 $\forall s' v, \text{steps } s \ t \ s' \ v \rightarrow Q \ v \ s'.$

The aim is to show that $\text{triple } t \ H \ Q$ entails that, for any s satisfying H , $\text{terminates } s \ t$ and $\text{safe } s \ t$ and $\text{correct } s \ t \ Q$ holds.

The judgment $\text{seval } s \ t \ Q$ asserts that any execution of (s,t) terminates and reaches a configuration satisfying Q . In the “base” case, $\text{seval } s \ v \ Q$ holds if the terminal configuration (s,v) satisfies Q . In the “step” case, $\text{seval } s \ t \ Q$ holds if (1) the configuration (s,t) is reducible, and (2) if for any step that (s,t) may take to (s',t') , the predicate $\text{seval } s' \ t' \ Q$ holds.

Inductive $\text{seval} : \text{heap} \rightarrow \text{trm} \rightarrow (\text{val} \rightarrow \text{hprop}) \rightarrow \text{Prop} :=$
 $| \text{seval\_val} : \forall s \ v \ Q,$
 $\quad Q \ v \ s \rightarrow$
 $\quad \text{seval } s \ v \ Q$
 $| \text{seval\_step} : \forall s \ t \ Q,$
 $\quad \text{reducible } s \ t \rightarrow$
 $\quad (\forall s' \ t', \text{step } s \ t \ s' \ t' \rightarrow \text{seval } s' \ t' \ Q) \rightarrow$
 $\quad \text{seval } s \ t \ Q.$

The judgment $\text{seval } s \ t \ Q$ satisfies the 3 targeted soundness properties.

Lemma $\text{seval\_terminates} : \forall s \ t \ Q,$
 $\text{seval } s \ t \ Q \rightarrow$
 $\text{terminates } s \ t.$

Proof using.

$\text{intros } M. \text{induction } M; \text{constructors}; \text{intros } R.$
 $\{ \text{inverts } R. \}$
 $\{ \text{eauto.} \}$

Qed.

Lemma $\text{seval\_safe} : \forall s \ t \ Q,$
 $\text{seval } s \ t \ Q \rightarrow$
 $\text{safe } s \ t.$

Proof using.

$\text{intros } M \ R. \text{gen } Q. \text{induction } R; \text{intros.}$
 $\{ \text{inverts } M. \{ \text{left. hnfs} \times. \} \{ \text{right} \times. \} \}$
 $\{ \text{rename } H \text{ into } S. \text{inverts } M. \{ \text{inverts } S. \} \{ \text{applies} \times \text{IHR.} \} \}$

Qed.

Lemma $\text{seval\_correct} : \forall s \ t \ Q,$
 $\text{seval } s \ t \ Q \rightarrow$
 $\text{correct } s \ t \ Q.$

Proof using.

$\text{intros } M. \text{induction } M; \text{intros } R.$
 $\{ \text{inverts } R \text{ as. } \{ \text{auto.} \} \{ \text{intros } S. \text{inverts } S. \} \}$
 $\{ \text{rename } H1 \text{ into } IH. \text{inverts } R. \{ \text{lets } (? \& ? \& R): H. \text{inverts } R. \} \text{applies} \times \text{IH.} \}$

Qed.

false_step is a handy tactic to get rid of goals with assumptions of the form **step** *s v s'* *t'* or **step** *s (op v) s' t'* for a binary operator *op*.

```
Ltac false_step :=
  solve [ match goal with
  | K: step _ (trm_app (trm_val (val_prim _)) (trm_val _)) _ _ ⊢ _ ⇒
      inversion K; clear K; subst; false_step
  | K: step _ (trm_val _) _ _ ⊢ _ ⇒
      inversion K; clear K; subst
  | _ ⇒ false
  end ].
```

The proof that **eval** *s t Q* entails **seval** *s t Q* begins with a bunch of auxiliary lemmas.

```
Lemma seval_fun : ∀ s x t1 Q,
  Q (val_fun x t1) s →
  seval s (trm_fun x t1) Q.
```

Proof using.

```
  introv M. applies seval_step.
  { do 2 esplit. constructor. }
  { introv R. inverts R. { applies seval_val. applies M. } }
Qed.
```

```
Lemma seval_fix : ∀ s f x t1 Q,
  Q (val_fix f x t1) s →
  seval s (trm_fix f x t1) Q.
```

Proof using.

```
  introv M. applies seval_step.
  { do 2 esplit. constructor. }
  { introv R. inverts R. { applies seval_val. applies M. } }
Qed.
```

```
Lemma seval_app_fun : ∀ s x v1 v2 t1 Q,
  v1 = val_fun x t1 →
  seval s (subst x v2 t1) Q →
  seval s (trm_app v1 v2) Q.
```

Proof using.

```
  introv E M. applies seval_step.
  { do 2 esplit. applies × step_app_fun. }
  { introv R. invert R; try solve [intros; false | introv R; inverts R]. introv → → → →
  → R. inverts E. applies M. }
Qed.
```

```
Lemma seval_app_fix : ∀ s f x v1 v2 t1 Q,
  v1 = val_fix f x t1 →
  seval s (subst x v2 (subst f v1 t1)) Q →
  seval s (trm_app v1 v2) Q.
```

Proof using.

```

  introv E M. applys seval_step.
  { do 2 esplit. applys× step_app_fix. }
  { introv R. invert R; try solve [intros; false | introv R; inverts R]. introv → → → →
→ R. inverts E. applys M. }

```

Qed.

Lemma seval_seq : $\forall s\ t1\ t2\ Q1\ Q,$
seval $s\ t1\ Q1 \rightarrow$
 $(\forall s1\ v1, Q1\ v1\ s1 \rightarrow \text{seval}\ s1\ t2\ Q) \rightarrow$
seval $s\ (\text{trm\_seq}\ t1\ t2)\ Q.$

Proof using.

```

  introv M1 M2. gen_eq Q1': Q1.
  induction M1; intros; subst.
  { apply seval_step.
    { do 2 esplit. applys step_seq. }
    { introv R. inverts R as R'. { inverts R'. } applys× M2. } }
  { rename t into t1, H1 into IH.
    destruct H as (s'&t'&RE). applys seval_step.
    { do 2 esplit. constructors. applys RE. }
    { introv R. inverts R as R'.
      { applys× IH R' M2. }
      { false. inverts RE. } } }

```

Qed.

Lemma seval_let : $\forall s\ x\ t1\ t2\ Q1\ Q,$
seval $s\ t1\ Q1 \rightarrow$
 $(\forall s1\ v1, Q1\ v1\ s1 \rightarrow \text{seval}\ s1\ (\text{subst}\ x\ v1\ t2)\ Q) \rightarrow$
seval $s\ (\text{trm\_let}\ x\ t1\ t2)\ Q.$

Proof using.

```

  introv M1 M2. gen_eq Q1': Q1.
  induction M1; intros; subst.
  { apply seval_step.
    { do 2 esplit. applys step_let. }
    { introv R. inverts R as R'. { inverts R'. } applys× M2. } }
  { rename t into t1, H1 into IH.
    destruct H as (s'&t'&RE). applys seval_step.
    { do 2 esplit. constructors. applys RE. }
    { introv R. inverts R as R'.
      { applys× IH R' M2. }
      { false. inverts RE. } } }

```

Qed.

Lemma seval_app_arg1 : $\forall t1\ t2\ Q1\ s1\ Q,$
seval $s1\ t1\ Q1 \rightarrow$

$(\forall v1\ s2, Q1\ v1\ s2 \rightarrow \mathbf{seval}\ s2\ (v1\ t2)\ Q) \rightarrow$
 $\mathbf{seval}\ s1\ (t1\ t2)\ Q.$

Proof using.

```

intros M1 M2. gen_eq Q1': Q1.
induction M1; intros; subst.
{ applys× M2. }
{ rename t into t1, H1 into IH.
  destruct H as (s'&t'&RE). applys seval_step.
  { do 2 esplit. applys step_app_arg1. applys RE. }
  { intros R. inverts R as R'; try solve [inverts RE]; try false_step.
    { applys× IH R'. } } }
```

Qed.

Lemma *seval_app_arg2* : $\forall v1\ t2\ Q1\ s1\ Q,$
 $\mathbf{seval}\ s1\ t2\ Q1 \rightarrow$
 $(\forall v2\ s2, Q1\ v2\ s2 \rightarrow \mathbf{seval}\ s2\ (v1\ v2)\ Q) \rightarrow$
 $\mathbf{seval}\ s1\ (v1\ t2)\ Q.$

Proof using.

```

intros M1 M2. gen_eq Q1': Q1.
induction M1; intros; subst.
{ applys× M2. }
{ rename t into t1, H1 into IH.
  destruct H as (s'&t'&RE). applys seval_step.
  { do 2 esplit. applys step_app_arg2. applys RE. }
  { intros R. inverts R as R'; try solve [inverts RE]; try false_step.
    { applys× IH R'. } } }
```

Qed.

Lemma *seval_if* : $\forall s\ b\ t1\ t2\ Q,$
 $\mathbf{seval}\ s\ (\mathbf{if}\ b\ \mathbf{then}\ t1\ \mathbf{else}\ t2)\ Q \rightarrow$
 $\mathbf{seval}\ s\ (\mathbf{trm\_if}\ b\ t1\ t2)\ Q.$

Proof using.

```

intros M. applys seval_step.
{ do 2 esplit. constructors×. }
{ intros R. inverts R; try false. { applys M. } }
```

Qed.

Lemma *seval_of_eval* : $\forall s\ t\ Q,$
 $\mathbf{eval}\ s\ t\ Q \rightarrow$
 $\mathbf{seval}\ s\ t\ Q.$

Proof using.

```

intros M. induction M.
{ applys× seval_val. }
{ applys× seval_fun. }
{ applys× seval_fix. }
```



```

{ applies × seval_app_arg1. }
{ applies × seval_app_arg2. }
{ applies × seval_app_fun. }
{ applies × seval_app_fix. }
{ applies × seval_seq. }
{ applies × seval_let. }
{ applies × seval_if. }
{ rename H into HE, H0 into K. unfolds purepostin.
  applies seval_step.
  { inverts HE; try solve [do 2 esplit; constructor; eauto].
    {  $\exists s\ 0. \text{applies step\_rand. math.}$  } }
  { introv R. inverts R; try false_step; inverts HE; applies × seval_val. } }
{ rename H into HE, H0 into K. unfolds purepostin.
  applies seval_step.
  { inverts HE; try solve [do 2 esplit; constructor; eauto]. }
  { introv R. inverts R; do 2 (try inverts_if_head step); inverts HE; applies × seval_val.
    { math_rewrite (p2 = p3). { applies eq_nat_of_eq_int. math. } eauto. } } }
{ applies seval_step.
  { forwards  $\neg (p \& D \& N)$ : (exists_fresh null s).
    do 2 esplit. applies × step_ref. }
  { introv R. do 2 (try inverts_if_head step). applies × seval_val. } }
{ applies seval_step.
  { do 2 esplit. applies × step_get. }
  { introv R. do 2 (try inverts_if_head step). applies × seval_val. } }
{ applies seval_step.
  { do 2 esplit. applies × step_set. }
  { introv R. do 3 (try inverts_if_head step). applies × seval_val. } }
{ applies seval_step.
  { do 2 esplit. applies × step_free. }
  { introv R. do 2 (try inverts_if_head step). applies × seval_val. } }

```

Qed.

Final soundness theorem with respect to the small-step semantics.

Lemma *triple_sound* : $\forall t\ H\ Q$,

triple t H Q $\rightarrow$

$\forall s, H\ s \rightarrow \text{terminates } s\ t \wedge \text{safe } s\ t \wedge \text{correct } s\ t\ Q$.

Proof using.

introv M Hs. specializes M Hs. lets M': seval_of_eval M. splits.

{ *applies* × *seval_terminates*. }

{ *applies* × *seval_safe*. }

{ *applies* × *seval_correct*. }

Qed.

35.5.3 Structural Properties of the `eval` Predicate

Section `EvalProp`.

Hint Constructors `eval`.

Covariance property

```
Lemma eval_conseq : ∀ s t Q1 Q2,
  eval s t Q1 →
  Q1 ==> Q2 →
  eval s t Q2.
```

Proof using.

```
  introv M W.
  asserts W': (∀ v h, Q1 v h → Q2 v h). { auto. } clear W.
  induction M; try solve [ constructors× ].
  { applies× eval_unop. unfolds× purepostin. }
  { applies× eval_binop. unfolds× purepostin. }
```

Qed.

Frame property

```
Lemma eval_frame : ∀ h1 h2 t Q,
  eval h1 t Q →
  Fmap.disjoint h1 h2 →
  eval (h1 \u h2) t (Q \*+ (= h2)).
```

Proof using.

```
  introv M HD. gen h2. induction M; intros;
  try solve [ hint hstar_intro; constructors× ].
  { rename M into M1, H into M2, IHM into IH1, H1 into IH2.
    specializes IH1 HD. applies× eval_app_arg1 IH1. introv HK.
    lets (h1'&h2'&K1'&K2'&KD&KU): hstar_inv HK. subst. applies× IH2. }
  { rename M into M1, H into M2, IHM into IH1, H1 into IH2.
    specializes IH1 HD. applies× eval_app_arg2 IH1. introv HK.
    lets (h1'&h2'&K1'&K2'&KD&KU): hstar_inv HK. subst. applies× IH2. }
  { rename M into M1, H into M2, IHM into IH1, H0 into IH2.
    specializes IH1 HD. applies eval_seq IH1. introv HK.
    lets (h1'&h2'&K1'&K2'&KD&KU): hstar_inv HK. subst. applies× IH2. }
  { rename M into M1, H into M2, IHM into IH1, H0 into IH2.
    specializes IH1 HD. applies eval_let IH1. introv HK.
    lets (h1'&h2'&K1'&K2'&KD&KU): hstar_inv HK. subst. applies× IH2. }
  { applies× eval_unop. unfolds× purepostin. introv Hv. applies× hstar_intro. }
  { applies× eval_binop. unfolds× purepostin. introv Hv. applies× hstar_intro. }
  { rename H into M. applies eval_ref. intros p Hp.
    rewrite Fmap.indom_union_eq in Hp. rew_logic in Hp. destruct Hp as [Hp1 Hp2].
    rewrite× Fmap.update_union_not_r. applies hstar_intro.
    { applies× M. } { auto. } { applies× Fmap.disjoint_update_not_r. } }
```

```

{ applies eval_get. { rewrite× Fmap.indom_union_eq. }
  { rewrite× Fmap.read_union_l. applies× hstar_intro. } }
{ applies eval_set. { rewrite× Fmap.indom_union_eq. }
  { rewrite× Fmap.update_union_l. applies hstar_intro.
    { auto. } { auto. } { applies× Fmap.disjoint_update_l. } } }
{ applies eval_free. { rewrite× Fmap.indom_union_eq. }
  { rewrite× Fmap.remove_disjoint_union_l. applies hstar_intro.
    { auto. } { auto. } { applies× Fmap.disjoint_remove_l. } } }

```

Qed.

End EvalProp.

35.5.4 Structural Rules

Consequence and frame rule.

Lemma triple_conseq : $\forall t H' Q' H Q$,
 triple $t H' Q' \rightarrow$
 $H \implies H' \rightarrow$
 $Q' \implies Q \rightarrow$
 triple $t H Q$.

Proof using. *unfolds* triple. *intros* $M MH MQ HF$. *applies*× eval_conseq. Qed.

Lemma triple_frame : $\forall t H Q H'$,
 triple $t H Q \rightarrow$
 triple $t (H \setminus * H') (Q \setminus * H')$.

Proof.

intros M . *intros* $h HF$. *lets* $(h1\&h2\&M1\&M2\&MD\&MU)$: hstar_inv (rm HF).
subst. *specializes* $M M1$. *applies* eval_conseq.
 $\{ \text{*applies* eval_frame } M MD. \} \{ \text{*xsimpl*. *intros* } h' \rightarrow. \text{*applies* } M2. \}$

Qed.

Extraction Rules

Lemma triple_hpure : $\forall t (P:\text{Prop}) H Q$,
 $(P \rightarrow \text{triple } t H Q) \rightarrow$
 triple $t (\setminus [P] \setminus * H) Q$.

Proof using.

intros M . *intros* $h (h1\&h2\&M1\&M2\&D\&U)$. *destruct* $M1$ as $(M1\&HP)$.
lets E : hempty_inv HP . *subst*. *rewrite* Fmap.union_empty_l. *applies* $\neg M$.

Qed.

Lemma triple_hexists : $\forall t (A:\text{Type}) (J:A\rightarrow\text{hprop}) Q$,
 $(\forall (x:A), \text{triple } t (J x) Q) \rightarrow$
 triple $t (\text{hexists } J) Q$.

Proof using. *intros* M . *intros* $h (x\&Hh)$. *applies* $M Hh$. Qed.

Lemma triple_hforall : $\forall t (A:\text{Type}) (x:A) (J:A\rightarrow\text{hprop}) Q$,

triple t ($J\ x$) $Q \rightarrow$
triple t ($\text{hforall } J$) Q .

Proof using. *intros* M . *applies* $\times$ triple_conseq M . *applies* hforall_specialize. Qed.

Lemma triple_hwand_hpure_l : $\forall t\ (P:\text{Prop})\ H\ Q,$
 $P \rightarrow$

triple $t\ H\ Q \rightarrow$
triple $t\ (\backslash[P]\ \backslash-*\ H)\ Q$.

Proof using. *intros* $HP\ M$. *applies* $\times$ triple_conseq M . *rewrite* $\times$ hwand_hpure_l. Qed.

A corollary of triple_hpure useful for the course

Parameter triple_hpure' : $\forall t\ (P:\text{Prop})\ Q,$
 $(P \rightarrow \text{triple } t\ \backslash[]\ Q) \rightarrow$
triple $t\ \backslash[P]\ Q$.

Heap-naming rule

Lemma triple_named_heap : $\forall t\ H\ Q,$
 $(\forall h, H\ h \rightarrow \text{triple } t\ (=h)\ Q) \rightarrow$
triple $t\ H\ Q$.

Proof using. *intros* $M\ Hs$. *applies* $M\ Hs$. *auto*. Qed.

Combined and ramified rules.

Lemma triple_conseq_frame : $\forall H2\ H1\ Q1\ t\ H\ Q,$
triple $t\ H1\ Q1 \rightarrow$
 $H \implies H1\ \backslash*\ H2 \rightarrow$
 $Q1\ \backslash*+\ H2 \implies Q \rightarrow$
triple $t\ H\ Q$.

Proof using.

intros $M\ WH\ WQ$. *applies* triple_conseq $WH\ WQ$. *applies* triple_frame M .

Qed.

Lemma triple_ramified_frame : $\forall H1\ Q1\ t\ H\ Q,$
triple $t\ H1\ Q1 \rightarrow$
 $H \implies H1\ \backslash*\ (Q1\ \backslash-*\ Q) \rightarrow$
triple $t\ H\ Q$.

Proof using.

intros $M\ W$. *applies* triple_conseq_frame $(Q1\ \backslash-*\ Q)\ M\ W$.
{ *rewrite* $\neg \leftarrow$ qwand_equiv. }

Qed.

35.5.5 Rules for Terms

Lemma triple_eval_like : $\forall t1\ t2\ H\ Q,$
eval_like $t1\ t2 \rightarrow$
triple $t1\ H\ Q \rightarrow$

```

triple t2 H Q.
Proof using. introv E M1 Hv. applies× E. Qed.

Lemma triple_val : ∀ v H Q,
  H ==> Q v →
  triple (trm_val v) H Q.
Proof using. introv M Hs. applies× eval_val. Qed.

Lemma triple_val_minimal : ∀ v,
  triple (trm_val v) \[] (fun r => \[r = v]).
Proof using. intros. applies triple_val. xsimpl×. Qed.

Lemma triple_fun : ∀ x t1 H Q,
  H ==> Q (val_fun x t1) →
  triple (trm_fun x t1) H Q.
Proof using. introv M Hs. applies× eval_fun. Qed.

Lemma triple_fix : ∀ f x t1 H Q,
  H ==> Q (val_fix f x t1) →
  triple (trm_fix f x t1) H Q.
Proof using. introv M Hs. applies× eval_fix. Qed.

Lemma triple_seq : ∀ t1 t2 H Q H1,
  triple t1 H (fun v => H1) →
  triple t2 H1 Q →
  triple (trm_seq t1 t2) H Q.
Proof using. introv M1 M2 Hs. applies× eval_seq. Qed.

Lemma triple_let : ∀ x t1 t2 Q1 H Q,
  triple t1 H Q1 →
  (∀ v1, triple (subst x v1 t2) (Q1 v1) Q) →
  triple (trm_let x t1 t2) H Q.
Proof using. introv M1 M2 Hs. applies× eval_let. Qed.

Lemma triple_let_val : ∀ x v1 t2 H Q,
  triple (subst x v1 t2) H Q →
  triple (trm_let x v1 t2) H Q.
Proof using.
  introv M. applies triple_let (fun v => \[v = v1] \* H).
  { applies triple_val. xsimpl×. }
  { intros v. applies triple_hpure. intros →. applies M. }
Qed.

Lemma triple_if : ∀ (b:bool) t1 t2 H Q,
  triple (if b then t1 else t2) H Q →
  triple (trm_if b t1 t2) H Q.
Proof using. introv M Hs. applies× eval_if. Qed.

Lemma triple_app_fun : ∀ x v1 v2 t1 H Q,

```

```

    v1 = val_fun x t1 →
    triple (subst x v2 t1) H Q →
    triple (trm_app v1 v2) H Q.
Proof using. intros E M Hs. applies× eval_app_fun. Qed.

Lemma triple_app_fun_direct : ∀ x v2 t1 H Q,
  triple (subst x v2 t1) H Q →
  triple (trm_app (val_fun x t1) v2) H Q.
Proof using. intros M. applies× triple_app_fun. Qed.

Lemma triple_app_fix : ∀ v1 v2 f x t1 H Q,
  v1 = val_fix f x t1 →
  triple (subst x v2 (subst f v1 t1)) H Q →
  triple (trm_app v1 v2) H Q.
Proof using. intros E M Hs. applies× eval_app_fix. Qed.

Lemma triple_app_fix_direct : ∀ v2 f x t1 H Q,
  f ≠ x →
  triple (subst x v2 (subst f (val_fix f x t1) t1)) H Q →
  triple (trm_app (val_fix f x t1) v2) H Q.
Proof using. intros N M. applies× triple_app_fix. Qed.

```

35.5.6 Rules for Heap-Manipulating Primitive Operations

```

Lemma triple_ref : ∀ v,
  triple (val_ref v)
  \ []
  (fun r ⇒ \∃ p, \[r = val_loc p] \* p ~~> v).
Proof using.
  intros. intros s1 K. applies eval_ref. intros p D.
  lets ->: hempty_inv K. rewrite Fmap.update_empty.
  applies hexists_intro p. rewrite hstar_hpure_l. split×.
  applies hsingle_intro.
Qed.

Lemma triple_get : ∀ v p,
  triple (val_get p)
  (p ~~> v)
  (fun r ⇒ \[r = v] \* (p ~~> v)).
Proof using.
  intros. intros s K. lets ->: hsingle_inv K.
  applies eval_get.
  { applies× Fmap.indom_single. }
  { rewrite hstar_hpure_l. split×. rewrite× Fmap.read_single. }
Qed.

```

Lemma triple_set : $\forall w\ p\ v,$
 triple (val_set (val_loc p) v)
 ($p \sim\sim> w$)
 (fun $r \Rightarrow \backslash[r = \text{val\_unit}] \backslash* (p \sim\sim> v)$).
 Proof using.
 intros. intros $s1\ K$. lets ->: hsingle_inv K . lets Hp : indom_single $p\ v$.
 applies eval_set.
 { applies $\times$ Fmap.indom_single. }
 { rewrite hstar_hpure_l. split $\times$. rewrite Fmap.update_single. applies hsingle_intro. }
 Qed.
 Lemma triple_free' : $\forall p\ v,$
 triple (val_free (val_loc p))
 ($p \sim\sim> v$)
 (fun $r \Rightarrow \backslash[r = \text{val\_unit}]$).
 Proof using.
 intros. intros $s1\ K$. lets ->: hsingle_inv K . lets Hp : indom_single $p\ v$.
 applies eval_free.
 { applies $\times$ Fmap.indom_single. }
 { rewrite $\times$ Fmap.remove_single. applies $\times$ hpure_intro. }
 Qed.
 Lemma triple_free : $\forall p\ v,$
 triple (val_free (val_loc p))
 ($p \sim\sim> v$)
 (fun $r \Rightarrow \backslash[]$).
 Proof using. intros. applies triple_conseq triple_free'; xsimpl $\times$. Qed.

35.5.7 Rules for Other Primitive Operations

Lemma triple_unop' : $\forall op\ v1\ (P:\text{val} \rightarrow \text{Prop})\ Q,$
 evalunop $op\ v1\ P \rightarrow$
 ($\forall v, P\ v \rightarrow Q\ v\ \text{Fmap.empty}$) $\rightarrow$
 triple ($op\ v1$) $\backslash[]\ Q$.
 Proof using.
 intro $M\ R\ Hs$. lets ->: hempty_inv Hs . applies eval_unop M . hnfs $\times$.
 Qed.
 Lemma triple_binop' : $\forall op\ v1\ v2\ (P:\text{val} \rightarrow \text{Prop})\ Q,$
 evalbinop $op\ v1\ v2\ P \rightarrow$
 ($\forall v, P\ v \rightarrow Q\ v\ \text{Fmap.empty}$) $\rightarrow$
 triple ($op\ v1\ v2$) $\backslash[]\ Q$.
 Proof using.
 intro $M\ R\ Hs$. lets ->: hempty_inv Hs . applies eval_binop M . hnfs $\times$.
 Qed.

Lemma triple_unop : $\forall op\ v1\ (P:\text{val} \rightarrow \text{Prop}),$
evalunop $op\ v1\ P \rightarrow$
triple ($op\ v1$) $\backslash []$ ($\text{fun } r \Rightarrow \backslash [P\ r]$).

Proof using.

introv $M\ Hs.$ *lets* $->:$ **hempty_inv** $Hs.$
applies **eval_conseq** ($\text{fun } v\ s \Rightarrow P\ v \wedge s = \text{Fmap.empty}$).
{ *applies* $\times$ **eval_unop** $M.$ *hnfs* $\times.$ }
{ *intros* $v\ s$ ($? \& ->$). *applies* $\times$ **hpure_intro**. }

Qed.

Lemma triple_binop : $\forall op\ v1\ v2\ (P:\text{val} \rightarrow \text{Prop}),$
evalbinop $op\ v1\ v2\ P \rightarrow$
triple ($op\ v1\ v2$) $\backslash []$ ($\text{fun } r \Rightarrow \backslash [P\ r]$).

Proof using.

introv $M\ Hs.$ *lets* $->:$ **hempty_inv** $Hs.$
applies **eval_conseq** ($\text{fun } v\ s \Rightarrow P\ v \wedge s = \text{Fmap.empty}$).
{ *applies* $\times$ **eval_binop** $M.$ *hnfs* $\times.$ }
{ *intros* $v\ s$ ($? \& ->$). *applies* $\times$ **hpure_intro**. }

Qed.

Lemma triple_add : $\forall n1\ n2,$
triple (**val_add** $n1\ n2$)
 $\backslash []$
($\text{fun } r \Rightarrow \backslash [r = \text{val\_int } (n1 + n2)]$).

Proof using. *intros.* *applies* $\times$ triple_binop. *applies* $\times$ evalbinop_add. Qed.

Lemma triple_div : $\forall n1\ n2,$
 $n2 \neq 0 \rightarrow$
triple (**val_div** $n1\ n2$)
 $\backslash []$
($\text{fun } r \Rightarrow \backslash [r = \text{val\_int } (\text{Z.quot } n1\ n2)]$).

Proof using. *intros.* *applies* $\times$ triple_binop. *applies* $\times$ evalbinop_div. Qed.

Lemma triple_rand : $\forall n,$
 $n > 0 \rightarrow$
triple (**val_rand** n)
 $\backslash []$
($\text{fun } r \Rightarrow \backslash [\exists n1, r = \text{val\_int } n1 \wedge 0 \leq n1 < n]$).

Proof using. *intros.* *applies* $\times$ triple_unop. *applies* $\times$ evalunop_rand. Qed.

Lemma triple_neg : $\forall (b1:\text{bool}),$
triple (**val_neg** $b1$)
 $\backslash []$
($\text{fun } r \Rightarrow \backslash [r = \text{val\_bool } (\text{neg } b1)]$).

Proof using. *intros.* *applies* $\times$ triple_unop. *applies* $\times$ evalunop_neg. Qed.

Lemma triple_opp : $\forall n1,$


```

triple (val_opp n1)
  \ []
  (fun r ⇒ \[r = val_int (- n1)]).
Proof using. intros. appls× triple_unop. appls× evalunop_opp. Qed.
Lemma triple_eq : ∀ v1 v2,
  triple (val_eq v1 v2)
    \ []
    (fun r ⇒ \[r = isTrue (v1 = v2)]).
Proof using. intros. appls× triple_binop. appls evalbinop_eq. Qed.
Lemma triple_neq : ∀ v1 v2,
  triple (val_neq v1 v2)
    \ []
    (fun r ⇒ \[r = isTrue (v1 ≠ v2)]).
Proof using. intros. appls× triple_binop. appls evalbinop_neq. Qed.
Lemma triple_sub : ∀ n1 n2,
  triple (val_sub n1 n2)
    \ []
    (fun r ⇒ \[r = val_int (n1 - n2)]).
Proof using. intros. appls× triple_binop. appls× evalbinop_sub. Qed.
Lemma triple_mul : ∀ n1 n2,
  triple (val_mul n1 n2)
    \ []
    (fun r ⇒ \[r = val_int (n1 × n2)]).
Proof using. intros. appls× triple_binop. appls× evalbinop_mul. Qed.
Lemma triple_mod : ∀ n1 n2,
  n2 ≠ 0 →
  triple (val_mod n1 n2)
    \ []
    (fun r ⇒ \[r = val_int (Z.rem n1 n2)]).
Proof using. intros. appls× triple_binop. appls× evalbinop_mod. Qed.
Lemma triple_le : ∀ n1 n2,
  triple (val_le n1 n2)
    \ []
    (fun r ⇒ \[r = isTrue (n1 ≤ n2)]).
Proof using. intros. appls× triple_binop. appls× evalbinop_le. Qed.
Lemma triple_lt : ∀ n1 n2,
  triple (val_lt n1 n2)
    \ []
    (fun r ⇒ \[r = isTrue (n1 < n2)]).
Proof using. intros. appls× triple_binop. appls× evalbinop_lt. Qed.
Lemma triple_ge : ∀ n1 n2,

```

```

triple (val_ge n1 n2)
  \ []
  (fun r => \[r = isTrue (n1 ≥ n2)]).
Proof using. intros. applies× triple_binop. applies× evalbinop_ge. Qed.

Lemma triple_gt : ∀ n1 n2,
  triple (val_gt n1 n2)
    \ []
    (fun r => \[r = isTrue (n1 > n2)]).
Proof using. intros. applies× triple_binop. applies× evalbinop_gt. Qed.

Lemma triple_ptr_add : ∀ p n,
  p + n ≥ 0 →
  triple (val_ptr_add p n)
    \ []
    (fun r => \[r = val_loc (abs (p + n))]).
Proof using.
  intros. applies× triple_binop. applies× evalbinop_ptr_add.
  { rewrite¬ abs_nonneg. }
Qed.

Lemma triple_ptr_add_nat : ∀ p (f:nat),
  triple (val_ptr_add p f)
    \ []
    (fun r => \[r = val_loc (p+f)%nat]).
Proof using.
  intros. applies triple_conseq triple_ptr_add. { math. } { xsimpl. }
  { xsimpl. intros. subst. fequals.
    applies eq_nat_of_eq_int. rewrite abs_nonneg; math. }
Qed.

```

Structural Rule for wp

Definition of wp

```

Definition wp (t:trm) (Q:val→hprop) : hprop :=
  fun s => eval s t Q.

```

Equivalence between wp and triple

```

Lemma wp_equiv : ∀ t H Q,
  (H ==> wp t Q) ↔ (triple t H Q).
Proof using. intros. unfold wp, triple. iff×. Qed.

```

Consequence rule for wp.

```

Lemma wp_conseq : ∀ t Q1 Q2,
  Q1 ==> Q2 →

```

$\text{wp } t \ Q1 \implies \text{wp } t \ Q2$.
 Proof using. unfold wp. *intros* M . *intros* $s \ Hs$. *applies* $\times$ eval_conseq. Qed.

Frame rule for wp.

Lemma wp_frame : $\forall t \ H \ Q$,
 $(\text{wp } t \ Q) \setminus * H \implies \text{wp } t \ (Q \setminus *+ H)$.

Proof using.

intros. unfold wp. *intros* $h \ HF$.
lets $(h1 \& h2 \& M1 \& M2 \& MD \& MU)$: *hstar_inv* (rm HF).
subst. *applies* eval_conseq.
 $\{ \text{applies eval\_frame } M1 \ MD. \}$
 $\{ \text{xsimpl. intros } h' \rightarrow. \text{applies } M2. \}$

Qed.

Corollaries, including ramified frame rule.

Lemma wp_ramified : $\forall t \ Q1 \ Q2$,
 $(\text{wp } t \ Q1) \setminus * (Q1 \setminus -* Q2) \implies (\text{wp } t \ Q2)$.

Proof using.

intros. *applies* himpl_trans.
 $\{ \text{applies wp\_frame. } \}$
 $\{ \text{applies wp\_conseq. xsimpl. } \}$

Qed.

Lemma wp_conseq_frame : $\forall t \ H \ Q1 \ Q2$,
 $Q1 \setminus *+ H \implies Q2 \rightarrow$
 $(\text{wp } t \ Q1) \setminus * H \implies (\text{wp } t \ Q2)$.

Proof using.

intros M . *rewrite* $\leftarrow$ *qwand_equiv* in M . *xchange* M . *applies* wp_ramified.

Qed.

Lemma wp_ramified_trans : $\forall t \ H \ Q1 \ Q2$,
 $H \implies (\text{wp } t \ Q1) \setminus * (Q1 \setminus -* Q2) \rightarrow$
 $H \implies (\text{wp } t \ Q2)$.

Proof using. *intros* M . *xchange* M . *applies* wp_ramified. Qed.

Weakest-Precondition Style Reasoning Rules for Terms.

Lemma wp_eval_like : $\forall t1 \ t2 \ Q$,
 $\text{eval\_like } t1 \ t2 \rightarrow$
 $\text{wp } t1 \ Q \implies \text{wp } t2 \ Q$.

Proof using. *intros* $E \ Hs$. *applies* $\times$ $E \ Hs$. Qed.

Lemma wp_val : $\forall v \ Q$,
 $Q \ v \implies \text{wp } (\text{trm\_val } v) \ Q$.

Proof using. unfold wp. *intros*. *intros* $h \ K$. *applies* $\times$ eval_val. Qed.

Lemma wp_fun : $\forall x t Q,$
 $Q (\text{val\_fun } x t) \Rightarrow \text{wp } (\text{trm\_fun } x t) Q.$
 Proof using. unfold wp. intros. intros $h K$. *applies* $\times$ eval_fun. Qed.

Lemma wp_fix : $\forall f x t Q,$
 $Q (\text{val\_fix } f x t) \Rightarrow \text{wp } (\text{trm\_fix } f x t) Q.$
 Proof using. unfold wp. intros. intros $h K$. *applies* $\times$ eval_fix. Qed.

Lemma wp_app_fun : $\forall x v1 v2 t1 Q,$
 $v1 = \text{val\_fun } x t1 \rightarrow$
 $\text{wp } (\text{subst } x v2 t1) Q \Rightarrow \text{wp } (\text{trm\_app } v1 v2) Q.$
 Proof using. unfold wp. intros. intros $h K$. *applies* $\times$ eval_app_fun. Qed.

Lemma wp_app_fix : $\forall f x v1 v2 t1 Q,$
 $v1 = \text{val\_fix } f x t1 \rightarrow$
 $\text{wp } (\text{subst } x v2 (\text{subst } f v1 t1)) Q \Rightarrow \text{wp } (\text{trm\_app } v1 v2) Q.$
 Proof using. unfold wp. intros. intros $h K$. *applies* $\times$ eval_app_fix. Qed.

Lemma wp_seq : $\forall t1 t2 Q,$
 $\text{wp } t1 (\text{fun } r \Rightarrow \text{wp } t2 Q) \Rightarrow \text{wp } (\text{trm\_seq } t1 t2) Q.$
 Proof using. unfold wp. intros. intros $h K$. *applies* $\times$ eval_seq. Qed.

Lemma wp_let : $\forall x t1 t2 Q,$
 $\text{wp } t1 (\text{fun } v \Rightarrow \text{wp } (\text{subst } x v t2) Q) \Rightarrow \text{wp } (\text{trm\_let } x t1 t2) Q.$
 Proof using. unfold wp. intros. intros $h K$. *applies* $\times$ eval_let. Qed.

Lemma wp_if : $\forall b t1 t2 Q,$
 $\text{wp } (\text{if } b \text{ then } t1 \text{ else } t2) Q \Rightarrow \text{wp } (\text{trm\_if } b t1 t2) Q.$
 Proof using. unfold wp. intros. intros $h K$. *applies* $\times$ eval_if. Qed.

Lemma wp_if_case : $\forall b t1 t2 Q,$
 $(\text{if } b \text{ then wp } t1 Q \text{ else wp } t2 Q) \Rightarrow \text{wp } (\text{trm\_if } b t1 t2) Q.$
 Proof using. intros. *applies* himpl_trans_r wp_if. *case_if* $\neg$. Qed.

35.6 WP Generator

This section defines a “weakest-precondition style characteristic formula generator”. This technology adapts the technique of “characteristic formulae” (originally developed in CFML 1.0) to produce weakest preconditions. (The formulae, their manipulation, and their correctness proofs are simpler in wp-style.)

The goal of the section is to define a function `wpgen t`, recursively over the structure of t , such that `wpgen t Q` entails `wp t Q`. Unlike `wp t Q`, which is defined semantically, `wpgen t Q` is defined following the syntax of t .

Technically, we define `wpgen E t`, where E is a list of bindings, to compute a formula that entails `wp (isubst E t)`, where `isubst E t` denotes the iterated substitution of bindings from E inside t .

In order to define a structurally recursive and relatively efficient characteristic formula generator, we need to introduce contexts, that essentially serve to apply substitutions lazily.

A context is an association list from variables to values.

lookup x E returns **Some** v if x is bound to a value v , and **None** otherwise.

$\text{rem } \mathbf{x} \ E$ denotes the removal of bindings on $\mathbf{x}$ from E .

`ctx_disjoint` *E1 E2* asserts that the two contexts have disjoint domains.

`ctx_equiv` *E1 E2* asserts that the two contexts bind same keys to same values.

Basic properties of context operations follow.

```

Lemma lookup_app :  $\forall E1 E2 x,$ 
  lookup  $x (E1 ++ E2) = \text{match lookup } x E1 \text{ with}$ 
    | None  $\Rightarrow$  lookup  $x E2$ 
    | Some  $v \Rightarrow$  Some  $v$ 
  end.

```

```

introv. induction E1 as [| (y,w) E1' ]; rew_list; simpl; intros.
{ auto. } { case_var ¬. }

```

Qed.

Lemma lookup_rem : $\forall x y E$,
lookup x (rem $y E$) = If $x = y$ then None else lookup $x E$.

Proof using.

```
intros. induction E as [(z,v) E'].
{ simpl. case_var $\neg$ . }
{ simpl. case_var $\sim$ ; simpl; case_var $\neg$ . }
```

Qed.

Lemma rem_app : $\forall x E1 E2$,
rem x ($E1 ++ E2$) = rem $x E1 ++$ rem $x E2$.

Proof using.

```
intros. induction E1 as [(y,w) E1']; rew_list; simpl. { auto. }
{ case_var $\neg$ . { rew_list. fequals. } }
```

Qed.

Lemma ctx_equiv_rem : $\forall x E1 E2$,
ctx_equiv $E1 E2 \rightarrow$
ctx_equiv (rem $x E1$) (rem $x E2$).

Proof using.

```
introv M. unfolds ctx_equiv. intros y.
do 2 rewrite lookup_rem. case_var $\neg$ .
```

Qed.

Lemma ctx_disjoint_rem : $\forall x E1 E2$,
ctx_disjoint $E1 E2 \rightarrow$
ctx_disjoint (rem $x E1$) (rem $x E2$).

Proof using.

```
introv D. intros y v1 v2 K1 K2. rewrite lookup_rem in *.
case_var $\neg$ . applies  $\times D K1 K2$ .
```

Qed.

Lemma ctx_disjoint_equiv_app : $\forall E1 E2$,
ctx_disjoint $E1 E2 \rightarrow$
ctx_equiv ($E1 ++ E2$) ($E2 ++ E1$).

Proof using.

```
introv D. intros x. do 2 rewrite $\neg$  lookup_app.
case_eq (lookup  $x E1$ ); case_eq (lookup  $x E2$ ); auto.
{ intros v2 K2 v1 K1. false  $\times D$ . }
```

Qed.

End CtxOps.

35.6.2 Multi-Substitution

Definition of Multi-Substitution

The specification of the characteristic formula generator is expressed using the multi-substitution function, which substitutes a list of bindings inside a term.

```
Fixpoint isubst (E:ctx) (t:trm) : trm :=
  match t with
  | trm_val v =>
    v
  | trm_var x =>
    match lookup x E with
    | None => t
    | Some v => v
    end
  | trm_fun x t1 =>
    trm_fun x (isubst (rem x E) t1)
  | trm_fix f x t1 =>
    trm_fix f x (isubst (rem x (rem f E)) t1)
  | trm_if t0 t1 t2 =>
    trm_if (isubst E t0) (isubst E t1) (isubst E t2)
  | trm_seq t1 t2 =>
    trm_seq (isubst E t1) (isubst E t2)
  | trm_let x t1 t2 =>
    trm_let x (isubst E t1) (isubst (rem x E) t2)
  | trm_app t1 t2 =>
    trm_app (isubst E t1) (isubst E t2)
  end.
```

Properties of Multi-Substitution

The goal of this entire section is only to establish `isubst_nil` and `isubst_rem`, which assert:

`isubst nil t = t` and `isubst ((x,v)::E) t = subst x v (isubst (rem x E) t)`

`isubst_nil` explains that the empty substitution is the identity.

Lemma `isubst_nil` : $\forall t,$
`isubst nil t = t.`

Proof using. `intros t. induction t; simpl; fequals. Qed.`

`subst_eq_isubst_one` relates `subst` and `isubst`.

Lemma `subst_eq_isubst_one` : $\forall x v t,$
`subst x v t = isubst ((x,v)::nil) t.`

Proof using.

`intros. induction t; simpl.`

```

{ fequals. }
{ case_var ¬. }
{ fequals. case_var ¬. { rewrite ¬ isubst_nil. } }
{ fequals. case_var; try case_var; simpl; try case_var; try rewrite isubst_nil; auto. }
{ fequals ×. }
{ fequals ×. }
{ fequals ×. case_var ¬. { rewrite ¬ isubst_nil. } }
{ fequals ×. }

```

Qed.

The next lemma shows that equivalent contexts produce equal results for `isubst`.

Lemma `isubst_ctx_equiv` : $\forall t \ E1 \ E2,$
`ctx_equiv E1 E2` $\rightarrow$
`isubst E1 t` = `isubst E2 t`.

Proof using.

```

hint ctx_equiv_rem.
intros t. induction t; introv EQ; simpl; fequals ¬.
{ rewrite ¬ EQ. }

```

Qed.

`isubst_app` asserts that `isubst` distribute over concatenation.

Lemma `isubst_app` : $\forall t \ E1 \ E2,$
`isubst (E1 ++ E2) t` = `isubst E2 (isubst E1 t)`.

Proof using.

```

hint ctx_disjoint_rem.
intros t. induction t; simpl; intros.
{ fequals. }
{ rename v into x. rewrite ¬ lookup_app.
  case_eq (lookup x E1); introv K1; case_eq (lookup x E2); introv K2; auto.
  { simpl. rewrite ¬ K2. }
  { simpl. rewrite ¬ K2. } }
{ fequals. rewrite × rem_app. }
{ fequals. do 2 rewrite × rem_app. }
{ fequals ×. }
{ fequals ×. }
{ fequals ×. { rewrite × rem_app. } }
{ fequals ×. }

```

Qed.

`isubst_cons` is a corollary

Lemma `isubst_cons` : $\forall x \ v \ xvs \ t,$
`isubst ((x, v) :: xvs) t` = `isubst xvs (subst x v t)`.

Proof using.

```

intros. rewrite ← app_cons_one_r. rewrite isubst_app.

```


rewrite $\times \leftarrow$ subst_eq_isubst_one.
Qed.

The next lemma asserts that the concatenation order is irrelevant in a substitution if the contexts have disjoint domains.

Lemma isubst_app_swap : $\forall t\ E1\ E2,$
 $\text{ctx\_disjoint } E1\ E2 \rightarrow$
 $\text{isubst } (E1 ++ E2)\ t = \text{isubst } (E2 ++ E1)\ t.$

Proof using.

intros D . applies isubst_ctx_equiv. applies $\neg$ ctx_disjoint_equiv_app.
Qed.

We are ready to derive the second targeted property of isubst.

Lemma isubst_rem : $\forall x\ v\ E\ t,$
 $\text{isubst } ((x, v) :: E)\ t = \text{subst } x\ v\ (\text{isubst } (\text{rem } x\ E)\ t).$

Proof using.

intros. rewrite subst_eq_isubst_one. rewrite $\leftarrow$ isubst_app.
 rewrite isubst_app_swap.
 { applies isubst_ctx_equiv. intros y . rew_list. simpl. rewrite lookup_rem. case_var $\neg$.
 }
 { intros $y\ v1\ v2\ K1\ K2$. simpls. rewrite lookup_rem in $K1$. case_var. }
 Qed.

A variant useful for trm_fix is proved next.

Lemma isubst_rem_2 : $\forall f\ x\ vf\ vx\ E\ t,$
 $\text{isubst } ((f, vf) :: (x, vx) :: E)\ t = \text{subst } x\ vx\ (\text{subst } f\ vf\ (\text{isubst } (\text{rem } x\ (\text{rem } f\ E))\ t)).$

Proof using.

intros. do 2 rewrite subst_eq_isubst_one. do 2 rewrite $\leftarrow$ isubst_app.
 rewrite isubst_app_swap.
 { applies isubst_ctx_equiv. intros y . rew_list. simpl. do 2 rewrite lookup_rem.
 case_var $\neg$. }
 { intros $y\ v1\ v2\ K1\ K2$. rew_list x in *. simpls. do 2 rewrite lookup_rem in $K1$.
 case_var. }
 Qed.

35.6.3 Definition of wpgen

The definition of wpgen $E\ t$ comes next. It depends on a predicate called mkstruct for handling structural rules, and on one auxiliary definition for each term rule.

Definition of mkstruct

Let formula denote the type of wp t and wpgen t .

Definition formula := (val $\rightarrow$ hprop) $\rightarrow$ hprop.

Implicit Type F : formula.

`mkstruct` F transforms a formula F into one that satisfies structural rules of Separation Logic. This predicate transformer enables integrating support for the frame rule (and other structural rules), in characteristic formulae.

Definition `mkstruct` (F :formula) : formula :=
`fun Q => \exists Q', F Q' \-* (Q' \-* Q).`

Lemma `mkstruct_ramified` : $\forall Q1\ Q2\ F,$
 $(\text{mkstruct } F\ Q1) \-* (Q1 \-* Q2) ==> (\text{mkstruct } F\ Q2).$

Proof using. `unfold mkstruct. xsimpl. Qed.`

Arguments `mkstruct_ramified` : `clear implicits.`

Lemma `mkstruct_erase` : $\forall Q\ F,$
 $F\ Q ==> \text{mkstruct } F\ Q.$

Proof using. `unfolds mkstruct. xsimpl. Qed.`

Arguments `mkstruct_erase` : `clear implicits.`

Lemma `mkstruct_conseq` : $\forall F\ Q1\ Q2,$
 $Q1 ==> Q2 \rightarrow$
 $\text{mkstruct } F\ Q1 ==> \text{mkstruct } F\ Q2.$

Proof using.

`introv WQ. unfolds mkstruct. xpull. intros Q. xsimpl Q. xchanges WQ.`
`Qed.`

Lemma `mkstruct_frame` : $\forall F\ H\ Q,$
 $(\text{mkstruct } F\ Q) \-* H ==> \text{mkstruct } F\ (Q \-* H).$

Proof using.

`intros. unfold mkstruct. xpull. intros Q'. xsimpl Q'.`
`Qed.`

Lemma `mkstruct_monotone` : $\forall F1\ F2\ Q,$
 $(\forall Q, F1\ Q ==> F2\ Q) \rightarrow$
 $\text{mkstruct } F1\ Q ==> \text{mkstruct } F2\ Q.$

Proof using.

`introv WF. unfolds mkstruct. xpull. intros Q'. xchange WF. xsimpl Q'.`
`Qed.`

Definition of Auxiliary Definition for `wpgen`

we state auxiliary definitions for `wpgen`, one per term construct. For simplicity, we here assume the term `t` to be in A-normal form. If it is not, the formula generated will be incomplete, that is, useless to prove triples about the term `t`. Note that the actual generator in CFML2 does support terms that are not in A-normal form.

Definition `wpgen_fail` : formula := `fun Q =>`
`\[False].`

Definition wpgen_val ($v:\mathbf{val}$) : formula := fun $Q \Rightarrow$
 $Q\ v$.

Definition wpgen_fun ($Fof:\mathbf{val} \rightarrow \text{formula}$) : formula := fun $Q \Rightarrow$
 $\backslash \forall\ vf, \backslash [\forall\ vx\ Q', Fof\ vx\ Q' \Rightarrow wp\ (\text{trm\_app}\ vf\ vx)\ Q'] \backslash -*\ Q\ vf$.

Definition wpgen_fix ($Fof:\mathbf{val} \rightarrow \mathbf{val} \rightarrow \text{formula}$) : formula := fun $Q \Rightarrow$
 $\backslash \forall\ vf, \backslash [\forall\ vx\ Q', Fof\ vf\ vx\ Q' \Rightarrow wp\ (\text{trm\_app}\ vf\ vx)\ Q'] \backslash -*\ Q\ vf$.

Definition wpgen_var ($E:\text{ctx}$) ($x:\text{var}$) : formula :=
 match lookup $x\ E$ with
 | None $\Rightarrow$ wpgen_fail
 | Some $v \Rightarrow$ wpgen_val v
 end.

Definition wpgen_seq ($F1\ F2:\text{formula}$) : formula := fun $Q \Rightarrow$
 $F1\ (\text{fun}\ v \Rightarrow F2\ Q)$.

Definition wpgen_let ($F1:\text{formula}$) ($F2of:\mathbf{val} \rightarrow \text{formula}$) : formula := fun $Q \Rightarrow$
 $F1\ (\text{fun}\ v \Rightarrow F2of\ v\ Q)$.

Definition wpgen_if ($t:\mathbf{trm}$) ($F1\ F2:\text{formula}$) : formula := fun $Q \Rightarrow$
 $\backslash \exists\ (b:\mathbf{bool}), \backslash [t = \text{trm\_val}\ (\text{val\_bool}\ b)] \backslash * (\text{if}\ b\ \text{then}\ F1\ Q\ \text{else}\ F2\ Q)$.

Definition wpgen_if_trm ($F0\ F1\ F2:\text{formula}$) : formula :=
 wpgen_let $F0\ (\text{fun}\ v \Rightarrow \text{mkstruct}\ (\text{wpgen\_if}\ v\ F1\ F2))$.

Definition wpgen_app ($t:\mathbf{trm}$) : formula := fun $Q \Rightarrow$
 $\backslash \exists\ H, H \backslash * \backslash [\text{triple}\ t\ H\ Q]$.

Recursive Definition of wpgen

$\text{wpgen}\ E\ t$ is structurally recursive on t . Note that this function does not recurse through values. Note also that the context E gets extended when traversing bindings, in the let-binding and the function cases.

Fixpoint wpgen ($E:\text{ctx}$) ($t:\mathbf{trm}$) : formula :=
 mkstruct match t with
 | $\text{trm\_val}\ v \Rightarrow$ wpgen_val v
 | $\text{trm\_var}\ x \Rightarrow$ wpgen_var $E\ x$
 | $\text{trm\_fun}\ x\ t1 \Rightarrow$ wpgen_fun ($\text{fun}\ v \Rightarrow \text{wpgen}\ ((x,v)::E)\ t1$)
 | $\text{trm\_fix}\ f\ x\ t1 \Rightarrow$ wpgen_fix ($\text{fun}\ vf\ v \Rightarrow \text{wpgen}\ ((f,vf)::(x,v)::E)\ t1$)
 | $\text{trm\_if}\ t0\ t1\ t2 \Rightarrow$ wpgen_if (isubst $E\ t0$) (wpgen $E\ t1$) (wpgen $E\ t2$)
 | $\text{trm\_seq}\ t1\ t2 \Rightarrow$ wpgen_seq (wpgen $E\ t1$) (wpgen $E\ t2$)
 | $\text{trm\_let}\ x\ t1\ t2 \Rightarrow$ wpgen_let (wpgen $E\ t1$) ($\text{fun}\ v \Rightarrow \text{wpgen}\ ((x,v)::E)\ t2$)
 | $\text{trm\_app}\ t1\ t2 \Rightarrow$ wpgen_app (isubst $E\ t$)
 end.

Soundness of `wpgen`

`formula_sound t F` asserts that, for any Q , the Separation Logic judgment `triple (F Q) t Q` is valid. In other words, it states that F is a stronger formula than `wp t`.

The soundness theorem that we are ultimately interested in asserts that `formula_sound (isubst E t) (wpgen E t)` holds for any E and t .

Definition `formula_sound (t:trm) (F:formula) : Prop :=`
 $\forall Q, F \ Q ==> \text{wp } t \ Q.$

Lemma `wp_sound :  $\forall t,$`
`formula_sound t (wp t).`

Proof using. `intros. intros Q. applies himpl_refl. Qed.`

One soundness lemma for `mkstruct`.

Lemma `mkstruct_wp :  $\forall t,$`
`mkstruct (wp t) = (wp t).`

Proof using.

`intros. applies fun_ext_1. intros Q. applies himpl_antisym.`
`{ unfold mkstruct. xpull. intros Q'. applies wp_ramified. }`
`{ applies mkstruct_erase. }`

Qed.

Lemma `mkstruct_sound :  $\forall t \ F,$`
`formula_sound t F  $\rightarrow$`
`formula_sound t (mkstruct F).`

Proof using.

`introv M. unfolds formula_sound. intros Q'.`
`rewrite  $\leftarrow$  mkstruct_wp. applies  $\times$  mkstruct_monotone M.`

Qed.

One soundness lemma for each term construct.

Lemma `wpgen_fail_sound :  $\forall t,$`
`formula_sound t wpgen_fail.`

Proof using. `intros. intros Q. unfold wpgen_fail. xpull. Qed.`

Lemma `wpgen_val_sound :  $\forall v,$`
`formula_sound (trm_val v) (wpgen_val v).`

Proof using. `intros. intros Q. unfolds wpgen_val. applies wp_val. Qed.`

Lemma `wpgen_fun_sound :  $\forall x \ t1 \ Fof,$`
`( $\forall vx, \text{formula\_sound (subst } x \ vx \ t1) (Fof \ vx)$ )  $\rightarrow$`
`formula_sound (trm_fun x t1) (wpgen_fun Fof).`

Proof using.

`introv M. intros Q. unfolds wpgen_fun. applies himpl_hforall_l (val_fun x t1).`
`xchange hwand_hpure_l.`
`{ intros. applies himpl_trans_r. { applies  $\times$  wp_app_fun. } { applies  $\times$  M. } }`

$\{ \text{applies wp\_fun.} \}$
 Qed.

Lemma wpgen_fix_sound : $\forall f\ x\ t1\ Fof,$
 $(\forall vf\ vx, \text{formula\_sound} (\text{subst } x\ vx (\text{subst } f\ vf\ t1)) (Fof\ vf\ vx)) \rightarrow$
 $\text{formula\_sound} (\text{trm\_fix } f\ x\ t1) (\text{wpgen\_fix } Fof).$

Proof using.
introv M. intros Q. unfolds wpgen_fix.
applies himpl_hforall_l (val_fix f x t1). xchange hwand_hpure_l.
 $\{ \text{intros. applies himpl\_trans\_r. } \{ \text{applies} \times \text{wp\_app\_fix.} \} \{ \text{applies} \times M. \} \}$
 $\{ \text{applies wp\_fix.} \}$

Qed.

Lemma wpgen_seq_sound : $\forall F1\ F2\ t1\ t2,$
 $\text{formula\_sound } t1\ F1 \rightarrow$
 $\text{formula\_sound } t2\ F2 \rightarrow$
 $\text{formula\_sound} (\text{trm\_seq } t1\ t2) (\text{wpgen\_seq } F1\ F2).$

Proof using.
introv S1 S2. intros Q. unfolds wpgen_seq. applies himpl_trans_r wp_seq.
applies himpl_trans S1. applies wp_conseq. intros v. applies S2.

Qed.

Lemma wpgen_let_sound : $\forall F1\ F2of\ x\ t1\ t2,$
 $\text{formula\_sound } t1\ F1 \rightarrow$
 $(\forall v, \text{formula\_sound} (\text{subst } x\ v\ t2) (F2of\ v)) \rightarrow$
 $\text{formula\_sound} (\text{trm\_let } x\ t1\ t2) (\text{wpgen\_let } F1\ F2of).$

Proof using.
introv S1 S2. intros Q. unfolds wpgen_let. applies himpl_trans_r wp_let.
applies himpl_trans S1. applies wp_conseq. intros v. applies S2.

Qed.

Lemma wpgen_if_sound : $\forall F1\ F2\ t0\ t1\ t2,$
 $\text{formula\_sound } t1\ F1 \rightarrow$
 $\text{formula\_sound } t2\ F2 \rightarrow$
 $\text{formula\_sound} (\text{trm\_if } t0\ t1\ t2) (\text{wpgen\_if } t0\ F1\ F2).$

Proof using.
introv S1 S2. intros Q. unfold wpgen_if. xpull. intros b $\rightarrow$.
applies himpl_trans_r wp_if. case_if. { applies S1. } { applies S2. }

Qed.

Lemma wpgen_app_sound : $\forall t,$
 $\text{formula\_sound } t (\text{wpgen\_app } t).$

Proof using.
intros t Q. unfold wpgen_app. xpull. intros H M. rewrite wp_equiv. apply M.

Qed.

The main inductive proof for the soundness theorem.

```

Lemma wpgen_sound :  $\forall E t$ ,
  formula_sound (isubst  $E t$ ) (wpgen  $E t$ ).
Proof using.
  intros. gen  $E$ . induction  $t$ ; intros; simpl;
  try applys mkstruct_sound.
  { applys wpgen_val_sound. }
  { rename  $v$  into  $x$ . unfold wpgen_var. case_eq (lookup  $x E$ ).
    { intros  $v EQ$ . applys wpgen_val_sound. }
    { intros  $N$ . applys wpgen_fail_sound. } }
  { applys wpgen_fun_sound.
    { intros  $vx$ . rewrite  $\times \leftarrow$  isubst_rem. } }
  { applys  $\times$  wpgen_fix_sound.
    { fold isubst. intros  $vf vx$ . rewrite  $\times \leftarrow$  isubst_rem_2. } }
  { applys wpgen_app_sound. }
  { applys  $\times$  wpgen_seq_sound. }
  { rename  $v$  into  $x$ . applys  $\times$  wpgen_let_sound.
    { intros  $v$ . rewrite  $\times \leftarrow$  isubst_rem. } }
  { applys  $\times$  wpgen_if_sound. }
Qed.

```

```

Lemma himpl_wpgen_wp :  $\forall t Q$ ,
  wpgen nil  $t Q \implies wp t Q$ .
Proof using.
  introv  $M$ . lets  $N$ : (wpgen_sound nil  $t$ ). rewrite isubst_nil in  $N$ . applys  $\times N$ .
Qed.

```

The final theorem for closed terms.

```

Lemma triple_of_wpgen :  $\forall t H Q$ ,
   $H \implies wpgen nil t Q \rightarrow$ 
  triple  $t H Q$ .
Proof using.
  introv  $M$ . rewrite  $\leftarrow$  wp_equiv. xchange  $M$ . applys himpl_wpgen_wp.
Qed.

```

35.7 Practical Proofs

This last section shows the techniques involved in setting up the lemmas and tactics required to carry out verification in practice, through concise proof scripts.

35.7.1 Lemmas for Tactics to Manipulate **wpgen** Formulae

```

Lemma xstruct_lemma :  $\forall F H Q$ ,
   $H \implies F Q \rightarrow$ 

```

```

    H ==> mkstruct F Q.
Proof using. intros M. xchange M. applies mkstruct_erase. Qed.

Lemma xval_lemma :  $\forall v H Q,$ 
  H ==> Q v  $\rightarrow$ 
  H ==> wpgen_val v Q.
Proof using. intros M. applies M. Qed.

Lemma xlet_lemma :  $\forall H F1 F2of Q,$ 
  H ==> F1 (fun v  $\Rightarrow$  F2of v Q)  $\rightarrow$ 
  H ==> wpgen_let F1 F2of Q.
Proof using. intros M. xchange M. Qed.

Lemma xseq_lemma :  $\forall H F1 F2 Q,$ 
  H ==> F1 (fun v  $\Rightarrow$  F2 Q)  $\rightarrow$ 
  H ==> wpgen_seq F1 F2 Q.
Proof using. intros M. xchange M. Qed.

Lemma xif_lemma :  $\forall b H F1 F2 Q,$ 
  (b = true  $\rightarrow$  H ==> F1 Q)  $\rightarrow$ 
  (b = false  $\rightarrow$  H ==> F2 Q)  $\rightarrow$ 
  H ==> wpgen_if b F1 F2 Q.
Proof using. intros M1 M2. unfold wpgen_if. xsimpl  $\times$  b. case_if  $\times$ . Qed.

Lemma xapp_lemma :  $\forall t Q1 H1 H Q,$ 
  triple t H1 Q1  $\rightarrow$ 
  H ==> H1  $\backslash$ * (Q1  $\backslash$ * protect Q)  $\rightarrow$ 
  H ==> wpgen_app t Q.
Proof using.
  intros M W. unfold wpgen_app. xsimpl.
  applies triple_ramified_frame M. applies W.
Qed.

Lemma xfun_spec_lemma :  $\forall (S:\mathbf{val} \rightarrow \mathbf{Prop}) H Q Fof,$ 
  ( $\forall vf,$ 
    ( $\forall vx H' Q', (H' ==> Fof vx Q') \rightarrow$  triple (trm_app vf vx) H' Q')  $\rightarrow$ 
    S vf)  $\rightarrow$ 
  ( $\forall vf, S vf \rightarrow (H ==> Q vf)) \rightarrow$ 
  H ==> wpgen_fun Fof Q.
Proof using.
  intros M1 M2. unfold wpgen_fun. xsimpl. intros vf N.
  applies M2. applies M1. intros K. rewrite  $\leftarrow$  wp_equiv. xchange K. applies N.
Qed.

Lemma xfun_nospec_lemma :  $\forall H Q Fof,$ 
  ( $\forall vf,$ 
    ( $\forall vx H' Q', (H' ==> Fof vx Q') \rightarrow$  triple (trm_app vf vx) H' Q')  $\rightarrow$ 
    (H ==> Q vf))  $\rightarrow$ 

```

$H \Rightarrow \text{wp\_gen\_fun } Fof \ Q.$
 Proof using.
 $\text{intros } M. \text{ unfold wp\_gen\_fun. xsimpl. intros } vf \ N. \text{ applies } M.$
 $\text{intros } K. \text{ rewrite } \leftarrow \text{wp\_equiv. xchange } K. \text{ applies } N.$
 Qed.

Lemma $\text{xfix\_spec\_lemma} : \forall (S : \text{val} \rightarrow \text{Prop}) \ H \ Q \ Fof,$
 $(\forall vf,$
 $(\forall vx \ H' \ Q', (H' \Rightarrow Fof \ vf \ vx \ Q') \rightarrow \text{triple } (\text{trm\_app } vf \ vx) \ H' \ Q') \rightarrow$
 $S \ vf) \rightarrow$
 $(\forall vf, S \ vf \rightarrow (H \Rightarrow Q \ vf)) \rightarrow$
 $H \Rightarrow \text{wp\_gen\_fix } Fof \ Q.$
 Proof using.
 $\text{intros } M1 \ M2. \text{ unfold wp\_gen\_fix. xsimpl. intros } vf \ N.$
 $\text{applies } M2. \text{ applies } M1. \text{ intros } K. \text{ rewrite } \leftarrow \text{wp\_equiv. xchange } K. \text{ applies } N.$
 Qed.

Lemma $\text{xfix\_nospec\_lemma} : \forall H \ Q \ Fof,$
 $(\forall vf,$
 $(\forall vx \ H' \ Q', (H' \Rightarrow Fof \ vf \ vx \ Q') \rightarrow \text{triple } (\text{trm\_app } vf \ vx) \ H' \ Q') \rightarrow$
 $(H \Rightarrow Q \ vf)) \rightarrow$
 $H \Rightarrow \text{wp\_gen\_fix } Fof \ Q.$
 Proof using.
 $\text{intros } M. \text{ unfold wp\_gen\_fix. xsimpl. intros } vf \ N. \text{ applies } M.$
 $\text{intros } K. \text{ rewrite } \leftarrow \text{wp\_equiv. xchange } K. \text{ applies } N.$
 Qed.

Lemma $\text{xwp\_lemma\_fun} : \forall v1 \ v2 \ x \ t \ H \ Q,$
 $v1 = \text{val\_fun } x \ t \rightarrow$
 $H \Rightarrow \text{wp\_gen } ((x, v2) :: \text{nil}) \ t \ Q \rightarrow$
 $\text{triple } (\text{trm\_app } v1 \ v2) \ H \ Q.$
 Proof using.
 $\text{intros } M1 \ M2. \text{ rewrite } \leftarrow \text{wp\_equiv. xchange } M2.$
 $\text{xchange } (\gg \text{wp\_gen\_sound } ((x, v2) :: \text{nil}) \ t \ Q).$
 $\text{rewrite } \leftarrow \text{subst\_eq\_isubst\_one. applies } \times \text{wp\_app\_fun.}$
 Qed.

Lemma $\text{xwp\_lemma\_fix} : \forall v1 \ v2 \ f \ x \ t \ H \ Q,$
 $v1 = \text{val\_fix } f \ x \ t \rightarrow$
 $f \neq x \rightarrow$
 $H \Rightarrow \text{wp\_gen } ((f, v1) :: (x, v2) :: \text{nil}) \ t \ Q \rightarrow$
 $\text{triple } (\text{trm\_app } v1 \ v2) \ H \ Q.$
 Proof using.
 $\text{intros } M1 \ N \ M2. \text{ rewrite } \leftarrow \text{wp\_equiv. xchange } M2.$
 $\text{xchange } (\gg \text{wp\_gen\_sound } ((f, v1) :: \text{nil}) ++ (x, v2) :: \text{nil}) \ t \ Q).$


```

    rewrite isubst_app. do 2 rewrite ← subst_eq_isubst_one.
    applies × wp_app_fix.
Qed.

Lemma xtriple_lemma : ∀ t H (Q:val→hprop),
  H ==> mkstruct (wpgen_app t) Q →
  triple t H Q.
Proof using.
  intros M. rewrite ← wp_equiv. xchange M. unfold mkstruct, wpgen_app.
  xpull. intros Q' H' N. rewrite ← wp_equiv in N. xchange N.
  applies wp_ramified.
Qed.

```

35.7.2 Tactics to Manipulate **wpgen** Formulae

The tactic are presented in chapter WPgen.

Database of hints for *xapp*.

```

#[global] Hint Resolve triple_get triple_set triple_ref triple_free : triple.
#[global] Hint Resolve triple_add triple_div triple_neg triple_opp triple_eq
  triple_neq triple_sub triple_mul triple_mod triple_le triple_lt
  triple_ge triple_gt triple_ptr_add triple_ptr_add_nat : triple.
xstruct removes the leading mkstruct.

```

Tactic Notation "xstruct" :=

applies xstruct_lemma.

xstruct_if_needed removes the leading mkstruct if there is one.

Tactic Notation "xstruct_if_needed" :=

try match goal with ⊢ ?H ==> mkstruct ?F ?Q ⇒ *xstruct* end.

Tactic Notation "xval" :=

xstruct_if_needed; *applies* xval_lemma.

Tactic Notation "xlet" :=

xstruct_if_needed; *applies* xlet_lemma.

Tactic Notation "xseq" :=

xstruct_if_needed; *applies* xseq_lemma.

Tactic Notation "xseq_xlet_if_needed" :=

```

try match goal with ⊢ ?H ==> mkstruct ?F ?Q ⇒
match F with
| wpgen_seq ?F1 ?F2 ⇒ xseq
| wpgen_let ?F1 ?F2of ⇒ xlet
end end.

```

Tactic Notation "xif" :=

```

xseq_xlet_if_needed; xstruct_if_needed;
applies xif_lemma; rew_bool_eq.

```

xapp_try_clear_unit_result implements some post-processing for cleaning up unused variables.

```

Tactic Notation "xapp_try_clear_unit_result" :=
  try match goal with ⊢ val → _ ⇒ intros _ end.

```

```

Tactic Notation "xtriple" :=
  intros; applies xtriple_lemma.

```

```

Tactic Notation "xtriple_if_needed" :=
  try match goal with ⊢ triple ?t ?H ?Q ⇒ applies xtriple_lemma end.

```

xapp_simpl performs the final step of the tactic *xapp*. It leverages *xsimpl_no_cancel_wand* *tt* which is a restricted version of *xsimpl* that does not attempt to cancel out magic wands.

```

Lemma xapp_simpl_lemma : ∀ F H Q,
  H ==> F Q →
  H ==> F Q \* (Q \-* protect Q).

```

Proof using. *introv M. xchange M. unfold protect. xsimpl. Qed.*

```

Tactic Notation "xapp_simpl" :=
  first [ eapply xapp_simpl_lemma
          | xsimpl_no_cancel_wand tt; unfold protect; xapp_try_clear_unit_result ].

```

```

Tactic Notation "xapp_pre" :=
  xtriple_if_needed; xseq_xlet_if_needed; xstruct_if_needed.

```

xapp_nosubst E implements the heart of *xapp E*. If the argument *E* was always a triple, it would suffice to run *applies xapp_lemma E; xapp_simpl*. Yet, *E* might be an specification involving quantifiers. These quantifiers need to be first instantiated. This instantiation is achieved by means of the tactic *forwards_nounfold_then* offered by the TLC library.

```

Tactic Notation "xapp_nosubst" constr(E) :=
  xapp_pre;
  forwards_nounfold_then E ltac:(fun K ⇒ applies xapp_lemma K; xapp_simpl).

```

xapp_apply_spec implements the heart of *xapp*, when called without argument. It finds out the specification triple, either in the hint data base named *triple*, or in the context by looking for an induction hypothesis. Disclaimer: as explained in chapter *WPgen*, the simple implementation of *xapp_apply_spec* which we use here does not apply when the specification includes premises that cannot be solved by *eauto*; in such cases, the tactic *xapp E* must be called, providing the specification *E* explicitly. This limitation is overcome using more involved *Hint Extern* tricks in CFML 2.0.

```

Tactic Notation "xapp_apply_spec" :=
  first [ solve [ eauto with triple ]
          | match goal with H: _ ⊢ _ ⇒ eapply H end ].

```

xapp_nosubst_for_records is place holder for implementing *xapp* on records. It is implemented further on.

```
Ltac xapp_nosubst_for_records tt :=
  fail.
```

xapp first calls *xtriple* if the goal is *triple t H Q* instead of $H \implies \text{wp } t \ Q$.

```
Tactic Notation "xapp_nosubst" :=
  xapp_pre;
  first [ applys xapp_lemma; [ xapp_apply_spec | xapp_simpl ]
        | xapp_nosubst_for_records tt ].
```

xapp_try_subst checks if the goal is of the form:

- either $\forall (r:\text{val}), (r = \dots) \rightarrow \dots$
- or $\forall (r:\text{val}), \forall x, (r = \dots) \rightarrow \dots$

in which case it substitutes *r* away.

```
Tactic Notation "xapp_try_subst" :=
  try match goal with
  |  $\vdash \forall (r:\text{val}), (r = \_) \rightarrow \_ \Rightarrow \text{intros } ? \rightarrow$ 
  |  $\vdash \forall (r:\text{val}), \forall x, (r = \_) \rightarrow \_ \Rightarrow$ 
    let y := fresh x in intros ? y ->; revert y
  end.
```

```
Tactic Notation "xapp" constr(E) :=
  xapp_nosubst E; xapp_try_subst.
```

```
Tactic Notation "xapp" :=
  xapp_nosubst; xapp_try_subst.
```

```
Tactic Notation "xapp_view" :=
  xseq_xlet_if_needed; xstruct_if_needed; applys xapp_lemma.
```

xapp is essentially equivalent to *xapp_view*; [*xapp_apply_spec* | *xapp_simpl*] .
xfun handles local functions, only for functions of one argument.

```
Tactic Notation "xfun" constr(S) :=
  xseq_xlet_if_needed; xstruct_if_needed;
  first [ applys xfun_spec_lemma S
        | applys xfix_spec_lemma S ].
```

```
Tactic Notation "xfun" :=
  xseq_xlet_if_needed; xstruct_if_needed;
  first [ applys xfun_nospec_lemma
        | applys xfix_nospec_lemma ].
```

xvars may be called for unfolding “program variables as definitions”, which take the form *Vars.x*, and revealing the underlying string.

Tactic Notation "xvars" :=

DefinitionsForVariables.libsepar_unfold.

xwp_simpl is a specialized version of *simpl* to be used for getting the function *wp* to compute properly.

From *Stdlib* Require Bool.

Ltac *xwp_simpl* :=

```
xvars;
cbn beta delta [
  LibListExec.combine List.combine
  wpgen wpgen_var isubst lookup var_eq
  string_dec string_rec string_rect
  sumbool_rec sumbool_rect
  Ascii.ascii_dec Ascii.ascii_rec Ascii.ascii_rect
  Bool.bool_dec bool_rec bool_rect ] iota zeta;
simpl.
```

xwp evaluates *wpgen* for the term appearing in the goal. This implementation of *xwp* is for functions of arity one only, refined further on for arbitrary arities.

Tactic Notation "xwp_arity_one_only" :=

```
intros;
first [ applies xwp_lemma_fun; [ reflexivity | ]
      | applies xwp_lemma_fix; [ reflexivity | ] ];
xwp_simpl.
```

35.7.3 Additional Tooling for *xapp*

xapp × is a convenient notation for *xapp*; *auto_star*. *xapp* × *E* is a convenient notation for *xapp E*; *auto_star*.

Tactic Notation "xapp" "*" :=

xapp; *auto_star*.

Tactic Notation "xapp" "*" constr(*E*) :=

xapp E; *auto_star*.

xapp_debug is a tactic to help debugging a call to *xapp*.

Tactic Notation "xapp_debug" :=

```
let step msg := (idtac msg; match goal with ⊢ ?H ⇒ idtac H end) in
xapp_pre;
step "===== goal after [xapp_pre] =====";
first
[ applies xapp_lemma;
  [ step "===== triple for [xapp_apply_spec] =====";
    first [ xapp_apply_spec; fail 1
```

```

| idtac "=====
  idtac "====> [xapp_apply_spec] failed!";
  idtac "====> [eauto with triple] could not solve find a specification";
  idtac "====> try [xapp_view. eapply myspec.] or [xapp myspec]. ";
  idtac "====> if side-conditions are not solved by [eauto], [xapp] won't
work."|
| idtac |
| applies xapp_lemma;
| [ xapp_apply_spec
| step "===== entailment for [xapp_simpl] =====";
  xapp_simpl | ].

```

35.7.4 Notations for Triples and **wpgen**

Declare Scope *wp_scope*.

Notation for printing proof obligations arising from **wpgen**.

Notation "'PRE' H 'CODE' F 'POST' Q" :=
 (*H* ==> (mkstruct *F*) *Q*)
 (at level 8, *H* at level 0, *F*, *Q* at level 0,
format "'[v' 'PRE' H '/' 'CODE' F '/' 'POST' Q ']'") : *wp_scope*.

Notation "' F" :=
 (mkstruct *F*)
 (at level 10,
format "' F") : *wp_scope*.

Custom grammar for the display of characteristic formulae.

Declare Custom Entry *wp*.

Notation "<[e]>" :=
e
 (at level 0, *e* custom *wp* at level 99) : *wp_scope*.

Notation "' F" :=
 (mkstruct *F*)
 (in custom *wp* at level 10,
format "' F") : *wp_scope*.

Notation "(x)" :=
x
 (in custom *wp*,
x at level 99) : *wp_scope*.

Notation "{ x }" :=
x
 (in custom *wp* at level 0,

```

    x constr,
    only parsing) : wp_scope.

Notation "x" :=
  x
  (in custom wp at level 0,
   x constr at level 0) : wp_scope.

Notation "'Fail'" :=
  ((wpgen_fail))
  (in custom wp at level 69) : wp_scope.

Notation "'Val' v" :=
  ((wpgen_val v))
  (in custom wp at level 69) : wp_scope.

Notation "'Let' x ':= ' F1 'in' F2" :=
  ((wpgen_let F1 (fun x => F2)))
  (in custom wp at level 69,
   x name,
   F1 custom wp at level 99,
   F2 custom wp at level 99,
   right associativity,
   format "[v' '[' 'Let' x ':= ' F1 'in' ']' ' / ' '[' F2 ']' ']'") : wp_scope.

Notation "'Seq' F1 ; F2" :=
  ((wpgen_seq F1 F2))
  (in custom wp at level 68,
   F1 custom wp at level 99,
   F2 custom wp at level 99,
   right associativity,
   format "[v' 'Seq' '[' F1 ']' ; '/' '[' F2 ']' ']'") : wp_scope.

Notation "'App' t0 t1 .. tn" :=
  ((wpgen_app (trm_app .. (trm_app t0 t1) .. tn)))
  (in custom wp at level 68,
   t0 constr at level 0,
   t1 constr at level 0,
   tn constr at level 0)
  : wp_scope.

Notation "'If_ ' v 'Then' F1 'Else' F2" :=
  ((wpgen_if v F1 F2))
  (in custom wp at level 69,
   F1 custom wp at level 99,
   F2 custom wp at level 99,
   left associativity,
   format "[v' '[' 'If_ ' v 'Then' ']' ' / ' '[' F1 ']' ' / ' 'Else' ' / ' '[' F2 ']' ']'") : wp_scope.

```

```

Notation "'Fun' x '=>' F1" :=
  ((wpgen_fun (fun x => F1)))
  (in custom wp at level 69,
   x name,
   F1 custom wp at level 99,
   right associativity,
   format "[v' '[' 'Fun' x '=>' F1 ']' ']'") : wp_scope.

Notation "'Fix' vf x '=>' F1" :=
  ((wpgen_fix (fun vf x => F1)))
  (in custom wp at level 69,
   vf name, x name,
   F1 custom wp at level 99,
   right associativity,
   format "[v' '[' 'Fix' vf x '=>' F1 ']' ']'") : wp_scope.

```

35.7.5 Notation for Concrete Terms

The custom grammar for **trm** is defined in `LibSepVar`.

Declare Scope *val_scope*.

Terms

```

Notation "<{ e }>" :=
  e
  (at level 0, e custom trm at level 99) : trm_scope.

Notation "( x )" :=
  x
  (in custom trm, x at level 99) : trm_scope.

Notation "'begin' e 'end'" :=
  e
  (in custom trm, e custom trm at level 99, only parsing) : trm_scope.

Notation "{ x }" :=
  x
  (in custom trm, x constr) : trm_scope.

Notation "x" := x
  (in custom trm at level 0,
   x constr at level 0) : trm_scope.

Notation "t1 t2" := (trm_app t1 t2)
  (in custom trm at level 30,
   left associativity,
   only parsing) : trm_scope.

Notation "'if' t0 'then' t1 'else' t2" :=

```

```

(trm_if t0 t1 t2)
(in custom trm at level 69,
 t0 custom trm at level 99,
 t1 custom trm at level 99,
 t2 custom trm at level 99,
 left associativity,
 format "[v' '[' 'if' t0 'then' ']' '/' '[' t1 ']' '/' 'else' '/' '[' t2 ']' ']'") : trm_scope.

Notation "if t0 'then' t1 'end'" :=
(trm_if t0 t1 (trm_val val_unit))
(in custom trm at level 69,
 t0 custom trm at level 99,
 t1 custom trm at level 99,
 left associativity,
 format "[v' '[' 'if' t0 'then' ']' '/' '[' t1 ']' '/' 'end' ']'") : trm_scope.

Notation "t1 ';' t2" :=
(trm_seq t1 t2)
(in custom trm at level 68,
 t2 custom trm at level 99,
 right associativity,
 format "[v' '[' t1 ']' ';' '/' '[' t2 ']' ']'") : trm_scope.

Notation "let x '=' t1 'in' t2" :=
(trm_let x t1 t2)
(in custom trm at level 69,
 x at level 0,
 t1 custom trm at level 99,
 t2 custom trm at level 99,
 right associativity,
 format "[v' '[' 'let' x '=' t1 'in' ']' '/' '[' t2 ']' ']'") : trm_scope.

Notation "let f x1 '=' t1 'in' t2" :=
(trm_let f (trm_fun x1 t1) t2)
(in custom trm at level 69,
 t1 custom trm,
 t2 custom trm,
 f, x1 at level 0,
 format "let f x1 '=' t1 'in' t2") : val_scope.

Notation "let f x1 .. xn '=' t1 'in' t2" :=
(trm_let f (trm_fun x1 .. (trm_fun xn t1) ..) t2)
(in custom trm at level 69,
 t1 custom trm,
 t2 custom trm,
 f, x1, xn at level 0,

```



```

    format "'let' f x1 .. xn '=' t1 'in' t2") : val_scope.
Notation "'let' 'rec' f x1 '=' t1 'in' t2" :=
  (trm_let f (trm_fix f x1 t1) t2)
  (in custom trm at level 69,
   t1 custom trm,
   t2 custom trm,
   f, x1 at level 0,
   format "'let' 'rec' f x1 '=' t1 'in' t2") : val_scope.
Notation "'let' 'rec' f x1 x2 .. xn '=' t1 'in' t2" :=
  (trm_let f (trm_fix f x1 (trm_fun x2 .. (trm_fun xn t1) ..)) t2)
  (in custom trm at level 69,
   t1 custom trm,
   t2 custom trm,
   f, x1, x2, xn at level 0,
   format "'let' 'rec' f x1 x2 .. xn '=' t1 'in' t2") : val_scope.

Notation "'fun' x1 '=>' t" :=
  (val_fun x1 t)
  (in custom trm at level 69,
   x1 at level 0,
   t custom trm at level 99,
   format "'fun' x1 '=>' t") : val_scope.
Notation "'fun' x1 x2 .. xn '=>' t" :=
  (val_fun x1 (trm_fun x2 .. (trm_fun xn t) ..))
  (in custom trm at level 69,
   t custom trm,
   x1, x2, xn at level 0,
   format "'fun' x1 x2 .. xn '=>' t") : val_scope.

Notation "'fix' f x1 '=>' t" :=
  (val_fix f x1 t)
  (in custom trm at level 69,
   f, x1 at level 0,
   t custom trm at level 99,
   format "'fix' f x1 '=>' t") : val_scope.
Notation "'fix' f x1 x2 .. xn '=>' t" :=
  (val_fix f x1 (trm_fun x2 .. (trm_fun xn t) ..))
  (in custom trm at level 69,
   t custom trm,
   f, x1, x2, xn at level 0,
   format "'fix' f x1 x2 .. xn '=>' t") : val_scope.

Notation "'fun_' x1 '=>' t" :=

```

```

(trm_fun x1 t)
(in custom trm at level 69,
 x1 at level 0,
 t custom trm at level 99,
 format "'fun_' x1 '=>' t") : trm_scope.

Notation "'fun_' x1 .. xn '=>' t" :=
(trm_fun x1 .. (trm_fun xn t) ..)
(in custom trm at level 69,
 t custom trm,
 x1, xn at level 0,
 format "'fun_' x1 .. xn '=>' t") : trm_scope.

Notation "'fix_' vf x1 '=>' t" :=
(trm_fix vf x1 t)
(in custom trm at level 69,
 vf, x1 at level 0,
 t custom trm at level 99,
 format "'fix_' vf x1 '=>' t") : trm_scope.

Notation "'fix_' vf x1 x2 .. xn '=>' t" :=
(trm_fix vf x1 (trm_fun x2 .. (trm_fun xn t) ..))
(in custom trm at level 69,
 t custom trm,
 vf, x1, x2, xn at level 0,
 format "'fix_' vf x1 x2 .. xn '=>' t") : trm_scope.

Notation "()" :=
(trm_val val_unit)
(in custom trm at level 0) : trm_scope.

Notation "()" :=
(val_unit)
(at level 0) : val_scope.

Notation for Primitive Operations.

Notation "'ref'" :=
(trm_val (val_prim val_ref))
(in custom trm at level 0, only parsing) : trm_scope.

Notation "'free'" :=
(trm_val (val_prim val_free))
(in custom trm at level 0, only parsing) : trm_scope.

Notation "'not'" :=
(trm_val (val_prim val_neg))
(in custom trm at level 0, only parsing) : trm_scope.

Notation "!" t :=

```

```

    (val_get t)
    (in custom trm at level 67,
     t custom trm at level 99) : trm_scope.
Notation "t1 := t2" :=
    (val_set t1 t2)
    (in custom trm at level 67) : trm_scope.
Notation "t1 + t2" :=
    (val_add t1 t2)
    (in custom trm at level 58) : trm_scope.
Notation "'- t" :=
    (val_opp t)
    (in custom trm at level 57,
     t custom trm at level 99) : trm_scope.
Notation "t1 - t2" :=
    (val_sub t1 t2)
    (in custom trm at level 58) : trm_scope.
Notation "t1 * t2" :=
    (val_mul t1 t2)
    (in custom trm at level 57) : trm_scope.
Notation "t1 / t2" :=
    (val_div t1 t2)
    (in custom trm at level 57) : trm_scope.
Notation "t1 'mod' t2" :=
    (val_mod t1 t2)
    (in custom trm at level 57) : trm_scope.
Notation "t1 = t2" :=
    (val_eq t1 t2)
    (in custom trm at level 58) : trm_scope.
Notation "t1 <> t2" :=
    (val_neq t1 t2)
    (in custom trm at level 58) : trm_scope.
Notation "t1 <= t2" :=
    (val_le t1 t2)
    (in custom trm at level 60) : trm_scope.
Notation "t1 < t2" :=
    (val_lt t1 t2)
    (in custom trm at level 60) : trm_scope.
Notation "t1 >= t2" :=
    (val_ge t1 t2)

```

```

    (in custom trm at level 60) : trm_scope.
Notation "t1 > t2" :=
  (val_gt t1 t2)
  (in custom trm at level 60) : trm_scope.

```

35.7.6 Scopes, Coercions and Notations for Concrete Programs

```

Module PROGRAMSYNTAX.
Export NOTATIONFORVARIABLES.
Module VARS := DEFINITIONSFORVARIABLES.
Close Scope fmap_scope.
Open Scope string_scope.
Open Scope val_scope.
Open Scope trm_scope.
Open Scope wp_scope.
Coercion string_to_var (x:string) : var := x.
End PROGRAMSYNTAX.

```

35.8 Additional Separation Logic Operators

35.8.1 Disjunction: Definition and Properties of *hor*

```

Definition hor (H1 H2 : hprop) : hprop :=
  \∃ (b:bool), if b then H1 else H2.
Lemma hor_sym : ∀ H1 H2,
  hor H1 H2 = hor H2 H1.
Proof using.
  intros. unfold hor. applies himpl_antisym.
  { applies himpl_hexists_l. intros b.
    applies himpl_hexists_r (neg b). destruct× b. }
  { applies himpl_hexists_l. intros b.
    applies himpl_hexists_r (neg b). destruct× b. }
Qed.
Lemma himpl_hor_r_l : ∀ H1 H2,
  H1 ==> hor H1 H2.
Proof using. intros. unfolds hor. ∃× true. Qed.
Lemma himpl_hor_r_r : ∀ H1 H2,
  H2 ==> hor H1 H2.
Proof using. intros. unfolds hor. ∃× false. Qed.

```

Lemma `himpl_hor_l` : $\forall H1\ H2\ H3,$
 $H1 \implies H3 \rightarrow$
 $H2 \implies H3 \rightarrow$
 $\text{hor } H1\ H2 \implies H3.$

Proof using.

`intros M1 M2. unfolds hor. applies himpl_hexists_l. intros b. case_if ×.`

Qed.

Lemma `triple_hor` : $\forall t\ H1\ H2\ Q,$
 $\text{triple } t\ H1\ Q \rightarrow$
 $\text{triple } t\ H2\ Q \rightarrow$
 $\text{triple } t\ (\text{hor } H1\ H2)\ Q.$

Proof using.

`intros M1 M2. unfold hor. applies triple_hexists.`
`intros b. destruct × b.`

Qed.

35.8.2 Conjunction: Definition and Properties of `hand`

Definition `hand` ($H1\ H2 : \text{hprop}$) : `hprop` :=
 $\backslash \forall (b:\text{bool}), \text{ if } b \text{ then } H1 \text{ else } H2.$

Lemma `hand_sym` : $\forall H1\ H2,$
 $\text{hand } H1\ H2 = \text{hand } H2\ H1.$

Proof using.

`intros. unfold hand. applies himpl_antisym.`
`{ applies himpl_hforall_r. intros b.`
`applies himpl_hforall_l (neg b). destruct × b. }`
`{ applies himpl_hforall_r. intros b.`
`applies himpl_hforall_l (neg b). destruct × b. }`

Qed.

Lemma `himpl_hand_l_r` : $\forall H1\ H2,$
 $\text{hand } H1\ H2 \implies H1.$

Proof using. `intros. unfolds hand. applies × himpl_hforall_l true. Qed.`

Lemma `himpl_hand_l_l` : $\forall H1\ H2,$
 $\text{hand } H1\ H2 \implies H2.$

Proof using. `intros. unfolds hand. applies × himpl_hforall_l false. Qed.`

Lemma `himpl_hand_r` : $\forall H1\ H2\ H3,$
 $H1 \implies H2 \rightarrow$
 $H1 \implies H3 \rightarrow$
 $H1 \implies \text{hand } H2\ H3.$

Proof using.

`intros M1 M2. unfold hand. applies himpl_hforall_r. intros b. case_if ×.`

Qed.

Lemma triple_hand_l : $\forall t\ H1\ H2\ Q$,
triple $t\ H1\ Q \rightarrow$
triple $t\ (\text{hand } H1\ H2)\ Q$.

Proof using. *introv M1*. unfold hand. *applies* $\neg$ triple_hforall true. Qed.

Lemma triple_hand_r : $\forall t\ H1\ H2\ Q$,
triple $t\ H2\ Q \rightarrow$
triple $t\ (\text{hand } H1\ H2)\ Q$.

Proof using. *introv M1*. unfold hand. *applies* $\neg$ triple_hforall false. Qed.

35.9 Generalization to N-ary Functions

35.9.1 Smart Constructors for N-ary Functions and Applications

Definition vars : Type := **list** var.

Definition vals : Type := **list** val.

Definition trms : Type := **list** trm.

trm_apps $f\ ts$ builds the application of the function f to the list of terms ts .

Fixpoint trm_apps (f :trm) (ts :trms) : **trm** :=
match ts with
| nil $\Rightarrow f$
| $ti::ts' \Rightarrow \text{trm\_apps } (\text{trm\_app } f\ ti)\ ts'$
end.

trm_funs $xs\ t$ builds a term describing a function that expects arguments with names xs , and that has t for body.

Fixpoint trm_funs (xs :vars) (t :trm) : **trm** :=
match xs with
| nil $\Rightarrow t$
| $x1::xs' \Rightarrow \text{trm\_fun } x1\ (\text{trm\_funs } xs'\ t)$
end.

val_funs $xs\ t$ is similar to trm_funs $xs\ t$ but produces a value.

Definition val_funs (xs :vars) (t :trm) : **val** :=
match xs with
| nil $\Rightarrow \text{arbitrary}$
| $x1::xs' \Rightarrow \text{val\_fun } x1\ (\text{trm\_funs } xs'\ t)$
end.

trm_fixs $f\ xs\ t$ builds a term describing a recursive function named f that expects arguments with names xs , and that has t for body.

Definition trm_fixs (f :var) (xs :vars) (t :trm) : **trm** :=

```

match xs with
| nil ⇒ arbitrary
| x1::xs' ⇒ trm_fix f x1 (trm_funs xs' t)
end.

```

`val_fixs xs t` is similar to `trm_fixs xs t` but produces a value.

```

Definition val_fixs (f:var) (xs:vars) (t:trm) : val :=
  match xs with
  | nil ⇒ arbitrary
  | x1::xs' ⇒ val_fix f x1 (trm_funs xs' t)
  end.

```

`var_funs xs n` asserts that *xs* is a nonempty list made of *n* distinct variable names.

```

Definition var_funs (xs:vars) (n:nat) : Prop :=
  LibList.noduplicates xs
  ∧ length xs = n
  ∧ xs ≠ nil.

```

`trms_vals vs` converts a list of values *vs* into a list of terms made of the corresponding values, each converted using `trm_val`.

```

Coercion trms_vals (vs:vals) : trms :=
  LibList.map trm_val vs.

```

```

Lemma trms_vals_nil :
  trms_vals nil = nil.

```

Proof using. auto. Qed.

```

Lemma trms_vals_cons : ∀ v vs,
  trms_vals (v :: vs) = trm_val v :: trms_vals vs.

```

Proof using. intros. unfold trms_vals. *rew_listx*×. Qed.

#[*local*] Hint Rewrite trms_vals_nil trms_vals_cons : *rew_listx*.

35.9.2 Evaluation Rules for N-ary Constructs

`subst_trm_funs` distributes `subst` over `trm_funs`.

```

Lemma subst_trm_funs : ∀ xs x v t1,
  ¬ mem x xs →
  subst x v (trm_funs xs t1) = trm_funs xs (subst x v t1).

```

Proof using.

```

  intros xs. induction xs as [|x xs']; introv N.
  { auto. }
  { rew_listx in N. rew_logic in N. destruct N. simpl. case_var. fequals×. }

```

Qed.

Evaluation rule for reducing `trm_funs` into `val_funs`

Lemma eval_like_trm_funs : $\forall xs\ t1,$
 $xs \neq \text{nil} \rightarrow$
 $\text{eval\_like} (\text{val\_funs } xs\ t1) (\text{trm\_funs } xs\ t1).$

Proof using.

intros N M. destruct xs; tryfalse. simpls. inverts M. applies $\times$ eval_fun.

Qed.

Auxiliary lemma: evaluation rule for reducing the first argument of a `val_funs` applied to an argument.

Lemma eval_like_app_val_funs_cons : $\forall x\ xs\ t1\ t0\ v,$
 $\text{eval\_like} (\text{val\_funs } (x :: xs)\ t1)\ t0 \rightarrow$
 $\neg \text{mem } x\ xs \rightarrow$
 $xs \neq \text{nil} \rightarrow$
 $\text{eval\_like} (\text{val\_funs } xs\ (\text{subst } x\ v\ t1)) (t0\ v).$

Proof using.

intros R Nx Hx M. unfolds eval_like. applies eval_app_arg1'.
 $\{ \text{applies } R. \text{applies eval\_val\_minimal.} \}$
 $\{ \text{simpl. intros ? ? } (-\>\&->). \text{applies } \times \text{ eval\_app\_fun.}$
 $\text{rewrite } \times \text{ subst\_trm\_funs. applies } \times \text{ eval\_like\_trm\_funs.} \}$

Qed.

Auxiliary lemma: evaluation rule for reducing a `val_funs` applied to its last argument.

Lemma eval_like_app_val_funs_one : $\forall x\ t1\ t0\ v,$
 $\text{eval\_like} ((\text{val\_funs } (x :: \text{nil})\ t1))\ t0 \rightarrow$
 $\text{eval\_like} (\text{subst } x\ v\ t1) (\text{trm\_app } t0\ v).$

Proof using.

intros R N. applies eval_app_arg1'.
 $\{ \text{applies } R. \text{applies eval\_val\_minimal.} \}$
 $\{ \text{simpl. intros ? ? } (-\>\&->). \text{applies } \times \text{ eval\_app\_fun.} \}$

Qed.

Evaluation rule for the application of a `val_funs` with `n` variables to a list of `n` values.

Lemma eval_like_trm_apps_funs : $\forall xs\ t0\ vs\ t1,$
 $\text{eval\_like} (\text{val\_funs } xs\ t1)\ t0 \rightarrow$
 $\text{var\_funs } xs\ (\text{length } vs) \rightarrow$
 $\text{eval\_like} (\text{isubst } (\text{combine } xs\ vs)\ t1) (\text{trm\_apps } t0\ (\text{trms\_vals } vs)).$

Proof using.

intros E (R1&R2&R3). gen t1 t0 vs. induction xs; intros.
 $\{ \text{false } \times. \}$
 $\{ \text{inverts } R1. \text{destruct } vs \text{ as } [|v\ vs]; \text{rew\_list in } *; \text{tryfalse.}$
 $\text{rew\_listx. rewrite isubst\_cons. simpl.}$
 $\text{tests: } (xs = \text{nil}).$
 $\{ \text{destruct } vs \text{ as } [|]; \text{tryfalse.}$
 $\text{rew\_listx. rewrite isubst\_nil. simpl.}$


```

    applies × eval_like_app_val_funs_one. }
  { applies × IHxs. applies × eval_like_app_val_funs_cons. } }

```

Qed.

Evaluation rule for application of a recursive n-ary function of the form `val_fix f (trm_funs xs t1)` to a list of values of the appropriate length.

```

Lemma eval_like_trm_apps_fixs : ∀ f xs vs t1 v0,
  v0 = val_fixs f xs t1 →
  var_funs xs (length vs) →
  ¬ mem f xs →
  xs ≠ nil →
  eval_like (isubst (combine (f :: xs) (v0 :: vs)) t1) (trm_apps v0 (trms_vals vs)).

```

Proof using.

```

  introv E F N1 N2. rew_listx. rewrite isubst_cons.
  destruct F as (F1 & F2 & F3).
  destruct xs as [|x xs']; tryfalse.
  destruct vs as [|v vs']; tryfalse.
  tests C: (xs' = nil).
  { destruct vs' as []; tryfalse.
    rew_listx. rewrite isubst_cons, isubst_nil. simpl.
    introv M. applies × eval_app_fix. }
  { rew_listx. rewrite isubst_cons. simpl. rew_listx in N1.
    rew_logic in N1. destruct N1 as (Nf & N1). inverts F1 as (Fx & F1).
    applies eval_like_trm_apps_funs.
    { introv M. applies × eval_app_fix.
      inverts M. rewrite × subst_trm_funs. rewrite × subst_trm_funs.
      applies × eval_like_trm_funs. applies × eval_val. }
    { rew_list in F2. splits ×. } }

```

Qed.

35.9.3 Tooling for N-ary Functions and Applications

`var_funs_exec xs n` is a computable version of `var_funs xs n`.

Section `Var_funs_exec`.

Import `LIBLISTEXEC.REWLISTEXEC`.

```

Definition var_funs_exec (xs:vars) (n:nat) : bool :=
  LibListExec.noduplicates var_eq xs
  && (LibNat.beq (LibListExec.length xs) n
  && LibListExec.is_not_nil xs).

```

```

Lemma var_funs_exec_eq : ∀ xs (n:nat),
  var_funs_exec xs n = isTrue (var_funs xs n).

```

Proof using.

```

intros. hint var_eq_spec.
unfold var_funs_exec, var_funs.
rewrite LibNat.beq_eq. rew_list_exec; auto.
extens; rew_istrue. splits×.
Qed.

End Var_funs_exec.

trms_to_vals is the reciprocal of trms_vals, in the sense that trms_to_vals ts = Some vs
ensure that ts = trms_vals vs.

Fixpoint trms_to_vals (ts:trms) : option vals :=
  match ts with
  | nil ⇒ Some nil
  | (trm_val v)::ts' ⇒
    match trms_to_vals ts' with
    | None ⇒ None
    | Some vs' ⇒ Some (v::vs')
    end
  | _ ⇒ None
end.

Lemma trms_to_vals_spec : ∀ ts vs,
  trms_to_vals ts = Some vs →
  ts = trms_vals vs.

Proof using.
  intros ts. induction ts as [|t ts']; simpl; introv E.
  { inverts E. auto. }
  { destruct t; inverts E as E. cases (trms_to_vals ts') as C; inverts E.
    rename v0 into vs'. rewrite× (IHts' vs'). }
Qed.

trm_apps_of_trm t is a tactic that takes a term t and returns a corresponding term in
the form trm_apps f ts, assuming it is possible.

Ltac trm_apps_of_trm t :=
  let rec aux acc t :=
    match t with
    | trm_app ?t0 ?t1 ⇒
      let acc' := constr:(t1::acc) in
      aux acc' t0
    | _ ⇒ constr:(trm_apps t acc)
  end in
  aux (@nil trm) t.

Lemma trm_apps_of_trm_demo : ∀ (f x1 x2 x3:trm) H Q,
  triple (f x1 x2 x3) H Q.

Proof using.

```

```

intros.
match goal with  $\vdash$  triple ?t _ _  $\Rightarrow$ 
  let  $r := \text{trm\_apps\_of\_trm } t$  in
  change  $t$  with  $r$  end.
Abort.

```

prove_eq_trm_apps applies to a goal of the form $t = \text{trm\_apps } ?f ?vs$, and solve it by instantiating f and vs , assuming it is possible.

```

Ltac prove_eq_trm_apps :=
  match goal with  $\vdash$  ?t = trm_apps _ _  $\Rightarrow$ 
    let  $r := \text{trm\_apps\_of\_trm } t$  in
    apply (refl_equal  $r$ ) end.

```

```

Lemma prove_eq_trm_apps_demo :  $\forall (f \ x1 \ x2 \ x3:\text{trm})$ ,
  ( $\forall t \ ts, f \ x1 \ x2 \ x3 = \text{trm\_apps } t \ ts \rightarrow (t, ts) = (t, ts) \rightarrow \text{True}$ )  $\rightarrow$ 
  True.

```

Proof using. intros. eapply *H.prove_eq_trm_apps*. Abort.

trm_funs_of_trm t is a tactic that takes a term t and returns a corresponding term in the form $\text{trm\_funs } xs \ t1$, assuming it is possible.

```

Ltac trm_funs_of_trm t :=
  let rec aux t :=
    match t with
    | trm_fun ?x ?t1  $\Rightarrow$ 
      match aux t1 with
      | trm_funs ?xs ?t0  $\Rightarrow$  constr:(trm_funs ( $x::xs$ ) t0)
      | ?t0  $\Rightarrow$  constr:(trm_funs ( $x::\text{nil}$ ) t0)
      end
    | _  $\Rightarrow$  constr:(t)
    end in
  let t := eval hnf in t in
  aux t.

```

val_funs_of_trm t is a tactic that takes a term t and returns a corresponding term in the form $\text{val\_funs } xs \ t1$, assuming it is possible.

```

Ltac val_funs_of_trm t :=
  let t := eval hnf in t in
  match t with
  | val_fun ?x ?t1  $\Rightarrow$ 
    match trm_funs_of_trm t1 with
    | trm_funs ?xs ?t0  $\Rightarrow$  constr:(val_funs ( $x::xs$ ) t0)
    | ?t0  $\Rightarrow$  constr:(val_funs ( $x::\text{nil}$ ) t0)
    end
  | _  $\Rightarrow$  constr:(t)
  end.

```

prove_eq_val_funs applies to a goal of the form $v = \text{val\_funs } ?xs \text{ ?}t1$ and solves it is possible by instantiating *xs* and *t1*

```
Ltac prove_eq_val_funs :=
  match goal with  $\vdash ?t = \text{val\_funs } \_ \Rightarrow$ 
    let  $r := \text{val\_funs\_of\_trm } t$  in
    apply (refl_equal  $r$ ) end.
```

```
Lemma prove_eq_val_funs_demo :  $\forall (x1 \ x2:\text{var}) (t0:\text{trm}),$ 
  ( $\forall xs \ t, \text{val\_fun } x1 (\text{trm\_fun } x2 \ t0) = \text{val\_funs } xs \ t \rightarrow (xs, t) = (xs, t) \rightarrow \text{True}$ )  $\rightarrow$ 
  True.
```

Proof using. intros. eapply *H. prove_eq_val_funs*. Abort.

val_fixs_of_trm *t* is a tactic that takes a term *t* and returns a corresponding term in the form *val_fixs xs t1*, if possible.

```
Ltac val_fixs_of_trm t :=
  let  $t := \text{eval hnf in } t$  in
  match  $t$  with
  |  $\text{val\_fix } ?f \ ?x \ ?t1 \Rightarrow$ 
    match  $\text{trm\_funs\_of\_trm } t1$  with
    |  $\text{trm\_funs } ?xs \ ?t0 \Rightarrow \text{constr}:(\text{val\_fixs } f (x::xs) \ t0)$ 
    |  $\_ \Rightarrow \text{constr}:(\text{val\_fixs } f (x::\text{nil}) \ t1)$ 
    end
  |  $\_ \Rightarrow \text{constr}:(t)$ 
  end.
```

prove_eq_val_fix_trm_funs applies to a goal of the form $v = \text{val\_fix } ?f (\text{trm\_funs } ?xs \ ?t1)$ and solves it is possible.

```
Ltac prove_eq_val_fix_trm_funs :=
  match goal with  $\vdash ?t = \text{val\_fixs } \_ \_ \_ \Rightarrow$ 
    let  $r := \text{val\_fixs\_of\_trm } t$  in
    apply (refl_equal  $r$ ) end.
```

```
Lemma prove_eq_val_fix_trm_funs_demo :  $\forall (x1 \ x2:\text{var}) (t0:\text{trm}),$ 
  ( $\forall f \ xs \ t, \text{val\_fix } f \ x1 (\text{trm\_fun } x2 \ t0) = \text{val\_fixs } f \ xs \ t \rightarrow (f, xs, t) = (f, xs, t) \rightarrow \text{True}$ )
 $\rightarrow$ 
  True.
```

Proof using. intros. eapply *H. prove_eq_val_fix_trm_funs*. Abort.

35.9.4 Implementation of the Tactic *xwp*

```
Lemma xwp_lemma_funs :  $\forall t \ v0 \ ts \ vs \ xs \ t1 \ H \ Q,$ 
   $t = \text{trm\_apps } v0 \ ts \rightarrow$ 
   $v0 = \text{val\_funs } xs \ t1 \rightarrow$ 
   $\text{trms\_to\_vals } ts = \text{Some } vs \rightarrow$ 
```

```

var_funs_exec xs (LibList.length vs) →
H ==> (wpgen (LibListExec.combine xs vs) t1) Q →
triple t H Q.

```

Proof using.

```

intros → E M1 F M2. rewrite var_funs_exec_eq in F. rew_bool_eq in F.
lets HL: (proj32 F). rewrite× LibListExec.combine_eq in M2.
lets ->: trms_to_vals_spec (rm M1).
rewrite ← wp_equiv. xchange (rm M2). xchange wpgen_sound.
applies wp_eval_like. applies× eval_like_trm_apps_funs.
{ subst. intros M. applies M. }

```

Qed.

```

Lemma xwp_lemma_fixs : ∀ t v0 ts vs f xs t1 H Q,
t = trm_apps v0 ts →
v0 = val_fixs f xs t1 →
trms_to_vals ts = Some vs →
var_funs_exec xs (LibList.length vs) →
negb (LibListExec.mem var_eq f xs) →
H ==> (wpgen (LibListExec.combine (f::xs) (v0::vs)) t1) Q →
triple t H Q.

```

Proof using.

```

intros → E M1 F N1 M2. rewrite var_funs_exec_eq in F. rew_bool_eq in F.
lets HL: (proj32 F). rewrite× LibListExec.combine_eq in M2; [| rew_list* ].
lets Nxs: (proj33 F). lets ->: trms_to_vals_spec (rm M1).
rewrite LibListExec.mem_eq in N1; [| applies var_eq_spec ]. rew_bool_eq in N1.
rewrite ← wp_equiv. xchange (rm M2). xchange wpgen_sound.
applies wp_eval_like. applies× eval_like_trm_apps_fixs.

```

Qed.

Tactic Notation "xwp" :=

```

intros;
first [ applies xwp_lemma_funs;
      [ prove_eq_trm_apps
        | prove_eq_val_funs
        | reflexivity
        | try reflexivity
        | ]
      | applies xwp_lemma_fixs;
      [ prove_eq_trm_apps
        | prove_eq_val_fix_trm_funs
        | reflexivity
        | try reflexivity
        | try reflexivity
        | ] ];

```

xwp_simpl.

35.9.5 Specification of Array Operations

See chapter *Array*.

35.9.6 Specification of Record Operations

The chapter *Struct* shows how to these specifications may be realized.

Implicit Types $k : \mathbf{nat}$.

Representation of Records

A field name is implemented as a natural number

Definition `field` : `Type` := `nat`.

A record field is described as the pair of a field and a value stored in that field.

Definition `hrecord_field` : `Type` := `(field × val)`.

A record consists of a list of fields.

Definition `hrecord_fields` : `Type` := `list hrecord_field`.

Implicit Types $kvs : \mathbf{hrecord\_fields}$.

Record fields syntax, e.g., `{ head := x; tail := q }`.

Notation `"{ k1 := v1 }"` :=
`((k1, (v1:val))::nil)`
(at level 0, $k1$ at level 0, *only parsing*)
: *val_scope*.

Notation `"{ k1 := v1 ; k2 := v2 }"` :=
`((k1, (v1:val))::(k2, (v2:val))::nil)`
(at level 0, $k1, k2$ at level 0, *only parsing*)
: *val_scope*.

Notation `"{ k1 := v1 ; k2 := v2 ; k3 := v3 }"` :=
`((k1, (v1:val))::(k2, (v2:val))::(k3, (v3:val))::nil)`
(at level 0, $k1, k2, k3$ at level 0, *only parsing*)
: *val_scope*.

Notation `"{ k1 := v1 }"` :=
`((k1, v1)::nil)`
(at level 0, $k1$ at level 0, *only printing*)
: *val_scope*.

Notation `"{ k1 := v1 ; k2 := v2 }"` :=
`((k1, v1)::(k2, v2)::nil)`

(at level 0, $k1$, $k2$ at level 0, *only printing*)
 $: val\_scope$.

Notation "{ $k1 := v1$; $k2 := v2$; $k3 := v3$ }" :=
 $((k1, v1) :: (k2, v2) :: (k3, v3) :: nil)$
 (at level 0, $k1$, $k2$, $k3$ at level 0, *only printing*)
 $: val\_scope$.

Open Scope val_scope .

$hrecord\ kvs\ p$, written $p \sim\sim\sim > kvs$, describes a record at location p , with fields described by the list kvs .

Parameter $hrecord : \forall (kvs : hrecord\_fields) (p : loc), hprop$.

Notation " $p \sim\sim\sim > kvs$ " := $(hrecord\ kvs\ p)$
 (at level 32) : $hprop\_scope$.

The heap predicate $hrecord\ kvs\ p$ captures in particular the invariant that the location p is not null.

Parameter $hrecord\_not\_null : \forall p\ kvs,$
 $hrecord\ kvs\ p \implies hrecord\ kvs\ p \setminus * \setminus [p \neq null]$.

Read Operation on Record Fields

$val\_get\_field\ k\ p$, written $p'.k$, reads field k from the record at location p .

Parameter $val\_get\_field : field \rightarrow val$.

Notation " $t1 \text{'.'} k$ " :=
 $(val\_get\_field\ k\ t1)$
 (in custom *trm* at level 56, k at level 0, *format* " $t1 \text{'.'} k$ ") : trm_scope .

Generator of specifications of val_get_field .

Fixpoint $hfields\_lookup (k : field) (kvs : hrecord\_fields) : option\ val :=$
 $match\ kvs\ with$
 $| nil \Rightarrow None$
 $| (ki, vi) :: kvs' \Rightarrow if\ Nat.eq\_dec\ k\ ki$
 $\quad then\ Some\ vi$
 $\quad else\ hfields\_lookup\ k\ kvs'$
 end .

Specification of val_get_field in terms of $hrecord$.

Parameter $triple\_get\_field\_hrecord : \forall\ kvs\ p\ k\ v,$
 $hfields\_lookup\ k\ kvs = Some\ v \rightarrow$
 $triple\ (val\_get\_field\ k\ p)$
 $(hrecord\ kvs\ p)$
 $(fun\ r \Rightarrow \setminus [r = v] \setminus * hrecord\ kvs\ p).$

Write Operation on Record Fields

`val_set_field` `k p v`, written `Set  $p.k$  := v`, update the contents of the field `k` from the record at location `p`, with the value `v`.

Parameter `val_set_field` : `field` $\rightarrow$ `val`.

Notation "`t1` `'.'` `k` `':='` `t2`" :=
(`val_set_field k t1 t2`)
(in `custom trm` at level 56,
`k` at level 0, `format` "`t1` `'.'` `k` `':='` `t2`")
: `trm_scope`.

Generator of specifications for `val_set_field`.

Fixpoint `hfields_update` (`k:field`) (`v:val`) (`kvs:hrecord_fields`)
: `option hrecord_fields` :=

```
match kvs with
| nil  $\Rightarrow$  None
| (ki, vi) :: kvs'  $\Rightarrow$  if Nat.eq_dec k ki
    then Some ((k, v) :: kvs')
    else match hfields_update k v kvs' with
        | None  $\Rightarrow$  None
        | Some LR  $\Rightarrow$  Some ((ki, vi) :: LR)
    end
end.
```

Specification of `val_set_field` in terms of `hrecord`.

Parameter `triple_set_field_hrecord` : $\forall$ `kvs kvs' k p v`,
`hfields_update k v kvs = Some kvs'  $\rightarrow$`
`triple (val_set_field k p v)`
`(hrecord kvs p)`
`(fun _  $\Rightarrow$  hrecord kvs' p).`

Allocation of Records

`val_alloc_hrecord` `ks` allocates a record with fields `ks`, storing uninitialized contents.

Parameter `val_alloc_hrecord` : $\forall$ (`ks:list field`), `trm`.

Parameter `triple_alloc_hrecord` : $\forall$ `ks`,
`ks = nat_seq 0 (LibListExec.length ks)  $\rightarrow$`
`triple (val_alloc_hrecord ks)`
`\ []`
`(funloc p  $\Rightarrow$  hrecord (LibListExec.map (fun k  $\Rightarrow$  (k, val_uninit)) ks) p).`

An arity-generic version of the definition of allocation combined with initialization is beyond the scope of this course. We only include constructors for arity 2 and 3.

val_new_record_2 *k1 k2 v1 v2*, written ‘{ *k1* := *v1* ; *k2* := *v2* }, allocates a record with two fields and initialize the fields.

Parameter *val_new_hrecord_2* : $\forall (k1:\text{field}) (k2:\text{field}), \text{val}$.

Notation "{ *k1* := *v1* ; *k2* := *v2* }" :=
 (*val_new_hrecord_2 k1 k2 v1 v2*)
 (in *custom trm* at level 65,
k1, *k2* at level 0,
v1, *v2* at level 65) : *trm_scope*.

Open Scope *trm_scope*.

Parameter *triple_new_hrecord_2* : $\forall k1\ k2\ v1\ v2,$
k1 = 0%nat $\rightarrow$
k2 = 1%nat $\rightarrow$
 triple <{ ‘{ *k1* := *v1* ; *k2* := *v2* } }>
 \ []
 (funloc *p* $\Rightarrow$ *p* ~~~> ‘{ *k1* := *v1* ; *k2* := *v2* }).

#[global] Hint Resolve *val_new_hrecord_2* : *triple*.

val_new_record_3 *k1 k2 k3 v1 v2 v3*, written ‘{ *k1* := *v1* ; *k2* := *v2* ; *k3* := *v3* }, allocates a record with three fields and initialize the fields.

Parameter *val_new_hrecord_3* : $\forall (k1:\text{field}) (k2:\text{field}) (k3:\text{field}), \text{val}$.

Notation "{ *k1* := *v1* ; *k2* := *v2* ; *k3* := *v3* }" :=
 (*val_new_hrecord_3 k1 k2 k3 v1 v2 v3*)
 (in *custom trm* at level 65,
k1, *k2*, *k3* at level 0,
v1, *v2*, *v3* at level 65) : *trm_scope*.

Parameter *triple_new_hrecord_3* : $\forall k1\ k2\ k3\ v1\ v2\ v3,$
k1 = 0%nat $\rightarrow$
k2 = 1%nat $\rightarrow$
k3 = 2%nat $\rightarrow$
 triple <{ ‘{ *k1* := *v1* ; *k2* := *v2* ; *k3* := *v3* } }>
 \ []
 (funloc *p* $\Rightarrow$ *p* ~~~> ‘{ *k1* := *v1* ; *k2* := *v2* ; *k3* := *v3* }).

#[global] Hint Resolve *val_new_hrecord_3* : *triple*.

Deallocation of Records

val_dealloc_hrecord *p*, written *delete p*, deallocates the record at location *p*.

Parameter *val_dealloc_hrecord* : **val**.

Notation "'delete'" :=
 (trm_val *val_dealloc_hrecord*)

```

(in custom trm at level 0) : trm_scope.
Parameter triple_dealloc_hrecord : ∀ kvs p,
  triple (val_dealloc_hrecord p)
    (hrecord kvs p)
    (fun _ => \[]).
#[global] Hint Resolve triple_dealloc_hrecord : triple.

```

Extending *xapp* to Support Record Access Operations

The tactic *xapp* is refined to automatically invoke the lemmas *triple_get_field_hrecord* and *triple_set_field_hrecord*.

```

Parameter xapp_get_field_lemma : ∀ H k p Q,
  H ==> \∃ kvs, (hrecord kvs p) \*
    match hfields_lookup k kvs with
    | None => \[False]
    | Some v => ((fun r => \[r = v] \* hrecord kvs p) \-* protect Q) end →
  H ==> wpgen_app (val_get_field k p) Q.
Parameter xapp_set_field_lemma : ∀ H k p v Q,
  H ==> \∃ kvs, (hrecord kvs p) \*
    match hfields_update k v kvs with
    | None => \[False]
    | Some kvs' => ((fun _ => hrecord kvs' p) \-* protect Q) end →
  H ==> wpgen_app (val_set_field k p v) Q.
Ltac xapp_nosubst_for_records tt ::=
  first [ applies xapp_set_field_lemma; xsimpl; simpl; xapp_simpl
    | applies xapp_get_field_lemma; xsimpl; simpl; xapp_simpl ].

```

Extending *xsimpl* to Simplify Record Equalities

fequals_fields simplifies equalities between values of types *hrecord_fields*, i.e. list of pairs of field names and values.

```

Ltac fequals_fields :=
  match goal with
  | ⊢ nil = nil => reflexivity
  | ⊢ cons _ _ = cons _ _ => applies args_eq_2; [ fequals | fequals_fields ]
  end.

```

At this point, the tactic *xsimpl* is refined to take into account simplifications of predicates of the form $p \sim\sim\sim > kvs$. The idea is to find a matching predicate of the form $p \sim\sim\sim > kvs'$ on the right-hand side of the entailment, then to simplify the list equality $kvs = kvs'$.

```

Ltac xsimpl_hook_hrecord p :=
  xsimpl_pick_st ltac:(fun H' =>

```

```

      match H' with hrecord ?kvs ' p ⇒
        constr:(true) end);
  apply xsimpl_lr_cancel_eq;
  [ fequal; first [ reflexivity | try fequals_fields ] | ].

Ltac xsimpl_hook H ::=
  match H with
  | hsingle ?p ?v ⇒ xsimpl_hook_hsingle p
  | hrecord ?kvs ?p ⇒ xsimpl_hook_hrecord p
  end.

```

35.10 Demo Programs

```

Module DEMOPROGRAMS.
Export PROGRAMSYNTAX.

```

Definition and Verification of *incr*.

Here is an implementation of the increment function, written in A-normal form.

OCaml:

```
let incr p = let n = !p in let m = n + 1 in p := m
```

The notation '*p*' stands for ("*x*":*var*). It is defined in *LibSepVar.v*.

Definition *incr* : **val** :=

```

<{ fun 'p ⇒
  let 'n = ! 'p in
  let 'm = 'n + 1 in
  'p := 'm }>.

```

Here is the Separation Logic triple specifying increment. And the proof follows. Note that the script contains explicit references to the specification lemmas of the functions being called (e.g. *triple_get* for the *get* operation). The actual CFML2 setup is able to automatically infer those names.

```

Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (incr p)
    (p ~~> n)
    (fun _ ⇒ (p ~~> (n+1))).

```

Proof using.

```
xwp. xapp. xapp. xapp. xsimpl×
```

Qed.

We register this specification so that it may be automatically invoked in further examples.

```
#[global] Hint Resolve triple_incr : triple.
```

Alternative definition using variable names without quotes, obtained by importing the module *Vars* from *LibSepVar.v*.

Module Export DEF_INCR. Import VARS.

Definition incr' : **val** :=

```
<{ fun p =>
  let n = ! p in
  let m = n + 1 in
  p := m }>.
```

End DEF_INCR.

Lemma triple_incr' : $\forall (p:\text{loc}) (n:\text{int}),$
 $\text{triple}(\text{incr}'\ p)$
 $(p \rightsquigarrow n)$
 $(\text{fun } - \Rightarrow (p \rightsquigarrow (n+1))).$

Proof using.

xwp. xapp. xapp. xapp. xsimpl×.

Qed.

The Decrement Function

Definition decr : **val** :=

```
<{ fun 'p =>
  let 'n = ! 'p in
  let 'm = 'n - 1 in
  'p := 'm }>.
```

Module Export DEF_DECR. Import VARS.

Definition decr : **val** :=

```
<{ fun p =>
  let n = !p in
  let m = n - 1 in
  p := m }>.
```

End DEF_DECR.

Lemma triple_decr : $\forall (p:\text{loc}) (n:\text{int}),$
 $\text{triple}(\text{trm\_app decr } p)$
 $(p \rightsquigarrow n)$
 $(\text{fun } - \Rightarrow p \rightsquigarrow (n-1)).$

Proof using.

xwp. xapp. xapp. xapp. xsimpl×.

Qed.

#[global] Hint Resolve triple_decr : triple.

Definition and Verification of **mysucc**.

Another example, involving a call to *incr*.

```
Module Export DEF_MYSUCC. Import VARS.
```

```
Definition mysucc : val :=
```

```
<{ fun n =>
  let r = ref n in
  incr r;
  let x = !r in
  free r;
  x }>.
```

```
End DEF_MYSUCC.
```

```
Lemma triple_mysucc :  $\forall$  n,
  triple (trm_app mysucc n)
    \ []
    (fun v => \ [v = n+1]).
```

Proof using.

```
xwp. xapp. intros r. xapp. xapp. xapp. xval. xsimpl. auto.
Qed.
```

Definition and Verification of **myrec**.

Another example, involving a function involving 3 arguments, as well as record operations.
OCaml:

```
let myrec r n1 n2 = r.myfield := r.myfield + n1 + n2
```

```
Definition myfield : field := 0%nat.
```

```
Module Export DEF_MYREC. Import VARS.
```

```
Definition myrec : val :=
```

```
<{ fun p n1 n2 =>
  let n = (p'.myfield) in
  let m1 = n + n1 in
  let m2 = m1 + n2 in
  p'.myfield := m2 }>.
```

```
Lemma triple_myrec :  $\forall$  (p:loc) (n n1 n2:int),
  triple (myrec p n1 n2)
    (p ~~~> '{ myfield := n})
    (fun _ => p ~~~> '{ myfield := (n+n1+n2) }).
```

Proof using.

```
xwp. xapp. xapp. xapp. xapp. xsimpl.
Qed.
```

```
End DEF_MYREC.
```

Definition and Verification of **myfun**.

Another example involving a local function definition.

```
Module Export DEF_MYFUN. Import VARS.
```

```
Definition myfun : val :=
```

```
  <{ fun p =>
    let f = (fun _ u => incr p) in
    f();
    f() }>.
```

```
End DEF_MYFUN.
```

```
Lemma triple_myfun :  $\forall (p:\text{loc}) (n:\text{int}),$   
  triple (myfun p)  
    ( $p \rightsquigarrow n$ )  
    (fun _ =>  $p \rightsquigarrow (n+2)$ ).
```

Proof using.

```
  xwp.  
  xfun (fun (f:val) =>  $\forall (m:\text{int}),$   
    triple (f())  
      ( $p \rightsquigarrow m$ )  
      (fun _ =>  $p \rightsquigarrow (m+1)$ )); intros f Hf.  
  { intros. applies Hf. clear Hf. xapp. xsimpl. }  
  xapp. xapp.  
  replace (n+1+1) with (n+2); [|math|. xsimpl.  
Qed.
```

Demo of a Function with Multiple Arguments.

```
Definition project_32 : val :=
```

```
  <{ fun 'a 'b 'c => 'b }>.
```

```
Lemma triple_project_32 :  $\forall (a\ b\ c : \text{int}),$   
  triple (project_32 a b c)  
    \ []  
    (fun r => \[r = b]).
```

Proof using. *xwp. xval. xsimpl*×. Qed.

```
Definition project_32_rec : val :=
```

```
  <{ fix 'f 'a 'b 'c => if false then 'f 'b end }>.
```

```
Lemma triple_project_32_rec :  $\forall (a\ b\ c : \text{int}),$   
  triple (project_32_rec a b c)  
    \ []  
    (fun _ => \ []).
```

Proof using. *xwp. xif; auto_false. intros _.* *xval. xsimpl.* Qed.

End DEMOPROGRAMS.

Chapter 36

Library SLF.Preface

36.1 Preface: Introduction to the Course

36.2 Welcome

This electronic book is Volume 6 of the *Software Foundations* series, which presents the mathematical underpinnings of reliable software.

COMPANION COURSE NOTES: A large part of this course is also written up in traditional LaTeX-style presentation. It's available from http://www.chargueraud.org/teach/verif/slf_notes.pdf.

This book will teach you about the foundations of Separation Logic, a practical approach to the modular verification of imperative programs. In particular, it presents the building blocks for constructing a program verification tool. It does not, however, focus on reasoning about data structures and algorithms using Separation Logic. This aspect is covered to some extent by Volume 5 of *Software Foundations*, which presents Verifiable C, a program logic and proof system for C. For OCaml programs, this aspect will be covered in a yet-to-be-written volume presenting CFML, a tool that builds upon all the techniques presented in this volume.

You are only assumed to understand the material in *Software Foundations* Volume 1 (*Logical Foundations*), and the two chapters on Hoare Logic (Hoare and Hoare2) from *Software Foundations* Volume 2 (*PL Foundations*). Volume 5 is not a prerequisite. The exposition here is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers.

36.3 Separation Logic

Separation Logic is a *program logic*: it enables one to establish that a program satisfies its specification. Specifications are expressed using triples of the form $\{H\} \text{ t } \{Q\}$. Whereas in Hoare logic the precondition H and the postcondition Q describe the whole memory state, in Separation Logic H and Q describe only a fragment of the memory state that includes

the resources necessary to the execution of t .

A key ingredient of Separation Logic is the frame rule, which enables modular proofs. It is stated as follows.

$$\{ H \} t \{ Q \}$$

$$\{ H \setminus^* H' \} t \{ Q \setminus^* H' \}$$

The above rule asserts that if, a term t executes correctly with the resources H and produces Q , then t admits the same behavior in a larger memory state, described by the union of H with a disjoint component H' , producing the postcondition Q extended with that same resource H' unmodified. The star symbol $\setminus^*$ denotes the *separating conjunction* operator of Separation Logic.

Separation Logic can be exploited in three kind of tools.

- Automated proofs: the user provides only the code, and the tool locates sources of potential bugs. A good automated tool provides feedback that (at least most of time) is relevant.
- Semi-automated proofs: the user provides not just the code, but also specifications and invariants. The tool then leverages automated solvers (e.g., SMT solvers) to discharge proof obligations.
- Interactive proofs: the user provides not just the code and its specifications, but also a detailed proof script justifying the correctness of the code. These proofs may be developed interactively using a proof assistant such as Rocq.

The present course focuses on the third approach, that is, the integration of Separation Logic in an interactive proof assistant. This approach has been successfully put to practice throughout the world, using various proof assistants (Rocq, Isabelle/HOL, HOL), targeting different languages (Assembly, C, SML, OCaml, Rust...) and for verifying various kind of programs, ranging from low-level operating system kernels to high-level data structures and algorithms.

36.4 Separation Logic in a Proof Assistant

The benefits of exploiting Separation Logic in a proof assistant include at least four major points:

- higher-order logic provides virtually unlimited expressiveness that enables formulating arbitrarily complex specifications and invariants;
- a proof assistant provides a unified framework to prove both the implementation details of the code and the underlying mathematical results, e.g., results from number theory or graph theory;

- proof scripts may be easily maintained to reflect changes to the source code; and
- the fact that Separation Logic itself is formalized in the proof assistant provides high confidence in the correctness of the tool.

Pretty much all the tools that leverage Separation Logic in proof assistants are constructed following the same schema:

- A formalization of the syntax and semantics of the source language. This is called a *deep embedding* of the programming language.
- A definition of Separation Logic predicates as predicates in higher-order logic. This is called a *shallow embedding* of the program logic.
- A definition of Separation Logic triples as a predicate, the statements of the reasoning rules as lemmas, and the proof of these reasoning rules with respect to the semantics.
- An infrastructure that consists of lemmas, tactics and notations, allowing for the verification of concrete programs to be carried out through relatively concise proof scripts.
- Applications of this infrastructure to the verification of concrete programs.

The purpose of this course is to explain how to set up such a construction. To that end, we consider in this course the simplest possible variant of Separation Logic, and apply it to a minimalistic imperative programming language. The language essentially consists of a lambda-calculus with references. This language admits a simple semantics. It avoids, in particular, the need to distinguish between stack variables and heap-allocated variables. Advanced chapters later in the course explain how to add support for loops, records, arrays, and n-ary functions.

36.5 Multiple Reading Depths

All chapters except the first one are organized in three parts.

- The *First Pass* section presents the most important ideas only. The course is designed in such a way that it is possible to read only the *First Pass* section of every chapter. The reader may be interested in going through all these sections to get the big picture, before revisiting each chapter in more detail.
- The *More Details* section presents additional material explaining in more depth the meaning and the consequences of the key results. This section also contains descriptions of the most important proofs. By default, readers would eventually read all this material.

- The *Optional Material* section typically contains the remaining proofs, as well as discussions of alternative definitions. The *Optional Material* sections are all independent from each other. These sections are intended for (1) readers who plan to continue studying Separation Logic beyond the present course, and (2) teachers using the course.

36.6 Chapters Overview

The first two chapters, **Basic** and **Repr**, give a primer on how to prove imperative programs in Separation Logic, i.e. they focus on the end user’s perspective. The eight following chapters focus on the implementor’s perspective, explaining how Separation Logic is defined and how a practical verification tool can be constructed. The last three chapters cover language extensions, from the perspective of both the user and the implementor.

The list of chapters appears below. The numbering corresponds to *teaching units*: if the chapters were taught as part of a University course, one could reasonably aim to cover one teaching unit per week.

- (1) **Basic**: introduction to the core features of Separation Logic,
- (2) **Repr**: introduction to representation predicates in Separation Logic, in particular for describing mutable lists and trees.
- (2) **Hprop**: definition of the core operators of Separation Logic.
- (2) **Himpl**: definition of the entailment relation, statement and proofs of its fundamental properties, and description of the simplification tactic for entailment.
- (3) **Triples**: definition of Separation Logic triples in terms of the semantics of the programming language.
- (3) **Rules**: statement and proofs of the reasoning rules of Separation Logic, and example proofs of programs using these rules.
- (4) **Wand**: introduction of the magic wand operator and other Separation Logic operators, and to the ramified frame rule.
- (4) **WPsem**: definition of the semantic notion of weakest precondition, and statement of rules in weakest-precondition style.
- (5) **WPgen**: presentation of a function that effectively computes the weakest precondition of a term, independently of its specification.
- (5) **WPsound**: soundness proof for the weakest precondition generator (mostly optional).

- (6) **Affine**: description of a generalization of Separation Logic with affine heap predicates, which are useful, in particular, for handling garbage-collected programming languages.
- (6) **Arrays**: specification of both ML-style arrays with headers and C-style arrays with pointer arithmetic.
- (6) **Records**: representation predicate for records, allowing us to isolate arbitrary subsets of a record's fields.

36.7 Other Distributed Files

The chapters listed above depend on a number of auxiliary files, which the reader does not need to go through but might be interested in looking at, either by curiosity, or for checking out a specific implementation detail.

- **LibSepReference**: a long file that defines the program verification tool that is used in the first two chapters, and whose implementation is discussed throughout the other chapters. Each chapter from the course imports this module, as opposed to importing earlier chapters.
- **LibSepVar**: a formalization of program variables, together with a bunch of notations for parsing variables.
- **LibSepFmap**: a formalization of finite maps, which are used for representing the memory state.
- **LibSepSimpl**: a functor that implements a powerful tactic for automatically simplifying entailments in Separation Logic.
- **LibSepMinimal**: a minimalistic formalization of a soundness proof for Separation Logic, up to the verification of the *incr* function.
- All other *Lib* $\times$ files are imports from the TLC library, which is described next.

The TLC library is a collection of general purpose theory and tactics developed over the years by Arthur Chaguéraud. The TLC library is exploited in this course to streamline the presentation. TLC provides, in particular, extensions for classical logic and tactics that are particularly well suited for meta-theory. Prior knowledge of TLC is not required, and all exercises can be completed without using TLC tactics.

The classical logic aspects of TLC are presented in chapter **Hprop**. Each TLC tactic is introduced when it is first used. Most of these tactics are also presented in the chapter *UseTactics* of Software Foundations Volume 2 (*Programming Language Foundations*).

36.8 Practicalities

36.8.1 System Requirements

Install instructions for Rocq and IDEs may be found on this page: <https://www.chargueraud.org/teach/verif/>

The files you are reading have been tested with Rocq version 9.0.0 but may also work with other versions.

36.8.2 Feedback Welcome

If you intend to use this course either in class or for self-study, the author would love to hear from you. Just knowing in which contexts the course has been used and how much of the text students were able to cover is very valuable information.

You can send feedback to `slf-at-chargueraud.org`.

If you plan on providing any non-small amount of feedback, do not hesitate to ask the author to be added as contributor to the github repository.

36.8.3 Exercises

Each chapter includes numerous exercises. The star rating scheme is described in the *Preface* of Software Foundations Volume 1 (*Logical Foundations*).

Disclaimer: the difficulty ratings currently in place are fairly speculative. Your feedback is very much welcome.

Disclaimer: (for instructors) the auto-grading system has not been tested for this volume. If you are interested in using auto-grading for this volume, please contact the author.

36.8.4 Recommended Citation Format

If you want to refer to this volume in your own writing, please do so as follows:

```
@book {Chargueraud:SF6, author = {Arthur Charguéraud}, editor = {Benjamin C.
Pierce}, title = "Separation Logic Foundations", series = "Software Foundations", volume =
"6", year = "2026", publisher = "Electronic textbook", note = {Version 3.0, \URL{http://softwarefoundation
}}, }
```

36.9 Thanks

The development of the technical infrastructure for the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep Specification*.

Chapter 37

Library **SLF.Basic**

37.1 Basic: Basic Proofs in Separation Logic

```
Set Implicit Arguments.
From SLF Require Import LibSepReference.
Import PROGRAMSYNTAX DEMOPROGRAMS.

Implicit Types n m : int.
Implicit Types p q : loc.
```

37.2 A First Taste

This chapter gives an overview of the basic features of Separation Logic, by means of examples. Example programs are specified and verified using a Separation Logic framework whose construction is explained throughout the course.

37.2.1 Parsing of Programs

The programs we consider are written within Rocq, using a “custom grammar” that allows writing code that reads almost like OCaml code. For example, consider the function *incr*, which increments the contents of a mutable cell that stores an integer. In OCaml syntax, this function could be defined as:

OCaml:

```
let incr = fun p -> let n = !p in let m = n + 1 in p := m
```

In Rocq, the corresponding program is described as shown below. The function defined, named *incr*, admits the type **val**. This type is defined by the framework. Observe that all variable names are prefixed with a quote symbol. This presentation avoids conflict between program variables and Rocq constants.

```
Definition incr : val :=
  <{ fun 'p =>
```

```

let 'n = !'p in
let 'm = 'n + 1 in
'p := 'm }>.

```

There is no need to learn how to write programs in this custom syntax: source code is provided for all the programs involved in this course.

To simplify the implementation of the framework and the reasoning about programs, we make throughout the course the simplifying assumption that programs are written in “A-normal form”: all intermediate expressions must be named using a let-binding.

37.2.2 Specification of the Increment Function

The specification of `incr p` is expressed using a “Separation Logic triple”, that is, a predicate of the form `triple t H Q`. The term `t` here corresponds to the application of the function `incr` to the argument `p`. We could write this application in the form `<{ incr p }>`, using the custom syntax for parsing programs.

The components `H` and `Q` correspond to the precondition and to the postcondition, which are explained next. To improve readability, we follow the convention of writing both the precondition and the postcondition on separate lines.

```

Lemma triple_incr : ∀ (p:loc) (n:int),
  triple <{ incr p }>
    (p ~~> n)
    (fun _ => (p ~~> (n+1))).

```

In the specification above, `p` denotes the “location” – that is, the address in memory of the reference cell provided as argument to the increment function. Locations have type `loc` in the framework.

The precondition is written `p ~~> n`. This Separation Logic predicate describes a memory state in which the contents of the location `p` is the value `n`. In the present example, `n` stands for an integer value.

The behavior of the operation `incr p` consists of updating the memory state by incrementing the contents of the cell at location `p`, updating its contents to `n+1`. Thus, the memory state posterior to the increment operation is described by the predicate `p ~~> (n+1)`.

The result value returned by `incr p` is the unit value, which does not carry any useful information. In the specification of `incr`, the postcondition is of the form `fun _ => ...`, indicating that there is no need to bind a name for the unit result value.

The general pattern of a specification admits the following scheme.

- Quantification of the arguments of the functions—here, the variable `p`.
- Quantification of the “ghost variables” used to describe the input state—here, the variable `n`.
- The application of the predicate `triple` to the function application `incr p`—here, the term being specified by the triple.

- The precondition describing the input state—here, the predicate $p \sim\sim > n$.
- The postcondition describing both the output value and the output state. The general pattern is $\text{fun } r \Rightarrow H'$, where r names the result and H' describes the final state. Here, r is just an underscore symbol, and the final state is described by $p \sim\sim > (n+1)$.

Note that we have to write $p \sim\sim > (n+1)$ using parentheses around $n+1$, because otherwise $p \sim\sim > n+1$ would get parsed as $(p \sim\sim > n) + 1$.

37.2.3 Verification of the Increment Function

Our next step is to prove the specification lemma `triple_incr` which specifies the behavior of the function `incr`. We conduct the proof using tactics provided by the frameworks, collectively called “x-tactics” because their names all start with the letter “x”. These tactics include `xwp` for starting a proof, `xapp` for reasoning about a function call, and `xsimpl` for proving that a description of a state entails another one.

Proof.

`xwp` begins the verification proof. `xwp`.

The proof obligation is displayed using a custom notation of the form *PRE H CODE F POST Q*. In the *CODE* section, one should be able to somewhat recognize the body of `incr`. Indeed, if we ignore the back-ticks and perform the alpha-renaming from v to n and $v0$ to m , the *CODE* section reads like:

```
< Let n := App val_get p in Let m := App val_add n 1 in App val_set p ) >
```

which is somewhat similar to the original source code, but displayed using a special syntax whose meaning will be explained in chapter `WPgen`.

The remainder of the proof performs essentially a symbolic execution of the code. At each step, one should not attempt to read the full proof obligation, but instead only look at the current state, described by the *PRE* part (here, $p \sim\sim > n$), and at the first line only of the *CODE* part, which corresponds to the next operation to reason about. Each of the operations involved here is handled using the tactic `xapp`.

First, we reason about the operation `!p` that reads into `p`; this read operation returns the value `n`. `xapp`.

Second, we reason about the addition operation `n+1`. `xapp`.

Third, we reason about the update operation `p := n+1`, thereby updating the state to $p \sim\sim > (n+1)$. `xapp`.

At this stage, the proof obligation takes the form $H1 \implies H2$. It requires us to check that the final state matches what is claimed in the postcondition. We discharge it using the tactic `xsimpl`. `xsimpl`.

Qed.

This completes the verification of the lemma `triple_incr`, which establishes a formal specification for the increment function. Before moving on to another function, we add the lemma `triple_incr` to a hint database called `triple`, using the command shown below. If at some point

we verify a function that includes a call to *incr*, the *xapp* tactic will be able to automatically invoke the lemma *triple_incr*.

```
#[global] Hint Resolve triple_incr : triple.
```

To minimize the amount of syntactic noise in specifications, we leverage an advanced feature of Rocq’s coercion mechanism. Concretely, instead of writing the specification in the form *triple* <{ *incr* *p* }> ..., we write it in the form *triple* (*incr* *p*) ..., that is, with just parentheses. Thanks to the coercion mechanism, explained in more detail in chapter *Rules*, when Rocq sees a “program value” *incr* being applied to an argument *p*, it automatically interprets this as a “program function call” of *incr* to *p*. Thus, the specification of the increment function can be written as follows.

```
Lemma triple_incr' : ∀ (p:loc) (n:int),
  triple (incr p)
    (p ~~> n)
    (fun _ => (p ~~> (n+1))).
```

Proof.

```
xwp. xapp. xapp. xapp. xsimpl.
```

Qed.

The existence of implicit coercions might be a little confusing at times, but coercions make specifications so much more readable that it would be a pity to not exploit them.

The reader may be curious to know what the notation *PRE H CODE F POST Q* stands for, and what the x-tactics are doing. Everything will be explained as we move through the course. This chapter and the next avoid such explanations to focus on surveying the features of Separation Logic and showing how x-tactics can be used to verify programs.

37.2.4 A Function with a Return Value

As a second example, let us specify a function that performs simple arithmetic computations. The function, whose code appears below, expects an integer argument *n* (in *Z*). It evaluates *a* as *n*+1, then evaluates *b* as *n*-1, and finally returns the sum *a*+*b*. The function thus always returns 2**n*.

```
Definition example_let : val :=
  <{ fun 'n =>
    let 'a = 'n + 1 in
    let 'b = 'n - 1 in
    'a + 'b }>.
```

The specification takes the form *triple* (*example_let* *n*) *H* (*fun* *r* => *H'*), where *r*, of type *val*, denotes the output value.

The precondition *H* describes what we need to assume about the input state. For this function, we need not assume anything, hence we write *\[]* to denote the empty precondition. The program might have allocated data prior to the call to the function *example_let*, but this function will not interfere in any way with this previously allocated data.

The postcondition describes what the function produces. More precisely, the postcondition specifies both the output that the function returns and the data from memory that the function has allocated, accessed, or updated. The function `example_let` does not interact with the memory, thus the postcondition could be described using the empty predicate `\[]`.

Yet, if we write just `fun (r:val) => \[]` as postcondition, we would have said nothing about the output value `r` produced by a call `example_let`. Instead, we would like to specify that the result `r` is equal to `2*n`. To that end, we write the postcondition `fun r => \[r = 2*n]`. Here, we use the predicate format `\[P]`, which allows to embed “pure facts”, of type `Prop` in preconditions and postconditions.

The equality `r = 2*n` actually resolves to `r = val_int (2*n)`, where `val_int` is a coercion that translates the integer value `2*n` into the corresponding integer value, of type `val`, from the programming language. If you do not know what a coercion is, just ignore the previous sentence.

```
Lemma triple_example_let : ∀ (n:int),
  triple (example_let n)
    \[]
    (fun r => \[r = 2×n]).
```

The proof script is quite similar to the previous one: `xwp` begins the proof, `xapp` performs symbolic execution. and `xsimpl` simplifies the entailment. Ultimately, we need to check that the expression computed, `(n + 1) + (n - 1)`, is equal to the specified result, that is, `2*n`. To prove this equality, we invoke the tactic `math` provided by the TLC library. Recall from the preface that this course leverages TLC for enhanced definitions and tactics. (Technically, `math` is a wrapper around the standard Rocq tactic `lia`; this wrapper is needed because TLC uses different definitions for arithmetic inequalities than Rocq’s standard library.)

Proof.

```
xwp. xapp. xapp. xapp. xsimpl. math.
```

Qed.

Exercise: 1 star, standard, especially useful (triple_quadruple) Consider the function `quadruple`, which expects an integer `n` and returns its quadruple, that is, the value `4*n`.

Definition `quadruple` : `val` :=

```
<{ fun 'n =>
  let 'm = 'n + 'n in
  'm + 'm }>.
```

Specify and verify the function `quadruple` to express that it returns `4*n`. Follow the pattern of the previous proof.

□

Exercise: 2 stars, standard, especially useful (triple_inplace_double) Consider the function `inplace_double`, which expects a reference to an integer, reads its contents, then updates the contents with the double of the original value.

Definition `inplace_double` : `val` :=

```
<{ fun 'p =>
  let 'n = !'p in
  let 'm = 'n + 'n in
  'p := 'm }>.
```

Specify and verify the function `inplace_double`, following the pattern of the first example, `triple_incr`.

□

From here on, we use the command `Proof using` for introducing a proof instead of writing just `Proof`. Writing `Proof using.` tells Rocq that the proof of the lemma does not depend on section variables others than the ones involved for typechecking the statement of the lemma. The `Proof using.` enables Rocq to compile proofs in parallel when the `-vos` flag is passed. For more details, see the “Need for Proof using” section from: <https://coq.inria.fr/refman/practical-tools/coq-commands.html>

37.3 Separation Logic Operators

37.3.1 Increment of Two References

Consider the following function, which expects the addresses of two reference cells and increments both of them.

Definition `incr_two` : `val` :=

```
<{ fun 'p 'q =>
  incr 'p;
  incr 'q }>.
```

The specification of this function takes the form `triple (incr_two p q) H (fun _ => H')`, where again the underscore symbol denotes the unit result value.

The precondition describes two references cells: $p \rightsquigarrow n$ and $q \rightsquigarrow m$. To assert that the two cells are distinct from each other, we separate their description with the operator $\backslash*$. Thus, the precondition is $(p \rightsquigarrow n) \backslash* (q \rightsquigarrow m)$, or simply $p \rightsquigarrow n \backslash* q \rightsquigarrow m$. The operator $\backslash*$ is called the “separating conjunction” of Separation Logic. It is also known as the “star” operator.

The postcondition describes the final state in a similar way, as $p \rightsquigarrow (n+1) \backslash* q \rightsquigarrow (m+1)$. This predicate reflects the fact that both references have their contents increased by one unit.

The specification triple for `incr_two` is thus as follows.

```
Lemma triple_incr_two :  $\forall (p\ q:\text{loc}) (n\ m:\text{int}),$ 
  triple (incr_two p q)
    ( $p \rightsquigarrow n \backslash* q \rightsquigarrow m$ )
    (fun _ =>  $p \rightsquigarrow (n+1) \backslash* q \rightsquigarrow (m+1)$ ).
```

Proof using.

xwp. xapp. xapp. xsimpl.

Qed.

We will make use of the function `incr_two` later in this chapter, so we register the specification `triple_incr_two` in the `triple` database.

#[global] Hint Resolve `triple_incr_two` : `triple`.

Separation Logic expressions such as $p \rightsquigarrow n$ or $\llbracket \rrbracket$ or $H1 \setminus^* H2$ are called “heap predicates”, because they corresponding to predicates over “heaps”, i.e., over memory states.

37.3.2 Aliased Arguments

The specification `triple_incr_two` describes the behavior of calls to the function `incr_two` *only* when the two arguments provided correspond to distinct reference cells. It says nothing at all about a call of the form `incr_two p p`. Indeed, in Separation Logic, a state described by $p \rightsquigarrow n$ cannot be matched against a state described by $p \rightsquigarrow n \setminus^* p \rightsquigarrow n$, because the star operator requires its operand to correspond to disjoint pieces of state.

What happens if we nevertheless try to exploit `triple_incr_two` to reason about a call of the form `incr_two p p`, that is, with aliased arguments? Let’s find out, by considering the operation `aliased_call p`, which does execute such a call.

Definition `aliased_call` : `val` :=

`<{ fun 'p =>`
`incr_two 'p 'p }>.`

A call to `aliased_call p` increases the contents of `p` by 2. This property can be specified as follows.

Lemma `triple_aliased_call` : $\forall (p:\text{loc}) (n:\text{int}),$
`triple (aliased_call p)`
 $(p \rightsquigarrow n)$
 $(\text{fun } _ \Rightarrow p \rightsquigarrow (n+2)).$

If we attempt the proof, we get stuck. The tactic *xapp* reports its failure by issuing a proof obligation of the form $\llbracket \rrbracket \Rightarrow (p \rightsquigarrow ?m) \setminus^* \_$. This proof obligation requires us to show that, from the empty heap predicate state, one can extract a heap predicate $p \rightsquigarrow ?m$ describing a reference at location `p` with some integer contents `?m`.

Proof using.

xwp. xapp.

Abort.

On the one hand, the precondition of the specification `triple_incr_two`, with $q = p$, requires providing $p \rightsquigarrow ?n \setminus^* p \rightsquigarrow ?m$. On the other hand, the current state is described as $p \rightsquigarrow n$. When trying to match the two, the internal simplification tactic *xsimpl* is able to cancel out one occurrence of $p \rightsquigarrow n$ from both expressions, but then there remains to match

the empty heap predicate $\llbracket \rrbracket$ against $(p \rightsquigarrow ?m)$. The issue here is that the specification `triple_incr_two` is specialized for the case of “non-aliased” references.

One thing we can do is to state and prove an alternative specification for the function `incr_two` to cover the case of aliased arguments. The precondition of this alternative specification mentions a single reference, $p \rightsquigarrow n$. Its postcondition asserts that the contents of that reference is increased by two units. This alternative specification is stated and proved as follows.

Lemma `triple_incr_two_aliased` : $\forall (p:\text{loc}) (n:\text{int}),$

`triple (incr_two p p)`
 $(p \rightsquigarrow n)$
 $(\text{fun } - \Rightarrow p \rightsquigarrow (n+2)).$

Proof using.

`xwp. xapp. xapp. xsimpl. math.`

Qed.

By exploiting the alternative specification for `incr_two`, we are able to prove the specification of the function `aliased_call`. In order to indicate to the tactic `xapp` that it should not invoke the lemma `triple_incr_two` registered for `incr_two`, but instead invoke the lemma `triple_incr_two_aliased`, we provide that lemma as an explicit argument to `xapp`, writing `xapp triple_incr_two_aliased`.

Lemma `triple_aliased_call` : $\forall (p:\text{loc}) (n:\text{int}),$

`triple (aliased_call p)`
 $(p \rightsquigarrow n)$
 $(\text{fun } - \Rightarrow p \rightsquigarrow (n+2)).$

Proof using.

`xwp. xapp triple_incr_two_aliased. xsimpl.`

Qed.

Taking a step back, it may be somewhat disappointing that we need two different specifications for the same function, depending on whether its arguments are aliased or not. There are advanced features of Separation Logic that do allow handling the two cases through a single specification. However, for such a simple function, it is easiest to just state and prove the two specifications separately.

37.3.3 A Function that Takes Two References and Increments One

Consider the following function, which expects the addresses of two reference cells and increments only the first one. What is interesting about this function is precisely the fact that it does nothing with its second argument.

Definition `incr_first` : `val` :=

`<{ fun 'p 'q  $\Rightarrow$`
`incr 'p }>.`

We can specify this function by describing its input state as $p \rightsquigarrow n \mid * q \rightsquigarrow m$ and describing its output state as $p \rightsquigarrow (n+1) \mid * q \rightsquigarrow m$. Formally:

```
Lemma triple_incr_first : ∀ (p q:loc) (n m:int),
  triple (incr_first p q)
    (p ~~~> n \* q ~~~> m)
    (fun _ => p ~~~> (n+1) \* q ~~~> m).
```

Proof using.

xwp. xapp. xsimpl.

Qed.

The second reference plays absolutely no role in the execution of the function. Thus, we could equally well consider a specification that mentions only the first reference.

```
Lemma triple_incr_first' : ∀ (p q:loc) (n:int),
  triple (incr_first p q)
    (p ~~~> n)
    (fun _ => p ~~~> (n+1)).
```

Proof using.

xwp. xapp. xsimpl.

Qed.

Interestingly, the specification `triple_incr_first`, which mentions the two references, is derivable from the specification `triple_incr_first'`, which mentions only the first. To prove the implication, it suffices to invoke the tactic *xapp* with argument `triple_incr_first'`.

```
Lemma triple_incr_first_derived : ∀ (p q:loc) (n m:int),
  triple (incr_first p q)
    (p ~~~> n \* q ~~~> m)
    (fun _ => p ~~~> (n+1) \* q ~~~> m).
```

Proof using.

intros. xapp triple_incr_first'. xsimpl.

Qed.

More generally, in Separation Logic, if a specification triple holds, then this triple remains valid when we add the same heap predicate to both the precondition and the postcondition. This is the “frame” principle, a key modularity feature that we’ll come back to later on in the course.

37.3.4 Transfer from one Reference to Another

Consider the `transfer` function, whose code appears below. Recall that, to simplify the implementation of the framework used in the course, we need to write in A-normal form, assigning a name to every intermediate result.

```
Definition transfer : val :=
  <{ fun 'p 'q =>
    let 'n = !'p in
```

```

let 'm = !'q in
let 's = 'n + 'm in
'p := 's;
'q := 0 }>.

```

Exercise: 1 star, standard, especially useful (triple_transfer) State and prove a lemma called *triple_transfer*, specifying the behavior of `transfer p q` in the case where `p` and `q` denote two distinct references.

□

Exercise: 1 star, standard, especially useful (triple_transfer_aliased) State and prove a lemma called *triple_transfer_aliased* specifying the behavior of `transfer` when it is applied twice to the same argument. It should take the form `triple (transfer p p) - ..`.

□

37.3.5 Specification of Allocation

Consider the operation `ref v`, which allocates a memory cell with contents `v`. How can we specify this operation using a triple? The precondition of this triple should be the empty heap predicate, written $\backslash []$, because the allocation can execute in an empty state. The postcondition should assert that the output value is a pointer `p`, such that the final state is described by $p \sim\sim v$.

It would be tempting to write the postcondition `fun p  $\Rightarrow$  p  $\sim\sim v$` . Yet, the triple would be ill-typed, because the postcondition of a triple must be a predicate over values, of type `val` in the framework, whereas here `p` is an address, of type `loc`. We thus need to write the postcondition in the form `fun (r:val)  $\Rightarrow H$` , where `r` denotes the result value, and somehow assert that `r` is the value that corresponds to the location `p`. This value is written `val_loc p`, where `val_loc` denotes the constructor that injects locations into the grammar of program values.

To formally quantify the variable `p`, we use the existential quantifier for heap predicates, written $\backslash \exists$. The correct postcondition for `ref v` is thus `fun r  $\Rightarrow \backslash \exists (p:loc), \backslash [r = val\_loc p] \backslash * (p \sim\sim v)$` . The complete statement of the specification of `ref` appears below. It is introduced as a `Parameter` instead of a `Lemma`, because the proof of the specification of this primitive operation is postponed to the chapter `Rules`.

```

Parameter triple_ref :  $\forall (v:val),$ 
  triple <{ ref v }>
     $\backslash []$ 
    (fun r  $\Rightarrow \backslash \exists p, \backslash [r = val\_loc p] \backslash * p \sim\sim v$ ).

```

The pattern `fun r  $\Rightarrow \backslash \exists p, \backslash [r = val\_loc p] \backslash * H$` occurs whenever a function returns a pointer. To improve concision for this frequent pattern, we introduce a specific notation: *funloc* `p  $\Rightarrow H$` .

Notation "'funloc' p '=>' H" :=
 (fun (r:val) => \exists p, \[r = val_loc p] * H)
 (at level 200, p name, format "'funloc' p '=>' H").

Using this notation, the specification *triple_ref* can be reformulated more concisely:

Parameter *triple_ref*' : $\forall (v:\text{val})$,
 triple <{ ref v }>
 \[]
 (funloc p => p ~~> v).

The CFML tool, which leverages techniques similar to those described in this course, leverages type-classes to generalize the notation *funloc* to all return types. Here, in order to avoid technical difficulties associated with type-classes, we will not go for the general presentation, but instead exploit the *funloc* notation, which is specific to the case where the return type is a location. For other types, we can quantify over the result value explicitly.

37.3.6 Allocation of a Reference with Greater Contents

Consider the function *ref_greater*, which takes as argument the address *p* of a memory cell with contents *n*, allocates a fresh memory cell with contents *n*+1, and returns the address of that fresh cell.

Definition *ref_greater* : val :=
 <{ fun 'p =>
 let 'n = !'p in
 let 'm = 'n + 1 in
 ref 'm }>.

The precondition of *ref_greater* asserts the existence of a cell *p* ~~> *n*. The postcondition asserts the existence of two cells, *p* ~~> *n* and *q* ~~> (*n*+1), where *q* denotes the location returned by the function. The postcondition is thus written *funloc* *q* => *p* ~~> *n* * *q* ~~> (*n*+1), which is a shorthand for fun (r:val) => \exists q, \[r = val_loc q] * *p* ~~> *n* * *q* ~~> (*n*+1). In the proof below, observe that the operation *ref* is displayed as *val_ref*, because this is the name of the operation in the internal abstrat syntax tree.

Lemma *triple_ref_greater* : $\forall (p:\text{loc}) (n:\text{int})$,
 triple (ref_greater p)
 (p ~~> n)
 (funloc q => p ~~> n * q ~~> (n+1)).

Proof using.

xwp. xapp. xapp. xapp. intros q. xsimpl. auto.

Qed.

Exercise: 2 stars, standard, especially useful (triple_ref_greater_abstract) State another specification for the function *ref_greater* with a postcondition that does not reveal the contents of the fresh reference *q*, but instead only asserts that it is greater than the contents

of p . To that end, introduce in the postcondition an existentially quantified variable called m , with $m > n$. This new specification, to be called *triple_ref_greater_abstract*, should be derived from *triple_ref_greater*, following the proof pattern employed in *triple_incr_first_derived*.

Hint: Remember that the notation $\backslash[P]$ injects a Rocq proposition into the language of Separation Logic predicates.

□

37.3.7 The Power of the Frame Rule with Respect to Allocation

Recall the specification *triple_ref'*, which describes the behavior of an allocation of a memory cell with contents v , performed at location p .

Parameter *triple_ref'* : forall (v:val), triple <{ ref v }> $\backslash\Box$ (funloc p => p $\sim\sim$ > v).

According to the frame rule presented in the preface, the above specification would remain valid if we extend the precondition and the postcondition with a heap predicate. Consider, for example, extending the precondition with the predicate $p' \sim\sim > v'$. By doing so, we derive a lemma describing the behavior of the allocation of a cell at location p , starting from a state that already contains a cell allocated at location p' .

Parameter *triple_ref_with_frame* : $\forall (p':loc) (v':val) (v:val)$,
triple <{ ref v }>
 $(p' \sim\sim > v')$
(funloc p => p $\sim\sim$ > v $\ast$ $p' \sim\sim > v'$).

The separating conjunction that appears in the above postcondition captures the fact that the location p is distinct from p' . As illustrated here, the frame rule indirectly captures the property that any piece of freshly allocated data is distinct from any piece of previously existing data. This property may seem obvious, but in the work on program verification prior to Separation Logic it was challenging to capture.

37.3.8 Deallocation in Separation Logic

Separation Logic, in its simplest form, enforces that every piece of allocated data is eventually deallocated. But OCaml is a programming language equipped with a garbage collector: programs do not contain explicit deallocation operations.

Thus, concretely, if we consider an OCaml program that allocates a reference, and if this reference is not described in the postcondition, we get stuck in the proof. Let us see how we get stuck and what we can do about it.

To begin with, consider the function shown below. This function computes the successor of an integer n . It does so using a reference: it first stores n into a reference cell, then it increments that reference, and finally it returns the new contents of the reference.

Definition *succ_using_incr_attempt* :=
<{ fun 'n =>
let 'p = ref 'n in

```

incr 'p;
! 'p }>.

```

A call to that function can be specified using an empty precondition and a postcondition asserting that the final result is equal to $n+1$. The postcondition has no reason to mention the reference used internally by the function. But we get stuck on the last step when trying to prove this specification.

```

Lemma triple_succ_using_incr_attempt :  $\forall (n:\text{int}),$ 
  triple (succ_using_incr_attempt  $n$ )
     $\square$ 
    (fun  $r \Rightarrow \square[r = n+1]$ ).

```

Proof using.

```

xwp. xapp. intros p. xapp. xapp. xsimpl. { auto. }
Abort.

```

We get stuck with the unprovable entailment $p \sim\sim > (n+1) ==> \square$, where the left-hand side describes a state with one reference, whereas the right-hand side describes an empty state. There are three possibilities to work around the issue.

The first possibility consists of extending the postcondition to account for the existence of the reference p . This yields a provable specification.

```

Lemma triple_succ_using_incr_attempt' :  $\forall (n:\text{int}),$ 
  triple (succ_using_incr_attempt  $n$ )
     $\square$ 
    (fun  $r \Rightarrow \square[r = n+1] \wedge \exists p, (p \sim\sim > (n+1))$ ).

```

Proof using.

```

xwp. xapp. intros p. xapp. xapp. xsimpl. { auto. }
Qed.

```

However, while the specification above is provable, it is pretty unsatisfying. The postcondition $\square \exists p, p \sim\sim > (n+1)$ is of absolutely no use to the caller of the function. Worse, the caller will get its own heap predicate polluted with $\square \exists p, p \sim\sim > (n+1)$, with no way of throwing away this predicate.

A second solution is to alter the code to include an explicit free operation, written *free* p , for deallocating the reference. This operation does not exist in OCaml, but let us nevertheless assume it to be able to demonstrate how Separation Logic supports reasoning about explicit deallocation.

Definition succ_using_incr :=

```

<{ fun 'n  $\Rightarrow$ 
  let 'p = ref 'n in
  incr 'p;
  let 'x = !'p in
  free 'p;
  'x }>.

```

This program may be proved correct with respect to the intended postcondition `fun r ⇒ \[r = n+1]`, without the need to mention `p`. In the proof below, the key step is the last call to `xapp`. This call is for reasoning about the operation `free p`, which consumes the heap predicate `p ~> _`. At the last proof step, we invoke the tactic `xval` for reasoning about the return value. When applied to a piece of code that consists of a value `v`, in a precondition `H` and a postcondition `Q`, `xval` transforms the goal into `H ==> Q v`, which asserts that the current state matches the state described by the postcondition, for the return value at hand.

```
Lemma triple_succ_using_incr : ∀ n,
  triple (succ_using_incr n)
    \[]
    (fun r ⇒ \[r = n+1]).
```

Proof using.

```
xwp. xapp. intros p. xapp. xapp.
xapp. xval. xsimpl. auto.
```

Qed.

The third solution for handling garbage collection involves a generalized version of Separation Logic in which specific classes of heap predicates may be freely discarded from the current state, at any point during a proof. This variant is described in the chapter [Affine](#). For the moment, we will keep assuming a language equipped with `free`.

37.3.9 Combined Reading and Freeing of a Reference

The function `get_and_free` takes as argument the address `p` of a reference cell. It reads the contents of that cell, frees the cell, and returns its contents.

```
Definition get_and_free : val :=
  <{ fun 'p ⇒
    let 'v = ! 'p in
    free 'p;
    'v }>.
```

Exercise: 2 stars, standard, especially useful (triple_get_and_free) Prove the correctness of the function `get_and_free`.

```
Lemma triple_get_and_free : ∀ p v,
  triple (get_and_free p)
    (p ~> v)
    (fun r ⇒ \[r = v]).
```

Proof using. *Admitted*.

□

```
#[global] Hint Resolve triple_get_and_free : triple.
```

37.3.10 Nondeterminism: Specifying Random Output Values

Given a positive integer n , the primitive operation `val_rand n` returns an integer in the range 0 inclusive to n exclusive. This operation may be specified by the following triple, which asserts that the output value r is an integer m satisfying $0 \leq m < n$.

Parameter *triple_rand* : $\forall n,$
 $n > 0 \rightarrow$
`triple (val_rand n)`
 $\sqcap$
 $(\text{fun } r \Rightarrow \exists m, [r = \text{val\_int } m] \wedge [0 \leq m < n]).$

Consider the function `two_dice`, which simulates the throw of two dice and returns their sum.

Definition `two_dice` : `val` :=
`<{ fun 'u =>`
`let 'n1 = val_rand 6 in`
`let 'n2 = val_rand 6 in`
`let 's = 'n1 + 'n2 in`
`'s + 2 }>.`

Exercise: 2 stars, standard, optional (`triple_two_dice`) Prove the correctness of the function `two_dice`. Hint: you'll need to use `xapp triple_rand`, because `xapp` is not able to discharge the side-condition $n > 0$ automatically. Use `math` for handling arithmetic proof obligations.

Lemma `triple_two_dice` :
`triple <{ two_dice () }>`
 $\sqcap$
 $(\text{fun } r \Rightarrow \exists n, [r = \text{val\_int } n] \wedge [2 \leq n \leq 12]).$

Proof using. *Admitted*.

□

37.4 Recursive Functions

37.4.1 Axiomatization of the Mathematical Factorial Function

Our next example consists of a program that evaluates the factorial function. To specify this function, we consider a Rocq axiomatization of the mathematical factorial function, named *facto*. We wrap the axiomatization inside a module, so that we can later refer to it from other files.

Module `FACTO`.

Parameter *facto* : `int` $\rightarrow$ `int`.

The factorial of 0 and 1 is equal to 1, and the factorial of n for $n > 1$ is equal to $n \times \text{facto } (n-1)$. Note that we purposely leave unspecified the value of *facto* on negative arguments.

Parameter *facto_init* : $\forall n,$

$0 \leq n \leq 1 \rightarrow$

facto $n = 1$.

Parameter *facto_step* : $\forall n,$

$n \geq 1 \rightarrow$

facto $n = n \times (\text{facto } (n-1))$.

Sometimes it is more convenient to simplify the value of an expression of the form *facto* $(n+1)$, as captured by the following lemma.

Lemma *facto_succ* : $\forall i,$

$i \geq 0 \rightarrow$

facto $(i + 1) = (i + 1) \times \text{facto } i$.

Proof using.

intros Hi. rewrite (@facto_step (i+1)). f_equal. f_equal. math. math.

Qed.

End FACTO.

Import FACTO.

37.4.2 A Partial Recursive Function, Without State

As a warm-up, we first consider a recursive function that does not involve any mutable state. The program function *factorec* computes the factorial of its argument: it implements the logical function *facto*.

OCaml:

```
let rec factorec n = if n <= 1 then 1 else n * factorec (n-1)
```

The corresponding code in A-normal form is slightly more verbose.

Definition *factorec* : **val** :=

```
<{ fix 'f 'n =>
  let 'b = 'n <= 1 in
  if 'b
  then 1
  else let 'x = 'n - 1 in
        let 'y = 'f 'x in
        'n × 'y }>.
```

A call to *factorec* n can be specified as follows:

- the initial state is empty,
- the final state is empty,

- the result value r is such that $r = \text{facto } n$, when $n \geq 0$.

In case $n < 0$, we have two choices:

- either we explicitly specify that the result is 1 in this case,
- or we rule out this possibility by requiring $n \geq 0$.

Let us follow the second approach, in order to illustrate the specification of partial functions.

There are two possibilities for expressing the constraint $n \geq 0$:

- either we use as precondition $\backslash[n \geq 0]$,
- or we use the empty precondition, that is, $\backslash[]$, and we place an assumption $(n \geq 0) \rightarrow \_$ at the front of the triple.

The two presentations are formally equivalent, but we prefer the second, which tends to improve both the readability of specifications and the conciseness of proof scripts. In that style, the specification of `factorec` is stated as follows.

Lemma triple_factorec : $\forall n,$
 $n \geq 0 \rightarrow$
 $\text{triple } (\text{factorec } n)$
 $\backslash []$
 $(\text{fun } r \Rightarrow \backslash[r = \text{facto } n]).$

In general, we prove specifications for recursive functions by exploiting a strong induction principle statement (“well-founded induction”) that allows us to assume, while we try to prove the specification, that the specification already holds for any “smaller input”. The (well-founded) order relation that defines whether an input is smaller than another one is specified by the user. In the present example of `factorec`, we use the well-founded relation `downto 0`, where `downto 0 m n` asserts that $0 \leq m < n$.

Let’s walk through the proof script in detail, to see how to set up the induction, how we exploit it for reasoning about the recursive call, and how we justify that the recursive call is made on a smaller input.

Proof using.

We set up a proof by induction on n to obtain an induction hypothesis for the recursive calls. The tactic `induction_wf`, provided by the TLC library, helps setting up well-founded inductions. Its usage is `induction_wf IH: R x`, where R denotes a well-founded relation, x is the name of a variable, and IH is the name assigned to the induction hypothesis. `intros n. induction_wf IH: (downto 0) n.`

Observe the induction hypothesis IH . By unfolding `downto` as in the next step, we can see that this hypothesis asserts that, given the current argument n , the specification of `factorec` can be exploited for any m such that $0 \leq m < n$. `unfold downto in IH. intros Hn. xwp.`

We reason about the evaluation of the boolean condition $n \leq 1$. *xapp.*

The result of the evaluation of $n \leq 1$ in the source program is described by the boolean value `isTrue` ($n \leq 1$), which appears in the *CODE* section after *Ifval*. The operation `isTrue` is provided by the TLC library as a conversion function from *Prop* to *bool*. The use of such a conversion function (which leverages classical logic) greatly simplifies the process of automatically performing substitutions after calls to *xapp*.

We next perform the case analysis on the test $n \leq 1$. *xif.*

Doing so gives two cases.

In the “then” branch, we can assume $n \leq 1$. *{ intros C.*

Here, the return value is 1. *xval. xsimpl.*

We check that $1 = \text{facto } n$ when $n \leq 1$. *rewrite facto_init. math. math. }*

In the “else” branch, we can assume $n > 1$. *{ intros C.*

We reason about the evaluation of $n-1$ *xapp.*

We reason about the recursive call, implicitly exploiting the induction hypothesis *IH* with $n-1$. *xapp.*

We justify that the recursive call is indeed made on a smaller argument than the current one, that is, a nonnegative integer smaller than n . *{ math. }*

We justify that the recursive call is also made on a nonnegative argument, as required by the specification. *{ math. }*

We reason about the multiplication $n \times \text{facto}(n-1)$. *xapp.*

We check that $n \times \text{facto } (n-1)$ matches $\text{facto } n$. *xsimpl. rewrite (@facto_step n). math. math. }*
Qed.

This completes our first proof of a recursive function. Further on, we will investigate a proof of an imperative implementation of a factorial function.

37.4.3 A Recursive Function with State

Let’s now tackle a recursive function involving some mutable state. The function `repeat_incr p m` makes, m times, a call to `incr p`, where m is assumed to be nonnegative.

OCaml:

```
let rec repeat_incr p m = if m > 0 then ( incr p; repeat_incr p (m - 1) )
```

In the concrete syntax for programs, conditionals without an ‘else’ branch are written `if t1 then t2 end`. The keyword `end` avoids ambiguities in cases where this construct is followed by a semicolon.

Definition `repeat_incr` : *val* :=

```
<{ fix 'f 'p 'm =>
  let 'b = 'm > 0 in
  if 'b then
    incr 'p;
    let 'x = 'm - 1 in
    'f 'p 'x
```

end }>.

The specification for `repeat_incr p` requires that the initial state contains a reference `p` with some integer contents `n`, that is, $p \sim\sim > n$. Its postcondition asserts that the resulting state is $p \sim\sim > (n+m)$, which is the result after incrementing, `m` times, the reference `p`. Observe that this postcondition is only valid under the assumption that $m \geq 0$.

Lemma `triple_repeat_incr` : $\forall (m\ n:\text{int}) (p:\text{loc}),$
 $m \geq 0 \rightarrow$
`triple (repeat_incr p m)`
 $(p \sim\sim > n)$
 $(\text{fun } - \Rightarrow p \sim\sim > (n + m)).$

Exercise: 2 stars, standard, especially useful (triple_repeat_incr) Prove the specification of the function `repeat_incr`, by following the template of the proof of `triple_factorec`.

Hint: begin the proof with `intros m. induction_wf IH`: ..., without introducing `n` or `p`, otherwise the induction principle obtained is too weak.

Proof using. *Admitted.*

□

In the previous examples of recursive functions, the induction was always performed on the first argument quantified in the specification. When the decreasing argument is not the first one, additional manipulations are required for re-generalizing into the goal the variables that may change during the course of the induction. Here is an example illustrating how to deal with such a situation.

Lemma `triple_repeat_incr'` : $\forall (p:\text{loc}) (n\ m:\text{int}),$
 $m \geq 0 \rightarrow$
`triple (repeat_incr p m)`
 $(p \sim\sim > n)$
 $(\text{fun } - \Rightarrow p \sim\sim > (n + m)).$

Proof using.

First, we introduce all variables and hypotheses. `intros p n m Hm.`

Next, we generalize variables and hypotheses that are not constant during the recursion, using the TLC tactic `gen`, which is similar to Rocq's tactics `revert` and `dependent generalize`. `gen n Hm.`

Then, we set up the induction. `induction_wf IH: (downto 0) m. unfold downto in IH.`

Finally, we re-introduce the generalized hypotheses. `intros.`

The rest of the proof is exactly the same as before. `Abort.`

37.4.4 Trying to Prove Incorrect Specifications

We established for `repeat_incr p m` a specification featuring the hypothesis $m \geq 0$, but what if we had omitted this hypothesis? At which step would we get stuck in the proof? What

feedback would we get at that point?

Certainly, we expect the proof to get stuck if $m < 0$. Indeed, in this case, the call to `repeat_incr p m` terminates immediately, so the final state is $p \rightsquigarrow n$, like the initial state, and the final state does not match the claimed postcondition $p \rightsquigarrow (n + m)$. Let us investigate how the proof of lemma `triple_repeat_incr` breaks.

```
Lemma triple_repeat_incr_incorrect : ∀ (p:loc) (n m:int),
  triple (repeat_incr p m)
    (p ~~~> n)
    (fun _ => p ~~~> (n + m)).
```

Proof using.

```
intros. gen n. induction_wf IH: (downto 0) m. unfold downto in IH.
intros. xwp. xapp. xif; intros C.
{
  xapp. xapp. xapp. { math. } xsimpl. math. }
{
  xval.
```

At this point, we are requested to justify that the current state $p \rightsquigarrow n$ matches the postcondition $p \rightsquigarrow (n + m)$, which amounts to proving $n = n + m$. *xsimpl*.

Abort.

When the specification includes the assumption $m \geq 0$, we can prove this equality because the fact that we are in the else branch means that $m \leq 0$, thus $m = 0$. However, without the assumption $m \geq 0$, the value of m could very well be negative. In that case, the equality $n = n + m$ is unprovable. As users, the proof obligation $(m \leq 0) \rightarrow (n = n + m)$ gives us a very strong hint that either the code or the specification is not handling the case $m < 0$ properly.

This concludes our example attempt at proving an incorrect specification.

There exists a valid specification for `repeat_incr` that does not constrain m but instead specifies that, regardless of the value of m , the state evolves from $p \rightsquigarrow n$ to $p \rightsquigarrow (n + \max 0 m)$. The corresponding proof script exploits two characteristic properties of the function `max`.

```
Lemma max_l : ∀ n m,
  n ≥ m →
  max n m = n.
```

Proof using. intros. unfold max. case-if; math. Qed.

```
Lemma max_r : ∀ n m,
  n ≤ m →
  max n m = m.
```

Proof using. intros. unfold max. case-if; math. Qed.

Here is the most general specification for the function `repeat_incr`.

Exercise: 2 stars, standard, optional (triple_repeat_incr') Prove the general specification for the function `repeat_incr`, covering also the case $m < 0$.

Lemma `triple_repeat_incr'` : $\forall (p:\text{loc}) (n\ m:\text{int}),$
 $\text{triple } (\text{repeat\_incr } p\ m)$
 $(p \sim\sim> n)$
 $(\text{fun } - \Rightarrow p \sim\sim> (n + \max\ 0\ m)).$

Proof using. *Admitted.*

□

37.4.5 A Recursive Function Involving two References

Consider the function `step_transfer` `p q`, which repeatedly increments a reference `p` and decrements a reference `q`, as long as the contents of `q` is positive.

OCaml:

```
let rec step_transfer p q = if !q > 0 then ( incr p; decr q; step_transfer p q )
```

Definition `step_transfer` :=

```
<{ fix 'f 'p 'q =>
  let 'm = !'q in
  let 'b = 'm > 0 in
  if 'b then
    incr 'p;
    decr 'q;
    'f 'p 'q
  end }>.
```

The specification of `step_transfer` is essentially the same as that of the function `transfer` presented previously, the only difference being that we now assume the contents of `q` to be nonnegative.

Lemma `triple_step_transfer` : $\forall p\ q\ n\ m,$
 $m \geq 0 \rightarrow$
 $\text{triple } (\text{step\_transfer } p\ q)$
 $(p \sim\sim> n \ \& \ q \sim\sim> m)$
 $(\text{fun } - \Rightarrow p \sim\sim> (n + m) \ \& \ q \sim\sim> 0).$

Exercise: 2 stars, standard, especially useful (triple_step_transfer) Verify the function `step_transfer`. Hint: to set up the induction, follow the pattern from `triple_repeat_incr'`.

Proof using. *Admitted.*

□

37.4.6 Computing Factorial using State

To conclude this chapter, consider a function that computes factorial using mutable state.

A standard OCaml implementation would be:

OCaml:

```
let factoimp n = let r = ref 1 in for i = 1 to n do r := !r * (i+1); done; !r
```

Rather than explaining how to handle for loops at this stage, let us instead encode the loop using an auxiliary recursive function.

The auxiliary function `factoimp_aux r i n` performs the operations of the above for-loop starting at loop index `i`, until reaching `n`. The main function `factoimp n` computes the factorial of `n` in a fresh reference cell named `r`. Then, it returns the contents of `r`.

OCaml:

```
let rec factoimp_aux r i n = if i < n then ( r := !r * (i+1); factoimp_aux r (i+1) n )
let factoimp n = let r = ref 1 in factoimp_aux r 1 n; !r
```

Definition `factoimp_aux` : `val` :=

```
<{ fix 'f 'r 'i 'n =>
  let 'b = 'i < 'n in
  if 'b then
    let 'j = 'i + 1 in
    let 's = !'r in
    let 'p = 's × 'j in
    'r := 'p;
    'f 'r 'j 'n
  end }>.
```

Definition `factoimp` : `val` :=

```
<{ fun 'n =>
  let 'r = ref 1 in
  factoimp_aux 'r 1 'n;
  let 'm = !'r in
  free 'r;
  'm }>.
```

Exercise: 3 stars, standard, especially useful (`triple_factoimp_aux`) Verify the function `triple_factoimp_aux`. Hint: the set up of the induction is provided. Use `facto_succ` in the proof.

Lemma `triple_factoimp_aux` : $\forall (r:\text{loc}) (i\ n:\text{int}),$

```
0 ≤ i ≤ n →
triple (factoimp_aux r i n)
  (r ~> facto i)
  (fun _ => r ~> facto n).
```

Proof using.

```
introv. induction_wf IH: (upto n) i. introv Hn.
```

Admitted.

□

Exercise: 3 stars, standard, especially useful (triple_factoimp) Now verify the function `factoimp`. Hint: use *xapp* `triple_factoimp_aux` to reason about the call to the auxiliary function `factoimp_aux`. Use *facto_init*. For simplicity, we assume $n \geq 1$ instead of $n \geq 0$. In the optional material section further on, we explain how to generalize the proof to handle the case $n = 0$

Lemma `triple_factoimp` : $\forall n,$
 $n \geq 1 \rightarrow$
 triple (factoimp n)
 \ []
 (fun $r \Rightarrow \backslash[r = \text{facto } n]$).

Proof using. *Admitted.*

□

37.5 Summary

This chapter introduced the following notions:

- “Heap predicates”, which are used to describe memory states in Separation Logic.
- “Specification triples” of the form `triple t H Q`, which relate a term t , a precondition H , and a postcondition Q .
- “Verification triples”, of the form `PRE H CODE F POST Q`, are triples of a specific form, produced by the framework.
- “Entailments”, of the form $H ==> H'$ or $Q ==> Q'$, which assert that one pre- or post-condition is weaker than another one.
- Custom proof tactics, called “x-tactics”, which are specialized tactics for discharging these proof obligations.

Several specific heap predicates for describing memory states were presented in this introductory chapter. They include:

- $p \sim\sim> n$, which describes a memory cell at location p with contents n ,
- $\backslash[]$, which describes an empty state,
- $\backslash[P]$, which also describes an empty state and moreover asserts that the proposition P is true,

- $H1 \setminus^* H2$, which describes a state made of two disjoint parts, one satisfying $H1$ and another satisfying $H2$,
- $\setminus \exists x, H$, which is used to quantify variables in postconditions.

All these heap predicates have type `hprop`, which describes predicates over memory states. A memory state has type `heap`, thus `hprop` is defined as `heap → Prop`.

The verification of practical programs is carried out using x-tactics, identified by the leading “x” letter in their name. These tactics include:

- `xwp` to begin a proof,
- `xapp` to reason about an application,
- `xval` to reason about a return value,
- `xif` to reason about a conditional,
- `xsimpl` to simplify or prove entailments ($H ==> H'$ and $Q ==> Q'$).

In addition to x-tactics, the proof scripts exploit standard Rocq tactics, as well as tactics from the TLC library, which provides a bunch of useful, general purpose tactics. In this chapter, we used a few TLC tactics:

- `math`, which is a variant of `lia` for proving mathematical goals,
- `induction_wf`, which sets up proofs by well-founded induction,
- `gen`, for generalizing variables for inductions.

37.6 Optional Material

The function `factoimp n` correctly computes the factorial of `n` not just when $n \geq 1$, but also when $n = 0$. Yet, the argument for justifying this fact is not entirely straightforward. A naive approach would be to develop a separate proof that `factoimp 0` returns 1. Such a separate proof would look as follows.

```
Lemma triple_factoimp_aux_zero :  $\forall (r:loc) (i n:int),$ 
   $i = 1 \rightarrow$ 
   $n = 0 \rightarrow$ 
  triple (factoimp_aux r i n)
    ( $r \sim\sim> facto i$ )
    ( $\text{fun } _ \Rightarrow r \sim\sim> facto n$ ).
```

Proof using.

```
introv. introv Hi Hn.
```

```

xwp. xapp. xif; intros C.
Case i ≥ n cannot happen { false. math. }
Case i < n holds because fact0 0 = fact0 1 { xval. xsimpl. subst.
  rewrite fact0_init; [|math|.
  rewrite fact0_init; [|math|. auto. }

```

Qed.

```

Lemma triple_factoimp_zero : ∀ n,
  n = 0 →
  triple (factoimp n)
    \[]
    (fun r ⇒ \[r = fact0 n]).

```

Proof using.

```

  introv Hn. xwp. xapp. intros r.
  Here we invoke the specification of factoimp_aux specialized for the case n = 0.
triple_factoimp_aux_zero. { math. } { math. }
{ rewrite× fact0_init. { math. } }
xapp. xapp. xval. xsimpl. auto.

```

Qed.

What we have done here is to go twice over the whole code, once to verify correctness in the case $n \geq 1$, and once for the case $n = 0$. Going twice over the same piece of code is, in general, bad practice. Indeed, the resulting script involves a fair amount of duplication. Fortunately, in most situations it is possible to verify code in a single pass. In the present example, it suffices to consider as precondition the disjunction of the two possible cases: either $i = 1 \wedge n = 0$ or $0 \leq i \leq n$.

Exercise: 4 stars, standard, especially useful (triple_factoimp_aux') Refine the proof of `triple_factoimp_aux` for a relaxed precondition accounting also for the case $n = 0$. Hint: use `destruct Hn as [(?&?)|?]` to perform the case analyses.

```

Lemma triple_factoimp_aux' : ∀ (r:loc) (i n:int),
  (i = 1 ∧ n = 0) ∨ (0 ≤ i ≤ n) →
  triple (factoimp_aux r i n)
    (r ~~~> fact0 i)
    (fun _ ⇒ r ~~~> fact0 n).

```

Proof using.

```

  introv. induction_wf IH: (upto n) i. introv Hn.
  Admitted.

```

Exercise: 4 stars, standard, especially useful (triple_factoimp') Refine the proof of `triple_factoimp` for a relaxed precondition accounting also for the case $n = 0$. Hint: use the tactic `destruct (classic (n = 0))` to perform a case analysis on whether n is zero; but make sure to perform the case analysis as late in the proof as possible, in order to avoid duplication.

```

Lemma triple_factoimp' :  $\forall n,$ 
   $n \geq 0 \rightarrow$ 
  triple (factoimp  $n$ )
    \ []
    (fun  $r \Rightarrow \backslash[r = \text{facto } n]$ ).
Proof using. Admitted.

```

37.7 Historical Notes

The key ideas of Separation Logic were devised by John Reynolds, inspired in part by older work by *BurSTALL* 1972 (in Bib.v). Reynolds presented his ideas in lectures given in the fall of 1999. The proposed rules turned out to be unsound, but *Ishtiaq and O'Hearn* 2001 (in Bib.v) noticed a strong relationship with the logic of bunched implications by *O'Hearn and Pym* 1999 (in Bib.v), leading to ideas on how to set up a sound program logic. Soon afterwards, the seminal publications on Separation Logic appeared at the CSL workshop *O'Hearn, Reynolds, and Yang* 2001 (in Bib.v) and at the LICS conference *Reynolds* 2002 (in Bib.v).

The Separation Logic specifications and proof scripts using x-tactics presented in this file are directly adapted from the CFML tool (2010-2023), developed mainly by Arthur Charguéraud. The notations for Separation Logic predicates are directly inspired from those introduced in the Ynot project *Chlipala et al* 2009 (in Bib.v). See chapter *Bib* for references.

Chapter 38

Library `SLF.Repr`

38.1 Repr: Representation Predicates

```
Set Implicit Arguments.
From SLF Require Import LibSepReference.
Import PROGRAMSYNTAX DEMOPROGRAMS.
From SLF Require Import Basic.
Open Scope liblist_scope.

Implicit Types n m : int.
Implicit Types p q s c : loc.
Implicit Types x : val.
```

38.2 First Pass

The previous chapter (*Basic*) covered simple programs that manipulate individual mutable cells. This chapter focuses on the specification and verification of programs manipulating mutable data structures like lists and trees. In Separation Logic, such data structures are specified using *representation predicates*.

A representation predicate is a heap predicate that describes a mutable data structure. For example, the heap predicate `MList L p` describes a mutable linked list whose head cell is at location `p` and whose elements are described by the Rocq list `L`. In this chapter, we will see how to define `MList` and how to use this predicate for specifying and verifying functions that operate on mutable linked lists. We will also study representation predicates for mutable trees, as well as for counter functions, which feature an internal state.

As explained in the *Preface*, this chapter, like all the following ones, is structured in three parts:

- The *First Pass* section presents the most important ideas only.

- The *More Details* section presents additional material explaining in more depth the meaning and the consequences of the key results. By default, readers would eventually read all this material.
- The *Optional Material* section contains more advanced material, for readers who can afford to invest more time in the topic.

While going through proofs, we will introduce a few additional tactics.

- *xpull* to extract pure facts and quantifiers from the LHS of $==>$.
- *xchange E* for exploiting a lemma *E* with a conclusion of the form $H1 ==> H2$ or $H1 = H2$.
- *xchange $\leftarrow E$* for exploiting an entailment $H2 ==> H1$ in the case *E* is a lemma with a conclusion of the form $H1 = H2$.
- *xchanges* is a shorthand for *xchange* followed with *xsimpl*.
- *xfun* to reason about function definitions.
- *xtriple* to establish specifications for abstract functions.
- *introv* (a TLC tactic) like *intros* but takes as arguments only the name of the hypotheses, not of all variables.
- *rew_list* (a TLC tactic) to normalize list expressions.

Furthermore, TLC tactics and most x-tactics may be followed with a star symbol to invoke *eauto*. For example, *xsimpl* $\times$ is a shorthand for *xsimpl*; *eauto*. The star symbol is used to make proof scripts more concise, yet exercises can all be solved without this feature.

38.2.1 The List Representation Predicate

The implementation of mutable lists and trees involves the use of records. For simplicity, field names are represented as natural numbers. For example, to represent a list cell with a head and a tail field, we define *head* as the constant zero, and *tail* as the constant one.

Definition *head* : field := 0%*nat*.

Definition *tail* : field := 1%*nat*.

The heap predicate $p \rightsquigarrow \{ \text{head} := x; \text{tail} := q \}$ describes a record allocated at location *p*, with a head field storing *x* and a tail field storing *q*. The arrow $\rightsquigarrow$ is provided by the framework for describing records. The notation $\{ \dots \}$ is a handy notation for a list of pairs of field names and values of arbitrary types. (Further details on the formalization of records are presented in chapter *Records*.)

A mutable list consists of a chain of cells. Each cell stores a tail pointer that gives the location of the next cell in the chain. The last cell stores the `null` value in its tail field.

The heap predicate `MList L p` describes a list whose head cell is at location `p` and whose elements are described by the list `L`. This predicate is defined recursively on the structure of `L`.

- If `L` is empty, then `p` is the null pointer.
- If `L` is of the form `x::L'`, then `p` is not null, the head field of `p` contains `x`, and the tail field of `p` contains a pointer `q` such that `MList L' q` describes the tail of the list.

This definition is formalized as follows.

```
Fixpoint MList (L:list val) (p:loc) : hprop :=
  match L with
  | nil => \[p = null]
  | x::L' => \exists q, (p ~~~> '{ head := x; tail := q}) \* (MList L' q)
  end.
```

38.2.2 Alternative Characterizations of MList

Carrying out proofs directly with `MList` can be slightly cumbersome, mainly due to Rocq's limited support for re-folding definitions. We find it more practical to explicitly state equalities that paraphrase the definition of `MList`. There is one equality for the `nil` case and one for the `cons` case.

```
Lemma MList_nil : ∀ p,
  (MList nil p) = \[p = null].
```

Proof using. auto. Qed.

```
Lemma MList_cons : ∀ p x L',
  MList (x::L') p =
  \exists q, (p ~~~> '{ head := x; tail := q}) \* (MList L' q).
```

Proof using. auto. Qed.

In addition, it is also very useful in proofs to reformulate the definition of `MList L p` in the form of a case analysis on whether the pointer `p` is null or not. This corresponds to the programming pattern `if p == null then ... else ....` This alternative characterization of `MList L p` asserts the following.

- If `p` is null, then `L` is empty.
- Otherwise, `L` decomposes as `x::L'`, the head field of `p` contains `x`, and the tail field of `p` contains a pointer `q` such that `MList L' q` describes the tail of the list.

The corresponding lemma, shown below, is stated using the `If P then X else Y` construction, which generalizes Rocq's construction `if b then X else Y` to discriminate over

a proposition P as opposed to a boolean value b . The *If* construct leverages classical logic; it is provided by the TLC library. The tactic *case_if* is convenient for performing the case analysis on whether P is true or false.

```
Lemma MList_if : ∀ (p:loc) (L:list val),
  (MList L p)
==> (If p = null
      then \[L = nil]
      else \∃ x q L', \[L = x::L']
      \* (p ~~~> '{ head := x; tail := q }') \* (MList L' q)).
```

The proof is a bit technical: it may be skipped on a first reading. *Proof using. Admitted.*

Note that the reciprocal entailment to the one stated in *MList_if* is also true, but we do not need it so we do not bother proving it here. In the rest of the course, we will never unfold the definition *MList*, but only work with *MList_nil*, *MList_cons*, and *MList_if*. We therefore make the definition of *MList* opaque to prevent Rocq from performing undesired simplifications.

Global Opaque MList.

38.2.3 In-place Concatenation of Two Mutable Lists

The function *append* expects two arguments: a pointer *p1* to a nonempty list, and a pointer *p2* to another list, possibly empty. The function updates the last cell from the first list in such a way that its tail points to the head cell of *p2*. After this operation, the pointer *p1* points to a list that corresponds to the concatenation of the two input lists.

OCaml:

```
let rec append p1 p2 = if p1.tail == null then p1.tail <- p2 else append p1.tail p2
```

Definition append : val :=

```
<{ fix 'f 'p1 'p2 =>
  let 'q1 = 'p1'.tail in
  let 'b = ('q1 = null) in
  if 'b
  then 'p1'.tail := 'p2
  else 'f 'q1 'p2 }>.
```

The *append* function is specified and verified as shown below. The proof pattern is representative of that of many list-manipulating functions, so it is essential that you take the time to play through every step of this proof. Several exercises will require a very similar proof pattern.

```
Lemma triple_append : ∀ (L1 L2:list val) (p1 p2:loc),
  p1 ≠ null →
  triple (append p1 p2)
  (MList L1 p1 \* MList L2 p2)
```

(fun _ $\Rightarrow$ MList ($L1++L2$) p1).

Proof using.

The TLC tactic *introv* is very convenient for providing explicit names for hypotheses without having to provide names for the variables already named in the lemma statement. *introv N*.

The induction principle provides a hypothesis for the tail of $L1$. The predicate **list_sub** $L1' L1$ asserts that $L1$ decomposes as $x::L1'$ for some x . *gen p1. induction_wf IH: list_sub L1. intros. xwp.*

To begin the proof, we reveal the head cell of $p1$, exploiting the assumption that $p1 \neq \text{null}$. *xchange (MList_if p1). case_if.*

Using *xpull*, we extract the existentially quantified variables: we let $q1$ denote the tail of $p1$, x the head element of $L1$, and $L1'$ the tail of $L1$. *xpull.*

The tactic *intros $\rightarrow$* is a shorthand for *intros E; rewrite E; clear E. intros x q1 L1' $\rightarrow$.*

We then reason about the if statement. *xapp. xapp. xif; intros Cq1.*

If $q1$ is null, then $L1'$ is empty. *{ xchange (MList_if q1). case_if. xpull.*

intros $\rightarrow$.

In this case, we reason about the assignment, then we fold back the head cell. To that end, we exploit the tactic *xchange $\leftarrow$ MList_cons*, which is similar to *xchange MList_cons* but exploits the equality asserted by *MList_cons* in the reverse direction. *xapp. xchange $\leftarrow$ MList_cons.*

Above, the tactic *xchange* discharges the goal because the LHS, after the operation, matches the RHS. To see the details, change the line of script to: *xapp. eapply himpl_trans. xchange $\leftarrow$ MList_cons. apply himpl_refl. }*

Otherwise, if $q1$ is not null, we reason about the recursive call using the induction hypothesis; then we re-fold the head cell. *{ xapp. xchange $\leftarrow$ MList_cons. }*

Qed.

38.2.4 Smart Constructors for Linked Lists

We next introduce two smart constructors for linked lists, called **mnil** and **mcons**. The operation **mnil**() creates an empty list. Its implementation simply returns the value **null**. Its specification asserts that the return value is a pointer p such that **MList nil** p . In the specification below, recall that *funloc* $p \Rightarrow H$ is a notation for *fun (r:val) $\Rightarrow \backslash \exists (p:\text{loc}), \backslash [r = \text{val\_loc } p] \backslash^* H$* .

Definition **mnil** : **val** :=

*<{ fun 'u $\Rightarrow$
null }>.*

Note: in theory, we could present **mnil** as a constant rather than as a function that takes unit as argument. However, viewing it as a function like all the other example programs helps keep the framework minimal.

Lemma **triple_mnil** :

```
triple (mnil ())
  \[]
  (funloc p ⇒ MList nil p).
```

Proof using.

xwp.

We are requested to prove that, under the empty precondition, the value `null` satisfies the postcondition `fun (r:val) ⇒ \∃ (p:loc), \[r = val_loc p] \* (MList nil p)`. As the tactic *xval* shows, this is equivalent to proving that the precondition `\[]` entails `\∃ p, \[null = val_loc p] \* (MList nil p)`. *xval*.

To conclude the proof, it suffices to instantiate `p` with `null`, and to argue that `MList nil null` can be created out of thin air. This is indeed true because `MList nil null` is equivalent to `\[]`. The first step is to obtain an explicit statement of this equivalence, in the form of an hypothesis named *E*. By instantiating the lemma `MList_nil`, which asserts that $\forall p, \text{MList nil } p = \backslash[p = \text{null}]$, we obtain the equality `MList nil null = \[null = null]`. *generalize (MList_nil null)*. *intros E*.

Then, we invoke *xchange* ← *E*. In exchange for proving `null = null`, this tactic replaces the empty heap predicate `\[]` with `MList nil null`. *xchange* ← *E*. { *auto*. }

At this stage, the tactic *xsimpl* is able to automatically instantiate `p` with `null`, and *auto* checks that `null = null`. *xsimpl*. *auto*.

Qed.

The proof above can be greatly shortened by using the tactic *xchanges*, which is a shorthand for *xchange* ← *E* followed by *xsimpl*. In fact, we even use *xchanges* × ← *E*, to invoke *xchanges* ← *E* followed by *eauto*.

Lemma `triple_mnil'` :

```
triple (mnil ())
  \[]
  (funloc p ⇒ MList nil p).
```

Proof using. *xwp*. *xval*. *xchanges* × ← (MList_nil null). *Qed*.

#[*global*] Hint Resolve *triple_mnil* : *triple*.

Observe that the specification `triple_mnil` does not mention the `null` pointer anywhere. Hence, this specification abstracts away from the implementation details of how mutable lists are represented internally.

The operation `mcons x q` creates a fresh list cell, with `x` in the head field and `q` in the tail. Its implementation allocates and initializes a fresh record made of two fields. The allocation operation leverages the allocation construct written `{ head := 'x; tail := 'q }` in the code. This construct is in fact a notation for a primitive operation called `val_new_hrecord_2`. The details of record constructions are explained in the chapter `Record`. There is no need to understand the details at this stage.

Definition `mcons` : `val` :=

```
<{ fun 'x 'q ⇒
  '{ head := 'x; tail := 'q } }>.
```

The operation `mcons` admits two specifications. The first one says that it produces a record described with a record heap predicate.

```
Lemma triple_mcons : ∀ x q,
  triple (mcons x q)
    \ []
    (funloc p ⇒ p ~~~> '{ head := x ; tail := q }).
```

To prove this lemma, we need to know that the record allocation operation is in fact a notation for a call to `val_new_hrecord_2`, hence we need to invoke a library lemma called `triple_new_hrecord_2` for reasoning about this call. (Don't worry: this proof pattern will not appear in exercises!)

```
Proof using. xwp. xapp triple_new_hrecord_2; auto. xsimpl×. Qed.
```

The second specification asserts that `mcons` can be used to extend a mutable list. It assumes that the argument `q` comes with a list representation of the form `Mlist q L`, and it specifies that the function `mcons` produces the representation predicate `Mlist p (x::L)`. This second specification is derivable from the first one, by folding the representation predicate `MList` using the tactic `xchange`.

```
Lemma triple_mcons' : ∀ L x q,
  triple (mcons x q)
    (MList L q)
    (funloc p ⇒ MList (x::L) p).
```

```
Proof using.
  intros. xapp triple_mcons.
  intros p. xchange ← MList_cons. xsimpl×.
Qed.
```

In practice, this second specification is more often useful than the first one, hence we register it in the database for `xapp`. It remains possible to invoke `xapp triple_mcons` to exploit the first specification, where needed.

```
#[global] Hint Resolve triple_mcons' : triple.
```

In what follows, we consider several classic operations on mutable lists, each illustrating an idiomatic proof pattern.

38.2.5 Copy Function for Lists

The function `mcopy` takes a mutable linked list and builds an independent copy of it, with the help of the functions `mnil` and `mcons`.

OCaml:

```
let rec mcopy p = if p == null then mnil () else mcons (p.head) (mcopy p.tail)
```

```
Definition mcopy : val :=
  <{ fix 'f 'p ⇒
    let 'b = ('p = null) in
```

```

    if 'b
    then mnil ()
    else
      let 'x = 'p'.head in
      let 'q = 'p'.tail in
      let 'q2 = ('f 'q) in
      mcons 'x 'q2 }>.

```

The precondition of `mcopy` requires a linked list described as `MList L p`. The postcondition asserts that the function returns a pointer `p'` and a list described as `MList L p'`, in addition to the original list `MList L p`. The two lists are totally disjoint and independent, as captured by the separating conjunction symbol (the star).

Lemma `triple_mcopy` : $\forall L p,$
`triple (mcopy p)`
`(MList L p)`
`(funloc p'  $\Rightarrow$  (MList L p)  $\backslash$ * (MList L p'))`.

The proof structure is like the previous ones. While playing the script, try to spot the places where - `mnil` produces an empty list of the form `MList nil p'`, - the recursive call produces a list of the form `MList L' q'`, and - `mcons` produces a list of the form `MList (x::L') p'`.

Proof using.

```

  intros. gen p. induction_wf IH: list_sub L.
  xwp. xapp. xchange MList_if. xif.
  { intros C.

```

Note that `C` is actually `val_loc p = val_loc null`, and not simply `p = null`. This explains why a call to `subst` does nothing. If needed, use the TLC tactic, `inverts C` or `invert C` to exploit the equality. In this specific proof, though, we don't need that. `case_if`.

```

xpull. intros E. subst L.
  xapp. xsimpl. { auto. }
  subst. xchange  $\leftarrow$  MList_nil. auto. }
{ intros C. case_if.      xpull. intros x q L' E. subst L.
  xapp. xapp. xapp. intros q'.
  xapp. intros p'.
  xchange  $\leftarrow$  MList_cons. xsimpl. auto. }

```

Qed.

A Rocq expert would typically write the same proof script in a more compact fashion, as follows. Recall that the token $\rightarrow$ can be provided instead of a fresh name to indicate on-the-fly substitution, and that the star token after a tactic indicates a call to `eauto`.

Lemma `triple_mcopy'` : $\forall L p,$
`triple (mcopy p)`
`(MList L p)`
`(funloc p'  $\Rightarrow$  (MList L p)  $\backslash$ * (MList L p'))`.

Proof using.

```
intros. gen p. induction_wf IH: list_sub L.
xwp. xapp. xchange MList_if. xif; intros C; case_if; xpull.
{ intros →. xapp. xsimpl×. subst. xchange× ← MList_nil. }
{ intros x q L' →. xapp. xapp. xapp. intros q'.
  xapp. intros p'. xchange ← MList_cons. xsimpl×. }
```

Qed.

38.2.6 Length Function for Lists

The function `mlength` computes the length of a mutable linked list.

OCaml:

```
let rec mlength p = if p == null then 0 else 1 + mlength p.tail
```

Definition `mlength` : **val** :=

```
<{ fix 'f 'p ⇒
  let 'b = ('p = null) in
  if 'b
  then 0
  else (let 'q = 'p'.tail in
        let 'n = 'f 'q in
        'n + 1) }>.
```

Exercise: 3 stars, standard, especially useful (`triple_mlength`) Prove the correctness of the function `mlength`. Hint: use the tactic `rew_list` to normalize list expressions – in particular, to prove `length L' + 1 = length (x :: L')`.

Lemma `triple_mlength` : $\forall L p,$

```
triple (mlength p)
(MList L p)
(fun r ⇒ \[r = val_int (length L)] \* (MList L p)).
```

Proof using. *Admitted*.

□

38.2.7 Alternative Length Function for Lists

In this section, we consider an alternative implementation of `mlength` that uses an auxiliary reference cell to keep track of the number of cells traversed so far. The list is traversed recursively, incrementing the contents of the reference once for every cell.

OCaml:

```
let rec listacc c p = if p == null then () else (incr c; listacc c p.tail)
let mlength' p = let c = ref 0 in listacc c p; !c
```

Definition `acclength` : **val** :=


```

<{ fix 'f 'c 'p ⇒
  let 'b = ('p ≠ null) in
  if 'b then
    incr 'c;
    let 'q = 'p'.tail in
    'f 'c 'q
  end }>.

```

Definition `mlength' : val :=`

```

<{ fun 'p ⇒
  let 'c = ref 0 in
  acclength 'c 'p;
  get_and_free 'c }>.

```

(Recall that `get_and_free` was defined in chapter [Basic](#).)

Exercise: 3 stars, standard, especially useful (`triple_mlength'`) Prove the correctness of the function `mlength'`. Hint: start by stating a lemma *triple_acclength* expressing the specification of the recursive function `acclength`. Make sure to generalize the appropriate variables before applying the well-founded induction tactic. Then complete the proof of the specification `triple_mlength'`, using *xapp triple_acclength* to reason about the call to the auxiliary function. Recall that *rew_list* can be used to simplify `length (x :: L')` into `length L' + 1` and that *math* can be used to prove arithmetic equalities such as $(n + 1) + m = n + (m + 1)$.

Lemma `triple_mlength' : ∀ L p,`

```

  triple (mlength' p)
    (MList L p)
    (fun r ⇒ \[r = val_int (length L)] \* (MList L p)).

```

Proof using. *Admitted*.

□

38.2.8 Free Function for Lists

The operation `mfree` deallocates all the cells in a given list. Each cell consists of a record that can be deallocated using the primitive operation *delete*. The deallocation of a list is implemented by recursively traversing the list, invoking *delete* on each cell after reading its tail pointer.

OCaml:

```
let rec mfree p = if p != null then begin let q = p.tail in delete p; mfree q end
```

Definition `mfree : val :=`

```

<{ fix 'f 'p ⇒
  let 'b = ('p ≠ null) in
  if 'b then

```

```

    let 'q = 'p'.tail in
    delete 'p;
    'f 'q
end }>.

```

The precondition of `mfree` requires a full list `MList L p`. The postcondition is empty: the entire list is destroyed.

Exercise: 3 stars, standard, especially useful (`Triple_mfree`) Verify the function `mfree`.

```

Lemma triple_mfree : ∀ L p,
  triple (mfree p)
    (MList L p)
    (fun _ ⇒ \ []).

```

Proof using. *Admitted*.

□

38.2.9 In-Place Reversal Function for Lists

The function `mrev` takes as argument a pointer to a mutable list, and returns a pointer on the reverse list, that is, a list with elements in the reverse order. The cells from the input list are reused for constructing the output list: the operation is said to be “in place”.

OCaml:

```

let rec mrev_aux p1 p2 = if p2 == null then p1 else (let p3 = p2.tail in p2.tail <- p1;
mrev_aux p2 p3)
let mrev p = mrev_aux null p

```

Definition `mrev_aux` : **val** :=

```

<{ fix 'f 'p1 'p2 ⇒
  let 'b = ('p2 = null) in
  if 'b
  then 'p1
  else (
    let 'p3 = 'p2'.tail in
    'p2'.tail := 'p1;
    'f 'p2 'p3) }>.

```

Definition `mrev` : **val** :=

```

<{ fun 'p ⇒
  mrev_aux null 'p }>.

```

Exercise: 4 stars, standard, optional (`triple_mrev`) Prove the correctness of the functions `mrev_aux` and `mrev`. Hint: here again, start by stating a lemma *triple_mrev_aux*

expressing the specification of the recursive function `mrev_aux`. Make sure to generalize the appropriate variables before applying the well-founded induction tactic.

Then complete the proof of `triple_mrev`, using *xapp triple_mrev_aux*. Hint: in `triple_mrev`, you will need to create an empty list out of thin air, using *xchange* $\leftarrow$ (`MList_nil` `null`). Also, make sure to call *rew_list* before *xsimpl*; otherwise *xsimpl* might not guess the right existential variable. The syntax *xsimpl* `v` is also available if you want to specify the witness `x` inside a goal of the form *H1* $\impl$ ($\exists$ `x`, *H2*).

```
Lemma triple_mrev :  $\forall$  L p,
  triple (mrev p)
    (MList L p)
    (funloc q  $\Rightarrow$  MList (rev L) q).
```

Proof using. *Admitted*.

□

38.3 More Details

The rest of this chapter presents the formal verification of more advanced data structures and algorithms. The remaining chapters in the book explain how to construct the program verification framework that we have been using so far.

The two aspects are completely orthogonal. Thus, if you have time and are interested in the practical aspects of program verification, you should complete the present chapter before moving on. On the other hand, if you are eager to dive into the construction of Separation Logic, then you may safely move on to the next chapter and come back to this later.

38.3.1 A Stack Represented as a List and a Size

Module `SIZEDSTACK`.

In this section, we consider the implementation of a mutable stack featuring a constant-time access to the size of the stack. This stack structure consists of a 2-field record storing a pointer to a mutable linked list plus an integer recording the length of that list. The implementation includes a function `create` to allocate an empty stack, a function `sizeof` for reading the size, and functions `push`, `top`, and `pop` for manipulating the top of the stack.

OCaml:

```
type 'a stack = { data : 'a list; size : int }
let create () = { data = null; size = 0 }
let sizeof s = s.size
let push p x = s.data <- mcons x s.data; s.size <- s.size + 1
let top s = let p = s.data in p.head
let pop s = let p = s.data in let x = p.head in let q = p.tail in delete p; s.data <- q in
s.size <- s.size - 1; x
```

The constants `data` and `size` are introduced as identifiers for the record fields of the type `'a stack`.

Definition `data : field := 0%nat`.

Definition `size : field := 1%nat`.

The representation predicate for the stack takes the form `Stack L s`, where `s` denotes the location of the record describing the stack and `L` denotes the list of items stored in the stack. The underlying mutable list is described as `MList L p`, where `p` is the location `p` stored in the first field of the record. The definition of `Stack` is as follows.

Definition `Stack (L:list val) (s:loc) : hprop :=`
`\∃ p, s ~~~> '{ data := p; size := length L } \* (MList L p).`

Observe that the predicate `Stack` does not expose the location of the mutable list; this location is existentially quantified in the definition. It also does not expose the size of the stack, as this value can be obtained from `length L`. Let's start with the specification and verification of `create` and `sizeof`.

Definition `create : val :=`
`<{ fun 'u =>`
`'{ data := null; size := 0 } }>.`

Lemma `triple_create :`
`triple (create ())`
`\ []`
`(funloc s => Stack nil s).`

Proof using.

`xwp. xapp triple_new_hrecord_2; auto. intros s.`
`unfold Stack. xsimpl×. xchange× ← (MList_nil null).`

Qed.

The `sizeof` operation returns the contents of the `size` field of a stack.

Definition `sizeof : val :=`
`<{ fun 'p =>`
`'p'.size }>.`

Lemma `triple_sizeof : ∀ L s,`
`triple (sizeof s)`
`(Stack L s)`
`(fun r => \[r = length L] \* Stack L s).`

Proof using.

`xwp. unfold Stack. xpull. intros p. xapp. xsimpl×.`

Qed.

The `push` operation extends the head of the list and increments the size field.

Definition `push : val :=`
`<{ fun 's 'x =>`

```

let 'p = 's'.data in
let 'p2 = mcons 'x 'p in
's'.data := 'p2;
let 'n = 's'.size in
let 'n2 = 'n + 1 in
's'.size := 'n2 }>.

```

Exercise: 3 stars, standard, especially useful (triple_push) Prove the following specification for the `push` operation.

Lemma `triple_push` : $\forall L\ s\ x$,
 $\text{triple } (\text{push } s\ x)$
 $(\text{Stack } L\ s)$
 $(\text{fun } u \Rightarrow \text{Stack } (x::L)\ s).$

Proof using. *Admitted*.

□

The `pop` operation extracts the element at the head of the list, updates the `data` field to the tail of the list and decrements the size field.

Definition `pop` : **val** :=

```

<{ fun 's =>
  let 'p = 's'.data in
  let 'x = 'p'.head in
  let 'p2 = 'p'.tail in
  delete 'p;
  's'.data := 'p2;
  let 'n = 's'.size in
  let 'n2 = 'n - 1 in
  's'.size := 'n2;
  'x }>.

```

Exercise: 4 stars, standard, especially useful (triple_pop) Prove the following specification for the `pop` operation. Hint: You'll need to unfold the definition of `Stack` as in the proofs above, and at some point you'll want to destruct L .

Lemma `triple_pop` : $\forall L\ s$,
 $L \neq \text{nil} \rightarrow$
 $\text{triple } (\text{pop } s)$
 $(\text{Stack } L\ s)$
 $(\text{fun } x \Rightarrow \exists L', \ [L = x::L'] \ \&* \text{Stack } L'\ s).$

Proof using. *Admitted*.

□

The `top` operation extracts the element at the head of the list.

Definition `top` : **val** :=

```

<{ fun 's =>
  let 'p = 's'.data in
  'p'.head }>.

```

Exercise: 2 stars, standard, optional (triple_top) Prove the following specification for the `top` operation.

```

Lemma triple_top : ∀ L s,
  L ≠ nil →
  triple (top s)
    (Stack L s)
    (fun x => \∃ L', \[L = x::L'] \* Stack L s).

```

Proof using. *Admitted*.

□

End SIZEDSTACK.

38.3.2 Formalization of the Tree Representation Predicate **MTree**

In this section, we generalize the ideas presented for linked lists to binary trees that store integer values in their nodes. Just as mutable lists are specified with respect to Rocq’s purely functional lists, mutable binary trees are specified with respect to Rocq trees.

Consider the following inductive definition of the type **tree**. A leaf is represented by the constructor `Leaf`, and a node takes the form `Node n T1 T2`, where `n` is an integer and `T1` and `T2` are its two subtrees.

```

Inductive tree : Type :=
| Leaf : tree
| Node : int → tree → tree → tree.

```

Implicit Types `T` : **tree**.

In a program manipulating a mutable tree, an empty tree is represented using the `null` pointer, and a node is represented in memory using a three-cell record. The first field, “item”, stores an integer. The other two fields, “left” and “right”, store pointers to the left and right subtrees, respectively.

Definition `item` : `field` := `0%nat`.

Definition `left` : `field` := `1%nat`.

Definition `right` : `field` := `2%nat`.

The heap predicate $p \sim\sim\sim \langle \{ \text{item} := n; \text{left} := p1; \text{right} := p2 \} \rangle$ describes a record allocated at location `p`, storing the integer `n` and the two pointers `p1` and `p2`.

The representation predicate `MTree T p`, of type `hprop`, asserts that the mutable tree structure with root at location `p` describes the logical tree `T`. The predicate is defined recursively on the structure of `T`.

- If `T` is a `Leaf`, then `p` is the null pointer.

- If T is a node `Node n T1 T2`, then p is not null and at location p one finds a record with field contents n , $p1$ and $p2$, with `MTree T1 p1` and `MTree T2 p2` describing the two subtrees.

```

Fixpoint MTree (T:tree) (p:loc) : hprop :=
  match T with
  | Leaf => \[p = null]
  | Node n T1 T2 => \exists p1 p2,
    (p ~~~> '{ item := n; left := p1; right := p2 })
    \* (MTree T1 p1)
    \* (MTree T2 p2)
  end.

```

38.3.3 Alternative Characterization of MTree

As for `MList`, it is very useful for proofs to state three lemmas that paraphrase the definition of `MTree`. The first two lemmas correspond to the folding/unfolding rules for leaves and nodes.

```

Lemma MTree_Leaf : ∀ p,
  (MTree Leaf p) = \[p = null].

```

Proof using. auto. Qed.

```

Lemma MTree_Node : ∀ p n T1 T2,
  (MTree (Node n T1 T2) p) =
  \exists p1 p2,
    (p ~~~> '{ item := n; left := p1; right := p2 })
    \* (MTree T1 p1) \* (MTree T2 p2).

```

Proof using. auto. Qed.

The third lemma reformulates `MTree T p` using a case analysis on whether p is the null pointer. This formulation matches the case analysis typically performed in the code of functions that operates on trees.

```

Lemma MTree_if : ∀ (p:loc) (T:tree),
  (MTree T p)
  ==> (If p = null
    then \[T = Leaf]
    else \exists n p1 p2 T1 T2, \[T = Node n T1 T2]
      \* (p ~~~> '{ item := n; left := p1; right := p2 })
      \* (MTree T1 p1) \* (MTree T2 p2)).

```

Proof using.

```

  intros. destruct T as [|n T1 T2].
  { xchange MTree_Leaf. intros M. case_if. xsimpl×. }
  { xchange MTree_Node. intros p1 p2.

```

```

    xchange hrecord_not_null. intros N. case-if. xsimpl×. }
Qed.

```

Beyond this point, the definition of *Mtree* not longer needs to be unfolded: we make it opaque.

```

Opaque MTree.

```

38.3.4 Additional Tooling for MTree

For reasoning about recursive functions over trees, it is useful to exploit the well-founded order associated with “immediate subtrees”. Concretely, `tree_sub T1 T` asserts that the tree *T1* is either the left or the right subtree of the tree *T*. This order may be exploited for verifying recursive functions over trees using the tactic *induction_wf IH: tree_sub T*. The relation `tree_sub` is defined as follows.

```

Inductive tree_sub : binary (tree) :=
| tree_sub_1 : ∀ n T1 T2,
    tree_sub T1 (Node n T1 T2)
| tree_sub_2 : ∀ n T1 T2,
    tree_sub T2 (Node n T1 T2).

```

Lemma `tree_sub_wf` : wf `tree_sub`.

Proof using.

```

    intros T. induction T; constructor; intros t' H; inversions¬ H.
Qed.

```

```

#[global] Hint Resolve tree_sub_wf : wf.

```

For allocating fresh tree nodes as a 3-field record, we introduce the operation `mnode n p1 p2`, defined and specified as follows.

```

Definition mnode : val :=
    val_new_hrecord_3 item left right.

```

A first specification of `mnode` describes the allocation of a record.

```

Lemma triple_mnode : ∀ n p1 p2,
    triple (mnode n p1 p2)
    \[]
    (funloc p => p ~~~> '{ item := n ; left := p1 ; right := p2 }).

```

Proof using. intros. apply `triple_new_hrecord_3`; auto. Qed.

A second specification, derived from the first, asserts that, when provided two subtrees *T1* and *T2* at locations `p1` and `p2`, the operation `mnode n p1 p2` builds, at a fresh location `p`, a tree described by *Mtree* `[Node n T1 T2] p`. Compared with the first specification, this second specification is said to “transfer ownership” of the two subtrees.

Exercise: 2 stars, standard, optional (triple_mnode') Prove the specification `triple_mnode'` for node allocation. Because this specification refines the previous specification `triple_mnode`, the proof should begin with *xapp* `triple_mnode`.

Lemma `triple_mnode'` : $\forall T1\ T2\ n\ p1\ p2,$
`triple (mnode  $n\ p1\ p2$ )`
`(MTree  $T1\ p1$   $\backslash*$  MTree  $T2\ p2$ )`
`(funloc  $p \Rightarrow$  MTree (Node  $n\ T1\ T2$ )  $p$ ).`

Proof using. *Admitted*.

#[*global*] Hint Resolve `triple_mnode'` : `triple`.

□

Similarly to what we had done for `mnil`, we could implement a function *mleaf* that returns the `null` pointer and derive a specification for it in terms of `MTree`. Instead, we will see through the next examples how one may directly reason about occurrences of `null` appearing in tree-manipulating code, using *xchange* $\leftarrow$ (`MTree_Leaf null`) directly.

38.3.5 Deep-Copy of a Tree

The operation `tree_copy` takes as argument a pointer `p` on a mutable tree and returns a fresh copy of that tree. It is defined in a similar way to the function `mcopy` for lists.

OCaml:

```
let rec tree_copy p = if p = null then null else mnode p.item (tree_copy p.left) (tree_copy p.right)
```

Definition `tree_copy` :=

```
<{ fix 'f 'p  $\Rightarrow$ 
  let 'b = ('p = null) in
  if 'b then null else (
    let 'n = 'p'.item in
    let 'p1 = 'p'.left in
    let 'p2 = 'p'.right in
    let 'q1 = 'f 'p1 in
    let 'q2 = 'f 'p2 in
    mnode 'n 'q1 'q2
  ) }>.
```

Exercise: 3 stars, standard, especially useful (triple_tree_copy) Prove the specification of `tree_copy`. Hint: you'll need to use *xchange* $\leftarrow$ (`MTree_Leaf null`) twice.

Lemma `triple_tree_copy` : $\forall p\ T,$
`triple (tree_copy  $p$ )`
`(MTree  $T\ p$ )`
`(funloc  $q \Rightarrow$  (MTree  $T\ p$ )  $\backslash*$  (MTree  $T\ q$ )).`

Proof using. *Admitted*.

□

38.3.6 Computing the Sum of the Items in a Tree

The operation `mtreesum` takes as argument the location `p` of a mutable tree, and it returns the sum of all the integers stored in the nodes of that tree. The implementation traverses the tree, using an auxiliary reference cell to maintain the sum of all the items visited so far.

OCaml:

```
let rec treeacc c p = if p <> null then ( c := !c + p.item; treeacc c p.left; treeacc c p.right)
```

```
let mtreesum p = let c = ref 0 in treeacc c p; !c
```

Definition `treeacc` : **val** :=

```
<{ fix 'f 'c 'p =>
  let 'b = ('p ≠ null) in
  if 'b then
    let 'm = ! 'c in
    let 'x = 'p'.item in
    let 'm2 = 'm + 'x in
    'c := 'm2;
    let 'p1 = 'p'.left in
    'f 'c 'p1;
    let 'p2 = 'p'.right in
    'f 'c 'p2
  end }>.
```

Definition `mtreesum` : **val** :=

```
<{ fun 'p =>
  let 'c = ref 0 in
  treeacc 'c 'p;
  get_and_free 'c }>.
```

The specification of `mtreesum` is expressed in terms of the Rocq function `treesum`, which computes the sum of the node items stored in a logical tree. This operation is defined by recursion over the tree.

Fixpoint `treesum` (*T*:**tree**) : **int** :=

```
match T with
| Leaf => 0
| Node n T1 T2 => n + treesum T1 + treesum T2
end.
```

Exercise: 4 stars, standard, optional (triple_mtreesum) Prove the correctness of the function `mtreesum`. Hint: to begin with, state and prove the specification lemma *triple_treeacc*.

Lemma `triple_mtreesum` : $\forall T\ p,$
`triple (mtreesum p)`
`(MTree T p)`
`(fun r  $\Rightarrow$   $\backslash[r = \text{treesum } T] \backslash * (\text{MTree } T\ p)$ )).`

Proof using. *Admitted*.

□

38.3.7 Verification of a Counter Function with Local State

This section is concerned with the verification of counter functions, which feature internal, mutable state. A counter function `f` is a function that, each time it is called, returns the next integer. Concretely, the first call to `f()` returns 1, the second call returns 2, the third returns 3, and so on.

A counter function can be implemented using a reference cell, `p`, that stores the integer last returned by the counter. Initially, the contents of the cell is zero. Each time the counter function is called, the contents is increased by one unit, and the new value of the contents is returned to the caller.

The function `create_counter` produces a fresh counter function. Concretely, `create_counter()` returns a counter function `f` that is independent from any other previously existing counter function.

OCaml:

```
let create_counter () = let p = ref 0 in (fun () -> (incr p; !p))
```

Definition `create_counter` : `val` :=

```
<{ fun 'u  $\Rightarrow$ 
  let 'p = ref 0 in
  (fun_ 'u  $\Rightarrow$  (incr 'p; !'p)) }>.
```

In this section, we present two specifications for counter functions. The first specification is the most direct, but it exposes the existence of the reference cell, revealing implementation details about the counter function. The second specification is more abstract, hiding from the client the internal representation of the counter using an abstract representation predicate.

Let us begin with the simple, direct specification. The proposition `CounterSpec f p` asserts that `f` is a counter function `f` whose internal state is stored in a reference cell at location `p`. Thus, invoking `f` in a state `p  $\sim\sim$  > m` updates the state to `p  $\sim\sim$  > (m+1)` and produces the output value `m+1`.

Definition `CounterSpec` (`f:val`) (`p:loc`) : `Prop` :=

```
 $\forall m,$  triple (f ())  

  (p  $\sim\sim$  > m)  

  (fun r  $\Rightarrow$   $\backslash[r = m+1] \backslash * p \sim\sim > (m+1)$ )).
```

Implicit Type `f` : `val`.

The function `create_counter` creates a fresh counter. Its precondition is empty. Its postcondition asserts that the function `f` being returned satisfies `CounterSpec f p` and the

output state contains a cell $p \rightsquigarrow 0$ for some existentially quantified location p .

Lemma `triple_create_counter` :

```
triple (create_counter ())
  \[]
  (fun f => \exists p, (p ~> 0) \* \[CounterSpec f p]).
```

Observe how the postcondition that appears in the triple above refers to `CounterSpec`, which itself involves a triple. In other words, a triple appears nested inside another triple. In technical terms, we are leveraging the “impredicativity of triples”, which holds as a consequence of the “impredicativity of predicates” in the logic of Rocq.

The proof involves the use of a new tactic, called `xfun`, for reasoning about local function definitions. Here, `xfun` gives us the hypothesis Hf that specifies the code of f . This hypothesis is of the form: $\forall v \ H' \ Q', (PRE \ H' \ CODE \ B \ POST \ Q') \rightarrow \text{triple} (f \ v) \ H' \ Q'$, where B denotes the code of the body of the function f instantiated on the argument v , and where H' and Q' denote arbitrary pre- and postconditions. Exploiting this assumption enables the user to subsequently derive triples about the local function f , by reasoning about the code of the f , just in the same way as when the user invokes `xwp` for reasoning about a top-level function. **Proof using.**

```
xwp. xapp. intros p.
xfun. intros f Hf.
xsimpl.
{ intros m.
  xapp.
  xapp. xapp. xsimpl. auto. }
```

Qed.

Let us move on to the presentation of more abstract specifications. Their purpose is to hide from the client the existence of the reference cell used to represent the internal state of the counter functions. To that end, we introduce the heap predicate `IsCounter f n`, which relates a function f , its current value n , and the piece of memory state involved in the implementation of this function. This piece of memory is of the form $p \rightsquigarrow n$, for some location p , such that `CounterSpec f p` holds.

Definition `IsCounter (f:val) (n:int) : hprop :=`

```
\exists p, p ~> n \* \[CounterSpec f p].
```

Using `IsCounter`, we can reformulate the specification of `create_counter` with a postcondition asserting that the output function f is described by the heap predicate `IsCounter f 0`.

Lemma `triple_create_counter_abstract` :

```
triple (create_counter ())
  \[]
  (fun f => IsCounter f 0).
```

This lemma is the same as `triple_create_counter`, except that the reference cell p is no longer explicitly mentioned in the postcondition. In other words, the address of the reference

cell `p` used internally by the counter becomes hidden from the user. *Proof using. unfold IsCounter. apply triple_create_counter. Qed.*

Next, we reformulate the specification of a call to a counter function. A call to `f()`, in a state satisfying `IsCounter f n`, produces a state satisfying `IsCounter f (n+1)`, and returns `n+1`.

Exercise: 4 stars, standard, especially useful (triple_apply_counter_abstract) Prove the abstract specification for a counter function. You will need to begin the proof using the tactic *xtriple*, for turning goal into a form on which other x-tactics can be invoked. Then, use *xpull* to extract facts from the precondition.

```
Lemma triple_apply_counter_abstract : ∀ f n,
  triple (f ())
    (IsCounter f n)
    (fun r => \[r = n+1] \* (IsCounter f (n+1))).
```

Proof using. Admitted.

□

`Opaque IsCounter.`

Finally, let us illustrate how a client might work with counter functions.

OCaml:

```
let test_counter () = let c1 = create_counter () in let c2 = create_counter () in let n1 =
c1() in let n2 = c2() in let n3 = c1() in n2 + n3
```

Definition `test_counter` : **val** :=

```
<{ fun 'u =>
  let 'c1 = create_counter () in
  let 'c2 = create_counter () in
  let 'n1 = 'c1 () in
  let 'n2 = 'c2 () in
  let 'n3 = 'c1 () in
  'n2 + 'n3 }>.
```

Exercise: 2 stars, standard, optional (triple_test_counter) Prove the example function manipulating abstract counters. In the specification below, the heap predicate $\exists H$, *H* corresponds to the two counters, which we currently have no way of deallocating. The necessary mechanism for garbage collection will be introduced later in chapter *Affine*.

Hint: use *xapp* `triple_create_counter_abstract` and *xapp* `triple_apply_counter_abstract` to invoke the specifications.

```
Lemma triple_test_counter :
  triple (test_counter ())
    \[]
    (fun r => \[r = 3] \* (\exists H, H)).
```

Proof using. *Admitted*.

□

38.4 Optional Material

The verification of higher-order functions, and of their interaction with mutable state, was a formidable challenge prior to the introduction of Separation Logic. The rest of this chapter illustrates how to reason about imperative, higher-order programs.

38.4.1 Specification of a Higher-Order Repeat Operator

Consider the higher-order iterator `repeat`: a call to `repeat f n` executes `n` times the call `f()`. OCaml:

```
let rec repeat f n = if n > 0 then (f(); repeat f (n-1))
```

Definition `repeat` : `val` :=

```
<{ fix 'g 'f 'n =>
  let 'b = ('n > 0) in
  if 'b then
    'f ();
    let 'n2 = 'n - 1 in
    'g 'f 'n2
  end }>.
```

For simplicity, let us assume for now $n \geq 0$. The specification of `repeat n f` can be expressed in terms of an invariant, named I , describing the state in between every two calls to `f`. We assume that the initial state satisfies $I\ 0$. Moreover, we assume that, for every index i in the range from 0 (inclusive) to n (exclusive), a call `f()` can execute in a state that satisfies $I\ i$ and produce a state that satisfies $I\ (i+1)$. The specification below asserts that, under these two assumptions, after the `n` calls to `f()`, the final state satisfies $I\ n$. The specification takes the form:

$n \geq 0 \rightarrow \text{Hypothesis\_on\_f} \rightarrow \text{triple } (\text{repeat } f\ n) (I\ 0) (\text{fun } u \Rightarrow I\ n)$
where *Hypothesis_on_f* is a proposition that captures the following specification:
 $\text{forall } i, 0 \leq i < n \rightarrow \text{triple } (f\ ()) (I\ i) (\text{fun } u \Rightarrow I\ (i+1))$
The complete specification of `repeat n f` is thus as shown below.

Lemma `triple_repeat` : $\forall (I:\text{int} \rightarrow \text{hprop}) (f:\text{val}) (n:\text{int}),$
 $n \geq 0 \rightarrow$
 $(\forall i, 0 \leq i < n \rightarrow$
 $\text{triple } (f\ ())$
 $(I\ i)$
 $(\text{fun } u \Rightarrow I\ (i+1))) \rightarrow$
 $\text{triple } (\text{repeat } f\ n)$
 $(I\ 0)$

```

    (fun u => I n).
Proof using.
  introv Hn Hf.

```

To establish this specification, we carry out a proof by induction over a generalized specification, covering the case where there remains `m` iterations to perform, for any value of `m` between 0 and `n` inclusive.

```
forall m, 0 <= m <= n -> triple (repeat f m) (I (n-m)) (fun u => I n))
```

We use the TLC tactics `cuts`, a variant of `cut`, to state show that the generalized specification entails the statement of `triple-repeat`. `cuts G: (∀ m, 0 ≤ m ≤ n →`

```

triple (repeat f m)
  (I (n-m))
  (fun u => I n)).

```

```
{ replace 0 with (n - n). { eapply G. math. } { math. } }
```

We then carry a proof by induction: during the execution, the value of `m` decreases step by step down to 0. `intros m. induction_wf IH: (downto 0) m. intros Hm.`

```
xwp. xapp. xif; intros C.
```

We reason about the call to `f` `{ xapp. { math. } xapp.`

We next reason about the recursive call. `xapp. { math. } { math. }`

We need to exploit an arithmetic equality. We do so using `math_rewrite`, which is a convenient TLC tactic to assert an equality proved by the `math` tactic, then immediately invoke `rewrite` with this equality. `math_rewrite ((n - m) + 1 = n - (m - 1)). xsimpl. }`

Finally, when `m` reaches zero, we check that we obtain `I n`. `{ xval. math_rewrite (n - m = n). xsimpl. }`

```
Qed.
```

38.4.2 Specification of an Iterator on Mutable Lists

The operation `miter` takes as argument a function `f` and a pointer `p` on a mutable list and invokes `f` on each of the items stored in that list.

OCaml:

```
let rec miter f p = if p <> null then (f p.head; miter f p.tail)
```

Definition `miter` : `val` :=

```

<{ fix 'g 'f 'p =>
  let 'b = ('p ≠ null) in
  if 'b then
    let 'x = 'p'.head in
    'f 'x;
    let 'q = 'p'.tail in
    'g 'f 'q
  end }>.

```

The specification of `miter` follows the same structure as that of the function `repeat` from the previous section, with two main differences. The first difference is that the invariant is

expressed not in terms of an index `i` ranging from 0 to `n`, but in terms of a prefix of the list `L` being traversed. This prefix ranges from `nil` to the full list `L`. The second difference is that the operation `miter f p` requires in its precondition, in addition to `I nil`, the description of the mutable list `MList L p`. This predicate is returned in the postcondition, unchanged, reflecting the fact that the iteration process does not alter the contents of the list.

Exercise: 5 stars, standard, especially useful (`triple_miter`) Prove the correctness of `triple_miter`.

```
Lemma triple_miter : ∀ (I:list val→hprop) L (f:val) p,
  (∀ x L1 L2, L = L1++x::L2 →
    triple (f x)
      (I L1)
      (fun u ⇒ I (L1++(x::nil)))) →
  triple (miter f p)
    (MList L p \* I nil)
    (fun u ⇒ MList L p \* I L).
```

Proof using. *Admitted*.

□

For exploiting the specification `triple_miter` to reason about a call to `miter`, it is necessary to provide an invariant `I` of type `list val → hprop`, that is, of the form `fun (K:list val) ⇒ ...`. This invariant, which generally cannot be inferred automatically, should describe the state at the point where the iteration has traversed a prefix `K` of the list `L`. Concretely, for reasoning about a call to `miter`, one should exploit the tactic `xapp (triple_miter (fun K ⇒ ...))`. An example appears next.

38.4.3 Computing the Length of a Mutable List using an Iterator

The function `mlength_using_miter` computes the length of a mutable list by iterating over that list a function that increments a reference cell once for every item.

OCaml:

```
let mlength_using_miter p = let c = ref 0 in miter (fun x -> incr c) p; !c
```

Exercise: 4 stars, standard, especially useful (`triple_mlength_using_miter`) Prove the correctness of `mlength_using_iter`. Hint: as explained earlier, use `xfun`; `intros f Hf` for reasoning about the function definition, then use `xapp` for reasoning about a call to `f`. Use of `xlet` is optional.

```
Definition mlength_using_miter : val :=
  <{ fun 'p ⇒
    let 'c = ref 0 in
    let 'f = (fun _ 'x ⇒ incr 'c) in
    miter 'f 'p;
```



```

    get_and_free 'c }>.
Lemma triple_mlength_using_miter :  $\forall$   $p$   $L$ ,
  triple (mlength_using_miter  $p$ )
    (MList  $L$   $p$ )
    (fun  $r \Rightarrow \backslash[r = \text{length } L] \backslash * \text{MList } L$   $p$ ).
Proof using. Admitted.
□

```

38.4.4 Factorial Function in Continuation-Passing Style

This section and the next one present examples of functions involving “continuations”. As a warm-up, we first consider a function that does not involve any mutable state.

The function `cps_facto_aux n k` performs a call to the function `k` on the value `facto n`. The function `cps_facto n` computes the value of `facto n` by applying `cps_facto_aux` to the identity function.

OCaml:

```

let rec cps_facto_aux n k = if n <= 1 then k 1 else cps_facto_aux (n-1) (fun r -> k (n *
r))
let cps_facto n = cps_facto_aux n (fun r -> r)

```

Definition `cps_facto_aux` : **val** :=

```

<{ fix 'f 'n 'k  $\Rightarrow$ 
  let 'b = 'n  $\leq$  1 in
  if 'b
  then 'k 1
  else let 'k2 = (fun_ 'r  $\Rightarrow$  let 'r2 = 'n  $\times$  'r in 'k 'r2) in
    let 'n2 = 'n - 1 in
    'f 'n2 'k2 }>.

```

Definition `cps_facto` : **val** :=

```

<{ fun 'n  $\Rightarrow$ 
  let 'k = (fun_ 'r  $\Rightarrow$  'r) in
  cps_facto_aux 'n 'k }>.

```

Import `FACTO`.

Exercise: 4 stars, standard, optional (`triple_cps_facto_aux`) Verify `cps_facto_aux`. Hints: To set up the induction, use the usual pattern *induction_wf IH*: (`downto 0`) `n`. To reason about the function definition, use *xfun*. To reason about the recursive call, you’ll need to exploit the induction hypothesis, called *IH*, which universally quantifies over a function *F* of type *int*→*int*. To do so, use the syntax *xapp* (*» IH* (`fun a  $\Rightarrow$  ..`)). The function provided should describe the behavior of the continuation `k2` that appears in the code. For the mathematical reasoning, use the same pattern as in the proof of `factorec`, applying the tactics `rewrite facto_init` and `rewrite (@facto_step n)`.

```

Lemma triple_cps_facto_aux :  $\forall$  ( $n$ :int) ( $k$ :val) ( $F$ :int $\rightarrow$ int),
   $n \geq 0 \rightarrow$ 
  ( $\forall$  ( $a$ :int), triple ( $k$   $a$ )  $\backslash []$  (fun  $r \Rightarrow \backslash[r = F\ a]$ ))  $\rightarrow$ 
  triple (cps_facto_aux  $n$   $k$ )
     $\backslash []$ 
    (fun  $r \Rightarrow \backslash[r = F$  (facto  $n$ )]).
Proof using. Admitted.
□

```

Exercise: 2 stars, standard, optional (triple_cps_facto) Verify `cps_facto`. Hint: use the syntax `xapp` ($\gg$ `triple_cps_append_aux` F) to provide the function F that describes the behavior of the identity continuation.

```

Lemma triple_cps_facto :  $\forall$   $n$ ,
   $n \geq 0 \rightarrow$ 
  triple (cps_facto  $n$ )
     $\backslash []$ 
    (fun  $r \Rightarrow \backslash[r =$  facto  $n]$ ).
Proof using. Admitted.
□

```

38.4.5 In-Place Concatenation Function in Continuation-Passing Style

This section presents an example verification of a function involving “continuations”. The function `cps_append` is similar to the function `append` presented previously: it also performs in-place concatenation of two lists. The main difference is that it is implemented using an auxiliary recursive function in “continuation-passing style” (CPS).

The presentation of `cps_append p1 p2` is also slightly different: this operation returns a pointer `p3` that describes the head of the result of the concatenation. In the general case, `p3` is equal to `p1`, but if `p1` is the null pointer, meaning that the first list is empty, then `p3` is equal to `p2`.

The code of `cps_append` involves the auxiliary function `cps_append_aux p1 p2 k`, which invokes the continuation function `k` on the result of concatenating the lists at locations `p1` and `p2`. Its code appears at first quite puzzling, because the recursive call is performed inside the continuation. It takes a good drawing and at least a couple minutes to figure out how the function works.

OCaml:

```

let rec cps_append_aux p1 p2 k = if p1 == null then k p2 else cps_append_aux p1.tail
p2 (fun r => (p1.tail <- r); k p1)
let cps_append p1 p2 = cps_append_aux p1 p2 (fun r => r)

```

Definition `cps_append_aux` : **val** :=

```

<{ fix 'f 'p1 'p2 'k ⇒
  let 'b = ('p1 = null) in
  if 'b
  then 'k 'p2
  else
    let 'q1 = 'p1'.tail in
    let 'k2 = (fun_ 'r ⇒ ('p1'.tail := 'r; 'k 'p1)) in
    'f 'q1 'p2 'k2 }>.

```

Definition `cps_append` : **val** :=

```

<{ fun 'p1 'p2 ⇒
  let 'f = (fun_ 'r ⇒ 'r) in
  cps_append_aux 'p1 'p2 'f }>.

```

The goal is to establish the following specification for `cps_append`.

Lemma `triple_cps_append` : forall (L1 L2:list val) (p1 p2:loc), triple (cps_append p1 p2) (MList L1 p1 * MList L2 p2) (funloc p3 => MList (L1++L2) p3).

If you are interested in the challenge of solving a 6-star exercise, then try to prove the above specification without reading any further. If, however, you are only looking for a 5-star exercise, keep reading.

Exercise: 5 stars, standard, optional (triple_cps_append_aux) The specification of `cps_append_aux` involves an hypothesis describing the behavior of the continuation `k`. For this function, and more generally for code in CPS form, we cannot easily leverage the frame property, thus we need to quantify explicitly over a heap predicate H for describing the “rest of the state”. Prove that specification. Hint: you can use the syntax *xapp* ($\gg IH\ H'$) to instantiate the induction hypothesis IH on a specific heap predicate H' .

Lemma `triple_cps_append_aux` : $\forall\ H\ Q\ (L1\ L2:\text{list val})\ (p1\ p2:\text{loc})\ (k:\text{val})$,
 $(\forall\ (p3:\text{loc}), \text{triple } (k\ p3)\ (\text{MList } (L1\ ++\ L2)\ p3\ \backslash * H)\ Q) \rightarrow$
 $\text{triple } (\text{cps\_append\_aux } p1\ p2\ k)$
 $(\text{MList } L1\ p1\ \backslash * \text{MList } L2\ p2\ \backslash * H)$
 Q .

Proof using. *Admitted*.

□

Exercise: 3 stars, standard, optional (triple_cps_append) Verify `cps_append`. Hint: use the syntax `@triple_cps_append_aux H' Q'` to specialize the specification of the auxiliary function to specific pre- and post-conditions.

Lemma `triple_cps_append` : $\forall\ (L1\ L2:\text{list val})\ (p1\ p2:\text{loc})$,
 $\text{triple } (\text{cps\_append } p1\ p2)$
 $(\text{MList } L1\ p1\ \backslash * \text{MList } L2\ p2)$
 $(\text{funloc } p3 \Rightarrow \text{MList } (L1++L2)\ p3)$.

Proof using. *Admitted*.

□

38.4.6 Historical Notes

The representation predicate for lists appears in the seminal publications on Separation Logic: the notes by Reynolds from the summer 1999, updated the next summer *Reynolds* 2000 (in Bib.v), and the paper by *O’Hearn, Reynolds, and Yang* 2001 (in Bib.v). The function `cps_append` was proposed in Reynolds’s article as an open challenge for verification. Zhaozhong Ni and Zhong Shao presented the framework XCAP, which was the first to support reasoning on embedded code pointers *Ni, Shao* 2006 (in Bib.v). Their proof, carried on a 23-line assembly program, consists of 1500+ lines of Rocq. Subsequently, Charguéraud presented a concise proof for a high-level implementation of the `cps_append` function, essentially equivalent to the formalization presented above *Charguéraud* 2011 (in Bib.v).

The specification of higher-order iterators requires higher-order Separation Logic. Being embedded in the higher-order logic of Rocq, the Separation Logic that we work with is inherently higher-order. Further information on the history of higher-order Separation Logic for higher-order programs may be found in the companion course notes, linked in the *Preface*.

Chapter 39

Library SLF.Hprop

39.1 Hprop: Heap Predicates

```
Set Implicit Arguments.  
From SLF Require Export LibSepReference.  
Import PROGRAMSYNTAX.
```

39.2 First Pass

The first two chapters have illustrated how to specify and verify programs using Separation Logic. The purpose of the coming chapters is to explain how to formally assign meaning to the tokens used in the statement of specifications (e.g., the star operator), formally define the notion of Separation Triples, and prove the reasoning rules of Separation Logic. These rules are exploited by the “x-tactics” used in the first two chapters.

This chapter presents the formal definitions of the key heap predicate operators from Separation Logic:

- $\backslash []$ denotes the empty heap predicate,
- $\backslash [P]$ denotes a pure fact,
- $p \sim\sim v$ denotes a predicate that characterizes a singleton heap,
- $H1 \backslash * H2$ denotes the separating conjunction,
- $Q1 \backslash * + H2$ denotes the separating conjunction between a postcondition and a heap predicate,
- $\backslash \exists x, H$ denotes an existential quantifier.

To begin with, we define the types `heap`, used to represent a memory state, or a piece of a memory state.

In Separation Logic, specifications are expressed using “heap predicates”, predicates over heaps, of type `heap → Prop`. We define `hprop` to be a shorthand for that type.

By convention, throughout the course:

- H denotes a heap predicate of type `hprop`; it describes a piece of state,
- Q denotes a postcondition, of type `val → hprop`; it describes both a result value and a piece of state.

39.2.1 Description of Memory States

To reason about imperative programs, Separation Logic relies on heap predicates for describing memory states. As the name suggests, a “heap predicate” is a predicate over heaps, i.e., over memory states. Heap predicates such as $p \rightsquigarrow n$, or $H1 \setminus^* H2$ admit the type `heap → Prop`, where the type `heap` corresponds to a finite map used to describe the contents of a piece of the memory. In what follows, we give the formal definition of `heap` and of the type of heap predicates.

In the previous chapters, we have assumed a type `val` to range over program values and a type `loc` to denote memory locations. Concrete definitions will be provided later on, in chapter [Rules](#). At this point, we are interested in the definition of the type `heap`, used to describe the contents of the memory during the execution of a program.

A `heap` is a finite map from locations to values. The file `LibSepFmap.v` provides a self-contained formalization of finite maps. An object of type `fmap A B` represents a finite map binding keys of type `A` to values of type `B`. We define the type `heap` as a map from locations, of type `loc`, to program values, of type `val`. Internally, `fmap A B` is implemented as a function of type `A → option B`, like in `Maps.v` from the LF volume.

Definition `heap : Type := fmap loc val`.

The file `LibSepReference` introduces the module name `FMAP` as a shorthand for `LibSepFmap`. The key operations associated with finite maps are:

- `Fmap.empty` denotes the empty state,
- `Fmap.single p v` denotes a singleton state, that is, a unique cell at location `p` with contents `v`,
- `Fmap.union h1 h2` denotes the union of the two states `h1` and `h2`.
- `Fmap.disjoint h1 h2` asserts that `h1` and `h2` have disjoint domains.

Note that the union operation is commutative only when its arguments have disjoint domains. Throughout the course, we only consider disjoint unions, for which commutativity holds.

Hereafter, to improve readability of statements in proofs, we introduce the notation `h1 \u h2` for heap union.

Notation `"h1 \u h2" := (Fmap.union h1 h2) (at level 37, right associativity)`.

39.2.2 Heap Predicates

In Separation Logic, the state is described using “heap predicates”, of type `hprop`.

Definition `hprop := heap → Prop`.

H ranges over heap predicates.

Implicit Type $H : \text{hprop}$.

An essential aspect of Separation Logic is that all heap predicates defined and used in practice are built using a few basic combinators. In other words, except for the definition of the combinators themselves, we will never define a custom heap predicate directly as a function of the state.

We next describe the most important combinators, which were used pervasively throughout the first two chapters.

The `hempty` predicate, usually written $\llbracket \rrbracket$, characterizes an empty heap.

Definition `hempty : hprop :=`

`fun (h:heap) ⇒ (h = Fmap.empty).`

Notation $\llbracket \rrbracket := (\text{hempty})$ (at level 0).

The pure fact predicate, written $\llbracket P \rrbracket$, characterizes an empty state and moreover asserts that the proposition P is true.

Definition `hpure (P:Prop) : hprop :=`

`fun (h:heap) ⇒ (h = Fmap.empty) ∧ P.`

Notation $\llbracket P \rrbracket := (\text{hpure } P)$ (at level 0, *format* $\llbracket P \rrbracket$).

The singleton heap predicate, written $p \rightsquigarrow v$, characterizes a state with a single allocated cell, at location p , storing the value v .

Definition `hsingle (p:loc) (v:val) : hprop :=`

`fun (h:heap) ⇒ (h = Fmap.single p v).`

Notation $p \rightsquigarrow v := (\text{hsingle } p \ v)$ (at level 32).

The “separating conjunction”, written $H1 \searrow^* H2$, characterizes a state that can be decomposed in two disjoint parts, one that satisfies $H1$ and another that satisfies $H2$.

Definition `hstar (H1 H2 : hprop) : hprop :=`

`fun (h:heap) ⇒ ∃ h1 h2, H1 h1
 ∧ H2 h2
 ∧ Fmap.disjoint h1 h2
 ∧ h = h1 \u h2.`

Notation $H1 \searrow^* H2 := (\text{hstar } H1 \ H2)$ (at level 41, right associativity).

The existential quantifier for heap predicates, written $\searrow \exists x, H$ characterizes a heap that satisfies H for some x . The variable x has type A , for some arbitrary A .

The notation $\searrow \exists x, H$ stands for `hexists (fun x ⇒ H)`. The generalized notation $\searrow \exists x1 \dots xn, H$ is also available.

The definition of `hexists` is a bit technical, and it is not essential to master it at this point. Additional explanations are provided near the end of this chapter.

Definition `hexists` ($A:\text{Type}$) ($J:A\rightarrow\text{hprop}$) : `hprop` :=
`fun (h:heap) => ∃ x, J x h.`

Notation `"'\exists' x1 .. xn , H"` :=
`(hexists (fun x1 => .. (hexists (fun xn => H)) ..))`
`(at level 39, x1 binder, H at level 50, right associativity,`
`format "'[\exists' '/' x1 .. xn , '/' H ]'")`.

Universal quantification in `hprop` is also possible, but it is only useful for more advanced features of Separation Logic. We postpone its introduction to chapter [Wand](#).

All the definitions above will eventually be made `Opaque`, after the appropriate introduction and elimination lemmas have been established, making it no longer possible to execute, say, `unfold hstar`. Opacity is essential to ensure that proofs do not depend on the details of the definitions.

39.2.3 Extensionality for Heap Predicates

To work in Separation Logic, it is extremely convenient to be able to state equalities between heap predicates. For example, we wish to establish the associativity property for the star operator in the form of an equality:

Parameter `hstar_assoc_statement` : $\forall H1\ H2\ H3,$
 $((H1 \star H2) \star H3) = (H1 \star (H2 \star H3)).$

How can we prove a goal of the form $H1 = H2$ when $H1$ and $H2$ have type `hprop`, that is, `heap→Prop`? Intuitively, H and H' are equal if and only if they characterize exactly the same set of heaps, that is, if $\forall (h:\text{heap}), H1\ h \leftrightarrow H2\ h$.

This reasoning principle, a specific form of extensionality property, is not available by default in Rocq. But we can safely assume it if we extend the logic of Rocq with a standard axiom called “predicate extensionality”.

Axiom `predicate_extensionality` : $\forall (A:\text{Type}) (P\ Q:A\rightarrow\text{Prop}),$
 $(\forall x, P\ x \leftrightarrow Q\ x) \rightarrow$
 $P = Q.$

By specializing P and Q above to the type `hprop`, we obtain exactly the desired extensionality principle.

Lemma `hprop_eq` : $\forall (H1\ H2:\text{hprop}),$
 $(\forall (h:\text{heap}), H1\ h \leftrightarrow H2\ h) \rightarrow$
 $H1 = H2.$

Proof using. `apply predicate_extensionality. Qed.`

More information about extensionality axioms may be found near the end of this chapter.

39.2.4 Fundamental Properties of Operators

By exploiting extensionality, the following properties of Separation Logic operators can be established (they are proved further on in the chapter).

(1) The star operator is associative.

Parameter *hstar_assoc* : $\forall H1\ H2\ H3,$
 $(H1 \ \backslash* \ H2) \ \backslash* \ H3 = H1 \ \backslash* \ (H2 \ \backslash* \ H3).$

(2) The star operator is commutative.

Parameter *hstar_comm* : $\forall H1\ H2,$
 $H1 \ \backslash* \ H2 = H2 \ \backslash* \ H1.$

(3) The empty heap predicate is a neutral element for the star.

Because star is commutative, it is equivalent to state that **hempty** is a left or a right neutral element for **hstar**. We choose, arbitrarily, to state the left-neutrality property.

Parameter *hstar_hempty_l* : $\forall H,$
 $\backslash [] \ \backslash* \ H = H.$

(4) Existentials can be “extruded” out of stars, that is: $(\backslash \exists x, J\ x) \ \backslash* \ H$ is equivalent to $\backslash \exists x, (J\ x \ \backslash* \ H).$

Parameter *hstar_hexists* : $\forall A\ (J:A \rightarrow \text{hprop})\ H,$
 $(\backslash \exists x, J\ x) \ \backslash* \ H = \backslash \exists x, (J\ x \ \backslash* \ H).$

(5) Pure facts can be extruded out of stars.

Parameter *hstar_hpure_l* : $\forall P\ H\ h,$
 $(\backslash [P] \ \backslash* \ H) \ \text{h} = (P \wedge H\ h).$

39.2.5 Postconditions: Type, Syntax, and Extensionality

A specification takes the form **triple** *t* *H* *Q*, where *t* is a term, *H* is a precondition of type **hprop**, and *Q* is a postcondition.

For example, a read at location *p*, written **!p** in OCaml, is specified as: **triple** $\langle \{ !p \} \rangle$ $(p \rightsquigarrow v) (\text{fun } r \Rightarrow (p \rightsquigarrow v) \ \backslash* \ [r = v]).$

In general, a postcondition has type **val** $\rightarrow$ **hprop**, which is equivalent to **val** $\rightarrow$ **heap** $\rightarrow$ **Prop**. The postcondition thereby capture the properties of both the output value and the output state.

Implicit Type *Q* : **val** $\rightarrow$ **hprop**.

One common operation is augmenting a postcondition with a description of another piece of state. This operation is written as $Q \ \backslash*+ \ H$, which is just a convenient notation for **fun** *x* $\Rightarrow (Q\ x \ \backslash* \ H)$. We will use this operator in particular in the statement of the frame rule in the next chapter.

Notation “ $Q \ \backslash*+ \ H$ ” := **(fun** *x* $\Rightarrow$ **hstar** (*Q* *x*) *H*) **(at level 40).**

Intuitively, in order to prove that two postconditions $Q1$ and $Q2$ are equal, it suffices to show that the heap predicates $Q1 \vee$ and $Q2 \vee$ are equal for any value v .

Again, the extensionality property that we need is not built into Rocq. We need another axiom called “functional extensionality”.

```
Axiom functional_extensionality :  $\forall A B (f g : A \rightarrow B),$ 
  ( $\forall x, f\ x = g\ x$ )  $\rightarrow$ 
   $f = g$ .
```

The desired equality property for postconditions follows directly from this axiom.

```
Lemma qprop_eq :  $\forall (Q1\ Q2 : \text{val} \rightarrow \text{hprop}),$ 
  ( $\forall (v : \text{val}), Q1\ v = Q2\ v$ )  $\rightarrow$ 
   $Q1 = Q2$ .
```

Proof using. apply *functional_extensionality*. Qed.

39.2.6 Introduction and Inversion Lemmas for Basic Operators

This section presents a collection of lemmas capturing the properties of the definitions of heap predicates. They include “introduction lemmas” for proving goals of the form $H\ h$, and “inversion lemmas” for extracting information from hypotheses of the form $H\ h$. These lemmas will be exploited pervasively in the next chapters for establishing reasoning rules.

```
Implicit Types  $P : \text{Prop}$ .
```

```
Implicit Types  $v : \text{val}$ .
```

The introduction lemmas show how to prove goals of the form $H\ h$, for various forms of the heap predicate H .

```
Lemma hempty_intro :
   $\backslash []$  Fmap.empty.
```

Proof using.

The tactic *hnf*, for “head normal form”, unfolds the head constants. *hnf. auto.*
Qed.

```
Lemma hpure_intro :  $\forall P,$ 
   $P \rightarrow$ 
   $\backslash [P]$  Fmap.empty.
```

Proof using. *introv M. hnf. auto.* Qed.

```
Lemma hsingle_intro :  $\forall p\ v,$ 
  ( $p \rightsquigarrow v$ ) (Fmap.single p v).
```

Proof using. *intros. hnf. auto.* Qed.

```
Lemma hstar_intro :  $\forall H1\ H2\ h1\ h2,$ 
   $H1\ h1 \rightarrow$ 
   $H2\ h2 \rightarrow$ 
  Fmap.disjoint  $h1\ h2 \rightarrow$ 
  ( $H1 \backslash * H2$ ) ( $h1 \backslash u h2$ ).
```

Proof using. `intros.  $\exists$ ×  $h1$   $h2$ . Qed.`

Lemma `hexists_intro` : $\forall A (x:A) (J:A \rightarrow \text{hprop}) h,$
 $J\ x\ h \rightarrow$
 $(\backslash \exists\ x, J\ x)\ h.$

Proof using. `introv  $M$ . hnf. eauto. Qed.`

The inversion lemmas show how to extract information from hypotheses of the form H h , for various forms of the heap predicate H .

Lemma `hempty_inv` : $\forall h,$
 $\backslash []\ h \rightarrow$
 $h = \text{Fmap.empty}.$

Proof using. `introv  $M$ . hnf in  $M$ . auto. Qed.`

Lemma `hpure_inv` : $\forall P\ h,$
 $\backslash [P]\ h \rightarrow$
 $P \wedge h = \text{Fmap.empty}.$

Proof using. `introv  $M$ . hnf in  $M$ . autos×. Qed.`

Lemma `hsingle_inv`: $\forall p\ v\ h,$
 $(p \sim\!\!\sim\!> v)\ h \rightarrow$
 $h = \text{Fmap.single}\ p\ v.$

Proof using. `introv  $M$ . hnf in  $M$ . auto. Qed.`

Lemma `hstar_inv` : $\forall H1\ H2\ h,$
 $(H1 \backslash * H2)\ h \rightarrow$
 $\exists h1\ h2, H1\ h1 \wedge H2\ h2 \wedge \text{Fmap.disjoint}\ h1\ h2 \wedge h = h1 \backslash \cup h2.$

Proof using. `introv  $M$ . hnf in  $M$ . eauto. Qed.`

Lemma `hexists_inv` : $\forall A (J:A \rightarrow \text{hprop}) h,$
 $(\backslash \exists\ x, J\ x)\ h \rightarrow$
 $\exists x, J\ x\ h.$

Proof using. `introv  $M$ . hnf in  $M$ . eauto. Qed.`

39.2.7 Proofs of Fundamental Properties

The fundamental properties of Separation Logic operators were stated earlier in this chapter. This section presents their proofs. To avoid name conflicts, we wrap the new statements in a module.

Module `HPPROOFS`.

Extraction of Existential Quantifiers

Let us prove that $(\backslash \exists\ x, J\ x) \backslash * H$ is equivalent to $\backslash \exists\ x, (J\ x \backslash * H)$.

Note that, somewhat confusingly, Rocq displays none of the parentheses. You may want to use `Set Printing Parentheses` to more easily follow through the proof.

Lemma `hstar_hexists` : $\forall A (J:A \rightarrow \text{hprop}) H,$
 $(\exists x, J x) \setminus * H = \exists x, (J x \setminus * H).$

Proof using.

First, we exploit extensionality, using `hprop_eq` or `himpl_antisym`. `intros. apply hprop_eq. split.`

Then we reveal the definitions of `==>`, `hexists` and `hstar`. `{ intros (h1&h2&M1&M2&D&U). destruct M1 as (x&M1).  $\exists x h1 h2.$  }`
`{ intros (x&M). destruct M as (h1&h2&M1&M2&D&U).  $\exists h1 h2.$  splits $\neg$ .  $\exists x.$  }`

Qed.

Neutral of Separating Conjunction

Exercise: 3 stars, standard, especially useful (`hstar_empty_l`) Prove that the empty heap predicate is a neutral element for the star. You will need to exploit the fact that the union with an empty map is the identity, as captured by lemma `Fmap.union_empty_l`.

Check `Fmap.union_empty_l` : forall h, Fmap.empty $\setminus u$ h = h.

Lemma `hstar_empty_l` : $\forall H,$
 $\setminus [] \setminus * H = H.$

Proof using. *Admitted.*

□

Commutativity of Separating Conjunction

We next aim to prove the commutativity of the star operator, i.e., that $H1 \setminus * H2$ is equivalent to $H2 \setminus * H1$. By symmetry, it suffices to prove the entailment in one direction, e.g., $H1 \setminus * H2 \Rightarrow H2 \setminus * H1$. The symmetry argument for any binary operator on heap predicates is captured by the lemma `hprop_op_comm` shown below.

Lemma `hprop_op_comm` : $\forall (op:\text{hprop} \rightarrow \text{hprop} \rightarrow \text{hprop}),$
 $(\forall H1 H2 h, op H1 H2 h \rightarrow op H2 H1 h) \rightarrow$
 $(\forall H1 H2, op H1 H2 = op H2 H1).$

Proof using. `introv M. intros. apply hprop_eq. split; apply M. Qed.`

To prove commutativity of star, we need to exploit the fact that the union of two finite maps with disjoint domains commutes. This fact is captured by a lemma coming from the finite map library.

Check `Fmap.union_comm_of_disjoint` : forall h1 h2, Fmap.disjoint h1 h2 $\rightarrow$ h1 $\setminus u$ h2 = h2 $\setminus u$ h1.

The commutativity result is then proved as follows.

Lemma `hstar_comm` : $\forall H1 H2,$
 $H1 \setminus * H2 = H2 \setminus * H1.$

Proof using.

`apply hprop_op_comm.`

```

  introv (h1&h2&M1&M2&D&U).
  ∃ h2 h1. subst. splits¬.
  { rewrite Fmap.union_comm_of_disjoint; auto. }
Qed.

```

Associativity of Separating Conjunction

Associativity of star is slightly more tedious to derive. We need to exploit the associativity of union on finite maps, as well as lemmas about the disjointness of a map with the result of the union of two maps.

Check Fmap.union_assoc : forall h1 h2 h3, (h1 \u h2) \u h3 = h1 \u (h2 \u h3).

Check Fmap.disjoint_union_eq_l : forall h1 h2 h3, Fmap.disjoint (h2 \u h3) h1 = (Fmap.disjoint h1 h2 /\ Fmap.disjoint h1 h3).

Check Fmap.disjoint_union_eq_r : forall h1 h2 h3, Fmap.disjoint h1 (h2 \u h3) = (Fmap.disjoint h1 h2 /\ Fmap.disjoint h1 h3).

Exercise: 1 star, standard, optional (hstar_assoc) Complete the right-to-left part of the proof of associativity of the star operator.

Lemma hstar_assoc : ∀ *H1 H2 H3*,
 (*H1 * H2*) * *H3* = *H1 * (H2 * H3)*.

Proof using.

```

  intros. apply hprop_eq. split.
  { intros (h'&h3&M1&M2&D&U). destruct M1 as (h1&h2&M3&M4&D'&U').
    subst h'. rewrite Fmap.disjoint_union_eq_l in D.
    ∃ h1 (h2 \u h3). splits.
    { apply M3. }
    { ∃× h2 h3. }
    { rewrite× @Fmap.disjoint_union_eq_r. }
    { rewrite× @Fmap.union_assoc in U. } }

```

Admitted.

□

The lemma showing that `hempty` is a right neutral can be derived from the fact that `hempty` is a left neutral, plus commutativity.

Lemma hstar_hempty_r : ∀ *H*,
H * \[] = *H*.

Proof using.

```

  intros. rewrite hstar_comm. rewrite hstar_hempty_l. auto.

```

Qed.

Exercise: 1 star, standard, especially useful (hstar_comm_assoc) The commutativity and associativity results combine into one result that is sometimes convenient to exploit in proofs.

Lemma `hstar_comm_assoc` : $\forall H1\ H2\ H3,$
 $H1 \setminus * H2 \setminus * H3 = H2 \setminus * H1 \setminus * H3.$

Proof using. *Admitted*.

□

Extracting Pure Facts from Separating Conjunctions

The lemma `hstar_hpure_l` stated below explains how to extract a pure fact from an assertion. It asserts that the proposition $(\setminus[P] \setminus * H)\ h$ is equivalent to $P \wedge H\ h$. This lemma will also be used pervasively in proofs in subsequent chapters.

Exercise: 4 stars, standard, especially useful (`hstar_hpure_l`) Prove that a heap `h` satisfies $\setminus[P] \setminus * H$ if and only if P is true and `h` satisfies H . The proof requires two lemmas on finite maps from *LibSepFmap.v*:

Check `Fmap.union_empty_l` : forall h, `Fmap.empty` $\setminus u\ h = h$. Check `Fmap.disjoint_empty_l` : forall h, `Fmap.disjoint` `Fmap.empty` h.

Note that `auto` can apply *`Fmap.disjoint_empty_l`* automatically.

Hint: begin the proof by applying *`propositional_extensionality`*.

Lemma `hstar_hpure_l` : $\forall P\ H\ h,$
 $(\setminus[P] \setminus * H)\ h = (P \wedge H\ h).$

Proof using. *Admitted*.

□

End HPROPPROOFS.

39.3 More Details

39.3.1 Alternative Definitions for Heap Predicates

We pause here to discuss some alternative, equivalent definitions for the fundamental heap predicates. We write these equivalences using equalities of the form $H1 = H2$. Recall that the lemma `hprop_eq` enables deriving such equalities by invoking predicate extensionality.

The empty heap predicate $\setminus[]$ is equivalent to the pure fact predicate $\setminus[\mathbf{True}]$.

Lemma `hempty_eq_hpure_true` :
 $\setminus[] = \setminus[\mathbf{True}].$

Proof using.

```
unfold hempty, hpure. apply hprop_eq. intros h. iff Hh.
{ auto. }
{ destruct Hh. auto. }
```

Qed.

Thus, `hempty` could be defined in terms of `hpure`, as `hpure True`, written $\setminus[\mathbf{True}]$.

Definition `hempty' : hprop :=`
`\[True].`

The pure fact predicate $\backslash P$ is equivalent to the existential quantification over a proof of P in the empty heap, that is, to the heap predicate $\backslash \exists (p:P), \backslash []$.

Lemma `hpure_eq_hexists_proof :  $\forall P,$`
 `$\backslash P = (\backslash \exists (p:P), \backslash []).$`

Proof using.

```

unfold hempty, hpure, hexists. intros P.
apply hprop_eq. intros h. iff Hh.
{ destruct Hh as (E&p).  $\exists p$ . auto. }
{ destruct Hh as (p&E). auto. }

```

Qed.

Thus, `hpure` could be defined in terms of `hexists` and `hempty`, as `hexists (fun (p:P)  $\Rightarrow$  hempty)`, also written $\backslash \exists (p:P), \backslash []$. In fact, this is how `hpure` is defined in the rest of the course.

Definition `hpure' (P:Prop) : hprop :=`
 `$\backslash \exists (p:P), \backslash []$ .`

It is useful to minimize the number of combinators, both for elegance and to reduce proof effort. We cannot do without `hexists`, thus there remains a choice between considering either `hpure` or `hempty` as primitive, and the other one as derived. The predicate `hempty` is simpler and appears as more fundamental. Hence, in the subsequent chapters, we define `hpure` in terms of `hexists` and `hempty`, as in the definition of `hpure'` shown above. In other words, we assume the definition:

Definition `hpure (P:Prop) : hprop := \exists (p:P), \square`.

39.4 Optional Material

39.4.1 More on the Definition of the Existential Quantifier

The heap predicate $\backslash \exists (n:int), p \rightsquigarrow (\text{val\_int } n)$ characterizes a state that contains a single memory cell, at address p , storing the integer value n , for “some” (unspecified) integer n .

Parameter $(p:\text{loc})$. Check $(\backslash \exists (n:int), p \rightsquigarrow (\text{val\_int } n)) : \text{hprop}$.

The type of $\backslash \exists$, which operates on `hprop`, is very similar to that of $\exists$, which operates on `Prop`. The key difference is that a witness for a $\backslash \exists$ can depend on the current heap, whereas a witness for a $\exists$ cannot.

The notation $\exists x, P$ stands for `ex (fun x  $\Rightarrow$  P)`, where `ex` has the following type:

Check `ex : forall A : Type, (A -> Prop) -> Prop`.

Likewise, $\backslash \exists x, H$ stands for `hexists (fun x  $\Rightarrow$  H)`, where `hexists` has the following type:

Check `hexists : forall A : Type, (A -> hprop) -> hprop`.

The notation for $\backslash \exists$ is directly adapted from that of $\exists$, which supports the quantification an arbitrary number of variables, and is defined in *Rocq.Init.Logic* as follows.

Notation “exists’ $x_1 \dots x_n$, p ” := $(\text{ex } (\text{fun } x_1 \Rightarrow \dots (\text{ex } (\text{fun } x_n \Rightarrow p)) \dots))$ (at level 200, x_1 binder, right associativity, format “’exists’ ’/’ ’ $x_1 \dots x_n$, ’/’ p ”).

Notation “\exists’ $x_1 \dots x_n$, H ” := $(\text{hexists } (\text{fun } x_1 \Rightarrow \dots (\text{hexists } (\text{fun } x_n \Rightarrow H)) \dots))$ (at level 39, x_1 binder, right associativity, H at level 50, format “’\exists’ ’/’ ’ $x_1 \dots x_n$, ’/’ H ”).

39.4.2 Formulation of the Extensionality Axioms

Module EXTENSIONALITY.

To establish extensionality of entailment, we have used the predicate extensionality axiom. In fact, this axiom can be derived by combining the axiom of “functional extensionality” with another one called “propositional extensionality”. The axiom of “propositional extensionality” asserts that two propositions that are logically equivalent, in the sense that they imply each other, can be considered equal.

Axiom *propositional_extensionality* : $\forall (P\ Q:\text{Prop}),$
 $(P \leftrightarrow Q) \rightarrow$
 $P = Q.$

The axiom of “functional extensionality”, as we saw above, asserts that two functions are equal if they provide equal result for every argument.

Axiom *functional_extensionality* : $\forall A\ B\ (f\ g:A \rightarrow B),$
 $(\forall x, f\ x = g\ x) \rightarrow$
 $f = g.$

Exercise: 1 star, standard, especially useful (*predicate_extensionality_derived*)

Use *propositional_extensionality* and *functional_extensionality* to derive *predicate_extensionality*.

Lemma *predicate_extensionality_derived* : $\forall A\ (P\ Q:A \rightarrow \text{Prop}),$
 $(\forall x, P\ x \leftrightarrow Q\ x) \rightarrow$
 $P = Q.$

Proof using. *Admitted*.

□

End EXTENSIONALITY.

39.4.3 Historical Notes

In this chapter, we have defined the Separation Logic operators, in particular the “separating conjunction” operator, as a Rocq predicate. This formalization style is called “shallow embedding”. It has been employed in the first mechanized formalizations of Separation Logic, in Rocq by *Yu, Hamid, and Shao* 2003 (in Bib.v), and in Isabelle/HOL by *Weber* 2004

(in Bib.v). These two formalizations were carried out soon after the invention of Separation Logic, by researchers who were used to mechanizing Hoare logic and had spotted the potential benefit of working with the separating conjunction.

Chapter 40

Library `SLF.Himpl`

40.1 Himpl: Heap Entailment

Set Implicit Arguments.
From *SLF* Require LibSepReference.
From *SLF* Require Export Hprop.

40.2 First Pass

In the previous chapter, we introduced the type `hprop` and the key operators over this type: `\[]`, `\[P]`, `p ~> v`, `H1 \* H2`, `Q1 \*+ H2`, and `\exists x, H`.

In order to state the reasoning rules of Separation Logic, we need to introduce the “entailment relation” on heap predicates. This relation, written $H1 ==> H2$, asserts that any heap that satisfies $H1$ also satisfies $H2$. We also need to extend the entailment relation to postconditions. We write $Q1 ===> Q2$ to asserts that, for any result value v , the entailment $Q1\ v ==> Q2\ v$ holds.

Both entailment relations $==>$ and $===>$ appear in the statement of the “consequence-frame rule”, exploited by the tactic *xapp* for reasoning about every function call. This rule explains how to derive a triple of the form `triple t H Q`, knowing that the term t admits the specification `triple t H1 Q1`. There are two requirements: First, the precondition H must decompose into $H1$, which denotes the precondition expected by the specification, and some remaining part, $H2$. Second, the targeted postcondition Q must be a consequence of the conjunction of $Q1$, which denotes the postcondition asserted by the specification, and $H2$, which again denotes the remaining part, which should not have been affected during the evaluation of the term t .

The formal statement of the frame rule is as follows.

Lemma `triple_conseq_frame` : forall H2 H1 Q1 t H Q, triple t H1 Q1 -> H ==> (H1 * H2) -> (Q1 *+ H2) ===> Q -> triple t H Q.

This chapter presents:

- the formal definition of the entailment relations for heap predicates and for postconditions,
- the lemmas capturing the interaction of entailment with the star operator, and
- the features of the tactics *xsimpl* and *xchange*, which are critically useful for manipulating entailments in practice.

The tactics *xsimpl* and *xchange* were already used in the first two chapters. The implementation of these tactics (from *LibSepSimpl.v*) is out of scope for the present course, but in this chapter we explore their features in more depth through examples.

This chapter exploits one additional TLC tactic for improving scripts: *applies* is an enhanced version of *eapply*. This tactic takes as argument a lemma or hypothesis, plus (optionally) a variable number of arguments, which are fed to the lemma on a first-match basis. In other words, *applies L x y* is equivalent to *eapply (L _ _ .. _ x _ _ .. _ y _ _ ..)* for an appropriate number of underscore symbols.

40.2.1 Definition of Entailment

The *entailment relationship* $H1 \Rightarrow H2$ asserts that any heap *h* that satisfies the heap predicate *H1* also satisfies *H2*.

Definition *himpl* (*H1 H2*:hprop) : Prop :=
 $\forall (h:\text{heap}), H1\ h \rightarrow H2\ h.$

Notation " $H1 \Rightarrow H2$ " := (*himpl H1 H2*) (at level 55).

Entailment is reflexive, transitive, and antisymmetric. It thus forms an order relation on heap predicates.

Lemma *himpl_refl* : $\forall H,$
 $H \Rightarrow H.$

Proof using. *intros h. hnf. auto. Qed.*

Lemma *himpl_trans* : $\forall H2\ H1\ H3,$
 $(H1 \Rightarrow H2) \rightarrow$
 $(H2 \Rightarrow H3) \rightarrow$
 $(H1 \Rightarrow H3).$

Proof using. *introv M1 M2. intros h H1h. eauto. Qed.*

Exercise: 1 star, standard, optional (himpl_antisym) Prove the antisymmetry of entailment result shown below. Hint: use *predicate_extensionality*.

Lemma *himpl_antisym* : $\forall H1\ H2,$
 $(H1 \Rightarrow H2) \rightarrow$
 $(H2 \Rightarrow H1) \rightarrow$
 $H1 = H2.$

Proof using. *Admitted.*

□

40.2.2 Entailment for Postconditions

The entailment relation on postconditions, written $Q1 \implies Q2$, asserts that the heap predicate $Q1 \ v$ entails $Q2 \ v$ for any value v .

Definition `qimpl` ($Q1 \ Q2$:`val $\rightarrow$ hprop`) : `Prop` :=
 $\forall (v:\text{val}), Q1 \ v \implies Q2 \ v$.

Notation " $Q1 \implies Q2$ " := (`qimpl` $Q1 \ Q2$) (at level 55).

In other words, $Q1 \implies Q2$ holds if and only if, for any value v and any heap h , the proposition $Q1 \ v \ h$ implies $Q2 \ v \ h$.

Entailment on postconditions also forms an order relation:

Lemma `qimpl_refl` : $\forall Q,$
 $Q \implies Q$.

Proof using. `intros Q v. apply himpl_refl. Qed.`

Lemma `qimpl_trans` : $\forall Q2 \ Q1 \ Q3,$
 $(Q1 \implies Q2) \rightarrow$
 $(Q2 \implies Q3) \rightarrow$
 $(Q1 \implies Q3).$

Proof using. `introv M1 M2. intros v. eapply himpl_trans; eauto. Qed.`

Exercise: 1 star, standard, optional (qimpl_antisym) Prove the antisymmetry of entailment on postconditions. Hint: exploit *functional_extensionality*.

Lemma `qimpl_antisym` : $\forall Q1 \ Q2,$
 $(Q1 \implies Q2) \rightarrow$
 $(Q2 \implies Q1) \rightarrow$
 $(Q1 = Q2).$

Proof using. *Admitted.*

□

40.2.3 Frame Rule for Entailment

A fundamental property is that the star operator is *monotone* with respect to entailment, meaning that if $H1 \implies H1'$ then $(H1 \ \backslash^* \ H2) \implies (H1' \ \backslash^* \ H2)$. In other words, an arbitrary heap predicate $H2$ can be “framed” on both sides of an entailment.

Viewed the other way around, if we have to prove the entailment relation $(H1 \ \backslash^* \ H2) \implies (H1' \ \backslash^* \ H2)$, we can “cancel out” $H2$ on both sides. In this view, the monotonicity property is a sort of frame rule for the entailment relation.

Due to commutativity of star, it suffices to state only the left version of the monotonicity property.

Parameter *himpl_frame_l* : $\forall H2\ H1\ H1',$
 $H1 \implies H1' \rightarrow$
 $(H1 \setminus * H2) \implies (H1' \setminus * H2).$

40.2.4 Rules for Proving Entailments

The rules for introducing and eliminating pure facts and existential quantifiers in entailments are essential. They are presented next.

The first rule is captured by the lemma *himpl_hstar_hpure_r*. Consider an entailment of the form $H \implies (\setminus [P] \setminus * H')$. To establish this entailment, we must prove that P is true and that H entails H' .

Exercise: 2 stars, standard, especially useful (*himpl_hstar_hpure_r*). Prove the rule *himpl_hstar_hpure_r*.

Hint: recall from *Hprop* the lemma *hstar_hpure_l*, which asserts the equality $(\setminus [P] \setminus * H) \text{ h } (P \wedge H \text{ h})$.

Lemma *himpl_hstar_hpure_r* : $\forall (P:\text{Prop})\ H\ H',$
 $P \rightarrow$
 $(H \implies H') \rightarrow$
 $H \implies (\setminus [P] \setminus * H').$

Proof using. *Admitted*.

□

Reciprocally, consider an entailment of the form $(\setminus [P] \setminus * H) \implies H'$. To establish this entailment, we must prove that H entails H' under the assumption that P is true. The “extraction rule for pure facts in the left of an entailment” captures the property that the pure fact $\setminus [P]$ can be extracted into the Rocq context. It is stated as follows.

Exercise: 2 stars, standard, especially useful (*himpl_hstar_hpure_l*). Prove the rule *himpl_hstar_hpure_l*.

Lemma *himpl_hstar_hpure_l* : $\forall (P:\text{Prop})\ (H\ H':\text{hprop}),$
 $(P \rightarrow H \implies H') \rightarrow$
 $(\setminus [P] \setminus * H) \implies H'.$

Proof using. *Admitted*.

□

Consider an entailment of the form $H \implies (\setminus \exists x, J\ x)$, where x has some type A and J has type $A \rightarrow \text{hprop}$. To establish this entailment, we must exhibit a value for x for which H entails $J\ x$.

Exercise: 2 stars, standard, optional (*himpl_hexists_r*). Prove the rule *himpl_hexists_r*.

Lemma *himpl_hexists_r* : $\forall A\ (x:A)\ H\ J,$

$(H \implies J\ x) \rightarrow$
 $H \implies (\exists x, J\ x).$

Proof using. *Admitted*.

□

Reciprocally, consider an entailment $(\exists x, (J\ x)) \implies H$. To establish this entailment, one has to prove that, whatever the value of x , the predicate $J\ x$ entails H . Indeed the former proposition asserts that if a heap h satisfies $J\ x$ for some x , then h satisfies H' , while the latter asserts that if, for some x , the predicate h satisfies $J\ x$, then h satisfies H' .

The “extraction rule for existentials in the left of an entailment” captures the property that existentials can be extracted into the Rocq context. It is stated as follows. (Observe how the existential quantifier on the left of the entailment becomes a universal quantifier outside of the arrow.)

Exercise: 2 stars, standard, especially useful (`himpl_hexists_l`). Prove the rule `himpl_hexists_l`.

Lemma `himpl_hexists_l` : $\forall (A:\text{Type}) (H:\text{hprop}) (J:A \rightarrow \text{hprop}),$
 $(\forall x, J\ x \implies H) \rightarrow$
 $(\exists x, J\ x) \implies H.$

Proof using. *Admitted*.

□

40.2.5 Extracting Information from Heap Predicates

We next present an example showing how entailments can be used to state lemmas allowing that extract information from particular heap predicates. Concretely, we show that, from a heap predicate of the form $(p \rightsquigarrow v1) \setminus^* (p \rightsquigarrow v2)$, describing two “disjoint” cells that are both “at location p ”, one can extract a contradiction. Indeed, such a state cannot exist.

Lemma `hstar_hsingle_same_loc` : $\forall (p:\text{loc}) (v1\ v2:\text{val}),$
 $(p \rightsquigarrow v1) \setminus^* (p \rightsquigarrow v2) \implies \setminus [\text{False}].$

The proof of this result exploits a result on finite maps. Essentially, the domain of a single singleton map that binds a location p to some value is the singleton set $\{p\}$, so such a singleton map cannot be disjoint from another singleton map that binds the same location p .

Check `disjoint_single_single_same_inv` : `forall (p:loc) (v1 v2:val), Fmap.disjoint (Fmap.single p v1) (Fmap.single p v2) -> False`.

Using this lemma, we can prove `hstar_hsingle_same_loc` by unfolding the definition of `hstar` to reveal the contradiction on the disjointness assumption.

Proof using.

`intros. unfold hsingle. intros h (h1&h2&E1&E2&D&E). false.`
`subst. eapply Fmap.disjoint_single_single_same_inv. eapply D.`

`Qed.`

More generally, a heap predicate of the form $H \setminus^* H$ is usually suspicious in Separation Logic. In the simple variant of Separation Logic that we consider in this course, there are only three typical situations where $H \setminus^* H$ makes sense:

- (1) if H is the empty heap predicate $\setminus[]$,
- (2) if H is a pure heap predicate of the form $\setminus[P]$, or
- (3) if H has the form $\setminus\exists (H0:hprop), H0$, which will be written $\setminus GC$ in chapter [Affine](#).

Rules for Naming Heaps

We write $= h$ as a shorthand for $\text{fun } h' \Rightarrow h' = h$, that is, the heap predicate that only accepts heaps exactly equal to h .

Exercise: 3 stars, standard, optional (hexists_named_eq) Prove that a heap predicate H is equivalent to $\setminus\exists h, \setminus[H h] \setminus^* (= h)$.

Hint: In one direction, use `hexists_intro` and `hstar_hpure_l`. In the other, use the extraction rules `himpl_hexists_l` and `himpl_hstar_hpure_l`.

Lemma `hexists_named_eq` : $\forall H,$
 $H = (\setminus\exists h, \setminus[H h] \setminus^* (= h)).$

Proof using. *Admitted.*

□

40.2.6 The *xsimpl* Tactic

Simplifying entailments by hand is extremely tedious. Fortunately, it can be automated using a specialized Rocq tactic. This tactic, called *xsimpl*, applies to an entailment and implements a 3-step process:

- extract pure facts and existential quantifiers from the LHS;
- instantiate existential quantifiers from the RHS, using either Rocq unification variables or user-provided hints (details are explained further on);
- cancel out predicates that occur on both the LHS and RHS; certain Rocq unification variables may be instantiated during that process.

These steps are detailed and illustrated in the subsections that follow.

To make the *xsimpl* tactic available, we import the `LibSepReference`, over a scope delimited by the module named `XSIMPLTACTIC`. When doing so, the Separation Logic operators and entailments refer to the definitions from `LibSepReference`. These definitions are essentially the same as those presented in `Hprop` and earlier in the present chapter.

Module `XSIMPLTACTIC`.

Import `LIBSEPREFERENCE`.

xsimpl Extracts Pure Facts and Quantifiers in LHS

The first feature of *xsimpl* is its ability to extract the pure facts and existential quantifiers from the left-hand side out into the Rocq context.

In the example below, the pure fact $n > 0$ appears in the LHS. After calling *xsimpl*, this pure fact is turned into a hypothesis, which may be introduced with a name into the Rocq context.

```
Lemma xsimpl_demo_lhs_hpure :  $\forall H1 H2 H3 H4 (n:\text{int}),$   
   $H1 \wedge H2 \wedge [n > 0] \wedge H3 \implies H4.$ 
```

Proof using.

```
  intros. xsimpl. intros Hn.
```

Abort.

In case the LHS includes a contradiction, such as the pure fact **False**, the goal gets solved immediately by *xsimpl*.

```
Lemma xsimpl_demo_lhs_hpure :  $\forall H1 H2 H3 H4,$   
   $H1 \wedge H2 \wedge [\text{False}] \wedge H3 \implies H4.$ 
```

Proof using.

```
  intros. xsimpl.
```

Qed.

The *xsimpl* tactic also extracts existential quantifiers from the LHS. It turns them into universally quantified variables outside of the entailment relation.

```
Lemma xsimpl_demo_lhs_hexists :  $\forall H1 H2 H3 H4 (p:\text{loc}),$   
   $H1 \wedge \exists (n:\text{int}), (p \rightsquigarrow n \wedge H2) \wedge H3 \implies H4.$ 
```

Proof using.

```
  intros. xsimpl. intros n.
```

Abort.

A single call to *xsimpl* extracts *all* the pure facts and quantifiers from the LHS.

```
Lemma xsimpl_demo_lhs_several :  $\forall H1 H2 H3 H4 (p q:\text{loc}),$   
   $H1 \wedge \exists (n:\text{int}), (p \rightsquigarrow n \wedge [n > 0] \wedge H2) \wedge [p \neq q] \wedge H3 \implies H4.$ 
```

Proof using.

```
  intros. xsimpl. intros n Hn Hp.
```

Abort.

xsimpl Cancels Out Heap Predicates from LHS and RHS

The second feature of *xsimpl* is its ability to cancel out similar heap predicates that occur on both sides of an entailment.

In the example below, $H2$ occurs on both sides, so it is canceled out by *xsimpl*.

```
Lemma xsimpl_demo_cancel_one :  $\forall H1 H2 H3 H4 H5 H6 H7,$   
   $H1 \wedge H2 \wedge H3 \wedge H4 \implies H5 \wedge H6 \wedge H2 \wedge H7.$ 
```

Proof using.


```

intros. xsimpl.
Abort.

```

xsimpl actually cancels out *all* the heap predicates that it can spot appearing on both sides. In the example below, *H2*, *H3*, and *H4* are canceled out.

```

Lemma xsimpl_demo_cancel_many :  $\forall H1 H2 H3 H4 H5,$ 
   $H1 \wedge H2 \wedge H3 \wedge H4 \implies H4 \wedge H3 \wedge H5 \wedge H2.$ 

```

Proof using.

```

intros. xsimpl.
Abort.

```

If all the pieces of heap predicate get canceled out, the remaining proof obligation is $\llbracket \implies \rrbracket$. In this case, *xsimpl* automatically solves the goal by invoking the reflexivity of entailment.

```

Lemma xsimpl_demo_cancel_all :  $\forall H1 H2 H3 H4,$ 
   $H1 \wedge H2 \wedge H3 \wedge H4 \implies H4 \wedge H3 \wedge H1 \wedge H2.$ 

```

Proof using.

```

intros. xsimpl.
Qed.

```

xsimpl Instantiates Pure Facts and Quantifiers in the RHS

The third feature of *xsimpl* is its ability to extract pure facts from the RHS as separate subgoals and to instantiate existential quantifiers from the RHS.

Let us first illustrate how it deals with pure facts. In the example below, the fact $n > 0$ gets spawned in a separated subgoal.

```

Lemma xsimpl_demo_rhs_hpure :  $\forall H1 H2 H3 (n:int),$ 
   $H1 \implies H2 \wedge [n > 0] \wedge H3.$ 

```

Proof using.

```

intros. xsimpl.
Abort.

```

When it encounters an existential quantifier in the RHS, the *xsimpl* tactic introduces a unification variable denoted by a question mark, that is, an “evar”, in Rocq terminology. Here, *xsimpl* turns $\exists n, .. p \rightsquigarrow n ..$ into $.. p \rightsquigarrow ?Goal ..$, where *?Goal* is the evar.

```

Lemma xsimpl_demo_rhs_hexists :  $\forall H1 H2 H3 H4 (p:loc),$ 
   $H1 \implies H2 \wedge \exists (n:int), (p \rightsquigarrow n \wedge H3) \wedge H4.$ 

```

Proof using.

```

intros. xsimpl.

```

The output of the command *Show Existentials* shows *Existential* 1, named *?Goal*, of type *int*. This “evar” corresponds to the value assigned to the existentially quantified variable *n*.
 Abort.

The “evar” often gets subsequently instantiated as a result of a cancellation step. For example, in the example below, *xsimpl* instantiates the existentially quantified variable *n* as

?Goal, then cancels out $p \sim\sim > ?Goal$ from the LHS against $p \sim\sim > 3$ on the right-hand-side, thereby unifying ?Goal with 3. As a result, the “evvar” ?Goal is never visible.

```
Lemma xsimpl_demo_rhs_hexists_unify : ∀ H1 H2 H3 H4 (p:loc),
  H1 \* (p \sim\sim > 3) ==> H2 \* \exists (n:int), (p \sim\sim > n \* H3) \* H4.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

The instantiation of “evvars” can only be observed if there is another occurrence of the same variable in the entailment. In the next example, which refines the previous one, observe how the assertion $n > 0$ that appears in the LHS gets turned into the proof obligation $3 > 0$. This results from n being unified with 3 when $p \sim\sim > n$ is cancelled against $p \sim\sim > 3$.

```
Lemma xsimpl_demo_rhs_hexists_unify_view : ∀ H1 H2 H4 (p:loc),
  H1 \* (p \sim\sim > 3) ==> H2 \* \exists (n:int), (p \sim\sim > n \* \[n > 0]) \* H4.
```

Proof using.

```
  intros. xsimpl.
```

Abort.

In practice, in most cases, *xsimpl* would instantiate existentials on the left-hand side, then discharge what remains of the entailment. Here are two typical examples.

```
Lemma xsimpl_demo_rhs_exists_solved_1 : ∀ (p:loc),
  p \sim\sim > 3 ==>
```

```
  \exists (n:int), p \sim\sim > n.
```

Proof using. *xsimpl*. Qed.

```
Lemma xsimpl_demo_rhs_exists_solved_2 : ∀ (p q:loc),
```

```
  p \sim\sim > 3 \* q \sim\sim > 3 ==>
```

```
  \exists (n:int), p \sim\sim > n \* q \sim\sim > n.
```

Proof using. *xsimpl*. Qed.

In certain situations, it may be desirable to provide an explicit value for instantiating an existential quantifier that occurs in the RHS. The *xsimpl* tactic accepts arguments, which will be used to instantiate the existentials (on a first-match basis). The syntax is *xsimpl* v1 .. vn.

```
Lemma xsimpl_demo_rhs_hints : ∀ H1 (p q:loc),
  H1 ==> \exists (n m:int), (p \sim\sim > n \* q \sim\sim > m).
```

Proof using.

```
  intros. xsimpl 8 9.
```

Abort.

If two existential quantifiers quantify variables of the same type, it is possible to provide a value for only the second quantifier by passing as first argument to *xsimpl* the special value `--`. The following example shows how, on LHS of the form $\exists n\ m, \dots$, the tactic *xsimpl* `--` 9 instantiates m with 9 while leaving n as an unresolved evvar.

```
Lemma xsimpl_demo_rhs_hints_evar : ∀ H1 (p q:loc),
```

```

    H1 ==> \∃ (n m:int), (p ~~> n \* q ~~> m).
Proof using.
  intros. xsimpl -- 9.
Abort.

```

xsimpl on Entailments Between Postconditions

The tactic *xsimpl* also applies on goals of the form $Q1 ==> Q2$.

For such goals, it unfolds the definition of $==>$ to reveal an entailment of the form $==>$, then invokes the *xsimpl* tactic.

```

Lemma qimpl_example_1 : ∀ (Q1 Q2:val→hprop) (H2 H3:hprop),
  Q1 \*+ H2 ==> Q2 \*+ H2 \*+ H3.
Proof using. intros. xsimpl. intros r. Abort.

```

Example: Entailment Proofs using *xpull* and *xsimpl*

The *xpull* tactic is a restricted version of *xsimpl*: it operates on the left-hand side of the entailment, but leaves the right-hand side untouched. In other words, *xpull* performs only the first half of the work that *xsimpl* would do.

```

Lemma xpull_example_1 : ∀ (p:loc),
  \∃ (n:int), p ~~> n ==>
  \∃ (m:int), p ~~> (m + 1).
Proof using.
  intros. xpull.
Abort.

```

```

Lemma xpull_example_2 : ∀ (H:hprop),
  \[False] ==> H.
Proof using. xpull. Qed.

```

The next example illustrates an interesting situation where *xpull* is required because a plain call to *xsimpl* leads to an awkward goal. This examples involves a nontrivial instantiation of an existential quantifier on the right-hand side of the entailment. However, we cannot provide an explicit hint to *xsimpl* before extracting an existential quantifier that appears on the left-hand side of the entailment. Here is the example.

```

Lemma xpull_example_3 : ∀ (p:loc),
  (\∃ (n:int), p ~~> n) ==>
  (\∃ (m:int), p ~~> (m + 1)).
Proof using.
  intros.
  xpull. intros n. xsimpl (n-1). math.
Qed.

```

40.2.7 The *xchange* Tactic

The tactic *xchange* is to entailment what *rewrite* is to equality.

Assume an entailment goal of the form $H1 \setminus * H2 \setminus * H3 ==> H4$. Assume an entailment assumption M , say $H2 ==> H2'$. Then *xchange* M turns the goal into $H1 \setminus * H2' \setminus * H3 ==> H4$, effectively replacing $H2$ with $H2'$.

```
Lemma xchange_demo_base : ∀ H1 H2 H2' H3 H4,
  H2 ==> H2' →
  H1 \* H2 \* H3 ==> H4.
```

Proof using.

```
  intro M. xchange M.
```

Abort.

The *xchange* tactic can also take an equality as an argument; *xchange* M exploits the left-to-right direction of an equality M , whereas *xchange* $\leftarrow M$ exploits the right-to-left direction.

```
Lemma xchange_demo_eq : ∀ H1 H2 H3 H4 H5,
  H1 \* H3 = H5 →
  H1 \* H2 \* H3 ==> H4.
```

Proof using.

```
  intro M. xchange M.
```

```
  xchange ← M.
```

Abort.

The tactic *xchange* M will accept a lemma or hypothesis M with universal quantifiers, as long as its conclusion is an equality or an entailment. In such case, *xchange* M instantiates M before attempting to perform a replacement.

```
Lemma xchange_demo_inst : ∀ H1 (J J':int→hprop) H3 H4,
  (∀ n, J n = J' (n+1)) →
  H1 \* J 3 \* H3 ==> H4.
```

Proof using.

```
  intro M. xchange M.
```

Abort.

40.2.8 Identifying True and False Entailments

Quiz: For each entailment below, indicate (without a Rocq proof) whether it is true or false. Solutions appear further on.

Module ENTAILMENTQUIZ.

```
Implicit Types p q : loc.
```

```
Implicit Types n m : int.
```

```
Parameter case_study_1 : ∀ p q,
  p ~~> 3 \* q ~~> 4
```

```

==> q ~> 4 \* p ~> 3.
Parameter case_study_2 : ∀ p q,
  p ~> 3
==> q ~> 4 \* p ~> 3.
Parameter case_study_3 : ∀ p q,
  q ~> 4 \* p ~> 3
==> p ~> 3.
Parameter case_study_4 : ∀ p q,
  \[False] \* p ~> 3
==> p ~> 4 \* q ~> 4.
Parameter case_study_5 : ∀ p q,
  p ~> 3 \* q ~> 4
==> \[False].
Parameter case_study_6 : ∀ p,
  p ~> 3 \* p ~> 4
==> \[False].
Parameter case_study_7 : ∀ p,
  p ~> 3 \* p ~> 3
==> \[False].
Parameter case_study_8 : ∀ p,
  p ~> 3
==> \∃ n, p ~> n.
Parameter case_study_9 : ∀ p,
  ∃ n, p ~> n
==> p ~> 3.
Parameter case_study_10 : ∀ p,
  \∃ n, p ~> n \* \[n > 0]
==> \∃ n, \[n > 1] \* p ~> (n-1).
Parameter case_study_11 : ∀ p q,
  p ~> 3 \* q ~> 3
==> \∃ n, p ~> n \* q ~> n.
Parameter case_study_12 : ∀ p n,
  p ~> n \* \[n > 0] \* \[n < 0]
==> p ~> n \* p ~> n.

```

End ENTAILMENTQUIZ.

Module ENTAILMENTQUIZANSWERS.

The answers to the quiz are as follows.

1. True, by commutativity.

2. False, because one cell does not entail two cells.
3. False, because two cells do not entail one cell.
4. True, because $\backslash \text{False}$ entails anything.
5. False, because a satisfiable heap predicate does not entail $\backslash \text{False}$.
6. True, because a cell cannot be starred with itself.
7. True, because a cell cannot be starred with itself.
8. True, by instantiating n with 3.
9. False, because n could be something else than 3.
10. True, by instantiating n in RHS with $n+1$ for the n of the LHS.
11. True, by instantiating n with 3.
12. True, because it is equivalent to $\backslash [\text{False}] ==> \backslash [\text{False}]$.

Proofs for the true results appear below, using the *xsimpl* tactic.

Implicit Types $p\ q : \text{loc}$.

Implicit Types $n\ m : \text{int}$.

Lemma case_study_1 : $\forall\ p\ q,$
 $p \rightsquigarrow 3 \backslash * q \rightsquigarrow 4$
 $==> q \rightsquigarrow 4 \backslash * p \rightsquigarrow 3$.

Proof using. *xsimpl*. Qed.

Lemma case_study_4 : $\forall\ p\ q,$
 $\backslash [\text{False}] \backslash * p \rightsquigarrow 3$
 $==> p \rightsquigarrow 4 \backslash * q \rightsquigarrow 4$.

Proof using. *xsimpl*. Qed.

Lemma case_study_6 : $\forall\ p,$
 $p \rightsquigarrow 3 \backslash * p \rightsquigarrow 4$
 $==> \backslash [\text{False}]$.

Proof using. intros. *xchange* (hstar_hsingle_same_loc p). Qed.

Lemma case_study_7 : $\forall\ p,$
 $p \rightsquigarrow 3 \backslash * p \rightsquigarrow 3$
 $==> \backslash [\text{False}]$.

Proof using. intros. *xchange* (hstar_hsingle_same_loc p). Qed.

Lemma case_study_8 : $\forall\ p,$
 $p \rightsquigarrow 3$
 $==> \backslash \exists\ n, p \rightsquigarrow n$.

Proof using. *xsimpl*. Qed.

Lemma case_study_10 : $\forall\ p,$
 $\backslash \exists\ n, p \rightsquigarrow n \backslash * \backslash [n > 0]$
 $==> \backslash \exists\ n, \backslash [n > 1] \backslash * p \rightsquigarrow (n-1)$.

Proof using.

intros. *xpull*. intros $n\ Hn$. *xsimpl* ($n+1$).
math. math.

Qed.

```
Lemma case_study_11 :  $\forall p q,$   
   $p \sim\sim > 3 \setminus * q \sim\sim > 3$   
   $\implies \exists n, p \sim\sim > n \setminus * q \sim\sim > n.$ 
```

Proof using. *xsimpl*. Qed.

```
Lemma case_study_12 :  $\forall p n,$   
   $p \sim\sim > n \setminus * \setminus [n > 0] \setminus * \setminus [n < 0] \implies p \sim\sim > n \setminus * p \sim\sim > n.$   
Proof using. intros. xsimpl. intros Hn1 Hn2. false. math. Qed.
```

End ENTAILMENTQUIZANSWERS.

40.3 More Details

40.3.1 Proving Entailments by Hand

Module ENTAILMENTPROOFS.

Implicit Types *p* : loc.

Implicit Types *n* : int.

Proving an entailment by hand is generally a tedious task. This is why most frameworks based on Separation Logic include an automated tactic for simplifying entailments. In this course, the relevant tactic is named *xsimpl*. In order to fully appreciate what a simplification tactic provides and better understand how it works internally, it is very useful to complete a few entailment proofs by hand.

Exercise: 3 stars, standard, optional (himpl_example_1) Prove the example entailment below. Hint: exploit *hstar_comm*, *hstar_assoc*, or *hstar_comm_assoc* which combines the two, and exploit *himpl_frame_l* or *himpl_frame_r* to cancel out matching pieces.

```
Lemma himpl_example_1 :  $\forall p1 p2 p3 p4,$   
   $p1 \sim\sim > 6 \setminus * p2 \sim\sim > 7 \setminus * p3 \sim\sim > 8 \setminus * p4 \sim\sim > 9$   
   $\implies p4 \sim\sim > 9 \setminus * p3 \sim\sim > 8 \setminus * p2 \sim\sim > 7 \setminus * p1 \sim\sim > 6.$ 
```

Proof using. *Admitted*.

□

Exercise: 2 stars, standard, optional (himpl_example_2) Prove the example entailment below. Hint: use *himpl_hstar_hpure_l* to extract pure facts, once they appear at the head of the left-hand side of the entailment. For arithmetic inequalities, use the *math* tactic.

```
Lemma himpl_example_2 :  $\forall p1 p2 p3 n,$   
   $p1 \sim\sim > 6 \setminus * \setminus [n > 0] \setminus * p2 \sim\sim > 7 \setminus * \setminus [n < 0]$   
   $\implies p3 \sim\sim > 8.$ 
```

Proof using. *Admitted*.

□

Exercise: 2 stars, standard, optional (himpl_example_3) Prove the example entailment below. Hint: use `himpl_hexists_r`, `himpl_hexists_l`, `himpl_hstar_hpure_l` and/or `himpl_hstar_hpure_r` to deal with pure facts and quantifiers. You may find the TLC tactic *math_rewrite* ($n+1-1 = n$) helpful here.

Lemma `himpl_example_3` : $\forall p,$
 $\backslash \exists n, p \sim\sim n \backslash * \backslash [n > 0]$
 $\impl \backslash \exists n, \backslash [n > 1] \backslash * p \sim\sim (n-1).$

Proof using. *Admitted*.

□

End `ENTAILMENTPROOFS`.

40.3.2 Inside the *xchange* Tactic

How does the *xchange* tactic work? Consider a goal of the form $H \impl H'$ and assume *xchange* is invoked with an hypothesis of type $H1 \impl H1'$ as argument. It attempts to decompose H as the star of $H1$ and the rest of H , call it $H2$. If it succeeds, then the goal $H \impl H'$ can be rewritten as $H1 \backslash * H2 \impl H'$. To exploit the hypothesis $H1 \impl H1'$, the tactic should replace the goal with the entailment $H1' \backslash * H2 \impl H'$. The lemma below mimics this piece of reasoning.

Exercise: 2 stars, standard, especially useful (xchange_lemma) Prove the following lemma without using *xchange*.

Lemma `xchange_lemma` : $\forall H1\ H1'\ H\ H'\ H2,$
 $H1 \impl H1' \rightarrow$
 $H \impl H1 \backslash * H2 \rightarrow$
 $H1' \backslash * H2 \impl H' \rightarrow$
 $H \impl H'.$

Proof using. *Admitted*.

□

End `XSIMPLTACTIC`.

40.4 Optional Material

40.4.1 Proofs of Rules for Entailment

We next show the details of the proofs establishing some of the fundamental properties of Separation Logic operators. All these results must be proved without help of *xsimpl*, because

the implementation of *xsimpl* itself depends on these properties.

We begin with the frame property, which is the simplest to prove.

Module ENTAILMENTRULESPROOFS.

Exercise: 1 star, standard, especially useful (himpl_frame_l) Prove the frame property for entailment. Hint: unfold the definition of *hstar*.

Lemma *himpl_frame_l* : $\forall H2\ H1\ H1',$
 $H1 \implies H1' \rightarrow$
 $(H1 \setminus * H2) \implies (H1' \setminus * H2).$

Proof using. *Admitted*.

□

The lemma *himpl_frame_l* admits two useful corollaries, presented next.

Exercise: 1 star, standard, especially useful (himpl_frame_r) Prove *himpl_frame_r*, which is the symmetric of *himpl_frame_l*.

Lemma *himpl_frame_r* : $\forall H1\ H2\ H2',$
 $H2 \implies H2' \rightarrow$
 $(H1 \setminus * H2) \implies (H1 \setminus * H2').$

Proof using. *Admitted*.

□

Exercise: 1 star, standard, especially useful (himpl_frame_lr) The monotonicity property of the star operator w.r.t. entailment can also be stated in a symmetric fashion, as shown next. Prove this result. Hint: exploit the transitivity of entailment (*himpl_trans*) and the asymmetric monotonicity results from the two prior exercises.

Lemma *himpl_frame_lr* : $\forall H1\ H1'\ H2\ H2',$
 $H1 \implies H1' \rightarrow$
 $H2 \implies H2' \rightarrow$
 $(H1 \setminus * H2) \implies (H1' \setminus * H2').$

Proof using. *Admitted*.

□

End ENTAILMENTRULESPROOFS.

40.4.2 Historical Notes

Chapter 41

Library SLF.Triples

41.1 Triples: Structural Reasoning Rules

Set Implicit Arguments.

The file *LibSepReference.v* contains definitions that are essentially the same as those from *Hprop.v* and *Himpl.v*, but with one key difference: `LibSepReference` makes the definition of Separation Logic operators opaque. This chapter and the following ones import *LibSepReference.v* instead of *Hprop.v* and *Himpl.v*.

As a result, we cannot unfold the definition of `hstar`, `hpure`, etc. any more. To reason, we must instead use the introduction and elimination lemmas, such as `hstar_intro` and `hstar_elim`. These lemmas enforce abstraction, ensuring that the proofs do not depend on the particular choice of the definitions used for constructing Separation Logic. For example, we should not depend on whether $\backslash[P]$ is internally implemented as `fun h => h = empty & P` or as $\backslash\exists (p:P), \backslash[]$.

From *SLF* Require Export `LibSepReference`.

From *SLF* Require `Basic`.

41.2 First Pass

In chapter *Hprop*, we formalized heap predicates, of type `hprop`, and several operators on them: $\backslash[], \backslash[P], p \sim\sim> v, H1 \backslash^* H2, Q1 \backslash^{*+} H2$, and $\backslash\exists x, H$. In chapter *Himpl*, we formalized the heap entailment relations $H1 ==> H2$ and $Q1 ===> Q2$. The goal of the present chapter and the next one is to formalize the definition of triples, written `triple t H Q`, and the reasoning rules of Separation Logic.

More precisely, the present chapter focuses on the definition of triples and the proofs of the “structural rules”, which are essential in particular for implementing the tactics *xpull* and *xapp*. The structural rules include the “frame rule”, which lies at the heart of Separation Logic, as well as the “consequence rule” and two “extraction rules”, which admit similar

statements as in Hoare Logic. The reasoning rules concerned with term constructs are discussed in the next chapter (Rules).

This chapter starts with a formalization of the syntax and the semantics of the programming language we are using for examples. The semantics is axiomatized using a variant of big-step semantics called “omni-big-step semantics”. This presentation is well suited for capturing safety and termination in nondeterministic programming languages. Near the end of the chapter, we explain how an omni-big-step semantics can be formally related to a standard small-step semantics.

This chapter exploits a few additional TLC tactics to enable concise proofs. Recall from chapter [Himpl](#) that *applies* is an enhanced version of *eapply*, which takes as argument a lemma or hypothesis, followed by a variable number of arguments, which are fed to the lemma on a first-match basis. For example, *applies* *L* *x* *y* is equivalent to *eapply* (*L* _ _ .. _ *x* _ _ .. _ _ *y* _ _ ..) for an appropriate number of underscore symbols.

Likewise, the tactics *lets*, *forwards*, and *specializes* can be used to instantiate a lemma or an existing hypothesis:

- *lets* *H*: *L* *x* *y* introduces an hypothesis named *H*, whose statement is the specialization of the lemma *L* on the arguments *x* and *y*. It is equivalent to *generalize* (*L* _ _ .. _ *x* _ _ .. _ _ *y*); *intros* *H*, for the appropriate number of underscore symbols.
- *forwards* *H*: *L* *x* *y* is very similar to *lets*, except that it attempts to instantiate all arguments of *L*. It is equivalent to *generalize* (*L* _ _ .. _ *x* _ _ .. _ _ *y* _ _ .. _ _); *intros* *H*.
- *specializes* *H* *x* *y* specializes an existing hypothesis *H* in place. It is equivalent to *lets* *H2*: *H* *x* *y*; *clear* *H*; *rename* *H2* into *H*.

The reader will find many occurrences of these tactics in proofs and solutions. However, exploiting these tactics is never required for solving exercises.

41.2.1 Formalized Syntax of the Programming Language

In what follows, we present the formalization of the syntax of the programming language. This material corresponds to a large subset of the material from the file [LibSepReference](#). Further on in this file, we will want to refer to the definitions from [LibSepReference](#), to avoid having to copy-paste all the construction of Separation Logic operators, which depend on the type *heap* and therefore depend on the type *val* associated with program values. Hence, we wrap the definitions of syntax presented below inside a module.

Module SYNTAX.

Values and Terms

The syntax of our example programming language is described using an *abstract syntax tree*. Its presentation distinguishes between closed values (i.e., values with no free variables inside)

and terms. This presentation simplifies the definition and evaluation of the substitution function. Indeed, since values are always closed, the substitution function never needs to traverse through values.

The grammar for values includes the unit value, booleans, integers, locations, functions, recursive functions, and primitive operations. For example, `val_int 3` denotes the integer value 3, while `val_fun x t` denotes the function `fun x => t`, and `val_fix f x t` denotes the function `fix f x => t`, which is written `let rec f x = t in f` in OCaml syntax.

For conciseness, we include just a few primitive operations: `val_ref`, `val_get`, `val_set`, and `val_free` for manipulating the mutable state, `val_add` to illustrate a simple arithmetic operation, `val_div` to illustrate a partial operation, and `val_rand` to illustrate nondeterminism.

```
Inductive val : Type :=
| val_unit : val
| val_bool : bool -> val
| val_int : int -> val
| val_loc : loc -> val
| val_fun : var -> trm -> val
| val_fix : var -> var -> trm -> val
| val_ref : val
| val_get : val
| val_set : val
| val_free : val
| val_add : val
| val_div : val
| val_rand : val
```

The grammar for terms includes values, variables, function definitions, recursive function definitions, function applications, sequences, let-bindings, and conditionals.

```
with trm : Type :=
| trm_val : val -> trm
| trm_var : var -> trm
| trm_fun : var -> trm -> trm
| trm_fix : var -> var -> trm -> trm
| trm_app : trm -> trm -> trm
| trm_seq : trm -> trm -> trm
| trm_let : var -> trm -> trm -> trm
| trm_if : trm -> trm -> trm -> trm.
```

Note that `trm_fun` and `trm_fix` denote functions that may feature free variables, unlike `val_fun` and `val_fix`, which denote closed values. More concretely, in the term `trm_fun x t`, the term `t` may refer to the variable `x` or to other free variables. In contrast, in the term `val_fun x t`, the only free variable that `t` may have is `x`.

We will see, below, evaluation rules asserting that `trm_fun x t` reduces, in the empty context, to `val_fun x t`. Likewise, `trm_fix f x t` reduces, in the empty context, to `val_fix f x t`.

At a high level, the term `val_fun x t` may be thought of as the term `trm_fun x t` augmented with a proof that this term `trm_fun x t` is a closed term. The syntax that we consider therefore embeds “proofs that values are closed”. Thanks to this presentation, the substitution function `subst`, which will be presented shortly afterwards, need not recurse through values. Moreover, we never need to manipulate proofs that specific terms are closed.

Functions of several arguments are encoded as nested functions (curried functions). For example, the syntax `fun x1 x2 x3 => t` used to define a top-level function in the chapters [Basic](#) and [Repr](#) is encoded as `val_fun x1 (trm_fun x2 (trm_fun x3 t))`. Observe how the outer constructor asserts that the whole function is a closed value, whereas the inner functions are terms. In particular `t` may refer to `x1` and `x2`, which are free variables for the term `trm_fun x3 t`.

Likewise, a closed recursive function `fix f x1 x2 x3 => t` is encoded as `val_fix f x1 (trm_fun x2 (trm_fun x3 t))`. Here only the outermost function is recursive (and closed).

In the rest of the course, we will focus on functions with a single argument. The treatment of n-ary functions is the topic of a yet-to-be-released chapter.

State

The language we consider is imperative, with primitive functions for manipulating the state. Thus, the statements of the evaluation rules involve memory states.

Recall from chapter [Hprop](#) that a state is represented as a finite map from location to values. Finite maps are presented using the type `fmap`. Details of the construction of finite maps may be found in the file [LibSepFmap.v](#). (The construction is beyond the scope of this course, because it leverages dependent pairs to realize the type of finite maps.)

Definition `heap : Type := fmap loc val`.

For technical reasons related to the internal representation of finite maps, to enable reading in a state, we need to justify that the grammar of values is inhabited. This property is captured by the following command, whose details are not relevant for understanding the rest of the chapter.

Global Instance `Inhab_val : Inhab val`.

Proof using. `apply (Inhab_of_val val_unit)`. `Qed`.

Substitution

The semantics of the evaluation of a program function is described by means of a substitution function, written `subst y w t`, which replaces all occurrences of a variable `y` with a value `w` inside a term `t`.

The substitution function is always the identity on values, because our language only considers closed values. In other words, we define `subst y w (trm_val v) = (trm_val v)`.

When it reaches a variable, substitution performs a comparison between two variables. To that end, it exploits the comparison function `var_eq x y`, which produces a boolean value indicating whether `x` and `y` denote the same variable.

The remainder of this subsection describes the implementation of the substitution function `subst`. This operation traverses all other language constructs in a structural manner. It takes care of avoiding “variable capture” when traversing binders: `subst y w t` does not recurse below the scope of binders whose bound name is equal to `y`. For example, the result of `subst y w (trm_let x t1 t2)` is `trm_let x (subst y w t1) (if var_eq x y then t2 else (subst y w t2))`.

The auxiliary function `if_y_eq`, which appears in the definition of `subst`, helps factorizing the comparisons required to prevent variable capture.

```
Fixpoint subst (y:var) (w:val) (t:trm) : trm :=
  let aux t := subst y w t in
  let if_y_eq x t1 t2 := if var_eq x y then t1 else t2 in
  match t with
  | trm_val v => trm_val v
  | trm_var x => if_y_eq x (trm_val w) t
  | trm_fun x t1 => trm_fun x (if_y_eq x t1 (aux t1))
  | trm_fix f x t1 => trm_fix f x (if_y_eq f t1 (if_y_eq x t1 (aux t1)))
  | trm_app t1 t2 => trm_app (aux t1) (aux t2)
  | trm_seq t1 t2 => trm_seq (aux t1) (aux t2)
  | trm_let x t1 t2 => trm_let x (aux t1) (if_y_eq x t2 (aux t2))
  | trm_if t0 t1 t2 => trm_if (aux t0) (aux t1) (aux t2)
  end.
```

Implicit Types and Coercions

To improve the readability of the evaluation rules below, we take advantage of Rocq’s *implicit types* and *coercions*.

First off, we give several Implicit Types declarations. For example, the first command indicates that variables whose name begins with the letter ‘b’ are, by default, of type `bool`.

```
Implicit Types b : bool.
Implicit Types v r : val.
Implicit Types t : trm.
```

Next, we give a collection of coercions. Coercions correspond to implicit function calls, which aim to make statements more concise. For example, `val_loc` is declared as a coercion, so that a location `p` of type `loc` can be viewed as the value `val_loc p` where an expression of type `val` is expected. Likewise, a boolean `b` may be viewed as the value `val_bool b`, and an integer `n` may be viewed as the value `val_int n`.

```
Coercion val_loc : loc >-> val.
Coercion val_bool : bool >-> val.
Coercion val_int : Z >-> val.
```

The constructor `trm_val` is also declared as a coercion. Thus, instead of writing `trm_val v` at a place where a term is expected, we can write just `v`.

Coercion `trm_val : val >-> trm`.

The constructor `trm_app` is declared as a “Funclass” coercion. This bit of magic enables us to write `t1 t2` as a shorthand for `trm_app t1 t2`. The idea of associating `trm_app` as the “Funclass” coercion for the type `trm` is that if a term `t1` of type `trm` is applied like a function to an argument, then `t1` should be interpreted as `trm_app t1`.

Coercion `trm_app : trm >-> Funclass`.

The “Funclass” coercion for `trm_app` can be iterated. The expression `t1 t2 t3` is parsed by Rocq as `(t1 t2) t3`. The first application `t1 t2` is interpreted as `trm_app t1 t2`. This expression, which itself has type `trm`, is applied to `t3`. Hence, `t1 t2 t3` is interpreted as `trm_app (trm_app t1 t2) t3`.

Loading Definitions from `LibSepReference`

As announced earlier, we will carry out proofs not with respect to the above definitions, but with respect to the definitions from *LibSepReference.v*.

Concretely, we close the scope of the module `Syntax`, and re-enter the scope of the definitions from `LibSepReference`.

End SYNTAX.

```
Implicit Types f : var.
Implicit Types b : bool.
Implicit Types p : loc.
Implicit Types n : int.
Implicit Types v w r : val.
Implicit Types t : trm.
Implicit Types h s : heap.
Implicit Types H : hprop.
```

41.2.2 Formalized Semantics of the Programming Language

Standard Big-Step Semantics

We are now ready to present a formal semantics for the language. To begin with, we present a big-step evaluation judgment, written `big s t s' v`. This judgment asserts that, starting from state `s`, the evaluation of the term `t` terminates in a state `s'`, and produces an output value `v`.

For simplicity, we assume terms to be in “A-normal form”: the arguments of applications and of conditionals are restricted to variables and values. Such a requirement does not limit expressiveness, but it simplifies the statement of the evaluation rules.

For example, if a source program includes a conditional `trm_if t0 t1 t2`, then it is required that `t0` be either a variable or a value. This is not a real restriction, because `trm_if t0 t1 t2` can always be encoded as `let x = t0 in if x then t1 else t2`.

The big-step judgment is inductively defined as follows. By convention, we let s range over entire memory states, and let h range over pieces of states that may correspond to a subset of the entire state.

Inductive **big** : heap $\rightarrow$ trm $\rightarrow$ heap $\rightarrow$ val $\rightarrow$ Prop :=

1. **big** for values and function definitions.

A value evaluates to itself. A term function evaluates to a value function. Likewise for a recursive function.

```
| big_val :  $\forall s v,$ 
  big  $s$  (trm_val  $v$ )  $s v$ 
| big_fun :  $\forall s x t1,$ 
  big  $s$  (trm_fun  $x t1$ )  $s$  (val_fun  $x t1$ )
| big_fix :  $\forall s f x t1,$ 
  big  $s$  (trm_fix  $f x t1$ )  $s$  (val_fix  $f x t1$ )
```

2. **big** for function applications.

The beta-reduction rule asserts that (val_fun $x t1$) $v2$ evaluates to the same result as subst $x v2 t1$. Likewise, (val_fix $f x t1$) $v2$ evaluates to subst $x v2$ (subst $f v1 t1$), where $v1$ denotes the recursive function itself, that is, val_fix $f x t1$.

```
| big_app_fun :  $\forall s1 s2 v1 v2 x t1 v,$ 
   $v1 = \text{val\_fun } x t1 \rightarrow$ 
  big  $s1$  (subst  $x v2 t1$ )  $s2 v \rightarrow$ 
  big  $s1$  (trm_app  $v1 v2$ )  $s2 v$ 
| big_app_fix :  $\forall s1 s2 v1 v2 f x t1 v,$ 
   $v1 = \text{val\_fix } f x t1 \rightarrow$ 
  big  $s1$  (subst  $x v2$  (subst  $f v1 t1$ ))  $s2 v \rightarrow$ 
  big  $s1$  (trm_app  $v1 v2$ )  $s2 v$ 
```

3. **big** for structural constructs.

A sequence trm_seq $t1 t2$ first evaluates $t1$, taking the state from $s1$ to $s2$, then drops the result of $t1$ and evaluates $t2$, taking the state from $s2$ to $s3$.

The let-binding trm_let $x t1 t2$ is similar, except that the variable x gets substituted with the result of $t1$ inside $t2$.

```
| big_seq :  $\forall s1 s2 s3 t1 t2 v1 v,$ 
  big  $s1 t1 s2 v1 \rightarrow$ 
  big  $s2 t2 s3 v \rightarrow$ 
  big  $s1$  (trm_seq  $t1 t2$ )  $s3 v$ 
| big_let :  $\forall s1 s2 s3 x t1 t2 v1 r,$ 
  big  $s1 t1 s2 v1 \rightarrow$ 
  big  $s2$  (subst  $x v1 t2$ )  $s3 r \rightarrow$ 
  big  $s1$  (trm_let  $x t1 t2$ )  $s3 r$ 
```


4. **big** for conditionals.

A conditional in a source program is assumed to be of the form `if  $t_0$  then  $t_1$  else  $t_2$` , where t_0 is either a variable or a value. If t_0 is a variable, then, by the time it reaches an evaluation position, the variable must have been substituted by a value. Thus, the evaluation rule only considers the form `if  $v_0$  then  $t_1$  else  $t_2$` . The value v_0 must be a boolean value, otherwise evaluation gets stuck.

The term `trm_if (val_bool true)  $t_1$   $t_2$` behaves like t_1 , whereas the term `trm_if (val_bool false)  $t_1$   $t_2$` behaves like t_2 . This behavior is described by a single rule, leveraging Rocq’s “if” constructor to factor out the two cases.

$$\begin{array}{l} | \text{big\_if} : \forall s1\ s2\ b\ v\ t1\ t2, \\ \quad \text{big } s1\ (\text{if } b\ \text{then } t1\ \text{else } t2)\ s2\ v \rightarrow \\ \quad \text{big } s1\ (\text{trm\_if } (\text{val\_bool } b)\ t1\ t2)\ s2\ v \end{array}$$

5. **big** for primitive stateless operations.

For similar reasons, the behavior of applied primitive functions only needs to be described for the case of value arguments.

An arithmetic operation expects integer arguments. The addition of `val_int  $n1$` and `val_int  $n2$` produces `val_int ( $n1 + n2$ )`.

The division operation, on the same arguments, produces the quotient `$n1 / n2$` , under the assumption that the divisor `$n2$` is non-zero. In other words, if a program performs a division by zero, then it cannot satisfy the **big** judgment.

The random number generator `val_rand  $n$` produces an integer `$n1$` in the range from 0 (inclusive) to `$n$` (exclusive).

$$\begin{array}{l} | \text{big\_add} : \forall s\ n1\ n2, \\ \quad \text{big } s\ (\text{val\_add } (\text{val\_int } n1)\ (\text{val\_int } n2))\ s\ (\text{val\_int } (n1 + n2)) \\ | \text{big\_div} : \forall s\ n1\ n2, \\ \quad n2 \neq 0 \rightarrow \\ \quad \text{big } s\ (\text{val\_div } (\text{val\_int } n1)\ (\text{val\_int } n2))\ s\ (\text{val\_int } (\text{Z.quot } n1\ n2)) \\ | \text{big\_rand} : \forall s\ n\ n1, \\ \quad 0 \leq n1 < n \rightarrow \\ \quad \text{big } s\ (\text{val\_rand } (\text{val\_int } n))\ s\ (\text{val\_int } n1) \end{array}$$

6. **big** for primitive operations on memory.

The term `val_ref  $v$` allocates a fresh cell with contents `$v$` and returns the location, `$p$` , of the new cell. This location must not be previously in the domain of the store `$s$` .

The term `val_get (val_loc  $p$ )` reads the value in the store `$s$` at location `$p$` . The location must be bound to a value in the store, otherwise evaluation is stuck.

The term `val_set (val_loc  $p$ )  $v$` updates the store at a location `$p$` assumed to be bound in the store `$s$` . The operation modifies the store and returns the unit value.

The term `val_free (val_loc  $p$ )` deallocates the cell at location `$p$` .

$$\begin{array}{l} | \text{big\_ref} : \forall s\ v\ p, \\ \quad \neg \text{Fmap.indom } s\ p \rightarrow \end{array}$$

```

    big s (val_ref v) (Fmap.update s p v) (val_loc p)
| big_get : ∀ s p,
    Fmap.indom s p →
    big s (val_get (val_loc p)) s (Fmap.read s p)
| big_set : ∀ s p v,
    Fmap.indom s p →
    big s (val_set (val_loc p) v) (Fmap.update s p v) val_unit
| big_free : ∀ s p,
    Fmap.indom s p →
    big s (val_free (val_loc p)) (Fmap.remove s p) val_unit.

```

Omni-Big-Step Semantics

A triple of the form `triple t H Q` aims to capture the property that, for any input state `s` satisfying the precondition `H`, any possible execution of the program starting from the initial configuration `(s,t)` will terminate and reach a final configuration satisfying the postcondition `Q`.

However, the standard big-step judgment, `big s t s' v`, only describes *one* possible execution. Using this judgment, we can capture the property that “if a particular execution terminates, then it reaches a final configuration satisfying `Q`”. But we do not capture the property that “all possible executions terminate”.

The omni-big-step judgment is a generalization of the standard big-step judgment that allows capturing properties of all possible executions. The judgment takes the form `eval s t Q`, where `Q` denotes a set of final configurations. It captures the fact that all possible executions starting on the configuration `(s,t)` terminate safely and reach a final configuration in the set `Q`. We can think of `Q` as a set of pairs made of values and a state. Equivalently, we can think of `Q` as a postcondition, of type `val → heap → Prop`.

Implicit Types `Q : val → heap → Prop`.

The inductive definition below defines the omni-big-step judgment, which has the form `eval s t Q`, where `Q` is a postcondition. The generalization from standard-big-step to omni-big-step follows a regular pattern, which should become apparent while stepping through the rules. **Inductive** `eval : heap → trm → (val → heap → Prop) → Prop :=`

1. `eval` for values and function definitions.

The judgment `eval s v Q` asserts that the value `v` in the state `s` satisfies the postcondition `Q`. The premise for deriving that judgment is thus that `Q v s` must hold.

```

| eval_val : ∀ s v Q,
    Q v s →
    eval s (trm_val v) Q
| eval_fun : ∀ s x t1 Q,
    Q (val_fun x t1) s →
    eval s (trm_fun x t1) Q

```

| $\text{eval\_fix} : \forall s f x t1 Q,$
 $Q (\text{val\_fix } f x t1) s \rightarrow$
 $\text{eval } s (\text{trm\_fix } f x t1) Q$

2. **eval** for function applications.

Consider a function **v1** of the form **val_fun** $x t1$. The term **trm_app** $v1 v2$ terminates with postcondition Q provided that the substituted term **subst** $x v2 t1$ terminates with postcondition Q . Hence, to prove **eval** $s1 (\text{trm\_app } v1 v2) Q$, we must establish **eval** $s1 (\text{subst } x v2 t1) Q$.

| $\text{eval\_app\_fun} : \forall s1 v1 v2 x t1 Q,$
 $v1 = \text{val\_fun } x t1 \rightarrow$
 $\text{eval } s1 (\text{subst } x v2 t1) Q \rightarrow$
 $\text{eval } s1 (\text{trm\_app } v1 v2) Q$
| $\text{eval\_app\_fix} : \forall s v1 v2 f x t1 Q,$
 $v1 = \text{val\_fix } f x t1 \rightarrow$
 $\text{eval } s (\text{subst } x v2 (\text{subst } f v1 t1)) Q \rightarrow$
 $\text{eval } s (\text{trm\_app } v1 v2) Q$

3. **eval** for sequencing.

Consider a sequence **trm_seq** $t1 t2$ in a state s . What is the requirement for this configuration to terminate with postcondition Q ? First, we need the evaluation of $t1$ in s to terminate. Let $Q1$ be (an overapproximation of) the set of possible results for the execution of $(s, t1)$. Then, to prove that the sequence **trm_seq** $t1 t2$ terminates in Q , we need to show that, for any intermediate state $s2$ satisfying $Q1$, the evaluation of $t2$ terminates with postcondition Q .

The case of let-bindings is similar, only with an additional substitution.

| $\text{eval\_seq} : \forall Q1 s t1 t2 Q,$
 $\text{eval } s t1 Q1 \rightarrow$
 $(\forall v1 s2, Q1 v1 s2 \rightarrow \text{eval } s2 t2 Q) \rightarrow$
 $\text{eval } s (\text{trm\_seq } t1 t2) Q$
| $\text{eval\_let} : \forall Q1 s x t1 t2 Q,$
 $\text{eval } s t1 Q1 \rightarrow$
 $(\forall v1 s2, Q1 v1 s2 \rightarrow \text{eval } s2 (\text{subst } x v1 t2) Q) \rightarrow$
 $\text{eval } s (\text{trm\_let } x t1 t2) Q$

4. **eval** for conditionals.

The term **(trm_if (val_bool b) t1 t2)** has the same behaviors as the term $t1$ when b is **true**, and as the term $t2$ when b is **false**.

| $\text{eval\_if} : \forall s (b:\text{bool}) t1 t2 Q,$
 $\text{eval } s (\text{if } b \text{ then } t1 \text{ else } t2) Q \rightarrow$
 $\text{eval } s (\text{trm\_if } (\text{val\_bool } b) t1 t2) Q$

5. **eval** for primitive stateless operations.

The judgment **eval** s (**val_add** $n1$ $n2$) Q asserts that the pair of the state s and the value $n1+n2$ produced by the addition operation satisfies the postcondition Q . Hence, the premise is Q (**val_int** ($n1 + n2$)) s .

Nondeterministic constructs are more interesting. The rule **eval_rand** gives the condition under which the term **val_rand** n , evaluated in a state s , produces an output satisfying Q . The first premise of the rule requires $n > 0$. The second premise requires that, for any value $n1$ that **val_rand** n may evaluate to – that is, such that $0 \leq n1 < n$ – the configuration made of $n1$ and s satisfies the postcondition Q . Formally: $\forall n1, (0 \leq n1 < n) \rightarrow (Q\ n1\ s)$.

$$\begin{array}{l} | \text{eval\_add} : \forall s\ n1\ n2\ Q, \\ \quad Q\ (\text{val\_int}\ (n1 + n2))\ s \rightarrow \\ \quad \text{eval}\ s\ (\text{val\_add}\ (\text{val\_int}\ n1)\ (\text{val\_int}\ n2))\ Q \\ | \text{eval\_div} : \forall s\ n1\ n2\ Q, \\ \quad n2 \neq 0 \rightarrow \\ \quad Q\ (\text{val\_int}\ (\text{Z.quot}\ n1\ n2))\ s \rightarrow \\ \quad \text{eval}\ s\ (\text{val\_div}\ (\text{val\_int}\ n1)\ (\text{val\_int}\ n2))\ Q \\ | \text{eval\_rand} : \forall s\ n\ Q, \\ \quad n > 0 \rightarrow \\ \quad (\forall n1, 0 \leq n1 < n \rightarrow Q\ n1\ s) \rightarrow \\ \quad \text{eval}\ s\ (\text{val\_rand}\ (\text{val\_int}\ n))\ Q \end{array}$$

6. **eval** for primitive operations on memory.

The most interesting case here is that of allocation, which is nondeterministic. For **val_ref** v , evaluated in a state s , to produce a configuration in Q , we require that, for any location p that is “fresh” for the state s , the pair made of p and “ s extended with a binding from p to v ” satisfies Q .

$$\begin{array}{l} | \text{eval\_ref} : \forall s\ v\ Q, \\ \quad (\forall p, \neg \text{Fmap.indom}\ s\ p \rightarrow \\ \quad \quad Q\ (\text{val\_loc}\ p)\ (\text{Fmap.update}\ s\ p\ v)) \rightarrow \\ \quad \text{eval}\ s\ (\text{val\_ref}\ v)\ Q \\ | \text{eval\_get} : \forall s\ p\ Q, \\ \quad \text{Fmap.indom}\ s\ p \rightarrow \\ \quad Q\ (\text{Fmap.read}\ s\ p)\ s \rightarrow \\ \quad \text{eval}\ s\ (\text{val\_get}\ (\text{val\_loc}\ p))\ Q \\ | \text{eval\_set} : \forall s\ p\ v\ Q, \\ \quad \text{Fmap.indom}\ s\ p \rightarrow \\ \quad Q\ \text{val\_unit}\ (\text{Fmap.update}\ s\ p\ v) \rightarrow \\ \quad \text{eval}\ s\ (\text{val\_set}\ (\text{val\_loc}\ p)\ v)\ Q \\ | \text{eval\_free} : \forall s\ p\ Q, \\ \quad \text{Fmap.indom}\ s\ p \rightarrow \\ \quad Q\ \text{val\_unit}\ (\text{Fmap.remove}\ s\ p) \rightarrow \end{array}$$

`eval s (val_free (val_loc p)) Q.`

Note that, given `eval s t Q`, it is *not* necessarily the case that Q is the exact set of configurations that are reachable from (s,t) ; it may rather be an *overapproximation* of that set. Trying to set up an inductive definition that relates (s,t) with *exactly* its set of reachable configurations would introduce unnecessary complications.

Note, also, that `eval s t Q` will not hold if there exist one or more executions of (s,t) that run into an error, i.e., that reach a configuration that is stuck. To see why, consider a program of the form `let n = rand 10 in (f n)`. When applying the rule `eval_let` to this program, the postcondition $Q1$ associated with `rand 10` must include all integers in the range 0..9. The second premise of the rule `eval_let` is of the form $\forall v1\ s2, Q1\ v1\ s2 \rightarrow \text{eval}\ s2\ (\text{subst}\ x\ v1\ t2)\ Q$. In our specific example, this premise asserts that for any result in $Q1$, which includes all integers n in the range 0..9, it is the case that the term `f n` terminates.

41.2.3 Definition of Triples

Now we come to the most important definition of the whole course!

A triple `triple t H Q` asserts that, for any input state s satisfying the precondition H , any possible execution of (s,t) terminates and reaches a final configuration satisfying the postcondition Q . This statement is captured in terms of the omni-big-step semantics as follows.

Definition `triple (t:trm) (H:hprop) (Q:val→hprop) : Prop :=`
 $\forall s, H\ s \rightarrow \text{eval}\ s\ t\ Q.$

Because the predicate `eval s t Q` captures termination, the predicate `triple t H Q` that we build on top of it also captures termination. It therefore yields a “total correctness” triple. If we had defined the predicate `eval` as `CoInductive` instead of `Inductive`, we would have allowed for infinite executions.

With such a coinductive definition of `eval`, we would have obtained a “partial correctness” triple instead. Such a partial correctness triple would still capture the property that none of the possible executions can get stuck – each possible execution either terminates safely with a result satisfying the postcondition, or else diverges.

Throughout the development of program logics, most research has concentrated on “partial correctness” logics. Depending on the set up, capturing “total correctness” is often much more challenging. In particular, when considering nondeterministic small-step semantics, and in particular when targeting concurrent programs, it can be nonobvious how to account for termination. But as long as we are concerned with sequential programs (i.e., without concurrency), omni-big-step semantics simplify reasoning about nondeterministic termination.

41.2.4 Structural Rules

Now that triples are defined, we can state and prove reasoning rules of Separation Logic. This chapter focuses on structural rules; the other rules will be presented in the next chapter.

We begin with the statements of the rules and present their proofs afterwards.

Module STRUCTURALRULES.

The frame rule asserts that an arbitrary heap predicate may be added to both the precondition and the postcondition of any triple.

Parameter *triple_frame* : $\forall t H Q H'$,
 $\text{triple } t H Q \rightarrow$
 $\text{triple } t (H \setminus^* H') (Q \setminus^{*+} H')$.

The consequence rule asserts, as in Hoare Logic, that the validity of a triple is preserved if we strengthen its precondition or weaken its postcondition.

Parameter *triple_conseq* : $\forall t H' Q' H Q$,
 $\text{triple } t H' Q' \rightarrow$
 $H ==> H' \rightarrow$
 $Q' ==> Q \rightarrow$
 $\text{triple } t H Q$.

The “extraction rule for pure facts” asserts that a judgment of the form $\text{triple } t (\setminus[P] \setminus^* H) Q$ is derivable from $P \rightarrow \text{triple } t H Q$. This structural rule captures the extraction of the pure facts out of the precondition of a triple, like `himpl_hstar_hpure_l` for entailments.

Parameter *triple_hpure* : $\forall t (P:\text{Prop}) H Q$,
 $(P \rightarrow \text{triple } t H Q) \rightarrow$
 $\text{triple } t (\setminus[P] \setminus^* H) Q$.

The “extraction rule for existentials” asserts that a judgment of the form $\text{triple } t (\setminus\exists x, J x) Q$ is derivable from $\forall x, \text{triple } t (J x) Q$. Again, this rule is the counterpart of the corresponding rule on entailments, `himpl_hexists_l`.

Parameter *triple_hexists* : $\forall t (A:\text{Type}) (J:A \rightarrow \text{hprop}) Q$,
 $(\forall x, \text{triple } t (J x) Q) \rightarrow$
 $\text{triple } t (\setminus\exists x, J x) Q$.

41.2.5 Derived Structural Rules

Given a triple to be established, it is very unlikely for the frame rule to be applicable directly, because the precondition must be exactly of the form $H \setminus^* H'$ and the postcondition exactly of the form $Q \setminus^{*+} H'$ for some H' . For example, the frame rule would not apply to a proof obligation of the form $\text{triple } t (H' \setminus^* H) (Q \setminus^{*+} H')$ because $H' \setminus^* H$ does not match $H \setminus^* H'$.

This limitation rule can be addressed by combining the frame rule and the rule of consequence into a single rule, called the “consequence-frame rule”. This rule, shown below, enables deriving a triple from another triple, without syntactic restrictions on the shape of the precondition and postcondition of the two triples involved.

Lemma *triple_conseq_frame* : $\forall H2 H1 Q1 t H Q$,
 $\text{triple } t H1 Q1 \rightarrow$

$H \implies H1 \setminus * H2 \rightarrow$
 $Q1 \setminus * H2 \implies Q \rightarrow$
 $\text{triple } t \ H \ Q.$

Exercise: 1 star, standard, especially useful (triple_conseq_frame) Prove the combined consequence-frame rule.

Proof using. Admitted.

□

The extraction rule `triple_hpure` assumes a precondition of the form $\setminus [P] \setminus * H$. The variant rule `triple_hpure'`, stated below, assumes instead a precondition with only the pure part, i.e., of the form $\setminus [P]$.

Exercise: 1 star, standard, optional (triple_hpure') Prove that `triple_hpure'` is indeed a corollary of `triple_hpure`.

`Lemma triple_hpure' :  $\forall t (P:\text{Prop}) \ Q,$`
`( $P \rightarrow \text{triple } t \ \setminus [] \ Q$ )  $\rightarrow$`
`$\text{triple } t \ \setminus [P] \ Q.$`

Proof using. Admitted.

□

End STRUCTURALRULES.

41.3 More Details

41.3.1 Proof of the Consequence Rule

The proof of the consequence rule relies on establishing that the omni-big-step judgment is preserved under weakening its postcondition. Recall that in `eval s t Q1`, the postcondition `Q1` is an overapproximation of the set of configurations reachable from (s, t) . Thus, if `Q1` is included in a larger set `Q2`, then `Q2` is also an overapproximation of the configurations reachable from (s, t) . The formalization is a straightforward induction.

`Lemma eval_conseq :  $\forall s \ t \ Q1 \ Q2,$`
`$\text{eval } s \ t \ Q1 \rightarrow$`
`$Q1 \implies Q2 \rightarrow$`
`$\text{eval } s \ t \ Q2.$`

Proof using.

introv M W.

asserts W': ($\forall v \ h, \ Q1 \ v \ h \rightarrow Q2 \ v \ h$). { auto. } clear W.

induction M; try solve [constructors×].

Qed.

From there, it is straightforward to derive the consequence rule.

```

Lemma triple_conseq :  $\forall t H' Q' H Q,$ 
  triple  $t H' Q' \rightarrow$ 
   $H \implies H' \rightarrow$ 
   $Q' \implies Q \rightarrow$ 
  triple  $t H Q$ .

```

Proof using. *unfolds triple. introv M MH MQ HF. applies $\times$ eval_conseq. Qed.*

41.3.2 Proof of the Frame Rule

The second key property that we wish to establish is the “frame rule”. The frame rule asserts that if a specification `triple  $H \ t \ Q$` holds, then one may derive `triple  $(H \setminus^* H') \ t \ (Q \setminus^* + H')$` for any H' . Recall from chapter `Hprop` that the operator $Q \setminus^* + H'$ is a notation for `fun x  $\Rightarrow (Q \ x \setminus^* H')$` .

Intuitively, if the term t executes safely in a heap H , then this term should behave similarly in any extension of H with a disjoint part H' . Moreover, its evaluation should leave this piece of state H' unmodified throughout the execution of t .

The crux of the proof of the frame rule is to argue that `eval` is stable under extension with a disjoint piece of heap.

```

Lemma eval_frame :  $\forall h1 h2 t Q,$ 
  eval  $h1 \ t \ Q \rightarrow$ 
  Fmap.disjoint  $h1 \ h2 \rightarrow$ 
  eval  $(h1 \setminus_u h2) \ t \ (Q \setminus^{*+} (= h2))$ .

```

This proof goes by induction. The most interesting step is for allocation. In a derivation built using `eval_ref`, we are given as assumption a property that holds for any location p fresh from $h1$, and we are requested to prove a property that holds for any location p fresh from $h1 \setminus_u h2$. We are thus restricting the set of p that can be considered for allocation, and so the result holds.

Proof using.

```

introv M HD. gen h2. induction M; intros;
  try solve [ hint hstar_intro; constructors $\times$  ].
  { rename M into M1, H into M2, IHM into IH1, H0 into IH2.
    specializes IH1 HD. applies eval_seq IH1. introv HK.
    lets (h1'&h2'&K1'&K2'&KD&KU): hstar_inv HK. subst. applies $\times$  IH2. }
  { rename M into M1, H into M2, IHM into IH1, H0 into IH2.
    specializes IH1 HD. applies eval_let IH1. introv HK.
    lets (h1'&h2'&K1'&K2'&KD&KU): hstar_inv HK. subst. applies $\times$  IH2. }
  {
    rename H into M. applies eval_ref. intros p Hp.
    rewrite Fmap.indom_union_eq in Hp. rew_logic in Hp.
    destruct Hp as [Hp1 Hp2].
    rewrite $\times$  Fmap.update_union_not_r. applies hstar_intro.
    { applies $\times$  M. } { auto. } { applies $\times$  Fmap.disjoint_update_not_r. } }

```



```

{ applies eval_get. { rewrite× Fmap.indom_union_eq. }
  { rewrite× Fmap.read_union_l. applies× hstar_intro. } }
{ applies eval_set. { rewrite× Fmap.indom_union_eq. }
  { rewrite× Fmap.update_union_l. applies hstar_intro.
    { auto. } { auto. } { applies× Fmap.disjoint_update_l. } } }
{ applies eval_free. { rewrite× Fmap.indom_union_eq. }
  { rewrite× Fmap.remove_disjoint_union_l. applies hstar_intro.
    { auto. } { auto. } { applies× Fmap.disjoint_remove_l. } } }

```

Qed.

The frame rule is derived from `eval_frame` and `eval_conseq`.

Lemma `triple_frame` : $\forall t\ H\ Q\ H'$,
 $\text{triple } t\ H\ Q \rightarrow$
 $\text{triple } t\ (H \setminus * H')\ (Q \setminus ** H')$.

Proof.

```

intros M. intros h HF. lets (h1&h2&M1&M2&MD&MU): hstar_inv (rm HF).
subst. specializes M M1. applies eval_conseq.
{ applies eval_frame M MD. } { xsimpl. intros h'  $\rightarrow$ . applies M2. }

```

Qed.

41.3.3 Proof of the Extraction Rules

The extraction rules are proved by unfolding the definition of `triple` and inverting the hypothesis capturing the fact that the input state satisfies the precondition.

Lemma `triple_hpure` : $\forall t\ (P:\text{Prop})\ H\ Q$,
 $(P \rightarrow \text{triple } t\ H\ Q) \rightarrow$
 $\text{triple } t\ (\setminus [P] \setminus * H)\ Q$.

Proof using.

```

intros M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).
lets E: empty_inv HP. subst. rewrite Fmap.union_empty_l. applies  $\neg$  M.

```

Qed.

Exercise: 1 star, standard, especially useful (`triple_hexists`) Prove `triple_hexists`.

Lemma `triple_hexists` : $\forall t\ (A:\text{Type})\ (J:A \rightarrow \text{hprop})\ Q$,
 $(\forall (x:A), \text{triple } t\ (J\ x)\ Q) \rightarrow$
 $\text{triple } t\ (\text{hexists } J)\ Q$.

Proof using. *Admitted*.

□

41.3.4 Rule for Naming Heaps

To prove $\text{triple } t \ H \ Q$, it suffices to show that, for any heap h satisfying H , the triple $\text{triple } t \ (= \ h) \ Q$ holds. In other words, even though the predicate triple abstracts away from the input heap, it remains technically possible to introduce an entity, namely h , that refers to the specific heap at hand during the evaluation of t . (An example use case will appear in a later chapter (WPsem, lemma `wp_equiv_2`).

Exercise: 1 star, standard, optional (`triple_named_heap`) Prove the reasoning rule *hoare_named_heap*.

Lemma `triple_named_heap` : $\forall t \ H \ Q,$
 $(\forall h, \ H \ h \rightarrow \text{triple } t \ (= \ h) \ Q) \rightarrow$
 $\text{triple } t \ H \ Q.$

Proof using. *Admitted*.

□

41.4 Optional Material

41.4.1 An Alternative Structural Rule for Existentials

Module `ALTERNATIVEEXISTENTIALRULE`.

Traditional papers on Separation Logic do not include `triple_hexists`, but instead include a rule called here `triple_hexists2`, which features an existential quantifier both in the precondition and in the postcondition. As we show next, in the presence of the consequence rule, the two rules are equivalent.

Exercise: 2 stars, standard, especially useful (`triple_hexists2`) Using `triple_hexists` and `triple_conseq`, as well as the tactic *xsimpl*, prove that `triple_hexists2` is derivable.

Lemma `triple_hexists2` : $\forall A \ (Hof : A \rightarrow \text{hprop}) \ (Qof : A \rightarrow \text{val} \rightarrow \text{hprop}) \ t,$
 $(\forall x, \text{triple } t \ (Hof \ x) \ (Qof \ x)) \rightarrow$
 $\text{triple } t \ (\backslash \exists x, \ Hof \ x) \ (\text{fun } v \Rightarrow \backslash \exists x, \ Qof \ x \ v).$

Proof using. *Admitted*.

Exercise: 2 stars, standard, especially useful (`triple_hexists_of_triple_hexists2`) Reciprocally, using `triple_hexists2` and `triple_conseq`, as well as the tactic *xsimpl*, prove that `triple_hexists` is derivable. Of course, you may not use `triple_hexists` in this proof.)

Lemma `triple_hexists_of_triple_hexists2` : $\forall t \ (A : \text{Type}) \ (Hof : A \rightarrow \text{hprop}) \ Q,$
 $(\forall x, \text{triple } t \ (Hof \ x) \ Q) \rightarrow$
 $\text{triple } t \ (\backslash \exists x, \ Hof \ x) \ Q.$

Proof using. *Admitted*.

Compared with `triple_hexists2`, the formulation of `triple_hexists` is more concise, and is easier to exploit in practice.

End `ALTERNATIVEEXISTENTIALRULE`.

41.4.2 Small-Step Semantics

Suppose we have a language with a formal semantics written not in big-step style but in small-step style. What are the options for formalizing total correctness triples? We discuss two approaches. The first one consists of formally relating the small-step judgment with the omni-big-step judgment. The second consists of directly defining triples in terms of the small-step judgment. Before explaining the two approaches, let us present the formalization of the small-step semantics.

The judgment `step s t s' t'` describes the small-step reduction relation: it asserts that the program configuration (s, t) can take one reduction step to reach the program configuration (s', t') . Its definition is standard.

Inductive `step` : `heap` $\rightarrow$ `trm` $\rightarrow$ `heap` $\rightarrow$ `trm` $\rightarrow$ `Prop` :=

```
| step_seq_ctx :  $\forall$  s1 s2 t1 t1' t2,
  step s1 t1 s2 t1'  $\rightarrow$ 
  step s1 (trm_seq t1 t2) s2 (trm_seq t1' t2)
| step_let_ctx :  $\forall$  s1 s2 x t1 t1' t2,
  step s1 t1 s2 t1'  $\rightarrow$ 
  step s1 (trm_let x t1 t2) s2 (trm_let x t1' t2)

| step_fun :  $\forall$  s x t1,
  step s (trm_fun x t1) s (val_fun x t1)
| step_fix :  $\forall$  s f x t1,
  step s (trm_fix f x t1) s (val_fix f x t1)
| step_app_fun :  $\forall$  s v1 v2 x t1,
  v1 = val_fun x t1  $\rightarrow$ 
  step s (trm_app v1 v2) s (subst x v2 t1)
| step_app_fix :  $\forall$  s v1 v2 f x t1,
  v1 = val_fix f x t1  $\rightarrow$ 
  step s (trm_app v1 v2) s (subst x v2 (subst f v1 t1))
| step_if :  $\forall$  s b t1 t2,
  step s (trm_if (val_bool b) t1 t2) s (if b then t1 else t2)
| step_seq :  $\forall$  s t2 v1,
  step s (trm_seq v1 t2) s t2
| step_let :  $\forall$  s x t2 v1,
  step s (trm_let x v1 t2) s (subst x v1 t2)
```

```

| step_add :  $\forall s\ n1\ n2,$ 
  step  $s$  (val_add (val_int  $n1$ ) (val_int  $n2$ ))  $s$  (val_int ( $n1 + n2$ ))
| step_div :  $\forall s\ n1\ n2,$ 
   $n2 \neq 0 \rightarrow$ 
  step  $s$  (val_div (val_int  $n1$ ) (val_int  $n2$ ))  $s$  (Z.quot  $n1\ n2$ )
| step_rand :  $\forall s\ n\ n1,$ 
   $0 \leq n1 < n \rightarrow$ 
  step  $s$  (val_rand (val_int  $n$ ))  $s$  (val_int  $n1$ )
| step_ref :  $\forall s\ v\ p,$ 
   $\neg \text{Fmap.indom } s\ p \rightarrow$ 
  step  $s$  (val_ref  $v$ ) (Fmap.update  $s\ p\ v$ ) (val_loc  $p$ )
| step_get :  $\forall s\ p,$ 
  Fmap.indom  $s\ p \rightarrow$ 
  step  $s$  (val_get (val_loc  $p$ ))  $s$  (Fmap.read  $s\ p$ )
| step_set :  $\forall s\ p\ v,$ 
  Fmap.indom  $s\ p \rightarrow$ 
  step  $s$  (val_set (val_loc  $p$ )  $v$ ) (Fmap.update  $s\ p\ v$ ) val_unit
| step_free :  $\forall s\ p,$ 
  Fmap.indom  $s\ p \rightarrow$ 
  step  $s$  (val_free (val_loc  $p$ )) (Fmap.remove  $s\ p$ ) val_unit.

```

The judgment **steps** $s\ t\ s'\ t'$ corresponds to the reflexive and transitive closure of **step**. Concretely, it asserts that the configuration (s,t) can reduce in zero, one, or several steps to (s',t') .

```

Inductive steps : heap  $\rightarrow$  trm  $\rightarrow$  heap  $\rightarrow$  trm  $\rightarrow$  Prop :=
| steps_refl :  $\forall s\ t,$ 
  steps  $s\ t\ s\ t$ 
| steps_step :  $\forall s1\ s2\ s3\ t1\ t2\ t3,$ 
  step  $s1\ t1\ s2\ t2 \rightarrow$ 
  steps  $s2\ t2\ s3\ t3 \rightarrow$ 
  steps  $s1\ t1\ s3\ t3$ .

```

```

Lemma steps_of_step :  $\forall s1\ s2\ t1\ t2,$ 
  step  $s1\ t1\ s2\ t2 \rightarrow$ 
  steps  $s1\ t1\ s2\ t2$ .

```

Proof using. *intros M. applys steps_step M. applys steps_refl. Qed.*

```

Lemma steps_trans :  $\forall s1\ s2\ s3\ t1\ t2\ t3,$ 
  steps  $s1\ t1\ s2\ t2 \rightarrow$ 
  steps  $s2\ t2\ s3\ t3 \rightarrow$ 
  steps  $s1\ t1\ s3\ t3$ .

```

Proof using.

intros M1. induction M1; intros M2. { auto. } { constructors ×. }
 Qed.

The predicate `trm_is_val t` asserts that `t` is a value.

Definition `trm_is_val (t:trm) : Prop :=`
`match t with trm_val v => True | _ => False end.`

Consider a configuration (s, t) , where `t` is not a value. If this configuration cannot take any reduction step, it is said to be *stuck*. Conversely, a configuration (s, t) that can take a step is said to be *reducible*.

Definition `reducible (s:heap) (t:trm) : Prop :=`
`∃ s' t', step s t s' t'.`

Values are not reducible.

Lemma `reducible_val_inv : ∀ s v,`
`¬ reducible s v.`

Proof using. *intros (s'&t'&M). inverts M. Qed.*

The predicate `notstuck s t` asserts that either `t` is a value or `t` is reducible.

Definition `notstuck (s:heap) (t:trm) : Prop :=`
`trm_is_val t ∨ reducible s t.`

41.4.3 Equivalence Between Small-Step and Omni-Big-Step

If we consider the omni-big-step semantics to be the starting point of a development, then the definition of `triple` is obviously sound with respect to that semantics, as it concludes `eval s t Q`. If, however, we consider the small-step semantics to be the starting point, then to establish soundness it is required to prove that `triple t H Q` ensures that all possible evaluations of `t` in a heap satisfying `H` are: (1) terminating, and (2) guaranteed to reach a value; moreover, (3) this value and the final state obtained must satisfy the postcondition `Q`. We begin by formalizing each of these properties.

The judgment `terminates s t` is defined inductively. The execution starting from (s, t) terminates if any possible step leads to a configuration that terminates. Note that a configuration that has reached a value cannot take a step, hence is considered terminating.

Inductive `terminates : heap → trm → Prop :=`
`| terminates_step : ∀ s t,`
 `(∀ s' t', step s t s' t' → terminates s' t') →`
 `terminates s t.`

The judgment `safe s t` asserts that no execution may reach a stuck term. In other words, for any configuration (s', t') reachable from (s, t) , it is the case that the configuration (s', t') is either a value or is reducible.

Definition `safe (s:heap) (t:trm) : Prop :=`
`∀ s' t', steps s t s' t' → notstuck s' t'.`

The judgment $\text{correct } s \ t \ Q$ asserts that if the execution of (s,t) reaches a final configuration, then this final configuration satisfies Q .

Definition $\text{correct } (s:\text{heap}) \ (t:\text{trm}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{Prop} :=$
 $\forall s' \ v, \text{steps } s \ t \ s' \ v \rightarrow Q \ v \ s'.$

The conjunction of **safe** and **correct** corresponds to “partial correctness”. The conjunction of **safe**, **correct** and **terminates** corresponds to “total correctness”. The soundness theorem that we aim for establishes that $\text{triple } t \ H \ Q$ entails total correctness.

$\text{triple } t \ H \ Q \rightarrow \text{forall } s, H \ s \rightarrow \text{terminates } s \ t \wedge \text{safe } s \ t \wedge \text{correct } s \ t \ Q.$

To prove soundness, we introduce an inductively predicate, named **seval**, which captures total correctness in small-step style. On the one hand, we prove that **seval** entails **safe**, **correct**, and **terminates**. On the other hand, we prove that **seval** is related to the omni-big-step judgment, **eval**.

The judgment $\text{seval } s \ t \ Q$ asserts that any execution of (s,t) terminates and reaches a configuration satisfying Q . In the “base” case, $\text{seval } s \ v \ Q$ holds if the terminal configuration (s,v) satisfies Q . In the “step” case, $\text{seval } s \ t \ Q$ holds if (1) the configuration (s,t) is reducible, and (2) if for any step that (s,t) may take to (s',t') , the predicate $\text{seval } s' \ t' \ Q$ holds.

Inductive $\text{seval} : \text{heap} \rightarrow \text{trm} \rightarrow (\text{val} \rightarrow \text{hprop}) \rightarrow \text{Prop} :=$
 $| \text{seval\_val} : \forall s \ v \ Q,$
 $\quad Q \ v \ s \rightarrow$
 $\quad \text{seval } s \ v \ Q$
 $| \text{seval\_step} : \forall s \ t \ Q,$
 $\quad \text{reducible } s \ t \rightarrow$
 $\quad (\forall s' \ t', \text{step } s \ t \ s' \ t' \rightarrow \text{seval } s' \ t' \ Q) \rightarrow$
 $\quad \text{seval } s \ t \ Q.$

Exercise: 2 stars, standard, optional (seval_val_inv) As a warm-up to get some familiar with **seval**, prove the following inversion lemma, which asserts that, given a value v , the property $\text{seval } s \ v \ Q$ implies $Q \ s \ v$.

Lemma $\text{seval\_val\_inv} : \forall s \ v \ Q,$
 $\text{seval } s \ v \ Q \rightarrow$
 $Q \ v \ s.$

Proof using. *Admitted.*

□

We begin with the three key properties of **seval**: termination, safety, and correctness.

Exercise: 2 stars, standard, especially useful (seval_terminates) Prove that **seval** captures termination.

Lemma $\text{seval\_terminates} : \forall s \ t \ Q,$
 $\text{seval } s \ t \ Q \rightarrow$
 $\text{terminates } s \ t.$

Proof using. *Admitted.*

□

Exercise: 3 stars, standard, especially useful (seval_safe) Prove that **seval** captures safety.

Lemma **seval_safe** : $\forall s\ t\ Q,$
 seval $s\ t\ Q \rightarrow$
 safe $s\ t$.

Proof using. *Admitted.*

□

Exercise: 3 stars, standard, especially useful (seval_correct) Prove that **seval** captures correctness.

Lemma **seval_correct** : $\forall s\ t\ Q,$
 seval $s\ t\ Q \rightarrow$
 correct $s\ t\ Q$.

Proof using. *Admitted.*

□

We can group the three results into a soundness theorem for **seval**.

Lemma **seval_sound** : $\forall s\ t\ Q,$
 seval $s\ t\ Q \rightarrow$
 terminates $s\ t \wedge$ safe $s\ t \wedge$ correct $s\ t\ Q$.

Proof using.

introv M. splits.
 { *applies* $\times$ *seval_terminates.* }
 { *applies* $\times$ *seval_safe.* }
 { *applies* $\times$ *seval_correct.* }

Qed.

We now establish that the omni-big-step evaluation judgment entails the small-step-based **seval** judgment. The proof is carried out by induction on the omni-big-step relation. It relies on a number of auxiliary results establishing that, for each term construct, the **seval** judgment admits an evaluation rule that mimics the omni-big-step evaluation rule. We begin with the statements of the auxiliary lemmas.

Lemma **seval_fun** : $\forall s\ x\ t1\ Q,$
 $Q\ (\text{val\_fun } x\ t1)\ s \rightarrow$
 seval $s\ (\text{trm\_fun } x\ t1)\ Q$.

Proof using.

introv M. applies seval_step.
 { *do 2 esplit. constructor.* }
 { *introv R. inverts R. { applies seval_val. applies M. } }*

Qed.

```
Lemma seval_fix : ∀ s f x t1 Q,  
  Q (val_fix f x t1) s →  
  seval s (trm_fix f x t1) Q.
```

Proof using.

```
  introv M. applies seval_step.  
  { do 2 esplit. constructor. }  
  { introv R. inverts R. { applies seval_val. applies M. } }
```

Qed.

```
Lemma seval_app_fun : ∀ s x v1 v2 t1 Q,  
  v1 = val_fun x t1 →  
  seval s (subst x v2 t1) Q →  
  seval s (trm_app v1 v2) Q.
```

Proof using.

```
  introv E M. applies seval_step.  
  { do 2 esplit. applies× step_app_fun. }  
  { introv R. invert R; try solve [intros; false]. introv → → → → → R. inverts E.  
    applies M. }
```

Qed.

```
Lemma seval_app_fix : ∀ s f x v1 v2 t1 Q,  
  v1 = val_fix f x t1 →  
  seval s (subst x v2 (subst f v1 t1)) Q →  
  seval s (trm_app v1 v2) Q.
```

Proof using.

```
  introv E M. applies seval_step.  
  { do 2 esplit. applies× step_app_fix. }  
  { introv R. invert R; try solve [intros; false]. introv → → → → → R. inverts E.  
    applies M. }
```

Qed.

Exercise: 5 stars, standard, especially useful (seval_seq) Prove the big-step reasoning rule for sequence for **seval**.

```
Lemma seval_seq : ∀ s t1 t2 Q1 Q,  
  seval s t1 Q1 →  
  (∀ s1 v1, Q1 v1 s1 → seval s1 t2 Q) →  
  seval s (trm_seq t1 t2) Q.
```

Proof using. *Admitted*.

□

Exercise: 5 stars, standard, optional (seval_let) The proof of **seval_let** is the same as that of **seval_seq**. We also present it as an exercise to avoid giving away the solution.

Lemma `seval_let` : $\forall s\ x\ t1\ t2\ Q1\ Q,$
`seval` $s\ t1\ Q1 \rightarrow$
 $(\forall s1\ v1, Q1\ v1\ s1 \rightarrow \text{seval}\ s1\ (\text{subst}\ x\ v1\ t2)\ Q) \rightarrow$
`seval` $s\ (\text{trm\_let}\ x\ t1\ t2)\ Q.$

Proof using. *Admitted.*

□

Lemma `seval_if` : $\forall s\ b\ t1\ t2\ Q,$
`seval` $s\ (\text{if}\ b\ \text{then}\ t1\ \text{else}\ t2)\ Q \rightarrow$
`seval` $s\ (\text{trm\_if}\ b\ t1\ t2)\ Q.$

Proof using.

intros M. applies `seval_step`.
 $\{ \text{do } 2\ \text{esplit. constructors} \times. \}$
 $\{ \text{intros } R. \text{inverts } R; \text{tryfalse. } \{ \text{applies } M. \} \}$

Qed.

We are now ready to prove by induction that the omni-big-step judgment entails the small-step-based judgment `seval`. Note that the reciprocal entailment also holds, however it is not needed for establishing soundness.

Lemma `seval_of_eval` : $\forall s\ t\ Q,$
`eval` $s\ t\ Q \rightarrow$
`seval` $s\ t\ Q.$

Proof using.

intros M. induction M.
 $\{ \text{applies} \times \text{seval\_val.} \}$
 $\{ \text{applies} \times \text{seval\_fun.} \}$
 $\{ \text{applies} \times \text{seval\_fix.} \}$
 $\{ \text{applies} \times \text{seval\_app\_fun.} \}$
 $\{ \text{applies} \times \text{seval\_app\_fix.} \}$
 $\{ \text{applies} \times \text{seval\_seq.} \}$
 $\{ \text{applies} \times \text{seval\_let.} \}$
 $\{ \text{applies} \times \text{seval\_if.} \}$
 $\{ \text{applies } \text{seval\_step.}$
 $\{ \text{do } 2\ \text{esplit. } \text{applies} \times \text{step\_add.} \}$
 $\{ \text{intros } K. \text{inverts } K. \text{applies} \times \text{seval\_val.} \} \}$
 $\{ \text{applies } \text{seval\_step.}$
 $\{ \text{do } 2\ \text{esplit. } \text{applies} \times \text{step\_div.} \}$
 $\{ \text{intros } K. \text{inverts } K. \text{applies} \times \text{seval\_val.} \} \}$
 $\{ \text{applies } \text{seval\_step.}$
 $\{ \text{do } 2\ \text{esplit. } \text{applies} \times \text{step\_rand } 0. \text{math.} \}$
 $\{ \text{intros } K. \text{inverts } K; \text{tryfalse. } \text{applies} \times \text{seval\_val.} \} \}$
 $\{ \text{applies } \text{seval\_step.}$
 $\{ \text{forwards} \neg (p \& D \& N): (\text{exists\_fresh null } s). \}$

```

    do 2 esplit. applies × step_ref. }
  { intros K. inverts K; tryfalse. applies × seval_val. } }
{ applies seval_step.
  { do 2 esplit. applies × step_get. }
  { intros K. inverts K; tryfalse. applies × seval_val. } }
{ applies seval_step.
  { do 2 esplit. applies × step_set. }
  { intros K. inverts K; tryfalse. applies × seval_val. } }
{ applies seval_step.
  { do 2 esplit. applies × step_free. }
  { intros K. inverts K; tryfalse. applies × seval_val. } }

```

Qed.

Putting it all together, we conclude on the soundness of Separation Logic with respect to the small-step semantics defined by the relation **step**.

Lemma `soundness_small_step` : $\forall t\ H\ Q,$
 $\text{triple } t\ H\ Q \rightarrow$
 $\forall s, H\ s \rightarrow \text{terminates } s\ t \wedge \text{safe } s\ t \wedge \text{correct } s\ t\ Q.$

Proof using.

```

  unfold triple. intros M Hs. specializes M Hs.
  lets R: seval_of_eval M. applies seval_sound R.

```

Qed.

41.4.4 Small-Step-Based Triples

A second possible approach to setting up a Separation Logic starting from a small-step semantics consists of defining **triple** directly in terms of **seval**, without introducing the omnibig-step judgment **eval**.

Let **striple** denote triples built from **seval**.

Definition `striple` ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : Prop :=
 $\forall (s:\text{heap}), H\ s \rightarrow \text{seval } s\ t\ Q.$

The first step is to check that this new definition of triple is sound. This property has been essentially been established in the previous section via the lemma **seval_sound**.

Lemma `striple_sound` : $\forall t\ H\ Q,$
 $\text{striple } t\ H\ Q \rightarrow$
 $\forall s, H\ s \rightarrow \text{terminates } s\ t \wedge \text{safe } s\ t \wedge \text{correct } s\ t\ Q.$

Proof using. *intros* *M* *Hs*. *specializes* *M* *Hs*. *applies* *seval_sound* *M*. Qed.

The second step is to establish reasoning rules for **striple**. Let us begin with the consequence rule and the frame rule. For those, we need to establish the consequence and the frame property directly for **seval**. The proofs differ from the corresponding proofs on **eval**.

The consequence rule for **seval** is established by induction.

Lemma `seval_conseq` : $\forall s\ t\ Q\ Q',$

```

seval  $s\ t\ Q' \rightarrow$ 
 $Q' \implies Q \rightarrow$ 
seval  $s\ t\ Q$ .

```

Proof using.

```

intros  $M\ WQ$ . induction  $M$ .
{ applies seval_val. applies $\times\ WQ$ . }
{ rename  $H1$  into  $IH$ .
  applies seval_step.
  { auto. }
  { intros  $HR$ . applies $\times\ IH$ . } }

```

Qed.

The consequence rule follows directly.

Lemma **stuple_conseq** : $\forall\ t\ H'\ Q'\ H\ Q$,

```

stuple  $t\ H'\ Q' \rightarrow$ 
 $H \implies H' \rightarrow$ 
 $Q' \implies Q \rightarrow$ 
stuple  $t\ H\ Q$ .

```

Proof using.

```

intros  $M\ MH\ MQ\ HF$ . applies seval_conseq  $M\ MQ$ . applies $\times\ MH$ .

```

Qed.

The frame property for **seval** is also established by induction, however the proof is more involved.

Section **stepsFrame**.

Hint Constructors **step**.

Lemma **seval_frame** : $\forall\ h1\ h2\ t\ Q$,

```

seval  $h1\ t\ Q \rightarrow$ 
Fmap.disjoint  $h1\ h2 \rightarrow$ 
seval  $(h1 \setminus u\ h2)\ t\ (Q \setminus *+ (= h2))$ .

```

Proof using.

```

intros  $M\ HD$ . gen  $h2$ . induction  $M$ ; intros.
{ applies seval_val. applies $\times\ hstar\_intro$ . }
{ rename  $H$  into  $M1$ ,  $H0$  into  $M2$ ,  $H1$  into  $IH2$ .
  applies seval_step.
  { unfolds reducible. clear  $M2\ IH2$ . destruct  $M1$  as  $(s' \& t' \& R)$ .
    induction  $R$ ; tryfalse; try solve [ do 2 esplit; constructors $\times$  ].
    { forwards $\times\ (s' \& t' \& R')$ :  $IHR$ . }
    { forwards $\times\ (s' \& t' \& R')$ :  $IHR$ . }
    { lets  $(p' \& F \& \_)$ : exists_fresh null (Fmap.union  $s\ h2$ ). do 2 esplit.
      applies step_ref  $v\ p'$ . eauto. }
    { do 2 esplit. applies step_get. applies $\times\ \text{Fmap.indom\_union\_l}$ . }
    { do 2 esplit. applies step_set. applies $\times\ \text{Fmap.indom\_union\_l}$ . }
  }
}

```

```

{ do 2 esplit. applies step_free. applies× Fmap.indom_union_l. } }
{ intros R. cuts (s1'&E'&D'&R'):
  (∃ s1', s' = s1' \u h2 ∧ Fmap.disjoint s1' h2 ∧ step s t s1' t').
  { subst. applies× IH2. }
  clear M2 IH2.
  gen_eq su: (s \u h2). gen s.
  unfolds reducible. induction R; intros; subst; eauto.
  { destruct M1 as (s0&t0&R0).
    rename R into R1. forwards× (s1'&E&D&R1'): IHR s.
    { inverts R0. { eauto. } { inverts R1. } } }
  { destruct M1 as (s0&t0&R0).
    rename R into R1. forwards× (s1'&E&D&R1'): IHR s.
    { inverts R0. { eauto. } { inverts R1. } } }
  { rename H into D. rewrite Fmap.indom_union_eq in D. rew_logic in D.
    destruct D as (D1&D2). esplit. splits.
    { rewrite× Fmap.update_union_not_r. }
    { applies× Fmap.disjoint_update_not_r. }
    { eauto. } }
  { destruct M1 as (se&te&Re). inverts Re; tryfalse.
    rewrite× Fmap.read_union_l. }
  { destruct M1 as (se&te&Re). inverts Re; tryfalse. esplit. splits.
    { rewrite× Fmap.update_union_l. }
    { applies× Fmap.disjoint_update_l. }
    { eauto. } } }
  { destruct M1 as (se&te&Re). inverts Re; tryfalse. esplit. splits.
    { rewrite× remove_disjoint_union_l. }
    { applies× Fmap.disjoint_remove_l. }
    { eauto. } } } }

```

Qed.

End *stepsFrame*.

The frame rule for *strip* then follows from the frame and the consequence properties for *seval*.

Lemma *strip_frame* : $\forall t H Q H'$,
strip *t H Q* $\rightarrow$
strip *t* (*H * H'*) (*Q *+ H'*).

Proof.

```

intros M. intros h HF. lets (h1&h2&M1&M2&MD&MU): hstar_inv (rm HF).
subst. specializes M M1. applies seval_conseq.
{ applies seval_frame M MD. } { xsimpl. intros h' \rightarrow. applies M2. }

```

Qed.

To establish other reasoning rules for *strip*, one need to prove big-step-style evaluation

rules for **seval**, such as the lemma **seval_let** stated earlier.

In summary, it is possible to set up a Separation Logic directly based on the small-step-based judgment **seval**, without going through a definition of the omni-big-step judgment **eval**. However, most of the proof effort remains, as one needs to establish the frame property and to derive big-step-style evaluation rules such as **seval_let**.

41.4.5 Separation Logic Triples for Quasi-Deterministic Semantics

Module **BAKEDINFRAME**.

Consider a programming language that is deterministic except for the allocation operation, which may pick any fresh location. For such languages, it is possible to define Separation Logic triples in terms of the standard big-step semantics, using a technique known as the “baked-in frame rule”. Let **beval** denote the standard big-step judgment: **beval** *s* *t* *s'* *v* asserts that the evaluation of *t* in a state *s* produces the value *v* in a state *s'*.

Parameter **beval** : **heap** → **trm** → **heap** → **val** → **Prop**.

First, we define a (total correctness) Hoare triple, written **hoare** *t* *H* *Q*. This judgment asserts that, starting from a state *s* satisfying the precondition *H*, the term *t* evaluates to a value *v* and to a state *s'* that, together, satisfy the postcondition *Q*. Formally:

Definition **hoare** (*t*:**trm**) (*H*:**hprop**) (*Q*:**val**→**hprop**) : **Prop** :=

 ∀ (*s*:**heap**), *H* *s* →
 ∃ (*s'*:**heap**) (*v*:**val**), **beval** *s* *t* *s'* *v* ∧ *Q* *v* *s'*.

Exercise: 2 stars, standard, especially useful (hoare_conseq) To gain familiarity with the **hoare** judgment, prove the consequence rule for Hoare triples.

Lemma **hoare_conseq** : ∀ *t* *H* *Q* *H'* *Q'*,

hoare *t* *H'* *Q'* →
 H ==> *H'* →
 Q' ==> *Q* →
 hoare *t* *H* *Q*.

Proof using. *Admitted*.

□

The judgment **hoare** *t* *H* *Q* satisfies the consequence rule, yet it does not satisfy the frame property. Can you figure out a counter-example?

It is nevertheless possible to define **triple** on top of **hoare** by “baking in” the frame rule. The relevant definition is as follows.

Definition **btriple** (*t*:**trm**) (*H*:**hprop**) (*Q*:**val**→**hprop**) : **Prop** :=

 ∀ (*H'*:**hprop**), **hoare** *t* (*H* * *H'*) (*Q* *+ *H'*).

This definition inherently satisfies the frame rule, as we show below. The proof essentially exploits the associativity of the star operator.

Exercise: 3 stars, standard, especially useful (btriple_frame) Prove that $\text{btriple } t \ H \ Q$ is a judgment satisfying the frame rule.

Lemma `btriple_frame` : $\forall t \ H \ Q \ H'$,

$\text{btriple } t \ H \ Q \rightarrow$
 $\text{btriple } t \ (H \setminus^* H') \ (Q \setminus^{*+} H')$.

Proof using. *Admitted*.

□

We have previously defined `btriple` on top of `hoare`, with the help of the separating conjunction operator, in the form: $\forall (H':\text{hprop}), \text{hoare } (H \setminus^* H') \ t \ (Q \setminus^{*+} H')$. In what follows, we give an equivalent characterization, expressed directly in terms of heaps and heap unions.

The alternative definition of $\text{triple } t \ H \ Q$ asserts that if $h1$ satisfies the precondition H and $h2$ describes the rest of the state, then the evaluation of t produces a value v in a final state made that can be decomposed between a part $h1'$ and $h2$ unchanged, in such a way that v and $h1'$ together satisfy the postcondition Q . Formally:

Definition `btriple_lowlevel` ($t:\text{trm}$) ($H:\text{hprop}$) ($Q:\text{val} \rightarrow \text{hprop}$) : `Prop` :=

$\forall h1 \ h2,$
 $\text{Fmap.disjoint } h1 \ h2 \rightarrow$
 $H \ h1 \rightarrow$
 $\exists h1' \ v,$
 $\text{Fmap.disjoint } h1' \ h2$
 $\wedge \text{beval } (h1 \setminus u \ h2) \ t \ (h1' \setminus u \ h2) \ v$
 $\wedge Q \ v \ h1'$.

Let us establish the equivalence between this alternative definition of `triple` and the original one.

Lemma `btriple_iff_btriple_lowlevel` : $\forall t \ H \ Q,$

$\text{btriple } t \ H \ Q \leftrightarrow \text{btriple\_lowlevel } t \ H \ Q$.

Proof using.

`unfold triple, btriple_lowlevel, hoare. iff M.`
`{ introv D P1.`
`forwards (h'&v&HR&HQ): M (=h2) (h1 \u h2). { applies× hstar_intro. }`
`destruct HQ as (h1'&h2'&N0&N1&N2&N3). subst.`
`∃ h1' v. auto. }`
`{ intros H' h. introv (h1&h2&N1&N2&D&U).`
`forwards (h1'&v&D'&HR&HQ): M h1 h2; auto. subst.`
`∃ (h1' \u h2) v. split. { eauto. } { applies× hstar_intro. } }`

`Qed.`

The low-level definition of `triple` leveraging the baked-in frame rule may be convenient for formalizing certain extensions of Separation Logic.

Finally, let us investigate how to derive a reasoning rule for triples defined using the baked-in frame rule, with respect to a standard big-step semantics. Consider `beval_let`, the

big-step evaluation rule for a let-binding.

Parameter *beval_let* : $\forall s1\ s2\ s3\ v1\ v2\ x\ t1\ t2,$
 beval *s1* *t1* *s2* *v1* $\rightarrow$
 beval *s2* (*subst* *x* *v1* *t2*) *s3* *v2* $\rightarrow$
 beval *s1* (*trm_let* *x* *t1* *t2*) *s3* *v2*.

Exercise: 3 stars, standard, especially useful (hoare_let) Prove the reasoning rule for let-bindings w.r.t. Hoare triples.

Lemma *hoare_let* : $\forall x\ t1\ t2\ H\ Q\ Q1,$
 hoare *t1* *H* *Q1* $\rightarrow$
 $(\forall v, \text{hoare } (\text{subst } x\ v\ t2)\ (Q1\ v)\ Q) \rightarrow$
 hoare (*trm_let* *x* *t1* *t2*) *H* *Q*.

Proof using. *Admitted*.

□

Exercise: 3 stars, standard, especially useful (btriple_let) Prove the reasoning rule for let-bindings w.r.t. Separation Logic triples defined using the baked-in frame rule.

Lemma *btriple_let* : $\forall x\ t1\ t2\ Q1\ H\ Q,$
 btriple *t1* *H* *Q1* $\rightarrow$
 $(\forall v1, \text{btriple } (\text{subst } x\ v1\ t2)\ (Q1\ v1)\ Q) \rightarrow$
 btriple (*trm_let* *x* *t1* *t2*) *H* *Q*.

Proof using. *Admitted*.

□

End BAKEDINFRAME.

41.4.6 Historical Notes

The definition of Separation Logic triples based on the use of an omni-big-step semantics is a contribution of *Charguéraud et al* 2023 (in Bib.v). Omni-big-step semantics was introduced by *Schäfer et al* 2016 (in Bib.v), with applications to a Hoare logic. The technique of the “baked-in frame rule” presented near the end of this chapter was introduced by *Birkedal, Torp-Smith and Yang* 2006 (in Bib.v), who developed the first Separation Logic for a higher-order programming language. Since then, this technique has been employed successfully in numerous formalizations of Separation Logic.

Chapter 42

Library SLF.Rules

42.1 Rules: Reasoning Rules for Term Constructs

```
Set Implicit Arguments.
From SLF Require Export LibSepReference.
From SLF Require Basic.

Implicit Types n : int.
Implicit Types t : trm.
Implicit Types r v : val.
Implicit Types p : loc.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
```

42.2 First Pass

This chapter establishes the remaining reasoning rules of Separation Logic: rules for reasoning about each language construct, and rules for reasoning about primitive operations. All these reasoning rules are expressed using the predicate **triple**, and are proved with respect to the formal semantics captured by the predicate **eval** introduced in the previous chapter (**Triples**). Throughout the chapter, one needs to have in mind the definition of **triple**:

Definition triple (t:trm) (H:hprop) (Q:val->hprop) : Prop := forall s, H s -> eval s t Q.

After establishing the reasoning rules, we present an example proof of a concrete program, using directly these reasoning rules, as opposed to using high-level “x-tactics” as in the first two chapters. This example proof reveals the proof patterns employed by the x-tactics. It also motivates the introduction of the x-tactics to avoid the tedious, repetitive proof patterns.

42.2.1 Rules for Term Constructors

Sequences

The Separation Logic reasoning rule for a sequence $t1;t2$ is essentially the same as that from Hoare logic. In the rule shown below, the underscore symbol indicates that the output value produced by $t1$ is irrelevant to the semantics, as captured by the rule `eval_seq`.

$$\frac{\{H\} t1 \{ \text{fun } \_ \Rightarrow H1 \} \{H1\} t2 \{Q\}}{\{H\} (t1;t2) \{Q\}}$$

The Rocq statement corresponding to the above rule is:

Lemma `triple_seq` : $\forall t1\ t2\ H\ Q\ H1,$
`triple` $t1\ H\ (\text{fun } \_ \Rightarrow H1) \rightarrow$
`triple` $t2\ H1\ Q \rightarrow$
`triple` $(\text{trm\_seq } t1\ t2)\ H\ Q.$

Exercise: 2 stars, standard, especially useful (`triple_seq`) Prove `triple_seq` by unfolding `triple` and using `eval_seq`.

Proof using. *Admitted.*

□

Let-Bindings

Next, we present the reasoning rule for let-bindings. Here again, there is nothing specific to Separation Logic: the rule is exactly the same as in Hoare Logic. The rule for a `let`-binding `let x = t1 in t2` could be stated, informally, in the form:

$$\{H\} t1 \{Q1\} (\text{forall } x, \{Q1\ x\} t2 \{Q\})$$

$$\frac{\{H\} (let\ x = t1\ in\ t2) \{Q\}}{\{H\} (let\ x = t1\ in\ t2) \{Q\}}$$

However, this presentation confuses the x that denotes a program variable in `let x = t1 in t2`, and the x that denotes a universally quantified Rocq value. The correct statement involves a substitution from the variable x to a value quantified as $\forall (v:\text{val})$.

$$\{H\} t1 \{Q1\} (\text{forall } v, \{Q1\ v\} (\text{subst } x\ v\ t2) \{Q\})$$

$$\frac{\{H\} (let\ x = t1\ in\ t2) \{Q\}}{\{H\} (let\ x = t1\ in\ t2) \{Q\}}$$

The corresponding Rocq statement is thus.

Lemma `triple_let` : $\forall x\ t1\ t2\ Q1\ H\ Q,$
`triple` $t1\ H\ Q1 \rightarrow$
 $(\forall v1, \text{triple } (\text{subst } x\ v1\ t2) (Q1\ v1)\ Q) \rightarrow$
`triple` $(\text{trm\_let } x\ t1\ t2)\ H\ Q.$

Proof using. *introv M1 M2 Hs. applies eval_let. Qed.*

When we carry out proofs in practice, the application of the rule *triple_let* produces a second sub-goal of the form, e.g.: $\forall v1, \text{triple}(\text{subst } "a" v1 \text{ rest\_of\_the\_program}) (\text{post1 } v1) \text{ post2}$, where “a” denotes the name of the program variable that was bound by the let-binding in the original source code. At this point, the user would typically type *intros a*, to introduce the Rocq variable *v1* under the same name that it had in the source code. This operations produces: $\text{triple}(\text{subst } "a" a \text{ rest\_of\_the\_program}) (\text{post1 } a) \text{ post2}$. At that point, the user may call *simpl* to effectively replace all occurrences of the program variable "a" with the Rocq variable *a* of type *val*. Doing so produces: $\text{triple } \text{rest\_of\_the\_program\_updated} (\text{post1 } a) \text{ post2}$, where *rest_of_the_program_updated* refers to the Rocq variable *a*. This process will be illustrated in the proof of *triple_incr* further on.

Conditionals

The rule for a conditional is, again, exactly like in Hoare logic.

$$b = \text{true} \rightarrow \{H\} t1 \{Q\} \quad b = \text{false} \rightarrow \{H\} t2 \{Q\}$$

$$\{H\} (\text{if } b \text{ then } t1 \text{ in } t2) \{Q\}$$

The corresponding Rocq statement is:

Lemma *triple_if_case* : $\forall b \ t1 \ t2 \ H \ Q,$
 $(b = \text{true} \rightarrow \text{triple } t1 \ H \ Q) \rightarrow$
 $(b = \text{false} \rightarrow \text{triple } t2 \ H \ Q) \rightarrow$
 $\text{triple } (\text{trm\_if } (\text{val\_bool } b) \ t1 \ t2) \ H \ Q.$

Exercise: 2 stars, standard, especially useful (*triple_if_case*) Prove *triple_if_case* by unfolding *triple* and using *eval_if*. Hint: use the tactic *case_if* to perform a case analysis.

Proof using. *Admitted*.

□

The two premises of the rule for if-statements may be factorized into a single one using Rocq’s conditional construct, as in *eval_if*.

Lemma *triple_if* : $\forall (b:\text{bool}) \ t1 \ t2 \ H \ Q,$
 $\text{triple } (\text{if } b \text{ then } t1 \text{ else } t2) \ H \ Q \rightarrow$
 $\text{triple } (\text{trm\_if } b \ t1 \ t2) \ H \ Q.$

Proof using. *intros M Hs. applies× eval_if. Qed.*

Values

The triple specifying the behavior of a value *v* can be written as a triple with an empty precondition and a postcondition asserting that the result value *r* is equal to *v*, in the empty heap. Formally:

$$\{\backslash \square\} v \ \{\text{fun } r \Rightarrow \backslash r = v\}$$

In practice, however, it is more convenient to work with a judgment whose conclusion has the form $\{H\} \text{ v } \{Q\}$, for an arbitrary precondition H and postcondition Q .

$H \implies Q \text{ v}$

$\{H\} \text{ v } \{Q\}$

The Rocq statement of the rule for values is thus as follows.

Lemma `triple_val` : $\forall v \ H \ Q$,

$H \implies Q \text{ v } \rightarrow$

`triple` (`trm_val` v) $H \ Q$.

Proof using. *introv* $M \ Hs$. *applies* $\times$ `eval_val`. Qed.

It may not be obvious at first sight why *triple_val* is equivalent to the “minimalistic” rule $\{\llbracket \cdot \rrbracket\} \text{ v } \{\text{fun } r \Rightarrow \llbracket r = v \rrbracket\}$. Let us prove the equivalence.

Exercise: 1 star, standard, especially useful (`triple_val_minimal`) Prove that the first rule for values is derivable from *triple_val*. Hint: use the tactic *xsimpl* to conclude the proof.

Lemma `triple_val_minimal` : $\forall v$,

`triple` (`trm_val` v) $\llbracket \cdot \rrbracket$ (`fun` $r \Rightarrow \llbracket r = v \rrbracket$).

Proof using. *Admitted*.

□

Exercise: 2 stars, standard, especially useful (`triple_val'`) More interestingly, prove that *triple_val* is derivable from `triple_val_minimal`. Concretely, your goal is to prove `triple_val'` without unfolding the definition of `triple`. Use *triple_conseq_frame*, `triple_val_minimal`, and *xsimpl*.

Lemma `triple_val'` : $\forall v \ H \ Q$,

$H \implies Q \text{ v } \rightarrow$

`triple` (`trm_val` v) $H \ Q$.

Proof using. *Admitted*.

□

Exercise: 4 stars, standard, especially useful (`triple_let_val`) Consider a term of the form `let x = v1 in t2`, that is, where the argument of the let-binding is already a value. State and prove a reasoning rule of the form:

Lemma `triple_let_val` : `forall x v1 t2 H Q, ... -> triple` (`trm_let` $x \ v1 \ t2$) $H \ Q$.

□

Function Definitions

Recall from the previous chapter the distinction between `val_fun x t1`, which denotes a closed value, and `trm_fun x t1`, which corresponds to a term that may contain free variables. By the time the evaluation of a program reaches a `trm_fun`, all its free variables should have been substituted by values. At this point, the rule `eval_fun` reduces the `trm_fun` into a `val_fun`.

We need a rule in the program logic to mimic this transition from `trm_fun` to `val_fun`. We have two ways to state this rule, similar to the two ways of stating the rule for reasoning about values. The first possibility is to consider a rule with an empty precondition.

$$\{\backslash\Box\} (\text{trm\_fun } x \ t1) \{\text{fun } r \Rightarrow \backslash r = \text{val\_fun } x \ t1\}$$

The second possibility, which leads to a more practical rule, features a conclusion of the form $\{H\} (\text{trm\_fun } x \ t1) \{Q\}$, with H and Q unconstrained. The resulting rule, *triple_fun* is structured similarly to *triple_val*.

Lemma *triple_fun* : $\forall x \ t1 \ H \ Q,$
 $H \implies Q \ (\text{val\_fun } x \ t1) \rightarrow$
 $\text{triple} (\text{trm\_fun } x \ t1) \ H \ Q.$

Proof using. *introv M Hs. applies eval_fun. Qed.*

This reasoning rules for functions generalizes to recursive functions. A term describing a recursive function is written `trm_fix f x t1`, and the corresponding value is written `val_fix f x t1`.

Lemma *triple_fix* : $\forall f \ x \ t1 \ H \ Q,$
 $H \implies Q \ (\text{val\_fix } f \ x \ t1) \rightarrow$
 $\text{triple} (\text{trm\_fix } f \ x \ t1) \ H \ Q.$

Proof using. *introv M Hs. applies eval_fix. Qed.*

Function Calls

Last but not least, we need a reasoning rule to reason about function applications. Consider an application `trm_app v1 v2`. Assume `v1` to be a function, that is, to be of the form `val_fun x t1`. Then, according to the beta-reduction rule, the semantics of `trm_app v1 v2` is the same as that of `subst x v2 t1`. This reasoning rule is thus:

$$v1 = \text{val\_fun } x \ t1 \ \{H\} \ (\text{subst } x \ v2 \ t1) \ \{Q\}$$

$$\{H\} (\text{trm\_app } v1 \ v2) \{Q\}$$

Observe that this reasoning rule requires `v1` to be a known value. In particular, `v1` cannot be a program variable or an unspecified program value. More precisely:

- If `v1` was a program variable (that is, a `trm_var`) in the original source code, then this program variable should have been substituted by a Rocq value or variable of type `val` by the time we reach this function call. Otherwise, the variable would correspond to a dangling free variable.

- If $v1$ is a Rocq variable of type **val** but with no assumption of the form $v1 = \text{val\_fun } x \ t1$, it is probably the case that we already have established a specification lemma for the function $v1$ earlier in the reasoning. In this case, the rule above generally cannot be applied. Instead, one should apply the specification lemma for $v1$, which is expected to have a conclusion of the form $\{..\} (\text{trm\_app } v1 \ ..) \{..\}$, matching the proof obligation at hand.

Let's come back to the case where we are interested in establishing a triple for a concrete function, whose code is known. The reasoning rule stated above translates in Rocq as shown below.

Lemma `triple_app_fun` : $\forall x \ v1 \ v2 \ t1 \ H \ Q$,
 $v1 = \text{val\_fun } x \ t1 \rightarrow$
 $\text{triple } (\text{subst } x \ v2 \ t1) \ H \ Q \rightarrow$
 $\text{triple } (\text{trm\_app } v1 \ v2) \ H \ Q$.

Proof using. `intros E M Hs. applies` $\times$ `eval_app_fun`. `Qed`.

The reasoning rule that corresponds to beta-reduction for a recursive function involves *two* substitutions: a first substitution for recursive occurrences of the function, followed with a second substitution for the argument provided to the call.

Lemma `triple_app_fix` : $\forall v1 \ v2 \ f \ x \ t1 \ H \ Q$,
 $v1 = \text{val\_fix } f \ x \ t1 \rightarrow$
 $\text{triple } (\text{subst } x \ v2 \ (\text{subst } f \ v1 \ t1)) \ H \ Q \rightarrow$
 $\text{triple } (\text{trm\_app } v1 \ v2) \ H \ Q$.

Proof using. `intros E M Hs. applies` $\times$ `eval_app_fix`. `Qed`.

This concludes the presentation of the reasoning rules for term constructs. Before we can tackle the verification of actual programs, there remains to present the specifications for the primitive operations. We start with arithmetic operations, then consider heap-manipulating operations.

42.2.2 Primitive Arithmetic Operations

Addition

Consider a term of the form `val_add n1 n2`, which is short for `trm_app (trm_app (trm_val val_add) (val_int n1)) (val_int n2)`, if we unfold the coercion `val_int`.

The addition operation may execute in an empty state. It does not modify the state, and returns the value `val_int (n1+n2)`. Recall the statement of the evaluation rule, in omni-big-step style.

Parameter `eval_add` : $\forall s \ n1 \ n2 \ Q$,
 $Q \ (\text{val\_int } (n1 + n2)) \ s \rightarrow$
 $\text{eval } s \ (\text{val\_add } (\text{val\_int } n1) \ (\text{val\_int } n2)) \ Q$.

In the specification shown below, the precondition is written $\llbracket \cdot \rrbracket$ and the postcondition binds a return value `r` of type **val** specified to be equal to `val_int (n1+n2)`.

```

Lemma triple_add :  $\forall$   $n1$   $n2$ ,
  triple (val_add  $n1$   $n2$ )
    \[]
    (fun  $r \Rightarrow \backslash[r = \text{val\_int } (n1 + n2)]$ ).

```

Proof using.

```

  introv  $Hs$ .  $\text{applies} \times \text{eval\_add}$ .  $\text{lets } ->: \text{hempty\_inv } Hs$ .
   $\text{applies}$  hpure_intro. auto.

```

Qed.

Technically, for a simple function such as addition, we could state a specification in a similar form as *triple_val*, with an entailment of the form $H \implies Q \ v$, as shown below.

```

Lemma triple_add' :  $\forall$   $H$   $Q$   $n1$   $n2$ ,
   $H \implies Q$  (val_int ( $n1 + n2$ ))  $\rightarrow$ 
  triple (val_add  $n1$   $n2$ )  $H$   $Q$ .

```

Proof using.

```

  introv  $M$ .  $\text{applies}$  triple_conseq_frame.
  {  $\text{applies}$  triple_add. }
  {  $\text{xsimpl}$ . }
  {  $\text{xpull}$ . intros ?  $\rightarrow$ . auto. }

```

Qed.

Yet, this pattern does not generalize well to more complex functions with nonempty preconditions. Instead, for specifying functions, we follow by convention the pattern of “minimal footprint specifications”, where the precondition of the function covers no more than what the function needs to access. For example:

- the *add* operation does not access the memory state, so it should be specified with \[] as precondition;
- the *get* operation on a reference accesses exactly one cell, so it should be specified using a precondition of the form $p \rightsquigarrow v$;
- the *mlist_length* operation traverses one mutable linked list, so it should be specified using a precondition of the form $\text{MList } p \ L$.

There are a few situations where the precondition of a function may be an over-approximation, to avoid complications. For example, a function that gives the index of the first occurrence of a value v in a mutable list at address p would typically be specified with a precondition of the form $\text{MList } p \ L$ describing the full linked list, rather than a precondition describing the the smallest prefix of the list that reaches a cell with value v .

Division

The specification of the division operation *val_div* $n1$ $n2$ is similar, yet with the extra requirement that the argument $n2$ must be nonzero. This requirement $n2 \neq 0$ appears in the evaluation rule as well as in the reasoning rule.

proves $(p \rightsquigarrow v) (Fmap.single\ p\ v)$. Also, the lemma `hstar_hpure_l` establishing $(\backslash[P] \backslash^* H)\ h = (P \wedge H\ h)$ will be used pervasively.

Allocation

Recall that `val_ref v` allocates a cell with contents `v`. A call to `val_ref v` does not depend on the current contents of the state. The operation extends the state with a fresh singleton cell, at some location `p`, assigning it `v` as contents. This fresh cell is described by the heap predicate $p \rightsquigarrow v$. The evaluation of `val_ref v` produces the value `val_loc p`. Thus, if `r` denotes the result value, we have $r = val\_loc\ p$ for some `p`. In the corresponding specification shown below, observe how the location `p` is existentially quantified in the postcondition.

```
Lemma triple_ref : ∀ v,
  triple (val_ref v)
    \[]
    (fun r => \∃ p, \[r = val_loc p] \* p \rightsquigarrow v).
```

Proof using.

```
intros. intros s1 K. applys eval_ref. intros p D.
lets ->: hempty_inv K. rewrite Fmap.update_empty.
applys hexists_intro p. rewrite hstar_hpure_l. split×.
applys hsingle_intro.
```

Qed.

Using the notation $funloc\ p \Rightarrow H$ as a shorthand for $fun\ (r:val) \Rightarrow \backslash\exists\ (p:loc), \backslash[r = val\_loc\ p] \backslash^* H$, the specification for `val_ref` becomes more concise.

```
Lemma triple_ref' : ∀ v,
  triple (val_ref v)
    \[]
    (funloc p => p \rightsquigarrow v).
```

Proof using. `apply triple_ref`. Qed.

Deallocation

Recall that `val_free` denotes the operation for deallocating a cell at a given address. A call of the form `val_free p` executes safely in a state described by $p \rightsquigarrow v$. The operation leaves an empty state, and asserts that the return value, named `r`, is equal to unit.

Exercise: 2 stars, standard, especially useful (`triple_free'`) Prove the reasoning rule for deallocation of a single cell. Hint: use `Fmap.indom_single` and `Fmap.remove_single`.

```
Lemma triple_free' : ∀ p v,
  triple (val_free (val_loc p))
    (p \rightsquigarrow v)
    (fun r => \[r = val_unit]).
```

Proof using. *Admitted*.

□

In practice, the information $r = \text{val\_unit}$ is generally useless. Thus, we state `triple_free` with an empty postcondition.

```
Lemma triple_free : ∀ p v,
  triple (val_free (val_loc p))
    (p ~~> v)
    (fun _ => \[]).
```

Proof using. `intros. applies triple_conseq triple_free'; xsimpl`×. Qed.

Read

Recall that `val_get` denotes the operation for reading a memory cell. A call of the form `val_get v'` executes safely if its argument v' is a value of the form `val_loc p` for some location p , in a state that features a memory cell at location p , storing some contents v . Such a state is described as $p \text{ ~~> } v$. The read operation returns a value r such that $r = v$, and the memory state of the operation remains unchanged. The specification of `val_get` is thus expressed as follows.

```
Lemma triple_get : ∀ v p,
  triple (val_get p)
    (p ~~> v)
    (fun r => \[r = v] \* (p ~~> v)).
```

Proof using.

```
  intros. intros s K. lets ->: hsingle_inv K. applies eval_get.
  { applies× Fmap.indom_single. }
  { rewrite hstar_hpure_l. split×. rewrite× Fmap.read_single. }
```

Qed.

Write

Recall that `val_set` denotes the operation for writing a memory cell. A call of the form `val_set v' w` executes safely if v' is of the form `val_loc p` for some location p , in a state $p \text{ ~~> } v$. The write operation updates this state to $p \text{ ~~> } w$, and returns the unit value, which can be ignored. Hence, `val_set` is specified as follows.

Exercise: 2 stars, standard, especially useful (`triple_set`) Prove the reasoning rule for a write operation. Hint: use `Fmap.update_single`.

```
Lemma triple_set : ∀ w p v,
  triple (val_set (val_loc p) v)
    (p ~~> w)
    (fun r => \[r = val_unit] \* (p ~~> v)).
```

Proof using. *Admitted*.

□

42.2.4 Program Verification using the Reasoning Rules

We now have at hand all the necessary rules for carrying out actual verification proofs in Separation Logic.

```
Module EXAMPLEPROGRAMS.  
Export PROGRAMSYNTAX.
```

Proof of *incr*

First, we consider the verification of the increment function, which is written in OCaml syntax as:

OCaml:

```
let incr p = p := !p + 1
```

and in “A-normal form” as:

OCaml:

```
let incr p = let n = !p in let m = n+1 in p := m
```

Using the construct from our embedded programming language, the definition of *incr* is formalized as shown below.

Definition *incr* : **val** :=

```
val_fun "p" (  
  trm_let "n" (trm_app val_get (trm_var "p")) (  
    trm_let "m" (trm_app (trm_app val_add  
      (trm_var "n")) (val_int 1)) (  
        trm_app (trm_app val_set (trm_var "p")) (trm_var "m")))).
```

Alternatively, using notation and coercions, the same program can be written more concisely.

Definition *incr'* : **val** :=

```
<{ fun 'p =>  
  let 'n = ! 'p in  
  let 'm = 'n + 1 in  
  'p := 'm }>.
```

Let us check that the two definitions are indeed the same.

Lemma *incr_eq_incr'* :

```
incr = incr'.
```

Proof using. *reflexivity*. Qed.

Recall from the first chapter the specification of the increment function. Its precondition requires a singleton state of the form $p \sim\sim> n$. Its postcondition describes a state of the form $p \sim\sim> (n+1)$.

```
Lemma triple_incr :  $\forall (p:loc) (n:int),$   
triple (trm_app incr p)  
(p  $\sim\sim> n$ )
```

(`fun _ => p ~> (n+1)`).

In chapter [Basic](#), we have seen that the proof of this lemma can be completed by the short script `xwp. xapp. xapp. xapp. xsimpl..` The purpose of the detailed proof that follows is to explain how the x-tactics operate. It shows how, under the hood, the proof exploits:

- the structural reasoning rules,
- the reasoning rules for terms,
- the specification of the primitive functions,
- the `xsimpl` tactic for simplifying entailments.

Executing the proof script step by step leads to the display of intermediate proof obligations that involve Rocq existential variables, a.k.a. “evars”, whose name is prefixed by a question mark, e.g., “?x”. These placeholders are typically introduced by `applys` (which calls `eapply`), and are typically instantiated when solving subgoals.

Proof using.

Initialize the proof `intros. applys triple_app_fun.`

Provide the argument name and body for `incr`. `{ reflexivity. }`

Compute the substitution from program variable “p” to Rocq variable `p` `simpl.`

Reason about `let n = ...`. Observe that this introduces an evvar `?Q1`, which denotes the postcondition of `!p`. `applys triple_let.`

In the body of the let-binding, reason about `!p`. Here we are in a favorable situation, where there is no need to apply a consequence or frame rule. `{ apply triple_get. }`

Observe how `?Q1` has been instantiated as `fun r => \[r = n] \* p ~> n`. We next introduce a name `n'` for the result of `!p` `intros n'.`

Once again, we compute the substitution of a program variable. `simpl.`

We substitute away the equality `n' = n`. `apply triple_hpure. intros ->.`

Reason about `let m = ...`. Observe the introduction of another evvar `?Q1`.

`applys triple_let.`

Apply the frame rule to put aside `p ~> n`. `{ applys triple_conseq_frame.`

We reason about `n+1` in the empty state. Doing so instantiates `?Q1`. `{ applys triple_add. }`

`{ xsimpl. }`

`{ xsimpl. }`

We name `m'` the result of `n+1`. `intros m'. simpl.`

We can extract the equality `m' = n + 1` and substitute it away. `apply triple_hpure. intros ->.`

Finally, we reason about `p := m`. `applys triple_conseq_frame.`

`{ applys triple_set. }`

`{ xsimpl. }`

`{ xsimpl. }`

Qed.

Proof of `succ_using_incr`

Recall from `Basic` the function `succ_using_incr`.

Definition `succ_using_incr` : `val` :=

```
<{ fun 'n =>
  let 'r = val_ref 'n in
  incr 'r;
  let 'x = ! 'r in
  val_free 'r;
  'x }>.
```

Recall the specification of `succ_using_incr`.

Lemma `triple_succ_using_incr` : $\forall (n:\text{int}),$
`triple (trm_app succ_using_incr n)`
 $\backslash []$
 $(\text{fun } v \Rightarrow \backslash [v = \text{val\_int } (n+1)]).$

Exercise: 4 stars, standard, especially useful (`triple_succ_using_incr`) Verify the function `triple_succ_using_incr`. Hint: follow the pattern of `triple_incr`. Use *applies* `triple_seq` for reasoning about a sequence. Use *applies* `triple_val` for reasoning about the final return value, namely `x`.

Proof using. *Admitted.*

□

End `EXAMPLEPROGRAMS`.

The job of the following chapters is to introduce additional technology to streamline the proof process, notably by:

- automating the application of the frame rule, and
- eliminating the need to manipulate program variables and substitutions during verification.

42.3 More Details

This “More Details” section presents several additional reasoning rules.

42.3.1 Triple for Terms with Same Semantics

Module `PROOFSSAMESEMANTICS`.

A general principle is that if, according to the semantics, the terms `t1` and `t2` admit the same behaviors, then `t1` and `t2` satisfy the same triples. The statement can also be

reformulated in an asymmetric fashion: if $t2$ admits no more behaviors than $t1$, then $t2$ satisfies all the triples that $t1$ satisfies. This section explains how to formalize this asymmetric statement, which is more general and more useful in practice.

The judgment `eval_like t1 t2` asserts that if $t1$ terminates and produces results in Q , then so does $t2$.

Definition `eval_like (t1 t2:trm) : Prop :=`
 $\forall s\ Q, \text{eval } s\ t1\ Q \rightarrow \text{eval } s\ t2\ Q.$

Exercise: 1 star, standard, especially useful (eval_like_eta_expansion) Prove that when the relation `eval_like t1 t2` holds, any triple that $t2$ satisfies any triple that $t1$ satisfies.

Lemma `triple_eval_like :  $\forall t1\ t2\ H\ Q,$`
`eval_like t1 t2  $\rightarrow$`
`triple t1 H Q  $\rightarrow$`
`triple t2 H Q.`

Proof using. *Admitted.*

□

The remaining of this section presents 4 examples applications of this reasoning principle. To begin with, consider the term `let x = t in x`, represented as `trm_let x t x` for a variable named x . This term reduces to t , hence the relation `eval_like t (trm_let x t x)` holds.

Lemma `eval_like_eta_reduction :  $\forall (t:\text{trm})\ (x:\text{var}),$`
`eval_like t (trm_let x t x).`

Proof using.

introv R. applies eval_let R.
`simpl. rewrite var_eq_spec. case_if.`
introv Hv. applies eval_val Hv.

Qed.

Exercise: 4 stars, standard, optional (eval_like_eta_expansion) Prove that the symmetric relation `eval_like (trm_let x t x) t` also holds.

Lemma `eval_like_eta_expansion :  $\forall (t:\text{trm})\ (x:\text{var}),$`
`eval_like (trm_let x t x) t.`

Proof using. *Admitted.*

□

Exercise: 2 stars, standard, especially useful (eta_same_triples) Conclude that t and `trm_let x t x` satisfy exactly the same set of triples. (This result will be exploited in chapter *Affine*, for proving the equivalence of two versions of the garbage collection rule.)

Lemma `eta_same_triples :  $\forall (t:\text{trm})\ (x:\text{var})\ H\ Q,$`
`triple t H Q  $\leftrightarrow$  triple (trm_let x t x) H Q.`

Proof using. *Admitted.*

□

Another application of the reasoning rule `triple_eval_like` is to revisit the proof of `triple_app_fun` by arguing that `trm_app (val_fun x t1) v2` reduces to `subst x v2 t1`, hence they admit the same behavior.

Lemma `triple_app_fun` : $\forall x\ v1\ v2\ t1\ H\ Q,$
 $v1 = \text{val\_fun } x\ t1 \rightarrow$
 $\text{triple } (\text{subst } x\ v2\ t1)\ H\ Q \rightarrow$
 $\text{triple } (\text{trm\_app } v1\ v2)\ H\ Q.$

Proof using.

introv E M1. applies triple_eval_like M1.

introv R. applies eval_app_fun E R.

Qed.

A even more interesting application is a rule that allows to modify the parenthesis structure of a sequence, from `t1; (t2; t3)` to `(t1;t2); t3`. Such a change in the parenthesis structure of a sequence might be helpful to apply the frame rule around `t1;t2`, for example.

Exercise: 3 stars, standard, optional (`triple_trm_seq_assoc`) Prove that the term `t1; (t2; t3)`, which corresponds to the natural parsing of `t1; t2; t3`, satisfies any triple that the term `(t1;t2); t3` satisfies.

Lemma `triple_trm_seq_assoc` : $\forall t1\ t2\ t3\ H\ Q,$
 $\text{triple } (\text{trm\_seq } (\text{trm\_seq } t1\ t2)\ t3)\ H\ Q \rightarrow$
 $\text{triple } (\text{trm\_seq } t1\ (\text{trm\_seq } t2\ t3))\ H\ Q.$

Proof using. *Admitted.*

□

End PROOFSSAMESEMANTICS.

42.3.2 The Combined Let-Frame Rule

Module `LETFRAME`.

Recall the Separation Logic let rule.

Parameter `triple_let` : $\forall x\ t1\ t2\ Q1\ H\ Q,$
 $\text{triple } t1\ H\ Q1 \rightarrow$
 $(\forall v1, \text{triple } (\text{subst } x\ v1\ t2)\ (Q1\ v1)\ Q) \rightarrow$
 $\text{triple } (\text{trm\_let } x\ t1\ t2)\ H\ Q.$

At first sight, it seems that, to reason about `let x = t1 in t2` in a state described by precondition `H`, we need to first reason about `t1` in that same state. But `t1` may well require only a subset of the state `H` to evaluate.

The “let-frame” rule combines the rule for let-bindings with the frame rule to make it more explicit that the precondition `H` may be decomposed in the form `H1 \* H2`, where `H1` is the part needed by `t1`, and `H2` denotes the rest of the state. The part of the state

covered by $H2$ remains unmodified during the evaluation of $t1$, and appears as part of the precondition of $t2$. The corresponding statement is as follows.

Lemma `triple_let_frame` : $\forall x\ t1\ t2\ Q1\ H\ H1\ H2\ Q,$
 $\text{triple } t1\ H1\ Q1 \rightarrow$
 $H \Rightarrow H1 \ \backslash * \ H2 \rightarrow$
 $(\forall v1, \text{triple } (\text{subst } x\ v1\ t2) (Q1\ v1\ \backslash * \ H2)\ Q) \rightarrow$
 $\text{triple } (\text{trm\_let } x\ t1\ t2)\ H\ Q.$

Exercise: 3 stars, standard, especially useful (`triple_let_frame`) Prove the let-frame rule.

Proof using. *Admitted.*

□

End LETFRAME.

42.4 Optional Material

42.4.1 Alternative Specification Style for Pure Preconditions

Module DIVSPEC.

Recall the specification for division.

Parameter `triple_div` : $\forall n1\ n2,$
 $n2 \neq 0 \rightarrow$
 $\text{triple } (\text{val\_div } n1\ n2)$
 $\ \backslash []$
 $(\text{fun } r \Rightarrow \backslash [r = \text{val\_int } (\text{Z.quot } n1\ n2)])$.

Equivalently, we could place the requirement $n2 \neq 0$ in the precondition:

Parameter `triple_div'` : $\forall n1\ n2,$
 $\text{triple } (\text{val\_div } n1\ n2)$
 $\ \backslash [n2 \neq 0]$
 $(\text{fun } r \Rightarrow \backslash [r = \text{val\_int } (\text{Z.quot } n1\ n2)])$.

The two presentations are equivalent. First, let us prove `triple_div'` using `triple_div`.

Lemma `triple_div'_from_triple_div` : $\forall n1\ n2,$
 $\text{triple } (\text{val\_div } n1\ n2)$
 $\ \backslash [n2 \neq 0]$
 $(\text{fun } r \Rightarrow \backslash [r = \text{val\_int } (\text{Z.quot } n1\ n2)])$.

Proof using.

`intros. applies triple_hpure'. applies triple_div.`

`Qed.`

Exercise: 2 stars, standard, especially useful (triple_div_from_triple_div') Prove *triple_div* using *triple_div'*.

```
Lemma triple_div_from_triple_div' : ∀ n1 n2,
  n2 ≠ 0 →
  triple (val_div n1 n2)
    \ []
    (fun r ⇒ \[r = val_int (Z.quot n1 n2)]).
```

Proof using. *Admitted*.

□

Placing pure preconditions outside of the triples makes it slightly more convenient to exploit specifications. For this reason, we adopt the style that precondition only contain the description of heap-allocated data structures.

End DIVSPEC.

42.4.2 Alternative Specification Style for Result Values

Module MATCHSTYLE.

Recall the specification for the function *ref*.

```
Parameter triple_ref : ∀ v,
  triple (val_ref v)
    \ []
    (fun r ⇒ \∃ p, \[r = val_loc p] \* p ~~> v).
```

Its postcondition could be equivalently stated by using, instead of an existential quantifier, a pattern matching construct.

```
Parameter triple_ref' : ∀ v,
  triple (val_ref v)
    \ []
    (fun r ⇒ match r with
      | val_loc p ⇒ (p ~~> v)
      | _ ⇒ \[False]
    end).
```

However, the pattern-matching presentation is less readable and would be fairly cumbersome to work with in practice.

End MATCHSTYLE.

Proof of factorec

This section presents a manual proof for a recursive function.

Module EXAMPLEPROGRAMS2.

Export PROGRAMSYNTAX.

Import BASIC.FACTO.

In this section, the corollary *triple_hpure'* established in the previous chapter, can be useful.

Parameter *triple_hpure'* : $\forall t (P:\text{Prop}) Q,$
 $(P \rightarrow \text{triple } t \setminus [] Q) \rightarrow$
 $\text{triple } t \setminus [P] Q.$

Recall the function `repeat_incr` from chapter `Basic`.

OCaml:

```
let rec factorec n = if n <= 1 then 1 else n * factorec (n-1)
```

Definition `factorec` : **val** :=

```
<{ fix 'f 'n ⇒
  let 'b = ('n ≤ 1) in
  if 'b
  then 1
  else let 'x = 'n - 1 in
        let 'y = 'f 'x in
        'n × 'y }>.
```

Exercise: 4 stars, standard, especially useful (`triple_factorec`) Verify the function `factorec`. Hint: exploit `triple_app_fix` for reasoning about the recursive function. Use *triple_hpure'*, the corollary of `triple_hpure`. Exploit `triple_le` and `triple_sub` and `triple_mul` to reason about the behavior of the primitive operations involved. Exploit *applies triple_if.case_if as C*. to reason about the conditional; alternatively, if using `triple_if_case`, you'll need to use the tactic *rew_bool_eq in ** to simplify, e.g., the expression `isTrue (m ≤ 1)` = `true`.

Lemma `triple_factorec` : $\forall n,$
 $n \geq 0 \rightarrow$
 $\text{triple } (\text{factorec } n)$
 $\setminus []$
 $(\text{fun } r \Rightarrow \setminus [r = \text{facto } n]).$

Proof using. *Admitted*.

□

End EXAMPLEPROGRAMS2.

42.4.3 Historical Notes

Gordon 1989 (in Bib.v) gave the first mechanization of Hoare logic in a proof assistant, using the HOL tool. Gordon's pioneering work was followed by numerous formalizations of Hoare logic, targeting various programming languages.

The original presentation of Separation Logic (1999-2001) consists of a set of rules written down on paper. These rules were not formally described in a proof assistant. Nevertheless, mechanized presentation of Separation Logic emerged a few years later.

Yu, Hamid, and Shao 2003 (in Bib.v) present the CAP framework for the verification in Coq of assembly-level code. This framework exploits separation logic style specifications, with predicate for lists and list segments involving the separating conjunction operator.

In parallel, *Weber* 2004 (in Bib.v), advised by Nipkow, developed the first mechanization of the rules of Separation Logic for a while language, using the Isabelle/HOL tool. His presentation is quite close from the original, paper presentation.

Numerous mechanized presentations of Separation Logic, targeting various languages (assembly, C, core-Java, ML, etc.) and using various tools (Isabelle/HOL, Coq/Rocq, PVS, HOL4, HOL). For a detailed list, see the last chapter of the companion notes, linked from the *Preface*.

Chapter 43

Library `SLF.Wand`

43.1 Wand: The Magic Wand and Other Operators

```
Set Implicit Arguments.
From SLF Require Import LibSepReference.
From SLF Require Repr.
Close Scope trm_scope.

Implicit Types h : heap.
Implicit Types P : Prop.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
```

43.2 First Pass

This chapter presents some additional Separation Logic operators:

- the disjunction, written `hor H1 H2`,
- the non-separating conjunction, written `hand H1 H2`,
- the “forall”, named `hforall` and written `\forall x, H`,
- the “magic wand”, named `hwand` and written `H1 \-* H2`,
- the “magic wand for postconditions”, named `qwand` and written `Q1 \-* Q2`.

The magic wand operator has multiple uses:

- it is key to formulating the “ramified frame rule”, a more practical rule for exploiting the frame and consequence properties,
- it will be used in the chapter `WPgen` to define a weakest-precondition generator, and

- it can be useful to state the specifications for certain data structures.

This chapter is organized as follows:

- definition and properties of `hand`, `hor`, and `hforall`,
- definition and properties of the magic wand operator,
- generalization of the magic wand to postconditions,
- extension of the `xsimpl` tactic to handle the magic wand,
- statement and benefits of the ramified frame rule.

The “optional material” presents several equivalent definitions for magic wand.

43.2.1 Definition of the Operator `hforall`

Module `HFORALL`.

The heap predicate $\backslash \forall \mathbf{x}, H$ holds of a heap `h` that, for any possible value of `x`, satisfies `H`. It may be puzzling at first what could possibly be the use case for such a universal quantification, but we will see several examples through this chapter – in particular the encoding of the non-separating conjunction (`hand`) and of the magic wand on postconditions (`qwand`).

The predicate $\backslash \forall \mathbf{x}, H$ stands for `hforall (fun x => H)`, where the definition of `hforall` follows the same pattern as `hexists`.

Definition `hforall` ($A : \text{Type}$) ($J : A \rightarrow \text{hprop}$) : `hprop` :=
`fun h =>  $\forall x, J x h$ .`

Notation `"'\forallforall' x1 .. xn , H"` :=
`(hforall (fun x1 => .. (hforall (fun xn => H)) ..))`
`(at level 39, x1 binder, H at level 50, right associativity,`
`format "'[' '\forallforall' '/' ' x1 .. xn , '/' ' H ']'")`.

To follow the rest of this chapter, it suffices to have in mind the introduction and elimination rules for `hforall`.

Lemma `hforall_intro` : $\forall A (J:A \rightarrow \text{hprop}) h,$
 $(\forall x, J x h) \rightarrow$
 $(\text{hforall } J) h.$

Proof using. `introv M. applies  $\times$  M. Qed.`

Lemma `hforall_inv` : $\forall A (J:A \rightarrow \text{hprop}) h,$
 $(\text{hforall } J) h \rightarrow$
 $\forall x, J x h.$

Proof using. `introv M. applies  $\times$  M. Qed.`

The introduction rule in an entailment for $\backslash\forall$ appears below. To prove that a heap satisfies $\backslash\forall x, J x$, we must show that, for any x , this heap satisfies $J x$.

Lemma `himpl_hforall_r` : $\forall A (J:A \rightarrow \text{hprop}) H$,

$(\forall x, H \Rightarrow J x) \rightarrow$

$H \Rightarrow (\backslash\forall x, J x)$.

Proof using. `intros M. intros h K x. apply  $\neg$  M. Qed.`

The elimination rule in an entailment for $\backslash\forall$ appears below. Assuming a heap satisfies $\backslash\forall x, J x$, we can derive that the same heap satisfies $J v$ for any desired value v .

Lemma `hforall_specialize` : $\forall A (v:A) (J:A \rightarrow \text{hprop})$,

$(\backslash\forall x, J x) \Rightarrow (J v)$.

Proof using. `intros. intros h K. apply  $\times$  K. Qed.`

The lemma `hforall_specialize` can equivalently be formulated in the following way, which makes it easier to apply in some cases.

Lemma `himpl_hforall_l` : $\forall A (v:A) (J:A \rightarrow \text{hprop}) H$,

$J v \Rightarrow H \rightarrow$

$(\backslash\forall x, J x) \Rightarrow H$.

Proof using. `intros M. applys himpl_trans_r M. applys hforall_specialize. Qed.`

Universal quantifiers that appear in the precondition of a triple may be specialized like universal quantifiers appearing on the left-hand side of an entailment.

Lemma `triple_hforall` : $\forall A (v:A) t (J:A \rightarrow \text{hprop}) Q$,

`triple` $t (J v) Q \rightarrow$

`triple` $t (\backslash\forall x, J x) Q$.

Proof.

`intros M. applys triple_conseq M.`

`{ applys hforall_specialize. }`

`{ applys qimpl_refl. }`

Qed.

End `HFORALL`.

43.2.2 Definition of the Operator `hor`

Module `HOR`.

The heap predicate `hor` $H1 H2$ describes a heap that satisfies $H1$ or satisfies $H2$ (or possibly both). In other words, the heap predicate `hor` $H1 H2$ lifts the disjunction operator $P1 \vee P2$ from `Prop` to `hprop`.

The disjunction operator does not appear to be critically useful in practice, because it can be encoded using Rocq's conditional construct, or using pattern matching. Nevertheless, there are situations where it proves handy.

The heap disjunction predicate admits a direct definition as a function over heaps, written `hor'`.

Definition hor' (*H1 H2* : hprop) : hprop :=
 fun *h* => *H1 h* ∨ *H2 h*.

An alternative definition leverages the $\exists$ quantifier. The definition, shown below, reads as follows: “there exists an unspecified boolean value *b* such that if *b* is true then *H1* holds, else if *b* is false then *H2* holds”.

The benefit of this definition is that its properties can be established without manipulating heaps explicitly.

Definition hor (*H1 H2* : hprop) : hprop :=
 $\exists (b:\text{bool}), \text{if } b \text{ then } H1 \text{ else } H2$.

Exercise: 3 stars, standard, optional (hor_eq_hor') Prove the equivalence of the definitions hor and hor'.

Lemma hor_eq_hor' :
 hor = hor'.

Proof using. *Admitted*.

□

The two introduction rules for hor assert that if *H1* holds, then hor *H1 H2* holds; and, symmetrically, if *H2* holds, then hor *H1 H2* holds.

Lemma himpl_hor_r_l : $\forall H1 H2$,
H1 ==> hor *H1 H2*.

Proof using. intros. *unfolds* hor. $\exists \times$ true. Qed.

Lemma himpl_hor_r_r : $\forall H1 H2$,
H2 ==> hor *H1 H2*.

Proof using. intros. *unfolds* hor. $\exists \times$ false. Qed.

In practice, these rules are easier to exploit when combined with a transitivity step.

Lemma himpl_hor_r_l_trans : $\forall H1 H2 H3$,
H3 ==> *H1* →
H3 ==> hor *H1 H2*.

Proof using. intro *W*. *applies* himpl_trans *W*. *applies* himpl_hor_r_l. Qed.

Lemma himpl_hor_r_r_trans : $\forall H1 H2 H3$,
H3 ==> *H2* →
H3 ==> hor *H1 H2*.

Proof using. intro *W*. *applies* himpl_trans *W*. *applies* himpl_hor_r_r. Qed.

The elimination rule asserts that, if hor *H1 H2* holds, then we can perform a case analysis on whether it is *H1* or *H2* that holds. Concretely, to show that hor *H1 H2* entails a heap predicate *H3*, we must show both that *H1* entails *H3* and that *H2* entails *H3*.

Lemma himpl_hor_l : $\forall H1 H2 H3$,
H1 ==> *H3* →
H2 ==> *H3* →

```

  hor H1 H2 ==> H3.
Proof using.
  introv M1 M2. unfolds hor. applies himpl_hexists_l. intros b. case_if ×.
Qed.

```

The operator `hor` is commutative. To establish this property, it is useful to exploit the following lemma, called `if_neg`, for swapping the two branches of a conditional by negating its boolean condition.

```

Lemma if_neg : ∀ (b:bool) A (X Y:A),
  (if b then X else Y) = (if neg b then Y else X).
Proof using. intros. case_if ×. Qed.

```

Exercise: 2 stars, standard, especially useful (`hor_comm`) Prove that `hor` is a symmetric operator. Hint: exploit `hprop_op_comm` and `if_neg` (from chapter `Himpl`).

```

Lemma hor_comm : ∀ H1 H2,
  hor H1 H2 = hor H2 H1.
Proof using. Admitted.

```

□

```

Module HOREXAMPLE.
Import REPR.
Implicit Types q : loc.

```

Exercise: 4 stars, standard, especially useful (`hor_comm`) Prove that the representation predicate `MList` introduced in chapter `Repr` can be equivalently characterized using the predicate `hor`, as shown below. Hint: to prove this equivalence, do not attempt a proof by induction, and do not attempt to unfold `MList`. Instead, work using the equalities `MList_nil` and `MList_cons`. You may want to begin the proof by applying `himpl_antisym` and calling `destruct` on `L`.

```

Lemma MList_using_hor : ∀ L p,
  MList L p =
    hor ( \ [L = nil ∧ p = null] )
      ( \ ∃ x q L', \ [L = x :: L']
        \* (p ~~~> '{ head := x; tail := q })
        \* (MList L' q) ).

```

Proof using. *Admitted*.

□

```

End HOREXAMPLE.
End HOR.

```

43.2.3 Definition of the Operator `hand`

Module HAND.

Import HFORALL HOR.

The heap predicate `hand H1 H2` describes a single heap that satisfies `H1` and at the same time satisfies `H2`. In other words, the heap predicate `hand H1 H2` lifts the disjunction operator `P1 ∧ P2` from `Prop` to `hprop`. The heap predicate `hand` admits a direct definition as a function over heaps.

Definition `hand'` (`H1 H2 : hprop`) : `hprop` :=
`fun h => H1 h ∧ H2 h.`

An alternative definition leverages the $\forall$ quantifier. The definition reads as follows: “for any boolean value `b`, if `b` is true then `H1` should hold, and if `b` is false then `H2` should hold”.

Definition `hand` (`H1 H2 : hprop`) : `hprop` :=
 $\forall (b:\text{bool}), \text{ if } b \text{ then } H1 \text{ else } H2.$

Exercise: 2 stars, standard, especially useful (`hand_eq_hand'`) Prove the equivalence of the definitions `hand` and `hand'`. Use functional extensionality to get started.

Lemma `hand_eq_hand'` :
`hand = hand'.`

Proof using. *Admitted.*

□

The introduction and elimination rules for `hand` are as follows.

- If “`H1` and `H2`” holds, then in particular `H1` holds.
- Symmetrically, if “`H1` and `H2`” holds, then in particular `H2` holds.
- Reciprocally, to prove that a heap predicate `H3` entails “`H1` and `H2`”, we must prove that `H3` entails `H1`, and that `H3` satisfies `H2`.

Lemma `himpl_hand_l_r` : $\forall H1 H2,$
`hand H1 H2 ==> H1.`

Proof using. `intros. unfolds hand. applys × himpl_hforall_l true. Qed.`

Lemma `himpl_hand_l_l` : $\forall H1 H2,$
`hand H1 H2 ==> H2.`

Proof using. `intros. unfolds hand. applys × himpl_hforall_l false. Qed.`

Lemma `himpl_hand_r` : $\forall H1 H2 H3,$
`H3 ==> H1 →`
`H3 ==> H2 →`
`H3 ==> hand H1 H2.`

Proof using. `introv M1 M2 Hh. intros b. case_if ×. Qed.`

Exercise: 1 star, standard, especially useful (hand_comm) Prove that `hand` is symmetric. Hint: use `hprop_op_comm` and `rewrite if_neg`, or a case analysis on the boolean value coming from `hand`.

Lemma `hand_comm` : $\forall H1\ H2,$
 $\text{hand } H1\ H2 = \text{hand } H2\ H1.$

Proof using. *Admitted.*

□

End HAND.

43.2.4 Definition of the Magic Wand Operator: `hwand`

Module HWAND.

Definition of `hwand`

The magic wand operation $H1 \setminus^* H2$, pronounced “H1 wand H2”, defines a heap predicate such that, if we extend it with $H1$, we obtain $H2$. Formally, the following entailment holds:

$$H1 \setminus^* (H1 \setminus^* H2) \implies H2.$$

Intuitively, if we think of the star $H1 \setminus^* H2$ as a sort of addition, $H1 + H2$, then we can think of $H1 \setminus^* H2$ as the subtraction $-H1 + H2$. The entailment above intuitively captures the idea that $(-H1 + H2) + H1$ simplifies to $H2$.

Note, however, that the operation $H1 \setminus^* H2$ only makes sense if $H1$ describes a piece of heap that “can” be subtracted from $H2$. Otherwise, the predicate $H1 \setminus^* H2$ characterizes a heap that cannot possibly exist. Informally speaking, $H1$ must somehow be a subset of $H2$ for the subtraction $-H1 + H2$ to make sense.

Another possible analogy is that of logical operators. If $P1$ and $P2$ were propositions (of type `Prop`), then $P1 \setminus^* P2$ would mean $P1 \wedge P2$ and $P1 \setminus^* P2$ would mean $P1 \rightarrow P2$. The entailment $P1 \setminus^* (P1 \setminus^* P2) \implies P2$ then corresponds to the tautology $(P1 \wedge (P1 \rightarrow P2)) \rightarrow P2$.

Technically, $H1 \setminus^* H2$ holds of a heap h if, for any heap h' disjoint from h and satisfying $H1$, the union of h and h' satisfies $H2$. The operator `hwand`, which implements the notation $H1 \setminus^* H2$, may thus be defined as follows.

Definition `hwand'` ($H1\ H2:hprop$) : `hprop` :=
`fun h =>  $\forall h', \text{Fmap.disjoint } h\ h' \rightarrow H1\ h' \rightarrow H2\ (h \setminus u\ h').$`

The definition above is perfectly fine, however it is more practical to use an alternative, equivalent definition of `hwand`, expressed in terms of previously introduced Separation Logic operators. Doing so enables us to establish all the properties of the magic wand by exploiting the tactic `xsimpl`, thereby conducting all the reasoning at the level of `hprop`, rather than having to work with concrete heaps.

The alternative definition asserts that $H1 \setminus^* H2$ corresponds to some heap predicate, called $H0$, such that $H0$ starred with $H1$ yields $H2$. In other words, $H0$ is such that $(H1 \setminus^* H0) \implies H2$. In the definition below, observe how $H0$ is existentially quantified.

Definition hwand ($H1\ H2$:hprop) : hprop :=
 $\backslash \exists\ H0,\ H0\ \backslash * \backslash [H1\ \backslash * H0 ==> H2]$.

Notation "H1 $\backslash *$ H2" := (hwand H1 H2) (at level 43, right associativity).

Characteristic Equivalence for hwand

The magic wand is not easy to make sense of at first. Reading its introduction and elimination rules may help further appreciate its meaning.

The operator $H1\ \backslash * H2$ satisfies the following equivalence. Informally speaking, think of $H0 = -H1 + H2$ and $H1 + H0 = H2$ being equivalent claims.

Lemma hwand_equiv : $\forall\ H0\ H1\ H2,$
 $(H0 ==> H1\ \backslash * H2) \leftrightarrow (H1\ \backslash * H0 ==> H2).$

Proof using.

unfold hwand. iff M.
 $\{ \text{exchange } M.\ \text{intros } H\ N.\ \text{exchange } N.\ }$
 $\{ \text{xsimpl } H0.\ \text{exchange } M.\ }$

Qed.

Indeed, we will see below that the magic wand operator is *uniquely defined* by the equivalence $(H0 ==> H1\ \backslash * H2) \leftrightarrow (H1\ \backslash * H0 ==> H2)$. In other words, any operator that satisfies the above equivalence is provably equal to hwand.

The right-to-left direction of the equivalence is an introduction rule: it tells what needs to be proved for constructing a magic wand $H1\ \backslash * H2$ from a state $H0$.

To establish that $H0$ entails $H1\ \backslash * H2$, we have to show that the conjunction of $H0$ and $H1$ yields $H2$.

Lemma himpl_hwand_r : $\forall\ H0\ H1\ H2,$
 $(H1\ \backslash * H0) ==> H2 \rightarrow$
 $H0 ==> (H1\ \backslash * H2).$

Proof using. introv M. applys hwand_equiv. applys M. Qed.

The left-to-right direction of the equivalence is an elimination rule: it tells what can be deduced from an entailment $H0 ==> (H1\ \backslash * H2)$ – namely, if $H0$ is starred with $H1$, then $H2$ can be recovered.

Lemma himpl_hwand_r_inv : $\forall\ H0\ H1\ H2,$
 $H0 ==> (H1\ \backslash * H2) \rightarrow$
 $(H1\ \backslash * H0) ==> H2.$

Proof using. introv M. applys hwand_equiv. applys M. Qed.

This elimination rule can be equivalently reformulated in the following handy form: $H1\ \backslash * H2$, when starred with $H1$, yields $H2$.

Lemma hwand_cancel : $\forall\ H1\ H2,$
 $H1\ \backslash * (H1\ \backslash * H2) ==> H2.$

Proof using. intros. applys himpl_hwand_r_inv. applys himpl_refl. Qed.

Arguments hwand_cancel : clear *implicit*s.

Further Properties of `hwand`

We next present the most important properties of the magic wand operator. Here, the tactic `xsimpl` is available only in a restricted form that does not support magic wand. (The full `xsimpl` tactic would trivially solve many of these lemmas, but using it here would be cheating because the implementation of `xsimpl` itself relies on several of these lemmas.)

The operator $H1 \setminus^* H2$ is contravariant in $H1$ and covariant in $H2$, like the implication operator $\rightarrow$.

```
Lemma hwand_himpl : ∀ H1 H1' H2 H2',
  H1' ==> H1 →
  H2 ==> H2' →
  (H1 \-* H2) ==> (H1' \-* H2').
```

Proof using.

```
  introv M1 M2. applys himpl_hwand_r. xchange M1.
  xchange (hwand_cancel H1 H2). applys M2.
```

Qed.

The predicates $H1 \setminus^* H2$ and $H2 \setminus^* H3$ together simplify to $H1 \setminus^* H3$. This is reminiscent of the arithmetic identity $(-H1 + H2) + (-H2 + H3) = (-H1 + H3)$ (and to the transitivity of ordinary implication).

```
Lemma hwand_trans_elim : ∀ H1 H2 H3,
  (H1 \-* H2) \* (H2 \-* H3) ==> (H1 \-* H3).
```

Proof using.

```
  intros. applys himpl_hwand_r. xchange (hwand_cancel H1 H2).
```

Qed.

The predicate $H \setminus^* H$ holds of the empty heap. Intuitively, we can rewrite 0 as $-H + H$.

```
Lemma himpl_hempty_hwand_same : ∀ H,
  \[] ==> (H \-* H).
```

Proof using. intros. apply himpl_hwand_r. `xsimpl`. Qed.

Let's now study the interaction of `hwand` with `hempty` and `hpure`.

The heap predicate $\setminus[] \setminus^* H$ is equivalent to H . Intuitively, we can rewrite $-0 + H$ as $+H$.

```
Lemma hwand_hempty_l : ∀ H,
  (\[] \-* H) = H.
```

Proof using.

```
  intros. unfold hwand. xsimpl.
  { intros H0 M. xchange M. }
  { xsimpl. }
```

Qed.

The lemma above shows that $\setminus[]$ can be removed from the LHS of a magic wand.

More generally, a pure predicate $\backslash[P]$ can be removed from the LHS of a magic wand as long as P is true. Formally:

Exercise: 2 stars, standard, especially useful (hwand_hpure_l) *Lemma hwand_hpure_l*

$: \forall P H,$
 $P \rightarrow$
 $(\backslash[P] \backslash-* H) = H.$

Proof using. Admitted.

□

Reciprocally, to prove that a heap satisfies $\backslash[P] \backslash-* H2$, it suffices to prove that this heap satisfies $H2$ under the assumption that P is true. Formally:

Lemma himpl_hwand_hpure_r : $\forall H1 H2 P,$
 $(P \rightarrow H1 ==> H2) \rightarrow$
 $H1 ==> (\backslash[P] \backslash-* H2).$

Proof using. introv M. applies himpl_hwand_r. xsimpl. applies M. Qed.

Exercise: 2 stars, standard, optional (himpl_hwand_hpure_lr) Prove that $\backslash[P1 \rightarrow P2]$ entails $\backslash[P1] \backslash-* \backslash[P2]$. Hint: use *xpull*, then use *xsimpl* by providing an explicit argument to indicate how the $\backslash\exists$ on the right-hand side of the entailment should be instantiated.

Lemma himpl_hwand_hpure_lr : $\forall (P1 P2:\text{Prop}),$
 $\backslash[P1 \rightarrow P2] ==> (\backslash[P1] \backslash-* \backslash[P2]).$

Proof using. Admitted.

□

Another interesting property is that arguments on the LHS of a magic wand can equivalently be “curried” or “uncurried”, just as a function of type $(A \times B) \rightarrow C$ is equivalent to a function of type $A \rightarrow B \rightarrow C$.

The heap predicates $(H1 \backslash* H2) \backslash-* H3$, $H1 \backslash-* (H2 \backslash-* H3)$, and $H2 \backslash-* (H1 \backslash-* H3)$ are all equivalent. Intuitively, they all describe $H3$ with the missing pieces $H1$ and $H2$.

The equivalence between the uncurried form $(H1 \backslash* H2) \backslash-* H3$ and the curried form $H1 \backslash-* (H2 \backslash-* H3)$ is formalized by the lemma shown below. The third form, $H2 \backslash-* (H1 \backslash-* H3)$, follows from the commutativity property $H1 \backslash* H2 = H2 \backslash* H1$.

Lemma hwand_curry_eq : $\forall H1 H2 H3,$
 $(H1 \backslash* H2) \backslash-* H3 = H1 \backslash-* (H2 \backslash-* H3).$

Proof using.

intros. applies himpl_antisym.
 $\{ \text{apply himpl_hwand_r. apply himpl_hwand_r.}$
 $\text{ xchange (hwand_cancel (H1 \backslash* H2) H3). } \}$
 $\{ \text{apply himpl_hwand_r. xchange (hwand_cancel H1 (H2 \backslash-* H3)).}$
 $\text{ xchange (hwand_cancel H2 H3). } \}$

Qed.

Yet another interesting property is that the RHS of a magic wand can absorb resources that the magic wand is starred with.

Concretely, from $(H1 \setminus^* H2) \setminus^* H3$, we can get the predicate $H3$ to be absorbed by the $H2$ in the magic wand, yielding $H1 \setminus^* (H2 \setminus^* H3)$. One way to read this: “if you own $H3$ and, when given $H1$, you own $H2$, then, when given $H1$, you own both $H2$ and $H3$.”

Lemma `hstar_hwand` : $\forall H1\ H2\ H3$,
 $(H1 \setminus^* H2) \setminus^* H3 \implies H1 \setminus^* (H2 \setminus^* H3)$.

Proof using.

`intros. applies himpl_hwand_r. xsimpl. xchange (hwand_cancel H1 H2).`

Qed.

The reciprocal entailment is false: $H1 \setminus^* (H2 \setminus^* H3)$ does not entail $(H1 \setminus^* H2) \setminus^* H3$.

To see why, instantiate $H1$ with $\llbracket \text{False} \rrbracket$. The predicate $\llbracket \text{False} \rrbracket \setminus^* (H2 \setminus^* H3)$ is equivalent to **True**, hence imposes no restriction on the heap. But, to establish $(\llbracket \text{False} \rrbracket \setminus^* H2) \setminus^* H3$, we would need to exhibit a piece of heap satisfying $H3$.

Exercise: 1 star, standard, especially useful (`himpl_hwand_hstar_same_r`) Prove that $H1$ entails $H2 \setminus^* (H2 \setminus^* H1)$.

Lemma `himpl_hwand_hstar_same_r` : $\forall H1\ H2$,
 $H1 \implies (H2 \setminus^* (H2 \setminus^* H1))$.

Proof using. *Admitted*.

□

Exercise: 2 stars, standard, especially useful (`hwand_cancel_part`) Prove that $H1 \setminus^* ((H1 \setminus^* H2) \setminus^* H3)$ simplifies to $H2 \setminus^* H3$.

Lemma `hwand_cancel_part` : $\forall H1\ H2\ H3$,
 $H1 \setminus^* ((H1 \setminus^* H2) \setminus^* H3) \implies (H2 \setminus^* H3)$.

Proof using. *Admitted*.

□

Exercise: 3 stars, standard, optional (`hwand_frame`) Prove that $H1 \setminus^* H2$ entails to $(H1 \setminus^* H3) \setminus^* (H2 \setminus^* H3)$. Hint: use *xsimpl*.

Lemma `hwand_frame` : $\forall H1\ H2\ H3$,
 $H1 \setminus^* H2 \implies (H1 \setminus^* H3) \setminus^* (H2 \setminus^* H3)$.

Proof using. *Admitted*.

□

Exercise: 3 stars, standard, optional (`hwand_inv`) Prove the following inversion lemma for `hwand`. This lemma essentially captures the fact that `hwand` entails the alternative definition named `hwand'`.

```

Lemma hwand_inv :  $\forall$   $h1$   $h2$   $H1$   $H2$ ,
  ( $H1 \setminus -* H2$ )  $h2 \rightarrow$ 
   $H1$   $h1 \rightarrow$ 
  Fmap.disjoint  $h1$   $h2 \rightarrow$ 
   $H2$  ( $h1 \setminus u h2$ ).

```

Proof using. *Admitted*.

□

End HWAND.

43.2.5 Definition of the Magic Wand Operator for Postconditions: **qwand**

Module QWAND.

Import HWAND.

In what follows, we generalize the magic wand to operate on postconditions, introducing a heap predicate of the form $Q1 \setminus -* Q2$, of type **hprop**.

Note that the magic wand between two postconditions produces a heap predicate, not a postcondition.

There are two ways to define the operator **qwand**. The first is to follow the same pattern as for **hwand**, that is, to quantify some heap predicate $H0$ such that $H0$ starred with $Q1$ yields $Q2$.

```

Definition qwand' ( $Q1$   $Q2$ :val $\rightarrow$ hprop) : hprop :=
   $\exists H0, H0 \setminus * \setminus [ Q1 \setminus ** H0 ==> Q2 ]$ .

```

The second possibility is to define **qwand** directly on top of **hwand**, by means of the **hforall** quantifier.

```

Definition qwand ( $Q1$   $Q2$ :val $\rightarrow$ hprop) : hprop :=
   $\forall v, (Q1 v) \setminus -* (Q2 v)$ .

```

Notation " $Q1 \setminus -* Q2$ " := (**qwand** $Q1$ $Q2$) (at level 43).

As we establish later in this chapter, **qwand** and **qwand'** both define the same operator. We prefer taking **qwand** as the definition because in practice instantiating the universal quantifier is the most useful way to exploit a magic wand between postconditions. This specialization operation is formalized next. This result is a direct consequence of the specialization result for $\forall$.

```

Lemma qwand_specialize :  $\forall (v$ :val) ( $Q1$   $Q2$ :val $\rightarrow$ hprop),
  ( $Q1 \setminus -* Q2$ )  $==>$  ( $Q1 v \setminus -* Q2 v$ ).

```

Proof using.

intros. unfold **qwand**. *applies* **himpl_hforall_l** v . *xsimpl*.

Qed.

The predicate `qwand` satisfies numerous properties that are direct counterparts of the properties on `hwand`. First, `qwand` satisfies a characteristic equivalence rule.

Lemma `qwand_equiv` : $\forall H \ Q1 \ Q2,$
 $H \implies (Q1 \setminus^* Q2)$
 $\leftrightarrow (Q1 \setminus^{*+} H) \implies Q2.$

Proof using.

```
intros. iff M.
{ intros x. xchange M. xchange (qwand_specialize x).
  xchange (hwand_cancel (Q1 x)). }
{ applies himpl_hforall_r. intros x. applies himpl_hwand_r.
  xchange (M x). }
```

Qed.

Second, `qwand` satisfies a cancellation rule.

Lemma `qwand_cancel` : $\forall Q1 \ Q2,$
 $Q1 \setminus^{*+} (Q1 \setminus^* Q2) \implies Q2.$

Proof using. `intros. rewrite ← qwand_equiv. applies qimpl_refl. Qed.`

Third, the operation $Q1 \setminus^* Q2$ is contravariant in $Q1$ and covariant in $Q2$.

Lemma `qwand_himpl` : $\forall Q1 \ Q1' \ Q2 \ Q2',$
 $Q1' \implies Q1 \rightarrow$
 $Q2 \implies Q2' \rightarrow$
 $(Q1 \setminus^* Q2) \implies (Q1' \setminus^* Q2').$

Proof using.

```
introv M1 M2. rewrite qwand_equiv. intros x.
xchange (qwand_specialize x). xchange M1.
xchange (hwand_cancel (Q1 x)). xchange M2.
```

Qed.

Fourth the operation $Q1 \setminus^* Q2$ can absorb in its RHS resources to which it is starred.

Lemma `hstar_qwand` : $\forall Q1 \ Q2 \ H,$
 $(Q1 \setminus^* Q2) \setminus^* H \implies Q1 \setminus^* (Q2 \setminus^{*+} H).$

Proof using.

```
intros. rewrite qwand_equiv. xchange (@qwand_cancel Q1).
```

Qed.

Exercise: 1 star, standard, especially useful (`himpl_qwand_hstar_same_r`) Prove that H entails $Q \setminus^* (Q \setminus^{*+} H)$.

Lemma `himpl_qwand_hstar_same_r` : $\forall H \ Q,$
 $H \implies Q \setminus^* (Q \setminus^{*+} H).$

Proof using. *Admitted.*

□

Exercise: 2 stars, standard, optional (qwand_cancel_part) Prove that $H \setminus^* ((Q1 \setminus^+ H) \setminus^* Q2)$ simplifies to $Q1 \setminus^* Q2$. Hint: use *xchange*.

Lemma `qwand_cancel_part` : $\forall H Q1 Q2,$
 $H \setminus^* ((Q1 \setminus^+ H) \setminus^* Q2) \implies (Q1 \setminus^* Q2).$

Proof using. *Admitted*.

□

End QWAND.

43.2.6 Simplifications of Magic Wands using *xsimpl*

We can extend the tactic *xsimpl* to recognize the magic wand and automatically perform a number of obvious simplifications. This extension is implemented in the file `LibSepSimpl`, which exports the version of *xsimpl* illustrated here.

Module `XSIMPLDEMO`.

xsimpl is able to spot a magic wand that cancels out. For example, if an iterated separating conjunction includes both $H2 \setminus^* H3$ and $H2$, then these two heap predicates can be simplified into $H3$.

Lemma `xsimpl_demo_hwand_cancel` : $\forall H1 H2 H3 H4 H5,$
 $H1 \setminus^* (H2 \setminus^* H3) \setminus^* H4 \setminus^* H2 \implies H5.$

Proof using. `intros. xsimpl. Abort.`

xsimpl is able to simplify uncurried magic wands. For example, if an iterated separating conjunction includes $(H1 \setminus^* H2 \setminus^* H3) \setminus^* H4$ and $H2$, the two predicates can be simplified into $(H1 \setminus^* H3) \setminus^* H4$.

Lemma `xsimpl_demo_hwand_cancel_partial` : $\forall H1 H2 H3 H4 H5 H6,$
 $((H1 \setminus^* H2 \setminus^* H3) \setminus^* H4) \setminus^* H5 \setminus^* H2 \implies H6.$

Proof using. `intros. xsimpl. Abort.`

xsimpl automatically applies the introduction rule `himpl_hwand_r` when the right-hand-side, after prior simplification, reduces to just a magic wand. In the example below, $H1$ is first cancelled out from both sides, then $H3$ is moved from the RHS to the LHS.

Lemma `xsimpl_demo_himpl_hwand_r` : $\forall H1 H2 H3 H4 H5,$
 $H1 \setminus^* H2 \implies H1 \setminus^* (H3 \setminus^* (H4 \setminus^* H5)).$

Proof using. `intros. xsimpl. Abort.`

xsimpl can iterate a number of simplifications involving different magic wands.

Lemma `xsimpl_demo_hwand_iter` : $\forall H1 H2 H3 H4 H5,$
 $H4 \setminus^* H3 \implies H5 \rightarrow$
 $H2 \setminus^* (H1 \setminus^* H3) \setminus^* H4 \setminus^* (H2 \setminus^* H1) \implies H5.$

Proof using. `intros. xsimpl. auto. Qed.`

xsimpl can iterate simplifications on both sides.


```

Lemma xsimpl_demo_hwand_iter_2 : ∀ H1 H2 H3 H4 H5,
  H1 \* H2 \* ((H1 \* H3) \-* (H4 \-* H5)) \* H4 ==> ((H2 \-* H3) \-* H5).
Proof using. intros. xsimpl. Qed.

```

xsimpl is also able to deal with the magic wand for postconditions. In particular, it is able to simplify the conjunction of $Q1 \setminus^* Q2$ and $Q1 \vee$ into $Q2 \vee$.

```

Lemma xsimpl_demo_qwand_cancel : ∀ v (Q1 Q2:val→hprop) H1 H2,
  (Q1 \-* Q2) \* H1 \* (Q1 v) ==> H2.
Proof using. intros. xsimpl. Abort.

```

xsimpl is able to prove entailments whose right-hand side is a magic wand.

```

Lemma xsimpl_hwand_frame : ∀ H1 H2 H3,
  (H1 \-* H2) ==> ((H1 \* H3) \-* (H2 \* H3)).
Proof using.
  intros.
  xsimpl.
Qed.

```

End XSIMPLEDEMO.

43.2.7 Summary of All the Separation Logic Operators

Module SUMMARYHPROP.

The core operators are defined as functions over heaps.

```

Definition hempty : hprop :=
  fun h => (h = Fmap.empty).

Definition hsingle (p:loc) (v:val) : hprop :=
  fun h => (h = Fmap.single p v).

Definition hstar (H1 H2 : hprop) : hprop :=
  fun h => ∃ h1 h2, H1 h1
    ∧ H2 h2
    ∧ Fmap.disjoint h1 h2
    ∧ h = Fmap.union h1 h2.

```

```

Definition hexists A (J:A→hprop) : hprop :=
  fun h => ∃ x, J x h.

Definition hforall (A : Type) (J : A → hprop) : hprop :=
  fun h => ∀ x, J x h.

```

The remaining operators can be defined in terms of the core operators.

Module REAMININGOPERATORSDERIVED.

```

Definition hpure (P:Prop) : hprop :=
  \∃ (p:P), \[].

```

```

Definition hand (H1 H2 : hprop) : hprop :=
  \∀ (b:bool), if b then H1 else H2.

Definition hor (H1 H2 : hprop) : hprop :=
  \∃ (b:bool), if b then H1 else H2.

Definition hwand (H1 H2 : hprop) : hprop :=
  \∃ H0, H0 \* \[ (H1 \* H0) ==> H2 ].

Definition qwand (Q1 Q2 : val→hprop) : hprop :=
  \∀ v, (Q1 v) \-* (Q2 v).

```

End REAMININGOPERATORSDERIVED.

Of course, these derived operators could also be defined directly as predicate over heaps. The definitions are shown below. However, establishing properties of such low-level definitions requires more effort than establishing properties for the derived definitions shown above. Indeed, when operators are defined as derived operations, their properties may be established with help of the powerful entailment simplification tactic *xsimpl*.

Module REAMININGOPERATORSDIRECT.

```

Definition hpure (P:Prop) : hprop :=
  fun h => (h = Fmap.empty) ∧ P.

Definition hor (H1 H2 : hprop) : hprop :=
  fun h => H1 h ∨ H2 h.

Definition hand (H1 H2 : hprop) : hprop :=
  fun h => H1 h ∧ H2 h.

Definition hwand (H1 H2:hprop) : hprop :=
  fun h => ∀ h', Fmap.disjoint h h' → H1 h' → H2 (h \u h').

Definition qwand (Q1 Q2:val→hprop) : hprop :=
  fun h => ∀ v h', Fmap.disjoint h h' → Q1 v h' → Q2 v (h \u h').

```

End REAMININGOPERATORSDIRECT.

End SUMMARYHPROP.

43.2.8 The Ramified Frame Rule

Module RAMIFIEDFRAME.

Import HWAND QWAND.

Recall the consequence-frame rule, which is used pervasively – for example by the tactic *xapp*, for reasoning about applications.

```

Parameter triple_conseq_frame : ∀ H2 H1 Q1 t H Q,
  triple t H1 Q1 →
  H ==> H1 \* H2 →
  Q1 \*+ H2 ==> Q →

```

triple t H Q .

This rule suffers from a practical issue, which we illustrate in detail on a concrete example further on. At a high-level, though, the problem stems from the fact that we need to instantiate $H2$ when applying the rule. Providing $H2$ by hand is not practical, thus we need to infer it. The value of $H2$ can be computed as the subtraction of H minus $H1$. The resulting value may then be exploited in the last premise for constructing $Q1 \setminus^* + H2$. This transfer of information via $H2$ from one subgoal to another can be obtained by introducing an “evar” (Rocq unification variable) for $H2$. However this approach does not work well in cases where H contains existential quantifiers. This is because such existential quantifiers are typically first extracted out of the entailment $H ==> H1 \setminus^* H2$ by the tactic *xsimpl*. But these existentially quantified variables are not in the scope of $H2$, so the instantiation of the evar associated with $H2$ typically fails.

The “ramified frame rule” exploits the magic wand operator to circumvent this problem, by merging the two premises $H ==> H1 \setminus^* H2$ and $Q1 \setminus^* + H2 ==> Q$ into a single premise that no longer mentions $H2$. This replacement premise is $H ==> H1 \setminus^* (Q1 \setminus^* Q)$. To understand where it comes from, observe first that the second premise $Q1 \setminus^* + H2 ==> Q$ is equivalent to $H2 ==> (Q1 \setminus^* Q)$. By replacing $H2$ with $Q1 \setminus^* Q$ inside the first premise $H ==> H1 \setminus^* H2$, we obtain the new premise $H ==> H1 \setminus^* (Q1 \setminus^* Q)$. This merge of the two entailments leads us to the statement of the “ramified frame rule” shown below.

Lemma triple_ramified_frame : $\forall H1 Q1 t H Q$,
 triple t $H1$ $Q1 \rightarrow$
 $H ==> H1 \setminus^* (Q1 \setminus^* Q) \rightarrow$
 triple t H Q .

Proof using.

introv M W . *applies* triple_conseq_frame ($Q1 \setminus^* Q$) M .
 { *applies* W . } { *applies* qwand_cancel. }

Qed.

Reciprocally, we can prove that the ramified frame rule entails the consequence-frame rule. Hence, the ramified frame rule has the same expressive power as the consequence-frame rule.

Lemma triple_conseq_frame_of_ramified_frame : $\forall H2 H1 Q1 t H Q$,
 triple t $H1$ $Q1 \rightarrow$
 $H ==> H1 \setminus^* H2 \rightarrow$
 $Q1 \setminus^* + H2 ==> Q \rightarrow$
 triple t H Q .

Proof using.

introv M WH WQ . *applies* triple_ramified_frame M .
xchange WH . *xsimpl*. *rewrite* qwand_equiv. *applies* WQ .

Qed.

43.3 More Details

43.3.1 Benefits of the Ramified Frame Rule

Earlier on, we sketched an argument claiming that the consequence-frame rule is not very well suited for carrying out proofs in practice, due to issues with working with evars for instantiating the heap predicate $H2$ in the rule. Let us come back to this point and describe the issue in depth on a concrete example, then show how the ramified frame rule smoothly handles that example.

Recall, once again, the consequence-frame rule.

```
Parameter triple_conseq_frame' : ∀ H2 H1 Q1 t H Q,  
  triple t H1 Q1 →  
  H ==> H1 \* H2 →  
  Q1 \*+ H2 ==> Q →  
  triple t H Q.
```

One practical caveat with this rule is that we must resolve $H2$, which corresponds to the difference between H and $H1$. In practice, providing $H2$ explicitly is extremely tedious. The alternative is to leave $H2$ as an evvar, and count on the fact that the tactic *xsimpl*, when applied to $H ==> H1 * H2$, will correctly instantiate $H2$. This approach works in simple cases, but fails in particular in the case where H contains an existential quantifier.

For a concrete example, recall the specification of the primitive *ref*, which allocates a reference.

```
Parameter triple_ref : ∀ (v:val),  
  triple (val_ref v)  
    \ []  
    (funloc p => p ~~~> v).
```

Assume that wish to derive the following triple, which extends both the precondition and the postcondition of *triple_ref* with the heap predicate $\exists l' v', l' \sim\sim v'$.

This predicate describes the existence of some other, totally unspecified, reference cell. It is a bit artificial but illustrates the issue.

```
Lemma triple_ref_extended : ∀ (v:val),  
  triple (val_ref v)  
    (\exists l' v', l' ~~~> v')  
    (funloc p => p ~~~> v \* \exists l' v', l' ~~~> v').
```

Let's try to prove that this specification is derivable from the original *triple_ref*.

Proof using.

```
intros. applys triple_conseq_frame.  
{ applys triple_ref. }  
{  
  xsimpl.}
```

Abort.

Now, let us apply the ramified frame rule to carry out the same proof, and observe how the problem does not show up.

```
Lemma triple_ref_extended' : ∀ (v:val),
  triple (val_ref v)
    (λ ∃ l' v', l' ~> v')
    (funloc p ⇒ p ~> v \* λ ∃ l' v', l' ~> v').
```

Proof using.

```
intros. applies triple_ramified_frame.
{ applies triple_ref. }
{ xsimpl.
  intros l' v'.
  rewrite qwand_equiv.      intros x. xsimpl. auto. }
```

Qed.

End RAMIFIEDFRAME.

43.3.2 Some Tempting, But False, Properties of the Magic Wand

Module HWANDFALSE.

Import HWAND.

The entailment $\backslash[] \implies (H \backslash-* H)$ holds for any H . But the symmetrical entailment $(H \backslash-* H) \implies \backslash[]$ is false. For a counterexample, instantiate H as $\backslash[\text{False}]$. Any heap satisfies $\backslash[\text{False}] \backslash-* \backslash[\text{False}]$. But only the empty heap satisfies $\backslash[]$.

Lemma himpl_hwand_same_empty_counterexample :

```
¬ (∀ H, (H \-* H) ==> \[]).
```

Proof using.

```
rew_logic.   ∃ \[\text{False}]. intros M.
lets (h,Hh): (@Fmap.exists_not_empty val _).
forwards K: M h.
{ applies× himpl_hwand_r (fun h ⇒ h ≠ Fmap.empty). xsimpl×. }
lets: empty_inv K. false× Hh.
```

Qed.

As another tempting yet false property of the magic wand, consider the converse of the cancellation lemma, that is, $H2 \implies H1 \backslash-* (H1 \backslash-* H2)$. It does not hold in general. As a counter-example, consider $H2 = \backslash[]$ and $H1 = \backslash[\text{False}]$. The empty heap satisfies the left-hand side of the entailment, but it does not satisfy $\backslash[\text{False}] \backslash-* (\backslash[\text{False}] \backslash-* \backslash[])$, because there is no way to establish **False** out of thin air.

Lemma not_himpl_hwand_r_inv_reciprocal : ∃ H1 H2,

```
¬ (H2 ==> H1 \-* (H1 \-* H2)).
```

Proof using.

```
∃ \[\text{False}] \[]. intros N. forwards K: N (Fmap.empty:heap).
applies empty_intro. rewrite hstar_hpure_l in K. destruct K. auto.
```

Qed.

More generally, we should be suspicious of any entailment that introduces wands “out of nowhere”.

The entailment `hwand_trans_elim`: $(H1 \setminus^* H2) \setminus^* (H2 \setminus^* H3) \implies (H1 \setminus^* H3)$ is correct because, intuitively, the left-hand-side captures that $H1 \leq H2$ and that $H2 \leq H3$ for some vaguely defined notion of $\leq$ as “being a subset of”. On the contrary, the reciprocal entailment $(H1 \setminus^* H3) \implies (H1 \setminus^* H2) \setminus^* (H2 \setminus^* H3)$ is false. Intuitively, from $H1 \leq H3$ there is no way to justify $H1 \leq H2$ nor $H2 \leq H3$.

End `HWANDFALSE`.

43.4 Optional Material

43.4.1 Equivalence Between Alternative Definitions of the Magic Wand

Module `HWANDEQUIV`.

Implicit Type `op` : `hprop` $\rightarrow$ `hprop` $\rightarrow$ `hprop`.

In what follows we prove the equivalence between the three characterizations of `hwand` $H1\ H2$ that we have presented:

1. The definition `hwand` expressed directly in terms of heaps: `fun h  $\Rightarrow$   $\forall h'$ , Fmap.disjoint  $h\ h' \rightarrow H1\ h' \rightarrow H2\ (h' \setminus u\ h)$`
2. The definition `hwand` expressed in terms of existing operators: `$\setminus \exists H0$ ,  $H0 \setminus^* \setminus [ (H1 \setminus^* H0) \implies H2 ]$`
3. The characterization via the equivalence `hwand_equiv`: `$\forall H0\ H1\ H2$ ,  $(H0 \implies H1 \setminus^* H2) \leftrightarrow (H1 \setminus^* H0 \implies H2)$` .
4. The characterization via the pair of the introduction rule `himpl_hwand_r` and the elimination rule `hwand_cancel`.

To prove the 4-way equivalence, we first prove the equivalence between (1) and (2), then prove the equivalence between (2) and (3), and finally the equivalence between (3) and (4).

Definition `hwand_characterization_1` (`op`:`hprop` $\rightarrow$ `hprop` $\rightarrow$ `hprop`) :=

`op = (fun $H1\ H2 \Rightarrow$
 $(\text{fun } h \Rightarrow \forall h', \text{Fmap.disjoint } h\ h' \rightarrow H1\ h' \rightarrow H2\ (h' \setminus u\ h))$).`

Definition `hwand_characterization_2` (`op`:`hprop` $\rightarrow$ `hprop` $\rightarrow$ `hprop`) :=

`op = (fun  $H1\ H2 \Rightarrow \setminus \exists H0$ ,  $H0 \setminus^* \setminus [ H1 \setminus^* H0 \implies H2 ]$ ).`

Definition `hwand_characterization_3` (`op`:`hprop` $\rightarrow$ `hprop` $\rightarrow$ `hprop`) :=

`$\forall H0\ H1\ H2$ ,  $(H0 \implies op\ H1\ H2) \leftrightarrow (H1 \setminus^* H0 \implies H2)$ .`

Definition `hwand_characterization_4` (`op`:`hprop` $\rightarrow$ `hprop` $\rightarrow$ `hprop`) :=

`$(\forall H0\ H1\ H2$, $(H1 \setminus^* H0 \implies H2) \rightarrow (H0 \implies op\ H1\ H2))$
 $\wedge (\forall H1\ H2$, $(H1 \setminus^* (op\ H1\ H2) \implies H2))$.`

The equivalence proofs are given here for reference. The reader needs not follow through the details of these proofs.

Lemma `hwand_characterization_1_eq_2` :

`hwand_characterization_1 = hwand_characterization_2.`

Proof using.

```
applies pred_ext_1. intros op.
unfold hwand_characterization_1, hwand_characterization_2.
asserts K: (∀ A B, A = B → (op = A ↔ op = B)).
{ intros. iff; subst×. } apply K; clear K.
apply pred_ext_3. intros H1 H2 h. iff M.
{ ∃ (=h). rewrite hstar_hpure_r. split.
  { auto. }
  { intros h3 K3. rewrite hstar_comm in K3.
    destruct K3 as (h1&h2&K1&K2&D&U). subst h1 h3.
    rewrites (» union_comm_of_disjoint D). applies M D K2. } }
{
  intros h1 D K1. destruct M as (H0&M).
  destruct M as (h0&h2&K0&K2&D'&U).
  lets (N&E): hpure_inv (rm K2). subst h h2.
  rewrite Fmap.union_empty_r in *.
  applies N. applies hstar_intro K1 K0. applies disjoint_sym D. }
```

Qed.

Lemma `hwand_characterization_2_eq_3` :

`hwand_characterization_2 = hwand_characterization_3.`

Proof using.

```
applies pred_ext_1. intros op.
unfold hwand_characterization_2, hwand_characterization_3. iff K.
{ subst. intros. iff M.
  { xchange M. intros H3 N. xchange N. }
  { xsimpl H0. xchange M. } }
{ apply fun_ext_2. intros H1 H2. apply himpl_antisym.
  { lets (M&-): (K (op H1 H2) H1 H2). xsimpl (op H1 H2).
    applies M. applies himpl_refl. }
  { xsimpl. intros H0 M. rewrite K. applies M. } }
```

Qed.

Lemma `hwand_characterization_3_eq_4` :

`hwand_characterization_3 = hwand_characterization_4.`

Proof using.

```
applies pred_ext_1. intros op.
unfold hwand_characterization_3, hwand_characterization_4. iff K.
{ split.
  { introv M. apply ← K. apply M. }
```

```

    { intros. apply K. auto. } }
  { destruct K as (K1&K2). intros. split.
    { introv M. xchange M. xchange (K2 H1 H2). }
    { introv M. applies K1. applies M. } }
Qed.
End HWANDEQUIV.

```

43.4.2 Equivalence Results for the Magic Wand for Postconditions

Module QWANDEQUIV.

Implicit Type *op* : (**val**→hprop)→(**val**→hprop)→hprop.

In what follows we prove the equivalence between five equivalent characterizations of *qwand* *H1 H2*:

1. The definition expressed directly in terms of heaps: $\text{fun } h \Rightarrow \forall v \ h', \text{Fmap.disjoint } h \ h' \rightarrow h' \rightarrow Q1 \ v \ h' \rightarrow Q2 \ v \ (h \setminus u \ h')$
2. The definition *qwand*, expressed in terms of existing operators: $\setminus \exists H0, H0 \setminus * \setminus [(Q1 \setminus * + H0) ==> Q2]$
3. The definition expressed using the universal quantifier: $\setminus \forall v, (Q1 \ v) \setminus * (Q2 \ v)$
4. The characterization via the equivalence *hwand_equiv*: $\forall H0 \ H1 \ H2, (H0 ==> H1 \setminus * H2) \leftrightarrow (H1 \setminus * H0 ==> H2)$.
5. The characterization via the pair of the introduction rule *himpl_qwand_r* and the elimination rule *qwand_cancel*.

The proof are essentially identical to the equivalence proofs for *hwand*, except for definition (3), which is specific to *qwand*.

Definition *qwand_characterization_1 op* :=

op = (fun *Q1 Q2* $\Rightarrow$ (fun *h* $\Rightarrow \forall v \ h', \text{Fmap.disjoint } h \ h' \rightarrow Q1 \ v \ h' \rightarrow Q2 \ v \ (h \setminus u \ h')$)).

Definition *qwand_characterization_2 op* :=

op = (fun *Q1 Q2* $\Rightarrow \setminus \exists H0, H0 \setminus * \setminus [Q1 \setminus * + H0 ==> Q2]$).

Definition *qwand_characterization_3 op* :=

op = (fun *Q1 Q2* $\Rightarrow \setminus \forall v, (Q1 \ v) \setminus * (Q2 \ v)$).

Definition *qwand_characterization_4 op* :=

$\forall H0 \ Q1 \ Q2, (H0 ==> op \ Q1 \ Q2) \leftrightarrow (Q1 \setminus * + H0 ==> Q2)$.

Definition *qwand_characterization_5 op* :=

$(\forall H0 \ Q1 \ Q2, (Q1 \setminus * + H0 ==> Q2) \rightarrow (H0 ==> op \ Q1 \ Q2))$
 $\wedge (\forall Q1 \ Q2, (Q1 \setminus * + (op \ Q1 \ Q2) ==> Q2))$.

Here again, we show proofs for the reference, but the reader needs not follow through the details.

Lemma *hwand_characterization_1_eq_2* :

qwand_characterization_1 = *qwand_characterization_2*.

Proof using.

```
applies pred_ext_1. intros op.
unfold qwand_characterization_1, qwand_characterization_2.
asserts K: (∀ A B, A = B → (op = A ↔ op = B)).
{ intros. iff; subst×. } apply K; clear K.
apply pred_ext_3. intros Q1 Q2 h. iff M.
{ ∃ (=h). rewrite hstar_hpure_r. split.
  { auto. }
  { intros v h3 K3. rewrite hstar_comm in K3.
    destruct K3 as (h1&h2&K1&K2&D&U). subst h1 h3. applies M D K2. } }
{ intros v h1 D K1. destruct M as (H0&M).
  destruct M as (h0&h2&K0&K2&D'&U).
  lets (N&E): hpure_inv (rm K2). subst h h2.
  rewrite Fmap.union_empty_r in *.
  applies N. rewrite hstar_comm. applies hstar_intro K0 K1 D. }
```

Qed.

Lemma qwand_characterization_2_eq_3 :

qwand_characterization_2 = qwand_characterization_3.

Proof using.

```
applies pred_ext_1. intros op.
unfold qwand_characterization_2, qwand_characterization_3.
asserts K: (∀ A B, A = B → (op = A ↔ op = B)).
{ intros. iff; subst×. } apply K; clear K.
apply fun_ext_2. intros Q1 Q2. apply himpl_antisym.
{ xpull. intros H0 M. applies himpl_hforall_r. intros v.
  rewrite hwand_equiv. xchange M. }
{ xsimpl (qwand Q1 Q2). applies qwand_cancel. }
```

Qed.

Lemma qwand_characterization_2_eq_4 :

qwand_characterization_2 = qwand_characterization_4.

Proof using.

```
applies pred_ext_1. intros op.
unfold qwand_characterization_2, qwand_characterization_4. iff K.
{ subst. intros. iff M.
  { xchange M. intros v H3 N. xchange N. }
  { xsimpl H0. xchange M. } }
{ apply fun_ext_2. intros H1 H2. apply himpl_antisym.
  { lets (M&-): (K (op H1 H2) H1 H2). xsimpl (op H1 H2).
    applies M. applies himpl_refl. }
  { xsimpl. intros H0 M. rewrite K. applies M. } }
```

Qed.

Lemma qwand_characterization_4_eq_5 :

```

    qwand_characterization_4 = qwand_characterization_5.
Proof using.
  applies pred_ext_1. intros op.
  unfold qwand_characterization_4, qwand_characterization_5. iff K.
  { split.
    { introv M. apply  $\leftarrow$  K. apply M. }
    { intros. apply K. auto. } }
  { destruct K as (K1 & K2). intros. split.
    { introv M. xchange M. xchange (K2 Q1 Q2). }
    { introv M. applies K1. applies M. } }
Qed.
End QWANDEQUIV.

```

43.4.3 Historical Notes

The magic wand is an operator that was introduced in the very first days of Separation Logic. From a logical perspective, it makes total sense to have it. From a practical perspective, however, it was not always entirely obvious how the magic wand could simplify specifications and proofs. Experience with CFML-1.0 shows that it is possible to develop a verification framework and verify thousands of lines of advanced data structures and algorithms without ever involving the magic wand operator. The magic wand, however, reveals its interest when exploited (1) in the ramified frame rule, and (2) in weakest-precondition style reasoning rules.

The idea of the ramified frame rule was introduced by *Krishnaswami, Birkedal, and Aldrich* 2010 (in Bib.v). Its general statement, as formulated in the present chapter, was proposed by *Hobor and Villard* 2013 (in Bib.v). The rule has later been popularized by the Iris framework, in particular.

Chapter 44

Library `SLF.WPsem`

44.1 WPsem: Semantics of Weakest Preconditions

```
Set Implicit Arguments.
From SLF Require Export Rules.

Implicit Types f : var.
Implicit Types b : bool.
Implicit Types v w : val.
Implicit Types p : loc.
Implicit Types h : heap.
Implicit Types P : Prop.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
```

44.2 First Pass

In this chapter, we introduce the notion of “weakest precondition” for Separation Logic triples, written `wp t Q`.

The intention is for `wp t Q` to be a heap predicate (of type `hprop`) such that $H \implies \text{wp } t \ Q$ iff `triple t H Q`.

The benefits of introducing weakest preconditions are twofold.

- The use of `wp` greatly reduces the number of structural rules required, thus reducing the number of tactics required for carrying out proofs in practice;
- The predicate `wp` will serve as guidelines for setting up (in the next chapter) a “characteristic formula generator”, which is the key ingredient at the heart of the CFML tool.

The high-level picture is:

- (1) wp and triples are equivalent presentations of the same thing.
- (2) wp has practical benefits, reducing the number of reasoning rules and tactics that are needed, by a fair amount.
- (3) specifications of functions remain more intuitive to read with triples.

The chapter presents:

- the notion of weakest precondition, as captured by `wp`,
- the reformulation of structural rules in `wp`-style, and
- the reformulation of reasoning rules in `wp`-style.

Throughout the chapter, we aim to establish reasoning rules in `wp`-style directly with respect to the semantics defined by `eval`, as opposed to deriving `wp`-style rules from the reasoning rules on triples proved in the previous chapters. The interest is to present the proofs involved in the most direct construction of a program verification framework based on `wp`.

The “optional material” section presents several equivalent definitions for `wp`, alternative proofs for deriving `wp`-style reasoning rules, as well as a presentation of “Texan triples”, which reformulate function specifications using magic wands instead of triples.

44.2.1 Definition of Weakest Preconditions

Consider a term `t` and a postcondition `Q`.

The expression `wp t Q` is called the “weakest precondition” of the term `t` with respect to the postcondition `Q`.

It is a heap predicate such that, for any heap predicate `H`, the entailment $H \implies \text{wp } t \text{ } Q$ holds if and only if `triple t H Q` does.

The concept of `wp` might seem a bit mysterious at this point. Hopefully it will become clearer in the rest of this chapter.

The first critical observation is that `wp` is in fact nothing but a reformulation of the predicate `eval`. To see why, recall the definition of `triple t H Q`.

Definition `triple (t:trm) (H:hprop) (Q:val->hprop) : Prop := forall s, H s -> eval s t Q`.

By definition, the predicate `wp` should be such that $H \implies \text{wp } t \text{ } Q$ is equivalent to `triple t H Q`. But $H \implies \text{wp } t \text{ } Q$ unfolds to $\forall s, H s \rightarrow \text{wp } t \text{ } Q s$. If we compare $\forall s, H s \rightarrow \text{eval } s \text{ } t \text{ } Q$ with $\forall s, H s \rightarrow \text{wp } t \text{ } Q s$, we see that `wp t Q s` should match `eval s t Q`. In other words, `wp t Q` should correspond to `fun s => eval s t Q`.

Definition `wp (t : trm) (Q : val->hprop) : hprop :=
fun s => eval s t Q`.

Remark: the reader may wonder why `wp` is not just defined to `*be*` `eval`, by simply modifying our earlier definition of `eval` to take its arguments in a different order. Certainly,

it could be done. However, the benefits of identifying `wp` and `eval` are limited, and there are two orthogonal motivations for not following that route.

Firstly, the omni-big-step predicate `eval s t Q` is devised as an immediate generalization of the big-step predicate `eval s t s' v`, by generalizing from out final configuration (s', v) to a set Q of final configurations. From the perspective of a semantics, we view the configuration (s, t) as an input and the set Q as an output, thus it would be fairly awkward to write `eval t Q s`, with the input state s taken as last argument.

Secondly, we are looking here at a simple Separation Logic where heap predicates are predicates over the physical state. They have type `heap → Prop`, where `heap` corresponds exactly to the type of of a state s . More advanced Separation Logic feature something known as “ghost state” (used, e.g., to represent invariants associated with locks). In such logics, heap predicates are predicates over both the physical and the ghost state. In other words, the type `heap` in general may consist of more than just a state s . The point of this discussion is that, for more advanced Separation Logics, `wp` is more than just `eval`. Thus, there is no reason to attempt to identify the two.

As intended, the entailment $H \implies wp\ t\ Q$ holds if and only if `triple t H Q` does.

Lemma `wp_equiv` : $\forall\ t\ H\ Q,$
 $(H \implies wp\ t\ Q) \leftrightarrow (triple\ t\ H\ Q).$

Proof using.

iff $M. \{ introv\ Hs. applys\ M\ Hs. \} \{ introv\ Hs. applys\ M\ Hs. \}$

Qed.

There are several other ways of defining `wp`. As we show near the end of this chapter, they are all equivalent. The definition above is the one that leads to the simplest proofs for the reasoning rules. Indeed, reasoning rules must be proved correct with respect to the semantics, and the semantics is captured by the predicate `eval`.

Let us now explain why `wp` is called a “weakest precondition”. First, `wp t Q` is always a *valid* precondition for t with respect to the postcondition Q .

Lemma `wp_pre` : $\forall\ t\ Q,$
 $triple\ t\ (wp\ t\ Q)\ Q.$

Proof using. `intros. rewrite ← wp_equiv. applys himpl_refl. Qed.`

Second, `wp t Q` is the “weakest” of all valid preconditions for t and Q , in the sense that, for any other valid precondition H (i.e., such that `triple t H Q` holds), it is the case that H entails `wp t Q`.

Lemma `wp_weakest` : $\forall\ t\ H\ Q,$
 $triple\ t\ H\ Q \rightarrow$
 $H \implies wp\ t\ Q.$

Proof using. `intros M. rewrite wp_equiv. applys M. Qed.`

44.2.2 Structural Rules in Weakest-Precondition Style

We next present reformulations of the frame rule and the rule of consequence in *weakest-precondition style*.

Given a term t and a postcondition Q , we say that “ t produces Q ” if t terminates and produces a output value and an output state that, together, satisfy the postcondition Q .

The Frame Rule

The frame rule for wp asserts that $(\text{wp } t \ Q) \setminus^* H$ entails $\text{wp } t \ (Q \setminus^* + H)$. This statement can be read as follows: if you own both a piece of state satisfying H and a piece of state in which the execution of t produces Q , then you own a piece of state in which the execution of t produces $Q \setminus^* + H$, that is, produces both Q and H .

Lemma wp_frame : $\forall t \ H \ Q,$
 $(\text{wp } t \ Q) \setminus^* H \implies \text{wp } t \ (Q \setminus^* + H).$

The lemma is proved by exploiting the frame property on eval . (It could also be derived using wp_equiv and triple_frame , but the point here is to derive properties of wp without mentioning triple .)

Proof using.

```
intros. unfold wp. intros h HF.
lets (h1&h2&M1&M2&MD&MU): hstar_inv HF.
subst. applies eval_conseq.
{ applies eval_frame M1 MD. }
{ xsimpl. intros h' →. applies M2. }
```

Qed.

Looking at the entailment expressed by wp_frame , the reader may wonder whether the reverse entailment holds, that is, if $\text{wp } t \ (Q \setminus^* + H)$ entails $(\text{wp } t \ Q) \setminus^* H$. This reverse entailment does not hold in general. Consider the following counterexample term t , defined as $\text{let } p = \text{ref } 1 \text{ in } 0$. This term returns 0 and produces a dangling reference cell that can be described as $\exists p, (p \sim > 1)$. Let Q be $\text{fun } r \Rightarrow [r = 0]$ and H be $\exists p, (p \sim > 1)$. The term t , executed in the empty heap, terminates with postcondition $Q \setminus^* + H$. Therefore, the predicate $\text{wp } t \ (Q \setminus^* + H)$ holds of the empty heap. However, $(\text{wp } t \ Q) \setminus^* H$ does not hold of the empty heap, because H , which describes one memory cell, cannot be carved out of the empty heap. We conclude that $\text{wp } t \ (Q \setminus^* + H)$ does not entail $(\text{wp } t \ Q) \setminus^* H$ in general.

The connection between wp_frame and the rule triple_frame is not be totally obvious. Recall the statement of the frame rule for triples.

$\text{triple } t \ H1 \ Q \rightarrow \text{triple } t \ (H1 \setminus^* H) \ (Q \setminus^* + H)$

Let us replace the pattern $\text{triple } t \ H \ Q$ with the pattern $H \implies \text{wp } t \ Q$, following the characteristic equivalence wp_equiv . We obtain the statement shown below.

Lemma wp_frame_trans : $\forall t \ H1 \ Q \ H,$
 $H1 \implies \text{wp } t \ Q \rightarrow$
 $(H1 \setminus^* H) \implies \text{wp } t \ (Q \setminus^* + H).$

If we exploit transitivity of entailment to eliminate $H1$, then we obtain exactly `wp_frame`, as illustrated by the proof script below.

Proof using. `introv M. xchange M. applies × wp_frame. Qed.`

The Rule of Consequence

The rule of consequence for `wp` is a covariance property: it asserts that `wp t Q` is covariant in Q . In other words, if $Q1$ entails $Q2$, then `wp t Q1` entails `wp t Q2`.

Lemma `wp_conseq` : $\forall t Q1 Q2,$

$Q1 \implies Q2 \rightarrow$

`wp t Q1` $\implies$ `wp t Q2`.

Proof using. `unfold wp. introv M. intros s Hs. applies × eval_conseq. Qed.`

Here again, the connection with the corresponding reasoning rule for triples is not totally obvious. Recall the statement of the rule of consequence.

`triple t H1 Q1 -> H2` $\implies$ `H1 -> Q1` $\implies$ `Q2 -> triple t H2 Q2`

Let us replace `triple t H Q` with $H \implies \text{wp } t \text{ } Q$ to obtain the following statement:

Lemma `wp_conseq_trans` : $\forall t H1 H2 Q1 Q2,$

$H1 \implies \text{wp } t \text{ } Q1 \rightarrow$

$H2 \implies H1 \rightarrow$

$Q1 \implies Q2 \rightarrow$

$H2 \implies \text{wp } t \text{ } Q2$.

If we exploit transitivity of entailment to eliminate $H1$ and $H2$, then we obtain exactly `wp_conseq`, as illustrated by the proof script below.

Proof using.

`introv M WH WQ. xchange WH. xchange M. applies wp_conseq WQ.`

Qed.

The Extraction Rules

The extraction rules `triple_hpure` and `triple_hexists` have no specific counterpart with the `wp` presentation. Indeed, in a weakest-precondition style presentation, the extraction rules for triples correspond exactly to the extraction rules for entailment.

To see why, consider the rule `triple_hpure`.

Parameter `triple_hpure` : $\forall t (P:\text{Prop}) H Q,$

$(P \rightarrow \text{triple } t \text{ } H \text{ } Q) \rightarrow$

`triple t (\[P] \* H) Q`.

Replacing `triple t H Q` with $H \implies \text{wp } t \text{ } Q$ yields the following:

Lemma `triple_hpure_with_wp` : $\forall t H Q (P:\text{Prop}),$

$(P \rightarrow (H \implies \text{wp } t \text{ } Q)) \rightarrow$

$(\text{wp } t \text{ } H) \implies \text{wp } t \text{ } Q$.

This implication is just a special case of the extraction lemma for pure facts on the left on an entailment, `himpl_hstar_hpure_l`:

$(P \rightarrow (H \Rightarrow H')) \rightarrow (\backslash P \backslash * H) \Rightarrow H'$.

Proof using. `intros M. applies himpl_hstar_hpure_l M. Qed.`

Similar reasoning applies to the extraction rule for existentials.

The Ramified Frame Rule

Recall the ramified frame rule:

Parameter `triple_ramified_frame` : $\forall H1\ Q1\ t\ H\ Q,$
 $\text{triple } t\ H1\ Q1 \rightarrow$
 $H \Rightarrow H1 \backslash * (Q1 \backslash - * Q) \rightarrow$
 $\text{triple } t\ H\ Q.$

The ramified frame rule in weakest-precondition style, named `wp_ramified`, admits a concise statement and subsumes all the other structural rules of Separation Logic. Its very elegant statement is as follows.

Lemma `wp_ramified` : $\forall t\ Q1\ Q2,$
 $(\text{wp } t\ Q1) \backslash * (Q1 \backslash - * Q2) \Rightarrow (\text{wp } t\ Q2).$

Proof using. `intros. applies wp_conseq_frame. applies qwand_cancel. Qed.`

The following reformulation of `wp_ramified` can be more practical to exploit in practice, because it applies to any goal of the form $H \Rightarrow \text{wp } t\ Q$.

Lemma `wp_ramified_trans` : $\forall t\ H\ Q1\ Q2,$
 $H \Rightarrow (\text{wp } t\ Q1) \backslash * (Q1 \backslash - * Q2) \rightarrow$
 $H \Rightarrow (\text{wp } t\ Q2).$

Proof using. `intros M. xchange M. applies wp_ramified. Qed.`

Exercise: 3 stars, standard, especially useful (`wp_conseq_of_wp_ramified`) Prove that `wp_conseq` is derivable from `wp_ramified`. Concretely, prove `wp_conseq` using only `wp_ramified` or `wp_ramified_trans`, plus properties of the entailment relation.

Lemma `wp_conseq_of_wp_ramified` : $\forall t\ Q1\ Q2,$
 $Q1 \Rightarrow Q2 \rightarrow$
 $\text{wp } t\ Q1 \Rightarrow \text{wp } t\ Q2.$

Proof using. `Admitted.`

□

Exercise: 2 stars, standard, especially useful (`wp_frame_of_wp_ramified`) Prove that `wp_frame` is derivable from `wp_ramified`. Concretely, prove `wp_frame` using only `wp_ramified` plus properties of entailment.

Lemma `wp_frame_of_wp_ramified` : $\forall t\ H\ Q,$

$(\text{wp } t \ Q) \ \backslash * \ H \implies \text{wp } t \ (Q \ \backslash * + \ H).$

Proof using. *Admitted*.

□

44.2.3 Reasoning Rules for Terms in Weakest-Precondition Style

Values

Recall the rule *triple_val* which establishes a triple for a value v .

Parameter *triple_val* : $\forall v \ H \ Q,$

$H \implies Q \ v \rightarrow$

$\text{triple} (\text{trm\_val } v) \ H \ Q.$

If we rewrite this rule in *wp* style, we obtain this:

$H \implies Q \ v \rightarrow H \implies \text{wp} (\text{trm\_val } v) \ Q.$

By exploiting transitivity of entailment, we can eliminate H . We obtain the following rule: If you own a state satisfying $Q \ v$, then you own a state from which the evaluation of the value v produces Q .

Lemma *wp_val* : $\forall v \ Q,$

$Q \ v \implies \text{wp} (\text{trm\_val } v) \ Q.$

Proof using. *unfold wp. intros. intros h K. applys × eval_val. Qed.*

Sequences

Recall the reasoning rule for a sequence *trm_seq* $t1 \ t2$.

Parameter *triple_seq* : $\forall t1 \ t2 \ H \ Q \ H1,$

$\text{triple } t1 \ H \ (\text{fun } _ \Rightarrow H1) \rightarrow$

$\text{triple } t2 \ H1 \ Q \rightarrow$

$\text{triple} (\text{trm\_seq } t1 \ t2) \ H \ Q.$

Replacing $\text{triple } t \ H \ Q$ with $H \implies \text{wp } t \ Q$ gives this:

$H \implies (\text{wp } t1) (\text{fun } _ \Rightarrow H1) \rightarrow H1 \implies (\text{wp } t2) \ Q \rightarrow H \implies \text{wp} (\text{trm\_seq } t1 \ t2) \ Q.$

This entailment holds for any H and $H1$. Let us specialize it to $H1 := (\text{wp } t2) \ Q$ and $H := (\text{wp } t1) (\text{fun } v \Rightarrow (\text{wp } t2) \ Q).$

This leads us to the following rule: If you own a state from which the evaluation of $t1$ produces a state from which the evaluation of $t2$ produces the postcondition Q , then you own a state from which the evaluation of the sequence $t1;t2$ produces Q .

Lemma *wp_seq* : $\forall t1 \ t2 \ Q,$

$\text{wp } t1 \ (\text{fun } v \Rightarrow \text{wp } t2 \ Q) \implies \text{wp} (\text{trm\_seq } t1 \ t2) \ Q.$

Proof using. *unfold wp. intros. intros h K. applys × eval_seq. Qed.*

Let-Bindings

Recall the reasoning rule for a term `trm_let x t1 t2`.

Parameter `triple_let` : $\forall x\ t1\ t2\ Q1\ H\ Q,$
 $\text{triple } t1\ H\ Q1 \rightarrow$
 $(\forall v1, \text{triple } (\text{subst } x\ v1\ t2)\ (Q1\ v1)\ Q) \rightarrow$
 $\text{triple } (\text{trm\_let } x\ t1\ t2)\ H\ Q.$

The rule of `trm_let x t1 t2` is very similar to that for `trm_seq`, the only difference being the substitution of `x` by `v` in `t2`, where `v` denotes the result of `t1`.

Lemma `wp_let` : $\forall x\ t1\ t2\ Q,$
 $\text{wp } t1\ (\text{fun } v1 \Rightarrow \text{wp } (\text{subst } x\ v1\ t2)\ Q) \Rightarrow \text{wp } (\text{trm\_let } x\ t1\ t2)\ Q.$
Proof using. `unfold wp. intros. intros h K. applys× eval_let. Qed.`

Functions

Recall the reasoning rule for a term `trm_fun x t1`, which evaluates to the value `val_fun x t1`.

Parameter `triple_fun` : $\forall x\ t1\ H\ Q,$
 $H \Rightarrow Q\ (\text{val\_fun } x\ t1) \rightarrow$
 $\text{triple } (\text{trm\_fun } x\ t1)\ H\ Q.$

The rule for functions follows exactly the same pattern as for values.

Lemma `wp_fun` : $\forall x\ t\ Q,$
 $Q\ (\text{val\_fun } x\ t) \Rightarrow \text{wp } (\text{trm\_fun } x\ t)\ Q.$
Proof using. `unfold wp. intros. intros h K. applys× eval_fun. Qed.`

A similar rule holds for the evaluation of a recursive function.

Lemma `wp_fix` : $\forall f\ x\ t\ Q,$
 $Q\ (\text{val\_fix } f\ x\ t) \Rightarrow \text{wp } (\text{trm\_fix } f\ x\ t)\ Q.$
Proof using. `unfold wp. intros. intros h K. applys× eval_fix. Qed.`

Conditionals

Recall the reasoning rule for a term `triple_if b t1 t2`.

Parameter `triple_if` : $\forall b\ t1\ t2\ H\ Q,$
 $\text{triple } (\text{if } b\ \text{then } t1\ \text{else } t2)\ H\ Q \rightarrow$
 $\text{triple } (\text{trm\_if } (\text{val\_bool } b)\ t1\ t2)\ H\ Q.$

Replacing `triple` using `wp` entailments yields

$H \Rightarrow \text{wp } (\text{if } b\ \text{then } t1\ \text{else } t2)\ Q \rightarrow H \Rightarrow \text{wp } (\text{trm\_if } (\text{val\_bool } b)\ t1\ t2)\ Q$

A necessary and sufficient condition to prove a proposition of the form $\forall H, (H \Rightarrow H1) \rightarrow (H \Rightarrow H2)$ is $H1 \Rightarrow H2$. Indeed, if we instantiate H with $H1$, we have $(H1 \Rightarrow H1) \rightarrow (H1 \Rightarrow H2)$, thus we deduce $H1 \Rightarrow H2$. Reciprocally, if we have $H1 \Rightarrow H2$ and $H \Rightarrow H1$, then by the transitivity property `himpl_trans`, we deduce $H \Rightarrow H2$.

Therefore, the previous entailment simplifies to:

$\text{wp} \text{ (if } b \text{ then } t1 \text{ else } t2) \ Q \implies \text{wp} \text{ (trm\_if (val\_bool } b) \ t1 \ t2) \ Q}.$

This is the wp-style reasoning rule for conditionals.

Lemma `wp_if` : $\forall \ b \ t1 \ t2 \ Q,$

$\text{wp} \text{ (if } b \text{ then } t1 \text{ else } t2) \ Q \implies \text{wp} \text{ (trm\_if (val\_bool } b) \ t1 \ t2) \ Q}.$

Proof using. `unfold wp. intros. intros h K. applys× eval_if. Qed.`

Equivalently, the rule may be stated with the conditional around the calls to `wp t1 Q` and `wp t2 Q`.

Exercise: 1 star, standard, optional (`wp_if'`) Prove the alternative statement of rule `wp_if`, either from `wp_if` or directly from `eval_if`.

Lemma `wp_if'` : $\forall \ b \ t1 \ t2 \ Q,$

$\text{(if } b \text{ then (wp } t1 \ Q) \text{ else (wp } t2 \ Q))} \implies \text{wp} \text{ (trm\_if } b \ t1 \ t2) \ Q.$

Proof using. *Admitted*.

□

Rule For Function Applications

Recall the reasoning rule for an application `(val_fun x t1) v2`.

Parameter `triple_app_fun` : $\forall \ x \ v1 \ v2 \ t1 \ H \ Q,$

$v1 = \text{val\_fun } x \ t1 \rightarrow$

$\text{triple} \text{ (subst } x \ v2 \ t1) \ H \ Q \rightarrow$

$\text{triple} \text{ (trm\_app } v1 \ v2) \ H \ Q.$

The corresponding `wp` rule is:

Lemma `wp_app_fun` : $\forall \ x \ v1 \ v2 \ t1 \ Q,$

$v1 = \text{val\_fun } x \ t1 \rightarrow$

$\text{wp} \text{ (subst } x \ v2 \ t1) \ Q \implies \text{wp} \text{ (trm\_app } v1 \ v2) \ Q.$

Proof using. `unfold wp. intros. intros h K. applys× eval_app_fun. Qed.`

A similar rule holds for the application of a recursive function.

Lemma `wp_app_fix` : $\forall \ f \ x \ v1 \ v2 \ t1 \ Q,$

$v1 = \text{val\_fix } f \ x \ t1 \rightarrow$

$\text{wp} \text{ (subst } x \ v2 \ (\text{subst } f \ v1 \ t1)) \ Q \implies \text{wp} \text{ (trm\_app } v1 \ v2) \ Q.$

Proof using. `unfold wp. intros. intros h K. applys× eval_app_fix. Qed.`

44.3 More Details

44.3.1 Combined Structural Rule

Recall the combined consequence-frame rule for `triple`.

Parameter *triple_conseq_frame* : $\forall H2\ H1\ Q1\ t\ H\ Q,$
 $\text{triple } t\ H1\ Q1 \rightarrow$
 $H \implies H1 \setminus * H2 \rightarrow$
 $Q1 \setminus *+ H2 \implies Q \rightarrow$
 $\text{triple } t\ H\ Q.$

Let us reformulate this rule using *wp*, replacing *triple t H Q* with $H \implies \text{wp } t\ Q$.

Exercise: 2 stars, standard, especially useful (*wp_conseq_frame_trans*) Prove the combined structural rule in *wp* style. Hint: exploit *wp_conseq_trans* and *wp_frame*.

Lemma *wp_conseq_frame_trans* : $\forall t\ H\ H1\ H2\ Q1\ Q,$
 $H1 \implies \text{wp } t\ Q1 \rightarrow$
 $H \implies H1 \setminus * H2 \rightarrow$
 $Q1 \setminus *+ H2 \implies Q \rightarrow$
 $H \implies \text{wp } t\ Q.$

Proof using. *Admitted*.

□

The combined structural rule for *wp* can actually be stated in a more concise way. The rule reads as follows: If you own a state from which the execution of *t* produces (a result and a state satisfying) *Q1* and you also own *H*; and if, furthermore, you can trade the combination of *Q1* and *H* against *Q2*; then you own a piece of state from which the execution of *t* produces *Q2*.

Exercise: 2 stars, standard, especially useful (*wp_conseq_frame*) Prove the concise version of the combined structural rule in *wp* style. (Many proofs are possible.)

Lemma *wp_conseq_frame* : $\forall t\ H\ Q1\ Q2,$
 $Q1 \setminus *+ H \implies Q2 \rightarrow$
 $(\text{wp } t\ Q1) \setminus * H \implies (\text{wp } t\ Q2).$

Proof using. *Admitted*.

□

Let us reformulate the rule *wp_conseq_frame* using a magic wand. The premise $Q1 \setminus *+ H \implies Q2$ can be rewritten as $H \implies (Q1 \setminus -^* Q2)$. By replacing *H* with $Q1 \setminus -^* Q2$ in the conclusion of *wp_conseq_frame*, we obtain the ramified rule for *wp*.

Lemma *wp_ramified* : forall t Q1 Q2, $(\text{wp } t\ Q1) \setminus * (Q1 \setminus -^* Q2) \implies (\text{wp } t\ Q2).$

This explanation suggests how one might have come up with the statement of the ramified frame rule.

44.4 Optional Material

44.4.1 Weakest Preconditions Derived from Triples, a First Route

Module WPFROMTRIPLE.

The lemma `wp_equiv` captures the characteristic property of `wp`, that is, $(H \implies \text{wp } t \ Q) \leftrightarrow (\text{triple } t \ H \ Q)$. The predicate `wp` can be defined in terms of `eval`, like `triple`. Interestingly, `wp` may also be defined in terms of `triple`. The idea is to define `wp t Q` as the predicate $\lambda H. H \ \backslash * \ \backslash [\text{triple } t \ H \ Q]$, which, reading literally, is satisfied by “any” heap predicate H which is a valid precondition for a triple for the term t and the postcondition Q .

Definition `wp_1 (t:trm) (Q:val→hprop) : hprop :=`
 $\lambda \exists (H:hprop), H \ \backslash * \ \backslash [\text{triple } t \ H \ Q]$.

Exercise: 3 stars, standard, especially useful (`wp_equiv_1`) Prove that the alternative definition `wp_1` satisfies the characteristic equivalence for weakest preconditions. Hint: use the consequence rule and the extraction rules.

Lemma `wp_equiv_1 :  $\forall t \ H \ Q,$`
 $(H \implies \text{wp\_1 } t \ Q) \leftrightarrow (\text{triple } t \ H \ Q)$.

Proof using. *Admitted*.

□

End WPFROMTRIPLE.

44.4.2 Weakest Preconditions Derived From Triples, a Second Route

Module WPFROMTRIPLE2.

The concrete definition for `wp` given above is expressed in terms of Separation Logic combinators. In contrast to this “high level” definition, there exists a more “low level” definition, expressed directly as a function over heaps. In this alternative definition, the heap predicate `wp t Q` is defined as a predicate that holds of a heap h if and only if the execution of t starting in exactly the heap h produces the post-condition Q . The latter statement is formally captured as `triple t (fun h' => h' = h) Q`. The low-level definition of `wp` is thus as shown below.

Definition `wp_2 (t:trm) (Q:val→hprop) : hprop :=`
 $\text{fun } (h:\text{heap}) \Rightarrow \text{triple } t \ (\text{fun } h' \Rightarrow (h' = h)) \ Q$.

Exercise: 4 stars, standard, optional (`wp_equiv_2`) Prove that the low-level definition `wp_2` also satisfies the characteristic equivalence $H \implies \text{wp } Q \leftrightarrow \text{triple } t \ H \ Q$. Hint: exploit the lemma `triple_named_heap` which was established as an exercise in the chapter Triples.

Lemma wp_equiv_2 : $\forall t H Q,$
 $(H \implies \text{wp } t Q) \leftrightarrow (\text{triple } t H Q).$

Proof using. *Admitted.*

□

End WPFROMTRIPLE2.

44.4.3 Equivalent Characterizations of **wp**

First, we establish that the equivalence $(\text{triple } t H Q) \leftrightarrow (H \implies \text{wp } t Q)$ defines a unique predicate **wp**. In other words, all possible definitions of **wp** are equivalent to each other. Thus, it really does not matter which concrete definition of **wp** we consider: they are all equivalent. Concretely, assume two predicates *wp1* and *wp2* to both satisfy the characteristic equivalence. We prove that they are equal.

Lemma wp_unique : $\forall wp1 wp2,$
 $(\forall t H Q, (\text{triple } t H Q) \leftrightarrow (H \implies wp1 t Q)) \rightarrow$
 $(\forall t H Q, (\text{triple } t H Q) \leftrightarrow (H \implies wp2 t Q)) \rightarrow$
 $wp1 = wp2.$

Proof using.

intros M1 M2. applies fun_ext_2. intros t Q. applies himpl_antisym.
 $\{ \text{rewrite} \leftarrow M2. \text{rewrite } M1. \text{auto.} \}$
 $\{ \text{rewrite} \leftarrow M1. \text{rewrite } M2. \text{auto.} \}$

Qed.

Second, we establish that the property of “being the weakest precondition” also uniquely characterizes the definition of **wp**. This property is the conjunction of two facts: $\text{wp } t Q$ must be a valid precondition for a triple involving *t* and *Q*; and $\text{wp } t Q$ must be entailed by any valid precondition of a triple involving *t* and *Q*. These two facts correspond to **wp_pre** and **wp_weakest**.

Exercise: 2 stars, standard, especially useful (wp_equiv_iff_wp_pre_and_wp_weakest)

Prove that the conjunction of the properties **wp_pre** and *wp_weakest* is equivalent to the property **wp_equiv**.

Lemma wp_equiv_iff_wp_pre_and_wp_weakest : $\forall wp',$
 $((\forall t Q, \text{triple } t (wp' t Q) Q) \wedge (\forall t H Q, \text{triple } t H Q \rightarrow H \implies wp' t Q))$
 $\leftrightarrow$
 $(\forall t H Q, H \implies wp' t Q \leftrightarrow \text{triple } t H Q).$ Proof using. *Admitted.*

□

44.4.4 Texan Triples

In this section, we show that specification triples can be presented in a different style using weakest preconditions. This style has appeared under the name “Texan triples” in certain work related to the Iris framework. The main motivation is to slightly simplify the implementation of the tactic *xapp*. The main drawback is that it makes specifications, one could argue, slightly less easy to read. For this reason, users of the Iris framework usually stick to the standard triple notation in their publications. Despite the readability issue, it may be interesting for the reader to be aware of the possibility of stating specifications directly using *wp*, as there might be useful applications of this technique in certain contexts.

Module **TEXANTRIPLES**.

1. An Example

Consider the specification triple for allocation.

Parameter *triple_ref* : $\forall v,$
 $\text{triple } (\text{val\_ref } v)$
 $\backslash []$
 $(\text{funloc } p \Rightarrow p \sim\sim > v).$

This specification may be equivalently reformulated in the form of an entailment, as shown below. Note that we purposely leave an empty heap predicate at the front to indicate where the precondition, if it were not empty, would go in the reformulation.

Parameter *wp_ref* : $\forall Q v,$
 $\backslash [] \backslash * (\backslash \forall p, p \sim\sim > v \backslash - * Q (\text{val\_loc } p)) \implies \text{wp } (\text{val\_ref } v) Q.$

In what follows, we describe a chain of transformations that can take us from the triple form to the *wp* form, and we establish the reciprocal. Afterwards, we will formalize the general pattern of the translation from a triple to a “texan triple”, that is, to a *wp*-based specification.

By replacing *triple t H Q* with $H \implies \text{wp } t Q$, the specification *triple_ref* can be reformulated as follows.

Lemma *wp_ref_0* : $\forall v,$
 $\backslash [] \implies \text{wp } (\text{val\_ref } v) (\text{funloc } p \Rightarrow p \sim\sim > v).$

Proof using. *intros. rewrite wp_equiv. applies triple_ref. Qed.*

We wish to cast the RHS in the form $\text{wp } (\text{val\_ref } v) Q$ for an abstract postcondition *Q*. To that end, we reformulate the above statement by including a magic wand relating the postcondition $(\text{funloc } p \Rightarrow p \sim\sim > v)$ with *Q*.

Lemma *wp_ref_1* : $\forall Q v,$
 $((\text{funloc } p \Rightarrow p \sim\sim > v) \backslash - * Q) \implies \text{wp } (\text{val\_ref } v) Q.$

Proof using. *intros. xchange (wp_ref_0 v). applies wp_ramified. Qed.*

This statement can be made slightly more readable by unfolding the definition of the magic wand for postconditions. Doing so amounts to quantifying explicitly on the result, named r .

Lemma `wp_ref_2` : $\forall Q\ v,$
 $(\forall r, (\exists p, \backslash[r = \text{val\_loc } p] \backslash* p \rightsquigarrow v) \backslash* Q\ r)$
 $\implies \text{wp } (\text{val\_ref } v)\ Q.$

Proof using. `intros. applys himpl_trans wp_ref_1. xsimpl. Qed.`

Interestingly, the variable r , which is equal to `val_loc p`, can now be substituted away, further increasing readability. We obtain the specification of `val_ref` in “Texan triple style”.

Lemma `wp_ref_3` : $\forall Q\ v,$
 $(\forall p, (p \rightsquigarrow v) \backslash* Q\ (\text{val\_loc } p)) \implies \text{wp } (\text{val\_ref } v)\ Q.$

Proof using.

`intros. applys himpl_trans_r wp_ref_2. xsimpl. intros ? p →.`
`xchange (hforall_specialize p).`

`Qed.`

2. The General Pattern

Most specification triples can be cast in the form: `triple t H (fun r ⇒ ∃ x1 .. xN,  $\backslash[r = v]$   $\backslash* H'$ )`. In such a specification:

- the value v may depend on the xi variables,
- the heap predicate H' may depend on r and the xi ,
- the number of existentials xi may vary, possibly be zero,
- the equality $\backslash[r = v]$ may be removed if no pure fact is needed about r .

Such a specification triple of the form `triple t H (fun r ⇒ ∃ x1 xN,  $\backslash[r = v]$   $\backslash* H'$ )` can be reformulated as the Texan triple: $(\forall x1\ xN, H \backslash* Q\ v) \implies \text{wp } t\ Q.$

We next formalize the equivalence between the two presentations, for the specific case where the specification involves a single auxiliary variable, named x . The statement below makes it explicit that v may depend on x , and that H may depend on r and x .

Lemma `texan_triple_equiv` : $\forall t\ H\ A\ (Hof:\text{val} \rightarrow A \rightarrow \text{hprop})\ (vof:A \rightarrow \text{val}),$
 $(\text{triple } t\ H\ (\text{fun } r \Rightarrow \exists x, \backslash[r = vof\ x] \backslash* Hof\ r\ x))$
 $\leftrightarrow (\forall Q, H \backslash* (\forall x, Hof\ (vof\ x)\ x \backslash* Q\ (vof\ x)) \implies \text{wp } t\ Q).$

Proof using.

`intros. rewrite ← wp_equiv. iff M.`
`{ intros Q. xchange M. applys wp_ramified_trans.`
`xsimpl. intros r x →.`
`xchange (hforall_specialize x). }`
`{ applys himpl_trans_r M. xsimpl. }`

`Qed.`

3. Further Examples

Section WpSpecRef.

The wp-style specifications of *ref*, *get*, and *set* are presented next.

Lemma wp_get : $\forall v p Q$,

$(p \sim\sim> v) \setminus * (p \sim\sim> v \setminus - * Q v) \implies \text{wp } (\text{val\_get } p) Q$.

Proof using.

intros. rewrite wp_equiv. *applies triple_conseq_frame*.

{ *applies triple_get*. } { *applies himpl_refl*. } { *xsimpl*. intros ? $\rightarrow$. auto. }

Qed.

Lemma wp_set : $\forall v w p Q$,

$(p \sim\sim> v) \setminus * (\forall r, p \sim\sim> w \setminus - * Q r) \implies \text{wp } (\text{val\_set } p w) Q$.

Proof using.

intros. rewrite wp_equiv. *applies triple_conseq_frame*.

{ *applies triple_set*. } { *applies himpl_refl*. }

{ intros *r*. *xchange* (hforall_specialize *r*). *xsimpl*. }

Qed.

Lemma wp_free : $\forall v p Q$,

$(p \sim\sim> v) \setminus * (\forall r, Q r) \implies \text{wp } (\text{val\_free } p) Q$.

Proof using.

intros. rewrite wp_equiv. *applies triple_conseq_frame*.

{ *applies triple_free*. } { *applies himpl_refl*. }

{ intros *r*. *xchange* (hforall_specialize *r*). }

Qed.

Alternatively, the specifications of *set* and *free* may advertize that the output value is the unit value.

Parameter *triple_set'* : $\forall w p v$,

triple (val_set *p w*)

$(p \sim\sim> v)$

$(\text{fun } r \Rightarrow \setminus [r = \text{val\_unit}] \setminus * p \sim\sim> w)$.

Lemma wp_set' : $\forall v w p Q$,

$(p \sim\sim> v) \setminus * (p \sim\sim> w \setminus - * Q \text{val\_unit}) \implies \text{wp } (\text{val\_set } p w) Q$.

Proof using.

intros. rewrite wp_equiv. *applies triple_conseq_frame*.

{ *applies triple_set'*. }

{ *applies himpl_refl*. }

{ *xsimpl*. intros ? $\rightarrow$. auto. }

Qed.

Parameter *triple_free'* : $\forall p v$,

triple (val_free *p*)

```

    (p ~~> v)
    (fun r => \[r = val_unit]).

```

Lemma wp_free' : $\forall v w p Q,$
 $(p ~~> v) \setminus * (Q \text{ val\_unit}) ==> \text{wp} (\text{val\_free } p) Q.$

Proof using.

```

    intros. rewrite wp_equiv. applys triple_conseq_frame.
    { applys triple_free'. }
    { applys himpl_refl. }
    { xsimpl. intros ? ->. auto. }

```

Qed.

End WpSpecRef.

4. Exercise

Let us put to practice the use of a Texan triple on a different example. Recall the function *incr* and its specification (from *Hprop.v*).

Parameter *incr* : val.

```

Parameter triple_incr :  $\forall (p:\text{loc}) (n:\text{int}),$ 
  triple (incr p)
    (p ~~> n)
    (fun v => \[v = val_unit] \setminus * (p ~~> (n+1))).

```

Exercise: 3 stars, standard, especially useful (wp_incr) State a Texan triple for *incr* as a lemma called *wp_incr*, then prove this lemma from *triple_incr*.

□

End TEXANTRIPLES.

44.4.5 Historical Notes

The idea of weakest precondition was introduced by *Dijkstra* 1975 (in Bib.v) in his seminal paper “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. Weakest preconditions provide a reformulation of Floyd-Hoare logic. Numerous practical verification tools leverage weakest preconditions, e.g. ESC/Java, Why3, Boogie, Spec

Chapter 45

Library SLF.WPgen

45.1 WPgen: Weakest Precondition Generator

```
Set Implicit Arguments.
From SLF Require Import WPsem.

Implicit Types f : var.
Implicit Types b : bool.
Implicit Types v : val.
Implicit Types h : heap.
Implicit Types P : Prop.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
```

45.2 First Pass

The previous chapter introduced a predicate called `wp` to describe the weakest precondition of a term `t` with respect to a given postcondition `Q`.

The weakest precondition `wp` was characterized by the equivalence $H \implies wp\ t\ Q$ iff `triple t H Q`. We proved “wp-style” reasoning rules, such as:

`wp t1 (fun r => wp t2 Q) ==> wp (trm_seq t1 t2) Q`

In this chapter, we introduce a function, called `wpgen`, to *compute* the weakest precondition of a term.

The value of `wpgen t Q` is defined recursively over the structure of the term `t`, and it computes a formula that is logically equivalent to `wp t Q`. The fundamental difference between `wp` and `wpgen` is that, whereas `wp t Q` is a predicate that we can reason about by applying reasoning rules, `wpgen t Q` is a predicate that we can reason about simply by *unfolding* its definition. (This sentence will probably make more sense to the reader later on in the chapter.)

Another key difference between `wp` and `wpgen` is that the formulae produced by `wpgen`

no longer refer to program syntax. In particular, all program variables involved in a term t in a statement of the form $\text{wp } t \ Q$ are replaced with Rocq variables when computing $\text{wpgen } t \ Q$. The benefit of eliminating program variables is that formulae produced by wpgen can be manipulated without the need to simplify substitutions of the form $\text{subst } x \ v \ t2$. Instead, the beta reduction mechanism of Rocq will automatically perform substitutions for Rocq variables.

The reader might have encountered the term “weakest precondition generator” in the past. Such generators take as input a program annotated with all function specifications and loop invariants and produce a set of proof obligations that need to be checked in order to conclude that the code indeed satisfies its logical annotations. In contrast, the weakest preconditions computed by wpgen apply to a piece of code independent of any specification or invariant. More precisely, the computation of $\text{wpgen } t \ Q$ treats the postcondition Q as an abstract variable.

At a high level, wpgen is a key ingredient for smoothing the user experience of conducting interactive proofs in Separation Logic. The x-tactics presented in the first two chapters of the course were built on top of wpgen . The matter of the present chapter is to show:

- how to define $\text{wpgen } t \ Q$ as a recursive function in Rocq,
- how to integrate support for the frame rule in this definition,
- how to carry out practical proofs using wpgen .

The rest of the “first pass” section gives a high-level tour of the steps of the construction of wpgen . The formal definitions and the set up of x-tactics follow in the “more details” section. The treatment of local functions is presented in the “optional material” section.

This chapter is probably the most technical of the course. The reader who finds it hard to follow the entire chapter may safely jump to the next chapter at any point after the “first pass” section.

45.2.1 The Basic Structure of wpgen

As first approximation, $\text{wpgen } t \ Q$ is defined as a recursive function that pattern matches on its argument t , and produces an appropriate heap predicate in each case. The definitions somewhat mimic the reasoning rules of wp . Let us show the pattern of the definition and comment it afterwards.

Fixpoint $\text{wpgen } (t:\text{trm}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } | \text{trm\_val } v \Rightarrow Q \ v \mid \text{trm\_var } x \Rightarrow \text{False} \mid \text{trm\_app } v1 \ v2 \Rightarrow \exists H, H \ \& \ \text{triple } t \ H \ Q \mid \text{trm\_let } x \ t1 \ t2 \Rightarrow \text{wpgen } t1 \ (\text{fun } v \Rightarrow \text{wpgen } (\text{subst } x \ v \ t2) \ Q) \dots \text{end}$. Recall from the previous chapter that the rule wp_val asserts that $Q \ v$ entails $\text{wp } (\text{trm\_val } v) \ Q$. Well, $\text{wpgen } (\text{trm\_val } v) \ Q$ is defined as $Q \ v$.

Likewise, recall that the rule wp_let asserts that $\text{wp } t1 \ (\text{fun } v \Rightarrow \text{wp } (\text{subst } x \ v \ t2) \ Q)$ entails $\text{wp } (\text{trm\_let } x \ t1 \ t2) \ Q$. Well, the definition of wpgen is such that $\text{wpgen } (\text{trm\_let } x$

$t1\ t2) \ Q$ is defined to be $wpgen\ t1\ (\text{fun } v \Rightarrow wpgen\ (\text{subst } x\ v\ t2)\ Q)$, that is, the same expression as in wp_let only with wp replaced by $wpgen$.

An interesting case is $wgen\ (\text{trm\_var } x)\ Q$. There are no rules on wp to reason about variables, because they correspond to stuck terms. To reflect the fact that $wgen\ (\text{trm\_var } x)\ Q$ cannot be derived, we define it as $\backslash[\text{False}]$.

The last key case is that of function calls. Consider a term t that consists of a function application $\text{trm\_app } v1\ v2$. How should we define $wpgen\ (\text{trm\_app } t1\ t2)\ Q$? We need to define it to be a heap predicate, call it H , such that the function call $\text{trm\_app } t1\ t2$ admits H as precondition and Q as postcondition. In other words, the definition of $wpgen\ (\text{trm\_app } t1\ t2)\ Q$ should consist of a heap predicate H such that $\text{triple } t\ H\ Q$ holds. We do not know what this H should be, so we simply quantify it existentially. This suggests why the definition of $wpgen\ (\text{trm\_app } t1\ t2)\ Q$ should be $\backslash\exists\ H,\ H\ \backslash*\ \backslash[\text{triple } t\ H\ Q]$.

To obtain the actual definition of $wpgen$, we need to refine the above definition in four steps.

45.2.2 Step 1: $wpgen$ as a Recursive Function over Terms

In a first step, we modify the definition in order to make it structurally recursive. Indeed, in the above the recursive call $wpgen\ (\text{subst } x\ v\ t2)$ is not made on a strict subterm of $\text{trm\_let } x\ t1\ t2$, so Rocq will reject this definition as it stands.

To fix the issue, we change the definition to the form $wpgen\ E\ t\ Q$, where E denotes an association list bindings values to variables. The intention is that $wpgen\ E\ t\ Q$ computes the weakest precondition for the term obtained by substituting all the bindings from E in t . This term is described by the operation $\text{isubst } E\ t$, which is formally defined later in the chapter.

The updated definition looks as follows. Observe how, when traversing $\text{trm\_let } x\ t1\ t2$, the context E gets extended as $(x,v)::E$. Observe also how, when reaching a variable x , a lookup for x into the context E is performed for recovering the value bound to x .

Fixpoint $wpgen\ (E:\text{ctx})\ (t:\text{trm})\ (Q:\text{val}\rightarrow\text{hprop}) : \text{hprop} := \text{match } t \text{ with } | \text{trm\_val } v \Rightarrow Q\ v \mid \text{trm\_var } x \Rightarrow \text{match lookup } x\ E \text{ with } | \text{Some } v \Rightarrow Q\ v \mid \text{None} \Rightarrow \backslash\text{False} \text{ end } | \text{trm\_app } v1\ v2 \Rightarrow \backslash\text{exists } H,\ H\ \backslash*\ \backslash\text{triple } (\text{isubst } E\ t)\ H\ Q \mid \text{trm\_let } x\ t1\ t2 \Rightarrow wpgen\ E\ t1\ (\text{fun } v \Rightarrow wpgen\ ((x,v)::E)\ t2\ Q) \dots \text{end}$.

45.2.3 Step 2: $wpgen$ Reformulated to Eagerly Match on the Term

In a second step, we slightly tweak the definition in order to move the $\text{fun } (Q:\text{val}\rightarrow\text{hprop})$, by which the postcondition is taken as argument, to place it inside every case of the pattern matching. The idea is to make it explicit that $wpgen\ E\ t$ can be computed without any knowledge of Q . In the definition shown below, formula is a shorthand for the type $(\text{val}\rightarrow\text{hprop})\rightarrow\text{hprop}$, which is the type of $wpgen\ E\ t$.

Fixpoint $wpgen\ (E:\text{ctx})\ (t:\text{trm}) : \text{formula} := \text{match } t \text{ with } | \text{trm\_val } v \Rightarrow \text{fun } (Q:\text{val}\rightarrow\text{hprop}) \Rightarrow Q\ v \mid \text{trm\_var } x \Rightarrow \text{fun } (Q:\text{val}\rightarrow\text{hprop}) \Rightarrow \text{match lookup } x\ E \text{ with } | \text{Some } v \Rightarrow Q\ v \mid \text{None} \Rightarrow \backslash\text{False} \text{ end } | \text{trm\_app } t1\ t2 \Rightarrow \text{fun } (Q:\text{val}\rightarrow\text{hprop}) \Rightarrow \backslash\text{exists } H,\ H\ \backslash*$

$\backslash \text{triple } (\text{isubst } E \ t) \ H \ Q \mid \text{trm\_let } x \ t1 \ t2 \Rightarrow \text{fun } (Q:\text{val} \rightarrow \text{hprop}) \Rightarrow \text{wpngen } E \ t1 \ (\text{fun } v \Rightarrow \text{wpngen } ((x,v)::E) \ t2 \ Q) \ \dots \text{end.}$

45.2.4 Step 3: **wpngen** Reformulated with Auxiliary Definitions

In a third step, we introduce auxiliary definitions to improve the readability of the output of **wpngen**. For example, we let **wpngen_val** v be a shorthand for $\text{fun } (Q:\text{val} \rightarrow \text{hprop}) \Rightarrow Q \ v$. Likewise, we let **wpngen_let** $F1 \ F2$ be a shorthand for $\text{fun } (Q:\text{val} \rightarrow \text{hprop}) \Rightarrow F1 \ (\text{fun } v \Rightarrow F2 \ Q)$, where v can appear in $F2$. Using these auxiliary definitions, the definition of **wpngen** can be reformulated as follows.

Fixpoint **wpngen** ($E:\text{ctx}$) ($t:\text{trm}$) : formula := match t with | $\text{trm\_val } v \Rightarrow \text{wpngen\_val } v \mid \text{trm\_var } x \Rightarrow \text{wpngen\_var } E \ x \mid \text{trm\_app } t1 \ t2 \Rightarrow \text{wpngen\_app } (\text{isubst } E \ t) \mid \text{trm\_let } x \ t1 \ t2 \Rightarrow \text{wpngen\_let } (\text{wpngen } E \ t1) \ (\text{fun } v \Rightarrow \text{wpngen } ((x,v)::E) \ t2) \ \dots \text{end.}$ Each of the auxiliary definitions introduced for **wpngen** comes with a custom notation that enables a nice display of the output of **wpngen**. For example, the notation $\text{Let}' \ v := F1 \text{ in } F2$ stands for **wpngen_let** $F1 \ (\text{fun } v \Rightarrow F2)$. Thanks to this notation, the result of computing **wpngen** on a source term $\text{Let } x := t1 \text{ in } t2$, which is an Rocq expression of type **trm**, is a Rocq expression of type **formula** displayed as $\text{Let}' \ x := F1 \text{ in } F2$.

Thanks to these auxiliary definitions and pieces of notation, the formula that **wpngen** produces as output reads pretty much like the source term provided as input. The benefits, illustrated in the first two chapters (**Basic** and **Repr**), is that the proof obligations can be easily related to the source code that they arise from.

45.2.5 Step 4: **wpngen** Augmented with Support for Structural Rules

In a fourth step, we refine the definition of **wpngen** in order to equip it with inherent support for applications of the structural rules of the logic, namely the frame rule and the rule of consequence. To achieve this, we carefully craft a predicate called **mkstruct** and insert it at the head of the output of every call to **wpngen**, including all its recursive calls. In other words, the definition of **wpngen** hereafter has the following structure.

Fixpoint **wpngen** ($E:\text{ctx}$) ($t:\text{trm}$) : formula := **mkstruct** (match t with | $\text{trm\_val } v \Rightarrow \dots \mid \dots \text{end}$). Without going into the details at this stage, the predicate **mkstruct** is a function of type **formula** $\rightarrow$ **formula** that captures the essence of the wp-style consequence-frame rule **wp_conseq_frame**, presented the previous chapter. This rule asserts that $Q1 \setminus^* H \implies Q2$ implies $(\text{wp } t \ Q1) \setminus^* H \implies (\text{wp } t \ Q2)$.

This concludes our little journey towards the definition of **wpngen**. For conducting proofs in practice, it remains to state lemmas and define “x-tactics” to assist the user in the manipulation of the formulae produced by **wpngen**. Ultimately, the end-user only manipulates “x-tactics” as in the first two chapters, without ever being required to understand how **wpngen** is defined.

The “more details” section recapitulates each of the four steps presented in the above summary, but explaining in detail all the Rocq definitions involved.

45.3 More Details

45.3.1 Definition of **wpgen** for Each Term Construct

The function **wpgen** computes a heap predicate that has the same meaning as **wp**. In essence, **wpgen** takes the form of a recursive function that, like **wp**, expects a term **t** and a postcondition **Q** and produces a heap predicate. The function is recursively defined, and its result is guided by the structure of the term **t**. In essence, the definition of **wpgen** has the following shape:

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => .. |
trm_seq t1 t2 => .. | trm_let x t1 t2 => .. | trm_var x => .. | trm_app t1 t2 => .. |
trm_fun x t1 => .. | trm_fix f x t1 => .. | trm_if v0 t1 t2 => .. end.
```

Our first goal is to figure out how to fill in the dots for each of the term constructors. The intention that guides us is the soundness theorem for **wpgen**, which has the following form:

$\text{wpgen } t \ Q \implies \text{wp } t \ Q$ This entailment asserts in particular that, if we are able to establish a statement of the form $H \implies \text{wpgen } t \ Q$, then we can derive $H \implies \text{wp } t \ Q$ from it. The latter is also equivalent to $\text{triple } t \ H \ Q$. Thus, **wpgen** can be viewed as a practical tool for establishing triples.

The reciprocal implication, namely $\text{wp } t \ Q \implies \text{wpgen } t \ Q$, would correspond to a completeness theorem. Completeness is beyond the scope of this course. The historical notes section, located at the end of the chapter, contains a few comments on this question.

Definition of **wpgen** for Values

Consider first the case of a value **v**. Recall the reasoning rule for values in weakest-precondition style.

Parameter $\text{wp\_val} : \forall v \ Q,$
 $Q \ v \implies \text{wp } (\text{trm\_val } v) \ Q.$

The soundness theorem for **wpgen** requires the entailment $\text{wpgen } (\text{trm\_val } v) \ Q \implies \text{wp } (\text{trm\_val } v) \ Q$ to hold.

To satisfy this entailment, according to the rule **wp_val**, it suffices to define $\text{wpgen } (\text{trm\_val } v) \ Q$ as $Q \ v$.

Concretely, we fill in the first dots as follows:

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v ...
```

Definition of **wpgen** for Functions

Consider the case of a function definition $\text{trm\_fun } x \ t$. Recall that the **wp** reasoning rule for functions is very similar to the one for values.

Parameter $\text{wp\_fun} : \forall x \ t1 \ Q,$
 $Q \ (\text{val\_fun } x \ t1) \implies \text{wp } (\text{trm\_fun } x \ t1) \ Q.$

Following this rule, we define $\text{wpgen } (\text{trm\_fun } x \ t1) \ Q$ as $Q \ (\text{val\_fun } x \ t1)$. Likewise for recursive functions, we define $\text{wpgen } (\text{trm\_fix } f \ x \ t1) \ Q$ as $Q \ (\text{val\_fix } f \ x \ t1)$.

Fixpoint $\text{wpgen } (t:\text{trm}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } \dots \mid \text{trm\_fun } x \ t1 \Rightarrow Q \ (\text{val\_fun } x \ t1) \mid \text{trm\_fix } f \ x \ t1 \Rightarrow Q \ (\text{val\_fix } f \ x \ t1) \dots$

An interesting question that arises here is: why does $\text{wpgen } (\text{trm\_fun } x \ t1) \ Q$ not trigger a recursive call to wpgen on $t1$? The long answer is detailed in the last section of this chapter (module WPGENREC). The short answer is:

- it is not technically needed to recurse in the body of local functions, in the sense that the user does not lose any ability to reason about local functions;
- it may be interesting to recurse the body of local functions because it may save the need for the user to manually invoke the tactic xwp for each local function.

Definition of wpgen for Sequences

Recall the wp reasoning rule for a sequence $\text{trm\_seq } t1 \ t2$.

Parameter $\text{wp\_seq} : \forall \ t1 \ t2 \ Q,$

$\text{wp } t1 \ (\text{fun } v \Rightarrow \text{wp } t2 \ Q) \Rightarrow \text{wp } (\text{trm\_seq } t1 \ t2) \ Q.$

The intention is for wpgen to admit the same semantics as wp . We thus expect the definition of $\text{wpgen } (\text{trm\_seq } t1 \ t2) \ Q$ to have a similar shape to $\text{wp } t1 \ (\text{fun } v \Rightarrow \text{wp } t2 \ Q)$.

We therefore define $\text{wpgen } (\text{trm\_seq } t1 \ t2) \ Q$ as $\text{wpgen } t1 \ (\text{fun } v \Rightarrow \text{wpgen } t2 \ Q)$. The definition of wpgen is extended as follows:

Fixpoint $\text{wpgen } (t:\text{trm}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } \dots \mid \text{trm\_seq } t1 \ t2 \Rightarrow \text{wpgen } t1 \ (\text{fun } v \Rightarrow \text{wpgen } t2 \ Q) \dots$

Definition of wpgen for Let-Bindings

The case of let bindings is similar to that of sequences, except that it involves a substitution. Recall the wp rule:

Parameter $\text{wp\_let} : \forall \ x \ t1 \ t2 \ Q,$

$\text{wp } t1 \ (\text{fun } v \Rightarrow \text{wp } (\text{subst } x \ v \ t2) \ Q) \Rightarrow \text{wp } (\text{trm\_let } x \ t1 \ t2) \ Q.$

We fill in the dots as follows:

Fixpoint $\text{wpgen } (t:\text{trm}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } \dots \mid \text{trm\_let } x \ t1 \ t2 \Rightarrow \text{wpgen } t1 \ (\text{fun } v \Rightarrow \text{wpgen } (\text{subst } x \ v \ t2) \ Q) \dots$ Observe that in the above definition, the second recursive call is invoked on $\text{subst } x \ v \ t2$, which is not a strict subterm of t . As explained earlier, this recursion pattern motivates the introduction of substitution contexts, written E , to delay the computation of substitutions until the leaves of the recursion.

Definition of `wpgen` for Variables

We have seen no reasoning rules for establishing a triple for a program variable, that is, to prove `triple (trm_var x) H Q`. Indeed, `trm_var x` is a stuck term: its execution does not produce an output. A source term may of course contain program variables, but all these variables should be substituted away before the execution reaches them.

In the case of the function `wpgen`, a variable bound by a let-binding get substituted while traversing that let-binding construct. Thus, if a free variable is reached by `wpgen`, it means that this variable was originally a dangling free variable, and therefore that the initial source term was invalid.

But, although we have no reasoning rules for `triple (trm_var x) H Q` nor for $H \implies \text{wp}(\text{trm\_var } x) Q$, we nevertheless have to provide some meaningful definition for `wpgen (trm_var x) Q`. This definition should capture the fact that this case must not happen. The heap predicate `\[False]` appropriately captures this intention.

Fixpoint `wpgen (t:trm) (Q:val->hprop) : hprop := match t with ... | trm_var x => \[False] ...`

Remark: the use of `\[False]` embodies the fact that, technically, we could have stated a Separation Logic rule for free variables, using `False` as precondition. There are three ways of presenting this rule:

Lemma `wp_var` : $\forall x Q$,

`\[False] ==> wp (trm_var x) Q`.

Proof using. `intros. intros h Hh. destruct (hpure_inv Hh). false. Qed.`

Lemma `triple_var` : $\forall x H Q$,

`False` $\rightarrow$

`triple (trm_var x) H Q`.

Proof using. `intros. false. Qed.`

Lemma `triple_var'` : $\forall x Q$,

`triple (trm_var x) \[False] Q`.

Proof using. `intros. rewrite <- wp_equiv. applys wp_var. Qed.`

All these rules are correct, albeit totally useless.

Definition of `wpgen` for Function Applications

Consider an application in A-normal form, that is, an application of the form `trm_app v1 v2`. We have seen `wp`-style rules to reason about the application of a known function, e.g. `trm_app (val_fun x t1) v2`. Yet, typically the function `v1` is a function specified in terms of `triple`. (If `v1` is a primitive function, or if it comes from a function argument, it might not even be the case that `v1` admits the shape `val_fun x t1`.)

Thus, we wish to reason about an application `trm_app v1 v2` by exhibiting a triple of the form `triple (trm_app v1 v2) H Q`. As suggested earlier in the chapter, the definition of `wpgen (trm_app t1 t2) Q` needs to correspond to some heap predicate H for which `triple (trm_app v1 v2) H Q` holds.

The formula that formalizes this intuition is:

Fixpoint $\text{wpgen} (t:\text{trm}) (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } \dots \mid \text{trm\_app } t1 \ t2 \Rightarrow \text{exists } H, H \ \backslash^* \ \text{triple } t \ H \ Q \ \dots$

Another possibility would be to define $\text{wpgen} (\text{trm\_app } t1 \ t2) \ Q$ as $\text{wp} (\text{trm\_app } t1 \ t2) \ Q$. In other words, to define wpgen for a function application, we could fall back to the semantic definition of wp .

Fixpoint $\text{wpgen} (t:\text{trm}) (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } \dots \mid \text{trm\_app } t1 \ t2 \Rightarrow \text{wp } t \ Q \ \dots$

However, this definition would require converting the wp into a triple each time, because specifications are, by convention, expressed using the predicate triple .

Remark: we assume throughout the course that terms are written in A-normal form. Nevertheless, we need to define wpgen even on terms that are not in A-normal form. One possibility is to map all these terms to $\backslash[\text{False}]$. However, even in the case of an application of the form $\text{trm\_app } t1 \ t2$ where $t1$ and $t2$ are not both values, it is still correct to define $\text{wpgen} (\text{trm\_app } t1 \ t2)$ as $\text{wp} (\text{trm\_app } t1 \ t2)$. So, we need not bother checking in the definition of wpgen that the arguments of trm_app are actually values.

Definition of wpgen for Conditionals

Finally, consider an if-statement. Recall the wp -style reasoning rule stated using a Rocq conditional.

Parameter $\text{wp\_if} : \forall (b:\text{bool}) \ t1 \ t2 \ Q,$
 $(\text{if } b \text{ then } (\text{wp } t1 \ Q) \text{ else } (\text{wp } t2 \ Q)) \Rightarrow \text{wp} (\text{trm\_if } (\text{val\_bool } b) \ t1 \ t2) \ Q.$

Typically, a source program may feature a conditional $\text{trm\_if } (\text{trm\_val } x) \ t1 \ t2$ that, after substitution for x , becomes $\text{trm\_if } (\text{trm\_val } v) \ t1 \ t2$, for some abstract v of type val . Yet, in general, because we are working here with possibly ill-typed programs, this value is not known to be a boolean value. We therefore need to define wpgen for all if-statements of the form $\text{trm\_if } (\text{trm\_val } v0) \ t1 \ t2$, where $v0$ could be a value of unknown shape.

However, the reasoning rule wp_if stated above features a right-hand side of the form $\text{wp} (\text{trm\_if } (\text{val\_bool } b) \ t1 \ t2) \ Q$. It only applies when the first argument of trm_if is syntactically a boolean value b . To resolve this mismatch, the definition of wpgen for a term $\text{trm\_if } t0 \ t1 \ t2$ quantifies existentially over a boolean value b such that $t0 = \text{trm\_val } (\text{val\_bool } b)$. This way, if $t0$ is not a boolean value, then $\text{wpgen} (\text{trm\_if } t0 \ t1 \ t2) \ Q$ is equivalent to $\backslash[\text{False}]$. Otherwise, if $t0$ is of the form $\text{val\_bool } b$, and we are able to perform a case analysis on b , to switch between $\text{wpgen } t1 \ Q$ and $\text{wpgen } t2 \ Q$ depending on the value of b .

The formal definition is:

Fixpoint $\text{wpgen} (t:\text{trm}) (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } \dots \mid \text{trm\_if } t0 \ t1 \ t2 \Rightarrow \backslash \text{exists } (b:\text{bool}), \backslash t0 = \text{trm\_val } (\text{val\_bool } b) \ \backslash^* (\text{if } b \text{ then } (\text{wpgen } t1) \ Q \text{ else } (\text{wpgen } t2) \ Q) \ \dots$

Summary of the Definition So Far

In summary, we have defined:

Fixpoint $\text{wpgen} (t:\text{trm}) (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} := \text{match } t \text{ with } | \text{trm\_val } v \Rightarrow Q \ v \mid \text{trm\_fun } x \ t1 \Rightarrow Q \ (\text{val\_fun } x \ t) \mid \text{trm\_fix } f \ x \ t1 \Rightarrow Q \ (\text{val\_fix } f \ x \ t) \mid \text{trm\_seq } t1 \ t2 \Rightarrow \text{wpgen } t1 \ (\text{fun } v \Rightarrow \text{wpgen } t2 \ Q) \mid \text{trm\_let } x \ t1 \ t2 \Rightarrow \text{wpgen } t1 \ (\text{fun } v \Rightarrow \text{wpgen } (\text{subst } x \ v \ t2) \ Q) \mid \text{trm\_var } x \Rightarrow \text{False} \mid \text{trm\_app } t1 \ t2 \Rightarrow \text{exists } H, H \ \ast \ \text{triple } t \ H \ Q \mid \text{trm\_if } t0 \ t1 \ t2 \Rightarrow \text{exists } (b:\text{bool}), \ t0 = \text{trm\_val } (\text{val\_bool } b) \ \ast \ (\text{if } b \text{ then } (\text{wpgen } t1) \ Q \text{ else } (\text{wpgen } t2) \ Q) \text{ end.}$

As pointed out earlier, this definition is not structurally recursive and is thus not accepted by Rocq, due to the recursive call $\text{wpgen} (\text{subst } x \ v \ t2) \ Q$. Our next step is to fix this issue.

45.3.2 Computing with wpgen

To make wpgen structurally recursive, the idea is to postpone the substitutions until the leaves of the recursion. To that end, we introduce a substitution context, written E , to record the substitutions that must be performed. Concretely, we modify the function to take the form $\text{wpgen } E \ t$, where E denotes a list of bindings from variables to values. The intention is that $\text{wpgen } E \ t$ should compute the weakest precondition for the term $\text{isubst } E \ t$, which denotes the result of substituting all bindings from E inside the term t .

Contexts

A context E is represented as an association list relating variables to values. Recall that all values are closed, i.e., without free variables.

Definition $\text{ctx} : \text{Type} := \text{list } (\text{var} \times \text{val})$.

The “iterated substitution” operation, written $\text{isubst } E \ t$, describes the substitution of *all* the bindings from a context E into a term t .

Its implementation is standard: the function traverses the term recursively and, when reaching a variable, performs a lookup in the context E . The operation needs to take care to respect variable shadowing. To that end, when traversing a binder for a variable x , all occurrences of x that might previously exist in E are removed.

The formal definition of isubst involves two auxiliary functions – lookup and removal – on association lists. The definition of the operation $\text{lookup } x \ E$ on association lists is standard. It returns an option on a value.

```
Fixpoint lookup (x:var) (E:ctx) : option val :=
  match E with
  | nil ⇒ None
  | (y,v)::E1 ⇒ if var_eq x y
                  then Some v
                  else lookup x E1
  end.
```

The definition of the removal operation, written $\text{rem } x \ E$, is also standard. It returns a filtered context.

```
Fixpoint rem (x:var) (E:ctx) : ctx :=
```

```

match  $E$  with
| nil  $\Rightarrow$  nil
|  $(y, v) :: E1 \Rightarrow$ 
    let  $E1' := \text{rem } x \ E1$  in
    if var_eq  $x \ y$  then  $E1'$  else  $(y, v) :: E1'$ 
end.

```

The definition of the operation $\text{isubst } E \ t$ can then be expressed as a recursive function over the term t . It invokes $\text{lookup } x \ E$ when it reaches a variable x . It invokes $\text{rem } x \ E$ when traversing a binding on the name x .

```

Fixpoint isubst ( $E:\text{ctx}$ ) ( $t:\text{trm}$ ) :  $\text{trm} :=$ 
  match  $t$  with
  | trm_val  $v \Rightarrow$ 
       $v$ 
  | trm_var  $x \Rightarrow$ 
      match lookup  $x \ E$  with
      | None  $\Rightarrow t$ 
      | Some  $v \Rightarrow v$ 
      end
  | trm_fun  $x \ t1 \Rightarrow$ 
      trm_fun  $x \ (\text{isubst } (\text{rem } x \ E) \ t1)$ 
  | trm_fix  $f \ x \ t1 \Rightarrow$ 
      trm_fix  $f \ x \ (\text{isubst } (\text{rem } x \ (\text{rem } f \ E)) \ t1)$ 
  | trm_app  $t1 \ t2 \Rightarrow$ 
      trm_app  $(\text{isubst } E \ t1) \ (\text{isubst } E \ t2)$ 
  | trm_seq  $t1 \ t2 \Rightarrow$ 
      trm_seq  $(\text{isubst } E \ t1) \ (\text{isubst } E \ t2)$ 
  | trm_let  $x \ t1 \ t2 \Rightarrow$ 
      trm_let  $x \ (\text{isubst } E \ t1) \ (\text{isubst } (\text{rem } x \ E) \ t2)$ 
  | trm_if  $t0 \ t1 \ t2 \Rightarrow$ 
      trm_if  $(\text{isubst } E \ t0) \ (\text{isubst } E \ t1) \ (\text{isubst } E \ t2)$ 
  end.

```

Remark: it is also possible to define the substitution by iterating the unary substitution subst over the list of bindings from E . However, this alternative approach yields a less efficient function and leads to more complicated proofs.

We now present the reformulated definition of $\text{wpgen } E \ t$, case by case. Throughout these definitions, recall that $\text{wpgen } E \ t$ is interpreted as the weakest precondition of $\text{isubst } E \ t$.

The Let-Binding Case

When the function wpgen traverses a let-binding, rather than eagerly performing a substitution, it simply extends the current context. Concretely, a call to $\text{wpgen } E \ (\text{trm\_let } x \ t1$

$t2$) triggers a recursive call to `wpgen ((x,v)::E) t2`. The corresponding definition is:

```
Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct (match t with ... | trm_let x t1 t2
=> fun Q => (wpgen E t1) (fun v => wpgen ((x,v)::E) t2) ... ) end.
```

The Variable Case

When `wpgen` reaches a variable, it looks for a binding on the variable x inside the context E . Concretely, the evaluation of `wpgen E (trm_var x)` triggers a call to `lookup x E`. If the context E binds the variable x to some value v , then the operation `lookup x E` returns `Some v`. In that case, `wpgen` returns the weakest precondition for the value v , that is, $Q\ v$. Otherwise, if E does not bind x , the lookup operation returns `None`. In that case, `wpgen` returns `\[False]`, the weakest precondition for a stuck program.

```
Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct (match t with ... | trm_var x =>
fun Q => match lookup x E with | Some v => Q v | None => \False end ... ) end.
```

The Application Case

Recall the definition of `wpgen` without substitution contexts:

```
Fixpoint wpgen (t:trm) (Q:val->hprop) : hprop := match t with .. | trm_app t1 t2 =>
fun Q => exists H, H \* \triple t H Q
```

In the definition with contexts, the term t appearing in `triple t H Q` needs to be replaced with `isubst E t`.

```
Fixpoint wpgen (t:trm) : formula := mkstruct (match t with ... | trm_app v1 v2 => fun
Q => exists H, H \* \triple (isubst E t) H Q ..
```

The Function Definition Case

Suppose t is a function definition, for example `trm_fun x t1`. Here again, the formula `wpgen E t` is interpreted as the weakest precondition of `isubst E t`.

By unfolding the definition of `isubst` in the case where t is `trm_fun x t1`, we obtain `trm_fun x (isubst (rem x E) t1)`.

The corresponding value is `trm_val x (isubst (rem x E) t1)`. The weakest precondition for that value is `fun Q => Q (val_fun x (isubst (rem x E) t1))`.

Thus, `wpgen E t` handles functions (and recursive functions) as follows:

```
Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct (match t with ... | trm_fun x t1
=> fun Q => Q (val_fun x (isubst (rem x E) t1)) | trm_fix f x t1 => fun Q => Q (val_fix
f x (isubst (rem x (rem f E)) t1)) ... ) end.
```

`wpgen`: At Last, an Executable Function

```
Module WPGENEXEC1.
```

At last, we arrive at a definition of `wpgen` that type-checks in Rocq and that can be used to effectively compute weakest preconditions in Separation Logic.

```

Fixpoint wpgen (E:ctx) (t:trm) (Q:val→hprop) : hprop :=
  match t with
  | trm_val v ⇒ Q v
  | trm_fun x t1 ⇒ Q (val_fun x (isubst (rem x E) t1))
  | trm_fix f x t1 ⇒ Q (val_fix f x (isubst (rem x (rem f E)) t1))
  | trm_seq t1 t2 ⇒ wpgen E t1 (fun v ⇒ wpgen E t2 Q)
  | trm_let x t1 t2 ⇒ wpgen E t1 (fun v ⇒ wpgen ((x,v)::E) t2 Q)
  | trm_var x ⇒
      match lookup x E with
      | Some v ⇒ Q v
      | None ⇒ \[False]
      end
  | trm_app v1 v2 ⇒ \∃ H, H \* \[triple (isubst E t) H Q]
  | trm_if t0 t1 t2 ⇒
      \∃ (b:bool), \[t0 = trm_val (val_bool b)]
      \* (if b then (wpgen E t1) Q else (wpgen E t2) Q)
  end.

```

In the next chapter, we will establish the soundness of the `wpgen`. For the moment, we simply state the soundness theorem, then explain how it can be exploited for verifying concrete programs.

Parameter `wpgen_sound` : $\forall E t Q,$
`wpgen E t Q ==> wp (isubst E t) Q.`

The entailment above asserts in particular that we can derive `triple t H Q` by proving $H ==> \text{wpgen } t \text{ } Q$. A useful corollary combines this soundness theorem with the rule `triple_app_fun`, which allows establishing triples for functions.

Recall `triple_app_fun` from chapter [Rules](#).

Parameter `triple_app_fun` : $\forall x v1 v2 t1 H Q,$
`v1 = val_fun x t1 →`
`triple (subst x v2 t1) H Q →`
`triple (trm_app v1 v2) H Q.`

Reformulating the rule above into a rule for `wpgen` takes 3 steps.

First, we rewrite the premise `triple (subst x v2 t1) H Q` using `wp`. It becomes $H ==> \text{wp (subst x v2 t1) } Q$.

Second, we observe that the term `subst x v2 t1` is equal to `isubst ((x,v2)::nil) t1`. (This equality is captured by the lemma `subst_eq_isubst_one` proved in the next chapter.) Thus, the heap predicate `wp (subst x v2 t1) Q` is equivalent to `wp (isubst ((x,v2)::nil) t1)`.

Third, according to `wpgen_sound`, `wp (isubst ((x,v2)::nil) t1)` is entailed by `wpgen ((x,v2)::nil) t1`. Thus, we can use the latter as premise in place of the former. We thereby obtain the following lemma, which is at the heart of the implementation of the tactic `xwp`.

Parameter `triple_app_fun_from_wpgen` : $\forall v1 v2 x t1 H Q,$
`v1 = val_fun x t1 →`

```

H ==> wpgen ((x, v2) : : nil) t1 Q →
triple (trm_app v1 v2) H Q.

```

Executing **wpgen** on a Concrete Program

Import EXAMPLEPROGRAMS.

Let us exploit `triple_app_fun_from_wpgen` to demonstrate the computation of **wpgen** on a practical program. Recall the function *incr* (defined in Rules), and its specification below.

```

Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
    (p ~~> n)
    (fun _ => p ~~> (n+1)).

```

Proof using.

```

intros. applys triple_app_fun_from_wpgen. { reflexivity. }
simpl. Abort.

```

At the end of the above proof fragment, the goal takes the form $H ==> F Q$, where H denotes the precondition, Q the postcondition, and F is a formula describing the body of the function *incr*. Inside this formula, the reader can spot triples for the primitive operations involved. Observe that the goal is nevertheless somewhat hard to relate to the original program. In what follows, we explain how to remedy the situation by setting up **wpgen** so that its output is human-readable and resembles the original source code.

Consequence and Frame Properties for **wpgen**

Exercise: 3 stars, standard, especially useful (wpgen_conseq) Prove that **wpgen** $E\ t$ satisfies the same consequence rule as **wp** t . The statement is directly adapted from `wp_conseq`, from **WPsem**. Hint: begin the proof with `intros t. induction t..`

```

Lemma wpgen_conseq : ∀ t E Q1 Q2,
  Q1 ==> Q2 →
  wpgen E t Q1 ==> wpgen E t Q2.

```

Proof using. *Admitted*.

□

Exercise: 4 stars, standard, especially useful (wpgen_frame) Prove that **wpgen** $E\ t$ satisfies the same frame rule as **wp** t . The statement is directly adapted from `wp_frame`, from **WPsem**. The sequence case is not easy.

```

Lemma wpgen_frame : ∀ t E H Q,
  (wpgen E t Q) \* H ==> wpgen E t (Q \*+ H).

```

Proof using. *Admitted*.

□

End WPGENEXEC1.

45.3.3 Optimizing the Readability of `wpngen`'s Output

To improve the readability of the formulae produced by `wpngen`, we take the following 3 steps:

- first, we modify the presentation of `wpngen` so that the `fun (Q:val→hprop) ⇒` appears inside the branches of the `match t with` rather than around it,
- second, we introduce one auxiliary definition for each branch of the `match t with`, and
- third, we introduce a Notation for each of these auxiliary definitions.

Reability Step 1: Moving the Function below the Branches.

We distribute the `fun Q` into the branches of the `match t`. Concretely, we change from:

Fixpoint `wpngen (E:ctx) (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v | trm_fun x t1 => Q (val_fun x (isubst (rem x E) t1)) ... end.`

to the equivalent form:

Fixpoint `wpngen (E:ctx) (t:trm) : (val->hprop)->hprop := match t with | trm_val v => fun (Q:val->hprop) => Q v | trm_fun x t1 => fun (Q:val->hprop) => Q (val_fun x (isubst (rem x E) t1)) ... end.`

The result type of `wpngen E t` is `(val→hprop)->hprop`. Hereafter, we let `formula` be a shorthand for this type. This shorthand will improve readability, especially when defining the `mkstruct` combinator, which has type `formula→formula`.

Definition `formula : Type := (val→hprop)->hprop.`

Readability Steps 2 and 3 for the Case of Sequences

We next introduce auxiliary definitions to denote the result of each of the branches of the `match t` construct. Concretely, we change from

Fixpoint `wpngen (E:ctx) (t:trm) : formula := match t with ... | trm_seq t1 t2 => fun Q => (wpngen E t1) (fun v => wpngen E t2 Q) ... end.`

to the equivalent form:

Fixpoint `wpngen (E:ctx) (t:trm) : formula := match t with ... | trm_seq t1 t2 => wpngen_seq (wpngen E t1) (wpngen E t2) ... end.`

where `wpngen_seq` is defined below.

Definition `wpngen_seq (F1 F2:formula) : formula := fun Q => F1 (fun v => F2 Q).`

Here, `F1` and `F2` denote the results of the recursive calls, `wpngen E t1` and `wpngen E t2`, respectively.

With the above definitions, `wpngen E (trm_seq t1 t2)` evaluates to `wp_seq (wpngen E t1) (wpngen E t2)`.

Finally, we introduce a piece of notation for each case. For sequence, we set up the notation defined next to so that any formula of the form `wpgen_seq F1 F2` gets displayed as *Seq F1 ; F2*.

```
Notation "'Seq' F1 ; F2" :=
  ((wpgen_seq F1 F2))
  (at level 68, right associativity,
   format "'[v' 'Seq' '[' F1 ']' ';' '/' '[' F2 ']' ']'").
```

Thanks to this notation, the `wpgen` of a sequence `t1 ; t2` displays as *Seq F1 ; F2* where *F1* and *F2* denote the `wpgen` of `t1` and `t2`, respectively. The “format” clause in the notation definition is there only to improve the pretty-printing of formulae. The semantics of `'/'` is to encourage a line break. The semantics of square brackets is to delimit blocks. The “v” character next to the leading bracket encourages a vertical alignment of the blocks. Double spaces in the format clause indicate the need to pretty-print a single space, as opposed to an empty spacing.

Readability Step 2: Auxiliary Definitions for other Constructs

We generalize the approach illustrated above for sequences to every other term construct...

```
Definition wpgen_val (v:val) : formula := fun Q =>
  Q v.
```

```
Definition wpgen_let (F1:formula) (F2of:val→formula) : formula := fun Q =>
  F1 (fun v => F2of v Q).
```

```
Definition wpgen_if (t:trm) (F1 F2:formula) : formula := fun Q =>
  \∃ (b:bool), \[t = trm_val (val_bool b)] \* (if b then F1 Q else F2 Q).
```

```
Definition wpgen_fail : formula := fun Q =>
  \[False].
```

```
Definition wpgen_var (E:ctx) (x:var) : formula :=
  match lookup x E with
  | None => wpgen_fail
  | Some v => wpgen_val v
  end.
```

```
Definition wpgen_app (t:trm) : formula := fun Q =>
  \∃ H, H \* \[triple t H Q].
```

The new definition of `wpgen` reads as follows:

```
Module WPGENEXEC2.
```

```
Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  match t with
  | trm_val v => wpgen_val v
  | trm_var x => wpgen_var E x
  | trm_fun x t1 => wpgen_val (val_fun x (isubst (rem x E) t1))
```

```

| trm_fix f x t1 ⇒ wpgen_val (val_fix f x (isubst (rem x (rem f E)) t1))
| trm_app t1 t2 ⇒ wpgen_app (isubst E t)
| trm_seq t1 t2 ⇒ wpgen_seq (wpgen E t1) (wpgen E t2)
| trm_let x t1 t2 ⇒ wpgen_let (wpgen E t1) (fun v ⇒ wpgen ((x,v)::E) t2)
| trm_if t0 t1 t2 ⇒ wpgen_if (isubst E t0) (wpgen E t1) (wpgen E t2)
end.

```

End WPGENEXEC2.

Readability Step 3: Notations for Auxiliary Definitions

We generalize the notation introduced for sequences to every other term construct. The corresponding notation is defined below. To avoid a conflict with the TLC notation for classical conditionals, we write *If_* in place of *If*.

Declare Scope *wpgen_scope*.

```

Notation "'Val' v" :=
  ((wpgen_val v))
  (at level 69) : wpgen_scope.

```

```

Notation "'Let' x ':= ' F1 'in' F2" :=
  ((wpgen_let F1 (fun x ⇒ F2)))
  (at level 69, x name, right associativity,
   format "'[v' '[' 'Let' x ':= ' F1 'in' ']' '/' '[' F2 ']' ']'")
  : wpgen_scope.

```

```

Notation "'If_ ' b 'Then' F1 'Else' F2" :=
  ((wpgen_if b F1 F2))
  (at level 69) : wpgen_scope.

```

```

Notation "'Fail'" :=
  ((wpgen_fail))
  (at level 69) : wpgen_scope.

```

In addition, we introduce a handy notation for the result of `wpgen t` where `t` denotes an application. (We cover here only functions of arity one or two. The file *LibSepReference.v* provides arity-generic definitions.)

```

Notation "'App' f v1 " :=
  ((wpgen_app (trm_app f v1)))
  (at level 68, f, v1 at level 0) : wpgen_scope.

```

```

Notation "'App' f v1 v2 " :=
  ((wpgen_app (trm_app (trm_app f v1) v2)))
  (at level 68, f, v1, v2 at level 0) : wpgen_scope.

```

Demo of `wpgen` with Notation.

Module WPGENWITHNOTATION.

```
Import EXAMPLEPROGRAMS WPGENEXEC2.
Open Scope wpgen_scope.
```

Here again, we temporarily axiomatize the soundness result for the purpose of looking at how that result can be exploited in practice.

```
Parameter triple_app_fun_from_wpgen :  $\forall v1\ v2\ x\ t1\ H\ Q,$   

   $v1 = \text{val\_fun } x\ t1 \rightarrow$   

   $H \Rightarrow \text{wpgen } ((x, v2) :: \text{nil})\ t1\ Q \rightarrow$   

   $\text{triple } (\text{trm\_app } v1\ v2)\ H\ Q.$ 
```

Consider again the example of *incr*.

```
Lemma triple_incr :  $\forall (p:\text{loc})\ (n:\text{int}),$   

   $\text{triple } (\text{trm\_app } \text{incr } p)$   

   $(p \sim\sim> n)$   

   $(\text{fun } _ \Rightarrow p \sim\sim> (n+1)).$ 
```

Proof using.

```
  intros. applies triple_app_fun_from_wpgen. { reflexivity. }  

  simpl. Abort.
```

Up to whitespace, alpha-renaming, removal of parentheses, and removal of quotes after *Let* and *If*, the formula *F* reads as:

Let $n := \text{App val\_get } p$ in Let $m := \text{App } (\text{val\_add } n)\ 1$ in $\text{App } (\text{val\_set } p)\ m$

With the introduction of intermediate definitions for *wpgen* and the introduction of associated notations for each term construct, what we have achieved is that the output of *wpgen* is, for any input term *t*, a human-readable formula whose display closely resembles the source code of *t*.

```
End WPGENWITHNOTATION.
```

45.3.4 Extension of *wpgen* to Handle Structural Rules

The definition of *wpgen* proposed so far integrates the logic of the reasoning rules for terms. The lemmas *wpgen_conseq* and *wpgen_frame* established earlier establish that *wpgen* moreover supports the structural rules of the logic. Yet, when we are in the middle of the verification of a concrete program *t*, the expression *wpgen nil t* is already simplified, making it hard to exploit the lemmas *wpgen_conseq* and *wpgen_frame*.

In order to support convenient application of these rules during the verification of concrete programs, we tweak the definition of *wpgen* in such a way that, even after simplification, it obviously satisfies both the rule of consequence and the frame rule.

Introduction of *mkstruct* in the Definition of *wpgen*

The tweak consists of introducing, at every step of the recursion of *wpgen*, a special predicate called *mkstruct* to capture the possibility of applying structural rules.

Fixpoint `wpgen (E:ctx) (t:trm) : (val->hprop)->hprop := mkstruct ( match t with |`

`trm_val v => wpgen_val v ... end`).
 With this definition, any output of `wpgen E t` is necessarily of the form `mkstruct F`, for some formula `F` describing the weakest precondition of `t`.

The next step is to investigate what properties `mkstruct` should satisfy.

Properties of `mkstruct`

Module `MKSTRUCTPROP`.

Because `mkstruct` appears between the prototype and the `match` in the body of `wpgen`, it must have type `formula→formula`.

Parameter `mkstruct` : `formula→formula`.

Next, `mkstruct` should satisfy reasoning rules that mimic the statements of the frame rule and the consequence rule in weakest-precondition style (`wp_frame` and `wp_conseq`).

Parameter `mkstruct_frame` : $\forall (F:\text{formula}) H Q,$
 $(\text{mkstruct } F \ Q) \setminus * H \implies \text{mkstruct } F (Q \setminus ** H).$

Parameter `mkstruct_conseq` : $\forall (F:\text{formula}) Q1 \ Q2,$
 $Q1 \implies Q2 \rightarrow$
 $\text{mkstruct } F \ Q1 \implies \text{mkstruct } F \ Q2.$

In addition, the user manipulating a formula produced by `wpgen` should be able to discard the predicate `mkstruct` from the head of the output when there is no need to apply the frame rule or the rule of consequence. The erasure property is captured by the following entailment.

Parameter `mkstruct_erase` : $\forall (F:\text{formula}) Q,$
 $F \ Q \implies \text{mkstruct } F \ Q.$

Moreover, in order to complete the soundness proof for `wpgen`, the predicate `mkstruct` should be covariant: if `F1` entails `F2`, then `mkstruct F1` should entail `mkstruct F2`.

Parameter `mkstruct_monotone` : $\forall F1 \ F2 \ Q,$
 $(\forall Q, F1 \ Q \implies F2 \ Q) \rightarrow$
 $\text{mkstruct } F1 \ Q \implies \text{mkstruct } F2 \ Q.$

End `MKSTRUCTPROP`.

Realization of `mkstruct`

The great news is that there exists a predicate satisfying the four required properties of `mkstruct`. The definition may appear magic at first, and indeed there is no need to deeply understand it or the proofs that follow. The only critical observation is that there exists a predicate `mkstruct` satisfying the required properties.

Definition `mkstruct (F:formula) : formula :=`

```
fun Q ⇒ \∃ Q1, F Q1 \* (Q1 \-* Q).
```

The magic wand may appear somewhat puzzling, but the above statement is reminiscent from the statement of `wp_ramified`, which captures the expressive power of all the structural reasoning rules of Separation Logic at once. If we unfold the definition of the magic wand, we can see more clearly that `mkstruct F` is a formula that describes a program that produces a postcondition `Q` when executed in the current state if and only if the formula `F` describes a program that produces a postcondition `Q1` in a subset of the current state and if the postcondition `Q1` augmented with the remainder of the current state (i.e., the piece described by `H`) corresponds to the postcondition `Q`.

```
Definition mkstruct' (F:formula) : formula :=
  fun (Q:val→hprop) ⇒ \∃ Q1 H, F Q1 \* H \* \[Q1 \*+ H ==> Q].
```

The four properties of `mkstruct` can be easily verified.

```
Lemma mkstruct_frame : ∀ F H Q,
  (mkstruct F Q) \* H ==> mkstruct F (Q \*+ H).
Proof using. intros. unfold mkstruct. xpull. intros Q'. xsimpl×. Qed.
```

```
Lemma mkstruct_conseq : ∀ F Q1 Q2,
  Q1 ==> Q2 →
  mkstruct F Q1 ==> mkstruct F Q2.
Proof using. introv WQ. unfold mkstruct. xpull. intros Q'. xsimpl×. Qed.
```

```
Lemma mkstruct_erase : ∀ F Q,
  F Q ==> mkstruct F Q.
Proof using. intros. unfold mkstruct. xsimpl. Qed.
```

```
Lemma mkstruct_monotone : ∀ F1 F2 Q,
  (∀ Q, F1 Q ==> F2 Q) →
  mkstruct F1 Q ==> mkstruct F2 Q.
Proof using. introv WF. unfolds mkstruct. xpull. intros Q'. xchanges WF. Qed.
```

Definition of `wpgen` with `mkstruct`

The final definition of `wpgen` refines the previous one by inserting the `mkstruct` predicate at the front of the `match t with` construct.

```
Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  mkstruct (match t with
  | trm_val v ⇒ wpgen_val v
  | trm_var x ⇒ wpgen_var E x
  | trm_fun x t1 ⇒ wpgen_val (val_fun x (isubst (rem x E) t1))
  | trm_fix f x t1 ⇒ wpgen_val (val_fix f x (isubst (rem x (rem f E)) t1))
  | trm_app t1 t2 ⇒ wpgen_app (isubst E t)
  | trm_seq t1 t2 ⇒ wpgen_seq (wpgen E t1) (wpgen E t2)
  | trm_let x t1 t2 ⇒ wpgen_let (wpgen E t1) (fun v ⇒ wpgen ((x,v)::E) t2)
```

```
| trm_if t0 t1 t2 => wpgen_if (isubst E t0) (wpgen E t1) (wpgen E t2)
end).
```

To avoid clutter in reading the output of `wpgen`, we introduce a lightweight shorthand to denote `mkstruct F`, allowing it to display simply as `F`.

Notation `"F" := (mkstruct F) (at level 10, format "F") : wpgen_scope.`

Once again, we temporarily axiomatize the soundness result for the purpose of looking at how that result can be exploited in practice.

Parameter `triple_app_fun_from_wpgen : ∀ v1 v2 x t1 H Q,`
`v1 = val_fun x t1 →`
`H ==> wpgen ((x, v2) :: nil) t1 Q →`
`triple (trm_app v1 v2) H Q.`

45.3.5 Implementation of X-tactics

X-lemmas for Implementing X-tactics

The last major step in the setup of our verification framework consists of lemmas and tactics to assist in the processing of formulas produced by `wpgen`. For each term construct, and for `mkstruct`, we introduce a dedicated lemma, called “x...-lemma”, to help with the elimination of that construct.

`xstruct_lemma` is a reformulation of `mkstruct_erase`.

Lemma `xstruct_lemma : ∀ F H Q,`
`H ==> F Q →`
`H ==> mkstruct F Q.`

Proof using. `intros M. xchange M. applies mkstruct_erase. Qed.`

`xval_lemma` reformulates the definition of `wpgen_val`. It just unfolds the definition.

Lemma `xval_lemma : ∀ v H Q,`
`H ==> Q v →`
`H ==> wpgen_val v Q.`

Proof using. `intros M. applies M. Qed.`

`xlet_lemma` reformulates the definition of `wpgen_let`. It just unfolds the definition.

Lemma `xlet_lemma : ∀ H F1 F2of Q,`
`H ==> F1 (fun v => F2of v Q) →`
`H ==> wpgen_let F1 F2of Q.`

Proof using. `intros M. xchange M. Qed.`

Likewise, `xseq_lemma` reformulates `wpgen_seq`.

Lemma `xseq_lemma : ∀ H F1 F2 Q,`
`H ==> F1 (fun v => F2 Q) →`
`H ==> wpgen_seq F1 F2 Q.`

Proof using. *intros M. xchange M. Qed.*

`xapp_lemma` applies to goals produced by `wpgen` on an application. In such cases, the proof obligation is of the form $H \implies (\text{wpgen\_app } t) \ Q$. `xapp_lemma` reformulates the frame-consequence rule in a way that enables exploiting a specification triple for a function to reason about a call to that function.

```
Lemma xapp_lemma : ∀ t Q1 H1 H2 H Q,
  triple t H1 Q1 →
  H ==> H1 \* H2 →
  Q1 \*+ H2 ==> Q →
  H ==> wpgen_app t Q.
```

Proof using.

```
intros M WH WQ. unfold wpgen_app. xsimpl. applys× triple_conseq_frame M.
Qed.
```

`xwp_lemma` is another name for `triple_app_fun_from_wpgen`. It is used for establishing a triple for a function application.

```
Lemma xwp_lemma : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wpgen ((x, v2) : : nil) t1 Q →
  triple (trm_app v1 v2) H Q.
```

Proof using. *intros M1 M2. applys× triple_app_fun_from_wpgen. Qed.*

Example Proofs using X-lemmas

Let us illustrate how “x-lemmas” help clarifying verification proof scripts.

```
Module PROOFWITHXLEMMAS.
```

```
Import EXAMPLEPROGRAMS.
```

```
Open Scope wpgen_scope.
```

```
Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
    (p ~~> n)
    (fun _ => p ~~> (n+1)).
```

Proof using.

```
intros.
  applys xwp_lemma. { reflexivity. }

  simpl.
  applys xstruct_lemma.
  applys xlet_lemma.
  applys xstruct_lemma.
  applys xapp_lemma. { apply triple_get. } { xsimpl. }
  xpull; intros ? →.
```

```

applies xstruct_lemma.
applies xlet_lemma.
applies xstruct_lemma.
applies xapp_lemma. { apply triple_add. } { xsimpl. }
xpull; intros ?  $\rightarrow$ .
applies xstruct_lemma.
applies xapp_lemma. { apply triple_set. } { xsimpl. }
xsimpl.

```

Qed.

Exercise: 2 stars, standard, especially useful (triple_succ_using_incr_with_xlemmas)
 Using x-lemmas, simplify the proof of `triple_succ_using_incr`, which was carried out using triples in chapter [Rules](#).

```

Lemma triple_succ_using_incr_with_xlemmas :  $\forall$  (n:int),
  triple (trm_app succ_using_incr n)
    \ []
    (fun v  $\Rightarrow$  \ [v = n+1]).

```

Proof using. *Admitted*.

□

End PROOFSWITHXLEMMAS.

Definition of X-tactics

Next, for each x-lemma, we introduce a dedicated tactic to apply that lemma and perform the associated bookkeeping. *xstruct* eliminates the leading `mkstruct` that appears in a goal of the form $H \Rightarrow \text{mkstruct } F \ Q$.

```

Tactic Notation "xstruct" :=
  applies xstruct_lemma.

```

The tactics *xval*, *xseq* and *xlet* invoke the corresponding x-lemmas, after calling *xstruct* if a leading `mkstruct` is in the way.

```

Tactic Notation "xstruct_if_needed" :=
  try match goal with  $\vdash$  ?H  $\Rightarrow$  mkstruct ?F ?Q  $\Rightarrow$  xstruct end.

```

```

Tactic Notation "xval" :=
  xstruct_if_needed; applies xval_lemma.

```

```

Tactic Notation "xlet" :=
  xstruct_if_needed; applies xlet_lemma.

```

```

Tactic Notation "xseq" :=
  xstruct_if_needed; applies xseq_lemma.

```


xapp_nosubst applies *xapp_lemma*, after calling *xseq* or *xlet* if applicable. Further on, we will define *xapp* as an enhanced version of *xapp_nosubst* that is able to automatically perform substitutions.

```
Tactic Notation "xseq_xlet_if_needed" :=
  try match goal with ⊢ ?H ==> mkstruct ?F ?Q =>
  match F with
  | wpgen_seq ?F1 ?F2 => xseq
  | wpgen_let ?F1 ?F2of => xlet
  end end.
```

```
Tactic Notation "xapp_nosubst" constr(E) :=
  xseq_xlet_if_needed; xstruct_if_needed;
  applies xapp_lemma E; [ xsimpl | xpull ].
```

xwp applies *xwp_lemma*, then requests Rocq to evaluate the *wpgen* function. (For technical reasons, in the definition shown below we need to explicitly request the unfolding of *wpgen_var*.)

```
Tactic Notation "xwp" :=
  intros; applies xwp_lemma;
  [ first [ reflexivity | eassumption ]
  | simpl; unfold wpgen_var; simpl ].
```

Now let us revisit the previous proof scripts using x-tactics instead of x-lemmas. The reader is invited to contemplate the gain in conciseness.

```
Module PROOFSWITHXTACTICS.
```

```
Import EXAMPLEPROGRAMS.
```

```
Open Scope wpgen_scope.
```

```
Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
    (p ~~> n)
    (fun _ => p ~~> (n+1)).
```

```
Proof using.
```

```
  xwp.
```

```
  xapp_nosubst triple_get. intros ? →.
```

```
  xapp_nosubst triple_add. intros ? →.
```

```
  xapp_nosubst triple_set.
```

```
  xsimpl.
```

```
Qed.
```

Exercise: 2 stars, standard, especially useful (triple_succ_using_incr_with_xtactics)

Using x-tactics, prove *succ_using_incr*.

```
Lemma triple_succ_using_incr_with_xtactics : ∀ (n:int),
  triple (trm_app succ_using_incr n)
```

```

\[]
(fun v => \[v = n+1]).
Proof using. Admitted.

```

□

End PROOFSWITHXTACTICS.

Improved Postprocessing for *xapp*.

The above proofs frequently exhibit the patterns `intros ? →` or `intros ? x →` for some variable `x`. This pattern typically occurs whenever the specification of the function being called in the code features a postcondition of the form `fun v => \[v = ..]` or of the form `fun v => \[v = ..] \* ...`. Let us devise a tactic called *xapp* to automatically handle this pattern.

The pattern in its implementation matches on the shape of the goal being produced by a call to *xapp_nosubst*. It attempts to perform the introduction and the substitution. The reader need not follow through the details of this Ltac definition.

```

Tactic Notation "xapp_try_subst" :=
  try match goal with
  | ⊢ ∀ (r:val), (r = _) → _ => intros ? →
  | ⊢ ∀ (r:val), ∀ x, (r = _) → _ =>
    let y := fresh x in intros ? y ->; revert y
  end.

```

```

Tactic Notation "xapp" constr(E) :=
  xapp_nosubst E; xapp_try_subst.

```

Database of Specification Lemmas for the *xapp* Tactic.

Explicitly providing arguments to *xapp_nosubst* or *xapp* is tedious. To avoid this effort, we set up a database of specification lemmas. Using this database, *xapp* can automatically look up the relevant specification. The actual CFML tool uses a proper lookup table binding function names to specification lemmas. In this course, however, we'll rely on a simpler mechanism, based on standard hints for *eauto*. In this setting, we can register specification lemmas using the `Hint Resolve ... : triple` command, as illustrated below.

```

#[global] Hint Resolve triple_get triple_set triple_ref
               triple_free triple_add : triple.

```

The argument-free variants *xapp_subst* and *xapp* are implemented by invoking *eauto* with `triple` to retrieve the relevant specification.

One shortcoming of the version of the *xapp* tactic that leverages the `triple` database is that it is not able to automatically apply specifications that involve a premise that *eauto* cannot solve. To exploit such specifications, we need to provide the specification explicitly using *xapp E*. This tactic is defined in *LibSepReference.v*. Its implementation is a bit technical, we do not describe it here.

```

Tactic Notation "xapp_nosubst" :=
  xseq_xlet_if_needed; xstruct_if_needed;
  applies xapp_lemma; [ solve [ eauto with triple ] | xsimpl | xpull ].

Tactic Notation "xapp" :=
  xapp_nosubst; xapp_try_subst.

```

Demo of Verification using X-tactics

```

Module PROOFWITHADVANCEDXTACTICS.
Import EXAMPLEPROGRAMS.
Open Scope wpgen_scope.

```

The tactics defined in the previous sections may be used to complete proofs like done in the first two chapters of the course. For example, here is the proof of the increment function.

```

Lemma triple_incr : ∀ (p:loc) (n:int),
  triple (trm_app incr p)
    (p ~~> n)
    (fun _ => p ~~> (n+1)).

```

Proof using.

```
xwp. xapp. xapp. xapp. xsimpl.
```

Qed.

```
#[global] Hint Resolve triple_incr : triple.
```

In summary, we have shown how to leverage `wpgen`, x-lemmas and x-tactics to implement the core of a practical verification framework based on Separation Logic.

```
End PROOFWITHADVANCEDXTACTICS.
```

45.3.6 Reasoning about Local Functions

```

Module EXAMPLELOCALFUNWPGEN.
Import DEMOPROGRAMS WPGENEXEC2.
Open Scope wpgen_scope.

```

Consider the following program, which defines a local function named `f` for incrementing a reference cell `p`, then invokes `f()` twice. The purpose of this example is to illustrate how to reason about a local function.

OCaml:

```
fun p -> let f = (fun () -> incr p) in f(); f()
```

Definition `incrtwice` : **val** :=

```

<{ fun 'p =>
  let 'f = (fun _ 'u => incr 'p) in
  'f ();
  'f () }>.

```

There are two possible approaches to reasoning about this code.

In the first approach, we store the hypothesis capturing that `f` has for code `fun () → incr p`, and each time we encounter a call to `f`, we “inline” the code of `f` at the call site. This approach is very effective when `f` has a unique occurrence. However, if `f` has multiple occurrences, then the user has to reason several times about the same code pattern.

The second approach yields a modular proof. When reasoning about the definition of `f`, we assert and prove a specification for `f`, then immediately discard any information revealing what the code of `f` is. Then, each time we encounter a call to `f`, we use `xapp` to leverage the specification of the local function `f`, exactly like we have been doing everywhere for reasoning about calls to top-level functions.

Let us illustrate the two approaches, and introduce on the way a few additional x-tactics relevant for local functions.

1. Reasoning about Local Functions by Inlining

To begin with, let us naively attempt to use the tactics `xlet` and `xval` to reason about the function definitions.

Lemma `triple_incrtwice_using_xval` : $\forall (p:\text{loc}) (n:\text{int}),$
 $\text{triple } (\text{incrtwice } p)$
 $(p \sim\sim> n)$
 $(\text{fun } _ \Rightarrow p \sim\sim> (n+2)).$

Proof using.

`xwp`.

At this point, we have the formula for the body of the function `incrtwice`, which binds (under the name `v`) the value `fun {"u"} ⇒ {incr p}`. `xlet`.

By typing `xlet`, we focus on this value `fun {"u"} ⇒ {incr p}`. The tactic `xval` then substitutes this value throughout the remaining of the formula.

`xval`.

Because the body of `incrtwice` contains two calls to the local function `f`, the resulting formula contains two copies of the function `fun {"u"} ⇒ {incr p}`. This kind of duplication, in general, is problematic, because the size of the formulae grow significantly. `Abort`.

We therefore introduce a variant of the `xlet` tactic, called `xletval`, that processes let-bindings by introducing an equality in the Rocq context. For example, for our local function `f`, a fresh hypothesis named `f` of type `val` is introduced, and an hypothesis of type asserting that `f` is equal to `fun {"u"} ⇒ {incr p}` is provided to characterize the value `f`. Let us first defined the tactic `xval`, then illustrate its usage.

The tactic `xletval` applies to formulae arising from terms of the form `let x = v1 in t2`. It processes the binding `let x = v1` by introducing in the goal a universal quantification over a variable `x` and over an hypothesis of type `x = v1`.

Lemma `xletval_lemma` : $\forall H v1 F2of Q,$
 $(\forall x, x = v1 \rightarrow H \implies F2of x Q) \rightarrow$
 $H \implies \text{wpgen\_let } (\text{mkstruct } (\text{wpgen\_val } v1)) F2of Q.$

Proof using.

```
introv M. applies xlet_lemma. apply xstruct_lemma.
apply xval_lemma. applies × M.
```

Qed.

Tactic Notation "xletval" :=

```
xstruct_if_needed; applies xletval_lemma.
```

Lemma triple_incrtwice_using_xletval : $\forall (p:\text{loc}) (n:\text{int})$,

```
triple (incrtwice p)
  (p ~~> n)
  (fun _ => p ~~> (n+2)).
```

Proof using.

```
xwp. xletval. intros f Hf.
```

As announced, reasoning about the local function definition using *xletval* introduces a name *f* and an hypothesis *Hf* of type $f = \langle \{ \text{fun } \{ "u" \} \Rightarrow \{ \text{incr } p \} \} \rangle$

There remains to reason about calls to the function *f*. Let us first show how it could be done step by step. We will subsequently define a tactic, named *xappfun*, to improve conciseness. To reason about the remaining of the code of *incrtwice*, we begin by focusing on the first call to *f()*, using *xseq*. *xseq*.

At this stage, we can remove the leading *mkstruct*, unfold the definition of *wpgen_app*, and simplify the resulting statement using *xsimpl*. *xstruct*. *unfold wpgen_app*. *xsimpl*.

We are left with proving that *f()* admits a certain behaviour, under the hypothesis that the code of *f* is *fun { "u" } => { incr p }*. We can make progress by invoking the tactic *xwp*, exactly as we do for reasoning about top-level functions. *xwp*.

There remains to reason about the code from the body of the function *f*, that is, the instruction *incr p*. We can use *xapp* for that purpose. *xapp*.

We could similarly handle the second call to *f*, then complete the proof. Let us show how to do so in the next lemma, after introducing a tactic to factorize the proof pattern for reasoning about calls to local functions. *Abort*.

The tactic *xappfun* applies to formulae arising from function calls of the form *f v* where *f* is a local function previously handled using *xletval*. Essentially, this tactic inline the formula associated with the body of *f*, specializing the body to the argument *v*.

Lemma xappfun_lemma : $\forall t H Q$,

```
triple t H Q →
H ==> wpgen_app t Q.
```

Proof using. *introv M*. *unfold wpgen_app*. *xsimpl* ×. *Qed*.

Tactic Notation "xappfun" :=

```
xseq_xlet_if_needed; xstruct_if_needed; applies xappfun_lemma; xwp.
```

Let us revisit the previous proof using this new tactic *xappfun*.

Lemma triple_incrtwice_using_xletva_and_xapp_fun : $\forall (p:\text{loc}) (n:\text{int})$,

```
triple (incrtwice p)
  (p ~~> n)
```

(fun _ => p ~~> (n+2)).

Proof using.

xwp.

xletval. intros f Hf. xappfun. xapp. xappfun. xapp. xsimpl. math. Qed.

This completes the presentation of the first approach. As illustrated in the example proof above, reasoning twice about the body of the function `f` leads to some duplication. The second approach shows how to avoid duplication, by stating and proving a specification for `f`.

2. Reasoning about Local Functions Modularly

In the second approach, immediately after obtaining the hypothesis that `f` has code `fun {"u"} => {incr p}`, we establish a specification for `f`, expressed using a triple, then discard the hypothesis about the code of `f`.

Lemma triple_incrtwice_with_assert : $\forall (p:\text{loc}) (n:\text{int}),$
 triple (incrtwice p)
 (p ~~> n)
 (fun _ => p ~~> (n+2)).

Proof using.

We begin the proof like the previous one. *xwp. xletval. intros f Hf.*

As soon as `f` is introduced, we assert the specification of this function.

*assert (Sf : $\forall (m:\text{int}),$
 triple <{ f () }> (p ~~> m) (fun _ => p ~~> (m+1))). {*

To establish this specification, we inline the code of `f`. This is the one and only place where we inline the code of `f`. *intros. subst f.*

Then, we complete the verification of `f` as usual. *xwp. xapp. xsimpl. }*

We have introduced an hypothesis `Sf` that characterizes the behavior of `f` by means of a triple. We can drop the hypothesis about the code of `f`. *clear Hf.*

The remaining of the proof now proceeds just like if `f` was a top-level function, using *xapp* for reasoning about each call to `f`. *xapp.*

xapp.

xsimpl.

math.

Qed.

The tactic *xfun* applies to a formula arising from terms of the form `let f = (fun x → t1) in t2`. This tactic simulates the invocation of `assert` over the desired specification for `f`. The tactic *xfun* takes as argument the desired specification for the function. For example, in the proof of `incrtwice` we will use a call of the form:

xfun (fun f => forall (m:int), triple <{ f () }> (p ~~> m) (fun _ => p ~~> (m+1)))

Lemma xfun_lemma : $\forall (Sof:\text{val} \rightarrow \text{Prop}) H v1 F2of Q,$
Sof v1 →
 $(\forall x, Sof\ x \rightarrow H \implies F2of\ x\ Q) \rightarrow$

$H \Rightarrow \text{wpgen\_let } (\text{mkstruct } (\text{wpgen\_val } v1)) \text{ } F2of \text{ } Q.$
 Proof using. *introv Sf M. applies xletval_lemma. intros ? →. applies × M. Qed.*

Tactic Notation "xfun" constr(*Sof*) :=
 let *f* := fresh in let *Hf* := fresh in
xstruct_if_needed; applies (@xfun_lemma *Sof*).

We are ready, at last, to show an elegant proof script for *incrtwice*.

Lemma triple_incrtwice_with_xfun : $\forall (p:\text{loc}) (n:\text{int}),$
 triple (incrtwice *p*)
 (*p* $\sim\sim>$ *n*)
 (fun _ $\Rightarrow$ *p* $\sim\sim>$ (*n*+2)).

Proof using.

xwp.

The key step is to invoke *xfun* by providing the desired specification for the local function
 f. *xfun* (fun (*f:val*) $\Rightarrow \forall (m:\text{int}),$
 triple $\langle\{ f \ () \} \rangle (p \sim\sim> m) (\text{fun } _ \Rightarrow p \sim\sim> (m+1)))$).

The verification of the local function *f* is similar to that of any top-level function. {
xwp. xapp. xsimpl. }

There remains to assign a name to the function and its specification. *intros f Sf.*

The rest of the proof is the same as before. *xapp.*

xapp.

xsimpl.

math.

Qed.

In summary, we have presented the tactic *xfun* for reasoning about local functions using the exact same mechanisms as for reasoning about top-level functions. The only difference is that the specification is not provided by the user as a lemma, but as an argument to *xfun*. Using the tactic *xfun* leads to modular proofs, whereby the body of the local function is only processed once by the user.

3. Exercises on Local Functions

This section presents an exercise involving the verification of a program featuring a simple local functions, which consists of the 'successor' function.

OCaml:

fun n -> let f = (fun m -> m + 1) in let a = f 2 in f a

Definition succtwice : val :=

$\langle\{ \text{fun } 'n \Rightarrow$
 let *'f* = (fun _ *'m* $\Rightarrow$ *'m* + 1) in
 let *'a* = *'f* *'n* in
'f *'a* $\rangle$.

Exercise: 2 stars, standard, optional (triple_succtwice_using_xletval) Verify the function `succtwice` using `xletval` and `xappfun`, without using `xfun`.

```
Lemma triple_succtwice_using_xletval : ∀ (n:int),
  triple (succtwice n)
    \ []
    (fun r ⇒ \[r = n + 2]).
```

Proof using. *Admitted*.

□

Exercise: 2 stars, standard, optional (triple_succtwice_using_xfun) Verify the function `succtwice` using `xfun`.

```
Lemma triple_succtwice_using_xfun : ∀ (n:int),
  triple (succtwice n)
    \ []
    (fun r ⇒ \[r = n + 2]).
```

Proof using. *Admitted*.

□

A more challenging exercise, named `forloop`, may be found near the end of the chapter. It can be resolved using the tactic `xfun`, even without reading all the text from the 'optional material' section.

End EXAMPLELOCALFUNWPGEN.

45.3.7 Proof Structure and Proof Automation

When looking at a proof of the form `xwp. xapp. xapp. xapp. xsimpl.`, where the application of tactics is fully directed by the code to be verified, one might wonder why we do not take x-tactics one step further in terms of automation. Concretely, why not introduce a tactic for repeatedly applying the relevant x-tactics as far as possible?

Such a tactic – let's call it `xgo` – can indeed be defined, and can prove handy for proving short programs compactly. However, it often turns out to be counterproductive when verifying nontrivial programs. Let us explain why.

In a nontrivial proof, calls to x-tactics do not appear in sequence: they are interleaved with conventional Rocq tactics for discharging side conditions. Typically, when reasoning about a function whose specification contains preconditions, `xapp` generates one subgoal for each precondition. Now, suppose that we have a sequence made of 3 such function calls. The proof script would look like:

```
xapp. { prove_side_1_1. } { prove_side_1_2. } xapp. { prove_side_2_1. } { prove_side_2_2. }
{ prove_side_2_3. } xapp. { prove_side_3_1. } { prove_side_3_2. } { prove_side_3_3. }
```

If we use `xgo` instead of 3 calls to `xapp`, we replace 3 calls to `xapp` with a single call to `xgo`, which may seem at first sight like to be improvement. However, the proof script now looks like:

xgo. { prove_side_1_1. } { prove_side_1_2. } { prove_side_2_1. } { prove_side_2_2. } { prove_side_2_3. } { prove_side_3_1. } { prove_side_3_2. } { prove_side_3_3. }

With *xgo*, all the side-conditions are produced at once. Why is this a problem? Well suppose that either the code or the specifications of the functions being called are later modified. When we re-play the proof script with *xgo*, a number of proof obligations will be added, others will be removed. The user will have no obvious way of associating proof obligations with the function call they arise from. In summary, in nontrivial proofs, the use of *xgo* just to make the proof script a tiny bit more concise makes proof scripts much harder to maintain.

One way to make proof script more concise without harming maintainability is to use a tactic call *xstep*. A call to *xstep* *n* invokes *n* times in a row the relevant x-tactic, with only the last of these x-tactics being allowed to produce subgoals. With *xstep* *n*, we are able to factorize consecutive calls to certain x-tactics, yet we are forced to keep all the x-tactics that participate in structuring the proof scripts.

A useful generalization is the tactic *xstep* $\times$ *n*, which is like *xstep* *n* except that the x-tactics before the last one are allowed to produce subgoals as long as these subgoals are all solved by automation. The tactic *xgo* $\times$ can now be defined as *xstep* $\times$ *n* for the maximal possible value of *n*, with a check that all the code has been processed, i.e., that the point of a *xsimpl* has been reached. Examples of usage of *xstep* and *xgo* are beyond the scope of this course.

45.4 Optional Material

45.4.1 Additional Tactics *xconseq* and *xframe*

The tactic *xconseq* applies the weakening rule; from the perspective of the user, it replaces the current postcondition with a stronger one. Optionally, the tactic can be passed an explicit argument, using the syntax *xconseq* *Q*.

The tactic is implemented by applying the lemma *xconseq_lemma*, stated below.

Lemma *xconseq_lemma* : $\forall Q1\ Q2\ H\ F,$
 $H \implies \text{mkstruct } F\ Q1 \rightarrow$
 $Q1 \implies Q2 \rightarrow$
 $H \implies \text{mkstruct } F\ Q2.$

Exercise: 1 star, standard, optional (*xconseq_lemma*) Prove the *xconseq_lemma*.

Proof using. *Admitted*.

□

Tactic Notation "xconseq" :=
applies xconseq_lemma.

Tactic Notation "xconseq" constr(*Q*) :=

applies *xconseq_lemma* *Q*.

The tactic *xframe* enables applying the frame rule on a formula produced by *wpgen*. The syntax *xframe H* is used to specify the heap predicate to keep, and the syntax *xframe_out H* is used to specify the heap predicate to frame out—everything else is kept.

```
Lemma xframe_lemma : ∀ H1 H2 H Q Q1 F,
  H ==> H1 \* H2 →
  H1 ==> mkstruct F Q1 →
  Q1 \*+ H2 ==> Q →
  H ==> mkstruct F Q.
```

Exercise: 2 stars, standard, optional (xframe_lemma) Prove the *xframe_lemma*. Exploit the properties of *mkstruct*; do not try to unfold the definition of *mkstruct*.

Proof using. *Admitted*.

□

```
Tactic Notation "xframe" constr(H) :=
  eapply (@xframe_lemma H); [ xsimpl | | ].
Tactic Notation "xframe_out" constr(H) :=
  eapply (@xframe_lemma - H); [ xsimpl | | ].
```

Let's illustrate the use of *xframe* on an example.

```
Module PROOFWITHSTRUCTURALXTACTICS.
Import EXAMPLEPROGRAMS.
Open Scope wpgen_scope.
```

```
Lemma triple_incr_frame : ∀ (p q:loc) (n m:int),
  triple (trm_app incr p)
    (p ~~> n \* q ~~> m)
    (fun _ => (p ~~> (n+1)) \* (q ~~> m)).
```

Proof using.

xwp.

Instead of calling *xapp*, we put aside $q \sim\sim> m$ and focus on $p \sim\sim> n$. *xframe* ($p \sim\sim> n$). *xapp*. *xapp*. *xapp*.

Finally we check the check that the current state augmented with the framed predicate $q \sim\sim> m$ matches with the claimed postcondition. *xsimpl*.

Qed.

```
End PROOFWITHSTRUCTURALXTACTICS.
```

45.4.2 Evaluation of *wpgen* Recursively inside Local Functions

```
Module WPGENREC.
Implicit Types vx vf : val.
```

1. Motivation for `wpgen` to Recurse in Local Functions

So far in the chapter, in the definition of `wpgen`, we have treated the constructions `trm_fun` and `trm_fix` as terms directly constructing a value, following the same pattern as for the construction `trm_val`.

Fixpoint `wpgen (t:trm) (Q:val->hprop) : hprop := match t with | trm_val v => Q v | trm_fun x t1 => Q (val_fun x t1) | trm_fix f x t1 => Q (val_fix f x t1) ... end.`

This section explains how to extend the definition of `wpgen` to make it recurse inside the body of the function definitions. Doing so does not increase expressiveness, because the user had the possibility of manually requesting the computation of `wpgen` on a function value of the form `val_fun` or `val_fix`. However, having `wpgen` automatically recurse into function bodies simplifies and improves the efficiency of the implementation of the `xfun` tactic.

The new definition of `wpgen` takes the shape shown below, for well-suited definitions of `wpgen_fun` and `wpgen_fix` yet to be introduced. In the code snippet below, `vx` denotes a value to which the function may be applied, and `vf` denotes the value associated with the function itself. This value is involved where the function defined by `trm_fix f x t1` invokes itself recursively inside its body `t1`.

Fixpoint `wpgen (E:ctx) (t:trm) : formula := mkstruct match t with | trm_val v => wpgen_val v | trm_fun x t1 => wpgen_fun (fun vx => wpgen ((x,vx)::E) t1) | trm_fix f x t1 => wpgen_fix (fun vf vx => wpgen ((f,vf)::(x,vx)::E) t1) ... end.`

2. A New Definition of `wpgen` that Recurses in Local Functions

For simplicity, let us assume for now the substitution context `E` to be empty and ignore the presence of the predicate `mkstruct`. Our first task is to provide a definition for `wpgen (trm_fun x t1) Q`, expressed in terms of `wpgen t1`.

Let `vf` denote the function `val_fun x t1`, which the term `trm_fun x t1` evaluates to. The heap predicate `wpgen (trm_fun x t1) Q` should assert that that the postcondition `Q` holds of the result value `vf`, i.e., that `Q vf` should hold.

Rather than specifying that `vf` is equal to `val_fun x t1` as we were doing previously, we would like to specify that `vf` denotes a function whose body admits `wpgen t1` as weakest precondition. To that end, we define the heap predicate `wpgen (trm_fun x t1) Q` to be of the form $\forall vf, \backslash[P\ vf] \backslash^* Q\ vf$ for a carefully crafted predicate `P` that describes the behavior of the function by means of its weakest precondition. This predicate `P` is defined further on.

The universal quantification on `vf` is intended to hide away from the user the fact that `vf` actually denotes `val_fun x t1`. It would be correct to replace $\forall vf, \dots$ with `let vf := val_fun x t1 in ...`, yet we are purposely trying to abstract away from the syntax of `t1`, hence the universal quantification of `vf`.

In the heap predicate $\forall vf, \backslash[P\ vf] \backslash^* Q\ vf$, the magic wand features a left-hand side of the form $\backslash[P\ vf]$ intended to provide to the user the knowledge of the weakest precondition of the body `t1` of the function, in such a way that the user is able to derive the specification aimed for the local function `vf`.

Concretely, the proposition $P\ vf$ should enable the user establishing properties of applications of the form $\text{trm\_app}\ vf\ vx$, where vf denotes the function and vx denotes an argument to which the function is applied.

To figure out the details of the statement of P , it is useful to recall from chapter WPgen the statement of the lemma $\text{triple\_app\_fun\_from\_wpgen}$, which we used for reasoning about top-level functions.

```
Parameter triple_app_fun_from_wpgen :  $\forall\ vf\ vx\ x\ t1\ H'\ Q'$ ,
   $vf = \text{val\_fun}\ x\ t1 \rightarrow$ 
   $H' \Rightarrow \text{wpgen}\ ((x, vx) :: \text{nil})\ t1\ Q' \rightarrow$ 
   $\text{triple}\ (\text{trm\_app}\ vf\ vx)\ H'\ Q'$ .
```

The lemma above enables establishing a triple for an application $\text{trm\_app}\ vf\ vx$ with precondition H' and postcondition Q' , from the premise $H' \Rightarrow \text{wgen}\ ((x, vx) :: \text{nil})\ t1\ Q'$.

It therefore makes sense, in our definition of the predicate $\text{wpgen}\ (\text{trm\_fun}\ x\ t1)\ Q$, which we said would take the form $\backslash\forall\ vf, \backslash[P\ vf]\ \backslash-*\ Q\ vf$, to define $P\ vf$ as:

```
forall vx H' Q', (H' ==> wpgen ((x,vx)::nil) t1 Q') -> triple (trm_app vf vx) H' Q'
```

Overall, the definition of $\text{wpgen}\ E\ t$ is as follows. Note that the occurrence of nil is replaced with E to account for the case of a nonempty context.

```
Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct match t with ... | trm_fun x t1 =>
fun Q => let P vf := (forall vx H' Q', H' ==> wpgen ((x,vx)::E) t1 Q' -> triple (trm_app
vf vx) H' Q') in \forall vf, \P vf \-* Q vf ... end.
```

The actual definition of wpgen exploits an intermediate definition called wpgen_fun , as shown below:

```
Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct match t with ... | trm_fun x t1
=> wpgen_fun (fun vx => wpgen ((x,vx)::E) t1) ... end.
```

where wpgen_fun could be defined as follows:

```
Definition wpgen_fun' (Fof:val→formula) : formula := fun Q =>
  \forall vf, \[\forall vx H' Q',
     $H' \Rightarrow Fof\ vx\ Q' \rightarrow$ 
     $\text{triple}\ (\text{trm\_app}\ vf\ vx)\ H'\ Q']$ 
  \-* Q vf.
```

In the definition above, the assumption about vf , enclosed above in brackets, can be slightly simplified, by using wp instead of triple , allowing to eliminate H' . This observation leads to a more concise definition of wpgen_fun .

```
Definition wpgen_fun (Fof:val→formula) : formula := fun Q =>
  \forall vf, \[\forall vx Q', Fof vx Q' ==> wp (trm_app vf vx) Q'] \-* Q vf.
```

Exercise: 2 stars, standard, optional (wpgen_fun_eq) Prove that $\text{wpgen\_fun}'$ is equivalent to wpgen_fun .

Lemma wpgen_fun_eq :

```
wpgen_fun = wpgen_fun'.
```

Proof using.

```

unfold wpgen_fun, wpgen_fun'. applies fun_ext_2.
intros Fof Q. applies himpl_antisym.
{ applies himpl_hforall_r. intros vf. xchange (hforall_specialize vf).
  xsimpl. intros M. rewrite hwand_hpure_l. xsimpl.
  introv. rewrite wp_equiv. applies M. xsimpl. }
Admitted.

```

□

Like for other auxiliary functions associated with `wpgen`, we introduce a custom notation for `wpgen_fun`. Here, we let $\text{Fun } x := F$ stand for `wpgen_fun (fun x  $\Rightarrow$  F)`.

Notation `"'Fun' x ' := ' F1"` :=
`((wpgen_fun (fun x  $\Rightarrow$  F1)))`
`(at level 69, x name, right associativity,`
`format "'[v' '[' 'Fun' x ' := ' F1 ']' ']'")`.

3. Treatment of Recursive Functions

The formula produced by `wpgen` for a recursive function `trm_fix f x t1` is almost the same as for a non-recursive function. The main difference is that we need to add a binding in the context from the name `f` to the value `vf`. Like for `trm_fun`, the heap predicate `wpgen (trm_fix f x t1) Q` is of the form $\forall vf, \forall [P \text{ } vf] \setminus^* Q \text{ } vf$. The predicate P is similar to that for `trm_fun`, the only difference is that the context E needs to be extended not with one but with two bindings: one for the argument, and one for the function. Concretely, P is defined as:

forall vx H' Q', H' ==> wpgen ((f,vf)::(x,vx)::E) t1 Q' -> triple (trm_app vf vx) H' Q'

Like for `wpgen_fun`, we can make the definition more concise by eliminating H' . Concretely, we define:

Definition `wpgen_fix (Fof:val $\rightarrow$ val $\rightarrow$ formula) : formula := fun Q  $\Rightarrow$`
 `$\forall vf, \forall [\forall vx Q', Fof \text{ } vf \text{ } vx \text{ } Q' ==> wp \text{ } (trm\_app \text{ } vf \text{ } vx) \text{ } Q'] \setminus^* Q \text{ } vf$ .`

Then we integrate this definition in `wpgen` as shown below.

Fixpoint wpgen (E:ctx) (t:trm) : formula := mkstruct match t with | .. | trm_fix f x t1
 $\Rightarrow$ wpgen_fix (fun vf v $\Rightarrow$ wpgen ((f,vf)::(x,v)::E) t1) | .. end

Here again, we introduce a piece of notation for `wpgen_fix`. We let $\text{Fix } f \text{ } x := F$ stand for `wpgen_fix (fun f x  $\Rightarrow$  F)`.

Notation `"'Fix' f x ' := ' F1"` :=
`((wpgen_fix (fun f x  $\Rightarrow$  F1)))`
`(at level 69, f name, x name, right associativity,`
`format "'[v' '[' 'Fix' f x ' := ' F1 ']' ']'")`.

4. Final Definition of `wpgen`, with Processing a Local Functions

The final definition of `wpgen`, including the recursive processing of local function definitions, appears below. Note that we here aim at supporting only local functions with one argument.

The proper treatment of n-ary local functions would require a nested recursive function for accumulating the list of arguments involved in each local function. It is beyond the scope of this course.

```

Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  mkstruct match t with
  | trm_val v ⇒ wpgen_val v
  | trm_var x ⇒ wpgen_var E x
  | trm_fun x t1 ⇒ wpgen_fun (fun v ⇒ wpgen ((x,v)::E) t1)
  | trm_fix f x t1 ⇒ wpgen_fix (fun vf v ⇒ wpgen ((f,vf)::(x,v)::E) t1)
  | trm_app t1 t2 ⇒ wpgen_app (isubst E t)
  | trm_seq t1 t2 ⇒ wpgen_seq (wpgen E t1) (wpgen E t2)
  | trm_let x t1 t2 ⇒ wpgen_let (wpgen E t1) (fun v ⇒ wpgen ((x,v)::E) t2)
  | trm_if t0 t1 t2 ⇒ wpgen_if (isubst E t0) (wpgen E t1) (wpgen E t2)
  end.

```

5. Tactic for Reasoning About Functions

This section refines the tactic `fun`, to reflect on the fact that the new definition of `wpgen` removes the need for the user to invoke `xwp` by hand for reasoning about the body of local functions. The tactic `xfun` presented here may be invoked either with or without providing a specification for the local function. If no specification is provided, the formula associated with the local function is taken as specification. This formula corresponds to the “most general specification” of the function. Exploiting this formula has a similar effect as inlining the code of the local function.

First, we describe the tactic `xfun S`, where `S` describes the specification of the local function. A typical call is of the form `xfun (fun (f:val) ⇒ ∀ ..., triple (f ..) .. ..)`. The tactic `xfun S` generates two subgoals. The first one requires the user to establish the specification `S` for the function. The second one requires the user to prove that the rest of the program is correct, in a context where the local function can be assumed to satisfy the specification `S`. The definition of `xfun S` appears next. It is not required to understand the details. An example use case appears further on.

```

Lemma xfun_spec_lemma : ∀ (S:val→Prop) H Q Fof,
  (∀ vf,
    (∀ vx H' Q', (H' ==> Fof vx Q') → triple (trm_app vf vx) H' Q') →
    S vf) →
  (∀ vf, S vf → (H ==> Q vf)) →
  H ==> wpgen_fun Fof Q.

```

Proof using.

```

  introv M1 M2. unfold wpgen_fun. xsimpl. intros vf N.
  applies M2. applies M1. introv K. rewrite ← wp_equiv.
  applies himpl_trans_r K. applies N.

```

Qed.

```
Tactic Notation "xfun" constr(S) :=
  xseq_xlet_if_needed; xstruct_if_needed; applies xfun_spec_lemma S.
```

Second, we describe the tactic *xfun* with no argument. It applies to a goal of the form $H \implies \text{wpngen\_fun } Fof \ Q$. The tactic *xfun* simply provides an hypothesis about the local function. The user may subsequently exploit this hypothesis for reasoning about a call to that function, just like if the code of the function was inlined at its call site. In practice, the use of *xfun* without argument is most relevant for local functions that are invoked exactly once. For example, higher-order iterators such as *List.iter* or *List.map* are typically invoked with an ad-hoc functions that appears exactly once.

```
Lemma xfun_nospec_lemma :  $\forall H \ Q \ Fof,$ 
  ( $\forall vf,$ 
    ( $\forall vx \ H' \ Q', (H' \implies Fof \ vx \ Q') \rightarrow \text{triple } (\text{trm\_app } vf \ vx) \ H' \ Q') \rightarrow$ 
    ( $H \implies Q \ vf$ ))  $\rightarrow$ 
   $H \implies \text{wpngen\_fun } Fof \ Q.$ 
```

Proof using.

```
  introv M. unfold wpngen_fun. xsimpl. intros vf N. applies M.
  introv K. rewrite  $\leftarrow$  wp_equiv. applies himpl_trans_r K. applies N.
```

Qed.

```
Tactic Notation "xfun" :=
  xseq_xlet_if_needed; xstruct_if_needed; applies xfun_nospec_lemma.
```

A generalization of *xfun* that handles recursive functions may be defined following exactly the same pattern. Details may be found in the file *LibSepReference.v*

This completes our presentation of a version of *wpngen* that recursively processes the local definition of non-recursive functions. A practical example is presented next.

6. Example Revisited with the New *wpngen*

```
Module EXAMPLELOCALFUNWPGENREC.
Import DEMOPROGRAMS EXAMPLELOCALFUNWPGEN.
```

We import here the definitions from *LibSepReference*, which provides the a definition of *wpngen* that recurse in function bodies exactly as defined in subsection

```
Import LIBSEPREFERENCE.
Open Scope wp_scope.
```

Consider again the example program *myincr*, whose code is:

OCaml:

```
fun p -> let f = (fun () -> incr p) in f(); f()
```

We first illustrate a call to *xfun* with an explicit specification. Here, the function *f* is specified as incrementing the reference *p*. Observe that the body function of the function *f* is verified only once. The reasoning about the two calls to the function *f* that appears in the code are done with respect to the specification that we provide as argument to *xfun* in the proof script.

```

Lemma triple_incr twice :  $\forall (p:\text{loc}) (n:\text{int}),$ 
  triple (trm_app incrtwice p)
    ( $p \rightsquigarrow\!\!\sim n$ )
    ( $\text{fun } _ \Rightarrow p \rightsquigarrow\!\!\sim (n+2)$ ).

```

Proof using.

```

xwp.
xfun (fun (f:val)  $\Rightarrow \forall (m:\text{int}),$ 
  triple (f ())
    ( $p \rightsquigarrow\!\!\sim m$ )
    ( $\text{fun } _ \Rightarrow p \rightsquigarrow\!\!\sim (m+1)$ )); intros f Hf.
{ intros.
  applies Hf.      clear Hf.
  xapp. xsimpl. }
xapp.  xapp.  xapp.  xsimpl. math.

```

Qed.

We next illustrate a call to *xfun* without argument. The “generic specification” that comes as hypothesis about the local function is a proposition that describes the behavior of the function in terms of the weakest-precondition of its body. When reasoning about a call to the function, one can invoke this generic specification. The effect is equivalent to that of inlining the code of the function at its call site. The example shown below contains two calls to the function *f*. The script invokes *xfun* without argument. Then, each of the two calls to *f* executes a call to *incr*. Thus the proof script involves two calls to *xapp* that each exploit the specification *triple_incr*.

```

Lemma triple_incr twice' :  $\forall (p:\text{loc}) (n:\text{int}),$ 
  triple (trm_app incrtwice p)
    ( $p \rightsquigarrow\!\!\sim n$ )
    ( $\text{fun } _ \Rightarrow p \rightsquigarrow\!\!\sim (n+2)$ ).

```

Proof using.

```

xwp.
xfun; intros f Hf.
xapp.  xapp.  xapp.  xapp.  xsimpl. math.

```

Qed.

7. Exercise on a Local Recursive Function

Consider a program that uses a local function to encode a for-loop. This program generalizes the *repeat* program presented in the chapter *Repr*. The loop function takes as argument the loop bounds *a* and *b*, as well a function *f* describing the body of the loop. A locally defined, recursive function named *g* is used to execute *f i* for every integer value of *i* in the range from *a* inclusive to *b* exclusive.

OCaml:

```
fun a b f -> let g = (fix g i -> if i < b then (f i; g (i+1)) end) in g a
```



```

Definition forloop : val :=
  <{ fun 'a 'b 'f =>
    let 'g = (fix_ 'g 'i =>
      let 'c = 'i < 'b in
      if 'c then
        'f 'i ;
        let 'j = 'i + 1 in
        'g 'j
      end) in
    'g 'a }>.

```

Like for the function `repeat`, the specification of `forloop` is expressed using an invariant, named I , of type $\text{int} \rightarrow \text{hprop}$. The hypothesis on f asserts that a call to $f\ i$ takes a state satisfying $I\ i$ to a state satisfying $I\ (i+1)$. For simplicity, we specify the function only in the case where $a \leq b$.

Exercise: 4 stars, standard, especially useful (triple_forloop) Verify the function `forloop`. Hint: provide the specification of g to `xfun`. To set up the induction, use the sequence: `intros i. induction_wf IH: (upto b) i. intros Hi..` Besides, the tactics `math_rewrite` is helpful at one point in the proof.

```

Lemma triple_forloop :  $\forall (I:\text{int} \rightarrow \text{hprop}) (a\ b:\text{int}) (f:\text{val}),$ 
   $a \leq b \rightarrow$ 
   $(\forall i, a \leq i \leq b \rightarrow$ 
    triple  $(f\ i)$ 
     $(I\ i)$ 
     $(\text{fun } u \Rightarrow I\ (i+1))) \rightarrow$ 
  triple  $(\text{forloop } a\ b\ f)$ 
   $(I\ a)$ 
   $(\text{fun } r \Rightarrow I\ b).$ 

```

Proof using. *Admitted*.

□

End EXAMPLELOCALFUNWPGENREC.

End WPGENREC.

45.4.3 Historical Notes

Many verification tools for Hoare Logic are based on weakest-precondition generators. Typically, a tool takes as input source code annotated with specifications and invariants and produces a logical formula that entails the correctness of the program. This logical formula is typically expressed in first-order logic and is discharged using automated tools such as SMT solvers.

In contrast, the weakest-precondition generator presented in this chapter applies to unannotated code. It thus produces a logical formula that does not depend at all on the specifications and invariants. Such formula, which can be constructed in a systematic manner, is called a “characteristic formula” in the literature. The notion of characteristic formulae was introduced in work by *Hennessey and Milner* 1985 (in Bib.v) on process calculi. It was first applied to program logics by *Honda, Berger, and Yoshida* 2006 (in Bib.v) and to Separation Logic in the PhD work of *Charguéraud* 2010 (in Bib.v), which led to CFML.

In general, a characteristic formula provides not just a sound but also a complete description of the semantics of a program. In Charguéraud’s PhD thesis, completeness is established on paper, and a mechanized proof in Rocq is provided only for the simple IMP language. Even if a result of the form $\text{wp } t \ Q \implies \text{wpgen } t \ Q$ were proved in Rocq, such a completeness result would be relatively unsatisfying. Indeed, this entailment asserts that if a program admits a behavior, then the program logic can be used to establish that this program admits that behavior. Yet, this statement does not capture the existence of a *pretty* Separation Logic proof featuring local reasoning that is, allowing for maximal usage of the frame rule.

Chapter 46

Library SLF.WPsound

46.1 WPsound: Soundness of the Weakest Precondition Generator

```
Set Implicit Arguments.
From SLF Require Import WPsem WPgen.
Import WPGENREC.

Implicit Types f : var.
Implicit Types b : bool.
Implicit Types v : val.
Implicit Types h : heap.
Implicit Types P : Prop.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
```

46.2 More Details

This chapter develops the soundness proof for `wpgen`. The key theorem asserts that, for any term and and postcondition, `wpgen` (on an empty substitution context) yields a formula that entails `wp`. Formally: `wpgen nil t Q ==> wp t Q`. The present “more details” section contains the proof of this theorem. The “optional material” section details the proofs of two technical lemmas related to permutation of substitutions. This chapter may be safely skipped.

46.2.1 Definition of the Predicate `formula_sound`

The soundness theorem that we aim to establish for `wpgen` is:

Parameter `wpgen_sound` : forall E t Q, `wpgen E t Q ==> wp (isubst E t) Q`.

Before entering into the details of the proof, let us reformulate the statement of the soundness theorem to make proof obligations and induction hypotheses easier to read. To that end, we introduce the predicate `formula_sound t F`, read “F is a formula sound for t”. It asserts that `F` is a weakest-precondition style formula that entails `wp t`. Formally:

Definition `formula_sound (t:trm) (F:formula) : Prop :=`
 $\forall Q, F \ Q ==> \text{wp } t \ Q.$

Using `formula_sound`, the soundness theorem for `wpgen` is reformulated as follows.

Parameter `wpgen_sound : forall E t, formula_sound (isubst E t) (wpgen E t)`.

The soundness theorem `wpgen_sound` is proved by induction on `t`. Each case of this inductive proof corresponds to a term construct. For each term construct, we need to exploit two lemmas. The first lemma, common to all term constructs, is called `mkstruct_sound`, and is used to argue that the presence of `mkstruct` in the definition of `wpgen` is sound. The second lemma is specific to the term construct. For example, `wpgen_sound_seq` is of the form `formula_sound (trm_seq t1 t2) (wpgen_seq F1 F2)`. We next explain the details of how to state and prove `wpgen_sound_seq`, then present the other auxiliary lemmas useful for completing the proof by induction of `wpgen_sound`.

46.2.2 Soundness for the Case of Sequences

Consider the soundness theorem `wpgen_sound` and let us specialize it to the particular case where `t` is of the form `trm_seq t1 t2`. The corresponding statement is:

Parameter `wpgen_sound_seq_1 :  $\forall E \ t1 \ t2,$`
`formula_sound (isubst E (trm_seq t1 t2))`
`(wpgen E (trm_seq t1 t2)).`

Let us ignore `mkstruct` for a minute – that is, pretend that `wpgen` is defined without `mkstruct`. Under this assumption, `wpgen E (trm_seq t1 t2)` evaluates to `wpgen_seq (wpgen E t1) (wpgen E t2)`. Besides, the expression `isubst E (trm_seq t1 t2)` evaluates to `trm_seq (isubst E t1) (isubst E t2)`. Therefore, the soundness statement to establish can be reformulated as:

Parameter `wpgen_sound_seq_2 :  $\forall E \ t1 \ t2,$`
`formula_sound (trm_seq (isubst E t1) (isubst E t2))`
`(wpgen_seq (wpgen E t1) (wpgen E t2)).`

By induction hypothesis, we can assume that `wpgen E t1` is a sound formula for `isubst E t1`, and likewise `wpgen E t2` is a sound formula for `isubst E t2`. Thus, what we actually will need to prove as part of the induction is:

Parameter `wpgen_sound_seq_3 :  $\forall E \ t1 \ t2,$`
`formula_sound (isubst E t1) (wpgen E t1)  $\rightarrow$`
`formula_sound (isubst E t2) (wpgen E t2)  $\rightarrow$`
`formula_sound (trm_seq (isubst E t1) (isubst E t2))`
`(wpgen_seq (wpgen E t1) (wpgen E t2)).`

In the above statement, let us abstract `isubst E t1` as $t1'$ and `wpgen t1` as $F1$, and similarly `isubst E t2` as $t2'$ and `wpgen t2` as $F2$. The statement reformulates as:

```
Lemma wpgen_seq_sound : ∀ t1' t2' F1 F2,
  formula_sound t1' F1 →
  formula_sound t2' F2 →
  formula_sound (trm_seq t1' t2') (wpgen_seq F1 F2).
```

This statement captures the essence of the correctness of the definition of `wpgen_seq`. Now let's prove it.

```
Proof using.
  introv S1 S2.
  unfolds formula_sound.
  intros Q.
  unfolds wpgen_seq.
  applies himpl_trans.
  2:{ applies wp_seq. }

  { applies himpl_trans.
    { applies S1. }
    { applies wp_conseq. intros v. applies S2. } }
Qed.
```

46.2.3 Soundness for the Other Term Constructs

Similarly, for every other language construct, we state and prove a lemma about `formula_sound`. The reader may skip over the proof details. What is most interesting here is the pattern followed by the statements.

```
Lemma wpgen_val_sound : ∀ v,
  formula_sound (trm_val v) (wpgen_val v).
```

```
Proof using. intros. intros Q. unfolds wpgen_val. applies wp_val. Qed.
```

```
Lemma wpgen_let_sound : ∀ F1 F2of x t1 t2,
  formula_sound t1 F1 →
  (∀ v, formula_sound (subst x v t2) (F2of v)) →
  formula_sound (trm_let x t1 t2) (wpgen_let F1 F2of).
```

```
Proof using.
  introv S1 S2. intros Q. unfolds wpgen_let. applies himpl_trans_r wp_let.
  applies himpl_trans S1. applies wp_conseq. intros v. applies S2.
```

```
Qed.
```

```
Lemma wpgen_if_sound : ∀ F1 F2 t0 t1 t2,
  formula_sound t1 F1 →
  formula_sound t2 F2 →
  formula_sound (trm_if t0 t1 t2) (wpgen_if t0 F1 F2).
```

Proof using.

```
introv S1 S2. intros Q. unfold wpgen_if. xpull. intros b →.  
applies himpl_trans_r wp_if. case_if. { applies S1. } { applies S2. }
```

Qed.

Lemma wpgen_app_sound : $\forall t$,
formula_sound t (wpgen_app t).

Proof using.

```
intros t Q. unfold wpgen_app. xpull. intros H M. rewrite wp_equiv. apply M.
```

Qed.

Lemma wpgen_fail_sound : $\forall t$,
formula_sound t wpgen_fail.

Proof using. intros. intros Q. unfold wpgen_fail. xpull. Qed.

Lemma wpgen_fun_sound : $\forall x t1 Fof$,
($\forall vx$, formula_sound (subst $x vx t1$) ($Fof vx$)) $\rightarrow$
formula_sound (trm_fun $x t1$) (wpgen_fun Fof).

Proof using.

```
introv M. intros Q. unfolds wpgen_fun. applies himpl_hforall_l (val_fun x t1).  
xchange hwand_hpure_l.  
{ intros. applies himpl_trans_r. { applies  $\times$  wp_app_fun. } { applies  $\times$  M. } }  
{ applies wp_fun. }
```

Qed.

Lemma wpgen_fix_sound : $\forall f x t1 Fof$,
($\forall vf vx$, formula_sound (subst $x vx$ (subst $f vf t1$)) ($Fof vf vx$)) $\rightarrow$
formula_sound (trm_fix $f x t1$) (wpgen_fix Fof).

Proof using.

```
introv M. intros Q. unfolds wpgen_fix.  
applies himpl_hforall_l (val_fix f x t1). xchange hwand_hpure_l.  
{ intros. applies himpl_trans_r. { applies  $\times$  wp_app_fix. } { applies  $\times$  M. } }  
{ applies wp_fix. }
```

Qed.

Additional lemmas establishing the soundness of a version of `wpgen` that does not recurse inside local function definitions.

Lemma wpgen_fun_val_sound : $\forall x t$,
formula_sound (trm_fun $x t$) (wpgen_val (val_fun $x t$)).

Proof using. intros. intros Q. unfolds wpgen_val. applies wp_fun. Qed.

Lemma wpgen_fix_val_sound : $\forall f x t$,
formula_sound (trm_fix $f x t$) (wpgen_val (val_fix $f x t$)).

Proof using. intros. intros Q. unfolds wpgen_val. applies wp_fix. Qed.

46.2.4 Soundness of the Predicate Transformer `mkstruct`

Above, we have ignored the existence of `mkstruct` at the head of formulae produced by `wpgen`. We next show that the insertion of `mkstruct` in formulae preserves their soundness. In other words, if the formula F is a valid weakest precondition for t , then so is `mkstruct` F .

Lemma `mkstruct_sound` : forall t F , `formula_sound` t F $\rightarrow$ `formula_sound` t (`mkstruct` F).

In order to prove this result, we first need to show that `mkstruct` is a predicate transformer that does not affect the meaning of any formula of the form `wp` t . Intuitively, `mkstruct` adds support for applying structural rules of Separation Logic; but a formula `wp` t *inherently* satisfies all the structural rules; hence wrapping `wp` t inside a call to `mkstruct` does not increase (nor decrease) its expressive power.

Lemma `mkstruct_wp` : $\forall t$,
`mkstruct` (`wp` t) = (`wp` t).

Proof using.

```
intros. applies fun_ext_1. intros  $Q$ . applies himpl_antisym.
{ unfold mkstruct. xpull. intros  $H$   $Q'$   $M$ . applies wp_conseq_frame  $M$ . xsimpl. }
{ applies mkstruct_erase. }
```

Qed.

We are now ready to prove the desired result `mkstruct_sound`.

Lemma `mkstruct_sound` : $\forall t$ F ,
`formula_sound` t F $\rightarrow$
`formula_sound` t (`mkstruct` F).

Proof using.

```
introv  $M$ . unfolds formula_sound. intros  $Q$ .
rewrite  $\leftarrow$  mkstruct_wp. applies mkstruct_monotone. applies  $M$ .
```

Qed.

Another, lower-level proof for the same result reveals the definition of `mkstruct` and exploits the consequence-frame rule for `wp`.

Lemma `mkstruct_sound'` : $\forall t$ F ,
`formula_sound` t F $\rightarrow$
`formula_sound` t (`mkstruct` F).

Proof using.

```
introv  $M$ . unfolds formula_sound.
intros  $Q$ .
unfolds mkstruct.
xsimpl. intros  $Q'$ .
xchange ( $M$   $Q'$ ).
applies wp_ramified.
```

Qed.

Another interesting property of `mkstruct` is its idempotence property, that is, it is such

that `mkstruct (mkstruct  $F$ ) = mkstruct  $F$` . Idempotence asserts that applying the predicate `mkstruct` more than once does not provide more expressiveness than applying it just once.

Intuitively, idempotence reflects in particular the fact that two nested applications of the rule of consequence can always be combined into a single application of that rule (due to the transitivity of entailment); and that, similarly, two nested applications of the frame rule can always be combined into a single application of that rule (framing on $H1$ then framing on $H2$ is equivalent to framing on $H1 \setminus^* H2$).

Exercise: 3 stars, standard, especially useful (`mkstruct_idempotent`) Complete the proof of the idempotence of `mkstruct`. Hint: leverage `xpull` and `xsimpl`.

Lemma `mkstruct_idempotent` : $\forall F$,
`mkstruct (mkstruct  $F$ ) = mkstruct  $F$ .`

Proof using.

`intros. apply fun_ext_1. intros  $Q$ . applies himpl_antisym.`
Admitted.

□

46.2.5 Lemmas Capturing Properties of Iterated Substitutions

In the soundness proof for `wpgen`, we need to exploit two basic properties of the iterated substitution function `isubst`. The first property asserts that the substitution for the empty context is the identity. This result is useful to beautify the final statement of the soundness theorem.

Parameter `isubst_nil` : $\forall t$,
`isubst nil  $t$  =  $t$ .`

The second property asserts that the substitution for a context $(x,v)::E$ is the same as the substitution for the context `rem  $x$   $E$` followed with the substitution of `subst  $x$   $v$` . This second property is needed to handle the case of let-bindings.

Parameter `isubst_rem` : $\forall x\ v\ E\ t$,
`isubst (( $x, v$ ) ::  $E$ )  $t$  = subst  $x$   $v$  (isubst (rem  $x$   $E$ )  $t$ ).`

The proofs for these two lemmas is fairly technical and of limited interest for the course. They can be found in the “optional material” near the end of this chapter.

46.2.6 Main Induction for the Soundness Proof of `wpgen`

At last, we are ready to establish the soundness of `wpgen  $E$   $t$` . As previously announced, the proof is by structural induction on t . For each term construct, the proof has two steps:

- first, invoke the lemma `mkstruct_sound` to justify soundness of the leading `mkstruct` produced by `wpgen`,

- second, invoke the the soundness lemma specific to that term construct, e.g. `wp-gen_seq_sound` for sequences.

Lemma `wpgen_sound_induct` : $\forall E t$,
`formula_sound (isubst  $E$   $t$ ) (wpgen  $E$   $t$ ).`

Proof using.

```

intros. gen E. induction t; intros; simpl;
  applies mkstruct_sound. { applies wpgen_val_sound. }
{ rename v into x. unfold wpgen_var. case_eq (lookup x E).
  { intros v EQ. applies wpgen_val_sound. }
  { intros N. applies wpgen_fail_sound. } }
{ applies wpgen_fun_sound.
  { intros vx. rewrite ← isubst_rem. applies IHt. } }
{ introv IH. applies wpgen_fix_sound IH.
  { intros vf vx. rewrite ← isubst_rem_2. applies IHt. } }
{ applies wpgen_app_sound. }
{ applies wpgen_seq_sound. { applies IHt1. } { applies IHt2. } }
{ rename v into x. applies wpgen_let_sound.
  { applies IHt1. }
  { intros v. rewrite ← isubst_rem. applies IHt2. } }
{ applies wpgen_if_sound. { applies IHt2. } { applies IHt3. } }

```

Qed.

46.2.7 Statement of Soundness of `wpgen` for Closed Terms

For a closed term t , the context E is instantiated as `nil`, and `isubst nil t` simplifies to t . We derive the main soundness statement for `wpgen`.

Lemma `wpgen_sound` : $\forall t Q$,
`wpgen nil  $t$   $Q$  ==> wp  $t$   $Q$ .`

Proof using.

```

introv M. lets N: (wpgen_sound nil t). rewrite isubst_nil in N. applies × N.

```

Qed.

The lemma `triple_of_wpgen` triggers the initial evaluation of `wpgen` for a given program t that appears in a triple.

Lemma `triple_of_wpgen` : $\forall t H Q$,
 $H ==> \text{wpgen nil } t \text{ } Q \rightarrow$
 $\text{triple } t \text{ } H \text{ } Q$.

Proof using.

```

introv M. rewrite ← wp_equiv. xchange M. applies wpgen_sound.

```

Qed.

The lemma `triple_app_fun_from_wpgen` is at the core of the implementation of the tactic `xwp`. It is a specialized version of `triple_of_wpgen` specialized for establishing a triple for a

function application. In essence, this lemma is a reformulation of the rule *triple_app_fun*, but triggering an evaluation of *wpgen* for the body of the function.

```
Lemma triple_app_fun_from_wpgen : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wpgen ((x, v2)::nil) t1 Q →
  triple (trm_app v1 v2) H Q.
```

Proof using.

```
introv M1 M2. applys triple_app_fun M1.
asserts_rewrite (subst x v2 t1 = isubst ((x, v2)::nil) t1).
{ rewrite isubst_rem. rewrite× isubst_nil. }
rewrite ← wp_equiv. xchange M2. applys wpgen_sound_induct.
```

Qed.

The lemma *triple_app_fix_from_wpgen* is similar but handles recursive functions.

```
Lemma triple_app_fix_from_wpgen : ∀ v1 v2 f x t1 H Q,
  v1 = val_fix f x t1 →
  H ==> wpgen ((f, v1)::(x, v2)::nil) t1 Q →
  triple (trm_app v1 v2) H Q.
```

Proof using.

```
introv M1 M2. applys triple_app_fix M1.
asserts_rewrite (subst x v2 (subst f v1 t1)
  = isubst ((f, v1)::(x, v2)::nil) t1).
{ rewrite isubst_rem_2. rewrite× isubst_nil. }
rewrite ← wp_equiv. xchange M2. applys wpgen_sound_induct.
```

Qed.

46.3 Optional Material

46.3.1 Proofs of Properties of Iterated Substitution

Module ISUBSTPROP.

Open Scope *liblist_scope*.

Implicit Types *E* : ctx.

Recall that *isubst E t* denotes the multi-substitution in the term *t* of all bindings from the association list *E*. The soundness proof for *wpgen* and the proof of its corollary *triple_app_fun_from_wpgen* rely on two key properties of iterated substitutions, captured by the lemmas called *isubst_nil* and *isubst_rem*, which we state and prove next.

isubst nil t = t

subst x v (isubst (rem x E) t) = isubst ((x,v)::E) t

The first lemma is straightforward by induction. The TLC tactic *fequals* is an enhanced variant of Rocq's tactic *f_equal*.

Lemma `isubst_nil` : $\forall t$,
`isubst nil t = t`.

Proof using.

`intros t. induction t; simpl; fequals.`

Qed.

The second lemma is much more involved to prove. We introduce the notion of “two equivalent contexts” $E1$ and $E2$, and argue that substitution for two equivalent contexts yields the same result. We then introduce the notion of “contexts with disjoint domains”, and argue that if $E1$ and $E2$ are disjoint then `isubst ( $E1 ++ E2$ ) t = isubst  $E1$  (isubst  $E2$  t)`.

Before we start, we describe the tactic `case_var`, which helps with the case-analyses on variable equalities, and we prove an auxiliary lemma that describes the result of a lookup on a context from which a binding has been removed. It is defined in the file `LibSepVar.v` as:

Tactic Notation “`case_var`” := repeat rewrite `var_eq_spec` in *; repeat `case_if`.

The tactic `case_var`× is a shorthand for `case_var` followed with automation (essentially, `eauto`).

A key auxiliary lemma relates `subst` and `isubst`.

Lemma `subst_eq_isubst_one` : $\forall x v t$,
`subst x v t = isubst ((x,v)::nil) t`.

Proof using.

```
intros. induction t; simpl.
{ fequals. }
{ case_var×. }
{ fequals. case_var×. { rewrite× isubst_nil. } }
{ fequals. case_var; try case_var; simpl; try case_var;
  try rewrite isubst_nil; auto. }
{ fequals×. }
{ fequals×. }
{ fequals×. case_var×. { rewrite× isubst_nil. } }
{ fequals×. }
```

Qed.

A lemma about the lookup in a removal.

Lemma `lookup_rem` : $\forall x y E$,
`lookup x (rem y E) = if var_eq x y then None else lookup x E`.

Proof using.

```
intros. induction E as [(z,v) E'].
{ simpl. case_var×. }
{ simpl. case_var*; simpl; case_var×. }
```

Qed.

A lemma about the removal over an append.

Lemma `rem_app` : $\forall x E1 E2$,

$\text{rem } x (E1 ++ E2) = \text{rem } x E1 ++ \text{rem } x E2.$

Proof using.

```
intros. induction E1 as [(y,w) E1']; rew_list; simpl. { auto. }
{ case_var ×. { rew_list. fequals. } }
```

Qed.

The definition of equivalent contexts.

Definition $\text{ctx\_equiv } E1 E2 :=$

$\forall x, \text{lookup } x E1 = \text{lookup } x E2.$

The fact that removal preserves equivalence.

Lemma $\text{ctx\_equiv\_rem} : \forall x E1 E2,$
 $\text{ctx\_equiv } E1 E2 \rightarrow$
 $\text{ctx\_equiv } (\text{rem } x E1) (\text{rem } x E2).$

Proof using.

```
intros M. unfolds ctx_equiv. intros y.
do 2 rewrite lookup_rem. case_var ×.
```

Qed.

The fact that substitution for equivalent contexts yields equal results.

Lemma $\text{isubst\_ctx\_equiv} : \forall t E1 E2,$
 $\text{ctx\_equiv } E1 E2 \rightarrow$
 $\text{isubst } E1 t = \text{isubst } E2 t.$

Proof using.

```
hint ctx_equiv_rem.
intros t. induction t; introv EQ; simpl; fequals ×.
{ rewrite EQ. auto. }
```

Qed.

The definition of disjoint contexts.

Definition $\text{ctx\_disjoint } E1 E2 :=$

$\forall x v1 v2, \text{lookup } x E1 = \text{Some } v1 \rightarrow \text{lookup } x E2 = \text{Some } v2 \rightarrow \text{False}.$

Removal preserves disjointness.

Lemma $\text{ctx\_disjoint\_rem} : \forall x E1 E2,$
 $\text{ctx\_disjoint } E1 E2 \rightarrow$
 $\text{ctx\_disjoint } (\text{rem } x E1) (\text{rem } x E2).$

Proof using.

```
intros D. intros y v1 v2 K1 K2. rewrite lookup_rem in *.
case_var. applies × D K1 K2.
```

Qed.

Lookup in the concatenation of two disjoint contexts $E1$ and $E2$ is equivalent to a lookup in $E1$ if it succeeds, or a lookup in $E2$ otherwise.

Lemma $\text{lookup\_app} : \forall x E1 E2,$

```

lookup x (E1 ++ E2) = match lookup x E1 with
  | None => lookup x E2
  | Some v => Some v
end.

```

Proof using.

```

introv. induction E1 as [(y,w) E1']; rew_list; simpl; intros.
{ auto. }
{ case_var ×. }

```

Qed.

The key induction shows that `isubst` distributes over context concatenation.

Lemma `isubst_app` : $\forall t \ E1 \ E2,$
`isubst (E1 ++ E2) t = isubst E2 (isubst E1 t).`

Proof using.

```

hint ctx_disjoint_rem.
intros t. induction t; simpl; intros.
{ fequals. }
{ rename v into x. rewrite × lookup_app.
  case_eq (lookup x E1); introv K1; case_eq (lookup x E2); introv K2; auto.
  { simpl. rewrite × K2. }
  { simpl. rewrite × K2. } }
{ fequals. rewrite × rem_app. }
{ fequals. do 2 rewrite × rem_app. }
{ fequals ×. }
{ fequals ×. }
{ fequals ×. { rewrite × rem_app. } }
{ fequals ×. }

```

Qed.

The next lemma asserts that the concatenation order is irrelevant in a substitution if the contexts have disjoint domains.

Lemma `isubst_app_swap` : $\forall t \ E1 \ E2,$
`ctx_disjoint E1 E2 →`
`isubst (E1 ++ E2) t = isubst (E2 ++ E1) t.`

Proof using.

```

introv D. applies isubst_ctx_equiv. applies × ctx_disjoint_equiv_app.

```

Qed.

At last, we derive the desired property of `isubst`.

Lemma `isubst_rem` : $\forall x \ v \ E \ t,$
`isubst ((x, v) :: E) t = subst x v (isubst (rem x E) t).`

Proof using.

```

intros. rewrite subst_eq_isubst_one. rewrite ← isubst_app.
rewrite isubst_app_swap.

```

```

{ applies isubst_ctx_equiv. intros y. rew_list. simpl. case_var ×.
  { rewrite lookup_rem. case_var ×. } }
{ intros y v1 v2 K1 K2. simpls. case_var.
  { subst. rewrite lookup_rem in K1. case_var ×. } }

```

Qed.

We also prove the variant lemma which is useful for handling recursive functions in the soundness proof for `wpgen`.

Lemma `isubst_rem_2` : $\forall f\ x\ vf\ vx\ E\ t,$
 $\text{isubst } ((f, vf) :: (x, vx) :: E) t$
 $= \text{subst } x\ vx\ (\text{subst } f\ vf\ (\text{isubst } (\text{rem } x\ (\text{rem } f\ E))\ t)).$

Proof using.

```

intros. do 2 rewrite subst_eq_isubst_one. do 2 rewrite ← isubst_app.
rewrite isubst_app_swap.
{ applies isubst_ctx_equiv. intros y. rew_list. simpl.
  do 2 rewrite lookup_rem. case_var ×. }
{ intros y v1 v2 K1 K2. rew_listx in *. simpls.
  do 2 rewrite lookup_rem in K1. case_var. }

```

Qed.

End ISUBSTPROP.

Chapter 47

Library `SLF.Affine`

47.1 Affine: Affine Separation Logic

```
Set Implicit Arguments.
From SLF Require Rules.
From SLF Require Export LibSepReference.

Implicit Types f : var.
Implicit Types b : bool.
Implicit Types v : val.
Implicit Types h : heap.
Implicit Types P : Prop.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
```

47.2 First Pass

The Separation Logic framework that we have constructed is well suited for a language with explicit deallocation, but cannot be used as-is for a language equipped with a garbage collector. As pointed out in chapter `Basic`, there is no rule in our basic Separation Logic that allows discarding a heap predicate from the precondition or the postcondition. From a theoretical perspective, discarding predicates from the postcondition is sufficient. From a practical perspective, however, the user may wish to clarify the proof obligations by discarding predicates appearing in the precondition as soon as they are no longer useful.

In this chapter, we explain how to generalize the Separation Logic framework to support a “discard rule”, which one may invoke to discard heap predicates from the precondition or postcondition.

The framework extended with the discard rule corresponds to an “affine” logic, where heap predicates may be freely discarded, as opposed to a “linear” logic, where heap predicates cannot be thrown away. As we argue, it may be interesting for a program logic to accomodate

both “affine” predicates and “linear” predicates. We will explain how to fine tune which heap predicates may be freely discarded.

For example, even in a programming language featuring a garbage collector, it may be useful to ensure that every file handle opened eventually gets closed, or that every lock acquired is eventually released. File handles and locks are example of resources that may be described in Separation Logic, yet that one should not be allowed to discard freely.

Regarding garbage-collected data, another possible approach is to require the programmer to integrate “free” operations defined as no-ops, to materialize the places where memory cells can and should be discarded. This approach, however, is cumbersome for freeing recursive data structures. For example, to free a mutable list, the programmer would need to iterate over the list to free its cells one by one. In contrast, with a “discard rule” in the program logic, the user can discard at once the representation predicate for the full list, without involving any form of iteration.

The chapter is organized as follows:

- first, we recall the example illustrating the limitation of a logic without a discard rule for a garbage-collected language;
- second, we present several versions of the “discard rule”;
- third, we show how to refine the definition of Separation Logic triples in such a way that the discard rules are satisfied for certain classes of heap predicates.

47.2.1 Motivation for the Discard Rule

Module `MOTIVATINGEXAMPLE`.

Export `DEMOPROGRAMS`.

Recall the example of the function `succ_using_incr_attempt` from chapter `Basic`. This function allocates a reference with contents `n`, then increments that reference, and finally returning its contents, that is, `n+1`. Let us revisit this example, this time with the intention of establishing for it a postcondition that does not leak the existence of a left-over reference cell.

Definition `succ_using_incr` :=

```
<{ fun 'n =>
  let 'p = ref 'n in
  incr 'p;
  ! 'p }>.
```

In the framework developed so far, the heap predicate describing the reference cell allocated by the function cannot be discarded, because the code does not include a deallocation operation. Thus, we are forced to include in the postcondition the description of a left-over reference with a heap predicate, e.g., $\backslash \exists p, p \sim \sim > (n+1)$, or $\backslash \exists p \ m, p \sim \sim > m$.


```

Lemma triple_succ_using_incr :  $\forall (n:\text{int}),$ 
  triple (succ_using_incr  $n$ )
     $\backslash []$ 
    (fun  $r \Rightarrow \backslash [r = n+1] \backslash * \backslash \exists p, (p \sim\sim > (n+1))$ ).

```

Proof using.

```

 $xwp.xapp.intros p.xapp.xapp.xsimpl.auto.$ 

```

Qed.

If we try to prove a specification that does not mention the left-over reference, then we get stuck with a proof obligation of the form $p \sim\sim > (n+1) ==> \backslash []$.

```

Lemma triple_succ_using_incr' :  $\forall (n:\text{int}),$ 
  triple (succ_using_incr  $n$ )
     $\backslash []$ 
    (fun  $r \Rightarrow \backslash [r = n+1]$ ).

```

Proof using.

```

 $xwp.xapp.intros p.xapp.xapp.xsimpl. \{ auto. \}$ 

```

Abort.

In what follows, we extend our Separation Logic with a “discard rule” that allows proving the above specification for `succ_using_incr`.

End MOTIVATINGEXAMPLE.

47.2.2 Statement of The Discard Rule

There are several ways to state the “discard rule”. Let us begin with two variants: one that discards a heap predicate H' from the postcondition, and one that discards a heap predicate H' from the precondition.

The first rule, named `triple_hany_post`, asserts that an arbitrary heap predicate H' can be dropped from the postcondition, simplifying it from $Q \backslash * + H'$ to Q .

```

Parameter triple_hany_post :  $\forall t H H' Q,$ 
  triple  $t H (Q \backslash * + H') \rightarrow$ 
  triple  $t H Q$ .

```

Let us show that, using the rule `triple_hany_post`, we can derive the desired specification for the motivating example from the specification that mentions the left-over postcondition.

Module MOTIVATINGEXAMPLESOLVED.

Export MOTIVATINGEXAMPLE.

```

Lemma triple_succ_using_incr' :  $\forall (n:\text{int}),$ 
  triple (succ_using_incr  $n$ )
     $\backslash []$ 
    (fun  $r \Rightarrow \backslash [r = n+1]$ ).

```

Proof using.

```

intros.  $applies triple_hany_post.applies triple_succ_using_incr.$ 

```

Qed.

End MOTIVATINGEXAMPLESOLVED.

A symmetric rule, named *triple_hany_pre*, asserts that an arbitrary heap predicate H' can be dropped from the precondition, simplifying it from $H \setminus^* H'$ to H .

Parameter *triple_hany_pre* : $\forall t H H' Q$,

triple $t H Q \rightarrow$

triple $t (H \setminus^* H') Q$.

Observe the difference between the two rules. In *triple_hany_post*, the discarded predicate H' appears in the premise, reflecting the fact that we discard it after the evaluation of the term t .

Conversely, in *triple_hany_pre*, the discarded predicate H' appears in the conclusion, reflecting the fact that we discard it before the evaluation of t .

The rules *triple_hany_pre* and *triple_hany_post* can be derived from each other. As we will establish further on, *triple_hany_pre* is derivable from *triple_hany_post* by a simple application of the frame rule. Reciprocally, *triple_hany_post* is derivable from *triple_hany_pre*, though the proof is slightly more involved. This proof appears in lemma *triple_hgc_post_from_hgc_pre* in the “optional material” section.

47.2.3 Fine-grained Control of Collectable Predicates

Axiomatization of *heap_affine* and *haffine*

As suggested in the introduction, it may be useful to constrain the discard rule in such a way that it can be used to discard only certain types of heap predicates, not arbitrary ones.

The idea is to restrict the discard rules so that only predicates satisfying a predicate called *haffine* may be discarded.

The two discard rules should thus feature an extra premise: to discard a heap predicate H' , the proposition *haffine* H' must hold.

Module PREVIEW.

Parameter *haffine* : $\text{hprop} \rightarrow \text{Prop}$.

Parameter *triple_haffine_post* : $\forall t H H' Q$,

haffine $H' \rightarrow$

triple $t H (Q \setminus^{*+} H') \rightarrow$

triple $t H Q$.

Parameter *triple_haffine_pre* : $\forall t H H' Q$,

haffine $H' \rightarrow$

triple $t H Q \rightarrow$

triple $t (H \setminus^* H') Q$.

End PREVIEW.

The predicate `haffine` H is defined in terms of a lower-level predicate `heap_affine`. The proposition `heap_affine` h asserts that the piece of heap h may be freely discarded, in the sense that it does not have to be accounted for in the program logic. The question of whether a heap or a heap predicate should be affine or not is to be decided on a case-by-case basis. It depends on the intention of the designer of the program logic. For the moment, we leave the predicate `heap_affine` abstract, and allow it to be customized later on.

Parameter `heap_affine` : `heap` $\rightarrow$ `Prop`.

The only assumptions we need to make about `heap_affine` is that it holds of the empty heap and that it is preserved by a disjoint union operation.

Parameter `heap_affine_empty` :
`heap_affine` `Fmap.empty`.

Parameter `heap_affine_union` : $\forall h1\ h2,$
`heap_affine` $h1 \rightarrow$
`heap_affine` $h2 \rightarrow$
`Fmap.disjoint` $h1\ h2 \rightarrow$
`heap_affine` (`Fmap.union` $h1\ h2$).

We will later show two extreme instantiations: one that leads to a logic where all predicates are affine (i.e. can be freely discarded), and one that leads to a logic where all predicates are linear (i.e. none can be freely discarded, like in our previous set up).

Definition and Properties of `heap_affine`

The predicate `haffine` H captures the notion of “affine heap predicate”.

A heap predicate is affine iff it holds only of affine heaps.

Definition `haffine` (H :`hprop`) : `Prop` :=
 $\forall h, H\ h \rightarrow \text{heap\_affine}\ h.$

The predicate `haffine` distributes in a natural way over each of the operators of Separation Logic – i.e., the combination of affine heap predicates yields affine heap predicates. In particular:

- $\backslash []$ and $\backslash [P]$, which describe empty heaps, can always be discarded;
- $H1 \backslash * H2$ can be discarded if both $H1$ and $H2$ can be discarded;
- $\backslash \forall x, H$ can be discarded if H can be discarded for at least one x ;
- $\backslash \exists x, H$ can be discarded if H can be discarded for every x .

The treatment of universal quantifiers can be explained as follows. Consider a heap h that satisfies $\backslash \forall (x:A), H$, and consider a particular value v of type A . The heap h also satisfies $[x:=v]H$. Thus, if the heap predicate $[x:=v]H$ can be discarded, the stronger predicate $\backslash \forall (x:A), H$ can also be discarded.

Existential quantifiers can be explained as follows. Consider a heap h that satisfies $\exists (x:A), H$. We know that there exists a value v of type A such that h satisfies $[x:=v]H$. However, we know nothing about this value v . Therefore, to discard $\exists (x:A), H$, we need to know that $[x:=v]H$ can be discarded for every possible v .

Lemma `haffine_hempty` :

`haffine \ [].`

Proof using.

`introv K. lets E: hempty_inv K. subst. applies heap_affine_empty.`

Qed.

Lemma `haffine_hpure` : $\forall P$,

`haffine \ [P].`

Proof using.

`intros. intros h K. lets (HP&M): hpure_inv K.`

`subst. applies heap_affine_empty.`

Qed.

Lemma `haffine_hstar` : $\forall H1\ H2$,

`haffine H1  $\rightarrow$`

`haffine H2  $\rightarrow$`

`haffine (H1  $\setminus^*$  H2).`

Proof using.

`introv M1 M2 K. lets (h1&h2&K1&K2&D&U): hstar_inv K.`

`subst. applies  $\times$  heap_affine_union.`

Qed.

Lemma `haffine_hexists` : $\forall A\ (J:A \rightarrow \text{hprop})$,

`( $\forall x$ , haffine (J x))  $\rightarrow$`

`haffine ( $\exists x$ , (J x)).`

Proof using. `introv F1 (x&Hx). applies  $\times$  F1. Qed.`

Lemma `haffine_hforall'` : $\forall A\ (J:A \rightarrow \text{hprop})$,

`( $\exists x$ , haffine (J x))  $\rightarrow$`

`haffine ( $\forall x$ , (J x)).`

Proof using.

`introv (x&Hx) M. lets N: hforall_inv M. applies  $\times$  Hx.`

Qed.

The rule `haffine_hforall'` requires the user to provide evidence that there exists at least one value x of type A for which `haffine (J x)` is true. In practice, the user is generally not interested in proving properties of a specific value x , but is happy to justify that $J\ x$ is affine for any x . The corresponding statement appears below, with an assumption `Inhab A` asserting that the type A is inhabited. In practice, the $\forall$ quantifier is always invoked on inhabited types, so this is a benign restriction.

Lemma `haffine_hforall` : $\forall A\ \{Inhab\ A\}\ (J:A \rightarrow \text{hprop})$,

`( $\forall x$ , haffine (J x))  $\rightarrow$`

```

  haffine ( $\forall x, (J\ x)$ ).
Proof using.
  introv IA M. applys haffine_hforall'.  $\exists$  (arbitrary (A:=A)). applys M.
Qed.

```

Another useful fact about `haffine` is that to prove `haffine ( $\backslash[P] \backslash^* H$ )` it is sufficient to prove `haffine H` under the hypothesis that `P` is true. Formally:

```

Lemma haffine_hstar_hpure_l :  $\forall P\ H,$ 
  ( $P \rightarrow$  haffine  $H$ )  $\rightarrow$ 
  haffine ( $\backslash[P] \backslash^* H$ ).
Proof using. introv M. intros h K. rewrite hstar_hpure_l in K. applys  $\times$  M. Qed.

```

47.2.4 The Affine-Top Heap Predicate: $\backslash GC$

Definition of The Affine-Top Predicate

We next introduce a new heap predicate, called “affine top”, that is very handy for describing “the possibility to discard a heap predicate”. We use this predicate to reformulate the discard rules in a more concise and usable manner. This predicate is written $\backslash GC$ and named `hgc` in the formalization. The predicate $\backslash GC$ holds of any affine heap.

The reason $\backslash GC$ is named “affine top” is because it refines a representation predicate called “top”, and written $\backslash Top$. The predicate $\backslash Top$ is the heap predicate that holds of any heap. It can be defined as `fun (h:heap)  $\Rightarrow$  True`, or, equivalently, as $\backslash \exists (H:hprop), H$. The characteristic property of $\backslash Top$ is the entailment $\forall H, (H \Rightarrow \backslash Top)$.

The affine top predicate $\backslash GC$ can be defined as $\exists H, \backslash[haffine\ H] \backslash^* H$. As we prove further on, it could be equivalently defined as `fun h  $\Rightarrow$  heap_affine h`. We prefer the former definition as it is easier to manipulate using the `xsimpl` tactic.

```

Definition hgc : hprop :=
   $\backslash \exists H, \backslash[haffine\ H] \backslash^* H$ .

Declare Scope hgc_scope.
Open Scope hgc_scope.

Notation " $\backslash GC$ " := (hgc) : hgc_scope.

```

The following introduction lemma asserts that $\backslash GC\ h$ holds when `h` satisfies `heap_affine`.

Exercise: 2 stars, standard, especially useful (triple_frame) Prove that the affine heap predicate holds of any affine heap.

```

Lemma hgc_intro :  $\forall h,$ 
  heap_affine h  $\rightarrow$ 
   $\backslash GC\ h$ .
Proof using. Admitted.

```

□

The elimination lemma asserts the converse.

Exercise: 2 stars, standard, optional (hgc_inv) Prove the elimination lemma for $\backslash GC$ expressed using *heap_affine*.

Lemma *hgc_inv* : $\forall h,$

$\backslash GC\ h \rightarrow$
heap_affine *h*.

Proof using. *Admitted*.

□

Together, the introduction and the elimination rule justify the fact that *hgc* could equivalently have been defined as *heap_affine*.

Lemma *hgc_eq_heap_affine* :

hgc = *heap_affine*.

Proof using.

intros. *applies* *himpl_antisym*.
 $\{ \text{intros } h\ M. \text{ applies } \times \text{ hgc\_inv. } \}$
 $\{ \text{intros } h\ M. \text{ applies } \times \text{ hgc\_intro. } \}$

Qed.

Properties of the Affine-Top Predicate

One fundamental property of $\backslash GC$ that is useful in the soundness proofs is that the conjunction of two occurrences of $\backslash GC$ may be merged into a single $\backslash GC$.

Lemma *hstar_hgc_hgc* :

$(\backslash GC\ \backslash * \backslash GC) = \backslash GC$.

Proof using.

unfold *hgc*. *applies* *himpl_antisym*.
 $\{ \text{xpull. intros } H1\ K1\ H2\ K2. \text{ xsimpl } (H1\ \backslash * H2). \text{ applies } \times \text{ haffine\_hstar. } \}$
 $\{ \text{xpull. intros } H\ K. \text{ xsimpl } H\ \backslash []. \text{ auto. applies haffine\_hempty. } \}$

Qed.

Another useful property is that the heap predicate $\backslash GC$ itself satisfies *haffine*.

Lemma *haffine_hgc* :

haffine $\backslash GC$.

Proof using.

applies *haffine_hexists*. *intros* *H*. *applies* *haffine_hstar_hpure_l*. *auto*.

Qed.

The process of exploiting $\backslash GC$ to “absorb” affine heap predicates is captured by the following lemma, which asserts that a heap predicate *H* entails $\backslash GC$ whenever *H* is affine.

Lemma *himpl_hgc_r* : $\forall H,$

haffine *H* $\rightarrow$
H $\impl \backslash GC$.

Proof using. *intro* *M*. *intros* *h* *K*. *applies* *hgc_intro*. *applies* *M* *K*. Qed.

In particular, the empty heap predicate $\backslash []$ entails $\backslash GC$, because the empty heap predicate is affine (lemma `haffine_empty`).

Lemma `empty_himpl_hgc` :

$\backslash [] \implies \backslash GC$.

Proof using. *applies* `himpl_hgc_r`. *applies* `haffine_empty`. Qed.

Exercise: 3 stars, standard, especially useful (`himpl_hgc_absorb`) Show that $\backslash GC \times H$ entails $\backslash GC$ for any H satisfying `haffine`.

Lemma `himpl_hgc_absorb_r` : $\forall H$,

`haffine` $H \rightarrow$

$\backslash GC \backslash * H \implies \backslash GC$.

Proof using. *Admitted*.

□

Using the predicate $\backslash GC$, we can reformulate the constrained discard rule `triple_haffine_post` as follows.

Parameter `triple_hgc_post` : $\forall t H Q$,

`triple` $t H (Q \backslash * \backslash GC) \rightarrow$

`triple` $t H Q$.

This rule is more concise than `triple_haffine_post`. Moreover, the piece of heap being discarded, previously described by H' , no longer needs to be provided up front, at the moment of applying the rule. Instead, it can be provided further on in the reasoning, by exploiting `himpl_hgc_r` to prove an entailment of the form $H' \implies \backslash GC$.

47.2.5 Example Instantiations of *heap_affine*

Instantiation of *heap_affine* for a Fully Affine Logic

Module `FULLYAFFINELOGIC`.

To set up a fully affine logic, we consider a definition of *heap_affine* that holds of any heap.

Parameter `heap_affine_def` : $\forall h$,

heap_affine $h = \text{True}$.

It is trivial to check that *heap_affine* satisfies the required distribution properties on the empty heap and the union of heaps.

Lemma `heap_affine_empty` :

heap_affine `Fmap.empty`.

Proof using. `rewrite` $\times$ *heap_affine_def*. Qed.

Lemma `heap_affine_union` : $\forall h1 h2$,

heap_affine $h1 \rightarrow$

```

    heap_affine h2 →
    Fmap.disjoint h1 h2 →
    heap_affine (Fmap.union h1 h2).
Proof using. intros. rewrite× heap_affine_def. Qed.

```

With that instantiation, `haffine` holds of any heap predicate.

```

Lemma haffine_equiv : ∀ H,
  (haffine H) ↔ True.
Proof using.
  intros. iff M.
  { auto. }
  { unfold haffine. intros. rewrite× heap_affine_def. }
Qed.

```

Moreover, the affine top predicate $\backslash GC$ is equivalent to the top predicate `htop`, defined as `fun h ⇒ True` or equivalently as $\backslash \exists H, H$.

```

Definition htop : hprop :=
  \∃ H, H.

```

```

Lemma hgc_eq_htop : hgc = htop.
Proof using.
  unfold hgc, htop. applys himpl-antisym.
  { xsimpl. }
  { xsimpl. intros. rewrite× haffine_equiv. }
Qed.

```

End FULLYAFFINELOGIC.

Instantiation of *heap_affine* for a Fully Linear Logic

Module FULLYLINEARLOGIC.

To set up a fully affine logic, we consider a definition of *heap_affine* that holds only of empty heaps.

```

Parameter heap_affine_def : ∀ h,
  heap_affine h = (h = Fmap.empty).

```

Again, it is not hard to check that *heap_affine* satisfies the required distributivity properties.

```

Lemma heap_affine_empty :
  heap_affine Fmap.empty.
Proof using. rewrite× heap_affine_def. Qed.

```

```

Lemma heap_affine_union : ∀ h1 h2,
  heap_affine h1 →
  heap_affine h2 →

```



```

Fmap.disjoint h1 h2 →
  heap_affine (Fmap.union h1 h2).
Proof using.
  intros K1 K2 D. rewrite heap_affine_def in *.
  subst. rewrite× Fmap.union_empty_r.
Qed.

```

With that instantiation, the predicate *haffine H* is equivalent to $H \implies \backslash []$, that is, it characterizes heap predicates that hold of the empty heap.

```

Lemma haffine_equiv : ∀ H,
  haffine H ↔ (H ==> \ []).

```

```

Proof using.
  intros. unfold haffine. iff M.
  { intros h K. specializes M K. rewrite heap_affine_def in M.
    subst. applies empty_intro. }
  { intros h K. specializes M K. rewrite heap_affine_def.
    applies empty_inv M. }
Qed.

```

Moreover, the affine top predicate $\backslash GC$ is equivalent to the empty heap predicate *empty*.

```

Lemma hgc_eq_empty : hgc = empty.
Proof using.
  unfold hgc. applies himpl_antisym.
  { xpull. introv N. rewrite× haffine_equiv in N. }
  { xsimpl \ []. applies haffine_empty. }
Qed.

```

End FULLYLINEARLOGIC.

47.2.6 Definition of a Partially Affine Separation Logic

Refined Definition of Triples

Module NEWTRIPLES.

We now explain how to refine the notion of Separation Logic triple so as to accomodate the discard rule. Recall the previous definition of triples, capturing a linear logic.

Definition *triple* (*t:trm*) (*H:hprop*) (*Q:val→hprop*) : Prop := forall *s*, *H s* -> eval *s t Q*.

The new discard rule *triple_hstop_post* asserts that postconditions may be freely extended with the $\backslash GC$ predicate. To support this rule, it suffices to modify the definition of *triple* to include the predicate $\backslash GC$ in the postcondition of the underlying Hoare triple, as follows.

```

Definition triple (t:trm) (H:hprop) (Q:val→hprop) : Prop :=
  ∀ s, H s → eval s t (Q \*+ \GC).

```

Next, for the updated definition of `triple` using $\backslash GC$, we establish:

- that all the existing reasoning rules of Separation Logic remain sound, and
- that the new discard rules `triple_hstop_post`, `triple_haffine_hpost`, and `triple_haffine_hpre` are provable.

Soundness of the Existing Rules

Let us update the soundness proof of the rules of Separation Logic to account for the addition of $\backslash GC$ in the definition of `triples`. All the rules are proved using exactly the same proof scripts as before, with `xsimpl` possibly performing extra work for cancelling out occurrences of $\backslash GC$ on both sides of an entailment.

```
Lemma triple_conseq : ∀ t H' Q' H Q,
  triple t H' Q' →
  H ==> H' →
  Q' ==> Q →
  triple t H Q.
```

Proof using.

```
unfolds triple. introv M MH MQ HF. applys× eval_conseq. xsimpl×.
```

Qed.

```
Lemma triple_frame : ∀ t H Q H',
  triple t H Q →
  triple t (H \* H') (Q \** H').
```

Proof using.

```
introv M HF. lets (h1&h2&M1&M2&MD&MU): hstar_inv (rm HF).
subst. specializes M M1. applys eval_conseq.
{ applys eval_frame M MD. } { xsimpl. intros h' →. applys M2. }
```

Qed.

```
Lemma triple_hpure : ∀ t (P:Prop) H Q,
  (P → triple t H Q) →
  triple t (\[P] \* H) Q.
```

Proof using.

```
introv M. intros h (h1&h2&M1&M2&D&U). destruct M1 as (M1&HP).
lets E: hempty_inv HP. subst. rewrite Fmap.union_empty_l. applys⊃ M.
```

Qed.

```
Lemma triple_hexists : ∀ t (A:Type) (J:A→hprop) Q,
  (∀ (x:A), triple t (J x) Q) →
  triple t (hexists J) Q.
```

Proof using. `introv M (x&K). applys M K. Qed.`

We also need to update the reasoning rules for terms. We present one representative example: the proof rule for sequences.

Lemma `triple_seq` : $\forall t1\ t2\ H\ Q\ H1,$
`triple` $t1\ H\ (\text{fun } v \Rightarrow H1) \rightarrow$
`triple` $t2\ H1\ Q \rightarrow$
`triple` $(\text{trm\_seq } t1\ t2)\ H\ Q.$

Proof using.

The proof is structured like in lemma `triple_seq` from chapter `Rules`. `unfold triple.`
`intros M1 M2. intros s Hs. applies eval_seq.`

`{ applies M1 Hs. }`
`{ simpl. intros v1 s2 Hs2. lets (h1&h2&N1&N2&D&->): hstar_inv (rm Hs2).`
`applies eval_conseq. { applies eval_frame D. applies× M2. }`

The main difference is the need to to invoke the lemma `hstar_hgc_hgc` for merging the left-over that originates from `t1` with the $\backslash GC$ that originates from `t2` into a single $\backslash GC$.

`asserts R: ((= h2) ==> \GC). { intros ? ->. auto. } xchange (rm R).`
`xchange hstar_hgc_hgc. xsimpl. }`

`Qed.`

Soundness of the Discard Rules

Let us first establish the soundness of the discard rule `triple_htop_post`.

Exercise: 3 stars, standard, especially useful (`triple_hgc_post`) Prove `triple_hgc_post` with respect to the refined definition of `triple` that includes $\backslash GC$ in the postcondition. Hint: exploit `hstar_hgc_hgc`, with help from the tactics `xchange` and `xsimpl`.

Lemma `triple_hgc_post` : $\forall t\ H\ Q,$
`triple` $t\ H\ (Q\ \backslash^{**}\ \backslash GC) \rightarrow$
`triple` $t\ H\ Q.$

Proof using. *Admitted.*

□

Exercise: 1 star, standard, especially useful (`triple_haffine_post`) Prove that `triple_haffine_post` is derivable from `triple_hgc_post`. Hint: use the property `himpl_hgc_r`, which asserts $H' \implies \backslash GC$.

Lemma `triple_haffine_post` : $\forall t\ H\ H'\ Q,$
`haffine` $H' \rightarrow$
`triple` $t\ H\ (Q\ \backslash^{**}\ H') \rightarrow$
`triple` $t\ H\ Q.$

Proof using. *Admitted.*

□

Exercise: 1 star, standard, optional (`triple_hgc_post_from_triple_haffine_post`) Conversely, prove that `triple_hgc_post` is derivable from `triple_haffine_post`.

Lemma `triple_hgc_post_from_triple_haffine_post` : $\forall t H Q,$
 $\text{triple } t H (Q \setminus^{*+} \setminus GC) \rightarrow$
 $\text{triple } t H Q.$

Proof using. *Admitted*.

□

Exercise: 1 star, standard, optional (`triple_haffine_pre`) Prove that `triple_haffine_pre` is derivable from `triple_hgc_post`. Hint: exploit the frame rule, and leverage `triple_hgc_post` either directly or by invoking its corollary `triple_haffine_post`.

Lemma `triple_haffine_pre` : $\forall t H H' Q,$
 $\text{haffine } H' \rightarrow$
 $\text{triple } t H Q \rightarrow$
 $\text{triple } t (H \setminus^* H') Q.$

Proof using. *Admitted*.

□

Combined Structural Rules

Exercise: 2 stars, standard, optional (`triple_conseq_frame_hgc`) Prove the combined structural rule `triple_conseq_frame_hgc`, which extends `triple_conseq_frame` with the discard rule, replacing $Q1 \setminus^{*+} H2 \implies Q$ with $Q1 \setminus^{*+} H2 \implies Q \setminus^{*+} \setminus GC$.

Lemma `triple_conseq_frame_hgc` : $\forall H2 H1 Q1 t H Q,$
 $\text{triple } t H1 Q1 \rightarrow$
 $H \implies H1 \setminus^* H2 \rightarrow$
 $Q1 \setminus^{*+} H2 \implies Q \setminus^{*+} \setminus GC \rightarrow$
 $\text{triple } t H Q.$

Proof using. *Admitted*.

□

Exercise: 1 star, standard, optional (`triple_ramified_frame_hgc`) Prove the following generalization of the ramified frame rule that includes the discard rule. Hint: take inspiration from the proof of `triple_ramified_frame` presented in the chapter `Wand`.

Lemma `triple_ramified_frame_hgc` : $\forall H1 Q1 t H Q,$
 $\text{triple } t H1 Q1 \rightarrow$
 $H \implies H1 \setminus^* (Q1 \setminus^* (Q \setminus^{*+} \setminus GC)) \rightarrow$
 $\text{triple } t H Q.$

Proof using. *Admitted*.

□

47.3 More Details

47.3.1 Discard Rules in WP Style

Definition of WP for a Partially Affine Separation Logic

In chapter WPsem, we defined $\text{wp } t \ Q$ s as $\text{eval } s \ t \ Q$. To account for affine heaps, we refine the definition to $\text{eval } s \ t \ (Q \setminus^{*+} \setminus GC)$.

Definition $\text{wp } (t:\text{trm}) \ (Q:\text{val} \rightarrow \text{hprop}) : \text{hprop} :=$
 $\text{fun } s \Rightarrow \text{eval } s \ t \ (Q \setminus^{*+} \setminus GC).$

The characteristic equivalence of wp remains valid, with respect to the definition of wp extended with $\setminus GC$ and the definition of triple extended with $\setminus GC$.

Lemma $\text{wp\_equiv} : \forall t \ H \ Q,$
 $(H \implies \text{wp } t \ Q) \leftrightarrow (\text{triple } t \ H \ Q).$

Proof using. *iff M; introv Hs; applies M Hs. Qed.*

The structural reasoning rules also remain valid.

Lemma $\text{wp\_conseq} : \forall t \ Q1 \ Q2,$
 $Q1 \implies Q2 \rightarrow$
 $\text{wp } t \ Q1 \implies \text{wp } t \ Q2.$

Proof using.

unfold wp. introv M. intros s Hs. applies $\times$ eval_conseq. xsimpl $\times$.

Qed.

Lemma $\text{wp\_frame} : \forall t \ H \ Q,$
 $(\text{wp } t \ Q) \setminus^* H \implies \text{wp } t \ (Q \setminus^{*+} H).$

Proof using.

intros. unfold wp. intros h HF.
lets (h1&h2&M1&M2&MD&MU): hstar_inv (rm HF).
subst. applies eval_conseq.
 $\{ \text{applies eval\_frame } M1 \ MD. \}$
 $\{ \text{xsimpl. intros } h' \rightarrow. \text{applies } M2. \}$

Qed.

In weakest precondition style, the discard rule *triple_hgc_post* translates into the entailment $\text{wp } t \ (Q \setminus^{*+} \setminus GC) \implies \text{wp } t \ Q$.

Exercise: 1 star, standard, optional (wp_hgc_post) **Lemma** $\text{wp\_hgc\_post} : \forall t \ Q,$
 $\text{wp } t \ (Q \setminus^{*+} \setminus GC) \implies \text{wp } t \ Q.$

Proof using. *Admitted.*

□

The wp-style presentation of the rule *triple_hgc_pre* is captured by the lemma *wp_haffine_pre* shown below.

Exercise: 3 stars, standard, optional (wp_haffine_pre) Prove `wp_haffine_pre`. Hint: one possibility is to use `eval_conseq` and `wp_frame`, as well as `himpl_hgc_r` and `hstar_hgc_hgc`. Another possibility is to refine the proof of `wp_frame`.

Lemma `wp_haffine_pre` : $\forall t\ H\ Q,$
 $\text{haffine } H \rightarrow$
 $(\text{wp } t\ Q) \setminus * H \implies \text{wp } t\ Q.$

Proof using. *Admitted.*

□

The revised presentation of the wp-style ramified frame rule includes an extra $\setminus GC$ predicate. This rule captures at once all the structural properties of Separation Logic, including the discard rule.

Exercise: 3 stars, standard, optional (wp_haffine_pre) Prove `wp_ramified` with support for $\setminus GC$.

Lemma `wp_ramified` : $\forall Q1\ Q2\ t,$
 $(\text{wp } t\ Q1) \setminus * (Q1 \setminus * (Q2 \setminus * \setminus GC)) \implies (\text{wp } t\ Q2).$

Proof using. *Admitted.*

□

Exploiting the Discard Rule in Proofs

In a practical verification proof, there are two useful ways to discard heap predicates that are no longer needed:

- by invoking `triple_haffine_pre` to remove a specific predicate from the current state, i.e., the precondition; or
- by invoking `triple_hstop_post` to add a $\setminus GC$ into the current postcondition and allow subsequent removal of any predicate that may be left-over in the final entailment justifying that the final state satisfies the postcondition.

Eager removal of predicates from the current state is never necessary: one can always be lazy and postpone the application of the discard rule until the last step of reasoning.

It is cumbersome for the user to anticipate, right from the beginning of the proof of a function, the need to discard heap predicates at the last line of the function body. To ease the work of the user, it suffices to systematically apply the rule `triple_hstop_post` as very first step of the proof of every function. The effect is to extend the postcondition of the function with a $\setminus GC$ predicate. This predicate may be used to absorb any garbage left-over at the end of the proof of the function body. If there is no such garbage, the tactic `xsimpl` automatically discards the $\setminus GC$ predicate.

We implement this strategy by integrating the rule `triple_hstop_post` into the tactic `xwp`, which sets up the verification proof by computing the characteristic formula. To that end,

we generalize the lemma *xwp_lemma*, which the tactic *xwp* applies. Its original statement is as follows.

```
Parameter xwp_lemma : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wpgen ((x, v2)::nil) t1 Q →
  triple (trm_app v1 v2) H Q.
```

Its generalized form extends the postcondition from Q to $Q \setminus^{*+} \setminus GC$, as shown below.

```
Lemma xwp_lemma' : ∀ v1 v2 x t1 H Q,
  v1 = val_fun x t1 →
  H ==> wpgen ((x, v2)::nil) t1 (Q \setminus^{*+} \setminus GC) →
  triple (trm_app v1 v2) H Q.
```

Proof using. *intros E M. applies triple_hgc_post. applies × xwp_lemma. Qed.*

We update the tactic *xwp* to exploit the lemma *xwp_lemma'* instead of *xwp_lemma*.

```
Tactic Notation "xwp" :=
  intros; applies xwp_lemma';
  [ reflexivity | simpl; unfold wpgen_var; simpl ].
```

Example Proof in an Affine Separation Logic

Using the updated version of *xwp*, let us revisit the proof of our motivating example *succ_using_incr* in a fully affine logic where any predicate can be discarded.

```
Module MOTIVATINGEXAMPLEWITHUPDATEDXWP.
Export MOTIVATINGEXAMPLE.
```

Assume a fully affine logic.

```
Parameter haffine_hany : ∀ (H:hprop),
  haffine H.
```

Observe, in the proof below, the $\setminus GC$ introduced in the postcondition by the call to *xwp*.

```
Lemma triple_succ_using_incr : ∀ (n:int),
  triple (succ_using_incr n)
    \[]
    (fun r => \[r = n+1]).
```

Proof using.

```
xwp. xapp. intros r. xapp. xapp. xsimpl. { auto. }
```

```
applies himpl_hgc_r. applies haffine_hany.
```

Qed.

We will show further on how to automate the work from the last line of the proof above, by setting up *xsimpl* to automatically resolve goals of the form $H ==> \setminus GC$.

```
End MOTIVATINGEXAMPLEWITHUPDATEDXWP.
```

47.3.2 Revised Definition of `mkstruct`

Recall the definition `mkstruct`, which is key to implementing `wpgen` and the `x-tactics`.

Definition `mkstruct` (F :formula) : formula := fun $Q \Rightarrow \backslash \text{exists } Q', (F \ Q') \backslash * (Q' \backslash - * Q)$.

This definition can be generalized to handle not just the consequence and frame rules, but also the discard rule. To that end, `mkstruct` is extended with an additional `\GC`, as follows.

Definition `mkstruct` (F :formula) : formula :=
 fun $Q \Rightarrow \backslash \exists Q', F \ Q' \backslash * (Q' \backslash - * (Q \backslash * + \backslash GC))$.

Let us prove that this revised definition of `mkstruct` does indeed satisfy the `wp`-style statement of the discard rule, which is stated in a way similar to `wp_hgc_post`.

Lemma `mkstruct_hgc` : $\forall Q \ F$,
 $\text{mkstruct } F \ (Q \backslash * + \backslash GC) \implies \text{mkstruct } F \ Q$.

Proof using.

intros. unfold `mkstruct`. set ($X := \text{hgc}$) at 3. replace X with $(\backslash GC \backslash * \backslash GC)$.
 $\{ \text{ximpl.} \} \{ \text{subst } X. \text{apply hstar\_hgc\_hgc.} \}$

Qed.

Further, let us prove that the revised definition of `mkstruct` still satisfies the four originally required properties: erasure, consequence, frame, and monotonicity.

Lemma `mkstruct_erase` : $\forall F \ Q$,
 $F \ Q \implies \text{mkstruct } F \ Q$.

Proof using.

intros. unfold `mkstruct`. $\text{ximpl } Q$. apply `himpl_hgc_r`. apply `haffine_hempty`.

Qed.

Lemma `mkstruct_conseq` : $\forall F \ Q1 \ Q2$,
 $Q1 \implies Q2 \rightarrow$
 $\text{mkstruct } F \ Q1 \implies \text{mkstruct } F \ Q2$.

Proof using.

intro WQ . unfold `mkstruct`. xpull . intros Q' . $\text{ximpl } Q'$. $\text{xchange } WQ$.

Qed.

Lemma `mkstruct_frame` : $\forall F \ H \ Q$,
 $(\text{mkstruct } F \ Q) \backslash * H \implies \text{mkstruct } F \ (Q \backslash * + H)$.

Proof using.

intros. unfold `mkstruct`. xpull . intros Q' . $\text{ximpl } Q'$.

Qed.

Lemma `mkstruct_monotone` : $\forall F1 \ F2 \ Q$,
 $(\forall Q, F1 \ Q \implies F2 \ Q) \rightarrow$
 $\text{mkstruct } F1 \ Q \implies \text{mkstruct } F2 \ Q$.

Proof using.

intro WF . unfolds `mkstruct`. xpull . intros Q' . $\text{xchange } WF$. $\text{ximpl } Q'$.

Qed.

Exercise: 2 stars, standard, optional (mkstruct_haffine_post) Prove the reformulation of `triple_haffine_post` adapted to `mkstruct`, for discarding an affine piece of postcondition.

```
Lemma mkstruct_haffine_post : ∀ H Q F,  
  haffine H →  
  mkstruct F (Q \*+ H) ==> mkstruct F Q.
```

Proof using. *Admitted*.

□

Exercise: 2 stars, standard, optional (mkstruct_haffine_pre) Prove the reformulation of `triple_haffine_pre` adapted to `mkstruct`, for discarding an affine piece of postcondition.

```
Lemma mkstruct_haffine_pre : ∀ H Q F,  
  haffine H →  
  (mkstruct F Q) \* H ==> mkstruct F Q.
```

Proof using. *Admitted*.

□

End NEWTRIPLES.

47.3.3 The Tactic *xaffine* and the Behavior of *xsimpl* on $\backslash GC$

Module XAFFINE.

The tactic *xaffine* applies to a goal of the form `haffine H`. It simplifies the goal using all the distributivity rules associated with `haffine`. Ultimately, it invokes `eauto with haffine`, which can leverage knowledge specific to the definition of `haffine` from the Separation Logic setup at hand.

Create HintDb *haffine*.

```
Tactic Notation "xaffine" :=  
  repeat match goal with ⊢ haffine ?H ⇒  
    match H with  
    | (hempty) ⇒ apply haffine_hempty  
    | (hpure _) ⇒ apply haffine_hpure  
    | (hstar - _) ⇒ apply haffine_hstar  
    | (hexists _) ⇒ apply haffine_hexists  
    | (hforall _) ⇒ apply haffine_hforall  
    | (hgc) ⇒ apply haffine_hgc  
    | _ ⇒ eauto with haffine  
  end
```

```

end.
Lemma xaffine_demo :  $\forall H1 H2 H3$ ,
  haffine  $H1 \rightarrow$ 
  haffine  $H3 \rightarrow$ 
  haffine  $(H1 \setminus * H2 \setminus * H3)$ .
Proof using. intros K1 KJ. xaffine. Abort.
End XAFFINE.

Module XSIMPLEXTENDED.
Import LIBSEPREFERENCE.

The tactic xsimpl is extended with support for simplifying goals of the form  $H ==> \setminus GC$  into haffine H, using lemma himpl_hgc_r. For example, xsimpl can simplify the goal  $H1 \setminus * H2 ==> H2 \setminus * \setminus GC$  into just haffine H1.

Lemma xsimpl_xgc_demo :  $\forall H1 H2$ ,
   $H1 \setminus * H2 ==> H2 \setminus * \setminus GC$ .
Proof using. intros. xsimpl. Abort.

In addition, xsimpl invokes the tactic xaffine to simplify side-conditions of the form haffine H. For example, xsimpl can prove the following lemma.

Lemma xsimpl_xaffine_demo :  $\forall H1 H2$ ,
  haffine  $H1 \rightarrow$ 
   $H1 \setminus * H2 ==> H2 \setminus * \setminus GC$ .
Proof using. intros K1. xsimpl. Qed.
End XSIMPLEXTENDED.

```

47.3.4 Tactics for Applying the Discard Rules

```

Module XGC.
Import LIBSEPREFERENCE.

```

This section presents the discard tactics *xgc*, *xc_keep*, and *xgc_post*. Their implementation leverages the discard property of *mkstruct*, reproduced below.

```

Parameter mkstruct_hgc :  $\forall Q F$ ,
  mkstruct  $F (Q \setminus *+ \setminus GC) ==> mkstruct F Q$ .

```

The tactic *xgc H1* removes *H1* from the precondition (i.e., from the current state), in the course of a proof exploiting a formula produced by *wpgen*. More precisely, the tactic *xgc H1* removes *H1* from the current precondition *H*. It leverages *xsimpl* to simplify the entailment $H ==> H1 \setminus * H2$ and infer *H2*, which describes what remains after removing *H1* from *H*.

```

Lemma xgc_lemma:  $\forall H1 H2 H F Q$ ,
   $H ==> H1 \setminus * H2 \rightarrow$ 
  haffine  $H1 \rightarrow$ 
   $H2 ==> mkstruct F Q \rightarrow$ 

```

$H \Rightarrow \text{mkstruct } F \ Q.$

Proof using.

intros WH K M. xchange WH. xchange M.
applies himpl_trans mkstruct_frame.
applies himpl_trans_r mkstruct_hgc.
applies mkstruct_conseq. xsimpl.

Qed.

Tactic Notation "xgc" constr(H) :=
eapply (@xgc_lemma H); [*xsimpl* | *xaffine* |].

Lemma xgc_demo : $\forall H1 \ H2 \ H3 \ F \ Q,$
haffine $H2 \rightarrow$
($H1 \setminus * H2 \setminus * H3$) $\Rightarrow \text{mkstruct } F \ Q.$

Proof using. *intros K2. xgc H2. Abort.*

The tactic *xgc_keep* H is a variant of *xgc* that enables discarding everything but H from the precondition.

The implementation of the tactic leverages the same lemma *xgc_lemma*, but providing $H2$ instead of $H1$.

Tactic Notation "xgc_keep" constr(H) :=
eapply (@xgc_lemma _ H); [*xsimpl* | *xaffine* |].

Lemma xgc_keep_demo : $\forall H1 \ H2 \ H3 \ F \ Q,$
haffine $H1 \rightarrow$
haffine $H3 \rightarrow$
($H1 \setminus * H2 \setminus * H3$) $\Rightarrow \text{mkstruct } F \ Q.$

Proof using. *intros K1 K3. xgc_keep H2. Abort.*

The tactic *xgc_post* simply extends the postcondition with a $\setminus GC$, to enable subsequent discarding of heap predicates in the final entailment.

Lemma xgc_post_lemma : $\forall H \ Q \ F,$
 $H \Rightarrow \text{mkstruct } F \ (Q \setminus *+ \setminus GC) \rightarrow$
 $H \Rightarrow \text{mkstruct } F \ Q.$

Proof using. *intros M. xchange M. applies mkstruct_hgc. Qed.*

Tactic Notation "xgc_post" :=
apply xgc_post_lemma.

The example below illustrates a usage of the *xgc_post* tactic.

Lemma xgc_keep_demo : $\forall H1 \ H2 \ H3 \ F \ Q,$
haffine $H1 \rightarrow$
haffine $H3 \rightarrow$
 $H1 \Rightarrow \text{mkstruct } F \ (Q \setminus *+ H2 \setminus *+ H3) \rightarrow$
 $H1 \Rightarrow \text{mkstruct } F \ Q.$

Proof using.

intros K1 K3 M.

```

    xgc_post.    xchange M. applies mkstruct_conseq.
    xsimpl.
Qed.
End XGC.

```

47.4 Optional Material

47.4.1 Alternative Statement for Distribution of **haffine** on Quantifiers

Module HAFFINEQUANTIFIERS.

Recall the lemmas **haffine_hexists** and **haffine_hforall**.

Lemma **haffine_hexists** : forall A (J:A->hprop), (forall x, haffine (J x)) -> haffine (\exists x, (J x)). Lemma **haffine_hforall** : forall A '{Inhab A} (J:A->hprop), (forall x, haffine (J x)) -> haffine (\forall x, (J x)).

They can be reformulated in a more concise way, as explained next.

First, to smoothly handle the distribution on the quantifiers, let us extend the notion of “affinity” to postconditions. The predicate **haffine_post** *J* asserts that **haffine** holds of *J* *x* for any *x*.

Definition **haffine_post** (A:Type) (J:A->hprop) : Prop :=
 $\forall x, \text{haffine } (J \ x).$

The rules then reformulate as follows.

Lemma **haffine_hexists** : $\forall A \ (J:A \rightarrow \text{hprop}),$
 $\text{haffine\_post } J \rightarrow$
 $\text{haffine } (\text{hexists } J).$

Proof using. *intros F1 (x&Hx). applies $\times$ F1. Qed.*

Lemma **haffine_hforall** : $\forall A \ \{ \text{Inhab } A \} \ (J:A \rightarrow \text{hprop}),$
 $\text{haffine\_post } J \rightarrow$
 $\text{haffine } (\text{hforall } J).$

Proof using.

intros IA F1 Hx. lets N: hforall_inv Hx. applies $\times$ F1 (arbitrary (A:=A)).

Qed.

End HAFFINEQUANTIFIERS.

47.4.2 Deriving the Rule GC-Post from the Rule GC-Pre

Module FROMPRETOPSTGC.

Import RULES PROOFS SAME SEMANTICS.

Earlier on, we proved that *triple_hgc_pre* is derivable from *triple_hgc_post*, through a simple application of the frame rule. We wrote that, reciprocally, the rule *triple_hgc_post* is derivable from *triple_hgc_pre*, yet with a slightly more involved proof. Let us present this proof. Concretely, assume *triple_hgc_pre* and let us prove the result *triple_hgc_post*.

```
Parameter triple_hgc_pre : ∀ t H Q,
  triple t H Q →
  triple t (H \* \GC) Q.
```

```
Parameter triple_hgc_post : ∀ t H Q,
  triple t H (Q \*+ \GC) →
  triple t H Q.
```

The key idea of the proof is that a term *t* admits the same behavior as *let x = t in x*. Recall from chapter [Rules](#) the lemma *eta_same_triples* which asserts the equivalence of *t* and *let x = t in x*.

```
Parameter eta_same_triples : ∀ (t:trm) (x:var) H Q,
  triple t H Q ↔ triple (trm_let x t x) H Q.
```

To discard a predicate from the postcondition of *t* if we only have at hand a rule for discarding predicates from preconditions, we replace *t* with *let x = t in x* and apply the discard rule after evaluating *t* but before returning the variable *x*.

```
Lemma triple_hgc_post_from_hgc_pre : ∀ t H Q,
  triple t H (Q \*+ \GC) →
  triple t H Q.
```

Proof using.

```
  introv M. rewrite (eta_same_triples t "x"). applys triple_let.
  { applys M. }
  { intros v. simpl. applys triple_hgc_pre. applys triple_val. auto. }
```

Qed.

End FROMPRETOPOSTGC.

47.4.3 Historical Notes

The seminal presentation of Separation Logic concerned a linear logic for a programming language with explicit deallocation. More recent works on Separation Logic for ML-style languages equipped with a garbage collector consider affine logics. For example, the original presentation of the Iris framework provides an affine entailment, for which $H \implies \llbracket \cdot \rrbracket$ is always true. Follow-up work on Iris provides encodings for supporting linear resources, i.e., resources that cannot be dropped on the floor.

The present chapter gives a presentation of Separation Logic featuring a customizable predicate *haffine* for controlling which resources should be treated as affine, and which ones should be treated as linear. This direct approach to controlling linearity was introduced in the context of CFML, in work by [Guéneau, Jourdan, Charguéraud, and Pottier](#) 2019 (in Bib.v).

Chapter 48

Library SLF.Arrays

48.1 Arrays: Reasoning about Arrays

```
Set Implicit Arguments.
From SLF Require Import LibSepReference LibSepTLCbuffer.
#[global] Hint Rewrite conseq_cons' : rew_listx.
Open Scope wp_scope.

Implicit Types P : Prop.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
Implicit Type p q : loc.
Implicit Type k : nat.
Implicit Type i n : int.
Implicit Type v : val.
Implicit Types z : nat.
```

48.2 First Pass

This chapter explains how to specify operations on ML-style arrays in Separation Logic. In ML-like languages, an array of size `n` is an allocated block of size `n+1` with a header field that stores the length of the array. To perform a *get* or a *set* operation on an array, the programmer provides the pointer to the array object – that is, the address of the header cell as well as the index on which to operate.

The chapter starts with a presentation of *large-footprint* operations for arrays, expressed with respect to a representation predicate of the form `harray L p`. Then it presents *small-footprint* specifications for the same operations, involving a representation predicate of the form `hcell v p i` for representing each cell, as well as a predicate, written `hheader p n` for representing the header field. Interestingly, the header cell predicate alone suffices for reading the length of an array.

We will illustrate the benefits of small-footprint specifications for reasoning about the recursive function *quicksort*, which operates on array segments. Segments are described using the heap predicate *harray_seg* $L\ p\ i$. Array segments allow “framing” the parts of arrays that are not involved in a recursive call.

The “More Details” part of the chapter explains how to define the predicates *hheader*, *hcell*, *hseg*, and *harray* with respect to the representation of heaps as a finite map from locations to values. The “optional material” part of the chapter shows how to implement and verify *make*, *length*, *get*, and *set* in terms of block allocation and pointer arithmetic.

48.2.1 Large-Footprint Specifications for Array Operations

The heap predicate *harray* $L\ p$ asserts that an array is allocated at address p and that its elements are described by the list L . At this stage, let us axiomatize this predicate.

Parameter *harray* : $\forall (L:\text{list val}) (p:\text{loc}), \text{hprop}$.

The primitive operation *val_array_make* $n\ v$ allocates a fresh array of length n , with each of its cells containing the value v .

Parameter *val_array_make* : **val**.

The specification of *val_array_make* requires the length n to be nonnegative. The output array is described as a list made of n copies of the value v . This list is built by the utility function *LibList.make* $(\text{abs } n)\ v$, where *abs* converts the integer n into a natural number.

Parameter *triple_array_make* : $\forall n\ v,$
 $n \geq 0 \rightarrow$
 $\text{triple } (\text{val\_array\_make } n\ v)$
 $\backslash []$
 $(\text{funloc } p \Rightarrow \text{harray } (\text{LibList.make } (\text{abs } n)\ v)\ p).$

Alternatively, the array produced by *make* $n\ v$ can be described extensionally, as a list L of length n such that all its elements are equal to v .

Parameter *triple_array_make'* : $\forall n\ v,$
 $n \geq 0 \rightarrow$
 $\text{triple } (\text{val\_array\_make } n\ v)$
 $\backslash []$
 $(\text{funloc } p \Rightarrow \backslash \exists L, \text{harray } L\ p$
 $\quad \backslash * \backslash [n = \text{length } L]$
 $\quad \backslash * \backslash [\forall i, 0 \leq i < n \rightarrow \text{LibList.nth } (\text{abs } i)\ L = v]).$

We’ll come back shortly afterwards on the benefits of using the first presentation, expressed using *LibList.make*.

The operation *val_array_length* p returns the length of the array allocated at location p .

Parameter *val_array_length* : **val**.

The specification for `val_array_length` expects a heap predicate of the form `harray L p` and asserts that the return value is the length of the logical list `L`. The postcondition repeats `harray L p` to capture the fact that the array is not modified during the operation.

Parameter `triple_array_length` : $\forall L p,$
`triple (val_array_length p)`
`(harray L p)`
`(fun r  $\Rightarrow$   $\backslash[r = \text{length } L] \backslash * \text{harray } L p$ ).`

The operation `val_array_get p i` returns the contents of the `i`-th cell of the array at location `p`.

Parameter `val_array_get` : **val**.

The specification of `val_array_get` is as follows. The precondition describes the array in the form `harray L p`, with a premise that requires the index `i` to be in the valid range, that is, between zero (inclusive) and the length of `L` (exclusive). The postcondition asserts that the result value is the `i`-th element of the list `L`, written `nth (abs i) L`. The postcondition again repeats `harray L p` because the array is unchanged.

Parameter `triple_array_get` : $\forall L p i,$
 $0 \leq i < \text{length } L \rightarrow$
`triple (val_array_get p i)`
`(harray L p)`
`(fun r  $\Rightarrow$   $\backslash[r = \text{LibList.nth (abs i) } L] \backslash * \text{harray } L p$ ).`

The operation `val_array_set p i v` updates the contents of the `i`-th cell of the array at location `p`.

Parameter `val_array_set` : **val**.

The specification of `val_array_set` admits the same precondition as `val_array_get`, with `harray L p` and the constraint $0 \leq i < \text{length } L$. Its postcondition describes the updated array using a predicate of the form `harray L' p`, where `L'` corresponds to the update of the `i`-th item of the list `L` with the value `v`, written `update (abs i) v L`.

Parameter `triple_array_set` : $\forall L p i v,$
 $0 \leq i < \text{length } L \rightarrow$
`triple (val_array_set p i v)`
`(harray L p)`
`(fun _  $\Rightarrow$  harray (LibList.update (abs i) v L) p).`

`#[global] Hint Resolve triple_array_get triple_array_set`
`triple_array_make triple_array_length : triple.`

Above, we saw two ways of stating the postcondition of the `array_make` operation: either describing the result using a corresponding logical operation, like in `triple_array_make`; or describing the result extensionally by specifying the value of each cell, as in `triple_array_make'`. A question of this kind appears not only here for `array_make`, but for numerous other operations involving arrays. There is no definite rule, but the following considerations motivate our preference for the first style.

A first aspect to take into account is that expressing the postcondition in terms of a logical operation like in `triple_array_make` leads to a more concise specification.

A second aspect to consider is how to reason about a sequence of operations involving arrays. For example, consider the following program fragment.

OCaml:

```
let t = Array.make n v in t.(i) <- w; t.(j)
```

When using logical operations, the resulting array may be described as `nth (abs j) (update (abs i) w (make n v))`. If $i \neq j$, this expression can be simplified by rewriting into `v`. In other words, all the reasoning is carried out by in-place rewriting without involving any hypothesis.

When using extensional characterization, the operation `Array.make n v` is described as an array L such that `nth (abs k) L = v` holds for any valid index k . In that same style, the operation `Array.set t i w` would be described as returning an array L' such that `nth (abs i) L' = w` and `nth (abs k) L' = nth (abs k) L` for any valid index k such that $k \neq i$. Then, the final value is described as `nth (abs j) L'`. To prove this value equal to `v` under the assumption $i \neq j$, we would need to perform two rewriting steps using each of the two hypotheses: `nth (abs j) L'` is equal to `nth (abs j) L`, hence equal to `v`.

In summary, both approach work. But when a logical function corresponding to the program function can be used, hypotheses are more concise and tend to be better suited for automated simplification by rewriting. Of course, there might be overheads to defining this logical function, but if it already exists in the library of our theorem prover, then the overhead is null.

48.2.2 Small-Footprint Specifications for Array Operations

The large-footprint specifications presented above are sufficiently expressive for verifying sequential programs manipulating arrays, but they are too limited for verifying a parallel program that concurrently updates two independent segments of an array. Indeed, even the verification of certain sequential programs can benefit from the use of smaller-footprint specifications, which require the ownership not of the whole array but only of a specific subset of the array cells.

For example, the divide-and-conquer `quicksort` function sorts the elements in a given range of cells. Even though `quicksort` is not a concurrent function, using a small-footprint specification avoids the need to explicitly state the fact that cells of the array outside of the segment targeted by a recursive call remain unmodified.

In what follows, we present small-footprint specifications for operating on individual cells and for operating on array segments, then present the proof of quicksort.

Small-footprint specifications are expressed using heap predicates for individual array and header cells.

The heap predicate `hcell v p i` asserts that the cell at index i in the array p stores the value v .

Internally, as explained below, `hcell v p i` can be defined as `(p+1+i) ~> v`, with the `+1` corresponding to the header cell.

Parameter $hcell : \forall (v:\text{val}) (p:\text{loc}) (i:\text{int}), hprop$.

The heap predicate $hheader\ n\ p$ asserts that the header cell of the array at address p stores the length n . Internally, $hheader\ n\ p$ can be defined as $p \rightsquigarrow n$.

Parameter $hheader : \forall (n:\text{int}) (p:\text{loc}), hprop$.

To read the length of an array, it is sufficient to provide the heap predicate associated with the header field of that array. In other words, there is no need to have the ownership of any cell of an array for reading its length.

Parameter $triple\_array\_length\_hheader : \forall\ n\ p,$
 $triple\ (val\_array\_length\ p)$
 $(hheader\ n\ p)$
 $(fun\ r \Rightarrow \backslash[r = (n:\text{int})] \backslash * hheader\ n\ p).$

To read or write in a given array cell, it is sufficient to provide the $hcell$ predicate associated with that cell. There is no need to own the header cell or other cells of the array.

Parameter $triple\_array\_get\_hcell : \forall\ p\ i\ v,$
 $triple\ (val\_array\_get\ p\ i)$
 $(hcell\ v\ p\ i)$
 $(fun\ r \Rightarrow \backslash[r = v] \backslash * hcell\ v\ p\ i).$

Parameter $triple\_array\_set\_hcell : \forall\ p\ i\ v\ w,$
 $triple\ (val\_array\_set\ p\ i\ v)$
 $(hcell\ w\ p\ i)$
 $(fun\ _ \Rightarrow hcell\ v\ p\ i).$

Note: technically, an ML runtime performs bound checks on get and set operations, to ensure that the indices provided fall within the array. These bound-checks operations do involve a read to the header field. Thus, it may appear that the header cell ought to be involved in the specification of get and set . However, providing the $hheader$ predicate is not actually required for verifying the correctness a program. If one wanted to formally verify a *runtime system*, on the other hand, one would argue instead that headers are read-only cells, and that the access permissions over such cells can be “divided” among the client code and the runtime system. The realization of this “division” mechanism is based on the use of “fractional permissions”, a Separation Logic concept that is just beyond the scope of the present course.

48.2.3 Heap Predicate for Array Segments

So far, we have presented small-footprint specifications expressed in terms of $hheader$ and $hcell$. But the creation of an array via $val\_array\_make\ n\ v$ produces a heap predicate of the form $harray\ L\ p$. Thus, it remains to explain how to convert an $harray$ predicate into the separating conjunction of an $hheader$ predicate and a set of $hcell$ predicates.

The auxiliary predicate $hseg\ L\ p\ j$ describes an “array segment”: it describes the iterated separating conjunction of the predicate $hcell$ over a set of consecutive cells starting at index

j , with elements described by the list L . For example, $\text{hseg } (x0::x1::x2::\text{nil}) \text{ p } j$ corresponds to $\text{hcell } x0 \text{ p } (j+0) \setminus * \text{hcell } x1 \text{ p } (j+1) \setminus * \text{hcell } x2 \text{ p } (j+2)$. Internally, this corresponds to the cells at address $p+1+j$, $p+2+j$, and $p+3+j$, skipping the header cell located at address $p+j$.

```
Fixpoint hseg (L:list val) (p:loc) (j:int) : hprop :=
  match L with
  | nil  $\Rightarrow \setminus []$ 
  |  $x::L' \Rightarrow (\text{hcell } x \text{ p } j) \setminus * (\text{hseg } L' \text{ p } (j+1))$ 
  end.
```

If the list L is empty, then hseg nil p j is equivalent to the empty heap predicate. In particular, it does not assert that j is a valid index in the array. (The proofs of this lemma and the following one appear further in the file.)

```
Parameter hseg_nil :  $\forall \text{ p } j$ ,
  hseg nil p j =  $\setminus []$ .
```

If the list L is a singleton, then $\text{hseg } (v::\text{nil}) \text{ p } j$ is equivalent to hcell v p j .

```
Parameter hseg_one :  $\forall v \text{ p } j$ ,
  hseg (v::nil) p j = hcell v p j.
```

A key result captures how a range of consecutive cells may be split in two parts. Concretely, a heap predicate describing a segment with elements $(L1 ++ L2)$ can be split into a predicate describing a first segment with elements $L1$ and another predicate describing a second segment with elements $L2$. The two parts can be merged back into the original form at any time, as captured by the equality symbol in the statement below.

```
Parameter hseg_app :  $\forall \text{ p } j \text{ L1 } L2$ ,
  hseg (L1 ++ L2) p j
  = hseg L1 p j  $\setminus * \text{hseg } L2 \text{ p } (j + \text{length } L1)$ .
```

For verifying programs, two corollaries are convenient. The lemma `hseg_cons` isolates the first cell of a segment, whereas the lemma `hseg_last` isolates the last cell of a segment.

```
Lemma hseg_cons :  $\forall v \text{ p } j \text{ L}$ ,
  hseg (v::L) p j = hcell v p j  $\setminus * \text{hseg } L \text{ p } (j+1)$ .
```

Proof using. `auto. Qed.`

```
Lemma hseg_last :  $\forall v \text{ p } j \text{ L}$ ,
  hseg (L&v) p j = hseg L p j  $\setminus * \text{hcell } v \text{ p } (j+\text{length } L)$ .
```

Proof using. `intros. rewrite hseg_app. rewrite hseg_cons, hseg_nil. xsimpl. Qed.`

These two corollaries themselves admit additional reformulations that help merge back the isolated head and tail cells.

Their statements do not constraint the offsets to be syntactically of the form $j + 1$ or $j + \text{length } L1$, but instead introduce arithmetic equalities that the tactic `math` can generally handle.

```
Lemma hseg_cons_r :  $\forall L v \text{ p } j1 \text{ j2}$ ,
```

```

j2 = j1 + 1 →
hcell v p j1 \* hseg L p j2 ==> hseg (v::L) p j1.
Proof using. intros. subst. rewrite× hseg_cons. Qed.

Lemma hseg_app_r : ∀ p L1 L2 j1 j2,
j2 = j1 + length L1 →
hseg L1 p j1 \* hseg L2 p j2 ==> hseg (L1 ++ L2) p j1.
Proof using. intros. subst. rewrite× hseg_app. Qed.

Lemma hseg_last_r : ∀ v p j L j',
j' = (j+length L) →
hseg L p j \* hcell v p j' ==> hseg (L&v) p j.
Proof using. intros. subst. rewrite× hseg_last. Qed.

```

48.2.4 Derived Segment-Based Specifications for Array Operations

For reasoning about programs that operate over array segments, such as *quicksort*, it is convenient to specify the functions *make*, *length*, *get* and *set* exclusively in terms of *hheader* and *hseg*. The lemma *triple_array_length_header*, already stated earlier, specifies *length* in terms of *hheader*. For the other operations, we consider the following derived specifications.

```

Parameter triple_array_make_hseg : ∀ n v,
n ≥ 0 →
triple (val_array_make n v)
\ []
(funloc p ⇒ hheader (abs n) p \* hseg (LibList.make (abs n) v) p 0).

Parameter triple_array_get_hseg : ∀ L p i j,
0 ≤ i - j < length L →
triple (val_array_get p i)
(hseg L p j)
(fun r ⇒ \[r = LibList.nth (abs (i-j)) L] \* hseg L p j).

Parameter triple_array_set_hseg : ∀ L p i j v,
0 ≤ i - j < length L →
triple (val_array_set p i v)
(hseg L p j)
(fun _ ⇒ hseg (LibList.update (abs (i-j)) v L) p j).

```

48.2.5 Verification of the Safety of QuickSort

```

Module QUICKSORT.
Export NOTATIONFORVARIABLES.

```

Let us illustrate the benefits of these segment-based specifications by reasoning about the divide-and-conquer *quicksort* function. For simplicity, let us consider an array that stores integer values. The implementation of quicksort is standard, using an auxiliary *pivot*

function. The operation *pivot* *p i n* processes the segment of array *p* that starts at index *i* and is made of *n* elements. It first chooses an arbitrary element from that segment as pivot. It then reorders the elements from the segment, separating the values less than or equal to the pivot value from the values greater than the pivot value. It returns the index at which the pivot value ends up being stored in the segment.

OCaml:

```
let pivot p i n = ...
let rec quicksort p i n = if n > 1 then begin let j = pivot p i n in let n1 = j - i in quicksort p i n1; quicksort p (j+1) (n-n1-1) end
```

As a warm-up before establishing the full functional correctness of *quicksort*, we begin by establishing its safety and termination. The core of this argument is proving that every array access is valid, through the use of array-segment specifications. The safety proof captures the essence of the ownership reasoning at play, including the framing process over recursive calls. In the “More Details” section, we generalize the proof to show that *quicksort* also correctly sorts its input array.

First, *pivot*.

Its safety specification asserts that *pivot* *p i n* operates on a range of size *n*, starting at index *i* of the array *p*. This range is described in the precondition by a list *L*, and in the postcondition by a list *L'*. This list *L'* decomposes as *L1* ++ *x* :: *L2*, where *x* denotes the pivot value.

In the full correctness proof, we will further assert that *L1* contains only values smaller than *x* and *L2* only values at least as large as *x*, but for establishing safety these assertions are not needed.

Parameter *val_pivot* : **val**.

Parameter *triple_pivot_safety* : $\forall p\ i\ n\ L,$
 $n = \text{length } L \rightarrow$
 $n \geq 1 \rightarrow$
 $\text{triple } (\text{val\_pivot } p\ i\ n)$
 $(\text{hseg } L\ p\ i)$
 $(\text{fun } r \Rightarrow \exists (j:\text{int}), \exists [r = \text{val\_int } j] \setminus *$
 $\setminus \exists (x:\text{int}) (L' \ L1\ L2:\text{list val}), \text{hseg } L'\ p\ i \setminus * \setminus [$
 $\text{length } L' = \text{length } L$
 $\wedge L' = L1 ++ \text{val\_int } x :: L2$
 $\wedge j - i = \text{length } L1]).$

Now, the recursive function *quicksort* *p i n* sorts an segment of length *n*, starting at index *i*, in the array *p*.

Definition *val_quicksort* : **val** :=

```
<{ fix 'f 'p 'i 'n =>
  let 'b = 'n > 1 in
  if 'b then
    let 'j = val_pivot 'p 'i 'n in
```

```

    let 'n1 = 'j - 'i in
    'f 'p 'i 'n1;
    let 'i2 = 'j + 1 in
    let 'n3 = 'n - 'n1 in
    let 'n2 = 'n3 - 1 in
    'f 'p 'i2 'n2
  end }>.

```

Exercise: 4 stars, standard, especially useful (`triple_quicksort_safety`) Prove that *quicksort* operates on the targeted array segment without interfering with the other cells of the array. This property is captured by the specification shown below. Hint: use *xapp* *triple_pivot_safety* to reason about the call to *pivot*.

```

Lemma triple_quicksort_safety : ∀ p i n L,
  n = length L →
  i ≥ 0 →
  triple (val_quicksort p i n)
    (hseg L p i)
    (fun _ => \∃ L', \[length L' = length L] \* hseg L' p i).

```

Proof using.

```

  introv Hn Hi. gen i L. induction_wf IH: (downto 0) n. intros.
  Admitted.

```

□

48.3 More Details

48.3.1 Formalization of Sorted Lists

Section `SortedLists`.

To refine the above safety proof into a functional correctness proof, we first need to formalize the notations of permutation and sortedness.

The predicate `permut L L'` asserts that *L'* is a permutation of *L*.

```

Inductive permut (A:Type) : list A → list A → Prop :=
| permut_mid : ∀ L1 L2 L3 L4,
  permut (L1 ++ L2 ++ L3 ++ L4) (L1 ++ L3 ++ L2 ++ L4)
| permut_trans : ∀ L3 L1 L2,
  permut L1 L2 →
  permut L2 L3 →
  permut L1 L3.

```

The predicate `permut` is reflexive and symmetric.

```

Lemma permut_refl : ∀ A (L:list A),

```

permut L L .

Proof using. intros. *applies* permut_mid (@nil A) (@nil A) (@nil A) L . Qed.

Lemma permut_sym : $\forall A (L1\ L2: \text{list } A),$

permut $L1\ L2 \rightarrow$

permut $L2\ L1$.

Proof using.

intros M . induction M .

{ *applies* $\times$ permut_mid. }

{ *applies* $\times$ permut_trans IHM2. }

Qed.

#[local] Hint Resolve permut_refl.

If L' is a permutation of L , then it has the same length as L .

Lemma permut_length : $\forall A (L\ L': \text{list } A),$

permut $L\ L' \rightarrow$

length L = length L' .

Proof using. *intros* M . induction M ; *rew_list* in *; try *math*. Qed.

If $L1'$ is a permutation of $L1$ and $L2'$ is a permutation of $L2$, then $L1' ++ L2'$ is a permutation of $L1 ++ L2$.

Lemma permut_app : $\forall A (L1\ L2\ L1'\ L2': \text{list } A),$

permut $L1\ L1' \rightarrow$

permut $L2\ L2' \rightarrow$

permut $(L1 ++ L2) (L1' ++ L2')$.

Proof using.

intros $M1\ M2$. *applies* permut_trans $(L1' ++ L2)$.

{ *clear* $M2$. induction $M1$.

{ *rew_list*. *applies* permut_mid. }

{ *applies* $\times$ permut_trans IHM1_2. } }

{ *clear* $M1$. induction $M2$.

{ *repeat* *rewrite* $\leftarrow$ (app_assoc $L1'$). *applies* permut_mid. }

{ *applies* $\times$ permut_trans IHM2_2. } }

Qed.

Useful corollaries for *cons* and *last*:

Lemma permut_cons : $\forall A (x:A) L1\ L1',$

permut $L1\ L1' \rightarrow$

permut $(x :: L1) (x :: L1')$.

Proof using. intros. *applies* $\times$ permut_app $(x :: \text{nil})\ L1\ (x :: \text{nil})\ L1'$. Qed.

Lemma permut_last : $\forall A (x:A) L1\ L1',$

permut $L1\ L1' \rightarrow$

permut $(L1 \& x) (L1' \& x)$.

Proof using. intros. *applies* $\times$ permut_app $L1\ (x :: \text{nil})\ L1'\ (x :: \text{nil})$. Qed.

Swapping of consecutive elements or moving the first element to the last position yield valid permutations.

Lemma `permut_swap_first_two` : $\forall A (x\ y : A) (L : \text{list } A)$,
`permut` $(x :: y :: L) (y :: x :: L)$.

Proof using.

`intros. applies permut_mid (@nil A) (x :: nil) (y :: nil) L.`

Qed.

Lemma `permut_swap_first_last` : $\forall A (x : A) (L : \text{list } A)$,
`permut` $(x :: L) (L \& x)$.

Proof using.

`intros. applies_eq (» permut_mid (@nil A) (x :: nil) L (@nil A)). rew_list ×.`

Qed.

If a property P holds for all elements of a list L , it also holds for all the elements of any permutation of L .

Lemma `permut_Forall` : $\forall A (P : A \rightarrow \text{Prop}) L L'$,
`Forall` $P L \rightarrow$
`permut` $L L' \rightarrow$
`Forall` $P L'$.

Proof using.

`introv N M. gen N. induction M; intros; auto.`
`{ repeat rewrite Forall_app_eq in *. autos ×. }`

Qed.

The predicate `sorted` L asserts that L is a sorted list of integers.

Inductive `sorted` : `list int` $\rightarrow$ `Prop` :=

| `sorted_nil` :
`sorted` `nil`
| `sorted_one` : $\forall x$,
`sorted` $(x :: \text{nil})$
| `sorted_cons` : $\forall x\ y\ L$,
 $x \leq y \rightarrow$
`sorted` $(y :: L) \rightarrow$
`sorted` $(x :: y :: L)$.

The proposition `list_of_gt` $x\ L$ asserts that all elements in L are greater than x . Symmetrically, the proposition `list_of_le` $x\ L$ asserts that all elements in L is no greater than x .

Definition `list_of_gt` $(x : \text{int}) (L : \text{list int})$: `Prop` :=
`Forall` $(\text{fun } y \Rightarrow y > x) L$.

Definition `list_of_le` $(x : \text{int}) (L : \text{list int})$: `Prop` :=
`Forall` $(\text{fun } y \Rightarrow y \leq x) L$.

Two key lemmas for verifying sorting algorithms are stated below.

First, if L is sorted, then adding an element x no greater than elements in L to the front of L yields a sorted list.

```
Lemma sorted_cons_gt : ∀ x L,
  list_of_gt x L →
  sorted L →
  sorted (x :: L).
```

Proof using.

```
  introv N M. destruct L as [].
  { applys sorted_one. }
  { lets (Hv&_): Forall_cons_inv N. applys× sorted_cons. math. }
```

Qed.

Second, if $L1$ is a sorted list, x is no less than the elements in $L1$, and $x::L2$ is a sorted list, then $L1 ++ x :: L2$ is sorted.

```
Lemma sorted_app_le : ∀ x L1 L2,
  list_of_le x L1 →
  sorted L1 →
  sorted (x :: L2) →
  sorted (L1 ++ x :: L2).
```

Proof using.

```
  introv N M1 M2. induction M1; rew_list in *. { auto. }
  { lets (Hv&_): Forall_cons_inv N. applys× sorted_cons. }
  { lets (&N'): Forall_cons_inv N. applys× sorted_cons. }
```

Qed.

End SortedLists.

48.3.2 Formalization of Arrays of Integer Values

To specify *quicksort* and other functions manipulating lists of integers, it is useful to introduce a conversion function `vals_int`, which converts a list of integers (type `list int`) into a list of values (type `list val`).

In particular, `hseg (vals_int L) p i` describes an array segment containing integer values.

```
Definition vals_int (L:list int) : list val :=
  LibList.map val_int L.
```

To ease the reasoning about `vals_int`, we add the rewriting rules, stated below, to the tactics `rew_list` and `rew_listx`. These tactics perform normalization by means of Rocq's `autorewrite` tactic. In an ideal world, we would use a single tactic `rew_list` to handle all rewriting rules related to lists. Unfortunately, `autorewrite` is undesirably slow when handling a large the data base of rewriting rules. The TLC library therefore relies on a 2-layer design: the tactic `rew_list` covers only the most frequently used rewriting rules, whereas the tactic `rew_listx` is slower but covers all rewriting rules.

```
Lemma vals_int_nil :
```

```

    vals_int nil = nil.
Proof using. intros. unfold vals_int. rew_listx×. Qed.

Lemma vals_int_cons : ∀ x L,
  vals_int (x :: L) = val_int x :: vals_int L.
Proof using. intros. unfold vals_int. rew_listx×. Qed.

Lemma vals_int_app : ∀ L1 L2,
  vals_int (L1 ++ L2) = vals_int L1 ++ vals_int L2.
Proof using. intros. unfold vals_int. rew_listx×. Qed.

Lemma vals_int_last : ∀ x L,
  vals_int (L & x) = vals_int L & val_int x.
Proof using. intros. unfold vals_int. rew_listx×. Qed.

Lemma length_vals_int : ∀ L,
  length (vals_int L) = length L.
Proof using. intros. unfold vals_int. rew_listx×. Qed.

#[export] Hint Rewrite vals_int_nil vals_int_cons vals_int_app
  vals_int_last length_vals_int : rew_list.
#[export] Hint Rewrite vals_int_nil vals_int_cons vals_int_app
  vals_int_last length_vals_int : rew_listx.

```

48.3.3 Functional Correctness of Quicksort

In order to verify *quicksort*, we need to refine the specification of the *pivot* function to include functional correctness properties.

The postcondition of the pivot operation describes the elements in the segment as the list L' . This list decomposes as $L1 ++ x :: L2$, where x corresponds to the pivot. This time, the assertions `list_of_le x L1` and `list_of_gt x L2` are included.

```

Parameter triple_pivot : ∀ n p i L,
  n = length L →
  n ≥ 1 →
  triple (val_pivot p i n)
    (hseg (vals_int L) p i)
    (fun r ⇒ \∃ j, \[r = val_int j] \*
      \∃ x L' L1 L2, hseg (vals_int L') p i \* \[
        permut L L'
        ∧ L' = L1 ++ x :: L2
        ∧ j - i = length L1
        ∧ list_of_le x L1
        ∧ list_of_gt x L2 ] ).

```

Exercise: 5 stars, standard, optional (triple_quicksort) Prove that *quicksort* sorts the array segment targeted by its arguments. Hint: use *xapp triple_pivot*.

Following the standard memory layout of allocated blocks in ML programs, we represent ML-style array of size n as a memory block of size $n+1$. The first cell, called the “header cell”, stores the length of the array. The remaining cells store the elements from the array.

The heap predicate $\text{hheader } n \ p$ describes a cell at location p with contents n . Recall that the definition of hheader is opaque when reasoning about array-using programs, so there is no risk that the programmer attempts to reason about code that modifies header fields.

Definition $\text{hheader } (n:\text{int}) \ (p:\text{loc}) : \text{hprop} :=$
 $p \rightsquigarrow (\text{val\_int } n).$

Because Rocq’s fold operation rarely applies as the user expects, we state a lemma that reformulates the definition of hheader as an equality. Performing a rewrite using this lemma corresponds either to an unfold or to a fold operation, depending on the direction. More generally, reformulating a definition as an equality is strongly recommended for every representation predicate.

Lemma $\text{hheader\_eq} : \forall \ p \ n,$
 $(\text{hheader } n \ p) = (p \rightsquigarrow (\text{val\_int } n)).$

Proof using. auto. Qed.

The predicate $\text{hcell } v \ p \ i$ asserts that the cell at index i from the array at address p stores the value v . We defined this predicate informally by asserting that the cell in memory at address $p+1+i$ stores the value v . Thus, as first approximation, the predicate $\text{hcell } v \ p \ i$ could be defined as $(p + 1 + \text{abs } i) \rightsquigarrow v$. The actual definition also embeds the assertion $i \geq 0$ to guarantee that i refers to a valid index and $p + 1 + \text{abs } i$ computes the expected offset.

Definition $\text{hcell } (v:\text{val}) \ (p:\text{loc}) \ (i:\text{int}) : \text{hprop} :=$
 $((p + 1 + \text{abs } i) \% \text{nat} \rightsquigarrow v) \ \& \ [i \geq 0].$

The following lemma is useful for folding or unfolding the definition.

Lemma $\text{hcell\_eq} : \forall \ v \ p \ i,$
 $(\text{hcell } v \ p \ i) = ((p + 1 + \text{abs } i) \% \text{nat} \rightsquigarrow v) \ \& \ [i \geq 0].$

Proof using. auto. Qed.

This one extracts the property $i \geq 0$ from $\text{hcell } v \ p \ i$.

Lemma $\text{hcell\_nonneg} : \forall \ v \ p \ i,$
 $\text{hcell } v \ p \ i \implies \text{hcell } v \ p \ i \ \& \ [i \geq 0].$

Proof using. $\text{unfold hcell. xsimpl} \times. \text{Qed.}$

48.3.5 Realization of hseg and harray

The heap predicate hseg for array segments can be defined in terms of hcell as suggested earlier.

Fixpoint $\text{hseg } (L:\text{list val}) \ (p:\text{loc}) \ (j:\text{int}) : \text{hprop} :=$
 $\text{match } L \text{ with}$
 $| \text{nil} \Rightarrow []$

```
| x::L' => (hcell x p j) \* (hseg L' p (j+1))
end.
```

The predicate for full arrays, `harray`, can be defined as the pair of an `hheader` predicate describing the header cell and a `hseg` covering the full contents of the array.

```
Definition harray (L:list val) (p:loc) : hprop :=
  hheader (length L) p \* hseg L p 0.
```

The lemma `harray_eq` can be used for folding or unfolding `harray`.

```
Lemma harray_eq : ∀ p L,
  harray L p = \∃ n, \[n = length L] \* hheader n p \* hseg L p 0.
Proof using. unfold harray. xsimpl×. { intros; subst×. } Qed.
```

When proving properties about `hseg L p j`, one frequently faces a mismatch between `hseg L p j1` and `hseg L p j2`, where `j1` and `j2` are provably equal yet do not unify. To avoid cluttering proofs, the following lemma and associated hint are very useful.

```
Lemma hseg_start_eq : ∀ L p j1 j2,
  j1 = j2 →
  hseg L p j1 ==> hseg L p j2.
Proof using. intros. subst×. Qed.
#[local] Hint Extern 1 (hseg ?L ?p ?j1 ==> hseg ?L ?p ?j2) =>
  apply hseg_start_eq; math.
```

We now prove the lemmas `hseg_nil` and `hseg_one` and `hseg_cons`, which we presented and used earlier in the chapter.

```
Lemma hseg_nil : ∀ p j,
  hseg nil p j = \[].
Proof using. auto. Qed.

Lemma hseg_one : ∀ v p j,
  hseg (v::nil) p j = hcell v p j.
Proof using. intros. simpl. xsimpl×. Qed.

Lemma hseg_cons : ∀ v p j L,
  hseg (v::L) p j = hcell v p j \* hseg L p (j+1).
Proof using. intros. simpl. xsimpl×. Qed.
```

Exercise: 3 stars, standard, especially useful (`hseg_app`) Prove the splitting lemma for array segments. Hint: `rew_list` is helpful for simplifying list operations. Recall that `xsimpl` helps proving equalities on `hprop`.

```
Lemma hseg_app : ∀ L1 L2 p j,
  hseg (L1 ++ L2) p j
  = hseg L1 p j \* hseg L2 p (j + length L1).
Proof using. Admitted.
```

□

48.3.6 Focus Lemmas for Array Segments

With a predicate `harray L p` at hand, it may be useful to isolate the cell at an index `i`, that is, to extract the predicate `hcell v p i`, where $0 \leq i < \text{length } L$ and $v = \text{LibList.nth } (\text{abs } i) L$. By “giving back” the predicate `hcell v p i`, we can get back to the original predicate `harray L p`.

```
Parameter harray_focus_read' : ∀ i L p,
  0 ≤ i < length L →
    harray L p
==> let v := LibList.nth (abs i) L in
  (hcell v p i) \* (hcell v p i \-* harray L p).
```

The lemma above, called a “focus” lemma or “borrowing” lemma in Rust’s terminology, only supports reading into the “focused” cell at index `i`; it does not allow modifying the contents of the cell. Indeed, if `hcell v p i` is updated to `hcell w p i`, then the magic wand `hcell v p i \-* harray L p` can no longer be exploited.

Fortunately, it is straightforward to generalize the focus lemma into a form that supports updates. The general focus lemma shown below makes use of a universal quantification over the value `w` that the cell `i` may contain after potential update operations.

```
Parameter harray_focus' : ∀ i L p,
  0 ≤ i < length L →
    harray L p
==> let v := LibList.nth (abs i) L in
  (hcell v p i)
  \* (∀ w, hcell w p i \-* harray (LibList.update (abs i) w L) p).
```

The focus lemmas are critically useful. Without them, we would need to repeat a number of tedious splitting and merging steps.

Exercise: 4 stars, standard, especially useful (hseg_focus) Prove the following focus lemma for array *segments*. Hint: although a proof by induction is possible, a simpler proof can be achieved by exploiting `LibList.list_middle_inv`, `LibList.nth_middle` and `LibList.update_middle`. Also, recall that lemma `Inhab_val` proves `Inhab val`.

```
Lemma hseg_focus_relative : ∀ (k:nat) L p j,
  0 ≤ k < length L →
    hseg L p j
==> let v := LibList.nth k L in
  hcell v p (j+k)
  \* (∀ w, hcell w p (j+k) \-* hseg (LibList.update k w L) p j).
```

Proof using. *Admitted*.

□

Arguments `hseg_focus_relative` : clear *implicit*s.

In the statement above, `k` is an index relative to the start of the segment. The focus lemma can also be expressed in terms of absolute indices. Below, `i` denotes a valid array index within the targeted segment.

```
Lemma hseg_focus : ∀ i L p j,
  0 ≤ i-j < length L →
    hseg L p j
  ==> let v := LibList.nth (abs (i-j)) L in
    hcell v p i
    \* (\∀ w, hcell w p i \-* hseg (LibList.update (abs (i-j)) w L) p j).
```

Proof using.

```
  introv Hk. xchange (hseg_focus_relative (abs (i-j))). { math. }
  math_rewrite (j + abs (i - j) = i). xsimpl.
```

Qed.

Arguments hseg_focus : clear *implicit*s.

We can derive the final focus lemma for `harray` from the one for `hseg`.

```
Lemma harray_focus : ∀ i L p,
  0 ≤ i < length L →
    harray L p
  ==> let v := LibList.nth (abs i) L in
    hcell v p i
    \* (\∀ w, hcell w p i \-* harray (LibList.update (abs i) w L) p).
```

Proof using.

```
  introv Hi. unfold harray. xchange (hseg_focus i). { math. }
  math_rewrite (i - 0 = i). xsimpl. intros w.
  xchange (hforall_specialize w). xsimpl.
  rewrite× LibList.length_update.
```

Qed.

Arguments harray_focus : clear *implicit*s.

Exercise: 3 stars, standard, optional (harray_focus_read) Prove that the “focus-read” lemma is a direct consequence of the general version of the focus lemma `harray_focus`. Hint: use `LibList.update_nth_same`.

```
Lemma harray_focus_read : ∀ i L p,
  0 ≤ i < length L →
    harray L p
  ==> let v := LibList.nth (abs i) L in
    hcell v p i \* (hcell v p i \-* harray L p).
```

Proof using. *Admitted*.

□

Arguments harray_focus_read : clear *implicit*s.

48.4 Optional Material

In this section, we show how to implement array operations in terms of lower-level primitive operations such as block allocation and pointer arithmetic.

48.4.1 Semantics of Pointer Arithmetic

The operation `val_ptr_add p n` applies to a pointer `p` and an integer `n`, and returns the address `p+n`. In other words, it computes the address of the `n`-th cell past the cell at location `p`. We assume `val_ptr_add` to be a primitive of the language.

Parameter `val_ptr_add` : **prim**.

The evaluation rule and the general specification for pointer addition are shown below. They both require `p+n` to be a nonnegative integer.

Parameter `eval_ptr_add` : $\forall p\ n\ s\ Q,$
 $p + n \geq 0 \rightarrow$
 $Q\ (\text{val\_loc}\ (\text{abs}\ (p + n)))\ s \rightarrow$
 $\text{eval}\ s\ (\text{val\_ptr\_add}\ (\text{val\_loc}\ p)\ (\text{val\_int}\ n))\ Q.$

Lemma `triple_ptr_add` : $\forall (p:\text{loc})\ (n:\text{int}),$
 $p + n \geq 0 \rightarrow$
 $\text{triple}\ (\text{val\_ptr\_add}\ p\ n)$
 $\ \ \ \backslash []$
 $\ (\text{fun}\ r \Rightarrow \backslash [r = \text{val\_loc}\ (\text{abs}\ (p + n))]).$

Proof using.

introv R Hs. applys × eval_ptr_add. lets ->: hempty_inv Hs.
applys hpure_intro. auto.

Qed.

For the `math` tactic provided by the TLC library to behave well on goals involving expressions of the form `p+n` involved in calls to `val_ptr_add`, we need a small tweak. We instantiate a hook of the `math` tactic to register `loc` as transparent type for this tactic.

Ltac `is_additional_arith_type T` ::=
`match T with`
`| loc => constr:(true)`
`end.`

For the purpose of this course, we only need to shift pointers by a nonnegative number of cells. We show below a simplified specification featuring the premise $n \geq 0$ instead of $p+n \geq 0$.

Lemma `triple_ptr_add_nonneg` : $\forall (p:\text{loc})\ (n:\text{int}),$
 $n \geq 0 \rightarrow$
 $\text{triple}\ (\text{val\_ptr\_add}\ p\ n)$
 $\ \ \ \backslash []$


```

    (fun r => \[r = val_loc (p + abs n)%nat]).
Proof using.
  introv Hn. applies triple_conseq triple_ptr_add.
  { math. } { xsimpl. } { xsimpl. intros ? ->. fequal. math. }
Qed.
#[export]Hint Resolve triple_ptr_add_nonneg : triple.

```

48.4.2 Semantics of Low-Level Block Allocation

The operation `val_alloc n` allocates a block of `n` consecutive cells. Starting from a state `sa`, it produces a state described as the union of `sb` and `sa`, where `sb` consists of consecutive of `n` consecutive cells. In the evaluation rule shown below, `Fmap.conseq ... p` builds a state with a range of cells starting a location `p` and with contents described by the list `L`. Each of these cells is specified as having the special value `val_uninit` as contents.

Parameter `val_alloc` : `val`.

```

Parameter eval_alloc : ∀ n sa Q,
  n ≥ 0 →
  (∀ p sb,
    sb = Fmap.conseq (LibList.make (abs n) val_uninit) p →
    p ≠ null →
    Fmap.disjoint sa sb →
    Q (val_loc p) (sb \u sa)) →
  eval sa (val_alloc (val_int n)) Q.

```

The specification of `val_alloc` is expressed using the heap predicate `hrange L p` to describe a range of consecutive cells – a heap produced by `Fmap.conseq L p`. The structure of the recursive definition of `hrange` resembles that of `hseg`.

```

Fixpoint hrange (L:list val) (p:loc) : hprop :=
  match L with
  | nil => \[]
  | x::L' => (p ~~> x) \* (hrange L' (p+1)%nat)
  end.

```

Lemma `hrange_intro` : $\forall L p,$
 $(\text{hrange } L p) (\text{Fmap.conseq } L p).$

Proof using.

```

  intros L. induction L as [|L']; intros; rew_listx.
  { applies empty_intro. }
  { simpl. applies hstar_intro.
    { applies × hsingle_intro. }
    { applies IHL. }
    { applies Fmap.disjoint_single_conseq. left. math. } }

```

Qed.

The allocation operation `val_alloc` is then specified as producing a heap described using `hrange` for a list of uninitialized values. Note: in ML, uninitialized values are never exposed to the programmer; `val_alloc` is only meant to be invoked internally by the runtime system.

```
Lemma triple_alloc : ∀ n,
  n ≥ 0 →
  triple (val_alloc n)
    \[]
    (funloc p ⇒ hrange (LibList.make (abs n) val_uninit) p \* \[p ≠ null]).
```

Proof using.

```
introv Hn Hsa. applys× eval_alloc. intros p sb Esb Hp D.
lets ->: hempty_inv Hsa. applys hexists_intro p.
rewrite hstar_hpure_l. split×. rewrite Fmap.union_empty_r.
rewrite hstar_hpure_r. split×. subst sb. applys× hrange_intro.
```

Qed.

```
#[export]Hint Resolve triple_alloc : triple.
```

48.4.3 Low-Level Implementation of Arrays

We are now ready to explain how a runtime system could implement the operations `length`, `get`, `set`, and `make` on arrays, in terms of `val_alloc` and `val_ptr_add` for allocating memory blocks and computing pointer offsets, as well as `val_get` and `val_set` for reading and writing into individual memory cells.

```
Import NOTATIONFORVARIABLES.
```

The operation that obtains the `length` of an array is implemented by reading the integer stored into the header field of that array.

```
Definition val_array_length : val :=
  <{ fun 'p ⇒ val_get 'p }>.
```

The operation that reads into an array cell, written `val_array_get p i`, is implemented as a read at the location `p + 1 + abs i`.

```
Definition val_array_get : val :=
  <{ fun 'p 'i ⇒
    let 'p1 = val_ptr_add 'p 1 in
    let 'q = val_ptr_add 'p1 'i in
    val_get 'q }>.
```

The operation that reads into an array cell, written `val_array_set p i v`, is implemented as a write of `v` at the location `p + 1 + abs i`.

```
Definition val_array_set : val :=
  <{ fun 'p 'i 'v ⇒
    let 'p1 = val_ptr_add 'p 1 in
    let 'q = val_ptr_add 'p1 'i in
```

```
val_set 'q 'v }>.
```

The auxiliary function *fill* is used to implement *make*. The operation *fill* *p i n v* fills the range of the array *p*, starting at index *i*, with *n* copies of the value *v*. The code below provides a naive recursive implementation of *fill*.

Definition *val_array_fill* : **val** :=

```
<{ fix 'f 'p 'i 'n 'v =>
  let 'b = 'n > 0 in
  if 'b then
    val_array_set 'p 'i 'v;
    let 'm = 'n - 1 in
    let 'j = 'i + 1 in
    'f 'p 'j 'm 'v
  end }>.
```

The operation *val_array_make* *n v* creates an array of length *n* filled with copies of the value *v*. It is implemented as the allocation of a block of length *n*+1 at a fresh address *p*, followed by the write of the value *n* in the header, and with a call to *fill* *p 0 n v* for filling the *n* cells of the array *p* with copies of the value *v*.

Definition *val_array_make* : **val** :=

```
<{ fun 'n 'v =>
  let 'm = 'n + 1 in
  let 'p = val_alloc 'm in
  val_set 'p 'n;
  val_array_fill 'p 0 'n 'v;
  'p }>.
```

48.4.4 Verification of Low-Level Operations for Arrays

Now we can prove that the implementations presented above for *length*, *get*, *set*, and *make* indeed satisfy the specifications axiomatized for them earlier in this chapter. For these proofs, we need to set the definition of *hheader* as transparent.

Global Transparent *hheader*.

We also add a hint for *xapp*, to automatically handle the arithmetic preconditions arising from calls to *val_ptr_add* and *val_alloc*.

```
#[local] Hint Extern 1 (- ≥ -) => math : triple.
```

Moreover, we customize the behavior of the “*” suffix that appears in next to Rocq tactics, for this symbol to trigger not a call to *eauto* but instead a call to *eauto* with *maths*.

Local Ltac *auto_star* ::= *eauto* with *maths*.

Here are the proofs for the small-footprint specifications.

Lemma *triple_array_length_hheader* : $\forall n p,$

```
triple (val_array_length p)
  (hheader n p)
  (fun r ⇒ \[r = n] \* hheader n p).
```

Proof using. *xwp*. unfold hheader. *xapp*. *xsimpl*×. Qed.

```
Lemma triple_array_get_hcell : ∀ v p i,
  triple (val_array_get p i)
    (hcell v p i)
    (fun r ⇒ \[r = v] \* hcell v p i).
```

Proof using.

```
xwp. unfold hcell. xpull. intros Hi. xapp. xapp. xapp. xsimpl×.
```

Qed.

```
Lemma triple_array_set_hcell : ∀ p i v w,
  triple (val_array_set p i v)
    (hcell w p i)
    (fun _ ⇒ hcell v p i).
```

Proof using.

```
xwp. unfold hcell. xpull. intros Hi. xapp. xapp. xapp. xsimpl×.
```

Qed.

Then, the proofs of specifications operating on array segments (*hseg*).

```
Lemma triple_array_get_hseg : ∀ L p i j,
  0 ≤ i - j < length L →
  triple (val_array_get p i)
    (hseg L p j)
    (fun r ⇒ \[r = LibList.nth (abs (i-j)) L] \* hseg L p j).
```

Proof using.

```
introv M. xtriple. xchange× (hseg_focus i).
xapp triple_array_get_hcell.
xchange (hforall_specialize (nth (abs (i - j)) L)).
rewrite× update_nth_same. { xsimpl×. }
```

Qed.

```
Lemma triple_array_set_hseg : ∀ L p i j v,
  0 ≤ i - j < length L →
  triple (val_array_set p i v)
    (hseg L p j)
    (fun _ ⇒ hseg (LibList.update (abs (i-j)) v L) p j).
```

Proof using.

```
introv M. xtriple. xchange× (hseg_focus i).
xapp triple_array_set_hcell. xchange (hforall_specialize v).
```

Qed.

```
Lemma hseg_eq_hrange : ∀ L p,
  hseg L p 0 = hrange L (p+1)%nat.
```

Proof using.

```

asserts M: (∀ L (k:nat) p, hseg L p k = hrange L (p+1+k)%nat).
{ intros L. induction L; intros; simpl.
  { auto. }
  { math_rewrite (p + 1 + k + 1 = p + 1 + (k + 1))%nat.
    rewrite ← IHL. unfold hcell. xsimpl.
    { intros -. applies himpl_of_eq; fequals_rec; math. }
    { math. }
    { applies himpl_of_eq; fequals_rec; math. } } }
intros. rewrite (M L 0%nat). fequals. math.

```

Qed.

```

Lemma triple_array_fill : ∀ n L p i v,
  n = length L →
  triple (val_array_fill p i n v)
    (hseg L p i)
    (fun _ ⇒ hseg (LibList.make (abs n) v) p i).

```

Proof using.

```

intros n. induction_wf IH: (downto 0) n.
introv Hn. xwp. xapp. xif; intros C.
{ xapp triple_array_set_hseg. { math. }
  math_rewrite (abs (i - i) = 0%nat).
  destruct L as [|x L']; rew_list in *. { false. math. }
  rew_listx. xchange hseg_cons. xapp. xapp.
  xapp; try math. xchange ← hseg_cons.
  rewrites × (» LibList.make_pos (abs n) v).
  math_rewrite × (abs (n-1) = abs n - 1)%nat. }
{ xval. math_rewrite (n = 0) in *.
  destruct L; rew_listx in *; tryfalse. auto. }

```

Qed.

```

Lemma triple_array_make_hseg : ∀ n v,
  n ≥ 0 →
  triple (val_array_make n v)
    \[]
    (funloc p ⇒ hheader (abs n) p \* hseg (LibList.make (abs n) v) p 0).

```

Proof using.

```

introv Hn. xwp. xapp. xapp. intros q Hq.
math_rewrite (abs (n+1) = S (abs n)). rew_listx. simpl.
xapp. xchange × ← (hheader_eq q). xchange ← hseg_eq_hrange.
xapp triple_array_fill. rew_listx ×. xval. xsimpl ×.
applies himpl_of_eq; fequals_rec; math.

```

Qed.

Finally, the proofs of specifications expressed in terms of `harray`.

```

Lemma triple_array_get :  $\forall L p i,$ 
   $0 \leq i < \text{length } L \rightarrow$ 
  triple (val_array_get  $p i$ )
    (harray  $L p$ )
    (fun  $r \Rightarrow \backslash[r = \text{LibList.nth (abs } i) L] \backslash * \text{harray } L p$ ).
Proof using.
  introv  $M$ . xtriple. unfold harray. xapp triple_array_get_hseg. { math. }
  math_rewrite ( $i - 0 = i$ ). xsimpl $\times$ .
Qed.

Lemma triple_array_set :  $\forall L p i v,$ 
   $0 \leq i < \text{length } L \rightarrow$ 
  triple (val_array_set  $p i v$ )
    (harray  $L p$ )
    (fun  $- \Rightarrow \text{harray (LibList.update (abs } i) v L) p$ ).
Proof using.
  introv  $M$ . xtriple. unfold harray. xapp triple_array_set_hseg. { math. }
  math_rewrite ( $i - 0 = i$ ). rew_listx. xsimpl $\times$ .
Qed.

Lemma triple_array_length :  $\forall L p,$ 
  triple (val_array_length  $p$ )
    (harray  $L p$ )
    (fun  $r \Rightarrow \backslash[r = \text{length } L] \backslash * \text{harray } L p$ ).
Proof using.
  intros. xtriple. unfold harray. xapp triple_array_length_hheader. xsimpl $\times$ .
Qed.

Lemma triple_array_make :  $\forall n v,$ 
   $n \geq 0 \rightarrow$ 
  triple (val_array_make  $n v$ )
     $\backslash []$ 
    (funloc  $p \Rightarrow \text{harray (LibList.make (abs } n) v) p$ ).
Proof using.
  intros. xtriple. unfold harray. xapp triple_array_make_hseg. { math. }
  rew_listx. xsimpl $\times$ .
Qed.
End REALIZATION.

```

48.5 Appendix

48.5.1 Verification of the Pivot Function

Module PIVOT.

Import QUICKSORT.

Local Ltac *auto_star* ::= eauto with *maths*.

For completeness, we include the formal verification of the *pivot* function. The proof is unfortunately rather cluttered with reasoning about the *vals_int* conversion function. The need for it stems from the fact that we are reasoning on untyped code. The actual CFML tool provides reasoning rule for well-typed code, and is thereby avoids all this kind of clutter.

We consider a simple, unoptimized implementation of the *pivot* function. This implementation is recursive and performs a series of *swap* operations.

OCaml:

```
let swap = fun p j1 j2 -> let x1 = p.(j1) in let x2 = p.(j2) in p.(j1) <- x2; p.(j2) <- x1
let pivot p i n = if n < 2 then i else begin let j = i+1 in let x = p.(i) in let y = p.(j) in if
y <= x then begin swap p i j; pivot p j (n-1); end else begin swap p j i; pivot p i (n-1) end
```

Definition *val_array_swap* : **val** :=

```
<{ fun 'p 'j1 'j2 =>
  let 'x1 = val_array_get 'p 'j1 in
  let 'x2 = val_array_get 'p 'j2 in
  val_array_set 'p 'j1 'x2;
  val_array_set 'p 'j2 'x1 }>.
```

Definition *val_pivot* : **val** :=

```
<{ fix 'f 'p 'i 'n =>
  let 'b = 'n < 2 in
  if 'b then
    'i
  else
    let 'x = val_array_get 'p 'i in
    let 'j = 'i + 1 in
    let 'y = val_array_get 'p 'j in
    let 'm = 'n - 1 in
    let 'c = 'y <= 'x in
    if 'c then
      val_array_swap 'p 'i 'j;
      'f 'p 'j 'm
    else
      let 'k = 'i + 'm in
      val_array_swap 'p 'j 'k;
      'f 'p 'i 'm }>.
```

We first state a specialized version of *array_get* for array segments storing integer values.

Lemma *triple_array_get_hseg_vals_int* : $\forall (L:\text{list int})\ p\ i\ j,$

$0 \leq i - j < \text{length } L \rightarrow$

$\text{triple } (\text{val\_array\_get } p\ i)$
 $(\text{hseg } (\text{vals\_int } L)\ p\ j)$

```

    (fun  $r \Rightarrow \backslash[r = \text{LibList.nth} (\text{abs } (i-j)) \text{ } L] \backslash * \text{hseg } (\text{vals\_int } L) \text{ } p \text{ } j).$ 
Proof using.
  introv  $M$ . xtriple. xapp triple_array_get_hseg. { rew_list. math. }
  xsimpl. unfolds vals_int. rewrite nth_map. fequals. math.
Qed.

```

Next, we verify the *swap* function, which permutes elements at two indices inside a given array segment.

```

Lemma triple_array_swap_seg :  $\forall p \text{ } i \text{ } j1 \text{ } j2 \text{ } (L:\text{list val}),$ 
   $0 \leq j1 - i < \text{length } L \rightarrow$ 
   $0 \leq j2 - i < \text{length } L \rightarrow$ 
  triple (val_array_swap  $p \text{ } j1 \text{ } j2$ )
    (hseg  $L \text{ } p \text{ } i$ )
    (fun  $_ \Rightarrow$ 
      hseg (LibList.update (abs (j2-i)) (LibList.nth (abs (j1-i))  $L$ )
        (LibList.update (abs (j1-i)) (LibList.nth (abs (j2-i))  $L$ )  $L$ ))  $p \text{ } i$ ).

```

```

Proof using.
  introv  $Hj1 \text{ } Hj2$ . xwp.
  xapp triple_array_get_hseg. { math. }
  xapp triple_array_get_hseg. { math. }
  xapp triple_array_set_hseg. { math. }
  xapp triple_array_set_hseg. { rew_listx. math. } xsimpl.
Qed.

```

We derive a specification for *swap* specialized for the case where the elements appear in array segments.

```

Lemma triple_array_swap_seg_lists :  $\forall L1 \text{ } L2 \text{ } L3 \text{ } x \text{ } y \text{ } p \text{ } i \text{ } j1 \text{ } j2,$ 
   $j1 = i + \text{length } L1 \rightarrow$ 
   $j2 = i + \text{length } L1 + 1 + \text{length } L2 \rightarrow$ 
  triple (val_array_swap  $p \text{ } j1 \text{ } j2$ )
    (hseg ( $L1 ++ x :: L2 ++ y :: L3$ )  $p \text{ } i$ )
    (fun  $_ \Rightarrow \text{hseg } (L1 ++ y :: L2 ++ x :: L3) \text{ } p \text{ } i$ ).

```

```

Proof using.
  hint Inhab_val. introv  $H1 \text{ } H2$ . xapp triple_array_swap_seg; rew_list $\times$ .
  applies himpl_of_eq. fequal.
  math_rewrite (abs (j1 - i) = length  $L1$ ).
  math_rewrite (abs (j2 - i) = (length  $L1 + 1 + \text{length } L2$ )%nat).
  rewrite nth_middle; [|rew_list*|.
  asserts_rewrite ( $L1 ++ x :: L2 ++ y :: L3 = (L1 ++ x :: L2) ++ y :: L3$ ).
  { rew_list $\times$ . }
  rewrite nth_middle; [|rew_list*|. rew_list.
  rewrite $\times$  update_middle.
  asserts_rewrite ( $L1 \& y ++ L2 ++ y :: L3 = (L1 \& y ++ L2) ++ y :: L3$ ).

```



```

{ rew_list × . }
rewrite × update_middle; rew_list ×.
Qed.

```

We also derive a specification for *swap* specialized for the case where it permutes an element with itself.

```

Lemma triple_array_swap_seg_self : ∀ L p i j1 j2,
  0 ≤ j1 - i < length L →
  j2 = j1 →
  triple (val_array_swap p j1 j2)
    (hseg L p i)
    (fun _ ⇒ hseg L p i).

```

Proof using.

```

  introv H1 →. xapp × triple_array_swap_seg.
  do 2 rewrite × LibList.update_nth_same.
Qed.

```

We are now ready to prove the *pivot* function. For the recursion, we need to furthermore assert in the postcondition that the pivot value *x* is, at each recursive call, located at the head of the segment on which the recursive call is performed. This property is captured by the additional assertion *x* = *LibList.nth* 0%*nat* *L*. This assertion was not needed for the verification of *quicksort*.

```

Lemma triple_pivot' : ∀ n p i L,
  n = length L →
  n ≥ 1 →
  triple (val_pivot p i n)
    (hseg (vals_int L) p i)
    (fun r ⇒ \∃ j, \[r = val_int j] \*
      \∃ x L' L1 L2, hseg (vals_int L') p i \* \[
        permut L L'
        ∧ x = LibList.nth 0%nat L
        ∧ L' = L1 ++ x :: L2
        ∧ j - i = length L1
        ∧ list_of_le x L1
        ∧ list_of_gt x L2 ] ).

```

Proof using.

```

  intros n. induction_wf IH: (downto 0) n; introv HL Hn. xwp.
  xapp. xif; intros C.
  { xval. xsimpl (⟩ i (LibList.nth 0%nat L) L (@nil int) (@nil int)).
    { auto. }
    { destruct L as [|x []]; rew_list in *; try (false; math). splits.
      { applies permut_refl. }
      { rew_list × . }
    }
  }

```

```

{ rew_listx ×. }
{ math. }
{ applys Forall_nil. }
{ applys Forall_nil. } } }
{ xapp × triple_array_get_hseg_vals_int.
  xapp. xapp × triple_array_get_hseg_vals_int.
  xapp. xapp.
  math_rewrite (abs (i - i) = 0%nat).
  math_rewrite (abs (i + 1 - i) = 1%nat).
  lets × (x & L' &->): length_pos_inv_cons L. rew_list in *.
  lets × (y & L'' &->): length_pos_inv_cons L'. rew_listx.
  xif; intros C'.
  { xapp (» triple_array_swap_seg_lists
    (@nil val) (@nil val) (vals_int L'')); rew_list ×.
    xchange hseg_cons. rew_list in *. xapp (» IH (x :: L'')); rew_list ×.
    intros j' x' L' L1' L2' (HP & Hx & HE & Hji & Hle & Hgt). rew_listx in Hx. subst x'.
    xchange ← hseg_cons. xsimpl × (» j' x (y :: L') (y :: L1') L2'). splits.
    { subst L'. applys permut_trans. applys permut_swap_first_two.
      applys × permut_cons. }
    { subst L'. rew_list ×. }
    { subst L'. rew_list ×. }
    { rew_list ×. }
    { applys × Forall_cons. }
    { auto. } } }
  { xapp. tests Cend: (L'' = nil).
    { rew_listx in *. xapp triple_array_swap_seg_self; rew_list ×.
      change (val_int x :: val_int y :: nil) with ((val_int x :: nil) & val_int y).
      xchange hseg_last. xapp (» IH (x :: nil)); rew_list ×.
      intros j' L' x' L1' L2' (HP & Hx & HE & Hji & Hle & Hgt).
      lets HEQ: permut_length HP. subst L'. subst x'. rew_list in *.
      asserts ->: (L1' = nil). applys × length_zero_inv.
      asserts ->: (L2' = nil). applys × length_zero_inv.
      xchange hseg_last_r. { rew_listx. math. } rew_listx.
      xsimpl × (» j' x (x :: y :: nil) (@nil int) (y :: nil)); rew_listx. splits ×.
      { applys permut_refl. }
      { applys × Forall_cons. } }
    { lets × (z & L''' &->): list_neq_nil_inv_last L''. rew_listx in *.
      xapp (» triple_array_swap_seg_lists (val_int x :: nil)
        (vals_int L''')) (@nil val) y z); rew_list ×.
      replace (val_int x :: val_int z :: vals_int L''' & val_int y)
        with ((val_int x :: val_int z :: vals_int L''') & val_int y);
        [|rew_list*].

```

```

xchange hseg_last.
xapp ( $\gg IH (x :: z :: L''')$ ); rew_list $\times$ .
intros  $j' x' L' L1' L2'$  ( $HP \& Hx \& HE \& Hji \& Hle \& Hgt$ ).
lets  $HEQ$ : permut_length  $HP$ . rew_listx in *. subst  $x'$ .
xchange hseg_last_r. { rew_list $\times$ . }
xsimpl $\times$  ( $\gg j' x (L1' ++ x :: L2' \& y) L1' (L2' \& y)$ ). splits $\times$ .
{ subst  $L'$ . applys permut_trans. applys permut_swap_first_two.
  applys permut_trans. applys permut_swap_first_last.
  asserts_rewrite ( $L1' ++ x :: L2' \& y = (L1' ++ x :: L2') \& y$ ).
  { rew_list $\times$ . }
  applys permut_last. applys permut_trans  $HP$ . applys permut_cons.
  applys permut_sym. applys permut_swap_first_last. }
{ subst  $L'$ . applys $\times$  Forall_last. }
{ subst  $L'$ . rew_listx $\times$ . } } } }

```

Qed.

As mentioned earlier, the proof is longer than we would like it to be. (It would be much simpler in the CFML tool, where invariants need not mention the `vals_int` conversion function.)

End PIVOT.

Chapter 49

Library `SLF.Records`

49.1 Records: Reasoning about Records

```
Set Implicit Arguments.
From SLF Require Import LibSepReference LibSepTLCbuffer.
From SLF Require Import Arrays. Import REALIZATION.

Implicit Types P : Prop.
Implicit Types H : hprop.
Implicit Types Q : val → hprop.
Implicit Type p q : loc.
Implicit Type k : nat.
Implicit Type i : int.
Implicit Type n z : nat.
Implicit Type v : val.
Implicit Type L : list val.
```

49.2 First Pass

This chapter explains how to specify operations on ML-style records in Separation Logic. In ML-like languages, a record is an allocated object whose header field stores the number of fields of the record. Concretely, each field is stored at a specific offset relative to the header. For example, consider a mutable list cell allocated at location `p`. It is represented in memory as:

- a header at location `p`, storing the number of fields, that is, the value 2;
- a cell at location `p+1`, storing the contents of the head field,
- a cell at location `p+2`, storing the contents of the tail field.

Thus, at a low level, such a record can be represented by the heap predicate: $(\text{hheader } 2 \text{ } p) \setminus * ((p+1) \sim\sim > x) \setminus * ((p+2) \sim\sim > q)$. To avoid exposing pointer arithmetic to the end user, we introduce the predicate $\text{hfield } p \text{ } k \text{ } v$, written $p'.k \sim\sim > v$, to describe the contents of the field named k of the record at location p .

The chapter starts by presenting the representation predicate hfield , as well as a higher-level predicate named hrecord for describing all fields at once. The second part of the chapter presents the specification of access and allocation operations. The last part explains how to implement and verify these operations using block allocation and pointer arithmetic.

49.2.1 Representation of Individual Record Fields

Let us assume that each field name corresponds to a nonnegative offset. Let field denote the type of field names, an alias for nat .

Definition $\text{field} : \text{Type} := \text{nat}$.

The predicate $\text{hfield } p \text{ } k \text{ } v$, written $p'.k \sim\sim > v$, asserts that the field k of the record p stores the value v . This predicate is presented as an abstract predicate to the end user. Internally, it may be defined using pointer arithmetic.

Definition $\text{hfield } (p:\text{loc}) (k:\text{field}) (v:\text{val}) : \text{hprop} :=$
 $(p+1+k)\% \text{nat} \sim\sim > v$.

Global Opaque hfield .

Notation $"p'.k \sim\sim > v" := (\text{hfield } p \text{ } k \text{ } v)$
 (at level 32, k at level 0, no associativity,
 $\text{format } "p'.k \sim\sim > v"$).

Writing $(\text{hheader } 2 \text{ } p) \setminus * (p'.\text{head} \sim\sim > x) \setminus * (p'.\text{tail} \sim\sim > q)$ to describe a list cell record is fairly verbose and cumbersome to manipulate. To improve the situation, we introduce a generic representation predicate for records. This predicate allows us to describe the same list cell much more concisely as $p \sim\sim > \{ \text{head} := x; \text{tail} := q \}$.

It what follows, we show how to implement this notation by introducing the heap predicates hfields and hrecords . These predicates will be used for stating both small-footprint and large-footprint specifications for record operations.

49.2.2 Representation of Sets of Fields

A record field is described as the pair of a field and a value stored in this field.

Definition $\text{hrecord\_field} : \text{Type} := (\text{field} \times \text{val})$.

A record is described as a list of fields.

Definition $\text{hrecord\_fields} : \text{Type} := \text{list } \text{hrecord\_field}$.

We let the meta-variable kvs range over such lists.

Implicit Types $kvs : \text{hrecord\_fields}$.

The heap predicate `hfields kvs p` asserts that, at location `p`, one finds the representation of the fields described by the list `kvs`. The predicate `hfields kvs p` is defined recursively on the list `kvs`. If `kvs` is empty, the predicate describes the empty heap predicate. Otherwise, `kvs` decomposes as a head (k, v) and a tail `kvs'`. The predicate $p.k \rightsquigarrow v$ describes the field at offset `k`, with contents `v`. The predicate `hfields kvs' p` describes the remaining fields.

```
Fixpoint hfields (kvs:hrecord_fields) (p:loc) : hprop :=
  match kvs with
  | nil => \[]
  | (k,v)::kvs' => (p'.k ~~~> v) \* (hfields kvs' p)
  end.
```

A list cell with head field `x` and tail field `q` is represented by the list $(\text{head}, x) :: (\text{tail}, q) :: \text{nil}$. To support the syntax `{ head := x; tail := q }`, we introduce the following notation. For technical reasons associated with coercions, we need to separately define notation for parsing, where a cast to the type `val` is included, from notation for printing, where such a cast is not included.

```
Notation "{ k1 := v1 }" :=
  ((k1, (v1:val))::nil)
  (at level 0, k1 at level 0, only parsing).

Notation "{ k1 := v1 ; k2 := v2 }" :=
  ((k1, (v1:val))::(k2, (v2:val))::nil)
  (at level 0, k1, k2 at level 0, only parsing).

Notation "{ k1 := v1 ; k2 := v2 ; k3 := v3 }" :=
  ((k1, (v1:val))::(k2, (v2:val))::(k3, (v3:val))::nil)
  (at level 0, k1, k2, k3 at level 0, only parsing).

Notation "{ k1 := v1 }" :=
  ((k1, v1)::nil)
  (at level 0, k1 at level 0, only printing).

Notation "{ k1 := v1 ; k2 := v2 }" :=
  ((k1, v1)::(k2, v2)::nil)
  (at level 0, k1, k2 at level 0, only printing).

Notation "{ k1 := v1 ; k2 := v2 ; k3 := v3 }" :=
  ((k1, v1)::(k2, v2)::(k3, v3)::nil)
  (at level 0, k1, k2, k3 at level 0, only printing).

Open Scope val_scope.
```

49.2.3 Representation of Records

The heap predicate `hrecord kvs p` describes a record by covering not just all the fields of the record but also the header. The cells are described by `hfields kvs p`, and the header is

described by `hheader n p`, where `n` should be such that the keys in the list `kvs` are between 0 inclusive and `n` exclusive.

A permissive definition of `hrecord kvs p` would allow the fields from the list `kvs` to be permuted arbitrarily. However, to avoid complications related to permutations, we make in this course the simplifying assumptions that fields are always listed in the order of their offsets.

The auxiliary predicate `maps_all_fields z kvs` asserts that the keys from the association list `kvs` correspond exactly to the sequence made of the first `n` natural numbers, that is, 0; 1; ...; `n-1`.

Definition `maps_all_fields (n:nat) (kvs:hrecord_fields) : Prop :=`

`LibList.map fst kvs = nat_seq 0 n.`

The predicate `hrecord kvs p` exploits `maps_all_fields n kvs` to relate the value `n` stored in the header with the association list `kvs` that describes the contents of the fields.

Definition `hrecord (kvs:hrecord_fields) (p:loc) : hprop :=`

`∃ n, hheader n p \* hfields kvs p \* \[maps_all_fields n kvs].`

We introduce the notation `p ~~~> kvs` for `hrecord kvs p`, allowing to write, e.g., `p ~~~> '{ head := x; tail := q }.`

Notation `"p ~~~>' kvs" := (hrecord kvs p)`
(at level 32).

The following lemma may be used to convert a concrete `hrecord` predicate into a `hheader` and a conjunction of `hfield` predicates. In the statement below, `LibListExec.length` is a variant of `LibList.length` that computes in Rocq using `simpl` or `reflexivity`.

Lemma `hrecord_elim : ∀ p kvs,`

`hrecord kvs p ==> hheader (LibListExec.length kvs) p \* hfields kvs p.`

Proof using.

`intros. unfold hrecord. xpull. intros z Hz. xsimpl. xsimpl.`
`rewrite LibListExec.length_eq. applys himpl_of_eq. fequal. gen Hz.`
`unfold maps_all_fields. generalize 0%nat. gen z.`
`induction kvs; intros; destruct z as [|z']; rew_listx in *; tryfalse.`
`{ math. } { inverts Hz. forwards*: IHkvs H1. math. }`

Qed.

For example, let us show how to convert from `p ~~~> '{ head := x ; tail := q }` to the conjunction of `hheader 2 p` and `p'.head ~~~> x` and `p'.tail ~~~> q`.

Definition `head : field := 0%nat.`

Definition `tail : field := 1%nat.`

Lemma `demo_hrecord_intro_elim : ∀ p x q,`

`p ~~~> '{ head := x ; tail := q } ==> \[[].`

Proof using. `intros. xchange hrecord_elim; simpl. Abort.`

49.2.4 Reasoning about Reads from Record Fields

Declare Scope *trm_scope_ext*.

The read operation is described by an expression of the form `val_get_field k p`, where `k` denotes a field name and `p` denotes the location of a record.

Observe that `k` is *not* a program value of type `val`, but rather a name for a natural number.

The expression `val_get_field` has type `field → val`, and for a given field `k` the expression `val_get_field k` is a value, which may be applied in the program syntax to a pointer `p`.

Parameter `val_get_field` : `field → val`.

The read operation `val_get_field k p` is written `p'.k`.

Notation `"t1 '.' k" :=`

```
(val_get_field k t1)
(in custom trm at level 56, k at level 0, format "t1 '.' k" )
: trm_scope_ext.
```

The operation `val_get_field k p` can be specified at three levels.

First, its small-footprint specification operates at the level of a single field, described by `p'.k ~~> v`. The specification is very similar to that of `val_get`. The return value is exactly `v`.

Parameter `triple_get_field` : $\forall p k v,$
`triple (val_get_field k p)`
`(p'.k ~~> v)`
`(fun r ⇒ \[r = v] \* (p'.k ~~> v)).`

Second, the read operation `val_get_field` can be specified with respect to a list of fields, described in the form `hfields kvs p`. To that end, we introduce a function called `hfields_lookup` for extracting the value `v` associated with a field `k` in a list of record fields `kvs`. Note that the operation `hfields_lookup k kvs` returns an `option val`, because we cannot presume that the field `k` occurs in `kvs`.

```
Fixpoint hfields_lookup (k:field) (kvs:hrecord_fields) : option val :=
  match kvs with
  | nil ⇒ None
  | (ki,vi)::kvs' ⇒ if Nat.eq_dec k ki
                     then Some vi
                     else hfields_lookup k kvs'
end.
```

Under the assumption that `hfields_lookup k kvs` returns `Some v`, the read operation `val_get_field k p` is specified to return `v`.

Parameter `triple_get_field_hfields` : $\forall kvs p k v,$
`hfields_lookup k kvs = Some v →`
`triple (val_get_field k p)`


```

(hfields kvs p)
(fun r ⇒ \[r = v] \* hfields kvs p).

```

Third, the read operation `val_get_field` can be specified with respect to the predicate `hrecord kvs p`, describing the full record, including its header. The corresponding specification shown below follows a similar pattern as the specification of `hfield`.

```

Parameter triple_get_field_hrecord : ∀ kvs p k v,
  hfields_lookup k kvs = Some v →
  triple (val_get_field k p)
    (hrecord kvs p)
    (fun r ⇒ \[r = v] \* hrecord kvs p).

```

49.2.5 Reasoning about Writes To Record Fields

The write operation is described by an expression of the form `val_set_field k p v`, where *k* denotes a field name, and where *p* denotes the location of a record, and where *v* is the new value for the field.

```

Parameter val_set_field : field → val.

```

The operation `val_set_field k p v` is abbreviated `Set p'.k := v`.

```

Notation "t1 '.' k ':=' t2" :=
  (val_set_field k t1 t2)
  (in custom trm at level 56, k at level 0, format "t1 '.' k ':=' t2")
  : trm_scope_ext.

```

As for the read operation, the write operation can be specified at three levels. First, it may be specified at the level of an individual field.

```

Parameter triple_set_field : ∀ v p k v',
  triple (val_set_field k p v)
    (p'.k ~~> v')
    (fun _ ⇒ p'.k ~~> v').

```

Alternatively, it may be specified in terms of `hfields` or `hrecord`, using an auxiliary function called `hrecord_update`. This function computes an updated list of fields to reflect the action of a write operation.

Concretely, `hrecord_update k w kvs` replaces the contents of the field named *k* with the value *w*. It returns a description *kvs'* of the record fields, provided the update operation succeeded, i.e., provided that the field *k* on which the update is to be performed actually occurs in the list *kvs*.

```

Fixpoint hfields_update (k:field) (v:val) (kvs:hrecord_fields)
  : option hrecord_fields :=
  match kvs with
  | nil ⇒ None
  | (ki, vi) :: kvs' ⇒ if Nat.eq_dec k ki

```

```

      then Some ((k, v) :: kvs')
    else match hfields_update k v kvs' with
      | None ⇒ None
      | Some LR ⇒ Some ((ki, vi) :: LR)
    end
  end.

```

The specification in terms of `hfields` is as follows.

Parameter `triple_set_field_hfields` : $\forall kvs\ kvs'\ k\ p\ v,$
`hfields_update k v kvs = Some kvs' →`
`triple (val_set_field k p v)`
`(hfields kvs p)`
`(fun _ ⇒ hfields kvs' p).`

The specification in terms of `hrecord` follows a similar pattern.

Parameter `triple_set_field_hrecord` : $\forall kvs\ kvs'\ k\ p\ v,$
`hfields_update k v kvs = Some kvs' →`
`triple (val_set_field k p v)`
`(hrecord kvs p)`
`(fun _ ⇒ hrecord kvs' p).`

49.2.6 Reasoning about Record Allocation with Initializers

In ML, records may only be allocated by providing values to initialize every field. This allocation operation can be defined in an arity-generic way. However, to avoid technicalities, we here present only its specialization for arity 2. For example, the notation ‘{ `head` := `x`; `tail` := `q` }’ stands for the term `(val_new_hrecord_2 head tail) x q`.

Here, `val_new_hrecord_2` is a Rocq function that expects two names as arguments and produces a value. For example, the expression `val_new_hrecord_2 head tail` is a value that may be applied to two arguments in the program syntax.

Parameter `val_new_hrecord_2` : `field → field → val`.

Notation “{ `k1` := `v1` ; `k2` := `v2` }” :=
`(val_new_hrecord_2 k1 k2 v1 v2)`
`(in custom trm at level 65,`
`k1, k2 at level 0,`
`v1, v2 at level 65).`

The record allocation operation ‘{ `k1` := `v1` ; `k2` := `v2` }’ is specified as follows. The premises `k1 = 0` and `k2 = 1` enforce the field names to be provided in increasing order, starting from zero. The postcondition describes a record at address `p` using the predicate `p ~~~>` ‘{ `k1` := `v1` ; `k2` := `v2` }, i.e., `hrecord ((k1,v1)::(k2,v2)::nil) p`.

Parameter `triple_new_hrecord_2` : $\forall (k1\ k2:\text{field})\ (v1\ v2:\text{val}),$
`k1 = 0%nat →`

```

k2 = 1%nat →
triple <{ ‘{ k1 := v1 ; k2 := v2 } }>
  \ []
(funloc p ⇒ p ~~~> ‘{ k1 := v1 ; k2 := v2 } ).

```

For example, the operation `mcons x q` allocates a list cell with head value `x` and tail pointer `q`.

Definition `mcons` : `val` :=
`val_new_hrecord_2` head tail.

Lemma `triple_mcons` : $\forall (x\ q:\text{val})$,
`triple (mcons x q)`
`\ []`
`(funloc p ⇒ p ~~~> ‘{ head := x ; tail := q } )`.

Proof using. `intros. applys × triple_new_hrecord_2. Qed.`

This completes the presentation of the formalization of the mechanisms for reasoning about records. These mechanisms were illustrated throughout the chapter [Repr](#).

49.3 Optional Material

```

Module REALIZATION.
Import PROGRAMSYNTAX.
Open Scope trm_scope_ext.

```

In this section, we explain how `val_get_field`, `val_set_field`, and `val_new_hrecord_2` can be realized in terms of block allocation and pointer arithmetic, following the same approach as in the previous chapter on arrays. In particular, we use a similar proof set up.

Global Transparent `hfield`.

```
#[local] Hint Extern 1 ( _ ≥ _ ) ⇒ math : triple.
```

```

Ltac is_additional_arith_type T ::=
  match T with
  | loc ⇒ constr:(true)
  | field ⇒ constr:(true)
  end.

```

49.3.1 Implementation of Record Accesses using Pointer Arithmetic

The get operation on a field, written `p·.k`, is encoded as `val_get (p+1+k)`. In the definition shown below, the braces around `nat_to_Z k` are used to embed the Rocq natural number `k` into the code, in the form of an integer represented in the grammar of terms using the implicit coercion `val_int`.

Definition `val_get_field` (`k:field`) : `val` :=

```

<{ fun 'p =>
  let 'p1 = val_ptr_add 'p 1 in
  let 'q = val_ptr_add 'p1 {nat_to_Z k} in
  val_get 'q }>.

```

Notation "t1 '`.`' f" :=
 (val_get_field f t1)
 (in custom trm at level 56, f at level 0, format "t1 '`.`' f").

#[export]Hint Resolve triple_ptr_add_nonneg : triple.

Lemma triple_get_field : $\forall p k v$,
 triple ((val_get_field k) p)
 (p'.k $\sim\sim$ > v)
 (fun r $\Rightarrow$ \[r = v] * (p'.k $\sim\sim$ > v)).

Proof using.

unfold hfield. xwp. xapp. xapp. unfolds field.
 math_rewrite (p + abs 1 + abs k = p+1+k)%nat. xapp. xsimpl.

Qed.

The set operation on a field, written **Set** p'.k := v, is encoded as **val_set** (p+1+k) v.

Definition val_set_field (k:field) : **val** :=

```

<{ fun 'p 'v =>
  let 'p1 = val_ptr_add 'p 1 in
  let 'q = val_ptr_add 'p1 {nat_to_Z k} in
  val_set 'q 'v }>.

```

Lemma triple_set_field : $\forall v1 p k v2$,
 triple ((val_set_field k) p v2)
 (p'.k $\sim\sim$ > v1)
 (fun _ $\Rightarrow$ p'.k $\sim\sim$ > v2).

Proof using.

unfold hfield. intros. xwp. xapp. xapp.
 math_rewrite (p + abs 1 + abs k = p+1+k)%nat. xapp. xsimpl.

Qed.

Notation "t1 '`.`' f ':= ' t2" :=
 (val_set_field f t1 t2)
 (in custom trm at level 56, f at level 0, format "t1 '`.`' f ':= ' t2").

49.3.2 Specification of Record Accesses w.r.t. **hfields**

The specifications **triple_get_field** and **triple_set_field** proved in the previous section correspond to small-footprint specifications, with preconditions mentioning a single field. Corollaries to these specifications can be established with preconditions mentioning a list of fields, with preconditions expressed using **hfields**.

Lemma triple_get_field_hfields : $\forall kvs\ p\ k\ v$,
 hfields_lookup $k\ kvs = \text{Some } v \rightarrow$
 triple (val_get_field $k\ p$)
 (hfields $kvs\ p$)
 (fun $r \Rightarrow \backslash[r = v] \backslash * \text{hfields } kvs\ p$).

Proof using.

```
intros L. induction L as [| [ki vi] L']; simpl; introv E.
{ inverts E. }
{ case_if.
  { inverts E. subst ki. applies triple_conseq_frame.
    { applies triple_get_field. } { xsimpl. } { xsimpl×. } }
  { applies triple_conseq_frame.
    { applies IHL' E. }
    { xsimpl. }
    { xsimpl×. } } } }
```

Qed.

Lemma triple_set_field_hfields : $\forall kvs\ kvs'\ k\ p\ v$,
 hfields_update $k\ v\ kvs = \text{Some } kvs' \rightarrow$
 triple (val_set_field $k\ p\ v$)
 (hfields $kvs\ p$)
 (fun $- \Rightarrow \text{hfields } kvs'\ p$).

Proof using.

```
intros kvs. induction kvs as [| [ki vi] kvs']; simpl; introv E.
{ inverts E. }
{ case_if.
  { inverts E. subst ki. applies triple_conseq_frame.
    { applies triple_set_field. } { xsimpl. } { xsimpl×. } }
  { cases (hfields_update  $k\ v\ kvs'$ ) as C2; tryfalse. inverts E.
    applies triple_conseq_frame. { applies IHkvs' C2. }
    { xsimpl. } { simpl. xsimpl×. } } }
```

Qed.

49.3.3 Specification of Record Accesses w.r.t. **hrecord**

This section further refines the specification for `val_get_field` and `val_set_field`, into specification expressed using `hrecord`. For `val_get_field`, the proof is straightforward.

Lemma triple_get_field_hrecord : $\forall kvs\ p\ k\ v$,
 hfields_lookup $k\ kvs = \text{Some } v \rightarrow$
 triple (val_get_field $k\ p$)
 (hrecord $kvs\ p$)
 (fun $r \Rightarrow \backslash[r = v] \backslash * \text{hrecord } kvs\ p$).

Proof using.

```

  introv M. unfold hrecord. xtriple. xpull. intros z Hz.
  xapp (» triple_get_field_hfields M). xsimpl×.

```

Qed.

For `val_set_field`, however, we need an auxiliary lemma about `hfields_update`, showing that the update operation preserves the names of the fields. This lemma is established in two steps.

Lemma `hfields_update_preserves_fields` : $\forall kvs\ kvs'\ k\ v$,
`hfields_update k v kvs = Some kvs' →`
`LibList.map fst kvs' = LibList.map fst kvs.`

Proof using.

```

  intros kvs. induction kvs as [[ki vi] kvs1]; simpl; introv E.
  { introv _ H. inverts H. }
  { case_if.
    { inverts E. rew_listx×. subst. fequals. }
    { cases (hfields_update k v kvs1).
      { inverts E. rew_listx. fequals×. }
      { inverts E. } } }

```

Qed.

Lemma `hfields_update_preserves_maps_all_fields` : $\forall kvs\ kvs'\ z\ k\ v$,
`hfields_update k v kvs = Some kvs' →`
`maps_all_fields z kvs = maps_all_fields z kvs'.`

Proof using.

```

  introv M. unfold maps_all_fields. extens.
  rewrites× (» hfields_update_preserves_fields M).

```

Qed.

We are then ready to derive the specification for `val_set_field`.

Lemma `triple_set_field_hrecord` : $\forall kvs\ kvs'\ k\ p\ v$,
`hfields_update k v kvs = Some kvs' →`
`triple (val_set_field k p v)`
`(hrecord kvs p)`
`(fun _ ⇒ hrecord kvs' p).`

Proof using.

```

  introv M. unfold hrecord. xtriple. xpull. intros z Hz.
  xapp (» triple_set_field_hfields M). xsimpl.
  rewrites× ← (» hfields_update_preserves_maps_all_fields z M).

```

Qed.

49.3.4 Implementation of Record Allocation without Initialization

Recall that a record with `n` fields is represented as a memory block of size `n+1`. The first memory cell in the block corresponds to the header, and stores the number of fields, that is,

the value n . The n other cells store the value associated with each of the fields.

The operation that allocates and initializes a fresh record with specific values can be implemented in two stages. The first function, named `val_alloc_hrecord`, takes a list of n field names, and allocates a record with n fields, with contents specified as “uninitialized”. The second function, named `val_new_hrecord`, takes n field names and n values. It invokes the first function to allocate the record, then performs one write per field to store the provided values into the corresponding memory cells.

The present section focuses on the first part, namely the allocation without initialization. The Rocq value `val_alloc_hrecord  $ks$` depends on a list of field names. This value corresponds to a function that, when applied to a unit argument, allocates a record with `length  $ks$` fields. The operation `(val_alloc_hrecord  $ks$ ) ()` is only specified when ks corresponds to a list of names whose unfolding is of the form $0 :: 1 :: 2 :: \dots (n-1)$.

The implementation of this operation allocates a block of size $n+1$ at a fresh location p , then sets the header cell to the value n , then returns the address p . In the statement below, the expression `{LibListExec.length  $ks$ }` in braces computes in Rocq the expression `length  $ks$` , and inserts the result in the code as a constant integer – that is, as a value of the form `val_int (LibListExec.length  $ks$ )`.

Definition `val_alloc_hrecord ( $ks$ :list field) : val :=`

```
<{ fun 'v =>
  let 'm = {LibListExec.length  $ks$ } + 1 in
  let 'p = val_alloc 'm in
  val_set 'p {LibListExec.length  $ks$ };
  'p }>.
```

A key auxiliary result asserts that, if kvs is a list of key-value pairs such that the keys correspond to consecutive field names, then a list of fields described as `hfields  $kvs$  p` corresponds to a range of consecutive cells as described by `hrange (List.map snd  $kvs$ ) (p+1)`, for the definition of `hrange` given in `Arrays`.

Lemma `hfields_eq_hrange :  $\forall z L p kvs$ ,`
`maps_all_fields  $z kvs$   $\rightarrow$`
 `$z$  = length  $L$   $\rightarrow$`
 `$L$  = LibList.map snd  $kvs$   $\rightarrow$`
`hfields  $kvs$   $p$  = hrange  $L$  ( $p+1$ )%nat.`

Proof using.

```
asserts Ind: ( $\forall L p kvs o$ ,
  LibList.map fst  $kvs$  = nat_seq o (length  $L$ )  $\rightarrow$ 
   $L$  = LibList.map snd  $kvs$   $\rightarrow$ 
  hfields  $kvs$   $p$  = hrange  $L$  ( $p+1+o$ )%nat).
{ intros  $L$ . induction  $L$  as [|v L']; introv  $M E$ ; rew_listx in *;
  destruct  $kvs$  as [| [ $k v'$ ]  $kvs'$ ]; tryfalse; rew_listx in *.
{ auto. }
{ simpls. inverts  $M$  as  $M'$ . inverts  $E$  as  $E'$ . rew_listx in *.
  unfold hfield. fequals. math_rewrite ( $p+1+o+1=p+1+(o+1)$ )%nat.
```

```

    applies× IHL'. math_rewrite× (o+1=S o)%nat. } }
  introv M HL E. subst z. rewrites (» Ind M E). fequals. unfold loc. math.
Qed.

```

The specification of `val_alloc_hrecord` *ks* involves an empty precondition and a postcondition of the form `hrecord kvs p`, where the list *kvs* maps the fields names from *ks* to the value `val_uninit`. The premise `ks = nat_seq 0 (length ks)` checks that the list *ks* contains consecutive offsets starting from zero.

Recall that `LibListExec.length` is a variant of `LibList.length` that computes in Rocq (using `simpl` or `reflexivity`). Likewise `LibListExec.map` is a computable version of `LibList.map`.

```

Lemma triple_alloc_hrecord : ∀ ks,
  ks = nat_seq 0 (LibListExec.length ks) →
  triple ((val_alloc_hrecord ks) ())
    \ []
    (funloc p ⇒ hrecord (LibListExec.map (fun k ⇒ (k, val_uninit)) ks) p).

```

Proof using.

```

  introv Hks. xwp. xapp. xapp. intros p Hp. unfolds field.
  rewrite LibListExec.length_eq in *. set (n := length ks) in *.
  math_rewrite (abs (n + 1) = S n). rew_listx. simpl.
  xapp. xval. xsimpl×. unfold hrecord. autorewrite with rew_list_exec.
  asserts R: (maps_all_fields n (LibList.map (fun k ⇒ (k, val_uninit)) ks)).
  { unfolds. rewrite Hks. clears ks p. generalize 0%nat as o.
    induction n as [|n']; intros; simpl; rew_listx; fequals×. }
  xsimpl× n. xchange× ← (@hfields_eq_hrange n). { rew_listx×. }
  { unfold n. clears n p. induction ks as [|ks']; rew_listx; fequals×. }

```

Qed.

```

#[global] Hint Resolve triple_alloc_hrecord : triple.

```

The interest of using the computable counterparts of the function `length` and `map` is that a call to `simpl` will yield exactly the expected result, with lists indexed using field names. For example, the allocation of a list cell is specified as follows.

```

Lemma triple_alloc_mcons :
  triple (val_alloc_hrecord (head::tail::nil) ())
    \ []
    (funloc p ⇒ p ~~~> '{ head := val_uninit ; tail := val_uninit }).

```

Proof using.

```

  dup.
  {
    applies triple_alloc_hrecord. simpl. rew_listx. reflexivity. }
  {
    applies× triple_alloc_hrecord. }

```

Qed.

49.3.5 Implementation of Record Allocation with Initialization

We now show the implementation of record initialization in the particular case of records with exactly 2 fields. Recall that `val_new_hrecord_2 k1 k2` denotes a Rocq value of type `val`, which may be applied in the grammar of terms to two arguments, `x1` and `x2`. The definition below shows an implementation for `val_new_hrecord_2`. In this definition, the expression in braces `val_alloc_hrecord (k1::k2::nil)` refers to a Rocq term, embedded in the syntax of program terms.

Definition `val_new_hrecord_2 (k1:field) (k2:field) : val :=`

```
<{ fun 'x1 'x2 =>
  let 'p = {val_alloc_hrecord (k1::k2::nil)} () in
  'p'.k1 := 'x1;
  'p'.k2 := 'x2;
  'p }>.
```

To improve readability, we introduce notation to allow writing, e.g., `{ head := x; tail := q }` for the allocation and initialization of a list cell.

Notation `"{ k1 := v1 ; k2 := v2 }" :=`

```
(val_new_hrecord_2 k1 k2 v1 v2)
(in custom trm at level 65,
 k1, k2 at level 0,
 v1, v2 at level 65) : trm_scope_ext.
```

This operation is specified as follows.

Lemma `triple_new_hrecord_2 : ∀ k1 k2 v1 v2,`
`k1 = 0%nat →`
`k2 = 1%nat →`
`triple <{ '{ k1 := v1 ; k2 := v2 } }>`
`\ []`
`(funloc p => p ~~~> '{ k1 := v1 ; k2 := v2 }).`

Proof using.

```
introv → →. xwp. xapp triple_alloc_hrecord. { reflexivity. }
intros p. simpl.
xapp triple_set_field_hrecord. { reflexivity. }
xapp triple_set_field_hrecord. { reflexivity. }
xval. xsimpl ×.
```

Qed.

End REALIZATION.

This completes the verification of the implementation of record operations in terms of block allocation and pointer arithmetic. The generalization of `val_new_hrecord_2` to handle arbitrary arities involves more technical meta-programming, beyond the scope of this course.

49.3.6 Extending *xapp* to Support Record Access Operations

This section generalizes the tactic *xapp* to enable it to automatically reason about read and write on records specified using *hrecord*. The lemma *xapp_get_field_lemma* reformulates the specification *triple_get_field_hrecord* in weakest-precondition form. Its statement takes the form: $H \implies \backslash \exists kvs, (\text{hrecord } kvs \text{ } p) \backslash * (\text{match } \dots \text{ with } \text{Some } .. \Rightarrow ..)$, where the *match* construct reflects the preconditions *hfields_lookup* *k kvs* = *Some v*.

```
Lemma xapp_get_field_lemma :  $\forall H \ k \ p \ Q,$ 
   $H \implies \backslash \exists kvs, (\text{hrecord } kvs \text{ } p) \backslash *$ 
    match hfields_lookup k kvs with
    | None  $\Rightarrow \backslash [\text{False}]$ 
    | Some v  $\Rightarrow ((\text{fun } r \Rightarrow \backslash [r = v]) \backslash * \text{hrecord } kvs \text{ } p) \backslash * \text{protect } Q) \text{ end} \rightarrow$ 
   $H \implies \text{wpgen\_app } (\text{val\_get\_field } k \text{ } p) \text{ } Q.$ 
```

Proof using.

```
introv N. xchange N. intros kvs. cases (hfields_lookup k kvs).
{ unfold wpgen_app. xsimpl.
  applies  $\times$  triple_conseq_frame triple_get_field_hrecord.
  { xsimpl. }
  { xpull. intros r  $\rightarrow$ . xchange (qwand_specialize v).
    rewrite  $\times$  hwand_hpure_l. } }
{ xpull. }
```

Qed.

Likewise, the lemma *xapp_set_field_lemma* reformulates the specification *triple_set_field_hrecord*. The assumption *hfields_update* *k v kvs* = *Some ks'* is also captured by a pattern matching.

```
Lemma xapp_set_field_lemma :  $\forall H \ k \ p \ v \ Q,$ 
   $H \implies \backslash \exists kvs, (\text{hrecord } kvs \text{ } p) \backslash *$ 
    match hfields_update k v kvs with
    | None  $\Rightarrow \backslash [\text{False}]$ 
    | Some kvs'  $\Rightarrow ((\text{fun } _ \Rightarrow \text{hrecord } kvs' \text{ } p) \backslash * \text{protect } Q) \text{ end} \rightarrow$ 
   $H \implies \text{wpgen\_app } (\text{val\_set\_field } k \text{ } p \text{ } v) \text{ } Q.$ 
```

Proof using.

```
introv N. xchange N. intros kvs. cases (hfields_update k v kvs).
{ unfold wpgen_app. xsimpl.
  applies  $\times$  triple_conseq_frame triple_set_field_hrecord.
  { xsimpl. }
  { xpull. intros r. xchange (qwand_specialize r). } }
{ xpull. }
```

Qed.

The hook *xapp_nosubst_for_records*, invoked by *xapp*, is then implemented by exploiting the two lemmas above, in conjunction with *xsimpl*.

```
Ltac xapp_nosubst_for_records tt ::=
  first [ applies xapp_set_field_lemma; xsimpl; simpl; xapp_simpl
```

| *appls* xapp_get_field_lemma; *xsimpl*; **simpl**; *xapp_simpl*].

The above definition is the one used in [LibSepReference](#). It was put to practice in the chapters [Basic](#) and [Repr](#).

Chapter 50

Library **SLF.Postscript**

50.1 Postscript: Conclusion and Perspectives

50.2 Beyond the Scope of This Course

This course has focused on the foundations of Separation Logic for sequential programs, and discussed the set up of tooling for computing weakest preconditions.

There are numerous aspects of Separation Logic that were not covered in this course, including:

- Concurrent Separation Logic, for reasoning about concurrent programs.
- Automated reasoning using Separation Logic, typically for a restricted fragment of Separation Logic.
- Separation Logic for low-level programs, at the byte level, and with assembly-style code written in continuation-passing style.
- Separation Logic for object-oriented programs.
- Reasoning about I/O, integer overflows, floating-point arithmetics, etc.

50.3 Tools Leveraging Separation Logic

Ideas of Separation Logic have had significant influence on the field of programming languages from various perspectives. For a broad survey of Separation Logic, we refer to Peter O’Hearn’s CACM paper 2019. <https://dl.acm.org/doi/10.1145/3211968> (Make sure to also download the “supplementary material” link at the bottom of the page.)

Here is a non-exhaustive list of active projects leveraging Separation Logic.

- The tool “Infer” leverages for static analysis, that is, fully automated program analysis to produce a list of potential bugs. <https://fbinfer.com/> <https://fbinfer.com/docs/separation-logic-and-bi-abduction/>
- The programming language “Rust” features a type system with “borrows”, a notion directly inspired by Separation Logic.
- The tool “VeriFast” targets C and Java programs. <https://people.cs.kuleuven.be/~bart.jacobs/verifast> and <https://github.com/verifast/verifast>
- The tool “Verifiable C” tool, from the VST project, targets the verification of C code, using interactive Rocq proofs based on Separation Logic. <https://vst.cs.princeton.edu/> See also volume 5 of Software Foundations.
- The tool “Viper” is based on a variant of Separation Logic, its front-end supports several languages, and its backend leverages the SMT solver Z3; it also supports user annotations in the code for guiding proofs. <https://www.pm.inf.ethz.ch/research/viper.html>
- The “Isabelle Refinement Framework” leverages Separation Logic is the process of refining monadic high-order logic definitions into efficient imperative code. It can be used to produce verified ML code, as well as LLVM IR code. https://www21.in.tum.de/~lammich/pub/itp15_
- The tool “CFML” targets the verification of OCaml programs. The original version of CFML leverages a version of characteristic formulae slightly more complicated than the one presented in this volume. The new version, CFML 2.0, directly leverages all the ideas from this course, in particular the weakest-precondition style `wpgen` function.
- The verified compiler “CakeML” implements technology similar to that of CFML, and leverages it to verify components from its standard library and from its runtime.
- The Iris project develops advanced logics for reasoning about concurrent programs using Separation Logic.

50.4 Related Courses

The following courses focus on reasoning about programs, and should be of interest to the reader:

- Verifiable C, volume 5 of Software Foundations, by Andrew Appel. <https://softwarefoundations.cis.upenn.edu/current/index.html>
- Formal Reasoning About Programs, by Adam Chlipala. <http://adam.chlipala.net/frap/>
- The Iris tutorial, by Birkedal and Bizjak, presents the core ideas of Iris’ concurrent Separation Logic. <https://iris-project.org/tutorial-material.html>

50.5 Acknowledgments

The lead author of this volume, Arthur Charguéraud, wishes to thank:

- Benjamin Pierce, for demonstrating the benefits of Rocq-based teaching, for coordinating the Software Foundations series, and for encouraging me to write this volume.
- The Software Foundations (SF) community, for helping to polish the course material and the tooling, and contributing to the success of teaching using this material.
- François Pottier, with whom I developed several extensions of Separation Logic, and thereby obtained a deeper understanding of this logic.
- Jacques-Henri Jourdan, who introduced me to a number of techniques used in the Iris project, and contributed to the definition of `mkstruct`.
- Xavier Leroy, with whom I had numerous discussions on mechanized semantics.
- Jean-Christophe Filliâtre, for numerous discussions on program verification.
- Andrew Appel, Lars Birkedal, Adam Chlipala, Magnus Myreen, Gerwin Klein, Peter Lammich, and Zhong Shao, who kindly answered questions on related work aspects.
- The many readers who helped fixing typos.

Chapter 51

Library **SLF.Bib**

51.1 Bib: Bibliography

51.2 Resources cited in this volume

An extended related work section on the history of Separation Logic may be found in the companion course notes, available from: http://www.chargueraud.org/teach/verif/slf_notes.pdf

The references most relevant to the contents of the course appear below.

Birkedal, Torp-Smith and Yang 2006 Lars Birkedal, Noah Torp-smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules for algol-like languages. In Logical Methods in Computer Science.

Burstall 1972 R. M. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. In Machine Intelligence 7, Edinburgh University Press.

Charguéraud 2010 Arthur Charguéraud. Characteristic Formulae for Mechanized Program Verification. PhD thesis, Université Paris Diderot, December 2010.

Charguéraud 2011 Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In International Conference on Functional Programming (ICFP). <https://doi.org/10.1145/2034773.2034828>

Charguéraud 2020 Arthur Charguéraud. Separation Logic for Sequential Programs. In International Conference on Function Programming (ICFP). <https://dl.acm.org/doi/abs/10.1145/3408998>

Charguéraud 2023 Arthur Charguéraud. A Modern Eye on Separation Logic for Sequential Programs. Habilitation Manuscript. https://www.chargueraud.org/research/2023/hdr/chargueraud_h
<https://inria.hal.science/tel-04076725/>

Charguéraud et al 2023 Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smoother Handling of Nondeterminism (TOPLAS). <http://www.chargueraud.org>

Chlipala et al 2009 Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-Order Imperative Programs. In International Conference on Functional Programming (ICFP). <https://doi.org/10.1145/1596550>.
See also the other papers: <https://ynot.cs.harvard.edu/>

- Guéneau, Jourdan, Charguéraud, and Pottier* 2019 Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In Interactive Theorem Proving (ITP). <https://doi.org/10.4230/LIPIcs.ITP>
- Guéneau, Myreen, Kumar and Norrish* 2017 Armaël Guéneau, Magnus O. Myreen, Ravana Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In European Symposium on Programming (ESOP). https://doi.org/10.1007/978-3-662-54434-1_22
- Dijkstra* 1975 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM. <https://doi.org/10.1145/360933.360975>
- Gordon* 1989 Michael J. C. Gordon. Mechanizing Programming Logics in Higher Order Logic. Springer-Verlag, Berlin, Heidelberg, https://doi.org/10.1007/978-1-4612-3658-0_10
- Krishnaswami, Birkedal, and Aldrich* 2010 Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying Event-Driven Programs Using Ramified Frame Properties. In Workshop on Types in Language Design and Implementation (TLDI). <https://doi.org/10.1145/1708016.170>
- Hennessy and Milner* 1985 Matthew Hennessy and Robin Milner. Algebraic laws for non-determinism and concurrency. Journal of the ACM, 32. <https://dl.acm.org/doi/10.1145/2455.2460>
- Honda, Berger, and Yoshida* 2006 Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. International Colloquium on Automata, Languages and Programming (ICALP). https://doi.org/10.1007/11787006_31
- Hobor and Villard* 2013 Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In Symposium on Principles of Programming Languages (POPL). <https://doi.org/10.1145/2429069.2429131>
- Ishtiaq and O’Hearn* 2001 Samin S. Ishtiaq and Peter W. O’Hearn. BI as an Assertion Language for Mutable Data Structures. SIGPLAN Notice 36. <https://doi.org/10.1145/373243.375719>
- Jung et al* 2018 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleksey Bizjak, Lars Birkedal, Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming. <https://doi.org/10.1017/S0956796818000>
- O’Hearn and Pym* 1999 Peter W. O’Hearn and David J. Pym. The Logic of Bunched Implications. The Bulletin of Symbolic Logic 5. <https://doi.org/10.2307/421090>
- O’Hearn, Reynolds, and Yang* 2001 Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. Workshop on Computer Science Logic (CSL). https://doi.org/10.1007/3-540-44802-0_1
- Ni, Shao* 2006 Zhaozhong Ni and Zhong Shao. Certified Assembly Programming with Embedded Code Pointers. Symposium on Principles of Programming Languages (POPL). <https://doi.org/10.1145/1111037.1111066>
- Reynolds* 2002 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. Annual IEEE Symposium on Logic in Computer Science (LICS). 10.1109/LICS.2002.1029817
- Reynolds* 2000 John C. Reynolds. Intuitionistic Reasoning about Shared Mutable Data Structure <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.5999&rep=rep1&type=pdf>
- Schäfer et al* 2016 Steven Schäfer, Sigurd Schneider, Gert Smolka. Axiomatic Semantics for Compiler Verification. In Conference on Certified Programs and Proofs (CPP). <https://dl.acm.org/doi/10.1145/2854065.2854083>
- Tuerk* 2010 Thomas Tuerk. Local Reasoning about While-Loops. In International Con-

ference on Verified Software: Theories, Tools and Experiments (VSTTE).

Weber 2004 Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In Computer Science Logic (CSL) https://doi.org/10.1007/978-3-540-30124-0_21

Yu, Hamid, and Shao 2003 Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. European Symposium on Programming (ESOP). https://doi.org/10.1007/3-540-36575-3_25