

# Contents

<b>1</b>	<b>Library Preface</b>	<b>2</b>
1.1	Preface . . . . .	2
1.2	Welcome . . . . .	2
1.3	Overview . . . . .	2
1.3.1	Logic . . . . .	3
1.3.2	Proof Assistants . . . . .	4
1.3.3	Functional Programming . . . . .	5
1.3.4	Program Verification . . . . .	6
1.3.5	Type Systems . . . . .	7
1.4	Practicalities . . . . .	7
1.4.1	Chapter Dependencies . . . . .	7
1.4.2	System Requirements . . . . .	7
1.4.3	Exercises . . . . .	8
1.4.4	Downloading the Coq Files . . . . .	8
1.5	Note for Instructors . . . . .	8
1.6	Translations . . . . .	9
<b>2</b>	<b>Library Basics</b>	<b>10</b>
2.1	Basics: Functional Programming in Coq . . . . .	10
2.2	Introduction . . . . .	10
2.3	Enumerated Types . . . . .	11
2.3.1	Days of the Week . . . . .	11
2.3.2	Booleans . . . . .	12
2.3.3	Function Types . . . . .	14
2.3.4	Modules . . . . .	15
2.3.5	Numbers . . . . .	15
2.4	Proof by Simplification . . . . .	20
2.5	Proof by Rewriting . . . . .	22
2.6	Proof by Case Analysis . . . . .	23
2.6.1	More on Notation (Optional) . . . . .	27
2.6.2	Fixpoints and Structural Recursion (Optional) . . . . .	27
2.7	More Exercises . . . . .	28

<b>3</b>	<b>Library Induction</b>	<b>30</b>
3.1	Induction: Proof by Induction . . . . .	30
3.2	Proof by Induction . . . . .	30
3.3	Proofs Within Proofs . . . . .	33
3.4	More Exercises . . . . .	34
3.5	Formal vs. Informal Proof (Optional) . . . . .	37
<b>4</b>	<b>Library Lists</b>	<b>38</b>
4.1	Lists: Working with Structured Data . . . . .	38
4.2	Pairs of Numbers . . . . .	38
4.3	Lists of Numbers . . . . .	40
4.4	Reasoning About Lists . . . . .	46
4.4.1	Induction on Lists . . . . .	47
4.4.2	<code>SearchAbout</code> . . . . .	50
4.4.3	List Exercises, Part 1 . . . . .	50
4.4.4	List Exercises, Part 2 . . . . .	51
4.5	Options . . . . .	52
4.6	Partial Maps . . . . .	54
<b>5</b>	<b>Library Poly</b>	<b>57</b>
5.1	Poly: Polymorphism and Higher-Order Functions . . . . .	57
5.2	Polymorphism . . . . .	57
5.2.1	Polymorphic Lists . . . . .	57
5.2.2	Polymorphic Pairs . . . . .	64
5.2.3	Polymorphic Options . . . . .	65
5.3	Functions as Data . . . . .	66
5.3.1	Higher-Order Functions . . . . .	66
5.3.2	Filter . . . . .	67
5.3.3	Anonymous Functions . . . . .	67
5.3.4	Map . . . . .	69
5.3.5	Fold . . . . .	70
5.3.6	Functions That Construct Functions . . . . .	71
5.4	Additional Exercises . . . . .	72
<b>6</b>	<b>Library Tactics</b>	<b>76</b>
6.1	Tactics: More Basic Tactics . . . . .	76
6.2	The <code>apply</code> Tactic . . . . .	76
6.3	The <code>apply ... with ...</code> Tactic . . . . .	78
6.4	The <code>inversion</code> tactic . . . . .	79
6.5	Using Tactics on Hypotheses . . . . .	82
6.6	Varying the Induction Hypothesis . . . . .	83
6.7	Unfolding Definitions . . . . .	89
6.8	Using <code>destruct</code> on Compound Expressions . . . . .	91

6.9	Review . . . . .	93
6.10	Additional Exercises . . . . .	94
<b>7</b>	<b>Library Logic</b>	<b>96</b>
7.1	Logic: Logic in Coq . . . . .	96
7.2	Logical Connectives . . . . .	97
7.2.1	Conjunction . . . . .	97
7.2.2	Disjunction . . . . .	100
7.2.3	Falsehood and Negation . . . . .	102
7.2.4	Truth . . . . .	104
7.2.5	Logical Equivalence . . . . .	105
7.2.6	Existential Quantification . . . . .	107
7.3	Programming with Propositions . . . . .	108
7.4	Applying Theorems to Arguments . . . . .	111
7.5	Coq vs. Set Theory . . . . .	112
7.5.1	Functional Extensionality . . . . .	113
7.5.2	Propositions and Booleans . . . . .	114
7.5.3	Classical vs. Constructive Logic . . . . .	118
<b>8</b>	<b>Library IndProp</b>	<b>121</b>
8.1	IndProp: Inductively Defined Propositions . . . . .	121
8.2	Inductively Defined Propositions . . . . .	121
8.3	Using Evidence in Proofs . . . . .	123
8.3.1	Inversion on Evidence . . . . .	123
8.3.2	Induction on Evidence . . . . .	126
8.4	Inductive Relations . . . . .	129
8.5	Case Study: Regular Expressions . . . . .	133
8.5.1	The <i>remember</i> Tactic . . . . .	139
8.6	Improving Reflection . . . . .	143
8.7	Additional Exercises . . . . .	145
<b>9</b>	<b>Library Maps</b>	<b>149</b>
9.1	Maps: Total and Partial Maps . . . . .	149
9.2	The Coq Standard Library . . . . .	149
9.3	Identifiers . . . . .	150
9.4	Total Maps . . . . .	151
9.5	Partial maps . . . . .	153
<b>10</b>	<b>Library ProofObjects</b>	<b>156</b>
10.1	ProofObjects: The Curry-Howard Correspondence . . . . .	156
10.2	Proof Scripts . . . . .	158
10.3	Quantifiers, Implications, Functions . . . . .	159
10.4	Connectives as Inductive Types . . . . .	160

10.4.1	Conjunction . . . . .	160
10.4.2	Disjunction . . . . .	161
10.4.3	Existential Quantification . . . . .	162
10.4.4	<b>True</b> and <b>False</b> . . . . .	162
10.5	Programming with Tactics . . . . .	163
10.6	Equality . . . . .	163
10.6.1	Inversion, Again . . . . .	164
<b>11</b>	<b>Library IndPrinciples</b>	<b>166</b>
11.1	IndPrinciples: Induction Principles . . . . .	166
11.2	Basics . . . . .	166
11.3	Polymorphism . . . . .	169
11.4	Induction Hypotheses . . . . .	170
11.5	More on the <code>induction</code> Tactic . . . . .	170
11.6	Induction Principles in <code>Prop</code> . . . . .	172
11.7	Formal vs. Informal Proofs by Induction . . . . .	174
11.7.1	Induction Over an Inductively Defined Set . . . . .	174
11.7.2	Induction Over an Inductively Defined Proposition . . . . .	175
<b>12</b>	<b>Library Sflib</b>	<b>177</b>
12.1	Sflib: Software Foundations Library . . . . .	177
<b>13</b>	<b>Library Rel</b>	<b>178</b>
13.1	Rel: Properties of Relations . . . . .	178
13.2	Basic Properties . . . . .	179
13.3	Reflexive, Transitive Closure . . . . .	183
<b>14</b>	<b>Library Imp</b>	<b>186</b>
14.1	Imp: Simple Imperative Programs . . . . .	186
14.2	Arithmetic and Boolean Expressions . . . . .	186
14.2.1	Syntax . . . . .	187
14.2.2	Evaluation . . . . .	188
14.2.3	Optimization . . . . .	188
14.3	Coq Automation . . . . .	190
14.3.1	Tacticals . . . . .	190
14.3.2	Defining New Tactic Notations . . . . .	193
14.3.3	The <code>omega</code> Tactic . . . . .	194
14.3.4	A Few More Handy Tactics . . . . .	195
14.4	Evaluation as a Relation . . . . .	195
14.4.1	Inference Rule Notation . . . . .	196
14.4.2	Equivalence of the Definitions . . . . .	197
14.4.3	Computational vs. Relational Definitions . . . . .	199
14.5	Expressions With Variables . . . . .	200

14.5.1	States . . . . .	200
14.5.2	Syntax . . . . .	201
14.5.3	Evaluation . . . . .	202
14.6	Commands . . . . .	202
14.6.1	Syntax . . . . .	202
14.6.2	Examples . . . . .	203
14.7	Evaluation . . . . .	204
14.7.1	Evaluation as a Function (Failed Attempt) . . . . .	204
14.7.2	Evaluation as a Relation . . . . .	205
14.7.3	Determinism of Evaluation . . . . .	208
14.8	Reasoning About Imp Programs . . . . .	209
14.9	Additional Exercises . . . . .	211
<b>15</b>	<b>Library ImpParser</b>	<b>217</b>
15.1	ImpParser: Lexing and Parsing in Coq . . . . .	217
15.2	Internals . . . . .	217
15.2.1	Lexical Analysis . . . . .	217
15.2.2	Parsing . . . . .	219
15.3	Examples . . . . .	226
<b>16</b>	<b>Library ImpCEvalFun</b>	<b>227</b>
16.1	ImpCEvalFun: Evaluation Function for Imp . . . . .	227
16.2	A Broken Evaluator . . . . .	227
16.3	A Step-Indexed Evaluator . . . . .	228
16.4	Relational vs. Step-Indexed Evaluation . . . . .	231
16.5	Determinism of Evaluation Again . . . . .	234
<b>17</b>	<b>Library Extraction</b>	<b>235</b>
17.1	Extraction: Extracting ML from Coq . . . . .	235
17.2	Basic Extraction . . . . .	235
17.3	Controlling Extraction of Specific Types . . . . .	235
17.4	A Complete Example . . . . .	236
17.5	Discussion . . . . .	237
<b>18</b>	<b>Library Equiv</b>	<b>238</b>
18.1	Equiv: Program Equivalence . . . . .	238
18.2	Behavioral Equivalence . . . . .	239
18.2.1	Definitions . . . . .	239
18.2.2	Examples . . . . .	240
18.3	Properties of Behavioral Equivalence . . . . .	247
18.3.1	Behavioral Equivalence is an Equivalence . . . . .	247
18.3.2	Behavioral Equivalence is a Congruence . . . . .	248
18.4	Program Transformations . . . . .	251

18.4.1	The Constant-Folding Transformation . . . . .	252
18.4.2	Soundness of Constant Folding . . . . .	255
18.5	Proving That Programs Are <i>Not</i> Equivalent . . . . .	258
18.6	Extended Exercise: Nondeterministic Imp . . . . .	261
18.7	Additional Exercises . . . . .	265
<b>19</b>	<b>Library Hoare</b>	<b>268</b>
19.1	Hoare: Hoare Logic, Part I . . . . .	268
19.2	Assertions . . . . .	269
19.3	Hoare Triples . . . . .	270
19.4	Proof Rules . . . . .	273
19.4.1	Assignment . . . . .	273
19.4.2	Consequence . . . . .	276
19.4.3	Digression: The <code>eapply</code> Tactic . . . . .	278
19.4.4	Skip . . . . .	280
19.4.5	Sequencing . . . . .	281
19.4.6	Conditionals . . . . .	283
19.4.7	Recap . . . . .	287
19.4.8	Loops . . . . .	288
19.5	Summary . . . . .	293
19.6	Additional Exercises . . . . .	294
<b>20</b>	<b>Library Hoare2</b>	<b>297</b>
20.1	Hoare2: Hoare Logic, Part II . . . . .	297
20.2	Decorated Programs . . . . .	297
20.2.1	Example: Swapping Using Addition and Subtraction . . . . .	299
20.2.2	Example: Simple Conditionals . . . . .	300
20.2.3	Example: Reduce to Zero . . . . .	301
20.2.4	Example: Division . . . . .	302
20.3	Finding Loop Invariants . . . . .	303
20.3.1	Example: Slow Subtraction . . . . .	303
20.3.2	Exercise: Slow Assignment . . . . .	305
20.3.3	Exercise: Slow Addition . . . . .	306
20.3.4	Example: Parity . . . . .	306
20.3.5	Example: Finding Square Roots . . . . .	307
20.3.6	Example: Squaring . . . . .	308
20.3.7	Exercise: Factorial . . . . .	309
20.3.8	Exercise: Min . . . . .	310
20.3.9	Exercise: Power Series . . . . .	311
20.4	Weakest Preconditions (Optional) . . . . .	311
20.5	Formal Decorated Programs (Optional) . . . . .	313
20.5.1	Syntax . . . . .	313
20.5.2	Extracting Verification Conditions . . . . .	316

20.5.3	Automation . . . . .	318
20.5.4	Examples . . . . .	320
<b>21</b>	<b>Library HoareAsLogic</b>	<b>332</b>
21.1	HoareAsLogic: Hoare Logic as a Logic . . . . .	332
21.2	Definitions . . . . .	332
21.3	Properties . . . . .	334
<b>22</b>	<b>Library Smallstep</b>	<b>338</b>
22.1	Smallstep: Small-step Operational Semantics . . . . .	338
22.2	A Toy Language . . . . .	339
22.3	Relations . . . . .	341
22.3.1	Values . . . . .	343
22.3.2	Strong Progress and Normal Forms . . . . .	345
22.4	Multi-Step Reduction . . . . .	352
22.4.1	Examples . . . . .	354
22.4.2	Normal Forms Again . . . . .	355
22.4.3	Equivalence of Big-Step and Small-Step Reduction . . . . .	358
22.4.4	Additional Exercises . . . . .	359
22.5	Small-Step Imp . . . . .	360
22.6	Concurrent Imp . . . . .	364
22.7	A Small-Step Stack Machine . . . . .	368
<b>23</b>	<b>Library Auto</b>	<b>370</b>
23.1	Auto: More Automation . . . . .	370
23.2	The <code>auto</code> and <code>eauto</code> tactics . . . . .	371
23.3	Searching Hypotheses . . . . .	376
<b>24</b>	<b>Library Types</b>	<b>382</b>
24.1	Types: Type Systems . . . . .	382
24.2	Typed Arithmetic Expressions . . . . .	382
24.2.1	Syntax . . . . .	382
24.2.2	Operational Semantics . . . . .	383
24.2.3	Normal Forms and Values . . . . .	385
24.2.4	Typing . . . . .	386
24.2.5	Progress . . . . .	388
24.2.6	Type Preservation . . . . .	390
24.2.7	Type Soundness . . . . .	391
24.3	Aside: the <i>normalize</i> Tactic . . . . .	392
24.3.1	Additional Exercises . . . . .	393

<b>25 Library Stlc</b>	<b>396</b>
25.1 Stlc: The Simply Typed Lambda-Calculus . . . . .	396
25.2 The Simply Typed Lambda-Calculus . . . . .	396
25.2.1 Overview . . . . .	396
25.2.2 Syntax . . . . .	398
25.2.3 Operational Semantics . . . . .	399
25.2.4 Typing . . . . .	405
<b>26 Library StlcProp</b>	<b>410</b>
26.1 StlcProp: Properties of STLC . . . . .	410
26.2 Canonical Forms . . . . .	410
26.3 Progress . . . . .	411
26.4 Preservation . . . . .	413
26.4.1 Free Occurrences . . . . .	413
26.4.2 Substitution . . . . .	414
26.4.3 Main Theorem . . . . .	419
26.5 Type Soundness . . . . .	420
26.6 Uniqueness of Types . . . . .	420
26.7 Additional Exercises . . . . .	421
26.7.1 Exercise: STLC with Arithmetic . . . . .	423
<b>27 Library MoreStlc</b>	<b>425</b>
27.1 MoreStlc: More on the Simply Typed Lambda-Calculus . . . . .	425
27.2 Simple Extensions to STLC . . . . .	425
27.2.1 Numbers . . . . .	425
27.2.2 <code>let</code> -bindings . . . . .	425
27.2.3 Pairs . . . . .	426
27.2.4 Unit . . . . .	428
27.2.5 Sums . . . . .	429
27.2.6 Lists . . . . .	431
27.2.7 General Recursion . . . . .	432
27.2.8 Records . . . . .	436
27.3 Exercise: Formalizing the Extensions . . . . .	439
27.3.1 Examples . . . . .	448
27.3.2 Properties of Typing . . . . .	452
<b>28 Library Sub</b>	<b>464</b>
28.1 Sub: Subtyping . . . . .	464
28.2 Concepts . . . . .	464
28.2.1 A Motivating Example . . . . .	464
28.2.2 Subtyping and Object-Oriented Languages . . . . .	465
28.2.3 The Subsumption Rule . . . . .	466
28.2.4 The Subtype Relation . . . . .	466



28.2.5	Exercises . . . . .	470
28.3	Formal Definitions . . . . .	473
28.3.1	Core Definitions . . . . .	473
28.3.2	Subtyping . . . . .	475
28.3.3	Typing . . . . .	477
28.3.4	Typing examples . . . . .	478
28.4	Properties . . . . .	478
28.4.1	Inversion Lemmas for Subtyping . . . . .	479
28.4.2	Canonical Forms . . . . .	479
28.4.3	Progress . . . . .	480
28.4.4	Inversion Lemmas for Typing . . . . .	482
28.4.5	Context Invariance . . . . .	485
28.4.6	Substitution . . . . .	486
28.4.7	Preservation . . . . .	488
28.4.8	Records, via Products and Top . . . . .	489
28.4.9	Exercises . . . . .	490
28.5	Exercise: Adding Products . . . . .	491
<b>29</b>	<b>Library Typechecking</b>	<b>492</b>
29.1	MoreStlc: A Typechecker for STLC . . . . .	492
29.1.1	Comparing Types . . . . .	492
29.1.2	The Typechecker . . . . .	493
29.1.3	Properties . . . . .	494
<b>30</b>	<b>Library Records</b>	<b>496</b>
30.1	Records: Adding Records to STLC . . . . .	496
30.2	Adding Records . . . . .	496
30.3	Formalizing Records . . . . .	497
30.3.1	Examples . . . . .	502
30.3.2	Properties of Typing . . . . .	503
<b>31</b>	<b>Library References</b>	<b>510</b>
31.1	References: Typing Mutable References . . . . .	510
31.2	Definitions . . . . .	511
31.3	Syntax . . . . .	511
31.4	Pragmatics . . . . .	514
31.4.1	Side Effects and Sequencing . . . . .	514
31.4.2	References and Aliasing . . . . .	515
31.4.3	Shared State . . . . .	516
31.4.4	Objects . . . . .	516
31.4.5	References to Compound Types . . . . .	517
31.4.6	Null References . . . . .	518
31.4.7	Garbage Collection . . . . .	518

31.5	Operational Semantics . . . . .	518
31.5.1	Locations . . . . .	518
31.5.2	Stores . . . . .	519
31.5.3	Reduction . . . . .	521
31.6	Typing . . . . .	525
31.6.1	Store typings . . . . .	525
31.6.2	The Typing Relation . . . . .	527
31.7	Properties . . . . .	529
31.7.1	Well-Typed Stores . . . . .	529
31.7.2	Extending Store Typings . . . . .	530
31.7.3	Preservation, Finally . . . . .	531
31.7.4	Substitution lemma . . . . .	532
31.7.5	Assignment Preserves Store Typing . . . . .	535
31.7.6	Weakening for Stores . . . . .	536
31.7.7	Preservation! . . . . .	537
31.7.8	Progress . . . . .	539
31.8	References and Nontermination . . . . .	542
31.9	Additional Exercises . . . . .	544
<b>32</b>	<b>Library RecordSub</b>	<b>545</b>
32.1	RecordSub: Subtyping with Records . . . . .	545
32.2	Core Definitions . . . . .	545
32.3	Subtyping . . . . .	548
32.3.1	Definition . . . . .	548
32.3.2	Subtyping Examples and Exercises . . . . .	549
32.3.3	Properties of Subtyping . . . . .	550
32.4	Typing . . . . .	552
32.4.1	Typing Examples . . . . .	553
32.4.2	Properties of Typing . . . . .	554
32.4.3	Exercises on Typing . . . . .	564
<b>33</b>	<b>Library Norm</b>	<b>566</b>
33.1	Norm: Normalization of STLC . . . . .	566
33.2	Language . . . . .	567
33.3	Normalization . . . . .	574
33.3.1	Membership in $R\_T$ is invariant under evaluation . . . . .	576
33.3.2	Closed instances of terms of type $T$ belong to $R\_T$ . . . . .	578
<b>34</b>	<b>Library LibTactics</b>	<b>587</b>
34.1	LibTactics: A Collection of Handy General-Purpose Tactics . . . . .	587
34.2	Tools for programming with Ltac . . . . .	588
34.2.1	Identity continuation . . . . .	588
34.2.2	Untyped arguments for tactics . . . . .	588

34.2.3	Optional arguments for tactics . . . . .	588
34.2.4	Wildcard arguments for tactics . . . . .	589
34.2.5	Position markers . . . . .	589
34.2.6	List of arguments for tactics . . . . .	589
34.2.7	Databases of lemmas . . . . .	592
34.2.8	On-the-fly removal of hypotheses . . . . .	593
34.2.9	Numbers as arguments . . . . .	594
34.2.10	Testing tactics . . . . .	595
34.2.11	Check no evar in goal . . . . .	595
34.2.12	Helper function for introducing evars . . . . .	595
34.2.13	Tagging of hypotheses . . . . .	596
34.2.14	Tagging of hypotheses . . . . .	596
34.2.15	Deconstructing terms . . . . .	596
34.2.16	Action at occurrence and action not at occurrence . . . . .	597
34.2.17	An alias for <code>eq</code> . . . . .	598
34.3	Common tactics for simplifying goals like <code>intuition</code> . . . . .	598
34.4	Backward and forward chaining . . . . .	598
34.4.1	Application . . . . .	598
34.4.2	Assertions . . . . .	600
34.4.3	Instantiation and forward-chaining . . . . .	602
34.4.4	Experimental tactics for application . . . . .	612
34.4.5	Adding assumptions . . . . .	613
34.4.6	Application of tautologies . . . . .	613
34.4.7	Application modulo equalities . . . . .	614
34.4.8	Absurd goals . . . . .	616
34.5	Introduction and generalization . . . . .	617
34.5.1	Introduction . . . . .	617
34.5.2	Generalization . . . . .	620
34.5.3	Naming . . . . .	621
34.6	Rewriting . . . . .	623
34.6.1	Replace . . . . .	625
34.6.2	Change . . . . .	626
34.6.3	Renaming . . . . .	626
34.6.4	Unfolding . . . . .	627
34.6.5	Simplification . . . . .	628
34.6.6	Evaluation . . . . .	629
34.6.7	Substitution . . . . .	629
34.6.8	Tactics to work with proof irrelevance . . . . .	631
34.6.9	Proving equalities . . . . .	631
34.7	Inversion . . . . .	632
34.7.1	Basic inversion . . . . .	632
34.7.2	Inversion with substitution . . . . .	633

34.7.3	Injection with substitution . . . . .	636
34.7.4	Inversion and injection with substitution –rough implementation . . .	637
34.7.5	Case analysis . . . . .	638
34.8	Induction . . . . .	641
34.9	Coinduction . . . . .	643
34.10	Decidable equality . . . . .	644
34.11	Equivalence . . . . .	644
34.12	N-ary Conjunctions and Disjunctions . . . . .	644
34.13	Tactics to prove typeclass instances . . . . .	650
34.14	Tactics to invoke automation . . . . .	650
34.14.1	Definitions for parsing compatibility . . . . .	650
34.14.2	<i>hint</i> to add hints local to a lemma . . . . .	651
34.14.3	<i>jauto</i> , a new automation tactics . . . . .	651
34.14.4	Definitions of automation tactics . . . . .	651
34.14.5	Parsing for light automation . . . . .	652
34.14.6	Parsing for strong automation . . . . .	661
34.15	Tactics to sort out the proof context . . . . .	669
34.15.1	Hiding hypotheses . . . . .	669
34.15.2	Sorting hypotheses . . . . .	672
34.15.3	Clearing hypotheses . . . . .	672
34.16	Tactics for development purposes . . . . .	674
34.16.1	Skipping subgoals . . . . .	674
34.17	Compatibility with standard library . . . . .	677
34.18	Additional notations for Coq . . . . .	677
34.18.1	N-ary Existentials –TODO: DEPRECATED, Coq now supports it. .	677
34.18.2	<i>let</i> bindings (EXPERIMENTAL). . . . .	677
<b>35</b>	<b>Library UseTactics</b>	<b>682</b>
35.1	UseTactics: Tactic Library for Coq: A Gentle Introduction . . . . .	682
35.2	Tactics for introduction and case analysis . . . . .	683
35.2.1	The tactic <i>introv</i> . . . . .	683
35.2.2	The tactic <i>inverts</i> . . . . .	684
35.3	Tactics for n-ary connectives . . . . .	687
35.3.1	The tactic <i>splits</i> . . . . .	687
35.3.2	The tactic <i>branch</i> . . . . .	687
35.3.3	The tactic $\exists$ . . . . .	688
35.4	Tactics for working with equality . . . . .	688
35.4.1	The tactics <i>asserts_rewrite</i> and <i>cuts_rewrite</i> . . . . .	689
35.4.2	The tactic <i>subst</i> . . . . .	689
35.4.3	The tactic <i>fequals</i> . . . . .	690
35.4.4	The tactic <i>applies_eq</i> . . . . .	690
35.5	Some convenient shorthands . . . . .	691
35.5.1	The tactic <i>unfolds</i> . . . . .	691

35.5.2	The tactics <code>false</code> and <code>tryfalse</code> . . . . .	692
35.5.3	The tactic <code>gen</code> . . . . .	692
35.5.4	The tactics <code>skip</code> , <code>skip_rewrite</code> and <code>skip_goal</code> . . . . .	693
35.5.5	The tactic <code>sort</code> . . . . .	694
35.6	Tactics for advanced lemma instantiation . . . . .	695
35.6.1	Working of <code>lets</code> . . . . .	695
35.6.2	Working of <code>applies</code> , <code>forwards</code> and <code>specializes</code> . . . . .	697
35.6.3	Example of instantiations . . . . .	698
35.7	Summary . . . . .	699
<b>36</b>	<b>Library UseAuto</b> . . . . .	<b>701</b>
36.1	UseAuto: Theory and Practice of Automation in Coq Proofs . . . . .	701
36.2	Basic Features of Proof Search . . . . .	702
36.2.1	Strength of Proof Search . . . . .	702
36.2.2	Basics . . . . .	702
36.2.3	Conjunctions . . . . .	703
36.2.4	Disjunctions . . . . .	704
36.2.5	Existentials . . . . .	705
36.2.6	Negation . . . . .	706
36.2.7	Equalities . . . . .	706
36.3	How Proof Search Works . . . . .	706
36.3.1	Search Depth . . . . .	706
36.3.2	Backtracking . . . . .	708
36.3.3	Adding Hints . . . . .	710
36.3.4	Integration of Automation in Tactics . . . . .	711
36.4	Examples of Use of Automation . . . . .	712
36.4.1	Determinism . . . . .	712
36.4.2	Preservation for STLC . . . . .	715
36.4.3	Progress for STLC . . . . .	716
36.4.4	BigStep and SmallStep . . . . .	717
36.4.5	Preservation for STLCRef . . . . .	718
36.4.6	Progress for STLCRef . . . . .	721
36.4.7	Subtyping . . . . .	722
36.5	Advanced Topics in Proof Search . . . . .	723
36.5.1	Stating Lemmas in the Right Way . . . . .	723
36.5.2	Unfolding of Definitions During Proof-Search . . . . .	724
36.5.3	Automation for Proving Absurd Goals . . . . .	725
36.5.4	Automation for Transitivity Lemmas . . . . .	727
36.6	Decision Procedures . . . . .	729
36.6.1	Omega . . . . .	729
36.6.2	Ring . . . . .	730
36.6.3	Congruence . . . . .	731
36.7	Summary . . . . .	732

<b>37 Library PE</b>	<b>733</b>
37.1 PE: Partial Evaluation . . . . .	733
37.2 Generalizing Constant Folding . . . . .	734
37.2.1 Partial States . . . . .	734
37.2.2 Arithmetic Expressions . . . . .	735
37.2.3 Boolean Expressions . . . . .	738
37.3 Partial Evaluation of Commands, Without Loops . . . . .	740
37.3.1 Assignment . . . . .	741
37.3.2 Conditional . . . . .	742
37.3.3 The Partial Evaluation Relation . . . . .	746
37.3.4 Examples . . . . .	747
37.3.5 Correctness of Partial Evaluation . . . . .	748
37.4 Partial Evaluation of Loops . . . . .	750
37.4.1 Examples . . . . .	753
37.4.2 Correctness . . . . .	755
37.5 Partial Evaluation of Flowchart Programs . . . . .	760
37.5.1 Basic blocks . . . . .	761
37.5.2 Flowchart programs . . . . .	762
37.5.3 Partial evaluation of basic blocks and flowchart programs . . . . .	763
<b>38 Library Postscript</b>	<b>766</b>
38.1 Postscript . . . . .	766
38.2 Looking back... . . . .	766
38.3 Looking forward... . . . .	767

# Chapter 1

## Library Preface

### 1.1 Preface

### 1.2 Welcome

This electronic book is a course on *Software Foundations*, the mathematical underpinnings of reliable software. Topics include basic concepts of logic, computer-assisted theorem proving, the Coq proof assistant, functional programming, operational semantics, Hoare logic, and static type systems. The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity will be helpful.

The principal novelty of the course is that it is one hundred percent formalized and machine-checked: the entire text is literally a script for Coq. It is intended to be read alongside an interactive session with Coq. All the details in the text are fully formalized in Coq, and the exercises are designed to be worked using Coq.

The files are organized into a sequence of core chapters, covering about one semester’s worth of material and organized into a coherent linear narrative, plus a number of “appendices” covering additional topics. All the core chapters are suitable for both upper-level undergraduate and graduate students.

### 1.3 Overview

Building reliable software is hard. The scale and complexity of modern systems, the number of people involved in building them, and the range of demands placed on them render it extremely difficult to build software that is even more-or-less correct, much less 100% correct. At the same time, the increasing degree to which information processing is woven into every aspect of society continually amplifies the cost of bugs and insecurities.

Computer scientists and software engineers have responded to these challenges by developing a whole host of techniques for improving software reliability, ranging from recommendations about managing software projects and organizing programming teams (e.g., extreme programming) to design philosophies for libraries (e.g., model-view-controller, publish-subscribe, etc.) and programming languages (e.g., object-oriented programming, aspect-oriented programming, functional programming, ...) to mathematical techniques for specifying and reasoning about properties of software and tools for helping validate these properties.

The present course is focused on this last set of techniques. The text weaves together five conceptual threads:

- (1) basic tools from *logic* for making and justifying precise claims about programs;
- (2) the use of *proof assistants* to construct rigorous logical arguments;
- (3) the idea of *functional programming*, both as a method of programming that simplifies reasoning about programs and as a bridge between programming and logic;
- (4) formal techniques for *reasoning about the properties of specific programs* (e.g., the fact that a sorting function or a compiler obeys some formal specification); and
- (5) the use of *type systems* for establishing well-behavedness guarantees for *all* programs in a given programming language (e.g., the fact that well-typed Java programs cannot be subverted at runtime).

Each of these topics is easily rich enough to fill a whole course in its own right, so tackling all of them together naturally means that much will be left unsaid. Nevertheless, we hope readers will find that the themes illuminate and amplify each other and that bringing them together creates a foundation from which it will be easy to dig into any of them more deeply. Some suggestions for further reading can be found in the *Postscript* chapter.

### 1.3.1 Logic

Logic is the field of study whose subject matter is *proofs* – unassailable arguments for the truth of particular propositions. Volumes have been written about the central role of logic in computer science. Manna and Waldinger called it “the calculus of computer science,” while Halpern et al.’s paper *On the Unusual Effectiveness of Logic in Computer Science* catalogs scores of ways in which logic offers critical tools and insights. Indeed, they observe that “As a matter of fact, logic has turned out to be significantly more effective in computer science than it has been in mathematics. This is quite remarkable, especially since much of the impetus for the development of logic during the past one hundred years came from mathematics.”

In particular, the fundamental notion of inductive proofs is ubiquitous in all of computer science. You have surely seen them before, in contexts from discrete math to analysis of algorithms, but in this course we will examine them much more deeply than you have probably done so far.



### 1.3.2 Proof Assistants

The flow of ideas between logic and computer science has not been in just one direction: CS has also made important contributions to logic. One of these has been the development of software tools for helping construct proofs of logical propositions. These tools fall into two broad categories:

- *Automated theorem provers* provide “push-button” operation: you give them a proposition and they return either *true*, *false*, or *ran out of time*. Although their capabilities are limited to fairly specific sorts of reasoning, they have matured tremendously in recent years and are used now in a huge variety of settings. Examples of such tools include SAT solvers, SMT solvers, and model checkers.
- *Proof assistants* are hybrid tools that automate the more routine aspects of building proofs while depending on human guidance for more difficult aspects. Widely used proof assistants include Isabelle, Agda, Twelf, ACL2, PVS, and Coq, among many others.

This course is based around Coq, a proof assistant that has been under development, mostly in France, since 1983 and that in recent years has attracted a large community of users in both research and industry. Coq provides a rich environment for interactive development of machine-checked formal reasoning. The kernel of the Coq system is a simple proof-checker, which guarantees that only correct deduction steps are performed. On top of this kernel, the Coq environment provides high-level facilities for proof development, including powerful tactics for constructing complex proofs semi-automatically, and a large library of common definitions and lemmas.

Coq has been a critical enabler for a huge variety of work across computer science and mathematics:

- As a *platform for modeling programming languages*, it has become a standard tool for researchers who need to describe and reason about complex language definitions. It has been used, for example, to check the security of the JavaCard platform, obtaining the highest level of common criteria certification, and for formal specifications of the x86 and LLVM instruction sets and programming languages such as C.
- As an *environment for developing formally certified software*, Coq has been used, for example, to build CompCert, a fully-verified optimizing compiler for C, for proving the correctness of subtle algorithms involving floating point numbers, and as the basis for CertiCrypt, an environment for reasoning about the security of cryptographic algorithms.
- As a *realistic environment for functional programming with dependent types*, it has inspired numerous innovations. For example, the Ynot project at Harvard embedded “relational Hoare reasoning” (an extension of the *Hoare Logic* we will see later in this course) in Coq.

- As a *proof assistant for higher-order logic*, it has been used to validate a number of important results in mathematics. For example, its ability to include complex computations inside proofs made it possible to develop the first formally verified proof of the 4-color theorem. This proof had previously been controversial among mathematicians because part of it included checking a large number of configurations using a program. In the Coq formalization, everything is checked, including the correctness of the computational part. More recently, an even more massive effort led to a Coq formalization of the Feit-Thompson Theorem – the first major step in the classification of finite simple groups.

By the way, in case you're wondering about the name, here's what the official Coq web site says: "Some French computer scientists have a tradition of naming their software as animal species: Caml, Elan, Foc or Phox are examples of this tacit convention. In French, 'coq' means rooster, and it sounds like the initials of the Calculus of Constructions (CoC) on which it is based." The rooster is also the national symbol of France, and C-o-q are the first three letters of the name of Thierry Coquand, one of Coq's early developers.

### 1.3.3 Functional Programming

The term *functional programming* refers both to a collection of programming idioms that can be used in almost any programming language and to a family of programming languages designed to emphasize these idioms, including Haskell, OCaml, Standard ML, F#, Scala, Scheme, Racket, Common Lisp, Clojure, Erlang, and Coq.

Functional programming has been developed over many decades – indeed, its roots go back to Church's lambda-calculus, which was invented in the 1930s, before there were even any computers! But since the early '90s it has enjoyed a surge of interest among industrial engineers and language designers, playing a key role in high-value systems at companies like Jane St. Capital, Microsoft, Facebook, and Ericsson.

The most basic tenet of functional programming is that, as much as possible, computation should be *pure*, in the sense that the only effect of execution should be to produce a result: the computation should be free from *side effects* such as I/O, assignments to mutable variables, redirecting pointers, etc. For example, whereas an *imperative* sorting function might take a list of numbers and rearrange its pointers to put the list in order, a pure sorting function would take the original list and return a *new* list containing the same numbers in sorted order.

One significant benefit of this style of programming is that it makes programs easier to understand and reason about. If every operation on a data structure yields a new data structure, leaving the old one intact, then there is no need to worry about how that structure is being shared and whether a change by one part of the program might break an invariant that another part of the program relies on. These considerations are particularly critical in concurrent programs, where every piece of mutable state that is shared between threads is a potential source of pernicious bugs. Indeed, a large part of the recent interest in functional programming in industry is due to its simpler behavior in the presence of concurrency.

Another reason for the current excitement about functional programming is related to the first: functional programs are often much easier to parallelize than their imperative counterparts. If running a computation has no effect other than producing a result, then it does not matter *where* it is run. Similarly, if a data structure is never modified destructively, then it can be copied freely, across cores or across the network. Indeed, the “Map-Reduce” idiom, which lies at the heart of massively distributed query processors like Hadoop and is used by Google to index the entire web is a classic example of functional programming.

For this course, functional programming has yet another significant attraction: it serves as a bridge between logic and computer science. Indeed, Coq itself can be viewed as a combination of a small but extremely expressive functional programming language plus with a set of tools for stating and proving logical assertions. Moreover, when we come to look more closely, we find that these two sides of Coq are actually aspects of the very same underlying machinery – i.e., *proofs are programs*.

### 1.3.4 Program Verification

Approximately the first third of the book is devoted to developing the conceptual framework of logic and functional programming and gaining enough fluency with Coq to use it for modeling and reasoning about nontrivial artifacts. From this point on, we increasingly turn our attention to two broad topics of critical importance to the enterprise of building reliable software (and hardware): techniques for proving specific properties of particular *programs* and for proving general properties of whole programming *languages*.

For both of these, the first thing we need is a way of representing programs as mathematical objects, so we can talk about them precisely, together with ways of describing their behavior in terms of mathematical functions or relations. Our tools for these tasks are *abstract syntax* and *operational semantics*, a method of specifying programming languages by writing abstract interpreters. At the beginning, we work with operational semantics in the so-called “big-step” style, which leads to somewhat simpler and more readable definitions when it is applicable. Later on, we switch to a more detailed “small-step” style, which helps make some useful distinctions between different sorts of “nonterminating” program behaviors and is applicable to a broader range of language features, including concurrency.

The first programming language we consider in detail is *Imp*, a tiny toy language capturing the core features of conventional imperative programming: variables, assignment, conditionals, and loops. We study two different ways of reasoning about the properties of *Imp* programs.

First, we consider what it means to say that two *Imp* programs are *equivalent* in the intuitive sense that they yield the same behavior when started in any initial memory state. This notion of equivalence then becomes a criterion for judging the correctness of *metaprograms* – programs that manipulate other programs, such as compilers and optimizers. We build a simple optimizer for *Imp* and prove that it is correct.

Second, we develop a methodology for proving that particular *Imp* programs satisfy formal specifications of their behavior. We introduce the notion of *Hoare triples* – *Imp* programs annotated with pre- and post-conditions describing what should be true about the

memory in which they are started and what they promise to make true about the memory in which they terminate – and the reasoning principles of *Hoare Logic*, a “domain-specific logic” specialized for convenient compositional reasoning about imperative programs, with concepts like “loop invariant” built in.

This part of the course is intended to give readers a taste of the key ideas and mathematical tools used in a wide variety of real-world software and hardware verification tasks.

### 1.3.5 Type Systems

Our final major topic, covering approximately the last third of the course, is *type systems*, a powerful set of tools for establishing properties of *all* programs in a given language.

Type systems are the best established and most popular example of a highly successful class of formal verification techniques known as *lightweight formal methods*. These are reasoning techniques of modest power – modest enough that automatic checkers can be built into compilers, linkers, or program analyzers and thus be applied even by programmers unfamiliar with the underlying theories. Other examples of lightweight formal methods include hardware and software model checkers, contract checkers, and run-time property monitoring techniques for detecting when some component of a system is not behaving according to specification.

This topic brings us full circle: the language whose properties we study in this part, the *simply typed lambda-calculus*, is essentially a simplified model of the core of Coq itself!

## 1.4 Practicalities

### 1.4.1 Chapter Dependencies

A diagram of the dependencies between chapters and some suggested paths through the material can be found in the file *deps.html*.

### 1.4.2 System Requirements

Coq runs on Windows, Linux, and OS X. You will need:

- A current installation of Coq, available from the Coq home page. Everything should work with version 8.4. (Version 8.5 will *not* work, due to a few incompatible changes in Coq between 8.4 and 8.5.)
- An IDE for interacting with Coq. Currently, there are two choices:
  - Proof General is an Emacs-based IDE. It tends to be preferred by users who are already comfortable with Emacs. It requires a separate installation (google “Proof General”).

- CoqIDE is a simpler stand-alone IDE. It is distributed with Coq, so it should “just work” once you have Coq installed. It can also be compiled from scratch, but on some platforms this may involve installing additional packages for GUI libraries and such.

### 1.4.3 Exercises

Each chapter includes numerous exercises. Each is marked with a “star rating,” which can be interpreted as follows:

- One star: easy exercises that underscore points in the text and that, for most readers, should take only a minute or two. Get in the habit of working these as you reach them.
- Two stars: straightforward exercises (five or ten minutes).
- Three stars: exercises requiring a bit of thought (ten minutes to half an hour).
- Four and five stars: more difficult exercises (half an hour and up).

Also, some exercises are marked “advanced”, and some are marked “optional.” Doing just the non-optional, non-advanced exercises should provide good coverage of the core material. Optional exercises provide a bit of extra practice with key concepts and introduce secondary themes that may be of interest to some readers. Advanced exercises are for readers who want an extra challenge (and, in return, a deeper contact with the material).

*Please do not post solutions to the exercises in any public place:* Software Foundations is widely used both for self-study and for university courses. Having solutions easily available makes it much less useful for courses, which typically have graded homework assignments. The authors especially request that readers not post solutions to the exercises anywhere where they can be found by search engines.

### 1.4.4 Downloading the Coq Files

A tar file containing the full sources for the “release version” of these notes (as a collection of Coq scripts and HTML files) is available here:

`http://www.cis.upenn.edu/~bcpierce/sf`

If you are using the notes as part of a class, you may be given access to a locally extended version of the files, which you should use instead of the release version.

## 1.5 Note for Instructors

If you intend to use these materials in your own course, you will undoubtedly find things you’d like to change, improve, or add. Your contributions are welcome!

Please send an email to Benjamin Pierce describing yourself and how you would like to use the materials, and including the result of doing “`htpasswd -s -n NAME`”, where NAME is your preferred user name. We’ll set you up with read/write access to our subversion repository and developers’ mailing list; in the repository you’ll find a *README* with further instructions.

## 1.6 Translations

Thanks to the efforts of a team of volunteer translators, *Software Foundations* can now be enjoyed in Japanese at <http://proofcafe.org/sf>. A Chinese translation is underway.

*Date* : 2016 – 03 – 04 09 : 33 : 20 – 0500 (*Fri*, 04 *Mar* 2016)

# Chapter 2

## Library Basics

### 2.1 Basics: Functional Programming in Coq

Definition admit  $\{T: \text{Type}\} : T.$  *Admitted.*

### 2.2 Introduction

The functional programming style brings programming closer to simple, everyday mathematics: If a procedure or method has no side effects, then (ignoring efficiency) all we need to understand about it is how it maps inputs to outputs – that is, we can think of it as just a concrete method for computing a mathematical function. This is one sense of the word “functional” in “functional programming.” The direct connection between programs and simple mathematical objects supports both formal correctness proofs and sound informal reasoning about program behavior.

The other sense in which functional programming is “functional” is that it emphasizes the use of functions (or methods) as *first-class* values – i.e., values that can be passed as arguments to other functions, returned as results, included in data structures, etc. The recognition that functions can be treated as data in this way enables a host of useful and powerful idioms.

Other common features of functional languages include *algebraic data types* and *pattern matching*, which make it easy to construct and manipulate rich data structures, and sophisticated *polymorphic type systems* supporting abstraction and code reuse. Coq shares all of these features.

The first half of this chapter introduces the most essential elements of Coq’s functional programming language. The second half introduces some basic *tactics* that can be used to prove simple properties of Coq programs.

## 2.3 Enumerated Types

One unusual aspect of Coq is that its set of built-in features is *extremely* small. For example, instead of providing the usual palette of atomic data types (booleans, integers, strings, etc.), Coq offers a powerful mechanism for defining new data types from scratch, from which all these familiar types arise as instances.

Naturally, the Coq distribution comes with an extensive standard library providing definitions of booleans, numbers, and many common data structures like lists and hash tables. But there is nothing magic or primitive about these library definitions. To illustrate this, we will explicitly recapitulate all the definitions we need in this course, rather than just getting them implicitly from the library.

To see how this definition mechanism works, let's start with a very simple example.

### 2.3.1 Days of the Week

The following declaration tells Coq that we are defining a new set of data values – a *type*.

Inductive **day** : Type :=

```
| monday : day
| tuesday : day
| wednesday : day
| thursday : day
| friday : day
| saturday : day
| sunday : day.
```

The type is called **day**, and its members are **monday**, **tuesday**, etc. The second and following lines of the definition can be read “**monday** is a **day**, **tuesday** is a **day**, etc.”

Having defined **day**, we can write functions that operate on days.

Definition next\_weekday (*d*:**day**) : **day** :=

```
match d with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end.
```

One thing to note is that the argument and return types of this function are explicitly declared. Like most functional programming languages, Coq can often figure out these types for itself when they are not given explicitly – i.e., it performs *type inference* – but we'll include them to make reading easier.



Having defined a function, we should check that it works on some examples. There are actually three different ways to do this in Coq.

First, we can use the command `Compute` to evaluate a compound expression involving `next_weekday`.

```
Compute (next_weekday friday).
```

```
Compute (next_weekday (next_weekday saturday)).
```

(We show Coq’s responses in comments, but, if you have a computer handy, this would be an excellent moment to fire up the Coq interpreter under your favorite IDE – either CoqIde or Proof General – and try this for yourself. Load this file, *Basics.v*, from the book’s accompanying Coq sources, find the above example, submit it to Coq, and observe the result.)

Second, we can record what we *expect* the result to be in the form of a Coq example:

Example `test_next_weekday`:

```
(next_weekday (next_weekday saturday)) = tuesday.
```

This declaration does two things: it makes an assertion (that the second weekday after `saturday` is `tuesday`), and it gives the assertion a name that can be used to refer to it later.

Having made the assertion, we can also ask Coq to verify it, like this:

```
Proof. simpl. reflexivity. Qed.
```

The details are not important for now (we’ll come back to them in a bit), but essentially this can be read as “The assertion we’ve just made can be proved by observing that both sides of the equality evaluate to the same thing, after some simplification.”

Third, we can ask Coq to *extract*, from our **Definition**, a program in some other, more conventional, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler. This facility is very interesting, since it gives us a way to construct *fully certified* programs in mainstream languages. Indeed, this is one of the main uses for which Coq was developed. We’ll come back to this topic in later chapters.

## 2.3.2 Booleans

In a similar way, we can define the standard type **bool** of booleans, with members **true** and **false**.

```
Inductive bool : Type :=  
  | true : bool  
  | false : bool.
```

Although we are rolling our own booleans here for the sake of building up everything from scratch, Coq does, of course, provide a default implementation of the booleans in its standard library, together with a multitude of useful functions and lemmas. (Take a look at `Coq.Init.Datatypes` in the Coq library documentation if you’re interested.) Whenever possible, we’ll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.

Functions over booleans can be defined in the same way as above:

```
Definition negb (b:bool) : bool :=  
  match b with  
  | true => false  
  | false => true  
  end.
```

```
Definition andb (b1:bool) (b2:bool) : bool :=  
  match b1 with  
  | true => b2  
  | false => false  
  end.
```

```
Definition orb (b1:bool) (b2:bool) : bool :=  
  match b1 with  
  | true => true  
  | false => b2  
  end.
```

The last two illustrate Coq’s syntax for multi-argument function definitions. The corresponding multi-argument application syntax is illustrated by the following four “unit tests,” which constitute a complete specification – a truth table – for the `orb` function:

```
Example test_orb1: (orb true false) = true.  
Proof. simpl. reflexivity. Qed.  
Example test_orb2: (orb false false) = false.  
Proof. simpl. reflexivity. Qed.  
Example test_orb3: (orb false true) = true.  
Proof. simpl. reflexivity. Qed.  
Example test_orb4: (orb true true) = true.  
Proof. simpl. reflexivity. Qed.
```

We can also introduce some familiar syntax for the boolean operations we have just defined. The `Infix` command defines new, infix notation for an existing definition.

```
Infix "&&" := andb.  
Infix "||" := orb.
```

```
Example test_orb5: false || false || true = true.  
Proof. simpl. reflexivity. Qed.
```

*A note on notation:* In `.v` files, we use square brackets to delimit fragments of Coq code within comments; this convention, also used by the *coqdoc* documentation tool, keeps them visually separate from the surrounding text. In the html version of the files, these pieces of text appear in a *different font*.

The special phrases *Admitted* and `admit` can be used as a placeholder for an incomplete definition or proof. We’ll use them in exercises, to indicate the parts that we’re leaving for you – i.e., your job is to replace `admit` or *Admitted* with real definitions or proofs.

**Exercise: 1 star (nandb)** Remove `admit` and complete the definition of the following function; then make sure that the `Example` assertions below can each be verified by Coq. (Remove “*Admitted.*” and fill in each proof, following the model of the `orb` tests above.) The function should return `true` if either or both of its inputs are `false`.

```
Definition nandb (b1:bool) (b2:bool) : bool :=  
  admit.
```

```
Example test_nandb1: (nandb true false) = true.  
  Admitted.
```

```
Example test_nandb2: (nandb false false) = true.  
  Admitted.
```

```
Example test_nandb3: (nandb false true) = true.  
  Admitted.
```

```
Example test_nandb4: (nandb true true) = false.  
  Admitted.
```

□

**Exercise: 1 star (andb3)** Do the same for the `andb3` function below. This function should return `true` when all of its inputs are `true`, and `false` otherwise.

```
Definition andb3 (b1:bool) (b2:bool) (b3:bool) : bool :=  
  admit.
```

```
Example test_andb31: (andb3 true true true) = true.  
  Admitted.
```

```
Example test_andb32: (andb3 false true true) = false.  
  Admitted.
```

```
Example test_andb33: (andb3 true false true) = false.  
  Admitted.
```

```
Example test_andb34: (andb3 true true false) = false.  
  Admitted.
```

□

### 2.3.3 Function Types

Every expression in Coq has a type, describing what sort of thing it computes. The `Check` command asks Coq to print the type of an expression.

For example, the type of `negb true` is `bool`.

```
Check true.
```

```
Check (negb true).
```

Functions like `negb` itself are also data values, just like `true` and `false`. Their types are called *function types*, and they are written with arrows.

```
Check negb.
```

The type of `negb`, written `bool → bool` and pronounced “**bool** arrow **bool**,” can be read, “Given an input of type **bool**, this function produces an output of type **bool**.” Similarly, the type of `andb`, written `bool → bool → bool`, can be read, “Given two inputs, both of type **bool**, this function produces an output of type **bool**.”

### 2.3.4 Modules

Coq provides a *module system*, to aid in organizing large developments. In this course we won’t need most of its features, but one is useful: If we enclose a collection of declarations between `Module X` and `End X` markers, then, in the remainder of the file after the `End`, these definitions are referred to by names like `X.foo` instead of just `foo`. Here, we use this feature to introduce the definition of the type `nat` in an inner module so that it does not interfere with the one from the standard library, which comes with a bit of special notational magic.

`Module PLAYGROUND1.`

### 2.3.5 Numbers

The types we have defined so far are examples of “enumerated types”: their definitions explicitly enumerate a finite set of elements. A more interesting way of defining a type is to give a collection of *inductive rules* describing its elements. For example, we can define the natural numbers as follows:

```
Inductive nat : Type :=
| O : nat
| S : nat → nat.
```

The clauses of this definition can be read:

- `O` is a natural number (note that this is the letter “O,” not the numeral “0”).
- `S` is a “constructor” that takes a natural number and yields another one – that is, if `n` is a natural number, then `S n` is too.

Let’s look at this in a little more detail.

Every inductively defined set (**day**, **nat**, **bool**, etc.) is actually a set of *expressions*. The definition of `nat` says how expressions in the set `nat` can be constructed:

- the expression `O` belongs to the set `nat`;
- if `n` is an expression belonging to the set `nat`, then `S n` is also an expression belonging to the set `nat`; and
- expressions formed in these two ways are the only ones belonging to the set `nat`.

The same rules apply for our definitions of **day** and **bool**. The annotations we used for their constructors are analogous to the one for the **O** constructor, indicating that they don't take any arguments.

These three conditions are the precise force of the **Inductive** declaration. They imply that the expression **O**, the expression **S O**, the expression **S (S O)**, the expression **S (S (S O))**, and so on all belong to the set **nat**, while other expressions like **true**, **andb true false**, and **S (S false)** do not.

We can write simple functions that pattern match on natural numbers just as we did above – for example, the predecessor function:

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.
```

The second branch can be read: “if **n** has the form **S n'** for some **n'**, then return **n'**.”

End PLAYGROUND1.

```
Definition minustwo (n : nat) : nat :=
  match n with
  | O => O
  | S O => O
  | S (S n') => n'
  end.
```

Because natural numbers are such a pervasive form of data, Coq provides a tiny bit of built-in magic for parsing and printing them: ordinary arabic numerals can be used as an alternative to the “unary” notation defined by the constructors **S** and **O**. Coq prints numbers in arabic form by default:

```
Check (S (S (S (S O)))).
Compute (minustwo 4).
```

The constructor **S** has the type **nat** → **nat**, just like the functions **minustwo** and **pred**:

```
Check S.
Check pred.
Check minustwo.
```

These are all things that can be applied to a number to yield a number. However, there is a fundamental difference between the first one and the other two: functions like **pred** and **minustwo** come with *computation rules* – e.g., the definition of **pred** says that **pred 2** can be simplified to **1** – while the definition of **S** has no such behavior attached. Although it is like a function in the sense that it can be applied to an argument, it does not *do* anything at all!

For most function definitions over numbers, just pattern matching is not enough: we also need recursion. For example, to check that a number **n** is even, we may need to recursively check whether **n-2** is even. To write such functions, we use the keyword **Fixpoint**.

```

Fixpoint evenb (n:nat) : bool :=
  match n with
  | 0 => true
  | S 0 => false
  | S (S n') => evenb n'
  end.

```

We can define `oddb` by a similar `Fixpoint` declaration, but here is a simpler definition that is a bit easier to work with:

```

Definition oddb (n:nat) : bool := negb (evenb n).

```

```

Example test_oddb1: oddb 1 = true.

```

```

Proof. simpl. reflexivity. Qed.

```

```

Example test_oddb2: oddb 4 = false.

```

```

Proof. simpl. reflexivity. Qed.

```

(You will notice if you step through these proofs that `simpl` actually has no effect on the goal – all of the work is done by `reflexivity`. We’ll see more about why that is shortly.)

Naturally, we can also define multi-argument functions by recursion.

```

Module PLAYGROUND2.

```

```

Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | 0 => m
  | S n' => S (plus n' m)
  end.

```

Adding three to two now gives us five, as we’d expect.

```

Compute (plus 3 2).

```

The simplification that Coq performs to reach this conclusion can be visualized as follows:

As a notational convenience, if two or more arguments have the same type, they can be written together. In the following definition, `(n m : nat)` means just the same as if we had written `(n : nat) (m : nat)`.

```

Fixpoint mult (n m : nat) : nat :=
  match n with
  | 0 => 0
  | S n' => plus m (mult n' m)
  end.

```

```

Example test_mult1: (mult 3 3) = 9.

```

```

Proof. simpl. reflexivity. Qed.

```

You can match two expressions at once by putting a comma between them:

```

Fixpoint minus (n m:nat) : nat :=

```

```

match n, m with
| O, _ => O
| S -, O => n
| S n', S m' => minus n' m'
end.

```

The `_` in the first line is a *wildcard pattern*. Writing `_` in a pattern is the same as writing some variable that doesn't get used on the right-hand side. This avoids the need to invent a bogus variable name.

End PLAYGROUND2.

```

Fixpoint exp (base power : nat) : nat :=
  match power with
  | O => S O
  | S p => mult base (exp base p)
  end.

```

**Exercise: 1 star (factorial)** Recall the standard mathematical factorial function:

$$\begin{aligned} \text{factorial}(0) &= 1 \\ \text{factorial}(n) &= n * \text{factorial}(n-1) \quad (\text{if } n > 0) \end{aligned}$$

Translate this into Coq.

```

Fixpoint factorial (n:nat) : nat :=
  admit.

```

Example test\_factorial1: (factorial 3) = 6.

*Admitted.*

Example test\_factorial2: (factorial 5) = (mult 10 12).

*Admitted.*

□

We can make numerical expressions a little easier to read and write by introducing *notations* for addition, multiplication, and subtraction.

```

Notation "x + y" := (plus x y)
                      (at level 50, left associativity)
                      : nat_scope.

```

```

Notation "x - y" := (minus x y)
                      (at level 50, left associativity)
                      : nat_scope.

```

```

Notation "x * y" := (mult x y)
                      (at level 40, left associativity)
                      : nat_scope.

```

Check ((0 + 1) + 1).

(The `level`, `associativity`, and `nat_scope` annotations control how these notations are treated by Coq’s parser. The details are not important, but interested readers can refer to the optional “More on Notation” section at the end of this chapter.)

Note that these do not change the definitions we’ve already made: they are simply instructions to the Coq parser to accept  $x + y$  in place of `plus x y` and, conversely, to the Coq pretty-printer to display `plus x y` as  $x + y$ .

When we say that Coq comes with nothing built-in, we really mean it: even equality testing for numbers is a user-defined operation!

The `beq_nat` function tests natural numbers for equality, yielding a boolean. Note the use of nested matches (we could also have used a simultaneous match, as we did in `minus`.)

```
Fixpoint beq_nat (n m : nat) : bool :=
  match n with
  | O => match m with
        | O => true
        | S m' => false
        end
  | S n' => match m with
            | O => false
            | S m' => beq_nat n' m'
            end
  end.
```

The `leb` function tests whether its first argument is less than or equal to its second argument, yielding a boolean.

```
Fixpoint leb (n m : nat) : bool :=
  match n with
  | O => true
  | S n' =>
    match m with
    | O => false
    | S m' => leb n' m'
    end
  end.
```

Example test\_leb1: `(leb 2 2) = true`.

Proof. `simpl. reflexivity. Qed`.

Example test\_leb2: `(leb 2 4) = true`.

Proof. `simpl. reflexivity. Qed`.

Example test\_leb3: `(leb 4 2) = false`.

Proof. `simpl. reflexivity. Qed`.

**Exercise: 1 star (`blt_nat`)** The `blt_nat` function tests natural numbers for less-than, yielding a boolean. Instead of making up a new `Fixpoint` for this one, define it in terms of



a previously defined function.

```
Definition blt_nat (n m : nat) : bool :=  
  admit.
```

```
Example test_blt_nat1: (blt_nat 2 2) = false.  
  Admitted.
```

```
Example test_blt_nat2: (blt_nat 2 4) = true.  
  Admitted.
```

```
Example test_blt_nat3: (blt_nat 4 2) = false.  
  Admitted.
```

□

## 2.4 Proof by Simplification

Now that we’ve defined a few datatypes and functions, let’s turn to stating and proving properties of their behavior. Actually, we’ve already started doing this: each **Example** in the previous sections makes a precise claim about the behavior of some function on some particular inputs. The proofs of these claims were always the same: use `simpl` to simplify both sides of the equation, then use `reflexivity` to check that both sides contain identical values.

The same sort of “proof by simplification” can be used to prove more interesting properties as well. For example, the fact that 0 is a “neutral element” for `+` on the left can be proved just by observing that `0 + n` reduces to `n` no matter what `n` is, a fact that can be read directly off the definition of `plus`.

```
Theorem plus_O_n : ∀ n : nat, 0 + n = n.
```

Proof.

```
  intros n. simpl. reflexivity. Qed.
```

(You may notice that the above statement looks different in the `.v` file in your IDE than it does in the HTML rendition in your browser, if you are viewing both. In `.v` files, we write the  $\forall$  universal quantifier using the reserved identifier “forall.” When the `.v` files are converted to HTML, this gets transformed into an upside-down-A symbol.)

This is a good place to mention that `reflexivity` is a bit more powerful than we have admitted. In the examples we have seen, the calls to `simpl` were actually not needed, because `reflexivity` can perform some simplification automatically when checking that two sides are equal; `simpl` was just added so that we could see the intermediate state – after simplification but before finishing the proof. Here is a shorter proof of the theorem:

```
Theorem plus_O_n' : ∀ n : nat, 0 + n = n.
```

Proof.

```
  intros n. reflexivity. Qed.
```

Moreover, it will be useful later to know that `reflexivity` does somewhat *more* simplification than `simpl` does – for example, it tries “unfolding” defined terms, replacing them with

their right-hand sides. The reason for this difference is that, if `reflexivity` succeeds, the whole goal is finished and we don't need to look at whatever expanded expressions `reflexivity` has created by all this simplification and unfolding; by contrast, `simpl` is used in situations where we may have to read and understand the new goal that it creates, so we would not want it blindly expanding definitions and leaving the goal in a messy state.

The form of the theorem we just stated and its proof are almost exactly the same as the simpler examples we saw earlier; there are just a few differences.

First, we've used the keyword `Theorem` instead of `Example`. This difference is purely a matter of style; the keywords `Example` and `Theorem` (and a few others, including `Lemma`, `Fact`, and `Remark`) mean exactly the same thing to Coq.

Second, we've added the quantifier  $\forall n:\text{nat}$ , so that our theorem talks about *all* natural numbers  $n$ . In order to prove theorems of this form, we need to be able to reason by *assuming* the existence of an arbitrary natural number  $n$ . This is achieved in the proof by `intros n`, which moves the quantifier from the goal to a *context* of current assumptions. In effect, we start the proof by saying "Suppose  $n$  is some arbitrary number..."

The keywords `intros`, `simpl`, and `reflexivity` are examples of *tactics*. A tactic is a command that is used between `Proof` and `Qed` to guide the process of checking some claim we are making. We will see several more tactics in the rest of this chapter and yet more in future chapters.

Other similar theorems can be proved with the same pattern.

`Theorem plus_1_l :  $\forall n:\text{nat}, 1 + n = S\ n$ .`

`Proof.`

`intros n. reflexivity. Qed.`

`Theorem mult_0_l :  $\forall n:\text{nat}, 0 \times n = 0$ .`

`Proof.`

`intros n. reflexivity. Qed.`

The `_l` suffix in the names of these theorems is pronounced "on the left."

It is worth stepping through these proofs to observe how the context and the goal change. You may want to add calls to `simpl` before `reflexivity` to see the simplifications that Coq performs on the terms before checking that they are equal.

Although simplification is powerful enough to prove some fairly general facts, there are many statements that cannot be handled by simplification alone. For instance, we cannot use it to prove that 0 is also a neutral element for  $+$  *on the right*.

`Theorem plus_n_0 :  $\forall n, n = n + 0$ .`

`Proof.`

`intros n. simpl.`

(Can you explain why this happens? Step through both proofs with Coq and notice how the goal and context change.)

When stuck in the middle of a proof, we can use the `Abort` command to give up on it for the moment.

`Abort.`

The next chapter will introduce *induction*, a powerful technique that can be used for proving this goal. For the moment, though, let's look at a few more simple tactics.

## 2.5 Proof by Rewriting

This theorem is a bit more interesting than the others we've seen:

**Theorem** `plus_id_example` :  $\forall n\ m:\text{nat},$

$n = m \rightarrow$

$n + n = m + m.$

Instead of making a universal claim about all numbers `n` and `m`, it talks about a more specialized property that only holds when `n = m`. The arrow symbol is pronounced “implies.”

As before, we need to be able to reason by assuming the existence of some numbers `n` and `m`. We also need to assume the hypothesis `n = m`. The `intros` tactic will serve to move all three of these from the goal into assumptions in the current context.

Since `n` and `m` are arbitrary numbers, we can't just use simplification to prove this theorem. Instead, we prove it by observing that, if we are assuming `n = m`, then we can replace `n` with `m` in the goal statement and obtain an equality with the same expression on both sides. The tactic that tells Coq to perform this replacement is called `rewrite`.

**Proof.**

`intros n m.`

`intros H.`

`rewrite  $\rightarrow$  H.`

`reflexivity. Qed.`

The first line of the proof moves the universally quantified variables `n` and `m` into the context. The second moves the hypothesis `n = m` into the context and gives it the name `H`. The third tells Coq to rewrite the current goal (`n + n = m + m`) by replacing the left side of the equality hypothesis `H` with the right side.

(The arrow symbol in the `rewrite` has nothing to do with implication: it tells Coq to apply the rewrite from left to right. To rewrite from right to left, you can use `rewrite  $\leftarrow$` . Try making this change in the above proof and see what difference it makes.)

**Exercise: 1 star (plus\_id\_exercise)** Remove “*Admitted.*” and fill in the proof.

**Theorem** `plus_id_exercise` :  $\forall n\ m\ o : \text{nat},$

$n = m \rightarrow m = o \rightarrow n + m = m + o.$

**Proof.**

*Admitted.*

□

The *Admitted* command tells Coq that we want to skip trying to prove this theorem and just accept it as a given. This can be useful for developing longer proofs, since we can state subsidiary lemmas that we believe will be useful for making some larger argument,

use *Admitted* to accept them on faith for the moment, and continue working on the main argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped. Be careful, though: every time you say *Admitted* (or `admit`) you are leaving a door open for total nonsense to enter Coq’s nice, rigorous, formally checked world!

We can also use the `rewrite` tactic with a previously proved theorem instead of a hypothesis from the context.

```
Theorem mult_0_plus : ∀ n m : nat,
  (0 + n) × m = n × m.
```

Proof.

```
  intros n m.
  rewrite → plus_0_n.
  reflexivity. Qed.
```

**Exercise: 2 stars (mult\_S\_1)** Theorem `mult_S_1` : ∀ n m : nat,

```
  m = S n →
  m × (1 + n) = m × m.
```

Proof.

*Admitted.*

□

## 2.6 Proof by Case Analysis

Of course, not everything can be proved by simple calculation and rewriting: In general, unknown, hypothetical values (arbitrary numbers, booleans, lists, etc.) can block simplification. For example, if we try to prove the following fact using the `simpl` tactic as above, we get stuck.

```
Theorem plus_1_neq_0_firsttry : ∀ n : nat,
  beq_nat (n + 1) 0 = false.
```

Proof.

```
  intros n.
  simpl. Abort.
```

The reason for this is that the definitions of both `beq_nat` and `+` begin by performing a `match` on their first argument. But here, the first argument to `+` is the unknown number `n` and the argument to `beq_nat` is the compound expression `n + 1`; neither can be simplified.

To make progress, we need to consider the possible forms of `n` separately. If `n` is `O`, then we can calculate the final result of `beq_nat (n + 1) 0` and check that it is, indeed, `false`. And if `n = S n'` for some `n'`, then, although we don’t know exactly what number `n + 1` yields, we can calculate that, at least, it will begin with one `S`, and this is enough to calculate that, again, `beq_nat (n + 1) 0` will yield `false`.

The tactic that tells Coq to consider, separately, the cases where `n = O` and where `n = S n'` is called `destruct`.

Theorem `plus_1_neq_0` :  $\forall n : \text{nat}$ ,  
`beq_nat (n + 1) 0 = false`.

Proof.

```
intros n. destruct n as [| n'].
- reflexivity.
- reflexivity. Qed.
```

The `destruct` generates *two* subgoals, which we must then prove, separately, in order to get Coq to accept the theorem. The annotation “`as [| n']`” is called an *intro pattern*. It tells Coq what variable names to introduce in each subgoal. In general, what goes between the square brackets is a *list of lists* of names, separated by `|`. In this case, the first component is empty, since the `O` constructor is nullary (it doesn’t have any arguments). The second component gives a single name, `n'`, since `S` is a unary constructor.

The `-` signs on the second and third lines are called *bullets*, and they mark the parts of the proof that correspond to each generated subgoal. The proof script that comes after a bullet is the entire proof for a subgoal. In this example, each of the subgoals is easily proved by a single use of `reflexivity`, which itself performs some simplification – e.g., the first one simplifies `beq_nat (S n' + 1) 0` to `false` by first rewriting `(S n' + 1)` to `S (n' + 1)`, then unfolding `beq_nat`, and then simplifying the `match`.

Marking cases with bullets is entirely optional: if bullets are not present, Coq simply asks you to prove each subgoal in sequence, one at a time. But it is a good idea to use bullets. For one thing, they make the structure of a proof apparent, making it more readable. Also, bullets instruct Coq to ensure that a subgoal is complete before trying to verify the next one, preventing proofs for different subgoals from getting mixed up. These issues become especially important in large developments, where fragile proofs lead to long debugging sessions.

There are no hard and fast rules for how proofs should be formatted in Coq – in particular, where lines should be broken and how sections of the proof should be indented to indicate their nested structure. However, if the places where multiple subgoals are generated are marked with explicit bullets at the beginning of lines, then the proof will be readable almost no matter what choices are made about other aspects of layout.

This is also a good place to mention one other piece of somewhat obvious advice about line lengths. Beginning Coq users sometimes tend to the extremes, either writing each tactic on its own line or writing entire proofs on one line. Good style lies somewhere in the middle. One reasonable convention is to limit yourself to 80-character lines.

The `destruct` tactic can be used with any inductively defined datatype. For example, we use it next to prove that boolean negation is involutive – i.e., that negation is its own inverse.

Theorem `negb_involutive` :  $\forall b : \text{bool}$ ,  
`negb (negb b) = b`.

Proof.

```
intros b. destruct b.
- reflexivity.
```

- reflexivity. Qed.

Note that the `destruct` here has no `as` clause because none of the subcases of the `destruct` need to bind any variables, so there is no need to specify any names. (We could also have written `as []`, or `as []`.) In fact, we can omit the `as` clause from *any* `destruct` and Coq will fill in variable names automatically. This is generally considered bad style, since Coq often makes confusing choices of names when left to its own devices.

It is sometimes useful to invoke `destruct` inside a subgoal, generating yet more proof obligations. In this case, we use different kinds of bullets to mark goals on different “levels.” For example:

Theorem `andb_commutative` :  $\forall b\ c, \text{andb } b\ c = \text{andb } c\ b$ .

Proof.

```
intros b c. destruct b.  
- destruct c.  
  + reflexivity.  
  + reflexivity.  
- destruct c.  
  + reflexivity.  
  + reflexivity.
```

Qed.

Each pair of calls to `reflexivity` corresponds to the subgoals that were generated after the execution of the `destruct c` line right above it. Besides - and +, Coq proofs can also use  $\times$  (asterisk) as a third kind of bullet. If we ever encounter a proof that generates more than three levels of subgoals, we can also enclose individual subgoals in curly braces (`{ ... }`):

Theorem `andb_commutative'` :  $\forall b\ c, \text{andb } b\ c = \text{andb } c\ b$ .

Proof.

```
intros b c. destruct b.  
{ destruct c.  
  { reflexivity. }  
  { reflexivity. } }  
{ destruct c.  
  { reflexivity. }  
  { reflexivity. } }
```

Qed.

Since curly braces mark both the beginning and the end of a proof, they can be used for multiple subgoal levels, as this example shows. Furthermore, curly braces allow us to reuse the same bullet shapes at multiple levels in a proof:

Theorem `andb3_exchange` :

$\forall b\ c\ d, \text{andb } (\text{andb } b\ c)\ d = \text{andb } (\text{andb } b\ d)\ c$ .

Proof.

```
intros b c d. destruct b.
```

```

- destruct c.
  { destruct d.
    - reflexivity.
    - reflexivity. }
  { destruct d.
    - reflexivity.
    - reflexivity. }
- destruct c.
  { destruct d.
    - reflexivity.
    - reflexivity. }
  { destruct d.
    - reflexivity.
    - reflexivity. }

```

Qed.

Before closing the chapter, let's mention one final convenience. As you may have noticed, many proofs perform case analysis on a variable right after introducing it: `intros x y. destruct y as |y.` This pattern is so common that Coq provides a shorthand for it: we can perform case analysis on a variable when introducing it by using an `intro` pattern instead of a variable name. For instance, here is a shorter proof of the `plus_1_neq_0` theorem above.

Theorem `plus_1_neq_0'` :  $\forall n : \text{nat},$   
`beq_nat (n + 1) 0 = false.`

Proof.

```

intros [|n].
- reflexivity.
- reflexivity. Qed.

```

If there are no arguments to name, we can just write `||`.

Theorem `andb_commutative''` :  
 $\forall b\ c, \text{andb } b\ c = \text{andb } c\ b.$

Proof.

```

intros [| |].
- reflexivity.
- reflexivity.
- reflexivity.
- reflexivity.

```

Qed.

**Exercise: 2 stars (`andb_true_elim2`)** Prove the following claim, marking cases (and subcases) with bullets when you use `destruct`.

Theorem `andb_true_elim2` :  $\forall b\ c : \text{bool},$   
`andb b c = true  $\rightarrow$  c = true.`

*Admitted.*

**Exercise: 1 star (zero\_nbeq\_plus\_1)** Theorem zero\_nbeq\_plus\_1 :  $\forall n : \text{nat}, \text{beq\_nat } 0 (n + 1) = \text{false}$ .

*Admitted.*

### 2.6.1 More on Notation (Optional)

(In general, sections marked Optional are not needed to follow the rest of the book, except possibly other Optional sections. On a first reading, you might want to skim these sections so that you know what's there for future reference.)

Recall the notation definitions for infix plus and times:

Notation " $x + y$ " := (plus  $x$   $y$ )  
 (at level 50, left associativity)  
 : *nat\_scope*.

Notation " $x * y$ " := (**mult**  $x$   $y$ )  
 (at level 40, left associativity)  
 : *nat\_scope*.

For each notation symbol in Coq, we can specify its *precedence level* and its *associativity*. The precedence level `n` is specified by writing `at level n`; this helps Coq parse compound expressions. The associativity setting helps to disambiguate expressions containing multiple occurrences of the same symbol. For example, the parameters specified above for `+` and `*` say that the expression `1+2*3*4` is shorthand for `(1+((2*3)*4))`. Coq uses precedence levels from 0 to 100, and *left*, *right*, or *no* associativity. We will see more examples of this later, e.g., in the `Lists` chapter.

Each notation symbol is also associated with a *notation scope*. Coq tries to guess what scope you mean from context, so when you write `S(O×O)` it guesses *nat\_scope*, but when you write the cartesian product (tuple) type `bool×bool` it guesses *type\_scope*. Occasionally, you may have to help it out with percent-notation by writing `(x×y)%nat`, and sometimes in Coq’s feedback to you it will use `%nat` to indicate what scope a notation is in.

Notation scopes also apply to numeral notation (3, 4, 5, etc.), so you may sometimes see `0%nat`, which means 0 (the natural number 0 that we’re using in this chapter), or `0%Z`, which means the Integer zero (which comes from a different part of the standard library).

### 2.6.2 Fixpoints and Structural Recursion (Optional)

Here is a copy of the definition of addition:



```

Fixpoint plus' (n : nat) (m : nat) : nat :=
  match n with
  | 0 => m
  | S n' => S (plus' n' m)
  end.

```

When Coq checks this definition, it notes that `plus'` is “decreasing on 1st argument.” What this means is that we are performing a *structural recursion* over the argument `n` – i.e., that we make recursive calls only on strictly smaller values of `n`. This implies that all calls to `plus'` will eventually terminate. Coq demands that some argument of *every* `Fixpoint` definition is “decreasing.”

This requirement is a fundamental feature of Coq’s design: In particular, it guarantees that every function that can be defined in Coq will terminate on all inputs. However, because Coq’s “decreasing analysis” is not very sophisticated, it is sometimes necessary to write functions in slightly unnatural ways.

**Exercise: 2 stars, optional (decreasing)** To get a concrete sense of this, find a way to write a sensible `Fixpoint` definition (of a simple function on numbers, say) that *does* terminate on all inputs, but that Coq will reject because of this restriction.

□

## 2.7 More Exercises

**Exercise: 2 stars (boolean\_functions)** Use the tactics you have learned so far to prove the following theorem about boolean functions.

Theorem `identity_fn_applied_twice` :

$$\begin{aligned} &\forall (f : \mathbf{bool} \rightarrow \mathbf{bool}), \\ &(\forall (x : \mathbf{bool}), f\ x = x) \rightarrow \\ &\forall (b : \mathbf{bool}), f\ (f\ b) = b. \end{aligned}$$

Proof.

*Admitted.*

Now state and prove a theorem `negation_fn_applied_twice` similar to the previous one but where the second hypothesis says that the function `f` has the property that `f x = negb x`.

□

**Exercise: 2 stars (andb\_eq\_orb)** Prove the following theorem. (You may want to first prove a subsidiary lemma or two. Alternatively, remember that you do not have to introduce all hypotheses at the same time.)

Theorem `andb_eq_orb` :

$$\begin{aligned} &\forall (b\ c : \mathbf{bool}), \\ &(\mathbf{andb}\ b\ c = \mathbf{orb}\ b\ c) \rightarrow \end{aligned}$$

$b = c.$

Proof.

*Admitted.*

□

**Exercise: 3 stars (binary)** Consider a different, more efficient representation of natural numbers using a binary rather than unary system. That is, instead of saying that each natural number is either zero or the successor of a natural number, we can say that each binary number is either

- zero,
- twice a binary number, or
- one more than twice a binary number.

(a) First, write an inductive definition of the type *bin* corresponding to this description of binary numbers.

(Hint: Recall that the definition of `nat` from class, `Inductive nat : Type := | O : nat | S : nat -> nat.` says nothing about what `O` and `S` “mean.” It just says “`O` is in the set called `nat`, and if `n` is in the set then so is `S n`.” The interpretation of `O` as zero and `S` as successor/plus one comes from the way that we *use* `nat` values, by writing functions to do things with them, proving things about them, and so on. Your definition of *bin* should be correspondingly simple; it is the functions you will write next that will give it mathematical meaning.)

(b) Next, write an increment function *incr* for binary numbers, and a function *bin\_to\_nat* to convert binary numbers to unary numbers.

(c) Write five unit tests *test\_bin\_incr1*, *test\_bin\_incr2*, etc. for your increment and binary-to-unary functions. Notice that incrementing a binary number and then converting it to unary should yield the same result as first converting it to unary and then incrementing.

□

*Date* : 2016 – 02 – 18 18 : 17 : 32 – 0500(Thu, 18Feb2016)

# Chapter 3

## Library Induction

### 3.1 Induction: Proof by Induction

The next line imports all of our definitions from the previous chapter.

Require Export Basics.

For it to work, you need to use *coqc* to compile *Basics.v* into *Basics.vo*. This is like making a .class file from a .java file, or a .o file from a .c file.

Here are two ways to compile your code:

- In CoqIDE:

Open *Basics.v*. In the “Compile” menu, click on “Compile Buffer”.

- From the command line:

Run *coqc Basics.v*

### 3.2 Proof by Induction

We proved in the last chapter that 0 is a neutral element for + on the left using an easy argument based on simplification. The fact that it is also a neutral element on the *right*...

Theorem plus\_n\_O\_firsttry :  $\forall n:\text{nat}$ ,  
 $n = n + 0$ .

... cannot be proved in the same simple way. Just applying **reflexivity** doesn’t work, since the **n** in  $n + 0$  is an arbitrary unknown number, so the **match** in the definition of + can’t be simplified.

Proof.

```
intros n.  
simpl. Abort.
```

And reasoning by cases using `destruct n` doesn't get us much further: the branch of the case analysis where we assume  $n = 0$  goes through fine, but in the branch where  $n = S\ n'$  for some  $n'$  we get stuck in exactly the same way. We could use `destruct n'` to get one step further, but, since  $n$  can be arbitrarily large, if we try to keep on like this we'll never be done.

Theorem `plus_n_O_secondtry` :  $\forall n:\text{nat},$   
 $n = n + 0.$

Proof.

```
intros n. destruct n as [| n'].
-
  reflexivity.
-
  simpl. Abort.
```

To prove interesting facts about numbers, lists, and other inductively defined sets, we usually need a more powerful reasoning principle: *induction*.

Recall (from high school, a discrete math course, etc.) the principle of induction over natural numbers: If  $P(n)$  is some proposition involving a natural number  $n$  and we want to show that  $P$  holds for *all* numbers  $n$ , we can reason like this:

- show that  $P(0)$  holds;
- show that, for any  $n'$ , if  $P(n')$  holds, then so does  $P(S\ n')$ ;
- conclude that  $P(n)$  holds for all  $n$ .

In Coq, the steps are the same but the order is backwards: we begin with the goal of proving  $P(n)$  for all  $n$  and break it down (by applying the `induction` tactic) into two separate subgoals: first showing  $P(0)$  and then showing  $P(n') \rightarrow P(S\ n')$ . Here's how this works for the theorem at hand:

Theorem `plus_n_O` :  $\forall n:\text{nat}, n = n + 0.$

Proof.

```
intros n. induction n as [| n' IHn'].
- reflexivity.
- simpl. rewrite <- IHn'. reflexivity. Qed.
```

Like `destruct`, the `induction` tactic takes an `as...` clause that specifies the names of the variables to be introduced in the subgoals. In the first branch,  $n$  is replaced by  $0$  and the goal becomes  $0 + 0 = 0$ , which follows by simplification. In the second,  $n$  is replaced by  $S\ n'$  and the assumption  $n' + 0 = n'$  is added to the context (with the name  $IHn'$ , i.e., the Induction Hypothesis for  $n'$  – notice that this name is explicitly chosen in the `as...` clause of the call to `induction` rather than letting Coq choose one arbitrarily). The goal in this case becomes  $(S\ n') + 0 = S\ n'$ , which simplifies to  $S\ (n' + 0) = S\ n'$ , which in turn follows from  $IHn'$ .

Theorem `minus_diag` :  $\forall n,$

**minus**  $n\ n = 0$ .

Proof.

```
intros n. induction n as [| n' IHn'].  
-  
  simpl. reflexivity.  
-  
  simpl. rewrite → IHn'. reflexivity. Qed.
```

**Exercise: 2 stars, recommended (basic\_induction)** Prove the following using induction. You might need previously proven results.

Theorem mult\_0\_r :  $\forall n:\mathbf{nat}$ ,  
 $n \times 0 = 0$ .

Proof.

*Admitted.*

Theorem plus\_n\_Sm :  $\forall n\ m : \mathbf{nat}$ ,  
 $\mathbf{S}\ (n + m) = n + (\mathbf{S}\ m)$ .

Proof.

*Admitted.*

Theorem plus\_comm :  $\forall n\ m : \mathbf{nat}$ ,  
 $n + m = m + n$ .

Proof.

*Admitted.*

Theorem plus\_assoc :  $\forall n\ m\ p : \mathbf{nat}$ ,  
 $n + (m + p) = (n + m) + p$ .

Proof.

*Admitted.*

□

**Exercise: 2 stars (double\_plus)** Consider the following function, which doubles its argument:

```
Fixpoint double (n: nat) :=  
  match n with  
  | 0 => 0  
  | S n' => S (S (double n'))  
end.
```

Use induction to prove this simple fact about **double**:

Lemma double\_plus :  $\forall n$ , **double**  $n = n + n$ .

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional (evenb\_S)** One inconvenient aspect of our definition of `evenb n` is that it may need to perform a recursive call on `n - 2`. This makes proofs about `evenb n` harder when done by induction on `n`, since we may need an induction hypothesis about `n - 2`. The following lemma gives a better characterization of `evenb (S n)`:

Theorem `evenb_S` :  $\forall n : \text{nat},$   
 $\text{evenb } (S\ n) = \text{negb } (\text{evenb } n).$

Proof.

*Admitted.*

□

**Exercise: 1 star (destruct\_induction)** Briefly explain the difference between the tactics `destruct` and `induction`.

□

### 3.3 Proofs Within Proofs

In Coq, as in informal mathematics, large proofs are often broken into a sequence of theorems, with later proofs referring to earlier theorems. But sometimes a proof will require some miscellaneous fact that is too trivial and of too little general interest to bother giving it its own top-level name. In such cases, it is convenient to be able to simply state and prove the needed “sub-theorem” right at the point where it is used. The `assert` tactic allows us to do this. For example, our earlier proof of the `mult_0_plus` theorem referred to a previous theorem named `plus_0_n`. We could instead use `assert` to state and prove `plus_0_n` in-line:

Theorem `mult_0_plus'` :  $\forall n\ m : \text{nat},$   
 $(0 + n) \times m = n \times m.$

Proof.

```
intros n m.
assert (H: 0 + n = n). { reflexivity. }
rewrite → H.
reflexivity. Qed.
```

The `assert` tactic introduces two sub-goals. The first is the assertion itself; by prefixing it with `H`: we name the assertion `H`. (We can also name the assertion with `as` just as we did above with `destruct` and `induction`, i.e., `assert (0 + n = n) as H`.) Note that we surround the proof of this assertion with curly braces `{ ... }`, both for readability and so that, when using Coq interactively, we can see more easily when we have finished this sub-proof. The second goal is the same as the one at the point where we invoke `assert` except that, in the context, we now have the assumption `H` that `0 + n = n`. That is, `assert` generates one subgoal where we must prove the asserted fact and a second subgoal where we can use the asserted fact to make progress on whatever we were trying to prove in the first place.

The `assert` tactic is handy in many sorts of situations. For example, suppose we want to prove that  $(n + m) + (p + q) = (m + n) + (p + q)$ . The only difference between the

two sides of the  $=$  is that the arguments  $m$  and  $n$  to the first inner  $+$  are swapped, so it seems we should be able to use the commutativity of addition (`plus_comm`) to rewrite one into the other. However, the `rewrite` tactic is a little stupid about *where* it applies the rewrite. There are three uses of  $+$  here, and it turns out that doing `rewrite → plus_comm` will affect only the *outer* one...

Theorem `plus_rearrange_firsttry` :  $\forall n\ m\ p\ q : \text{nat}$ ,  
 $(n + m) + (p + q) = (m + n) + (p + q)$ .

Proof.

```
intros n m p q.
rewrite → plus_comm.
```

Abort.

To get `plus_comm` to apply at the point where we want it to, we can introduce a local lemma stating that  $n + m = m + n$  (for the particular  $m$  and  $n$  that we are talking about here), prove this lemma using `plus_comm`, and then use it to do the desired rewrite.

Theorem `plus_rearrange` :  $\forall n\ m\ p\ q : \text{nat}$ ,  
 $(n + m) + (p + q) = (m + n) + (p + q)$ .

Proof.

```
intros n m p q.
assert (H: n + m = m + n).
{ rewrite → plus_comm. reflexivity. }
rewrite → H. reflexivity. Qed.
```

## 3.4 More Exercises

**Exercise: 3 stars, recommended (`mult_comm`)** Use `assert` to help prove this theorem. You shouldn't need to use induction on `plus_swap`.

Theorem `plus_swap` :  $\forall n\ m\ p : \text{nat}$ ,  
 $n + (m + p) = m + (n + p)$ .

Proof.

*Admitted.*

Now prove commutativity of multiplication. (You will probably need to define and prove a separate subsidiary theorem to be used in the proof of this one. You may find that `plus_swap` comes in handy.)

Theorem `mult_comm` :  $\forall m\ n : \text{nat}$ ,  
 $m \times n = n \times m$ .

Proof.

*Admitted.*

□

**Exercise: 3 stars, optional (more\_exercises)** Take a piece of paper. For each of the following theorems, first *think* about whether (a) it can be proved using only simplification and rewriting, (b) it also requires case analysis (**destruct**), or (c) it also requires induction. Write down your prediction. Then fill in the proof. (There is no need to turn in your piece of paper; this is just to encourage you to reflect before you hack!)

Theorem leb\_refl :  $\forall n:\text{nat}$ ,  
 $\text{true} = \text{leb } n \ n$ .

Proof.

*Admitted.*

Theorem zero\_nbeq\_S :  $\forall n:\text{nat}$ ,  
 $\text{beq\_nat } 0 \ (\text{S } n) = \text{false}$ .

Proof.

*Admitted.*

Theorem andb\_false\_r :  $\forall b : \text{bool}$ ,  
 $\text{andb } b \ \text{false} = \text{false}$ .

Proof.

*Admitted.*

Theorem plus\_ble\_compat\_l :  $\forall n \ m \ p : \text{nat}$ ,  
 $\text{leb } n \ m = \text{true} \rightarrow \text{leb } (p + n) \ (p + m) = \text{true}$ .

Proof.

*Admitted.*

Theorem S\_nbeq\_0 :  $\forall n:\text{nat}$ ,  
 $\text{beq\_nat } (\text{S } n) \ 0 = \text{false}$ .

Proof.

*Admitted.*

Theorem mult\_1\_l :  $\forall n:\text{nat}$ ,  $1 \times n = n$ .

Proof.

*Admitted.*

Theorem all3\_spec :  $\forall b \ c : \text{bool}$ ,

$$\begin{aligned} & \text{orb} \\ & \quad (\text{andb } b \ c) \\ & \quad (\text{orb } (\text{negb } b) \\ & \quad \quad (\text{negb } c)) \end{aligned}$$

$= \text{true}$ .

Proof.

*Admitted.*

Theorem mult\_plus\_distr\_r :  $\forall n \ m \ p : \text{nat}$ ,  
 $(n + m) \times p = (n \times p) + (m \times p)$ .

Proof.

*Admitted.*



Theorem `mult_assoc` :  $\forall n\ m\ p : \text{nat},$   
 $n \times (m \times p) = (n \times m) \times p.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional (`beq_nat_refl`)** Prove the following theorem. (Putting the `true` on the left-hand side of the equality may look odd, but this is how the theorem is stated in the Coq standard library, so we follow suit. Rewriting works equally well in either direction, so we will have no problem using the theorem no matter which way we state it.)

Theorem `beq_nat_refl` :  $\forall n : \text{nat},$   
`true` = `beq_nat n n.`

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional (`plus_swap'`)** The `replace` tactic allows you to specify a particular subterm to rewrite and what you want it rewritten to: `replace (t) with (u)` replaces (all copies of) expression `t` in the goal by expression `u`, and generates `t = u` as an additional subgoal. This is often useful when a plain `rewrite` acts on the wrong part of the goal.

Use the `replace` tactic to do a proof of `plus_swap'`, just like `plus_swap` but without needing `assert (n + m = m + n).`

Theorem `plus_swap'` :  $\forall n\ m\ p : \text{nat},$   
 $n + (m + p) = m + (n + p).$

Proof.

*Admitted.*

□

**Exercise: 3 stars, recommended (`binary_commute`)** Recall the *increment* and *binary-to-unary* functions that you wrote for the *binary* exercise in the Basics chapter. Prove that the following diagram commutes: `binnat`  $\xrightarrow{\text{incr}}$  `binnat`  $\parallel$  `bin2unary` `bin2unary`  
 $\parallel$  `v v nat`  $\xrightarrow{S}$  `nat` That is, incrementing a binary number and then converting it to unary yields the same result as first converting it to unary and then incrementing. Name your theorem *bin\_to\_nat\_pres\_incr*.

Before you start working on this exercise, please copy the definitions from your solution to the *binary* exercise here so that this file can be graded on its own. If you find yourself wanting to change your original definitions to make the property easier to prove, feel free to do so!

□

**Exercise: 5 stars, advanced (binary\_inverse)** This exercise is a continuation of the previous exercise about binary numbers. You will need your definitions and theorems from there to complete this one.

(a) First, write a function to convert natural numbers to binary numbers. Then prove that starting with any natural number, converting to binary, then converting back yields the same natural number you started with.

(b) You might naturally think that we should also prove the opposite direction: that starting with a binary number, converting to a natural, and then back to binary yields the same number we started with. However, this is not true! Explain what the problem is.

(c) Define a “direct” normalization function – i.e., a function *normalize* from binary numbers to binary numbers such that, for any binary number *b*, converting to a natural and then back to binary yields (*normalize b*). Prove it. (Warning: This part is tricky!)

Again, feel free to change your earlier definitions if this helps here.

□

### 3.5 Formal vs. Informal Proof (Optional)

# Chapter 4

## Library Lists

### 4.1 Lists: Working with Structured Data

Require Export Induction.  
Module NATLIST.

### 4.2 Pairs of Numbers

In an `Inductive` type definition, each constructor can take any number of arguments – none (as with `true` and `O`), one (as with `S`), or more than one, as here:

```
Inductive natprod : Type :=  
| pair : nat → nat → natprod.
```

This declaration can be read: “There is one way to construct a pair of numbers: by applying the constructor `pair` to two arguments of type `nat`.”

Check (pair 3 5).

Here are two simple functions for extracting the first and second components of a pair. The definitions also illustrate how to do pattern matching on two-argument constructors.

```
Definition fst (p : natprod) : nat :=  
  match p with  
  | pair x y ⇒ x  
  end.
```

```
Definition snd (p : natprod) : nat :=  
  match p with  
  | pair x y ⇒ y  
  end.
```

Compute (fst (pair 3 5)).

Since pairs are used quite a bit, it is nice to be able to write them with the standard mathematical notation  $(x,y)$  instead of `pair x y`. We can tell Coq to allow this with a `Notation` declaration.

```
Notation "( x , y )" := (pair x y).
```

The new notation can be used both in expressions and in pattern matches (indeed, we've seen it already in the previous chapter – this works because the pair notation is actually provided as part of the standard library):

```
Compute (fst (3,5)).
```

```
Definition fst' (p : natprod) : nat :=
  match p with
  | (x,y) => x
  end.
```

```
Definition snd' (p : natprod) : nat :=
  match p with
  | (x,y) => y
  end.
```

```
Definition swap_pair (p : natprod) : natprod :=
  match p with
  | (x,y) => (y,x)
  end.
```

Let's try to prove a few simple facts about pairs.

If we state things in a particular (and slightly peculiar) way, we can complete proofs with just reflexivity (and its built-in simplification):

```
Theorem surjective_pairing' : ∀ (n m : nat),
  (n,m) = (fst (n,m), snd (n,m)).
```

Proof.

```
  reflexivity. Qed.
```

But reflexivity is not enough if we state the lemma in a more natural way:

```
Theorem surjective_pairing_stuck : ∀ (p : natprod),
  p = (fst p, snd p).
```

Proof.

```
  simpl. Abort.
```

We have to expose the structure of  $p$  so that `simpl` can perform the pattern match in `fst` and `snd`. We can do this with `destruct`.

```
Theorem surjective_pairing : ∀ (p : natprod),
  p = (fst p, snd p).
```

Proof.

```
  intros p. destruct p as [n m]. simpl. reflexivity. Qed.
```

Notice that, unlike its behavior with `nats`, `destruct` doesn't generate an extra subgoal here. That's because `natprods` can only be constructed in one way.

**Exercise: 1 star (snd\_fst\_is\_swap)** Theorem `snd_fst_is_swap` :  $\forall (p : \text{natprod}),$   
 $(\text{snd } p, \text{fst } p) = \text{swap\_pair } p.$

Proof.

*Admitted.*

□

**Exercise: 1 star, optional (fst\_swap\_is\_snd)** Theorem `fst_swap_is_snd` :  $\forall (p : \text{natprod}),$   
 $\text{fst } (\text{swap\_pair } p) = \text{snd } p.$

Proof.

*Admitted.*

□

## 4.3 Lists of Numbers

Generalizing the definition of pairs, we can describe the type of *lists* of numbers like this: “A list is either the empty list or else a pair of a number and another list.”

Inductive `natlist` : Type :=

| `nil` : `natlist`  
| `cons` : `nat` → `natlist` → `natlist`.

For example, here is a three-element list:

Definition `mylist` := `cons 1 (cons 2 (cons 3 nil))`.

As with pairs, it is more convenient to write lists in familiar programming notation. The following declarations allow us to use `::` as an infix `cons` operator and square brackets as an “outfix” notation for constructing lists.

Notation "`x :: l`" := `(cons x l)`  
(at level 60, right associativity).

Notation "`[ ]`" := `nil`.

Notation "`[ x ; .. ; y ]`" := `(cons x .. (cons y nil) ..)`.

It is not necessary to understand the details of these declarations, but in case you are interested, here is roughly what's going on. The `right associativity` annotation tells Coq how to parenthesize expressions involving several uses of `::` so that, for example, the next three declarations mean exactly the same thing:

Definition `mylist1` := `1 :: (2 :: (3 :: nil))`.

Definition `mylist2` := `1 :: 2 :: 3 :: nil`.

Definition `mylist3` := `[1;2;3]`.

The `at level 60` part tells Coq how to parenthesize expressions that involve both `::` and some other infix operator. For example, since we defined `+` as infix notation for the `plus` function at level 50, Notation “`x + y`” `:= (plus x y)` (at level 50, left associativity). the `+` operator will bind tighter than `::`, so `1 + 2 :: [3]` will be parsed, as we’d expect, as `(1 + 2) :: [3]` rather than `1 + (2 :: [3])`.

(By the way, it’s worth noting in passing that expressions like “`1 + 2 :: [3]`” can be a little confusing when you read them in a `.v` file. The inner brackets, around 3, indicate a list, but the outer brackets, which are invisible in the HTML rendering, are there to instruct the “coqdoc” tool that the bracketed part should be displayed as Coq code rather than running text.)

The second and third `Notation` declarations above introduce the standard square-bracket notation for lists; the right-hand side of the third one illustrates Coq’s syntax for declaring `n`-ary notations and translating them to nested sequences of binary constructors.

## Repeat

A number of functions are useful for manipulating lists. For example, the `repeat` function takes a number `n` and a `count` and returns a list of length `count` where every element is `n`.

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | 0 => nil
  | S count' => n :: (repeat n count')
  end.
```

## Length

The `length` function calculates the length of a list.

```
Fixpoint length (l : natlist) : nat :=
  match l with
  | nil => 0
  | h :: t => S (length t)
  end.
```

## Append

The `app` function concatenates (appends) two lists.

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil => l2
  | h :: t => h :: (app t l2)
  end.
```

Actually, `app` will be used a lot in some parts of what follows, so it is convenient to have an infix operator for it.

Notation "x ++ y" := (app x y)  
(right associativity, at level 60).

Example test\_app1: [1;2;3] ++ [4;5] = [1;2;3;4;5].

Proof. reflexivity. Qed.

Example test\_app2: nil ++ [4;5] = [4;5].

Proof. reflexivity. Qed.

Example test\_app3: [1;2;3] ++ nil = [1;2;3].

Proof. reflexivity. Qed.

## Head (with default) and Tail

Here are two smaller examples of programming with lists. The `hd` function returns the first element (the “head”) of the list, while `tl` returns everything but the first element (the “tail”). Of course, the empty list has no first element, so we must pass a default value to be returned in that case.

Definition `hd (default:nat) (l:natlist) : nat :=`  
  match `l` with  
  | `nil`  $\Rightarrow$  `default`  
  | `h :: t`  $\Rightarrow$  `h`  
  end.

Definition `tl (l:natlist) : natlist :=`  
  match `l` with  
  | `nil`  $\Rightarrow$  `nil`  
  | `h :: t`  $\Rightarrow$  `t`  
  end.

Example test\_hd1: `hd 0 [1;2;3]` = 1.

Proof. reflexivity. Qed.

Example test\_hd2: `hd 0 []` = 0.

Proof. reflexivity. Qed.

Example test\_tl: `tl [1;2;3]` = [2;3].

Proof. reflexivity. Qed.

## Exercises

**Exercise: 2 stars, recommended (list\_funs)** Complete the definitions of `nonzeros`, `oddmembers` and `countoddmembers` below. Have a look at the tests to understand what these functions should do.

Fixpoint `nonzeros (l:natlist) : natlist :=`  
  `admit`.

Example test\_nonzeros:

`nonzeros [0;1;0;2;3;0;0]` = [1;2;3].

*Admitted.*

```
Fixpoint oddmembers (l:natlist) : natlist :=  
  admit.
```

Example test\_oddmembers:

```
  oddmembers [0;1;0;2;3;0;0] = [1;3].
```

*Admitted.*

```
Fixpoint countoddmembers (l:natlist) : nat :=  
  admit.
```

Example test\_countoddmembers1:

```
  countoddmembers [1;0;3;1;4;5] = 4.
```

*Admitted.*

Example test\_countoddmembers2:

```
  countoddmembers [0;2;4] = 0.
```

*Admitted.*

Example test\_countoddmembers3:

```
  countoddmembers nil = 0.
```

*Admitted.*

□

**Exercise: 3 stars, advanced (alternate)** Complete the definition of `alternate`, which “zips up” two lists into one, alternating between elements taken from the first list and elements from the second. See the tests below for more specific examples.

Note: one natural and elegant way of writing `alternate` will fail to satisfy Coq’s requirement that all `Fixpoint` definitions be “obviously terminating.” If you find yourself in this rut, look for a slightly more verbose solution that considers elements of both lists at the same time. (One possible solution requires defining a new kind of pairs, but this is not the only way.)

```
Fixpoint alternate (l1 l2 : natlist) : natlist :=  
  admit.
```

Example test\_alternate1:

```
  alternate [1;2;3] [4;5;6] = [1;4;2;5;3;6].
```

*Admitted.*

Example test\_alternate2:

```
  alternate [1] [4;5;6] = [1;4;5;6].
```

*Admitted.*

Example test\_alternate3:

```
  alternate [1;2;3] [4] = [1;4;2;3].
```

*Admitted.*

Example test\_alternate4:



alternate [] [20;30] = [20;30].

*Admitted.*

□

## Bags via Lists

A **bag** (or *multiset*) is like a set, except that each element can appear multiple times rather than just once. One possible implementation is to represent a bag of numbers as a list.

Definition bag := natlist.

**Exercise: 3 stars, recommended (bag\_functions)** Complete the following definitions for the functions count, sum, add, and member for bags.

Fixpoint count (v:nat) (s:bag) : nat :=  
admit.

All these proofs can be done just by reflexivity.

Example test\_count1: count 1 [1;2;3;1;4;1] = 3.

*Admitted.*

Example test\_count2: count 6 [1;2;3;1;4;1] = 0.

*Admitted.*

Multiset **sum** is similar to set *union*: **sum** a b contains all the elements of a and of b. (Mathematicians usually define *union* on multisets a little bit differently, which is why we don't use that name for this operation.) For **sum** we're giving you a header that does not give explicit names to the arguments. Moreover, it uses the keyword **Definition** instead of **Fixpoint**, so even if you had names for the arguments, you wouldn't be able to process them recursively. The point of stating the question this way is to encourage you to think about whether **sum** can be implemented in another way – perhaps by using functions that have already been defined.

Definition sum : bag → bag → bag :=  
admit.

Example test\_sum1: count 1 (sum [1;2;3] [1;4;1]) = 3.

*Admitted.*

Definition add (v:nat) (s:bag) : bag :=  
admit.

Example test\_add1: count 1 (add 1 [1;4;1]) = 3.

*Admitted.*

Example test\_add2: count 5 (add 1 [1;4;1]) = 0.

*Admitted.*

Definition member (v:nat) (s:bag) : bool :=  
admit.

Example test\_member1: member 1 [1;4;1] = true.

*Admitted.*

Example test\_member2: member 2 [1;4;1] = false.

*Admitted.*

□

**Exercise: 3 stars, optional (bag\_more\_functions)** Here are some more bag functions for you to practice with.

Fixpoint remove\_one ( $v:\text{nat}$ ) ( $s:\text{bag}$ ) : bag :=

*admit.*

Example test\_remove\_one1:

count 5 (remove\_one 5 [2;1;5;4;1]) = 0.

*Admitted.*

Example test\_remove\_one2:

count 5 (remove\_one 5 [2;1;4;1]) = 0.

*Admitted.*

Example test\_remove\_one3:

count 4 (remove\_one 5 [2;1;4;5;1;4]) = 2.

*Admitted.*

Example test\_remove\_one4:

count 5 (remove\_one 5 [2;1;5;4;5;1;4]) = 1.

*Admitted.*

Fixpoint remove\_all ( $v:\text{nat}$ ) ( $s:\text{bag}$ ) : bag :=

*admit.*

Example test\_remove\_all1: count 5 (remove\_all 5 [2;1;5;4;1]) = 0.

*Admitted.*

Example test\_remove\_all2: count 5 (remove\_all 5 [2;1;4;1]) = 0.

*Admitted.*

Example test\_remove\_all3: count 4 (remove\_all 5 [2;1;4;5;1;4]) = 2.

*Admitted.*

Example test\_remove\_all4: count 5 (remove\_all 5 [2;1;5;4;5;1;4;5;1;4]) = 0.

*Admitted.*

Fixpoint subset ( $s1:\text{bag}$ ) ( $s2:\text{bag}$ ) : bool :=

*admit.*

Example test\_subset1: subset [1;2] [2;1;4;1] = true.

*Admitted.*

Example test\_subset2: subset [1;2;2] [2;1;4;1] = false.

*Admitted.*

□

**Exercise: 3 stars, recommended (bag\_theorem)** Write down an interesting theorem `bag_theorem` about bags involving the functions `count` and `add`, and prove it. For this, replace the `admit` command below with the statement of your theorem. Note that, since this problem is somewhat open-ended, it's possible that you may come up with a theorem which is true, but whose proof requires techniques you haven't learned yet. Feel free to ask for help if you get stuck!

Theorem `bag_theorem` :

*admit.*

*Admitted.*

□

## 4.4 Reasoning About Lists

As with numbers, simple facts about list-processing functions can sometimes be proved entirely by simplification. For example, the simplification performed by `reflexivity` is enough for this theorem...

Theorem `nil_app` :  $\forall l:\text{natlist},$

`[] ++ l = l.`

Proof. `reflexivity. Qed.`

... because the `[]` is substituted into the match “scrutinee” in the definition of `app`, allowing the match itself to be simplified.

Also, as with numbers, it is sometimes helpful to perform case analysis on the possible shapes (empty or non-empty) of an unknown list.

Theorem `tl_length_pred` :  $\forall l:\text{natlist},$

`pred (length l) = length (tl l).`

Proof.

`intros l. destruct l as [| n l'].`

-

`reflexivity.`

-

`reflexivity. Qed.`

Here, the `nil` case works because we've chosen to define `tl nil = nil`. Notice that the `as` annotation on the `destruct` tactic here introduces two names, `n` and `l'`, corresponding to the fact that the `cons` constructor for lists takes two arguments (the head and tail of the list it is constructing).

Usually, though, interesting theorems about lists require induction for their proofs.

### Micro-Sermon

Simply reading example proof scripts will not get you very far! It is important to work through the details of each one, using Coq and thinking about what each step achieves.

Otherwise it is more or less guaranteed that the exercises will make no sense when you get to them. 'Nuff said.

### 4.4.1 Induction on Lists

Proofs by induction over datatypes like **natlist** are a little less familiar than standard natural number induction, but the idea is equally simple. Each **Inductive** declaration defines a set of data values that can be built up using the declared constructors: a boolean can be either **true** or **false**; a number can be either **O** or **S** applied to another number; a list can be either **nil** or **cons** applied to a number and a list.

Moreover, applications of the declared constructors to one another are the *only* possible shapes that elements of an inductively defined set can have, and this fact directly gives rise to a way of reasoning about inductively defined sets: a number is either **O** or else it is **S** applied to some *smaller* number; a list is either **nil** or else it is **cons** applied to some number and some *smaller* list; etc. So, if we have in mind some proposition  $P$  that mentions a list  $l$  and we want to argue that  $P$  holds for *all* lists, we can reason as follows:

- First, show that  $P$  is true of  $l$  when  $l$  is **nil**.
- Then show that  $P$  is true of  $l$  when  $l$  is **cons**  $n$   $l'$  for some number  $n$  and some smaller list  $l'$ , assuming that  $P$  is true for  $l'$ .

Since larger lists can only be built up from smaller ones, eventually reaching **nil**, these two arguments together establish the truth of  $P$  for all lists  $l$ . Here's a concrete example:

**Theorem** `app_assoc` :  $\forall l1\ l2\ l3 : \text{natlist},$

$(l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).$

**Proof.**

```
intros l1 l2 l3. induction l1 as [| n l1' IHl1'].
-
  reflexivity.
-
  simpl. rewrite → IHl1'. reflexivity. Qed.
```

Notice that, as when doing induction on natural numbers, the **as...** clause provided to the **induction** tactic gives a name to the induction hypothesis corresponding to the smaller list  $l1'$  in the **cons** case. Once again, this Coq proof is not especially illuminating as a static written document – it is easy to see what's going on if you are reading the proof in an interactive Coq session and you can see the current goal and context at each point, but this state is not visible in the written-down parts of the Coq proof. So a natural-language proof – one written for human readers – will need to include more explicit signposts; in particular, it will help the reader stay oriented if we remind them exactly what the induction hypothesis is in the second case.

For comparison, here is an informal proof of the same theorem.

*Theorem:* For all lists  $l1$ ,  $l2$ , and  $l3$ ,  $(l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3)$ .

*Proof:* By induction on  $l1$ .

- First, suppose  $l1 = []$ . We must show  $([] ++ l2) ++ l3 = [] ++ (l2 ++ l3)$ , which follows directly from the definition of  $++$ .
- Next, suppose  $l1 = n::l1'$ , with  $(l1' ++ l2) ++ l3 = l1' ++ (l2 ++ l3)$  (the induction hypothesis). We must show  $((n :: l1') ++ l2) ++ l3 = (n :: l1') ++ (l2 ++ l3)$ . By the definition of  $++$ , this follows from  $n :: ((l1' ++ l2) ++ l3) = n :: (l1' ++ (l2 ++ l3))$ , which is immediate from the induction hypothesis.  $\square$

## Reversing a list

For a slightly more involved example of inductive proof over lists, suppose we use `app` to define a list-reversing function `rev`:

```
Fixpoint rev (l: natlist) : natlist :=
  match l with
  | nil => nil
  | h :: t => rev t ++ [h]
  end.
```

Example test\_rev1: `rev [1;2;3] = [3;2;1]`.

Proof. `reflexivity. Qed.`

Example test\_rev2: `rev nil = nil`.

Proof. `reflexivity. Qed.`

## Proofs about reverse

Now let's prove some theorems about our newly defined `rev`. For something a bit more challenging than what we've seen, let's prove that reversing a list does not change its length. Our first attempt gets stuck in the successor case...

Theorem `rev_length_firsttry` :  $\forall l : \text{natlist},$   
 $\text{length} (\text{rev } l) = \text{length } l$ .

Proof.

```
intros l. induction l as [| n l' IHl'].
```

```
-
```

```
  reflexivity.
```

```
-
```

```
  simpl.
```

```
  rewrite <- IHl'.
```

Abort.

So let's take the equation relating  $++$  and `length` that would have enabled us to make progress and prove it as a separate lemma.

Theorem `app_length` :  $\forall l1 \ l2 : \text{natlist},$   
 $\text{length} (l1 ++ l2) = (\text{length } l1) + (\text{length } l2)$ .

Proof.

```

intros l1 l2. induction l1 as [| n l1' IHL1'].
-
  reflexivity.
-
  simpl. rewrite → IHL1'. reflexivity. Qed.

```

Note that, to make the lemma as general as possible, we quantify over *all* **natlists**, not just those that result from an application of **rev**. This should seem natural, because the truth of the goal clearly doesn't depend on the list having been reversed. Moreover, it is easier to prove the more general property.

Now we can complete the original proof.

Theorem `rev_length` :  $\forall l : \text{natlist}$ ,  
`length (rev l) = length l`.

Proof.

```

intros l. induction l as [| n l' IHL'].
-
  reflexivity.
-
  simpl. rewrite → app_length, plus_comm.
  rewrite → IHL'. reflexivity. Qed.

```

For comparison, here are informal proofs of these two theorems:

*Theorem:* For all lists  $l1$  and  $l2$ ,  $\text{length } (l1 ++ l2) = \text{length } l1 + \text{length } l2$ .

*Proof:* By induction on  $l1$ .

- First, suppose  $l1 = []$ . We must show  $\text{length } ([] ++ l2) = \text{length } [] + \text{length } l2$ , which follows directly from the definitions of **length** and **++**.
- Next, suppose  $l1 = n::l1'$ , with  $\text{length } (l1' ++ l2) = \text{length } l1' + \text{length } l2$ . We must show  $\text{length } ((n::l1') ++ l2) = \text{length } (n::l1') + \text{length } l2$ . This follows directly from the definitions of **length** and **++** together with the induction hypothesis.  $\square$

*Theorem:* For all lists  $l$ ,  $\text{length } (\text{rev } l) = \text{length } l$ .

*Proof:* By induction on  $l$ .

- First, suppose  $l = []$ . We must show  $\text{length } (\text{rev } []) = \text{length } []$ , which follows directly from the definitions of **length** and **rev**.
- Next, suppose  $l = n::l'$ , with  $\text{length } (\text{rev } l') = \text{length } l'$ . We must show  $\text{length } (\text{rev } (n :: l')) = \text{length } (n :: l')$ . By the definition of **rev**, this follows from  $\text{length } ((\text{rev } l') ++ n) = S (\text{length } l')$  which, by the previous lemma, is the same as  $\text{length } (\text{rev } l') + \text{length } n = S (\text{length } l')$ . This follows directly from the induction hypothesis and the definition of **length**.  $\square$

The style of these proofs is rather longwinded and pedantic. After the first few, we might find it easier to follow proofs that give fewer details (which can easily work out in our own minds or on scratch paper if necessary) and just highlight the non-obvious steps. In this more compressed style, the above proof might look like this:

*Theorem:* For all lists  $l$ ,  $\text{length} (\text{rev } l) = \text{length } l$ .

*Proof:* First, observe that  $\text{length} (l ++ [n]) = S (\text{length } l)$  for any  $l$  (this follows by a straightforward induction on  $l$ ). The main property again follows by induction on  $l$ , using the observation together with the induction hypothesis in the case where  $l = n'::l'$ .  $\square$

Which style is preferable in a given situation depends on the sophistication of the expected audience and how similar the proof at hand is to ones that the audience will already be familiar with. The more pedantic style is a good default for our present purposes.

#### 4.4.2 SearchAbout

We've seen that proofs can make use of other theorems we've already proved, e.g., using `rewrite`. But in order to refer to a theorem, we need to know its name! Indeed, it is often hard even to remember what theorems have been proven, much less what they are called.

Coq's `SearchAbout` command is quite helpful with this. Typing `SearchAbout foo` will cause Coq to display a list of all theorems involving `foo`. For example, try uncommenting the following to see a list of theorems that we have proved about `rev`:

Keep `SearchAbout` in mind as you do the following exercises and throughout the rest of the book; it can save you a lot of time!

If you are using ProofGeneral, you can run `SearchAbout` with `C-c C-a C-a`. Pasting its response into your buffer can be accomplished with `C-c C-;`.

#### 4.4.3 List Exercises, Part 1

**Exercise: 3 stars (list\_exercises)** More practice with lists:

**Theorem** `app_nil_r` :  $\forall l : \text{natlist},$   
 $l ++ [] = l$ .

**Proof.**

*Admitted.*

**Theorem** `rev_involutive` :  $\forall l : \text{natlist},$   
 $\text{rev} (\text{rev } l) = l$ .

**Proof.**

*Admitted.*

There is a short solution to the next one. If you find yourself getting tangled up, step back and try to look for a simpler way.

**Theorem** `app_assoc4` :  $\forall l1\ l2\ l3\ l4 : \text{natlist},$   
 $l1 ++ (l2 ++ (l3 ++ l4)) = ((l1 ++ l2) ++ l3) ++ l4$ .

**Proof.**

*Admitted.*

An exercise about your implementation of `nonzeros`:

Lemma `nonzeros_app` :  $\forall l1\ l2 : \text{natlist}$ ,  
`nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2)`.

Proof.

*Admitted.*

□

**Exercise: 2 stars (`beq_natlist`)** Fill in the definition of `beq_natlist`, which compares lists of numbers for equality. Prove that `beq_natlist l l` yields `true` for every list `l`.

Fixpoint `beq_natlist (l1 l2 : natlist) : bool :=`  
`admit`.

Example `test_beq_natlist1` :  
`(beq_natlist nil nil = true)`.  
*Admitted.*

Example `test_beq_natlist2` :  
`beq_natlist [1;2;3] [1;2;3] = true`.  
*Admitted.*

Example `test_beq_natlist3` :  
`beq_natlist [1;2;3] [1;2;4] = false`.  
*Admitted.*

Theorem `beq_natlist_refl` :  $\forall l : \text{natlist}$ ,  
`true = beq_natlist l l`.

Proof.

*Admitted.*

□

#### 4.4.4 List Exercises, Part 2

**Exercise: 3 stars, advanced (`bag_proofs`)** Here are a couple of little theorems to prove about your definitions about bags earlier in the file.

Theorem `count_member_nonzero` :  $\forall (s : \text{bag})$ ,  
`leb 1 (count 1 (1 :: s)) = true`.

Proof.

*Admitted.*

The following lemma about `leb` might help you in the next proof.

Theorem `ble_n_Sn` :  $\forall n$ ,  
`leb n (S n) = true`.

Proof.



```

intros n. induction n as [| n' IHn'].
-
  simpl. reflexivity.
-
  simpl. rewrite IHn'. reflexivity. Qed.
Theorem remove_decreases_count:  $\forall (s : \text{bag})$ ,
  leb (count 0 (remove_one 0 s)) (count 0 s) = true.
Proof.
  Admitted.
□

```

**Exercise: 3 stars, optional (bag\_count\_sum)** Write down an interesting theorem *bag\_count\_sum* about bags involving the functions `count` and `sum`, and prove it.

□

**Exercise: 4 stars, advanced (rev\_injective)** Prove that the `rev` function is injective – that is, forall  $(l1\ l2 : \text{natlist})$ ,  $\text{rev } l1 = \text{rev } l2 \rightarrow l1 = l2$ . (There is a hard way and an easy way to do this.)

□

## 4.5 Options

Suppose we want to write a function that returns the  $n$ th element of some list. If we give it type  $\text{nat} \rightarrow \text{natlist} \rightarrow \text{nat}$ , then we'll have to choose some number to return when the list is too short...

```

Fixpoint nth_bad (l:natlist) (n:nat) : nat :=
  match l with
  | nil  $\Rightarrow$  42
  | a :: l'  $\Rightarrow$  match beq_nat n 0 with
                | true  $\Rightarrow$  a
                | false  $\Rightarrow$  nth_bad l' (pred n)
                end
  end.

```

This solution is not so good: If `nth_bad` returns 42, we can't tell whether that value actually appears on the input without further processing. A better alternative is to change the return type of `nth_bad` to include an error value as a possible outcome. We call this type **natoption**.

```

Inductive natoption : Type :=
| Some : nat  $\rightarrow$  natoption
| None : natoption.

```

We can then change the above definition of `nth_bad` to return `None` when the list is too short and `Some a` when the list has enough members and `a` appears at position `n`. We call this new function `nth_error` to indicate that it may result in an error.

```
Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
  match l with
  | nil => None
  | a :: l' => match beq_nat n 0 with
    | true => Some a
    | false => nth_error l' (pred n)
  end
end.
```

Example test\_nth\_error1 : nth\_error [4;5;6;7] 0 = Some 4.

Proof. reflexivity. Qed.

Example test\_nth\_error2 : nth\_error [4;5;6;7] 3 = Some 7.

Proof. reflexivity. Qed.

Example test\_nth\_error3 : nth\_error [4;5;6;7] 9 = None.

Proof. reflexivity. Qed.

(In the HTML version, the boilerplate proofs of these examples are elided. Click on a box if you want to see one.)

This example is also an opportunity to introduce one more small feature of Coq's programming language: conditional expressions...

```
Fixpoint nth_error' (l:natlist) (n:nat) : natoption :=
  match l with
  | nil => None
  | a :: l' => if beq_nat n 0 then Some a
    else nth_error' l' (pred n)
  end.
```

Coq's conditionals are exactly like those found in any other language, with one small generalization. Since the boolean type is not built in, Coq actually allows conditional expressions over *any* inductively defined type with exactly two constructors. The guard is considered true if it evaluates to the first constructor in the `Inductive` definition and false if it evaluates to the second.

The function below pulls the `nat` out of a `natoption`, returning a supplied default in the `None` case.

```
Definition option_elim (d : nat) (o : natoption) : nat :=
  match o with
  | Some n' => n'
  | None => d
  end.
```

**Exercise: 2 stars (hd\_error)** Using the same idea, fix the `hd` function from earlier so we don't have to pass a default element for the `nil` case.

Definition `hd_error (l : natlist) : natoption :=`  
`admit.`

Example `test_hd_error1 : hd_error [] = None.`  
*Admitted.*

Example `test_hd_error2 : hd_error [1] = Some 1.`  
*Admitted.*

Example `test_hd_error3 : hd_error [5;6] = Some 5.`  
*Admitted.*

□

**Exercise: 1 star, optional (option\_elim\_hd)** This exercise relates your new `hd_error` to the old `hd`.

Theorem `option_elim_hd : ∀ (l:natlist) (default:nat),`  
`hd default l = option_elim default (hd_error l).`

Proof.

*Admitted.*

□

End NATLIST.

## 4.6 Partial Maps

As a final illustration of how data structures can be defined in Coq, here is a simple *partial map* data type, analogous to the map or dictionary data structures found in most programming languages.

First, we define a new inductive datatype **id** to serve as the “keys” of our partial maps.

Inductive **id** : Type :=  
| `ld : nat → id.`

Internally, an **id** is just a number. Introducing a separate type by wrapping each `nat` with the tag `ld` makes definitions more readable and gives us the flexibility to change representations later if we wish.

We'll also need an equality test for **ids**:

Definition `beq_id x1 x2 :=`  
`match x1, x2 with`  
| `ld n1, ld n2 ⇒ beq_nat n1 n2`  
`end.`

**Exercise: 1 star (beq\_id\_refl)** Theorem `beq_id_refl` :  $\forall x, \text{true} = \text{beq\_id } x \ x$ .

Proof.

*Admitted.*

□

Now we define the type of partial maps:

Module `PARTIALMAP`.

Import `NatList`.

Inductive `partial_map` : Type :=

| `empty` : `partial_map`

| `record` : `id` → `nat` → `partial_map` → `partial_map`.

This declaration can be read: “There are two ways to construct a `partial_map`: either using the constructor `empty` to represent an empty partial map, or by applying the constructor `record` to a key, a value, and an existing `partial_map` to construct a `partial_map` with an additional key-to-value mapping.”

The `update` function overrides the entry for a given key in a partial map (or adds a new entry if the given key is not already present).

Definition `update` ( $d$  : `partial_map`)  
                   ( $key$  : `id`) ( $value$  : `nat`)  
                   : `partial_map` :=

`record key value d`.

Last, the `find` function searches a `partial_map` for a given key. It returns `None` if the key was not found and `Some val` if the key was associated with `val`. If the same key is mapped to multiple values, `find` will return the first one it encounters.

Fixpoint `find` ( $key$  : `id`) ( $d$  : `partial_map`) : `natoption` :=

`match d with`

| `empty` ⇒ `None`

| `record k v d'` ⇒ if `beq_id key k`  
                           then `Some v`  
                           else `find key d'`

`end`.

**Exercise: 1 star (update\_eq)** Theorem `update_eq` :

$\forall (d$  : `partial_map`) ( $k$  : `id`) ( $v$  : `nat`),  
   `find k (update d k v) = Some v`.

Proof.

*Admitted.*

□

**Exercise: 1 star (update\_neq)** Theorem `update_neq` :

$\forall (d$  : `partial_map`) ( $m \ n$  : `id`) ( $o$  : `nat`),

$\text{beq\_id } m \ n = \text{false} \rightarrow \text{find } m \ (\text{update } d \ n \ o) = \text{find } m \ d.$

Proof.

*Admitted.*

□

End PARTIALMAP.

**Exercise: 2 stars (baz\_num\_elts)** Consider the following inductive definition:

Inductive **baz** : Type :=

| Baz1 : **baz** → **baz**

| Baz2 : **baz** → **bool** → **baz**.

How *many* elements does the type **baz** have?

□

*Date* : 2016 – 03 – 04 09 : 33 : 20 – 0500(*Fri, 04Mar2016*)

# Chapter 5

## Library Poly

### 5.1 Poly: Polymorphism and Higher-Order Functions

Require Export Lists.

### 5.2 Polymorphism

In this chapter we continue our development of basic concepts of functional programming. The critical new ideas are *polymorphism* (abstracting functions over the types of the data they manipulate) and *higher-order functions* (treating functions as data). We begin with polymorphism.

#### 5.2.1 Polymorphic Lists

For the last couple of chapters, we've been working just with lists of numbers. Obviously, interesting programs also need to be able to manipulate lists with elements from other types – lists of strings, lists of booleans, lists of lists, etc. We *could* just define a new inductive datatype for each of these, for example...

```
Inductive boollist : Type :=  
  | bool_nil : boollist  
  | bool_cons : bool → boollist → boollist.
```

... but this would quickly become tedious, partly because we have to make up different constructor names for each datatype, but mostly because we would also need to define new versions of all our list manipulating functions (**length**, **rev**, etc.) for each new datatype definition.

To avoid all this repetition, Coq supports *polymorphic* inductive type definitions. For example, here is a *polymorphic list* datatype.

```
Inductive list (X:Type) : Type :=
```

```
| nil : list X
| cons : X → list X → list X.
```

This is exactly like the definition of **natlist** from the previous chapter, except that the **nat** argument to the **cons** constructor has been replaced by an arbitrary type **X**, a binding for **X** has been added to the header, and the occurrences of **natlist** in the types of the constructors have been replaced by **list X**. (We can re-use the constructor names **nil** and **cons** because the earlier definition of **natlist** was inside of a **Module** definition that is now out of scope.)

What sort of thing is **list** itself? One good way to think about it is that **list** is a *function* from **Types** to **Inductive** definitions; or, to put it another way, **list** is a function from **Types** to **Types**. For any particular type **X**, the type **list X** is an **Inductively** defined set of lists whose elements are things of type **X**.

With this definition, when we use the constructors **nil** and **cons** to build lists, we need to tell Coq the type of the elements in the lists we are building – that is, **nil** and **cons** are now *polymorphic constructors*. Observe the types of these constructors:

```
Check nil.
Check cons.
```

(Side note on notation: In **.v** files, the “forall” quantifier is spelled out in letters. In the generated HTML files,  $\forall$  is usually typeset as the usual mathematical “upside down A,” but you’ll see the spelled-out “forall” in a few places, as in the above comments. This is just a quirk of typesetting: there is no difference in meaning.)

The “ $\forall X$ ” in these types can be read as an additional argument to the constructors that determines the expected types of the arguments that follow. When **nil** and **cons** are used, these arguments are supplied in the same way as the others. For example, the list containing 2 and 1 is written like this:

```
Check (cons nat 2 (cons nat 1 (nil nat))).
```

(We’ve written **nil** and **cons** explicitly here because we haven’t yet defined the **[]** and **::** notations for the new version of lists. We’ll do that in a bit.)

We can now go back and make polymorphic versions of all the list-processing functions that we wrote before. Here is **repeat**, for example:

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 ⇒ nil X
  | S count' ⇒ cons X x (repeat X x count')
  end.
```

As with **nil** and **cons**, we can use **repeat** by applying it first to a type and then to its list argument:

```
Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
Proof. reflexivity. Qed.
```

To use `repeat` to build other kinds of lists, we simply instantiate it with an appropriate type parameter:

Example `test_repeat2` :

```
repeat bool false 1 = cons bool false (nil bool).
```

Proof. `reflexivity`. `Qed`.

Module `MUMBLEGRUMBLE`.

**Exercise: 2 stars (mumble\_grumble)** Consider the following two inductively defined types.

Inductive `mumble` : Type :=

```
| a : mumble  
| b : mumble → nat → mumble  
| c : mumble.
```

Inductive `grumble` (X:Type) : Type :=

```
| d : mumble → grumble X  
| e : X → grumble X.
```

Which of the following are well-typed elements of `grumble X` for some type X?

- `d (b a 5)`
- `d mumble (b a 5)`
- `d bool (b a 5)`
- `e bool true`
- `e mumble (b c 0)`
- `e bool (b c 0)`
- `c`

□

End `MUMBLEGRUMBLE`.

## Type Annotation Inference

Let's write the definition of `repeat` again, but this time we won't specify the types of any of the arguments. Will Coq still accept it?

Fixpoint `repeat' X x count` : `list X` :=

```
match count with  
| 0 ⇒ nil X  
| S count' ⇒ cons X x (repeat' X x count')
```



end.

Indeed it will. Let's see what type Coq has assigned to `repeat'`:

Check `repeat'`.

Check `repeat`.

It has exactly the same type type as `repeat`. Coq was able to use *type inference* to deduce what the types of `X`, `x`, and `count` must be, based on how they are used. For example, since `X` is used as an argument to `cons`, it must be a `Type`, since `cons` expects a `Type` as its first argument; matching `count` with `0` and `S` means it must be a `nat`; and so on.

This powerful facility means we don't always have to write explicit type annotations everywhere, although explicit type annotations are still quite useful as documentation and sanity checks, so we will continue to use them most of the time. You should try to find a balance in your own code between too many type annotations (which can clutter and distract) and too few (which forces readers to perform type inference in their heads in order to understand your code).

## Type Argument Synthesis

To we use a polymorphic function, we need to pass it one or more types in addition to its other arguments. For example, the recursive call in the body of the `repeat` function above must pass along the type `X`. But since the second argument to `repeat` is an element of `X`, it seems entirely obvious that the first argument can only be `X` – why should we have to write it explicitly?

Fortunately, Coq permits us to avoid this kind of redundancy. In place of any type argument we can write the “implicit argument” `_`, which can be read as “Please try to figure out for yourself what belongs here.” More precisely, when Coq encounters a `_`, it will attempt to *unify* all locally available information – the type of the function being applied, the types of the other arguments, and the type expected by the context in which the application appears – to determine what concrete type should replace the `_`.

This may sound similar to type annotation inference – indeed, the two procedures rely on the same underlying mechanisms. Instead of simply omitting the types of some arguments to a function, like `repeat' X x count : list X :=` we can also replace the types with `_` `repeat' (X : _) (x : _) (count : _) : list X :=` to tell Coq to attempt to infer the missing information.

Using implicit arguments, the `count` function can be written like this:

```
Fixpoint repeat'' X x count : list X :=
  match count with
  | 0 => nil _
  | S count' => cons _ x (repeat'' _ x count')
end.
```

In this instance, we don't save much by writing `_` instead of `X`. But in many cases the difference in both keystrokes and readability is nontrivial. For example, suppose we want to write down a list containing the numbers 1, 2, and 3. Instead of writing this...

```
Definition list123 :=
  cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).
```

...we can use argument synthesis to write this:

```
Definition list123' :=
  cons _ 1 (cons _ 2 (cons _ 3 (nil _))).
```

## Implicit Arguments

We can go further and even avoid writing `_`'s in most cases by telling Coq *always* to infer the type argument(s) of a given function. The *Arguments* directive specifies the name of the function (or constructor) and then lists its argument names, with curly braces around any arguments to be treated as implicit. (If some arguments of a definition don't have a name, as is often the case for constructors, they can be marked with a wildcard pattern `_`.)

```
Arguments nil {X}.
Arguments cons {X} _ -.
Arguments repeat {X} x count.
```

Now, we don't have to supply type arguments at all:

```
Definition list123'' := cons 1 (cons 2 (cons 3 nil)).
```

Alternatively, we can declare an argument to be implicit when defining the function itself, by surrounding it in curly braces. For example:

```
Fixpoint repeat''' {X : Type} (x : X) (count : nat) : list X :=
  match count with
  | 0 => nil
  | S count' => cons x (repeat''' x count')
  end.
```

(Note that we didn't even have to provide a type argument to the recursive call to `repeat'''`; indeed, it would be invalid to provide one!)

We will use the latter style whenever possible, but we will continue to use explicit *Argument* declarations for **Inductive** constructors. The reason for this is that marking the parameter of an inductive type as implicit causes it to become implicit for the type itself, not just for its constructors. For instance, consider the following alternative definition of the **list** type:

```
Inductive list' {X:Type} : Type :=
  | nil' : list'
  | cons' : X → list' → list'.
```

Because `X` is declared as implicit for the *entire* inductive definition including **list'** itself, we now have to write just **list'** whether we are talking about lists of numbers or booleans or anything else, rather than **list' nat** or **list' bool** or whatever; this is a step too far.

Let's finish by re-implementing a few other standard list functions on our new polymorphic lists...

```

Fixpoint app {X : Type} (l1 l2 : list X)
  : (list X) :=
  match l1 with
  | nil => l2
  | cons h t => cons h (app t l2)
  end.

Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil => nil
  | cons h t => app (rev t) (cons h nil)
  end.

Fixpoint length {X : Type} (l : list X) : nat :=
  match l with
  | nil => 0
  | cons _ l' => S (length l')
  end.

```

```

Example test_rev1 :
  rev (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
Proof. reflexivity. Qed.

```

```

Example test_rev2:
  rev (cons true nil) = cons true nil.
Proof. reflexivity. Qed.

```

```

Example test_length1: length (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof. reflexivity. Qed.

```

One small problem with declaring arguments `Implicit` is that, occasionally, Coq does not have enough local information to determine a type argument; in such cases, we need to tell Coq that we want to give the argument explicitly just this time. For example, suppose we write this:

```

Fail Definition mynil := nil.

```

(The *Fail* qualifier that appears before `Definition` can be used with *any* command, and is used to ensure that that command indeed fails when executed. If the command does fail, Coq prints the corresponding error message, but continues processing the rest of the file.)

Here, Coq gives us an error because it doesn't know what type argument to supply to `nil`. We can help it by providing an explicit type declaration (so that Coq has more information available when it gets to the “application” of `nil`):

```

Definition mynil : list nat := nil.

```

Alternatively, we can force the implicit arguments to be explicit by prefixing the function name with `@`.

```

Check @nil.

```

Definition mynil' := @nil **nat**.

Using argument synthesis and implicit arguments, we can define convenient notation for lists, as before. Since we have made the constructor type arguments implicit, Coq will know to automatically infer these when we use the notations.

Notation "x :: y" := (cons x y)  
(at level 60, right associativity).

Notation "[ ]" := nil.

Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).

Notation "x ++ y" := (app x y)  
(at level 60, right associativity).

Now lists can be written just the way we'd hope:

Definition list123''' := [1; 2; 3].

## Exercises

**Exercise: 2 stars, optional (poly\_exercises)** Here are a few simple exercises, just like ones in the Lists chapter, for practice with polymorphism. Complete the proofs below.

Theorem app\_nil\_r :  $\forall (X:\text{Type}), \forall l:\text{list } X,$   
 $l ++ [] = l.$

Proof.

*Admitted.*

Theorem app\_assoc :  $\forall A (l\ m\ n:\text{list } A),$   
 $l ++ m ++ n = (l ++ m) ++ n.$

Proof.

*Admitted.*

Lemma app\_length :  $\forall (X:\text{Type}) (l1\ l2 : \text{list } X),$   
 $\text{length } (l1 ++ l2) = \text{length } l1 + \text{length } l2.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional (more\_poly\_exercises)** Here are some slightly more interesting ones...

Theorem rev\_app\_distr:  $\forall X (l1\ l2 : \text{list } X),$   
 $\text{rev } (l1 ++ l2) = \text{rev } l2 ++ \text{rev } l1.$

Proof.

*Admitted.*

Theorem rev\_involutive :  $\forall X : \text{Type}, \forall l : \text{list } X,$   
 $\text{rev } (\text{rev } l) = l.$

Proof.

*Admitted.*

□

### 5.2.2 Polymorphic Pairs

Following the same pattern, the type definition we gave in the last chapter for pairs of numbers can be generalized to *polymorphic pairs*, often called *products*:

```
Inductive prod (X Y : Type) : Type :=  
| pair : X → Y → prod X Y.
```

*Arguments* **pair** {X} {Y} - ..

As with lists, we make the type arguments implicit and define the familiar concrete notation.

**Notation** "( x , y )" := (**pair** x y).

We can also use the **Notation** mechanism to define the standard notation for product *types*:

**Notation** "X \* Y" := (**prod** X Y) : *type\_scope*.

(The annotation : *type\_scope* tells Coq that this abbreviation should only be used when parsing types. This avoids a clash with the multiplication symbol.)

It is easy at first to get (x,y) and X×Y confused. Remember that (x,y) is a *value* built from two other values, while X×Y is a *type* built from two other types. If x has type X and y has type Y, then (x,y) has type X×Y.

The first and second projection functions now look pretty much as they would in any functional programming language.

```
Definition fst {X Y : Type} (p : X × Y) : X :=  
  match p with  
  | (x, y) ⇒ x  
  end.
```

```
Definition snd {X Y : Type} (p : X × Y) : Y :=  
  match p with  
  | (x, y) ⇒ y  
  end.
```

The following function takes two lists and combines them into a list of pairs. In other functional languages, it is often called *zip*; we call it **combine** for consistency with Coq's standard library.

```
Fixpoint combine {X Y : Type} (lx : list X) (ly : list Y)  
  : list (X × Y) :=  
  match lx, ly with  
  | [], _ ⇒ []  
  | _, [] ⇒ []
```

```
| x :: tx, y :: ty => (x, y) :: (combine tx ty)
end.
```

**Exercise: 1 star, optional (combine\_checks)** Try answering the following questions on paper and checking your answers in coq:

- What is the type of `combine` (i.e., what does `Check @combine` print?)
- What does `Compute (combine 1;2 false;false;true;true).` print?  $\square$

**Exercise: 2 stars, recommended (split)** The function `split` is the right inverse of `combine`: it takes a list of pairs and returns a pair of lists. In many functional languages, it is called *unzip*.

Uncomment the material below and fill in the definition of `split`. Make sure it passes the given unit test.

```
Fixpoint split {X Y : Type} (l : list (X × Y))
  : (list X) × (list Y) :=
```

*admit.*

Example test\_split:

```
split [(1,false);(2,false)] = ([1;2], [false;false]).
```

Proof.

*Admitted.*

$\square$

## 5.2.3 Polymorphic Options

One last polymorphic type for now: *polymorphic options*, which generalize `natoption` from the previous chapter:

```
Inductive option (X:Type) : Type :=
| Some : X → option X
| None : option X.
```

*Arguments* Some {X} \_.

*Arguments* None {X}.

We can now rewrite the `nth_error` function so that it works with any type of lists.

```
Fixpoint nth_error {X : Type} (l : list X) (n : nat)
  : option X :=
```

```
  match l with
  | [] => None
  | a :: l' => if beq_nat n 0 then Some a else nth_error l' (pred n)
  end.
```

Example test\_nth\_error1 : `nth_error [4;5;6;7] 0 = Some 4.`

Proof. reflexivity. Qed.

Example test\_nth\_error2 : nth\_error [[1];[2]] 1 = Some [2].

Proof. reflexivity. Qed.

Example test\_nth\_error3 : nth\_error [true] 2 = None.

Proof. reflexivity. Qed.

**Exercise: 1 star, optional (hd\_error\_poly)** Complete the definition of a polymorphic version of the `hd_error` function from the last chapter. Be sure that it passes the unit tests below.

Definition `hd_error {X : Type} (l : list X) : option X :=`  
    `admit.`

Once again, to force the implicit arguments to be explicit, we can use `@` before the name of the function.

Check `@hd_error`.

Example test\_hd\_error1 : `hd_error [1;2] = Some 1`.

*Admitted.*

Example test\_hd\_error2 : `hd_error [[1];[2]] = Some [1]`.

*Admitted.*

□

## 5.3 Functions as Data

Like many other modern programming languages – including all functional languages (ML, Haskell, Scheme, Scala, Clojure, etc.) – Coq treats functions as first-class citizens, allowing them to be passed as arguments to other functions, returned as results, stored in data structures, etc.

### 5.3.1 Higher-Order Functions

Functions that manipulate other functions are often called *higher-order* functions. Here's a simple one:

Definition `doit3times {X:Type} (f:X→X) (n:X) : X :=`  
    `f (f (f n)).`

The argument `f` here is itself a function (from `X` to `X`); the body of `doit3times` applies `f` three times to some value `n`.

Check `@doit3times`.

Example test\_doit3times: `doit3times minustwo 9 = 3`.

Proof. reflexivity. Qed.

Example test\_doit3times': `doit3times negb true = false`.

Proof. reflexivity. Qed.

### 5.3.2 Filter

Here is a more useful higher-order function, taking a list of  $X$ s and a *predicate* on  $X$  (a function from  $X$  to **bool**) and “filtering” the list, returning a new list containing just those elements for which the predicate returns **true**.

```
Fixpoint filter {X:Type} (test: X→bool) (l:list X)
  : (list X) :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ if test h then h :: (filter test t)
              else filter test t
  end.
```

For example, if we apply **filter** to the predicate **evenb** and a list of numbers **l**, it returns a list containing just the even members of **l**.

Example test\_filter1: filter evenb [1;2;3;4] = [2;4].

Proof. reflexivity. Qed.

```
Definition length_is_1 {X : Type} (l : list X) : bool :=
  beq_nat (length l) 1.
```

```
Example test_filter2:
  filter length_is_1
    [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
```

Proof. reflexivity. Qed.

We can use **filter** to give a concise version of the **countoddmembers** function from the Lists chapter.

```
Definition countoddmembers' (l:list nat) : nat :=
  length (filter oddb l).
```

Example test\_countoddmembers'1: countoddmembers' [1;0;3;1;4;5] = 4.

Proof. reflexivity. Qed.

Example test\_countoddmembers'2: countoddmembers' [0;2;4] = 0.

Proof. reflexivity. Qed.

Example test\_countoddmembers'3: countoddmembers' nil = 0.

Proof. reflexivity. Qed.

### 5.3.3 Anonymous Functions

It is arguably a little sad, in the example just above, to be forced to define the function **length\_is\_1** and give it a name just to be able to pass it as an argument to **filter**, since we will probably never use it again. Moreover, this is not an isolated example: when using higher-order functions, we often want to pass as arguments “one-off” functions that we will never use again; having to give each of these functions a name would be tedious.



Fortunately, there is a better way. We can construct a function “on the fly” without declaring it at the top level or giving it a name.

Example test\_anon\_fun':

```
doit3times (fun n => n × n) 2 = 256.
```

Proof. reflexivity. Qed.

The expression (fun n => n × n) can be read as “the function that, given a number n, yields n × n.”

Here is the filter example, rewritten to use an anonymous function.

Example test\_filter2':

```
filter (fun l => beq_nat (length l) 1)
  [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
= [ [3]; [4]; [8] ].
```

Proof. reflexivity. Qed.

**Exercise: 2 stars (filter\_even\_gt7)** Use filter (instead of Fixpoint) to write a Coq function filter\_even\_gt7 that takes a list of natural numbers as input and returns a list of just those that are even and greater than 7.

Definition filter\_even\_gt7 (l : list nat) : list nat :=  
admit.

Example test\_filter\_even\_gt7\_1 :

```
filter_even_gt7 [1;2;6;9;10;3;12;8] = [10;12;8].  
Admitted.
```

Example test\_filter\_even\_gt7\_2 :

```
filter_even_gt7 [5;2;6;19;129] = [].  
Admitted.
```

□

**Exercise: 3 stars (partition)** Use filter to write a Coq function partition: partition : forall X : Type, (X -> bool) -> list X -> list X \* list X Given a set X, a test function of type X → bool and a list X, partition should return a pair of lists. The first member of the pair is the sublist of the original list containing the elements that satisfy the test, and the second is the sublist containing those that fail the test. The order of elements in the two sublists should be the same as their order in the original list.

Definition partition {X : Type}  
 (test : X → bool)  
 (l : list X)  
 : list X × list X :=  
admit.

Example test\_partition1: partition oddb [1;2;3;4;5] = ([1;3;5], [2;4]).

Admitted.

Example test\_partition2: partition (fun x  $\Rightarrow$  false) [5;9;0] = ([], [5;9;0]).

*Admitted.*

□

### 5.3.4 Map

Another handy higher-order function is called **map**.

```
Fixpoint map {X Y:Type} (f:X $\rightarrow$ Y) (l:list X) : (list Y) :=
  match l with
  | []  $\Rightarrow$  []
  | h :: t  $\Rightarrow$  (f h) :: (map f t)
end.
```

It takes a function **f** and a list **l** = [*n1*, *n2*, *n3*, ...] and returns the list [**f** *n1*, **f** *n2*, **f** *n3*,...] , where **f** has been applied to each element of **l** in turn. For example:

Example test\_map1: map (fun x  $\Rightarrow$  plus 3 x) [2;0;2] = [5;3;5].

Proof. reflexivity. Qed.

The element types of the input and output lists need not be the same, since **map** takes *two* type arguments, **X** and **Y**; it can thus be applied to a list of numbers and a function from numbers to booleans to yield a list of booleans:

Example test\_map2:

map oddb [2;1;2;5] = [false;true;false;true].

Proof. reflexivity. Qed.

It can even be applied to a list of numbers and a function from numbers to *lists* of booleans to yield a *list of lists* of booleans:

Example test\_map3:

map (fun n  $\Rightarrow$  [evenb n;oddb n]) [2;1;2;5]  
= [[true;false]; [false;true]; [true;false]; [false;true]].

Proof. reflexivity. Qed.

### Exercises

**Exercise: 3 stars (map\_rev)** Show that **map** and **rev** commute. You may need to define an auxiliary lemma.

Theorem map\_rev :  $\forall$  (X Y : Type) (f : X  $\rightarrow$  Y) (l : list X),  
map f (rev l) = rev (map f l).

Proof.

*Admitted.*

□

**Exercise: 2 stars, recommended (flat\_map)** The function `map` maps a **list** `X` to a **list** `Y` using a function of type `X → Y`. We can define a similar function, `flat_map`, which maps a **list** `X` to a **list** `Y` using a function `f` of type `X → list Y`. Your definition should work by ‘flattening’ the results of `f`, like so: `flat_map (fun n => n;n+1;n+2) 1;5;10 = 1; 2; 3; 5; 6; 7; 10; 11; 12`.

```
Fixpoint flat_map {X Y:Type} (f:X → list Y) (l:list X)
  : (list Y) :=
```

*admit.*

Example `test_flat_map1`:

```
flat_map (fun n => [n;n;n]) [1;5;4]
= [1; 1; 1; 5; 5; 5; 4; 4; 4].
```

*Admitted.*

□

Lists are not the only inductive type that we can write a `map` function for. Here is the definition of `map` for the **option** type:

```
Definition option_map {X Y : Type} (f : X → Y) (xo : option X)
  : option Y :=
```

```
match xo with
| None => None
| Some x => Some (f x)
end.
```

**Exercise: 2 stars, optional (implicit\_args)** The definitions and uses of `filter` and `map` use implicit arguments in many places. Replace the curly braces around the implicit arguments with parentheses, and then fill in explicit type parameters where necessary and use Coq to check that you’ve done so correctly. (This exercise is not to be turned in; it is probably easiest to do it on a *copy* of this file that you can throw away afterwards.) □

### 5.3.5 Fold

An even more powerful higher-order function is called `fold`. This function is the inspiration for the “*reduce*” operation that lies at the heart of Google’s map/reduce distributed programming framework.

```
Fixpoint fold {X Y:Type} (f: X→Y→Y) (l:list X) (b:Y)
  : Y :=
```

```
match l with
| nil => b
| h :: t => f h (fold f t b)
end.
```

Intuitively, the behavior of the `fold` operation is to insert a given binary operator `f` between every pair of elements in a given list. For example, `fold plus [1;2;3;4]` intuitively

means  $1+2+3+4$ . To make this precise, we also need a “starting element” that serves as the initial second input to  $f$ . So, for example, `fold plus 1;2;3;4 0` yields  $1 + (2 + (3 + (4 + 0)))$ . Some more examples:

Check `(fold andb)`.

Example `fold_example1` :

`fold mult [1;2;3;4] 1 = 24.`

Proof. `reflexivity. Qed.`

Example `fold_example2` :

`fold andb [true;true;false;true] true = false.`

Proof. `reflexivity. Qed.`

Example `fold_example3` :

`fold app [[1];[];[2;3];[4]] [] = [1;2;3;4].`

Proof. `reflexivity. Qed.`

**Exercise: 1 star, advanced (fold\_types\_different)** Observe that the type of `fold` is parameterized by *two* type variables,  $X$  and  $Y$ , and the parameter  $f$  is a binary operator that takes an  $X$  and a  $Y$  and returns a  $Y$ . Can you think of a situation where it would be useful for  $X$  and  $Y$  to be different?

### 5.3.6 Functions That Construct Functions

Most of the higher-order functions we have talked about so far take functions as arguments. Let’s look at some examples that involve *returning* functions as the results of other functions. To begin, here is a function that takes a value  $x$  (drawn from some type  $X$ ) and returns a function from  $\text{nat}$  to  $X$  that yields  $x$  whenever it is called, ignoring its  $\text{nat}$  argument.

Definition `constfun {X: Type} (x: X) : nat  $\rightarrow$  X :=`

`fun (k:nat)  $\Rightarrow$  x.`

Definition `fttrue := constfun true.`

Example `constfun_example1` : `fttrue 0 = true.`

Proof. `reflexivity. Qed.`

Example `constfun_example2` : `(constfun 5) 99 = 5.`

Proof. `reflexivity. Qed.`

In fact, the multiple-argument functions we have already seen are also examples of passing functions as data. To see why, recall the type of `plus`.

Check `plus`.

Each  $\rightarrow$  in this expression is actually a *binary* operator on types. This operator is *right-associative*, so the type of `plus` is really a shorthand for  $\text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})$  – i.e., it can be read as saying that “`plus` is a one-argument function that takes a `nat` and returns a one-argument function that takes another `nat` and returns a `nat`.” In the examples above, we have always

applied `plus` to both of its arguments at once, but if we like we can supply just the first. This is called *partial application*.

Definition `plus3` := `plus 3`.

Check `plus3`.

Example `test_plus3` : `plus3 4 = 7`.

Proof. `reflexivity`. Qed.

Example `test_plus3'` : `doit3times plus3 0 = 9`.

Proof. `reflexivity`. Qed.

Example `test_plus3''` : `doit3times (plus 3) 0 = 9`.

Proof. `reflexivity`. Qed.

## 5.4 Additional Exercises

Module EXERCISES.

**Exercise: 2 stars (fold\_length)** Many common functions on lists can be implemented in terms of `fold`. For example, here is an alternative definition of `length`:

Definition `fold_length` {*X* : Type} (*l* : `list X`) : `nat` :=  
`fold (fun _ n => S n) l 0`.

Example `test_fold_length1` : `fold_length [4;7;0] = 3`.

Proof. `reflexivity`. Qed.

Prove the correctness of `fold_length`.

Theorem `fold_length_correct` :  $\forall X (l : \text{list } X),$   
`fold_length l = length l`.

*Admitted.*

□

**Exercise: 3 stars (fold\_map)** We can also define `map` in terms of `fold`. Finish `fold_map` below.

Definition `fold_map` {*X* *Y*:Type} (*f* : *X* → *Y*) (*l* : `list X`) : `list Y` :=  
*admit*.

Write down a theorem *fold\_map\_correct* in Coq stating that `fold_map` is correct, and prove it.

□

**Exercise: 2 stars, advanced (currying)** In Coq, a function `f` : *A* → *B* → *C* really has the type *A* → (*B* → *C*). That is, if you give `f` a value of type *A*, it will give you function *f'* : *B* → *C*. If you then give *f'* a value of type *B*, it will return a value of type *C*. This allows

for partial application, as in `plus3`. Processing a list of arguments with functions that return functions is called *currying*, in honor of the logician Haskell Curry.

Conversely, we can reinterpret the type  $A \rightarrow B \rightarrow C$  as  $(A \times B) \rightarrow C$ . This is called *uncurrying*. With an uncurried binary function, both arguments must be given at once as a pair; there is no partial application.

We can define currying as follows:

**Definition** `prod_curry`  $\{X\ Y\ Z : \text{Type}\}$   
 $(f : X \times Y \rightarrow Z) (x : X) (y : Y) : Z := f\ (x, y).$

As an exercise, define its inverse, `prod_uncurry`. Then prove the theorems below to show that the two are inverses.

**Definition** `prod_uncurry`  $\{X\ Y\ Z : \text{Type}\}$   
 $(f : X \rightarrow Y \rightarrow Z) (p : X \times Y) : Z :=$   
*admit.*

As a trivial example of the usefulness of currying, we can use it to shorten one of the examples that we saw above:

**Example** `test_map2`: `map (fun x => plus 3 x) [2;0;2] = [5;3;5].`

**Proof.** `reflexivity. Qed.`

Thought exercise: before running the following commands, can you calculate the types of `prod_curry` and `prod_uncurry`?

`Check @prod_curry.`

`Check @prod_uncurry.`

**Theorem** `uncurry_curry` :  $\forall (X\ Y\ Z : \text{Type})$   
 $(f : X \rightarrow Y \rightarrow Z)$   
 $x\ y,$   
 $\text{prod\_curry} (\text{prod\_uncurry } f) x\ y = f\ x\ y.$

**Proof.**

*Admitted.*

**Theorem** `curry_uncurry` :  $\forall (X\ Y\ Z : \text{Type})$   
 $(f : (X \times Y) \rightarrow Z) (p : X \times Y),$   
 $\text{prod\_uncurry} (\text{prod\_curry } f) p = f\ p.$

**Proof.**

*Admitted.*

□

**Exercise: 2 stars, advanced (`nth_error_informal`)** Recall the definition of the `nth_error` function: `Fixpoint nth_error {X : Type} (l : list X) (n : nat) : option X := match l with | [] => None | a :: l' => if beq_nat n 0 then Some a else nth_error l' (pred n) end.` Write an informal proof of the following theorem: `forall X n l, length l = n -> @nth_error X l n = None` □

**Exercise: 4 stars, advanced (church\_numerals)** This exercise explores an alternative way of defining natural numbers, using the so-called *Church numerals*, named after mathematician Alonzo Church. We can represent a natural number  $n$  as a function that takes a function  $f$  as a parameter and returns  $f$  iterated  $n$  times.

Module CHURCH.

Definition nat :=  $\forall X : \text{Type}, (X \rightarrow X) \rightarrow X \rightarrow X$ .

Let's see how to write some numbers with this notation. Iterating a function once should be the same as just applying it. Thus,

Definition one : nat :=

fun (X : Type) (f : X → X) (x : X) ⇒ f x.

Similarly, two should apply  $f$  twice to its argument:

Definition two : nat :=

fun (X : Type) (f : X → X) (x : X) ⇒ f (f x).

Defining zero is somewhat trickier: how can we “apply a function zero times”? The answer is actually simple: just return the argument untouched.

Definition zero : nat :=

fun (X : Type) (f : X → X) (x : X) ⇒ x.

More generally, a number  $n$  can be written as `fun X f x ⇒ f (f ... (f x) ...)`, with  $n$  occurrences of  $f$ . Notice in particular how the `doit3times` function we've defined previously is actually just the Church representation of 3.

Definition three : nat := @doit3times.

Complete the definitions of the following functions. Make sure that the corresponding unit tests pass by proving them with `reflexivity`.

Successor of a natural number:

Definition succ (n : nat) : nat :=

admit.

Example succ\_1 : succ zero = one.

Proof. Admitted.

Example succ\_2 : succ one = two.

Proof. Admitted.

Example succ\_3 : succ two = three.

Proof. Admitted.

Addition of two natural numbers:

Definition plus (n m : nat) : nat :=

admit.

Example plus\_1 : plus zero one = one.

Proof. Admitted.

Example plus\_2 : plus two three = plus three two.

Proof. *Admitted.*

Example plus\_3 :

plus (plus two two) three = plus one (plus three three).

Proof. *Admitted.*

Multiplication:

Definition mult (n m : nat) : nat :=  
admit.

Example mult\_1 : mult one one = one.

Proof. *Admitted.*

Example mult\_2 : mult zero (plus three three) = zero.

Proof. *Admitted.*

Example mult\_3 : mult two three = plus three three.

Proof. *Admitted.*

Exponentiation:

(*Hint:* Polymorphism plays a crucial role here. However, choosing the right type to iterate over can be tricky. If you hit a “Universe inconsistency” error, try iterating over a different type: nat itself is usually problematic.)

Definition exp (n m : nat) : nat :=  
admit.

Example exp\_1 : exp two two = plus two two.

Proof. *Admitted.*

Example exp\_2 : exp three two = plus (mult two (mult two two)) one.

Proof. *Admitted.*

Example exp\_3 : exp three zero = one.

Proof. *Admitted.*

End CHURCH.

□

End EXERCISES.

Date : 2016 – 02 – 25 10 : 37 : 36 – 0500(*Thu, 25Feb2016*)



# Chapter 6

## Library Tactics

### 6.1 Tactics: More Basic Tactics

Require Export Poly.

This chapter introduces several more proof strategies and tactics that allow us to prove more interesting properties of functional programs. We will see:

- how to use auxiliary lemmas in both “forward-style” and “backward-style” proofs;
- how to reason about data constructors (in particular, how to use the fact that they are injective and disjoint);
- how to create a strong induction hypothesis (and when such strengthening is required); and
- more details on how to reason by case analysis.

### 6.2 The apply Tactic

We often encounter situations where the goal to be proved is exactly the same as some hypothesis in the context or some previously proved lemma.

```
Theorem silly1 :  $\forall (n\ m\ o\ p : \text{nat}),$   
   $n = m \rightarrow$   
   $[n;o] = [n;p] \rightarrow$   
   $[n;o] = [m;p].$ 
```

Proof.

```
intros n m o p eq1 eq2.  
rewrite  $\leftarrow$  eq1.
```

At this point, we could finish with “`rewrite → eq2. reflexivity.`” as we have done several times before. We can achieve the same effect in a single step by using the `apply` tactic instead:

```
apply eq2. Qed.
```

The `apply` tactic also works with *conditional* hypotheses and lemmas: if the statement being applied is an implication, then the premises of this implication will be added to the list of subgoals needing to be proved.

```
Theorem silly2 : ∀ (n m o p : nat),
  n = m →
  (∀ (q r : nat), q = r → [q;o] = [r;p]) →
  [n;o] = [m;p].
```

Proof.

```
intros n m o p eq1 eq2.
apply eq2. apply eq1. Qed.
```

You may find it instructive to experiment with this proof and see if there is a way to complete it using just `rewrite` instead of `apply`.

Typically, when we use `apply H`, the statement  $H$  will begin with a  $\forall$  that binds some *universal variables*. When Coq matches the current goal against the conclusion of  $H$ , it will try to find appropriate values for these variables. For example, when we do `apply eq2` in the following proof, the universal variable  $q$  in `eq2` gets instantiated with `n` and `r` gets instantiated with `m`.

```
Theorem silly2a : ∀ (n m : nat),
  (n,n) = (m,m) →
  (∀ (q r : nat), (q,q) = (r,r) → [q] = [r]) →
  [n] = [m].
```

Proof.

```
intros n m eq1 eq2.
apply eq2. apply eq1. Qed.
```

**Exercise: 2 stars, optional (silly\_ex)** Complete the following proof without using `simpl`.

```
Theorem silly_ex :
  (∀ n, evenb n = true → oddb (S n) = true) →
  evenb 3 = true →
  oddb 4 = true.
```

Proof.

*Admitted.*

□

To use the `apply` tactic, the (conclusion of the) fact being applied must match the goal exactly – for example, `apply` will not work if the left and right sides of the equality are swapped.

```
Theorem silly3_firsttry :  $\forall (n : \text{nat}),$ 
  true = beq_nat n 5  $\rightarrow$ 
  beq_nat (S (S n)) 7 = true.
```

Proof.

```
  intros n H.
```

```
  simpl.
```

Abort.

In this case we can use the `symmetry` tactic, which switches the left and right sides of an equality in the goal.

```
Theorem silly3 :  $\forall (n : \text{nat}),$ 
  true = beq_nat n 5  $\rightarrow$ 
  beq_nat (S (S n)) 7 = true.
```

Proof.

```
  intros n H.
```

```
  symmetry.
```

```
  simpl.   apply H. Qed.
```

**Exercise: 3 stars (apply\_exercise1)** (*Hint:* You can use `apply` with previously defined lemmas, not just hypotheses in the context. Remember that `SearchAbout` is your friend.)

```
Theorem rev_exercise1 :  $\forall (l \ l' : \text{list nat}),$ 
  l = rev l'  $\rightarrow$ 
  l' = rev l.
```

Proof.

```
  Admitted.
```

□

**Exercise: 1 star, optional (apply\_rewrite)** Briefly explain the difference between the tactics `apply` and `rewrite`. What are the situations where both can usefully be applied?

□

## 6.3 The `apply ... with ...` Tactic

The following silly example uses two rewrites in a row to get from `[a,b]` to `[e,f]`.

```
Example trans_eq_example :  $\forall (a \ b \ c \ d \ e \ f : \text{nat}),$ 
  [a;b] = [c;d]  $\rightarrow$ 
  [c;d] = [e;f]  $\rightarrow$ 
  [a;b] = [e;f].
```

Proof.

```
  intros a b c d e f eq1 eq2.
```

```
  rewrite  $\rightarrow$  eq1. rewrite  $\rightarrow$  eq2. reflexivity. Qed.
```

Since this is a common pattern, we might like to pull it out as a lemma recording, once and for all, the fact that equality is transitive.

**Theorem** `trans_eq` :  $\forall (X:\text{Type}) (n\ m\ o : X),$   
 $n = m \rightarrow m = o \rightarrow n = o.$

**Proof.**

```
intros X n m o eq1 eq2. rewrite → eq1. rewrite → eq2.
reflexivity. Qed.
```

Now, we should be able to use `trans_eq` to prove the above example. However, to do this we need a slight refinement of the `apply` tactic.

**Example** `trans_eq_example'` :  $\forall (a\ b\ c\ d\ e\ f : \text{nat}),$   
 $[a;b] = [c;d] \rightarrow$   
 $[c;d] = [e;f] \rightarrow$   
 $[a;b] = [e;f].$

**Proof.**

```
intros a b c d e f eq1 eq2.
```

If we simply tell Coq `apply trans_eq` at this point, it can tell (by matching the goal against the conclusion of the lemma) that it should instantiate `X` with `[nat]`, `n` with `[a,b]`, and `o` with `[e,f]`. However, the matching process doesn't determine an instantiation for `m`: we have to supply one explicitly by adding `with (m:=[c;d])` to the invocation of `apply`.

```
apply trans_eq with (m:=[c;d]). apply eq1. apply eq2. Qed.
```

Actually, we usually don't have to include the name `m` in the `with` clause; Coq is often smart enough to figure out which instantiation we're giving. We could instead write: `apply trans_eq with [c;d]`.

**Exercise: 3 stars, optional (apply\_with\_exercise)** **Example** `trans_eq_exercise` :  $\forall (n\ m\ o\ p : \text{nat}),$   
 $m = (\text{minustwo } o) \rightarrow$   
 $(n + p) = m \rightarrow$   
 $(n + p) = (\text{minustwo } o).$

**Proof.**

*Admitted.*

□

## 6.4 The inversion tactic

Recall the definition of natural numbers: `Inductive nat : Type := | O : nat | S : nat -> nat.` It is obvious from this definition that every number has one of two forms: either it is the constructor `O` or it is built by applying the constructor `S` to another number. But there is more here than meets the eye: implicit in the definition (and in our informal understanding of how datatype declarations work in other programming languages) are two more facts:

- The constructor `S` is *injective*. That is, if `S n = S m`, it must be the case that `n = m`.
- The constructors `O` and `S` are *disjoint*. That is, `O` is not equal to `S n` for any `n`.

Similar principles apply to all inductively defined types: all constructors are injective, and the values built from distinct constructors are never equal. For lists, the `cons` constructor is injective and `nil` is different from every non-empty list. For booleans, `true` and `false` are different. (Since neither `true` nor `false` take any arguments, their injectivity is not an issue.) And so on.

Coq provides a tactic called `inversion` that allows us to exploit these principles in proofs. To see how to use it, let's show explicitly that the `S` constructor is injective:

**Theorem** `S_injective` :  $\forall (n\ m : \text{nat}),$

$S\ n = S\ m \rightarrow$

$n = m.$

**Proof.**

`intros n m H.`

By writing `inversion H` at this point, we ask Coq to generate all equations that it can infer from `H` as additional hypotheses, replacing variables in the goal as it goes. In the present example, this amounts to adding a new hypothesis `H1 : n = m` and replacing `n` by `m` in the goal.

`inversion H. reflexivity. Qed.`

Here's a more interesting example that shows how multiple equations can be derived at once.

**Theorem** `inversion_ex1` :  $\forall (n\ m\ o : \text{nat}),$

$[n; m] = [o; o] \rightarrow$

$[n] = [m].$

**Proof.**

`intros n m o H. inversion H. reflexivity. Qed.`

It is possible to name the equations that `inversion` generates with an `as ...` clause:

**Theorem** `inversion_ex2` :  $\forall (n\ m : \text{nat}),$

$[n] = [m] \rightarrow$

$n = m.$

**Proof.**

`intros n o H. inversion H as [Hno]. reflexivity. Qed.`

**Exercise: 1 star (inversion\_ex3)** Example `inversion_ex3` :  $\forall (X : \text{Type}) (x\ y\ z : X) (l\ j : \text{list } X),$

$x :: y :: l = z :: j \rightarrow$

$y :: l = x :: j \rightarrow$

$x = y.$

**Proof.**

*Admitted.*

□

While the injectivity of constructors allows us to reason that  $\forall (n\ m : \text{nat}), S\ n = S\ m \rightarrow n = m$ , the converse of this implication is an instance of a more general fact about constructors and functions, which we will find useful below:

**Theorem** `f_equal` :  $\forall (A\ B : \text{Type}) (f : A \rightarrow B) (x\ y : A),$   
 $x = y \rightarrow f\ x = f\ y.$

**Proof.** `intros A B f x y eq. rewrite eq. reflexivity. Qed.`

When used on a hypothesis involving an equality between *different* constructors (e.g.,  $S\ n = 0$ ), `inversion` solves the goal immediately. To see why this makes sense, consider the following proof:

**Theorem** `beq_nat_0_l` :  $\forall n,$   
 $\text{beq\_nat}\ 0\ n = \text{true} \rightarrow n = 0.$

**Proof.**

`intros n.`

We can proceed by case analysis on `n`. The first case is trivial.

`destruct n as [| n'].`

-

`intros H. reflexivity.`

However, the second one doesn't look so simple: assuming  $\text{beq\_nat}\ 0\ (S\ n') = \text{true}$ , we must show  $S\ n' = 0$ , but the latter clearly contradictory! The way forward lies in the assumption. After simplifying the goal state, we see that  $\text{beq\_nat}\ 0\ (S\ n') = \text{true}$  has become  $\text{false} = \text{true}$ :

-

`simpl.`

If we use `inversion` on this hypothesis, Coq notices that the subgoal we are working on is impossible, and therefore removes it from further consideration.

`intros H. inversion H. Qed.`

This is an instance of a general logical principle known as the *principle of explosion*, which asserts that a contradiction entails anything, even false things. For instance:

**Theorem** `inversion_ex4` :  $\forall (n : \text{nat}),$   
 $S\ n = 0 \rightarrow$   
 $2 + 2 = 5.$

**Proof.**

`intros n contra. inversion contra. Qed.`

**Theorem** `inversion_ex5` :  $\forall (n\ m : \text{nat}),$   
 $\text{false} = \text{true} \rightarrow$   
 $[n] = [m].$

**Proof.**

`intros n m contra. inversion contra. Qed.`

If you find the principle of explosion confusing, remember that these proofs are not actually showing that the conclusion of the statement holds. Rather, they are arguing that the situation described by the premise can never arise, so the implication is vacuous. We'll explore the principle of explosion of more detail in the next chapter.

**Exercise: 1 star (inversion\_ex6)** Example `inversion_ex6` :  $\forall (X : \text{Type})$   
 $(x\ y\ z : X) (l\ j : \text{list } X),$

$x :: y :: l = [] \rightarrow$   
 $y :: l = z :: j \rightarrow$   
 $x = z.$

**Proof.**

*Admitted.*

□

To summarize this discussion, suppose  $H$  is a hypothesis in the context or a previously proven lemma of the form `c a1 a2 ... an = d b1 b2 ... bm` for some constructors `c` and `d` and arguments  $a1 \dots an$  and  $b1 \dots bm$ . Then `inversion H` has the following effect:

- If `c` and `d` are the same constructor, then, by the injectivity of this constructor, we know that  $a1 = b1$ ,  $a2 = b2$ , etc.; `inversion H` adds these facts to the context, and tries to use them to rewrite the goal.
- If `c` and `d` are different constructors, then the hypothesis  $H$  is contradictory, and the current goal doesn't have to be considered. In this case, `inversion H` marks the current goal as completed and pops it off the goal stack.

## 6.5 Using Tactics on Hypotheses

By default, most tactics work on the goal formula and leave the context unchanged. However, most tactics also have a variant that performs a similar operation on a statement in the context.

For example, the tactic `simpl in H` performs simplification in the hypothesis named  $H$  in the context.

**Theorem S\_inj** :  $\forall (n\ m : \text{nat}) (b : \text{bool}),$   
 $\text{beq\_nat } (\text{S } n) (\text{S } m) = b \rightarrow$   
 $\text{beq\_nat } n\ m = b.$

**Proof.**

`intros n m b H. simpl in H. apply H. Qed.`

Similarly, `apply L in H` matches some conditional statement  $L$  (of the form  $L1 \rightarrow L2$ , say) against a hypothesis  $H$  in the context. However, unlike ordinary `apply` (which rewrites a

goal matching  $L2$  into a subgoal  $L1$ ), `apply L in H` matches  $H$  against  $L1$  and, if successful, replaces it with  $L2$ .

In other words, `apply L in H` gives us a form of “forward reasoning”: from  $L1 \rightarrow L2$  and a hypothesis matching  $L1$ , it produces a hypothesis matching  $L2$ . By contrast, `apply L` is “backward reasoning”: it says that if we know  $L1 \rightarrow L2$  and we are trying to prove  $L2$ , it suffices to prove  $L1$ .

Here is a variant of a proof from above, using forward reasoning throughout instead of backward reasoning.

```
Theorem silly3' : ∀ (n : nat),
  (beq_nat n 5 = true → beq_nat (S (S n)) 7 = true) →
  true = beq_nat n 5 →
  true = beq_nat (S (S n)) 7.
```

Proof.

```
intros n eq H.
symmetry in H. apply eq in H. symmetry in H.
apply H. Qed.
```

Forward reasoning starts from what is *given* (premises, previously proven theorems) and iteratively draws conclusions from them until the goal is reached. Backward reasoning starts from the *goal*, and iteratively reasons about what would imply the goal, until premises or previously proven theorems are reached. If you’ve seen informal proofs before (for example, in a math or computer science class), they probably used forward reasoning. In general, idiomatic use of Coq tends to favor backward reasoning, but in some situations the forward style can be easier to think about.

**Exercise: 3 stars, recommended (plus\_n\_n\_injective)** Practice using “in” variants in this exercise. (Hint: use `plus_n_Sm`.)

```
Theorem plus_n_n_injective : ∀ n m,
  n + n = m + m →
  n = m.
```

Proof.

```
intros n. induction n as [| n'].
Admitted.
□
```

## 6.6 Varying the Induction Hypothesis

Sometimes it is important to control the exact form of the induction hypothesis when carrying out inductive proofs in Coq. In particular, we need to be careful about which of the assumptions we move (using `intros`) from the goal to the context before invoking the `induction` tactic. For example, suppose we want to show that the `double` function is injective – i.e., that it always maps different arguments to different results: Theorem `double_injective`:



forall n m, double n = double m -> n = m. The way we *start* this proof is a bit delicate: if we begin with `intros n. induction n.` all is well. But if we begin it with `intros n m. induction n.` we get stuck in the middle of the inductive case...

Theorem `double_injective_FAILED` :  $\forall n\ m,$   
`double n = double m →`  
`n = m.`

Proof.

```
intros n m. induction n as [| n'].
- simpl. intros eq. destruct m as [| m'].
  + reflexivity.
  + inversion eq.
- intros eq. destruct m as [| m'].
  + inversion eq.
  + apply f_equal.
```

At this point, the induction hypothesis,  $IHn'$ , does not give us  $n' = m'$  – there is an extra  $S$  in the way – so the goal is not provable.

Abort.

What went wrong?

The problem is that, at the point we invoke the induction hypothesis, we have already introduced  $m$  into the context – intuitively, we have told Coq, “Let’s consider some particular  $n$  and  $m$ ...” and we now have to prove that, if `double n = double m` for *these particular*  $n$  and  $m$ , then  $n = m$ .

The next tactic, `induction n` says to Coq: We are going to show the goal by induction on  $n$ . That is, we are going to prove, for *all*  $n$ , that the proposition

- $P\ n =$  “if `double n = double m`, then  $n = m$ ”

holds, by showing

- $P\ 0$   
 (i.e., “if `double 0 = double m` then  $0 = m$ ”) and
- $P\ n \rightarrow P\ (S\ n)$   
 (i.e., “if `double n = double m` then  $n = m$ ” implies “if `double (S n) = double m` then  $S\ n = m$ ”).

If we look closely at the second statement, it is saying something rather strange: it says that, for a *particular*  $m$ , if we know

- “if `double n = double m` then  $n = m$ ”

then we can prove

- “if  $\text{double } (S\ n) = \text{double } m$  then  $S\ n = m$ ”.

To see why this is strange, let’s think of a particular  $m$  – say, 5. The statement is then saying that, if we know

- $Q = \text{“if } \text{double } n = 10 \text{ then } n = 5\text{”}$

then we can prove

- $R = \text{“if } \text{double } (S\ n) = 10 \text{ then } S\ n = 5\text{”}$ .

But knowing  $Q$  doesn’t give us any help at all with proving  $R$ ! (If we tried to prove  $R$  from  $Q$ , we would start with something like “Suppose  $\text{double } (S\ n) = 10\dots$ ” but then we’d be stuck: knowing that  $\text{double } (S\ n)$  is 10 tells us nothing about whether  $\text{double } n$  is 10, so  $Q$  is useless.)

To summarize: Trying to carry out this proof by induction on  $n$  when  $m$  is already in the context doesn’t work because we are then trying to prove a relation involving *every*  $n$  but just a *single*  $m$ .

The good proof of `double_injective` leaves  $m$  in the goal statement at the point where the `induction` tactic is invoked on  $n$ :

```
Theorem double_injective : ∀ n m,
  double n = double m →
  n = m.
```

Proof.

```
intros n. induction n as [| n'].
- simpl. intros m eq. destruct m as [| m'].
  + reflexivity.
  + inversion eq.
- simpl.
```

Notice that both the goal and the induction hypothesis are different this time: the goal asks us to prove something more general (i.e., to prove the statement for *every*  $m$ ), but the IH is correspondingly more flexible, allowing us to choose any  $m$  we like when we apply the IH.

```
intros m eq.
```

Now we’ve chosen a particular  $m$  and introduced the assumption that  $\text{double } n = \text{double } m$ . Since we are doing a case analysis on  $n$ , we also need a case analysis on  $m$  to keep the two “in sync.”

```
destruct m as [| m'].
+ simpl.
```

The 0 case is trivial:

```
inversion eq.
```

```
+
  apply f_equal.
```

At this point, since we are in the second branch of the `destruct m`, the  $m'$  mentioned in the context is the predecessor of the `m` we started out talking about. Since we are also in the `S` branch of the induction, this is perfect: if we instantiate the generic `m` in the IH with the current  $m'$  (this instantiation is performed automatically by the `apply` in the next step), then `IHn'` gives us exactly what we need to finish the proof.

```
  apply IHn'. inversion eq. reflexivity. Qed.
```

What you should take away from all this is that we need to be careful about using induction to try to prove something too specific: If we're proving a property of `n` and `m` by induction on `n`, we may need to leave `m` generic.

The following exercise requires the same pattern.

**Exercise: 2 stars (beq\_nat\_true)** Theorem `beq_nat_true` :  $\forall n m,$   
`beq_nat n m = true  $\rightarrow$  n = m.`

Proof.

*Admitted.*

□

**Exercise: 2 stars, advanced (beq\_nat\_true\_informal)** Give a careful informal proof of `beq_nat_true`, being as explicit as possible about quantifiers.

□

The strategy of doing fewer `intros` before an `induction` to obtain a more general IH doesn't always work by itself; sometimes a little *rearrangement* of quantified variables is needed. Suppose, for example, that we wanted to prove `double_injective` by induction on `m` instead of `n`.

Theorem `double_injective_take2_FAILED` :  $\forall n m,$   
`double n = double m  $\rightarrow$`   
`n = m.`

Proof.

```
intros n m. induction m as [| m'].
- simpl. intros eq. destruct n as [| n'].
  + reflexivity.
  + inversion eq.
- intros eq. destruct n as [| n'].
  + inversion eq.
  + apply f_equal.
```

Abort.

The problem here is that, to do induction on `m`, we must first introduce `n`. (If we simply say `induction m` without introducing anything first, Coq will automatically introduce `n` for us!)

What can we do about this? One possibility is to rewrite the statement of the lemma so that  $m$  is quantified before  $n$ . This will work, but it's not nice: We don't want to have to twist the statements of lemmas to fit the needs of a particular strategy for proving them – we want to state them in the most clear and natural way.

What we can do instead is to first introduce all the quantified variables and then *re-generalize* one or more of them, selectively taking variables out of the context and putting them back at the beginning of the goal. The `generalize dependent` tactic does this.

Theorem `double_injective_take2` :  $\forall n\ m,$

`double n = double m`  $\rightarrow$

`n = m`.

Proof.

`intros n m.`

`generalize dependent n.`

`induction m as [| m'].`

- `simpl. intros n eq. destruct n as [| n'].`

`+ reflexivity.`

`+ inversion eq.`

- `intros n eq. destruct n as [| n'].`

`+ inversion eq.`

`+ apply f_equal.`

`apply IHm'. inversion eq. reflexivity. Qed.`

Let's look at an informal proof of this theorem. Note that the proposition we prove by induction leaves  $n$  quantified, corresponding to the use of `generalize dependent` in our formal proof.

*Theorem:* For any nats  $n$  and  $m$ , if `double n = double m`, then  $n = m$ .

*Proof:* Let  $m$  be a nat. We prove by induction on  $m$  that, for any  $n$ , if `double n = double m` then  $n = m$ .

- First, suppose  $m = 0$ , and suppose  $n$  is a number such that `double n = double m`. We must show that  $n = 0$ .

Since  $m = 0$ , by the definition of `double` we have `double n = 0`. There are two cases to consider for  $n$ . If  $n = 0$  we are done, since  $m = 0 = n$ , as required. Otherwise, if  $n = S\ n'$  for some  $n'$ , we derive a contradiction: by the definition of `double`, we can calculate `double n = S (S (double n'))`, but this contradicts the assumption that `double n = 0`.

- Second, suppose  $m = S\ m'$  and that  $n$  is again a number such that `double n = double m`. We must show that  $n = S\ m'$ , with the induction hypothesis that for every number  $s$ , if `double s = double m'` then  $s = m'$ .

By the fact that  $m = S\ m'$  and the definition of `double`, we have `double n = S (S (double m'))`. There are two cases to consider for  $n$ .

If  $n = 0$ , then by definition `double n = 0`, a contradiction.

Thus, we may assume that  $n = S \ n'$  for some  $n'$ , and again by the definition of `double` we have  $S \ (S \ (\text{double } n')) = S \ (S \ (\text{double } m'))$ , which implies by inversion that  $\text{double } n' = \text{double } m'$ . Instantiating the induction hypothesis with  $n'$  thus allows us to conclude that  $n' = m'$ , and it follows immediately that  $S \ n' = S \ m'$ . Since  $S \ n' = n$  and  $S \ m' = m$ , this is just what we wanted to show.  $\square$

Before we close this section and move on to some exercises, let's digress briefly and use `beq_nat_true` to prove a similar property about identifiers that we'll need in later chapters:

Theorem `beq_id_true` :  $\forall x \ y,$   
 $\text{beq\_id } x \ y = \text{true} \rightarrow x = y.$

Proof.

```
intros [m] [n]. simpl. intros H.
assert (H' : m = n). { apply beq_nat_true. apply H. }
rewrite H'. reflexivity.
```

Qed.

**Exercise: 3 stars, recommended (gen\_dep\_practice)** Prove this by induction on `l`.

Theorem `nth_error_after_last`:  $\forall (n : \text{nat}) (X : \text{Type}) (l : \text{list } X),$   
 $\text{length } l = n \rightarrow$   
 $\text{nth\_error } l \ n = \text{None}.$

Proof.

*Admitted.*

$\square$

**Exercise: 3 stars, optional (app\_length\_cons)** Prove this by induction on `l1`, without using `app_length` from `Lists`.

Theorem `app_length_cons` :  $\forall (X : \text{Type}) (l1 \ l2 : \text{list } X)$   
 $(x : X) (n : \text{nat}),$   
 $\text{length } (l1 ++ (x :: l2)) = n \rightarrow$   
 $S \ (\text{length } (l1 ++ l2)) = n.$

Proof.

*Admitted.*

$\square$

**Exercise: 4 stars, optional (app\_length\_twice)** Prove this by induction on `l`, without using `app_length` from `Lists`.

Theorem `app_length_twice` :  $\forall (X : \text{Type}) (n : \text{nat}) (l : \text{list } X),$   
 $\text{length } l = n \rightarrow$   
 $\text{length } (l ++ l) = n + n.$

Proof.

*Admitted.*

$\square$

**Exercise: 3 stars, optional (double\_induction)** Prove the following principle of induction over two naturals.

Theorem double\_induction:  $\forall (P : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop}),$

$P\ 0\ 0 \rightarrow$   
 $(\forall m, P\ m\ 0 \rightarrow P\ (\text{S } m)\ 0) \rightarrow$   
 $(\forall n, P\ 0\ n \rightarrow P\ 0\ (\text{S } n)) \rightarrow$   
 $(\forall m\ n, P\ m\ n \rightarrow P\ (\text{S } m)\ (\text{S } n)) \rightarrow$   
 $\forall m\ n, P\ m\ n.$

Proof.

*Admitted.*

□

## 6.7 Unfolding Definitions

It sometimes happens that we need to manually unfold a Definition so that we can manipulate its right-hand side. For example, if we define

Definition square  $n := n \times n.$

and try to prove a simple fact about square...

Lemma square\_mult :  $\forall n\ m, \text{square } (n \times m) = \text{square } n \times \text{square } m.$

Proof.

intros  $n\ m.$   
simpl.

... we get stuck: `simpl` doesn't simplify anything at this point, and since we haven't proved any other facts about `square`, there is nothing we can `apply` or `rewrite` with.

To make progress, we can manually `unfold` the definition of `square`:

`unfold square.`

Now we have plenty to work with: both sides of the equality are expressions involving multiplication, and we have lots of facts about multiplication at our disposal. In particular, we know that it is commutative and associative, and from these facts it is not hard to finish the proof.

`rewrite mult_assoc.`  
`assert (H :  $n \times m \times n = n \times n \times m$ ).`  
`{ rewrite mult_comm. apply mult_assoc. }`  
`rewrite H. rewrite mult_assoc. reflexivity.`

Qed.

At this point, a slightly deeper discussion of unfolding and simplification is in order.

You may already have observed that tactics like `simpl`, `reflexivity`, and `apply` will often unfold the definitions of functions automatically when it allows them to make progress. For example, if we define `foo m` to be the constant 5,

Definition foo ( $x$ : nat) := 5.

then the `simpl` in the following proof (or the `reflexivity`, if we omit the `simpl`) will unfold `foo m` to `(fun x => 5) m` and then further simplify this expression to just 5.

Fact silly\_fact\_1 :  $\forall m$ , `foo m + 1 = foo (m + 1) + 1`.

Proof.

```
intros m.
simpl.
reflexivity.
```

Qed.

However, this automatic unfolding is rather conservative. For example, if we define a slightly more complicated function involving a pattern match...

Definition bar  $x$  :=

```
match x with
| O => 5
| S _ => 5
end.
```

...then the analogous proof will get stuck:

Fact silly\_fact\_2\_FAILED :  $\forall m$ , `bar m + 1 = bar (m + 1) + 1`.

Proof.

```
intros m.
simpl. Abort.
```

The reason that `simpl` doesn't make progress here is that it notices that, after tentatively unfolding `bar m`, it is left with a match whose scrutinee, `m`, is a variable, so the `match` cannot be simplified further. (It is not smart enough to notice that the two branches of the `match` are identical.) So it gives up on unfolding `bar m` and leaves it alone. Similarly, tentatively unfolding `bar (m+1)` leaves a `match` whose scrutinee is a function application (that, itself, cannot be simplified, even after unfolding the definition of `+`), so `simpl` leaves it alone.

At this point, there are two ways to make progress. One is to use `destruct m` to break the proof into two cases, each focusing on a more concrete choice of `m` (`O` vs `S _`). In each case, the `match` inside of `bar` can now make progress, and the proof is easy to complete.

Fact silly\_fact\_2 :  $\forall m$ , `bar m + 1 = bar (m + 1) + 1`.

Proof.

```
intros m.
destruct m.
- simpl. reflexivity.
- simpl. reflexivity.
```

Qed.

This approach works, but it depends on our recognizing that the `match` hidden inside `bar` is what was preventing us from making progress.

A more straightforward way to finish the proof is to explicitly tell Coq to unfold `bar`.

Fact silly\_fact\_2' :  $\forall m, \text{bar } m + 1 = \text{bar } (m + 1) + 1$ .

Proof.

```
intros m.
unfold bar.
```

Now it is apparent that we are stuck on the `match` expressions on both sides of the `=`, and we can use `destruct` to finish the proof without thinking too hard.

```
destruct m.
- reflexivity.
- reflexivity.
Qed.
```

## 6.8 Using destruct on Compound Expressions

We have seen many examples where `destruct` is used to perform case analysis of the value of some variable. But sometimes we need to reason by cases on the result of some *expression*. We can also do this with `destruct`.

Here are some examples:

Definition sillyfun ( $n : \text{nat}$ ) : `bool` :=  
 if beq\_nat n 3 then false  
 else if beq\_nat n 5 then false  
 else false.

Theorem sillyfun\_false :  $\forall (n : \text{nat})$ ,  
 sillyfun n = false.

Proof.

```
intros n. unfold sillyfun.
destruct (beq_nat n 3).
- reflexivity.
- destruct (beq_nat n 5).
  + reflexivity.
  + reflexivity. Qed.
```

After unfolding `sillyfun` in the above proof, we find that we are stuck on `if (beq_nat n 3) then ... else ...`. But either `n` is equal to 3 or it isn't, so we can use `destruct (beq_nat n 3)` to let us reason about the two cases.

In general, the `destruct` tactic can be used to perform case analysis of the results of arbitrary computations. If `e` is an expression whose type is some inductively defined type `T`, then, for each constructor `c` of `T`, `destruct e` generates a subgoal in which all occurrences of `e` (in the goal and in the context) are replaced by `c`.

**Exercise: 3 stars, optional (combine\_split)** Theorem combine\_split :  $\forall X Y (l : \text{list } (X \times Y)) \ l1 \ l2$ ,



```
split l = (l1, l2) →
combine l1 l2 = l.
```

Proof.

*Admitted.*

□

However, **destructing** compound expressions requires a bit of care, as such **destructs** can sometimes erase information we need to complete a proof. For example, suppose we define a function `sillyfun1` like this:

```
Definition sillyfun1 (n : nat) : bool :=
  if beq_nat n 3 then true
  else if beq_nat n 5 then true
  else false.
```

Now suppose that we want to convince Coq of the (rather obvious) fact that `sillyfun1 n` yields `true` only when `n` is odd. By analogy with the proofs we did with `sillyfun` above, it is natural to start the proof like this:

```
Theorem sillyfun1_odd_FAILED : ∀ (n : nat),
  sillyfun1 n = true →
  oddb n = true.
```

Proof.

```
intros n eq. unfold sillyfun1 in eq.
destruct (beq_nat n 3).
```

Abort.

We get stuck at this point because the context does not contain enough information to prove the goal! The problem is that the substitution performed by **destruct** is too brutal – it threw away every occurrence of `beq_nat n 3`, but we need to keep some memory of this expression and how it was destructed, because we need to be able to reason that, since `beq_nat n 3 = true` in this branch of the case analysis, it must be that `n = 3`, from which it follows that `n` is odd.

What we would really like is to substitute away all existing occurrences of `beq_nat n 3`, but at the same time add an equation to the context that records which case we are in. The *eqn*: qualifier allows us to introduce such an equation, giving it a name that we choose.

```
Theorem sillyfun1_odd : ∀ (n : nat),
  sillyfun1 n = true →
  oddb n = true.
```

Proof.

```
intros n eq. unfold sillyfun1 in eq.
destruct (beq_nat n 3) eqn:Heq3.
- apply beq_nat_true in Heq3.
  rewrite → Heq3. reflexivity.
-
```

```

destruct (beq_nat n 5) eqn:Heqe5.
+
  apply beq_nat_true in Heqe5.
  rewrite → Heqe5.reflexivity.
+ inversion eq. Qed.

```

**Exercise: 2 stars (destruct\_eqn\_practice)** Theorem `bool_fn_applied_thrice` :

$\forall (f : \mathbf{bool} \rightarrow \mathbf{bool}) (b : \mathbf{bool}),$   
 $f (f (f b)) = f b.$

Proof.

*Admitted.*

□

## 6.9 Review

We've now seen many of Coq's most fundamental tactics. We'll introduce a few more in the coming chapters, and later on we'll see some more powerful *automation* tactics that make Coq help us with low-level details. But basically we've got what we need to get work done.

Here are the ones we've seen:

- **intros**: move hypotheses/variables from goal to context
- **reflexivity**: finish the proof (when the goal looks like `e = e`)
- **apply**: prove goal using a hypothesis, lemma, or constructor
- **apply... in *H***: apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning)
- **apply... with...**: explicitly specify values for variables that cannot be determined by pattern matching
- **simpl**: simplify computations in the goal
- **simpl in *H*: ...** or a hypothesis
- **rewrite**: use an equality hypothesis (or lemma) to rewrite the goal
- **rewrite ... in *H*: ...** or a hypothesis
- **symmetry**: changes a goal of the form  $t=u$  into  $u=t$
- **symmetry in *H***: changes a hypothesis of the form  $t=u$  into  $u=t$
- **unfold**: replace a defined constant by its right-hand side in the goal

- `unfold... in H: ...` or a hypothesis
- `destruct... as...:` case analysis on values of inductively defined types
- `destruct... eqn:...:` specify the name of an equation to be added to the context, recording the result of the case analysis
- `induction... as...:` induction on values of inductively defined types
- `inversion:` reason by injectivity and distinctness of constructors
- `assert (e) as H:` introduce a “local lemma” `e` and call it `H`
- `generalize dependent x:` move the variable `x` (and anything else that depends on it) from the context back to an explicit hypothesis in the goal formula

## 6.10 Additional Exercises

**Exercise: 3 stars (beq\_nat\_sym)** Theorem `beq_nat_sym` :  $\forall (n\ m : \text{nat}),$   
`beq_nat n m = beq_nat m n.`

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced, optional (beq\_nat\_sym\_informal)** Give an informal proof of this lemma that corresponds to your formal proof above:

Theorem: For any nats `n m`, `beq_nat n m = beq_nat m n.`

Proof: □

**Exercise: 3 stars, optional (beq\_nat\_trans)** Theorem `beq_nat_trans` :  $\forall\ n\ m\ p,$

`beq_nat n m = true`  $\rightarrow$

`beq_nat m p = true`  $\rightarrow$

`beq_nat n p = true.`

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced (split\_combine)** We proved, in an exercise above, that for all lists of pairs, `combine` is the inverse of `split`. How would you formalize the statement that `split` is the inverse of `combine`? When is this property true?

Complete the definition of `split_combine_statement` below with a property that states that `split` is the inverse of `combine`. Then, prove that the property holds. (Be sure to leave your

induction hypothesis general by not doing `intros` on more things than necessary. Hint: what property do you need of `l1` and `l2` for `split combine l1 l2 = (l1,l2)` to be true?)

Definition `split_combine_statement` : Prop :=  
*admit.*

Theorem `split_combine` : `split_combine_statement`.

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced (filter\_exercise)** This one is a bit challenging. Pay attention to the form of your induction hypothesis.

Theorem `filter_exercise` :  $\forall (X : \text{Type}) (test : X \rightarrow \text{bool})$   
 $(x : X) (l \text{ lf} : \text{list } X),$

`filter test l = x :: lf` →  
`test x = true.`

Proof.

*Admitted.*

□

**Exercise: 4 stars, advanced, recommended (forall\_exists\_challenge)** Define two recursive *Fixpoints*, `forallb` and `existsb`. The first checks whether every element in a list satisfies a given predicate: `forallb oddb 1;3;5;7;9 = true`

`forallb negb false>false = true`

`forallb evenb 0;2;4;5 = false`

`forallb (beq_nat 5) [] = true` The second checks whether there exists an element in the list that satisfies a given predicate: `existsb (beq_nat 5) 0;2;3;6 = false`

`existsb (andb true) true>true>false = true`

`existsb oddb 1;0;0;0;0;3 = true`

`existsb evenb [] = false` Next, define a *nonrecursive* version of `existsb` – call it `existsb'` – using `forallb` and `negb`.

Finally, prove a theorem `existsb_existsb'` stating that `existsb'` and `existsb` have the same behavior.

□

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500(Wed, 17Feb2016)

# Chapter 7

## Library Logic

### 7.1 Logic: Logic in Coq

Require Export Tactics.

In previous chapters, we have seen many examples of factual claims (*propositions*) and ways of presenting evidence of their truth (*proofs*). In particular, we have worked extensively with *equality propositions* of the form  $e1 = e2$ , with implications ( $P \rightarrow Q$ ), and with quantified propositions ( $\forall x, P$ ). In this chapter, we will see how Coq can be used to carry out other familiar forms of logical reasoning.

Before diving into details, let's talk a bit about the status of mathematical statements in Coq. Recall that Coq is a *typed* language, which means that every sensible expression in its world has an associated type. Logical claims are no exception: any statement we might try to prove in Coq has a type, namely **Prop**, the type of *propositions*. We can see this with the **Check** command:

Check 3 = 3.

Check  $\forall n\ m : \text{nat}, n + m = m + n$ .

Note that all well-formed propositions have type **Prop** in Coq, regardless of whether they are true or not. Simply *being* a proposition is one thing; being *provable* is something else!

Check  $\forall n : \text{nat}, n = 2$ .

Check 3 = 4.

Indeed, propositions don't just have types: they are *first-class objects* that can be manipulated in the same ways as the other entities in Coq's world. So far, we've seen one primary place that propositions can appear: in **Theorem** (and **Lemma** and **Example**) declarations.

**Theorem** plus\_2\_2\_is\_4 :

2 + 2 = 4.

**Proof.** reflexivity. **Qed.**

But propositions can be used in many other ways. For example, we can give a name to a proposition using a **Definition**, just as we have given names to expressions of other sorts.

```
Definition plus_fact : Prop := 2 + 2 = 4.
Check plus_fact.
```

We can later use this name in any situation where a proposition is expected – for example, as the claim in a **Theorem** declaration.

```
Theorem plus_fact_is_true :
  plus_fact.
```

```
Proof. reflexivity. Qed.
```

We can also write *parameterized* propositions – that is, functions that take arguments of some type and return a proposition. For instance, the following function takes a number and returns a proposition asserting that this number is equal to three:

```
Definition is_three (n : nat) : Prop :=
  n = 3.
Check is_three.
```

In Coq, functions that return propositions are said to define *properties* of their arguments. For instance, here’s a polymorphic property defining the familiar notion of an *injective function*.

```
Definition injective {A B} (f : A → B) :=
  ∀ x y : A, f x = f y → x = y.
```

```
Lemma succ_inj : injective S.
```

```
Proof.
```

```
  intros n m H. inversion H. reflexivity.
```

```
Qed.
```

The equality operator `=` that we have been using so far is also just a function that returns a **Prop**. The expression `n = m` is just syntactic sugar for `eq n m`, defined using Coq’s **Notation** mechanism. Because `=` can be used with elements of any type, it is also polymorphic:

```
Check @eq.
```

(Notice that we wrote `@eq` instead of `eq`: The type argument **A** to `eq` is declared as implicit, so we need to turn off implicit arguments to see the full type of `eq`.)

## 7.2 Logical Connectives

### 7.2.1 Conjunction

The *conjunction* or *logical and* of propositions **A** and **B** is written  $A \wedge B$ , denoting the claim that both **A** and **B** are true.

```
Example and_example : 3 + 4 = 7 ∧ 2 × 2 = 4.
```

To prove a conjunction, use the `split` tactic. Its effect is to generate two subgoals, one for each part of the statement:

Proof.

```
split.  
- reflexivity.  
- reflexivity.
```

Qed.

More generally, the following principle works for any two propositions  $A$  and  $B$ :

Lemma `and_intro` :  $\forall A B : \text{Prop}, A \rightarrow B \rightarrow A \wedge B$ .

Proof.

```
intros A B HA HB. split.  
- apply HA.  
- apply HB.
```

Qed.

A logical statement with multiple arrows is just a theorem that has several hypotheses. Here, `and_intro` says that, for any propositions  $A$  and  $B$ , if we assume that  $A$  is true and we assume that  $B$  is true, then  $A \wedge B$  is also true.

Since applying a theorem with hypotheses to some goal has the effect of generating as many subgoals as there are hypotheses for that theorem, we can, apply `and_intro` to achieve the same effect as `split`.

Example `and_example'` :  $3 + 4 = 7 \wedge 2 \times 2 = 4$ .

Proof.

```
apply and_intro.  
- reflexivity.  
- reflexivity.
```

Qed.

**Exercise: 2 stars (`and_exercise`)** Example `and_exercise` :

$\forall n m : \text{nat}, n + m = 0 \rightarrow n = 0 \wedge m = 0$ .

Proof.

*Admitted.*

□

So much for proving conjunctive statements. To go in the other direction – i.e., to *use* a conjunctive hypothesis to prove something else – we employ the `destruct` tactic.

If the proof context contains a hypothesis  $H$  of the form  $A \wedge B$ , writing `destruct H as [HA HB]` will remove  $H$  from the context and add two new hypotheses:  $HA$ , stating that  $A$  is true, and  $HB$ , stating that  $B$  is true. For instance:

Lemma `and_example2` :

$\forall n m : \text{nat}, n = 0 \wedge m = 0 \rightarrow n + m = 0$ .

Proof.

```
intros n m H.  
destruct H as [Hn Hm].  
rewrite Hn. rewrite Hm.
```

```

    reflexivity.
Qed.

```

As usual, we can also destruct  $H$  when we introduce it instead of introducing and then destructing it:

```

Lemma and_example2' :
   $\forall n\ m : \text{nat}, n = 0 \wedge m = 0 \rightarrow n + m = 0.$ 

```

```

Proof.
  intros n m [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.

```

You may wonder why we bothered packing the two hypotheses  $n = 0$  and  $m = 0$  into a single conjunction, since we could have also stated the theorem with two separate premises:

```

Lemma and_example2'' :
   $\forall n\ m : \text{nat}, n = 0 \rightarrow m = 0 \rightarrow n + m = 0.$ 

```

```

Proof.
  intros n m Hn Hm.
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.

```

In this case, there is not much difference between the two theorems. But it is often necessary to explicitly decompose conjunctions that arise from intermediate steps in proofs, especially in bigger developments. Here's a simplified example:

```

Lemma and_example3 :
   $\forall n\ m : \text{nat}, n + m = 0 \rightarrow n \times m = 0.$ 

```

```

Proof.
  intros n m H.
  assert (H' :  $n = 0 \wedge m = 0$ ).
  { apply and_exercise. apply H. }
  destruct H' as [Hn Hm].
  rewrite Hn. reflexivity.
Qed.

```

Another common situation with conjunctions is that we know  $A \wedge B$  but in some context we need just  $A$  (or just  $B$ ). The following lemmas are useful in such cases:

```

Lemma proj1 :  $\forall P\ Q : \text{Prop},$ 
   $P \wedge Q \rightarrow P.$ 

```

```

Proof.
  intros P Q [HP HQ].
  apply HP. Qed.

```



**Exercise: 1 star, optional (proj2)** Lemma `proj2` :  $\forall P Q : \text{Prop},$   
 $P \wedge Q \rightarrow Q.$

Proof.

*Admitted.*

□

Finally, we sometimes need to rearrange the order of conjunctions and/or the grouping of conjuncts in multi-way conjunctions. The following commutativity and associativity theorems come in handy in such cases.

Theorem `and_commut` :  $\forall P Q : \text{Prop},$   
 $P \wedge Q \rightarrow Q \wedge P.$

Proof.

`intros P Q [HP HQ].`

`split.`

- `apply HQ.`

- `apply HP. Qed.`

**Exercise: 2 stars (and\_assoc)** (In the following proof of associativity, notice how the *nested* intro pattern breaks the hypothesis  $H : P \wedge (Q \wedge R)$  down into  $HP : P$ ,  $HQ : Q$ , and  $HR : R$ . Finish the proof from there.)

Theorem `and_assoc` :  $\forall P Q R : \text{Prop},$   
 $P \wedge (Q \wedge R) \rightarrow (P \wedge Q) \wedge R.$

Proof.

`intros P Q R [HP [HQ HR]].`

*Admitted.*

□

By the way, the infix notation  $\wedge$  is actually just syntactic sugar for **and**  $A B$ . That is, **and** is a Coq operator that takes two propositions as arguments and yields a proposition.

Check **and**.

## 7.2.2 Disjunction

Another important connective is the *disjunction*, or *logical or* of two propositions:  $A \vee B$  is true when either  $A$  or  $B$  is. (Alternatively, we can write **or**  $A B$ , where **or** :  $\text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop}.$ )

To use a disjunctive hypothesis in a proof, we proceed by case analysis, which, as for `nat` or other data types, can be done with `destruct` or `intros`. Here is an example:

Lemma `or_example` :

$\forall n m : \text{nat}, n = 0 \vee m = 0 \rightarrow n \times m = 0.$

Proof.

`intros n m [Hn | Hm].`

-

```

    rewrite Hn. reflexivity.
-
    rewrite Hm. rewrite ← mult_n_0.
    reflexivity.
Qed.

```

We can see in this example that, when we perform case analysis on a disjunction  $A \vee B$ , we must satisfy two proof obligations, each showing that the conclusion holds under a different assumption –  $A$  in the first subgoal and  $B$  in the second. Note that the case analysis pattern  $(Hn \mid Hm)$  allows us to name the hypothesis that is generated in each subgoal.

Conversely, to show that a disjunction holds, we need to show that one of its sides does. This is done via two tactics, **left** and **right**. As their names imply, the first one requires proving the left side of the disjunction, while the second requires proving its right side. Here is a trivial use...

Lemma `or_intro` :  $\forall A B : \text{Prop}, A \rightarrow A \vee B$ .

Proof.

```

  intros A B HA.

```

```

  left.

```

```

  apply HA.

```

Qed.

... and a slightly more interesting example requiring the use of both **left** and **right**:

Lemma `zero_or_succ` :

$\forall n : \text{nat}, n = 0 \vee n = S(\text{pred } n)$ .

Proof.

```

  intros [|n].

```

```

  - left. reflexivity.

```

```

  - right. reflexivity.

```

Qed.

**Exercise: 1 star (mult\_eq\_0)** Lemma `mult_eq_0` :

$\forall n m, n \times m = 0 \rightarrow n = 0 \vee m = 0$ .

Proof.

*Admitted.*

□

**Exercise: 1 star (or\_commut)** Theorem `or_commut` :  $\forall P Q : \text{Prop},$

$P \vee Q \rightarrow Q \vee P$ .

Proof.

*Admitted.*

□

### 7.2.3 Falsehood and Negation

So far, we have mostly been concerned with proving that certain things are *true* – addition is commutative, appending lists is associative, etc. Of course, we may also be interested in *negative* results, showing that certain propositions are *not* true. In Coq, such negative statements are expressed with the negation operator  $\neg$ .

To see how negation works, recall the discussion of the *principle of explosion* from the Tactics chapter; it asserts that, if we assume a contradiction, then any other proposition can be derived. Following this intuition, we could define  $\neg P$  (“not  $P$ ”) as  $\forall Q, P \rightarrow Q$ . Coq actually makes a slightly different choice, defining  $\neg P$  as  $P \rightarrow \mathbf{False}$ , where **False** is a *particular* contradictory proposition defined in the standard library.

Module MYNOT.

Definition not (P:Prop) := P  $\rightarrow$  **False**.

Notation "~ x" := (not x) : type\_scope.

Check not.

End MYNOT.

Since **False** is a contradictory proposition, the principle of explosion also applies to it. If we get **False** into the proof context, we can **destruct** it to complete any goal:

Theorem ex\_falso\_quodlibet :  $\forall (P:\text{Prop})$ ,  
**False**  $\rightarrow P$ .

Proof.

intros P contra.

destruct contra. Qed.

The Latin *ex falso quodlibet* means, literally, “from falsehood follows whatever you like”; this is another common name for the principle of explosion.

**Exercise: 2 stars, optional (not\_implies\_our\_not)** Show that Coq’s definition of negation implies the intuitive one mentioned above:

Fact not\_implies\_our\_not :  $\forall (P:\text{Prop})$ ,  
 $\neg P \rightarrow (\forall (Q:\text{Prop}), P \rightarrow Q)$ .

Proof.

Admitted.

□

This is how we use **not** to state that 0 and 1 are different elements of **nat**:

Theorem zero\_not\_one :  $\sim (0 = 1)$ .

Proof.

intros contra. inversion contra.

Qed.

Such inequality statements are frequent enough to warrant a special notation,  $x \neq y$ :

Check ( $0 \neq 1$ ).

Theorem zero\_not\_one' :  $0 \neq 1$ .

Proof.

intros  $H$ . inversion  $H$ .

Qed.

It takes a little practice to get used to working with negation in Coq. Even though you can see perfectly well why a statement involving negation is true, it can be a little tricky at first to get things into the right configuration so that Coq can understand it! Here are proofs of a few familiar facts to get you warmed up.

Theorem not\_False :

$\neg \text{False}$ .

Proof.

unfold not. intros  $H$ . destruct  $H$ . Qed.

Theorem contradiction\_implies\_anything :  $\forall P Q : \text{Prop}$ ,

$(P \wedge \neg P) \rightarrow Q$ .

Proof.

intros  $P Q$  [ $HP HNA$ ]. unfold not in  $HNA$ .

apply  $HNA$  in  $HP$ . destruct  $HP$ . Qed.

Theorem double\_neg :  $\forall P : \text{Prop}$ ,

$P \rightarrow \sim\sim P$ .

Proof.

intros  $P H$ . unfold not. intros  $G$ . apply  $G$ . apply  $H$ . Qed.

**Exercise: 2 stars, advanced, recommended (double\_neg\_inf)** Write an informal proof of double\_neg:

*Theorem:*  $P$  implies  $\sim\sim P$ , for any proposition  $P$ .

*Proof:*  $\square$

**Exercise: 2 stars, recommended (contrapositive)** Theorem contrapositive :  $\forall P Q : \text{Prop}$ ,

$(P \rightarrow Q) \rightarrow (\neg Q \rightarrow \neg P)$ .

Proof.

*Admitted.*

$\square$

**Exercise: 1 star (not\_both\_true\_and\_false)** Theorem not\_both\_true\_and\_false :  $\forall P : \text{Prop}$ ,

$\neg (P \wedge \neg P)$ .

Proof.

*Admitted.*

$\square$

**Exercise: 1 star, advanced (informal\_not\_PNP)** Write an informal proof (in English) of the proposition  $\forall P : \text{Prop}, \sim(P \wedge \neg P)$ .

□

Similarly, since inequality involves a negation, it requires a little practice to be able to work with it fluently. Here is one useful trick. If you are trying to prove a goal that is nonsensical (e.g., the goal state is `false = true`), apply `ex_falso_quodlibet` to change the goal to **False**. This makes it easier to use assumptions of the form  $\neg P$  that may be available in the context – in particular, assumptions of the form  $x \neq y$ .

Theorem `not_true_is_false` :  $\forall b : \text{bool}$ ,

$b \neq \text{true} \rightarrow b = \text{false}$ .

Proof.

```
intros [] H.
-
  unfold not in H.
  apply ex_falso_quodlibet.
  apply H. reflexivity.
-
  reflexivity.
```

Qed.

Since reasoning with `ex_falso_quodlibet` is quite common, Coq provides a built-in tactic, *exfalso*, for applying it.

Theorem `not_true_is_false'` :  $\forall b : \text{bool}$ ,

$b \neq \text{true} \rightarrow b = \text{false}$ .

Proof.

```
intros [] H.
-
  unfold not in H.
  exfalso.      apply H. reflexivity.
- reflexivity.
```

Qed.

## 7.2.4 Truth

Besides **False**, Coq's standard library also defines **True**, a proposition that is trivially true. To prove it, we use the predefined constant `I` : **True**:

Lemma `True_is_true` : **True**.

Proof. apply `I`. Qed.

Unlike **False**, which is used extensively, **True** is used quite rarely, since it is trivial (and therefore uninteresting) to prove as a goal, and it carries no useful information as a hypothesis. But it can be quite useful when defining complex Props using conditionals or as a parameter to higher-order Props. We will see some examples such uses of **True** later on.

## 7.2.5 Logical Equivalence

The handy “if and only if” connective, which asserts that two propositions have the same truth value, is just the conjunction of two implications.

Module MYIFF.

Definition iff ( $P\ Q : \text{Prop}$ ) := ( $P \rightarrow Q$ )  $\wedge$  ( $Q \rightarrow P$ ).

Notation " $P \leftrightarrow Q$ " := (iff  $P\ Q$ )  
(at level 95, no associativity)  
: *type\_scope*.

End MYIFF.

Theorem iff\_sym :  $\forall P\ Q : \text{Prop}$ ,  
( $P \leftrightarrow Q$ )  $\rightarrow$  ( $Q \leftrightarrow P$ ).

Proof.

intros  $P\ Q$  [ $HAB\ HBA$ ].  
split.  
- apply  $HBA$ .  
- apply  $HAB$ . Qed.

Lemma not\_true\_iff\_false :  $\forall b$ ,  
 $b \neq \text{true} \leftrightarrow b = \text{false}$ .

Proof.

intros  $b$ . split.  
- apply not\_true\_is\_false.  
-  
intros  $H$ . rewrite  $H$ . intros  $H'$ . inversion  $H'$ .

Qed.

**Exercise: 1 star, optional (iff\_properties)** Using the above proof that  $\leftrightarrow$  is symmetric (iff\_sym) as a guide, prove that it is also reflexive and transitive.

Theorem iff\_refl :  $\forall P : \text{Prop}$ ,  
 $P \leftrightarrow P$ .

Proof.

*Admitted.*

Theorem iff\_trans :  $\forall P\ Q\ R : \text{Prop}$ ,  
( $P \leftrightarrow Q$ )  $\rightarrow$  ( $Q \leftrightarrow R$ )  $\rightarrow$  ( $P \leftrightarrow R$ ).

Proof.

*Admitted.*

□

**Exercise: 3 stars (or\_distributes\_over\_and)** Theorem or\_distributes\_over\_and :  $\forall P\ Q\ R : \text{Prop}$ ,

$$P \vee (Q \wedge R) \leftrightarrow (P \vee Q) \wedge (P \vee R).$$

Proof.

*Admitted.*

□

Some of Coq's tactics treat iff statements specially, avoiding the need for some low-level proof-state manipulation. In particular, `rewrite` and `reflexivity` can be used with iff statements, not just equalities. To enable this behavior, we need to import a special Coq library that allows rewriting with other formulas besides equality:

Require Import `Coq.Setoids.Setoid`.

Here is a simple example demonstrating how these tactics work with iff. First, let's prove a couple of basic iff equivalences:

Lemma `mult_0` :  $\forall n\ m, n \times m = 0 \leftrightarrow n = 0 \vee m = 0$ .

Proof.

`split.`

- `apply mult_eq_0.`

- `apply or_example.`

Qed.

Lemma `or_assoc` :

$\forall P\ Q\ R : \text{Prop}, P \vee (Q \vee R) \leftrightarrow (P \vee Q) \vee R$ .

Proof.

`intros P Q R. split.`

- `intros [H | [H | H]].`

  + `left. left. apply H.`

  + `left. right. apply H.`

  + `right. apply H.`

- `intros [[H | H] | H].`

  + `left. apply H.`

  + `right. left. apply H.`

  + `right. right. apply H.`

Qed.

We can now use these facts with `rewrite` and `reflexivity` to give smooth proofs of statements involving equivalences. Here is a ternary version of the previous `mult_0` result:

Lemma `mult_0_3` :

$\forall n\ m\ p, n \times m \times p = 0 \leftrightarrow n = 0 \vee m = 0 \vee p = 0$ .

Proof.

`intros n m p.`

`rewrite mult_0. rewrite mult_0. rewrite or_assoc.`

`reflexivity.`

Qed.

The `apply` tactic can also be used with  $\leftrightarrow$ . When given an equivalence as its argument, `apply` tries to guess which side of the equivalence to use.

Lemma apply\_iff\_example :

$\forall n\ m : \text{nat}, n \times m = 0 \rightarrow n = 0 \vee m = 0.$

Proof.

intros  $n\ m\ H$ . apply mult\_0. apply  $H$ .

Qed.

## 7.2.6 Existential Quantification

Another important logical connective is *existential quantification*. To say that there is some  $x$  of type  $T$  such that some property  $P$  holds of  $x$ , we write  $\exists x : T, P$ . As with  $\forall$ , the type annotation  $: T$  can be omitted if Coq is able to infer from the context what the type of  $x$  should be.

To prove a statement of the form  $\exists x, P$ , we must show that  $P$  holds for some specific choice of value for  $x$ , known as the *witness* of the existential. This is done in two steps: First, we explicitly tell Coq which witness  $t$  we have in mind by invoking the tactic  $\exists t$ ; then we prove that  $P$  holds after all occurrences of  $x$  are replaced by  $t$ . Here is an example:

Lemma four\_is\_even :  $\exists n : \text{nat}, 4 = n + n.$

Proof.

$\exists 2$ . reflexivity.

Qed.

Conversely, if we have an existential hypothesis  $\exists x, P$  in the context, we can destruct it to obtain a witness  $x$  and a hypothesis stating that  $P$  holds of  $x$ .

Theorem exists\_example\_2 :  $\forall n,$

$(\exists m, n = 4 + m) \rightarrow$

$(\exists o, n = 2 + o).$

Proof.

intros  $n\ [m\ Hm]$ .

$\exists (2 + m).$

apply  $Hm$ . Qed.

**Exercise: 1 star (dist\_not\_exists)** Prove that “ $P$  holds for all  $x$ ” implies “there is no  $x$  for which  $P$  does not hold.”

Theorem dist\_not\_exists :  $\forall (X:\text{Type}) (P : X \rightarrow \text{Prop}),$

$(\forall x, P\ x) \rightarrow \neg (\exists x, \neg P\ x).$

Proof.

Admitted.

□

**Exercise: 2 stars (dist\_exists\_or)** Prove that existential quantification distributes over disjunction.

Theorem dist\_exists\_or :  $\forall (X:\text{Type}) (P\ Q : X \rightarrow \text{Prop}),$



$$(\exists x, P x \vee Q x) \leftrightarrow (\exists x, P x) \vee (\exists x, Q x).$$

Proof.

*Admitted.*

□

## 7.3 Programming with Propositions

The logical connectives that we have seen provide a rich vocabulary for defining complex propositions from simpler ones. To illustrate, let's look at how to express the claim that an element  $x$  occurs in a list  $l$ . Notice that this property has a simple recursive structure:

- If  $l$  is the empty list, then  $x$  cannot occur on it, so the property “ $x$  appears in  $l$ ” is simply false.
- Otherwise,  $l$  has the form  $x' :: l'$ . In this case,  $x$  occurs in  $l$  if either it is equal to  $x'$  or it occurs in  $l'$ .

We can translate this directly into a straightforward Coq function, `ln`. (It can also be found in the Coq standard library.)

```
Fixpoint ln {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x ∨ ln x l'
  end.
```

When `ln` is applied to a concrete list, it expands into a concrete sequence of nested conjunctions.

```
Example ln_example_1 : ln 4 [3; 4; 5].
```

Proof.

```
simpl. right. left. reflexivity.
```

Qed.

```
Example ln_example_2 :
```

```
  ∀ n, ln n [2; 4] →
  ∃ n', n = 2 × n'.
```

Proof.

```
simpl.
intros n [H | [H | []]].
- ∃ 1. rewrite ← H. reflexivity.
- ∃ 2. rewrite ← H. reflexivity.
```

Qed.

(Notice the use of the empty pattern to discharge the last case *en passant*.)

We can also prove more generic, higher-level lemmas about `ln`. Note, in the next, how `ln` starts out applied to a variable and only gets expanded when we do case analysis on this variable:

```
Lemma ln_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
    ln x l →
    ln (f x) (map f l).
```

Proof.

```
  intros A B f l x.
  induction l as [|x' l' IHL'].
  -
    simpl. intros [].
  -
    simpl. intros [H | H].
    + rewrite H. left. reflexivity.
    + right. apply IHL'. apply H.
```

Qed.

This way of defining propositions, though convenient in some cases, also has some drawbacks. In particular, it is subject to Coq’s usual restrictions regarding the definition of recursive functions, e.g., the requirement that they be “obviously terminating.” In the next chapter, we will see how to define propositions *inductively*, a different technique with its own set of strengths and limitations.

**Exercise: 2 stars (In\_map\_iff)** Lemma `ln_map_iff` :

```
  ∀ (A B : Type) (f : A → B) (l : list A) (y : B),
    ln y (map f l) ↔
    ∃ x, f x = y ∧ ln x l.
```

Proof.

*Admitted.*

□

**Exercise: 2 stars (in\_app\_iff)** Lemma `in_app_iff` :  $\forall A\ l\ l'\ (a:A),$

```
  ln a (l++l') ↔ ln a l ∨ ln a l'.
```

Proof.

*Admitted.*

□

**Exercise: 3 stars (All)** Recall that functions returning propositions can be seen as *properties* of their arguments. For instance, if  $P$  has type  $\text{nat} \rightarrow \text{Prop}$ , then  $P\ n$  states that property  $P$  holds of  $n$ .

Drawing inspiration from `ln`, write a recursive function `All` stating that some property  $P$  holds of all elements of a list `l`. To make sure your definition is correct, prove the `All_ln` lemma below. (Of course, your definition should *not* just restate the left-hand side of `All_ln`.)

Fixpoint `All` { $T$ } ( $P : T \rightarrow \text{Prop}$ ) (`l` : `list`  $T$ ) : `Prop` :=  
*admit.*

Lemma `All_ln` :

$\forall T (P : T \rightarrow \text{Prop}) (l : \text{list } T),$   
 $(\forall x, \text{ln } x \ l \rightarrow P \ x) \leftrightarrow$   
`All`  $P \ l$ .

Proof.

*Admitted.*

□

**Exercise: 3 stars (`combine_odd_even`)** Complete the definition of the `combine_odd_even` function below. It takes as arguments two properties of numbers, `Podd` and `Peven`, and it should return a property  $P$  such that  $P \ n$  is equivalent to `Podd`  $n$  when  $n$  is odd and equivalent to `Peven`  $n$  otherwise.

Definition `combine_odd_even` (`Podd` `Peven` : `nat`  $\rightarrow$  `Prop`) : `nat`  $\rightarrow$  `Prop` :=  
*admit.*

To test your definition, prove the following facts:

Theorem `combine_odd_even_intro` :

$\forall (Podd \ Peven : \text{nat} \rightarrow \text{Prop}) (n : \text{nat}),$   
 $(\text{oddb } n = \text{true} \rightarrow Podd \ n) \rightarrow$   
 $(\text{oddb } n = \text{false} \rightarrow Peven \ n) \rightarrow$   
`combine_odd_even` `Podd` `Peven`  $n$ .

Proof.

*Admitted.*

Theorem `combine_odd_even_elim_odd` :

$\forall (Podd \ Peven : \text{nat} \rightarrow \text{Prop}) (n : \text{nat}),$   
`combine_odd_even` `Podd` `Peven`  $n \rightarrow$   
 $\text{oddb } n = \text{true} \rightarrow$   
`Podd`  $n$ .

Proof.

*Admitted.*

Theorem `combine_odd_even_elim_even` :

$\forall (Podd \ Peven : \text{nat} \rightarrow \text{Prop}) (n : \text{nat}),$   
`combine_odd_even` `Podd` `Peven`  $n \rightarrow$   
 $\text{oddb } n = \text{false} \rightarrow$   
`Peven`  $n$ .

Proof.

*Admitted.*

□

## 7.4 Applying Theorems to Arguments

One feature of Coq that distinguishes it from many other proof assistants is that it treats *proofs* as first-class objects.

There is a great deal to be said about this, but it is not necessary to understand it in detail in order to use Coq. This section gives just a taste, while a deeper exploration can be found in the optional chapters `ProofObjects` and `IndPrinciples`.

We have seen that we can use the `Check` command to ask Coq to print the type of an expression. We can also use `Check` to ask what theorem a particular identifier refers to.

`Check plus_comm.`

Coq prints the *statement* of the `plus_comm` theorem in the same way that it prints the *type* of any term that we ask it to `Check`. Why?

The reason is that the identifier `plus_comm` actually refers to a *proof object* – a data structure that represents a logical derivation establishing of the truth of the statement  $\forall n\ m : \text{nat}, n + m = m + n$ . The type of this object *is* the statement of the theorem that it is a proof of.

Intuitively, this makes sense because the statement of a theorem tells us what we can use that theorem for, just as the type of a computational object tells us what we can do with that object – e.g., if we have a term of type `nat → nat → nat`, we can give it two `nats` as arguments and get a `nat` back. Similarly, if we have an object of type `n = m → n + n = m + m` and we provide it an “argument” of type `n = m`, we can derive `n + n = m + m`.

Operationally, this analogy goes even further: by applying a theorem, as if it were a function, to hypotheses with matching types, we can specialize its result without having to resort to intermediate assertions. For example, suppose we wanted to prove the following result:

**Lemma** `plus_comm3` :

$\forall n\ m\ p, n + (m + p) = (p + m) + n.$

It appears at first sight that we ought to be able to prove this by rewriting with `plus_comm` twice to make the two sides match. The problem, however, is that the second `rewrite` will undo the effect of the first.

**Proof.**

```
intros n m p.
rewrite plus_comm.
rewrite plus_comm.
```

One simple way of fixing this problem, using only tools that we already know, is to use `assert` to derive a specialized version of `plus_comm` that can be used to rewrite exactly where we want.

Qed.

Lemma plus\_comm3\_take2 :

Proof.

Qed.

Example lemma\_application\_ex :

Proof.

Qed.

125

at most one type. This difference often leads to slightly different ways of capturing informal mathematical concepts, though these are by and large quite natural and easy to work with. For example, instead of saying that a natural number  $n$  belongs to the set of even numbers, we would say in Coq that **ev**  $n$  holds, where **ev** :  $\text{nat} \rightarrow \text{Prop}$  is a property describing even numbers.

However, there are some cases where translating standard mathematical reasoning into Coq can be either cumbersome or sometimes even impossible, unless we enrich the core logic with additional axioms. We conclude this chapter with a brief discussion of some of the most significant differences between the two worlds.

### 7.5.1 Functional Extensionality

The equality assertions that we have seen so far mostly have concerned elements of inductive types (**nat**, **bool**, etc.). But since Coq’s equality operator is polymorphic, these are not the only possibilities – in particular, we can write propositions claiming that two *functions* are equal to each other:

Example `function_equality_ex` : **plus** 3 = **plus** (**pred** 4).

Proof. `reflexivity`. Qed.

In common mathematical practice, two functions  $f$  and  $g$  are considered equal if they produce the same outputs:  $(\text{forall } x, f\ x = g\ x) \rightarrow f = g$ . This is known as the principle of *functional extensionality*.

Informally speaking, an “extensional property” is one that pertains to an object’s observable behavior. Thus, functional extensionality simply means that a function’s identity is completely determined by what we can observe from it – i.e., in Coq terms, the results we obtain after applying it.

Functional extensionality is not part of Coq’s basic axioms: the only way to show that two functions are equal is by simplification (as we did in the proof of `function_equality_ex`). But we can add it to Coq’s core logic using the `Axiom` command.

`Axiom functional_extensionality` :  $\forall \{X\ Y : \text{Type}\}$   
 $\{f\ g : X \rightarrow Y\},$   
 $(\forall (x : X), f\ x = g\ x) \rightarrow f = g.$

Using `Axiom` has the same effect as stating a theorem and skipping its proof using *Admitted*, but it alerts the reader that this isn’t just something we’re going to come back and fill in later!

We can now invoke functional extensionality in proofs:

Lemma `plus_comm_ext` : **plus** = **fun**  $n\ m \Rightarrow m + n$ .

Proof.

`apply functional_extensionality. intros n.`  
`apply functional_extensionality. intros m.`  
`apply plus_comm.`

Qed.

Naturally, we must be careful when adding new axioms into Coq's logic, as they may render it inconsistent – that is, it may become possible to prove every proposition, including **False**! Unfortunately, there is no simple way of telling whether an axiom is safe: hard work is generally required to establish the consistency of any particular combination of axioms. Fortunately, it is known that adding functional extensionality, in particular, *is* consistent.

Note that it is possible to check whether a particular proof relies on any additional axioms, using the `Print Assumptions` command. For instance, if we run it on `plus_comm_ext`, we see that it uses *functional\_extensionality*:

```
Print Assumptions plus_comm_ext.
```

**Exercise: 5 stars (tr\_rev)** One problem with the definition of the list-reversing function `rev` that we have is that it performs a call to `app` on each step; running `app` takes time asymptotically linear in the size of the list, which means that `rev` has quadratic running time. We can improve this with the following definition:

```
Fixpoint rev_append {X} (l1 l2 : list X) : list X :=
  match l1 with
  | [] => l2
  | x :: l1' => rev_append l1' (x :: l2)
  end.
```

```
Definition tr_rev {X} (l : list X) : list X :=
  rev_append l [].
```

This version is said to be *tail-recursive*, because the recursive call to the function is the last operation that needs to be performed (i.e., we don't have to execute `++` after the recursive call); a decent compiler will generate very efficient code in this case. Prove that both definitions are indeed equivalent.

```
Lemma tr_rev_correct : ∀ X, @tr_rev X = @rev X.
```

*Admitted.*

□

## 7.5.2 Propositions and Booleans

We've seen that Coq has two different ways of encoding logical facts: with *booleans* (of type `bool`), and with *propositions* (of type `Prop`). For instance, to claim that a number `n` is even, we can say either (1) that `evenb n` returns `true` or (2) that there exists some `k` such that `n = double k`. Indeed, these two notions of evenness are equivalent, as can easily be shown with a couple of auxiliary lemmas (one of which is left as an exercise).

We often say that the boolean `evenb n` *reflects* the proposition  $\exists k, n = \text{double } k$ .

```
Theorem evenb_double : ∀ k, evenb (double k) = true.
```

*Proof.*

```
  intros k. induction k as [|k' IHk'].
  - reflexivity.
```

- simpl. apply *IHk*’.

Qed.

**Exercise: 3 stars (evenb\_double\_conv)** Theorem evenb\_double\_conv :  $\forall n,$   
 $\exists k, n = \text{if evenb } n \text{ then double } k$   
 else **S** (double *k*).

Proof.

*Admitted.*

□

Theorem even\_bool\_prop :  $\forall n,$   
 evenb *n* = true  $\leftrightarrow \exists k, n = \text{double } k$ .

Proof.

intros *n*. split.

- intros *H*. destruct (evenb\_double\_conv *n*) as [*k Hk*].  
 rewrite *Hk*. rewrite *H*.  $\exists k$ . reflexivity.

- intros [*k Hk*]. rewrite *Hk*. apply evenb\_double.

Qed.

Similarly, to state that two numbers *n* and *m* are equal, we can say either (1) that beq\_nat *n m* returns true or (2) that *n* = *m*. These two notions are equivalent.

Theorem beq\_nat\_true\_iff :  $\forall n1\ n2 : \text{nat},$   
 beq\_nat *n1 n2* = true  $\leftrightarrow n1 = n2$ .

Proof.

intros *n1 n2*. split.

- apply beq\_nat\_true.

- intros *H*. rewrite *H*. rewrite  $\leftarrow$  beq\_nat\_refl. reflexivity.

Qed.

However, while the boolean and propositional formulations of a claim are equivalent from a purely logical perspective, we have also seen that they need not be equivalent *operationally*. Equality provides an extreme example: knowing that beq\_nat *n m* = true is generally of little help in the middle of a proof involving *n* and *m*; however, if we convert the statement to the equivalent form *n* = *m*, we can rewrite with it.

The case of even numbers is also interesting. Recall that, when proving the backwards direction of even\_bool\_prop (evenb\_double, going from the propositional to the boolean claim), we used a simple induction on *k*. On the other hand, the converse (the evenb\_double\_conv exercise) required a clever generalization, since we can’t directly prove  $(\exists k, n = \text{double } k) \rightarrow \text{evenb } n = \text{true}$ .

For these examples, the propositional claims were more useful than their boolean counterparts, but this is not always the case. For instance, we cannot test whether a general proposition is true or not in a function definition; as a consequence, the following code fragment is rejected:

*Fail* Definition is\_even\_prime *n* :=



```
if n = 2 then true
else false.
```

Coq complains that `n = 2` has type **Prop**, while it expects an element of **bool** (or some other inductive type with two elements). The reason for this error message has to do with the *computational* nature of Coq's core language, which is designed so that every function that it can express is computable and total. One reason for this is to allow the extraction of executable programs from Coq developments. As a consequence, **Prop** in Coq does *not* have a universal case analysis operation telling whether any given proposition is true or false, since such an operation would allow us to write non-computable functions.

Although general non-computable properties cannot be phrased as boolean computations, it is worth noting that even many *computable* properties are easier to express using **Prop** than **bool**, since recursive function definitions are subject to significant restrictions in Coq. For instance, the next chapter shows how to define the property that a regular expression matches a given string using **Prop**. Doing the same with **bool** would amount to writing a regular expression matcher, which would be more complicated, harder to understand, and harder to reason about.

Conversely, an important side benefit of stating facts using booleans is enabling some proof automation through computation with Coq terms, a technique known as *proof by reflection*. Consider the following statement:

Example `even_1000` :  $\exists k, 1000 = \text{double } k$ .

The most direct proof of this fact is to give the value of `k` explicitly.

Proof.  `$\exists 500$ . reflexivity. Qed.`

On the other hand, the proof of the corresponding boolean statement is even simpler:

Example `even_1000'` : `evenb 1000 = true`.

Proof. `reflexivity. Qed.`

What is interesting is that, since the two notions are equivalent, we can use the boolean formulation to prove the other one without mentioning 500 explicitly:

Example `even_1000''` :  $\exists k, 1000 = \text{double } k$ .

Proof. `apply even_bool_prop. reflexivity. Qed.`

Although we haven't gained much in terms of proof size in this case, larger proofs can often be made considerably simpler by the use of reflection. As an extreme example, the Coq proof of the famous *4-color theorem* uses reflection to reduce the analysis of hundreds of different cases to a boolean computation. We won't cover reflection in great detail, but it serves as a good example showing the complementary strengths of booleans and general propositions.

**Exercise: 2 stars (logical\_connectives)** The following lemmas relate the propositional connectives studied in this chapter to the corresponding boolean operations.

Lemma `andb_true_iff` :  $\forall b1\ b2:\text{bool}$ ,

$b1 \ \&\& \ b2 = \text{true} \leftrightarrow b1 = \text{true} \wedge b2 = \text{true}.$

Proof.

*Admitted.*

Lemma `orb_true_iff` :  $\forall b1 \ b2,$

$b1 \ || \ b2 = \text{true} \leftrightarrow b1 = \text{true} \vee b2 = \text{true}.$

Proof.

*Admitted.*

□

**Exercise: 1 star (`beq_nat_false_iff`)** The following theorem is an alternate “negative” formulation of `beq_nat_true_iff` that is more convenient in certain situations (we’ll see examples in later chapters).

Theorem `beq_nat_false_iff` :  $\forall x \ y : \text{nat},$

$\text{beq\_nat } x \ y = \text{false} \leftrightarrow x \neq y.$

Proof.

*Admitted.*

□

**Exercise: 3 stars (`beq_list`)** Given a boolean operator *beq* for testing equality of elements of some type **A**, we can define a function `beq_list beq` for testing equality of lists with elements in **A**. Complete the definition of the `beq_list` function below. To make sure that your definition is correct, prove the lemma `beq_list_true_iff`.

Fixpoint `beq_list` {*A*} (*beq* : *A* → *A* → **bool**)  
                           (*l1 l2* : **list** *A*) : **bool** :=

*admit.*

Lemma `beq_list_true_iff` :

$\forall A \ (beq : A \rightarrow A \rightarrow \text{bool}),$   
 $(\forall a1 \ a2, \text{beq } a1 \ a2 = \text{true} \leftrightarrow a1 = a2) \rightarrow$   
 $\forall l1 \ l2, \text{beq\_list } beq \ l1 \ l2 = \text{true} \leftrightarrow l1 = l2.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, recommended (`All_forallb`)** Recall the function `forallb`, from the exercise *forall\_exists\_challenge* in chapter Tactics:

Fixpoint `forallb` {*X* : Type} (*test* : *X* → **bool**) (*l* : **list** *X*) : **bool** :=  
   match *l* with  
   | [] ⇒ true  
   | *x* :: *l'* ⇒ andb (*test* *x*) (`forallb test l'`)  
   end.

Prove the theorem below, which relates `forallb` to the `All` property of the above exercise.

**Theorem** `forallb_true_iff` :  $\forall X \text{ test } (l : \text{list } X),$   
 $\text{forallb test } l = \text{true} \leftrightarrow \text{All } (\text{fun } x \Rightarrow \text{test } x = \text{true}) \text{ } l.$

**Proof.**

*Admitted.*

Are there any important properties of the function `forallb` which are not captured by your specification?

□

### 7.5.3 Classical vs. Constructive Logic

We have seen that it is not possible to test whether or not a proposition  $P$  holds while defining a Coq function. You may be surprised to learn that a similar restriction applies to *proofs*! In other words, the following intuitive reasoning principle is not derivable in Coq:

**Definition** `excluded_middle` :=  $\forall P : \text{Prop},$   
 $P \vee \neg P.$

To understand operationally why this is the case, recall that, to prove a statement of the form  $P \vee Q$ , we use the `left` and `right` tactics, which effectively require knowing which side of the disjunction holds. However, the universally quantified  $P$  in `excluded_middle` is an *arbitrary* proposition, which we know nothing about. We don't have enough information to choose which of `left` or `right` to apply, just as Coq doesn't have enough information to mechanically decide whether  $P$  holds or not inside a function. On the other hand, if we happen to know that  $P$  is reflected in some boolean term `b`, then knowing whether it holds or not is trivial: we just have to check the value of `b`. This leads to the following theorem:

**Theorem** `restricted_excluded_middle` :  $\forall P \text{ } b,$   
 $(P \leftrightarrow b = \text{true}) \rightarrow P \vee \neg P.$

**Proof.**

`intros P [] H.`

`- left. rewrite H. reflexivity.`

`- right. rewrite H. intros contra. inversion contra.`

**Qed.**

In particular, the excluded middle is valid for equations `n = m`, between natural numbers `n` and `m`.

You may find it strange that the general excluded middle is not available by default in Coq; after all, any given claim must be either true or false. Nonetheless, there is an advantage in not assuming the excluded middle: statements in Coq can make stronger claims than the analogous statements in standard mathematics. Notably, if there is a Coq proof of  $\exists x, P \text{ } x$ , it is possible to explicitly exhibit a value of `x` for which we can prove  $P \text{ } x$  – in other words, every proof of existence is necessarily *constructive*. Because of this, logics like Coq's, which do not assume the excluded middle, are referred to as *constructive logics*. More conventional logical

systems such as ZFC, in which the excluded middle does hold for arbitrary propositions, are referred to as *classical*.

The following example illustrates why assuming the excluded middle may lead to non-constructive proofs:

*Claim:* There exist irrational numbers  $a$  and  $b$  such that  $a \wedge b$  is rational.

*Proof:* It is not difficult to show that  $\sqrt{2}$  is irrational. If  $\sqrt{2} \wedge \sqrt{2}$  is rational, it suffices to take  $a = b = \sqrt{2}$  and we are done. Otherwise,  $\sqrt{2} \wedge \sqrt{2}$  is irrational. In this case, we can take  $a = \sqrt{2} \wedge \sqrt{2}$  and  $b = \sqrt{2}$ , since  $a \wedge b = \sqrt{2} \wedge (\sqrt{2} \times \sqrt{2}) = \sqrt{2} \wedge 2 = 2$ .  $\square$

Do you see what happened here? We used the excluded middle to consider separately the cases where  $\sqrt{2} \wedge \sqrt{2}$  is rational and where it is not, without knowing which one actually holds! Because of that, we wind up knowing that such  $a$  and  $b$  exist but we cannot determine what their actual values are (at least, using this line of argument).

As useful as constructive logic is, it does have its limitations: There are many statements that can easily be proven in classical logic but that have much more complicated constructive proofs, and there are some that are known to have no constructive proof at all! Fortunately, like functional extensionality, the excluded middle is known to be compatible with Coq's logic, allowing us to add it safely as an axiom. However, we will not need to do so in this book: the results that we cover can be developed entirely within constructive logic at negligible extra cost.

It takes some practice to understand which proof techniques must be avoided in constructive reasoning, but arguments by contradiction, in particular, are infamous for leading to non-constructive proofs. Here's a typical example: suppose that we want to show that there exists  $x$  with some property  $P$ , i.e., such that  $P\ x$ . We start by assuming that our conclusion is false; that is,  $\neg \exists x, P\ x$ . From this premise, it is not hard to derive  $\forall x, \neg P\ x$ . If we manage to show that this intermediate fact results in a contradiction, we arrive at an existence proof without ever exhibiting a value of  $x$  for which  $P\ x$  holds!

The technical flaw here, from a constructive standpoint, is that we claimed to prove  $\exists x, P\ x$  using a proof of  $\neg \neg \exists x, P\ x$ . However, allowing ourselves to remove double negations from arbitrary statements is equivalent to assuming the excluded middle, as shown in one of the exercises below. Thus, this line of reasoning cannot be encoded in Coq without assuming additional axioms.

**Exercise: 3 stars (excluded\_middle\_irrefutable)** The consistency of Coq with the general excluded middle axiom requires complicated reasoning that cannot be carried out within Coq itself. However, the following theorem implies that it is always safe to assume a decidability axiom (i.e., an instance of excluded middle) for any *particular* Prop  $P$ . Why? Because we cannot prove the negation of such an axiom; if we could, we would have both  $\neg (P \vee \neg P)$  and  $\neg \neg (P \vee \neg P)$ , a contradiction.

Theorem excluded\_middle\_irrefutable:  $\forall (P:\text{Prop}),$   
 $\neg \neg (P \vee \neg P).$

Proof.

*Admitted.*

□

**Exercise: 3 stars, optional (not\_exists\_dist)** It is a theorem of classical logic that the following two assertions are equivalent:  $\sim (\exists x, \sim P x)$  for all  $x, P x$ . The `dist_not_exists` theorem above proves one side of this equivalence. Interestingly, the other direction cannot be proved in constructive logic. Your job is to show that it is implied by the excluded middle.

Theorem `not_exists_dist` :

`excluded_middle`  $\rightarrow$

$\forall (X:\text{Type}) (P : X \rightarrow \text{Prop}),$

$\neg (\exists x, \neg P x) \rightarrow (\forall x, P x).$

Proof.

*Admitted.*

□

**Exercise: 5 stars, advanced, optional (classical\_axioms)** For those who like a challenge, here is an exercise taken from the Coq'Art book by Bertot and Casteran (p. 123). Each of the following four statements, together with `excluded_middle`, can be considered as characterizing classical logic. We can't prove any of them in Coq, but we can consistently add any one of them as an axiom if we wish to work in classical logic.

Prove that all five propositions (these four plus `excluded_middle`) are equivalent.

Definition `peirce` :=  $\forall P Q:\text{Prop},$

$((P \rightarrow Q) \rightarrow P) \rightarrow P.$

Definition `double_negation_elimination` :=  $\forall P:\text{Prop},$

$\neg\neg P \rightarrow P.$

Definition `de_morgan_not_and_not` :=  $\forall P Q:\text{Prop},$

$\neg(\neg P \wedge \neg Q) \rightarrow P \vee Q.$

Definition `implies_to_or` :=  $\forall P Q:\text{Prop},$

$(P \rightarrow Q) \rightarrow (\neg P \vee Q).$

□

*Date* : 2015 – 08 – 11 12 : 03 : 04 – 0400(*Tue, 11 Aug 2015*)

# Chapter 8

## Library IndProp

### 8.1 IndProp: Inductively Defined Propositions

Require Export Logic.

### 8.2 Inductively Defined Propositions

In the `Logic` chapter we looked at several ways of writing propositions, including conjunction, disjunction, and quantifiers. In this chapter, we bring a new tool into the mix: *inductive definitions*.

Recall that we have seen two ways of stating that a number `n` is even: We can say (1) `evenb n = true`, or (2)  $\exists k, n = \text{double } k$ . Yet another possibility is to say that `n` is even if we can establish its evenness from the following rules:

- Rule `ev_0`: The number 0 is even.
- Rule `ev_SS`: If `n` is even, then `S (S n)` is even.

To illustrate how this new definition of evenness works, let's use its rules to show that 4 is even. By rule `ev_SS`, it suffices to show that 2 is even. This, in turn, is again guaranteed by rule `ev_SS`, as long as we can show that 0 is even. But this last fact follows directly from the `ev_0` rule.

We will see many definitions like this one during the rest of the course. For purposes of informal discussions, it is helpful to have a lightweight notation that makes them easy to read and write. *Inference rules* are one such notation:

---

$$\frac{(\text{ev\_0}) \text{ ev } 0}{\text{ev } n}$$

---

$$(\text{ev\_SS}) \text{ ev } (S (S n))$$

Each of the textual rules above is reformatted here as an inference rule; the intended reading is that, if the *premises* above the line all hold, then the *conclusion* below the line follows. For example, the rule `ev_SS` says that, if `n` satisfies `ev`, then `S (S n)` also does. If a rule has no premises above the line, then its conclusion holds unconditionally.

We can represent a proof using these rules by combining rule applications into a *proof tree*. Here’s how we might transcribe the above proof that 4 is even:

---

(ev\_0) ev 0

---

(ev\_SS) ev 2

---

(ev\_SS) ev 4

Why call this a “tree” (rather than a “stack”, for example)? Because, in general, inference rules can have multiple premises. We will see examples of this below.

Putting all of this together, we can translate the definition of evenness into a formal Coq definition using an `Inductive` declaration, where each constructor corresponds to an inference rule:

```
Inductive ev : nat → Prop :=
| ev_0 : ev 0
| ev_SS : ∀ n : nat, ev n → ev (S (S n)).
```

This definition is different in one crucial respect from previous uses of `Inductive`: its result is not a `Type`, but rather a function from `nat` to `Prop` – that is, a property of numbers. Note that we’ve already seen other inductive definitions that result in functions, such as `list`, whose type is `Type → Type`. What is new here is that, because the `nat` argument of `ev` appears *unnamed*, to the *right* of the colon, it is allowed to take different values in the types of different constructors: 0 in the type of `ev_0` and `S (S n)` in the type of `ev_SS`.

In contrast, the definition of `list` names the `X` parameter *globally*, to the *left* of the colon, forcing the result of `nil` and `cons` to be the same (`list X`). Had we tried to bring `nat` to the left in defining `ev`, we would have seen an error:

```
Fail Inductive wrong_ev (n : nat) : Prop :=
| wrong_ev_0 : wrong_ev 0
| wrong_ev_SS : ∀ n, wrong_ev n → wrong_ev (S (S n)).
```

(“Parameter” here is Coq jargon for an argument on the left of the colon in an `Inductive` definition; “index” is used to refer to arguments on the right of the colon.)

We can think of the definition of `ev` as defining a Coq property `ev : nat → Prop`, together with theorems `ev_0 : ev 0` and `ev_SS : ∀ n, ev n → ev (S (S n))`. Such “constructor theorems” have the same status as proven theorems. In particular, we can use Coq’s `apply` tactic with the rule names to prove `ev` for particular numbers...

Theorem `ev_4` : `ev 4`.

Proof. `apply ev_SS. apply ev_SS. apply ev_0. Qed.`

... or we can use function application syntax:

Theorem `ev_4'` : `ev 4`.

Proof. apply (ev\_SS 2 (ev\_SS 0 ev\_0)). Qed.

We can also prove theorems that have hypotheses involving **ev**.

Theorem ev\_plus4 :  $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (4 + n)$ .

Proof.

intros n. simpl. intros Hn.

apply ev\_SS. apply ev\_SS. apply Hn.

Qed.

More generally, we can show that any number multiplied by 2 is even:

**Exercise: 1 star (ev\_double)** Theorem ev\_double :  $\forall n, \mathbf{ev} \ (\text{double } n)$ .

Proof.

*Admitted.*

□

## 8.3 Using Evidence in Proofs

Besides *constructing* evidence that numbers are even, we can also *reason about* such evidence.

Introducing **ev** with an **Inductive** declaration tells Coq not only that the constructors **ev\_0** and **ev\_SS** are valid ways to build evidence that some number is even, but also that these two constructors are the *only* ways to build evidence that numbers are even (in the sense of **ev**). In other words, if someone gives us evidence *E* for the assertion **ev** *n*, then we know that *E* must have one of two shapes:

- *E* is **ev\_0** (and *n* is 0), or
- *E* is **ev\_SS** *n'* *E'* (and *n* is S (S *n'*), where *E'* is evidence for **ev** *n'*).

This suggests that it should be possible to analyze a hypothesis of the form **ev** *n* much as we do inductively defined data structures; in particular, it should be possible to argue by *induction* and *case analysis* on such evidence. Let's look at a few examples to see what this means in practice.

### 8.3.1 Inversion on Evidence

Subtracting two from an even number yields another even number. We can easily prove this claim with the techniques that we've already seen, provided that we phrase it in the right way. If we state it in terms of **evenb**, for instance, we can proceed by a simple case analysis on *n*:

Theorem evenb\_minus2:  $\forall n,$

$\mathbf{evenb} \ n = \mathbf{true} \rightarrow \mathbf{evenb} \ (\text{pred} \ (\text{pred} \ n)) = \mathbf{true}.$



Proof.

```
intros [| [| n' ] ].
- reflexivity.
- intros H. inversion H.
- simpl. intros H. apply H.
```

Qed.

We can state the same claim in terms of **ev**, but this quickly leads us to an obstacle: Since **ev** is defined inductively – rather than as a function – Coq doesn’t know how to simplify a goal involving **ev** *n* after case analysis on *n*. As a consequence, the same proof strategy fails:

Theorem ev\_minus2:  $\forall n,$   
 $\mathbf{ev} \ n \rightarrow \mathbf{ev} \ (\mathbf{pred} \ (\mathbf{pred} \ n)).$

Proof.

```
intros [| [| n' ] ].
- simpl. intros _. apply ev_0.
- simpl.
```

Abort.

The solution is to perform case analysis on the evidence that **ev** *n* *directly*. By the definition of **ev**, there are two cases to consider:

- If that evidence is of the form **ev\_0**, we know that *n* = 0. Therefore, it suffices to show that **ev** (**pred** (**pred** 0)) holds. By the definition of **pred**, this is equivalent to showing that **ev** 0 holds, which directly follows from **ev\_0**.
- Otherwise, that evidence must have the form **ev\_SS** *n'* *E'*, where *n* = **S** (**S** *n'*) and *E'* is evidence for **ev** *n'*. We must then show that **ev** (**pred** (**pred** (**S** (**S** *n'*)))) holds, which, after simplification, follows directly from *E'*.

We can invoke this kind of argument in Coq using the **inversion** tactic. Besides allowing us to reason about equalities involving constructors, **inversion** provides a case-analysis principle for inductively defined propositions. When used in this way, its syntax is similar to **destruct**: We pass it a list of identifiers separated by | characters to name the arguments to each of the possible constructors. For instance:

Theorem ev\_minus2 :  $\forall n,$   
 $\mathbf{ev} \ n \rightarrow \mathbf{ev} \ (\mathbf{pred} \ (\mathbf{pred} \ n)).$

Proof.

```
intros n E.
inversion E as [| n' E'].
- simpl. apply ev_0.
- simpl. apply E'. Qed.
```

Note that, in this particular case, it is also possible to replace **inversion** by **destruct**:

Theorem ev\_minus2' :  $\forall n,$

`ev n → ev (pred (pred n)).`

Proof.

```
intros n E.
destruct E as [| n' E'].
- simpl. apply ev_0.
- simpl. apply E'. Qed.
```

The difference between the two forms is that `inversion` is more convenient when used on a hypothesis that consists of an inductive property applied to a complex expression (as opposed to a single variable). Here’s is a concrete example. Suppose that we wanted to prove the following variation of `ev_minus2`:

Theorem `evSS_ev` :  $\forall n,$   
`ev (S (S n)) → ev n.`

Intuitively, we know that evidence for the hypothesis cannot consist just of the `ev_0` constructor, since `O` and `S` are different constructors of the type `nat`; hence, `ev_SS` is the only case that applies. Unfortunately, `destruct` is not smart enough to realize this, and it still generates two subgoals. Even worse, in doing so, it keeps the final goal unchanged, failing to provide any useful information for completing the proof.

Proof.

```
intros n E.
destruct E as [| n' E'].
-
```

Abort.

What happened, exactly? Calling `destruct` has the effect of replacing all occurrences of the property argument by the values that correspond to each constructor. This is enough in the case of `ev_minus2'` because that argument, `n`, is mentioned directly in the final goal. However, it doesn’t help in the case of `evSS_ev` since the term that gets replaced (`S (S n)`) is not mentioned anywhere.

The `inversion` tactic, on the other hand, can detect (1) that the first case does not apply, and (2) that the `n'` that appears on the `ev_SS` case must be the same as `n`. This allows us to complete the proof:

Theorem `evSS_ev` :  $\forall n,$   
`ev (S (S n)) → ev n.`

Proof.

```
intros n E.
inversion E as [| n' E'].
apply E'.
```

Qed.

By using `inversion`, we can also apply the principle of explosion to “obviously contradictory” hypotheses involving inductive properties. For example:

Theorem one\_not\_even :  $\neg$  **ev** 1.

Proof.

intros *H*. inversion *H*. Qed.

**Exercise: 1 star (inversion\_practice)** Prove the following results using **inversion**.

Theorem SSSSev\_\_even :  $\forall$  *n*,

**ev** (**S** (**S** (**S** (**S** *n*))))  $\rightarrow$  **ev** *n*.

Proof.

*Admitted.*

Theorem even5\_nonsense :

**ev** 5  $\rightarrow$  2 + 2 = 9.

Proof.

*Admitted.*

□

The way we've used **inversion** here may seem a bit mysterious at first. Until now, we've only used **inversion** on equality propositions, to utilize injectivity of constructors or to discriminate between different constructors. But we see here that **inversion** can also be applied to analyzing evidence for inductively defined propositions.

Here's how **inversion** works in general. Suppose the name *l* refers to an assumption *P* in the current context, where *P* has been defined by an **Inductive** declaration. Then, for each of the constructors of *P*, **inversion** *l* generates a subgoal in which *l* has been replaced by the exact, specific conditions under which this constructor could have been used to prove *P*. Some of these subgoals will be self-contradictory; **inversion** throws these away. The ones that are left represent the cases that must be proved to establish the original goal. For those, **inversion** adds all equations into the proof context that must hold of the arguments given to *P* (e.g., **S** (**S** *n'*) = *n* in the proof of **evSS\_ev**).

### 8.3.2 Induction on Evidence

The **ev\_double** exercise above shows that our new notion of evenness is implied by the two earlier ones (since, by **even\_bool\_prop**, we already know that those are equivalent to each other). To show that all three coincide, we just need the following lemma:

Lemma **ev\_even** :  $\forall$  *n*,

**ev** *n*  $\rightarrow \exists$  *k*, *n* = **double** *k*.

Proof.

We could try to proceed by case analysis or induction on *n*. But since **ev** is mentioned in a premise, this strategy would probably lead to a dead end, as in the previous section. Thus, it seems better to first try inversion on the evidence for **ev**. Indeed, the first case can be solved trivially.

intros *n* *E*. inversion *E* as [| *n'* *E'*].

-

```

    ∃ 0. reflexivity.
- simpl.

```

Unfortunately, the second case is harder. We need to show  $\exists k, S (S n') = \text{double } k$ , but the only available assumption is  $E'$ , which states that **ev**  $n'$  holds. Since this isn't directly useful, it seems that we are stuck and that performing case analysis on  $E$  was a waste of time.

If we look more closely at our second goal, however, we can see that something interesting happened: By performing case analysis on  $E$ , we were able to reduce the original result to an similar one that involves a *different* piece of evidence for **ev**:  $E'$ . More formally, we can finish our proof by showing that exists  $k'$ ,  $n' = \text{double } k'$ , which is the same as the original statement, but with  $n'$  instead of  $n$ . Indeed, it is not difficult to convince Coq that this intermediate result suffices.

```

assert (I : (∃ k', n' = double k') →
          (∃ k, S (S n') = double k)).
{ intros [k' Hk']. rewrite Hk'. ∃ (S k').
  reflexivity. }
apply I.

```

If this looks familiar, it is no coincidence: We've encountered similar problems in the **Induction** chapter, when trying to use case analysis to prove results that required induction. And once again the solution is... induction!

The behavior of **induction** on evidence is the same as its behavior on data: It causes Coq to generate one subgoal for each constructor that could have used to build that evidence, while providing an induction hypotheses for each recursive occurrence of the property in question.

Let's try our current lemma again:

Abort.

```

Lemma ev_even : ∀ n,
  ev n → ∃ k, n = double k.

```

Proof.

```

  intros n E.
  induction E as [|n' E' IH].
-
  ∃ 0. reflexivity.
-
  destruct IH as [k' Hk'].
  rewrite Hk'. ∃ (S k'). reflexivity.

```

Qed.

Here, we can see that Coq produced an  $IH$  that corresponds to  $E'$ , the single recursive occurrence of **ev** in its own definition. Since  $E'$  mentions  $n'$ , the induction hypothesis talks about  $n'$ , as opposed to  $n$  or some other number.

The equivalence between the second and third definitions of evenness now follows.

Theorem `ev_even_iff` :  $\forall n,$   
 $\mathbf{ev} \ n \leftrightarrow \exists k, n = \text{double } k.$

Proof.

```
intros n. split.
- apply ev_even.
- intros [k Hk]. rewrite Hk. apply ev_double.
```

Qed.

As we will see in later chapters, induction on evidence is a recurring technique when studying the semantics of programming languages, where many properties of interest are defined inductively. The following exercises provide simple examples of this technique, to help you familiarize yourself with it.

**Exercise: 2 stars (`ev_sum`)** Theorem `ev_sum` :  $\forall n \ m, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ m \rightarrow \mathbf{ev} \ (n + m).$

Proof.

*Admitted.*

□

**Exercise: 4 stars, advanced (`ev_alternate`)** In general, there may be multiple ways of defining a property inductively. For example, here's a (slightly contrived) alternative definition for `ev`:

```
Inductive ev' : nat → Prop :=
| ev'_0 : ev' 0
| ev'_2 : ev' 2
| ev'_sum :  $\forall n \ m, \mathbf{ev}' \ n \rightarrow \mathbf{ev}' \ m \rightarrow \mathbf{ev}' \ (n + m).$ 
```

Prove that this definition is logically equivalent to the old one.

Theorem `ev'_ev` :  $\forall n, \mathbf{ev}' \ n \leftrightarrow \mathbf{ev} \ n.$

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced, recommended (`ev_ev_ev`)** Finding the appropriate thing to do induction on is a bit tricky here:

Theorem `ev_ev_ev` :  $\forall n \ m,$   
 $\mathbf{ev} \ (n+m) \rightarrow \mathbf{ev} \ n \rightarrow \mathbf{ev} \ m.$

Proof.

*Admitted.*

□

**Exercise: 3 stars, optional (`ev_plus_plus`)** This exercise just requires applying existing lemmas. No induction or even case analysis is needed, though some of the rewriting may be tedious.

Theorem `ev_plus_plus` :  $\forall n\ m\ p,$   
 $\mathbf{ev}\ (n+m) \rightarrow \mathbf{ev}\ (n+p) \rightarrow \mathbf{ev}\ (m+p).$

Proof.

*Admitted.*

□

## 8.4 Inductive Relations

A proposition parameterized by a number (such as `ev`) can be thought of as a *property* – i.e., it defines a subset of `nat`, namely those numbers for which the proposition is provable. In the same way, a two-argument proposition can be thought of as a *relation* – i.e., it defines a set of pairs for which the proposition is provable.

Module `LEMODULE`.

One useful example is the “less than or equal to” relation on numbers.

The following definition should be fairly intuitive. It says that there are two ways to give evidence that one number is less than or equal to another: either observe that they are the same number, or give evidence that the first is less than or equal to the predecessor of the second.

Inductive `le` : `nat`  $\rightarrow$  `nat`  $\rightarrow$  `Prop` :=  
 | `le_n` :  $\forall n, \mathbf{le}\ n\ n$   
 | `le_S` :  $\forall n\ m, (\mathbf{le}\ n\ m) \rightarrow (\mathbf{le}\ n\ (\mathbf{S}\ m)).$

Notation "`m <= n`" :=  $(\mathbf{le}\ m\ n).$

Proofs of facts about  $\leq$  using the constructors `le_n` and `le_S` follow the same patterns as proofs about properties, like `ev` above. We can **apply** the constructors to prove  $\leq$  goals (e.g., to show that  $3 \leq 3$  or  $3 \leq 6$ ), and we can use tactics like `inversion` to extract information from  $\leq$  hypotheses in the context (e.g., to prove that  $(2 \leq 1) \rightarrow 2+2=5$ .)

Here are some sanity checks on the definition. (Notice that, although these are the same kind of simple “unit tests” as we gave for the testing functions we wrote in the first few lectures, we must construct their proofs explicitly – `simpl` and `reflexivity` don’t do the job, because the proofs aren’t just a matter of simplifying computations.)

Theorem `test_le1` :

$3 \leq 3.$

Proof.

`apply le_n. Qed.`

Theorem `test_le2` :

$3 \leq 6.$

Proof.

`apply le_S. apply le_S. apply le_S. apply le_n. Qed.`

Theorem `test_le3` :

$(2 \leq 1) \rightarrow 2 + 2 = 5.$

Proof.

intros *H*. inversion *H*. inversion *H2*. Qed.

The “strictly less than” relation  $n < m$  can now be defined in terms of **le**.

End LEMODULE.

Definition lt (*n m*:nat) := le (S *n*) *m*.

Notation "m < n" := (lt *m n*).

Here are a few more simple relations on numbers:

Inductive square\_of : nat → nat → Prop :=

sq : ∀ *n*:nat, square\_of *n* (*n* × *n*).

Inductive next\_nat : nat → nat → Prop :=

| nn : ∀ *n*:nat, next\_nat *n* (S *n*).

Inductive next\_even : nat → nat → Prop :=

| ne\_1 : ∀ *n*, ev (S *n*) → next\_even *n* (S *n*)

| ne\_2 : ∀ *n*, ev (S (S *n*)) → next\_even *n* (S (S *n*)).

**Exercise: 2 stars, recommended (total\_relation)** Define an inductive binary relation *total\_relation* that holds between every pair of natural numbers.

□

**Exercise: 2 stars (empty\_relation)** Define an inductive binary relation *empty\_relation* (on numbers) that never holds.

□

**Exercise: 3 stars, optional (le\_exercises)** Here are a number of facts about the  $\leq$  and  $<$  relations that we are going to need later in the course. The proofs make good practice exercises.

Lemma le\_trans : ∀ *m n o*,  $m \leq n \rightarrow n \leq o \rightarrow m \leq o$ .

Proof.

*Admitted.*

Theorem O\_le\_n : ∀ *n*,

$0 \leq n$ .

Proof.

*Admitted.*

Theorem n\_le\_m\_\_Sn\_le\_Sm : ∀ *n m*,

$n \leq m \rightarrow S\ n \leq S\ m$ .

Proof.

*Admitted.*

Theorem Sn\_le\_Sm\_\_n\_le\_m : ∀ *n m*,

$\text{S } n \leq \text{S } m \rightarrow n \leq m.$

Proof.

*Admitted.*

Theorem `le_plus_l` :  $\forall a b,$

$a \leq a + b.$

Proof.

*Admitted.*

Theorem `plus_lt` :  $\forall n1\ n2\ m,$

$n1 + n2 < m \rightarrow$

$n1 < m \wedge n2 < m.$

Proof.

`unfold lt.`

*Admitted.*

Theorem `lt_S` :  $\forall n\ m,$

$n < m \rightarrow$

$n < \text{S } m.$

Proof.

*Admitted.*

Theorem `leb_complete` :  $\forall n\ m,$

$\text{leb } n\ m = \text{true} \rightarrow n \leq m.$

Proof.

*Admitted.*

Hint: The next one may be easiest to prove by induction on `m`.

Theorem `leb_correct` :  $\forall n\ m,$

$n \leq m \rightarrow$

$\text{leb } n\ m = \text{true}.$

Proof.

*Admitted.*

Hint: This theorem can easily be proved without using `induction`.

Theorem `leb_true_trans` :  $\forall n\ m\ o,$

$\text{leb } n\ m = \text{true} \rightarrow \text{leb } m\ o = \text{true} \rightarrow \text{leb } n\ o = \text{true}.$

Proof.

*Admitted.*

**Exercise: 2 stars, optional (`leb_iff`)** Theorem `leb_iff` :  $\forall n\ m,$

$\text{leb } n\ m = \text{true} \leftrightarrow n \leq m.$

Proof.

*Admitted.*

□

Module `R`.



**Exercise: 3 stars, recommended (R\_provability2)** We can define three-place relations, four-place relations, etc., in just the same way as binary relations. For example, consider the following three-place relation on numbers:

```
Inductive R : nat → nat → nat → Prop :=
| c1 : R 0 0 0
| c2 : ∀ m n o, R m n o → R (S m) n (S o)
| c3 : ∀ m n o, R m n o → R m (S n) (S o)
| c4 : ∀ m n o, R (S m) (S n) (S (S o)) → R m n o
| c5 : ∀ m n o, R m n o → R n m o.
```

- Which of the following propositions are provable?
  - R 1 1 2
  - R 2 2 6
- If we dropped constructor c5 from the definition of R, would the set of provable propositions change? Briefly (1 sentence) explain your answer.
- If we dropped constructor c4 from the definition of R, would the set of provable propositions change? Briefly (1 sentence) explain your answer.

□

**Exercise: 3 stars, optional (R\_fact)** The relation R above actually encodes a familiar function. Figure out which function; then state and prove this equivalence in Coq?

Definition fR : nat → nat → nat :=

*admit.*

Theorem R\_equiv\_fR : ∀ m n o, R m n o ↔ fR m n = o.

Proof.

*Admitted.*

□

End R.

**Exercise: 4 stars, advanced (subsequence)** A list is a *subsequence* of another list if all of the elements in the first list occur in the same order in the second list, possibly with some extra elements in between. For example, 1;2;3 is a subsequence of each of the lists 1;2;3 1;1;1;2;2;3 1;2;7;3 5;6;1;9;9;2;7;3;8 but it is *not* a subsequence of any of the lists 1;2 1;3 5;6;2;1;7;3;8

- Define an inductive proposition *subseq* on **list nat** that captures what it means to be a subsequence. (Hint: You'll need three cases.)

- Prove *subseq\_refl* that subsequence is reflexive, that is, any list is a subsequence of itself.
- Prove *subseq\_app* that for any lists *l1*, *l2*, and *l3*, if *l1* is a subsequence of *l2*, then *l1* is also a subsequence of *l2* ++ *l3*.
- (Optional, harder) Prove *subseq\_trans* that subsequence is transitive – that is, if *l1* is a subsequence of *l2* and *l2* is a subsequence of *l3*, then *l1* is a subsequence of *l3*. Hint: choose your induction carefully!

□

**Exercise: 2 stars, optional (R\_provability)** Suppose we give Coq the following definition: Inductive *R* : nat -> list nat -> Prop := | c1 : *R* 0 [] | c2 : forall n l, *R* n l -> *R* (S n) (n :: l) | c3 : forall n l, *R* (S n) l -> *R* n l. Which of the following propositions are provable?

- *R* 2 [1;0]
- *R* 1 [1;2;1;0]
- *R* 6 [3;2;1;0]

□

## 8.5 Case Study: Regular Expressions

The **ev** property provides a simple example for illustrating inductive definitions and the basic techniques for reasoning about them, but it is not terribly exciting – after all, it is equivalent to the two non-inductive of evenness that we had already seen, and does not seem to offer any concrete benefit over them. To give a better sense of the power of inductive definitions, we now show how to use them to model a classic concept in computer science: *regular expressions*.

Regular expressions are a simple language for describing strings, defined as elements of the following inductive type. (The names of the constructors should become clear once we explain their meaning below.)

```
Inductive reg_exp (T : Type) : Type :=
| EmptySet : reg_exp T
| EmptyStr : reg_exp T
| Char : T -> reg_exp T
| App : reg_exp T -> reg_exp T -> reg_exp T
| Union : reg_exp T -> reg_exp T -> reg_exp T
| Star : reg_exp T -> reg_exp T.
```

*Arguments* EmptySet {T}.

*Arguments* EmptyStr { $T$ }.  
*Arguments* Char { $T$ } ..  
*Arguments* App { $T$ } - ..  
*Arguments* Union { $T$ } - ..  
*Arguments* Star { $T$ } ..

Note that this definition is *polymorphic*: Regular expressions in **reg\_exp**  $T$  describe strings with characters drawn from  $T$  – that is, lists of elements of  $T$ . (We depart slightly from standard practice in that we do not require the type  $T$  to be finite. This results in a somewhat different theory of regular expressions, but the difference is not significant for our purposes.)

We connect regular expressions and strings via the following rules, which define when a regular expression *matches* some string:

- The expression EmptySet does not match any string.
- The expression EmptyStr matches the empty string [].
- The expression Char  $x$  matches the one-character string [x].
- If  $re1$  matches  $s1$ , and  $re2$  matches  $s2$ , then **App**  $re1$   $re2$  matches  $s1 ++ s2$ .
- If at least one of  $re1$  and  $re2$  matches  $s$ , then **Union**  $re1$   $re2$  matches  $s$ .
- Finally, if we can write some string  $s$  as the concatenation of a sequence of strings  $s = s_1 ++ \dots ++ s_k$ , and the expression  $re$  matches each one of the strings  $s_i$ , then **Star**  $re$  matches  $s$ . (As a special case, the sequence of strings may be empty, so **Star**  $re$  always matches the empty string [] no matter what  $re$  is.)

We can easily translate this informal definition into an **Inductive** one as follows:

```

Inductive exp_match {T} : list T → reg_exp T → Prop :=
| MEmpty : exp_match [] EmptyStr
| MChar : ∀ x, exp_match [x] (Char x)
| MApp : ∀ s1 re1 s2 re2,
    exp_match s1 re1 →
    exp_match s2 re2 →
    exp_match (s1 ++ s2) (App re1 re2)
| MUnionL : ∀ s1 re1 re2,
    exp_match s1 re1 →
    exp_match s1 (Union re1 re2)
| MUnionR : ∀ re1 s2 re2,
    exp_match s2 re2 →
    exp_match s2 (Union re1 re2)
| MStar0 : ∀ re, exp_match [] (Star re)
| MStarApp : ∀ s1 s2 re,

```

```

exp_match s1 re →
exp_match s2 (Star re) →
exp_match (s1 ++ s2) (Star re).

```

Once again, for readability, we can also display this definition using inference-rule notation. At the same time, let's introduce a more readable infix notation.

Notation "s =~ re" := (**exp\_match** s re) (at level 80).

---

(MEmpty) □ =~ EmptyStr

---

(MChar) x =~ Char x  
s1 =~ re1 s2 =~ re2

---

(MApp) s1 ++ s2 =~ App re1 re2  
s1 =~ re1

---

(MUnionL) s1 =~ Union re1 re2  
s2 =~ re2

---

(MUnionR) s2 =~ Union re1 re2

---

(MStar0) □ =~ Star re  
s1 =~ re s2 =~ Star re

---

(MStarApp) s1 ++ s2 =~ Star re

---

Notice that these rules are not *quite* the same as the informal ones that we gave at the beginning of the section. First, we don't need to include a rule explicitly stating that no string matches **EmptySet**; we just don't happen to include any rule that would have the effect of some string matching **EmptySet**. (Indeed, the syntax of inductive definitions doesn't even *allow* us to give such a "negative rule.")

Furthermore, the informal rules for **Union** and **Star** correspond to two constructors each: **MUnionL** / **MUnionR**, and **MStar0** / **MStarApp**. The result is logically equivalent to the original rules, but more convenient to use in Coq, since the recursive occurrences of **exp\_match** are given as direct arguments to the constructors, making it easier to perform induction on evidence. (The *exp\_match\_ex1* and *exp\_match\_ex2* exercises below ask you to prove that the constructors given in the inductive declaration and the ones that would arise from a more literal transcription of the informal rules are indeed equivalent.)

Let's illustrate these rules with a few examples.

Example `reg_exp_ex1` : [1] =~ Char 1.

Proof.

    apply MChar.

Qed.

Example `reg_exp_ex2` : `[1; 2] =~ App (Char 1) (Char 2)`.

Proof.

```
  apply (MApp [1] - [2]).
- apply MChar.
- apply MChar.
```

Qed.

(Notice how the last example applies `MApp` to the strings `[1]` and `[2]` directly. Since the goal mentions `[1; 2]` instead of `[1] ++ [2]`, Coq wouldn't be able to figure out how to split the string on its own.)

Using `inversion`, we can also show that certain strings do *not* match a regular expression:

Example `reg_exp_ex3` : `¬ ([1; 2] =~ Char 1)`.

Proof.

```
  intros H. inversion H.
```

Qed.

We can define helper functions to help write down regular expressions. The `reg_exp_of_list` function constructs a regular expression that matches exactly the list that it receives as an argument:

```
Fixpoint reg_exp_of_list {T} (l : list T) :=
  match l with
  | [] => EmptyStr
  | x :: l' => App (Char x) (reg_exp_of_list l')
  end.
```

Example `reg_exp_ex4` : `[1; 2; 3] =~ reg_exp_of_list [1; 2; 3]`.

Proof.

```
  simpl. apply (MApp [1]).
  { apply MChar. }
  apply (MApp [2]).
  { apply MChar. }
  apply (MApp [3]).
  { apply MChar. }
  apply MEmpty.
```

Qed.

We can also prove general facts about `exp_match`. For instance, the following lemma shows that every string `s` that matches `re` also matches `Star re`.

Lemma `MStar1` :

```
  ∀ T s (re : reg_exp T) ,
    s =~ re →
    s =~ Star re.
```

Proof.

```
  intros T s re H.
```

```

rewrite ← (app_nil_r _ s).
apply (MStarApp s [] re).
- apply H.
- apply MStar0.

```

Qed.

(Note the use of `app_nil_r` to change the goal of the theorem to exactly the same shape expected by `MStarApp`.)

**Exercise: 3 stars (exp\_match\_ex1)** The following lemmas show that the informal matching rules given at the beginning of the chapter can be obtained from the formal inductive definition.

Lemma empty\_is\_empty :  $\forall T (s : \text{list } T),$   
 $\neg (s =^{\sim} \text{EmptySet}).$

Proof.

*Admitted.*

Lemma MUnion' :  $\forall T (s : \text{list } T) (re1\ re2 : \text{reg\_exp } T),$   
 $s =^{\sim} re1 \vee s =^{\sim} re2 \rightarrow$   
 $s =^{\sim} \text{Union } re1\ re2.$

Proof.

*Admitted.*

The next lemma is stated in terms of the `fold` function from the `Poly` chapter: If  $ss : \text{list } (\text{list } T)$  represents a sequence of strings  $s_1, \dots, s_n$ , then `fold app ss []` is the result of concatenating them all together.

Lemma MStar' :  $\forall T (ss : \text{list } (\text{list } T)) (re : \text{reg\_exp } T),$   
 $(\forall s, \text{In } s\ ss \rightarrow s =^{\sim} re) \rightarrow$   
 $\text{fold app } ss\ [] =^{\sim} \text{Star } re.$

Proof.

*Admitted.*

□

**Exercise: 4 stars (reg\_exp\_of\_list)** Prove that `reg_exp_of_list` satisfies the following specification:

Lemma reg\_exp\_of\_list\_spec :  $\forall T (s1\ s2 : \text{list } T),$   
 $s1 =^{\sim} \text{reg\_exp\_of\_list } s2 \leftrightarrow s1 = s2.$

Proof.

*Admitted.*

□

Since the definition of `exp_match` has a recursive structure, we might expect that proofs involving regular expressions will often require induction on evidence. For example, suppose that we wanted to prove the following intuitive result: If a regular expression  $re$  matches

some string  $s$ , then all elements of  $s$  must occur somewhere in  $re$ . To state this theorem, we first define a function `re_chars` that lists all characters that occur in a regular expression:

```
Fixpoint re_chars {T} (re : reg_exp T) : list T :=
  match re with
  | EmptySet => []
  | EmptyStr => []
  | Char x => [x]
  | App re1 re2 => re_chars re1 ++ re_chars re2
  | Union re1 re2 => re_chars re1 ++ re_chars re2
  | Star re => re_chars re
end.
```

We can then phrase our theorem as follows:

```
Theorem in_re_match : ∀ T (s : list T) (re : reg_exp T) (x : T),
  s =~ re →
  In x s →
  In x (re_chars re).
```

Proof.

```
intros T s re x Hmatch Hin.
induction Hmatch
as [
  | x'
  | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
  | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
  | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
-
  apply Hin.
-
  apply Hin.
- simpl. rewrite in_app_iff in *.
  destruct Hin as [Hin | Hin].
  +
    left. apply (IH1 Hin).
  +
    right. apply (IH2 Hin).
-
  simpl. rewrite in_app_iff.
  left. apply (IH Hin).
-
  simpl. rewrite in_app_iff.
  right. apply (IH Hin).
-
  destruct Hin.
```

Something interesting happens in the `MStarApp` case. We obtain *two* induction hypotheses: One that applies when `x` occurs in `s1` (which matches `re`), and a second one that applies when `x` occurs in `s2` (which matches `Star re`). This is a good illustration of why we need induction on evidence for `exp_match`, as opposed to `re`: The latter would only provide an induction hypothesis for strings that match `re`, which would not allow us to reason about the case `ln x s2`.

```
-
  simpl. rewrite in_app_iff in Hin.
  destruct Hin as [Hin | Hin].
  +
    apply (IH1 Hin).
  +
    apply (IH2 Hin).
Qed.
```

**Exercise: 4 stars (re\_not\_empty)** Write a recursive function `re_not_empty` that tests whether a regular expression matches some string. Prove that your function is correct.

```
Fixpoint re_not_empty {T} (re : reg_exp T) : bool :=
  admit.
```

```
Lemma re_not_empty_correct : ∀ T (re : reg_exp T),
  (∃ s, s =~ re) ↔ re_not_empty re = true.
```

Proof.

*Admitted.*

□

### 8.5.1 The *remember* Tactic

One potentially confusing feature of the `induction` tactic is that it happily lets you try to set up an induction over a term that isn't sufficiently general. The net effect of this will be to lose information (much as `destruct` can do), and leave you unable to complete the proof. Here's an example:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
```

Proof.

```
  intros T s1 s2 re H1.
```

Just doing an `inversion` on `H1` won't get us very far in the recursive cases. (Try it!). So we need induction. Here is a naive first attempt:

```
  induction H1
```



```

as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
   |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
   |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].

```

But now, although we get seven cases (as we would expect from the definition of **exp\_match**), we lost a very important bit of information from *H1*: the fact that *s1* matched something of the form **Star** *re*. This means that we have to give proofs for *all* seven constructors of this definition, even though all but two of them (**MStar0** and **MStarApp**) are contradictory. We can still get the proof to go through for a few constructors, such as **EMempty**...

```

-
  simpl. intros H. apply H.

```

... but most of them get stuck. For **MChar**, for instance, we must show that  $s2 = \sim \text{Char } x' \rightarrow x' :: s2 = \sim \text{Char } x'$ , which is clearly impossible.

```

-

```

Abort.

The problem is that **induction** over a Prop hypothesis only works properly with hypotheses that are completely general, i.e., ones in which all the arguments are variables, as opposed to more complex expressions, such as **Star** *re*. In this respect it behaves more like **destruct** than like **inversion**.

We can solve this problem by generalizing over the problematic expressions with an explicit equality:

```

Lemma star_app: ∀ T (s1 s2 : list T) (re re' : reg_exp T),
  s1 =~ re' →
  re' = Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.

```

We can now proceed by performing induction over evidence directly, because the argument to the first hypothesis is sufficiently general, which means that we can discharge most cases by inverting the  $re' = \text{Star } re$  equality in the context.

This idiom is so common that Coq provides a tactic to automatically generate such equations for us, avoiding thus the need for changing the statements of our theorems. Calling *remember e as x* causes Coq to (1) replace all occurrences of the expression *e* by the variable *x*, and (2) add an equation  $x = e$  to the context. Here's how we can use it to show the above result:

Abort.

```

Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.

```

Proof.

```
intros T s1 s2 re H1.
remember (Star re) as re'.
```

We now have  $Hegre' : re' = \text{Star } re$ .

```
generalize dependent s2.
induction H1
```

```
as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
   |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
   |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

The  $Hegre'$  is contradictory in most cases, which allows us to conclude immediately.

```
- inversion Hegre'.
- inversion Hegre'.
- inversion Hegre'.
- inversion Hegre'.
- inversion Hegre'.
```

In the interesting cases (those that correspond to **Star**), we can proceed as usual. Note that the induction hypothesis  $IH2$  on the **MStarApp** case mentions an additional premise  $\text{Star } re'' = \text{Star } re'$ , which results from the equality generated by *remember*.

```
-
  inversion Hegre'. intros s H. apply H.
-
  inversion Hegre'. rewrite H0 in IH2, Hmatch1.
  intros s2 H1. rewrite ← app_assoc.
  apply MStarApp.
  + apply Hmatch1.
  + apply IH2.
    × reflexivity.
    × apply H1.
```

Qed.

**Exercise: 4 stars (exp\_match\_ex2)** The **MStar''** lemma below (combined with its converse, the **MStar'** exercise above), shows that our definition of **exp\_match** for **Star** is equivalent to the informal one given previously.

Lemma **MStar''** :  $\forall T (s : \text{list } T) (re : \text{reg\_exp } T),$

```
s =~ Star re →
∃ ss : list (list T),
s = fold app ss []
∧ ∀ s', ln s' ss → s' =~ re.
```

Proof.

*Admitted.*

□

**Exercise: 5 stars, advanced (pumping)** Module PUMPING.

One of the first interesting theorems in the theory of regular expressions is the so-called *pumping lemma*, which states, informally, that any sufficiently long string  $s$  matching a regular expression  $re$  can be “pumped” by repeating some middle section of  $s$  an arbitrary number of times to produce a new string also matching  $re$ .

To begin, we need to define “sufficiently long.” Since we are working in a constructive logic, we actually need to be able to calculate, for each regular expression  $re$ , the minimum length for strings  $s$  to guarantee “pumpability.”

```
Fixpoint pumping_constant {T} (re : reg_exp T) : nat :=
  match re with
  | EmptySet => 0
  | EmptyStr => 1
  | Char _ => 2
  | App re1 re2 =>
    pumping_constant re1 + pumping_constant re2
  | Union re1 re2 =>
    pumping_constant re1 + pumping_constant re2
  | Star _ => 1
  end.
```

Next, it is useful to define an auxiliary function that repeats a string (appends it to itself) some number of times.

```
Fixpoint napp {T} (n : nat) (l : list T) : list T :=
  match n with
  | 0 => []
  | S n' => l ++ napp n' l
  end.
```

Lemma napp\_plus:  $\forall T (n m : nat) (l : list T)$ ,  
napp (n + m) l = napp n l ++ napp m l.

Proof.

```
intros T n m l.
induction n as [|n IHn].
- reflexivity.
- simpl. rewrite IHn, app_assoc. reflexivity.
```

Qed.

Now, the pumping lemma itself says that, if  $s \sim re$  and if the length of  $s$  is at least the pumping constant of  $re$ , then  $s$  can be split into three substrings  $s1 ++ s2 ++ s3$  in such a way that  $s2$  can be repeated any number of times and the result, when combined with  $s1$  and  $s3$  will still match  $re$ . Since  $s2$  is also guaranteed not to be the empty string, this gives us a (constructive!) way to generate strings matching  $re$  that are as long as we like.

Lemma pumping :  $\forall T (re : reg\_exp T) s$ ,  
 $s \sim re \rightarrow$

```

pumping_constant re ≤ length s →
∃ s1 s2 s3 ,
  s = s1 ++ s2 ++ s3 ∧
  s2 ≠ [] ∧
  ∀ m, s1 ++ napp m s2 ++ s3 =~ re.

```

To streamline the proof (which you are to fill in), the `omega` tactic, which is enabled by the following `Require`, is helpful in several places for automatically completing tedious low-level arguments involving equalities or inequalities over natural numbers. We'll return to `omega` in a later chapter, but feel free to experiment with it now if you like. The first case of the induction gives an example of how it is used.

`Require Import Coq.omega.Omega.`

`Proof.`

```

intros T re s Hmatch.
induction Hmatch
as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
  | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
  | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
-
  simpl. omega.
  Admitted.

```

`End PUMPING.`

□

## 8.6 Improving Reflection

We've seen in the `Logic` chapter that we often need to relate boolean computations to statements in `Prop`. Unfortunately, performing this conversion by hand can result in tedious proof scripts. Consider the proof of the following theorem:

`Theorem filter_not_empty_In : ∀ n l,`  
`filter (beq_nat n) l ≠ [] →`  
`In n l.`

`Proof.`

```

intros n l. induction l as [|m l' IHL'].
-
  simpl. intros H. apply H. reflexivity.
-
  simpl. destruct (beq_nat n m) eqn:H.
  +
    intros .. rewrite beq_nat_true_iff in H. rewrite H.
    left. reflexivity.

```

```

+
  intros H'. right. apply IHl'. apply H'.
Qed.

```

In the first branch after `destruct`, we explicitly apply the `beq_nat_true_iff` lemma to the equation generated by destructing `beq_nat n m`, to convert the assumption `beq_nat n m = true` into the assumption `n = m`, which is what we need to complete this case.

We can streamline this proof by defining an inductive proposition that yields a better case-analysis principle for `beq_nat n m`. Instead of generating an equation such as `beq_nat n m = true`, which is not directly useful, this principle gives us right away the assumption we need: `n = m`. We'll actually define something a bit more general, which can be used with arbitrary properties (and not just equalities):

```

Inductive reflect (P : Prop) : bool → Prop :=
| ReflectT : P → reflect P true
| ReflectF : ¬ P → reflect P false.

```

The **reflect** property takes two arguments: a proposition  $P$  and a boolean  $b$ . Intuitively, it states that the property  $P$  is *reflected* in (i.e., equivalent to) the boolean  $b$ :  $P$  holds if and only if  $b = \text{true}$ . To see this, notice that, by definition, the only way we can produce evidence that `reflect P true` holds is by showing that  $P$  is true and using the `ReflectT` constructor. If we invert this statement, this means that it should be possible to extract evidence for  $P$  from a proof of `reflect P true`. Conversely, the only way to show `reflect P false` is by combining evidence for  $\neg P$  with the `ReflectF` constructor.

It is easy to formalize this intuition and show that the two statements are indeed equivalent:

```

Theorem iff_reflect : ∀ P b, (P ↔ b = true) → reflect P b.

```

Proof.

```

  intros P || H.
  - apply ReflectT. rewrite H. reflexivity.
  - apply ReflectF. rewrite H. intros H'. inversion H'.

```

Qed.

**Exercise: 2 stars, recommended (reflect\_iff)** Theorem `reflect_iff` :  $\forall P b, \text{reflect } P b \rightarrow (P \leftrightarrow b = \text{true})$ .

Proof.

*Admitted.*

□

The advantage of **reflect** over the normal “if and only if” connective is that, by destructing a hypothesis or lemma of the form `reflect P b`, we can perform case analysis on  $b$  while at the same time generating appropriate hypothesis in the two branches ( $P$  in the first subgoal and  $\neg P$  in the second).

To use **reflect** to produce a better proof of `filter_not_empty_in`, we begin by recasting the `beq_nat_iff_true` lemma into a more convenient form in terms of **reflect**:

Lemma `beq_natP` :  $\forall n m$ , **reflect** ( $n = m$ ) (`beq_nat`  $n m$ ).

Proof.

`intros n m.`

`apply iff_reflect. rewrite beq_nat_true_iff. reflexivity.`

Qed.

The new proof of `filter_not_empty_in` now goes as follows. Notice how the calls to `destruct` and `apply` are combined into a single call to `destruct`. (To see this clearly, look at the two proofs of `filter_not_empty_in` in your Coq browser and observe the differences in proof state at the beginning of the first case of the `destruct`.)

Theorem `filter_not_empty_in'` :  $\forall n l$ ,

`filter (beq_nat  $n$ )  $l \neq [] \rightarrow$`

`in  $n l$ .`

Proof.

`intros n l. induction l as [|m l' IHL'].`

-

`simpl. intros H. apply H. reflexivity.`

-

`simpl. destruct (beq_natP  $n m$ ) as [H | H].`

+

`intros .. rewrite H. left. reflexivity.`

+

`intros H'. right. apply IHL'. apply H'.`

Qed.

Although this technique arguably gives us only a small gain in convenience for this particular proof, using **reflect** consistently often leads to shorter and clearer proofs. We'll see many more examples where **reflect** comes in handy in later chapters.

The use of the **reflect** property was popularized by *SSReflect*, a Coq library that has been used to formalize important results in mathematics, including as the 4-color theorem and the Feit-Thompson theorem. The name *SSReflect* stands for *small-scale reflection*, i.e., the pervasive use of reflection to simplify small proof steps with boolean computations.

## 8.7 Additional Exercises

**Exercise: 4 stars, recommended (palindromes)** A palindrome is a sequence that reads the same backwards as forwards.

- Define an inductive proposition *pal* on **list**  $X$  that captures what it means to be a palindrome. (Hint: You'll need three cases. Your definition should be based on the structure of the list; just having a single constructor  $c : \text{forall } l, l = \text{rev } l \rightarrow \text{pal } l$  may seem obvious, but will not work very well.)
- Prove *pal\_app\_rev* that  $\text{forall } l, \text{pal } (l ++ \text{rev } l)$ .

- Prove *pal\_rev* that forall l, pal l -> l = rev l.

□

**Exercise: 5 stars, optional (palindrome\_converse)** Again, the converse direction is significantly more difficult, due to the lack of evidence. Using your definition of *pal* from the previous exercise, prove that forall l, l = rev l -> pal l.

□

**Exercise: 4 stars, advanced (filter\_challenge)** Let's prove that our definition of *filter* from the Poly chapter matches an abstract specification. Here is the specification, written out informally in English:

A list *l* is an “in-order merge” of *l1* and *l2* if it contains all the same elements as *l1* and *l2*, in the same order as *l1* and *l2*, but possibly interleaved. For example, 1;4;6;2;3 is an in-order merge of 1;6;2 and 4;3. Now, suppose we have a set *X*, a function **test**: *X*→**bool**, and a list *l* of type **list** *X*. Suppose further that *l* is an in-order merge of two lists, *l1* and *l2*, such that every item in *l1* satisfies **test** and no item in *l2* satisfies **test**. Then **filter test** *l* = *l1*.

Translate this specification into a Coq theorem and prove it. (You'll need to begin by defining what it means for one list to be a merge of two others. Do this with an inductive relation, not a **Fixpoint**.)

□

**Exercise: 5 stars, advanced, optional (filter\_challenge\_2)** A different way to characterize the behavior of *filter* goes like this: Among all subsequences of *l* with the property that **test** evaluates to **true** on all their members, **filter test** *l* is the longest. Formalize this claim and prove it.

□

**Exercise: 4 stars, advanced (NoDup)** Recall the definition of the *ln* property from the Logic chapter, which asserts that a value *x* appears at least once in a list *l*:

Your first task is to use *ln* to define a proposition *disjoint* *X* *l1* *l2*, which should be provable exactly when *l1* and *l2* are lists (with elements of type *X*) that have no elements in common.

Next, use *ln* to define an inductive proposition *NoDup* *X* *l*, which should be provable exactly when *l* is a list (with elements of type *X*) where every member is different from every other. For example, *NoDup* **nat** [1;2;3;4] and *NoDup* **bool** [] should be provable, while *NoDup* **nat** [1;2;1] and *NoDup* **bool** [true;true] should not be.

Finally, state and prove one or more interesting theorems relating *disjoint*, *NoDup* and ++ (list append).

□

**Exercise: 3 stars, recommended (nostutter)** Formulating inductive definitions of properties is an important skill you'll need in this course. Try to solve this exercise without any help at all.

We say that a list “stutters” if it repeats the same element consecutively. The property “**nostutter** *mylist*” means that *mylist* does not stutter. Formulate an inductive definition for **nostutter**. (This is different from the *NoDup* property in the exercise above; the sequence 1;4;1 repeats but does not stutter.)

Inductive **nostutter** {X:Type} : list X → Prop :=

.

Make sure each of these tests succeeds, but feel free to change the suggested proof (in comments) if the given one doesn't work for you. Your definition might be different from ours and still be correct, in which case the examples might need a different proof. (You'll notice that the suggested proofs use a number of tactics we haven't talked about, to make them more robust to different possible ways of defining **nostutter**. You can probably just uncomment and use them as-is, but you can also prove each example with more basic tactics.)

Example test\_nostutter\_1: **nostutter** [3;1;4;1;5;6].

*Admitted.*

Example test\_nostutter\_2: **nostutter** (@nil **nat**).

*Admitted.*

Example test\_nostutter\_3: **nostutter** [5].

*Admitted.*

Example test\_nostutter\_4: **not** (**nostutter** [3;1;1;4]).

*Admitted.*

□

**Exercise: 4 stars, advanced (pigeonhole principle)** The *pigeonhole principle* states a basic fact about counting: if we distribute more than *n* items into *n* pigeonholes, some pigeonhole must contain at least two items. As often happens, this apparently trivial fact about numbers requires non-trivial machinery to prove, but we now have enough...

First prove an easy useful lemma.

Lemma in\_split : ∀ (X:Type) (x:X) (l:list X),

in x l →

∃ l1 l2 , l = l1 ++ x :: l2.

Proof.

*Admitted.*

Now define a property **repeats** such that **repeats** X l asserts that l contains at least one repeated element (of type X).

Inductive **repeats** {X:Type} : list X → Prop :=



Now, here's a way to formalize the pigeonhole principle. Suppose list `l2` represents a list of pigeonhole labels, and list `l1` represents the labels assigned to a list of items. If there are more items than labels, at least two items must have the same label – i.e., list `l1` must contain repeats.

This proof is much easier if you use the `excluded_middle` hypothesis to show that `ln` is decidable, i.e.,  $\forall x\ l, (\text{ln } x\ l) \vee \neg (\text{ln } x\ l)$ . However, it is also possible to make the proof go through *without* assuming that `ln` is decidable; if you manage to do this, you will not need the `excluded_middle` hypothesis.

```
Theorem pigeonhole_principle:  $\forall$  ( $X$ :Type) ( $l1\ l2$ :list  $X$ ),
  excluded_middle  $\rightarrow$ 
  ( $\forall x, \text{ln } x\ l1 \rightarrow \text{ln } x\ l2$ )  $\rightarrow$ 
  length  $l2 <$  length  $l1 \rightarrow$ 
  repeats  $l1$ .
```

Proof.

```
  intros  $X\ l1$ . induction  $l1$  as [ $x\ l1'$   $IHL1'$ ].
```

```
  Admitted.
```

□

```
  Date : 2015 – 08 – 11 12 : 03 : 04 – 0400(Tue, 11Aug2015)
```

# Chapter 9

## Library Maps

### 9.1 Maps: Total and Partial Maps

Maps (or dictionaries) are ubiquitous data structures, both in software construction generally and in the theory of programming languages in particular; we’re going to need them in many places in the coming chapters. They also make a nice case study using ideas we’ve seen in previous chapters, including building data structures out of higher-order functions (from `Basics` and `Poly`) and the use of reflection to streamline proofs (from `IndProp`).

We’ll define two flavors of maps: *total* maps, which include a “default” element to be returned when a key being looked up doesn’t exist, and *partial* maps, which return an **option** to indicate success or failure. The latter is defined in terms of the former, using `None` as the default element.

### 9.2 The Coq Standard Library

One small digression before we start.

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (and, transitively, all the earlier chapters). Instead, in this chapter and from now, on we’re going to import the definitions and theorems we need directly from Coq’s standard library stuff. You should not notice much difference, though, because we’ve been careful to name our own definitions and theorems the same as their counterparts in the standard library, wherever they overlap.

```
Require Import Coq.Arith.Arith.  
Require Import Coq.Bool.Bool.  
Require Import Coq.Logic.FunctionalExtensionality.
```

Documentation for the standard library can be found at <http://coq.inria.fr/library/>.

The `SearchAbout` command is a good way to look for theorems involving objects of specific types.

## 9.3 Identifiers

First, we need a type for the keys that we use to index into our maps. For this purpose, we again use the type **id** from the **Lists** chapter. To make this chapter self contained, we repeat its definition here, together with the equality comparison function for **ids** and its fundamental property.

```
Inductive id : Type :=
| ld : nat → id.
```

```
Definition beq_id id1 id2 :=
  match id1, id2 with
  | ld n1, ld n2 => beq_nat n1 n2
  end.
```

Theorem `beq_id_refl` :  $\forall id, \text{true} = \text{beq\_id } id \text{ } id$ .

Proof.

```
intros [n]. simpl. rewrite ← beq_nat_refl.
reflexivity. Qed.
```

The following useful property of `beq_id` follows from an analogous lemma about numbers:

Theorem `beq_id_true_iff` :  $\forall id1 \ id2 : \text{id},$   
 $\text{beq\_id } id1 \ id2 = \text{true} \leftrightarrow id1 = id2$ .

Proof.

```
intros [n1] [n2].
unfold beq_id.
rewrite beq_nat_true_iff.
split.
- intros H. rewrite H. reflexivity.
- intros H. inversion H. reflexivity.
```

Qed.

Similarly:

Theorem `beq_id_false_iff` :  $\forall x \ y : \text{id},$   
 $\text{beq\_id } x \ y = \text{false}$   
 $\leftrightarrow x \neq y$ .

Proof.

```
intros x y. rewrite ← beq_id_true_iff.
rewrite not_true_iff_false. reflexivity. Qed.
```

This useful variant follows just by rewriting:

Theorem `false_beq_id` :  $\forall x \ y : \text{id},$   
 $x \neq y$   
 $\rightarrow \text{beq\_id } x \ y = \text{false}.$

Proof.

```
intros x y. rewrite beq_id_false_iff.
```

intros  $H$ . apply  $H$ . Qed.

## 9.4 Total Maps

Our main job in this chapter will be to build a definition of partial maps that is similar in behavior to the one we saw in the `Lists` chapter, plus accompanying lemmas about their behavior.

This time around, though, we're going to use *functions*, rather than lists of key-value pairs, to build maps. The advantage of this representation is that it offers a more *extensional* view of maps, where two maps that respond to queries in the same way will be represented as literally the same thing (the same function), rather than just "equivalent" data structures. This, in turn, simplifies proofs that use maps.

We build partial maps in two steps. First, we define a type of *total maps* that return a default value when we look up a key that is not present in the map.

**Definition** `total_map` ( $A:\text{Type}$ ) := `id`  $\rightarrow A$ .

Intuitively, a total map over an element type  $A$  is just a function that can be used to look up `ids`, yielding  $A$ s.

The function `t_empty` yields an empty total map, given a default element; this map always returns the default element when applied to any `id`.

**Definition** `t_empty` { $A:\text{Type}$ } ( $v : A$ ) : `total_map`  $A$  :=  
(`fun`  $_ \Rightarrow v$ ).

More interesting is the `update` function, which (as before) takes a map  $m$ , a key  $x$ , and a value  $v$  and returns a new map that takes  $x$  to  $v$  and takes every other key to whatever  $m$  does.

**Definition** `t_update` { $A:\text{Type}$ } ( $m : \text{total\_map } A$ )  
( $x : \text{id}$ ) ( $v : A$ ) :=  
`fun`  $x' \Rightarrow \text{if beq\_id } x \ x' \text{ then } v \text{ else } m \ x'$ .

This definition is a nice example of higher-order programming. The `t_update` function takes a *function*  $m$  and yields a new function `fun`  $x' \Rightarrow \dots$  that behaves like the desired map.

For example, we can build a map taking `ids` to `bools`, where `ld 3` is mapped to `true` and every other key is mapped to `false`, like this:

**Definition** `examplemap` :=  
`t_update` (`t_update` (`t_empty false`) (`ld 1`) `false`)  
(`ld 3`) `true`.

This completes the definition of total maps. Note that we don't need to define a `find` operation because it is just function application!

**Example** `update_example1` : `examplemap` (`ld 0`) = `false`.

**Proof.** `reflexivity`. `Qed`.

**Example** `update_example2` : `examplemap` (`ld 1`) = `false`.

Proof. reflexivity. Qed.

Example update\_example3 : examplemap (Id 2) = false.

Proof. reflexivity. Qed.

Example update\_example4 : examplemap (Id 3) = true.

Proof. reflexivity. Qed.

To use maps in later chapters, we'll need several fundamental facts about how they behave. Even if you don't work the following exercises, make sure you thoroughly understand the statements of the lemmas! (Some of the proofs require the functional extensionality axiom discussed in the Logic chapter, which is also included in the standard library.)

**Exercise: 2 stars, optional (t\_update\_eq)** First, if we update a map  $m$  at a key  $x$  with a new value  $v$  and then look up  $x$  in the map resulting from the `update`, we get back  $v$ :

Lemma t\_update\_eq :  $\forall A (m : \text{total\_map } A) x v$ ,

(t\_update  $m x v$ )  $x = v$ .

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional (t\_update\_neq)** On the other hand, if we update a map  $m$  at a key  $x1$  and then look up a *different* key  $x2$  in the resulting map, we get the same result that  $m$  would have given:

Theorem t\_update\_neq :  $\forall (X : \text{Type}) v x1 x2$   
( $m : \text{total\_map } X$ ),

$x1 \neq x2 \rightarrow$

(t\_update  $m x1 v$ )  $x2 = m x2$ .

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional (t\_update\_shadow)** If we update a map  $m$  at a key  $x$  with a value  $v1$  and then update again with the same key  $x$  and another value  $v2$ , the resulting map behaves the same (gives the same result when applied to any key) as the simpler map obtained by performing just the second `update` on  $m$ :

Lemma t\_update\_shadow :  $\forall A (m : \text{total\_map } A) v1 v2 x$ ,

t\_update (t\_update  $m x v1$ )  $x v2$

= t\_update  $m x v2$ .

Proof.

*Admitted.*

□

For the final two lemmas about total maps, it's convenient to use the reflection idioms introduced in chapter `IndProp`. We begin by proving a fundamental *reflection lemma* relating the equality proposition on `ids` with the boolean function `beq_id`.

**Exercise: 2 stars (`beq_idP`)** Use the proof of `beq_natP` in chapter `IndProp` as a template to prove the following:

Lemma `beq_idP` :  $\forall x y, \text{reflect } (x = y) (\text{beq\_id } x y)$ .

Proof.

*Admitted.*

□

Now, given `ids`  $x1$  and  $x2$ , we can use the `destruct (beq_idP  $x1$   $x2$ )` to simultaneously perform case analysis on the result of `beq_id  $x1$   $x2$`  and generate hypotheses about the equality (in the sense of  $=$ ) of  $x1$  and  $x2$ .

**Exercise: 2 stars (`t_update_same`)** Using the example in chapter `IndProp` as a template, use `beq_idP` to prove the following theorem, which states that if we update a map to assign key  $x$  the same value as it already has in  $m$ , then the result is equal to  $m$ :

Theorem `t_update_same` :  $\forall X x (m : \text{total\_map } X),$   
 $\text{t\_update } m x (m x) = m$ .

Proof.

*Admitted.*

□

**Exercise: 3 stars, recommended (`t_update_permute`)** Use `beq_idP` to prove one final property of the `update` function: If we update a map  $m$  at two distinct keys, it doesn't matter in which order we do the updates.

Theorem `t_update_permute` :  $\forall (X:\text{Type}) v1 v2 x1 x2$   
 $(m : \text{total\_map } X),$

$x2 \neq x1 \rightarrow$   
 $(\text{t\_update } (\text{t\_update } m x2 v2) x1 v1)$   
 $= (\text{t\_update } (\text{t\_update } m x1 v1) x2 v2).$

Proof.

*Admitted.*

□

## 9.5 Partial maps

Finally, we define *partial maps* on top of total maps. A partial map with elements of type  $A$  is simply a total map with elements of type `option A` and default element `None`.

Definition `partial_map (A:Type)` := `total_map (option A)`.

Definition empty {A:Type} : partial\_map A :=  
 t\_empty None.

Definition update {A:Type} (m : partial\_map A)  
 (x : id) (v : A) :=  
 t\_update m x (Some v).

We can now lift all of the basic lemmas about total maps to partial maps.

Lemma update\_eq :  $\forall A (m : \text{partial\_map } A) x v$ ,  
 (update m x v) x = Some v.

Proof.

intros. unfold update. rewrite t\_update\_eq.  
 reflexivity.

Qed.

Theorem update\_neq :  $\forall (X:\text{Type}) v x1 x2$   
 (m : partial\_map X),

$x2 \neq x1 \rightarrow$   
 (update m x2 v) x1 = m x1.

Proof.

intros X v x1 x2 m H.  
 unfold update. rewrite t\_update\_neq. reflexivity.  
 apply H. Qed.

Lemma update\_shadow :  $\forall A (m : \text{partial\_map } A) v1 v2 x$ ,  
 update (update m x v1) x v2 = update m x v2.

Proof.

intros A m v1 v2 x1. unfold update. rewrite t\_update\_shadow.  
 reflexivity.

Qed.

Theorem update\_same :  $\forall X v x (m : \text{partial\_map } X)$ ,  
 m x = Some v  $\rightarrow$   
 update m x v = m.

Proof.

intros X v x m H. unfold update. rewrite  $\leftarrow$  H.  
 apply t\_update\_same.

Qed.

Theorem update\_permute :  $\forall (X:\text{Type}) v1 v2 x1 x2$   
 (m : partial\_map X),

$x2 \neq x1 \rightarrow$   
 (update (update m x2 v2) x1 v1)  
 = (update (update m x1 v1) x2 v2).

Proof.

intros X v1 v2 x1 x2 m. unfold update.  
 apply t\_update\_permute.

Qed.

*Date : 2015 – 12 – 11 17 : 17 : 29 – 0500(Fri, 11Dec2015)*



# Chapter 10

## Library ProofObjects

### 10.1 ProofObjects: The Curry-Howard Correspondence

"Algorithms are the computational content of proofs." –Robert Harper

Require Export IndProp.

We have seen that Coq has mechanisms both for *programming*, using inductive data types like `nat` or `list` and functions over these types, and for *proving* properties of these programs, using inductive propositions (like `ev`), implication, universal quantification, and the like. So far, we have mostly treated these mechanisms as if they were quite separate, and for many purposes this is a good way to think. But we have also seen hints that Coq's programming and proving facilities are closely related. For example, the keyword `Inductive` is used to declare both data types and propositions, and  $\rightarrow$  is used both to describe the type of functions on data and logical implication. This is not just a syntactic accident! In fact, programs and proofs in Coq are almost the same thing. In this chapter we will study how this works.

We have already seen the fundamental idea: provability in Coq is represented by concrete *evidence*. When we construct the proof of a basic proposition, we are actually building a tree of evidence, which can be thought of as a data structure.

If the proposition is an implication like  $A \rightarrow B$ , then its proof will be an evidence *transformer*: a recipe for converting evidence for A into evidence for B. So at a fundamental level, proofs are simply programs that manipulate evidence.

Question: If evidence is data, what are propositions themselves?

Answer: They are types!

Look again at the formal definition of the `ev` property.

Print ev.

Suppose we introduce an alternative pronunciation of `:`. Instead of "has type," we can say "is a proof of." For example, the second line in the definition of `ev` declares that `ev_0` : `ev 0`. Instead of "ev\_0 has type `ev 0`," we can say that "ev\_0 is a proof of `ev 0`."

This pun between types and propositions – between `:` as "has type" and `:` as "is a proof

of“ or ”is evidence for“ – is called the *Curry-Howard correspondence*. It proposes a deep connection between the world of logic and the world of computation:

propositions	~	types
proofs	~	data values

Many useful insights follow from this connection. To begin with, it gives us a natural interpretation of the type of the `ev_SS` constructor:

Check `ev_SS`.

This can be read ”`ev_SS` is a constructor that takes two arguments – a number `n` and evidence for the proposition `ev n` – and yields evidence for the proposition `ev (S (S n))`.”

Now let’s look again at a previous proof involving `ev`.

Theorem `ev_4` : `ev 4`.

Proof.

`apply ev_SS. apply ev_SS. apply ev_0. Qed.`

As with ordinary data values and functions, we can use the `Print` command to see the *proof object* that results from this proof script.

`Print ev_4.`

As a matter of fact, we can also write down this proof object *directly*, without the need for a separate proof script:

Check `(ev_SS 2 (ev_SS 0 ev_0))`.

The expression `ev_SS 2 (ev_SS 0 ev_0)` can be thought of as instantiating the parameterized constructor `ev_SS` with the specific arguments 2 and 0 plus the corresponding proof objects for its premises `ev 2` and `ev 0`. Alternatively, we can think of `ev_SS` as a primitive ”evidence constructor“ that, when applied to a particular number, wants to be further applied to evidence that that number is even; its type, forall n, ev n -> ev (S (S n)), expresses this functionality, in the same way that the polymorphic type  $\forall X, \text{list } X$  expresses the fact that the constructor `nil` can be thought of as a function from types to empty lists with elements of that type.

You may recall (as seen in the `Logic` chapter) that we can use function application syntax to instantiate universally quantified variables in lemmas, as well as to supply evidence for assumptions that these lemmas impose. For instance,

Theorem `ev_4'`: `ev 4`.

Proof.

`apply (ev_SS 2 (ev_SS 0 ev_0)).`

`Qed.`

We can now see that this feature is a trivial consequence of the status the Coq grants to proofs and propositions: Lemmas and hypotheses can be combined in expressions (i.e., proof objects) according to the same basic rules used for programs in the language.

## 10.2 Proof Scripts

The *proof objects* we've been discussing lie at the core of how Coq operates. When Coq is following a proof script, what is happening internally is that it is gradually constructing a proof object – a term whose type is the proposition being proved. The tactics between **Proof** and **Qed** tell it how to build up a term of the required type. To see this process in action, let's use the **Show Proof** command to display the current state of the proof tree at various points in the following tactic proof.

Theorem `ev_4''` : **ev** 4.

Proof.

  Show Proof.

  apply ev\_SS.

  Show Proof.

  apply ev\_SS.

  Show Proof.

  apply ev\_0.

  Show Proof.

Qed.

At any given moment, Coq has constructed a term with some "holes" (indicated by ?1, ?2, and so on), and it knows what type of evidence is needed at each hole.

Each of the holes corresponds to a subgoal, and the proof is finished when there are no more subgoals. At this point, the evidence we've built stored in the global context under the name given in the **Theorem** command.

Tactic proofs are useful and convenient, but they are not essential: in principle, we can always construct the required evidence by hand, as shown above. Then we can use **Definition** (rather than **Theorem**) to give a global name directly to a piece of evidence.

Definition `ev_4'''` : **ev** 4 :=

  ev\_SS 2 (ev\_SS 0 ev\_0).

All these different ways of building the proof lead to exactly the same evidence being saved in the global environment.

Print `ev_4`.

Print `ev_4'`.

Print `ev_4''`.

Print `ev_4'''`.

**Exercise: 1 star (eight\_is\_even)** Give a tactic proof and a proof object showing that **ev** 8.

Theorem `ev_8` : **ev** 8.

Proof.

*Admitted.*

Definition `ev_8'` : **ev** 8 :=

*admit.*

□

## 10.3 Quantifiers, Implications, Functions

In Coq's computational universe (where data structures and programs live), there are two sorts of values with arrows in their types: *constructors* introduced by **Inductive**-ly defined data types, and *functions*.

Similarly, in Coq's logical universe (where we carry out proofs), there are two ways of giving evidence for an implication: constructors introduced by **Inductive**-ly defined propositions, and... functions!

For example, consider this statement:

Theorem `ev_plus4` :  $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (4 + n)$ .

Proof.

```
intros n H. simpl.  
apply ev_SS.  
apply ev_SS.  
apply H.
```

Qed.

What is the proof object corresponding to `ev_plus4`?

We're looking for an expression whose *type* is  $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (4 + n)$  – that is, a *function* that takes two arguments (one number and a piece of evidence) and returns a piece of evidence! Here it is:

Definition `ev_plus4'` :  $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (4 + n) :=$

```
fun (n : nat) => fun (H : ev n) =>  
  ev_SS (S (S n)) (ev_SS n H).
```

Check `ev_plus4'`.

Recall that `fun n => blah` means "the function that, given `n`, yields *blah*," and that Coq treats `4 + n` and `S (S (S (S n)))` as synonyms. Another equivalent way to write this definition is:

Definition `ev_plus4''` (n : nat) (H : ev n) : ev (4 + n) :=

```
ev_SS (S (S n)) (ev_SS n H).
```

Check `ev_plus4''`.

When we view the proposition being proved by `ev_plus4` as a function type, one aspect of it may seem a little unusual. The second argument's type, `ev n`, mentions the *value* of the first argument, `n`. While such *dependent types* are not found in conventional programming languages, they can be useful in programming too, as recent work in the functional programming community demonstrates.

Notice that both implication ( $\rightarrow$ ) and quantification ( $\forall$ ) correspond to functions on evidence. In fact, they are really the same thing:  $\rightarrow$  is just a shorthand for a degenerate use

of  $\forall$  where there is no dependency, i.e., no need to give a name to the type on the LHS of the arrow.

For example, consider this proposition:

```
Definition ev_plus2 : Prop :=
   $\forall n, \forall (E : \mathbf{ev} \ n), \mathbf{ev} \ (n + 2).$ 
```

A proof term inhabiting this proposition would be a function with two arguments: a number  $n$  and some evidence  $E$  that  $n$  is even. But the name  $E$  for this evidence is not used in the rest of the statement of `ev_plus2`, so it's a bit silly to bother making up a name for it. We could write it like this instead, using the dummy identifier `_` in place of a real name:

```
Definition ev_plus2' : Prop :=
   $\forall n, \forall (_ : \mathbf{ev} \ n), \mathbf{ev} \ (n + 2).$ 
```

Or, equivalently, we can write it in more familiar notation:

```
Definition ev_plus2'' : Prop :=
   $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (n + 2).$ 
```

In general, " $P \rightarrow Q$ " is just syntactic sugar for " $\forall (_ : P), Q$ ".

## 10.4 Connectives as Inductive Types

Inductive definitions are powerful enough to express most of the connectives and quantifiers we have seen so far. Indeed, only universal quantification (and thus implication) is built into Coq; all the others are defined inductively. We study these definitions in this section.

```
Module PROPS.
```

### 10.4.1 Conjunction

To prove that  $P \wedge Q$  holds, we must present evidence for both  $P$  and  $Q$ . Thus, it makes sense to define a proof object for  $P \wedge Q$  as consisting of a pair of two proofs: one for  $P$  and another one for  $Q$ . This leads to the following definition.

```
Module AND.
```

```
Inductive and (P Q : Prop) : Prop :=
| conj : P  $\rightarrow$  Q  $\rightarrow$  and P Q.
```

```
End AND.
```

Notice the similarity with the definition of the **prod** type, given in chapter `Poly`; the only difference is that **prod** takes `Type` arguments, whereas **and** takes `Prop` arguments.

```
Print prod.
```

This should clarify why `destruct` and `intros` patterns can be used on a conjunctive hypothesis. Case analysis allows us to consider all possible ways in which  $P \wedge Q$  was proved

– here just one (the `conj` constructor). Similarly, the `split` tactic actually works for any inductively defined proposition with only one constructor. In particular, it works for **and**:

Lemma `and_comm` :  $\forall P Q : \text{Prop}, P \wedge Q \leftrightarrow Q \wedge P$ .

Proof.

```
intros P Q. split.
- intros [HP HQ]. split.
  + apply HQ.
  + apply HP.
- intros [HP HQ]. split.
  + apply HQ.
  + apply HP.
```

Qed.

This shows why the inductive definition of **and** can be manipulated by tactics as we’ve been doing. We can also use it to build proofs directly, using pattern-matching. For instance:

Definition `and_comm'_aux`  $P Q (H : P \wedge Q) :=$

```
match H with
| conj HP HQ => conj HQ HP
end.
```

Definition `and_comm'`  $P Q : P \wedge Q \leftrightarrow Q \wedge P :=$

```
conj (and_comm'_aux P Q) (and_comm'_aux Q P).
```

**Exercise: 2 stars, optional (`conj_fact`)** Construct a proof object demonstrating the following proposition.

Definition `conj_fact` :  $\forall P Q R, P \wedge Q \rightarrow Q \wedge R \rightarrow P \wedge R :=$

```
admit.
□
```

## 10.4.2 Disjunction

The inductive definition of disjunction uses two constructors, one for each side of the disjunct:

Module `OR`.

Inductive `or` ( $P Q : \text{Prop}$ ) : `Prop` :=

```
| or_introl : P → or P Q
| or_intror : Q → or P Q.
```

End `OR`.

This declaration explains the behavior of the `destruct` tactic on a disjunctive hypothesis, since the generated subgoals match the shape of the `or_introl` and `or_intror` constructors.

Once again, we can also directly write proof objects for theorems involving **or**, without resorting to tactics.

**Exercise: 2 stars, optional (or\_commut’)** Try to write down an explicit proof object for `or_commut` (without using `Print` to peek at the ones we already defined!).

```
Definition or_comm : ∀ P Q, P ∨ Q → Q ∨ P :=
  admit.
□
```

### 10.4.3 Existential Quantification

To give evidence for an existential quantifier, we package a witness `x` together with a proof that `x` satisfies the property `P`:

Module EX.

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
| ex_intro : ∀ x : A, P x → ex P.
```

End EX.

This may benefit from a little unpacking. The core definition is for a type former `ex` that can be used to build propositions of the form `ex P`, where `P` itself is a *function* from witness values in the type `A` to propositions. The `ex_intro` constructor then offers a way of constructing evidence for `ex P`, given a witness `x` and a proof of `P x`.

The more familiar form  $\exists x, P\ x$  desugars to an expression involving `ex`:

Check `ex (fun n ⇒ ev n)`.

Here’s how to define an explicit proof object involving `ex`:

```
Definition some_nat_is_even : ∃ n, ev n :=
  ex_intro ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

**Exercise: 2 stars, optional (ex\_ev\_Sn)** Complete the definition of the following proof object:

```
Definition ex_ev_Sn : ex (fun n ⇒ ev (S n)) :=
  admit.
□
```

### 10.4.4 True and False

The inductive definition of the `True` proposition is simple:

```
Inductive True : Prop :=
| : True.
```

It has one constructor (so every proof of `True` is the same, so being given a proof of `True` is not informative.)

`False` is equally simple – indeed, so simple it may look syntactically wrong at first glance!

```
Inductive False : Prop :=.
```

That is, **False** is an inductive type with *no* constructors – i.e., no way to build evidence for it.

End PROPS.

## 10.5 Programming with Tactics

If we can build proofs by giving explicit terms rather than executing tactic scripts, you may be wondering whether we can build *programs* using *tactics* rather than explicit terms. Naturally, the answer is yes!

Definition add1 : **nat** → **nat**.

intro *n*.

Show Proof.

apply **S**.

Show Proof.

apply *n*. Defined.

Print add1.

Compute add1 2.

Notice that we terminate the **Definition** with a **.** rather than with **:=** followed by a term. This tells Coq to enter *proof scripting mode* to build an object of type **nat** → **nat**. Also, we terminate the proof with **Defined** rather than **Qed**; this makes the definition *transparent* so that it can be used in computation like a normally-defined function. (**Qed**-defined objects are opaque during computation.)

This feature is mainly useful for writing functions with dependent types, which we won't explore much further in this book. But it does illustrate the uniformity and orthogonality of the basic ideas in Coq.

## 10.6 Equality

Even Coq's equality relation is not built in. It has the following inductive definition. (Actually, the definition in the standard library is a small variant of this, which gives an induction principle that is slightly easier to use.)

Module MYEQUALITY.

Inductive **eq** {*X*:Type} : *X* → *X* → Prop :=

| eq\_refl : ∀ *x*, **eq** *x* *x*.

Notation "*x* = *y*" := (**eq** *x* *y*)

(at level 70, no associativity)

: *type\_scope*.

The way to think about this definition is that, given a set *X*, it defines a *family* of propositions "*x* is equal to *y*," indexed by pairs of values (*x* and *y*) from *X*. There is just



one way of constructing evidence for each member of this family: applying the constructor `eq_refl` to a type `X` and a value `x : X` yields evidence that `x` is equal to `x`.

**Exercise: 2 stars (leibniz\_equality)** The inductive definition of equality corresponds to *Leibniz equality*: what we mean when we say “`x` and `y` are equal” is that every property on `P` that is true of `x` is also true of `y`.

Lemma `leibniz_equality` :  $\forall (X : \text{Type}) (x\ y : X),$   
 $x = y \rightarrow \forall P : X \rightarrow \text{Prop}, P\ x \rightarrow P\ y.$

Proof.

*Admitted.*

□

We can use `eq_refl` to construct evidence that, for example,  $2 = 2$ . Can we also use it to construct evidence that  $1 + 1 = 2$ ? Yes, we can. Indeed, it is the very same piece of evidence! The reason is that Coq treats as “the same” any two terms that are *convertible* according to a simple set of computation rules. These rules, which are similar to those used by `Compute`, include evaluation of function application, inlining of definitions, and simplification of matches.

Lemma four:  $2 + 2 = 1 + 3.$

Proof.

`apply eq_refl.`

Qed.

The `reflexivity` tactic that we have used to prove equalities up to now is essentially just short-hand for `apply refl_equal`.

In tactic-based proofs of equality, the conversion rules are normally hidden in uses of `simpl` (either explicit or implicit in other tactics such as `reflexivity`). But you can see them directly at work in the following explicit proof objects:

Definition four' :  $2 + 2 = 1 + 3 :=$   
`eq_refl 4.`

Definition singleton :  $\forall (X : \text{Set}) (x : X), [] ++ [x] = x :: [] :=$   
`fun (X : Set) (x : X) => eq_refl [x].`

End MYEQUALITY.

Definition quiz6 :  $\exists x, x + 3 = 4$   
`:= ex_intro (fun z => (z + 3 = 4)) 1 (refl_equal 4).`

## 10.6.1 Inversion, Again

We’ve seen `inversion` used with both equality hypotheses and hypotheses about inductively defined propositions. Now that we’ve seen that these are actually the same thing, we’re in a position to take a closer look at how `inversion` behaves.

In general, the `inversion` tactic...

- takes a hypothesis  $H$  whose type  $P$  is inductively defined, and
- for each constructor  $C$  in  $P$ 's definition,
  - generates a new subgoal in which we assume  $H$  was built with  $C$ ,
  - adds the arguments (premises) of  $C$  to the context of the subgoal as extra hypotheses,
  - matches the conclusion (result type) of  $C$  against the current goal and calculates a set of equalities that must hold in order for  $C$  to be applicable,
  - adds these equalities to the context (and, for convenience, rewrites them in the goal), and
  - if the equalities are not satisfiable (e.g., they involve things like  $S\ n = O$ ), immediately solves the subgoal.

*Example:* If we invert a hypothesis built with **or**, there are two constructors, so two subgoals get generated. The conclusion (result type) of the constructor  $(P \vee Q)$  doesn't place any restrictions on the form of  $P$  or  $Q$ , so we don't get any extra equalities in the context of the subgoal.

*Example:* If we invert a hypothesis built with **and**, there is only one constructor, so only one subgoal gets generated. Again, the conclusion (result type) of the constructor  $(P \wedge Q)$  doesn't place any restrictions on the form of  $P$  or  $Q$ , so we don't get any extra equalities in the context of the subgoal. The constructor does have two arguments, though, and these can be seen in the context in the subgoal.

*Example:* If we invert a hypothesis built with **eq**, there is again only one constructor, so only one subgoal gets generated. Now, though, the form of the **refl\_equal** constructor does give us some extra information: it tells us that the two arguments to **eq** must be the same! The **inversion** tactic adds this fact to the context.

*Date :* 2016 – 02 – 17 17 : 39 : 13 – 0500(*Wed, 17Feb2016*)

# Chapter 11

## Library IndPrinciples

### 11.1 IndPrinciples: Induction Principles

Require Export ProofObjects.

With the Curry-Howard correspondence and its realization in Coq in mind, we can now take a deeper look at induction principles.

### 11.2 Basics

Every time we declare a new `Inductive` datatype, Coq automatically generates an *induction principle* for this type. This induction principle is a theorem like any other: If  $t$  is defined inductively, the corresponding induction principle is called  $t\_ind$ . Here is the one for natural numbers:

Check `nat_ind`.

The `induction` tactic is a straightforward wrapper that, at its core, simply performs `apply t_ind`. To see this more clearly, let's experiment with directly using `apply nat_ind`, instead of the `induction` tactic, to carry out some proofs. Here, for example, is an alternate proof of a theorem that we saw in the Basics chapter.

Theorem `mult_0_r'` :  $\forall n:\text{nat}$ ,  
 $n \times 0 = 0$ .

Proof.

```
  apply nat_ind.  
  - reflexivity.  
  - simpl. intros n' IHn'. rewrite → IHn'.  
    reflexivity. Qed.
```

This proof is basically the same as the earlier one, but a few minor differences are worth noting.

First, in the induction step of the proof (the "S" case), we have to do a little bookkeeping manually (the `intros`) that `induction` does automatically.

Second, we do not introduce `n` into the context before applying `nat_ind` – the conclusion of `nat_ind` is a quantified formula, and `apply` needs this conclusion to exactly match the shape of the goal state, including the quantifier. By contrast, the `induction` tactic works either with a variable in the context or a quantified variable in the goal.

These conveniences make `induction` nicer to use in practice than applying induction principles like `nat_ind` directly. But it is important to realize that, modulo these bits of bookkeeping, applying `nat_ind` is what we are really doing.

**Exercise: 2 stars, optional (plus\_one\_r')** Complete this proof without using the `induction` tactic.

Theorem `plus_one_r'` :  $\forall n:\text{nat}$ ,

$n + 1 = \text{S } n$ .

Proof.

*Admitted.*

□

Coq generates induction principles for every datatype defined with `Inductive`, including those that aren't recursive. Although of course we don't need induction to prove properties of non-recursive datatypes, the idea of an induction principle still makes sense for them: it gives a way to prove that a property holds for all values of the type.

These generated principles follow a similar pattern. If we define a type `t` with constructors `c1 ... cn`, Coq generates a theorem with this shape: `t_ind : forall P : t -> Prop, ... case for c1 ... -> ... case for c2 ... -> ... case for cn ... -> forall n : t, P n`. The specific shape of each case depends on the arguments to the corresponding constructor. Before trying to write down a general rule, let's look at some more examples. First, an example where the constructors take no arguments:

```
Inductive yesno : Type :=
```

```
| yes : yesno
```

```
| no : yesno.
```

```
Check yesno_ind.
```

**Exercise: 1 star, optional (rgb)** Write out the induction principle that Coq will generate for the following datatype. Write down your answer on paper or type it into a comment, and then compare it with what Coq prints.

```
Inductive rgb : Type :=
```

```
| red : rgb
```

```
| green : rgb
```

```
| blue : rgb.
```

```
Check rgb_ind.
```

□

Here's another example, this time with one of the constructors taking some arguments.

```
Inductive natlist : Type :=
| nnil : natlist
| ncons : nat → natlist → natlist.
```

Check natlist\_ind.

**Exercise: 1 star, optional (natlist1)** Suppose we had written the above definition a little differently:

```
Inductive natlist1 : Type :=
| nnil1 : natlist1
| nsnoc1 : natlist1 → nat → natlist1.
```

Now what will the induction principle look like? □

From these examples, we can extract this general rule:

- The type declaration gives several constructors; each corresponds to one clause of the induction principle.
- Each constructor  $c$  takes argument types  $a1 \dots an$ .
- Each  $ai$  can be either  $t$  (the datatype we are defining) or some other type  $s$ .
- The corresponding case of the induction principle says:
  - "For all values  $x1 \dots xn$  of types  $a1 \dots an$ , if  $P$  holds for each of the inductive arguments (each  $xi$  of type  $t$ ), then  $P$  holds for  $c \ x1 \dots xn$ ".

**Exercise: 1 star, optional (byntree\_ind)** Write out the induction principle that Coq will generate for the following datatype. (Again, write down your answer on paper or type it into a comment, and then compare it with what Coq prints.)

```
Inductive byntree : Type :=
| bempty : byntree
| bleaf : yesno → byntree
| nbranch : yesno → byntree → byntree → byntree.
```

□

**Exercise: 1 star, optional (ex\_set)** Here is an induction principle for an inductively defined set. `ExSet_ind : forall P : ExSet -> Prop, (forall b : bool, P (con1 b)) -> (forall (n : nat) (e : ExSet), P e -> P (con2 n e)) -> forall e : ExSet, P e` Give an **Inductive** definition of **ExSet**:

```
Inductive ExSet : Type :=
```

.

□

## 11.3 Polymorphism

Next, what about polymorphic datatypes?

The inductive definition of polymorphic lists `Inductive list (X:Type) : Type := | nil : list X | cons : X -> list X -> list X`. is very similar to that of **natlist**. The main difference is that, here, the whole definition is *parameterized* on a set **X**: that is, we are defining a *family* of inductive types **list X**, one for each **X**. (Note that, wherever **list** appears in the body of the declaration, it is always applied to the parameter **X**.) The induction principle is likewise parameterized on **X**: `list_ind : forall (X : Type) (P : list X -> Prop), P nil -> (forall (x : X) (l : list X), P (cons x l) -> P l)`. Note that the *whole* induction principle is parameterized on **X**. That is, `list_ind` can be thought of as a polymorphic function that, when applied to a type **X**, gives us back an induction principle specialized to the type **list X**.

**Exercise: 1 star, optional (tree)** Write out the induction principle that Coq will generate for the following datatype. Compare your answer with what Coq prints.

```
Inductive tree (X:Type) : Type :=
| leaf : X -> tree X
| node : tree X -> tree X -> tree X.
```

Check `tree_ind`.

□

**Exercise: 1 star, optional (mytype)** Find an inductive definition that gives rise to the following induction principle: `mytype_ind : forall (X : Type) (P : mytype X -> Prop), (forall x : X, P (constr1 X x)) -> (forall n : nat, P (constr2 X n)) -> (forall m : mytype X, P m -> forall n : nat, P (constr3 X m n)) -> forall m : mytype X, P m` □

**Exercise: 1 star, optional (foo)** Find an inductive definition that gives rise to the following induction principle: `foo_ind : forall (X Y : Type) (P : foo X Y -> Prop), (forall x : X, P (bar X Y x)) -> (forall y : Y, P (baz X Y y)) -> (forall f1 : nat -> foo X Y, (forall n : nat, P (f1 n)) -> P (quux X Y f1)) -> forall f2 : foo X Y, P f2` □

**Exercise: 1 star, optional (foo')** Consider the following inductive definition:

```
Inductive foo' (X:Type) : Type :=
| C1 : list X -> foo' X -> foo' X
| C2 : foo' X.
```

What induction principle will Coq generate for `foo'`? Fill in the blanks, then check your answer with Coq.) `foo'_ind : forall (X : Type) (P : foo' X -> Prop), (forall (l : list X) (f : foo' X), _____ -> _____) -> _____ -> forall f : foo' X, _____`

□

## 11.4 Induction Hypotheses

Where does the phrase “induction hypothesis” fit into this story?

The induction principle for numbers forall  $P : \text{nat} \rightarrow \text{Prop}$ ,  $P\ 0 \rightarrow (\text{forall } n : \text{nat}, P\ n \rightarrow P\ (S\ n)) \rightarrow \text{forall } n : \text{nat}, P\ n$  is a generic statement that holds for all propositions  $P$  (or rather, strictly speaking, for all families of propositions  $P$  indexed by a number  $n$ ). Each time we use this principle, we are choosing  $P$  to be a particular expression of type  $\text{nat} \rightarrow \text{Prop}$ .

We can make proofs by induction more explicit by giving this expression a name. For example, instead of stating the theorem `mult_0_r` as “ $\forall n, n \times 0 = 0$ ,” we can write it as “ $\forall n, P\_m0r\ n$ ,” where `P_m0r` is defined as...

```
Definition P_m0r (n: nat) : Prop :=  
  n × 0 = 0.
```

... or equivalently:

```
Definition P_m0r' : nat → Prop :=  
  fun n => n × 0 = 0.
```

Now it is easier to see where `P_m0r` appears in the proof.

```
Theorem mult_0_r'' : ∀ n: nat,  
  P_m0r n.
```

Proof.

```
  apply nat_ind.  
  - reflexivity.  
  -
```

```
    intros n IHn.  
    unfold P_m0r in IHn. unfold P_m0r. simpl. apply IHn. Qed.
```

This extra naming step isn’t something that we do in normal proofs, but it is useful to do it explicitly for an example or two, because it allows us to see exactly what the induction hypothesis is. If we prove  $\forall n, P\_m0r\ n$  by induction on  $n$  (using either `induction` or `apply nat_ind`), we see that the first subgoal requires us to prove `P_m0r 0` (“ $P$  holds for zero”), while the second subgoal requires us to prove  $\forall n', P\_m0r\ n' \rightarrow P\_m0r\ n' (S\ n')$  (that is “ $P$  holds of  $S\ n'$  if it holds of  $n'$ ” or, more elegantly, “ $P$  is preserved by  $S$ ”). The *induction hypothesis* is the premise of this latter implication – the assumption that  $P$  holds of  $n'$ , which we are allowed to use in proving that  $P$  holds for  $S\ n'$ .

## 11.5 More on the induction Tactic

The `induction` tactic actually does even more low-level bookkeeping for us than we discussed above.

Recall the informal statement of the induction principle for natural numbers:

- If  $P\ n$  is some proposition involving a natural number  $n$ , and we want to show that  $P$  holds for *all* numbers  $n$ , we can reason like this:
  - show that  $P\ 0$  holds
  - show that, if  $P\ n'$  holds, then so does  $P\ (S\ n')$
  - conclude that  $P\ n$  holds for all  $n$ .

So, when we begin a proof with `intros n` and then `induction n`, we are first telling Coq to consider a *particular*  $n$  (by introducing it into the context) and then telling it to prove something about *all* numbers (by using induction).

What Coq actually does in this situation, internally, is to "re-generalize" the variable we perform induction on. For example, in our original proof that `plus` is associative...

Theorem `plus_assoc'` :  $\forall\ n\ m\ p : \text{nat},$   
 $n + (m + p) = (n + m) + p.$

Proof.

```
intros n m p.
induction n as [| n'].
- reflexivity.
-
```

```
simpl. rewrite → IHn'. reflexivity. Qed.
```

It also works to apply `induction` to a variable that is quantified in the goal.

Theorem `plus_comm'` :  $\forall\ n\ m : \text{nat},$   
 $n + m = m + n.$

Proof.

```
induction n as [| n'].
- intros m. rewrite ← plus_n_0. reflexivity.
- intros m. simpl. rewrite → IHn'.
  rewrite ← plus_n_Sm. reflexivity. Qed.
```

Note that `induction n` leaves `m` still bound in the goal – i.e., what we are proving inductively is a statement beginning with  $\forall\ m$ .

If we do `induction` on a variable that is quantified in the goal *after* some other quantifiers, the `induction` tactic will automatically introduce the variables bound by these quantifiers into the context.

Theorem `plus_comm''` :  $\forall\ n\ m : \text{nat},$   
 $n + m = m + n.$

Proof.

```
induction m as [| m'].
- simpl. rewrite ← plus_n_0. reflexivity.
- simpl. rewrite ← IHm'.
  rewrite ← plus_n_Sm. reflexivity. Qed.
```



**Exercise: 1 star, optional (plus\_explicit\_prop)** Rewrite both `plus_assoc'` and `plus_comm'` and their proofs in the same style as `mult_0_r''` above – that is, for each theorem, give an explicit **Definition** of the proposition being proved by induction, and state the theorem and proof in terms of this defined proposition.

□

## 11.6 Induction Principles in Prop

Earlier, we looked in detail at the induction principles that Coq generates for inductively defined *sets*. The induction principles for inductively defined *propositions* like **ev** are a tiny bit more complicated. As with all induction principles, we want to use the induction principle on **ev** to prove things by inductively considering the possible shapes that something in **ev** can have. Intuitively speaking, however, what we want to prove are not statements about *evidence* but statements about *numbers*: accordingly, we want an induction principle that lets us prove properties of numbers by induction on evidence.

For example, from what we've said so far, you might expect the inductive definition of **ev**... Inductive `ev : nat -> Prop := | ev_0 : ev 0 | ev_SS : forall n : nat, ev n -> ev (S (S n))`. ...to give rise to an induction principle that looks like this... `ev_ind_max : forall P : (forall n : nat, ev n -> Prop), P O ev_0 -> (forall (m : nat) (E : ev m), P m E -> P (S (S m)) (ev_SS m E)) -> forall (n : nat) (E : ev n), P n E` ... because:

- Since **ev** is indexed by a number *n* (every **ev** object *E* is a piece of evidence that some particular number *n* is even), the proposition *P* is parameterized by both *n* and *E* – that is, the induction principle can be used to prove assertions involving both an even number and the evidence that it is even.
- Since there are two ways of giving evidence of evenness (**ev** has two constructors), applying the induction principle generates two subgoals:
  - We must prove that *P* holds for *O* and `ev_0`.
  - We must prove that, whenever *n* is an even number and *E* is an evidence of its evenness, if *P* holds of *n* and *E*, then it also holds of `S (S n)` and `ev_SS n E`.
- If these subgoals can be proved, then the induction principle tells us that *P* is true for *all* even numbers *n* and evidence *E* of their evenness.

This is more flexibility than we normally need or want: it is giving us a way to prove logical assertions where the assertion involves properties of some piece of *evidence* of evenness, while all we really care about is proving properties of *numbers* that are even – we are interested in assertions about numbers, not about evidence. It would therefore be more convenient to have an induction principle for proving propositions *P* that are parameterized just by *n* and whose conclusion establishes *P* for all even numbers *n*: `forall P : nat -> Prop, ... -> forall n :`

`nat`, even  $n \rightarrow P\ n$  For this reason, Coq actually generates the following simplified induction principle for **ev**:

Check `ev_ind`.

In particular, Coq has dropped the evidence term  $E$  as a parameter of the the proposition  $P$ .

In English, `ev_ind` says:

- Suppose,  $P$  is a property of natural numbers (that is,  $P\ n$  is a `Prop` for every  $n$ ). To show that  $P\ n$  holds whenever  $n$  is even, it suffices to show:
  - $P$  holds for 0,
  - for any  $n$ , if  $n$  is even and  $P$  holds for  $n$ , then  $P$  holds for  $S\ (S\ n)$ .

As expected, we can apply `ev_ind` directly instead of using `induction`.

Theorem `ev_ev' :  $\forall\ n, \mathbf{ev}\ n \rightarrow \mathbf{ev}'\ n$` .

Proof.

```

apply ev_ind.
-
  apply ev'_0.
-
  intros m Hm IH.
  apply (ev'_sum 2 m).
  + apply ev'_2.
  + apply IH.
```

Qed.

The precise form of an **Inductive** definition can affect the induction principle Coq generates.

For example, in chapter `IndProp`, we defined  $\leq$  as:

This definition can be streamlined a little by observing that the left-hand argument  $n$  is the same everywhere in the definition, so we can actually make it a "general parameter" to the whole definition, rather than an argument to each constructor.

```

Inductive le (n:nat) : nat → Prop :=
| le_n : le n n
| le_S :  $\forall\ m, (\mathbf{le}\ n\ m) \rightarrow (\mathbf{le}\ n\ (S\ m))$ .
```

Notation "`m <= n`" := `(le m n)`.

The second one is better, even though it looks less symmetric. Why? Because it gives us a simpler induction principle.

Check `le_ind`.

## 11.7 Formal vs. Informal Proofs by Induction

Question: What is the relation between a formal proof of a proposition  $P$  and an informal proof of the same proposition  $P$ ?

Answer: The latter should *teach* the reader how to produce the former.

Question: How much detail is needed??

Unfortunately, there is no single right answer; rather, there is a range of choices.

At one end of the spectrum, we can essentially give the reader the whole formal proof (i.e., the "informal" proof will amount to just transcribing the formal one into words). This may give the reader the ability to reproduce the formal one for themselves, but it probably doesn't *teach* them anything much.

At the other end of the spectrum, we can say "The theorem is true and you can figure out why for yourself if you think about it hard enough." This is also not a good teaching strategy, because often writing the proof requires one or more significant insights into the thing we're proving, and most readers will give up before they rediscover all the same insights as we did.

In the middle is the golden mean – a proof that includes all of the essential insights (saving the reader the hard work that we went through to find the proof in the first place) plus high-level suggestions for the more routine parts to save the reader from spending too much time reconstructing these (e.g., what the IH says and what must be shown in each case of an inductive proof), but not so much detail that the main ideas are obscured.

Since we've spent much of this chapter looking "under the hood" at formal proofs by induction, now is a good moment to talk a little about *informal* proofs by induction.

In the real world of mathematical communication, written proofs range from extremely longwinded and pedantic to extremely brief and telegraphic. Although the ideal is somewhere in between, while one is getting used to the style it is better to start out at the pedantic end. Also, during the learning phase, it is probably helpful to have a clear standard to compare against. With this in mind, we offer two templates – one for proofs by induction over *data* (i.e., where the thing we're doing induction on lives in **Type**) and one for proofs by induction over *evidence* (i.e., where the inductively defined thing lives in **Prop**).

### 11.7.1 Induction Over an Inductively Defined Set

*Template:*

- *Theorem:* <Universally quantified proposition of the form "For all  $n:S$ ,  $P(n)$ ," where  $S$  is some inductively defined set.>

*Proof:* By induction on  $n$ .

<one case for each constructor  $c$  of  $S$ ...>

- Suppose  $n = c\ a1\ \dots\ ak$ , where <...and here we state the IH for each of the  $a$ 's that has type  $S$ , if any>. We must show <...and here we restate  $P(c\ a1\ \dots\ ak)$ >.  
<go on and prove  $P(n)$  to finish the case...>

– <other cases similarly...>  $\square$

*Example:*

- *Theorem:* For all sets  $X$ , lists  $l : \mathbf{list} X$ , and numbers  $n$ , if  $\mathbf{length} l = n$  then  $\mathbf{index} (S n) l = \mathbf{None}$ .

*Proof:* By induction on  $l$ .

- Suppose  $l = []$ . We must show, for all numbers  $n$ , that, if  $\mathbf{length} [] = n$ , then  $\mathbf{index} (S n) [] = \mathbf{None}$ .

This follows immediately from the definition of *index*.

- Suppose  $l = x :: l'$  for some  $x$  and  $l'$ , where  $\mathbf{length} l' = n'$  implies  $\mathbf{index} (S n') l' = \mathbf{None}$ , for any number  $n'$ . We must show, for all  $n$ , that, if  $\mathbf{length} (x :: l') = n$  then  $\mathbf{index} (S n) (x :: l') = \mathbf{None}$ .

Let  $n$  be a number with  $\mathbf{length} l = n$ . Since  $\mathbf{length} l = \mathbf{length} (x :: l') = S (\mathbf{length} l')$ , it suffices to show that  $\mathbf{index} (S (\mathbf{length} l')) l' = \mathbf{None}$ . But this follows directly from the induction hypothesis, picking  $n'$  to be  $\mathbf{length} l'$ .  $\square$

## 11.7.2 Induction Over an Inductively Defined Proposition

Since inductively defined proof objects are often called "derivation trees," this form of proof is also known as *induction on derivations*.

*Template:*

- *Theorem:* <Proposition of the form " $Q \rightarrow P$ ," where  $Q$  is some inductively defined proposition (more generally, "For all  $x y z$ ,  $Q x y z \rightarrow P x y z$ ")>

*Proof:* By induction on a derivation of  $Q$ . <Or, more generally, "Suppose we are given  $x, y$ , and  $z$ . We show that  $Q x y z$  implies  $P x y z$ , by induction on a derivation of  $Q x y z$ "...>

<one case for each constructor  $c$  of  $Q$ ...>

- Suppose the final rule used to show  $Q$  is  $c$ . Then <...and here we state the types of all of the  $a$ 's together with any equalities that follow from the definition of the constructor and the IH for each of the  $a$ 's that has type  $Q$ , if there are any>. We must show <...and here we restate  $P$ >.

<go on and prove  $P$  to finish the case...>

- <other cases similarly...>  $\square$

*Example*

- *Theorem:* The  $\leq$  relation is transitive – i.e., for all numbers  $n, m$ , and  $o$ , if  $n \leq m$  and  $m \leq o$ , then  $n \leq o$ .

*Proof:* By induction on a derivation of  $m \leq o$ .

- Suppose the final rule used to show  $\mathbf{m} \leq o$  is  $\mathbf{le\_n}$ . Then  $\mathbf{m} = o$  and we must show that  $\mathbf{n} \leq \mathbf{m}$ , which is immediate by hypothesis.
- Suppose the final rule used to show  $\mathbf{m} \leq o$  is  $\mathbf{le\_S}$ . Then  $o = \mathbf{S } o'$  for some  $o'$  with  $\mathbf{m} \leq o'$ . We must show that  $\mathbf{n} \leq \mathbf{S } o'$ . By induction hypothesis,  $\mathbf{n} \leq o'$ . But then, by  $\mathbf{le\_S}$ ,  $\mathbf{n} \leq \mathbf{S } o'$ .  $\square$

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500(*Wed, 17Feb*2016)

# Chapter 12

## Library SfLib

### 12.1 SfLib: Software Foundations Library

Here we collect together a few useful definitions from earlier chapters that are not provided as part of the Coq standard library. Later chapters will `Import` or `Export` just this file, instead of cluttering the top-level environment with all the examples and false starts in those files.

Definition `admit {T: Type} : T. Admitted.`

```
Tactic Notation "solve_by_inversion_step" tactic(t) :=  
  match goal with  
  | H : _  $\vdash$  _  $\Rightarrow$  solve [ inversion H; subst; t ]  
  end  
|| fail "because the goal is not solvable by inversion."
```

```
Tactic Notation "solve" "by" "inversion" "1" :=  
  solve_by_inversion_step idtac.
```

```
Tactic Notation "solve" "by" "inversion" "2" :=  
  solve_by_inversion_step (solve by inversion 1).
```

```
Tactic Notation "solve" "by" "inversion" "3" :=  
  solve_by_inversion_step (solve by inversion 2).
```

```
Tactic Notation "solve" "by" "inversion" :=  
  solve by inversion 1.
```

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500(*Wed, 17Feb2016*)

# Chapter 13

## Library Rel

### 13.1 Rel: Properties of Relations

Require Export IndProp.

This short (and optional) chapter develops some basic definitions and a few theorems about binary relations in Coq. The key definitions are repeated where they are actually used (in the **Smallstep** chapter), so readers who are already comfortable with these ideas can safely skim or skip this chapter. However, relations are also a good source of exercises for developing facility with Coq's basic reasoning facilities, so it may be useful to look at this material just after the **IndProp** chapter.

A binary *relation* on a set  $X$  is a family of propositions parameterized by two elements of  $X$  – i.e., a proposition about pairs of elements of  $X$ .

**Definition** `relation (X: Type) := X → X → Prop`.

Confusingly, the Coq standard library hijacks the generic term "relation" for this specific instance of the idea. To maintain consistency with the library, we will do the same. So, henceforth the Coq identifier `relation` will always refer to a binary relation between some set and itself, whereas the English word "relation" can refer either to the specific Coq concept or the more general concept of a relation between any number of possibly different sets. The context of the discussion should always make clear which is meant.

An example relation on `nat` is `le`, the less-than-or-equal-to relation, which we usually write  $n1 \leq n2$ .

Print `le`.

Check `le : nat → nat → Prop`.

Check `le : relation nat`.

(Why did we write it this way instead of starting with `Inductive le : relation nat...`? Because we wanted to put the first `nat` to the left of the `:`, which makes Coq generate a somewhat nicer induction principle for reasoning about `≤`.)

## 13.2 Basic Properties

As anyone knows who has taken an undergraduate discrete math course, there is a lot to be said about relations in general, including ways of classifying relations (as reflexive, transitive, etc.), theorems that can be proved generically about certain sorts of relations, constructions that build one relation from another, etc. For example...

### Partial Functions

A relation  $R$  on a set  $X$  is a *partial function* if, for every  $x$ , there is at most one  $y$  such that  $R\ x\ y$  – i.e.,  $R\ x\ y1$  and  $R\ x\ y2$  together imply  $y1 = y2$ .

**Definition** `partial_function`  $\{X: \text{Type}\}$  ( $R$ : relation  $X$ ) :=

$\forall\ x\ y1\ y2 : X, R\ x\ y1 \rightarrow R\ x\ y2 \rightarrow y1 = y2$ .

For example, the `next_nat` relation defined earlier is a partial function.

Print `next_nat`.

Check `next_nat` : relation `nat`.

Theorem `next_nat_partial_function` :  
    `partial_function next_nat`.

Proof.

```
unfold partial_function.
intros x y1 y2 H1 H2.
inversion H1. inversion H2.
reflexivity. Qed.
```

However, the  $\leq$  relation on numbers is not a partial function. (Assume, for a contradiction, that  $\leq$  is a partial function. But then, since  $0 \leq 0$  and  $0 \leq 1$ , it follows that  $0 = 1$ . This is nonsense, so our assumption was contradictory.)

Theorem `le_not_a_partial_function` :  
     $\neg$  (`partial_function le`).

Proof.

```
unfold not. unfold partial_function. intros Hc.
assert (0 = 1) as Nonsense. {
  apply Hc with (x := 0).
  - apply le_n.
  - apply le_S. apply le_n. }
inversion Nonsense. Qed.
```

**Exercise: 2 stars, optional** Show that the *total\_relation* defined in earlier is not a partial function.

□



**Exercise: 2 stars, optional** Show that the *empty\_relation* that we defined earlier is a partial function.

□

## Reflexive Relations

A *reflexive* relation on a set  $X$  is one for which every element of  $X$  is related to itself.

Definition `reflexive`  $\{X: \text{Type}\} (R: \text{relation } X) :=$

$\forall a : X, R \ a \ a.$

Theorem `le_reflexive` :

`reflexive` **le**.

Proof.

`unfold reflexive. intros n. apply le_n. Qed.`

## Transitive Relations

A relation  $R$  is *transitive* if  $R \ a \ c$  holds whenever  $R \ a \ b$  and  $R \ b \ c$  do.

Definition `transitive`  $\{X: \text{Type}\} (R: \text{relation } X) :=$

$\forall a \ b \ c : X, (R \ a \ b) \rightarrow (R \ b \ c) \rightarrow (R \ a \ c).$

Theorem `le_trans` :

`transitive` **le**.

Proof.

`intros n m o Hnm Hmo.`

`induction Hmo.`

`- apply Hnm.`

`- apply le_S. apply IHHmo. Qed.`

Theorem `lt_trans`:

`transitive` `lt`.

Proof.

`unfold lt. unfold transitive.`

`intros n m o Hnm Hmo.`

`apply le_S in Hnm.`

`apply le_trans with (a := (S n)) (b := (S m)) (c := o).`

`apply Hnm.`

`apply Hmo. Qed.`

**Exercise: 2 stars, optional** We can also prove `lt_trans` more laboriously by induction, without using `le_trans`. Do this.

Theorem `lt_trans'` :

`transitive` `lt`.

Proof.

```

unfold lt. unfold transitive.
intros n m o Hnm Hmo.
induction Hmo as [| m' Hm'o].
  Admitted.
□

```

**Exercise: 2 stars, optional** Prove the same thing again by induction on  $o$ .

Theorem `lt_trans''` :

transitive `lt`.

Proof.

```

unfold lt. unfold transitive.
intros n m o Hnm Hmo.
induction o as [| o'].
  Admitted.
□

```

The transitivity of **le**, in turn, can be used to prove some facts that will be useful later (e.g., for the proof of antisymmetry below)...

Theorem `le_Sn_le` :  $\forall n m, \text{S } n \leq m \rightarrow n \leq m$ .

Proof.

```

intros n m H. apply le_trans with (S n).
- apply le_S. apply le_n.
- apply H.

```

Qed.

**Exercise: 1 star, optional** Theorem `le_S_n` :  $\forall n m, (\text{S } n \leq \text{S } m) \rightarrow (n \leq m)$ .

Proof.

```

Admitted.
□

```

**Exercise: 2 stars, optional (`le_Sn_n_inf`)** Provide an informal proof of the following theorem:

Theorem: For every  $n$ ,  $\neg(\text{S } n \leq n)$

A formal proof of this is an optional exercise below, but try writing an informal proof without doing the formal proof first.

Proof:  $\square$

**Exercise: 1 star, optional** Theorem `le_Sn_n` :  $\forall n, \neg(\text{S } n \leq n)$ .

Proof.

```

Admitted.

```

□

Reflexivity and transitivity are the main concepts we'll need for later chapters, but, for a bit of additional practice working with relations in Coq, let's look at a few other common ones...

## Symmetric and Antisymmetric Relations

A relation  $R$  is *symmetric* if  $R\ a\ b$  implies  $R\ b\ a$ .

Definition symmetric  $\{X:\text{Type}\}\ (R:\text{relation } X) :=$   
 $\forall\ a\ b : X, (R\ a\ b) \rightarrow (R\ b\ a).$

**Exercise: 2 stars, optional** Theorem le\_not\_symmetric :  
 $\neg (\text{symmetric le}).$

Proof.

*Admitted.*

□

A relation  $R$  is *antisymmetric* if  $R\ a\ b$  and  $R\ b\ a$  together imply  $a = b$  – that is, if the only "cycles" in  $R$  are trivial ones.

Definition antisymmetric  $\{X:\text{Type}\}\ (R:\text{relation } X) :=$   
 $\forall\ a\ b : X, (R\ a\ b) \rightarrow (R\ b\ a) \rightarrow a = b.$

**Exercise: 2 stars, optional** Theorem le\_antisymmetric :  
antisymmetric le.

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional** Theorem le\_step :  $\forall\ n\ m\ p,$   
 $n < m \rightarrow$   
 $m \leq S\ p \rightarrow$   
 $n \leq p.$

Proof.

*Admitted.*

□

## Equivalence Relations

A relation is an *equivalence* if it's reflexive, symmetric, and transitive.

Definition equivalence  $\{X:\text{Type}\}\ (R:\text{relation } X) :=$   
 $(\text{reflexive } R) \wedge (\text{symmetric } R) \wedge (\text{transitive } R).$

## Partial Orders and Preorders

A relation is a *partial order* when it's reflexive, *anti*-symmetric, and transitive. In the Coq standard library it's called just "order" for short.

Definition order  $\{X:\text{Type}\}$  ( $R$ : relation  $X$ ) :=  
 (reflexive  $R$ )  $\wedge$  (antisymmetric  $R$ )  $\wedge$  (transitive  $R$ ).

A preorder is almost like a partial order, but doesn't have to be antisymmetric.

Definition preorder  $\{X:\text{Type}\}$  ( $R$ : relation  $X$ ) :=  
 (reflexive  $R$ )  $\wedge$  (transitive  $R$ ).

Theorem le\_order :  
 order **le**.

Proof.

```

unfold order. split.
- apply le_reflexive.
- split.
  + apply le_antisymmetric.
  + apply le_trans. Qed.
```

## 13.3 Reflexive, Transitive Closure

The *reflexive, transitive closure* of a relation  $R$  is the smallest relation that contains  $R$  and that is both reflexive and transitive. Formally, it is defined like this in the Relations module of the Coq standard library:

Inductive clos\_refl\_trans  $\{A:\text{Type}\}$  ( $R$ : relation  $A$ ) : relation  $A$  :=  
 | rt\_step :  $\forall x y, R x y \rightarrow \text{clos\_refl\_trans } R x y$   
 | rt\_refl :  $\forall x, \text{clos\_refl\_trans } R x x$   
 | rt\_trans :  $\forall x y z,$   
      $\text{clos\_refl\_trans } R x y \rightarrow$   
      $\text{clos\_refl\_trans } R y z \rightarrow$   
      $\text{clos\_refl\_trans } R x z.$

For example, the reflexive and transitive closure of the **next\_nat** relation coincides with the **le** relation.

Theorem next\_nat\_closure\_is\_le :  $\forall n m,$   
 ( $n \leq m$ )  $\leftrightarrow$  ((clos\_refl\_trans next\_nat)  $n m$ ).

Proof.

```

intros n m. split.
-
  intro H. induction H.
  + apply rt_refl.
  +
```

```

    apply rt_trans with m. apply IHle. apply rt_step.
    apply nn.
-
  intro H. induction H.
+ inversion H. apply le_S. apply le_n.
+ apply le_n.
+
  apply le_trans with y.
  apply IHclos_refl_trans1.
  apply IHclos_refl_trans2. Qed.

```

The above definition of reflexive, transitive closure is natural: it says, explicitly, that the reflexive and transitive closure of  $R$  is the least relation that includes  $R$  and that is closed under rules of reflexivity and transitivity. But it turns out that this definition is not very convenient for doing proofs, since the "nondeterminism" of the `rt_trans` rule can sometimes lead to tricky inductions.

Here is a more useful definition:

```

Inductive clos_refl_trans_1n {A : Type}
  (R : relation A) (x : A)
  : A → Prop :=
| rtln_refl : clos_refl_trans_1n R x x
| rtln_trans (y z : A) :
  R x y → clos_refl_trans_1n R y z →
  clos_refl_trans_1n R x z.

```

Our new definition of reflexive, transitive closure "bundles" the `rt_step` and `rt_trans` rules into the single rule `step`. The left-hand premise of this step is a single use of  $R$ , leading to a much simpler induction principle.

Before we go on, we should check that the two definitions do indeed define the same relation...

First, we prove two lemmas showing that `clos_refl_trans_1n` mimics the behavior of the two "missing" `clos_refl_trans` constructors.

```

Lemma rsc_R : ∀ (X:Type) (R:relation X) (x y : X),
  R x y → clos_refl_trans_1n R x y.

```

Proof.

```

  intros X R x y H.
  apply rtln_trans with y. apply H. apply rtln_refl. Qed.

```

**Exercise: 2 stars, optional (rsc\_trans)** Lemma `rsc_trans` :

```

  ∀ (X:Type) (R: relation X) (x y z : X),
    clos_refl_trans_1n R x y →
    clos_refl_trans_1n R y z →
    clos_refl_trans_1n R x z.

```

Proof.

*Admitted.*

□

Then we use these facts to prove that the two definitions of reflexive, transitive closure do indeed define the same relation.

**Exercise: 3 stars, optional (rtc\_rsc\_coincide)** Theorem `rtc_rsc_coincide` :

$\forall (X:\text{Type}) (R:\text{relation } X) (x\ y : X),$

`clos_refl_trans`  $R\ x\ y \leftrightarrow$  `clos_refl_trans_1n`  $R\ x\ y$ .

Proof.

*Admitted.*

□

*Date* : 2016 – 03 – 04 09 : 33 : 20 – 0500(*Fri*, 04*Mar*2016)

# Chapter 14

## Library Imp

### 14.1 Imp: Simple Imperative Programs

In this chapter, we begin a new direction that will continue for the rest of the course. Up to now most of our attention has been focused on various aspects of Coq itself, while from now on we'll mostly be using Coq to formalize other things. (We'll continue to pause from time to time to introduce a few additional aspects of Coq.)

Our first case study is a *simple imperative programming language* called Imp, embodying a tiny core fragment of conventional mainstream languages such as C and Java. Here is a familiar mathematical function written in Imp.

```
Z ::= X;; Y ::= 1;; WHILE not (Z = 0) DO Y ::= Y * Z;; Z ::= Z - 1 END
```

This chapter looks at how to define the *syntax* and *semantics* of Imp; the chapters that follow develop a theory of *program equivalence* and introduce *Hoare Logic*, a widely used logic for reasoning about imperative programs.

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Lists.List.
Import ListNotations.

Require Import Maps.
Require Import SfLib.
```

### 14.2 Arithmetic and Boolean Expressions

We'll present Imp in three parts: first a core language of *arithmetic and boolean expressions*, then an extension of these expressions with *variables*, and finally a language of *commands* including assignment, conditions, sequencing, and loops.

## 14.2.1 Syntax

Module AEXP.

These two definitions specify the *abstract syntax* of arithmetic and boolean expressions.

Inductive **aexp** : Type :=

```
| ANum : nat → aexp
| APlus : aexp → aexp → aexp
| AMinus : aexp → aexp → aexp
| AMult : aexp → aexp → aexp.
```

Inductive **bexp** : Type :=

```
| BTrue : bexp
| BFalse : bexp
| BEq : aexp → aexp → bexp
| BLe : aexp → aexp → bexp
| BNot : bexp → bexp
| BAnd : bexp → bexp → bexp.
```

In this chapter, we'll elide the translation from the concrete syntax that a programmer would actually write to these abstract syntax trees – the process that, for example, would translate the string "1+2\*3" to the AST `APlus (ANum 1) (AMult (ANum 2) (ANum 3))`. The optional chapter `ImpParser` develops a simple implementation of a lexical analyzer and parser that can perform this translation. You do *not* need to understand that chapter to understand this one, but if you haven't taken a course where these techniques are covered (e.g., a compilers course) you may want to skim it.

For comparison, here's a conventional BNF (Backus-Naur Form) grammar defining the same abstract syntax:  $a ::= \text{nat} \mid a + a \mid a - a \mid a * a$

$b ::= \text{true} \mid \text{false} \mid a = a \mid a \leq a \mid \text{not } b \mid b \text{ and } b$

Compared to the Coq version above...

- The BNF is more informal – for example, it gives some suggestions about the surface syntax of expressions (like the fact that the addition operation is written `+` and is an infix symbol) while leaving other aspects of lexical analysis and parsing (like the relative precedence of `+`, `-`, and `*`, the use of parens to explicitly group subexpressions, etc.) unspecified. Some additional information (and human intelligence) would be required to turn this description into a formal definition, for example when implementing a compiler).

The Coq version consistently omits all this information and concentrates on the abstract syntax only.

- On the other hand, the BNF version is lighter and easier to read. Its informality makes it flexible, which is a huge advantage in situations like discussions at the blackboard, where conveying general ideas is more important than getting every detail nailed down precisely.



Indeed, there are dozens of BNF-like notations and people switch freely among them, usually without bothering to say which form of BNF they're using because there is no need to: a rough-and-ready informal understanding is all that's important.

It's good to be comfortable with both sorts of notations: informal ones for communicating between humans and formal ones for carrying out implementations and proofs.

### 14.2.2 Evaluation

*Evaluating* an arithmetic expression produces a number.

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) × (aeval a2)
  end.
```

Example test\_aeval1:

```
aeval (APlus (ANum 2) (ANum 2)) = 4.
```

Proof. reflexivity. Qed.

Similarly, evaluating a boolean expression yields a boolean.

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ beq_nat (aeval a1) (aeval a2)
  | BLe a1 a2 ⇒ leb (aeval a1) (aeval a2)
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

### 14.2.3 Optimization

We haven't defined very much yet, but we can already get some mileage out of the definitions. Suppose we define a function that takes an arithmetic expression and slightly simplifies it, changing every occurrence of 0+e (i.e., (APlus (ANum 0) e) into just e.

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒
    ANum n
  | APlus (ANum 0) e2 ⇒
```

```

      optimize_0plus e2
| APlus e1 e2 ⇒
  APlus (optimize_0plus e1) (optimize_0plus e2)
| AMinus e1 e2 ⇒
  AMinus (optimize_0plus e1) (optimize_0plus e2)
| AMult e1 e2 ⇒
  AMult (optimize_0plus e1) (optimize_0plus e2)
end.

```

To make sure our optimization is doing the right thing we can test it on some examples and see if the output looks OK.

Example test\_optimize\_0plus:

```

optimize_0plus (APlus (ANum 2)
                    (APlus (ANum 0)
                          (APlus (ANum 0) (ANum 1))))
= APlus (ANum 2) (ANum 1).

```

Proof. reflexivity. Qed.

But if we want to be sure the optimization is correct – i.e., that evaluating an optimized expression gives the same result as the original – we should prove it.

Theorem optimize\_0plus\_sound:  $\forall a,$   
 $\text{aeval } (\text{optimize\_0plus } a) = \text{aeval } a.$

Proof.

```

intros a. induction a.
- reflexivity.
- destruct a1.
  + destruct n.
    × simpl. apply IHa2.
    × simpl. rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity. Qed.

```

## 14.3 Coq Automation

The repetition in this last proof is starting to be a little annoying. If either the language of arithmetic expressions or the optimization being proved sound were significantly more complex, it would begin to be a real problem.

So far, we've been doing all our proofs using just a small handful of Coq's tactics and completely ignoring its powerful facilities for constructing parts of proofs automatically. This section introduces some of these facilities, and we will see more over the next several chapters. Getting used to them will take some energy – Coq's automation is a power tool – but it will allow us to scale up our efforts to more complex definitions and more interesting properties without becoming overwhelmed by boring, repetitive, low-level details.

### 14.3.1 Tacticals

*Tacticals* is Coq's term for tactics that take other tactics as arguments – "higher-order tactics," if you will.

#### The `try` Tactical

If  $T$  is a tactic, then `try  $T$`  is a tactic that is just like  $T$  except that, if  $T$  fails, `try  $T$`  *successfully* does nothing at all (instead of failing).

Theorem `silly1` :  $\forall ae, aeval\ ae = aeval\ ae$ .

Proof. `try reflexivity. Qed.`

Theorem `silly2` :  $\forall (P : Prop), P \rightarrow P$ .

Proof.

`intros P HP.`

`try reflexivity.    apply HP. Qed.`

There is no real reason to use `try` in completely manual proofs like these, but we'll see below that it is very useful for doing automated proofs in conjunction with the `; tactical`.

#### The `; Tactical` (Simple Form)

In its most common form, the `; tactical` takes two tactics as arguments. The compound tactic  $T;T'$  first performs  $T$  and then performs  $T'$  on *each subgoal* generated by  $T$ .

For example, consider the following trivial lemma:

Lemma `foo` :  $\forall n, leb\ 0\ n = true$ .

Proof.

`intros.`

`destruct n.`

`- simpl. reflexivity.`

`- simpl. reflexivity.`

`Qed.`

We can simplify this proof using the `; tactical`:

Lemma `foo'` :  $\forall n$ , `leb 0 n = true`.

Proof.

```
intros.
destruct n;

simpl;

reflexivity.
Qed.
```

Using `try` and `; together`, we can get rid of the repetition in the proof that was bothering us a little while ago.

Theorem `optimize_0plus_sound'`:  $\forall a$ ,  
`aeval (optimize_0plus a) = aeval a`.

Proof.

```
intros a.
induction a;

  try (simpl; rewrite IHa1; rewrite IHa2; reflexivity).
- reflexivity.
-
  destruct a1;

  try (simpl; simpl in IHa1; rewrite IHa1;
    rewrite IHa2; reflexivity).
+ destruct n;
  simpl; rewrite IHa2; reflexivity. Qed.
```

Coq experts often use this “...; try...” idiom after a tactic like `induction` to take care of many similar cases all at once. Naturally, this practice has an analog in informal proofs.

Here is an informal proof of this theorem that matches the structure of the formal one:

*Theorem:* For all arithmetic expressions `a`, `aeval (optimize_0plus a) = aeval a`. *Proof:* By induction on `a`. Most cases follow directly from the IH. The remaining cases are as follows:

- Suppose `a = ANum n` for some `n`. We must show `aeval (optimize_0plus (ANum n)) = aeval (ANum n)`. This is immediate from the definition of `optimize_0plus`.
- Suppose `a = APlus a1 a2` for some `a1` and `a2`. We must show `aeval (optimize_0plus (APlus a1 a2)) = aeval (APlus a1 a2)`. Consider the possible forms of `a1`. For most of them, `optimize_0plus` simply calls itself recursively for the subexpressions and rebuilds a new expression of the same form as `a1`; in these cases, the result follows directly from the IH.

The interesting case is when  $a1 = \text{ANum } n$  for some  $n$ . If  $n = \text{ANum } 0$ , then `optimize_0plus (APlus a1 a2) = optimize_0plus a2` and the IH for  $a2$  is exactly what we need. On the other hand, if  $n = \text{S } n'$  for some  $n'$ , then again `optimize_0plus` simply calls itself recursively, and the result follows from the IH.  $\square$

This proof can still be improved: the first case (for  $a = \text{ANum } n$ ) is very trivial – even more trivial than the cases that we said simply followed from the IH – yet we have chosen to write it out in full. It would be better and clearer to drop it and just say, at the top, “Most cases are either immediate or direct from the IH. The only interesting case is the one for `APlus...`” We can make the same improvement in our formal proof too. Here’s how it looks:

**Theorem** `optimize_0plus_sound''`:  $\forall a,$

`aeval (optimize_0plus a) = aeval a.`

**Proof.**

`intros a.`

`induction a;`

`try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);`

`try reflexivity.`

`-`

`destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;  
rewrite IHa2; reflexivity).`

`+ destruct n;`

`simpl; rewrite IHa2; reflexivity. Qed.`

## The ; Tactical (General Form)

The `;` tactical also has a more general form than the simple  $T;T'$  we’ve seen above. If  $T, T1, \dots, Tn$  are tactics, then `T; T1 | T2 | ... | Tn` is a tactic that first performs  $T$  and then performs  $T1$  on the first subgoal generated by  $T$ , performs  $T2$  on the second subgoal, etc.

So  $T;T'$  is just special notation for the case when all of the  $Ti$ ’s are the same tactic – i.e.,  $T;T'$  is shorthand for: `T; T' | T' | ... | T'`

## The repeat Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails. Here is an example showing that 10 is in a long list using `repeat`.

**Theorem** `ln10` : `ln 10 [1;2;3;4;5;6;7;8;9;10].`

**Proof.**

`repeat (try (left; reflexivity); right).`

`Qed.`

The `repeat T` tactic never fails: if the tactic  $T$  doesn’t apply to the original goal, then `repeat` still succeeds without changing the original goal (i.e., it repeats zero times).

Theorem `ln10' : ln 10 [1;2;3;4;5;6;7;8;9;10]`.

Proof.

`repeat (left; reflexivity).`

`repeat (right; try (left; reflexivity)).`

Qed.

The `repeat T` tactic also does not have any upper bound on the number of times it applies  $T$ . If  $T$  is a tactic that always succeeds, then `repeat T` will loop forever (e.g., `repeat simpl` loops forever, since `simpl` always succeeds). While Coq's term language is guaranteed to terminate, Coq's tactic language is not!

**Exercise: 3 stars (optimize\_0plus\_b)** Since the `optimize_0plus` transformation doesn't change the value of `aexps`, we should be able to apply it to all the `aexps` that appear in a `bexp` without changing the `bexp`'s value. Write a function which performs that transformation on `bexps`, and prove it is sound. Use the tacticals we've just seen to make the proof as elegant as possible.

Fixpoint `optimize_0plus_b (b : bexp) : bexp :=`  
`admit.`

Theorem `optimize_0plus_b_sound : ∀ b,`  
`beval (optimize_0plus_b b) = beval b.`

Proof.

*Admitted.*

□

**Exercise: 4 stars, optional (optimizer)** *Design exercise:* The optimization implemented by our `optimize_0plus` function is only one of many possible optimizations on arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct.

□

## 14.3.2 Defining New Tactic Notations

Coq also provides several ways of "programming" tactic scripts.

- The **Tactic Notation** idiom illustrated below gives a handy way to define "shorthand tactics" that bundle several tactics into a single command.
- For more sophisticated programming, Coq offers a small built-in programming language called **Ltac** with primitives that can examine and modify the proof state. The details are a bit too complicated to get into here (and it is generally agreed that **Ltac** is not the most beautiful part of Coq's design!), but they can be found in the reference manual and other books on Coq, and there are many examples of **Ltac** definitions in the Coq standard library that you can use as examples.

- There is also an OCaml API, which can be used to build tactics that access Coq's internal structures at a lower level, but this is seldom worth the trouble for ordinary Coq users.

The **Tactic Notation** mechanism is the easiest to come to grips with, and it offers plenty of power for many purposes. Here's an example.

```
Tactic Notation "simpl_and_try" tactic(c) :=
  simpl;
  try c.
```

This defines a new tactical called *simpl\_and\_try* that takes one tactic *c* as an argument and is defined to be equivalent to the tactic `simpl; try c`. For example, writing "*simpl\_and\_try reflexivity*." in a proof would be the same as writing "`simpl; try reflexivity`."

### 14.3.3 The omega Tactic

The **omega** tactic implements a decision procedure for a subset of first-order logic called *Presburger arithmetic*. It is based on the Omega algorithm invented in 1992 by William Pugh.

If the goal is a universally quantified formula made out of

- numeric constants, addition (+ and `S`), subtraction (- and `pred`), and multiplication by constants (this is what makes it Presburger arithmetic),
- equality (= and  $\neq$ ) and inequality ( $\leq$ ), and
- the logical connectives  $\wedge$ ,  $\vee$ ,  $\neg$ , and  $\rightarrow$ ,

then invoking **omega** will either solve the goal or tell you that it is actually false.

```
Require Import Coq.omega.Omega.
```

```
Example silly_presburger_example :  $\forall m\ n\ o\ p,$ 
```

```
   $m + n \leq n + o \wedge o + 3 = p + 3 \rightarrow$ 
```

```
   $m \leq p.$ 
```

```
Proof.
```

```
  intros. omega.
```

```
Qed.
```

Leibniz wrote, "It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could be relegated to anyone else if machines were used." We recommend that excellent people of all genders use the omega tactic whenever possible.

### 14.3.4 A Few More Handy Tactics

Finally, here are some miscellaneous tactics that you may find convenient.

- **clear**  $H$ : Delete hypothesis  $H$  from the context.
- **subst**  $x$ : Find an assumption  $x = e$  or  $e = x$  in the context, replace  $x$  with  $e$  throughout the context and current goal, and clear the assumption.
- **subst**: Substitute away *all* assumptions of the form  $x = e$  or  $e = x$ .
- **rename... into...**: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named  $x$ , then **rename**  $x$  *into*  $y$  will change all occurrences of  $x$  to  $y$ .
- **assumption**: Try to find a hypothesis  $H$  in the context that exactly matches the goal; if one is found, behave just like **apply**  $H$ .
- **contradiction**: Try to find a hypothesis  $H$  in the current context that is logically equivalent to **False**. If one is found, solve the goal.
- **constructor**: Try to find a constructor  $c$  (from some **Inductive** definition in the current environment) that can be applied to solve the current goal. If one is found, behave like **apply**  $c$ .

We'll see many examples of these in the proofs below.

## 14.4 Evaluation as a Relation

We have presented **aeval** and **beval** as functions defined by **Fixpoints**. Another way to think about evaluation – one that we will see is often more flexible – is as a *relation* between expressions and their values. This leads naturally to **Inductive** definitions like the following one for arithmetic expressions...

Module AEVALR\_FIRST\_TRY.

```
Inductive aevalR : aexp → nat → Prop :=
| E_ANum : ∀ (n: nat),
  aevalR (ANum n) n
| E_APlus : ∀ (e1 e2: aexp) (n1 n2: nat),
  aevalR e1 n1 →
  aevalR e2 n2 →
  aevalR (APlus e1 e2) (n1 + n2)
| E_AMinus : ∀ (e1 e2: aexp) (n1 n2: nat),
  aevalR e1 n1 →
  aevalR e2 n2 →
```



```

    aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult : ∀ (e1 e2: aexp) (n1 n2: nat),
    aevalR e1 n1 →
    aevalR e2 n2 →
    aevalR (AMult e1 e2) (n1 × n2).

```

As is often the case with relations, we'll find it convenient to define infix notation for **aevalR**. We'll write  $e \backslash\backslash n$  to mean that arithmetic expression  $e$  evaluates to value  $n$ . (This notation is one place where the limitation to ASCII symbols becomes a little bothersome. The standard notation for the evaluation relation is a double down-arrow. We'll typeset it like this in the HTML version of the notes and use a double slash as the closest approximation in *.v* files.)

```

Notation "e '\!\!' n"
      := (aevalR e n)
      (at level 50, left associativity)
      : type_scope.

```

End AEVALR\_FIRST\_TRY.

In fact, Coq provides a way to use this notation in the definition of **aevalR** itself. This avoids situations where we're working on a proof involving statements in the form  $e \backslash\backslash n$  but we have to refer back to a definition written using the form **aevalR**  $e$   $n$ .

We do this by first "reserving" the notation, then giving the definition together with a declaration of what the notation means.

```

Reserved Notation "e '\!\!' n" (at level 50, left associativity).

```

```

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum : ∀ (n:nat),
    (ANum n) \!\! n
  | E_APlus : ∀ (e1 e2: aexp) (n1 n2 : nat),
    (e1 \!\! n1) → (e2 \!\! n2) → (APlus e1 e2) \!\! (n1 + n2)
  | E_AMinus : ∀ (e1 e2: aexp) (n1 n2 : nat),
    (e1 \!\! n1) → (e2 \!\! n2) → (AMinus e1 e2) \!\! (n1 - n2)
  | E_AMult : ∀ (e1 e2: aexp) (n1 n2 : nat),
    (e1 \!\! n1) → (e2 \!\! n2) → (AMult e1 e2) \!\! (n1 × n2)

```

```

where "e '\!\!' n" := (aevalR e n) : type_scope.

```

### 14.4.1 Inference Rule Notation

In informal discussions, it is convenient to write the rules for **aevalR** and similar relations in the more readable graphical form of *inference rules*, where the premises above the line justify the conclusion below the line (we have already seen them in the Prop chapter).

For example, the constructor **E\_APlus**... | **E\_APlus** : forall (e1 e2: aexp) (n1 n2: nat), aevalR e1 n1 -> aevalR e2 n2 -> aevalR (APlus e1 e2) (n1 + n2) ..would be written like

this as an inference rule:  $e1 \ \backslash\backslash \ n1 \ e2 \ \backslash\backslash \ n2$

---

(E\_APlus) APlus  $e1 \ e2 \ \backslash\backslash \ n1+n2$

Formally, there is nothing deep about inference rules: they are just implications. You can read the rule name on the right as the name of the constructor and read each of the linebreaks between the premises above the line and the line itself as  $\rightarrow$ . All the variables mentioned in the rule ( $e1$ ,  $n1$ , etc.) are implicitly bound by universal quantifiers at the beginning. (Such variables are often called *metavariables* to distinguish them from the variables of the language we are defining. At the moment, our arithmetic expressions don't include variables, but we'll soon be adding them.) The whole collection of rules is understood as being wrapped in an **Inductive** declaration. In informal prose, this is either elided or else indicated by saying something like "Let **aevalR** be the smallest relation closed under the following rules...".

For example,  $\backslash\backslash$  is the smallest relation closed under these rules:

---

(E\_ANum) ANum  $n \ \backslash\backslash \ n$   
 $e1 \ \backslash\backslash \ n1 \ e2 \ \backslash\backslash \ n2$

---

(E\_APlus) APlus  $e1 \ e2 \ \backslash\backslash \ n1+n2$   
 $e1 \ \backslash\backslash \ n1 \ e2 \ \backslash\backslash \ n2$

---

(E\_AMinus) AMinus  $e1 \ e2 \ \backslash\backslash \ n1-n2$   
 $e1 \ \backslash\backslash \ n1 \ e2 \ \backslash\backslash \ n2$

---

(E\_AMult) AMult  $e1 \ e2 \ \backslash\backslash \ n1*n2$

## 14.4.2 Equivalence of the Definitions

It is straightforward to prove that the relational and functional definitions of evaluation agree, for all arithmetic expressions...

**Theorem** `aeval_iff_aevalR` :  $\forall \ a \ n,$

$(a \ \backslash\backslash \ n) \leftrightarrow \text{aeval } a = n.$

**Proof.**

`split.`

-

`intros H.`

`induction H; simpl.`

+

`reflexivity.`

+

`rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.`

+

`rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.`

```

+
  rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
-
  generalize dependent n.
  induction a;
    simpl; intros; subst.
+
  apply E_ANum.
+
  apply E_APlus.
    apply IHa1. reflexivity.
    apply IHa2. reflexivity.
+
  apply E_AMinus.
    apply IHa1. reflexivity.
    apply IHa2. reflexivity.
+
  apply E_AMult.
    apply IHa1. reflexivity.
    apply IHa2. reflexivity.
Qed.

```

We can make the proof quite a bit shorter by making more use of tacticals...

Theorem *aeval\_iff\_aevalR'* :  $\forall a\ n,$

$(a \setminus n) \leftrightarrow \text{aeval } a = n.$

Proof.

```

split.
-
  intros H; induction H; subst; reflexivity.
-
  generalize dependent n.
  induction a; simpl; intros; subst; constructor;
    try apply IHa1; try apply IHa2; reflexivity.

```

Qed.

**Exercise: 3 stars (bevalR)** Write a relation *bevalR* in the same style as *aevalR*, and prove that it is equivalent to *beval*.

□

End AEXP.

### 14.4.3 Computational vs. Relational Definitions

For the definitions of evaluation for arithmetic and boolean expressions, the choice of whether to use functional or relational definitions is mainly a matter of taste. In general, Coq has somewhat better support for working with relations. On the other hand, in some sense function definitions carry more information, because functions are by definition deterministic and defined on all arguments; for a relation we have to show these properties explicitly if we need them. Functions also take advantage of Coq's computation mechanism.

However, there are circumstances where relational definitions of evaluation are preferable to functional ones.

Module AEVALR\_DIVISION.

For example, suppose that we wanted to extend the arithmetic operations by considering also a division operation:

```
Inductive aexp : Type :=
| ANum : nat → aexp
| APlus : aexp → aexp → aexp
| AMinus : aexp → aexp → aexp
| AMult : aexp → aexp → aexp
| ADiv : aexp → aexp → aexp.
```

Extending the definition of `aeval` to handle this new operation would not be straightforward (what should we return as the result of `ADiv (ANum 5) (ANum 0)`?). But extending `aevalR` is straightforward.

```
Reserved Notation "e '\'' n"
(at level 50, left associativity).
```

```
Inductive aevalR : aexp → nat → Prop :=
| E_ANum : ∀ (n:nat),
  (ANum n) '\'' n
| E_APlus : ∀ (a1 a2: aexp) (n1 n2 : nat),
  (a1 '\'' n1) → (a2 '\'' n2) → (APlus a1 a2) '\'' (n1 + n2)
| E_AMinus : ∀ (a1 a2: aexp) (n1 n2 : nat),
  (a1 '\'' n1) → (a2 '\'' n2) → (AMinus a1 a2) '\'' (n1 - n2)
| E_AMult : ∀ (a1 a2: aexp) (n1 n2 : nat),
  (a1 '\'' n1) → (a2 '\'' n2) → (AMult a1 a2) '\'' (n1 × n2)
| E_ADiv : ∀ (a1 a2: aexp) (n1 n2 n3: nat),
  (a1 '\'' n1) → (a2 '\'' n2) → (n2 > 0) →
  (mult n2 n3 = n1) → (ADiv a1 a2) '\'' n3
```

```
where "a '\'' n" := (aevalR a n) : type_scope.
```

```
End AEVALR_DIVISION.
```

```
Module AEVALR_EXTENDED.
```

## Adding Nondeterminism

Suppose, instead, that we want to extend the arithmetic operations by a nondeterministic number generator *any* that, when evaluated, may yield any number. (Note that this is not the same as making a *probabilistic* choice among all possible numbers – we’re not specifying any particular distribution of results, but just saying what results are *possible*.)

Reserved Notation "e '\'' n" (at level 50, left associativity).

```
Inductive aexp : Type :=
| AAny : aexp
| ANum : nat → aexp
| APlus : aexp → aexp → aexp
| AMinus : aexp → aexp → aexp
| AMult : aexp → aexp → aexp.
```

Again, extending `aeval` would be tricky, since now evaluation is *not* a deterministic function from expressions to numbers, but extending `aevalR` is no problem:

```
Inductive aevalR : aexp → nat → Prop :=
| E_Any : ∀ (n:nat),
  AAny \\\ n
| E_ANum : ∀ (n:nat),
  (ANum n) \\\ n
| E_APlus : ∀ (a1 a2: aexp) (n1 n2 : nat),
  (a1 \\\ n1) → (a2 \\\ n2) → (APlus a1 a2) \\\ (n1 + n2)
| E_AMinus : ∀ (a1 a2: aexp) (n1 n2 : nat),
  (a1 \\\ n1) → (a2 \\\ n2) → (AMinus a1 a2) \\\ (n1 - n2)
| E_AMult : ∀ (a1 a2: aexp) (n1 n2 : nat),
  (a1 \\\ n1) → (a2 \\\ n2) → (AMult a1 a2) \\\ (n1 × n2)
```

```
where "a '\'' n" := (aevalR a n) : type_scope.
```

```
End AEVALR-EXTENDED.
```

## 14.5 Expressions With Variables

Let’s turn our attention back to defining `Imp`. The next thing we need to do is to enrich our arithmetic and boolean expressions with variables. To keep things simple, we’ll assume that all variables are global and that they only hold numbers.

### 14.5.1 States

Since we’ll want to look variables up to find out their current values, we’ll reuse the type `id` from the `Maps` chapter for the type of variables in `Imp`.

A *machine state* (or just *state*) represents the current values of *all* variables at some point in the execution of a program.

For simplicity, we assume that the state is defined for *all* variables, even though any given program is only going to mention a finite number of them. The state captures all of the information stored in memory. For Imp programs, because each variable stores a natural number, we can represent the state as a mapping from identifiers to **nat**. For more complex programming languages, the state might have more structure.

Definition `state := total_map nat`.

Definition `empty_state : state :=  
t_empty 0`.

## 14.5.2 Syntax

We can add variables to the arithmetic expressions we had before by simply adding one more constructor:

```
Inductive aexp : Type :=
| ANum : nat → aexp
| Ald : id → aexp
| APlus : aexp → aexp → aexp
| AMinus : aexp → aexp → aexp
| AMult : aexp → aexp → aexp.
```

Defining a few variable names as notational shorthands will make examples easier to read:

Definition `X : id := ld 0`.

Definition `Y : id := ld 1`.

Definition `Z : id := ld 2`.

(This convention for naming program variables (`X`, `Y`, `Z`) clashes a bit with our earlier use of uppercase letters for types. Since we're not using polymorphism heavily in this part of the course, this overloading should not cause confusion.)

The definition of **bexps** is unchanged (except for using the new **aexps**):

```
Inductive bexp : Type :=
| BTrue : bexp
| BFalse : bexp
| BEq : aexp → aexp → bexp
| BLe : aexp → aexp → bexp
| BNot : bexp → bexp
| BAnd : bexp → bexp → bexp.
```

### 14.5.3 Evaluation

The arith and boolean evaluators are extended to handle variables in the obvious way, taking a state as an extra argument:

```
Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | Ald x ⇒ st x
  | APlus a1 a2 ⇒ (aeval st a1) + (aeval st a2)
  | AMinus a1 a2 ⇒ (aeval st a1) - (aeval st a2)
  | AMult a1 a2 ⇒ (aeval st a1) × (aeval st a2)
  end.
```

```
Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ beq_nat (aeval st a1) (aeval st a2)
  | BLe a1 a2 ⇒ leb (aeval st a1) (aeval st a2)
  | BNot b1 ⇒ negb (beval st b1)
  | BAnd b1 b2 ⇒ andb (beval st b1) (beval st b2)
  end.
```

```
Example aexpl :
  aeval (t_update empty_state X 5)
    (APlus (ANum 3) (AMult (Ald X) (ANum 2)))
  = 13.
```

Proof. reflexivity. Qed.

```
Example bexpl :
  beval (t_update empty_state X 5)
    (BAnd BTrue (BNot (BLe (Ald X) (ANum 4))))
  = true.
```

Proof. reflexivity. Qed.

## 14.6 Commands

Now we are ready define the syntax and behavior of Imp *commands* (sometimes called *statements*).

### 14.6.1 Syntax

Informally, commands  $c$  are described by the following BNF grammar. (We choose this slightly awkward concrete syntax for the sake of being able to define Imp syntax using Coq's

Notation mechanism. In particular, we use *IFB* to avoid conflicting with the *if* notation from the standard library.) `c ::= SKIP | x ::= a | c ;; c | IFB b THEN c ELSE c FI | WHILE b DO c END`

For example, here's factorial in Imp: `Z ::= X;; Y ::= 1;; WHILE not (Z = 0) DO Y ::= Y * Z;; Z ::= Z - 1 END` When this command terminates, the variable *Y* will contain the factorial of the initial value of *X*.

Here is the formal definition of the abstract syntax of commands:

```
Inductive com : Type :=
| CSkip : com
| CAss : id → aexp → com
| CSeq : com → com → com
| Clf : bexp → com → com → com
| CWhile : bexp → com → com.
```

As usual, we can use a few `Notation` declarations to make things more readable. To avoid conflicts with Coq's built-in notations, we keep this light – in particular, we don't introduce any notations for *aexprs* and *bexprs* to avoid confusion with the numeric and boolean operators we've already defined.

```
Notation "'SKIP'" :=
  CSkip.
Notation "x ' ::= ' a" :=
  (CAss x a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (Clf c1 c2 c3) (at level 80, right associativity).
```

For example, here is the factorial function again, written as a formal definition to Coq:

```
Definition fact_in_coq : com :=
  Z ::= Ald X;;
  Y ::= ANum 1;;
  WHILE BNot (BEq (Ald Z) (ANum 0)) DO
    Y ::= AMult (Ald Y) (Ald Z);;
    Z ::= AMinus (Ald Z) (ANum 1)
  END.
```

## 14.6.2 Examples

Assignment:

```
Definition plus2 : com :=
  X ::= (APlus (Ald X) (ANum 2)).
```



```
Definition XtimesYinZ : com :=
  Z ::= (AMult (Ald X) (Ald Y)).
```

```
Definition subtract_slowly_body : com :=
  Z ::= AMinus (Ald Z) (ANum 1) ;;
  X ::= AMinus (Ald X) (ANum 1).
```

## Loops

```
Definition subtract_slowly : com :=
  WHILE BNot (BEq (Ald X) (ANum 0)) DO
    subtract_slowly_body
  END.
```

```
Definition subtract_3_from_5_slowly : com :=
  X ::= ANum 3 ;;
  Z ::= ANum 5 ;;
  subtract_slowly.
```

## An infinite loop:

```
Definition loop : com :=
  WHILE BTrue DO
    SKIP
  END.
```

## 14.7 Evaluation

Next we need to define what it means to evaluate an Imp command. The fact that *WHILE* loops don't necessarily terminate makes defining an evaluation function tricky...

### 14.7.1 Evaluation as a Function (Failed Attempt)

Here's an attempt at defining an evaluation function for commands, omitting the *WHILE* case.

```
Fixpoint ceval_fun_no_while (st : state) (c : com)
  : state :=
  match c with
  | SKIP =>
    st
  | x ::= a1 =>
    t_update st x (aeval st a1)
  | c1 ;; c2 =>
```

```

    let st' := ceval_fun_no_while st c1 in
    ceval_fun_no_while st' c2
  | IFB b THEN c1 ELSE c2 FI =>
    if (beval st b)
    then ceval_fun_no_while st c1
    else ceval_fun_no_while st c2
  | WHILE b DO c END =>
    st
end.

```

In a traditional functional programming language like OCaml or Haskell we could add the *WHILE* case as follows:

```

Fixpoint ceval_fun (st : state) (c : com) : state :=
  match c with
  ...
  | WHILE b DO c END =>
    if (beval st b)
    then ceval_fun st (c; WHILE b DO c END)
    else st
  end.

```

Coq doesn't accept such a definition ("Error: Cannot guess decreasing argument of fix") because the function we want to define is not guaranteed to terminate. Indeed, it *doesn't* always terminate: for example, the full version of the *ceval\_fun* function applied to the *loop* program above would never terminate. Since Coq is not just a functional programming language, but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an (invalid!) Coq program showing what would go wrong if Coq allowed non-terminating recursive functions:

```

Fixpoint loop_false (n : nat) : False := loop_false n.

```

That is, propositions like **False** would become provable (e.g., *loop\_false* 0 would be a proof of **False**), which would be a disaster for Coq's logical consistency.

Thus, because it doesn't terminate on all inputs, the full version of *ceval\_fun* cannot be written in Coq – at least not without additional tricks (see chapter `ImpCEvalFun` if you're curious about what those might be).

## 14.7.2 Evaluation as a Relation

Here's a better way: define **ceval** as a *relation* rather than a *function* – i.e., define it in `Prop` instead of `Type`, as we did for **aevalR** above.

This is an important change. Besides freeing us from the awkward workarounds that would be needed to define evaluation as a function, it gives us a lot more flexibility in the definition. For example, if we add concurrency features like *any* to the language, we want

the definition of evaluation to be nondeterministic – i.e., not only will it not be total, it will not even be a function! We'll use the notation  $c / st \Downarrow st'$  for our **ceval** relation:  $c / st \Downarrow st'$  means that executing program  $c$  in a starting state  $st$  results in an ending state  $st'$ . This can be pronounced "c takes state  $st$  to  $st'$ ".

## Operational Semantics

Here is an informal definition of evaluation, presented as inference rules for the sake of readability:

(E_Skip) SKIP / $st \Downarrow st$ aeval $st$ $a1 = n$
(E_Ass) $x := a1 / st \Downarrow (t\_update\ st\ x\ n)$ $c1 / st \Downarrow st' \ c2 / st' \Downarrow st''$
(E_Seq) $c1;;c2 / st \Downarrow st''$ beval $st\ b1 = true \ c1 / st \Downarrow st'$
(E_IfTrue) IF $b1$ THEN $c1$ ELSE $c2$ FI / $st \Downarrow st'$ beval $st\ b1 = false \ c2 / st \Downarrow st'$
(E_IfFalse) IF $b1$ THEN $c1$ ELSE $c2$ FI / $st \Downarrow st'$ beval $st\ b = false$
(E_WhileEnd) WHILE $b$ DO $c$ END / $st \Downarrow st$ beval $st\ b = true \ c / st \Downarrow st' \ WHILE\ b\ DO\ c\ END / st' \Downarrow st''$
(E_WhileLoop) WHILE $b$ DO $c$ END / $st \Downarrow st''$

Here is the formal definition. Make sure you understand how it corresponds to the inference rules.

Reserved Notation " $c1 \text{ '}' st \text{ '}' \Downarrow st'$ "  
(at level 40,  $st$  at level 39).

Inductive **ceval** : **com**  $\rightarrow$  state  $\rightarrow$  state  $\rightarrow$  Prop :=

- | E\_Skip :  $\forall st,$   
SKIP /  $st \Downarrow st$
- | E\_Ass :  $\forall st\ a1\ n\ x,$   
aeval  $st\ a1 = n \rightarrow$   
( $x ::= a1$ ) /  $st \Downarrow (t\_update\ st\ x\ n)$
- | E\_Seq :  $\forall c1\ c2\ st\ st'\ st'',$   
 $c1 / st \Downarrow st' \rightarrow$   
 $c2 / st' \Downarrow st'' \rightarrow$

```

      (c1 ;; c2) / st \\\ st''
| E_IfTrue : ∀ st st' b c1 c2,
  beval st b = true →
  c1 / st \\\ st' →
  (IFB b THEN c1 ELSE c2 FI) / st \\\ st'
| E_IfFalse : ∀ st st' b c1 c2,
  beval st b = false →
  c2 / st \\\ st' →
  (IFB b THEN c1 ELSE c2 FI) / st \\\ st'
| E_WhileEnd : ∀ b st c,
  beval st b = false →
  (WHILE b DO c END) / st \\\ st
| E_WhileLoop : ∀ st st' st'' b c,
  beval st b = true →
  c / st \\\ st' →
  (WHILE b DO c END) / st' \\\ st'' →
  (WHILE b DO c END) / st \\\ st''

```

where "c1 '/' st '\\\' st'" := (ceval c1 st st').

The cost of defining evaluation as a relation instead of a function is that we now need to construct *proofs* that some program evaluates to some result state, rather than just letting Coq's computation mechanism do it for us.

Example ceval\_example1:

```

(X ::= ANum 2;;
 IFB BLe (Ald X) (ANum 1)
 THEN Y ::= ANum 3
 ELSE Z ::= ANum 4
 FI)
/ empty_state
\\ (t_update (t_update empty_state X 2) Z 4).

```

Proof.

```

apply E_Seq with (t_update empty_state X 2).
-
  apply E_Ass. reflexivity.
-
  apply E_IfFalse.
  reflexivity.
  apply E_Ass. reflexivity. Qed.

```

**Exercise: 2 stars (ceval\_example2)** Example ceval\_example2:

```

(X ::= ANum 0;; Y ::= ANum 1;; Z ::= ANum 2) / empty_state \\
(t_update (t_update (t_update empty_state X 0) Y 1) Z 2).

```

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced (pup\_to\_n)** Write an Imp program that sums the numbers from 1 to  $X$  (inclusive:  $1 + 2 + \dots + X$ ) in the variable  $Y$ . Prove that this program executes as intended for  $X = 2$  (the latter is trickier than you might expect).

Definition pup\_to\_n : com :=

*admit.*

Theorem pup\_to\_2\_ceval :

pup\_to\_n / (t\_update empty\_state X 2) \\  
 t\_update (t\_update (t\_update (t\_update (t\_update (t\_update empty\_state  
 X 2) Y 0) Y 2) X 1) Y 3) X 0.

Proof.

*Admitted.*

□

### 14.7.3 Determinism of Evaluation

Changing from a computational to a relational definition of evaluation is a good move because it allows us to escape from the artificial requirement that evaluation should be a total function. But it also raises a question: Is the second definition of evaluation really a partial function? Or is it possible that, beginning from the same state  $st$ , we could evaluate some command  $c$  in different ways to reach two different output states  $st'$  and  $st''$ ?

In fact, this cannot happen: **ceval** is a partial function:

Theorem ceval\_deterministic:  $\forall c \ st \ st1 \ st2$ ,

$c / st \ \backslash \backslash \ st1 \rightarrow$   
 $c / st \ \backslash \backslash \ st2 \rightarrow$   
 $st1 = st2$ .

Proof.

```
intros c st st1 st2 E1 E2.
generalize dependent st2.
induction E1;
  intros st2 E2; inversion E2; subst.
- reflexivity.
- reflexivity.
-
  assert (st' = st'0) as EQ1.
  { apply IHE1_1; assumption. }
  subst st'0.
  apply IHE1_2. assumption.
-
```

```

    apply IHE1. assumption.
-
    rewrite H in H5. inversion H5.
-
    rewrite H in H5. inversion H5.
-
    apply IHE1. assumption.
-
    reflexivity.
-
    rewrite H in H2. inversion H2.
-
    rewrite H in H4. inversion H4.
-
    assert (st' = st'0) as EQ1.
    { apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption. Qed.

```

## 14.8 Reasoning About Imp Programs

We'll get much deeper into systematic techniques for reasoning about Imp programs in the following chapters, but we can do quite a bit just working with the bare definitions.

This section explores some examples.

**Theorem** `plus2_spec` :  $\forall st\ n\ st'$ ,

$$\begin{aligned}
 &st\ X = n \rightarrow \\
 &\text{plus2} / st \setminus st' \rightarrow \\
 &st'\ X = n + 2.
 \end{aligned}$$

**Proof.**

```
intros st n st' HX Heval.
```

Inverting *Heval* essentially forces Coq to expand one step of the **ceval** computation – in this case revealing that *st'* must be *st* extended with the new value of *X*, since **plus2** is an assignment

```
inversion Heval. subst. clear Heval. simpl.
apply t_update_eq. Qed.
```

**Exercise: 3 stars, recommended (XtimesYinZ\_spec)** State and prove a specification of `XtimesYinZ`.

□

**Exercise: 3 stars, recommended (loop\_never\_stops)** Theorem `loop_never_stops` :  $\forall st\ st',$   
 $\sim(\text{loop} / st \setminus st').$

Proof.

```
intros st st' contra. unfold loop in contra.
remember (WHILE BTrue DO SKIP END) as loopdef
eqn:Hegloopdef.
```

Proceed by induction on the assumed derivation showing that *loopdef* terminates. Most of the cases are immediately contradictory (and so can be solved in one step with `inversion`).

*Admitted.*

□

**Exercise: 3 stars (no\_whilesR)** Consider the definition of the `no_whiles` boolean predicate below:

```
Fixpoint no_whiles (c : com) : bool :=
  match c with
  | SKIP =>
    true
  | _ ::= _ =>
    true
  | c1 ;; c2 =>
    andb (no_whiles c1) (no_whiles c2)
  | IFB _ THEN ct ELSE cf FI =>
    andb (no_whiles ct) (no_whiles cf)
  | WHILE _ DO _ END =>
    false
  end.
```

This predicate yields `true` just on programs that have no while loops. Using `Inductive`, write a property `no_whilesR` such that `no_whilesR c` is provable exactly when `c` is a program with no while loops. Then prove its equivalence with `no_whiles`.

Inductive `no_whilesR`: `com`  $\rightarrow$  `Prop` :=

.

Theorem `no_whiles_eqv`:

$\forall c, \text{no\_whiles } c = \text{true} \leftrightarrow \text{no\_whilesR } c.$

Proof.

*Admitted.*

□

**Exercise: 4 stars (no\_whiles\_terminating)** Imp programs that don't involve while loops always terminate. State and prove a theorem *no\_whiles\_terminating* that says this.

Use either `no_whiles` or `no_whilesR`, as you prefer.

□

## 14.9 Additional Exercises

**Exercise: 3 stars (stack\_compiler)** HP Calculators, programming languages like Forth and Postscript, and abstract machines like the Java Virtual Machine all evaluate arithmetic expressions using a stack. For instance, the expression

$(2*3)+(3*(4-2))$

would be entered as

`2 3 * 3 4 2 - * +`

and evaluated like this (where we show the program being evaluated on the right and the contents of the stack on the left):

<code>[]</code>		<code>2 3 * 3 4 2 - * +</code>
<code>[2]</code>		<code>3 * 3 4 2 - * +</code>
<code>[3, 2]</code>		<code>* 3 4 2 - * +</code>
<code>[6]</code>		<code>3 4 2 - * +</code>
<code>[3, 6]</code>		<code>4 2 - * +</code>
<code>[4, 3, 6]</code>		<code>2 - * +</code>
<code>[2, 4, 3, 6]</code>		<code>- * +</code>
<code>[2, 3, 6]</code>		<code>* +</code>
<code>[6, 6]</code>		<code>+</code>
<code>[12]</code>		

The task of this exercise is to write a small compiler that translates **aexps** into stack machine instructions.

The instruction set for our stack language will consist of the following instructions:

- **SPush n**: Push the number `n` on the stack.
- **SLoad x**: Load the identifier `x` from the store and push it on the stack
- **SPlus**: Pop the two top numbers from the stack, add them, and push the result onto the stack.
- **SMinus**: Similar, but subtract.
- **SMult**: Similar, but multiply.



```

Inductive sinstr : Type :=
| SPush : nat → sinstr
| SLoad : id → sinstr
| SPlus : sinstr
| SMinus : sinstr
| SMult : sinstr.

```

Write a function to evaluate programs in the stack language. It should take as input a state, a stack represented as a list of numbers (top stack item is the head of the list), and a program represented as a list of instructions, and it should return the stack after executing the program. Test your function on the examples below.

Note that the specification leaves unspecified what to do when encountering an SPlus, SMinus, or SMult instruction if the stack contains less than two elements. In a sense, it is immaterial what we do, since our compiler will never emit such a malformed program.

```

Fixpoint s_execute (st : state) (stack : list nat)
  (prog : list sinstr)
  : list nat :=

```

*admit.*

Example s\_execute1 :

```

  s_execute empty_state []
    [SPush 5; SPush 3; SPush 1; SMinus]
= [2; 5].
Admitted.

```

Example s\_execute2 :

```

  s_execute (t_update empty_state X 3) [3;4]
    [SPush 4; SLoad X; SMult; SPlus]
= [15; 4].
Admitted.

```

Next, write a function that compiles an **aexp** into a stack machine program. The effect of running the program should be the same as pushing the value of the expression on the stack.

```

Fixpoint s_compile (e : aexp) : list sinstr :=
admit.

```

After you've defined s\_compile, prove the following to test that it works.

Example s\_compile1 :

```

  s_compile (AMinus (ALd X) (AMult (ANum 2) (ALd Y)))
= [SLoad X; SPush 2; SLoad Y; SMult; SMinus].
Admitted.
□

```

**Exercise: 4 stars, advanced (stack\_compiler\_correct)** Now we'll prove the correctness of the compiler implemented in the previous exercise. Remember that the specification left unspecified what to do when encountering an `SPlus`, `SMinus`, or `SMult` instruction if the stack contains less than two elements. (In order to make your correctness proof easier you might find it helpful to go back and change your implementation!)

Prove the following theorem. You will need to start by stating a more general lemma to get a usable induction hypothesis; the main theorem will then be a simple corollary of this lemma.

Theorem `s_compile_correct` :  $\forall (st : \text{state}) (e : \text{aexp}),$   
`s_execute st [] (s_compile e) = [ aeval st e ]`.

Proof.

*Admitted.*

□

**Exercise: 5 stars, advanced (break\_imp)** Module `BREAKIMP`.

Imperative languages like C and Java often include a *break* or similar statement for interrupting the execution of loops. In this exercise we consider how to add *break* to `Imp`. First, we need to enrich the language of commands with an additional case.

Inductive `com` : Type :=

| `CSkip` : `com`  
 | `CBreak` : `com`  
 | `CAss` : `id` → `aexp` → `com`  
 | `CSeq` : `com` → `com` → `com`  
 | `Clf` : `bexp` → `com` → `com` → `com`  
 | `CWhile` : `bexp` → `com` → `com`.

Notation "'SKIP'" :=

`CSkip`.

Notation "'BREAK'" :=

`CBreak`.

Notation "x '::=' a" :=

(`CAss x a`) (at level 60).

Notation "c1 ;; c2" :=

(`CSeq c1 c2`) (at level 80, right associativity).

Notation "'WHILE' b 'DO' c 'END'" :=

(`CWhile b c`) (at level 80, right associativity).

Notation "'IFB' c1 'THEN' c2 'ELSE' c3 'FI'" :=

(`Clf c1 c2 c3`) (at level 80, right associativity).

Next, we need to define the behavior of *BREAK*. Informally, whenever *BREAK* is executed in a sequence of commands, it stops the execution of that sequence and signals that the innermost enclosing loop should terminate. (If there aren't any enclosing loops, then the

whole program simply terminates.) The final state should be the same as the one in which the *BREAK* statement was executed.

One important point is what to do when there are multiple loops enclosing a given *BREAK*. In those cases, *BREAK* should only terminate the *innermost* loop. Thus, after executing the following... `X ::= 0;; Y ::= 1;; WHILE 0 <> Y DO WHILE TRUE DO BREAK END;; X ::= 1;; Y ::= Y - 1 END ...` the value of *X* should be 1, and not 0.

One way of expressing this behavior is to add another parameter to the evaluation relation that specifies whether evaluation of a command executes a *BREAK* statement:

Inductive **status** : Type :=

| SContinue : **status**  
| SBreak : **status**.

Reserved Notation "`c1 '/' st '\\ s '/' st'`"  
(at level 40, *st*, *s* at level 39).

Intuitively, `c / st \\ s / st'` means that, if *c* is started in state *st*, then it terminates in state *st'* and either signals that the innermost surrounding loop (or the whole program) should exit immediately (*s* = SBreak) or that execution should continue normally (*s* = SContinue).

The definition of the "`c / st \\ s / st'`" relation is very similar to the one we gave above for the regular evaluation relation (`c / st \\ st'`) – we just need to handle the termination signals appropriately:

- If the command is *SKIP*, then the state doesn't change, and execution of any enclosing loop can continue normally.
- If the command is *BREAK*, the state stays unchanged, but we signal a SBreak.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution can continue normally.
- If the command is of the form *IFB b THEN c1 ELSE c2 FI*, then the state is updated as in the original semantics of Imp, except that we also propagate the signal from the execution of whichever branch was taken.
- If the command is a sequence `c1 ;; c2`, we first execute *c1*. If this yields a SBreak, we skip the execution of *c2* and propagate the SBreak signal to the surrounding context; the resulting state is the same as the one obtained by executing *c1* alone. Otherwise, we execute *c2* on the state obtained after executing *c1*, and propagate the signal generated there.
- Finally, for a loop of the form *WHILE b DO c END*, the semantics is almost the same as before. The only difference is that, when *b* evaluates to true, we execute *c* and check the signal that it raises. If that signal is SContinue, then the execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop, and the resulting state is the same as the one resulting from the execution of the current

iteration. In either case, since *BREAK* only terminates the innermost loop, *WHILE* signals *SContinue*.

Based on the above description, complete the definition of the **ceval** relation.

Inductive **ceval** : **com** → state → **status** → state → Prop :=  
 | E\_Skip : ∀ st,  
     CSkip / st \\\ SContinue / st

where "c1 '/' st '\\' s '/' st'" := (**ceval** c1 st s st').

Now prove the following properties of your definition of **ceval**:

Theorem break\_ignore : ∀ c st st' s,  
 (BREAK;; c) / st \\\ s / st' →  
 st = st'.

Proof.

*Admitted.*

Theorem while\_continue : ∀ b c st st' s,  
 (WHILE b DO c END) / st \\\ s / st' →  
 s = SContinue.

Proof.

*Admitted.*

Theorem while\_stops\_on\_break : ∀ b c st st',  
 beval st b = true →  
 c / st \\\ SBreak / st' →  
 (WHILE b DO c END) / st \\\ SContinue / st'.

Proof.

*Admitted.*

**Exercise: 3 stars, advanced, optional (while\_break\_true)** Theorem while\_break\_true

: ∀ b c st st',  
 (WHILE b DO c END) / st \\\ SContinue / st' →  
 beval st' b = true →  
 ∃ st'', c / st'' \\\ SBreak / st'.

Proof.

*Admitted.*

**Exercise: 4 stars, advanced, optional (ceval\_deterministic)** Theorem ceval\_deterministic:

∀ (c:com) st st1 st2 s1 s2,  
 c / st \\\ s1 / st1 →  
 c / st \\\ s2 / st2 →  
 st1 = st2 ∧ s1 = s2.

Proof.

*Admitted.*

End BREAKIMP.

□

**Exercise: 3 stars, optional (short\_circuit)** Most modern programming languages use a “short-circuit” evaluation rule for boolean **and**: to evaluate **BAnd** *b1* *b2*, first evaluate *b1*. If it evaluates to **false**, then the entire **BAnd** expression evaluates to **false** immediately, without evaluating *b2*. Otherwise, *b2* is evaluated to determine the result of the **BAnd** expression.

Write an alternate version of **beval** that performs short-circuit evaluation of **BAnd** in this manner, and prove that it is equivalent to **beval**.

□

**Exercise: 4 stars, optional (add\_for\_loop)** Add C-style **for** loops to the language of commands, update the **ceval** definition to define the semantics of **for** loops, and add cases for **for** loops as needed so that all the proofs in this file are accepted by Coq.

A **for** loop should be parameterized by (a) a statement executed initially, (b) a test that is run on each iteration of the loop to determine whether the loop should continue, (c) a statement executed at the end of each loop iteration, and (d) a statement that makes up the body of the loop. (You don’t need to worry about making up a concrete Notation for **for** loops, but feel free to play with this too if you like.)

□

# Chapter 15

## Library ImpParser

### 15.1 ImpParser: Lexing and Parsing in Coq

The development of the Imp language in *Imp.v* completely ignores issues of concrete syntax – how an ascii string that a programmer might write gets translated into abstract syntax trees defined by the datatypes **aexp**, **bexp**, and **com**. In this chapter, we illustrate how the rest of the story can be filled in by building a simple lexical analyzer and parser using Coq’s functional programming facilities.

It is not important to understand all the details here (and accordingly, the explanations are fairly terse and there are no exercises). The main point is simply to demonstrate that it can be done. You are invited to look through the code – most of it is not very complicated, though the parser relies on some “monadic” programming idioms that may require a little work to make out – but most readers will probably want to just skim down to the Examples section at the very end to get the punchline.

### 15.2 Internals

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.Lists.List.
Import ListNotations.

Require Import Maps.
Require Import Imp.
```

#### 15.2.1 Lexical Analysis

```
Definition isWhite (c : ascii) : bool :=
```

```

let n := nat_of_ascii c in
orb (orb (beq_nat n 32)
        (beq_nat n 9))
    (orb (beq_nat n 10)
        (beq_nat n 13)).

Notation "x '<=?' y" := (leb x y)
(at level 70, no associativity) : nat_scope.

Definition isLowerAlpha (c : ascii) : bool :=
let n := nat_of_ascii c in
andb (97 <=? n) (n <=? 122).

Definition isAlpha (c : ascii) : bool :=
let n := nat_of_ascii c in
orb (andb (65 <=? n) (n <=? 90))
    (andb (97 <=? n) (n <=? 122)).

Definition isDigit (c : ascii) : bool :=
let n := nat_of_ascii c in
andb (48 <=? n) (n <=? 57).

Inductive chartype := white | alpha | digit | other.

Definition classifyChar (c : ascii) : chartype :=
if isWhite c then
  white
else if isAlpha c then
  alpha
else if isDigit c then
  digit
else
  other.

Fixpoint list_of_string (s : string) : list ascii :=
match s with
| EmptyString => []
| String c s => c :: (list_of_string s)
end.

Fixpoint string_of_list (xs : list ascii) : string :=
fold_right String EmptyString xs.

Definition token := string.

Fixpoint tokenize_helper (cls : chartype) (acc xs : list ascii)
  : list (list ascii) :=
let tk := match acc with [] => [] | _ :: _ => [rev acc] end in
match xs with
| [] => tk

```

```

| (x :: xs') ⇒
  match cls, classifyChar x, x with
  | -, -, "(" ⇒
    tk ++ ["(" :: (tokenize_helper other [] xs')]
  | -, -, ")" ⇒
    tk ++ [")" :: (tokenize_helper other [] xs')]
  | -, white, _ ⇒
    tk ++ (tokenize_helper white [] xs')
  | alpha, alpha, x ⇒
    tokenize_helper alpha (x :: acc) xs'
  | digit, digit, x ⇒
    tokenize_helper digit (x :: acc) xs'
  | other, other, x ⇒
    tokenize_helper other (x :: acc) xs'
  | -, tp, x ⇒
    tk ++ (tokenize_helper tp [x] xs')
end
end %char.

```

Definition `tokenize (s : string) : list string :=`  
`map string_of_list (tokenize_helper white [] (list_of_string s)).`

Example `tokenize_ex1 :`  
`tokenize "abc12==3 223*(3+(a+c))" %string`  
`= ["abc"; "12"; "=="; "3"; "223";`  
`"*"; "("; "3"; "+"; "(";`  
`"a"; "+"; "c"; ")" ; "]" %string.`

Proof. reflexivity. Qed.

## 15.2.2 Parsing

### Options with Errors

An **option** type with error messages:

```

Inductive optionE (X:Type) : Type :=
| SomeE : X → optionE X
| NoneE : string → optionE X.

```

Implicit Arguments SomeE [[X]].

Implicit Arguments NoneE [[X]].

Some syntactic sugar to make writing nested match-expressions on `optionE` more convenient.

```

Notation "'DO' ( x , y ) <== e1 ; e2"
:= (match e1 with

```



```

      | SomeE (x, y) ⇒ e2
      | NoneE err ⇒ NoneE err
    end)
(right associativity, at level 60).
Notation "'DO' ( x , y ) <- e1 ; e2 'OR' e3"
:= (match e1 with
    | SomeE (x, y) ⇒ e2
    | NoneE err ⇒ e3
    end)
(right associativity, at level 60, e2 at next level).

```

## Symbol Table

Build a mapping from *tokens* to *nats*. A real parser would do this incrementally as it encountered new symbols, but passing around the symbol table inside the parsing functions is a bit inconvenient, so instead we do it as a first pass.

```

Fixpoint build_symtable (xs : list token) (n : nat)
  : (token → nat) :=
  match xs with
  | [] ⇒ (fun s ⇒ n)
  | x :: xs ⇒
    if (forallb isLowerAlpha (list_of_string x))
    then (fun s ⇒ if string_dec s x then n
                  else (build_symtable xs (S n) s))
    else build_symtable xs n
  end.

```

## Generic Combinators for Building Parsers

Open Scope *string\_scope*.

```

Definition parser (T : Type) :=
  list token → optionE (T × list token).

```

```

Fixpoint many_helper {T} (p : parser T) acc steps xs :=
  match steps, p xs with
  | 0, _ ⇒
    NoneE "Too many recursive calls"
  | _, NoneE _ ⇒
    SomeE ((rev acc), xs)
  | S steps', SomeE (t, xs') ⇒
    many_helper p (t :: acc) steps' xs'
  end.

```

A (step-indexed) parser that expects zero or more *ps*:

Fixpoint many  $\{T\}$  ( $p$  : parser  $T$ ) ( $steps$  : nat) : parser (list  $T$ ) :=  
 many\_helper  $p$  []  $steps$ .

A parser that expects a given token, followed by  $p$ :

Definition firstExpect  $\{T\}$  ( $t$  : token) ( $p$  : parser  $T$ )  
 : parser  $T$  :=

```
fun xs => match xs with
| x :: xs' =>
  if string_dec x t
  then p xs'
  else NoneE ("expected '" ++ t ++ "'")
| [] =>
  NoneE ("expected '" ++ t ++ "'")
end.
```

A parser that expects a particular token:

Definition expect ( $t$  : token) : parser unit :=  
 firstExpect  $t$  (fun xs => SomeE(tt, xs)).

## A Recursive-Descent Parser for Imp

Identifiers:

Definition parselIdentifier ( $symtable$  : string  $\rightarrow$  nat)  
 ( $xs$  : list token)  
 : optionE (id  $\times$  list token) :=

```
match xs with
| [] => NoneE "Expected identifier"
| x :: xs' =>
  if forallb isLowerAlpha (list_of_string x) then
    SomeE (Id (symtable x), xs')
  else
    NoneE ("Illegal identifier:'" ++ x ++ "'")
end.
```

Numbers:

Definition parseNumber ( $xs$  : list token)  
 : optionE (nat  $\times$  list token) :=

```
match xs with
| [] => NoneE "Expected number"
| x :: xs' =>
  if forallb isDigit (list_of_string x) then
    SomeE (fold_left
      (fun n d =>
        10  $\times$  n + (nat_of_ascii d -
```

```

                                nat_of_ascii "0"%char))
      (list_of_string x)
      0,
      xs')
    else
      NoneE "Expected number"
    end.
end.

Parse arithmetic expressions
Fixpoint parsePrimaryExp (steps:nat) symtable
  (xs : list token)
  : optionE (aexp × list token) :=
match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
  DO (i, rest) <- parselIdentifier symtable xs ;
  SomeE (Ald i, rest)
  OR DO (n, rest) <- parseNumber xs ;
  SomeE (ANum n, rest)
  OR (DO (e, rest) <== firstExpect "("
      (parseSumExp steps' symtable) xs ;
  DO (u, rest') <== expect ")" rest ;
  SomeE(e, rest'))
end
with parseProductExp (steps:nat) symtable
  (xs : list token) :=
match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
  DO (e, rest) <==
    parsePrimaryExp steps' symtable xs ;
  DO (es, rest') <==
    many (firstExpect "*" (parsePrimaryExp steps' symtable))
      steps' rest ;
  SomeE (fold_left AMult es e, rest')
end
with parseSumExp (steps:nat) symtable (xs : list token) :=
match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
  DO (e, rest) <==
    parseProductExp steps' symtable xs ;
  DO (es, rest') <==

```

```

many (fun xs ⇒
  DO (e, rest') <-
    firstExpect "+"
    (parseProductExp steps' symtable) xs;
  SomeE ( (true, e), rest')
OR DO (e, rest') <==
  firstExpect "-"
  (parseProductExp steps' symtable) xs;
  SomeE ( (false, e), rest'))
steps' rest;
SomeE (fold_left (fun e0 term ⇒
  match term with
    (true, e) ⇒ APlus e0 e
  | (false, e) ⇒ AMinus e0 e
  end)
  es e,
  rest')
end.

```

Definition parseAExp := parseSumExp.

Parsing boolean expressions:

Fixpoint parseAtomicExp (steps: nat) (symtable : string → nat)  
 (xs : list token) :=

```

match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
  DO (u, rest) <- expect "true" xs;
  SomeE (BTrue, rest)
OR DO (u, rest) <- expect "false" xs;
  SomeE (BFalse, rest)
OR DO (e, rest) <-
  firstExpect "not"
  (parseAtomicExp steps' symtable)
  xs;
  SomeE (BNot e, rest)
OR DO (e, rest) <-
  firstExpect "("
  (parseConjunctionExp steps' symtable) xs;
  (DO (u, rest') <== expect ")" rest;
  SomeE (e, rest'))
OR DO (e, rest) <== parseProductExp steps' symtable xs;
  (DO (e', rest') <-
  firstExpect "=="

```

```

      (parseAExp steps' symtable) rest;
    SomeE (BEq e e', rest')
  OR DO (e', rest') <-
    firstExpect "<="
      (parseAExp steps' symtable) rest;
    SomeE (BLe e e', rest')
  OR
    NoneE
    "Expected '==' or '<=' after arithmetic expression")
end
with parseConjunctionExp (steps:nat)
  (symtable : string→nat)
  (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    DO (e, rest) <==
      parseAtomicExp steps' symtable xs ;
    DO (es, rest') <==
      many (firstExpect "&&"
        (parseAtomicExp steps' symtable))
        steps' rest;
    SomeE (fold_left BAnd es e, rest')
  end.

```

Definition parseBExp := parseConjunctionExp.

Check parseConjunctionExp.

```

Definition testParsing {X : Type}
  (p : nat → (string → nat) →
    list token →
    optionE (X × list token))
  (s : string) :=
  let t := tokenize s in
  p 100 (build_symtable t 0) t.

```

Parsing commands:

```

Fixpoint parseSimpleCommand (steps:nat)
  (symtable:string→nat)
  (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    DO (u, rest) <- expect "SKIP" xs;

```

```

    SomeE (SKIP, rest)
OR DO (e, rest) <-
    firstExpect "IF" (parseBExp steps' symtable) xs;
    DO (c, rest') <==
        firstExpect "THEN"
            (parseSequencedCommand steps' symtable) rest;
    DO (c', rest'') <==
        firstExpect "ELSE"
            (parseSequencedCommand steps' symtable) rest';
    DO (u, rest''') <==
        expect "END" rest'';
    SomeE(IFB e THEN c ELSE c' FI, rest''')
OR DO (e, rest) <-
    firstExpect "WHILE"
        (parseBExp steps' symtable) xs;
    DO (c, rest') <==
        firstExpect "DO"
            (parseSequencedCommand steps' symtable) rest;
    DO (u, rest'') <==
        expect "END" rest';
    SomeE(WHILE e DO c END, rest'')
OR DO (i, rest) <==
    parselIdentifier symtable xs;
    DO (e, rest') <==
        firstExpect ":@" (parseAExp steps' symtable) rest;
    SomeE(i ::= e, rest')
end

with parseSequencedCommand (steps:nat)
                           (symtable:string→nat)
                           (xs : list token) :=

match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
    DO (c, rest) <==
        parseSimpleCommand steps' symtable xs;
    DO (c', rest') <-
        firstExpect ";;"
            (parseSequencedCommand steps' symtable) rest;
    SomeE(c ;; c', rest')
OR
    SomeE(c, rest)

```

end.

Definition bignumber := 1000.

Definition parse (*str* : **string**) : **optionE** (**com** × **list** token) :=  
 let *tokens* := tokenize *str* in  
 parseSequencedCommand bignumber  
 (build\_symtable *tokens* 0) *tokens*.

## 15.3 Examples

*Date* : 2016 – 02 – 29 10 : 14 : 50 – 0500(*Mon*, 29*Feb*2016)

# Chapter 16

## Library ImpCEvalFun

### 16.1 ImpCEvalFun: Evaluation Function for Imp

We saw in the `Imp` chapter how a naive approach to defining a function representing evaluation for `Imp` runs into difficulties. There, we adopted the solution of changing from a functional to a relational definition of evaluation. In this optional chapter, we consider strategies for getting the functional approach to work.

### 16.2 A Broken Evaluator

```
Require Import Coq.omega.Omega.  
Require Import Coq.Arith.Arith.  
  
Require Import Sflib.  
Require Import Imp.  
Require Import Maps.
```

Here was our first try at an evaluation function for commands, omitting *WHILE*.

```
Fixpoint ceval_step1 (st : state) (c : com) : state :=  
  match c with  
  | SKIP =>  
    st  
  | l ::= a1 =>  
    t_update st l (aeval st a1)  
  | c1 ;; c2 =>  
    let st' := ceval_step1 st c1 in  
    ceval_step1 st' c2  
  | IFB b THEN c1 ELSE c2 FI =>  
    if (beval st b)  
    then ceval_step1 st c1  
    else ceval_step1 st c2
```



```
| WHILE b1 DO c1 END =>
  st
end.
```

As we remarked in chapter `Imp`, in a traditional functional programming language like ML or Haskell we could write the WHILE case as follows:

```
| WHILE b1 DO c1 END => if (beval st b1) then ceval_step1 st (c1;;
  WHILE b1 DO c1 END) else st
```

Coq doesn't accept such a definition (*Error: Cannot guess decreasing argument of fix*) because the function we want to define is not guaranteed to terminate. Indeed, the changed `ceval_step1` function applied to the `loop` program from *Imp.v* would never terminate. Since Coq is not just a functional programming language, but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an invalid(!) Coq program showing what would go wrong if Coq allowed non-terminating recursive functions:

```
Fixpoint loop_false (n : nat) : False := loop_false n.
```

That is, propositions like **False** would become provable (e.g., `loop_false 0` would be a proof of **False**), which would be a disaster for Coq's logical consistency.

Thus, because it doesn't terminate on all inputs, the full version of `ceval_step1` cannot be written in Coq – at least not without one additional trick...

## 16.3 A Step-Indexed Evaluator

The trick we need is to pass an *additional* parameter to the evaluation function that tells it how long to run. Informally, we start the evaluator with a certain amount of "gas" in its tank, and we allow it to run until either it terminates in the usual way *or* it runs out of gas, at which point we simply stop evaluating and say that the final result is the empty memory. (We could also say that the result is the current state at the point where the evaluator runs out of gas – it doesn't really matter because the result is going to be wrong in either case!)

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) : state :=
  match i with
  | 0 => empty_state
  | S i' =>
    match c with
    | SKIP =>
      st
    | l ::= a1 =>
      t_update st l (aeval st a1)
    | c1 ;; c2 =>
      let st' := ceval_step2 st c1 i' in
      ceval_step2 st' c2 i'
```

```

| IFB b THEN c1 ELSE c2 FI ⇒
  if (beval st b)
    then ceval_step2 st c1 i'
    else ceval_step2 st c2 i'
| WHILE b1 DO c1 END ⇒
  if (beval st b1)
    then let st' := ceval_step2 st c1 i' in
      ceval_step2 st' c i'
    else st
end
end.

```

*Note:* It is tempting to think that the index  $i$  here is counting the “number of steps of evaluation.” But if you look closely you’ll see that this is not the case: for example, in the rule for sequencing, the same  $i$  is passed to both recursive calls. Understanding the exact way that  $i$  is treated will be important in the proof of `ceval__ceval_step`, which is given as an exercise below.

One thing that is not so nice about this evaluator is that we can’t tell, from its result, whether it stopped because the program terminated normally or because it ran out of gas. Our next version returns an **option state** instead of just a **state**, so that we can distinguish between normal and abnormal termination.

```

Fixpoint ceval_step3 (st : state) (c : com) (i : nat)
  : option state :=
  match i with
  | 0 ⇒ None
  | S i' ⇒
    match c with
    | SKIP ⇒
      Some st
    | l ::= a1 ⇒
      Some (t_update st l (aeval st a1))
    | c1 ;; c2 ⇒
      match (ceval_step3 st c1 i') with
      | Some st' ⇒ ceval_step3 st' c2 i'
      | None ⇒ None
      end
    | IFB b THEN c1 ELSE c2 FI ⇒
      if (beval st b)
        then ceval_step3 st c1 i'
        else ceval_step3 st c2 i'
    | WHILE b1 DO c1 END ⇒
      if (beval st b1)
        then match (ceval_step3 st c1 i') with

```

```

        | Some st' ⇒ ceval_step3 st' c i'
        | None ⇒ None
      end
    else Some st
  end
end.

```

We can improve the readability of this version by introducing a bit of auxiliary notation to hide the plumbing involved in repeatedly matching against optional states.

Notation "'LETOPT' x <== e1 'IN' e2"

```

:= (match e1 with
   | Some x ⇒ e2
   | None ⇒ None
end)

```

(right associativity, at level 60).

```

Fixpoint ceval_step (st : state) (c : com) (i : nat)
  : option state :=

```

```

match i with
| O ⇒ None
| S i' ⇒
  match c with
  | SKIP ⇒
    Some st
  | l ::= a1 ⇒
    Some (t_update st l (aeval st a1))
  | c1 ;; c2 ⇒
    LETOPT st' <== ceval_step st c1 i' IN
    ceval_step st' c2 i'
  | IFB b THEN c1 ELSE c2 FI ⇒
    if (beval st b)
    then ceval_step st c1 i'
    else ceval_step st c2 i'
  | WHILE b1 DO c1 END ⇒
    if (beval st b1)
    then LETOPT st' <== ceval_step st c1 i' IN
      ceval_step st' c i'
    else Some st
  end
end.

```

```

Definition test_ceval (st:state) (c:com) :=
  match ceval_step st c 500 with
  | None ⇒ None

```

```
| Some st  $\Rightarrow$  Some (st X, st Y, st Z)
end.
```

**Exercise: 2 stars, recommended (pup\_to\_n)** Write an *Imp* program that sums the numbers from 1 to X (inclusive:  $1 + 2 + \dots + X$ ) in the variable Y. Make sure your solution satisfies the test that follows.

Definition pup\_to\_n : **com** :=  
*admit*.

□

**Exercise: 2 stars, optional (peven)** Write a *While* program that sets Z to 0 if X is even and sets Z to 1 otherwise. Use *ceval\_test* to test your program.

□

## 16.4 Relational vs. Step-Indexed Evaluation

As for arithmetic and boolean expressions, we'd hope that the two alternative definitions of evaluation would actually amount to the same thing in the end. This section shows that this is the case.

Theorem ceval\_step\_\_ceval:  $\forall c \ st \ st',$   
 $(\exists i, \text{ceval\_step } st \ c \ i = \text{Some } st') \rightarrow$   
 $c / st \ \backslash\backslash \ st'.$

Proof.

```
intros c st st' H.
inversion H as [i E].
clear H.
generalize dependent st'.
generalize dependent st.
generalize dependent c.
induction i as [| i' ].
-
  intros c st st' H. inversion H.
-
  intros c st st' H.
  destruct c;
    simpl in H; inversion H; subst; clear H.
  + apply E_Skip.
  + apply E_Ass. reflexivity.
  +
```

```

destruct (ceval_step st c1 i') eqn:Hegr1.
×
  apply E_Seq with s.
  apply IHi'. rewrite Hegr1. reflexivity.
  apply IHi'. simpl in H1. assumption.
×
  inversion H1.
+
destruct (beval st b) eqn:Hegr.
×
  apply E_IfTrue. rewrite Hegr. reflexivity.
  apply IHi'. assumption.
×
  apply E_IfFalse. rewrite Hegr. reflexivity.
  apply IHi'. assumption.
+ destruct (beval st b) eqn :Hegr.
×
  destruct (ceval_step st c i') eqn:Hegr1.
  {
    apply E_WhileLoop with s. rewrite Hegr.
    reflexivity.
    apply IHi'. rewrite Hegr1. reflexivity.
    apply IHi'. simpl in H1. assumption. }
  { inversion H1. }
×
  inversion H1.
  apply E_WhileEnd.
  rewrite ← Hegr. subst. reflexivity. Qed.

```

**Exercise: 4 stars (ceval\_step\_\_ceval\_inf)** Write an informal proof of `ceval_step__ceval`, following the usual template. (The template for case analysis on an inductively defined value should look the same as for induction, except that there is no induction hypothesis.) Make your proof communicate the main ideas to a human reader; do not simply transcribe the steps of the formal proof.

□

Theorem `ceval_step_more`:  $\forall i1\ i2\ st\ st'\ c,$

$i1 \leq i2 \rightarrow$

$\text{ceval\_step}\ st\ c\ i1 = \text{Some}\ st' \rightarrow$

$\text{ceval\_step}\ st\ c\ i2 = \text{Some}\ st'.$

Proof.

induction  $i1$  as [| $i1'$ ]; intros  $i2\ st\ st'\ c\ Hle\ Hceval.$

-

```

    simpl in Hceval. inversion Hceval.
-
    destruct i2 as [|i2']. inversion Hle.
    assert (Hle': i1' ≤ i2') by omega.
    destruct c.
+
    simpl in Hceval. inversion Hceval.
    reflexivity.
+
    simpl in Hceval. inversion Hceval.
    reflexivity.
+
    simpl in Hceval. simpl.
    destruct (ceval_step st c1 i1') eqn:Hegst1'o.
    ×
    apply (IH i1' i2') in Hegst1'o; try assumption.
    rewrite Hegst1'o. simpl. simpl in Hceval.
    apply (IH i1' i2') in Hceval; try assumption.
    ×
    inversion Hceval.
+
    simpl in Hceval. simpl.
    destruct (beval st b); apply (IH i1' i2') in Hceval;
    assumption.
+
    simpl in Hceval. simpl.
    destruct (beval st b); try assumption.
    destruct (ceval_step st c i1') eqn: Hegst1'o.
    ×
    apply (IH i1' i2') in Hegst1'o; try assumption.
    rewrite → Hegst1'o. simpl. simpl in Hceval.
    apply (IH i1' i2') in Hceval; try assumption.
    ×
    simpl in Hceval. inversion Hceval. Qed.

```

**Exercise: 3 stars, recommended (ceval\_\_ceval\_step)** Finish the following proof. You'll need `ceval_step_more` in a few places, as well as some basic facts about  $\leq$  and `plus`.

Theorem `ceval__ceval_step`:  $\forall c \ st \ st',$   
 $c / st \setminus\setminus st' \rightarrow$   
 $\exists i, \text{ceval\_step } st \ c \ i = \text{Some } st'.$

Proof.

```

    intros c st st' Hce.

```

induction *Hce*.

*Admitted.*

□

Theorem *ceval\_and\_ceval\_step\_coincide*:  $\forall c\ st\ st',$   
 $c / st \ \backslash\backslash\ st'$

$\leftrightarrow \exists i, \text{ceval\_step } st\ c\ i = \text{Some } st'.$

Proof.

intros *c st st'*.

split. apply *ceval\_\_ceval\_step*. apply *ceval\_step\_\_ceval*.

Qed.

## 16.5 Determinism of Evaluation Again

Using the fact that the relational and step-indexed definition of evaluation are the same, we can give a slicker proof that the evaluation *relation* is deterministic.

Theorem *ceval\_deterministic'* :  $\forall c\ st\ st1\ st2,$

$c / st \ \backslash\backslash\ st1 \rightarrow$

$c / st \ \backslash\backslash\ st2 \rightarrow$

$st1 = st2.$

Proof.

intros *c st st1 st2 He1 He2*.

apply *ceval\_\_ceval\_step* in *He1*.

apply *ceval\_\_ceval\_step* in *He2*.

inversion *He1* as [*i1 E1*].

inversion *He2* as [*i2 E2*].

apply *ceval\_step\_more* with (*i2* := *i1* + *i2*) in *E1*.

apply *ceval\_step\_more* with (*i2* := *i1* + *i2*) in *E2*.

rewrite *E1* in *E2*. inversion *E2*. reflexivity.

omega. omega. Qed.

*Date* : 2016 – 03 – 0409 : 33 : 20 – 0500(*Fri*, 04Mar2016)

# Chapter 17

## Library Extraction

### 17.1 Extraction: Extracting ML from Coq

### 17.2 Basic Extraction

In its simplest form, extracting an efficient program from one written in Coq is completely straightforward.

First we say what language we want to extract into. Options are OCaml (the most mature), Haskell (which mostly works), and Scheme (a bit out of date).

*Extraction Language Ocaml.*

Now we load up the Coq environment with some definitions, either directly or by importing them from other modules.

```
Require Import Coq.Arith.Arith.
```

```
Require Import Coq.Arith.EqNat.
```

```
Require Import Sflib.
```

```
Require Import ImpCEvalFun.
```

Finally, we tell Coq the name of a definition to extract and the name of a file to put the extracted code into.

```
Extraction "impl.ml" ceval_step.
```

When Coq processes this command, it generates a file *impl.ml* containing an extracted version of *ceval\_step*, together with everything that it recursively depends on. Compile the present *.v* file and have a look at *impl.ml* now.

### 17.3 Controlling Extraction of Specific Types

We can tell Coq to extract certain Inductive definitions to specific OCaml types. For each one, we must say



- how the Coq type itself should be represented in OCaml, and
- how each constructor should be translated.

```
Extract Inductive bool ⇒ "bool" [ "true" "false" ].
```

Also, for non-enumeration types (where the constructors take arguments), we give an OCaml expression that can be used as a "recursor" over elements of the type. (Think Church numerals.)

```
Extract Inductive nat ⇒ "int"
[ "0" "(fun x -> x + 1)" ]
"(fun zero succ n -> if n=0 then zero () else succ (n-1))".
```

We can also extract defined constants to specific OCaml terms or operators.

```
Extract Constant plus ⇒ "( + )".
Extract Constant mult ⇒ "( * )".
Extract Constant beq_nat ⇒ "( = )".
```

Important: It is entirely *your responsibility* to make sure that the translations you're proving make sense. For example, it might be tempting to include this one Extract Constant minus => "(-)". but doing so could lead to serious confusion! (Why?)

```
Extraction "imp2.ml" ceval_step.
```

Have a look at the file *imp2.ml*. Notice how the fundamental definitions have changed from *imp1.ml*.

## 17.4 A Complete Example

To use our extracted evaluator to run Imp programs, all we need to add is a tiny driver program that calls the evaluator and prints out the result.

For simplicity, we'll print results by dumping out the first four memory locations in the final state.

Also, to make it easier to type in examples, let's extract a parser from the `ImpParser` Coq module. To do this, we need a few magic declarations to set up the right correspondence between Coq strings and lists of OCaml characters.

```
Require Import Ascii String.
```

```
Extract Inductive ascii ⇒ char
```

```
[
"(* If this appears, you're using Ascii internals. Please don't *) (fun (b0,b1,b2,b3,b4,b5,b6,b7)
-> let f b i = if b then 1 lsl i else 0 in Char.chr (f b0 0 + f b1 1 + f b2 2 + f b3 3 + f b4 4
+ f b5 5 + f b6 6 + f b7 7))"
]
"(* If this appears, you're using Ascii internals. Please don't *) (fun f c -> let n = Char.code
c in let h i = (n land (1 lsl i)) <> 0 in f (h 0) (h 1) (h 2) (h 3) (h 4) (h 5) (h 6) (h 7))".
```

```

Extract Constant zero ⇒ "'\000'".
Extract Constant one ⇒ "'\001'".
Extract Constant shift ⇒
  "fun b c -> Char.chr (((Char.code c) lsl 1) land 255 + if b then 1 else 0)".
Extract Inlined Constant ascii_dec ⇒ "(=)".

```

We also need one more variant of booleans.

```

Extract Inductive sumbool ⇒ "bool" ["true" "false"].

```

The extraction is the same as always.

```

Require Import Imp.
Require Import ImpParser.
Extraction "imp.ml" empty_state ceval_step parse.

```

Now let's run our generated Imp evaluator. First, have a look at *impdriver.ml*. (This was written by hand, not extracted.)

Next, compile the driver together with the extracted code and execute it, as follows.

```

ocamlc -w -20 -w -26 -o impdriver imp.mli imp.ml impdriver.ml
./impdriver

```

(The *-w* flags to *ocamlc* are just there to suppress a few spurious warnings.)

## 17.5 Discussion

Since we've proved that the *ceval\_step* function behaves the same as the **ceval** relation in an appropriate sense, the extracted program can be viewed as a *certified* Imp interpreter. Of course, the parser we're using is not certified, since we didn't prove anything about it!

*Date* : 2016 – 02 – 29 10 : 14 : 50 – 0500 (*Mon, 29 Feb 2016*)

# Chapter 18

## Library Equiv

### 18.1 Equiv: Program Equivalence

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Lists.List.
Require Import Coq.Logic.FunctionalExtensionality.
Import ListNotations.

Require Import Sflib.
Require Import Maps.
Require Import Imp.
```

**Some general advice for working on exercises:**

- Most of the Coq proofs we ask you to do are similar to proofs that we’ve provided. Before starting to work on exercises problems, take the time to work through our proofs (both informally, on paper, and in Coq) and make sure you understand them in detail. This will save you a lot of time.
- The Coq proofs we’re doing now are sufficiently complicated that it is more or less impossible to complete them simply by random experimentation or “following your nose.” You need to start with an idea about why the property is true and how the proof is going to go. The best way to do this is to write out at least a sketch of an informal proof on paper – one that intuitively convinces you of the truth of the theorem – before starting to work on the formal one. Alternately, grab a friend and try to convince them that the theorem is true; then try to formalize your explanation.
- Use automation to save work! Some of the proofs in this chapter’s exercises are pretty long if you try to write out all the cases explicitly.

## 18.2 Behavioral Equivalence

In an earlier chapter, we investigated the correctness of a very simple program transformation: the `optimize_0plus` function. The programming language we were considering was the first version of the language of arithmetic expressions – with no variables – so in that setting it was very easy to define what it means for a program transformation to be correct: it should always yield a program that evaluates to the same number as the original.

To go further and talk about the correctness of program transformations in the full Imp language, we need to consider the role of variables and state.

### 18.2.1 Definitions

For **aexprs** and **bexprs** with variables, the definition we want is clear. We say that two **aexprs** or **bexprs** are *behaviorally equivalent* if they evaluate to the same result *in every state*.

**Definition** `aequiv (a1 a2 : aexp) : Prop :=`

`∀ (st:state),  
    aeval st a1 = aeval st a2.`

**Definition** `bequiv (b1 b2 : bexp) : Prop :=`

`∀ (st:state),  
    beval st b1 = beval st b2.`

For commands, the situation is a little more subtle. We can't simply say "two commands are behaviorally equivalent if they evaluate to the same ending state whenever they are started in the same initial state," because some commands, when run in some starting states, don't terminate in any final state at all! What we need instead is this: two commands are behaviorally equivalent if, for any given starting state, they either both diverge or both terminate in the same final state. A compact way to express this is "if the first one terminates in a particular state then so does the second, and vice versa."

**Definition** `cequiv (c1 c2 : com) : Prop :=`

`∀ (st st' : state),  
    (c1 / st \\\ st') ↔ (c2 / st \\\ st').`

**Exercise: 2 stars (equiv\_classes)** Given the following programs, group together those that are equivalent in Imp. Your answer should be given as a list of lists, where each sub-list represents a group of equivalent programs. For example, if you think programs (a) through (h) are all equivalent to each other, but not to (i), your answer should look like this:

`[prog_a;prog_b;prog_c;prog_d;prog_e;prog_f;prog_g;prog_h] ; [prog_i]`

Write down your answer below in the definition of `equiv_classes`.

**Definition** `prog_a : com :=`

`WHILE BNot (BLe (Ald X) (ANum 0)) DO  
    X ::= APlus (Ald X) (ANum 1)  
  END.`

```

Definition prog_b : com :=
  IFB BEq (Ald X) (ANum 0) THEN
    X ::= APlus (Ald X) (ANum 1);;
    Y ::= ANum 1
  ELSE
    Y ::= ANum 0
  FI;;
  X ::= AMinus (Ald X) (Ald Y);;
  Y ::= ANum 0.

Definition prog_c : com :=
  SKIP.

Definition prog_d : com :=
  WHILE BNot (BEq (Ald X) (ANum 0)) DO
    X ::= APlus (AMult (Ald X) (Ald Y)) (ANum 1)
  END.

Definition prog_e : com :=
  Y ::= ANum 0.

Definition prog_f : com :=
  Y ::= APlus (Ald X) (ANum 1);;
  WHILE BNot (BEq (Ald X) (Ald Y)) DO
    Y ::= APlus (Ald X) (ANum 1)
  END.

Definition prog_g : com :=
  WHILE BTrue DO
    SKIP
  END.

Definition prog_h : com :=
  WHILE BNot (BEq (Ald X) (Ald X)) DO
    X ::= APlus (Ald X) (ANum 1)
  END.

Definition prog_i : com :=
  WHILE BNot (BEq (Ald X) (Ald Y)) DO
    X ::= APlus (Ald Y) (ANum 1)
  END.

Definition equiv_classes : list (list com) :=
  admit.
  □

```

## 18.2.2 Examples

Here are some simple examples of equivalences of arithmetic and boolean expressions.

Theorem aequiv\_example:

aequiv (AMinus (Ald X) (Ald X)) (ANum 0).

Proof.

intros *st*. simpl. omega.

Qed.

Theorem bequiv\_example:

bequiv (BEq (AMinus (Ald X) (Ald X)) (ANum 0)) BTrue.

Proof.

intros *st*. unfold beval.

rewrite aequiv\_example. reflexivity.

Qed.

For examples of command equivalence, let's start by looking at some trivial program transformations involving *SKIP*:

Theorem skip\_left:  $\forall c$ ,

cequiv

(SKIP;; *c*)

*c*.

Proof.

intros *c st st'*.

split; intros *H*.

-

inversion *H*. subst.

inversion *H2*. subst.

assumption.

-

apply E\_Seq with *st*.

apply E\_Skip.

assumption.

Qed.

**Exercise: 2 stars (skip\_right)** Prove that adding a *SKIP* after a command results in an equivalent program

Theorem skip\_right:  $\forall c$ ,

cequiv

(*c* ;; SKIP)

*c*.

Proof.

*Admitted.*

□

Similarly, here is a simple transformations that optimizes *IFB* commands:

Theorem IFB\_true\_simple:  $\forall c1\ c2$ ,

```
cequiv
  (IFB BTrue THEN c1 ELSE c2 FI)
  c1.
```

Proof.

```
intros c1 c2.
split; intros H.
-
  inversion H; subst. assumption. inversion H5.
-
  apply E_IfTrue. reflexivity. assumption. Qed.
```

Of course, few programmers would be tempted to write a conditional whose guard is literally BTrue. A more interesting case is when the guard is *equivalent* to true:

*Theorem:* If  $b$  is equivalent to BTrue, then  $IFB\ b\ THEN\ c1\ ELSE\ c2\ FI$  is equivalent to  $c1$ .

*Proof:*

- ( $\rightarrow$ ) We must show, for all  $st$  and  $st'$ , that if  $IFB\ b\ THEN\ c1\ ELSE\ c2\ FI\ /\ st\ \backslash\backslash\ st'$  then  $c1\ /\ st\ \backslash\backslash\ st'$ .

Proceed by cases on the rules that could possibly have been used to show  $IFB\ b\ THEN\ c1\ ELSE\ c2\ FI\ /\ st\ \backslash\backslash\ st'$ , namely E\_IfTrue and E\_IfFalse.

- Suppose the final rule rule in the derivation of  $IFB\ b\ THEN\ c1\ ELSE\ c2\ FI\ /\ st\ \backslash\backslash\ st'$  was E\_IfTrue. We then have, by the premises of E\_IfTrue, that  $c1\ /\ st\ \backslash\backslash\ st'$ . This is exactly what we set out to prove.
- On the other hand, suppose the final rule in the derivation of  $IFB\ b\ THEN\ c1\ ELSE\ c2\ FI\ /\ st\ \backslash\backslash\ st'$  was E\_IfFalse. We then know that  $beval\ st\ b = false$  and  $c2\ /\ st\ \backslash\backslash\ st'$ .

Recall that  $b$  is equivalent to BTrue, i.e., for all  $st$ ,  $beval\ st\ b = beval\ st\ BTrue$ . In particular, this means that  $beval\ st\ b = true$ , since  $beval\ st\ BTrue = true$ . But this is a contradiction, since E\_IfFalse requires that  $beval\ st\ b = false$ . Thus, the final rule could not have been E\_IfFalse.

- ( $\leftarrow$ ) We must show, for all  $st$  and  $st'$ , that if  $c1\ /\ st\ \backslash\backslash\ st'$  then  $IFB\ b\ THEN\ c1\ ELSE\ c2\ FI\ /\ st\ \backslash\backslash\ st'$ .

Since  $b$  is equivalent to BTrue, we know that  $beval\ st\ b = beval\ st\ BTrue = true$ . Together with the assumption that  $c1\ /\ st\ \backslash\backslash\ st'$ , we can apply E\_IfTrue to derive  $IFB\ b\ THEN\ c1\ ELSE\ c2\ FI\ /\ st\ \backslash\backslash\ st'$ .  $\square$

Here is the formal version of this proof:

Theorem IFB\_true:  $\forall\ b\ c1\ c2,$   
 $bequiv\ b\ BTrue \rightarrow$   
 $cequiv$

(IFB  $b$  THEN  $c1$  ELSE  $c2$  FI)  
 $c1$ .

Proof.

```

intros b c1 c2 Hb.
split; intros H.
-
  inversion H; subst.
+
  assumption.
+
  unfold bequiv in Hb. simpl in Hb.
  rewrite Hb in H5.
  inversion H5.
-
  apply E_lfTrue; try assumption.
  unfold bequiv in Hb. simpl in Hb.
  rewrite Hb. reflexivity. Qed.

```

**Exercise: 2 stars, recommended (IFB\_false)** Theorem IFB\_false:  $\forall b\ c1\ c2$ ,  
 bequiv  $b$  BFalse  $\rightarrow$

cequiv  
 (IFB  $b$  THEN  $c1$  ELSE  $c2$  FI)  
 $c2$ .

Proof.

*Admitted.*  
 $\square$

**Exercise: 3 stars (swap\_if\_branches)** Show that we can swap the branches of an IF  
 by negating its condition

Theorem swap\_if\_branches:  $\forall b\ e1\ e2$ ,  
 cequiv  
 (IFB  $b$  THEN  $e1$  ELSE  $e2$  FI)  
 (IFB BNot  $b$  THEN  $e2$  ELSE  $e1$  FI).

Proof.

*Admitted.*  
 $\square$

For *WHILE* loops, we can give a similar pair of theorems. A loop whose guard is  
 equivalent to BFalse is equivalent to *SKIP*, while a loop whose guard is equivalent to BTrue  
 is equivalent to *WHILE* BTrue *DO SKIP END* (or any other non-terminating program).  
 The first of these facts is easy.

Theorem WHILE\_false :  $\forall b\ c$ ,  
 bequiv  $b$  BFalse  $\rightarrow$



```

cequiv
  (WHILE b DO c END)
SKIP.

```

Proof.

```

intros b c Hb. split; intros H.
-
  inversion H; subst.
+
  apply E_Skip.
+
  rewrite Hb in H2. inversion H2.
-
  inversion H; subst.
  apply E_WhileEnd.
  rewrite Hb.
  reflexivity. Qed.

```

**Exercise: 2 stars, advanced, optional (WHILE\_false\_informal)** Write an informal proof of WHILE\_false.

□

To prove the second fact, we need an auxiliary lemma stating that *WHILE* loops whose guards are equivalent to BTrue never terminate:

*Lemma:* If *b* is equivalent to BTrue, then it cannot be the case that  $(WHILE\ b\ DO\ c\ END) / st \\\ st'$ .

*Proof:* Suppose that  $(WHILE\ b\ DO\ c\ END) / st \\\ st'$ . We show, by induction on a derivation of  $(WHILE\ b\ DO\ c\ END) / st \\\ st'$ , that this assumption leads to a contradiction.

- Suppose  $(WHILE\ b\ DO\ c\ END) / st \\\ st'$  is proved using rule E\_WhileEnd. Then by assumption  $beval\ st\ b = false$ . But this contradicts the assumption that *b* is equivalent to BTrue.
- Suppose  $(WHILE\ b\ DO\ c\ END) / st \\\ st'$  is proved using rule E\_WhileLoop. Then we are given the induction hypothesis that  $(WHILE\ b\ DO\ c\ END) / st \\\ st'$  is contradictory, which is exactly what we are trying to prove!
- Since these are the only rules that could have been used to prove  $(WHILE\ b\ DO\ c\ END) / st \\\ st'$ , the other cases of the induction are immediately contradictory. □

```

Lemma WHILE_true_nonterm : ∀ b c st st',
  bequiv b BTrue →
  ~ ( (WHILE b DO c END) / st \\\ st' ).

```

Proof.

```

intros b c st st' Hb.

```

```

intros H.
remember (WHILE b DO c END) as cw eqn:Heqcw.
induction H;

```

```

    inversion Heqcw; subst; clear Heqcw.
-
    unfold bequiv in Hb.
    rewrite Hb in H. inversion H.
-
    apply IHceval2. reflexivity. Qed.

```

**Exercise: 2 stars, optional (WHILE\_true\_nonterm\_informal)** Explain what the lemma WHILE\_true\_nonterm means in English.

□

**Exercise: 2 stars, recommended (WHILE\_true)** Prove the following theorem. *Hint:* You'll want to use WHILE\_true\_nonterm here.

Theorem WHILE\_true:  $\forall b \ c,$   
     bequiv  $b$  BTrue  $\rightarrow$   
     cequiv  
         (WHILE  $b$  DO  $c$  END)  
         (WHILE BTrue DO SKIP END).

Proof.

*Admitted.*

□

Theorem loop\_unrolling:  $\forall b \ c,$   
     cequiv  
         (WHILE  $b$  DO  $c$  END)  
         (IFB  $b$  THEN ( $c;;$  WHILE  $b$  DO  $c$  END) ELSE SKIP FI).

Proof.

```

intros b c st st'.
split; intros Hce.
-
    inversion Hce; subst.
    +
        apply E_IfFalse. assumption. apply E_Skip.
    +
        apply E_IfTrue. assumption.
        apply E_Seq with (st' := st'0). assumption. assumption.
-
    inversion Hce; subst.
    +

```

```

inversion H5; subst.
apply E_WhileLoop with (st' := st'0).
assumption. assumption. assumption.
+
inversion H5; subst. apply E_WhileEnd. assumption. Qed.

```

**Exercise: 2 stars, optional (seq\_assoc)** Theorem seq\_assoc :  $\forall c1\ c2\ c3$ ,  
 cequiv  $((c1;;c2);;c3)\ (c1;;(c2;;c3))$ .

Proof.

*Admitted.*

□

Proving program properties involving assignments is one place where the functional-extensionality axiom introduced in the Logic chapter often comes in handy. Here are an example and an exercise.

Theorem identity\_assignment :  $\forall (X:\text{id})$ ,

cequiv

$(X ::= \text{Ald } X)$

SKIP.

Proof.

intros. split; intro H.

-

```

inversion H; subst. simpl.
replace (t_update st X (st X)) with st.
constructor.
apply functional_extensionality. intro.
rewrite t_update_same; reflexivity.

```

-

```

inversion H; subst.
assert (st' = (t_update st' X (st' X))).
  apply functional_extensionality. intro.
  rewrite t_update_same; reflexivity.
rewrite H0 at 2.
constructor. reflexivity.

```

Qed.

**Exercise: 2 stars, recommended (assign\_aequiv)** Theorem assign\_aequiv :  $\forall X\ e$ ,  
 aequiv  $(\text{Ald } X)\ e \rightarrow$   
 cequiv SKIP  $(X ::= e)$ .

Proof.

*Admitted.*

□

## 18.3 Properties of Behavioral Equivalence

We now turn to developing some of the properties of the program equivalences we have defined.

### 18.3.1 Behavioral Equivalence is an Equivalence

First, we verify that the equivalences on *aexprs*, *bexprs*, and **coms** really are *equivalences* – i.e., that they are reflexive, symmetric, and transitive. The proofs are all easy.

Lemma `refl_aequiv` :  $\forall (a : \mathbf{aexp}), \text{aequiv } a \ a$ .

Proof.

`intros a st. reflexivity. Qed.`

Lemma `sym_aequiv` :  $\forall (a1 \ a2 : \mathbf{aexp}),$

`aequiv a1 a2  $\rightarrow$  aequiv a2 a1.`

Proof.

`intros a1 a2 H. intros st. symmetry. apply H. Qed.`

Lemma `trans_aequiv` :  $\forall (a1 \ a2 \ a3 : \mathbf{aexp}),$

`aequiv a1 a2  $\rightarrow$  aequiv a2 a3  $\rightarrow$  aequiv a1 a3.`

Proof.

`unfold aequiv. intros a1 a2 a3 H12 H23 st.  
rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.`

Lemma `refl_bequiv` :  $\forall (b : \mathbf{bexp}), \text{bequiv } b \ b$ .

Proof.

`unfold bequiv. intros b st. reflexivity. Qed.`

Lemma `sym_bequiv` :  $\forall (b1 \ b2 : \mathbf{bexp}),$

`bequiv b1 b2  $\rightarrow$  bequiv b2 b1.`

Proof.

`unfold bequiv. intros b1 b2 H. intros st. symmetry. apply H. Qed.`

Lemma `trans_bequiv` :  $\forall (b1 \ b2 \ b3 : \mathbf{bexp}),$

`bequiv b1 b2  $\rightarrow$  bequiv b2 b3  $\rightarrow$  bequiv b1 b3.`

Proof.

`unfold bequiv. intros b1 b2 b3 H12 H23 st.  
rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.`

Lemma `refl_cequiv` :  $\forall (c : \mathbf{com}), \text{cequiv } c \ c$ .

Proof.

`unfold cequiv. intros c st st'. apply iff_refl. Qed.`

Lemma `sym_cequiv` :  $\forall (c1 \ c2 : \mathbf{com}),$

`cequiv c1 c2  $\rightarrow$  cequiv c2 c1.`

Proof.

`unfold cequiv. intros c1 c2 H st st'.`

```

    assert (c1 / st \\\ st'  $\leftrightarrow$  c2 / st \\\ st') as H'.
    { apply H. }
    apply iff_sym. assumption.
Qed.

Lemma iff_trans :  $\forall$  (P1 P2 P3 : Prop),
  (P1  $\leftrightarrow$  P2)  $\rightarrow$  (P2  $\leftrightarrow$  P3)  $\rightarrow$  (P1  $\leftrightarrow$  P3).
Proof.
  intros P1 P2 P3 H12 H23.
  inversion H12. inversion H23.
  split; intros A.
  apply H1. apply H. apply A.
  apply H0. apply H2. apply A. Qed.

Lemma trans_cequiv :  $\forall$  (c1 c2 c3 : com),
  cequiv c1 c2  $\rightarrow$  cequiv c2 c3  $\rightarrow$  cequiv c1 c3.
Proof.
  unfold cequiv. intros c1 c2 c3 H12 H23 st st'.
  apply iff_trans with (c2 / st \\\ st'). apply H12. apply H23. Qed.

```

### 18.3.2 Behavioral Equivalence is a Congruence

Less obviously, behavioral equivalence is also a *congruence*. That is, the equivalence of two subprograms implies the equivalence of the larger programs in which they are embedded:  $\text{aequiv } a1 \ a1'$

---

```

cequiv (i ::= a1) (i ::= a1')
  cequiv c1 c1' cequiv c2 c2'

```

---

$\text{cequiv } (c1;;c2) \ (c1';;c2')$  ...and so on for the other forms of commands.

(Note that we are using the inference rule notation here not as part of a definition, but simply to write down some valid implications in a readable format. We prove these implications below.)

We will see a concrete example of why these congruence properties are important in the following section (in the proof of `fold_constants_com_sound`), but the main idea is that they allow us to replace a small part of a large program with an equivalent small part and know that the whole large programs are equivalent *without* doing an explicit proof about the non-varying parts – i.e., the “proof burden” of a small change to a large program is proportional to the size of the change, not the program.

```

Theorem CAss_congruence :  $\forall$  i a1 a1',
  aequiv a1 a1'  $\rightarrow$ 
  cequiv (CAss i a1) (CAss i a1').

```

```

Proof.
  intros i a1 a2 Heqv st st'.

```

```

split; intros Hceval.
-
  inversion Hceval. subst. apply E_Ass.
  rewrite Heqv. reflexivity.
-
  inversion Hceval. subst. apply E_Ass.
  rewrite Heqv. reflexivity. Qed.

```

The congruence property for loops is a little more interesting, since it requires induction.

*Theorem:* Equivalence is a congruence for *WHILE* – that is, if  $b1$  is equivalent to  $b1'$  and  $c1$  is equivalent to  $c1'$ , then *WHILE*  $b1$  *DO*  $c1$  *END* is equivalent to *WHILE*  $b1'$  *DO*  $c1'$  *END*.

*Proof:* Suppose  $b1$  is equivalent to  $b1'$  and  $c1$  is equivalent to  $c1'$ . We must show, for every  $st$  and  $st'$ , that *WHILE*  $b1$  *DO*  $c1$  *END* /  $st \parallel st'$  iff *WHILE*  $b1'$  *DO*  $c1'$  *END* /  $st \parallel st'$ . We consider the two directions separately.

- ( $\rightarrow$ ) We show that *WHILE*  $b1$  *DO*  $c1$  *END* /  $st \parallel st'$  implies *WHILE*  $b1'$  *DO*  $c1'$  *END* /  $st \parallel st'$ , by induction on a derivation of *WHILE*  $b1$  *DO*  $c1$  *END* /  $st \parallel st'$ . The only nontrivial cases are when the final rule in the derivation is *E\_WhileEnd* or *E\_WhileLoop*.
  - *E\_WhileEnd*: In this case, the form of the rule gives us  $\text{beval } st \ b1 = \text{false}$  and  $st = st'$ . But then, since  $b1$  and  $b1'$  are equivalent, we have  $\text{beval } st \ b1' = \text{false}$ , and *E\_WhileEnd* applies, giving us *WHILE*  $b1'$  *DO*  $c1'$  *END* /  $st \parallel st'$ , as required.
  - *E\_WhileLoop*: The form of the rule now gives us  $\text{beval } st \ b1 = \text{true}$ , with  $c1 / st \parallel st'0$  and *WHILE*  $b1$  *DO*  $c1$  *END* /  $st'0 \parallel st'$  for some state  $st'0$ , with the induction hypothesis *WHILE*  $b1'$  *DO*  $c1'$  *END* /  $st'0 \parallel st'$ . Since  $c1$  and  $c1'$  are equivalent, we know that  $c1' / st \parallel st'0$ . And since  $b1$  and  $b1'$  are equivalent, we have  $\text{beval } st \ b1' = \text{true}$ . Now *E\_WhileLoop* applies, giving us *WHILE*  $b1'$  *DO*  $c1'$  *END* /  $st \parallel st'$ , as required.
- ( $\leftarrow$ ) Similar.  $\square$

**Theorem CWhile\_congruence** :  $\forall b1 \ b1' \ c1 \ c1'$ ,  
 $\text{bequiv } b1 \ b1' \rightarrow \text{cequiv } c1 \ c1' \rightarrow$   
 $\text{cequiv } (\text{WHILE } b1 \text{ DO } c1 \text{ END}) (\text{WHILE } b1' \text{ DO } c1' \text{ END}).$

*Proof.*

```

unfold bequiv,cequiv.
intros b1 b1' c1 c1' Hb1e Hc1e st st'.
split; intros Hce.
-
  remember (WHILE b1 DO c1 END) as cwhile
  eqn:Heqcwhile.

```

```

induction Hce; inversion Heqcwhile; subst.
+
  apply E_WhileEnd. rewrite ← Hb1e. apply H.
+
  apply E_WhileLoop with (st' := st').
  × rewrite ← Hb1e. apply H.
  ×
    apply (Hc1e st st'). apply Hce1.
  ×
    apply IHHce2. reflexivity.
-
  remember (WHILE b1' DO c1' END) as c'while
    eqn:Heqc'while.
  induction Hce; inversion Heqc'while; subst.
+
  apply E_WhileEnd. rewrite → Hb1e. apply H.
+
  apply E_WhileLoop with (st' := st').
  × rewrite → Hb1e. apply H.
  ×
    apply (Hc1e st st'). apply Hce1.
  ×
    apply IHHce2. reflexivity. Qed.

```

**Exercise: 3 stars, optional (CSeq\_congruence)** Theorem CSeq\_congruence :  $\forall c1\ c1'\ c2\ c2'$ ,

$\text{cequiv } c1\ c1' \rightarrow \text{cequiv } c2\ c2' \rightarrow$   
 $\text{cequiv } (c1;;c2)\ (c1';;c2').$

Proof.

*Admitted.*

□

**Exercise: 3 stars (CIf\_congruence)** Theorem CIf\_congruence :  $\forall b\ b'\ c1\ c1'\ c2\ c2'$ ,  
 $\text{bequiv } b\ b' \rightarrow \text{cequiv } c1\ c1' \rightarrow \text{cequiv } c2\ c2' \rightarrow$   
 $\text{cequiv } (\text{IFB } b\ \text{THEN } c1\ \text{ELSE } c2\ \text{FI})$   
 $(\text{IFB } b'\ \text{THEN } c1'\ \text{ELSE } c2'\ \text{FI}).$

Proof.

*Admitted.*

□

For example, here are two equivalent programs and a proof of their equivalence...

Example congruence\_example:

cequiv

```

(X ::= ANum 0;;
 IFB (BEq (Ald X) (ANum 0))
 THEN
   Y ::= ANum 0
 ELSE
   Y ::= ANum 42
 FI)

```

```

(X ::= ANum 0;;
 IFB (BEq (Ald X) (ANum 0))
 THEN
   Y ::= AMinus (Ald X) (Ald X)
 ELSE
   Y ::= ANum 42
 FI).

```

Proof.

```

apply CSeq_congruence.
apply refl_cequiv.
apply Clf_congruence.
apply refl_bequiv.
apply CAss_congruence. unfold aequiv. simpl.
  symmetry. apply minus_diag.
apply refl_cequiv.

```

Qed.

## 18.4 Program Transformations

A *program transformation* is a function that takes a program as input and produces some variant of the program as output. Compiler optimizations such as constant folding are a canonical example, but there are many others.

A program transformation is *sound* if it preserves the behavior of the original program.

**Definition** `atrans_sound` (*atrans* : **aexp** → **aexp**) : Prop :=  
 $\forall (a : \mathbf{aexp}),$   
 $\text{aequiv } a \text{ (atrans } a).$

**Definition** `btrans_sound` (*btrans* : **bexp** → **bexp**) : Prop :=  
 $\forall (b : \mathbf{bexp}),$   
 $\text{bequiv } b \text{ (btrans } b).$

**Definition** `ctrans_sound` (*ctrans* : **com** → **com**) : Prop :=  
 $\forall (c : \mathbf{com}),$   
 $\text{cequiv } c \text{ (ctrans } c).$



### 18.4.1 The Constant-Folding Transformation

An expression is *constant* when it contains no variable references.

Constant folding is an optimization that finds constant expressions and replaces them by their values.

```
Fixpoint fold_constants_aexp (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | Ald i ⇒ Ald i
  | APlus a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
    | (a1', a2') ⇒ APlus a1' a2'
    end
  | AMinus a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
    | (a1', a2') ⇒ AMinus a1' a2'
    end
  | AMult a1 a2 ⇒
    match (fold_constants_aexp a1, fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 × n2)
    | (a1', a2') ⇒ AMult a1' a2'
    end
  end
end.
```

```
Example fold_aexp_ex1 :
  fold_constants_aexp
    (AMult (APlus (ANum 1) (ANum 2)) (Ald X))
  = AMult (ANum 3) (Ald X).
```

Proof. reflexivity. Qed.

Note that this version of constant folding doesn't eliminate trivial additions, etc. – we are focusing attention on a single optimization for the sake of simplicity. It is not hard to incorporate other ways of simplifying expressions; the definitions and proofs just get longer.

```
Example fold_aexp_ex2 :
  fold_constants_aexp
    (AMinus (Ald X) (APlus (AMult (ANum 0) (ANum 6))
                          (Ald Y)))
  = AMinus (Ald X) (APlus (ANum 0) (Ald Y)).
```

Proof. reflexivity. Qed.

Not only can we lift `fold_constants_aexp` to **bexps** (in the `BEq` and `BLe` cases), we can also find constant *boolean* expressions and reduce them in-place.

```
Fixpoint fold_constants_bexp (b : bexp) : bexp :=
  match b with
  | BTrue ⇒ BTrue
  | BFalse ⇒ BFalse
  | BEq a1 a2 ⇒
    match (fold_constants_aexp a1 , fold_constants_aexp a2)
    with
    | (ANum n1 , ANum n2) ⇒
      if beq_nat n1 n2 then BTrue else BFalse
    | (a1' , a2') ⇒
      BEq a1' a2'
    end
  | BLe a1 a2 ⇒
    match (fold_constants_aexp a1 , fold_constants_aexp a2)
    with
    | (ANum n1 , ANum n2) ⇒
      if leb n1 n2 then BTrue else BFalse
    | (a1' , a2') ⇒
      BLe a1' a2'
    end
  | BNot b1 ⇒
    match (fold_constants_bexp b1) with
    | BTrue ⇒ BFalse
    | BFalse ⇒ BTrue
    | b1' ⇒ BNot b1'
    end
  | BAnd b1 b2 ⇒
    match (fold_constants_bexp b1 , fold_constants_bexp b2)
    with
    | (BTrue , BTrue) ⇒ BTrue
    | (BTrue , BFalse) ⇒ BFalse
    | (BFalse , BTrue) ⇒ BFalse
    | (BFalse , BFalse) ⇒ BFalse
    | (b1' , b2') ⇒ BAnd b1' b2'
    end
  end.
```

```
Example fold_bexp_ex1 :
  fold_constants_bexp (BAnd BTrue (BNot (BAnd BFalse BTrue)))
  = BTrue.
```

Proof. reflexivity. Qed.

Example fold\_bexp\_ex2 :

```

fold_constants_bexp
  (BAnd (BEq (Ald X) (Ald Y))
        (BEq (ANum 0)
              (AMinus (ANum 2) (APlus (ANum 1)
                                       (ANum 1))))))

```

= BAnd (BEq (Ald X) (Ald Y)) BTrue.

Proof. reflexivity. Qed.

To fold constants in a command, we apply the appropriate folding functions on all embedded expressions.

```

Fixpoint fold_constants_com (c : com) : com :=
  match c with
  | SKIP =>
    SKIP
  | i ::= a =>
    CAss i (fold_constants_aexp a)
  | c1 ;; c2 =>
    (fold_constants_com c1) ;; (fold_constants_com c2)
  | IFB b THEN c1 ELSE c2 FI =>
    match fold_constants_bexp b with
    | BTrue => fold_constants_com c1
    | BFalse => fold_constants_com c2
    | b' => IFB b' THEN fold_constants_com c1
              ELSE fold_constants_com c2 FI
    end
  | WHILE b DO c END =>
    match fold_constants_bexp b with
    | BTrue => WHILE BTrue DO SKIP END
    | BFalse => SKIP
    | b' => WHILE b' DO (fold_constants_com c) END
    end
  end.

```

Example fold\_com\_ex1 :

```

fold_constants_com

(X ::= APlus (ANum 4) (ANum 5));;
Y ::= AMinus (Ald X) (ANum 3);;
IFB BEq (AMinus (Ald X) (Ald Y))
  (APlus (ANum 2) (ANum 4)) THEN
  SKIP
ELSE
  Y ::= ANum 0

```

```

FI;;
IFB BLe (ANum 0)
    (AMinus (ANum 4) (APlus (ANum 2) (ANum 1)))
THEN
    Y ::= ANum 0
ELSE
    SKIP
FI;;
WHILE BEq (Ald Y) (ANum 0) DO
    X ::= APlus (Ald X) (ANum 1)
END)
=
(X ::= ANum 9;;
 Y ::= AMinus (Ald X) (ANum 3));;
IFB BEq (AMinus (Ald X) (Ald Y)) (ANum 6) THEN
    SKIP
ELSE
    (Y ::= ANum 0)
FI;;
Y ::= ANum 0;;
WHILE BEq (Ald Y) (ANum 0) DO
    X ::= APlus (Ald X) (ANum 1)
END).

```

Proof. reflexivity. Qed.

## 18.4.2 Soundness of Constant Folding

Now we need to show that what we've done is correct.

Here's the proof for arithmetic expressions:

Theorem `fold_constants_aexp_sound` :  
`atrans_sound fold_constants_aexp.`

Proof.

```

unfold atrans_sound. intros a. unfold aequiv. intros st.
induction a; simpl;

try reflexivity;

try (destruct (fold_constants_aexp a1);
    destruct (fold_constants_aexp a2);
    rewrite IHa1; rewrite IHa2; reflexivity). Qed.

```

**Exercise: 3 stars, optional (fold\_bexp\_Eq\_informal)** Here is an informal proof of the BEq case of the soundness argument for boolean expression constant folding. Read it carefully and compare it to the formal proof that follows. Then fill in the BLe case of the formal proof (without looking at the BEq case, if possible).

*Theorem:* The constant folding function for booleans, `fold_constants_bexp`, is sound.

*Proof:* We must show that `b` is equivalent to `fold_constants_bexp b`, for all boolean expressions `b`. Proceed by induction on `b`. We show just the case where `b` has the form `BEq a1 a2`.

In this case, we must show `beval st (BEq a1 a2) = beval st (fold_constants_bexp (BEq a1 a2))`. There are two cases to consider:

- First, suppose `fold_constants_aexp a1 = ANum n1` and `fold_constants_aexp a2 = ANum n2` for some `n1` and `n2`.

In this case, we have `fold_constants_bexp (BEq a1 a2) = if beq_nat n1 n2 then BTrue else BFalse` and `beval st (BEq a1 a2) = beq_nat (aeval st a1) (aeval st a2)`. By the soundness of constant folding for arithmetic expressions (Lemma `fold_constants_aexp_sound`), we know `aeval st a1 = aeval st (fold_constants_aexp a1) = aeval st (ANum n1) = n1` and `aeval st a2 = aeval st (fold_constants_aexp a2) = aeval st (ANum n2) = n2`, so `beval st (BEq a1 a2) = beq_nat (aeval st a1) (aeval st a2) = beq_nat n1 n2`. Also, it is easy to see (by considering the cases `n1 = n2` and `n1 ≠ n2` separately) that `beval st (if beq_nat n1 n2 then BTrue else BFalse) = if beq_nat n1 n2 then beval st BTrue else beval st BFalse = if beq_nat n1 n2 then true else false = beq_nat n1 n2`. So `beval st (BEq a1 a2) = beq_nat n1 n2 = beval st (if beq_nat n1 n2 then BTrue else BFalse)`, as required.

- Otherwise, one of `fold_constants_aexp a1` and `fold_constants_aexp a2` is not a constant. In this case, we must show `beval st (BEq a1 a2) = beval st (BEq (fold_constants_aexp a1) (fold_constants_aexp a2))`, which, by the definition of `beval`, is the same as showing `beq_nat (aeval st a1) (aeval st a2) = beq_nat (aeval st (fold_constants_aexp a1)) (aeval st (fold_constants_aexp a2))`. But the soundness of constant folding for arithmetic expressions (`fold_constants_aexp_sound`) gives us `aeval st a1 = aeval st (fold_constants_aexp a1)` and `aeval st a2 = aeval st (fold_constants_aexp a2)`, completing the case.  $\square$

**Theorem `fold_constants_bexp_sound`:**

`btrans_sound fold_constants_bexp.`

*Proof.*

```

unfold btrans_sound. intros b. unfold bequiv. intros st.
induction b;

  try reflexivity.
-
  rename a into a1. rename a0 into a2. simpl.
```

(Doing induction when there are a lot of constructors makes specifying variable names a chore, but Coq doesn't always choose nice variable names. We can rename entries in the context with the `rename` tactic: `rename a into a1` will change `a` to `a1` in the current goal and context.)

```
remember (fold_constants_aexp a1) as a1' eqn:Heqa1'.
remember (fold_constants_aexp a2) as a2' eqn:Heqa2'.
replace (aeval st a1) with (aeval st a1') by
  (subst a1'; rewrite ← fold_constants_aexp_sound; reflexivity).
replace (aeval st a2) with (aeval st a2') by
  (subst a2'; rewrite ← fold_constants_aexp_sound; reflexivity).
destruct a1'; destruct a2'; try reflexivity.
```

The only interesting case is when both `a1` and `a2` become constants after folding

```
simpl. destruct (beq_nat n n0); reflexivity.
-
  admit.
-
  simpl. remember (fold_constants_bexp b) as b' eqn:Heqb'.
  rewrite IHb.
  destruct b'; reflexivity.
-
  simpl.
  remember (fold_constants_bexp b1) as b1' eqn:Heqb1'.
  remember (fold_constants_bexp b2) as b2' eqn:Heqb2'.
  rewrite IHb1. rewrite IHb2.
  destruct b1'; destruct b2'; reflexivity.
Admitted.
□
```

**Exercise: 3 stars (fold\_constants\_com\_sound)** Complete the *WHILE* case of the following proof.

Theorem `fold_constants_com_sound` :

`ctrans_sound fold_constants_com.`

Proof.

```
unfold ctrans_sound. intros c.
induction c; simpl.
- apply refl_cequiv.
- apply CAss_congruence.
  apply fold_constants_aexp_sound.
- apply CSeq_congruence; assumption.
-
  assert (bequiv b (fold_constants_bexp b)). {
```

```

    apply fold_constants_bexp_sound. }
destruct (fold_constants_bexp b) eqn:Heqb;
try (apply Clf_congruence; assumption).

```

(If the optimization doesn't eliminate the if, then the result is easy to prove from the IH and fold\_constants\_bexp\_sound.)

```

+
  apply trans_cequiv with c1; try assumption.
  apply IFB_true; assumption.
+
  apply trans_cequiv with c2; try assumption.
  apply IFB_false; assumption.
-

```

*Admitted.*

□

## Soundness of (0 + n) Elimination, Redux

**Exercise: 4 stars, advanced, optional (optimize\_0plus)** Recall the definition `optimize_0plus` from the `Imp` chapter: Fixpoint `optimize_0plus (e:aexp) : aexp := match e with | ANum n => ANum n | APlus (ANum 0) e2 => optimize_0plus e2 | APlus e1 e2 => APlus (optimize_0plus e1) (optimize_0plus e2) | AMinus e1 e2 => AMinus (optimize_0plus e1) (optimize_0plus e2) | AMult e1 e2 => AMult (optimize_0plus e1) (optimize_0plus e2) end`. Note that this function is defined over the old **aexps**, without states.

Write a new version of this function that accounts for variables, plus analogous ones for **bexps** and commands: `optimize_0plus_aexp` `optimize_0plus_bexp` `optimize_0plus_com`. Prove that these three functions are sound, as we did for `fold_constants_*`. Make sure you use the congruence lemmas in the proof of `optimize_0plus_com` – otherwise it will be *long*!

Then define an optimizer on commands that first folds constants (using `fold_constants_com`) and then eliminates 0 + n terms (using `optimize_0plus_com`).

- Give a meaningful example of this optimizer's output.
- Prove that the optimizer is sound. (This part should be *very* easy.)

□

## 18.5 Proving That Programs Are *Not* Equivalent

Suppose that `c1` is a command of the form `X ::= a1;; Y ::= a2` and `c2` is the command `X ::= a1;; Y ::= a2'`, where `a2'` is formed by substituting `a1` for all occurrences of `X` in `a2`. For example, `c1` and `c2` might be: `c1 = (X ::= 42 + 53;; Y ::= Y + X)` `c2 = (X ::= 42 + 53;; Y ::= Y + (42 + 53))`. Clearly, this *particular* `c1` and `c2` are equivalent. Is this true in general?

We will see in a moment that it is not, but it is worthwhile to pause, now, and see if you can find a counter-example on your own.

Here, formally, is the function that substitutes an arithmetic expression for each occurrence of a given variable in another expression:

```
Fixpoint subst_aexp (i : id) (u : aexp) (a : aexp) : aexp :=
  match a with
  | ANum n =>
    ANum n
  | AId i' =>
    if beq_id i i' then u else AId i'
  | APlus a1 a2 =>
    APlus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMinus a1 a2 =>
    AMinus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMult a1 a2 =>
    AMult (subst_aexp i u a1) (subst_aexp i u a2)
  end.
```

```
Example subst_aexp_ex :
  subst_aexp X (APlus (ANum 42) (ANum 53))
    (APlus (AId Y) (AId X))
= (APlus (AId Y) (APlus (ANum 42) (ANum 53))).
```

Proof. reflexivity. Qed.

And here is the property we are interested in, expressing the claim that commands `c1` and `c2` as described above are always equivalent.

```
Definition subst_equiv_property := ∀ i1 i2 a1 a2,
  cequiv (i1 ::= a1;; i2 ::= a2)
    (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2).
```

Sadly, the property does *not* always hold.

*Theorem:* It is not the case that, for all `i1`, `i2`, `a1`, and `a2`, `cequiv (i1 ::= a1;; i2 ::= a2) (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2)`. *Proof:* Suppose, for a contradiction, that for all `i1`, `i2`, `a1`, and `a2`, we have `cequiv (i1 ::= a1;; i2 ::= a2) (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2)`. Consider the following program: `X ::= APlus (AId X) (ANum 1);; Y ::= AId X` Note that `(X ::= APlus (AId X) (ANum 1);; Y ::= AId X) / empty_state \ st1`, where `st1 = { X |-> 1, Y |-> 1 }`.

By our assumption, we know that `cequiv (X ::= APlus (AId X) (ANum 1);; Y ::= AId X) (X ::= APlus (AId X) (ANum 1);; Y ::= APlus (AId X) (ANum 1))` so, by the definition of `cequiv`, we have `(X ::= APlus (AId X) (ANum 1);; Y ::= APlus (AId X) (ANum 1)) / empty_state \ st1`. But we can also derive `(X ::= APlus (AId X) (ANum 1);; Y ::= APlus (AId X) (ANum 1)) / empty_state \ st2`, where `st2 = { X |-> 1, Y |-> 2 }`. Note that `st1 ≠ st2`; this is a contradiction, since **ceval** is deterministic! □

Theorem subst\_inequiv :



```

  ¬ subst_equiv_property.
Proof.
  unfold subst_equiv_property.
  intros Contra.
  remember (X ::= APlus (Ald X) (ANum 1));
    Y ::= Ald X)
    as c1.
  remember (X ::= APlus (Ald X) (ANum 1));
    Y ::= APlus (Ald X) (ANum 1))
    as c2.
  assert (cequiv c1 c2) by (subst; apply Contra).
  remember (t_update (t_update empty_state X 1) Y 1) as st1.
  remember (t_update (t_update empty_state X 1) Y 2) as st2.
  assert (H1: c1 / empty_state \\ st1);
  assert (H2: c2 / empty_state \\ st2);
  try (subst;
    apply E_Seq with (st' := (t_update empty_state X 1));
    apply E_Ass; reflexivity).
  apply H in H1.
  assert (Hcontra: st1 = st2)
    by (apply (ceval_deterministic c2 empty_state); assumption).
  assert (Hcontra': st1 Y = st2 Y)
    by (rewrite Hcontra; reflexivity).
  subst. inversion Hcontra'. Qed.

```

**Exercise: 4 stars, optional (better\_subst\_equiv)** The equivalence we had in mind above was not complete nonsense – it was actually almost right. To make it correct, we just need to exclude the case where the variable *X* occurs in the right-hand-side of the first assignment statement.

```

Inductive var_not_used_in_aexp (X:id) : aexp → Prop :=
| VNUNum: ∀ n, var_not_used_in_aexp X (ANum n)
| VNUld: ∀ Y, X ≠ Y → var_not_used_in_aexp X (Ald Y)
| VNUPlus: ∀ a1 a2,
  var_not_used_in_aexp X a1 →
  var_not_used_in_aexp X a2 →
  var_not_used_in_aexp X (APlus a1 a2)
| VNUMinus: ∀ a1 a2,
  var_not_used_in_aexp X a1 →
  var_not_used_in_aexp X a2 →
  var_not_used_in_aexp X (AMinus a1 a2)
| VNUMult: ∀ a1 a2,

```

`var_not_used_in_aexp`  $X$   $a1 \rightarrow$   
`var_not_used_in_aexp`  $X$   $a2 \rightarrow$   
`var_not_used_in_aexp`  $X$  (`AMult`  $a1$   $a2$ ).

Lemma `aeval_weakening` :  $\forall i \ st \ a \ ni,$   
`var_not_used_in_aexp`  $i \ a \rightarrow$   
`aeval` (`t_update`  $st \ i \ ni$ )  $a = \text{aeval } st \ a$ .

Proof.

*Admitted.*

Using `var_not_used_in_aexp`, formalize and prove a correct version of `subst_equiv_property`.

□

**Exercise: 3 stars, optional (`inequiv_exercise`)** Prove that an infinite loop is not equivalent to `SKIP`

Theorem `inequiv_exercise`:

$\neg \text{cequiv } (\text{WHILE } \text{BTrue} \text{ DO } \text{SKIP} \text{ END}) \text{ SKIP}.$

Proof.

*Admitted.*

□

## 18.6 Extended Exercise: Nondeterministic Imp

As we have seen (in theorem `ceval_deterministic` in the `Imp` chapter), `Imp`'s evaluation relation is deterministic. However, *non*-determinism is an important part of the definition of many real programming languages. For example, in many imperative languages (such as C and its relatives), the order in which function arguments are evaluated is unspecified. The program fragment `x = 0;; f(++x, x)` might call `f` with arguments (1, 0) or (1, 1), depending how the compiler chooses to order things. This can be a little confusing for programmers, but it gives the compiler writer useful freedom.

In this exercise, we will extend `Imp` with a simple nondeterministic command and study how this change affects program equivalence. The new command has the syntax `HAVOC X`, where `X` is an identifier. The effect of executing `HAVOC X` is to assign an *arbitrary* number to the variable `X`, nondeterministically. For example, after executing the program: `HAVOC Y;; Z ::= Y * 2` the value of `Y` can be any number, while the value of `Z` is twice that of `Y` (so `Z` is always even). Note that we are not saying anything about the *probabilities* of the outcomes – just that there are (infinitely) many different outcomes that can possibly happen after executing this nondeterministic code.

In a sense, a variable on which we do `HAVOC` roughly corresponds to an uninitialized variable in a low-level language like C. After the `HAVOC`, the variable holds a fixed but arbitrary number. Most sources of nondeterminism in language definitions are there precisely

because programmers don't care which choice is made (and so it is good to leave it open to the compiler to choose whichever will run faster).

We call this new language *Himp* ("Imp extended with *HAVOC*").

Module HIMP.

To formalize Himp, we first add a clause to the definition of commands.

Inductive **com** : Type :=

| CSkip : **com**  
 | CAss : **id** → **aexp** → **com**  
 | CSeq : **com** → **com** → **com**  
 | Clf : **bexp** → **com** → **com** → **com**  
 | CWhile : **bexp** → **com** → **com**  
 | CHavoc : **id** → **com**.

Notation "'SKIP'" :=

CSkip.

Notation "X '::=' a" :=

(CAss X a) (at level 60).

Notation "c1 ;; c2" :=

(CSeq c1 c2) (at level 80, right associativity).

Notation "'WHILE' b 'DO' c 'END'" :=

(CWhile b c) (at level 80, right associativity).

Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=

(Clf e1 e2 e3) (at level 80, right associativity).

Notation "'HAVOC' l" := (CHavoc l) (at level 60).

**Exercise: 2 stars (himp\_ceval)** Now, we must extend the operational semantics. We have provided a template for the **ceval** relation below, specifying the big-step semantics. What rule(s) must be added to the definition of **ceval** to formalize the behavior of the *HAVOC* command?

Reserved Notation "c1 '/' st '\\ st'"

(at level 40, st at level 39).

Inductive **ceval** : **com** → state → state → Prop :=

| E\_Skip : ∀ st : state, SKIP / st \\ st  
 | E\_Ass : ∀ (st : state) (a1 : **aexp**) (n : **nat**) (X : **id**),  
     aeval st a1 = n →  
     (X ::= a1) / st \\ t\_update st X n  
 | E\_Seq : ∀ (c1 c2 : **com**) (st st' st'' : state),  
     c1 / st \\ st' →  
     c2 / st' \\ st'' →  
     (c1 ;; c2) / st \\ st''  
 | E\_IfTrue : ∀ (st st' : state) (b1 : **bexp**) (c1 c2 : **com**),  
     beval st b1 = **true** →

```

      c1 / st \\\ st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st \\\ st'
| E_IfFalse : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
  beval st b1 = false →
  c2 / st \\\ st' →
  (IFB b1 THEN c1 ELSE c2 FI) / st \\\ st'
| E_WhileEnd : ∀ (b1 : bexp) (st : state) (c1 : com),
  beval st b1 = false →
  (WHILE b1 DO c1 END) / st \\\ st
| E_WhileLoop : ∀ (st st' st'' : state) (b1 : bexp) (c1 : com),
  beval st b1 = true →
  c1 / st \\\ st' →
  (WHILE b1 DO c1 END) / st' \\\ st'' →
  (WHILE b1 DO c1 END) / st \\\ st''

```

where "c1 '/' st '\\\' st'" := (ceval c1 st st').

As a sanity check, the following claims should be provable for your definition:

Example havoc\_example1 : (HAVOC X) / empty\_state \\\ t\_update empty\_state X 0.

Proof.

*Admitted.*

Example havoc\_example2 :

(SKIP;; HAVOC Z) / empty\_state \\\ t\_update empty\_state Z 42.

Proof.

*Admitted.*

□

Finally, we repeat the definition of command equivalence from above:

Definition cequiv (c1 c2 : com) : Prop := ∀ st st' : state,  
 c1 / st \\\ st' ↔ c2 / st \\\ st'.

Let's apply this definition to prove some nondeterministic programs equivalent / inequivalent.

**Exercise: 3 stars (havoc\_swap)** Are the following two programs equivalent?

Definition pXY :=

HAVOC X;; HAVOC Y.

Definition pYX :=

HAVOC Y;; HAVOC X.

If you think they are equivalent, prove it. If you think they are not, prove that.

Theorem pXY\_cequiv\_pYX :

cequiv pXY pYX ∨ ¬cequiv pXY pYX.

Proof. *Admitted.*

□

**Exercise: 4 stars, optional (havoc\_copy)** Are the following two programs equivalent?

Definition ptwice :=

HAVOC X;; HAVOC Y.

Definition pcopy :=

HAVOC X;; Y ::= Ald X.

If you think they are equivalent, then prove it. If you think they are not, then prove that. (Hint: You may find the `assert` tactic useful.)

Theorem ptwice\_cequiv\_pcopy :

cequiv ptwice pcopy  $\vee \neg$ cequiv ptwice pcopy.

Proof. *Admitted.*

□

The definition of program equivalence we are using here has some subtle consequences on programs that may loop forever. What `cequiv` says is that the set of possible *terminating* outcomes of two equivalent programs is the same. However, in a language with nondeterminism, like Himp, some programs always terminate, some programs always diverge, and some programs can nondeterministically terminate in some runs and diverge in others. The final part of the following exercise illustrates this phenomenon.

**Exercise: 5 stars, advanced (p1\_p2\_equiv)** Prove that `p1` and `p2` are equivalent. In this and the following exercises, try to understand why the `cequiv` definition has the behavior it has on these examples.

Definition p1 : **com** :=

WHILE (BNot (BEq (Ald X) (ANum 0))) DO  
  HAVOC Y;;  
  X ::= APlus (Ald X) (ANum 1)  
END.

Definition p2 : **com** :=

WHILE (BNot (BEq (Ald X) (ANum 0))) DO  
  SKIP  
END.

Intuitively, the programs have the same termination behavior: either they loop forever, or they terminate in the same state they started in. We can capture the termination behavior of `p1` and `p2` individually with these lemmas:

Lemma p1\_may\_diverge :  $\forall st\ st', st\ X \neq 0 \rightarrow$   
 $\neg p1\ /\ st\ \backslash\backslash\ st'.$

Proof. *Admitted.*

Lemma p2\_may\_diverge :  $\forall st\ st', st\ X \neq 0 \rightarrow$

¬ p2 / st \\\ st'.

Proof.

*Admitted.*

You should use these lemmas to prove that p1 and p2 are actually equivalent.

Theorem p1\_p2\_equiv : cequiv p1 p2.

Proof. *Admitted.*

□

**Exercise: 4 stars, advanced (p3\_p4\_inequiv)** Prove that the following programs are *not* equivalent.

Definition p3 : com :=

  Z ::= ANum 1;;

  WHILE (BNot (BEq (Ald X) (ANum 0))) DO

    HAVOC X;;

    HAVOC Z

  END.

Definition p4 : com :=

  X ::= (ANum 0);;

  Z ::= (ANum 1).

Theorem p3\_p4\_inequiv : ¬ cequiv p3 p4.

Proof. *Admitted.*

□

**Exercise: 5 stars, advanced, optional (p5\_p6\_equiv)** Definition p5 : com :=

  WHILE (BNot (BEq (Ald X) (ANum 1))) DO

    HAVOC X

  END.

Definition p6 : com :=

  X ::= ANum 1.

Theorem p5\_p6\_equiv : cequiv p5 p6.

Proof. *Admitted.*

□

End HIMP.

## 18.7 Additional Exercises

**Exercise: 4 stars, optional (for\_while\_equiv)** This exercise extends the optional *add\_for\_loop* exercise from the *Imp* chapter, where you were asked to extend the language of commands with C-style *for* loops. Prove that the command: *for* (c1 ; b ; c2) { c3 } is equivalent to: c1 ; WHILE b DO c3 ; c2 END □

**Exercise: 3 stars, optional (swap\_noninterfering\_assignments)** (Hint: You'll need *functional\_extensionality* for this one.)

Theorem swap\_noninterfering\_assignments:  $\forall l1\ l2\ a1\ a2,$

$l1 \neq l2 \rightarrow$

$\text{var\_not\_used\_in\_aexp } l1\ a2 \rightarrow$

$\text{var\_not\_used\_in\_aexp } l2\ a1 \rightarrow$

$\text{cequiv}$

$(l1 ::= a1;; l2 ::= a2)$

$(l2 ::= a2;; l1 ::= a1).$

Proof.

*Admitted.*

□

**Exercise: 4 stars, advanced, optional (capprox)** In this exercise we define an asymmetric variant of program equivalence we call *program approximation*. We say that a program  $c1$  *approximates* a program  $c2$  when, for each of the initial states for which  $c1$  terminates,  $c2$  also terminates and produces the same final state. Formally, program approximation is defined as follows:

Definition capprox ( $c1\ c2 : \mathbf{com}$ ) : Prop :=  $\forall (st\ st' : \text{state}),$

$c1 / st \searrow st' \rightarrow c2 / st \searrow st'.$

For example, the program  $c1 = \text{WHILE } X \neq 1 \text{ DO } X ::= X - 1 \text{ END}$  approximates  $c2 = X ::= 1$ , but  $c2$  does not approximate  $c1$  since  $c1$  does not terminate when  $X = 0$  but  $c2$  does. If two programs approximate each other in both directions, then they are equivalent.

Find two programs  $c3$  and  $c4$  such that neither approximates the other.

Definition c3 :  $\mathbf{com} := \text{admit}.$

Definition c4 :  $\mathbf{com} := \text{admit}.$

Theorem c3\_c4\_different :  $\neg \text{capprox } c3\ c4 \wedge \neg \text{capprox } c4\ c3.$

Proof. *Admitted.*

Find a program  $c_{\min}$  that approximates every other program.

Definition cmin :  $\mathbf{com} :=$

*admit.*

Theorem cmin\_minimal :  $\forall c, \text{capprox } c_{\min}\ c.$

Proof. *Admitted.*

Finally, find a non-trivial property which is preserved by program approximation (when going from left to right).

Definition zprop ( $c : \mathbf{com}$ ) : Prop :=

*admit.*

Theorem zprop\_preserving :  $\forall c\ c',$

$\text{zprop } c \rightarrow \text{capprox } c\ c' \rightarrow \text{zprop } c'.$

Proof. *Admitted.*

□



# Chapter 19

## Library Hoare

### 19.1 Hoare: Hoare Logic, Part I

```
Require Import Coq.Bool.Bool.  
Require Import Coq.Arith.Arith.  
Require Import Coq.Arith.EqNat.  
Require Import Coq.omega.Omega.  
  
Require Import Sflib.  
Require Import Imp.  
Require Import Maps.
```

In the past couple of chapters, we’ve begun applying the mathematical tools developed in the first part of the course to studying the theory of a small programming language, Imp.

- We defined a type of *abstract syntax trees* for Imp, together with an *evaluation relation* (a partial function on states) that specifies the *operational semantics* of programs.

The language we defined, though small, captures some of the key features of full-blown languages like C, C++, and Java, including the fundamental notion of mutable state and some common control structures.

- We proved a number of *metatheoretic properties* – “meta” in the sense that they are properties of the language as a whole, rather than of particular programs in the language. These included:
  - determinism of evaluation
  - equivalence of some different ways of writing down the definitions (e.g., functional and relational definitions of arithmetic expression evaluation)
  - guaranteed termination of certain classes of programs
  - correctness (in the sense of preserving meaning) of a number of useful program transformations

- behavioral equivalence of programs (in the *Equiv* chapter).

If we stopped here, we would already have something useful: a set of tools for defining and discussing programming languages and language features that are mathematically precise, flexible, and easy to work with, applied to a set of key properties. All of these properties are things that language designers, compiler writers, and users might care about knowing. Indeed, many of them are so fundamental to our understanding of the programming languages we deal with that we might not consciously recognize them as "theorems." But properties that seem intuitively obvious can sometimes be quite subtle (sometimes also subtly wrong!).

We'll return to the theme of metatheoretic properties of whole languages later in the book when we discuss *types* and *type soundness*. In this chapter, though, we turn to a different set of issues.

Our goal is to carry out some simple examples of *program verification* – i.e., to use the precise definition of *Imp* to prove formally that particular programs satisfy particular specifications of their behavior. We'll develop a reasoning system called *Floyd-Hoare Logic* – often shortened to just *Hoare Logic* – in which each of the syntactic constructs of *Imp* is equipped with a generic "proof rule" that can be used to reason compositionally about the correctness of programs involving this construct.

Hoare Logic originated in the 1960s, and it continues to be the subject of intensive research right up to the present day. It lies at the core of a multitude of tools that are being used in academia and industry to specify and verify real software systems.

Hoare Logic combines two beautiful ideas: a natural way of writing down *specifications* of programs, and a *compositional proof technique* for proving that programs are correct with respect to such specifications – where by "compositional" we mean that the structure of proofs directly mirrors the structure of the programs that they are about.

## 19.2 Assertions

To talk about specifications of programs, the first thing we need is a way of making *assertions* about properties that hold at particular points during a program's execution – i.e., claims about the current state of the memory when execution reaches that point. Formally, an assertion is just a family of propositions indexed by a **state**.

**Definition**  $\text{Assertion} := \text{state} \rightarrow \text{Prop}$ .

**Exercise: 1 star, optional (assertions)** Module EXASSERTIONS.

Paraphrase the following assertions in English (or your favorite natural language).

**Definition**  $\text{as1} : \text{Assertion} := \text{fun } st \Rightarrow st \ X = 3$ .

**Definition**  $\text{as2} : \text{Assertion} := \text{fun } st \Rightarrow st \ X \leq st \ Y$ .

**Definition**  $\text{as3} : \text{Assertion} :=$

$\text{fun } st \Rightarrow st \ X = 3 \ \vee \ st \ X \leq st \ Y$ .

**Definition**  $\text{as4} : \text{Assertion} :=$

```

fun st => st Z × st Z ≤ st X ∧
  ¬ (((S (st Z)) × (S (st Z))) ≤ st X).

```

Definition as5 : Assertion := fun st => **True**.

Definition as6 : Assertion := fun st => **False**.

End EXASSERTIONS.

□

This way of writing assertions can be a little bit heavy, for two reasons: (1) every single assertion that we ever write is going to begin with `fun st =>`; and (2) this state `st` is the only one that we ever use to look up variables in assertions (we will never need to talk about two different memory states at the same time). For discussing examples informally, we'll adopt some simplifying conventions: we'll drop the initial `fun st =>`, and we'll write just `X` to mean `st X`. Thus, instead of writing

`fun st => (st Z) * (st Z) <= m /\ ~ ((S (st Z)) * (S (st Z)) <= m)` we'll write just `Z * Z <= m /\ ~ ((S Z) * (S Z) <= m)`.

Given two assertions  $P$  and  $Q$ , we say that  $P$  *implies*  $Q$ , written  $P \multimap Q$  (in ASCII,  $P \multimap Q$ ), if, whenever  $P$  holds in some state  $st$ ,  $Q$  also holds.

Definition assert\_implies ( $P \ Q$  : Assertion) : Prop :=

$\forall st, P \ st \rightarrow Q \ st$ .

Notation "P  $\multimap$  Q" := (assert\_implies P Q)

(at level 80) : *hoare\_spec\_scope*.

Open Scope *hoare\_spec\_scope*.

(The *hoare\_spec\_scope* annotation here tells Coq that this notation is not global but is intended to be used in particular contexts. The `Open Scope` tells Coq that this file is one such context.)

We'll also want the "iff" variant of implication between assertions:

Notation "P  $\Leftrightarrow$  Q" :=

( $P \multimap Q \wedge Q \multimap P$ ) (at level 80) : *hoare\_spec\_scope*.

## 19.3 Hoare Triples

Next, we need a way of making formal claims about the behavior of commands.

In general, the behavior of a command is to transform one state to another, so it is natural to express claims about commands in terms of assertions that are true before and after the command executes:

- "If command `c` is started in a state satisfying assertion  $P$ , and if `c` eventually terminates in some final state, then this final state will satisfy the assertion  $Q$ ."

Such a claim is called a *Hoare Triple*. The property  $P$  is called the *precondition* of `c`, while  $Q$  is the *postcondition*. Formally:

**Definition** hoare\_triple

$$(P:\text{Assertion}) (c:\text{com}) (Q:\text{Assertion}) : \text{Prop} :=$$

$$\forall st\ st',$$

$$c / st \ \backslash\ st' \rightarrow$$

$$P\ st \rightarrow$$

$$Q\ st'.$$

Since we'll be working a lot with Hoare triples, it's useful to have a compact notation:  
<sup>1</sup> <sup>c</sup> <sup>2</sup>. (The traditional notation is  $\{P\} c \{Q\}$ , but single braces are already used for other things in Coq.)

**Notation** " $\{\{ P \}\} c \{\{ Q \}\}$ " :=  
 (hoare\_triple  $P\ c\ Q$ ) (at level 90,  $c$  at next level)  
 : hoare\_spec\_scope.

**Exercise: 1 star, optional (triples)** Paraphrase the following Hoare triples in English.

- 1) <sup>3</sup> <sup>c</sup> <sup>4</sup>
  - 2) <sup>5</sup> <sup>c</sup> <sup>6</sup>
  - 3) <sup>7</sup> <sup>c</sup> <sup>8</sup>
  - 4) <sup>9</sup> <sup>c</sup> <sup>10</sup>
  - 5) <sup>11</sup> <sup>c</sup> <sup>12</sup>.
  - 6) <sup>13</sup> <sup>c</sup> <sup>14</sup>  $\parallel$
- 

**Exercise: 1 star, optional (valid\_triples)** Which of the following Hoare triples are *valid* – i.e., the claimed relation between  $P$ ,  $c$ , and  $Q$  is true? 1) <sup>15</sup>  $X ::= 5$  <sup>16</sup>

- 2) <sup>17</sup>  $X ::= X + 1$  <sup>18</sup>

---

- <sup>1</sup>P
- <sup>2</sup>Q
- <sup>3</sup>True
- <sup>4</sup>X=5
- <sup>5</sup>X=m
- <sup>6</sup>X=m+5)
- <sup>7</sup>X<=Y
- <sup>8</sup>Y<=X
- <sup>9</sup>True
- <sup>10</sup>False
- <sup>11</sup>X=m
- <sup>12</sup>Y=real\_factm
- <sup>13</sup>True
- <sup>14</sup>(Z\*Z)<=m/\~(((SZ)\*(SZ))<=m)
- <sup>15</sup>True
- <sup>16</sup>X=5
- <sup>17</sup>X=2
- <sup>18</sup>X=3

```

3) 19 X ::= 5; Y ::= 0 20
4) 21 X ::= 5 22
5) 23 SKIP 24
6) 25 SKIP 26
7) 27 WHILE True DO SKIP END 28
8) 29 WHILE X == 0 DO X ::= X + 1 END 30
9) 31 WHILE X <> 0 DO X ::= X + 1 END 32
□

```

(Note that we're using informal mathematical notations for expressions inside of commands, for readability, rather than their formal **aexp** and **bexp** encodings. We'll continue doing so throughout the chapter.)

To get us warmed up for what's coming, here are two simple facts about Hoare triples.

Theorem `hoare_post_true` :  $\forall (P \ Q : \text{Assertion}) \ c,$   
 $(\forall \ st, \ Q \ st) \rightarrow$   
 $\{\{P\}\} \ c \ \{\{Q\}\}.$

Proof.

```

intros P Q c H. unfold hoare_triple.
intros st st' Heval HP.
apply H. Qed.

```

Theorem `hoare_pre_false` :  $\forall (P \ Q : \text{Assertion}) \ c,$   
 $(\forall \ st, \sim (P \ st)) \rightarrow$   
 $\{\{P\}\} \ c \ \{\{Q\}\}.$

Proof.

```

intros P Q c H. unfold hoare_triple.
intros st st' Heval HP.
unfold not in H. apply H in HP.
inversion HP. Qed.

```

---

```

19True
20X=5
21X=2/\X=3
22X=0
23True
24False
25False
26True
27True
28False
29X=0
30X=1
31X=1
32X=100

```

## 19.4 Proof Rules

The goal of Hoare logic is to provide a *compositional* method for proving the validity of specific Hoare triples. That is, we want the structure of a program's correctness proof to mirror the structure of the program itself. To this end, in the sections below, we'll introduce a rule for reasoning about each of the different syntactic forms of commands in Imp – one for assignment, one for sequencing, one for conditionals, etc. – plus a couple of "structural" rules for gluing things together. We will then be able to prove programs correct using these proof rules, without ever unfolding the definition of `hoare_triple`.

### 19.4.1 Assignment

The rule for assignment is the most fundamental of the Hoare logic proof rules. Here's how it works.

Consider this valid Hoare triple: <sup>33</sup>  $X ::= Y$  <sup>34</sup> In English: if we start out in a state where the value of  $Y$  is 1 and we assign  $Y$  to  $X$ , then we'll finish in a state where  $X$  is 1. That is, the property of being equal to 1 gets transferred from  $Y$  to  $X$ .

Similarly, in <sup>35</sup>  $X ::= Y + Z$  <sup>36</sup> the same property (being equal to one) gets transferred to  $X$  from the expression  $Y + Z$  on the right-hand side of the assignment.

More generally, if  $a$  is *any* arithmetic expression, then <sup>37</sup>  $X ::= a$  <sup>38</sup> is a valid Hoare triple.

This can be made even more general. To conclude that an arbitrary property  $Q$  holds after  $X ::= a$ , we need to assume that  $Q$  holds before  $X ::= a$ , but *with all occurrences of*  $X$  replaced by  $a$  in  $Q$ . This leads to the Hoare rule for assignment <sup>39</sup>  $X ::= a$  <sup>40</sup> where " $Q [X \mapsto a]$ " is pronounced " $Q$  where  $a$  is substituted for  $X$ ".

For example, these are valid applications of the assignment rule: <sup>41</sup>  $X ::= X + 1$  <sup>42</sup>

<sup>43</sup>  $X ::= 3$  <sup>44</sup>

<sup>45</sup>  $X ::= 3$  <sup>46</sup>

To formalize the rule, we must first formalize the idea of "substituting an expression for an Imp variable in an assertion." That is, given a proposition  $P$ , a variable  $X$ , and an arithmetic expression  $a$ , we want to derive another proposition  $P'$  that is just the same as  $P$  except that, wherever  $P$  mentions  $X$ ,  $P'$  should instead mention  $a$ .

---

<sup>33</sup>  $Y=1$   
<sup>34</sup>  $X=1$   
<sup>35</sup>  $Y+Z=1$   
<sup>36</sup>  $X=1$   
<sup>37</sup>  $a=1$   
<sup>38</sup>  $X=1$   
<sup>39</sup>  $Q [X \mapsto a]$   
<sup>40</sup>  $Q$   
<sup>41</sup>  $(X \leq 5) [X \mapsto X+1] \text{ i.e., } X+1 \leq 5$   
<sup>42</sup>  $X \leq 5$   
<sup>43</sup>  $(X=3) [X \mapsto 3] \text{ i.e., } 3=3$   
<sup>44</sup>  $X=3$   
<sup>45</sup>  $(0 \leq X \wedge X \leq 5) [X \mapsto 3] \text{ i.e., } (0 \leq 3 \wedge 3 \leq 5)$   
<sup>46</sup>  $0 \leq X \wedge X \leq 5$

Since  $P$  is an arbitrary Coq proposition, we can't directly "edit" its text. Instead, we can achieve the effect we want by evaluating  $P$  in an updated state:

**Definition** `assn_sub`  $X$   $a$   $P$  : Assertion :=

```
fun (st : state) =>
  P (t_update st X (aeval st a)).
```

**Notation** " $P$  [ $X \mid\rightarrow a$ ]" := (`assn_sub`  $X$   $a$   $P$ ) (at level 10).

That is,  $P$  [ $X \mid\rightarrow a$ ] is an assertion – let's call it  $P'$  – that is just like  $P$  except that, wherever  $P$  looks up the variable  $X$  in the current state,  $P'$  instead uses the value of the expression  $a$ .

To see how this works, let's calculate what happens with a couple of examples. First, suppose  $P'$  is  $(X \leq 5)$  [ $X \mid\rightarrow 3$ ] – that is, more formally,  $P'$  is the Coq expression `fun st => (fun st' => st' X <= 5) (t_update st X (aeval st (ANum 3)))`, which simplifies to `fun st => (fun st' => st' X <= 5) (t_update st X 3)` and further simplifies to `fun st => ((t_update st X 3) X) <= 5` and by further simplification to `fun st => (3 <= 5)`. That is,  $P'$  is the assertion that 3 is less than or equal to 5 (as expected).

For a more interesting example, suppose  $P'$  is  $(X \leq 5)$  [ $X \mid\rightarrow X+1$ ]. Formally,  $P'$  is the Coq expression `fun st => (fun st' => st' X <= 5) (t_update st X (aeval st (APlus (AId X) (ANum 1))))`, which simplifies to `fun st => (((t_update st X (aeval st (APlus (AId X) (ANum 1)))) X) <= 5)` and further simplifies to `fun st => (aeval st (APlus (AId X) (ANum 1))) <= 5`. That is,  $P'$  is the assertion that  $X+1$  is at most 5.

Now we can give the precise proof rule for assignment:

---

(`hoare_asgn`) <sup>47</sup>  $X ::= a$  <sup>48</sup>

We can prove formally that this rule is indeed valid.

**Theorem** `hoare_asgn` :  $\forall Q$   $X$   $a$ ,

$\{\{Q$  [ $X \mid\rightarrow a$ ]\}\} ( $X ::= a$ )  $\{\{Q\}\}$ .

**Proof.**

```
unfold hoare_triple.
intros Q X a st st' HE HQ.
inversion HE. subst.
unfold assn_sub in HQ. assumption. Qed.
```

Here's a first formal proof using this rule.

**Example** `assn_sub_example` :

```
{{(fun st => st X = 3) [X |-> ANum 3]}}
(X ::= (ANum 3))
{(fun st => st X = 3)}.
```

**Proof.**

```
apply hoare_asgn. Qed.
```

---

<sup>47</sup>  $Q[X \mid\rightarrow a]$

<sup>48</sup>  $Q$

**Exercise: 2 stars (hoare\_asgn\_examples)** Translate these informal Hoare triples... 1)

<sup>49</sup>  $X ::= X + 1$  <sup>50</sup>

2) <sup>51</sup>  $X ::= 3$  <sup>52</sup> ...into formal statements (use the names *assn\_sub\_ex1* and *assn\_sub\_ex2*) and use *hoare\_asgn* to prove them.

□

**Exercise: 2 stars (hoare\_asgn\_wrong)** The assignment rule looks backward to almost everyone the first time they see it. If it still seems puzzling, it may help to think a little about alternative "forward" rules. Here is a seemingly natural one:

---

(hoare\_asgn\_wrong) <sup>53</sup>  $X ::= a$  <sup>54</sup> Give a counterexample showing that this rule is incorrect and argue informally that it is really a counterexample. (Hint: The rule universally quantifies over the arithmetic expression *a*, and your counterexample needs to exhibit an *a* for which the rule doesn't work.)

□

**Exercise: 3 stars, advanced (hoare\_asgn\_fwd)** However, by using an auxiliary variable *m* to remember the original value of *X* we can define a Hoare rule for assignment that does, intuitively, "work forwards" rather than backwards.

---

(hoare\_asgn\_fwd) <sup>55</sup>  $X ::= a$  <sup>56</sup> (where  $st' = t\_update\ st\ X\ m$ ) Note that we use the original value of *X* to reconstruct the state *st'* before the assignment took place. Prove that this rule is correct (the first hypothesis is the functional extensionality axiom, which you will need at some point). Also note that this rule is more complicated than *hoare\_asgn*.

**Theorem hoare\_asgn\_fwd :**

$$\begin{aligned} & (\forall \{X\ Y: \text{Type}\} \{f\ g : X \rightarrow Y\}, \\ & \quad (\forall (x: X), f\ x = g\ x) \rightarrow f = g) \rightarrow \\ & \forall m\ a\ P, \\ & \{\{\text{fun } st \Rightarrow P\ st \wedge st\ X = m\}\} \\ & \quad X ::= a \\ & \{\{\text{fun } st \Rightarrow P\ (t\_update\ st\ X\ m) \\ & \quad \wedge st\ X = aeval\ (t\_update\ st\ X\ m)\ a\}\}\}. \end{aligned}$$

**Proof.**

*intros functional\_extensionality m a P.*

---

```

49 (X<=5) [X| ->X+1]
50 X<=5
51 (0<=X/\X<=5) [X| ->3]
52 0<=X/\X<=5
53 True
54 X=a
55 funst=>Pst /\ stX=m
56 funst=>Pst' /\ stX=aevalst'a

```



*Admitted.*

□

**Exercise: 2 stars, advanced (hoare\_asgn\_fwd\_exists)** Another way to define a forward rule for assignment is to existentially quantify over the previous value of the assigned variable.

---

(hoare\_asgn\_fwd\_exists) <sup>57</sup>  $X ::= a$  <sup>58</sup>

Theorem hoare\_asgn\_fwd\_exists :

( $\forall \{X \ Y: \text{Type}\} \{f \ g : X \rightarrow Y\},$   
 $(\forall (x: X), f \ x = g \ x) \rightarrow f = g) \rightarrow$   
 $\forall a \ P,$   
 $\{\{\text{fun } st \Rightarrow P \ st\}\}$   
 $X ::= a$   
 $\{\{\text{fun } st \Rightarrow \exists m, P \ (t\_update \ st \ X \ m) \wedge$   
 $st \ X = aeval \ (t\_update \ st \ X \ m) \ a \ \}\}\}.$

Proof.

intros functional\_extensionality a P.

*Admitted.*

□

## 19.4.2 Consequence

Sometimes the preconditions and postconditions we get from the Hoare rules won't quite be the ones we want in the particular situation at hand – they may be logically equivalent but have a different syntactic form that fails to unify with the goal we are trying to prove, or they actually may be logically weaker (for preconditions) or stronger (for postconditions) than what we need. For instance, while <sup>59</sup>  $X ::= 3$  <sup>60</sup>, follows directly from the assignment rule, <sup>61</sup>  $X ::= 3$  <sup>62</sup> does not. This triple is valid, but it is not an instance of **hoare\_asgn** because **True** and  $(X = 3) [X \mapsto 3]$  are not syntactically equal assertions. However, they are logically equivalent, so if one triple is valid, then the other must certainly be as well. We can capture this observation with the following rule: <sup>63</sup>  $c$  <sup>64</sup>  $P \multimap P'$

---

```

57 fun st => P st
58 fun st => exists m, P (t_update st X m) /\ st X = aeval (t_update st X m) a
59 (X=3) [X|->3]
60 X=3
61 True
62 X=3
63 P',
64 Q

```

(hoare\_consequence\_pre\_equiv) <sup>65</sup> c <sup>66</sup> Taking this line of thought a bit further, we can see that strengthening the precondition or weakening the postcondition of a valid triple always produces another valid triple. This observation is captured by two *Rules of Consequence*. <sup>67</sup>  
<sup>68</sup>  $P \multimap P'$

---

(hoare\_consequence\_pre) <sup>69</sup> c <sup>70</sup>  
<sup>71</sup> c <sup>72</sup>  $Q' \multimap Q$

---

(hoare\_consequence\_post) <sup>73</sup> c <sup>74</sup>  
 Here are the formal versions:

Theorem hoare\_consequence\_pre :  $\forall (P P' Q : \text{Assertion}) \ c,$   
 $\{\{P'\}\} \ c \ \{\{Q\}\} \rightarrow$   
 $P \multimap P' \rightarrow$   
 $\{\{P\}\} \ c \ \{\{Q\}\}.$

Proof.

```
intros P P' Q c Hhoare Himp.
intros st st' Hc HP. apply (Hhoare st st').
assumption. apply Himp. assumption. Qed.
```

Theorem hoare\_consequence\_post :  $\forall (P Q Q' : \text{Assertion}) \ c,$   
 $\{\{P\}\} \ c \ \{\{Q'\}\} \rightarrow$   
 $Q' \multimap Q \rightarrow$   
 $\{\{P\}\} \ c \ \{\{Q\}\}.$

Proof.

```
intros P Q Q' c Hhoare Himp.
intros st st' Hc HP.
apply Himp.
apply (Hhoare st st').
assumption. assumption. Qed.
```

For example, we can use the first consequence rule like this: <sup>75</sup>  $\multimap$  <sup>76</sup>  $X ::= 1$  <sup>77</sup> Or, formally...

---

```
65P
66Q
67P',
68Q
69P
70Q
71P
72Q',
73P
74Q
75True
761=1
77X=1
```

```

Example hoare_asgn_example1 :
  {{fun st => True}} (X ::= (ANum 1)) {{fun st => st X = 1}}.
Proof.
  apply hoare_consequence_pre
    with (P' := (fun st => st X = 1) [X |-> ANum 1]).
  apply hoare_asgn.
  intros st H. unfold assn_sub, t_update. simpl. reflexivity.
Qed.

```

Finally, for convenience in some proofs, we can state a combined rule of consequence that allows us to vary both the precondition and the postcondition at the same time. <sup>78</sup> <sup>c</sup> <sup>79</sup>  $P \multimap P' \ Q' \multimap Q$

---

```

(hoare_consequence) 80 c 81
Theorem hoare_consequence :  $\forall (P \ P' \ Q \ Q' : \text{Assertion}) \ c,$ 
  {{P'}} c {{Q'}}  $\rightarrow$ 
  P  $\multimap$  P'  $\rightarrow$ 
  Q'  $\multimap$  Q  $\rightarrow$ 
  {{P}} c {{Q}}.
Proof.
  intros P P' Q Q' c Hht HPP' HQ'Q.
  apply hoare_consequence_pre with (P' := P').
  apply hoare_consequence_post with (Q' := Q').
  assumption. assumption. assumption. Qed.

```

### 19.4.3 Digression: The `eapply` Tactic

This is a good moment to introduce another convenient feature of Coq. We had to write “`with (P' := ...)`” explicitly in the proof of `hoare_asgn_example1` and `hoare_consequence` above, to make sure that all of the metavariables in the premises to the `hoare_consequence_pre` rule would be set to specific values. (Since  $P'$  doesn't appear in the conclusion of `hoare_consequence_pre`, the process of unifying the conclusion with the current goal doesn't constrain  $P'$  to a specific assertion.)

This is annoying, both because the assertion is a bit long and also because, in `hoare_asgn_example1`, the very next thing we are going to do – applying the `hoare_asgn` rule – will tell us exactly what it should be! We can use `eapply` instead of `apply` to tell Coq, essentially, “Be patient: The missing part is going to be filled in later in the proof.”

```

Example hoare_asgn_example1' :
  {{fun st => True}}

```

---

<sup>78</sup>  $P'$   
<sup>79</sup>  $Q'$   
<sup>80</sup>  $P$   
<sup>81</sup>  $Q$

```
(X ::= (ANum 1))
{{fun st => st X = 1}}.
```

Proof.

```
eapply hoare_consequence_pre.
apply hoare_asgn.
intros st H. reflexivity. Qed.
```

In general, `eapply H` tactic works just like `apply H` except that, instead of failing if unifying the goal with the conclusion of  $H$  does not determine how to instantiate all of the variables appearing in the premises of  $H$ , `eapply H` will replace these variables with *existential variables* (written `?nnn`), which function as placeholders for expressions that will be determined (by further unification) later in the proof.

In order for `Qed` to succeed, all existential variables need to be determined by the end of the proof. Otherwise Coq will (rightly) refuse to accept the proof. Remember that the Coq tactics build proof objects, and proof objects containing existential variables are not complete.

```
Lemma silly1 : ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∀ x y : nat, P x y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
```

Proof.

```
intros P Q HP HQ. eapply HQ. apply HP.
```

Coq gives a warning after `apply HP`. (The warnings look different between Coq 8.4 and Coq 8.5. In 8.4, the warning says "No more subgoals but non-instantiated existential variables." In 8.5, it says "All the remaining goals are on the shelf," meaning that we've finished all our top-level proof obligations but along the way we've put some aside to be done later, and we have not finished those.) Trying to close the proof with `Qed` gives an error.

Abort.

An additional constraint is that existential variables cannot be instantiated with terms containing ordinary variables that did not exist at the time the existential variable was created. (The reason for this technical restriction is that allowing such instantiation would lead to inconsistency of Coq's logic.)

```
Lemma silly2 :
  ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.
```

Proof.

```
intros P Q HP HQ. eapply HQ. destruct HP as [y HP'].
```

Doing `apply HP'` above fails with the following error: Error: Impossible to unify "?175" with "y". In this case there is an easy fix: doing `destruct HP` before doing `eapply HQ`.

Abort.

Lemma silly2\_fixed :

$\forall (P : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop}) (Q : \text{nat} \rightarrow \text{Prop}),$   
 $(\exists y, P \ 42 \ y) \rightarrow$   
 $(\forall x \ y : \text{nat}, P \ x \ y \rightarrow Q \ x) \rightarrow$   
 $Q \ 42.$

Proof.

intros P Q HP HQ. destruct HP as [y HP'].  
 eapply HQ. apply HP'.

Qed.

The `apply HP'` in the last step unifies the existential variable in the goal with the variable `y`.

Note that the `assumption` tactic doesn't work in this case, since it cannot handle existential variables. However, Coq also provides an *eassumption* tactic that solves the goal if one of the premises matches the goal up to instantiations of existential variables. We can use it instead of `apply HP'` if we like.

Lemma silly2\_eassumption :  $\forall (P : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop}) (Q : \text{nat} \rightarrow \text{Prop}),$   
 $(\exists y, P \ 42 \ y) \rightarrow$   
 $(\forall x \ y : \text{nat}, P \ x \ y \rightarrow Q \ x) \rightarrow$   
 $Q \ 42.$

Proof.

intros P Q HP HQ. destruct HP as [y HP']. eapply HQ. *eassumption*.

Qed.

**Exercise: 2 stars (hoare\_asgn\_examples\_2)** Translate these informal Hoare triples...

<sup>82</sup>  $X ::= X + 1$  <sup>83</sup>  $X ::= 3$  <sup>85</sup> ...into formal statements (name them *assn\_sub\_ex1'* and *assn\_sub\_ex2'*) and use `hoare_asgn` and `hoare_consequence_pre` to prove them.

□

## 19.4.4 Skip

Since *SKIP* doesn't change the state, it preserves any property *P*:

---

(hoare\_skip) <sup>86</sup> *SKIP* <sup>87</sup>

Theorem hoare\_skip :  $\forall P,$

---

<sup>82</sup>  $X+1 \leq 5$   
<sup>83</sup>  $X \leq 5$   
<sup>84</sup>  $0 \leq 3 \wedge 3 \leq 5$   
<sup>85</sup>  $0 \leq X \wedge X \leq 5$   
<sup>86</sup>  $P$   
<sup>87</sup>  $P$

$\{\{P\}\} \text{ SKIP } \{\{P\}\}.$

Proof.

```
intros P st st' H HP. inversion H. subst.
assumption. Qed.
```

## 19.4.5 Sequencing

More interestingly, if the command `c1` takes any state where  $P$  holds to a state where  $Q$  holds, and if `c2` takes any state where  $Q$  holds to one where  $R$  holds, then doing `c1` followed by `c2` will take any state where  $P$  holds to one where  $R$  holds: <sup>88</sup> `c1` <sup>89</sup> `c2` <sup>91</sup>

---

(hoare\_seq) <sup>92</sup> `c1;;c2` <sup>93</sup>

Theorem hoare\_seq :  $\forall P Q R c1 c2,$

$\{\{Q\}\} c2 \{\{R\}\} \rightarrow$   
 $\{\{P\}\} c1 \{\{Q\}\} \rightarrow$   
 $\{\{P\}\} c1;;c2 \{\{R\}\}.$

Proof.

```
intros P Q R c1 c2 H1 H2 st st' H12 Pre.
inversion H12; subst.
apply (H1 st'0 st'); try assumption.
apply (H2 st st'0); assumption. Qed.
```

Note that, in the formal rule `hoare_seq`, the premises are given in backwards order (`c2` before `c1`). This matches the natural flow of information in many of the situations where we'll use the rule, since the natural way to construct a Hoare-logic proof is to begin at the end of the program (with the final postcondition) and push postconditions backwards through commands until we reach the beginning.

Informally, a nice way of displaying a proof using the sequencing rule is as a "decorated program" where the intermediate assertion  $Q$  is written between `c1` and `c2`: <sup>94</sup> `X ::= a;` <sup>95</sup> `<— decoration for Q SKIP` <sup>96</sup>

Here's an example of a program involving both assignment and sequencing.

Example hoare\_asgn\_example3 :  $\forall a n,$

$\{\{\text{fun } st \Rightarrow \text{aeval } st \ a = n\}\}$   
 $(X ::= a;; \text{SKIP})$   
 $\{\{\text{fun } st \Rightarrow st \ X = n\}\}.$

---

<sup>88</sup> **P**  
<sup>89</sup> **Q**  
<sup>90</sup> **Q**  
<sup>91</sup> **R**  
<sup>92</sup> **P**  
<sup>93</sup> **R**  
<sup>94</sup> **a=n**  
<sup>95</sup> **X=n**  
<sup>96</sup> **X=n**

Proof.

```
intros a n. eapply hoare_seq.
-
  apply hoare_skip.
-
  eapply hoare_consequence_pre. apply hoare_asgn.
  intros st H. subst. reflexivity.
```

Qed.

We typically use `hoare_seq` in conjunction with `hoare_consequence_pre` and the `eapply` tactic, as in this example.

**Exercise: 2 stars (hoare\_asgn\_example4)** Translate this "decorated program" into a formal proof: <sup>97</sup>  $\rightarrow$  <sup>98</sup>  $X ::= 1$ ; <sup>99</sup>  $\rightarrow$  <sup>100</sup>  $Y ::= 2$  <sup>101</sup>

Example `hoare_asgn_example4` :

```
{fun st => True} (X ::= (ANum 1);; Y ::= (ANum 2))
{fun st => st X = 1 /\ st Y = 2}.
```

Proof.

*Admitted.*

□

**Exercise: 3 stars (swap\_exercise)** Write an Imp program `c` that swaps the values of `X` and `Y` and show that it satisfies the following specification: <sup>102</sup>  $c$  <sup>103</sup>

Definition `swap_program` : `com` :=

*admit.*

Theorem `swap_exercise` :

```
{fun st => st X <= st Y}
swap_program
{fun st => st Y <= st X}.
```

Proof.

*Admitted.*

□

---

<sup>97</sup> `True`  
<sup>98</sup> `1=1`  
<sup>99</sup> `X=1`  
<sup>100</sup> `X=1 /\ 2=2`  
<sup>101</sup> `X=1 /\ Y=2`  
<sup>102</sup> `X<=Y`  
<sup>103</sup> `Y<=X`

**Exercise: 3 stars (hoarestate1)** Explain why the following proposition can't be proven:  
 $\text{forall } (a : \text{aexp}) (n : \text{nat}), \text{ }^{104} (X ::= (\text{ANum } 3)); Y ::= a) \text{ }^{105}.$

□

## 19.4.6 Conditionals

What sort of rule do we want for reasoning about conditional commands?

Certainly, if the same assertion  $Q$  holds after executing either of the branches, then it holds after the whole conditional. So we might be tempted to write:  $\text{ }^{106} c1 \text{ }^{107} \text{ }^{108} c2 \text{ }^{109}$

---

$\text{ }^{110} \text{IFB } b \text{ THEN } c1 \text{ ELSE } c2 \text{ }^{111}$  However, this is rather weak. For example, using this rule, we cannot show  $\text{ }^{112} \text{IFB } X == 0 \text{ THEN } Y ::= 2 \text{ ELSE } Y ::= X + 1 \text{ FI } \text{ }^{113}$  since the rule tells us nothing about the state in which the assignments take place in the "then" and "else" branches.

Fortunately, we can say something more precise. In the "then" branch, we know that the boolean expression  $b$  evaluates to **true**, and in the "else" branch, we know it evaluates to **false**. Making this information available in the premises of the rule gives us more information to work with when reasoning about the behavior of  $c1$  and  $c2$  (i.e., the reasons why they establish the postcondition  $Q$ ).

$\text{ }^{114} c1 \text{ }^{115} \text{ }^{116} c2 \text{ }^{117}$

---

(hoare\_if)  $\text{ }^{118} \text{IFB } b \text{ THEN } c1 \text{ ELSE } c2 \text{ FI } \text{ }^{119}$

To interpret this rule formally, we need to do a little work. Strictly speaking, the assertion we've written,  $P \wedge b$ , is the conjunction of an assertion and a boolean expression – i.e., it doesn't typecheck. To fix this, we need a way of formally "lifting" any bexp  $b$  to an assertion. We'll write  $\text{bassn } b$  for the assertion "the boolean expression  $b$  evaluates to **true** (in the given state)".

**Definition**  $\text{bassn } b : \text{Assertion} :=$

---

```

104 funst=>aevalsta=n
105 funst=>stY=n
106 P
107 Q
108 P
109 Q
110 P
111 Q
112 True
113 X<=Y
114 P/\b
115 Q
116 P/\~b
117 Q
118 P
119 Q

```



`fun st  $\Rightarrow$  (beval st b = true).`

A couple of useful facts about `bassn`:

Lemma `bexp_eval_true` :  $\forall b\ st,$   
`beval st b = true  $\rightarrow$  (bassn b) st.`

Proof.

`intros b st Hbe.`  
`unfold bassn. assumption. Qed.`

Lemma `bexp_eval_false` :  $\forall b\ st,$   
`beval st b = false  $\rightarrow$   $\neg$  ((bassn b) st).`

Proof.

`intros b st Hbe contra.`  
`unfold bassn in contra.`  
`rewrite  $\rightarrow$  contra in Hbe. inversion Hbe. Qed.`

Now we can formalize the Hoare proof rule for conditionals and prove it correct.

Theorem `hoare_if` :  $\forall P\ Q\ b\ c1\ c2,$   
`{{fun st  $\Rightarrow$  P st  $\wedge$  bassn b st}} c1 {{Q}}  $\rightarrow$`   
`{{fun st  $\Rightarrow$  P st  $\wedge$   $\neg$  (bassn b st)}} c2 {{Q}}  $\rightarrow$`   
`{{P}} (IFB b THEN c1 ELSE c2 FI) {{Q}}.`

Proof.

`intros P Q b c1 c2 HTrue HFalse st st' HE HP.`  
`inversion HE; subst.`  
`-`  
`apply (HTrue st st').`  
`assumption.`  
`split. assumption.`  
`apply bexp_eval_true. assumption.`  
`-`  
`apply (HFalse st st').`  
`assumption.`  
`split. assumption.`  
`apply bexp_eval_false. assumption. Qed.`

## Example

Here is a formal proof that the program we used to motivate the rule satisfies the specification we gave.

Example `if_example` :

`{{fun st  $\Rightarrow$  True}}`  
`IFB (BEq (Ald X) (ANum 0))`  
`THEN (Y ::= (ANum 2))`  
`ELSE (Y ::= APlus (Ald X) (ANum 1))`

```

FI
  {{fun st  $\Rightarrow$  st X  $\leq$  st Y}}.
Proof.
  apply hoare_if.
-
  eapply hoare_consequence_pre. apply hoare_asgn.
  unfold bassn, assn_sub, t_update, assert_implies.
  simpl. intros st [- H].
  apply beq_nat_true in H.
  rewrite H. omega.
-
  eapply hoare_consequence_pre. apply hoare_asgn.
  unfold assn_sub, t_update, assert_implies.
  simpl; intros st -. omega.
Qed.

```

**Exercise: 2 stars (if\_minus\_plus)** Prove the following hoare triple using hoare\_if:

Theorem if\_minus\_plus :

```

  {{fun st  $\Rightarrow$  True}}
  IFB (BLe (Ald X) (Ald Y))
    THEN (Z ::= AMinus (Ald Y) (Ald X))
    ELSE (Y ::= APlus (Ald X) (Ald Z))
FI
  {{fun st  $\Rightarrow$  st Y = st X + st Z}}.
Proof.
  Admitted.

```

## Exercise: One-sided conditionals

**Exercise: 4 stars (if1\_hoare)** In this exercise we consider extending Imp with "one-sided conditionals" of the form *IF1 b THEN c FI*. Here *b* is a boolean expression, and *c* is a command. If *b* evaluates to *true*, then command *c* is evaluated. If *b* evaluates to *false*, then *IF1 b THEN c FI* does nothing.

We recommend that you do this exercise before the ones that follow, as it should help solidify your understanding of the material.

The first step is to extend the syntax of commands and introduce the usual notations. (We've done this for you. We use a separate module to prevent polluting the global name space.)

Module IF1.

```

Inductive com : Type :=
| CSkip : com
| CAss : id  $\rightarrow$  aexp  $\rightarrow$  com

```

| CSeq : **com** → **com** → **com**  
 | Clf : **bexp** → **com** → **com** → **com**  
 | CWhile : **bexp** → **com** → **com**  
 | Clf1 : **bexp** → **com** → **com**.

Notation "'SKIP'" :=

CSkip.

Notation "c1 ;; c2" :=

(CSeq c1 c2) (at level 80, right associativity).

Notation "X '::~=' a" :=

(CAss X a) (at level 60).

Notation "'WHILE' b 'DO' c 'END'" :=

(CWhile b c) (at level 80, right associativity).

Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=

(Clf e1 e2 e3) (at level 80, right associativity).

Notation "'IF1' b 'THEN' c 'FI'" :=

(Clf1 b c) (at level 80, right associativity).

Next we need to extend the evaluation relation to accommodate *IF1* branches. This is for you to do... What rule(s) need to be added to **ceval** to evaluate one-sided conditionals?

Reserved Notation "c1 '/' st '\\ st'" (at level 40, st at level 39).

Inductive **ceval** : **com** → state → state → Prop :=

| E\_Skip : ∀ st : state, SKIP / st \\ st  
 | E\_Ass : ∀ (st : state) (a1 : **aexp**) (n : **nat**) (X : **id**),  
     aeval st a1 = n → (X ::= a1) / st \\ t\_update st X n  
 | E\_Seq : ∀ (c1 c2 : **com**) (st st' st'' : state),  
     c1 / st \\ st' → c2 / st' \\ st'' → (c1 ;; c2) / st \\ st''  
 | E\_IfTrue : ∀ (st st' : state) (b1 : **bexp**) (c1 c2 : **com**),  
     beval st b1 = **true** →  
     c1 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'  
 | E\_IfFalse : ∀ (st st' : state) (b1 : **bexp**) (c1 c2 : **com**),  
     beval st b1 = **false** →  
     c2 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'  
 | E\_WhileEnd : ∀ (b1 : **bexp**) (st : state) (c1 : **com**),  
     beval st b1 = **false** → (WHILE b1 DO c1 END) / st \\ st  
 | E\_WhileLoop : ∀ (st st' st'' : state) (b1 : **bexp**) (c1 : **com**),  
     beval st b1 = **true** →  
     c1 / st \\ st' →  
     (WHILE b1 DO c1 END) / st' \\ st'' →  
     (WHILE b1 DO c1 END) / st \\ st''

where "c1 '/' st '\\ st'" := (**ceval** c1 st st').

Now we repeat (verbatim) the definition and notation of Hoare triples.

```

Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st',
    c / st \\\ st' →
    P st →
    Q st'.

Notation "{ { P } } c { { Q } }" := (hoare_triple P c Q)
                                   (at level 90, c at next level)
                                   : hoare_spec_scope.

```

Finally, we (i.e., you) need to state and prove a theorem, *hoare\_if1*, that expresses an appropriate Hoare logic proof rule for one-sided conditionals. Try to come up with a rule that is both sound and as precise as possible.

For full credit, prove formally *hoare\_if1\_good* that your rule is precise enough to show the following valid Hoare triple: <sup>120</sup> IF1 Y <> 0 THEN X ::= X + Y FI <sup>121</sup>

Hint: Your proof of this triple may need to use the other proof rules also. Because we're working in a separate module, you'll need to copy here the rules you find necessary.

```

Lemma hoare_if1_good :
  { { fun st => st X + st Y = st Z } }
  IF1 BNot (BEq (Ald Y) (ANum 0)) THEN
    X ::= APlus (Ald X) (Ald Y)
  FI
  { { fun st => st X = st Z } }.

```

Proof. *Admitted*.

End IF1.

□

## 19.4.7 Recap

Idea: create a *domain specific logic* for reasoning about properties of Imp programs.

- This hides the low-level details of the semantics of the program
- Leads to a compositional reasoning process

The basic structure is given by *Hoare triples* of the form: <sup>122</sup> c <sup>123</sup>

- *P* and *Q* are properties about the state of the Imp program

---

<sup>120</sup> X+Y=Z

<sup>121</sup> X=Z

<sup>122</sup> P

<sup>123</sup> Q

- "If command  $c$  is started in a state satisfying assertion  $P$ , and if  $c$  eventually terminates in some final state, then this final state will satisfy the assertion  $Q$ ."

The rules of Hoare Logic (so far):

---

(hoare\_asgn) <sup>124</sup>  $X ::= a$  <sup>125</sup>

---

(hoare\_skip) <sup>126</sup> SKIP <sup>127</sup>  
<sup>128</sup>  $c_1$  <sup>129</sup>  $c_2$  <sup>130</sup>  $c_1$  <sup>131</sup>

---

(hoare\_seq) <sup>132</sup>  $c_1;;c_2$  <sup>133</sup>  
<sup>134</sup>  $c_1$  <sup>135</sup>  $c_2$  <sup>136</sup>  $c_1$  <sup>137</sup>

---

(hoare\_if) <sup>138</sup> IFB  $b$  THEN  $c_1$  ELSE  $c_2$  FI <sup>139</sup>  
<sup>140</sup>  $c$  <sup>141</sup>  $P \rightarrow P' \quad Q' \rightarrow Q$

---

(hoare\_consequence) <sup>142</sup>  $c$  <sup>143</sup>

---

### 19.4.8 Loops

Finally, we need a rule for reasoning about while loops.

Suppose we have a loop WHILE  $b$  DO  $c$  END and we want to find a pre-condition  $P$  and a post-condition  $Q$  such that <sup>144</sup> WHILE  $b$  DO  $c$  END <sup>145</sup> is a valid triple.

---

<sup>124</sup>  $Q[x] \rightarrow a$   
<sup>125</sup>  $Q$   
<sup>126</sup>  $P$   
<sup>127</sup>  $P$   
<sup>128</sup>  $P$   
<sup>129</sup>  $Q$   
<sup>130</sup>  $Q$   
<sup>131</sup>  $R$   
<sup>132</sup>  $P$   
<sup>133</sup>  $R$   
<sup>134</sup>  $P/\wedge b$   
<sup>135</sup>  $Q$   
<sup>136</sup>  $P/\wedge \sim b$   
<sup>137</sup>  $Q$   
<sup>138</sup>  $P$   
<sup>139</sup>  $Q$   
<sup>140</sup>  $P,$   
<sup>141</sup>  $Q,$   
<sup>142</sup>  $P$   
<sup>143</sup>  $Q$   
<sup>144</sup>  $P$   
<sup>145</sup>  $Q$

First of all, let's think about the case where  $b$  is false at the beginning – i.e., let's assume that the loop body never executes at all. In this case, the loop behaves like *SKIP*, so we might be tempted to write:

<sup>146</sup> WHILE  $b$  DO  $c$  END <sup>147</sup>.

But, as we remarked above for the conditional, we know a little more at the end – not just  $P$ , but also the fact that  $b$  is false in the current state. So we can enrich the postcondition a little:

<sup>148</sup> WHILE  $b$  DO  $c$  END <sup>149</sup>

What about the case where the loop body *does* get executed? In order to ensure that  $P$  holds when the loop finally exits, we certainly need to make sure that the command  $c$  guarantees that  $P$  holds whenever  $c$  is finished. Moreover, since  $P$  holds at the beginning of the first execution of  $c$ , and since each execution of  $c$  re-establishes  $P$  when it finishes, we can always assume that  $P$  holds at the beginning of  $c$ . This leads us to the following rule:

<sup>150</sup>  $c$  <sup>151</sup>

---

<sup>152</sup> WHILE  $b$  DO  $c$  END <sup>153</sup>

This is almost the rule we want, but again it can be improved a little: at the beginning of the loop body, we know not only that  $P$  holds, but also that the guard  $b$  is true in the current state. This gives us a little more information to use in reasoning about  $c$  (showing that it establishes the invariant by the time it finishes). This gives us the final version of the rule:

<sup>154</sup>  $c$  <sup>155</sup>

---

(hoare\_while) <sup>156</sup> WHILE  $b$  DO  $c$  END <sup>157</sup> The proposition  $P$  is called an *invariant* of the loop.

Lemma hoare\_while :  $\forall P \ b \ c,$

$\{\{\text{fun } st \Rightarrow P \ st \wedge \text{bassn } b \ st\}\} \ c \ \{\{P\}\} \rightarrow$   
 $\{\{P\}\} \text{ WHILE } b \text{ DO } c \text{ END } \{\{\text{fun } st \Rightarrow P \ st \wedge \neg (\text{bassn } b \ st)\}\}.$

Proof.

intros  $P \ b \ c \ Hhoare \ st \ st' \ He \ HP.$

remember (WHILE  $b$  DO  $c$  END) as  $wcom$  eqn:Heqwcom.

induction  $He$ ;

---

<sup>146</sup>  $P$   
<sup>147</sup>  $P$   
<sup>148</sup>  $P$   
<sup>149</sup>  $P \wedge \neg b$   
<sup>150</sup>  $P$   
<sup>151</sup>  $P$   
<sup>152</sup>  $P$   
<sup>153</sup>  $P \wedge \neg b$   
<sup>154</sup>  $P \wedge b$   
<sup>155</sup>  $P$   
<sup>156</sup>  $P$   
<sup>157</sup>  $P \wedge \neg b$

```

    try (inversion Heqwcom); subst; clear Heqwcom.
-
    split. assumption. apply bexp_eval_false. assumption.
-
    apply IHHe2. reflexivity.
    apply (Hhoare st st'). assumption.
    split. assumption. apply bexp_eval_true. assumption.
Qed.

```

One subtlety in the terminology is that calling some assertion  $P$  a "loop invariant" doesn't just mean that it is preserved by the body of the loop in question (i.e.,  $\{\{P\}\} \text{ c } \{\{P\}\}$ , where  $\text{c}$  is the loop body), but rather that  $P$  *together with the fact that the loop's guard is true* is a sufficient precondition for  $\text{c}$  to ensure  $P$  as a postcondition.

This is a slightly (but significantly) weaker requirement. For example, if  $P$  is the assertion  $X = 0$ , then  $P$  is an invariant of the loop `WHILE X = 2 DO X := 1 END` although it is clearly *not* preserved by the body of the loop.

Example while\_example :

```

    {{fun st => st X ≤ 3}}
    WHILE (BLe (Ald X) (ANum 2))
    DO X ::= APlus (Ald X) (ANum 1) END
    {{fun st => st X = 3}}.

```

Proof.

```

    eapply hoare_consequence_post.
    apply hoare_while.
    eapply hoare_consequence_pre.
    apply hoare_asgn.
    unfold bassn, assn_sub, assert_implies, t_update. simpl.
    intros st [H1 H2]. apply leb_complete in H2. omega.
    unfold bassn, assert_implies. intros st [Hle Hb].
    simpl in Hb. destruct (leb (st X) 2) eqn : Heql.
    exfalso. apply Hb; reflexivity.
    apply leb_iff_conv in Heql. omega.

```

Qed.

We can use the while rule to prove the following Hoare triple, which may seem surprising at first...

Theorem always\_loop\_hoare :  $\forall P Q,$   
 $\{\{P\}\} \text{ WHILE BTrue DO SKIP END } \{\{Q\}\}.$

Proof.

```

    intros P Q.
    apply hoare_consequence_pre with (P' := fun st : state => True).
    eapply hoare_consequence_post.
    apply hoare_while.

```

```

-
  apply hoare_post_true. intros st. apply !.
-
  simpl. intros st [Hinv Hguard].
  exfalso. apply Hguard. reflexivity.
-
  intros st H. constructor. Qed.

```

Of course, this result is not surprising if we remember that the definition of `hoare_triple` asserts that the postcondition must hold *only* when the command terminates. If the command doesn't terminate, we can prove anything we like about the post-condition.

Hoare rules that only talk about terminating commands are often said to describe a logic of "partial" correctness. It is also possible to give Hoare rules for "total" correctness, which build in the fact that the commands terminate. However, in this course we will only talk about partial correctness.

### Exercise: *REPEAT*

Module REPEATEXERCISE.

**Exercise: 4 stars, advanced (`hoare_repeat`)** In this exercise, we'll add a new command to our language of commands: *REPEAT* *c* *UNTIL* *a* *END*. You will write the evaluation rule for `repeat` and add a new Hoare rule to the language for programs involving it.

```

Inductive com : Type :=
| CSkip : com
| CAsgn : id → aexp → com
| CSeq : com → com → com
| Clf : bexp → com → com → com
| CWhile : bexp → com → com
| CRepeat : com → bexp → com.

```

*REPEAT* behaves like *WHILE*, except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*. Because of this, the body will always execute at least once.

Notation "'SKIP'" :=

CSkip.

Notation "c1 ;; c2" :=

(CSeq *c1 c2*) (at level 80, right associativity).

Notation "X ':=:' a" :=

(CAsgn *X a*) (at level 60).

Notation "'WHILE' b 'DO' c 'END'" :=

(CWhile *b c*) (at level 80, right associativity).

Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=



(CIf  $e1\ e2\ e3$ ) (at level 80, right associativity).  
 Notation "'REPEAT'  $e1$  'UNTIL'  $b2$  'END'" :=  
 (CRepeat  $e1\ b2$ ) (at level 80, right associativity).

Add new rules for *REPEAT* to **ceval** below. You can use the rules for *WHILE* as a guide, but remember that the body of a *REPEAT* should always execute at least once, and that the loop ends when the guard becomes true. Then update the *ceval\_cases* tactic to handle these added cases.

Inductive **ceval** : state → com → state → Prop :=

| E\_Skip : ∀  $st$ ,  
     **ceval**  $st$  SKIP  $st$   
 | E\_Ass : ∀  $st\ a1\ n\ X$ ,  
     aeval  $st\ a1 = n$  →  
     **ceval**  $st\ (X ::= a1)\ (t\_update\ st\ X\ n)$   
 | E\_Seq : ∀  $c1\ c2\ st\ st'\ st''$ ,  
     **ceval**  $st\ c1\ st' →$   
     **ceval**  $st'\ c2\ st'' →$   
     **ceval**  $st\ (c1 ;; c2)\ st''$   
 | E\_IfTrue : ∀  $st\ st'\ b1\ c1\ c2$ ,  
     beval  $st\ b1 = \text{true}$  →  
     **ceval**  $st\ c1\ st' →$   
     **ceval**  $st\ (\text{IFB } b1\ \text{THEN } c1\ \text{ELSE } c2\ \text{FI})\ st'$   
 | E\_IfFalse : ∀  $st\ st'\ b1\ c1\ c2$ ,  
     beval  $st\ b1 = \text{false}$  →  
     **ceval**  $st\ c2\ st' →$   
     **ceval**  $st\ (\text{IFB } b1\ \text{THEN } c1\ \text{ELSE } c2\ \text{FI})\ st'$   
 | E\_WhileEnd : ∀  $b1\ st\ c1$ ,  
     beval  $st\ b1 = \text{false}$  →  
     **ceval**  $st\ (\text{WHILE } b1\ \text{DO } c1\ \text{END})\ st$   
 | E\_WhileLoop : ∀  $st\ st'\ st''\ b1\ c1$ ,  
     beval  $st\ b1 = \text{true}$  →  
     **ceval**  $st\ c1\ st' →$   
     **ceval**  $st'\ (\text{WHILE } b1\ \text{DO } c1\ \text{END})\ st'' →$   
     **ceval**  $st\ (\text{WHILE } b1\ \text{DO } c1\ \text{END})\ st''$

.

A couple of definitions from above, copied here so they use the new **ceval**.

Notation " $c1\ /\ st\ \backslash\backslash\ st'$ " := (**ceval**  $st\ c1\ st'$ )  
 (at level 40,  $st$  at level 39).

Definition hoare\_triple ( $P$ :Assertion) ( $c$ :com) ( $Q$ :Assertion)

: Prop :=

∀  $st\ st', (c /\ st \backslash\backslash st') → P\ st → Q\ st'.$

Notation " $\{\{ P \} \} c \{\{ Q \} \}$ " :=  
 (hoare\_triple  $P \ c \ Q$ ) (at level 90,  $c$  at next level).

To make sure you've got the evaluation rules for *REPEAT* right, prove that `ex1_repeat` evaluates correctly.

Definition `ex1_repeat` :=  
 REPEAT  
 X ::= ANum 1;;  
 Y ::= APlus (Ald Y) (ANum 1)  
 UNTIL (BEq (Ald X) (ANum 1)) END.

Theorem `ex1_repeat_works` :  
 $\text{ex1\_repeat} / \text{empty\_state} \ \backslash \backslash$   
 $\text{t\_update} (\text{t\_update empty\_state X } 1) \ Y \ 1.$

Proof.  
*Admitted.*

Now state and prove a theorem, *hoare\_repeat*, that expresses an appropriate proof rule for `repeat` commands. Use `hoare_while` as a model, and try to make your rule as precise as possible.

For full credit, make sure (informally) that your rule can be used to prove the following valid Hoare triple: <sup>158</sup> REPEAT  $Y ::= X;; X ::= X - 1$  UNTIL  $X = 0$  END <sup>159</sup>

End REPEATEXERCISE.  
 □

## 19.5 Summary

Above, we've introduced Hoare Logic as a tool to reasoning about Imp programs. In the reminder of this chapter we will explore a systematic way to use Hoare Logic to prove properties about programs. The rules of Hoare Logic are the following:

---

(hoare\_asgn) <sup>160</sup>  $X ::= a$  <sup>161</sup>

---

(hoare\_skip) <sup>162</sup> SKIP <sup>163</sup>

---

<sup>158</sup>  $X > 0$

<sup>159</sup>  $X = 0 / \backslash Y > 0$

<sup>160</sup>  $Q[X] \rightarrow a$

<sup>161</sup>  $Q$

<sup>162</sup>  $P$

<sup>163</sup>  $P$

164  $c_1$  165 166  $c_2$  167

---

(hoare\_seq) <sup>168</sup>  $c_1;;c_2$  <sup>169</sup>  
<sup>170</sup>  $c_1$  <sup>171</sup>  $c_2$  <sup>172</sup>  $c_2$  <sup>173</sup>

---

(hoare\_if) <sup>174</sup> IFB  $b$  THEN  $c_1$  ELSE  $c_2$  FI <sup>175</sup>  
<sup>176</sup>  $c$  <sup>177</sup>

---

(hoare\_while) <sup>178</sup> WHILE  $b$  DO  $c$  END <sup>179</sup>  
<sup>180</sup>  $c$  <sup>181</sup>  $P \multimap P' \ Q' \multimap Q$

---

(hoare\_consequence) <sup>182</sup>  $c$  <sup>183</sup> In the next chapter, we'll see how these rules are used to prove that programs satisfy specifications of their behavior.

## 19.6 Additional Exercises

**Exercise: 3 stars (himp\_hoare)** In this exercise, we will derive proof rules for the *HAVOC* command, which we studied in the last chapter.

First, we enclose this work in a separate module, and recall the syntax and big-step semantics of Himp commands.

Module HIMP.

Inductive **com** : Type :=  
 | CSkip : **com**  
 | CAsgn : **id** → **aexp** → **com**

---

164  $P$   
 165  $Q$   
 166  $Q$   
 167  $R$   
 168  $P$   
 169  $R$   
 170  $P/\wedge b$   
 171  $Q$   
 172  $P/\wedge \sim b$   
 173  $Q$   
 174  $P$   
 175  $Q$   
 176  $P/\wedge b$   
 177  $P$   
 178  $P$   
 179  $P/\wedge \sim b$   
 180  $P'$   
 181  $Q'$   
 182  $P$   
 183  $Q$

```

| CSeq : com → com → com
| Clf : bexp → com → com → com
| CWhile : bexp → com → com
| CHavoc : id → com.

```

Notation "'SKIP'" :=

CSkip.

Notation "X ::= a" :=

(CAsgn X a) (at level 60).

Notation "c1 ;; c2" :=

(CSeq c1 c2) (at level 80, right associativity).

Notation "'WHILE' b 'DO' c 'END'" :=

(CWhile b c) (at level 80, right associativity).

Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=

(Clf e1 e2 e3) (at level 80, right associativity).

Notation "'HAVOC' X" := (CHavoc X) (at level 60).

Reserved Notation "c1 '/' st '\\ st'" (at level 40, st at level 39).

Inductive **ceval** : com → state → state → Prop :=

```

| E_Skip : ∀ st : state, SKIP / st \\ st
| E_Ass : ∀ (st : state) (a1 : aexp) (n : nat) (X : id),
    aeval st a1 = n → (X ::= a1) / st \\ t_update st X n
| E_Seq : ∀ (c1 c2 : com) (st st' st'' : state),
    c1 / st \\ st' → c2 / st' \\ st'' → (c1 ;; c2) / st \\ st''
| E_IfTrue : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
    beval st b1 = true →
    c1 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
| E_IfFalse : ∀ (st st' : state) (b1 : bexp) (c1 c2 : com),
    beval st b1 = false →
    c2 / st \\ st' → (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
| E_WhileEnd : ∀ (b1 : bexp) (st : state) (c1 : com),
    beval st b1 = false → (WHILE b1 DO c1 END) / st \\ st
| E_WhileLoop : ∀ (st st' st'' : state) (b1 : bexp) (c1 : com),
    beval st b1 = true →
    c1 / st \\ st' →
    (WHILE b1 DO c1 END) / st' \\ st'' →
    (WHILE b1 DO c1 END) / st \\ st''
| E_Havoc : ∀ (st : state) (X : id) (n : nat),
    (HAVOC X) / st \\ t_update st X n

```

where "c1 '/' st '\\ st'" := (**ceval** c1 st st').

The definition of Hoare triples is exactly as before. Unlike our notion of program equivalence, which had subtle consequences with occasionally nonterminating commands (exercise

*havoc\_diverge*), this definition is still fully satisfactory. Convince yourself of this before proceeding.

**Definition** `hoare_triple` ( $P$ :Assertion) ( $c$ :com) ( $Q$ :Assertion) : Prop :=  
 $\forall st\ st', c / st \ \backslash\ st' \rightarrow P\ st \rightarrow Q\ st'.$

**Notation** " $\{\{ P \}\} c \{\{ Q \}\}$ " := (`hoare_triple`  $P\ c\ Q$ )  
 (at level 90,  $c$  at next level)  
 : *hoare\_spec\_scope*.

Complete the Hoare rule for *HAVOC* commands below by defining `havoc_pre` and prove that the resulting rule is correct.

**Definition** `havoc_pre` ( $X$  : id) ( $Q$  : Assertion) : Assertion :=  
*admit*.

**Theorem** `hoare_havoc` :  $\forall (Q : \text{Assertion}) (X : \text{id}),$   
 $\{\{ \text{havoc\_pre } X\ Q \}\} \text{HAVOC } X \{\{ Q \}\}.$

**Proof.**

*Admitted.*

**End HIMP.**

□

*Date* : 2016 – 03 – 0409 : 33 : 20 – 0500(*Fri*, 04*Mar*2016)

# Chapter 20

## Library Hoare2

### 20.1 Hoare2: Hoare Logic, Part II

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.

Require Import Sflib.
Require Import Maps.
Require Import Imp.
Require Import Hoare.
```

### 20.2 Decorated Programs

The beauty of Hoare Logic is that it is *compositional*: the structure of proofs exactly follows the structure of programs. This suggests that we can record the essential ideas of a proof informally (leaving out some low-level calculational details) by decorating a program with appropriate assertions on each of its commands. Such a *decorated program* carries with it an (informal) proof of its own correctness.

For example, here is a complete decorated program:

<sup>1</sup> -> <sup>2</sup> X ::= m;; <sup>3</sup> -> <sup>4</sup> Z ::= p; <sup>5</sup> -> <sup>6</sup> WHILE X <> 0 DO <sup>7</sup> -> <sup>8</sup> Z ::= Z - 1;; <sup>9</sup> X ::= X -

---

<sup>1</sup> True  
<sup>2</sup> m=m  
<sup>3</sup> X=m  
<sup>4</sup> X=m/\p=p  
<sup>5</sup> X=m/\Z=p  
<sup>6</sup> Z-X=p-m  
<sup>7</sup> Z-X=p-m/\X<>0  
<sup>8</sup> (Z-1)-(X-1)=p-m  
<sup>9</sup> Z-(X-1)=p-m

1 <sup>10</sup> END; <sup>11</sup> -» <sup>12</sup>

Concretely, a decorated program consists of the program text interleaved with assertions (either a single assertion or possibly two assertions separated by an implication). To check that a decorated program represents a valid proof, we check that each individual command is *locally consistent* with its nearby assertions in the following sense:

- *SKIP* is locally consistent if its precondition and postcondition are the same: <sup>13</sup> SKIP <sub>14</sub>
- The sequential composition of *c1* and *c2* is locally consistent (with respect to assertions *P* and *R*) if *c1* is locally consistent (with respect to *P* and *Q*) and *c2* is locally consistent (with respect to *Q* and *R*): <sup>15</sup> *c1*; <sup>16</sup> *c2* <sup>17</sup>
- An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition: <sup>18</sup> *X* ::= *a* <sup>19</sup>
- A conditional is locally consistent (with respect to assertions *P* and *Q*) if the assertions at the top of its "then" and "else" branches are exactly *P* ∧ *b* and *P* ∧ ¬*b* and if its "then" branch is locally consistent (with respect to *P* ∧ *b* and *Q*) and its "else" branch is locally consistent (with respect to *P* ∧ ¬*b* and *Q*): <sup>20</sup> IFB *b* THEN <sup>21</sup> *c1* <sup>22</sup> ELSE <sup>23</sup> *c2* <sup>24</sup> FI <sup>25</sup>
- A while loop with precondition *P* is locally consistent if its postcondition is *P* ∧ ¬*b*, if the pre- and postconditions of its body are exactly *P* ∧ *b* and *P*, and if its body is locally consistent: <sup>26</sup> WHILE *b* DO <sup>27</sup> *c1* <sup>28</sup> END <sup>29</sup>
- A pair of assertions separated by -» is locally consistent if the first implies the second (in all states): <sup>30</sup> -» <sup>31</sup>

---

<sup>10</sup> *Z-X=p-m*  
<sup>11</sup> *Z-X=p-m/∧~(X<>0)*  
<sup>12</sup> *Z=p-m*  
<sup>13</sup> *P*  
<sup>14</sup> *P*  
<sup>15</sup> *P*  
<sup>16</sup> *Q*  
<sup>17</sup> *R*  
<sup>18</sup> *P[X]->a]*  
<sup>19</sup> *P*  
<sup>20</sup> *P*  
<sup>21</sup> *P/∧b*  
<sup>22</sup> *Q*  
<sup>23</sup> *P/∧~b*  
<sup>24</sup> *Q*  
<sup>25</sup> *Q*  
<sup>26</sup> *P*  
<sup>27</sup> *P/∧b*  
<sup>28</sup> *P*  
<sup>29</sup> *P/∧~b*  
<sup>30</sup> *P*  
<sup>31</sup> *P,*

This corresponds to the application of `hoare_consequence` and is the only place in a decorated program where checking if decorations are correct is not fully mechanical and syntactic, but rather may involve logical and/or arithmetic reasoning.

The above essentially describes a procedure for *verifying* the correctness of a given proof involves checking that every single command is locally consistent with the accompanying assertions. If we are instead interested in *finding* a proof for a given specification, we need to discover the right assertions. This can be done in an almost mechanical way, with the exception of finding loop invariants, which is the subject of the next section. In the remainder of this section we explain in detail how to construct decorations for several simple programs that don't involve non-trivial loop invariants.

### 20.2.1 Example: Swapping Using Addition and Subtraction

Here is a program that swaps the values of two variables using addition and subtraction (instead of by assigning to a temporary variable).  $X ::= X + Y;; Y ::= X - Y;; X ::= X - Y$  We can prove using decorations that this program is correct – i.e., it always swaps the values of variables  $X$  and  $Y$ .

(1) <sup>32</sup>  $\rightarrow$  (2) <sup>33</sup>  $X ::= X + Y;;$  (3) <sup>34</sup>  $Y ::= X - Y;;$  (4) <sup>35</sup>  $X ::= X - Y$  (5) <sup>36</sup>  $\parallel$  These decorations can be constructed as follows:

- We begin with the undecorated program (the unnumbered lines).
- We then add the specification – i.e., the outer precondition (1) and postcondition (5). In the precondition we use auxiliary variables (parameters)  $m$  and  $n$  to remember the initial values of variables  $X$  and  $Y$ , so that we can refer to them in the postcondition (5).
- We work backwards mechanically, starting from (5) and proceeding until we get to (2). At each step, we obtain the precondition of the assignment from its postcondition by substituting the assigned variable with the right-hand-side of the assignment. For instance, we obtain (4) by substituting  $X$  with  $X - Y$  in (5), and (3) by substituting  $Y$  with  $X - Y$  in (4).
- Finally, we verify that (1) logically implies (2) – i.e., that the step from (1) to (2) is a valid use of the law of consequence. For this we substitute  $X$  by  $m$  and  $Y$  by  $n$  and calculate as follows:  $(m + n) - ((m + n) - n) = n \wedge (m + n) - n = m \quad (m + n) - m = n \wedge m = m \quad n = n \wedge m = m$

---

<sup>32</sup>  $X=m/\backslash Y=n$

<sup>33</sup>  $(X+Y) - ((X+Y) - Y) = n/\backslash (X+Y) - Y = m$

<sup>34</sup>  $X - (X - Y) = n/\backslash X - Y = m$

<sup>35</sup>  $X - Y = n/\backslash Y = m$

<sup>36</sup>  $X=n/\backslash Y=m$



Note that, since we are working with natural numbers, not fixed-width machine integers, we don't need to worry about the possibility of arithmetic overflow anywhere in this argument. (This makes life quite a bit simpler!)

## 20.2.2 Example: Simple Conditionals

Here is a simple decorated program using conditionals: (1) <sup>37</sup> IFB  $X \leq Y$  THEN (2) <sup>38</sup>  $\neg$ » (3) <sup>39</sup>  $Z ::= Y - X$  (4) <sup>40</sup> ELSE (5) <sup>41</sup>  $\neg$ » (6) <sup>42</sup>  $Z ::= X - Y$  (7) <sup>43</sup> FI (8) <sup>44</sup>

These decorations were constructed as follows:

- We start with the outer precondition (1) and postcondition (8).
- We follow the format dictated by the `hoare_if` rule and copy the postcondition (8) to (4) and (7). We conjoin the precondition (1) with the guard of the conditional to obtain (2). We conjoin (1) with the negated guard of the conditional to obtain (5).
- In order to use the assignment rule and obtain (3), we substitute  $Z$  by  $Y - X$  in (4). To obtain (6) we substitute  $Z$  by  $X - Y$  in (7).
- Finally, we verify that (2) implies (3) and (5) implies (6). Both of these implications crucially depend on the ordering of  $X$  and  $Y$  obtained from the guard. For instance, knowing that  $X \leq Y$  ensures that subtracting  $X$  from  $Y$  and then adding back  $X$  produces  $Y$ , as required by the first disjunct of (3). Similarly, knowing that  $\neg(X \leq Y)$  ensures that subtracting  $Y$  from  $X$  and then adding back  $Y$  produces  $X$ , as needed by the second disjunct of (6). Note that  $n - m + m = n$  does *not* hold for arbitrary natural numbers  $n$  and  $m$  (for example,  $3 - 5 + 5 = 5$ ).

The  $\neg$  in that paragraph is typeset wrong. Maybe it's time to give up on all the unicode hacks in the generated HTML...?

---

```

37True
38True/\X<=Y
39(Y-X)+X=Y\/(Y-X)+Y=X
40Z+X=Y\Z+Y=X
41True/\~(X<=Y)
42(X-Y)+X=Y\/(X-Y)+Y=X
43Z+X=Y\Z+Y=X
44Z+X=Y\Z+Y=X

```

**Exercise: 2 stars (if\_minus\_plus\_reloaded)** Fill in valid decorations for the following program: <sup>45</sup> IFB X <= Y THEN <sup>46</sup> -» <sup>47</sup> Z ::= Y - X <sup>48</sup> ELSE <sup>49</sup> -» <sup>50</sup> Y ::= X + Z <sup>51</sup> FI <sup>52</sup> □

### 20.2.3 Example: Reduce to Zero

Here is a *WHILE* loop that is so simple it needs no invariant (i.e., the invariant **True** will do the job). (1) <sup>53</sup> WHILE X <> 0 DO (2) <sup>54</sup> -» (3) <sup>55</sup> X ::= X - 1 (4) <sup>56</sup> END (5) <sup>57</sup> -» (6) <sup>58</sup> The decorations can be constructed as follows:

- Start with the outer precondition (1) and postcondition (6).
- Following the format dictated by the **hoare\_while** rule, we copy (1) to (4). We conjoin (1) with the guard to obtain (2) and with the negation of the guard to obtain (5). Note that, because the outer postcondition (6) does not syntactically match (5), we need a trivial use of the consequence rule from (5) to (6).
- Assertion (3) is the same as (4), because X does not appear in 4, so the substitution in the assignment rule is trivial.
- Finally, the implication between (2) and (3) is also trivial.

From this informal proof, it is easy to read off a formal proof using the Coq versions of the Hoare rules. Note that we do *not* unfold the definition of **hoare\_triple** anywhere in this proof – the idea is to use the Hoare rules as a "self-contained" logic for reasoning about programs.

Definition reduce\_to\_zero' : **com** :=  
 WHILE BNot (BEq (Ald X) (ANum 0)) DO  
 X ::= AMinus (Ald X) (ANum 1)  
 END.

Theorem reduce\_to\_zero\_correct' :

---

<sup>45</sup>True  
<sup>46</sup>  
<sup>47</sup>  
<sup>48</sup>  
<sup>49</sup>  
<sup>50</sup>  
<sup>51</sup>  
<sup>52</sup>Y=X+Z  
<sup>53</sup>True  
<sup>54</sup>True/\X<>0  
<sup>55</sup>True  
<sup>56</sup>True  
<sup>57</sup>True/\X=0  
<sup>58</sup>X=0

```

{{fun st  $\Rightarrow$  True}}
reduce_to_zero'
{{fun st  $\Rightarrow$  st X = 0}}.

```

Proof.

```

unfold reduce_to_zero'.
eapply hoare_consequence_post.
apply hoare_while.
-

eapply hoare_consequence_pre. apply hoare_asgn.
intros st [HT Hbp]. unfold assn_sub. apply l.
-

intros st [Inv GuardFalse].
unfold bassn in GuardFalse. simpl in GuardFalse.
SearchAbout [not true].
rewrite not_true_iff_false in GuardFalse.
SearchAbout [negb false].
rewrite negb_false_iff in GuardFalse.
SearchAbout [beq_nat true].
apply beq_nat_true in GuardFalse.
apply GuardFalse. Qed.

```

## 20.2.4 Example: Division

The following Imp program calculates the integer quotient and remainder of two numbers  $m$  and  $n$  that are arbitrary constants in the program.  $X ::= m;; Y ::= 0;; \text{WHILE } n \leq X \text{ DO } X ::= X - n;; Y ::= Y + 1 \text{ END};$  In we replace  $m$  and  $n$  by concrete numbers and execute the program, it will terminate with the variable  $X$  set to the remainder when  $m$  is divided by  $n$  and  $Y$  set to the quotient.

In order to give a specification to this program we need to remember that dividing  $m$  by  $n$  produces a reminder  $X$  and a quotient  $Y$  such that  $n \times Y + X = m \wedge X < n$ .

It turns out that we get lucky with this program and don't have to think very hard about the loop invariant: the invariant is just the first conjunct  $n \times Y + X = m$ , and we can use this to decorate the program.

(1) <sup>59</sup>  $\rightarrow$  (2) <sup>60</sup>  $X ::= m;;$  (3) <sup>61</sup>  $Y ::= 0;;$  (4) <sup>62</sup>  $\text{WHILE } n \leq X \text{ DO}$  (5) <sup>63</sup>  $\rightarrow$  (6) <sup>64</sup>  $X$

---

```

59True
60 $n*0+m=m$ 
61 $n*0+X=m$ 
62 $n*Y+X=m$ 
63 $n*Y+X=m/\backslash n \leq X$ 
64 $n*(Y+1)+(X-n)=m$ 

```

$::= X - n;$  (7) <sup>65</sup>  $Y ::= Y + 1$  (8) <sup>66</sup>  $\text{END}$  (9) <sup>67</sup>

Assertions (4), (5), (8), and (9) are derived mechanically from the invariant and the loop's guard. Assertions (8), (7), and (6) are derived using the assignment rule going backwards from (8) to (6). Assertions (4), (3), and (2) are again backwards applications of the assignment rule.

Now that we've decorated the program it only remains to check that the two uses of the consequence rule are correct – i.e., that (1) implies (2) and that (5) implies (6). This is indeed the case, so we have a valid decorated program.

## 20.3 Finding Loop Invariants

Once the outermost precondition and postcondition are chosen, the only creative part in verifying programs using Hoare Logic is finding the right loop invariants. The reason this is difficult is the same as the reason that inductive mathematical proofs are: strengthening the loop invariant (or the induction hypothesis) means that you have a stronger assumption to work with when trying to establish the postcondition of the loop body (or complete the induction step of the proof), but it also means that the loop body postcondition itself (or the statement being proved inductively) is stronger and thus harder to prove!

This section shows how to approach the challenge of finding loop invariants through a series of examples and exercises.

### 20.3.1 Example: Slow Subtraction

The following program subtracts the value of  $X$  from the value of  $Y$  by repeatedly decrementing both  $X$  and  $Y$ . We want to verify its correctness with respect to the following specification:

<sup>68</sup>  $\text{WHILE } X <> 0 \text{ DO } Y ::= Y - 1;; X ::= X - 1 \text{ END}$  <sup>69</sup>

To verify this program, we need to find an invariant  $I$  for the loop. As a first step we can leave  $I$  as an unknown and build a *skeleton* for the proof by applying (backward) the rules for local consistency. This process leads to the following skeleton: (1) <sup>70</sup>  $\text{--}\gg$  (a) (2) <sup>71</sup>  $\text{WHILE } X <> 0 \text{ DO}$  (3) <sup>72</sup>  $\text{--}\gg$  (c) (4) <sup>73</sup>  $Y ::= Y - 1;;$  (5) <sup>74</sup>  $X ::= X - 1$  (6) <sup>75</sup>  $\text{END}$  (7) <sup>76</sup>  $\text{--}\gg$  (b) (8) <sup>77</sup>

---

<sup>65</sup>  $n * (Y + 1) + X = m$   
<sup>66</sup>  $n * Y + X = m$   
<sup>67</sup>  $n * Y + X = m / \wedge X < n$   
<sup>68</sup>  $X = m / \wedge Y = n$   
<sup>69</sup>  $Y = n - m$   
<sup>70</sup>  $X = m / \wedge Y = n$   
<sup>71</sup>  $I$   
<sup>72</sup>  $I / \wedge X <> 0$   
<sup>73</sup>  $I [X | \rightarrow X - 1] [Y | \rightarrow Y - 1]$   
<sup>74</sup>  $I [X | \rightarrow X - 1]$   
<sup>75</sup>  $I$   
<sup>76</sup>  $I / \wedge \sim (X <> 0)$   
<sup>77</sup>  $Y = n - m$

By examining this skeleton, we can see that any valid  $I$  will have to respect three conditions:

- (a) it must be weak enough to be implied by the loop's precondition, i.e., (1) must imply (2);
- (b) it must be strong enough to imply the loop's postcondition, i.e., (7) must imply (8);
- (c) it must be preserved by one iteration of the loop, i.e., (3) must imply (4).

Here's another bad  $\neg$  (in the proof)...

These conditions are actually independent of the particular program and specification we are considering. Indeed, every loop invariant has to satisfy them. One way to find an invariant that simultaneously satisfies these three conditions is by using an iterative process: start with a "candidate" invariant (e.g., a guess or a heuristic choice) and check the three conditions above; if any of the checks fails, try to use the information that we get from the failure to produce another – hopefully better – candidate invariant, and repeat the process.

For instance, in the reduce-to-zero example above, we saw that, for a very simple loop, choosing **True** as an invariant did the job. So let's try instantiating  $I$  with **True** in the skeleton above see what we get... (1) <sup>78</sup>  $\neg \gg$  (a - OK) (2) <sup>79</sup> WHILE  $X \neq 0$  DO (3) <sup>80</sup>  $\neg \gg$  (c - OK) (4) <sup>81</sup>  $Y ::= Y - 1$ ; (5) <sup>82</sup>  $X ::= X - 1$  (6) <sup>83</sup> END (7) <sup>84</sup>  $\neg \gg$  (b - WRONG!) (8) <sup>85</sup>

While conditions (a) and (c) are trivially satisfied, condition (b) is wrong, i.e., it is not the case that (7) **True**  $\wedge X = 0$  implies (8)  $Y = n - m$ . In fact, the two assertions are completely unrelated, so it is very easy to find a counterexample to the implication (say,  $Y = X = m = 0$  and  $n = 1$ ).

If we want (b) to hold, we need to strengthen the invariant so that it implies the postcondition (8). One simple way to do this is to let the invariant *be* the postcondition. So let's return to our skeleton, instantiate  $I$  with  $Y = n - m$ , and check conditions (a) to (c) again. (1) <sup>86</sup>  $\neg \gg$  (a - WRONG!) (2) <sup>87</sup> WHILE  $X \neq 0$  DO (3) <sup>88</sup>  $\neg \gg$  (c - WRONG!) (4) <sup>89</sup>  $Y ::= Y - 1$ ; (5) <sup>90</sup>  $X ::= X - 1$  (6) <sup>91</sup> END (7) <sup>92</sup>  $\neg \gg$  (b - OK) (8) <sup>93</sup>

---

```

78 $X=m/\wedge Y=n$ 
79True
80 $\text{True}/\wedge X \neq 0$ 
81True
82True
83True
84 $\text{True}/\wedge X=0$ 
85 $Y=n-m$ 
86 $X=m/\wedge Y=n$ 
87 $Y=n-m$ 
88 $Y=n-m/\wedge X \neq 0$ 
89 $Y-1=n-m$ 
90 $Y=n-m$ 
91 $Y=n-m$ 
92 $Y=n-m/\wedge X=0$ 
93 $Y=n-m$ 

```

This time, condition (b) holds trivially, but (a) and (c) are broken. Condition (a) requires that (1)  $X = m \wedge Y = n$  implies (2)  $Y = n - m$ . If we substitute  $Y$  by  $n$  we have to show that  $n = n - m$  for arbitrary  $m$  and  $n$ , which is not the case (for instance, when  $m = n = 1$ ). Condition (c) requires that  $n - m - 1 = n - m$ , which fails, for instance, for  $n = 1$  and  $m = 0$ . So, although  $Y = n - m$  holds at the end of the loop, it does not hold from the start, and it doesn't hold on each iteration; it is not a correct invariant.

This failure is not very surprising: the variable  $Y$  changes during the loop, while  $m$  and  $n$  are constant, so the assertion we chose didn't have much chance of being an invariant!

To do better, we need to generalize (8) to some statement that is equivalent to (8) when  $X$  is 0, since this will be the case when the loop terminates, and that "fills the gap" in some appropriate way when  $X$  is nonzero. Looking at how the loop works, we can observe that  $X$  and  $Y$  are decremented together until  $X$  reaches 0. So, if  $X = 2$  and  $Y = 5$  initially, after one iteration of the loop we obtain  $X = 1$  and  $Y = 4$ ; after two iterations  $X = 0$  and  $Y = 3$ ; and then the loop stops. Notice that the difference between  $Y$  and  $X$  stays constant between iterations: initially,  $Y = n$  and  $X = m$ , and the difference is always  $n - m$ . So let's try instantiating  $I$  in the skeleton above with  $Y - X = n - m$ . (1) <sup>94</sup> -» (a - OK) (2) <sup>95</sup> WHILE  $X \neq 0$  DO (3) <sup>96</sup> -» (c - OK) (4) <sup>97</sup>  $Y ::= Y - 1$ ; (5) <sup>98</sup>  $X ::= X - 1$  (6) <sup>99</sup> END (7) <sup>100</sup> -» (b - OK) (8) <sup>101</sup>

Success! Conditions (a), (b) and (c) all hold now. (To verify (c), we need to check that, under the assumption that  $X \neq 0$ , we have  $Y - X = (Y - 1) - (X - 1)$ ; this holds for all natural numbers  $X$  and  $Y$ .)

### 20.3.2 Exercise: Slow Assignment

**Exercise: 2 stars (slow\_assignment)** A roundabout way of assigning a number currently stored in  $X$  to the variable  $Y$  is to start  $Y$  at 0, then decrement  $X$  until it hits 0, incrementing  $Y$  at each step. Here is a program that implements this idea: <sup>102</sup>  $Y ::= 0$ ; WHILE  $X \neq 0$  DO  $X ::= X - 1$ ;  $Y ::= Y + 1$  END <sup>103</sup> Write an informal decorated program showing that this procedure is correct.

□

---

<sup>94</sup>  $X=m/\backslash Y=n$   
<sup>95</sup>  $Y-X=n-m$   
<sup>96</sup>  $Y-X=n-m/\backslash X \neq 0$   
<sup>97</sup>  $(Y-1)-(X-1)=n-m$   
<sup>98</sup>  $Y-(X-1)=n-m$   
<sup>99</sup>  $Y-X=n-m$   
<sup>100</sup>  $Y-X=n-m/\backslash X=0$   
<sup>101</sup>  $Y=n-m$   
<sup>102</sup>  $X=m$   
<sup>103</sup>  $Y=m$

### 20.3.3 Exercise: Slow Addition

**Exercise: 3 stars, optional (add\_slowly\_decoration)** The following program adds the variable  $X$  into the variable  $Z$  by repeatedly decrementing  $X$  and incrementing  $Z$ . WHILE  $X \neq 0$  DO  $Z ::= Z + 1;; X ::= X - 1$  END

Following the pattern of the `subtract_slowly` example above, pick a precondition and postcondition that give an appropriate specification of *add\_slowly*; then (informally) decorate the program accordingly.

□

### 20.3.4 Example: Parity

Here is a cute little program for computing the parity of the value initially stored in  $X$  (due to Daniel Cristofani).<sup>104</sup> WHILE  $2 \leq X$  DO  $X ::= X - 2$  END<sup>105</sup> The mathematical parity function used in the specification is defined in Coq as follows:

```
Fixpoint parity x :=
  match x with
  | 0 => 0
  | 1 => 1
  | S (S x') => parity x'
  end.
```

The postcondition does not hold at the beginning of the loop, since  $m = \text{parity } m$  does not hold for an arbitrary  $m$ , so we cannot use that as an invariant. To find an invariant that works, let's think a bit about what this loop does. On each iteration it decrements  $X$  by 2, which preserves the parity of  $X$ . So the parity of  $X$  does not change, i.e., it is invariant. The initial value of  $X$  is  $m$ , so the parity of  $X$  is always equal to the parity of  $m$ . Using  $\text{parity } X = \text{parity } m$  as an invariant we obtain the following decorated program:<sup>106</sup>  $\dashv\!\!\dashv$  (a - OK)<sup>107</sup> WHILE  $2 \leq X$  DO<sup>108</sup>  $\dashv\!\!\dashv$  (c - OK)<sup>109</sup>  $X ::= X - 2$ <sup>110</sup> END<sup>111</sup>  $\dashv\!\!\dashv$  (b - OK)<sup>112</sup>

With this invariant, conditions (a), (b), and (c) are all satisfied. For verifying (b), we observe that, when  $X < 2$ , we have  $\text{parity } X = X$  (we can easily see this in the definition of `parity`). For verifying (c), we observe that, when  $2 \leq X$ , we have  $\text{parity } X = \text{parity } (X-2)$ .

**Exercise: 3 stars, optional (parity\_formal)** Translate this proof to Coq. Refer to the reduce-to-zero example for ideas. You may find the following two lemmas useful:

---

```
104 X=m
105 X=paritym
106 X=m
107 parityX=paritym
108 parityX=paritym /\ 2<=X
109 parity(X-2)=paritym
110 parityX=paritym
111 parityX=paritym /\ X<2
112 X=paritym
```

Lemma parity\_ge\_2 :  $\forall x,$   
 $2 \leq x \rightarrow$   
 $\text{parity } (x - 2) = \text{parity } x.$

Proof.

induction x; intro. reflexivity.  
destruct x. inversion H. inversion H1.  
simpl. rewrite  $\leftarrow$  minus\_n\_O. reflexivity.

Qed.

Lemma parity\_lt\_2 :  $\forall x,$

$\neg 2 \leq x \rightarrow$   
 $\text{parity } (x) = x.$

Proof.

intros. induction x. reflexivity. destruct x. reflexivity.  
exfalso. apply H. omega.

Qed.

Theorem parity\_correct :  $\forall m,$

{ { fun st  $\Rightarrow$  st X = m } }  
WHILE BLe (ANum 2) (Ald X) DO  
X ::= AMinus (Ald X) (ANum 2)  
END  
{ { fun st  $\Rightarrow$  st X = parity m } }.

Proof.

*Admitted.*

□

### 20.3.5 Example: Finding Square Roots

The following program computes the square root of X by naive iteration: <sup>113</sup> Z ::= 0;; WHILE (Z+1)\*(Z+1) <= X DO Z ::= Z+1 END <sup>114</sup>

As above, we can try to use the postcondition as a candidate invariant, obtaining the following decorated program: (1) <sup>115</sup>  $\neg$ » (a - second conjunct of (2) WRONG!) (2) <sup>116</sup> Z ::= 0;; (3) <sup>117</sup> WHILE (Z+1)\*(Z+1) <= X DO (4) <sup>118</sup>  $\neg$ » (c - WRONG!) (5) <sup>119</sup> Z ::= Z+1 (6) <sup>120</sup> END (7) <sup>121</sup>  $\neg$ » (b - OK) (8) <sup>122</sup>

---

<sup>113</sup> X=m  
<sup>114</sup> Z\*Z<=m/\m<(Z+1)\*(Z+1)  
<sup>115</sup> X=m  
<sup>116</sup> 0\*0<=m/\m<1\*1  
<sup>117</sup> Z\*Z<=m/\m<(Z+1)\*(Z+1)  
<sup>118</sup> Z\*Z<=m/\(Z+1)\*(Z+1)<=X  
<sup>119</sup> (Z+1)\*(Z+1)<=m/\m<(Z+2)\*(Z+2)  
<sup>120</sup> Z\*Z<=m/\m<(Z+1)\*(Z+1)  
<sup>121</sup> Z\*Z<=m/\m<(Z+1)\*(Z+1)/\X<(Z+1)\*(Z+1)  
<sup>122</sup> Z\*Z<=m/\m<(Z+1)\*(Z+1)



This didn't work very well: conditions (a) and (c) both failed. Looking at condition (c), we see that the second conjunct of (4) is almost the same as the first conjunct of (5), except that (4) mentions  $X$  while (5) mentions  $m$ . But note that  $X$  is never assigned in this program, so we should always have  $X=m$ , but we didn't propagate this information from (1) into the loop invariant.

Also, looking at the second conjunct of (8), it seems quite hopeless as an invariant; fortunately and we don't even need it, since we can obtain it from the negation of the guard – the third conjunct in (7) – again under the assumption that  $X=m$ .

So we now try  $X=m \wedge Z \times Z \leq m$  as the loop invariant: <sup>123</sup> -» (a - OK) <sup>124</sup>  $Z ::= 0$ ; <sup>125</sup> WHILE  $(Z+1) \times (Z+1) \leq X$  DO <sup>126</sup> -» (c - OK) <sup>127</sup>  $Z ::= Z+1$  <sup>128</sup> END <sup>129</sup> -» (b - OK) <sup>130</sup>

This works, since conditions (a), (b), and (c) are now all trivially satisfied.

Very often, even if a variable is used in a loop in a read-only fashion (i.e., it is referred to by the program or by the specification and it is not changed by the loop), it is necessary to add the fact that it doesn't change to the loop invariant.

### 20.3.6 Example: Squaring

Here is a program that squares  $X$  by repeated addition:

<sup>131</sup>  $Y ::= 0$ ;  $Z ::= 0$ ; WHILE  $Y <> X$  DO  $Z ::= Z + X$ ;  $Y ::= Y + 1$  END <sup>132</sup>

The first thing to note is that the loop reads  $X$  but doesn't change its value. As we saw in the previous example, it is a good idea in such cases to add  $X = m$  to the invariant. The other thing that we know is often useful in the invariant is the postcondition, so let's add that too, leading to the invariant candidate  $Z = m \times m \wedge X = m$ . <sup>133</sup> -» (a - WRONG) <sup>134</sup>  $Y ::= 0$ ; <sup>135</sup>  $Z ::= 0$ ; <sup>136</sup> WHILE  $Y <> X$  DO <sup>137</sup> -» (c - WRONG) <sup>138</sup>  $Z ::= Z + X$ ; <sup>139</sup>  $Y ::= Y + 1$  <sup>140</sup> END <sup>141</sup> -» (b - OK) <sup>142</sup>

---

<sup>123</sup>  $X=m$   
<sup>124</sup>  $X=m/\backslash 0 \times 0 \leq m$   
<sup>125</sup>  $X=m/\backslash Z \times Z \leq m$   
<sup>126</sup>  $X=m/\backslash Z \times Z \leq m/\backslash (Z+1) \times (Z+1) \leq X$   
<sup>127</sup>  $X=m/\backslash (Z+1) \times (Z+1) \leq m$   
<sup>128</sup>  $X=m/\backslash Z \times Z \leq m$   
<sup>129</sup>  $X=m/\backslash Z \times Z \leq m/\backslash X < (Z+1) \times (Z+1)$   
<sup>130</sup>  $Z \times Z \leq m/\backslash m < (Z+1) \times (Z+1)$   
<sup>131</sup>  $X=m$   
<sup>132</sup>  $Z=m \times m$   
<sup>133</sup>  $X=m$   
<sup>134</sup>  $0=m \times m/\backslash X=m$   
<sup>135</sup>  $0=m \times m/\backslash X=m$   
<sup>136</sup>  $Z=m \times m/\backslash X=m$   
<sup>137</sup>  $Z=Y \times m/\backslash X=m/\backslash Y <> X$   
<sup>138</sup>  $Z+X=m \times m/\backslash X=m$   
<sup>139</sup>  $Z=m \times m/\backslash X=m$   
<sup>140</sup>  $Z=m \times m/\backslash X=m$   
<sup>141</sup>  $Z=m \times m/\backslash X=m/\backslash Y=X$   
<sup>142</sup>  $Z=m \times m$

Conditions (a) and (c) fail because of the  $Z = m \times m$  part. While  $Z$  starts at 0 and works itself up to  $m \times m$ , we can't expect  $Z$  to be  $m \times m$  from the start. If we look at how  $Z$  progresses in the loop, after the 1st iteration  $Z = m$ , after the 2nd iteration  $Z = 2 * m$ , and at the end  $Z = m \times m$ . Since the variable  $Y$  tracks how many times we go through the loop, this leads us to derive a new invariant candidate:  $Z = Y \times m \wedge X = m$ .<sup>143</sup> -» (a - OK)<sup>144</sup>  $Y ::= 0$ ;;<sup>145</sup>  $Z ::= 0$ ;;<sup>146</sup> WHILE  $Y <> X$  DO<sup>147</sup> -» (c - OK)<sup>148</sup>  $Z ::= Z + X$ ;<sup>149</sup>  $Y ::= Y + 1$ <sup>150</sup> END<sup>151</sup> -» (b - OK)<sup>152</sup>

This new invariant makes the proof go through: all three conditions are easy to check.

It is worth comparing the postcondition  $Z = m \times m$  and the  $Z = Y \times m$  conjunct of the invariant. It is often the case that one has to replace auxiliary variables (parameters) with variables – or with expressions involving both variables and parameters, like  $m - Y$  – when going from postconditions to invariants.

## 20.3.7 Exercise: Factorial

**Exercise: 3 stars (factorial)** Recall that  $n!$  denotes the factorial of  $n$  (i.e.,  $n! = 1 * 2 * \dots * n$ ). Here is an Imp program that calculates the factorial of the number initially stored in the variable  $X$  and puts it in the variable  $Y$ :<sup>153</sup>  $Y ::= 1$  ;; WHILE  $X <> 0$  DO  $Y ::= Y * X$  ;;  $X ::= X - 1$  END<sup>154</sup>

Fill in the blanks in following decorated program:<sup>155</sup> -»<sup>156</sup>  $Y ::= 1$ ;;<sup>157</sup> WHILE  $X <> 0$  DO<sup>158</sup> -»<sup>159</sup>  $Y ::= Y * X$ ;;<sup>160</sup>  $X ::= X - 1$ <sup>161</sup> END<sup>162</sup> -»<sup>163</sup>

□

---

```

143 X=m
144 O=O*m/\X=m
145 O=Y*m/\X=m
146 Z=Y*m/\X=m
147 Z=Y*m/\X=m/\Y<>X
148 Z+X=(Y+1)*m/\X=m
149 Z=(Y+1)*m/\X=m
150 Z=Y*m/\X=m
151 Z=Y*m/\X=m/\Y=X
152 Z=m*m
153 X=m
154 Y=m!
155 X=m
156
157
158
159
160
161
162
163 Y=m!
```

### 20.3.8 Exercise: Min

**Exercise: 3 stars (Min\_Hoare)** Fill in valid decorations for the following program. For the  $\Rightarrow$  steps in your annotations, you may rely (silently) on the following facts about min  
 Lemma lemma1 : forall x y, (x=0  $\vee$  y=0)  $\rightarrow$  min x y = 0. Lemma lemma2 : forall x y, min  
 (x-1) (y-1) = (min x y) - 1. plus standard high-school algebra, as always.

```

164 -» 165 X ::= a;; 166 Y ::= b;; 167 Z ::= 0;; 168 WHILE (X <> 0 /\ Y <> 0) DO 169 -»
170 X := X - 1;; 171 Y := Y - 1;; 172 Z := Z + 1 173 END 174 -» 175
□

```

**Exercise: 3 stars (two\_loops)** Here is a very inefficient way of adding 3 numbers: X  
 ::= 0;; Y ::= 0;; Z ::= c;; WHILE X <> a DO X ::= X + 1;; Z ::= Z + 1 END;; WHILE Y  
 <> b DO Y ::= Y + 1;; Z ::= Z + 1 END

Show that it does what it should by filling in the blanks in the following decorated  
 program.

```

176 -» 177 X ::= 0;; 178 Y ::= 0;; 179 Z ::= c;; 180 WHILE X <> a DO 181 -» 182 X ::= X
+ 1;; 183 Z ::= Z + 1 184 END;; 185 -» 186 WHILE Y <> b DO 187 -» 188 Y ::= Y + 1;; 189 Z
::= Z + 1 190 END 191 -» 192

```

---

```

164 True
165
166
167
168
169
170
171
172
173
174
175 Z=minab
176 True
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192 Z=a+b+c

```

□

### 20.3.9 Exercise: Power Series

**Exercise: 4 stars, optional (dpow2\_down)** Here is a program that computes the series:  
 $1 + 2 + 2^2 + \dots + 2^m = 2^{(m+1)} - 1$   
 $X ::= 0;; Y ::= 1;; Z ::= 1;; \text{WHILE } X <> m \text{ DO}$   
 $Z ::= 2 * Z;; Y ::= Y + Z;; X ::= X + 1 \text{ END}$   
 Write a decorated program for this.

## 20.4 Weakest Preconditions (Optional)

Some Hoare triples are more interesting than others. For example, <sup>193</sup>  $X ::= Y + 1$  <sup>194</sup> is *not* very interesting: although it is perfectly valid, it tells us nothing useful. Since the precondition isn't satisfied by any state, it doesn't describe any situations where we can use the command  $X ::= Y + 1$  to achieve the postcondition  $X \leq 5$ .

By contrast, <sup>195</sup>  $X ::= Y + 1$  <sup>196</sup> is useful: it tells us that, if we can somehow create a situation in which we know that  $Y \leq 4 \wedge Z = 0$ , then running this command will produce a state satisfying the postcondition. However, this triple is still not as useful as it could be, because the  $Z = 0$  clause in the precondition actually has nothing to do with the postcondition  $X \leq 5$ . The *most* useful triple (for this command and postcondition) is this one: <sup>197</sup>  $X ::= Y + 1$  <sup>198</sup> In other words,  $Y \leq 4$  is the *weakest* valid precondition of the command  $X ::= Y + 1$  for the postcondition  $X \leq 5$ .

In general, we say that " $P$  is the weakest precondition of command  $c$  for postcondition  $Q$ " if  $\{\{P\}\} c \{\{Q\}\}$  and if, whenever  $P'$  is an assertion such that  $\{\{P'\}\} c \{\{Q\}\}$ , it is the case that  $P'$  *st* implies  $P$  *st* for all states  $st$ .

**Definition**  $\text{is\_wp } P \ c \ Q :=$

$$\{\{P\}\} c \{\{Q\}\} \wedge \\ \forall P', \{\{P'\}\} c \{\{Q\}\} \rightarrow (P' \multimap P).$$

That is,  $P$  is the weakest precondition of  $c$  for  $Q$  if (a)  $P$  is a precondition for  $Q$  and  $c$ , and (b)  $P$  is the *weakest* (easiest to satisfy) assertion that guarantees that  $Q$  will hold after executing  $c$ .

**Exercise: 1 star, optional (wp)** What are the weakest preconditions of the following commands for the following postconditions? 1) <sup>199</sup> SKIP <sup>200</sup>

- 
- <sup>193</sup> False
  - <sup>194</sup>  $X \leq 5$
  - <sup>195</sup>  $Y \leq 4 \wedge Z = 0$
  - <sup>196</sup>  $X \leq 5$
  - <sup>197</sup>  $Y \leq 4$
  - <sup>198</sup>  $X \leq 5$
  - <sup>199</sup> ?
  - <sup>200</sup>  $X = 5$

- 2) <sup>201</sup>  $X ::= Y + Z$  <sup>202</sup>
- 3) <sup>203</sup>  $X ::= Y$  <sup>204</sup>
- 4) <sup>205</sup> IFB  $X == 0$  THEN  $Y ::= Z + 1$  ELSE  $Y ::= W + 2$  FI <sup>206</sup>
- 5) <sup>207</sup>  $X ::= 5$  <sup>208</sup>
- 6) <sup>209</sup> WHILE True DO  $X ::= 0$  END <sup>210</sup>  $\square$

**Exercise: 3 stars, advanced, optional (is\_wp\_formal)** Prove formally, using the definition of `hoare_triple`, that  $Y \leq 4$  is indeed the weakest precondition of  $X ::= Y + 1$  with respect to postcondition  $X \leq 5$ .

Theorem `is_wp_example` :

`is_wp (fun st  $\Rightarrow$  st Y  $\leq$  4)`  
`(X ::= APlus (Ald Y) (ANum 1)) (fun st  $\Rightarrow$  st X  $\leq$  5).`

Proof.

*Admitted.*

$\square$

**Exercise: 2 stars, advanced, optional (hoare\_asgn\_weakest)** Show that the precondition in the rule `hoare_asgn` is in fact the weakest precondition.

Theorem `hoare_asgn_weakest` :  $\forall Q X a,$   
`is_wp (Q [X  $\mapsto$  a]) (X ::= a) Q.`

Proof.

*Admitted.*

$\square$

**Exercise: 2 stars, advanced, optional (hoare\_havoc\_weakest)** Show that your `havoc_pre` rule from the *himp\_hoare* exercise in the Hoare chapter returns the weakest precondition. Module `HIMP2`.

Import *Himp*.

Lemma `hoare_havoc_weakest` :  $\forall (P Q : \text{Assertion}) (X : \text{id}),$   
 $\{\{ P \}\} \text{HAVOC } X \{\{ Q \}\} \rightarrow$   
 $P \multimap \text{havoc\_pre } X Q.$

Proof.

---

<sup>201</sup> ?  
<sup>202</sup>  $X=5$   
<sup>203</sup> ?  
<sup>204</sup>  $X=Y$   
<sup>205</sup> ?  
<sup>206</sup>  $Y=5$   
<sup>207</sup> ?  
<sup>208</sup>  $X=0$   
<sup>209</sup> ?  
<sup>210</sup>  $X=0$

*Admitted.*  
End HIMP2.  
□

## 20.5 Formal Decorated Programs (Optional)

Our informal conventions for decorated programs amount to a way of displaying Hoare triples, in which commands are annotated with enough embedded assertions that checking the validity of a triple is reduced to simple logical and algebraic calculations showing that some assertions imply others. In this section, we show that this informal presentation style can actually be made completely formal and indeed that checking the validity of decorated programs can mostly be automated.

### 20.5.1 Syntax

The first thing we need to do is to formalize a variant of the syntax of commands with embedded assertions. We call the new commands *decorated commands*, or **dcoms**.

Inductive **dcom** : Type :=

| DCSkip : Assertion → **dcom**  
| DCSeq : **dcom** → **dcom** → **dcom**  
| DCAsgn : **id** → **aexp** → Assertion → **dcom**  
| DCIf : **bexp** → Assertion → **dcom** → Assertion → **dcom**  
| DCWhile : **bexp** → Assertion → **dcom** → Assertion → **dcom**  
| DCPre : Assertion → **dcom** → **dcom**  
| DCPPost : **dcom** → Assertion → **dcom**.

Notation "'SKIP' {{ P }}"

:= (DCSkip P)  
(at level 10) : *dcom\_scope*.

Notation "l '::=' a {{ P }}"

:= (DCAsgn l a P)  
(at level 60, *a* at next level) : *dcom\_scope*.

Notation "'WHILE' b 'DO' {{ Pbody }} d 'END' {{ Ppost }}"

:= (DCWhile b Pbody d Ppost)  
(at level 80, right associativity) : *dcom\_scope*.

Notation "'IFB' b 'THEN' {{ P }} d 'ELSE' {{ P' }} d' 'FI' {{ Q }}"

:= (DCIf b P d P' d' Q)  
(at level 80, right associativity) : *dcom\_scope*.

Notation "'-»' {{ P }} d"

:= (DCPre P d)  
(at level 90, right associativity) : *dcom\_scope*.

```

Notation "{ { P } } d"
      := (DCPre P d)
      (at level 90) : dcom_scope.
Notation "d '->' { { P } }"
      := (DCPost d P)
      (at level 80, right associativity) : dcom_scope.
Notation "d ;; d'"
      := (DCSeq d d')
      (at level 80, right associativity) : dcom_scope.
Delimit Scope dcom_scope with dcom.

```

To avoid clashing with the existing **Notation** definitions for ordinary **commands**, we introduce these notations in a special scope called *dcom\_scope*, and we wrap examples with the declaration `% dcom` to signal that we want the notations to be interpreted in this scope.

Careful readers will note that we've defined two notations for the **DCPre** constructor, one with and one without a `->`. The "without" version is intended to be used to supply the initial precondition at the very top of the program.

```

Example dec_while : dcom := (
  { { fun st => True } }
  WHILE (BNot (BEq (Ald X) (ANum 0)))
  DO
    { { fun st => True & st X ≠ 0 } }
    X ::= (AMinus (Ald X) (ANum 1))
    { { fun _ => True } }
  END
  { { fun st => True & st X = 0 } } ->
  { { fun st => st X = 0 } }
) % dcom.

```

It is easy to go from a **dcom** to a **com** by erasing all annotations.

```

Fixpoint extract (d:dcom) : com :=
  match d with
  | DCSkip _ => SKIP
  | DCSeq d1 d2 => (extract d1 ;; extract d2)
  | DCAsgn X a _ => X ::= a
  | DCIf b _ d1 _ d2 _ => IFB b THEN extract d1 ELSE extract d2 FI
  | DCWhile b _ d _ => WHILE b DO extract d END
  | DCPre _ d => extract d
  | DCPost d _ => extract d
  end.

```

The choice of exactly where to put assertions in the definition of **dcom** is a bit subtle. The simplest thing to do would be to annotate every **dcom** with a precondition and postcondition. But this would result in very verbose programs with a lot of repeated annotations: for

example, a program like  $SKIP;SKIP$  would have to be annotated as <sup>211</sup> (<sup>212</sup>  $SKIP$  <sup>213</sup>) ;; (<sup>214</sup>  $SKIP$  <sup>215</sup>) <sup>216</sup>, with pre- and post-conditions on each  $SKIP$ , plus identical pre- and post-conditions on the semicolon!

Instead, the rule we've followed is this:

- The *post*-condition expected by each **dcom**  $d$  is embedded in  $d$ .
- The *pre*-condition is supplied by the context.

In other words, the invariant of the representation is that a **dcom**  $d$  together with a precondition  $P$  determines a Hoare triple  $\{\{P\}\} (\text{extract } d) \{\{\text{post } d\}\}$ , where **post** is defined as follows:

```
Fixpoint post (d:dcom) : Assertion :=
  match d with
  | DCSkip P => P
  | DCSeq d1 d2 => post d2
  | DCAsgn X a Q => Q
  | DCIf _ _ d1 _ d2 Q => Q
  | DCWhile b Pbody c Ppost => Ppost
  | DCPre _ d => post d
  | DCPost c Q => Q
  end.
```

Similarly, we can extract the "initial precondition" from a decorated program.

```
Fixpoint pre (d:dcom) : Assertion :=
  match d with
  | DCSkip P => fun st => True
  | DCSeq c1 c2 => pre c1
  | DCAsgn X a Q => fun st => True
  | DCIf _ _ t _ e _ => fun st => True
  | DCWhile b Pbody c Ppost => fun st => True
  | DCPre P c => P
  | DCPost c Q => pre c
  end.
```

This function is not doing anything sophisticated like calculating a weakest precondition; it just recursively searches for an explicit annotation at the very beginning of the program, returning default answers for programs that lack an explicit precondition (like a bare assignment or  $SKIP$ ).

---

<sup>211</sup> **P**  
<sup>212</sup> **P**  
<sup>213</sup> **P**  
<sup>214</sup> **P**  
<sup>215</sup> **P**  
<sup>216</sup> **P**



Using `pre` and `post`, and assuming that we adopt the convention of always supplying an explicit precondition annotation at the very beginning of our decorated programs, we can express what it means for a decorated program to be correct as follows:

**Definition** `dec_correct (d:dcom) :=`  
`{{pre d}} (extract d) {{post d}}.`

To check whether this Hoare triple is *valid*, we need a way to extract the “proof obligations” from a decorated program. These obligations are often called *verification conditions*, because they are the facts that must be verified to see that the decorations are logically consistent and thus add up to a complete proof of correctness.

## 20.5.2 Extracting Verification Conditions

The function `verification_conditions` takes a **dcom** `d` together with a precondition `P` and returns a *proposition* that, if it can be proved, implies that the triple `{{P}} (extract d) {{post d}}` is valid.

It does this by walking over `d` and generating a big conjunction including all the “local checks” that we listed when we described the informal rules for decorated programs. (Strictly speaking, we need to massage the informal rules a little bit to add some uses of the rule of consequence, but the correspondence should be clear.)

**Fixpoint** `verification_conditions (P : Assertion) (d:dcom)`  
`: Prop :=`

```
match d with
| DCSkip Q =>
  (P -> Q)
| DCSeq d1 d2 =>
  verification_conditions P d1
  ^ verification_conditions (post d1) d2
| DCAsgn X a Q =>
  (P -> Q [X |-> a])
| DCIf b P1 d1 P2 d2 Q =>
  ((fun st => P st ^ bassn b st) -> P1)
  ^ ((fun st => P st ^ ~ (bassn b st)) -> P2)
  ^ (Q <-> post d1) ^ (Q <-> post d2)
  ^ verification_conditions P1 d1
  ^ verification_conditions P2 d2
| DCWhile b Pbody d Ppost =>
  (P -> post d)
  ^ (Pbody <-> (fun st => post d st ^ bassn b st))
  ^ (Ppost <-> (fun st => post d st ^ ~ (bassn b st)))
  ^ verification_conditions Pbody d
| DCPre P' d =>
```

```

      (P -> P') ∧ verification_conditions P' d
| DCPost d Q =>
  verification_conditions P d ∧ (post d -> Q)
end.

```

And now the key theorem, stating that `verification_conditions` does its job correctly. Not surprisingly, we need to use each of the Hoare Logic rules at some point in the proof.

Theorem `verification_correct` :  $\forall d P,$   
 $\text{verification\_conditions } P d \rightarrow \{\{P\}\} (\text{extract } d) \{\{\text{post } d\}\}.$

Proof.

```

induction d; intros P H; simpl in *.
-
  eapply hoare_consequence_pre.
  apply hoare_skip.
  assumption.
-
  inversion H as [H1 H2]. clear H.
  eapply hoare_seq.
  apply IHd2. apply H2.
  apply IHd1. apply H1.
-
  eapply hoare_consequence_pre.
  apply hoare_asgn.
  assumption.
-
  inversion H as [HPre1 [HPre2 [[Hd11 Hd12]
                                [[Hd21 Hd22] [HThen HElse]]]]].
  clear H.
  apply IHd1 in HThen. clear IHd1.
  apply IHd2 in HElse. clear IHd2.
  apply hoare_if.
  eapply hoare_consequence_pre; eauto.
  eapply hoare_consequence_post; eauto.
  eapply hoare_consequence_pre; eauto.
  eapply hoare_consequence_post; eauto.
-
  inversion H as [Hpre [[Hbody1 Hbody2] [[Hpost1 Hpost2] Hd]]];
  subst; clear H.
  eapply hoare_consequence_pre; eauto.
  eapply hoare_consequence_post; eauto.
  apply hoare_while.
  eapply hoare_consequence_pre; eauto.
-

```

```

inversion H as [HP Hd]; clear H.
eapply hoare_consequence_pre. apply IHd. apply Hd. assumption.
-
inversion H as [Hd HQ]; clear H.
eapply hoare_consequence_post. apply IHd. apply Hd. assumption.
Qed.

```

(If you expand the proof, you'll see that it uses an unfamiliar idiom: `simpl in *`. We have used `...in...` variants of several tactics before, to apply them to values in the context rather than the goal. The syntax `tactic in *` extends this idea, applying `tactic` in the goal and every hypothesis in the context.)

### 20.5.3 Automation

The propositions generated by `verification_conditions` are fairly big, and they contain many conjuncts that are essentially trivial.

Eval `simpl in (verification_conditions (fun st => True) dec_while)`.

```

==> (((fun _ : state => True) -> (fun _ : state => True)) /\ ((fun _ : state => True) ->
(fun _ : state => True)) /\ (fun st : state => True /\ bassn (BNot (BEq (AId X) (ANum 0))) st) = (fun st : state => True /\ bassn (BNot (BEq (AId X) (ANum 0))) st) /\ (fun st : state => True /\ ~ bassn (BNot (BEq (AId X) (ANum 0))) st) = (fun st : state => True /\ ~ bassn (BNot (BEq (AId X) (ANum 0))) st) /\ (fun st : state => True /\ bassn (BNot (BEq (AId X) (ANum 0))) st) -> (fun _ : state => True) X |-> AMinus (AId X) (ANum 1)) /\ (fun st : state => True /\ ~ bassn (BNot (BEq (AId X) (ANum 0))) st) -> (fun st : state => st X = 0)

```

In principle, we could work with such propositions using just the tactics we have so far, but we can make things much smoother with a bit of automation. We first define a custom *verify* tactic that uses `split` repeatedly to turn all the conjunctions into separate subgoals and then uses `omega` and `eauto` (a handy general-purpose automation tactic that we'll discuss in detail later) to deal with as many of them as possible.

```

Tactic Notation "verify" :=
  apply verification_correct;
  repeat split;
  simpl; unfold assert_implies;
  unfold bassn in *; unfold beval in *; unfold aeval in *;
  unfold assn_sub; intros;
  repeat rewrite t_update_eq;
  repeat (rewrite t_update_neq; [| (intro X; inversion X)]);
  simpl in *;
  repeat match goal with [H : _ /\ _ <- _] => destruct H end;
  repeat rewrite not_true_iff_false in *;
  repeat rewrite not_false_iff_true in *;
  repeat rewrite negb_true_iff in *;

```

```

repeat rewrite negb_false_iff in *;
repeat rewrite beq_nat_true_iff in *;
repeat rewrite beq_nat_false_iff in *;
repeat rewrite leb_iff in *;
repeat rewrite leb_iff_conv in *;
try subst;
repeat
  match goal with
  [st : state ⊢ _] ⇒
    match goal with
    [H : st _ = _ ⊢ _] ⇒ rewrite → H in *; clear H
    | [H : _ = st _ ⊢ _] ⇒ rewrite ← H in *; clear H
    end
  end;
try eauto; try omega.

```

What's left after *verify* does its thing is "just the interesting parts" of checking that the decorations are correct. For very simple examples *verify* immediately solves the goal (provided that the annotations are correct).

Theorem dec\_while\_correct :  
 dec\_correct dec\_while.

Proof. *verify*. Qed.

Another example (formalizing a decorated program we've seen before):

```

Example subtract_slowly_dec (m:nat) (p:nat) : dcom := (
  {{ fun st ⇒ st X = m ∧ st Z = p }} ->
  {{ fun st ⇒ st Z - st X = p - m }}
  WHILE BNot (BEq (Ald X) (ANum 0))
  DO {{ fun st ⇒ st Z - st X = p - m ∧ st X ≠ 0 }} ->
    {{ fun st ⇒ (st Z - 1) - (st X - 1) = p - m }}
    Z ::= AMinus (Ald Z) (ANum 1)
    {{ fun st ⇒ st Z - (st X - 1) = p - m }} ;;
    X ::= AMinus (Ald X) (ANum 1)
    {{ fun st ⇒ st Z - st X = p - m }}
  END
  {{ fun st ⇒ st Z - st X = p - m ∧ st X = 0 }} ->
  {{ fun st ⇒ st Z = p - m }}
) % dcom.

```

Theorem subtract\_slowly\_dec\_correct : ∀ m p,  
 dec\_correct (subtract\_slowly\_dec m p).

Proof. intros m p. *verify*. Qed.

**Exercise: 3 stars, advanced (slow\_assignment\_dec)** In the *slow\_assignment* exercise above, we saw a roundabout way of assigning a number currently stored in  $X$  to the variable  $Y$ : start  $Y$  at 0, then decrement  $X$  until it hits 0, incrementing  $Y$  at each step. Write a formal version of this decorated program and prove it correct.

Example `slow_assignment_dec (m:nat) : dcom := admit.`

Theorem `slow_assignment_dec_correct :  $\forall m$ ,  
dec_correct (slow_assignment_dec m).`

Proof. *Admitted.*

□

**Exercise: 4 stars, advanced (factorial\_dec)** Remember the factorial function we worked with before:

```
Fixpoint real_fact (n:nat) : nat :=
  match n with
  | 0 => 1
  | S n' => n * (real_fact n')
  end.
```

Following the pattern of `subtract_slowly_dec`, write a decorated program *factorial\_dec* that implements the factorial function and prove it correct as *factorial\_dec\_correct*.

□

## 20.5.4 Examples

In this section, we use the automation developed above to verify formal decorated programs corresponding to most of the informal ones we have seen.

### Swapping Using Addition and Subtraction

Definition `swap : com :=`

```
X ::= APlus (Ald X) (Ald Y);;
Y ::= AMinus (Ald X) (Ald Y);;
X ::= AMinus (Ald X) (Ald Y).
```

Definition `swap_dec m n : dcom :=`

```
{ { fun st => st X = m ∧ st Y = n } } ->
  { { fun st => (st X + st Y) - ((st X + st Y) - st Y) = n
    ∧ (st X + st Y) - st Y = m } }
X ::= APlus (Ald X) (Ald Y)
  { { fun st => st X - (st X - st Y) = n ∧ st X - st Y = m } };;
Y ::= AMinus (Ald X) (Ald Y)
  { { fun st => st X - st Y = n ∧ st Y = m } };;
```

$X ::= \text{AMinus } (\text{Ald } X) (\text{Ald } Y)$   
 $\{\{ \text{fun } st \Rightarrow st\ X = n \wedge st\ Y = m \}\}\} \% dcom.$

Theorem swap\_correct :  $\forall\ m\ n,$   
 dec\_correct (swap\_dec  $m\ n$ ).

Proof. intros; verify. Qed.

## Simple Conditionals

Definition if\_minus\_plus\_com :=

IFB (BLe (Ald X) (Ald Y))  
 THEN (Z ::= AMinus (Ald Y) (Ald X))  
 ELSE (Y ::= APlus (Ald X) (Ald Z))  
 FI.

Definition if\_minus\_plus\_dec :=

( $\{\{ \text{fun } st \Rightarrow \text{True} \}\}$ )  
 IFB (BLe (Ald X) (Ald Y)) THEN  
 $\{\{ \text{fun } st \Rightarrow \text{True} \wedge st\ X \leq st\ Y \}\} -\gg$   
 $\{\{ \text{fun } st \Rightarrow st\ Y = st\ X + (st\ Y - st\ X) \}\}$   
 Z ::= AMinus (Ald Y) (Ald X)  
 $\{\{ \text{fun } st \Rightarrow st\ Y = st\ X + st\ Z \}\}$   
 ELSE  
 $\{\{ \text{fun } st \Rightarrow \text{True} \wedge \sim(st\ X \leq st\ Y) \}\} -\gg$   
 $\{\{ \text{fun } st \Rightarrow st\ X + st\ Z = st\ X + st\ Z \}\}$   
 Y ::= APlus (Ald X) (Ald Z)  
 $\{\{ \text{fun } st \Rightarrow st\ Y = st\ X + st\ Z \}\}$   
 FI  
 $\{\{ \text{fun } st \Rightarrow st\ Y = st\ X + st\ Z \}\}\} \% dcom.$

Theorem if\_minus\_plus\_correct :  
 dec\_correct if\_minus\_plus\_dec.

Proof. intros; verify. Qed.

Definition if\_minus\_dec :=

( $\{\{ \text{fun } st \Rightarrow \text{True} \}\}$ )  
 IFB (BLe (Ald X) (Ald Y)) THEN  
 $\{\{ \text{fun } st \Rightarrow \text{True} \wedge st\ X \leq st\ Y \}\} -\gg$   
 $\{\{ \text{fun } st \Rightarrow (st\ Y - st\ X) + st\ X = st\ Y$   
 $\vee (st\ Y - st\ X) + st\ Y = st\ X \}\}$   
 Z ::= AMinus (Ald Y) (Ald X)  
 $\{\{ \text{fun } st \Rightarrow st\ Z + st\ X = st\ Y \vee st\ Z + st\ Y = st\ X \}\}$   
 ELSE  
 $\{\{ \text{fun } st \Rightarrow \text{True} \wedge \sim(st\ X \leq st\ Y) \}\} -\gg$   
 $\{\{ \text{fun } st \Rightarrow (st\ X - st\ Y) + st\ X = st\ Y$

```

       $\vee (st\ X - st\ Y) + st\ Y = st\ X\}$ 
Z ::= AMinus (Ald X) (Ald Y)
   $\{\{fun\ st \Rightarrow st\ Z + st\ X = st\ Y \vee st\ Z + st\ Y = st\ X\}\}$ 
FI
   $\{\{fun\ st \Rightarrow st\ Z + st\ X = st\ Y \vee st\ Z + st\ Y = st\ X\}\})\%dcom.$ 
Theorem if_minus_correct :
  dec_correct if_minus_dec.
Proof. verify. Qed.

```

## Division

```

Definition div_mod_dec (a b : nat) : dcom := (
 $\{\{fun\ st \Rightarrow \text{True}\}\} -\gg$ 
   $\{\{fun\ st \Rightarrow b \times 0 + a = a\}\}$ 
  X ::= ANum a
   $\{\{fun\ st \Rightarrow b \times 0 + st\ X = a\}\}\};$ 
  Y ::= ANum 0
   $\{\{fun\ st \Rightarrow b \times st\ Y + st\ X = a\}\}\};$ 
  WHILE (BLe (ANum b) (Ald X)) DO
     $\{\{fun\ st \Rightarrow b \times st\ Y + st\ X = a \wedge b \leq st\ X\}\} -\gg$ 
     $\{\{fun\ st \Rightarrow b \times (st\ Y + 1) + (st\ X - b) = a\}\}$ 
    X ::= AMinus (Ald X) (ANum b)
     $\{\{fun\ st \Rightarrow b \times (st\ Y + 1) + st\ X = a\}\}\};$ 
    Y ::= APlus (Ald Y) (ANum 1)
     $\{\{fun\ st \Rightarrow b \times st\ Y + st\ X = a\}\}$ 
  END
   $\{\{fun\ st \Rightarrow b \times st\ Y + st\ X = a \wedge \sim(b \leq st\ X)\}\} -\gg$ 
   $\{\{fun\ st \Rightarrow b \times st\ Y + st\ X = a \wedge (st\ X < b)\}\}$ 
)%dcom.

```

```

Theorem div_mod_dec_correct :  $\forall\ a\ b,$ 
  dec_correct (div_mod_dec a b).
Proof. intros a b. verify.
  rewrite mult_plus_distr_l. omega.
Qed.

```

## Parity

```

Definition find_parity : com :=
  WHILE (BLe (ANum 2) (Ald X)) DO
    X ::= AMinus (Ald X) (ANum 2)
  END.

```

There are actually several ways to phrase the loop invariant for this program. Here is one natural one, which leads to a rather long proof:

Inductive **ev** : **nat** → Prop :=

| ev\_0 : **ev** 0  
| ev\_SS : ∀ n : **nat**, **ev** n → **ev** (S (S n)).

Definition find\_parity\_dec m : **dcom** :=

{ { fun st ⇒ st X = m } } ->  
{ { fun st ⇒ st X ≤ m ∧ **ev** (m - st X) } }  
**WHILE** (BLe (ANum 2) (Ald X)) **DO**  
{ { fun st ⇒ (st X ≤ m ∧ **ev** (m - st X)) ∧ 2 ≤ st X } } ->  
{ { fun st ⇒ st X - 2 ≤ m ∧ (**ev** (m - (st X - 2))) } }  
X ::= AMinus (Ald X) (ANum 2)  
{ { fun st ⇒ st X ≤ m ∧ **ev** (m - st X) } }  
**END**  
{ { fun st ⇒ (st X ≤ m ∧ **ev** (m - st X)) ∧ st X < 2 } } ->  
{ { fun st ⇒ st X = 0 ⇔ **ev** m } } ) % dcom.

Lemma l1 : ∀ m n p,

p ≤ n →  
n ≤ m →  
m - (n - p) = m - n + p.

Proof. intros. omega. Qed.

Lemma l2 : ∀ m,

**ev** m →  
**ev** (m + 2).

Proof. intros. rewrite plus\_comm. simpl. constructor. assumption. Qed.

Lemma l3' : ∀ m,

**ev** m →  
¬ **ev** (S m).

Proof. induction m; intros H1 H2. inversion H2. apply IHm.  
inversion H2; subst; assumption. assumption. Qed.

Lemma l3 : ∀ m,

1 ≤ m →  
**ev** m →  
**ev** (m - 1) →  
**False**.

Proof. intros. apply l2 in H1.

assert (G : m - 1 + 2 = S m). clear H0 H1. omega.  
rewrite G in H1. apply l3' in H0. apply H0. assumption. Qed.

Theorem find\_parity\_correct : ∀ m,

dec\_correct (find\_parity\_dec m).

Proof.



```

intro m. verify;

fold (leb 2 (st X)) in *;
try rewrite leb_iff in *;
try rewrite leb_iff_conv in *; eauto; try omega.
-
  rewrite minus_diag. constructor.
-
  rewrite l1; try assumption.
  apply l2; assumption.
-
  rewrite ← minus_n_O in H2. assumption.
-
  destruct (st X) as [| | n]].      +
    reflexivity.
+
  apply l3 in H; try assumption. inversion H.
+
  clear H0 H2.      omega.

```

Qed.

Here is a more intuitive way of writing the invariant:

Definition find\_parity\_dec' m : dcom :=

```

({{ fun st => st X = m }} ->
  {{ fun st => ev (st X) ↔ ev m }}
  WHILE (BLe (ANum 2) (Ald X)) DO
    {{ fun st => (ev (st X) ↔ ev m) ∧ 2 ≤ st X }} ->
    {{ fun st => (ev (st X - 2) ↔ ev m) }}
    X := AMinus (Ald X) (ANum 2)
    {{ fun st => (ev (st X) ↔ ev m) }}
  END
  {{ fun st => (ev (st X) ↔ ev m) ∧ ~(2 ≤ st X) }} ->
  {{ fun st => st X=0 ↔ ev m }})%dcom.

```

Lemma l4 : ∀ m,

```

2 ≤ m →
  (ev (m - 2) ↔ ev m).

```

Proof.

```

induction m; intros. split; intro; constructor.
destruct m. inversion H. inversion H1. simpl in *.
rewrite ← minus_n_O in *. split; intro.
  constructor. assumption.
  inversion H0. assumption.

```

Qed.

Theorem find\_parity\_correct' :  $\forall m$ ,  
 dec\_correct (find\_parity\_dec' m).

Proof.

```

intros m. verify;

fold (leb 2 (st X)) in *;
try rewrite leb_iff in *;
try rewrite leb_iff_conv in *; intuition; eauto; try omega.
-
rewrite l4 in H0; eauto.
-
rewrite l4; eauto.
-
apply H0. constructor.
-
destruct (st X) as [| | n]].      +
  reflexivity.
+
  inversion H.
+
  clear H0 H H3.      omega.

```

Qed.

Here is the simplest invariant we've found for this program:

Definition parity\_dec m : dcom :=

```

({{ fun st => st X = m }} ->
  {{ fun st => parity (st X) = parity m }}
  WHILE (BLe (ANum 2) (Ald X)) DO
    {{ fun st => parity (st X) = parity m  $\wedge$  2  $\leq$  st X }} ->
    {{ fun st => parity (st X - 2) = parity m }}
    X := AMinus (Ald X) (ANum 2)
    {{ fun st => parity (st X) = parity m }}
  END
  {{ fun st => parity (st X) = parity m  $\wedge$   $\sim$ (2  $\leq$  st X) }} ->
  {{ fun st => st X = parity m }})%dcom.

```

Theorem parity\_dec\_correct :  $\forall m$ ,  
 dec\_correct (parity\_dec m).

Proof.

```

intros. verify;

fold (leb 2 (st X)) in *;
try rewrite leb_iff in *;
try rewrite leb_iff_conv in *; eauto; try omega.

```

```

-
  rewrite ← H. apply parity_ge_2. assumption.
-
  rewrite ← H. symmetry. apply parity_lt_2. assumption.
Qed.

```

## Square Roots

```

Definition sqrt_dec m : dcom := (
  {{ fun st ⇒ st X = m }} ->
  {{ fun st ⇒ st X = m ∧ 0 × 0 ≤ m }}
  Z ::= ANum 0
  {{ fun st ⇒ st X = m ∧ st Z × st Z ≤ m }};;
  WHILE BLe (AMult (APlus (Ald Z) (ANum 1))
    (APlus (Ald Z) (ANum 1)))
    (Ald X) DO
    {{ fun st ⇒ (st X = m ∧ st Z × st Z ≤ m)
      ∧ (st Z + 1) × (st Z + 1) ≤ st X }} ->
    {{ fun st ⇒ st X = m ∧ (st Z + 1) × (st Z + 1) ≤ m }}
    Z ::= APlus (Ald Z) (ANum 1)
    {{ fun st ⇒ st X = m ∧ st Z × st Z ≤ m }}
  END
  {{ fun st ⇒ (st X = m ∧ st Z × st Z ≤ m)
    ∧ ~((st Z + 1) × (st Z + 1) ≤ st X) }} ->
  {{ fun st ⇒ st Z × st Z ≤ m ∧ m < (st Z + 1) × (st Z + 1) }})%dcom.

```

Theorem sqrt\_correct :  $\forall m$ ,  
 dec\_correct (sqrt\_dec m).  
 Proof. intro m. verify. Qed.

## Squaring

Again, there are several ways of annotating the squaring program. The simplest variant we've found, square\_simpler\_dec, is given last.

```

Definition square_dec (m : nat) : dcom := (
  {{ fun st ⇒ st X = m }}
  Y ::= Ald X
  {{ fun st ⇒ st X = m ∧ st Y = m }};;
  Z ::= ANum 0
  {{ fun st ⇒ st X = m ∧ st Y = m ∧ st Z = 0 }};;
  {{ fun st ⇒ st Z + st X × st Y = m × m }}
  WHILE BNot (BEq (Ald Y) (ANum 0)) DO
    {{ fun st ⇒ st Z + st X × st Y = m × m ∧ st Y ≠ 0 }} ->

```

```

  {{ fun st => (st Z + st X) + st X × (st Y - 1) = m × m }}
  Z ::= APlus (Ald Z) (Ald X)
  {{ fun st => st Z + st X × (st Y - 1) = m × m }};;
  Y ::= AMinus (Ald Y) (ANum 1)
  {{ fun st => st Z + st X × st Y = m × m }}
END
{{ fun st => st Z + st X × st Y = m × m ∧ st Y = 0 }} ->
{{ fun st => st Z = m × m }}
)%dcom.

```

Theorem square\_dec\_correct :  $\forall m$ ,  
 dec\_correct (square\_dec m).

Proof.

```

  intro n. verify.
  -
    destruct (st Y) as [| y']. apply False_ind. apply H0.
    reflexivity.
    simpl. rewrite ← minus_n_O.
    assert (G :  $\forall n m, n \times S m = n + n \times m$ ). {
      clear. intros. induction n. reflexivity. simpl.
      rewrite IHn. omega. }
    rewrite ← H. rewrite G. rewrite plus_assoc. reflexivity.

```

Qed.

```

Definition square_dec' (n : nat) : dcom := (
  {{ fun st => True }}
  X ::= ANum n
  {{ fun st => st X = n }};;
  Y ::= Ald X
  {{ fun st => st X = n ∧ st Y = n }};;
  Z ::= ANum 0
  {{ fun st => st X = n ∧ st Y = n ∧ st Z = 0 }};;
  {{ fun st => st Z = st X × (st X - st Y)
    ∧ st X = n ∧ st Y ≤ st X }}
  WHILE BNot (BEq (Ald Y) (ANum 0)) DO
    {{ fun st => (st Z = st X × (st X - st Y)
      ∧ st X = n ∧ st Y ≤ st X)
      ∧ st Y ≠ 0 }}
    Z ::= APlus (Ald Z) (Ald X)
    {{ fun st => st Z = st X × (st X - (st Y - 1))
      ∧ st X = n ∧ st Y ≤ st X }};;
    Y ::= AMinus (Ald Y) (ANum 1)
    {{ fun st => st Z = st X × (st X - st Y)
      ∧ st X = n ∧ st Y ≤ st X }}

```

```

END
{{ fun st  $\Rightarrow$  (st Z = st X  $\times$  (st X - st Y)
   $\wedge$  st X = n  $\wedge$  st Y  $\leq$  st X)
   $\wedge$  st Y = 0 }} - $\gg$ 
{{ fun st  $\Rightarrow$  st Z = n  $\times$  n }}
)%dcom.

Theorem square_dec'_correct :  $\forall$  n,
  dec_correct (square_dec' n).
Proof.
  intro n. verify.
  -
    rewrite minus_diag. omega.
  - subst.
    rewrite mult_minus_distr_l.
    repeat rewrite mult_minus_distr_l. rewrite mult_1_r.
    assert (G :  $\forall$  n m p, m  $\leq$  n  $\rightarrow$  p  $\leq$  m  $\rightarrow$  n - (m - p) = n - m + p).
    intros. omega.
    rewrite G. reflexivity. apply mult_le_compat_l. assumption.
    destruct (st Y). apply False_ind. apply H0. reflexivity.
    clear. rewrite mult_succ_r. rewrite plus_comm.
    apply le_plus_l.
  -
    rewrite  $\leftarrow$  minus_n_0. reflexivity.
Qed.

Definition square_simpler_dec (m : nat) : dcom := (
  {{ fun st  $\Rightarrow$  st X = m }} - $\gg$ 
  {{ fun st  $\Rightarrow$  0 = 0  $\times$  m  $\wedge$  st X = m }}
  Y ::= ANum 0
  {{ fun st  $\Rightarrow$  0 = (st Y) * m  $\wedge$  st X = m }};;
  Z ::= ANum 0
  {{ fun st  $\Rightarrow$  st Z = (st Y) * m  $\wedge$  st X = m }};;
  {{ fun st  $\Rightarrow$  st Z = (st Y) * m  $\wedge$  st X = m }}
  WHILE BNot (BEq (Ald Y) (Ald X)) DO
    {{ fun st  $\Rightarrow$  (st Z = (st Y) * m  $\wedge$  st X = m)
       $\wedge$  st Y  $\neq$  st X }} - $\gg$ 
    {{ fun st  $\Rightarrow$  st Z + st X = ((st Y) + 1) * m  $\wedge$  st X = m }}
    Z ::= APlus (Ald Z) (Ald X)
    {{ fun st  $\Rightarrow$  st Z = ((st Y) + 1) * m  $\wedge$  st X = m }};;
    Y ::= APlus (Ald Y) (ANum 1)
    {{ fun st  $\Rightarrow$  st Z = (st Y) * m  $\wedge$  st X = m }}
  END
  {{ fun st  $\Rightarrow$  (st Z = (st Y) * m  $\wedge$  st X = m)  $\wedge$  st Y = st X }} - $\gg$ 

```

```

  {{ fun st  $\Rightarrow$  st Z = m  $\times$  m }}
)%dcom.
Theorem square_simpler_dec_correct :  $\forall$  m,
  dec_correct (square_simpler_dec m).
Proof.
  intro m. verify.
  rewrite mult_plus_distr_r. simpl. rewrite  $\leftarrow$  plus_n_O.
  reflexivity.
Qed.

```

## Two loops

```

Definition two_loops_dec (a b c : nat) :=
( {{ fun st  $\Rightarrow$  True }} ->
  {{ fun st  $\Rightarrow$  c = 0 + c  $\wedge$  0 = 0 }}
  X ::= ANum 0
  {{ fun st  $\Rightarrow$  c = st X + c  $\wedge$  0 = 0 }};;
  Y ::= ANum 0
  {{ fun st  $\Rightarrow$  c = st X + c  $\wedge$  st Y = 0 }};;
  Z ::= ANum c
  {{ fun st  $\Rightarrow$  st Z = st X + c  $\wedge$  st Y = 0 }};;
  WHILE BNot (BEq (Ald X) (ANum a)) DO
    {{ fun st  $\Rightarrow$  (st Z = st X + c  $\wedge$  st Y = 0)  $\wedge$  st X  $\neq$  a }} ->
    {{ fun st  $\Rightarrow$  st Z + 1 = st X + 1 + c  $\wedge$  st Y = 0 }}
    X ::= APlus (Ald X) (ANum 1)
    {{ fun st  $\Rightarrow$  st Z + 1 = st X + c  $\wedge$  st Y = 0 }};;
    Z ::= APlus (Ald Z) (ANum 1)
    {{ fun st  $\Rightarrow$  st Z = st X + c  $\wedge$  st Y = 0 }}
  END
  {{ fun st  $\Rightarrow$  (st Z = st X + c  $\wedge$  st Y = 0)  $\wedge$  st X = a }} ->
  {{ fun st  $\Rightarrow$  st Z = a + st Y + c }};;
  WHILE BNot (BEq (Ald Y) (ANum b)) DO
    {{ fun st  $\Rightarrow$  st Z = a + st Y + c  $\wedge$  st Y  $\neq$  b }} ->
    {{ fun st  $\Rightarrow$  st Z + 1 = a + st Y + 1 + c }}
    Y ::= APlus (Ald Y) (ANum 1)
    {{ fun st  $\Rightarrow$  st Z + 1 = a + st Y + c }};;
    Z ::= APlus (Ald Z) (ANum 1)
    {{ fun st  $\Rightarrow$  st Z = a + st Y + c }}
  END
  {{ fun st  $\Rightarrow$  (st Z = a + st Y + c)  $\wedge$  st Y = b }} ->
  {{ fun st  $\Rightarrow$  st Z = a + b + c }}
)%dcom.

```

Theorem two\_loops\_correct :  $\forall a b c$ ,  
 dec\_correct (two\_loops\_dec a b c).  
 Proof. intros a b c. *verify*. Qed.

## Power series

Fixpoint pow2 n :=

```
  match n with
  | 0 => 1
  | S n' => 2 × (pow2 n')
  end.
```

Definition dpow2\_down n :=

```
( {{ fun st => True }} ->
  {{ fun st => 1 = (pow2 (0 + 1)) - 1 ∧ 1 = pow2 0 }}
  X ::= ANum 0
  {{ fun st => 1 = (pow2 (0 + 1)) - 1 ∧ 1 = pow2 (st X) }};;
  Y ::= ANum 1
  {{ fun st => st Y = (pow2 (st X + 1)) - 1 ∧ 1 = pow2 (st X) }};;
  Z ::= ANum 1
  {{ fun st => st Y = (pow2 (st X + 1)) - 1 ∧ st Z = pow2 (st X) }};;
  WHILE BNot (BEq (Ald X) (ANum n)) DO
    {{ fun st => (st Y = (pow2 (st X + 1)) - 1 ∧ st Z = pow2 (st X))
      ∧ st X ≠ n }} ->
    {{ fun st => st Y + 2 × st Z = (pow2 (st X + 2)) - 1
      ∧ 2 × st Z = pow2 (st X + 1) }}
    Z ::= AMult (ANum 2) (Ald Z)
    {{ fun st => st Y + st Z = (pow2 (st X + 2)) - 1
      ∧ st Z = pow2 (st X + 1) }};;
    Y ::= APlus (Ald Y) (Ald Z)
    {{ fun st => st Y = (pow2 (st X + 2)) - 1
      ∧ st Z = pow2 (st X + 1) }};;
    X ::= APlus (Ald X) (ANum 1)
    {{ fun st => st Y = (pow2 (st X + 1)) - 1
      ∧ st Z = pow2 (st X) }}
  END
  {{ fun st => (st Y = (pow2 (st X + 1)) - 1 ∧ st Z = pow2 (st X))
    ∧ st X = n }} ->
  {{ fun st => st Y = pow2 (n+1) - 1 }}
)%dcom.
```

Lemma pow2\_plus\_1 :  $\forall n$ ,

pow2 (n+1) = pow2 n + pow2 n.

Proof. induction n; simpl. reflexivity. omega. Qed.

Lemma pow2\_le\_1 :  $\forall n, \text{pow2 } n \geq 1$ .  
Proof. induction n. simpl. constructor. simpl. omega. Qed.

Theorem dpow2\_down\_correct :  $\forall n,$   
dec\_correct (dpow2\_down n).  
Proof.  
intro m. verify.  
-  
rewrite pow2\_plus\_1. rewrite  $\leftarrow$  H0. reflexivity.  
-  
rewrite  $\leftarrow$  plus\_n\_O.  
rewrite  $\leftarrow$  pow2\_plus\_1. remember (st X) as n.  
replace (pow2 (n + 1) - 1 + pow2 (n + 1))  
with (pow2 (n + 1) + pow2 (n + 1) - 1) by omega.  
rewrite  $\leftarrow$  pow2\_plus\_1.  
replace (n + 1 + 1) with (n + 2) by omega.  
reflexivity.  
-  
rewrite  $\leftarrow$  plus\_n\_O. rewrite  $\leftarrow$  pow2\_plus\_1.  
reflexivity.  
-  
replace (st X + 1 + 1) with (st X + 2) by omega.  
reflexivity.  
Qed.  
Date : 2016 - 03 - 04 13 : 37 : 06 - 0500(Fri, 04Mar2016)



# Chapter 21

## Library HoareAsLogic

### 21.1 HoareAsLogic: Hoare Logic as a Logic

```
Require Import Imp.  
Require Import Hoare.
```

The presentation of Hoare logic in chapter `Hoare` could be described as ”model-theoretic“: the proof rules for each of the constructors were presented as *theorems* about the evaluation behavior of programs, and proofs of program correctness (validity of Hoare triples) were constructed by combining these theorems directly in Coq.

Another way of presenting Hoare logic is to define a completely separate proof system – a set of axioms and inference rules that talk about commands, Hoare triples, etc. – and then say that a proof of a Hoare triple is a valid derivation in *that* logic. We can do this by giving an inductive definition of *valid derivations* in this new logic.

This chapter is optional. Before reading it, you’ll want to read the `ProofObjects` chapter.

### 21.2 Definitions

```
Inductive hoare_proof : Assertion → com → Assertion → Type :=  
| H_Skip : ∀ P,  
  hoare_proof P (SKIP) P  
| H_Asgn : ∀ Q V a,  
  hoare_proof (assn_sub V a Q) (V ::= a) Q  
| H_Seq : ∀ P c Q d R,  
  hoare_proof P c Q → hoare_proof Q d R → hoare_proof P (c ;; d) R  
| H_If : ∀ P Q b c1 c2,  
  hoare_proof (fun st ⇒ P st ∧ bassn b st) c1 Q →  
  hoare_proof (fun st ⇒ P st ∧ ~(bassn b st)) c2 Q →  
  hoare_proof P (IFB b THEN c1 ELSE c2 FI) Q  
| H_While : ∀ P b c,
```

```

hoare_proof (fun st  $\Rightarrow$  P st  $\wedge$  bassn b st) c P  $\rightarrow$ 
hoare_proof P (WHILE b DO c END) (fun st  $\Rightarrow$  P st  $\wedge$   $\neg$  (bassn b st))
| H_Consequence :  $\forall$  (P Q P' Q' : Assertion) c,
hoare_proof P' c Q'  $\rightarrow$ 
( $\forall$  st, P st  $\rightarrow$  P' st)  $\rightarrow$ 
( $\forall$  st, Q' st  $\rightarrow$  Q st)  $\rightarrow$ 
hoare_proof P c Q.

```

We don't need to include axioms corresponding to `hoare_consequence_pre` or `hoare_consequence_post`, because these can be proven easily from `H_Consequence`.

```

Lemma H_Consequence_pre :  $\forall$  (P Q P' : Assertion) c,
hoare_proof P' c Q  $\rightarrow$ 
( $\forall$  st, P st  $\rightarrow$  P' st)  $\rightarrow$ 
hoare_proof P c Q.

```

Proof.

```

intros. eapply H_Consequence.
apply X. apply H. intros. apply H0. Qed.

```

```

Lemma H_Consequence_post :  $\forall$  (P Q Q' : Assertion) c,
hoare_proof P c Q'  $\rightarrow$ 
( $\forall$  st, Q' st  $\rightarrow$  Q st)  $\rightarrow$ 
hoare_proof P c Q.

```

Proof.

```

intros. eapply H_Consequence.
apply X. intros. apply H0. apply H. Qed.

```

As an example, let's construct a proof object representing a derivation for the hoare triple<sup>1</sup>  $X ::= X+1 \;; X ::= X+2$ <sup>2</sup>. We can use Coq's tactics to help us construct the proof object.

Example `sample_proof` :

```

hoare_proof
  (assn_sub X (APlus (Ald X) (ANum 1))
    (assn_sub X (APlus (Ald X) (ANum 2))
      (fun st  $\Rightarrow$  st X = 3) ))
  (X ::= APlus (Ald X) (ANum 1) ;; (X ::= APlus (Ald X) (ANum 2)))
  (fun st  $\Rightarrow$  st X = 3).

```

Proof.

```

eapply H_Seq; apply H_Asgn.
Qed.

```

---

<sup>1</sup>`assn_subX(X+1) (assn_subX(X+2) (X=3))`  
<sup>2</sup>`X=3`

## 21.3 Properties

**Exercise: 2 stars (hoare\_proof\_sound)** Prove that such proof objects represent true claims.

Theorem `hoare_proof_sound` :  $\forall P \ c \ Q,$   
`hoare_proof`  $P \ c \ Q \rightarrow \{\{P\}\} \ c \ \{\{Q\}\}.$

Proof.

*Admitted.*

□

We can also use Coq’s reasoning facilities to prove metatheorems about Hoare Logic. For example, here are the analogs of two theorems we saw in chapter `Hoare` – this time expressed in terms of the syntax of Hoare Logic derivations (provability) rather than directly in terms of the semantics of Hoare triples.

The first one says that, for every  $P$  and  $c$ , the assertion  $\{\{P\}\} \ c \ \{\{\mathbf{True}\}\}$  is *provable* in Hoare Logic. Note that the proof is more complex than the semantic proof in `Hoare`: we actually need to perform an induction over the structure of the command  $c$ .

Theorem `H_Post_True_deriv`:

$\forall \ c \ P, \text{hoare\_proof } P \ c \ (\text{fun } \_ \Rightarrow \mathbf{True}).$

Proof.

```

intro c.
induction c; intro P.
-
  eapply H_Consequence.
  apply H_Skip.
  intros. apply H.
  intros. apply I.
-
  eapply H_Consequence_pre.
  apply H_Asgn.
  intros. apply I.
-
  eapply H_Consequence_pre.
  eapply H_Seq.
  apply (IHc1 (fun _ => True)).
  apply IHc2.
  intros. apply I.
-
  apply H_Consequence_pre with (fun _ => True).
  apply H_If.
  apply IHc1.
  apply IHc2.
  intros. apply I.
```

```

-
  eapply H_Consequence.
  eapply H_While.
  eapply IHc.
  intros; apply !.
  intros; apply !.

```

Qed.

Similarly, we can show that  $\{\{\mathbf{False}\}\} c \{\{Q\}\}$  is provable for any  $c$  and  $Q$ .

Lemma False\_and\_P\_imp:  $\forall P Q,$   
 $\mathbf{False} \wedge P \rightarrow Q.$

Proof.

```

  intros P Q [CONTRA HP].
  destruct CONTRA.

```

Qed.

Tactic Notation "pre\_false\_helper" constr(CONSTR) :=  
 eapply H\_Consequence\_pre;  
 [eapply CONSTR | intros ? CONTRA; destruct CONTRA].

Theorem H\_Pre\_False\_deriv:

$\forall c Q, \mathbf{hoare\_proof} (\text{fun } _ \Rightarrow \mathbf{False}) c Q.$

Proof.

```

  intros c.
  induction c; intro Q.
  - pre_false_helper H_Skip.
  - pre_false_helper H_Asgn.
  - pre_false_helper H_Seq. apply IHc1. apply IHc2.
  -
    apply H_If; eapply H_Consequence_pre.
    apply IHc1. intro. eapply False_and_P_imp.
    apply IHc2. intro. eapply False_and_P_imp.
  -
    eapply H_Consequence_post.
    eapply H_While.
    eapply H_Consequence_pre.
    apply IHc.
    intro. eapply False_and_P_imp.
    intro. simpl. eapply False_and_P_imp.

```

Qed.

As a last step, we can show that the set of **hoare\_proof** axioms is sufficient to prove any true fact about (partial) correctness. More precisely, any semantic Hoare triple that we can prove can also be proved from these axioms. Such a set of axioms is said to be *relatively complete*.

This proof is inspired by the one at <http://www.ps.uni-saarland.de/courses/sem-ws11/script/Hoare.htm>

To prove this fact, we'll need to invent some intermediate assertions using a technical device known as *weakest preconditions*. Given a command  $c$  and a desired postcondition assertion  $Q$ , the weakest precondition  $\text{wp } c \ Q$  is an assertion  $P$  such that  $\{\{P\}\} c \ \{\{Q\}\}$  holds, and moreover, for any other assertion  $P'$ , if  $\{\{P'\}\} c \ \{\{Q\}\}$  holds then  $P' \rightarrow P$ . We can more directly define this as follows:

**Definition**  $\text{wp } (c:\text{com}) \ (Q:\text{Assertion}) : \text{Assertion} :=$   
 $\text{fun } s \Rightarrow \forall s', c / s \ \backslash \backslash s' \rightarrow Q \ s'.$

**Exercise: 1 star (wp\_is\_precondition)** Lemma  $\text{wp\_is\_precondition}: \forall c \ Q,$   
 $\{\{\text{wp } c \ Q\}\} c \ \{\{Q\}\}.$   
*Admitted.*  
 $\square$

**Exercise: 1 star (wp\_is\_weakest)** Lemma  $\text{wp\_is\_weakest}: \forall c \ Q \ P',$   
 $\{\{P'\}\} c \ \{\{Q\}\} \rightarrow \forall st, P' \ st \rightarrow \text{wp } c \ Q \ st.$   
*Admitted.*

The following utility lemma will also be useful.

Lemma  $\text{bassn\_eval\_false} : \forall b \ st, \neg \text{bassn } b \ st \rightarrow \text{beval } st \ b = \text{false}.$

Proof.

intros  $b \ st \ H$ . unfold  $\text{bassn}$  in  $H$ . destruct  $(\text{beval } st \ b)$ .  
 $\text{exfalso. apply } H.$  reflexivity.  
 reflexivity.

Qed.

$\square$

**Exercise: 5 stars (hoare\_proof\_complete)** Complete the proof of the theorem.

Theorem  $\text{hoare\_proof\_complete}: \forall P \ c \ Q,$   
 $\{\{P\}\} c \ \{\{Q\}\} \rightarrow \text{hoare\_proof } P \ c \ Q.$

Proof.

intros  $P \ c$ . generalize dependent  $P$ .  
 induction  $c$ ; intros  $P \ Q \ HT$ .  
 -  
 eapply H\_Consequence.  
 eapply H\_Skip.  
 intros. *eassumption*.  
 intro  $st$ . apply  $HT$ . apply E\_Skip.  
 -  
 eapply H\_Consequence.  
 eapply H\_Asgn.

```

  intro st. apply HT. econstructor. reflexivity.
  intros; assumption.

```

-

```

  apply H_Seq with (wp c2 Q).
  eapply IHc1.
    intros st st' E1 H. unfold wp. intros st'' E2.
      eapply HT. econstructor; eassumption. assumption.
    eapply IHc2. intros st st' E1 H. apply H; assumption.
  Admitted.

```

□

Finally, we might hope that our axiomatic Hoare logic is *decidable*; that is, that there is an (terminating) algorithm (a *decision procedure*) that can determine whether or not a given Hoare triple is valid (derivable). But such a decision procedure cannot exist!

Consider the triple  $\{\{\mathbf{True}\}\} \mathbf{c} \{\{\mathbf{False}\}\}$ . This triple is valid if and only if  $\mathbf{c}$  is non-terminating. So any algorithm that could determine validity of arbitrary triples could solve the Halting Problem.

Similarly, the triple  $\{\{\mathbf{True}\} \text{ SKIP } \{\{P\}\}$  is valid if and only if  $\forall s, P\ s$  is valid, where  $P$  is an arbitrary assertion of Coq's logic. But it is known that there can be no decision procedure for this logic.

Overall, this axiomatic style of presentation gives a clearer picture of what it means to "give a proof in Hoare logic." However, it is not entirely satisfactory from the point of view of writing down such proofs in practice: it is quite verbose. The section of chapter Hoare2 on formalizing decorated programs shows how we can do even better.

*Date* : 2016 – 03 – 04 10 : 01 : 05 – 0500(*Fri*, 04*Mar*2016)

# Chapter 22

## Library Smallstep

### 22.1 Smallstep: Small-step Operational Semantics

```
Require Import Coq.Arith.Arith.  
Require Import Coq.Arith.EqNat.  
Require Import Coq.omega.Omega.  
Require Import Coq.Lists.List.  
Import ListNotations.  
  
Require Import Sflib.  
Require Import Maps.  
Require Import Imp.
```

The evaluators we have seen so far (e.g., the ones for **aexps**, **bexps**, and commands) have been formulated in a "big-step" style: they specify how a given expression can be evaluated to its final value (or a command plus a store to a final store) "all in one big step."

This style is simple and natural for many purposes – indeed, Gilles Kahn, who popularized it, called it *natural semantics*. But there are some things it does not do well. In particular, it does not give us a natural way of talking about *concurrent* programming languages, where the semantics of a program – i.e., the essence of how it behaves – is not just which input states get mapped to which output states, but also includes the intermediate states that it passes through along the way, since these states can also be observed by concurrently executing code.

Another shortcoming of the big-step style is more technical, but critical in many situations. Suppose we want to define a variant of Imp where variables could hold *either* numbers *or* lists of numbers. In the syntax of this extended language, it will be possible to write strange expressions like  $2 + \text{nil}$ , and our semantics for arithmetic expressions will then need to say something about how such expressions behave. One possibility is to maintain the convention that every arithmetic expressions evaluates to some number by choosing some way of viewing a list as a number – e.g., by specifying that a list should be interpreted as 0 when it occurs in a context expecting a number. But this is really a bit of a hack.

A much more natural approach is simply to say that the behavior of an expression like

$2 + \text{nil}$  is *undefined* – i.e., it doesn’t evaluate to any result at all. And we can easily do this: we just have to formulate **aeval** and **beval** as **Inductive** propositions rather than **Fixpoints**, so that we can make them partial functions instead of total ones.

Now, however, we encounter a serious deficiency. In this language, a command might fail to map a given starting state to any ending state for *two quite different reasons*: either because the execution gets into an infinite loop or because, at some point, the program tries to do an operation that makes no sense, such as adding a number to a list, so that none of the evaluation rules can be applied.

These two outcomes – nontermination vs. getting stuck in an erroneous configuration – are quite different. In particular, we want to allow the first (permitting the possibility of infinite loops is the price we pay for the convenience of programming with general looping constructs like *while*) but prevent the second (which is just wrong), for example by adding some form of *typechecking* to the language. Indeed, this will be a major topic for the rest of the course. As a first step, we need a way of presenting the semantics that allows us to distinguish nontermination from erroneous “stuck states.”

So, for lots of reasons, we’d like to have a finer-grained way of defining and reasoning about program behaviors. This is the topic of the present chapter. We replace the “big-step” **eval** relation with a “small-step” relation that specifies, for a given program, how the “atomic steps” of computation are performed.

## 22.2 A Toy Language

To save space in the discussion, let’s go back to an incredibly simple language containing just constants and addition. (We use single letters – **C** and **P** (for **Command** and **Plus**) – as constructor names, for brevity.) At the end of the chapter, we’ll see how to apply the same techniques to the full **Imp** language.

```
Inductive tm : Type :=
| C : nat → tm
| P : tm → tm → tm.
```

Here is a standard evaluator for this language, written in the big-step style that we’ve been using up to this point.

```
Fixpoint evalF (t : tm) : nat :=
  match t with
  | C n ⇒ n
  | P a1 a2 ⇒ evalF a1 + evalF a2
  end.
```

Here is the same evaluator, written in exactly the same style, but formulated as an inductively defined relation. Again, we use the notation  $t \backslash\backslash n$  for “ $t$  evaluates to  $n$ .”

---

```
(E_Const) C n \\\ n
          t1 \\\ n1 t2 \\\ n2
```



---

(E\_Plus)  $P \ t1 \ t2 \ \backslash\backslash \ n1 + n2$   
 Reserved Notation "  $t \ '\backslash\backslash' \ n$  " (at level 50, left associativity).  
 Inductive **eval** : **tm**  $\rightarrow$  **nat**  $\rightarrow$  Prop :=  
 | E\_Const :  $\forall \ n,$   
      $C \ n \ \backslash\backslash \ n$   
 | E\_Plus :  $\forall \ t1 \ t2 \ n1 \ n2,$   
      $t1 \ \backslash\backslash \ n1 \rightarrow$   
      $t2 \ \backslash\backslash \ n2 \rightarrow$   
      $P \ t1 \ t2 \ \backslash\backslash \ (n1 + n2)$   
  
 where "  $t \ '\backslash\backslash' \ n$  " := (**eval**  $t \ n$ ).

Module SIMPLEARITH1.

Now, here is the corresponding *small-step* evaluation relation.

---

(ST\_PlusConstConst)  $P \ (C \ n1) \ (C \ n2) ==> C \ (n1 + n2)$   
 $t1 ==> t1'$

---

(ST\_Plus1)  $P \ t1 \ t2 ==> P \ t1' \ t2$   
 $t2 ==> t2'$

---

(ST\_Plus2)  $P \ (C \ n1) \ t2 ==> P \ (C \ n1) \ t2'$

Reserved Notation "  $t \ '==>' \ t'$  " (at level 40).  
 Inductive **step** : **tm**  $\rightarrow$  **tm**  $\rightarrow$  Prop :=  
 | ST\_PlusConstConst :  $\forall \ n1 \ n2,$   
      $P \ (C \ n1) \ (C \ n2) ==> C \ (n1 + n2)$   
 | ST\_Plus1 :  $\forall \ t1 \ t1' \ t2,$   
      $t1 ==> t1' \rightarrow$   
      $P \ t1 \ t2 ==> P \ t1' \ t2$   
 | ST\_Plus2 :  $\forall \ n1 \ t2 \ t2',$   
      $t2 ==> t2' \rightarrow$   
      $P \ (C \ n1) \ t2 ==> P \ (C \ n1) \ t2'$   
  
 where "  $t \ '==>' \ t'$  " := (**step**  $t \ t'$ ).

Things to notice:

- We are defining just a single reduction step, in which one  $P$  node is replaced by its value.
- Each step finds the *leftmost*  $P$  node that is ready to go (both of its operands are constants) and rewrites it in place. The first rule tells how to rewrite this  $P$  node itself; the other two rules tell how to find it.

- A term that is just a constant cannot take a step.

Let's pause and check a couple of examples of reasoning with the **step** relation...

If  $t1$  can take a step to  $t1'$ , then  $P\ t1\ t2$  steps to  $P\ t1'\ t2$ :

Example test\_step\_1 :

```

P
  (P (C 0) (C 3))
  (P (C 2) (C 4))
==>
P
  (C (0 + 3))
  (P (C 2) (C 4)).

```

Proof.

apply ST\_Plus1. apply ST\_PlusConstConst. Qed.

**Exercise: 1 star (test\_step\_2)** Right-hand sides of sums can take a step only when the left-hand side is finished: if  $t2$  can take a step to  $t2'$ , then  $P\ (C\ n)\ t2$  steps to  $P\ (C\ n)\ t2'$ :

Example test\_step\_2 :

```

P
  (C 0)
  (P
    (C 2)
    (P (C 0) (C 3)))
==>
P
  (C 0)
  (P
    (C 2)
    (C (0 + 3))).

```

Proof.

*Admitted.*

□

End SIMPLEARITH1.

## 22.3 Relations

We will be working with several different single-step relations, so it is helpful to generalize a bit and state a few definitions and theorems about relations in general. (The optional chapter *Rel.v* develops some of these ideas in a bit more detail; it may be useful if the treatment here is too dense.)

A *binary relation* on a set  $X$  is a family of propositions parameterized by two elements of  $X$  – i.e., a proposition about pairs of elements of  $X$ .

Should we be getting this (and deterministic, multi, etc.) from the standard library? Arguably yes, though the naming in the library is awkward in places. **Definition** `relation (X: Type) := X → X → Prop`.

Our main examples of such relations in this chapter will be the single-step reduction relation, `==>`, and its multi-step variant, `==>*` (defined below), but there are many other examples – e.g., the “equals,” “less than,” “less than or equal to,” and “is the square of” relations on numbers, and the “prefix of” relation on lists and strings.

One simple property of the `==>` relation is that, like the big-step evaluation relation for `Imp`, it is *deterministic*.

*Theorem:* For each  $t$ , there is at most one  $t'$  such that  $t$  steps to  $t'$  ( $t ==> t'$  is provable). Formally, this is the same as saying that `==>` is deterministic.

*Proof sketch:* We show that if  $x$  steps to both  $y1$  and  $y2$ , then  $y1$  and  $y2$  are equal, by induction on a derivation of `step x y1`. There are several cases to consider, depending on the last rule used in this derivation and the last rule in the given derivation of `step x y2`.

- If both are `ST_PlusConstConst`, the result is immediate.
- The cases when both derivations end with `ST_Plus1` or `ST_Plus2` follow by the induction hypothesis.
- It cannot happen that one is `ST_PlusConstConst` and the other is `ST_Plus1` or `ST_Plus2`, since this would imply that  $x$  has the form  $P \ t1 \ t2$  where both  $t1$  and  $t2$  are constants (by `ST_PlusConstConst`) and one of  $t1$  or  $t2$  has the form  $P \ _$ .
- Similarly, it cannot happen that one is `ST_Plus1` and the other is `ST_Plus2`, since this would imply that  $x$  has the form  $P \ t1 \ t2$  where  $t1$  has both the form  $P \ t11 \ t12$  and the form  $C \ n$ .  $\square$

Formally:

**Definition** `deterministic {X: Type} (R: relation X) :=`

`$\forall x \ y1 \ y2 : X, R \ x \ y1 \rightarrow R \ x \ y2 \rightarrow y1 = y2$ .`

`Module SIMPLEARITH2.`

`Import SimpleArith1.`

**Theorem** `step_deterministic:`

`deterministic step.`

**Proof.**

`unfold deterministic. intros x y1 y2 Hy1 Hy2.`

`generalize dependent y2.`

`induction Hy1; intros y2 Hy2.`

`- inversion Hy2.`

`+ reflexivity.`

```

+ inversion H2.
+ inversion H2.
- inversion Hy2.
+ rewrite ← H0 in Hy1. inversion Hy1.
+
  rewrite ← (IHHy1 t1'0).
  reflexivity. assumption.
+ rewrite ← H in Hy1. inversion Hy1.
- inversion Hy2.
+ rewrite ← H1 in Hy1. inversion Hy1.
+ inversion H2.
+
  rewrite ← (IHHy1 t2'0).
  reflexivity. assumption.

```

Qed.

There is some annoying repetition in this proof. Each use of `inversion Hy2` results in three subcases, only one of which is relevant (the one that matches the current case in the induction on *Hy1*). The other two subcases need to be dismissed by finding the contradiction among the hypotheses and doing inversion on it.

The tactic `solve by inversion`, which is defined in the small accompanying file *SfLib.v*, can be helpful in such cases. It will solve the goal if it can be solved by inverting some hypothesis; otherwise, it fails. (The variants `solve by inversion 2` and `solve by inversion 3` that work if two or three consecutive inversions will solve the goal.)

Let's see how a proof of the previous theorem can be simplified using this tactic...

Theorem `step_deterministic_alt`: deterministic **step**.

Proof.

```

intros x y1 y2 Hy1 Hy2.
generalize dependent y2.
induction Hy1; intros y2 Hy2;
  inversion Hy2; subst; try (solve by inversion).
- reflexivity.
-
  apply IHHy1 in H2. rewrite H2. reflexivity.
-
  apply IHHy1 in H2. rewrite H2. reflexivity.

```

Qed.

End SIMPLEARITH2.

### 22.3.1 Values

Next, it will be useful to slightly reformulate the definition of single-step reduction by stating it in terms of "values."

It is useful to think of the  $\Rightarrow$  relation as defining an *abstract machine*:

- At any moment, the *state* of the machine is a term.
- A *step* of the machine is an atomic unit of computation – here, a single “add” operation.
- The *halting states* of the machine are ones where there is no more computation to be done.

We can then execute a term  $t$  as follows:

- Take  $t$  as the starting state of the machine.
- Repeatedly use the  $\Rightarrow$  relation to find a sequence of machine states, starting with  $t$ , where each state steps to the next.
- When no more reduction is possible, “read out” the final state of the machine as the result of execution.

Intuitively, it is clear that the final states of the machine are always terms of the form  $C\ n$  for some  $n$ . We call such terms *values*.

**Inductive value : tm → Prop :=**  
**v\_const : ∀ n, value (C n).**

Having introduced the idea of values, we can use it in the definition of the  $\Rightarrow$  relation to write **ST\_Plus2** rule in a slightly more elegant way:

---

(ST\_PlusConstConst)  $P\ (C\ n1)\ (C\ n2) \Rightarrow C\ (n1 + n2)$   
 $t1 \Rightarrow t1'$

---

(ST\_Plus1)  $P\ t1\ t2 \Rightarrow P\ t1'\ t2$   
 $\text{value } v1\ t2 \Rightarrow t2'$

---

(ST\_Plus2)  $P\ v1\ t2 \Rightarrow P\ v1\ t2'$  Again, the variable names here carry important information: by convention,  $v1$  ranges only over values, while  $t1$  and  $t2$  range over arbitrary terms. (Given this convention, the explicit **value** hypothesis is arguably redundant. We’ll keep it for now, to maintain a close correspondence between the informal and Coq versions of the rules, but later on we’ll drop it in informal rules for brevity.)

Here are the formal rules:

**Reserved Notation** "  $t \Rightarrow t'$  " (at level 40).

**Inductive step : tm → tm → Prop :=**  
**| ST\_PlusConstConst : ∀ n1 n2,**  
**P (C n1) (C n2)**  
**$\Rightarrow C\ (n1 + n2)$**

```

| ST_Plus1 :  $\forall t1\ t1'\ t2,$ 
     $t1 \implies t1' \rightarrow$ 
     $P\ t1\ t2 \implies P\ t1'\ t2$ 
| ST_Plus2 :  $\forall v1\ t2\ t2',$ 
    value  $v1 \rightarrow$ 
     $t2 \implies t2' \rightarrow$ 
     $P\ v1\ t2 \implies P\ v1\ t2'$ 

```

where " $t \implies t'$ " := (**step**  $t\ t'$ ).

**Exercise: 3 stars, recommended (redo\_determinism)** As a sanity check on this change, let's re-verify determinism.

*Proof sketch:* We must show that if  $x$  steps to both  $y1$  and  $y2$ , then  $y1$  and  $y2$  are equal. Consider the final rules used in the derivations of **step**  $x\ y1$  and **step**  $x\ y2$ .

- If both are **ST\_PlusConstConst**, the result is immediate.
- It cannot happen that one is **ST\_PlusConstConst** and the other is **ST\_Plus1** or **ST\_Plus2**, since this would imply that  $x$  has the form  $P\ t1\ t2$  where both  $t1$  and  $t2$  are constants (by **ST\_PlusConstConst**) and one of  $t1$  or  $t2$  has the form  $P\ \_$ .
- Similarly, it cannot happen that one is **ST\_Plus1** and the other is **ST\_Plus2**, since this would imply that  $x$  has the form  $P\ t1\ t2$  where  $t1$  both has the form  $P\ t11\ t12$  and is a value (hence has the form **C**  $n$ ).
- The cases when both derivations end with **ST\_Plus1** or **ST\_Plus2** follow by the induction hypothesis.  $\square$

Most of this proof is the same as the one above. But to get maximum benefit from the exercise you should try to write your formal version from scratch and just use the earlier one if you get stuck.

**Theorem** `step_deterministic` :  
deterministic **step**.

**Proof.**

*Admitted.*

$\square$

## 22.3.2 Strong Progress and Normal Forms

The definition of single-step reduction for our toy language is fairly simple, but for a larger language it would be easy to forget one of the rules and accidentally create a situation where some term cannot take a step even though it has not been completely reduced to a value. The following theorem shows that we did not, in fact, make such a mistake here.

*Theorem (Strong Progress):* If  $t$  is a term, then either  $t$  is a value or else there exists a term  $t'$  such that  $t \Rightarrow t'$ .

*Proof:* By induction on  $t$ .

- Suppose  $t = C\ n$ . Then  $t$  is a value.
- Suppose  $t = P\ t1\ t2$ , where (by the IH)  $t1$  either is a value or can step to some  $t1'$ , and where  $t2$  is either a value or can step to some  $t2'$ . We must show  $P\ t1\ t2$  is either a value or steps to some  $t'$ .
  - If  $t1$  and  $t2$  are both values, then  $t$  can take a step, by `ST_PlusConstConst`.
  - If  $t1$  is a value and  $t2$  can take a step, then so can  $t$ , by `ST_Plus2`.
  - If  $t1$  can take a step, then so can  $t$ , by `ST_Plus1`.  $\square$

Or, formally:

`Theorem strong_progress :  $\forall\ t$ ,  
 value  $t \vee (\exists\ t',\ t \Rightarrow t')$ .`

*Proof.*

```

induction t.
- left. apply v_const.
- right. inversion IHt1.
  + inversion IHt2.
    × inversion H. inversion H0.
       $\exists\ (C\ (n + n0))$ .
      apply ST_PlusConstConst.
    × inversion H0 as [t' H1].
       $\exists\ (P\ t1\ t')$ .
      apply ST_Plus2. apply H. apply H1.
  + inversion H as [t' H0].
     $\exists\ (P\ t'\ t2)$ .
    apply ST_Plus1. apply H0. Qed.

```

This important property is called *strong progress*, because every term either is a value or can "make progress" by stepping to some other term. (The qualifier "strong" distinguishes it from a more refined version that we'll see in later chapters, called just *progress*.)

The idea of "making progress" can be extended to tell us something interesting about values: in this language, values are exactly the terms that *cannot* make progress in this sense.

To state this observation formally, let's begin by giving a name to terms that cannot make progress. We'll call them *normal forms*.

**Definition** `normal_form`  $\{X:\text{Type}\}$   $(R:\text{relation } X)$   $(t:X) : \text{Prop} :=$   
 $\neg \exists\ t',\ R\ t\ t'$ .

Note that this definition specifies what it is to be a normal form for an *arbitrary* relation  $R$  over an arbitrary set  $X$ , not just for the particular single-step reduction relation over terms that we are interested in at the moment. We'll re-use the same terminology for talking about other relations later in the course.

We can use this terminology to generalize the observation we made in the strong progress theorem: in this language, normal forms and values are actually the same thing.

```
Lemma value_is_nf :  $\forall v$ ,
  value  $v \rightarrow$  normal_form step  $v$ .
```

Proof.

```
  unfold normal_form. intros  $v$   $H$ . inversion  $H$ .
  intros contra. inversion contra. inversion  $H1$ .
```

Qed.

```
Lemma nf_is_value :  $\forall t$ ,
  normal_form step  $t \rightarrow$  value  $t$ .
```

Proof. unfold normal\_form. intros  $t$   $H$ .

```
  assert ( $G : \text{value } t \vee \exists t', t ==> t'$ ).
  { apply strong_progress. }
  inversion  $G$ .
  + apply  $H0$ .
  + ex falso. apply  $H$ . assumption. Qed.
```

```
Corollary nf_same_as_value :  $\forall t$ ,
  normal_form step  $t \leftrightarrow$  value  $t$ .
```

Proof.

```
  split. apply nf_is_value. apply value_is_nf. Qed.
```

Why is this interesting?

It's interesting, because **value** is a syntactic concept – it is defined by looking at the form of a term – while **normal\_form** is a semantic one – it is defined by looking at how the term steps. It is not obvious that these concepts should coincide! Indeed, we could easily have written the definitions so that they would *not* coincide.

**Exercise: 3 stars, optional (value\_not\_same\_as\_normal\_form)** We might, for example, mistakenly define **value** so that it includes some terms that are not finished reducing. (Even if you don't work this exercise and the following ones in Coq, make sure you can think of an example of such a term.)

Module TEMP1.

```
Inductive value : tm  $\rightarrow$  Prop :=
| v_const :  $\forall n$ , value (C  $n$ )
| v_funny :  $\forall t1\ n2$ ,
  value (P  $t1$  (C  $n2$ )).
```

```
Reserved Notation "  $t ==>$  "  $t$  " (at level 40).
```



```

Inductive step : tm → tm → Prop :=
| ST_PlusConstConst : ∀ n1 n2,
  P (C n1) (C n2) ==> C (n1 + n2)
| ST_Plus1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  P t1 t2 ==> P t1' t2
| ST_Plus2 : ∀ v1 t2 t2',
  value v1 →
  t2 ==> t2' →
  P v1 t2 ==> P v1 t2'

```

where " t '==>' t' " := (**step** t t').

Lemma value\_not\_same\_as\_normal\_form :

∃ v, **value** v ∧ ¬ normal\_form **step** v.

Proof.

*Admitted.*

□ End TEMP1.

**Exercise: 2 stars, optional (value\_not\_same\_as\_normal\_form)** Alternatively, we might mistakenly define **step** so that it permits something designated as a value to reduce further.

Module TEMP2.

```

Inductive value : tm → Prop :=
| v_const : ∀ n, value (C n).

```

Reserved Notation " t '==>' t' " (at level 40).

```

Inductive step : tm → tm → Prop :=

```

```

| ST_Funny : ∀ n,
  C n ==> P (C n) (C 0)
| ST_PlusConstConst : ∀ n1 n2,
  P (C n1) (C n2) ==> C (n1 + n2)
| ST_Plus1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  P t1 t2 ==> P t1' t2
| ST_Plus2 : ∀ v1 t2 t2',
  value v1 →
  t2 ==> t2' →
  P v1 t2 ==> P v1 t2'

```

where " t '==>' t' " := (**step** t t').

Lemma value\_not\_same\_as\_normal\_form :

∃ v, **value** v ∧ ¬ normal\_form **step** v.

Proof.

*Admitted.*

□ End TEMP2.

**Exercise: 3 stars, optional (value\_not\_same\_as\_normal\_form')** Finally, we might define **value** and **step** so that there is some term that is not a value but that cannot take a step in the **step** relation. Such terms are said to be *stuck*. In this case this is caused by a mistake in the semantics, but we will also see situations where, even in a correct language definition, it makes sense to allow some terms to be stuck.

Module TEMP3.

Inductive **value** : **tm** → Prop :=

| v\_const : ∀ n, **value** (C n).

Reserved Notation "t '==>' t'" (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

| ST\_PlusConstConst : ∀ n1 n2,  
P (C n1) (C n2) ==> C (n1 + n2)

| ST\_Plus1 : ∀ t1 t1' t2,  
t1 ==> t1' →  
P t1 t2 ==> P t1' t2

where "t '==>' t'" := (**step** t t').

(Note that ST\_Plus2 is missing.)

Lemma value\_not\_same\_as\_normal\_form :

∃ t, ¬ **value** t ∧ normal\_form **step** t.

Proof.

*Admitted.*

□

End TEMP3.

## Additional Exercises

Module TEMP4.

Here is another very simple language whose terms, instead of being just addition expressions and numbers, are just the booleans true and false and a conditional expression...

Inductive **tm** : Type :=

| ttrue : **tm**  
| tfalse : **tm**  
| tif : **tm** → **tm** → **tm** → **tm**.

Inductive **value** : **tm** → Prop :=

```

| v_true : value ttrue
| v_false : value tfalse.

```

Reserved Notation " t '==>' t' " (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

```

| ST_IfTrue : ∀ t1 t2,
    tif ttrue t1 t2 ==> t1
| ST_IfFalse : ∀ t1 t2,
    tif tfalse t1 t2 ==> t2
| ST_If : ∀ t1 t1' t2 t3,
    t1 ==> t1' →
    tif t1 t2 t3 ==> tif t1' t2 t3

```

where " t '==>' t' " := (**step** t t').

**Exercise: 1 star (smallstep\_bools)** Which of the following propositions are provable? (This is just a thought exercise, but for an extra challenge feel free to prove your answers in Coq.)

Definition bool\_step\_prop1 :=  
 tfalse ==> tfalse.

Definition bool\_step\_prop2 :=  
 tif  
 ttrue  
 (tif ttrue ttrue ttrue)  
 (tif tfalse tfalse tfalse)  
 ==>  
 ttrue.

Definition bool\_step\_prop3 :=  
 tif  
 (tif ttrue ttrue ttrue)  
 (tif ttrue ttrue ttrue)  
 tfalse  
 ==>  
 tif  
 ttrue  
 (tif ttrue ttrue ttrue)  
 tfalse.

□

**Exercise: 3 stars, optional (progress\_bool)** Just as we proved a progress theorem for plus expressions, we can do so for boolean expressions, as well.

Theorem strong\_progress :  $\forall t,$   
**value**  $t \vee (\exists t', t \Rightarrow t')$ .

Proof.

*Admitted.*

□

**Exercise: 2 stars, optional (step\_deterministic)** Theorem step\_deterministic :  
deterministic **step**.

Proof.

*Admitted.*

□

Module TEMP5.

**Exercise: 2 stars (smallstep\_bool\_shortcut)** Suppose we want to add a "short circuit" to the step relation for boolean expressions, so that it can recognize when the **then** and **else** branches of a conditional are the same value (either **ttrue** or **tfalse**) and reduce the whole conditional to this value in a single step, even if the guard has not yet been reduced to a value. For example, we would like this proposition to be provable:  $\text{tif}(\text{tif } \text{ttrue } \text{ttrue } \text{ttrue}) \text{tfalse} \Rightarrow \text{tfalse}$ .

Write an extra clause for the step relation that achieves this effect and prove **bool\_step\_prop4**.

Reserved Notation "  $t' \Rightarrow t$  " (at level 40).

Inductive **step** : **tm**  $\rightarrow$  **tm**  $\rightarrow$  Prop :=

| ST\_IfTrue :  $\forall t1\ t2,$   
 $\text{tif } \text{ttrue } t1\ t2 \Rightarrow t1$   
| ST\_IfFalse :  $\forall t1\ t2,$   
 $\text{tif } \text{tfalse } t1\ t2 \Rightarrow t2$   
| ST\_If :  $\forall t1\ t1'\ t2\ t3,$   
 $t1 \Rightarrow t1' \rightarrow$   
 $\text{tif } t1\ t2\ t3 \Rightarrow \text{tif } t1'\ t2\ t3$

where "  $t' \Rightarrow t$  " := (**step**  $t\ t'$ ).

Definition bool\_step\_prop4 :=

$\text{tif}$   
 $(\text{tif } \text{ttrue } \text{ttrue } \text{ttrue})$   
 $\text{tfalse}$   
 $\text{tfalse}$   
 $\Rightarrow$   
 $\text{tfalse}.$

Example bool\_step\_prop4\_holds :

bool\_step\_prop4.

Proof.

*Admitted.*

□

**Exercise: 3 stars, optional (properties\_of\_altered\_step)** It can be shown that the determinism and strong progress theorems for the step relation in the lecture notes also hold for the definition of step given above. After we add the clause *ST\_ShortCircuit...*

- Is the **step** relation still deterministic? Write yes or no and briefly (1 sentence) explain your answer.

Optional: prove your answer correct in Coq.

- Does a strong progress theorem hold? Write yes or no and briefly (1 sentence) explain your answer.

Optional: prove your answer correct in Coq.

- In general, is there any way we could cause strong progress to fail if we took away one or more constructors from the original step relation? Write yes or no and briefly (1 sentence) explain your answer.

□

End TEMP5.

End TEMP4.

## 22.4 Multi-Step Reduction

We've been working so far with the *single-step reduction* relation  $\Rightarrow$ , which formalizes the individual steps of an abstract machine for executing programs.

We can use the same machine to reduce programs to completion – to find out what final result they yield. This can be formalized as follows:

- First, we define a *multi-step reduction relation*  $\Rightarrow^*$ , which relates terms  $t$  and  $t'$  if  $t$  can reach  $t'$  by any number (including zero) of single reduction steps.
- Then we define a "result" of a term  $t$  as a normal form that  $t$  can reach by multi-step reduction.

Since we'll want to reuse the idea of multi-step reduction many times, let's take a little extra trouble and define it generically.

Given a relation  $R$ , we define a relation **multi**  $R$ , called the *multi-step closure of  $R$*  as follows.

```
Inductive multi {X:Type} (R: relation X) : relation X :=
| multi_refl : ∀ (x : X), multi R x x
| multi_step : ∀ (x y z : X),
    R x y →
    multi R y z →
    multi R x z.
```

(In the Rel chapter and the Coq standard library, this relation is called **clos\_refl\_trans\_1n**. We give it a shorter name here for the sake of readability.)

The effect of this definition is that **multi**  $R$  relates two elements  $x$  and  $y$  if

- $x = y$ , or
- $R \ x \ y$ , or
- there is some nonempty sequence  $z1, z2, \dots, zn$  such that  $R \ x \ z1 \ R \ z1 \ z2 \ \dots \ R \ zn \ y$ .

Thus, if  $R$  describes a single-step of computation, then  $z1\dots zn$  is the sequence of intermediate steps of computation between  $x$  and  $y$ .

We write  $==>^*$  for the **multi step** relation on terms.

Notation "  $t' ==>^* t$  " := (**multi step**  $t \ t'$ ) (at level 40).

The relation **multi**  $R$  has several crucial properties.

First, it is obviously *reflexive* (that is,  $\forall x, \text{multi } R \ x \ x$ ). In the case of the  $==>^*$  (i.e., **multi step**) relation, the intuition is that a term can execute to itself by taking zero steps of execution.

Second, it contains  $R$  – that is, single-step executions are a particular case of multi-step executions. (It is this fact that justifies the word "closure" in the term "multi-step closure of  $R$ .")

```
Theorem multi_R : ∀ (X:Type) (R:relation X) (x y : X),
    R x y → (multi R) x y.
```

Proof.

```
intros X R x y H.
```

```
apply multi_step with y. apply H. apply multi_refl. Qed.
```

Third, **multi**  $R$  is *transitive*.

```
Theorem multi_trans :
  ∀ (X:Type) (R: relation X) (x y z : X),
    multi R x y →
    multi R y z →
```

**multi**  $R\ x\ z$ .

Proof.

intros  $X\ R\ x\ y\ z\ G\ H$ .

induction  $G$ .

- assumption.

-

apply multi\_step with  $y$ . assumption.

apply  $IHG$ . assumption. Qed.

In particular, for the **multi step** relation on terms, if  $t1 ==>^* t2$  and  $t2 ==>^* t3$ , then  $t1 ==>^* t3$ .

## 22.4.1 Examples

Here's a specific instance of the **multi step** relation:

Lemma test\_multistep\_1:

P

(P (C 0) (C 3))

(P (C 2) (C 4))

==>\*

C ((0 + 3) + (2 + 4)).

Proof.

apply multi\_step with

(P

(C (0 + 3))

(P (C 2) (C 4))).

apply ST\_Plus1. apply ST\_PlusConstConst.

apply multi\_step with

(P

(C (0 + 3))

(C (2 + 4))).

apply ST\_Plus2. apply v\_const.

apply ST\_PlusConstConst.

apply multi\_R.

apply ST\_PlusConstConst. Qed.

Here's an alternate proof of the same fact that uses **eapply** to avoid explicitly constructing all the intermediate terms.

Lemma test\_multistep\_1':

P

(P (C 0) (C 3))

(P (C 2) (C 4))

==>\*

$C ((0 + 3) + (2 + 4)).$

Proof.

```
eapply multi_step. apply ST_Plus1. apply ST_PlusConstConst.
eapply multi_step. apply ST_Plus2. apply v_const.
apply ST_PlusConstConst.
eapply multi_step. apply ST_PlusConstConst.
apply multi_refl. Qed.
```

**Exercise: 1 star, optional (test\_multistep\_2)** Lemma test\_multistep\_2:

$C\ 3 ==>* C\ 3.$

Proof.

*Admitted.*

□

**Exercise: 1 star, optional (test\_multistep\_3)** Lemma test\_multistep\_3:

$P\ (C\ 0)\ (C\ 3)$   
 $==>* P\ (C\ 0)\ (C\ 3).$

Proof.

*Admitted.*

□

**Exercise: 2 stars (test\_multistep\_4)** Lemma test\_multistep\_4:

$P$   
 $(C\ 0)$   
 $(P$   
 $(C\ 2)$   
 $(P\ (C\ 0)\ (C\ 3)))$   
 $==>* P$   
 $(C\ 0)$   
 $(C\ (2 + (0 + 3))).$

Proof.

*Admitted.*

□

## 22.4.2 Normal Forms Again

If  $t$  reduces to  $t'$  in zero or more steps and  $t'$  is a normal form, we say that " $t'$  is a normal form of  $t$ ."

**Definition** step\_normal\_form := normal\_form **step**.

**Definition** normal\_form\_of ( $t\ t' : \mathbf{tm}$ ) :=



$(t \Rightarrow^* t' \wedge \text{step\_normal\_form } t')$ .

We have already seen that, for our language, single-step reduction is deterministic – i.e., a given term can take a single step in at most one way. It follows from this that, if  $t$  can reach a normal form, then this normal form is unique. In other words, we can actually pronounce  $\text{normal\_form } t$  as “ $t$  is the normal form of  $t$ .”

**Exercise: 3 stars, optional (normal-forms-unique)** Theorem `normal-forms-unique`:  
deterministic `normal_form_of`.

Proof.

```
unfold deterministic. unfold normal_form_of.
intros x y1 y2 P1 P2.
inversion P1 as [P11 P12]; clear P1.
inversion P2 as [P21 P22]; clear P2.
generalize dependent y2.
Admitted.
□
```

Indeed, something stronger is true for this language (though not for all languages): the reduction of *any* term  $t$  will eventually reach a normal form – i.e., `normal_form_of` is a *total* function. Formally, we say the **step** relation is *normalizing*.

Definition `normalizing {X:Type} (R:relation X) :=`

```
∀ t, ∃ t',
  (multi R) t t' ∧ normal_form R t'.
```

To prove that **step** is normalizing, we need a couple of lemmas.

First, we observe that, if  $t$  reduces to  $t'$  in many steps, then the same sequence of reduction steps within  $t$  is also possible when  $t$  appears as the left-hand child of a  $P$  node, and similarly when  $t$  appears as the right-hand child of a  $P$  node whose left-hand child is a value.

Lemma `multistep_congr_1` :  $\forall t1\ t1'\ t2,$   
 $t1 \Rightarrow^* t1' \rightarrow$   
 $P\ t1\ t2 \Rightarrow^* P\ t1'\ t2.$

Proof.

```
intros t1 t1' t2 H. induction H.
- apply multi_refl.
- apply multi_step with (P y t2).
  apply ST_Plus1. apply H.
  apply IHmulti. Qed.
```

**Exercise: 2 stars (multistep\_congr\_2)** Lemma `multistep_congr_2` :  $\forall t1\ t2\ t2',$

```
value t1 →
t2 ⇒* t2' →
P t1 t2 ⇒* P t1 t2'.
```

Proof.

*Admitted.*

□

With these lemmas in hand, the main proof is a straightforward induction.

*Theorem:* The **step** function is normalizing – i.e., for every  $t$  there exists some  $t'$  such that  $t$  steps to  $t'$  and  $t'$  is a normal form.

*Proof sketch:* By induction on terms. There are two cases to consider:

- $t = C\ n$  for some  $n$ . Here  $t$  doesn't take a step, and we have  $t' = t$ . We can derive the left-hand side by reflexivity and the right-hand side by observing (a) that values are normal forms (by `nf_same_as_value`) and (b) that  $t$  is a value (by `v_const`).
- $t = P\ t1\ t2$  for some  $t1$  and  $t2$ . By the IH,  $t1$  and  $t2$  have normal forms  $t1'$  and  $t2'$ . Recall that normal forms are values (by `nf_same_as_value`); we know that  $t1' = C\ n1$  and  $t2' = C\ n2$ , for some  $n1$  and  $n2$ . We can combine the  $==>^*$  derivations for  $t1$  and  $t2$  using `multi_congr_1` and `multi_congr_1` to prove that  $P\ t1\ t2$  reduces in many steps to  $C\ (n1 + n2)$ .

It is clear that our choice of  $t' = C\ (n1 + n2)$  is a value, which is in turn a normal form. □

Theorem `step_normalizing` :  
normalizing **step**.

Proof.

`unfold normalizing.`

`induction t.`

-

`∃ (C n).`

`split.`

`+ apply multi_refl.`

`+`

`rewrite nf_same_as_value. apply v_const.`

-

`inversion IHt1 as [t1' H1]; clear IHt1. inversion IHt2 as [t2' H2]; clear IHt2.`

`inversion H1 as [H11 H12]; clear H1. inversion H2 as [H21 H22]; clear H2.`

`rewrite nf_same_as_value in H12. rewrite nf_same_as_value in H22.`

`inversion H12 as [n1]. inversion H22 as [n2].`

`rewrite ← H in H11.`

`rewrite ← H0 in H21.`

`∃ (C (n1 + n2)).`

`split.`

`+`

`apply multi_trans with (P (C n1) t2).`

```

    apply multistep_congr_1. apply H11.
    apply multi_trans with
      (P (C n1) (C n2)).
    apply multistep_congr_2. apply v_const. apply H21.
    apply multi_R. apply ST_PlusConstConst.
+
  rewrite nf_same_as_value. apply v_const. Qed.

```

### 22.4.3 Equivalence of Big-Step and Small-Step Reduction

Having defined the operational semantics of our tiny programming language in two different ways (big-step and small-step), it makes sense to ask whether these definitions actually define the same thing! They do, though it takes a little work to show it. The details are left as an exercise.

**Exercise: 3 stars (eval\_\_multistep)** Theorem `eval__multistep` :  $\forall t n,$   
 $t \setminus n \rightarrow t ==>^* C n.$

The key ideas in the proof can be seen in the following picture:  $P\ t1\ t2 ==>$  (by `ST_Plus1`)  $P\ t1'\ t2 ==>$  (by `ST_Plus1`)  $P\ t1''\ t2 ==>$  (by `ST_Plus1`) ...  $P\ (C\ n1)\ t2 ==>$  (by `ST_Plus2`)  $P\ (C\ n1)\ t2' ==>$  (by `ST_Plus2`)  $P\ (C\ n1)\ t2'' ==>$  (by `ST_Plus2`) ...  $P\ (C\ n1)\ (C\ n2) ==>$  (by `ST_PlusConstConst`)  $C\ (n1 + n2)$  That is, the multistep reduction of a term of the form  $P\ t1\ t2$  proceeds in three phases:

- First, we use `ST_Plus1` some number of times to reduce  $t1$  to a normal form, which must (by `nf_same_as_value`) be a term of the form  $C\ n1$  for some  $n1$ .
- Next, we use `ST_Plus2` some number of times to reduce  $t2$  to a normal form, which must again be a term of the form  $C\ n2$  for some  $n2$ .
- Finally, we use `ST_PlusConstConst` one time to reduce  $P\ (C\ n1)\ (C\ n2)$  to  $C\ (n1 + n2)$ .

To formalize this intuition, you'll need to use the congruence lemmas from above (you might want to review them now, so that you'll be able to recognize when they are useful), plus some basic properties of  $==>^*$ : that it is reflexive, transitive, and includes  $==>$ .

**Proof.**

*Admitted.*

□

**Exercise: 3 stars, advanced (eval\_\_multistep\_inf)** Write a detailed informal version of the proof of `eval__multistep`.

□

For the other direction, we need one lemma, which establishes a relation between single-step reduction and big-step evaluation.

**Exercise: 3 stars (step\_eval)** Lemma `step_eval` :  $\forall t t' n,$

$t \Rightarrow t' \rightarrow$

$t' \ll n \rightarrow$

$t \ll n.$

Proof.

`intros t t' n Hs. generalize dependent n.`

*Admitted.*

□

The fact that small-step reduction implies big-step is now straightforward to prove, once it is stated correctly.

The proof proceeds by induction on the multi-step reduction sequence that is buried in the hypothesis `normal_form_of t t'`. Make sure you understand the statement before you start to work on the proof.

**Exercise: 3 stars (multistep\_eval)** Theorem `multistep_eval` :  $\forall t t',$

`normal_form_of t t' → ∃ n, t' = C n ∧ t ≪ n.`

Proof.

*Admitted.*

□

## 22.4.4 Additional Exercises

**Exercise: 3 stars, optional (interp\_tm)** Remember that we also defined big-step evaluation of terms as a function `evalF`. Prove that it is equivalent to the existing semantics. (Hint: we just proved that `eval` and `multistep` are equivalent, so logically it doesn't matter which you choose. One will be easier than the other, though!)

Theorem `evalF_eval` :  $\forall t n,$

`evalF t = n ↔ t ≪ n.`

Proof.

*Admitted.*

□

**Exercise: 4 stars (combined\_properties)** We've considered arithmetic and conditional expressions separately. This exercise explores how the two interact.

Module `COMBINED`.

Inductive `tm` : Type :=

| `C` : `nat` → `tm`

| `P` : `tm` → `tm` → `tm`

| `ttrue` : `tm`

| `tfalse` : `tm`

| `tif` : `tm` → `tm` → `tm` → `tm`.

```

Inductive value : tm → Prop :=
| v_const : ∀ n, value (C n)
| v_true : value ttrue
| v_false : value tfalse.

```

Reserved Notation "t'==>t'" (at level 40).

```

Inductive step : tm → tm → Prop :=
| ST_PlusConstConst : ∀ n1 n2,
  P (C n1) (C n2) ==> C (n1 + n2)
| ST_Plus1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  P t1 t2 ==> P t1' t2
| ST_Plus2 : ∀ v1 t2 t2',
  value v1 →
  t2 ==> t2' →
  P v1 t2 ==> P v1 t2'
| ST_IfTrue : ∀ t1 t2,
  tif ttrue t1 t2 ==> t1
| ST_IfFalse : ∀ t1 t2,
  tif tfalse t1 t2 ==> t2
| ST_If : ∀ t1 t1' t2 t3,
  t1 ==> t1' →
  tif t1 t2 t3 ==> tif t1' t2 t3

```

where "t'==>t'" := (step t t').

Earlier, we separately proved for both plus- and if-expressions...

- that the step relation was deterministic, and
- a strong progress lemma, stating that every term is either a value or can take a step.

Prove or disprove these two properties for the combined language.

□

End COMBINED.

## 22.5 Small-Step Imp

Now for a more serious example: a small-step version of the Imp operational semantics.

The small-step evaluation relations for arithmetic and boolean expressions are straightforward extensions of the tiny language we've been working up to now. To make them easier to read, we introduce the symbolic notations  $\Rightarrow_a$  and  $\Rightarrow_b$  for the arithmetic and boolean step relations.

Inductive **aval** : **aexp** → Prop :=  
 av\_num : ∀ n, **aval** (ANum n).

We are not actually going to bother to define boolean values, since they aren't needed in the definition of  $\Rightarrow_b$  below (why?), though they might be if our language were a bit larger (why?).

Reserved Notation " t '/' st '==>a' t' " (at level 40, st at level 39).

Inductive **astep** : state → **aexp** → **aexp** → Prop :=

| AS\_Id : ∀ st i,  
     Ald i / st ==>a ANum (st i)  
 | AS\_Plus : ∀ st n1 n2,  
     APlus (ANum n1) (ANum n2) / st ==>a ANum (n1 + n2)  
 | AS\_Plus1 : ∀ st a1 a1' a2,  
     a1 / st ==>a a1' →  
     (APlus a1 a2) / st ==>a (APlus a1' a2)  
 | AS\_Plus2 : ∀ st v1 a2 a2',  
     **aval** v1 →  
     a2 / st ==>a a2' →  
     (APlus v1 a2) / st ==>a (APlus v1 a2')  
 | AS\_Minus : ∀ st n1 n2,  
     (AMinus (ANum n1) (ANum n2)) / st ==>a (ANum (minus n1 n2))  
 | AS\_Minus1 : ∀ st a1 a1' a2,  
     a1 / st ==>a a1' →  
     (AMinus a1 a2) / st ==>a (AMinus a1' a2)  
 | AS\_Minus2 : ∀ st v1 a2 a2',  
     **aval** v1 →  
     a2 / st ==>a a2' →  
     (AMinus v1 a2) / st ==>a (AMinus v1 a2')  
 | AS\_Mult : ∀ st n1 n2,  
     (AMult (ANum n1) (ANum n2)) / st ==>a (ANum (mult n1 n2))  
 | AS\_Mult1 : ∀ st a1 a1' a2,  
     a1 / st ==>a a1' →  
     (AMult (a1) (a2)) / st ==>a (AMult (a1') (a2))  
 | AS\_Mult2 : ∀ st v1 a2 a2',  
     **aval** v1 →  
     a2 / st ==>a a2' →  
     (AMult v1 a2) / st ==>a (AMult v1 a2')

where " t '/' st '==>a' t' " := (**astep** st t t').

Reserved Notation " t '/' st '==>b' t' " (at level 40, st at level 39).

Inductive **bstep** : state → **bexp** → **bexp** → Prop :=

| BS\_Eq : ∀ st n1 n2,

```

      (BEq (ANum n1) (ANum n2)) / st ==>b
      (if (beq_nat n1 n2) then BTrue else BFalse)
| BS_Eq1 : ∀ st a1 a1' a2,
  a1 / st ==>a a1' →
  (BEq a1 a2) / st ==>b (BEq a1' a2)
| BS_Eq2 : ∀ st v1 a2 a2',
  aval v1 →
  a2 / st ==>a a2' →
  (BEq v1 a2) / st ==>b (BEq v1 a2')
| BS_LtEq : ∀ st n1 n2,
  (BLe (ANum n1) (ANum n2)) / st ==>b
  (if (leb n1 n2) then BTrue else BFalse)
| BS_LtEq1 : ∀ st a1 a1' a2,
  a1 / st ==>a a1' →
  (BLe a1 a2) / st ==>b (BLe a1' a2)
| BS_LtEq2 : ∀ st v1 a2 a2',
  aval v1 →
  a2 / st ==>a a2' →
  (BLe v1 a2) / st ==>b (BLe v1 (a2'))
| BS_NotTrue : ∀ st,
  (BNot BTrue) / st ==>b BFalse
| BS_NotFalse : ∀ st,
  (BNot BFalse) / st ==>b BTrue
| BS_NotStep : ∀ st b1 b1',
  b1 / st ==>b b1' →
  (BNot b1) / st ==>b (BNot b1')
| BS_AndTrueTrue : ∀ st,
  (BAnd BTrue BTrue) / st ==>b BTrue
| BS_AndTrueFalse : ∀ st,
  (BAnd BTrue BFalse) / st ==>b BFalse
| BS_AndFalse : ∀ st b2,
  (BAnd BFalse b2) / st ==>b BFalse
| BS_AndTrueStep : ∀ st b2 b2',
  b2 / st ==>b b2' →
  (BAnd BTrue b2) / st ==>b (BAnd BTrue b2')
| BS_AndStep : ∀ st b1 b1' b2,
  b1 / st ==>b b1' →
  (BAnd b1 b2) / st ==>b (BAnd b1' b2)

```

where "t '/' st '==>b' t' " := (**bstep** st t t').

The semantics of commands is the interesting part. We need two small tricks to make it work:

- We use *SKIP* as a "command value" – i.e., a command that has reached a normal form.
  - An assignment command reduces to *SKIP* (and an updated state).
  - The sequencing command waits until its left-hand subcommand has reduced to *SKIP*, then throws it away so that reduction can continue with the right-hand subcommand.
- We reduce a *WHILE* command by transforming it into a conditional followed by the same *WHILE*.

(There are other ways of achieving the effect of the latter trick, but they all share the feature that the original *WHILE* command needs to be saved somewhere while a single copy of the loop body is being evaluated.)

Reserved Notation " $t \text{ '/' } st \text{ '==>' } t' \text{ '/' } st'$ "  
(at level 40,  $st$  at level 39,  $t'$  at level 39).

Inductive **cstep** : (**com**  $\times$  state)  $\rightarrow$  (**com**  $\times$  state)  $\rightarrow$  Prop :=

```

| CS_AssStep :  $\forall st \ i \ a \ a',$ 
   $a \ / \ st \ ==> a \ a' \rightarrow$ 
   $(i ::= a) \ / \ st \ ==> (i ::= a') \ / \ st$ 
| CS_Ass :  $\forall st \ i \ n,$ 
   $(i ::= (ANum \ n)) \ / \ st \ ==> SKIP \ / \ (t\_update \ st \ i \ n)$ 
| CS_SeqStep :  $\forall st \ c1 \ c1' \ st' \ c2,$ 
   $c1 \ / \ st \ ==> c1' \ / \ st' \rightarrow$ 
   $(c1 ;; c2) \ / \ st \ ==> (c1' ;; c2) \ / \ st'$ 
| CS_SeqFinish :  $\forall st \ c2,$ 
   $(SKIP ;; c2) \ / \ st \ ==> c2 \ / \ st$ 
| CS_IfTrue :  $\forall st \ c1 \ c2,$ 
   $IFB \ BTrue \ THEN \ c1 \ ELSE \ c2 \ FI \ / \ st \ ==> c1 \ / \ st$ 
| CS_IfFalse :  $\forall st \ c1 \ c2,$ 
   $IFB \ BFalse \ THEN \ c1 \ ELSE \ c2 \ FI \ / \ st \ ==> c2 \ / \ st$ 
| CS_IfStep :  $\forall st \ b \ b' \ c1 \ c2,$ 
   $b \ / \ st \ ==> b \ b' \rightarrow$ 
   $IFB \ b \ THEN \ c1 \ ELSE \ c2 \ FI \ / \ st$ 
   $==> (IFB \ b' \ THEN \ c1 \ ELSE \ c2 \ FI) \ / \ st$ 
| CS_While :  $\forall st \ b \ c1,$ 
   $(WHILE \ b \ DO \ c1 \ END) \ / \ st$ 
   $==> (IFB \ b \ THEN \ (c1 ;; (WHILE \ b \ DO \ c1 \ END)) \ ELSE \ SKIP \ FI) \ / \ st$ 

```

where " $t \text{ '/' } st \text{ '==>' } t' \text{ '/' } st'$ " := (**cstep** ( $t, st$ ) ( $t', st'$ )).



## 22.6 Concurrent Imp

Finally, to show the power of this definitional style, let's enrich Imp with a new form of command that runs two subcommands in parallel and terminates when both have terminated. To reflect the unpredictability of scheduling, the actions of the subcommands may be interleaved in any order, but they share the same memory and can communicate by reading and writing the same variables.

Module CIMP.

Inductive **com** : Type :=

- | CSkip : **com**
- | CAss : **id** → **aexp** → **com**
- | CSeq : **com** → **com** → **com**
- | Clf : **bexp** → **com** → **com** → **com**
- | CWhile : **bexp** → **com** → **com**
- | CPar : **com** → **com** → **com**.

Notation "'SKIP'" :=

CSkip.

Notation "x ' ::= ' a" :=

(CAss x a) (at level 60).

Notation "c1 ;; c2" :=

(CSeq c1 c2) (at level 80, right associativity).

Notation "'WHILE' b 'DO' c 'END'" :=

(CWhile b c) (at level 80, right associativity).

Notation "'IFB' b 'THEN' c1 'ELSE' c2 'FI'" :=

(Clf b c1 c2) (at level 80, right associativity).

Notation "'PAR' c1 'WITH' c2 'END'" :=

(CPar c1 c2) (at level 80, right associativity).

Inductive **cstep** : (**com** × state) → (**com** × state) → Prop :=

- | CS\_AssStep : ∀ st i a a',  
a / st ==> a a' →  
(i ::= a) / st ==> (i ::= a') / st
- | CS\_Ass : ∀ st i n,  
(i ::= (ANum n)) / st ==> SKIP / (t\_update st i n)
- | CS\_SeqStep : ∀ st c1 c1' st' c2,  
c1 / st ==> c1' / st' →  
(c1 ;; c2) / st ==> (c1' ;; c2) / st'
- | CS\_SeqFinish : ∀ st c2,  
(SKIP ;; c2) / st ==> c2 / st
- | CS\_IfTrue : ∀ st c1 c2,  
(IFB BTrue THEN c1 ELSE c2 FI) / st ==> c1 / st

```

| CS_IfFalse :  $\forall st\ c1\ c2,$ 
  (IFB BFalse THEN  $c1$  ELSE  $c2$  FI) /  $st \implies c2$  /  $st$ 
| CS_IfStep :  $\forall st\ b\ b'\ c1\ c2,$ 
   $b / st \implies b' \rightarrow$ 
  (IFB  $b$  THEN  $c1$  ELSE  $c2$  FI) /  $st$ 
   $\implies$  (IFB  $b'$  THEN  $c1$  ELSE  $c2$  FI) /  $st$ 
| CS_While :  $\forall st\ b\ c1,$ 
  (WHILE  $b$  DO  $c1$  END) /  $st$ 
   $\implies$  (IFB  $b$  THEN ( $c1$ ;; (WHILE  $b$  DO  $c1$  END)) ELSE SKIP FI) /  $st$ 

| CS_Par1 :  $\forall st\ c1\ c1'\ c2\ st',$ 
   $c1 / st \implies c1' / st' \rightarrow$ 
  (PAR  $c1$  WITH  $c2$  END) /  $st \implies$  (PAR  $c1'$  WITH  $c2$  END) /  $st'$ 
| CS_Par2 :  $\forall st\ c1\ c2\ c2'\ st',$ 
   $c2 / st \implies c2' / st' \rightarrow$ 
  (PAR  $c1$  WITH  $c2$  END) /  $st \implies$  (PAR  $c1$  WITH  $c2'$  END) /  $st'$ 
| CS_ParDone :  $\forall st,$ 
  (PAR SKIP WITH SKIP END) /  $st \implies$  SKIP /  $st$ 
where "  $t' / st' \implies^* t' / st'$  " := (cstep ( $t, st$ ) ( $t', st'$ )).

```

Definition cmultistep := **multi cstep**.

Notation "  $t' / st' \implies^* t' / st'$  " :=  
 (**multi cstep** ( $t, st$ ) ( $t', st'$ ))  
 (at level 40,  $st$  at level 39,  $t'$  at level 39).

Among the many interesting properties of this language is the fact that the following program can terminate with the variable  $X$  set to any value.

```

Definition par_loop : com :=
  PAR
    Y ::= ANum 1
  WITH
    WHILE BEq (Ald Y) (ANum 0) DO
      X ::= APlus (Ald X) (ANum 1)
    END
  END.

```

In particular, it can terminate with  $X$  set to 0:

Example par\_loop\_example\_0:

```

 $\exists st',$ 
  par_loop / empty_state  $\implies^*$  SKIP /  $st'$ 
 $\wedge st' X = 0.$ 

```

Proof.

```

eapply ex_intro. split.
unfold par_loop.

```

```

eapply multi_step. apply CS_Par1.
  apply CS_Ass.
eapply multi_step. apply CS_Par2. apply CS_While.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq1. apply AS_Id.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq. simpl.
eapply multi_step. apply CS_Par2. apply CS_IfFalse.
eapply multi_step. apply CS_ParDone.
eapply multi_refl.
reflexivity. Qed.

```

It can also terminate with  $X$  set to 2:

Example par\_loop\_example\_2:

```

 $\exists st',$ 
  par_loop / empty_state ==>* SKIP /  $st'$ 
 $\wedge st' X = 2.$ 

```

Proof.

```

eapply ex_intro. split.
eapply multi_step. apply CS_Par2. apply CS_While.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq1. apply AS_Id.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq. simpl.
eapply multi_step. apply CS_Par2. apply CS_IfTrue.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_AssStep. apply AS_Plus1. apply AS_Id.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_AssStep. apply AS_Plus.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_Ass.
eapply multi_step. apply CS_Par2. apply CS_SeqFinish.

eapply multi_step. apply CS_Par2. apply CS_While.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq1. apply AS_Id.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq. simpl.
eapply multi_step. apply CS_Par2. apply CS_IfTrue.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_AssStep. apply AS_Plus1. apply AS_Id.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.
  apply CS_AssStep. apply AS_Plus.
eapply multi_step. apply CS_Par2. apply CS_SeqStep.

```

```

    apply CS_Ass.
  eapply multi_step. apply CS_Par1. apply CS_Ass.
  eapply multi_step. apply CS_Par2. apply CS_SeqFinish.
  eapply multi_step. apply CS_Par2. apply CS_While.
  eapply multi_step. apply CS_Par2. apply CS_IfStep.
    apply BS_Eq1. apply AS_Id.
  eapply multi_step. apply CS_Par2. apply CS_IfStep.
    apply BS_Eq. simpl.
  eapply multi_step. apply CS_Par2. apply CS_IfFalse.
  eapply multi_step. apply CS_ParDone.
  eapply multi_refl.
  reflexivity. Qed.

  More generally...

```

**Exercise: 3 stars, optional** Lemma `par_body_n__Sn` :  $\forall n\ st,$   
 $st\ X = n \wedge st\ Y = 0 \rightarrow$   
 $par\_loop\ /\ st ==>* par\_loop\ /\ (t\_update\ st\ X\ (S\ n)).$   
 Proof.  
*Admitted.*  
 □

**Exercise: 3 stars, optional** Lemma `par_body_n` :  $\forall n\ st,$   
 $st\ X = 0 \wedge st\ Y = 0 \rightarrow$   
 $\exists st',$   
 $par\_loop\ /\ st ==>* par\_loop\ /\ st' \wedge st'\ X = n \wedge st'\ Y = 0.$   
 Proof.  
*Admitted.*  
 □  
 ... the above loop can exit with `X` having any value whatsoever.

**Theorem** `par_loop_any_X`:  
 $\forall n, \exists st',$   
 $par\_loop\ /\ empty\_state ==>* SKIP\ /\ st'$   
 $\wedge st'\ X = n.$

Proof.  
 intros *n*.  
 destruct (*par\_body\_n* *n* empty\_state).  
 split; unfold *t\_update*; reflexivity.  
 rename *x* into *st*.  
 inversion *H* as [*H'* [*HX HY*]]; clear *H*.  
 $\exists (t\_update\ st\ Y\ 1).$  split.  
 eapply multi\_trans with (*par\_loop*, *st*). apply *H'*.

```

eapply multi_step. apply CS_Par1. apply CS_Ass.
eapply multi_step. apply CS_Par2. apply CS_While.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq1. apply AS_Id. rewrite t_update_eq.
eapply multi_step. apply CS_Par2. apply CS_IfStep.
  apply BS_Eq. simpl.
eapply multi_step. apply CS_Par2. apply CS_IfFalse.
eapply multi_step. apply CS_ParDone.
eapply multi_refl.

rewrite t_update_neq. assumption. intro X; inversion X.
Qed.

End CIMP.

```

## 22.7 A Small-Step Stack Machine

Our last example is a small-step semantics for the stack machine example from the `Imp` chapter.

Definition `stack` := **list nat**.

Definition `prog` := **list sinstr**.

Inductive `stack_step` : `state`  $\rightarrow$  `prog`  $\times$  `stack`  $\rightarrow$  `prog`  $\times$  `stack`  $\rightarrow$  `Prop` :=

```

| SS_Push :  $\forall$  st stk n p',
  stack_step st (SPush n :: p', stk) (p', n :: stk)
| SS_Load :  $\forall$  st stk i p',
  stack_step st (SLoad i :: p', stk) (p', st i :: stk)
| SS_Plus :  $\forall$  st stk n m p',
  stack_step st (SPlus :: p', n :: m :: stk) (p', (m+n) :: stk)
| SS_Minus :  $\forall$  st stk n m p',
  stack_step st (SMinus :: p', n :: m :: stk) (p', (m-n) :: stk)
| SS_Mult :  $\forall$  st stk n m p',
  stack_step st (SMult :: p', n :: m :: stk) (p', (m  $\times$  n) :: stk).

```

Theorem `stack_step_deterministic` :  $\forall$  `st`,  
 deterministic (**stack\_step** `st`).

Proof.

```

unfold deterministic. intros st x y1 y2 H1 H2.
induction H1; inversion H2; reflexivity.

```

Qed.

Definition `stack_multistep` `st` := **multi** (**stack\_step** `st`).

**Exercise: 3 stars, advanced (compiler\_is\_correct)** Remember the definition of *compile* for `aexp` given in the `Imp` chapter. We want now to prove *compile* correct with respect

to the stack machine.

State what it means for the compiler to be correct according to the stack machine small step semantics and then prove it.

Definition `compiler_is_correct_statement` : Prop :=  
*admit*.

Theorem `compiler_is_correct` : `compiler_is_correct_statement`.

Proof.

*Admitted.*

□

*Date* : 2016 – 03 – 07 12 : 44 : 57 – 0500 (*Mon, 07 Mar 2016*)

# Chapter 23

## Library Auto

### 23.1 Auto: More Automation

Require Import **Coq.omega.Omega**.

Require Import Maps.

Require Import Imp.

Up to now, we've used a quite restricted set of Coq's tactic facilities. In this chapter, we'll learn more about two very powerful features of Coq's tactic language: proof search via the `auto` and `eauto` tactics, and automated forward reasoning via the Ltac hypothesis matching machinery. Using these features together with Ltac's scripting facilities will enable us to make our proofs startlingly short! Used properly, they can also make proofs more maintainable and robust in the face of incremental changes to underlying definitions.

There's a third major category of automation we haven't fully studied yet, namely built-in decision procedures for specific kinds of problems: `omega` is one example, but there are others. This topic will be deferred for a while longer.

Our motivating example will be this proof, repeated with just a few small changes from `Imp`. We will try to simplify this proof in several stages.

```
Ltac inv H := inversion H; subst; clear H.
```

Theorem `ceval_deterministic`:  $\forall c\ st\ st1\ st2,$

$c / st \ \backslash\ st1 \rightarrow$

$c / st \ \backslash\ st2 \rightarrow$

$st1 = st2.$

Proof.

```
intros c st st1 st2 E1 E2;
```

```
generalize dependent st2;
```

```
induction E1; intros st2 E2; inv E2.
```

```
- reflexivity.
```

```
- reflexivity.
```

```
-
```

```

    assert (st' = st'0) as EQ1.
    { apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption.
-
    apply IHE1. assumption.
-
    rewrite H in H5. inversion H5.
-
    rewrite H in H5. inversion H5.
-
    apply IHE1. assumption.
-
    reflexivity.
-
    rewrite H in H2. inversion H2.
-
    rewrite H in H4. inversion H4.
-
    assert (st' = st'0) as EQ1.
    { apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption. Qed.

```

## 23.2 The auto and eauto tactics

Thus far, we have generally written proof scripts that apply relevant hypotheses or lemmas by name. In particular, when a chain of hypothesis applications is needed, we have specified them explicitly. (The only exceptions we've seen to this are using `assumption` to find a matching unqualified hypothesis or `(e)constructor` to find a matching constructor.)

Example `auto_example_1` :  $\forall (P \ Q \ R: \text{Prop}), (P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$ .

Proof.

```

    intros P Q R H1 H2 H3.
    apply H2. apply H1. assumption.
Qed.

```

The `auto` tactic frees us from this drudgery by *searching* for a sequence of applications that will prove the goal

Example `auto_example_1'` :  $\forall (P \ Q \ R: \text{Prop}), (P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$ .

Proof.

```

    intros P Q R H1 H2 H3.
    auto.

```



Qed.

The `auto` tactic solves goals that are solvable by any combination of

- `intros`,
- `apply` (with a local hypothesis, by default).

The `eauto` tactic works just like `auto`, except that it uses `eapply` instead of `apply`.

Using `auto` is always "safe" in the sense that it will never fail and will never change the proof state: either it completely solves the current goal, or it does nothing.

A more complicated example:

Example `auto_example_2` :  $\forall P Q R S T U : \text{Prop}$ ,

$(P \rightarrow Q) \rightarrow$   
 $(P \rightarrow R) \rightarrow$   
 $(T \rightarrow R) \rightarrow$   
 $(S \rightarrow T \rightarrow U) \rightarrow$   
 $((P \rightarrow Q) \rightarrow (P \rightarrow S)) \rightarrow$   
 $T \rightarrow$   
 $P \rightarrow$   
 $U$ .

Proof. `auto`. Qed.

Search can take an arbitrarily long time, so there are limits to how far `auto` will search by default.

Example `auto_example_3` :  $\forall (P Q R S T U : \text{Prop})$ ,

$(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow (R \rightarrow S) \rightarrow$   
 $(S \rightarrow T) \rightarrow (T \rightarrow U) \rightarrow P \rightarrow U$ .

Proof.

`auto`.

`auto 6`.

Qed.

When searching for potential proofs of the current goal, `auto` and `eauto` consider the hypotheses in the current context together with a *hint database* of other lemmas and constructors. Some of the lemmas and constructors we've already seen – e.g., `eq_refl`, `conj`, `or_introl`, and `or_intror` – are installed in this hint database by default.

Example `auto_example_4` :  $\forall P Q R : \text{Prop}$ ,

$Q \rightarrow$   
 $(Q \rightarrow R) \rightarrow$   
 $P \vee (Q \wedge R)$ .

Proof.

`auto`. Qed.

If we want to see which facts `auto` is using, we can use `info_auto` instead.

Example auto\_example\_5:  $2 = 2$ .

Proof.

*info\_auto.*

Qed.

We can extend the hint database just for the purposes of one application of `auto` or `eauto` by writing `auto using ....`

Lemma le\_antisym :  $\forall n m : \text{nat}, (n \leq m \wedge m \leq n) \rightarrow n = m$ .

Proof. intros. omega. Qed.

Example auto\_example\_6 :  $\forall n m p : \text{nat},$

$(n \leq p \rightarrow (n \leq m \wedge m \leq n)) \rightarrow$

$n \leq p \rightarrow$

$n = m$ .

Proof.

intros.

auto. auto using le\_antisym.

Qed.

Of course, in any given development there will probably be some specific constructors and lemmas that are used very often in proofs. We can add these to the global hint database by writing `Hint Resolve T.` at the top level, where  $T$  is a top-level theorem or a constructor of an inductively defined proposition (i.e., anything whose type is an implication). As a shorthand, we can write `Hint Constructors c.` to tell Coq to do a `Hint Resolve` for *all* of the constructors from the inductive definition of  $c$ .

It is also sometimes necessary to add `Hint Unfold d.` where  $d$  is a defined symbol, so that `auto` knows to expand uses of  $d$  and enable further possibilities for applying lemmas that it knows about.

Hint Resolve le\_antisym.

Example auto\_example\_6' :  $\forall n m p : \text{nat},$

$(n \leq p \rightarrow (n \leq m \wedge m \leq n)) \rightarrow$

$n \leq p \rightarrow$

$n = m$ .

Proof.

intros.

auto. Qed.

Definition is\_fortytwo  $x := x = 42$ .

Example auto\_example\_7 :  $\forall x, (x \leq 42 \wedge 42 \leq x) \rightarrow \text{is\_fortytwo } x$ .

Proof.

auto. Abort.

Hint Unfold is\_fortytwo.

Example auto\_example\_7' :  $\forall x, (x \leq 42 \wedge 42 \leq x) \rightarrow \text{is\_fortytwo } x$ .

Proof.

*info\_auto.*

Qed.

Hint Constructors **ceval**.

Hint Transparent state.

Hint Transparent total\_map.

Definition stl2 := t\_update (t\_update empty\_state X 1) Y 2.

Definition st21 := t\_update (t\_update empty\_state X 2) Y 1.

Example auto\_example\_8 :  $\exists s'$ ,  
 (IFB (BLe (Ald X) (Ald Y))  
 THEN (Z ::= AMinus (Ald Y) (Ald X))  
 ELSE (Y ::= APlus (Ald X) (Ald Z))  
 FI) / st21  $\setminus\setminus s'$ .

Proof. eauto. Qed.

Example auto\_example\_8' :  $\exists s'$ ,  
 (IFB (BLe (Ald X) (Ald Y))  
 THEN (Z ::= AMinus (Ald Y) (Ald X))  
 ELSE (Y ::= APlus (Ald X) (Ald Z))  
 FI) / stl2  $\setminus\setminus s'$ .

Proof. *info\_eauto*. Qed.

Now let's take a pass over **ceval\_deterministic** using **auto** to simplify the proof script. We see that all simple sequences of hypothesis applications and all uses of **reflexivity** can be replaced by **auto**, which we add to the default tactic to be applied to each case.

Theorem **ceval\_deterministic'**:  $\forall c \ st \ st1 \ st2$ ,  
  $c / st \setminus\setminus st1 \rightarrow$   
  $c / st \setminus\setminus st2 \rightarrow$   
  $st1 = st2$ .

Proof.

```
intros c st st1 st2 E1 E2;  
generalize dependent st2;  
induction E1;  
  intros st2 E2; inv E2; auto.  
-  
  assert (st' = st'0) as EQ1 by auto.  
  subst st'0.  
  auto.  
-  
  +  
    rewrite H in H5. inversion H5.  
-  
  +  
    rewrite H in H5. inversion H5.
```

```

-
+
  rewrite H in H2. inversion H2.
-
  rewrite H in H4. inversion H4.
-
  assert (st' = st'0) as EQ1 by auto.
  subst st'0.
  auto.
Qed.

```

When we are using a particular tactic many times in a proof, we can use a variant of the `Proof` command to make that tactic into a default within the proof. Saying `Proof with t` (where  $t$  is an arbitrary tactic) allows us to use  $t1...$  as a shorthand for  $t1;t$  within the proof. As an illustration, here is an alternate version of the previous proof, using `Proof with auto`.

```

Theorem ceval_deterministic'_alt:  $\forall c\ st\ st1\ st2,$ 
   $c / st \setminus \setminus st1 \rightarrow$ 
   $c / st \setminus \setminus st2 \rightarrow$ 
   $st1 = st2.$ 

```

Proof with auto.

```

  intros c st st1 st2 E1 E2;
  generalize dependent st2;
  induction E1;
    intros st2 E2; inv E2...
-
  assert (st' = st'0) as EQ1...
  subst st'0...
-
+
  rewrite H in H5. inversion H5.
-
+
  rewrite H in H5. inversion H5.
-
+
  rewrite H in H2. inversion H2.
-
  rewrite H in H4. inversion H4.
-
  assert (st' = st'0) as EQ1...
  subst st'0...
Qed.

```

## 23.3 Searching Hypotheses

The proof has become simpler, but there is still an annoying amount of repetition. Let's start by tackling the contradiction cases. Each of them occurs in a situation where we have both

*H1*: `beval st b = false`

and

*H2*: `beval st b = true`

as hypotheses. The contradiction is evident, but demonstrating it is a little complicated: we have to locate the two hypotheses *H1* and *H2* and do a `rewrite` following by an `inversion`. We'd like to automate this process.

(Note: In fact, Coq has a built-in tactic `congruence` that will do the job. But we'll ignore the existence of this tactic for now, in order to demonstrate how to build forward search tactics by hand.)

As a first step, we can abstract out the piece of script in question by writing a small amount of parameterized Ltac.

```
Ltac rwinv H1 H2 := rewrite H1 in H2; inv H2.
```

```
Theorem ceval_deterministic': ∀ c st st1 st2,
```

```
  c / st \ \ st1 →
```

```
  c / st \ \ st2 →
```

```
  st1 = st2.
```

Proof.

```
  intros c st st1 st2 E1 E2;
```

```
  generalize dependent st2;
```

```
  induction E1; intros st2 E2; inv E2; auto.
```

```
-
```

```
  assert (st' = st'0) as EQ1 by auto.
```

```
  subst st'0.
```

```
  auto.
```

```
-
```

```
  +
```

```
    rwinv H H5.
```

```
-
```

```
  +
```

```
    rwinv H H5.
```

```
-
```

```
  +
```

```
    rwinv H H2.
```

```
-
```

```
  rwinv H H4.
```

```
-
```

```
  assert (st' = st'0) as EQ1 by auto.
```

```

subst st'0.
auto. Qed.

```

But this is not much better. We really want Coq to discover the relevant hypotheses for us. We can do this by using the `match goal with ... end` facility of Ltac.

```

Ltac find_rwinv :=
  match goal with
    H1: ?E = true, H2: ?E = false ⊢ _ ⇒ rwinv H1 H2
  end.

```

In words, this `match goal` looks for two distinct hypotheses that have the form of equalities with the same arbitrary expression  $E$  on the left and conflicting boolean values on the right; if such hypotheses are found, it binds  $H1$  and  $H2$  to their names, and applies the `rwinv` tactic.

Adding this tactic to our default string handles all the contradiction cases.

```

Theorem ceval_deterministic''': ∀ c st st1 st2,
  c / st \\\ st1 →
  c / st \\\ st2 →
  st1 = st2.

```

Proof.

```

intros c st st1 st2 E1 E2;
generalize dependent st2;
induction E1; intros st2 E2; inv E2; try find_rwinv; auto.
-
  assert (st' = st'0) as EQ1 by auto.
  subst st'0.
  auto.
-
  +
  assert (st' = st'0) as EQ1 by auto.
  subst st'0.
  auto. Qed.

```

Let's see about the remaining cases. Each of them involves applying a conditional hypothesis to extract an equality. Currently we have phrased these as assertions, so that we have to predict what the resulting equality will be (although we can then use `auto` to prove it.) An alternative is to pick the relevant hypotheses to use, and then rewrite with them, as follows:

```

Theorem ceval_deterministic''''': ∀ c st st1 st2,
  c / st \\\ st1 →
  c / st \\\ st2 →
  st1 = st2.

```

Proof.

```

intros c st st1 st2 E1 E2;

```

```

generalize dependent st2;
induction E1; intros st2 E2; inv E2; try find_rwinv; auto.
-
  rewrite (IHE1_1 st'0 H1) in *. auto.
-
+
  rewrite (IHE1_1 st'0 H3) in *. auto. Qed.

```

Now we can automate the task of finding the relevant hypotheses to rewrite with.

```

Ltac find_eqn :=
  match goal with
  | H1:  $\forall x, ?P\ x \rightarrow ?L = ?R$ , H2:  $?P\ ?X \vdash \_ \Rightarrow$ 
    rewrite (H1 X H2) in *
  end.

```

But there are several pairs of hypotheses that have the right general form, and it seems tricky to pick out the ones we actually need. A key trick is to realize that we can *try them all*! Here's how this works:

- `rewrite` will fail given a trivial equation of the form  $X = X$ ;
- each execution of `match goal` will keep trying to find a valid pair of hypotheses until the tactic on the RHS of the match succeeds; if there are no such pairs, it fails;
- we can wrap the whole thing in a `repeat`, which will keep doing useful rewrites until only trivial ones are left.

```

Theorem ceval_deterministic''':  $\forall c\ st\ st1\ st2,$ 
   $c / st \setminus \setminus st1 \rightarrow$ 
   $c / st \setminus \setminus st2 \rightarrow$ 
   $st1 = st2.$ 

```

Proof.

```

intros c st st1 st2 E1 E2;
generalize dependent st2;
induction E1; intros st2 E2; inv E2; try find_rwinv;
  repeat find_eqn; auto.
Qed.

```

The big payoff in this approach is that our proof script should be robust in the face of modest changes to our language. For example, we can add a *REPEAT* command to the language. (This was an exercise in *Hoare.v*.)

Module REPEAT.

```

Inductive com : Type :=
| CSkip : com
| CAsgn : id  $\rightarrow$  aexp  $\rightarrow$  com

```

| CSeq : **com** → **com** → **com**  
 | Clf : **bexp** → **com** → **com** → **com**  
 | CWhile : **bexp** → **com** → **com**  
 | CRepeat : **com** → **bexp** → **com**.

*REPEAT* behaves like *WHILE*, except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*. Because of this, the body will always execute at least once.

Notation "'SKIP'" :=

CSkip.

Notation "c1 ; c2" :=

(CSeq c1 c2) (at level 80, right associativity).

Notation "X ':= ' a" :=

(CAsgn X a) (at level 60).

Notation "'WHILE' b 'DO' c 'END'" :=

(CWhile b c) (at level 80, right associativity).

Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=

(Clf e1 e2 e3) (at level 80, right associativity).

Notation "'REPEAT' e1 'UNTIL' b2 'END'" :=

(CRepeat e1 b2) (at level 80, right associativity).

Inductive **ceval** : state → **com** → state → Prop :=

| E\_Skip : ∀ st,  
     **ceval** st SKIP st  
 | E\_Ass : ∀ st a1 n X,  
     aeval st a1 = n →  
     **ceval** st (X ::= a1) (t\_update st X n)  
 | E\_Seq : ∀ c1 c2 st st' st'',  
     **ceval** st c1 st' →  
     **ceval** st' c2 st'' →  
     **ceval** st (c1 ; c2) st''  
 | E\_IfTrue : ∀ st st' b1 c1 c2,  
     beval st b1 = true →  
     **ceval** st c1 st' →  
     **ceval** st (IFB b1 THEN c1 ELSE c2 FI) st'  
 | E\_IfFalse : ∀ st st' b1 c1 c2,  
     beval st b1 = false →  
     **ceval** st c2 st' →  
     **ceval** st (IFB b1 THEN c1 ELSE c2 FI) st'  
 | E\_WhileEnd : ∀ b1 st c1,  
     beval st b1 = false →  
     **ceval** st (WHILE b1 DO c1 END) st  
 | E\_WhileLoop : ∀ st st' st'' b1 c1,  
     beval st b1 = true →



```

    ceval st c1 st' →
    ceval st' (WHILE b1 DO c1 END) st'' →
    ceval st (WHILE b1 DO c1 END) st''
| E_RepeatEnd : ∀ st st' b1 c1,
    ceval st c1 st' →
    beval st' b1 = true →
    ceval st (CRepeat c1 b1) st'
| E_RepeatLoop : ∀ st st' st'' b1 c1,
    ceval st c1 st' →
    beval st' b1 = false →
    ceval st' (CRepeat c1 b1) st'' →
    ceval st (CRepeat c1 b1) st''
.

```

Notation " $c1 \text{ '}' st \text{ '}' st''$ " := (ceval st c1 st')  
(at level 40, st at level 39).

Theorem ceval\_deterministic:  $\forall c \ st \ st1 \ st2$ ,  
 $c / st \ \backslash \ st1 \rightarrow$   
 $c / st \ \backslash \ st2 \rightarrow$   
 $st1 = st2$ .

Proof.

```

intros c st st1 st2 E1 E2;
generalize dependent st2;
induction E1;
    intros st2 E2; inv E2; try find_rwinv; repeat find_eqn; auto.
-
+
  find_rwinv.
-
+
  find_rwinv.

```

Qed.

Theorem ceval\_deterministic':  $\forall c \ st \ st1 \ st2$ ,  
 $c / st \ \backslash \ st1 \rightarrow$   
 $c / st \ \backslash \ st2 \rightarrow$   
 $st1 = st2$ .

Proof.

```

intros c st st1 st2 E1 E2;
generalize dependent st2;
induction E1;
    intros st2 E2; inv E2; repeat find_eqn; try find_rwinv; auto.

```

Qed.

End REPEAT.

These examples just give a flavor of what "hyper-automation" can do. The details of using `match goal` are a bit tricky, and debugging scripts using it is not very pleasant. But it is well worth adding at least simple uses to your proofs to avoid tedium and to "future proof" your scripts.

*Date* : 2016 – 03 – 07 12 : 57 : 38 – 0500 (*Mon, 07 Mar 2016*)

# Chapter 24

## Library Types

### 24.1 Types: Type Systems

```
Require Import Coq.Arith.Arith.
```

```
Require Import Sflib.
```

```
Require Import Maps.
```

```
Require Import Imp.
```

```
Require Import Smallstep.
```

```
Hint Constructors multi.
```

Our next major topic is *type systems* – static program analyses that classify expressions according to the “shapes” of their results. We’ll begin with a typed version of a very simple language with just booleans and numbers, to introduce the basic ideas of types, typing rules, and the fundamental theorems about type systems: *type preservation* and *progress*. In the next chapter we’ll move on to the *simply typed lambda-calculus*, which lives at the core of every modern functional programming language (including Coq).

### 24.2 Typed Arithmetic Expressions

To motivate the discussion of type systems, let’s begin as usual with an extremely simple toy language. We want it to have the potential for programs going wrong because of runtime type errors, so we need something a tiny bit more complex than the language of constants and addition that we used in chapter `Smallstep`: a single kind of data (e.g., numbers) is too simple, but just two kinds (numbers and booleans) gives us enough material to tell an interesting story.

The language definition is completely routine.

#### 24.2.1 Syntax

Informally:  $t ::= \text{true} \mid \text{false} \mid \text{if } t \text{ then } t \text{ else } t \mid 0 \mid \text{succ } t \mid \text{pred } t \mid \text{iszero } t$  Formally:

Inductive **tm** : Type :=

| ttrue : **tm**  
| tfalse : **tm**  
| tif : **tm** → **tm** → **tm** → **tm**  
| tzero : **tm**  
| tsucc : **tm** → **tm**  
| tpred : **tm** → **tm**  
| tiszero : **tm** → **tm**.

*Values* are true, false, and numeric values...

Inductive **bvalue** : **tm** → Prop :=

| bv\_true : **bvalue** ttrue  
| bv\_false : **bvalue** tfalse.

Inductive **nvalue** : **tm** → Prop :=

| nv\_zero : **nvalue** tzero  
| nv\_succ : ∀ t, **nvalue** t → **nvalue** (tsucc t).

Definition value (t:**tm**) := **bvalue** t ∨ **nvalue** t.

Hint Constructors **bvalue nvalue**.

Hint Unfold value.

Hint Unfold update.

## 24.2.2 Operational Semantics

Informally:

---

(ST\_IfTrue) if true then t1 else t2 ==> t1

---

(ST\_IfFalse) if false then t1 else t2 ==> t2  
t1 ==> t1'

---

(ST\_If) if t1 then t2 else t3 ==> if t1' then t2 else t3  
t1 ==> t1'

---

(ST\_Succ) succ t1 ==> succ t1'

---

(ST\_PredZero) pred 0 ==> 0  
numeric value v1

---

(ST\_PredSucc) pred (succ v1) ==> v1  
t1 ==> t1'

---

(ST\_Pred) pred t1 ==> pred t1'

---

(ST\_IszeroZero) iszero 0 ==> true  
numeric value v1

---

(ST\_IszeroSucc) iszero (succ v1) ==> false  
t1 ==> t1'

---

(ST\_Iszero) iszero t1 ==> iszero t1'  
Formally:

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

| ST\_IfTrue : ∀ t1 t2,  
    (tif ttrue t1 t2) ==> t1  
| ST\_IfFalse : ∀ t1 t2,  
    (tif tfalse t1 t2) ==> t2  
| ST\_If : ∀ t1 t1' t2 t3,  
    t1 ==> t1' →  
    (tif t1 t2 t3) ==> (tif t1' t2 t3)  
| ST\_Succ : ∀ t1 t1',  
    t1 ==> t1' →  
    (tsucc t1) ==> (tsucc t1')  
| ST\_PredZero :  
    (tpred tzero) ==> tzero  
| ST\_PredSucc : ∀ t1,  
    nvalue t1 →  
    (tpred (tsucc t1)) ==> t1  
| ST\_Pred : ∀ t1 t1',  
    t1 ==> t1' →  
    (tpred t1) ==> (tpred t1')  
| ST\_IszeroZero :  
    (tiszero tzero) ==> ttrue  
| ST\_IszeroSucc : ∀ t1,  
    nvalue t1 →  
    (tiszero (tsucc t1)) ==> tfalse  
| ST\_Iszero : ∀ t1 t1',  
    t1 ==> t1' →  
    (tiszero t1) ==> (tiszero t1')

where "t1 '==>' t2" := (**step** t1 t2).

Hint Constructors **step**.

Notice that the **step** relation doesn't care about whether expressions make global sense

– it just checks that the operation in the *next* reduction step is being applied to the right kinds of operands.

For example, the term `succ true` (i.e., `tsucc ttrue` in the formal syntax) cannot take a step, but the almost-as-obviously nonsensical term `succ (if true then true else true)` can take a step (once, before becoming stuck).

### 24.2.3 Normal Forms and Values

The first interesting thing to notice about the **step** relation in this language is that the strong progress theorem from the Smallstep chapter fails. That is, there are terms that are normal forms (they can't take a step) but not values (because we have not included them in our definition of possible "results of evaluation"). Such terms are *stuck*.

Notation `step_normal_form` := (normal\_form **step**).

Definition `stuck (t:tm) : Prop` :=  
`step_normal_form t ∧ ¬ value t.`

Hint `Unfold stuck`.

**Exercise: 2 stars (some\_term\_is\_stuck)** Example `some_term_is_stuck` :

`∃ t, stuck t.`

Proof.

*Admitted.*

□

However, although values and normal forms are not the same in this language, the set of values is included in the set of normal forms. This is important because it shows we did not accidentally define things so that some value could still take a step.

**Exercise: 3 stars, advanced (value\_is\_nf)** (Hint: You will reach a point in this proof where you need to use an induction to reason about a term that is known to be a numeric value. This induction can be performed either over the term itself or over the evidence that it is a numeric value. The proof goes through in either case, but you will find that one way is quite a bit shorter than the other. For the sake of the exercise, try to complete the proof both ways.)

Lemma `value_is_nf` :  $\forall t,$   
`value t → step_normal_form t.`

Proof.

*Admitted.*

□

**Exercise: 3 stars, optional (step\_deterministic)** Use `value_is_nf` to show that the **step** relation is also deterministic.

Theorem `step_deterministic`:

deterministic **step**.  
 Proof with `eauto`.  
*Admitted.*  
 $\square$

## 24.2.4 Typing

The next critical observation is that, although this language has stuck terms, they are all "nonsensical," mixing booleans and numbers in a way that we don't even *want* to have a meaning. We can easily exclude such ill-typed terms by defining a *typing relation* that relates terms to the types (either numeric or boolean) of their final results.

Inductive **ty** : Type :=  
 | TBool : **ty**  
 | TNat : **ty**.

In informal notation, the typing relation is often written  $\vdash t \text{ \textit{in} } T$  and pronounced "*t* has type *T*." The  $\vdash$  symbol is called a "turnstile". Below, we're going to see richer typing relations where an additional "context" argument is written to the left of the turnstile. Here, the context is always empty.

---

(T_True)	$\vdash \text{true} \text{ \textit{in} } \text{Bool}$
----------	---

---

(T_False)	$\vdash \text{false} \text{ \textit{in} } \text{Bool}$
	$\vdash t1 \text{ \textit{in} } \text{Bool} \vdash t2 \text{ \textit{in} } T \vdash t3 \text{ \textit{in} } T$

---

(T_If)	$\vdash \text{if } t1 \text{ then } t2 \text{ else } t3 \text{ \textit{in} } T$
--------	---

---

(T_Zero)	$\vdash 0 \text{ \textit{in} } \text{Nat}$
	$\vdash t1 \text{ \textit{in} } \text{Nat}$

---

(T_Succ)	$\vdash \text{succ } t1 \text{ \textit{in} } \text{Nat}$
	$\vdash t1 \text{ \textit{in} } \text{Nat}$

---

(T_Pred)	$\vdash \text{pred } t1 \text{ \textit{in} } \text{Nat}$
	$\vdash t1 \text{ \textit{in} } \text{Nat}$

---

(T_IsZero)	$\vdash \text{iszero } t1 \text{ \textit{in} } \text{Bool}$
------------	---

Reserved Notation " $\vdash t \text{ \textit{in} } T$ " (at level 40).

Inductive **has\_type** : **tm**  $\rightarrow$  **ty**  $\rightarrow$  Prop :=  
 | T\_True :  
    $\vdash \text{ttrue} \text{ \textit{in} } \text{TBool}$   
 | T\_False :

```

      ⊢ tfalse \in TBool
| T_If : ∀ t1 t2 t3 T,
      ⊢ t1 \in TBool →
      ⊢ t2 \in T →
      ⊢ t3 \in T →
      ⊢ tif t1 t2 t3 \in T
| T_Zero :
      ⊢ tzero \in TNat
| T_Succ : ∀ t1,
      ⊢ t1 \in TNat →
      ⊢ tsucc t1 \in TNat
| T_Pred : ∀ t1,
      ⊢ t1 \in TNat →
      ⊢ tpred t1 \in TNat
| T_Iszero : ∀ t1,
      ⊢ t1 \in TNat →
      ⊢ tiszero t1 \in TBool

```

where "'|-' t '\in' T" := (**has\_type** t T).

Hint Constructors **has\_type**.

## Examples

It's important to realize that the typing relation is a *conservative* (or *static*) approximation: it does not calculate the type of the normal form of a term.

Example has\_type\_1 :

```
⊢ tif tfalse tzero (tsucc tzero) \in TNat.
```

Proof.

```

apply T_If.
- apply T_False.
- apply T_Zero.
- apply T_Succ.
  + apply T_Zero.

```

Qed.

(Since we've included all the constructors of the typing relation in the hint database, the `auto` tactic can actually find this proof automatically.)

Example has\_type\_not :

```
¬ (⊢ tif tfalse tzero ttrue \in TBool).
```

Proof.

```
intros Contra. solve by inversion 2. Qed.
```



**Exercise: 1 star, optional (succ\_hastype\_nat\_\_hastype\_nat)** Example succ\_hastype\_nat\_\_hastype.

```
: ∀ t,
  ⊢ tsucc t \in TNat →
  ⊢ t \in TNat.
```

Proof.

*Admitted.*

□

## Canonical forms

The following two lemmas capture the basic property that defines the shape of well-typed values. They say that the definition of value and the typing relation agree.

```
Lemma bool_canonical : ∀ t,
  ⊢ t \in TBool → value t → bvalue t.
```

Proof.

```
  intros t HT HV.
  inversion HV; auto.

  induction H; inversion HT; auto.
```

Qed.

```
Lemma nat_canonical : ∀ t,
  ⊢ t \in TNat → value t → nvalue t.
```

Proof.

```
  intros t HT HV.
  inversion HV.
  inversion H; subst; inversion HT.
```

```
  auto.
```

Qed.

## 24.2.5 Progress

The typing relation enjoys two critical properties. The first is that well-typed normal forms are values (i.e., not stuck).

```
Theorem progress : ∀ t T,
  ⊢ t \in T →
  value t ∨ ∃ t', t ==> t'.
```

**Exercise: 3 stars (finish\_progress)** Complete the formal proof of the **progress** property. (Make sure you understand the informal proof fragment in the following exercise before starting – this will save you a lot of time.)

Proof with auto.

```
  intros t T HT.
```

```

induction HT...
-
  right. inversion IHHT1; clear IHHT1.
  +
  apply (bool_canonical t1 HT1) in H.
  inversion H; subst; clear H.
  ∃ t2...
  ∃ t3...
  +
  inversion H as [t1' H1].
  ∃ (tif t1' t2 t3)...
Admitted.
□

```

**Exercise: 3 stars, advanced (finish\_progress\_informal)** Complete the corresponding informal proof:

*Theorem:* If  $\vdash t \text{ \texttt{in} } T$ , then either  $t$  is a value or else  $t ==> t'$  for some  $t'$ .

*Proof:* By induction on a derivation of  $\vdash t \text{ \texttt{in} } T$ .

- If the last rule in the derivation is  $\text{\texttt{T\_If}}$ , then  $t = \text{\texttt{if } } t1 \text{ \texttt{ then } } t2 \text{ \texttt{ else } } t3$ , with  $\vdash t1 \text{ \texttt{ in } } \text{\texttt{Bool}}$ ,  $\vdash t2 \text{ \texttt{ in } } T$  and  $\vdash t3 \text{ \texttt{ in } } T$ . By the IH, either  $t1$  is a value or else  $t1$  can step to some  $t1'$ .
  - If  $t1$  is a value, then by the canonical forms lemmas and the fact that  $\vdash t1 \text{ \texttt{ in } } \text{\texttt{Bool}}$  we have that  $t1$  is a **bvalue** – i.e., it is either **true** or **false**. If  $t1 = \text{\texttt{true}}$ , then  $t$  steps to  $t2$  by  $\text{\texttt{ST\_IfTrue}}$ , while if  $t1 = \text{\texttt{false}}$ , then  $t$  steps to  $t3$  by  $\text{\texttt{ST\_IfFalse}}$ . Either way,  $t$  can step, which is what we wanted to show.
  - If  $t1$  itself can take a step, then, by  $\text{\texttt{ST\_If}}$ , so can  $t$ .

□

This is more interesting than the strong progress theorem that we saw in the **Smallstep** chapter, where *all* normal forms were values. Here, a term can be stuck, but only if it is ill typed.

**Exercise: 1 star (step\_review)** Quick review. Answer *true* or *false*. In this language...

- Every well-typed normal form is a value.
- Every value is a normal form.
- The single-step evaluation relation is a partial function (i.e., it is deterministic).
- The single-step evaluation relation is a *total* function.

□

## 24.2.6 Type Preservation

The second critical property of typing is that, when a well-typed term takes a step, the result is also a well-typed term.

This theorem is often called the *subject reduction* property, because it tells us what happens when the "subject" of the typing relation is reduced. This terminology comes from thinking of typing statements as sentences, where the term is the subject and the type is the predicate.

**Theorem preservation :**  $\forall t\ t'\ T,$   
 $\vdash t\ \text{in}\ T \rightarrow$   
 $t ==> t' \rightarrow$   
 $\vdash t'\ \text{in}\ T.$

**Exercise: 2 stars (finish\_preservation)** Complete the formal proof of the preservation property. (Again, make sure you understand the informal proof fragment in the following exercise first.)

Proof with auto.

```
intros t t' T HT HE.
generalize dependent t'.
induction HT;

    intros t' HE;

    try (solve by inversion).
- inversion HE; subst; clear HE.
  + assumption.
  + assumption.
  + apply T_If; try assumption.
    apply IHHT1; assumption.
Admitted.
□
```

**Exercise: 3 stars, advanced (finish\_preservation\_informal)** Complete the following proof:

*Theorem:* If  $\vdash t\ \text{in}\ T$  and  $t ==> t'$ , then  $\vdash t'\ \text{in}\ T$ .

*Proof:* By induction on a derivation of  $\vdash t\ \text{in}\ T$ .

- If the last rule in the derivation is `T_If`, then  $t = \text{if } t1 \text{ then } t2 \text{ else } t3$ , with  $\vdash t1\ \text{in}\ \text{Bool}$ ,  $\vdash t2\ \text{in}\ T$  and  $\vdash t3\ \text{in}\ T$ .

Inspecting the rules for the small-step reduction relation and remembering that  $t$  has the form `if ...`, we see that the only ones that could have been used to prove  $t ==> t'$  are `ST_IfTrue`, `ST_IfFalse`, or `ST_If`.

- If the last rule was `ST_IfTrue`, then  $t' = t2$ . But we know that  $\vdash t2 \text{ \texttt{in} } T$ , so we are done.
- If the last rule was `ST_IfFalse`, then  $t' = t3$ . But we know that  $\vdash t3 \text{ \texttt{in} } T$ , so we are done.
- If the last rule was `ST_If`, then  $t' = \text{if } t1' \text{ then } t2 \text{ else } t3$ , where  $t1 ==> t1'$ . We know  $\vdash t1 \text{ \texttt{in} } \text{Bool}$  so, by the IH,  $\vdash t1' \text{ \texttt{in} } \text{Bool}$ . The `T_If` rule then gives us  $\vdash \text{if } t1' \text{ then } t2 \text{ else } t3 \text{ \texttt{in} } T$ , as required.

•

□

**Exercise: 3 stars (preservation\_alterate\_proof)** Now prove the same property again by induction on the *evaluation* derivation instead of on the typing derivation. Begin by carefully reading and thinking about the first few lines of the above proof to make sure you understand what each one is doing. The set-up for this proof is similar, but not exactly the same.

Theorem `preservation'` :  $\forall t t' T,$   
 $\vdash t \text{ \texttt{in} } T \rightarrow$   
 $t ==> t' \rightarrow$   
 $\vdash t' \text{ \texttt{in} } T.$

Proof with `eauto`.

*Admitted.*

□

## 24.2.7 Type Soundness

Putting progress and preservation together, we can see that a well-typed term can *never* reach a stuck state.

Definition `multistep` := (**multi step**).

Notation " $t1' ==>^* t2$ " := (`multistep t1 t2`) (at level 40).

Corollary `soundness` :  $\forall t t' T,$   
 $\vdash t \text{ \texttt{in} } T \rightarrow$   
 $t ==>^* t' \rightarrow$   
 $\sim(\text{stuck } t').$

Proof.

```
intros t t' T HT P. induction P; intros [R S].
destruct (progress x T HT); auto.
apply IHP. apply (preservation x y T HT H).
unfold stuck. split; auto. Qed.
```

## 24.3 Aside: the *normalize* Tactic

When experimenting with definitions of programming languages in Coq, we often want to see what a particular concrete term steps to – i.e., we want to find proofs for goals of the form  $t \Rightarrow^* t'$ , where  $t$  is a completely concrete term and  $t'$  is unknown. These proofs are simple but repetitive to do by hand. Consider for example reducing an arithmetic expression using the small-step relation **astep**.

```
Notation " t '/' st '==>a*' t' " := (multi (astep st) t t')
                                   (at level 40, st at level 39).
```

Example astep\_example1 :

```
(APlus (ANum 3) (AMult (ANum 3) (ANum 4))) / empty_state
==>a× (ANum 15).
```

Proof.

```
apply multi_step with (APlus (ANum 3) (ANum 12)).
  apply AS_Plus2.
    apply av_num.
    apply AS_Mult.
  apply multi_step with (ANum 15).
    apply AS_Plus.
  apply multi_refl.
```

Qed.

We repeatedly apply `multi_step` until we get to a normal form. The proofs for the intermediate steps are simple enough that `auto`, with appropriate hints, can solve them.

Hint Constructors **astep aval**.

Example astep\_example1' :

```
(APlus (ANum 3) (AMult (ANum 3) (ANum 4))) / empty_state
==>a× (ANum 15).
```

Proof.

```
eapply multi_step. auto. simpl.
eapply multi_step. auto. simpl.
apply multi_refl.
```

Qed.

The following custom Tactic Notation definition captures this pattern. In addition, before each step, we print out the current goal, so that we can follow how the term is being evaluated.

```
Tactic Notation "print_goal" := match goal with ⊢ ?x ⇒ idtac x end.
```

```
Tactic Notation "normalize" :=
```

```
  repeat (print_goal; eapply multi_step ;
    [ (eauto 10; fail) | (instantiate; simpl)]) ;
  apply multi_refl.
```

Example astep\_example1'' :

(APlus (ANum 3) (AMult (ANum 3) (ANum 4))) / empty\_state  
 $\Rightarrow_{a \times}$  (ANum 15).

Proof.

*normalize.*

Qed.

The *normalize* tactic also provides a simple way to calculate what the normal form of a term is, by proving a goal with an existential variable in it.

Example `astep_example1'''` :  $\exists e'$ ,

(APlus (ANum 3) (AMult (ANum 3) (ANum 4))) / empty\_state  
 $\Rightarrow_{a \times} e'$ .

Proof.

`eapply ex_intro.` *normalize.*

Qed.

**Exercise: 1 star (normalize\_ex)** Theorem `normalize_ex` :  $\exists e'$ ,

(AMult (ANum 3) (AMult (ANum 2) (ANum 1))) / empty\_state  
 $\Rightarrow_{a \times} e'$ .

Proof.

*Admitted.*

□

**Exercise: 1 star, optional (normalize\_ex')** For comparison, prove it using `apply` instead of `eapply`.

Theorem `normalize_ex'` :  $\exists e'$ ,

(AMult (ANum 3) (AMult (ANum 2) (ANum 1))) / empty\_state  
 $\Rightarrow_{a \times} e'$ .

Proof.

*Admitted.*

□

### 24.3.1 Additional Exercises

**Exercise: 2 stars, recommended (subject\_expansion)** Having seen the subject reduction property, it is reasonable to wonder whether the opposite property – subject *expansion* – also holds. That is, is it always the case that, if  $t \Rightarrow t'$  and  $\vdash t' \text{ in } T$ , then  $\vdash t \text{ in } T$ ? If so, prove it. If not, give a counter-example. (You do not need to prove your counter-example in Coq, but feel free to do so if you like.)

□

**Exercise: 2 stars (variation1)** Suppose, that we add this new rule to the typing relation:  
 $| T\_SuccBool : \text{forall } t, |- t \text{ \textit{in} } TBool \rightarrow |- tsucc\ t \text{ \textit{in} } TBool$  Which of the following properties remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

**Exercise: 2 stars (variation2)** Suppose, instead, that we add this new rule to the **step** relation:  
 $| ST\_Funny1 : \text{forall } t2\ t3, (t\text{if } ttrue\ t2\ t3) \Rightarrow t3$  Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

**Exercise: 2 stars, optional (variation3)** Suppose instead that we add this rule:  
 $| ST\_Funny2 : \text{forall } t1\ t2\ t2'\ t3, t2 \Rightarrow t2' \rightarrow (t\text{if } t1\ t2\ t3) \Rightarrow (t\text{if } t1\ t2'\ t3)$  Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

**Exercise: 2 stars, optional (variation4)** Suppose instead that we add this rule:  
 $| ST\_Funny3 : (tpred\ tfalse) \Rightarrow (tpred\ (tpred\ tfalse))$  Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

**Exercise: 2 stars, optional (variation5)** Suppose instead that we add this rule:  
 $| T\_Funny4 : |- tzero \text{ \textit{in} } TBool$  Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

**Exercise: 2 stars, optional (variation6)** Suppose instead that we add this rule:  
 $| T\_Funny5 : |- tpred\ tzero \text{ \textit{in} } TBool$  Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

**Exercise: 3 stars, optional (more\_variations)** Make up some exercises of your own along the same lines as the ones above. Try to find ways of selectively breaking properties – i.e., ways of changing the definitions that break just one of the properties and leave the others alone. □

**Exercise: 1 star (remove\_predzero)** The evaluation rule  $E\_PredZero$  is a bit counter-intuitive: we might feel that it makes more sense for the predecessor of zero to be undefined, rather than being defined to be zero. Can we achieve this simply by removing the rule from the definition of **step**? Would doing so create any problems elsewhere?

□

**Exercise: 4 stars, advanced (prog\_pres\_bigstep)** Suppose our evaluation relation is defined in the big-step style. What are the appropriate analogs of the progress and preservation properties? (You do not need to prove them.)

□

*Date : 2016 – 03 – 07 17 : 05 : 52 – 0500 (Mon, 07 Mar 2016)*



# Chapter 25

## Library Stlc

### 25.1 Stlc: The Simply Typed Lambda-Calculus

```
Require Import Maps.  
Require Import Smallstep.  
Require Import Types.
```

### 25.2 The Simply Typed Lambda-Calculus

The simply typed lambda-calculus (STLC) is a tiny core calculus embodying the key concept of *functional abstraction*, which shows up in pretty much every real-world programming language in some form (functions, procedures, methods, etc.).

We will follow exactly the same pattern as in the previous chapter when formalizing this calculus (syntax, small-step semantics, typing rules) and its main properties (progress and preservation). The new technical challenges all arise from the mechanisms of *variable binding* and *substitution*. It which will take some work to deal with these.

#### 25.2.1 Overview

The STLC is built on some collection of *base types* – booleans, numbers, strings, etc. The exact choice of base types doesn’t matter much – the construction of the language and its theoretical properties work out the same no matter what we choose – so for the sake of brevity let’s take just `Bool` for the moment. At the end of the chapter we’ll see how to add more base types, and in later chapters we’ll enrich the pure STLC with other useful constructs like pairs, records, subtyping, and mutable state.

Starting from boolean constants and conditionals, we add three things:

- variables
- function abstractions

- application

This gives us the following collection of abstract syntax constructors (written out first in informal BNF notation – we’ll formalize it below).

$t ::= x$  variable |  $\backslash x:T1.t2$  abstraction |  $t1\ t2$  application | **true** constant true | **false** constant false | **if**  $t1$  **then**  $t2$  **else**  $t3$  conditional

The  $\backslash$  symbol in a function abstraction  $\backslash x:T1.t2$  is generally written as a greek letter ”lambda“ (hence the name of the calculus). The variable  $x$  is called the *parameter* to the function; the term  $t2$  is its *body*. The annotation  $:T1$  specifies the type of arguments that the function can be applied to.

Some examples:

- $\backslash x:\text{Bool}. x$

The identity function for booleans.

- $(\backslash x:\text{Bool}. x)\ \text{true}$

The identity function for booleans, applied to the boolean **true**.

- $\backslash x:\text{Bool}. \text{if } x \text{ then false else true}$

The boolean ”not“ function.

- $\backslash x:\text{Bool}. \text{true}$

The constant function that takes every (boolean) argument to **true**.

- $\backslash x:\text{Bool}. \backslash y:\text{Bool}. x$

A two-argument function that takes two booleans and returns the first one. (Note that, as in Coq, a two-argument function is really a one-argument function whose body is also a one-argument function.)

- $(\backslash x:\text{Bool}. \backslash y:\text{Bool}. x)\ \text{false}\ \text{true}$

A two-argument function that takes two booleans and returns the first one, applied to the booleans **false** and **true**.

Note that, as in Coq, application associates to the left – i.e., this expression is parsed as  $((\backslash x:\text{Bool}. \backslash y:\text{Bool}. x)\ \text{false})\ \text{true}$ .

- $\backslash f:\text{Bool} \rightarrow \text{Bool}. f\ (f\ \text{true})$

A higher-order function that takes a *function*  $f$  (from booleans to booleans) as an argument, applies  $f$  to **true**, and applies  $f$  again to the result.

- $(\backslash f:\text{Bool} \rightarrow \text{Bool}. f\ (f\ \text{true}))\ (\backslash x:\text{Bool}. \text{false})$

The same higher-order function, applied to the constantly **false** function.

As the last several examples show, the STLC is a language of *higher-order* functions: we can write down functions that take other functions as arguments and/or return other functions as results.

Another point to note is that the STLC doesn't provide any primitive syntax for defining *named* functions – all functions are "anonymous." We'll see in chapter **MoreStlc** that it is easy to add named functions to what we've got – indeed, the fundamental naming and binding mechanisms are exactly the same.

The *types* of the STLC include **Bool**, which classifies the boolean constants **true** and **false** as well as more complex computations that yield booleans, plus *arrow types* that classify functions.

$T ::= \text{Bool} \mid T_1 \rightarrow T_2$  For example:

- $\lambda x:\text{Bool}. \text{false}$  has type  $\text{Bool} \rightarrow \text{Bool}$
- $\lambda x:\text{Bool}. x$  has type  $\text{Bool} \rightarrow \text{Bool}$
- $(\lambda x:\text{Bool}. x) \text{true}$  has type **Bool**
- $\lambda x:\text{Bool}. \lambda y:\text{Bool}. x$  has type  $\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$  (i.e.,  $\text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$ )
- $(\lambda x:\text{Bool}. \lambda y:\text{Bool}. x) \text{false}$  has type  $\text{Bool} \rightarrow \text{Bool}$
- $(\lambda x:\text{Bool}. \lambda y:\text{Bool}. x) \text{false true}$  has type **Bool**

## 25.2.2 Syntax

Module STLC.

### Types

```
Inductive ty : Type :=
| TBool : ty
| TArrow : ty → ty → ty.
```

### Terms

```
Inductive tm : Type :=
| tvar : id → tm
| tapp : tm → tm → tm
| tabs : id → ty → tm → tm
| ttrue : tm
| tfalse : tm
| tif : tm → tm → tm → tm.
```

Note that an abstraction  $\lambda x:T.t$  (formally, `tabs x T t`) is always annotated with the type  $T$  of its parameter, in contrast to Coq (and other functional languages like ML, Haskell, etc.), which use *type inference* to fill in missing annotations. We're not considering type inference here, to keep things simple.

Some examples...

`Definition x := (ld 0).`

`Definition y := (ld 1).`

`Definition z := (ld 2).`

`Hint Unfold x.`

`Hint Unfold y.`

`Hint Unfold z.`

`idB = \x:Bool. x`

`Notation idB :=`

`(tabs x TBool (tvar x)).`

`idBB = \x:Bool→Bool. x`

`Notation idBB :=`

`(tabs x (TArrow TBool TBool) (tvar x)).`

`idBBBB = \x:(Bool→Bool) → (Bool→Bool). x`

`Notation idBBBB :=`

`(tabs x (TArrow (TArrow TBool TBool)  
                  (TArrow TBool TBool))  
  (tvar x)).`

`k = \x:Bool. \y:Bool. x`

`Notation k := (tabs x TBool (tabs y TBool (tvar x))).`

`notB = \x:Bool. if x then false else true`

`Notation notB := (tabs x TBool (tif (tvar x) tfalse ttrue)).`

(We write these as `Notations` rather than `Definitions` to make things easier for `auto`.)

### 25.2.3 Operational Semantics

To define the small-step semantics of STLC terms, we begin – as always – by defining the set of values. Next, we define the critical notions of *free variables* and *substitution*, which are used in the reduction rule for application expressions. And finally we give the small-step relation itself.

#### Values

To define the values of the STLC, we have a few cases to consider.

First, for the boolean part of the language, the situation is clear: **true** and **false** are the only values. An **if** expression is never a value.

Second, an application is clearly not a value: It represents a function being invoked on some argument, which clearly still has work left to do.

Third, for abstractions, we have a choice:

- We can say that  $\lambda x:T.t1$  is a value only when  $t1$  is a value – i.e., only if the function’s body has been reduced (as much as it can be without knowing what argument it is going to be applied to).
- Or we can say that  $\lambda x:T.t1$  is always a value, no matter whether  $t1$  is one or not – in other words, we can say that reduction stops at abstractions.

Coq, in its built-in functional programming language, makes the first choice – for example, `Compute (fun x:bool => 3 + 4)` yields **fun x:bool => 7**.

Most real-world functional programming languages make the second choice – reduction of a function’s body only begins when the function is actually applied to an argument. We also make the second choice here.

```
Inductive value : tm → Prop :=
| v_abs : ∀ x T t,
    value (tabs x T t)
| v_true :
    value ttrue
| v_false :
    value tfalse.
```

Hint Constructors **value**.

Finally, we must consider what constitutes a *complete* program.

Intuitively, a “complete” program must not refer to any undefined variables. We’ll see shortly how to define the “free” variables in a STLC term. A program is “closed”, that is, it contains no free variables.

Having made the choice not to reduce under abstractions, we don’t need to worry about whether variables are values, since we’ll always be reducing programs “from the outside in,” and that means the **step** relation will always be working with closed terms (ones with no free variables).

## Substitution

Now we come to the heart of the STLC: the operation of substituting one term for a variable in another term.

This operation will be used below to define the operational semantics of function application, where we will need to substitute the argument term for the function parameter in the function’s body. For example, we reduce  $(\lambda x:\text{Bool}. \text{if } x \text{ then true else } x) \text{ false}$  to **if false then true else false** by substituting **false** for the parameter  $x$  in the body of the function.

In general, we need to be able to substitute some given term  $s$  for occurrences of some variable  $x$  in another term  $t$ . In informal discussions, this is usually written  $[x:=s]t$  and pronounced "substitute  $x$  with  $s$  in  $t$ ."

Here are some examples:

```
x:=true (if x then x else false) yields if true then true else false
x:=true x yields true
x:=true (if x then x else y) yields if true then true else y
x:=true y yields y
x:=true false yields false (vacuous substitution)
x:=true (\y:Bool. if y then x else false) yields \y:Bool. if y then true else false
x:=true (\y:Bool. x) yields \y:Bool. true
x:=true (\y:Bool. y) yields \y:Bool. y
x:=true (\x:Bool. x) yields \x:Bool. x
```

The last example is very important: substituting  $x$  with  $true$  in  $\backslash x:Bool. x$  does *not* yield  $\backslash x:Bool. true$ ! The reason for this is that the  $x$  in the body of  $\backslash x:Bool. x$  is *bound* by the abstraction: it is a new, local name that just happens to be spelled the same as some global name  $x$ .

Here is the definition, informally...  $x:=sx = s$   $x:=sy = y$  if  $x <> y$   $x:=s(\backslash x:T11.t12) = \backslash x:T11. t12$   $x:=s(\backslash y:T11.t12) = \backslash y:T11. x:=st12$  if  $x <> y$   $x:=s(t1 t2) = (x:=st1) (x:=st2)$   $x:=strue = true$   $x:=sfalse = false$   $x:=s(if t1 then t2 else t3) = if x:=st1 then x:=st2 else x:=st3$

... and formally:

Reserved Notation "'[ x ' := ' s ' ]' t" (at level 20).

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' =>
    if beq_id x x' then s else t
  | tabs x' T t1 =>
    tabs x' T (if beq_id x x' then t1 else ([x:=s] t1))
  | tapp t1 t2 =>
    tapp ([x:=s] t1) ([x:=s] t2)
  | ttrue =>
    ttrue
  | tfalse =>
    tfalse
```

```

| tif t1 t2 t3 =>
  tif ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
end

```

where `"[' x ':= ' s ']' t" := (subst x s t)`.

*Technical note:* Substitution becomes trickier to define if we consider the case where `s`, the term being substituted for a variable in some other term, may itself contain free variables. Since we are only interested here in defining the **step** relation on closed terms (i.e., terms like `\x:Bool. x`, that do not mention variables are not bound by some enclosing lambda), we can skip this extra complexity here, but it must be dealt with when formalizing richer languages.

**Exercise: 3 stars (subst)** The definition that we gave above uses Coq's `Fixpoint` facility to define substitution as a *function*. Suppose, instead, we wanted to define substitution as an inductive *relation* **subst**. We've begun the definition by providing the `Inductive` header and one of the constructors; your job is to fill in the rest of the constructors.

```

Inductive substi (s:tm) (x:id) : tm -> tm -> Prop :=
| s_var1 :
  substi s x (tvar x) s

```

.

Hint Constructors **substi**.

Theorem `substi_correct` :  $\forall s x t t', [x:=s] t = t' \leftrightarrow \text{substi } s x t t'$ .

Proof.

*Admitted.*

□

## Reduction

The small-step reduction relation for STLC now follows the same pattern as the ones we have seen before. Intuitively, to reduce a function application, we first reduce its left-hand side until it becomes a literal function; then we reduce its right-hand side (the argument) until it is also a value; and finally we substitute the argument for the bound variable in the body of the function. This last rule, written informally as  $(\lambda x:T.t12) v2 \implies x:=v2t12$  is traditionally called "beta-reduction".

value `v2`

---

$(\text{ST\_AppAbs}) (\lambda x:T.t12) v2 \implies x:=v2t12$   
 $t1 \implies t1'$

---

$(\text{ST\_App1}) t1 t2 \implies t1' t2$

---

value  $v1\ t2 \implies t2'$

---

(ST\_App2)  $v1\ t2 \implies v1\ t2'$  ... plus the usual rules for booleans:

---

(ST\_IfTrue) (if true then  $t1$  else  $t2$ )  $\implies t1$

---

(ST\_IfFalse) (if false then  $t1$  else  $t2$ )  $\implies t2$   
 $t1 \implies t1'$

---

(ST\_If) (if  $t1$  then  $t2$  else  $t3$ )  $\implies$  (if  $t1'$  then  $t2$  else  $t3$ )

Reserved Notation " $t1 \implies t2$ " (at level 40).

Inductive **step** : **tm**  $\rightarrow$  **tm**  $\rightarrow$  Prop :=

| ST\_AppAbs :  $\forall x\ T\ t12\ v2$ ,  
    **value**  $v2 \rightarrow$   
    (**tapp** (**tabs**  $x\ T\ t12$ )  $v2$ )  $\implies [x:=v2]\ t12$   
| ST\_App1 :  $\forall t1\ t1'\ t2$ ,  
     $t1 \implies t1' \rightarrow$   
    **tapp**  $t1\ t2 \implies$  **tapp**  $t1'\ t2$   
| ST\_App2 :  $\forall v1\ t2\ t2'$ ,  
    **value**  $v1 \rightarrow$   
     $t2 \implies t2' \rightarrow$   
    **tapp**  $v1\ t2 \implies$  **tapp**  $v1\ t2'$   
| ST\_IfTrue :  $\forall t1\ t2$ ,  
    (**tif** **ttrue**  $t1\ t2$ )  $\implies t1$   
| ST\_IfFalse :  $\forall t1\ t2$ ,  
    (**tif** **tfalse**  $t1\ t2$ )  $\implies t2$   
| ST\_If :  $\forall t1\ t1'\ t2\ t3$ ,  
     $t1 \implies t1' \rightarrow$   
    (**tif**  $t1\ t2\ t3$ )  $\implies$  (**tif**  $t1'\ t2\ t3$ )

where " $t1 \implies t2$ " := (**step**  $t1\ t2$ ).

Hint Constructors **step**.

Notation **multistep** := (**multi** **step**).

Notation " $t1 \implies^* t2$ " := (**multistep**  $t1\ t2$ ) (at level 40).

## Examples

Example:  $((\lambda x:\text{Bool}. \rightarrow \text{Bool}. x) (\lambda x:\text{Bool}. x)) \implies^* (\lambda x:\text{Bool}. x)$  i.e., (**idBB** **idB**)  $\implies^*$  **idB**

Lemma **step\_example1** :

(**tapp** **idBB** **idB**)  $\implies^*$  **idB**.



Proof.

```
eapply multi_step.  
  apply ST_AppAbs.  
  apply v_abs.  
simpl.  
apply multi_refl. Qed.
```

Example:  $((\lambda x:\text{Bool} \rightarrow \text{Bool}. x) ((\lambda x:\text{Bool} \rightarrow \text{Bool}. x) (\lambda x:\text{Bool}. x))) \implies^* (\lambda x:\text{Bool}. x)$   
i.e.,  $(\text{idBB} (\text{idBB idB})) \implies^* \text{idB}$ .

Lemma step\_example2 :

$(\text{tapp idBB} (\text{tapp idBB idB})) \implies^* \text{idB}$ .

Proof.

```
eapply multi_step.  
  apply ST_App2. auto.  
  apply ST_AppAbs. auto.  
eapply multi_step.  
  apply ST_AppAbs. simpl. auto.  
simpl. apply multi_refl. Qed.
```

Example:  $((\lambda x:\text{Bool} \rightarrow \text{Bool}. x) (\lambda x:\text{Bool}. \text{if } x \text{ then false else true})) \text{ true} \implies^* \text{false}$  i.e.,  
 $((\text{idBB notB}) \text{ ttrue}) \implies^* \text{tfalse}$ .

Lemma step\_example3 :

$\text{tapp} (\text{tapp idBB notB}) \text{ ttrue} \implies^* \text{tfalse}$ .

Proof.

```
eapply multi_step.  
  apply ST_App1. apply ST_AppAbs. auto. simpl.  
eapply multi_step.  
  apply ST_AppAbs. auto. simpl.  
eapply multi_step.  
  apply ST_IfTrue. apply multi_refl. Qed.
```

Example:  $((\lambda x:\text{Bool} \rightarrow \text{Bool}. x) ((\lambda x:\text{Bool}. \text{if } x \text{ then false else true}) \text{ true})) \implies^* \text{false}$   
i.e.,  $(\text{idBB} (\text{notB ttrue})) \implies^* \text{tfalse}$ .

Lemma step\_example4 :

$\text{tapp idBB} (\text{tapp notB ttrue}) \implies^* \text{tfalse}$ .

Proof.

```
eapply multi_step.  
  apply ST_App2. auto.  
  apply ST_AppAbs. auto. simpl.  
eapply multi_step.  
  apply ST_App2. auto.  
  apply ST_IfTrue.  
eapply multi_step.  
  apply ST_AppAbs. auto. simpl.
```

apply multi\_refl. Qed.

A more automatic proof

Lemma step\_example1' :  
 (tapp idBB idB) ==>\* idB.

Proof. *normalize*. Qed.

Again, we can use the *normalize* tactic from above to simplify the proof.

Lemma step\_example2' :  
 (tapp idBB (tapp idBB idB)) ==>\* idB.

Proof.  
 *normalize*.

Qed.

Lemma step\_example3' :  
 tapp (tapp idBB notB) ttrue ==>\* tfalse.

Proof. *normalize*. Qed.

Lemma step\_example4' :  
 tapp idBB (tapp notB ttrue) ==>\* tfalse.

Proof. *normalize*. Qed.

**Exercise: 2 stars (step\_example3)** Try to do this one both with and without *normalize*.

Lemma step\_example5 :  
 (tapp (tapp idBBBB idBB) idB)  
 ==>\* idB.

Proof.  
 *Admitted*.

□

## 25.2.4 Typing

### Contexts

*Question:* What is the type of the term "x y"?

*Answer:* It depends on the types of x and y!

I.e., in order to assign a type to a term, we need to know what assumptions we should make about the types of its free variables.

This leads us to a three-place "typing judgment", informally written  $\Gamma \vdash t \text{ in } T$ , where  $\Gamma$  is a "typing context" – a mapping from variables to their types.

Informally, we'll write  $\Gamma, x:T$  for "extend the partial function  $\Gamma$  to also map x to T." Formally, we use the function *extend* to add a binding to a partial map.

**Definition** context := partial\_map ty.

## Typing Relation

$\Gamma \vdash x = T$

---

(T\_Var)  $\Gamma \vdash x \text{ \textbackslash in } T$   
 $\Gamma, x:T11 \vdash t12 \text{ \textbackslash in } T12$

---

(T\_Abs)  $\Gamma \vdash \lambda x:T11.t12 \text{ \textbackslash in } T11 \rightarrow T12$   
 $\Gamma \vdash t1 \text{ \textbackslash in } T11 \rightarrow T12 \quad \Gamma \vdash t2 \text{ \textbackslash in } T11$

---

(T\_App)  $\Gamma \vdash t1 \ t2 \text{ \textbackslash in } T12$

---

(T\_True)  $\Gamma \vdash \text{true} \text{ \textbackslash in } \text{Bool}$

---

(T\_False)  $\Gamma \vdash \text{false} \text{ \textbackslash in } \text{Bool}$   
 $\Gamma \vdash t1 \text{ \textbackslash in } \text{Bool} \quad \Gamma \vdash t2 \text{ \textbackslash in } T \quad \Gamma \vdash t3 \text{ \textbackslash in } T$

---

(T\_If)  $\Gamma \vdash \text{if } t1 \text{ then } t2 \text{ else } t3 \text{ \textbackslash in } T$

We can read the three-place relation  $\Gamma \vdash t \text{ \textbackslash in } T$  as: "to the term  $t$  we can assign the type  $T$  using as types for the free variables of  $t$  the ones specified in the context  $\Gamma$ ."

Reserved Notation " $\Gamma \vdash t \text{ \textbackslash in } T$ " (at level 40).

Inductive **has\_type** : context  $\rightarrow$  **tm**  $\rightarrow$  **ty**  $\rightarrow$  Prop :=

```

| T_Var :  $\forall \Gamma \ x \ T,$ 
   $\Gamma \vdash x = \text{Some } T \rightarrow$ 
   $\Gamma \vdash \text{tvar } x \text{ \textbackslash in } T$ 
| T_Abs :  $\forall \Gamma \ x \ T11 \ T12 \ t12,$ 
   $\text{update } \Gamma \ x \ T11 \vdash t12 \text{ \textbackslash in } T12 \rightarrow$ 
   $\Gamma \vdash \text{tabs } x \ T11 \ t12 \text{ \textbackslash in } T\text{Arrow } T11 \ T12$ 
| T_App :  $\forall T11 \ T12 \ \Gamma \ t1 \ t2,$ 
   $\Gamma \vdash t1 \text{ \textbackslash in } T\text{Arrow } T11 \ T12 \rightarrow$ 
   $\Gamma \vdash t2 \text{ \textbackslash in } T11 \rightarrow$ 
   $\Gamma \vdash \text{tapp } t1 \ t2 \text{ \textbackslash in } T12$ 
| T_True :  $\forall \Gamma,$ 
   $\Gamma \vdash \text{ttrue} \text{ \textbackslash in } T\text{Bool}$ 
| T_False :  $\forall \Gamma,$ 
   $\Gamma \vdash \text{tfalse} \text{ \textbackslash in } T\text{Bool}$ 
| T_If :  $\forall t1 \ t2 \ t3 \ T \ \Gamma,$ 
   $\Gamma \vdash t1 \text{ \textbackslash in } T\text{Bool} \rightarrow$ 
   $\Gamma \vdash t2 \text{ \textbackslash in } T \rightarrow$ 
   $\Gamma \vdash t3 \text{ \textbackslash in } T \rightarrow$ 
   $\Gamma \vdash \text{tif } t1 \ t2 \ t3 \text{ \textbackslash in } T$ 

```

where "Gamma '|-' t '\in' T" := (**has\_type** *Gamma* t T).

Hint Constructors **has\_type**.

## Examples

Example typing\_example\_1 :

empty  $\vdash$  tabs x TBool (tvar x) \in TArrow TBool TBool.

Proof.

apply T\_Abs. apply T\_Var. reflexivity. Qed.

Note that since we added the **has\_type** constructors to the hints database, auto can actually solve this one immediately.

Example typing\_example\_1' :

empty  $\vdash$  tabs x TBool (tvar x) \in TArrow TBool TBool.

Proof. auto. Qed.

Another example: empty  $\vdash$  \x:A. \y:A->A. y (y x)) \in A -> (A->A) -> A.

Example typing\_example\_2 :

empty  $\vdash$   
 (tabs x TBool  
 (tabs y (TArrow TBool TBool)  
 (tapp (tvar y) (tapp (tvar y) (tvar x)))))) \in  
 (TArrow TBool (TArrow (TArrow TBool TBool) TBool)).

Proof with auto using update\_eq.

apply T\_Abs.  
apply T\_Abs.  
eapply T\_App. apply T\_Var...  
eapply T\_App. apply T\_Var...  
apply T\_Var...

Qed.

**Exercise: 2 stars, optional (typing\_example\_2\_full)** Prove the same result without using auto, eauto, or eapply.

Example typing\_example\_2\_full :

empty  $\vdash$   
 (tabs x TBool  
 (tabs y (TArrow TBool TBool)  
 (tapp (tvar y) (tapp (tvar y) (tvar x)))))) \in  
 (TArrow TBool (TArrow (TArrow TBool TBool) TBool)).

Proof.

*Admitted.*

□

**Exercise: 2 stars (typing\_example\_3)** Formally prove the following typing derivation holds:

$\text{empty} \vdash \lambda x:\text{Bool} \rightarrow B. \lambda y:\text{Bool} \rightarrow \text{Bool}. \lambda z:\text{Bool}. y (x z) \text{ in } T.$

Example typing\_example\_3 :

```

 $\exists T,$ 
  empty  $\vdash$ 
    (tabs x (TArrow TBool TBool)
      (tabs y (TArrow TBool TBool)
        (tabs z TBool
          (tapp (tvar y) (tapp (tvar x) (tvar z)))))) \in
      T.

```

Proof with auto.

*Admitted.*

□

We can also show that terms are *not* typable. For example, let's formally check that there is no typing derivation assigning a type to the term  $\lambda x:\text{Bool}. \lambda y:\text{Bool}. x y$  – i.e.,  $\sim$  exists  $T$ ,  $\text{empty} \vdash \lambda x:\text{Bool}. \lambda y:\text{Bool}. x y : T$ .

Example typing\_nonexample\_1 :

```

 $\neg \exists T,$ 
  empty  $\vdash$ 
    (tabs x TBool
      (tabs y TBool
        (tapp (tvar x) (tvar y)))) \in
      T.

```

Proof.

```

intros Hc. inversion Hc.
inversion H. subst. clear H.
inversion H5. subst. clear H5.
inversion H4. subst. clear H4.
inversion H2. subst. clear H2.
inversion H5. subst. clear H5.
inversion H1. Qed.

```

**Exercise: 3 stars, optional (typing\_nonexample\_3)** Another nonexample:  $\sim$  (exists  $S$ , exists  $T$ ,  $\text{empty} \vdash \lambda x:S. x x : T$ ).

Example typing\_nonexample\_3 :

```

 $\neg (\exists S, \exists T,$ 
  empty  $\vdash$ 
    (tabs x S
      (tapp (tvar x) (tvar x))) \in
      T).

```

Proof.

*Admitted.*

□

End STLC.

*Date* : 2016 – 03 – 07 17 : 05 : 52 – 0500 (*Mon, 07 Mar 2016*)

# Chapter 26

## Library StlcProp

### 26.1 StlcProp: Properties of STLC

```
Require Import Sflib.
Require Import Maps.
Require Import Types.
Require Import Stlc.
Require Import Smallstep.
```

```
Module STLCProp.
Import STLC.
```

In this chapter, we develop the fundamental theory of the Simply Typed Lambda Calculus – in particular, the type safety theorem.

### 26.2 Canonical Forms

```
Lemma canonical_forms_bool :  $\forall t$ ,
  empty  $\vdash t$   $\text{in TBool}$   $\rightarrow$ 
  value  $t \rightarrow$ 
   $(t = \text{ttrue}) \vee (t = \text{tfalse})$ .
```

Proof.

```
  intros  $t$   $HT$   $HVal$ .
  inversion  $HVal$ ; intros; subst; try inversion  $HT$ ; auto.
```

Qed.

```
Lemma canonical_forms_fun :  $\forall t$   $T1$   $T2$ ,
  empty  $\vdash t$   $\text{in (TArrow } T1 \ T2)$   $\rightarrow$ 
  value  $t \rightarrow$ 
   $\exists x \ u, t = \text{tabs } x \ T1 \ u$ .
```

Proof.

```

intros t T1 T2 HT HVal.
inversion HVal; intros; subst; try inversion HT; subst; auto.
∃ x0. ∃ t0. auto.
Qed.

```

## 26.3 Progress

As before, the *progress* theorem tells us that closed, well-typed terms are not stuck: either a well-typed term is a value, or it can take an evaluation step. The proof is a relatively straightforward extension of the progress proof we saw in the **Types** chapter.

**Theorem progress** :  $\forall t\ T,$   
 $\text{empty} \vdash t \text{ \texttt{in} } T \rightarrow$   
 $\text{value } t \vee \exists t', t \Rightarrow t'.$

*Proof*: by induction on the derivation of  $\vdash t \text{ \texttt{in} } T$ .

- The last rule of the derivation cannot be **T\_Var**, since a variable is never well typed in an empty context.
- The **T\_True**, **T\_False**, and **T\_Abs** cases are trivial, since in each of these cases we know immediately that  $t$  is a value.
- If the last rule of the derivation was **T\_App**, then  $t = t1\ t2$ , and we know that  $t1$  and  $t2$  are also well typed in the empty context; in particular, there exists a type  $T2$  such that  $\vdash t1 \text{ \texttt{in} } T2 \rightarrow T$  and  $\vdash t2 \text{ \texttt{in} } T2$ . By the induction hypothesis, either  $t1$  is a value or it can take an evaluation step.
  - If  $t1$  is a value, we now consider  $t2$ , which by the other induction hypothesis must also either be a value or take an evaluation step.
    - \* Suppose  $t2$  is a value. Since  $t1$  is a value with an arrow type, it must be a lambda abstraction; hence  $t1\ t2$  can take a step by **ST\_AppAbs**.
    - \* Otherwise,  $t2$  can take a step, and hence so can  $t1\ t2$  by **ST\_App2**.
  - If  $t1$  can take a step, then so can  $t1\ t2$  by **ST\_App1**.
- If the last rule of the derivation was **T\_If**, then  $t = \text{if } t1 \text{ then } t2 \text{ else } t3$ , where  $t1$  has type **Bool**. By the IH,  $t1$  either is a value or takes a step.
  - If  $t1$  is a value, then since it has type **Bool** it must be either **true** or **false**. If it is **true**, then  $t$  steps to  $t2$ ; otherwise it steps to  $t3$ .
  - Otherwise,  $t1$  takes a step, and therefore so does  $t$  (by **ST\_If**).



Proof with eauto.

```

intros t T Ht.
remember (@empty ty) as Gamma.
induction Ht; subst Gamma...
-
  inversion H.
-
  right. destruct IHHt1...
+
  destruct IHHt2...
×
  assert (∃ x0 t0, t1 = tabs x0 T11 t0).
  eapply canonical_forms_fun; eauto.
  destruct H1 as [x0 [t0 Heq]]. subst.
  ∃ ([x0:=t2] t0)...
×
  inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
+
  inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...
-
  right. destruct IHHt1...
+
  destruct (canonical_forms_bool t1); subst; eauto.
+
  inversion H as [t1' Hstp]. ∃ (tif t1' t2 t3)...
Qed.

```

**Exercise: 3 stars, optional (progress\_from\_term\_ind)** Show that progress can also be proved by induction on terms instead of induction on typing derivations.

Theorem progress' :  $\forall t T,$   
 $\text{empty} \vdash t \text{ \texttt{in} } T \rightarrow$   
 $\text{value } t \vee \exists t', t ==> t'.$

Proof.

```

intros t.
induction t; intros T Ht; auto.
Admitted.
□

```

## 26.4 Preservation

The other half of the type soundness property is the preservation of types during reduction. For this, we need to develop some technical machinery for reasoning about variables and substitution. Working from top to bottom (the high-level property we are actually interested in to the lowest-level technical lemmas that are needed by various cases of the more interesting proofs), the story goes like this:

- The *preservation theorem* is proved by induction on a typing derivation, pretty much as we did in the **Types** chapter. The one case that is significantly different is the one for the **ST\_AppAbs** rule, which is defined using the substitution operation. To see that this step preserves typing, we need to know that the substitution itself does. So we prove a...
- *substitution lemma*, stating that substituting a (closed) term  $s$  for a variable  $x$  in a term  $t$  preserves the type of  $t$ . The proof goes by induction on the form of  $t$  and requires looking at all the different cases in the definition of substitution. This time, the tricky cases are the ones for variables and for function abstractions. In both cases, we discover that we need to take a term  $s$  that has been shown to be well-typed in some context  $\Gamma$  and consider the same term  $s$  in a slightly different context  $\Gamma'$ . For this we prove a...
- *context invariance* lemma, showing that typing is preserved under “inessential changes” to the context  $\Gamma$  – in particular, changes that do not affect any of the free variables of the term. For this, we need a careful definition of
- the *free variables* of a term – i.e., the variables occurring in the term that are not in the scope of a function abstraction that binds them.

### 26.4.1 Free Occurrences

A variable  $x$  *appears free in* a term  $t$  if  $t$  contains some occurrence of  $x$  that is not under an abstraction labeled  $x$ . For example:

- $y$  appears free, but  $x$  does not, in  $\lambda x:T \rightarrow U. x y$
- both  $x$  and  $y$  appear free in  $(\lambda x:T \rightarrow U. x y) x$
- no variables appear free in  $\lambda x:T \rightarrow U. \lambda y:T. x y$

```

Inductive appears_free_in : id → tm → Prop :=
| afi_var : ∀ x,
  appears_free_in x (tvar x)
| afi_app1 : ∀ x t1 t2,

```

```

    appears_free_in x t1 → appears_free_in x (tapp t1 t2)
| afi_app2 : ∀ x t1 t2,
    appears_free_in x t2 → appears_free_in x (tapp t1 t2)
| afi_abs : ∀ x y T11 t12,
    y ≠ x →
    appears_free_in x t12 →
    appears_free_in x (tabs y T11 t12)
| afi_if1 : ∀ x t1 t2 t3,
    appears_free_in x t1 →
    appears_free_in x (tif t1 t2 t3)
| afi_if2 : ∀ x t1 t2 t3,
    appears_free_in x t2 →
    appears_free_in x (tif t1 t2 t3)
| afi_if3 : ∀ x t1 t2 t3,
    appears_free_in x t3 →
    appears_free_in x (tif t1 t2 t3).

```

Hint Constructors **appears\_free\_in**.

A term in which no variables appear free is said to be *closed*.

Definition closed (*t*:tm) :=  
 ∀ x, ¬ appears\_free\_in x t.

## 26.4.2 Substitution

We first need a technical lemma connecting free variables and typing contexts. If a variable  $x$  appears free in a term  $t$ , and if we know  $t$  is well typed in context  $\Gamma$ , then it must be the case that  $\Gamma$  assigns a type to  $x$ .

Lemma free\_in\_context : ∀ x t T  $\Gamma$ ,  
 appears\_free\_in x t →  
 $\Gamma \vdash t \text{ in } T \rightarrow$   
 $\exists T', \Gamma x = \text{Some } T'$ .

*Proof*: We show, by induction on the proof that  $x$  appears free in  $t$ , that, for all contexts  $\Gamma$ , if  $t$  is well typed under  $\Gamma$ , then  $\Gamma$  assigns some type to  $x$ .

- If the last rule used was **afi\_var**, then  $t = x$ , and from the assumption that  $t$  is well typed under  $\Gamma$  we have immediately that  $\Gamma$  assigns a type to  $x$ .
- If the last rule used was **afi\_app1**, then  $t = t1\ t2$  and  $x$  appears free in  $t1$ . Since  $t$  is well typed under  $\Gamma$ , we can see from the typing rules that  $t1$  must also be, and the IH then tells us that  $\Gamma$  assigns  $x$  a type.
- Almost all the other cases are similar:  $x$  appears free in a subterm of  $t$ , and since  $t$  is well typed under  $\Gamma$ , we know the subterm of  $t$  in which  $x$  appears is well typed under  $\Gamma$  as well, and the IH gives us exactly the conclusion we want.

- The only remaining case is `afi_abs`. In this case  $t = \backslash y:T11.t12$ , and  $x$  appears free in  $t12$ ; we also know that  $x$  is different from  $y$ . The difference from the previous cases is that whereas  $t$  is well typed under  $\Gamma$ , its body  $t12$  is well typed under  $(\Gamma, y:T11)$ , so the IH allows us to conclude that  $x$  is assigned some type by the extended context  $(\Gamma, y:T11)$ . To conclude that  $\Gamma$  assigns a type to  $x$ , we appeal to lemma `update_neq`, noting that  $x$  and  $y$  are different variables.

Proof.

```
intros x t T Gamma H H0. generalize dependent Gamma.
generalize dependent T.
induction H;
  intros; try solve [inversion H0; eauto].
-
  inversion H1; subst.
  apply IHappears_free_in in H7.
  rewrite update_neq in H7; assumption.
```

Qed.

Next, we'll need the fact that any term  $t$  which is well typed in the empty context is closed – that is, it has no free variables.

**Exercise: 2 stars, optional (`typable_empty__closed`)** Corollary `typable_empty__closed`

$:\forall t T,$   
 $\text{empty} \vdash t \text{ \texttt{in} } T \rightarrow$   
 $\text{closed } t.$

Proof.

*Admitted.*

□

Sometimes, when we have a proof  $\Gamma \vdash t : T$ , we will need to replace  $\Gamma$  by a different context  $\Gamma'$ . When is it safe to do this? Intuitively, it must at least be the case that  $\Gamma'$  assigns the same types as  $\Gamma$  to all the variables that appear free in  $t$ . In fact, this is the only condition that is needed.

**Lemma** `context_invariance` :  $\forall \Gamma \Gamma' t T,$   
 $\Gamma \vdash t \text{ \texttt{in} } T \rightarrow$   
 $(\forall x, \text{appears\_free\_in } x t \rightarrow \Gamma x = \Gamma' x) \rightarrow$   
 $\Gamma' \vdash t \text{ \texttt{in} } T.$

*Proof:* By induction on the derivation of  $\Gamma \vdash t \text{ \texttt{in} } T$ .

- If the last rule in the derivation was `T_Var`, then  $t = x$  and  $\Gamma x = T$ . By assumption,  $\Gamma' x = T$  as well, and hence  $\Gamma' \vdash t \text{ \texttt{in} } T$  by `T_Var`.
- If the last rule was `T_Abs`, then  $t = \backslash y:T11. t12$ , with  $T = T11 \rightarrow T12$  and  $\Gamma, y:T11 \vdash t12 \text{ \texttt{in} } T12$ . The induction hypothesis is that for any context  $\Gamma''$ , if

$\Gamma, y:T11$  and  $\Gamma''$  assign the same types to all the free variables in  $t12$ , then  $t12$  has type  $T12$  under  $\Gamma''$ . Let  $\Gamma'$  be a context which agrees with  $\Gamma$  on the free variables in  $t$ ; we must show  $\Gamma' \vdash \lambda y:T11. t12 \text{ in } T11 \rightarrow T12$ .

By  $T\_Abs$ , it suffices to show that  $\Gamma', y:T11 \vdash t12 \text{ in } T12$ . By the IH (setting  $\Gamma'' = \Gamma', y:T11$ ), it suffices to show that  $\Gamma, y:T11$  and  $\Gamma', y:T11$  agree on all the variables that appear free in  $t12$ .

Any variable occurring free in  $t12$  must either be  $y$ , or some other variable.  $\Gamma, y:T11$  and  $\Gamma', y:T11$  clearly agree on  $y$ . Otherwise, we note that any variable other than  $y$  which occurs free in  $t12$  also occurs free in  $t = \lambda y:T11. t12$ , and by assumption  $\Gamma$  and  $\Gamma'$  agree on all such variables, and hence so do  $\Gamma, y:T11$  and  $\Gamma', y:T11$ .

- If the last rule was  $T\_App$ , then  $t = t1 \ t2$ , with  $\Gamma \vdash t1 \text{ in } T2 \rightarrow T$  and  $\Gamma \vdash t2 \text{ in } T2$ . One induction hypothesis states that for all contexts  $\Gamma'$ , if  $\Gamma'$  agrees with  $\Gamma$  on the free variables in  $t1$ , then  $t1$  has type  $T2 \rightarrow T$  under  $\Gamma'$ ; there is a similar IH for  $t2$ . We must show that  $t1 \ t2$  also has type  $T$  under  $\Gamma'$ , given the assumption that  $\Gamma'$  agrees with  $\Gamma$  on all the free variables in  $t1 \ t2$ . By  $T\_App$ , it suffices to show that  $t1$  and  $t2$  each have the same type under  $\Gamma'$  as under  $\Gamma$ . However, we note that all free variables in  $t1$  are also free in  $t1 \ t2$ , and similarly for free variables in  $t2$ ; hence the desired result follows by the two IHs.

Proof with eauto.

```

intros.
generalize dependent  $\Gamma'$ .
induction H; intros; auto.
-
  apply T_Var. rewrite  $\leftarrow H0$ ...
-
  apply T_Abs.
  apply IHhas_type. intros x1 Hafi.
  unfold update. unfold t_update. destruct (beq_id x0 x1) eqn: Hx0x1...
  rewrite beq_id_false_iff in Hx0x1. auto.
-
  apply T_App with T11...

```

Qed.

Now we come to the conceptual heart of the proof that reduction preserves types – namely, the observation that *substitution* preserves types.

Formally, the so-called *Substitution Lemma* says this: suppose we have a term  $t$  with a free variable  $x$ , and suppose we've been able to assign a type  $T$  to  $t$  under the assumption that  $x$  has some type  $U$ . Also, suppose that we have some other term  $v$  and that we've shown

that  $v$  has type  $U$ . Then, since  $v$  satisfies the assumption we made about  $x$  when typing  $t$ , we should be able to substitute  $v$  for each of the occurrences of  $x$  in  $t$  and obtain a new term that still has type  $T$ .

*Lemma:* If  $\Gamma, x:U \vdash t \text{ in } T$  and  $\vdash v \text{ in } U$ , then  $\Gamma \vdash [x:=v]t \text{ in } T$ .

**Lemma substitution\_preserves\_typing :**  $\forall \Gamma x U t v T,$

$\text{update } \Gamma x U \vdash t \text{ in } T \rightarrow$

$\text{empty} \vdash v \text{ in } U \rightarrow$

$\Gamma \vdash [x:=v]t \text{ in } T.$

One technical subtlety in the statement of the lemma is that we assign  $v$  the type  $U$  in the *empty* context – in other words, we assume  $v$  is closed. This assumption considerably simplifies the  $T\_Abs$  case of the proof (compared to assuming  $\Gamma \vdash v \text{ in } U$ , which would be the other reasonable assumption at this point) because the context invariance lemma then tells us that  $v$  has type  $U$  in any context at all – we don't have to worry about free variables in  $v$  clashing with the variable being introduced into the context by  $T\_Abs$ .

*Proof:* We prove, by induction on  $t$ , that, for all  $T$  and  $\Gamma$ , if  $\Gamma, x:U \vdash t \text{ in } T$  and  $\vdash v \text{ in } U$ , then  $\Gamma \vdash [x:=v]t \text{ in } T$ .

- If  $t$  is a variable, there are two cases to consider, depending on whether  $t$  is  $x$  or some other variable.
  - If  $t = x$ , then from the fact that  $\Gamma, x:U \vdash x \text{ in } T$  we conclude that  $U = T$ . We must show that  $[x:=v]x = v$  has type  $T$  under  $\Gamma$ , given the assumption that  $v$  has type  $U = T$  under the empty context. This follows from context invariance: if a closed term has type  $T$  in the empty context, it has that type in any context.
  - If  $t$  is some variable  $y$  that is not equal to  $x$ , then we need only note that  $y$  has the same type under  $\Gamma, x:U$  as under  $\Gamma$ .
- If  $t$  is an abstraction  $\lambda y:T11. t12$ , then the IH tells us, for all  $\Gamma'$  and  $T'$ , that if  $\Gamma', x:U \vdash t12 \text{ in } T'$  and  $\vdash v \text{ in } U$ , then  $\Gamma' \vdash [x:=v]t12 \text{ in } T'$ .

The substitution in the conclusion behaves differently, depending on whether  $x$  and  $y$  are the same variable name.

First, suppose  $x = y$ . Then, by the definition of substitution,  $[x:=v]t = t$ , so we just need to show  $\Gamma \vdash t \text{ in } T$ . But we know  $\Gamma, x:U \vdash t : T$ , and since the variable  $y$  does not appear free in  $\lambda y:T11. t12$ , the context invariance lemma yields  $\Gamma \vdash t \text{ in } T$ .

Second, suppose  $x \neq y$ . We know  $\Gamma, x:U, y:T11 \vdash t12 \text{ in } T12$  by inversion of the typing relation, and  $\Gamma, y:T11, x:U \vdash t12 \text{ in } T12$  follows from this by the context invariance lemma, so the IH applies, giving us  $\Gamma, y:T11 \vdash [x:=v]t12 \text{ in } T12$ . By  $T\_Abs$ ,  $\Gamma \vdash \lambda y:T11. [x:=v]t12 \text{ in } T11 \rightarrow T12$ , and by the definition of substitution (noting that  $x \neq y$ ),  $\Gamma \vdash \lambda y:T11. [x:=v]t12 \text{ in } T11 \rightarrow T12$  as required.

- If  $t$  is an application  $t1\ t2$ , the result follows straightforwardly from the definition of substitution and the induction hypotheses.
- The remaining cases are similar to the application case.

Another technical note: This proof is a rare case where an induction on terms, rather than typing derivations, yields a simpler argument. The reason for this is that the assumption  $\text{update } \Gamma \times U \vdash t \text{ in } T$  is not completely generic, in the sense that one of the "slots" in the typing relation – namely the context – is not just a variable, and this means that Coq's native induction tactic does not give us the induction hypothesis that we want. It is possible to work around this, but the needed generalization is a little tricky. The term  $t$ , on the other hand, *is* completely generic.

Proof with `eauto`.

```

intros  $\Gamma$   $x$   $U$   $t$   $v$   $T$   $Ht$   $Ht'$ .
generalize dependent  $\Gamma$ . generalize dependent  $T$ .
induction t; intros  $T$   $\Gamma$   $H$ ;

  inversion  $H$ ; subst; simpl...
-
  rename  $i$  into  $y$ . destruct (beq_idP  $x$   $y$ ) as [ $Hxy$ | $Hxy$ ].
+
  subst.
  rewrite update_eq in  $H2$ .
  inversion  $H2$ ; subst. clear  $H2$ .
    eapply context_invariance... intros  $x$   $Hcontra$ .
    destruct (free_in_context _ _  $T$  empty  $Hcontra$ ) as [ $T'$   $HT'$ ]...
    inversion  $HT'$ .
+
  apply T_Var. rewrite update_neq in  $H2$ ...
-
  rename  $i$  into  $y$ . apply T_Abs.
  destruct (beq_idP  $x$   $y$ ) as [ $Hxy$  |  $Hxy$ ].
+
  subst.
  eapply context_invariance...
  intros  $x$   $Hafi$ . unfold update, t_update.
  destruct (beq_id  $y$   $x$ ) eqn:  $Hyx$ ...
+
  apply IHt. eapply context_invariance...
  intros  $z$   $Hafi$ . unfold update, t_update.
  destruct (beq_idP  $y$   $z$ ) as [ $Hyz$  |  $Hyz$ ]; subst; trivial.
  rewrite  $\leftarrow$  beq_id_false_iff in  $Hxy$ .
  rewrite  $Hxy$ ...

```

Qed.

The substitution lemma can be viewed as a kind of "commutation" property. Intuitively, it says that substitution and typing can be done in either order: we can either assign types to the terms  $t$  and  $v$  separately (under suitable contexts) and then combine them using substitution, or we can substitute first and then assign a type to  $[x:=v] t$  – the result is the same either way.

### 26.4.3 Main Theorem

We now have the tools we need to prove preservation: if a closed term  $t$  has type  $T$ , and takes an evaluation step to  $t'$ , then  $t'$  is also a closed term with type  $T$ . In other words, the small-step evaluation relation preserves types.

**Theorem preservation :**  $\forall t t' T,$   
 $\text{empty} \vdash t \text{ \texttt{in} } T \rightarrow$   
 $t ==> t' \rightarrow$   
 $\text{empty} \vdash t' \text{ \texttt{in} } T.$

*Proof:* by induction on the derivation of  $\vdash t \text{ \texttt{in} } T$ .

- We can immediately rule out  $T\_Var$ ,  $T\_Abs$ ,  $T\_True$ , and  $T\_False$  as the final rules in the derivation, since in each of these cases  $t$  cannot take a step.
- If the last rule in the derivation was  $T\_App$ , then  $t = t1 \ t2$ . There are three cases to consider, one for each rule that could have been used to show that  $t1 \ t2$  takes a step to  $t'$ .
  - If  $t1 \ t2$  takes a step by  $ST\_App1$ , with  $t1$  stepping to  $t1'$ , then by the IH  $t1'$  has the same type as  $t1$ , and hence  $t1' \ t2$  has the same type as  $t1 \ t2$ .
  - The  $ST\_App2$  case is similar.
  - If  $t1 \ t2$  takes a step by  $ST\_AppAbs$ , then  $t1 = \backslash x:T11.t12$  and  $t1 \ t2$  steps to  $[x:=t2]t12$ ; the desired result now follows from the fact that substitution preserves types.
- If the last rule in the derivation was  $T\_If$ , then  $t = \texttt{if } t1 \texttt{ then } t2 \texttt{ else } t3$ , and there are again three cases depending on how  $t$  steps.
  - If  $t$  steps to  $t2$  or  $t3$ , the result is immediate, since  $t2$  and  $t3$  have the same type as  $t$ .
  - Otherwise,  $t$  steps by  $ST\_If$ , and the desired conclusion follows directly from the induction hypothesis.



Proof with eauto.

```
remember (@empty ty) as Gamma.
intros t t' T HT. generalize dependent t'.
induction HT;
  intros t' HE; subst Gamma; subst;
  try solve [inversion HE; subst; auto].
-
  inversion HE; subst...
+
  apply substitution_preserves_typing with T11...
  inversion HT1...
```

Qed.

**Exercise: 2 stars, recommended (subject\_expansion\_stlc)** An exercise in the *Types* chapter asked about the subject expansion property for the simple language of arithmetic and boolean expressions. Does this property hold for STLC? That is, is it always the case that, if  $t \Rightarrow t'$  and **has\_type**  $t' T$ , then  $\text{empty} \vdash t \text{ in } T$ ? If so, prove it. If not, give a counter-example not involving conditionals.

□

## 26.5 Type Soundness

**Exercise: 2 stars, optional (type\_soundness)** Put progress and preservation together and show that a well-typed term can *never* reach a stuck state.

Definition stuck ( $t:\mathbf{tm}$ ) : Prop :=  
 (normal\_form step)  $t \wedge \neg$  value  $t$ .

Corollary soundness :  $\forall t t' T$ ,  
 $\text{empty} \vdash t \text{ in } T \rightarrow$   
 $t \Rightarrow^* t' \rightarrow$   
 $\sim(\text{stuck } t')$ .

Proof.

```
intros t t' T Hhas_type Hmulti. unfold stuck.
intros [Hnf Hnot_val]. unfold normal_form in Hnf.
induction Hmulti.
Admitted.
```

## 26.6 Uniqueness of Types

**Exercise: 3 stars (types\_unique)** Another pleasant property of the STLC is that types are unique: a given term (in a given context) has at most one type. Formalize this statement and prove it.

□

## 26.7 Additional Exercises

**Exercise: 1 star (progress\_preservation\_statement)** Without peeking, write down the progress and preservation theorems for the simply typed lambda-calculus. □

**Exercise: 2 stars (stlc\_variation1)** Suppose we add a new term *zap* with the following reduction rule:

---

(ST\_Zap)  $t \implies \text{zap}$  and the following typing rule:

---

(T\_Zap)  $\Gamma \vdash \text{zap} : T$  Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

**Exercise: 2 stars (stlc\_variation2)** Suppose instead that we add a new term *foo* with the following reduction rules:

---

(ST\_Foo1)  $(\lambda x:A. x) \implies \text{foo}$

---

(ST\_Foo2)  $\text{foo} \implies \text{true}$  Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

**Exercise: 2 stars (stlc\_variation3)** Suppose instead that we remove the rule `ST_App1` from the **step** relation. Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

**Exercise: 2 stars, optional (stlc\_variation4)** Suppose instead that we add the following new rule to the reduction relation:

---

(`ST_FunnyIfTrue`) (if true then  $t_1$  else  $t_2$ )  $\Rightarrow$  true Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

**Exercise: 2 stars, optional (stlc\_variation5)** Suppose instead that we add the following new rule to the typing relation:  $\Gamma \vdash t_1 \text{ \texttt{\textbackslash in Bool} } \rightarrow \text{Bool} \rightarrow \text{Bool}$   $\Gamma \vdash t_2 \text{ \texttt{\textbackslash in Bool} }$

---

(`T_FunnyApp`)  $\Gamma \vdash t_1 \text{ \texttt{\textbackslash in Bool} } \rightarrow \text{Bool}$  Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

**Exercise: 2 stars, optional (stlc\_variation6)** Suppose instead that we add the following new rule to the typing relation:  $\Gamma \vdash t_1 \text{ \texttt{\textbackslash in Bool}} \quad \Gamma \vdash t_2 \text{ \texttt{\textbackslash in Bool}}$

---

(T\_FunnyApp')  $\Gamma \vdash t_1 \ t_2 \text{ \texttt{\textbackslash in Bool}}$  Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

**Exercise: 2 stars, optional (stlc\_variation7)** Suppose we add the following new rule to the typing relation of the STLC:

---

(T\_FunnyAbs)  $\vdash \lambda x:\texttt{Bool.t} \text{ \texttt{\textbackslash in Bool}}$  Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of **step**
- Progress
- Preservation

□

End STLCPROP.

## 26.7.1 Exercise: STLC with Arithmetic

To see how the STLC might function as the core of a real programming language, let's extend it with a concrete base type of numbers and some constants and primitive operators.

Module STLCARITH.

To types, we add a base type of natural numbers (and remove booleans, for brevity)

```
Inductive ty : Type :=
| TArrow : ty → ty → ty
| TNat : ty.
```

To terms, we add natural number constants, along with successor, predecessor, multiplication, and zero-testing...

```
Inductive tm : Type :=
| tvar : id → tm
```

```

| tapp : tm → tm → tm
| tabs : id → ty → tm → tm
| tnat : nat → tm
| tsucc : tm → tm
| tpred : tm → tm
| tmult : tm → tm → tm
| tif0 : tm → tm → tm → tm.

```

**Exercise: 4 stars (stlc\_arith)** Finish formalizing the definition and properties of the STLC extended with arithmetic. Specifically:

- Copy the whole development of STLC that we went through above (from the definition of values through the Progress theorem), and paste it into the file at this point.
- Extend the definitions of the **subst** operation and the **step** relation to include appropriate clauses for the arithmetic operators.
- Extend the proofs of all the properties (up to **soundness**) of the original STLC to deal with the new syntactic forms. Make sure Coq accepts the whole file.

□

End STLCARITH.

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500 (*Wed, 17 Feb 2016*)

# Chapter 27

## Library MoreStlc

### 27.1 MoreStlc: More on the Simply Typed Lambda-Calculus

```
Require Import Sflib.  
Require Import Maps.  
Require Import Types.  
Require Import Smallstep.  
Require Import Stlc.
```

### 27.2 Simple Extensions to STLC

The simply typed lambda-calculus has enough structure to make its theoretical properties interesting, but it is not much of a programming language. In this chapter, we begin to close the gap with real-world languages by introducing a number of familiar features that have straightforward treatments at the level of typing.

#### 27.2.1 Numbers

Adding types, constants, and primitive operations for numbers is easy – just a matter of combining the `Types` and `Stlc` chapters.

#### 27.2.2 `let`-bindings

When writing a complex expression, it is often useful to give names to some of its subexpressions: this avoids repetition and often increases readability. Most languages provide one or more ways of doing this. In OCaml (and Coq), for example, we can write `let x=t1 in t2` to mean “evaluate the expression *t1* and bind the name *x* to the resulting value while evaluating *t2*.”

Our `let`-binder follows OCaml’s in choosing a call-by-value evaluation order, where the `let`-bound term must be fully evaluated before evaluation of the `let`-body can begin. The

typing rule  $T\_Let$  tells us that the type of a `let` can be calculated by calculating the type of the `let`-bound term, extending the context with a binding with this type, and in this enriched context calculating the type of the body, which is then the type of the whole `let` expression.

At this point in the course, it's probably easier simply to look at the rules defining this new feature as to wade through a lot of english text conveying the same information. Here they are:

Syntax:

$t ::=$	Terms
...	(other terms same as before)
<code>let x=t in t</code>	let-binding

Reduction:  $t1 ==> t1'$

---

(ST-Let1)  $\text{let } x=t1 \text{ in } t2 ==> \text{let } x=t1' \text{ in } t2$

---

(ST-LetValue)  $\text{let } x=v1 \text{ in } t2 ==> x:=v1t2$  Typing:  $\Gamma \vdash t1 : T1 \quad \Gamma, x:T1 \vdash t2 : T2$

---

(T-Let)  $\Gamma \vdash \text{let } x=t1 \text{ in } t2 : T2$

---

### 27.2.3 Pairs

Our functional programming examples in Coq have made frequent use of *pairs* of values. The type of such pairs is called a *product type*.

The formalization of pairs is almost too simple to be worth discussing. However, let's look briefly at the various parts of the definition to emphasize the common pattern.

In Coq, the primitive way of extracting the components of a pair is *pattern matching*. An alternative style is to take `fst` and `snd` – the first- and second-projection operators – as primitives. Just for fun, let's do our products this way. For example, here's how we'd write a function that takes a pair of numbers and returns the pair of their sum and difference:

```
\x:Nat*Nat.
  let sum = x.fst + x.snd in
  let diff = x.fst - x.snd in
  (sum,diff)
```

Adding pairs to the simply typed lambda-calculus, then, involves adding two new forms of term – pairing, written  $(t1,t2)$ , and projection, written  $t.fst$  for the first projection from  $t$  and  $t.snd$  for the second projection – plus one new type constructor,  $T1 \times T2$ , called the *product* of  $T1$  and  $T2$ .

Syntax:

$t ::=$	Terms
...	
$(t, t)$	pair
$t.fst$	first projection
$t.snd$	second projection
$v ::=$	Values
...	
$(v, v)$	pair value
$T ::=$	Types
...	
$T * T$	product type

For evaluation, we need several new rules specifying how pairs and projection behave.  $t1 \implies t1'$

---


$$\begin{array}{l} (ST\_Pair1) \quad t1, t2 \implies (t1', t2) \\ \quad t2 \implies t2' \end{array}$$


---

$$\begin{array}{l} (ST\_Pair2) \quad (v1, t2) \implies (v1, t2') \\ \quad t1 \implies t1' \end{array}$$


---

$$(ST\_Fst1) \quad t1.fst \implies t1'.fst$$


---

$$\begin{array}{l} (ST\_FstPair) \quad (v1, v2).fst \implies v1 \\ \quad t1 \implies t1' \end{array}$$


---

$$(ST\_Snd1) \quad t1.snd \implies t1'.snd$$


---

$$(ST\_SndPair) \quad (v1, v2).snd \implies v2$$


---

Rules  $ST\_FstPair$  and  $ST\_SndPair$  specify that, when a fully evaluated pair meets a first or second projection, the result is the appropriate component. The congruence rules  $ST\_Fst1$  and  $ST\_Snd1$  allow reduction to proceed under projections, when the term being projected from has not yet been fully evaluated.  $ST\_Pair1$  and  $ST\_Pair2$  evaluate the parts of pairs: first the left part, and then – when a value appears on the left – the right part. The ordering arising from the use of the metavariables  $v$  and  $t$  in these rules enforces a left-to-right evaluation strategy for pairs. (Note the implicit convention that metavariables like  $v$  and  $v1$  can only denote values.) We’ve also added a clause to the definition of values, above, specifying that  $(v1, v2)$  is a value. The fact that the components of a pair value must themselves be values ensures that a pair passed as an argument to a function will be fully evaluated before the function body starts executing.



The typing rules for pairs and projections are straightforward.  $\Gamma \vdash t1 : T1$   $\Gamma \vdash t2 : T2$

---

(T\_Pair)  $\Gamma \vdash (t1, t2) : T1 * T2$   
 $\Gamma \vdash t1 : T11 * T12$

---

(T\_Fst)  $\Gamma \vdash t1.fst : T11$   
 $\Gamma \vdash t1 : T11 * T12$

---

(T\_Snd)  $\Gamma \vdash t1.snd : T12$

The rule T\_Pair says that  $(t1, t2)$  has type  $T1 \times T2$  if  $t1$  has type  $T1$  and  $t2$  has type  $T2$ . Conversely, the rules T\_Fst and T\_Snd tell us that, if  $t1$  has a product type  $T11 \times T12$  (i.e., if it will evaluate to a pair), then the types of the projections from this pair are  $T11$  and  $T12$ .

## 27.2.4 Unit

Another handy base type, found especially in languages in the ML family, is the singleton type *Unit*. It has a single element – the term constant **unit** (with a small *u*) – and a typing rule making **unit** an element of *Unit*. We also add **unit** to the set of possible result values of computations – indeed, **unit** is the *only* possible result of evaluating an expression of type *Unit*.

Syntax:

<b>t ::=</b>	<b>Terms</b>
...	
<b>unit</b>	<b>unit value</b>
 <b>v ::=</b>	 <b>Values</b>
...	
<b>unit</b>	<b>unit</b>
 <b>T ::=</b>	 <b>Types</b>
...	
<b>Unit</b>	<b>Unit type</b>

Typing:

---

(T\_Unit)  $\Gamma \vdash \text{unit} : \text{Unit}$

It may seem a little strange to bother defining a type that has just one element – after all, wouldn't every computation living in such a type be trivial?

This is a fair question, and indeed in the STLC the *Unit* type is not especially critical (though we'll see two uses for it below). Where *Unit* really comes in handy is in richer languages with various sorts of *side effects* – e.g., assignment statements that mutate variables

or pointers, exceptions and other sorts of nonlocal control structures, etc. In such languages, it is convenient to have a type for the (trivial) result of an expression that is evaluated only for its effect.

## 27.2.5 Sums

Many programs need to deal with values that can take two distinct forms. For example, we might identify employees in an accounting application using either their name *or* their id number. A search function might return *either* a matching value *or* an error code.

These are specific examples of a binary *sum type*, which describes a set of values drawn from exactly two given types, e.g.

`Nat + Bool`

We create elements of these types by *tagging* elements of the component types. For example, if `n` is a `Nat` then `inl n` is an element of `Nat+Bool`; similarly, if `b` is a `Bool` then `inr b` is a `Nat+Bool`. The names of the tags `inl` and `inr` arise from thinking of them as functions

```
inl : Nat -> Nat + Bool
inr : Bool -> Nat + Bool
```

that “inject” elements of `Nat` or `Bool` into the left and right components of the sum type `Nat+Bool`. (But note that we don’t actually treat them as functions in the way we formalize them: `inl` and `inr` are keywords, and `inl t` and `inr t` are primitive syntactic forms, not function applications. This allows us to give them their own special typing rules.)

In general, the elements of a type `T1 + T2` consist of the elements of `T1` tagged with the token `inl`, plus the elements of `T2` tagged with `inr`.

One important usage of sums is signaling errors:

```
div : Nat -> Nat -> (Nat + Unit) =
div =
  \x:Nat. \y:Nat.
    if iszero y then
      inr unit
    else
      inl ...
```

The type `Nat + Unit` above is in fact isomorphic to **option nat** in Coq, and we’ve already seen how to signal errors with options.

To *use* elements of sum types, we introduce a **case** construct (a very simplified form of Coq’s **match**) to destruct them. For example, the following procedure converts a `Nat+Bool` into a `Nat`:

```

getNat =
  \x:Nat+Bool.
    case x of
      inl n => n
    | inr b => if b then 1 else 0

```

More formally...

Syntax:

$t ::=$	$\text{Terms}$
$\begin{array}{l}   \dots \\   \text{inl } T \ t \\   \text{inr } T \ t \\   \text{case } t \text{ of} \\ \quad \text{inl } x \Rightarrow t \\ \quad   \text{inr } x \Rightarrow t \end{array}$	$\begin{array}{l} \\ \text{tagging (left)} \\ \text{tagging (right)} \\ \text{case} \end{array}$
$v ::=$	$\text{Values}$
$\begin{array}{l}   \dots \\   \text{inl } T \ v \\   \text{inr } T \ v \end{array}$	$\begin{array}{l} \\ \text{tagged value (left)} \\ \text{tagged value (right)} \end{array}$
$T ::=$	$\text{Types}$
$\begin{array}{l}   \dots \\   T + T \end{array}$	$\begin{array}{l} \\ \text{sum type} \end{array}$

Evaluation:

$t1 \implies t1'$

---

(ST\_Inl)  $\text{inl } T \ t1 \implies \text{inl } T \ t1'$   
 $t1 \implies t1'$

---

(ST\_Inr)  $\text{inr } T \ t1 \implies \text{inr } T \ t1'$   
 $t0 \implies t0'$

---

(ST\_Case)  $\text{case } t0 \text{ of } \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \implies \text{case } t0' \text{ of } \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2$

---

(ST\_CaseInl)  $\text{case } (\text{inl } T \ v0) \text{ of } \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \implies x1 := v0 t1$

---

(ST\_CaseInr)  $\text{case } (\text{inr } T \ v0) \text{ of } \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 \implies x2 := v0 t2$   
 Typing:  $\Gamma \vdash t1 : T1$

---

(T\_Inl)  $\Gamma \vdash \text{inl } T2 \ t1 : T1 + T2$

$\Gamma \vdash t1 : T2$

---

(T\_Inr)  $\Gamma \vdash \text{inr } T1 \ t1 : T1 + T2$

$\Gamma \vdash t0 : T1+T2 \quad \Gamma, x1:T1 \vdash t1 : T \quad \Gamma, x2:T2 \vdash t2 : T$

---

(T\_Case)  $\Gamma \vdash \text{case } t0 \text{ of } \text{inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 : T$

We use the type annotation in *inl* and *inr* to make the typing simpler, similarly to what we did for functions. Without this extra information, the typing rule T\_Inl, for example, would have to say that, once we have shown that *t1* is an element of type *T1*, we can derive that *inl t1* is an element of *T1 + T2* for *any* type *T2*. For example, we could derive both *inl 5 : Nat + Nat* and *inl 5 : Nat + Bool* (and infinitely many other types). This failure of uniqueness of types would mean that we cannot build a typechecking algorithm simply by “reading the rules from bottom to top” as we could for all the other features seen so far.

There are various ways to deal with this difficulty. One simple one – which we’ve adopted here – forces the programmer to explicitly annotate the “other side” of a sum type when performing an injection. This is rather heavyweight for programmers (and so real languages adopt other solutions), but it is easy to understand and formalize.

## 27.2.6 Lists

The typing features we have seen can be classified into *base types* like **Bool**, and *type constructors* like  $\rightarrow$  and  $\times$  that build new types from old ones. Another useful type constructor is **List**. For every type *T*, the type **List T** describes finite-length lists whose elements are drawn from *T*.

In principle, we could encode lists using pairs, sums and *recursive* types. But giving semantics to recursive types is non-trivial. Instead, we’ll just discuss the special case of lists directly.

Below we give the syntax, semantics, and typing rules for lists. Except for the fact that explicit type annotations are mandatory on **nil** and cannot appear on **cons**, these lists are essentially identical to those we built in Coq. We use *lcase* to destruct lists, to avoid dealing with questions like “what is the *head* of the empty list?”

For example, here is a function that calculates the sum of the first two elements of a list of numbers:

```
\x:List Nat.
lcase x of nil -> 0
| a::x' -> lcase x' of nil -> a
| b::x'' -> a+b
```

Syntax:

$t ::=$ $\mid \dots$ $\mid \text{nil } T$	<b>Terms</b>
---	--------------

```

      | cons t t
      | lcase t of nil -> t | x::x -> t

v ::=                               Values
  | ...
  | nil T                          nil value
  | cons v v                        cons value

T ::=                               Types
  | ...
  | List T                          list of Ts

```

Reduction:  $t1 ==> t1'$

---

(ST\_Cons1)  $\text{cons } t1 \ t2 ==> \text{cons } t1' \ t2$   
 $t2 ==> t2'$

---

(ST\_Cons2)  $\text{cons } v1 \ t2 ==> \text{cons } v1 \ t2'$   
 $t1 ==> t1'$

---

(ST\_Lcase1)  $(\text{lcase } t1 \text{ of nil} \rightarrow t2 \mid xh::xt \rightarrow t3) ==> (\text{lcase } t1' \text{ of nil} \rightarrow t2 \mid xh::xt \rightarrow t3)$

---

(ST\_LcaseNil)  $(\text{lcase nil } T \text{ of nil} \rightarrow t2 \mid xh::xt \rightarrow t3) ==> t2$

---

(ST\_LcaseCons)  $(\text{lcase } (\text{cons } vh \ vt) \text{ of nil} \rightarrow t2 \mid xh::xt \rightarrow t3) ==> xh:=vh, xt:=vtt3$   
 Typing:

---

(T\_Nil)  $\Gamma \vdash \text{nil } T : \text{List } T$   
 $\Gamma \vdash t1 : T \quad \Gamma \vdash t2 : \text{List } T$

---

(T\_Cons)  $\Gamma \vdash \text{cons } t1 \ t2 : \text{List } T$   
 $\Gamma \vdash t1 : \text{List } T1 \quad \Gamma \vdash t2 : T \quad \Gamma, h:T1, t:\text{List } T1 \vdash t3 : T$

---

(T\_Lcase)  $\Gamma \vdash (\text{lcase } t1 \text{ of nil} \rightarrow t2 \mid h::t \rightarrow t3) : T$

---

## 27.2.7 General Recursion

Another facility found in most programming languages (including Coq) is the ability to define recursive functions. For example, we might like to be able to define the factorial function like this:

```

fact = \x:Nat.
      if x=0 then 1 else x * (fact (pred x))

```

But this would require quite a bit of work to formalize: we'd have to introduce a notion of "function definitions" and carry around an "environment" of such definitions in the definition of the **step** relation.

Here is another way that is straightforward to formalize: instead of writing recursive definitions where the right-hand side can contain the identifier being defined, we can define a *fixed-point operator* that performs the "unfolding" of the recursive definition in the right-hand side lazily during reduction.

```
fact =
  fix
    (\f:Nat->Nat.
      \x:Nat.
        if x=0 then 1 else x * (f (pred x)))
```

The intuition is that the higher-order function *f* passed to **fix** is a *generator* for the **fact** function: if *f* is applied to a function that approximates the desired behavior of **fact** up to some number *n* (that is, a function that returns correct results on inputs less than or equal to *n*), then it returns a better approximation to **fact** – a function that returns correct results for inputs up to *n*+1. Applying **fix** to this generator returns its *fixed point* – a function that gives the desired behavior for all inputs *n*.

(The term "fixed point" has exactly the same sense as in ordinary mathematics, where a fixed point of a function *f* is an input *x* such that *f*(*x*) = *x*. Here, a fixed point of a function *F* of type (say)  $(Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)$  is a function *f* such that *F f* is behaviorally equivalent to *f*.)

Syntax:

<b>t ::=</b>	<b>Terms</b>
...	
<b>fix t</b>	<b>fixed-point operator</b>

Reduction:  $t1 ==> t1'$

---

(ST\_Fix1)  $\text{fix } t1 ==> \text{fix } t1'$   
 $F = \lambda x:f:T1. t2$

---

(ST\_FixAbs)  $\text{fix } F ==> x f := \text{fix } F t2$  Typing:  $\Gamma \vdash t1 : T1 \rightarrow T1$

---

(T\_Fix)  $\Gamma \vdash \text{fix } t1 : T1$

Let's see how *ST\_FixAbs* works by reducing **fact 3** = **fix F 3**, where  $F = (\lambda f. \lambda x. \text{if } x=0 \text{ then } 1 \text{ else } x \times (f (\text{pred } x)))$  (we are omitting type annotations for brevity here).

**fix F 3**

$==> ST\_FixAbs$

```

(\x. if x=0 then 1 else x * (fix F (pred x))) 3
==> ST_AppAbs
if 3=0 then 1 else 3 * (fix F (pred 3))
==> ST_If0_Nonzero
3 * (fix F (pred 3))
==> ST_FixAbs + ST_Mult2
3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 3))
==> ST_PredNat + ST_Mult2 + ST_App2
3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 2)
==> ST_AppAbs + ST_Mult2
3 * (if 2=0 then 1 else 2 * (fix F (pred 2)))
==> ST_If0_Nonzero + ST_Mult2
3 * (2 * (fix F (pred 2)))
==> ST_FixAbs + 2 x ST_Mult2
3 * (2 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 2)))
==> ST_PredNat + 2 x ST_Mult2 + ST_App2
3 * (2 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 1))
==> ST_AppAbs + 2 x ST_Mult2
3 * (2 * (if 1=0 then 1 else 1 * (fix F (pred 1))))
==> ST_If0_Nonzero + 2 x ST_Mult2
3 * (2 * (1 * (fix F (pred 1))))
==> ST_FixAbs + 3 x ST_Mult2
3 * (2 * (1 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 1))))
==> ST_PredNat + 3 x ST_Mult2 + ST_App2
3 * (2 * (1 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 0)))
==> ST_AppAbs + 3 x ST_Mult2

```

```

3 * (2 * (1 * (if 0=0 then 1 else 0 * (fix F (pred 0)))))
==> ST_If0Zero + 3 x ST_Mult2

3 * (2 * (1 * 1))
==> ST_MultNats + 2 x ST_Mult2

3 * (2 * 1)
==> ST_MultNats + ST_Mult2

3 * 2
==> ST_MultNats

6

```

**Exercise: 1 star, optional (halve\_fix)** Translate this informal recursive definition into one using `fix`:

```

halve =
  \x:Nat.
    if x=0 then 0
    else if (pred x)=0 then 0
    else 1 + (halve (pred (pred x)))

```

□

**Exercise: 1 star, optional (fact\_steps)** Write down the sequence of steps that the term `fact 1` goes through to reduce to a normal form (assuming the usual reduction rules for arithmetic operations).

□

The ability to form the fixed point of a function of type  $T \rightarrow T$  for any  $T$  has some surprising consequences. In particular, it implies that *every* type is inhabited by some term. To see this, observe that, for every type  $T$ , we can define the term `fix (\x:T.x)` By  $T\_Fix$  and  $T\_Abs$ , this term has type  $T$ . By  $ST\_FixAbs$  it reduces to itself, over and over again. Thus it is an *undefined element* of  $T$ .

More usefully, here's an example using `fix` to define a two-argument recursive function:

```

equal =
  fix
    (\eq:Nat->Nat->Bool.
      \m:Nat. \n:Nat.
        if m=0 then iszero n
        else if n=0 then false
        else eq (pred m) (pred n))

```



And finally, here is an example where `fix` is used to define a *pair* of recursive functions (illustrating the fact that the type  $T1$  in the rule  $T\_Fix$  need not be a function type):

```
evenodd =
  fix
    (\eo: (Nat->Bool * Nat->Bool).
      let e = \n:Nat. if n=0 then true  else eo.snd (pred n) in
      let o = \n:Nat. if n=0 then false else eo.fst  (pred n) in
      (e,o))

even = evenodd.fst
odd  = evenodd.snd
```

## 27.2.8 Records

As a final example of a basic extension of the STLC, let's look briefly at how to define *records* and their types. Intuitively, records can be obtained from pairs by two kinds of generalization: they are  $n$ -ary products (rather than just binary) and their fields are accessed by *label* (rather than position).

Conceptually, this extension is a straightforward generalization of pairs and product types, but notationally it becomes a little heavier; for this reason, we postpone its formal treatment to a separate chapter (Records).

Records are not included in the extended exercise below, but they will be useful to motivate the `Sub` chapter.

Syntax:

<code>t ::=</code>	Terms
...	
{i1=t1, ..., in=tn}	record
t.i	projection
 <code>v ::=</code>	 Values
...	
{i1=v1, ..., in=vn}	record value
 <code>T ::=</code>	 Types
...	
{i1:T1, ..., in:Tn}	record type

Intuitively, the generalization is pretty obvious. But it's worth noticing that what we've actually written is rather informal: in particular, we've written "..." in several places to mean "any number of these," and we've omitted explicit mention of the usual side-condition that the labels of a record should not contain repetitions.

It is possible to devise informal notations that are more precise, but these tend to be quite heavy and to obscure the main points of the definitions. So we'll leave these a bit loose here (they are informal anyway, after all) and do the work of tightening things up elsewhere (in chapter **Records**).

Reduction:  $ti \implies ti'$

---

(ST\_Rcd)  $\{i1=v1, \dots, im=vm, in=ti, \dots\} \implies \{i1=v1, \dots, im=vm, in=ti', \dots\}$   
 $t1 \implies t1'$

---

(ST\_Proj1)  $t1.i \implies t1'.i$

---

(ST\_ProjRcd)  $\{\dots, i=vi, \dots\}.i \implies vi$  Again, these rules are a bit informal. For example, the first rule is intended to be read "if  $ti$  is the leftmost field that is not a value and if  $ti$  steps to  $ti'$ , then the whole record steps..." In the last rule, the intention is that there should only be one field called  $i$ , and that all the other fields must contain values.

Typing:  $\Gamma \vdash t1 : T1 \dots \Gamma \vdash tn : Tn$

---

(T\_Rcd)  $\Gamma \vdash \{i1=t1, \dots, in=tn\} : \{i1:T1, \dots, in:Tn\}$   
 $\Gamma \vdash t : \{\dots, i:Ti, \dots\}$

---

(T\_Proj)  $\Gamma \vdash t.i : Ti$

---

## Encoding Records (Optional)

There are several ways to make the above definitions precise.

- We can directly formalize the syntactic forms and inference rules, staying as close as possible to the form we've given them above. This is conceptually straightforward, and it's probably what we'd want to do if we were building a real compiler – in particular, it will allow us to print error messages in the form that programmers will find easy to understand. But the formal versions of the rules will not be pretty at all!
- We could look for a smoother way of presenting records – for example, a binary presentation with one constructor for the empty record and another constructor for adding a single field to an existing record, instead of a single monolithic constructor that builds a whole record at once. This is the right way to go if we are primarily interested in studying the metatheory of the calculi with records, since it leads to clean and elegant definitions and proofs. Chapter **Records** shows how this can be done.
- Alternatively, if we like, we can avoid formalizing records altogether, by stipulating that record notations are just informal shorthands for more complex expressions involving pairs and product types. We sketch this approach here.

First, observe that we can encode arbitrary-size tuples using nested pairs and the **unit** value. To avoid overloading the pair notation  $(t1, t2)$ , we'll use curly braces without labels to write down tuples, so  $\{\}$  is the empty tuple,  $\{5\}$  is a singleton tuple,  $\{5,6\}$  is a 2-tuple (morally the same as a pair),  $\{5,6,7\}$  is a triple, etc.

```

{}          ---->  unit
{t1, t2, ..., tn} ----> (t1, trest)
                        where {t2, ..., tn} ----> trest

```

Similarly, we can encode tuple types using nested product types:

```

{}          ---->  Unit
{T1, T2, ..., Tn} ----> T1 * TRest
                        where {T2, ..., Tn} ----> TRest

```

The operation of projecting a field from a tuple can be encoded using a sequence of second projections followed by a first projection:

```

t.0          ---->  t.fst
t.(n+1)      ---->  (t.snd).n

```

Next, suppose that there is some total ordering on record labels, so that we can associate each label with a unique natural number. This number is called the *position* of the label. For example, we might assign positions like this:

LABEL	POSITION
a	0
b	1
c	2
...	...
foo	1004
...	...
bar	10562
...	...

We use these positions to encode record values as tuples (i.e., as nested pairs) by sorting the fields according to their positions. For example:

```

{a=5, b=6}      ---->  {5,6}
{a=5, c=7}      ---->  {5,unit,7}
{c=7, a=5}      ---->  {5,unit,7}
{c=5, b=3}      ---->  {unit,3,5}
{f=8,c=5,a=7}   ---->  {7,unit,5,unit,unit,8}
{f=8,c=5}       ---->  {unit,unit,5,unit,unit,8}

```

Note that each field appears in the position associated with its label, that the size of the tuple is determined by the label with the highest position, and that we fill in unused positions with **unit**.

We do exactly the same thing with record types:

<code>{a:Nat, b:Nat}</code>	<code>----&gt;</code>	<code>{Nat,Nat}</code>
<code>{c:Nat, a:Nat}</code>	<code>----&gt;</code>	<code>{Nat,Unit,Nat}</code>
<code>{f:Nat,c:Nat}</code>	<code>----&gt;</code>	<code>{Unit,Unit,Nat,Unit,Unit,Nat}</code>

Finally, record projection is encoded as a tuple projection from the appropriate position:

```
t.l ----> t.(position of l)
```

It is not hard to check that all the typing rules for the original "direct" presentation of records are validated by this encoding. (The reduction rules are "almost validated" – not quite, because the encoding reorders fields.)

Of course, this encoding will not be very efficient if we happen to use a record with label **bar**! But things are not actually as bad as they might seem: for example, if we assume that our compiler can see the whole program at the same time, we can *choose* the numbering of labels so that we assign small positions to the most frequently used labels. Indeed, there are industrial compilers that essentially do this!

## Variants (Optional Reading)

Just as products can be generalized to records, sums can be generalized to n-ary labeled types called *variants*. Instead of  $T1 + T2$ , we can write something like  $\langle l1:T1, l2:T2, \dots, ln:Tn \rangle$  where  $l1, l2, \dots$  are field labels which are used both to build instances and as case arm labels.

These n-ary variants give us almost enough mechanism to build arbitrary inductive data types like lists and trees from scratch – the only thing missing is a way to allow *recursion* in type definitions. We won't cover this here, but detailed treatments can be found in many textbooks – e.g., *Types and Programming Languages*.

## 27.3 Exercise: Formalizing the Extensions

**Exercise: 4 stars, optional (STLC\_extensions)** In this problem you will formalize a couple of the extensions described above. We've provided the necessary additions to the syntax of terms and types, and we've included a few examples that you can test your definitions with to make sure they are working as expected. You'll fill in the rest of the definitions and extend all the proofs accordingly.

To get you started, we've provided implementations for:

- numbers
- pairs and units

- sums
- lists

You need to complete the implementations for:

- `let` (which involves binding)
- `fix`

A good strategy is to work on the extensions one at a time, in multiple passes, rather than trying to work through the file from start to finish in a single pass. For each definition or proof, begin by reading carefully through the parts that are provided for you, referring to the text in the `Stlc` chapter for high-level intuitions and the embedded comments for detailed mechanics.

Module `STLCExtended`.

## Syntax and Operational Semantics

```
Inductive ty : Type :=
| TArrow : ty → ty → ty
| TNat : ty
| TUnit : ty
| TProd : ty → ty → ty
| TSum : ty → ty → ty
| TList : ty → ty.
```

```
Inductive tm : Type :=

| tvar : id → tm
| tapp : tm → tm → tm
| tabs : id → ty → tm → tm

| tnat : nat → tm
| tsucc : tm → tm
| tpred : tm → tm
| tmult : tm → tm → tm
| tif0 : tm → tm → tm → tm

| tpair : tm → tm → tm
| tfst : tm → tm
| tsnd : tm → tm

| tunit : tm
```

| tlet : **id** → **tm** → **tm** → **tm**

| tinl : **ty** → **tm** → **tm**

| tinr : **ty** → **tm** → **tm**

| tcase : **tm** → **id** → **tm** → **id** → **tm** → **tm**

| tnil : **ty** → **tm**

| tcons : **tm** → **tm** → **tm**

| tlcas : **tm** → **tm** → **id** → **id** → **tm** → **tm**

| tfix : **tm** → **tm**.

Note that, for brevity, we've omitted booleans and instead provided a single *if0* form combining a zero test and a conditional. That is, instead of writing

if  $x = 0$  then ... else ...

we'll write this:

if0  $x$  then ... else ...

## Substitution

Fixpoint subst ( $x:\mathbf{id}$ ) ( $s:\mathbf{tm}$ ) ( $t:\mathbf{tm}$ ) : **tm** :=  
 match  $t$  with  
 | tvar  $y \Rightarrow$   
 if beq\_id  $x y$  then  $s$  else  $t$   
 | tabs  $y T t1 \Rightarrow$   
 tabs  $y T$  (if beq\_id  $x y$  then  $t1$  else (subst  $x s t1$ ))  
 | tapp  $t1 t2 \Rightarrow$   
 tapp (subst  $x s t1$ ) (subst  $x s t2$ )  
 | tnat  $n \Rightarrow$   
 tnat  $n$   
 | tsucc  $t1 \Rightarrow$   
 tsucc (subst  $x s t1$ )  
 | tpred  $t1 \Rightarrow$   
 tpred (subst  $x s t1$ )  
 | tmult  $t1 t2 \Rightarrow$   
 tmult (subst  $x s t1$ ) (subst  $x s t2$ )  
 | tif0  $t1 t2 t3 \Rightarrow$

```

      tif0 (subst x s t1) (subst x s t2) (subst x s t3)
| tpair t1 t2 ⇒
      tpair (subst x s t1) (subst x s t2)
| tfst t1 ⇒
      tfst (subst x s t1)
| tsnd t1 ⇒
      tsnd (subst x s t1)
| tunit ⇒ tunit

| tinl T t1 ⇒
      tinl T (subst x s t1)
| tinr T t1 ⇒
      tinr T (subst x s t1)
| tcase t0 y1 t1 y2 t2 ⇒
      tcase (subst x s t0)
            y1 (if beq_id x y1 then t1 else (subst x s t1))
            y2 (if beq_id x y2 then t2 else (subst x s t2))
| tnil T ⇒
      tnil T
| tcons t1 t2 ⇒
      tcons (subst x s t1) (subst x s t2)
| tlcas t1 t2 y1 y2 t3 ⇒
      tlcas (subst x s t1) (subst x s t2) y1 y2
            (if beq_id x y1 then
              t3
            else if beq_id x y2 then t3
              else (subst x s t3))

| _ ⇒ t
end.

```

Notation "'[ x ' := ' s ' ]' t" := (subst x s t) (at level 20).

## Reduction

Next we define the values of our language.

Inductive **value** : **tm** → Prop :=

```

| v_abs : ∀ x T11 t12,
  value (tabs x T11 t12)

| v_nat : ∀ n1,
  value (tnat n1)

```

```

| v_pair : ∀ v1 v2,
  value v1 →
  value v2 →
  value (tpair v1 v2)

| v_unit : value tunit

| v_inl : ∀ v T,
  value v →
  value (tinl T v)
| v_inr : ∀ v T,
  value v →
  value (tinr T v)

| v_lnil : ∀ T, value (tnil T)
| v_lcons : ∀ v1 vl,
  value v1 →
  value vl →
  value (tcons v1 vl)
.

```

Hint Constructors **value**.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

```

| ST_AppAbs : ∀ x T11 t12 v2,
  value v2 →
  (tapp (tabs x T11 t12) v2) ==> [x:=v2] t12
| ST_App1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  (tapp t1 t2) ==> (tapp t1' t2)
| ST_App2 : ∀ v1 t2 t2',
  value v1 →
  t2 ==> t2' →
  (tapp v1 t2) ==> (tapp v1 t2')

| ST_Succ1 : ∀ t1 t1',
  t1 ==> t1' →
  (tsucc t1) ==> (tsucc t1')
| ST_SuccNat : ∀ n1,
  (tsucc (tnat n1)) ==> (tnat (S n1))
| ST_Pred : ∀ t1 t1',
  t1 ==> t1' →
  (tpred t1) ==> (tpred t1')

```



```

| ST_PredNat :  $\forall n1,$ 
  (tpred (tnat  $n1$ )) ==> (tnat (pred  $n1$ ))
| ST_Mult1 :  $\forall t1\ t1'\ t2,$ 
   $t1 ==> t1' \rightarrow$ 
  (tmult  $t1\ t2$ ) ==> (tmult  $t1'\ t2$ )
| ST_Mult2 :  $\forall v1\ t2\ t2',$ 
  value  $v1 \rightarrow$ 
   $t2 ==> t2' \rightarrow$ 
  (tmult  $v1\ t2$ ) ==> (tmult  $v1\ t2'$ )
| ST_MultNats :  $\forall n1\ n2,$ 
  (tmult (tnat  $n1$ ) (tnat  $n2$ )) ==> (tnat (mult  $n1\ n2$ ))
| ST_If01 :  $\forall t1\ t1'\ t2\ t3,$ 
   $t1 ==> t1' \rightarrow$ 
  (tif0  $t1\ t2\ t3$ ) ==> (tif0  $t1'\ t2\ t3$ )
| ST_If0Zero :  $\forall t2\ t3,$ 
  (tif0 (tnat 0)  $t2\ t3$ ) ==>  $t2$ 
| ST_If0Nonzero :  $\forall n\ t2\ t3,$ 
  (tif0 (tnat (S  $n$ ))  $t2\ t3$ ) ==>  $t3$ 

| ST_Pair1 :  $\forall t1\ t1'\ t2,$ 
   $t1 ==> t1' \rightarrow$ 
  (tpair  $t1\ t2$ ) ==> (tpair  $t1'\ t2$ )
| ST_Pair2 :  $\forall v1\ t2\ t2',$ 
  value  $v1 \rightarrow$ 
   $t2 ==> t2' \rightarrow$ 
  (tpair  $v1\ t2$ ) ==> (tpair  $v1\ t2'$ )
| ST_Fst1 :  $\forall t1\ t1',$ 
   $t1 ==> t1' \rightarrow$ 
  (tfst  $t1$ ) ==> (tfst  $t1'$ )
| ST_FstPair :  $\forall v1\ v2,$ 
  value  $v1 \rightarrow$ 
  value  $v2 \rightarrow$ 
  (tfst (tpair  $v1\ v2$ )) ==>  $v1$ 
| ST_Snd1 :  $\forall t1\ t1',$ 
   $t1 ==> t1' \rightarrow$ 
  (tsnd  $t1$ ) ==> (tsnd  $t1'$ )
| ST_SndPair :  $\forall v1\ v2,$ 
  value  $v1 \rightarrow$ 
  value  $v2 \rightarrow$ 
  (tsnd (tpair  $v1\ v2$ )) ==>  $v2$ 

```

```

| ST_Inl :  $\forall t1\ t1'\ T,$ 
   $t1 \implies t1' \rightarrow$ 
   $(\text{tinl } T\ t1) \implies (\text{tinl } T\ t1')$ 
| ST_Inr :  $\forall t1\ t1'\ T,$ 
   $t1 \implies t1' \rightarrow$ 
   $(\text{tinr } T\ t1) \implies (\text{tinr } T\ t1')$ 
| ST_Case :  $\forall t0\ t0'\ x1\ t1\ x2\ t2,$ 
   $t0 \implies t0' \rightarrow$ 
   $(\text{tcase } t0\ x1\ t1\ x2\ t2) \implies (\text{tcase } t0'\ x1\ t1\ x2\ t2)$ 
| ST_CaseInl :  $\forall v0\ x1\ t1\ x2\ t2\ T,$ 
  value  $v0 \rightarrow$ 
   $(\text{tcase } (\text{tinl } T\ v0)\ x1\ t1\ x2\ t2) \implies [x1 := v0] t1$ 
| ST_CaseInr :  $\forall v0\ x1\ t1\ x2\ t2\ T,$ 
  value  $v0 \rightarrow$ 
   $(\text{tcase } (\text{tinr } T\ v0)\ x1\ t1\ x2\ t2) \implies [x2 := v0] t2$ 

| ST_Cons1 :  $\forall t1\ t1'\ t2,$ 
   $t1 \implies t1' \rightarrow$ 
   $(\text{tcons } t1\ t2) \implies (\text{tcons } t1'\ t2)$ 
| ST_Cons2 :  $\forall v1\ t2\ t2',$ 
  value  $v1 \rightarrow$ 
   $t2 \implies t2' \rightarrow$ 
   $(\text{tcons } v1\ t2) \implies (\text{tcons } v1\ t2')$ 
| ST_Lcase1 :  $\forall t1\ t1'\ t2\ x1\ x2\ t3,$ 
   $t1 \implies t1' \rightarrow$ 
   $(\text{tlcase } t1\ t2\ x1\ x2\ t3) \implies (\text{tlcase } t1'\ t2\ x1\ x2\ t3)$ 
| ST_LcaseNil :  $\forall T\ t2\ x1\ x2\ t3,$ 
   $(\text{tlcase } (\text{tnil } T)\ t2\ x1\ x2\ t3) \implies t2$ 
| ST_LcaseCons :  $\forall v1\ vl\ t2\ x1\ x2\ t3,$ 
  value  $v1 \rightarrow$ 
  value  $vl \rightarrow$ 
   $(\text{tlcase } (\text{tcons } v1\ vl)\ t2\ x1\ x2\ t3) \implies (\text{subst } x2\ vl\ (\text{subst } x1\ v1\ t3))$ 

```

where " $t1 \implies t2$ " := (**step**  $t1\ t2$ ).

Notation **multistep** := (**multi step**).

Notation " $t1 \implies^* t2$ " := (**multistep**  $t1\ t2$ ) (at level 40).

Hint Constructors **step**.

## Typing

Definition context := partial\_map ty.

Next we define the typing rules. These are nearly direct transcriptions of the inference rules shown above.

Reserved Notation "Gamma '⊢' t 'in' T" (at level 40).

Inductive has\_type : context → tm → ty → Prop :=

- | T\_Var : ∀ Gamma x T,  
Gamma x = Some T →  
Gamma ⊢ (tvar x) \in T
- | T\_Abs : ∀ Gamma x T11 T12 t12,  
(update Gamma x T11) ⊢ t12 \in T12 →  
Gamma ⊢ (tabs x T11 t12) \in (TArrow T11 T12)
- | T\_App : ∀ T1 T2 Gamma t1 t2,  
Gamma ⊢ t1 \in (TArrow T1 T2) →  
Gamma ⊢ t2 \in T1 →  
Gamma ⊢ (tapp t1 t2) \in T2
  
- | T\_Nat : ∀ Gamma n1,  
Gamma ⊢ (tnat n1) \in TNat
- | T\_Succ : ∀ Gamma t1,  
Gamma ⊢ t1 \in TNat →  
Gamma ⊢ (tsucc t1) \in TNat
- | T\_Pred : ∀ Gamma t1,  
Gamma ⊢ t1 \in TNat →  
Gamma ⊢ (tpred t1) \in TNat
- | T\_Mult : ∀ Gamma t1 t2,  
Gamma ⊢ t1 \in TNat →  
Gamma ⊢ t2 \in TNat →  
Gamma ⊢ (tmult t1 t2) \in TNat
- | T\_If0 : ∀ Gamma t1 t2 t3 T1,  
Gamma ⊢ t1 \in TNat →  
Gamma ⊢ t2 \in T1 →  
Gamma ⊢ t3 \in T1 →  
Gamma ⊢ (tif0 t1 t2 t3) \in T1
  
- | T\_Pair : ∀ Gamma t1 t2 T1 T2,  
Gamma ⊢ t1 \in T1 →  
Gamma ⊢ t2 \in T2 →  
Gamma ⊢ (tpair t1 t2) \in (TProd T1 T2)
- | T\_Fst : ∀ Gamma t T1 T2,

```

      Gamma ⊢ t \in (TProd T1 T2) →
      Gamma ⊢ (tfst t) \in T1
| T_Snd : ∀ Gamma t T1 T2,
      Gamma ⊢ t \in (TProd T1 T2) →
      Gamma ⊢ (tsnd t) \in T2

| T_Unit : ∀ Gamma,
      Gamma ⊢ tunit \in TUnit

| T_Inl : ∀ Gamma t1 T1 T2,
      Gamma ⊢ t1 \in T1 →
      Gamma ⊢ (tinl T2 t1) \in (TSum T1 T2)
| T_Inr : ∀ Gamma t2 T1 T2,
      Gamma ⊢ t2 \in T2 →
      Gamma ⊢ (tinr T1 t2) \in (TSum T1 T2)
| T_Case : ∀ Gamma t0 x1 T1 t1 x2 T2 t2 T,
      Gamma ⊢ t0 \in (TSum T1 T2) →
      (update Gamma x1 T1) ⊢ t1 \in T →
      (update Gamma x2 T2) ⊢ t2 \in T →
      Gamma ⊢ (tcase t0 x1 t1 x2 t2) \in T

| T_Nil : ∀ Gamma T,
      Gamma ⊢ (tnil T) \in (TList T)
| T_Cons : ∀ Gamma t1 t2 T1,
      Gamma ⊢ t1 \in T1 →
      Gamma ⊢ t2 \in (TList T1) →
      Gamma ⊢ (tcons t1 t2) \in (TList T1)
| T_Lcase : ∀ Gamma t1 T1 t2 x1 x2 t3 T2,
      Gamma ⊢ t1 \in (TList T1) →
      Gamma ⊢ t2 \in T2 →
      (update (update Gamma x2 (TList T1)) x1 T1) ⊢ t3 \in T2 →
      Gamma ⊢ (tlcase t1 t2 x1 x2 t3) \in T2

```

where "Gamma |- t \in T" := (**has\_type** Gamma t T).

Hint Constructors **has\_type**.

### 27.3.1 Examples

This section presents formalized versions of the examples from above (plus several more). The ones at the beginning focus on specific features; you can use these to make sure your definition of a given feature is reasonable before moving on to extending the proofs later in the file with the cases relating to this feature. The later examples require all the features together, so you'll need to come back to these when you've got all the definitions filled in.

Module EXAMPLES.

#### Preliminaries

First, let's define a few variable names:

```
Notation a := (ld 0).
Notation f := (ld 1).
Notation g := (ld 2).
Notation l := (ld 3).
Notation k := (ld 6).
Notation i1 := (ld 7).
Notation i2 := (ld 8).
Notation x := (ld 9).
Notation y := (ld 10).
Notation processSum := (ld 11).
Notation n := (ld 12).
Notation eq := (ld 13).
Notation m := (ld 14).
Notation evenodd := (ld 15).
Notation even := (ld 16).
Notation odd := (ld 17).
Notation eo := (ld 18).
```

Next, a bit of Coq hackery to automate searching for typing derivations. You don't need to understand this bit in detail – just have a look over it so that you'll know what to look for if you ever find yourself needing to make custom extensions to `auto`.

The following `Hint` declarations say that, whenever `auto` arrives at a goal of the form  $(\Gamma \vdash (\text{tapp } e1 \ e2) \text{ in } T)$ , it should consider `eapply T_App`, leaving an existential variable for the middle type `T1`, and similar for `lcase`. That variable will then be filled in during the search for type derivations for `e1` and `e2`. We also include a hint to "try harder" when solving equality goals; this is useful to automate uses of `T_Var` (which includes an equality as a precondition).

```
Hint Extern 2 (has_type _ (tapp _ _) _) =>
  eapply T_App; auto.
Hint Extern 2 (_ = _) => compute; reflexivity.
```

## Numbers

Module NUMTEST.

Definition test :=

```
tif0
  (tpred
    (tsucc
      (tpred
        (tmult
          (tnat 2)
          (tnat 0))))))
  (tnat 5)
  (tnat 6).
```

Remove the comment braces once you've implemented enough of the definitions that you think this should work.

End NUMTEST.

## Products

Module PRODTEST.

Definition test :=

```
tsnd
  (tfst
    (tpair
      (tpair
        (tnat 5)
        (tnat 6))
      (tnat 7))).
```

End PRODTEST.

let

Module LETTEST.

Definition test :=

```
tlet
  x
  (tpred (tnat 6))
  (tsucc (tvar x)).
```

End LETTEST.

## Sums

Module SUMTEST1.

Definition test :=  
 tcase (tinl TNat (tnat 5))  
 x (tvar x)  
 y (tvar y).

End SUMTEST1.

Module SUMTEST2.

Definition test :=  
 tlet  
 processSum  
 (tabs x (TSum TNat TNat)  
 (tcase (tvar x)  
 n (tvar n)  
 n (tif0 (tvar n) (tnat 1) (tnat 0))))  
 (tpair  
 (tapp (tvar processSum) (tinl TNat (tnat 5)))  
 (tapp (tvar processSum) (tinr TNat (tnat 5)))).

End SUMTEST2.

## Lists

Module LISTTEST.

Definition test :=  
 tlet l  
 (tcons (tnat 5) (tcons (tnat 6) (tnil TNat)))  
 (tlcase (tvar l)  
 (tnat 0)  
 x y (tmult (tvar x) (tvar y))).

End LISTTEST.

fix

Module FIXTEST1.

Definition fact :=  
 tfix  
 (tabs f (TArrow TNat TNat)

```

(tabs a TNat
  (tif0
    (tvar a)
    (tnat 1)
    (tmult
      (tvar a)
      (tapp (tvar f) (tpred (tvar a))))))).

```

(Warning: you may be able to typecheck fact but still have some rules wrong!)

End FIXTEST1.

Module FIXTEST2.

Definition map :=

```

  tabs g (TArrow TNat TNat)
  (tfix
    (tabs f (TArrow (TList TNat) (TList TNat))
      (tabs l (TList TNat)
        (tlcase (tvar l)
          (tnil TNat)
          a l (tcons (tapp (tvar g) (tvar a))
            (tapp (tvar f) (tvar l))))))).

```

End FIXTEST2.

Module FIXTEST3.

Definition equal :=

```

  tfix
  (tabs eq (TArrow TNat (TArrow TNat TNat))
    (tabs m TNat
      (tabs n TNat
        (tif0 (tvar m)
          (tif0 (tvar n) (tnat 1) (tnat 0))
          (tif0 (tvar n)
            (tnat 0)
            (tapp (tapp (tvar eq)
              (tpred (tvar m)))
              (tpred (tvar n))))))).

```

End FIXTEST3.

Module FIXTEST4.

Definition eotest :=

```

  tlet evenodd

```



```

(tfix
  (tabs eo (TProd (TArrow TNat TNat) (TArrow TNat TNat))
    (tpair
      (tabs n TNat
        (tif0 (tvar n)
          (tnat 1)
          (tapp (tsnd (tvar eo)) (tpred (tvar n))))))
      (tabs n TNat
        (tif0 (tvar n)
          (tnat 0)
          (tapp (tfst (tvar eo)) (tpred (tvar n))))))))
  (tlet even (tfst (tvar evenodd))
    (tlet odd (tsnd (tvar evenodd))
      (tpair
        (tapp (tvar even) (tnat 3))
        (tapp (tvar even) (tnat 4)))))).

```

End FIXTEST4.

End EXAMPLES.

### 27.3.2 Properties of Typing

The proofs of progress and preservation for this system are essentially the same (though of course somewhat longer) as for the pure simply typed lambda-calculus.

#### Progress

Theorem progress :  $\forall t \ T,$   
 $\text{empty} \vdash t \ \text{in } T \rightarrow$   
 $\text{value } t \vee \exists t', t \Rightarrow t'.$

Proof with eauto.

```

intros t T Ht.
remember (@empty ty) as Gamma.
generalize dependent HeqGamma.
induction Ht; intros HeqGamma; subst.
-
  inversion H.
-
  left...
-

```

```

right.
destruct IHHt1; subst...
+
  destruct IHHt2; subst...
  ×

  inversion H; subst; try (solve by inversion).
  ∃ (subst x t2 t12)...
  ×

  inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
+
  inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...
-
left...
-
right.
destruct IHHt...
+
  inversion H; subst; try solve by inversion.
  ∃ (tnat (S n1))...
+
  inversion H as [t1' Hstp].
  ∃ (tsucc t1')...
-
right.
destruct IHHt...
+
  inversion H; subst; try solve by inversion.
  ∃ (tnat (pred n1))...
+
  inversion H as [t1' Hstp].
  ∃ (tpred t1')...
-
right.
destruct IHHt1...
+
  destruct IHHt2...
  ×
  inversion H; subst; try solve by inversion.

```

```

    inversion H0; subst; try solve by inversion.
    ∃ (tnat (mult n1 n0))...
  ×
    inversion H0 as [t2' Hstp].
    ∃ (tmult t1 t2')...
+
  inversion H as [t1' Hstp].
  ∃ (tmult t1' t2)...
-
right.
destruct IHHt1...
+
  inversion H; subst; try solve by inversion.
  destruct n1 as [|n1'].
  ×
    ∃ t2...
  ×
    ∃ t3...
+
  inversion H as [t1' H0].
  ∃ (tif0 t1' t2 t3)...
-
destruct IHHt1...
+
  destruct IHHt2...
  ×
    right. inversion H0 as [t2' Hstp].
    ∃ (tpair t1 t2')...
+
  right. inversion H as [t1' Hstp].
  ∃ (tpair t1' t2)...
-
right.
destruct IHHt...
+
  inversion H; subst; try solve by inversion.
  ∃ v1...
+
  inversion H as [t1' Hstp].
  ∃ (tfst t1')...
-
right.

```

```

destruct IHHt...
+
  inversion H; subst; try solve by inversion.
  ∃ v2...
+
  inversion H as [t1' Hstp].
  ∃ (tsnd t1')...
-
left...
-
destruct IHHt...
+
  right. inversion H as [t1' Hstp]...
-
destruct IHHt...
+
  right. inversion H as [t1' Hstp]...
-
right.
destruct IHHt1...
+
  inversion H; subst; try solve by inversion.
  ×
  ∃ ([x1 := v] t1)...
  ×
  ∃ ([x2 := v] t2)...
+
  inversion H as [t0' Hstp].
  ∃ (tcase t0' x1 t1 x2 t2)...
-
left...
-
destruct IHHt1...
+
  destruct IHHt2...
  ×
  right. inversion H0 as [t2' Hstp].
  ∃ (tcons t1 t2')...
+
  right. inversion H as [t1' Hstp].
  ∃ (tcons t1' t2)...
-

```

```

right.
destruct IHHt1...
+
  inversion H; subst; try solve by inversion.
  ×
    ∃ t2...
  ×
    ∃ ([x2:=v1] ([x1:=v1] t3))...
+
  inversion H as [t1' Hstp].
  ∃ (tlcase t1' t2 x1 x2 t3)...
Qed.

```

## Context Invariance

```

Inductive appears_free_in : id → tm → Prop :=
| afi_var : ∀ x,
  appears_free_in x (tvar x)
| afi_app1 : ∀ x t1 t2,
  appears_free_in x t1 → appears_free_in x (tapp t1 t2)
| afi_app2 : ∀ x t1 t2,
  appears_free_in x t2 → appears_free_in x (tapp t1 t2)
| afi_abs : ∀ x y T11 t12,
  y ≠ x →
  appears_free_in x t12 →
  appears_free_in x (tabs y T11 t12)

| afi_succ : ∀ x t,
  appears_free_in x t →
  appears_free_in x (tsucc t)
| afi_pred : ∀ x t,
  appears_free_in x t →
  appears_free_in x (tpred t)
| afi_mult1 : ∀ x t1 t2,
  appears_free_in x t1 →
  appears_free_in x (tmult t1 t2)
| afi_mult2 : ∀ x t1 t2,
  appears_free_in x t2 →
  appears_free_in x (tmult t1 t2)
| afi_if01 : ∀ x t1 t2 t3,
  appears_free_in x t1 →
  appears_free_in x (tif0 t1 t2 t3)
| afi_if02 : ∀ x t1 t2 t3,

```

```

    appears_free_in x t2 →
    appears_free_in x (tif0 t1 t2 t3)
| afi_if03 : ∀ x t1 t2 t3,
    appears_free_in x t3 →
    appears_free_in x (tif0 t1 t2 t3)

| afi_pair1 : ∀ x t1 t2,
    appears_free_in x t1 →
    appears_free_in x (tpair t1 t2)
| afi_pair2 : ∀ x t1 t2,
    appears_free_in x t2 →
    appears_free_in x (tpair t1 t2)
| afi_fst : ∀ x t,
    appears_free_in x t →
    appears_free_in x (tfst t)
| afi_snd : ∀ x t,
    appears_free_in x t →
    appears_free_in x (tsnd t)

| afi_inl : ∀ x t T,
    appears_free_in x t →
    appears_free_in x (tinl T t)
| afi_inr : ∀ x t T,
    appears_free_in x t →
    appears_free_in x (tinr T t)
| afi_case0 : ∀ x t0 x1 t1 x2 t2,
    appears_free_in x t0 →
    appears_free_in x (tcase t0 x1 t1 x2 t2)
| afi_case1 : ∀ x t0 x1 t1 x2 t2,
    x1 ≠ x →
    appears_free_in x t1 →
    appears_free_in x (tcase t0 x1 t1 x2 t2)
| afi_case2 : ∀ x t0 x1 t1 x2 t2,
    x2 ≠ x →
    appears_free_in x t2 →
    appears_free_in x (tcase t0 x1 t1 x2 t2)

| afi_cons1 : ∀ x t1 t2,
    appears_free_in x t1 →
    appears_free_in x (tcons t1 t2)

```

```

| afi_cons2 : ∀ x t1 t2,
  appears_free_in x t2 →
  appears_free_in x (tcons t1 t2)
| afi_lcase1 : ∀ x t1 t2 y1 y2 t3,
  appears_free_in x t1 →
  appears_free_in x (tlcase t1 t2 y1 y2 t3)
| afi_lcase2 : ∀ x t1 t2 y1 y2 t3,
  appears_free_in x t2 →
  appears_free_in x (tlcase t1 t2 y1 y2 t3)
| afi_lcase3 : ∀ x t1 t2 y1 y2 t3,
  y1 ≠ x →
  y2 ≠ x →
  appears_free_in x t3 →
  appears_free_in x (tlcase t1 t2 y1 y2 t3)

```

.

Hint Constructors **appears\_free\_in**.

Lemma context\_invariance : ∀ *Gamma Gamma' t S*,  
   *Gamma* ⊢ *t* \in *S* →  
   (∀ *x*, **appears\_free\_in** *x t* → *Gamma x* = *Gamma' x*) →  
   *Gamma'* ⊢ *t* \in *S*.

Proof with eauto.

```

intros. generalize dependent Gamma'.
induction H;
  intros Gamma' Heqv...
-
  apply T_Var... rewrite ← Heqv...
-
  apply T_Abs... apply IHhas_type. intros y Hafi.
  unfold update, t_update.
  destruct (beq_idP x y)...
-
  apply T_Mult...
-
  apply T_If0...
-
  apply T_Pair...
-
  eapply T_Case...
+ apply IHhas_type2. intros y Hafi.
  unfold update, t_update.

```

```

    destruct (beq_idP x1 y)...
+ apply IHhas_type3. intros y Hafi.
  unfold update, t_update.
  destruct (beq_idP x2 y)...
-
  apply T_Cons...
-
  eapply T_Lcase... apply IHhas_type3. intros y Hafi.
  unfold update, t_update.
  destruct (beq_idP x1 y)...
  destruct (beq_idP x2 y)...
Qed.

Lemma free_in_context :  $\forall x t T \text{ Gamma},$ 
  appears_free_in  $x t \rightarrow$ 
   $\text{Gamma} \vdash t \setminus \text{in } T \rightarrow$ 
   $\exists T', \text{Gamma } x = \text{Some } T'.$ 
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  induction Htyp; inversion Hafi; subst...
-
  destruct IHHtyp as [T' Hctx]...  $\exists T'.$ 
  unfold update, t_update in Hctx.
  rewrite false_beq_id in Hctx...
-
  destruct IHHtyp2 as [T' Hctx]...  $\exists T'.$ 
  unfold update, t_update in Hctx. rewrite false_beq_id in Hctx...
-
  destruct IHHtyp3 as [T' Hctx]...  $\exists T'.$ 
  unfold update, t_update in Hctx. rewrite false_beq_id in Hctx...
-
  clear Htyp1 IHHtyp1 Htyp2 IHHtyp2.
  destruct IHHtyp3 as [T' Hctx]...  $\exists T'.$ 
  unfold update, t_update in Hctx.
  rewrite false_beq_id in Hctx...
  rewrite false_beq_id in Hctx...
Qed.

```

## Substitution

```

Lemma substitution_preserves_typing :  $\forall \text{Gamma } x U v t S,$ 
   $(\text{update } \text{Gamma } x U) \vdash t \setminus \text{in } S \rightarrow$ 
   $\text{empty} \vdash v \setminus \text{in } U \rightarrow$ 

```



```

     $\Gamma \vdash ([x:=v]t) \setminus \text{in } S.$ 
Proof with eauto.
  intros  $\Gamma$   $x$   $U$   $v$   $t$   $S$   $Htypt$   $Htypv$ .
  generalize dependent  $\Gamma$ . generalize dependent  $S$ .
  induction t;
    intros  $S$   $\Gamma$   $Htypt$ ; simpl; inversion  $Htypt$ ; subst...
-
  simpl. rename  $i$  into  $y$ .
  unfold update, t_update in  $H1$ .
  destruct (beq_idP  $x$   $y$ ).
+

  subst.
  inversion  $H1$ ; subst. clear  $H1$ .
  eapply context_invariance...
  intros  $x$   $Hcontra$ .
  destruct (free_in_context _ _  $S$  empty  $Hcontra$ ) as [ $T'$   $HT'$ ]...
  inversion  $HT'$ .
+

  apply T_Var...
-
  rename  $i$  into  $y$ . rename  $t$  into  $T11$ .
  apply T_Abs...
  destruct (beq_idP  $x$   $y$ ) as [ $Hxy|Hxy$ ].
+

  eapply context_invariance...
  subst.
  intros  $x$   $Hafi$ . unfold update, t_update.
  destruct (beq_id  $y$   $x$ )...
+

  apply  $IHt$ . eapply context_invariance...
  intros  $z$   $Hafi$ . unfold update, t_update.
  destruct (beq_idP  $y$   $z$ ) as [ $Hyx|Hyx$ ]...
  subst.
  rewrite false_beq_id...
-
  rename  $i$  into  $x1$ . rename  $i0$  into  $x2$ .
  eapply T_Case...
+

```

```

destruct (beq_idP x x1) as [Hxx1|Hxx1].
×
  eapply context_invariance...
  subst.
  intros z Hafi. unfold update, t_update.
  destruct (beq_id x1 z)...
×
  apply IHt2. eapply context_invariance...
  intros z Hafi. unfold update, t_update.
  destruct (beq_idP x1 z) as [Hx1z|Hx1z]...
  subst. rewrite false_beq_id...
+
destruct (beq_idP x x2) as [Hxx2|Hxx2].
×
  eapply context_invariance...
  subst.
  intros z Hafi. unfold update, t_update.
  destruct (beq_id x2 z)...
×
  apply IHt3. eapply context_invariance...
  intros z Hafi. unfold update, t_update.
  destruct (beq_idP x2 z)...
  subst. rewrite false_beq_id...
-
rename i into y1. rename i0 into y2.
eapply T_Lcase...
destruct (beq_idP x y1).
+
  simpl.
  eapply context_invariance...
  subst.
  intros z Hafi. unfold update, t_update.
  destruct (beq_idP y1 z)...
+
destruct (beq_idP x y2).
×
  eapply context_invariance...
  subst.
  intros z Hafi. unfold update, t_update.
  destruct (beq_idP y2 z)...
×
  apply IHt3. eapply context_invariance...

```

```

    intros z Hafi. unfold update, t_update.
    destruct (beq_idP y1 z)...
    subst. rewrite false_beq_id...
    destruct (beq_idP y2 z)...
    subst. rewrite false_beq_id...

```

Qed.

## Preservation

Theorem preservation :  $\forall t t' T,$

```

    empty  $\vdash t \setminus \text{in } T \rightarrow$ 
     $t \Rightarrow t' \rightarrow$ 
    empty  $\vdash t' \setminus \text{in } T.$ 

```

Proof with eauto.

```

    intros t t' T HT.
    remember (@empty ty) as Gamma. generalize dependent HeqGamma.
    generalize dependent t'.
    induction HT;
    intros t' HeqGamma HE; subst; inversion HE; subst...
-
    inversion HE; subst...
+
    apply substitution_preserves_typing with T1...
    inversion HT1...
-
    inversion HT...
-
    inversion HT...
-
    inversion HT1; subst.
    eapply substitution_preserves_typing...
-
    inversion HT1; subst.
    eapply substitution_preserves_typing...
+
    inversion HT1; subst.
    apply substitution_preserves_typing with (TList T1)...
    apply substitution_preserves_typing with T1...

```

Qed.

□

End STLCEXTENDED.

# Chapter 28

## Library Sub

### 28.1 Sub: Subtyping

```
Require Import Sflib.  
Require Import Maps.  
Require Import Types.
```

### 28.2 Concepts

We now turn to the study of *subtyping*, perhaps the most characteristic feature of the static type systems of recently designed programming languages and a key feature needed to support the object-oriented programming style.

#### 28.2.1 A Motivating Example

Suppose we are writing a program involving two record types defined as follows:

```
Person  = {name:String, age:Nat}  
Student = {name:String, age:Nat, gpa:Nat}
```

In the simply typed lamdba-calculus with records, the term

```
(\r:Person. (r.age)+1) {name="Pat",age=21,gpa=1}
```

is not typable: it involves an application of a function that wants a one-field record to an argument that actually provides two fields, while the **T\_App** rule demands that the domain type of the function being applied must match the type of the argument precisely.

But this is silly: we're passing the function a *better* argument than it needs! The only thing the body of the function can possibly do with its record argument *r* is project the field *age* from it: nothing else is allowed by the type, and the presence or absence of an extra *gpa* field makes no difference at all. So, intuitively, it seems that this function should be applicable to any record value that has at least an *age* field.

Looking at the same thing from another point of view, a record with more fields is "at least as good in any context" as one with just a subset of these fields, in the sense that any value belonging to the longer record type can be used *safely* in any context expecting the shorter record type. If the context expects something with the shorter type but we actually give it something with the longer type, nothing bad will happen (formally, the program will not get stuck).

The general principle at work here is called *subtyping*. We say that "S is a subtype of T", informally written  $S <: T$ , if a value of type S can safely be used in any context where a value of type T is expected. The idea of subtyping applies not only to records, but to all of the type constructors in the language – functions, pairs, etc.

## 28.2.2 Subtyping and Object-Oriented Languages

Subtyping plays a fundamental role in many programming languages – in particular, it is closely related to the notion of *subclassing* in object-oriented languages.

An *object* in Java, C#, etc. can be thought of as a record, some of whose fields are functions ("methods") and some of whose fields are data values ("fields" or "instance variables"). Invoking a method *m* of an object *o* on some arguments *a1..an* consists of projecting out the *m* field of *o* and applying it to *a1..an*.

The type of an object can be given as either a *class* or an *interface*. Both of these provide a description of which methods and which data fields the object offers.

Classes and interfaces are related by the *subclass* and *subinterface* relations. An object belonging to a subclass (or subinterface) is required to provide all the methods and fields of one belonging to a superclass (or superinterface), plus possibly some more.

The fact that an object from a subclass (or sub-interface) can be used in place of one from a superclass (or super-interface) provides a degree of flexibility that is extremely handy for organizing complex libraries. For example, a GUI toolkit like Java's Swing framework might define an abstract interface *Component* that collects together the common fields and methods of all objects having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of such object would include the buttons, checkboxes, and scrollbars of a typical GUI. A method that relies only on this common interface can now be applied to any of these objects.

Of course, real object-oriented languages include many other features besides these. For example, fields can be updated. Fields and methods can be declared *private*. Classes also give *code* that is used when constructing objects and implementing their methods, and the code in subclasses cooperate with code in superclasses via *inheritance*. Classes can have static methods and fields, initializers, etc., etc.

To keep things simple here, we won't deal with any of these issues – in fact, we won't even talk any more about objects or classes. (There is a lot of discussion in *Types and Programming Languages*, if you are interested.) Instead, we'll study the core concepts behind the subclass / subinterface relation in the simplified setting of the STLC.

Of course, real OO languages have lots of other features...

- mutable fields
- *private* and other visibility modifiers
- method inheritance
- static components
- etc., etc.

We'll ignore all these and focus on core mechanisms.

### 28.2.3 The Subsumption Rule

Our goal for this chapter is to add subtyping to the simply typed lambda-calculus (with some of the basic extensions from **MoreStlc**). This involves two steps:

- Defining a binary *subtype relation* between types.
- Enriching the typing relation to take subtyping into account.

The second step is actually very simple. We add just a single rule to the typing relation: the so-called *rule of subsumption*:  $\Gamma \vdash t : S \quad S <: T$

---

(T\_Sub)  $\Gamma \vdash t : T$  This rule says, intuitively, that it is OK to "forget" some of what we know about a term. For example, we may know that  $t$  is a record with two fields (e.g.,  $S = \{x:A \rightarrow A, y:B \rightarrow B\}$ ), but choose to forget about one of the fields ( $T = \{y:B \rightarrow B\}$ ) so that we can pass  $t$  to a function that requires just a single-field record.

### 28.2.4 The Subtype Relation

The first step – the definition of the relation  $S <: T$  – is where all the action is. Let's look at each of the clauses of its definition.

#### Structural Rules

To start off, we impose two "structural rules" that are independent of any particular type constructor: a rule of *transitivity*, which says intuitively that, if  $S$  is better than  $U$  and  $U$  is better than  $T$ , then  $S$  is better than  $T$ ...  $S <: U \quad U <: T$

---

(S\_Trans)  $S <: T \dots$  and a rule of *reflexivity*, since certainly any type  $T$  is as good as itself:

---

(S\_Refl)  $T <: T$

## Products

Now we consider the individual type constructors, one by one, beginning with product types. We consider one pair to be "better than" another if each of its components is.  $S1 <: T1$   $S2 <: T2$

---

(S\_Prod)  $S1 * S2 <: T1 * T2$

## Arrows

Suppose we have two functions  $f$  and  $g$  with these types:  $f : C \rightarrow \text{Student}$   $g : (C \rightarrow \text{Person}) \rightarrow D$  That is,  $f$  is a function that yields a record of type **Student**, and  $g$  is a (higher-order) function that expects its (function) argument to yield a record of type **Person**. Also suppose, even though we haven't yet discussed subtyping for records, that **Student** is a subtype of **Person**. Then the application  $g\ f$  is safe even though their types do not match up precisely, because the only thing  $g$  can do with  $f$  is to apply it to some argument (of type  $C$ ); the result will actually be a **Student**, while  $g$  will be expecting a **Person**, but this is safe because the only thing  $g$  can then do is to project out the two fields that it knows about (*name* and *age*), and these will certainly be among the fields that are present.

This example suggests that the subtyping rule for arrow types should say that two arrow types are in the subtype relation if their results are:  $S2 <: T2$

---

(S\_Arrow\_Co)  $S1 \rightarrow S2 <: S1 \rightarrow T2$  We can generalize this to allow the arguments of the two arrow types to be in the subtype relation as well:  $T1 <: S1$   $S2 <: T2$

---

(S\_Arrow)  $S1 \rightarrow S2 <: T1 \rightarrow T2$  Notice that the argument types are subtypes "the other way round": in order to conclude that  $S1 \rightarrow S2$  to be a subtype of  $T1 \rightarrow T2$ , it must be the case that  $T1$  is a subtype of  $S1$ . The arrow constructor is said to be *contravariant* in its first argument and *covariant* in its second.

Here is an example that illustrates this:  $f : \text{Person} \rightarrow C$   $g : (\text{Student} \rightarrow C) \rightarrow D$  The application  $g\ f$  is safe, because the only thing the body of  $g$  can do with  $f$  is to apply it to some argument of type **Student**. Since  $f$  requires records having (at least) the fields of a **Person**, this will always work. So  $\text{Person} \rightarrow C$  is a subtype of  $\text{Student} \rightarrow C$  since **Student** is a subtype of **Person**.

The intuition is that, if we have a function  $f$  of type  $S1 \rightarrow S2$ , then we know that  $f$  accepts elements of type  $S1$ ; clearly,  $f$  will also accept elements of any subtype  $T1$  of  $S1$ . The type of  $f$  also tells us that it returns elements of type  $S2$ ; we can also view these results belonging to any supertype  $T2$  of  $S2$ . That is, any function  $f$  of type  $S1 \rightarrow S2$  can also be viewed as having type  $T1 \rightarrow T2$ .

## Records

What about subtyping for record types?



The basic intuition about subtyping for record types is that it is always safe to use a "bigger" record in place of a "smaller" one. That is, given a record type, adding extra fields will always result in a subtype. If some code is expecting a record with fields  $x$  and  $y$ , it is perfectly safe for it to receive a record with fields  $x$ ,  $y$ , and  $z$ ; the  $z$  field will simply be ignored. For example,  $\{\text{name:String, age:Nat, gpa:Nat}\} <: \{\text{name:String, age:Nat}\}$   $\{\text{name:String, age:Nat}\} <: \{\text{name:String}\}$   $\{\text{name:String}\} <: \{\}$  This is known as "width subtyping" for records.

We can also create a subtype of a record type by replacing the type of one of its fields with a subtype. If some code is expecting a record with a field  $x$  of type  $T$ , it will be happy with a record having a field  $x$  of type  $S$  as long as  $S$  is a subtype of  $T$ . For example,  $\{x:\text{Student}\} <: \{x:\text{Person}\}$  This is known as "depth subtyping".

Finally, although the fields of a record type are written in a particular order, the order does not really matter. For example,  $\{\text{name:String, age:Nat}\} <: \{\text{age:Nat, name:String}\}$  This is known as "permutation subtyping".

We could formalize these requirements in a single subtyping rule for records as follows: for each  $j_k$  in  $j_1..j_n$ , exists  $i_p$  in  $i_1..i_m$ , such that  $j_k = i_p$  and  $S_p <: T_k$

---

(S\_Rcd)  $\{i_1:S_1...i_m:S_m\} <: \{j_1:T_1...j_n:T_n\}$  That is, the record on the left should have all the field labels of the one on the right (and possibly more), while the types of the common fields should be in the subtype relation. However, this rule is rather heavy and hard to read. If we like, we can decompose it into three simpler rules, which can be combined using **S\_Trans** to achieve all the same effects.

First, adding fields to the end of a record type gives a subtype:  $n > m$

---

(S\_RcdWidth)  $\{i_1:T_1...i_n:T_n\} <: \{i_1:T_1...i_m:T_m\}$  We can use **S\_RcdWidth** to drop later fields of a multi-field record while keeping earlier fields, showing for example that  $\{\text{age:Nat, name:String}\} <: \{\text{name:String}\}$ .

Second, we can apply subtyping inside the components of a compound record type:  $S_1 <: T_1 \dots S_n <: T_n$

---

(S\_RcdDepth)  $\{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\}$  For example, we can use **S\_RcdDepth** and **S\_RcdWidth** together to show that  $\{y:\text{Student}, x:\text{Nat}\} <: \{y:\text{Person}\}$ .

Third, we need to be able to reorder fields. For example, we might expect that  $\{\text{name:String, gpa:Nat, age:Nat}\} <: \text{Person}$ . We haven't quite achieved this yet: using just **S\_RcdDepth** and **S\_RcdWidth** we can only drop fields from the *end* of a record type. So we need:  $\{i_1:S_1...i_n:S_n\}$  is a permutation of  $\{i_1:T_1...i_n:T_n\}$

---

(S\_RcdPerm)  $\{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\}$

It is worth noting that full-blown language designs may choose not to adopt all of these subtyping rules. For example, in Java:

- A subclass may not change the argument or result types of a method of its superclass (i.e., no depth subtyping or no arrow subtyping, depending how you look at it).

- Each class has just one superclass ("single inheritance" of classes).
- Each class member (field or method) can be assigned a single index, adding new indices "on the right" as more members are added in subclasses (i.e., no permutation for classes).
- A class may implement multiple interfaces – so-called "multiple inheritance" of interfaces (i.e., permutation is allowed for interfaces).

**Exercise: 2 stars, recommended (arrow\_sub\_wrong)** Suppose we had incorrectly defined subtyping as covariant on both the right and the left of arrow types:  $S1 <: T1 \rightarrow S2 <: T2$

---

(S\_Arrow\_wrong)  $S1 \rightarrow S2 <: T1 \rightarrow T2$  Give a concrete example of functions  $f$  and  $g$  with the following types...  $f : \text{Student} \rightarrow \text{Nat}$   $g : (\text{Person} \rightarrow \text{Nat}) \rightarrow \text{Nat}$  ... such that the application  $g f$  will get stuck during execution.

□

## Top

Finally, it is natural to give the subtype relation a maximum element – a type that lies above every other type and is inhabited by all (well-typed) values. We do this by adding to the language one new type constant, called *Top*, together with a subtyping rule that places it above every other type in the subtype relation:

---

(S\_Top)  $S <: \text{Top}$  The *Top* type is an analog of the *Object* type in Java and C#.

## Summary

In summary, we form the STLC with subtyping by starting with the pure STLC (over some set of base types) and...

- adding a base type *Top*,
- adding the rule of subsumption  $\Gamma \vdash t : S \quad S <: T$

– ————— (T\_Sub)  $\Gamma \vdash t : T$

to the typing relation, and

- defining a subtype relation as follows:  $S <: U \quad U <: T$

– ————— (S\_Trans)  $S <: T$

\* — (S\_Refl)

$$\begin{array}{l}
T <: T \\
* \longrightarrow (S\_Top) \\
S <: Top \\
\\
S1 <: T1 \quad S2 <: T2 \\
- \longrightarrow (S\_Prod) \quad S1 * S2 <: T1 * T2 \\
\\
T1 <: S1 \quad S2 <: T2 \\
- \longrightarrow (S\_Arrow) \\
\\
S1 \rightarrow S2 <: T1 \rightarrow T2 \\
n > m \\
- \longrightarrow (S\_RcdWidth) \\
\\
\{i1:T1...in:Tn\} <: \{i1:T1...im:Tm\} \\
S1 <: T1 \dots Sn <: Tn \\
- \longrightarrow (S\_RcdDepth) \\
\\
\{i1:S1...in:Sn\} <: \{i1:T1...in:Tn\} \\
\{i1:S1...in:Sn\} \text{ is a permutation of } \{i1:T1...in:Tn\} \\
- \longrightarrow (S\_RcdPerm) \quad \{i1:S1...in:Sn\} <: \{i1:T1...in:Tn\}
\end{array}$$

### 28.2.5 Exercises

**Exercise: 1 star, optional (subtype\_instances\_tf\_1)** Suppose we have types  $S$ ,  $T$ ,  $U$ , and  $V$  with  $S <: T$  and  $U <: V$ . Which of the following subtyping assertions are then true? Write *true* or *false* after each one. ( $A$ ,  $B$ , and  $C$  here are base types.)

- $T \rightarrow S <: T \rightarrow S$
- $Top \rightarrow U <: S \rightarrow Top$
- $(C \rightarrow C) \rightarrow (A \times B) <: (C \rightarrow C) \rightarrow (Top \times B)$
- $T \rightarrow T \rightarrow U <: S \rightarrow S \rightarrow V$
- $(T \rightarrow T) \rightarrow U <: (S \rightarrow S) \rightarrow V$
- $((T \rightarrow S) \rightarrow T) \rightarrow U <: ((S \rightarrow T) \rightarrow S) \rightarrow V$
- $S \times V <: T \times U$

□

**Exercise: 2 stars (subtype\_order)** The following types happen to form a linear order with respect to subtyping:

- $Top$
- $Top \rightarrow Student$
- $Student \rightarrow Person$
- $Student \rightarrow Top$
- $Person \rightarrow Student$

Write these types in order from the most specific to the most general.  
Where does the type  $Top \rightarrow Top \rightarrow Student$  fit into this order?

**Exercise: 1 star (subtype\_instances\_tf\_2)** Which of the following statements are true? Write *true* or *false* after each one. forall S T, S <: T -> S->S <: T->T

forall S, S <: A->A -> exists T, S = T->T /\ T <: A

forall S T1 T2, (S <: T1 -> T2) -> exists S1 S2, S = S1 -> S2 /\ T1 <: S1 /\ S2 <: T2

exists S, S <: S->S

exists S, S->S <: S

forall S T1 T2, S <: T1\*T2 -> exists S1 S2, S = S1\*S2 /\ S1 <: T1 /\ S2 <: T2  $\square$

**Exercise: 1 star (subtype\_concepts\_tf)** Which of the following statements are true, and which are false?

- There exists a type that is a supertype of every other type.
- There exists a type that is a subtype of every other type.
- There exists a pair type that is a supertype of every other pair type.
- There exists a pair type that is a subtype of every other pair type.
- There exists an arrow type that is a supertype of every other arrow type.
- There exists an arrow type that is a subtype of every other arrow type.
- There is an infinite descending chain of distinct types in the subtype relation—that is, an infinite sequence of types  $S_0, S_1$ , etc., such that all the  $S_i$ 's are different and each  $S_{i+1}$  is a subtype of  $S_i$ .
- There is an infinite *ascending* chain of distinct types in the subtype relation—that is, an infinite sequence of types  $S_0, S_1$ , etc., such that all the  $S_i$ 's are different and each  $S_{i+1}$  is a supertype of  $S_i$ .

$\square$

**Exercise: 2 stars (proper\_subtypes)** Is the following statement true or false? Briefly explain your answer.  $\text{forall } T, \sim(\text{exists } n, T = \text{Base } n) \rightarrow \text{exists } S, S <: T \wedge S <> T \parallel \square$

**Exercise: 2 stars (small\_large\_1)**

- What is the *smallest* type  $T$  ("smallest" in the subtype relation) that makes the following assertion true? (Assume we have *Unit* among the base types and **unit** as a constant of this type.)  $\text{empty} \vdash (\lambda p:T*\text{Top}. p.\text{fst}) ((\lambda z:A.z), \text{unit}) : A \rightarrow A$
- What is the *largest* type  $T$  that makes the same assertion true?

$\square$

**Exercise: 2 stars (small\_large\_2)**

- What is the *smallest* type  $T$  that makes the following assertion true?  $\text{empty} \vdash (\lambda p:(A \rightarrow A * B \rightarrow B). p) ((\lambda z:A.z), (\lambda z:B.z)) : T$
- What is the *largest* type  $T$  that makes the same assertion true?

$\square$

**Exercise: 2 stars, optional (small\_large\_3)**

- What is the *smallest* type  $T$  that makes the following assertion true?  $a:A \vdash (\lambda p:(A*T). (p.\text{snd}) (p.\text{fst})) (a, \lambda z:A.z) : A$
- What is the *largest* type  $T$  that makes the same assertion true?

$\square$

**Exercise: 2 stars (small\_large\_4)**

- What is the *smallest* type  $T$  that makes the following assertion true?  $\text{exists } S, \text{empty} \vdash (\lambda p:(A*T). (p.\text{snd}) (p.\text{fst})) : S$
- What is the *largest* type  $T$  that makes the same assertion true?

$\square$

**Exercise: 2 stars (smallest\_1)** What is the *smallest* type  $T$  that makes the following assertion true?  $\text{exists } S, \text{exists } t, \text{empty} \vdash (\lambda x:T. x \ x) \ t : S \parallel \square$

**Exercise: 2 stars (smallest\_2)** What is the *smallest* type  $T$  that makes the following assertion true?  $\text{empty} \vdash (\lambda x:\text{Top}. x) ((\lambda z:A.z), (\lambda z:B.z)) : T \parallel \square$

**Exercise: 3 stars, optional (count\_supertypes)** How many supertypes does the record type  $\{x:A, y:C \rightarrow C\}$  have? That is, how many different types  $T$  are there such that  $\{x:A, y:C \rightarrow C\} <: T$ ? (We consider two types to be different if they are written differently, even if each is a subtype of the other. For example,  $\{x:A, y:B\}$  and  $\{y:B, x:A\}$  are different.)

□

**Exercise: 2 stars (pair\_permutation)** The subtyping rule for product types  $S1 <: T1$   $S2 <: T2$

---

(S\_Prod)  $S1 * S2 <: T1 * T2$  intuitively corresponds to the "depth" subtyping rule for records. Extending the analogy, we might consider adding a "permutation" rule

---

$T1 * T2 <: T2 * T1$  for products. Is this a good idea? Briefly explain why or why not.

□

## 28.3 Formal Definitions

Most of the definitions – in particular, the syntax and operational semantics of the language – are identical to what we saw in the last chapter. We just need to extend the typing relation with the subsumption rule and add a new **Inductive** definition for the subtyping relation. Let's first do the identical bits.

### 28.3.1 Core Definitions

#### Syntax

For the sake of more interesting examples below, we'll allow an arbitrary set of additional base types like `String`, `Float`, etc. We won't bother adding any constants belonging to these types or any operators on them, but we could easily do so.

In the rest of the chapter, we formalize just base types, booleans, arrow types, *Unit*, and *Top*, omitting record types and leaving product types as an exercise.

```
Inductive ty : Type :=
| TTop : ty
| TBool : ty
| TBase : id → ty
| TArrow : ty → ty → ty
| TUnit : ty
.
```

```
Inductive tm : Type :=
| tvar : id → tm
| tapp : tm → tm → tm
```

```

| tabs : id → ty → tm → tm
| ttrue : tm
| tfalse : tm
| tif : tm → tm → tm → tm
| tunit : tm

```

## Substitution

The definition of substitution remains exactly the same as for the pure STLC.

```

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒
    if beq_id x y then s else t
  | tabs y T t1 ⇒
    tabs y T (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒
    tapp (subst x s t1) (subst x s t2)
  | ttrue ⇒
    ttrue
  | tfalse ⇒
    tfalse
  | tif t1 t2 t3 ⇒
    tif (subst x s t1) (subst x s t2) (subst x s t3)
  | tunit ⇒
    tunit
  end.

```

Notation "'[ x := s ]' t" := (subst x s t) (at level 20).

## Reduction

Likewise the definitions of the **value** property and the **step** relation.

```

Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
    value (tabs x T t)
  | v_true :
    value ttrue
  | v_false :
    value tfalse
  | v_unit :
    value tunit

```

Hint Constructors **value**.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=  
| ST\_AppAbs : ∀ x T t12 v2,  
    **value** v2 →  
    (tapp (tabs x T t12) v2) ==> [x:=v2] t12  
| ST\_App1 : ∀ t1 t1' t2,  
    t1 ==> t1' →  
    (tapp t1 t2) ==> (tapp t1' t2)  
| ST\_App2 : ∀ v1 t2 t2',  
    **value** v1 →  
    t2 ==> t2' →  
    (tapp v1 t2) ==> (tapp v1 t2')  
| ST\_IfTrue : ∀ t1 t2,  
    (tif ttrue t1 t2) ==> t1  
| ST\_IfFalse : ∀ t1 t2,  
    (tif tfalse t1 t2) ==> t2  
| ST\_If : ∀ t1 t1' t2 t3,  
    t1 ==> t1' →  
    (tif t1 t2 t3) ==> (tif t1' t2 t3)  
where "t1 '==>' t2" := (**step** t1 t2).

Hint Constructors **step**.

### 28.3.2 Subtyping

Now we come to the most interesting part. We begin by defining the subtyping relation and developing some of its important technical properties.

The definition of subtyping is just what we sketched in the motivating discussion.

Reserved Notation "T '<:' U" (at level 40).

Inductive **subtype** : **ty** → **ty** → Prop :=  
| S\_Refl : ∀ T,  
    T <: T  
| S\_Trans : ∀ S U T,  
    S <: U →  
    U <: T →  
    S <: T  
| S\_Top : ∀ S,  
    S <: TTop  
| S\_Arrow : ∀ S1 S2 T1 T2,  
    T1 <: S1 →  
    S2 <: T2 →



$(\text{TArrow } S1 \ S2) <: (\text{TArrow } T1 \ T2)$   
 where "T '<:' U" := (**subtype** T U).

Note that we don't need any special rules for base types: they are automatically subtypes of themselves (by `S_Refl`) and `Top` (by `S_Top`), and that's all we want.

Hint Constructors **subtype**.

Module EXAMPLES.

Notation x := (ld 0).

Notation y := (ld 1).

Notation z := (ld 2).

Notation A := (TBase (ld 6)).

Notation B := (TBase (ld 7)).

Notation C := (TBase (ld 8)).

Notation String := (TBase (ld 9)).

Notation Float := (TBase (ld 10)).

Notation Integer := (TBase (ld 11)).

**Exercise: 2 stars, optional (subtyping\_judgements)** (Do this exercise after you have added product types to the language, at least up to this point in the file).

Using the encoding of records into pairs, define pair types representing the record types  
`Person := { name : String }` `Student := { name : String ; gpa : Float }` `Employee := { name : String ; ssN : Integer }`

Recall that in chapter `MoreStlc`, the optional subsection "Encoding Records" describes how records can be encoded as pairs.

Definition `Person : ty` :=

*admit.*

Definition `Student : ty` :=

*admit.*

Definition `Employee : ty` :=

*admit.*

Example `sub_student_person` :

`Student <: Person.`

Proof.

*Admitted.*

Example `sub_employee_person` :

`Employee <: Person.`

Proof.

*Admitted.*

□

Example `subtyping_example_0` :

```

(TArrow C Person) <: (TArrow C TTop).
Proof.
  apply S_Arrow.
  apply S_Refl. auto.
Qed.

```

The following facts are mostly easy to prove in Coq. To get full benefit from the exercises, make sure you also understand how to prove them on paper!

**Exercise: 1 star, optional (subtyping\_example\_1)** Example subtyping\_example\_1 :  
 (TArrow TTop Student) <: (TArrow (TArrow C C) Person).  
 Proof with eauto.  
*Admitted.*  
 □

**Exercise: 1 star, optional (subtyping\_example\_2)** Example subtyping\_example\_2 :  
 (TArrow TTop Person) <: (TArrow Person TTop).  
 Proof with eauto.  
*Admitted.*  
 □

End EXAMPLES.

### 28.3.3 Typing

The only change to the typing relation is the addition of the rule of subsumption, T\_Sub.

Definition context := partial\_map ty.

Reserved Notation "Gamma '|-' t '\in' T" (at level 40).

Inductive has\_type : context → tm → ty → Prop :=

```

| T_Var : ∀ Gamma x T,
  Gamma x = Some T →
  Gamma ⊢ (tvar x) \in T
| T_Abs : ∀ Gamma x T11 T12 t12,
  (update Gamma x T11) ⊢ t12 \in T12 →
  Gamma ⊢ (tabs x T11 t12) \in (TArrow T11 T12)
| T_App : ∀ T1 T2 Gamma t1 t2,
  Gamma ⊢ t1 \in (TArrow T1 T2) →
  Gamma ⊢ t2 \in T1 →
  Gamma ⊢ (tapp t1 t2) \in T2
| T_True : ∀ Gamma,
  Gamma ⊢ ttrue \in TBool
| T_False : ∀ Gamma,

```

```

      Gamma ⊢ tfalse \in TBool
| T_If : ∀ t1 t2 t3 T Gamma,
      Gamma ⊢ t1 \in TBool →
      Gamma ⊢ t2 \in T →
      Gamma ⊢ t3 \in T →
      Gamma ⊢ (tif t1 t2 t3) \in T
| T_Unit : ∀ Gamma,
      Gamma ⊢ tunit \in TUnit

| T_Sub : ∀ Gamma t S T,
      Gamma ⊢ t \in S →
      S <: T →
      Gamma ⊢ t \in T

```

where "Gamma |- t \in T" := (**has\_type** Gamma t T).

Hint Constructors **has\_type**.

Hint Extern 2 (**has\_type** - (tapp - -) -) ⇒  
 eapply T\_App; auto.

Hint Extern 2 (- = -) ⇒ compute; reflexivity.

### 28.3.4 Typing examples

Module EXAMPLES2.

Import *Examples*.

Do the following exercises after you have added product types to the language. For each informal typing judgement, write it as a formal statement in Coq and prove it.

**Exercise: 1 star, optional (typing\_example\_0)** □

**Exercise: 2 stars, optional (typing\_example\_1)** □

**Exercise: 2 stars, optional (typing\_example\_2)** □

End EXAMPLES2.

## 28.4 Properties

The fundamental properties of the system that we want to check are the same as always: progress and preservation. Unlike the extension of the STLC with references, we don't need to change the *statements* of these properties to take subtyping into account. However, their proofs do become a little bit more involved.

### 28.4.1 Inversion Lemmas for Subtyping

Before we look at the properties of the typing relation, we need to record a couple of critical structural properties of the subtype relation:

- Bool is the only subtype of Bool
- every subtype of an arrow type is itself an arrow type.

These are called *inversion lemmas* because they play the same role in later proofs as the built-in **inversion** tactic: given a hypothesis that there exists a derivation of some subtyping statement  $S <: T$  and some constraints on the shape of  $S$  and/or  $T$ , each one reasons about what this derivation must look like to tell us something further about the shapes of  $S$  and  $T$  and the existence of subtype relations between their parts.

**Exercise: 2 stars, optional (sub\_inversion\_Bool)** Lemma `sub_inversion_Bool` :  $\forall U,$   
 $U <: \text{TBool} \rightarrow$   
 $U = \text{TBool}.$

Proof with `auto`.

```
intros U Hs.
remember TBool as V.
Admitted.
```

**Exercise: 3 stars, optional (sub\_inversion\_arrow)** Lemma `sub_inversion_arrow` :  $\forall U$   
 $V1\ V2,$   
 $U <: (\text{TArrow}\ V1\ V2) \rightarrow$   
 $\exists U1, \exists U2,$   
 $U = (\text{TArrow}\ U1\ U2) \wedge (V1 <: U1) \wedge (U2 <: V2).$

Proof with `eauto`.

```
intros U V1 V2 Hs.
remember (TArrow V1 V2) as V.
generalize dependent V2. generalize dependent V1.
Admitted.
```

□

### 28.4.2 Canonical Forms

We'll see first that the proof of the progress theorem doesn't change too much – we just need one small refinement. When we're considering the case where the term in question is an application  $t1\ t2$  where both  $t1$  and  $t2$  are values, we need to know that  $t1$  has the *form* of a lambda-abstraction, so that we can apply the **ST\_AppAbs** reduction rule. In the ordinary STLC, this is obvious: we know that  $t1$  has a function type  $T11 \rightarrow T12$ , and there is only one rule that can be used to give a function type to a value – rule **T\_Abs** – and the form of the conclusion of this rule forces  $t1$  to be an abstraction.

In the STLC with subtyping, this reasoning doesn't quite work because there's another rule that can be used to show that a value has a function type: subsumption. Fortunately, this possibility doesn't change things much: if the last rule used to show  $\Gamma \vdash t1 : T1 \rightarrow T2$  is subsumption, then there is some *sub*-derivation whose subject is also  $t1$ , and we can reason by induction until we finally bottom out at a use of **T\_Abs**.

This bit of reasoning is packaged up in the following lemma, which tells us the possible "canonical forms" (i.e., values) of function type.

**Exercise: 3 stars, optional (canonical\_forms\_of\_arrow\_types)** Lemma canonical\_forms\_of\_arrow\_ty

```
: ∀ Γ s T1 T2,
  Γ ⊢ s \in (TArrow T1 T2) →
  value s →
  ∃ x, ∃ S1, ∃ s2,
    s = tabs x S1 s2.
```

Proof with eauto.

*Admitted.*

□

Similarly, the canonical forms of type **Bool** are the constants **true** and **false**.

Lemma canonical\_forms\_of\_Bool : ∀ Γ s,

```
Γ ⊢ s \in TBool →
```

```
value s →
(s = ttrue ∨ s = tfalse).
```

Proof with eauto.

```
intros Γ s Hty Hv.
```

```
remember TBool as T.
```

```
induction Hty; try solve by inversion...
```

```
-
```

```
subst. apply sub_inversion_Bool in H. subst...
```

Qed.

### 28.4.3 Progress

The proof of progress proceeds like the one for the pure STLC, except that in several places we invoke canonical forms lemmas...

*Theorem* (Progress): For any term  $t$  and type  $T$ , if  $\text{empty} \vdash t : T$  then  $t$  is a value or  $t \Rightarrow t'$  for some term  $t'$ .

*Proof:* Let  $t$  and  $T$  be given, with  $\text{empty} \vdash t : T$ . Proceed by induction on the typing derivation.

The cases for **T\_Abs**, **T\_Unit**, **T\_True** and **T\_False** are immediate because abstractions, **unit**, **true**, and **false** are already values. The **T\_Var** case is vacuous because variables cannot be typed in the empty context. The remaining cases are more interesting:

- If the last step in the typing derivation uses rule **T\_App**, then there are terms  $t1$   $t2$  and types  $T1$  and  $T2$  such that  $t = t1\ t2$ ,  $T = T2$ ,  $\text{empty} \vdash t1 : T1 \rightarrow T2$ , and  $\text{empty} \vdash t2 : T1$ . Moreover, by the induction hypothesis, either  $t1$  is a value or it steps, and either  $t2$  is a value or it steps. There are three possibilities to consider:
  - Suppose  $t1 ==> t1'$  for some term  $t1'$ . Then  $t1\ t2 ==> t1'\ t2$  by **ST\_App1**.
  - Suppose  $t1$  is a value and  $t2 ==> t2'$  for some term  $t2'$ . Then  $t1\ t2 ==> t1\ t2'$  by rule **ST\_App2** because  $t1$  is a value.
  - Finally, suppose  $t1$  and  $t2$  are both values. By Lemma *canonical\_forms\_for\_arrow\_types*, we know that  $t1$  has the form  $\backslash x:S1.s2$  for some  $x$ ,  $S1$ , and  $s2$ . But then  $(\backslash x:S1.s2)\ t2 ==> [x:=t2]s2$  by **ST\_AppAbs**, since  $t2$  is a value.
- If the final step of the derivation uses rule **T\_If**, then there are terms  $t1$ ,  $t2$ , and  $t3$  such that  $t = \text{if } t1 \text{ then } t2 \text{ else } t3$ , with  $\text{empty} \vdash t1 : \text{Bool}$  and with  $\text{empty} \vdash t2 : T$  and  $\text{empty} \vdash t3 : T$ . Moreover, by the induction hypothesis, either  $t1$  is a value or it steps.
  - If  $t1$  is a value, then by the canonical forms lemma for booleans, either  $t1 = \text{true}$  or  $t1 = \text{false}$ . In either case,  $t$  can step, using rule **ST\_IfTrue** or **ST\_IfFalse**.
  - If  $t1$  can step, then so can  $t$ , by rule **ST\_If**.
- If the final step of the derivation is by **T\_Sub**, then there is a type  $S$  such that  $S <: T$  and  $\text{empty} \vdash t : S$ . The desired result is exactly the induction hypothesis for the typing subderivation.

Theorem progress :  $\forall t\ T,$   
 $\text{empty} \vdash t \backslash \text{in } T \rightarrow$   
 $\text{value } t \vee \exists t', t ==> t'.$

Proof with eauto.

```

intros t T Ht.
remember empty as Gamma.
revert HeqGamma.
induction Ht;
  intros HeqGamma; subst...
-
  inversion H.
-
  right.
  destruct IHHT1; subst...
+
  destruct IHHT2; subst...
×

```

```

    destruct (canonical_forms_of_arrow_types empty t1 T1 T2)
      as [x [S1 [t12 Heqt1]]]...
    subst.  $\exists$  ([x:=t2] t12)...
  ×
    inversion H0 as [t2' Hstp].  $\exists$  (tapp t1 t2')...
  +
    inversion H as [t1' Hstp].  $\exists$  (tapp t1' t2)...
  -
    right.
    destruct IHht1.
    + eauto.
    + assert (t1 = ttrue  $\vee$  t1 = tfalse)
      by (eapply canonical_forms_of_Bool; eauto).
      inversion H0; subst...
    + inversion H. rename x into t1'. eauto.
Qed.

```

## 28.4.4 Inversion Lemmas for Typing

The proof of the preservation theorem also becomes a little more complex with the addition of subtyping. The reason is that, as with the "inversion lemmas for subtyping" above, there are a number of facts about the typing relation that are "obvious from the definition" in the pure STLC (and hence can be obtained directly from the `inversion` tactic) but that require real proofs in the presence of subtyping because there are multiple ways to derive the same `has_type` statement.

The following "inversion lemma" tells us that, if we have a derivation of some typing statement  $\Gamma \vdash \lambda x:S1.t2 : T$  whose subject is an abstraction, then there must be some subderivation giving a type to the body  $t2$ .

*Lemma:* If  $\Gamma \vdash \lambda x:S1.t2 : T$ , then there is a type  $S2$  such that  $\Gamma, x:S1 \vdash t2 : S2$  and  $S1 \rightarrow S2 <: T$ .

(Notice that the lemma does *not* say, "then  $T$  itself is an arrow type" – this is tempting, but false!)

*Proof:* Let  $\Gamma, x, S1, t2$  and  $T$  be given as described. Proceed by induction on the derivation of  $\Gamma \vdash \lambda x:S1.t2 : T$ . Cases `T_Var`, `T_App`, are vacuous as those rules cannot be used to give a type to a syntactic abstraction.

- If the last step of the derivation is a use of `T_Abs` then there is a type  $T12$  such that  $T = S1 \rightarrow T12$  and  $\Gamma, x:S1 \vdash t2 : T12$ . Picking  $T12$  for  $S2$  gives us what we need:  $S1 \rightarrow T12 <: S1 \rightarrow T12$  follows from `S_Refl`.
- If the last step of the derivation is a use of `T_Sub` then there is a type  $S$  such that  $S <: T$  and  $\Gamma \vdash \lambda x:S1.t2 : S$ . The IH for the typing subderivation tell us that there

is some type  $S2$  with  $S1 \rightarrow S2 <: S$  and  $\Gamma, x:S1 \vdash t2 : S2$ . Picking type  $S2$  gives us what we need, since  $S1 \rightarrow S2 <: T$  then follows by  $S\_Trans$ .

```
Lemma typing_inversion_abs : ∀ Γ x S1 t2 T,
  Γ ⊢ (tabs x S1 t2) \in T →
  (∃ S2, (TArrow S1 S2) <: T
    ∧ (update Γ x S1) ⊢ t2 \in S2).
```

Proof with eauto.

```
intros Γ x S1 t2 T H.
remember (tabs x S1 t2) as t.
induction H;
  inversion Heqt; subst; intros; try solve by inversion.
-
  ∃ T12...
-
  destruct IHhas_type as [S2 [Hsub Hty]]...
Qed.
```

Similarly...

```
Lemma typing_inversion_var : ∀ Γ x T,
  Γ ⊢ (tvar x) \in T →
  ∃ S,
  Γ x = Some S ∧ S <: T.
```

Proof with eauto.

```
intros Γ x T Hty.
remember (tvar x) as t.
induction Hty; intros;
  inversion Heqt; subst; try solve by inversion.
-
  ∃ T...
-
  destruct IHHty as [U [Hctx HsubU]]... Qed.
```

```
Lemma typing_inversion_app : ∀ Γ t1 t2 T2,
  Γ ⊢ (tapp t1 t2) \in T2 →
  ∃ T1,
  Γ ⊢ t1 \in (TArrow T1 T2) ∧
  Γ ⊢ t2 \in T1.
```

Proof with eauto.

```
intros Γ t1 t2 T2 Hty.
remember (tapp t1 t2) as t.
induction Hty; intros;
  inversion Heqt; subst; try solve by inversion.
-
```



```

    ∃ T1...
  -
    destruct IHHty as [U1 [Hty1 Hty2]]...
Qed.

Lemma typing_inversion_true : ∀ Gamma T,
  Gamma ⊢ ttrue \in T →
  TBool <: T.
Proof with eauto.
  intros Gamma T Htyp. remember ttrue as tu.
  induction Htyp;
    inversion Heqtu; subst; intros...
Qed.

Lemma typing_inversion_false : ∀ Gamma T,
  Gamma ⊢ tfalse \in T →
  TBool <: T.
Proof with eauto.
  intros Gamma T Htyp. remember tfalse as tu.
  induction Htyp;
    inversion Heqtu; subst; intros...
Qed.

Lemma typing_inversion_if : ∀ Gamma t1 t2 t3 T,
  Gamma ⊢ (tif t1 t2 t3) \in T →
  Gamma ⊢ t1 \in TBool
  ∧ Gamma ⊢ t2 \in T
  ∧ Gamma ⊢ t3 \in T.
Proof with eauto.
  intros Gamma t1 t2 t3 T Hty.
  remember (tif t1 t2 t3) as t.
  induction Hty; intros;
    inversion Heqt; subst; try solve by inversion.
  -
    auto.
  -
    destruct (IHHty H0) as [H1 [H2 H3]]...
Qed.

Lemma typing_inversion_unit : ∀ Gamma T,
  Gamma ⊢ tunit \in T →
  TUnit <: T.
Proof with eauto.
  intros Gamma T Htyp. remember tunit as tu.
  induction Htyp;

```

inversion *Heqtu*; subst; intros...  
 Qed.

The inversion lemmas for typing and for subtyping between arrow types can be packaged up as a useful "combination lemma" telling us exactly what we'll actually require below.

Lemma *abs\_arrow* :  $\forall x \ S1 \ s2 \ T1 \ T2,$   
 $\text{empty} \vdash (\text{tabs } x \ S1 \ s2) \setminus \text{in } (\text{TArrow } T1 \ T2) \rightarrow$   
 $T1 <: S1$   
 $\wedge (\text{update empty } x \ S1) \vdash s2 \setminus \text{in } T2.$

Proof with *eauto*.

intros *x S1 s2 T1 T2 Hty*.  
 apply *typing\_inversion\_abs* in *Hty*.  
 inversion *Hty* as [*S2 [Hsub Hty1]*].  
 apply *sub\_inversion\_arrow* in *Hsub*.  
 inversion *Hsub* as [*U1 [U2 [Heq [Hsub1 Hsub2]]]*].  
 inversion *Heq*; subst... Qed.

## 28.4.5 Context Invariance

The context invariance lemma follows the same pattern as in the pure STLC.

Inductive *appears\_free\_in* : *id*  $\rightarrow$  *tm*  $\rightarrow$  Prop :=  
 | *afi\_var* :  $\forall x,$   
 $\text{appears\_free\_in } x \ (\text{tvar } x)$   
 | *afi\_app1* :  $\forall x \ t1 \ t2,$   
 $\text{appears\_free\_in } x \ t1 \rightarrow \text{appears\_free\_in } x \ (\text{tapp } t1 \ t2)$   
 | *afi\_app2* :  $\forall x \ t1 \ t2,$   
 $\text{appears\_free\_in } x \ t2 \rightarrow \text{appears\_free\_in } x \ (\text{tapp } t1 \ t2)$   
 | *afi\_abs* :  $\forall x \ y \ T11 \ t12,$   
 $y \neq x \rightarrow$   
 $\text{appears\_free\_in } x \ t12 \rightarrow$   
 $\text{appears\_free\_in } x \ (\text{tabs } y \ T11 \ t12)$   
 | *afi\_if1* :  $\forall x \ t1 \ t2 \ t3,$   
 $\text{appears\_free\_in } x \ t1 \rightarrow$   
 $\text{appears\_free\_in } x \ (\text{tif } t1 \ t2 \ t3)$   
 | *afi\_if2* :  $\forall x \ t1 \ t2 \ t3,$   
 $\text{appears\_free\_in } x \ t2 \rightarrow$   
 $\text{appears\_free\_in } x \ (\text{tif } t1 \ t2 \ t3)$   
 | *afi\_if3* :  $\forall x \ t1 \ t2 \ t3,$   
 $\text{appears\_free\_in } x \ t3 \rightarrow$   
 $\text{appears\_free\_in } x \ (\text{tif } t1 \ t2 \ t3)$   
 .

Hint Constructors *appears\_free\_in*.

Lemma context\_invariance :  $\forall \text{Gamma } \text{Gamma}' \ t \ S,$   
 $\text{Gamma} \vdash t \setminus \text{in } S \rightarrow$   
 $(\forall x, \text{appears\_free\_in } x \ t \rightarrow \text{Gamma } x = \text{Gamma}' \ x) \rightarrow$   
 $\text{Gamma}' \vdash t \setminus \text{in } S.$

Proof with eauto.

```

intros. generalize dependent Gamma'.
induction H;
  intros Gamma' Heqv...
-
  apply T_Var... rewrite ← Heqv...
-
  apply T_Abs... apply IHhas_type. intros x0 Hafi.
  unfold update, t_update. destruct (beq_idP x x0)...
-
  apply T_If...

```

Qed.

Lemma free\_in\_context :  $\forall x \ t \ T \ \text{Gamma},$   
 $\text{appears\_free\_in } x \ t \rightarrow$   
 $\text{Gamma} \vdash t \setminus \text{in } T \rightarrow$   
 $\exists T', \text{Gamma } x = \text{Some } T'.$

Proof with eauto.

```

intros x t T Gamma Hafi Htyp.
induction Htyp;
  subst; inversion Hafi; subst...
-
  destruct (IHHtyp H4) as [T Hctx].  $\exists T.$ 
  unfold update, t_update in Hctx.
  rewrite ← beq_id_false_iff in H2.
  rewrite H2 in Hctx... Qed.

```

## 28.4.6 Substitution

The *substitution lemma* is proved along the same lines as for the pure STLC. The only significant change is that there are several places where, instead of the built-in `inversion` tactic, we need to use the inversion lemmas that we proved above to extract structural information from assumptions about the well-typedness of subterms.

Lemma substitution\_preserves\_typing :  $\forall \text{Gamma } x \ U \ v \ t \ S,$   
 $(\text{update } \text{Gamma } x \ U) \vdash t \setminus \text{in } S \rightarrow$   
 $\text{empty} \vdash v \setminus \text{in } U \rightarrow$   
 $\text{Gamma} \vdash ([x:=v]t) \setminus \text{in } S.$

Proof with eauto.

```

intros Gamma x U v t S Htypt Htypv.

```

generalize dependent  $S$ . generalize dependent  $\Gamma$ .  
induction  $t$ ; intros; simpl.

```
-
  rename  $i$  into  $y$ .
  destruct (typing_inversion_var _ _ _ Htypt)
    as [ $T$  [ $Hctx$   $Hsub$ ]].
  unfold update, t_update in  $Hctx$ .
  destruct (beq_idP  $x$   $y$ ) as [ $Hxy$ | $Hxy$ ]; eauto;
  subst.
  inversion  $Hctx$ ; subst. clear  $Hctx$ .
  apply context_invariance with empty...
  intros  $x$   $Hcontra$ .
  destruct (free_in_context _ _  $S$  empty  $Hcontra$ )
    as [ $T'$   $HT'$ ]...
  inversion  $HT'$ .

-
  destruct (typing_inversion_app _ _ _ _ Htypt)
    as [ $T1$  [ $Htypt1$   $Htypt2$ ]].
  eapply T_App...

-
  rename  $i$  into  $y$ . rename  $t$  into  $T1$ .
  destruct (typing_inversion_abs _ _ _ _ _ Htypt)
    as [ $T2$  [ $Hsub$   $Htypt2$ ]].
  apply T_Sub with (TArrow  $T1$   $T2$ )... apply T_Abs...
  destruct (beq_idP  $x$   $y$ ) as [ $Hxy$ | $Hxy$ ].
  +
    eapply context_invariance...
    subst.
    intros  $x$   $Hafi$ . unfold update, t_update.
    destruct (beq_id  $y$   $x$ )...
  +
    apply IHt. eapply context_invariance...
    intros  $z$   $Hafi$ . unfold update, t_update.
    destruct (beq_idP  $y$   $z$ )...
    subst.
    rewrite ← beq_id_false_iff in  $Hxy$ . rewrite  $Hxy$ ...

-
  assert (TBool <:  $S$ )
    by apply (typing_inversion_true _ _ _ Htypt)...

-
  assert (TBool <:  $S$ )
    by apply (typing_inversion_false _ _ _ Htypt)...
```

```

-
  assert ((update Gamma x U) ⊢ t1 \in TBool
    ∧ (update Gamma x U) ⊢ t2 \in S
    ∧ (update Gamma x U) ⊢ t3 \in S)
    by apply (typing_inversion_if _ _ _ _ Htypt).
  inversion H as [H1 [H2 H3]].
  apply IHt1 in H1. apply IHt2 in H2. apply IHt3 in H3.
  auto.
-
  assert (TUnit <: S)
    by apply (typing_inversion_unit _ _ Htypt)...
Qed.

```

### 28.4.7 Preservation

The proof of preservation now proceeds pretty much as in earlier chapters, using the substitution lemma at the appropriate point and again using inversion lemmas from above to extract structural information from typing assumptions.

*Theorem* (Preservation): If  $t, t'$  are terms and  $T$  is a type such that  $\text{empty} \vdash t : T$  and  $t \Rightarrow t'$ , then  $\text{empty} \vdash t' : T$ .

*Proof:* Let  $t$  and  $T$  be given such that  $\text{empty} \vdash t : T$ . We proceed by induction on the structure of this typing derivation, leaving  $t'$  general. The cases  $\text{T\_Abs}$ ,  $\text{T\_Unit}$ ,  $\text{T\_True}$ , and  $\text{T\_False}$  cases are vacuous because abstractions and constants don't step. Case  $\text{T\_Var}$  is vacuous as well, since the context is empty.

- If the final step of the derivation is by  $\text{T\_App}$ , then there are terms  $t1$  and  $t2$  and types  $T1$  and  $T2$  such that  $t = t1 \ t2$ ,  $T = T2$ ,  $\text{empty} \vdash t1 : T1 \rightarrow T2$ , and  $\text{empty} \vdash t2 : T1$ .

By the definition of the step relation, there are three ways  $t1 \ t2$  can step. Cases  $\text{ST\_App1}$  and  $\text{ST\_App2}$  follow immediately by the induction hypotheses for the typing subderivations and a use of  $\text{T\_App}$ .

Suppose instead  $t1 \ t2$  steps by  $\text{ST\_AppAbs}$ . Then  $t1 = \lambda x:S.t12$  for some type  $S$  and term  $t12$ , and  $t' = [x:=t2]t12$ .

By lemma `abs_arrow`, we have  $T1 <: S$  and  $x:S1 \vdash s2 : T2$ . It then follows by the substitution lemma (`substitution_preserves_typing`) that  $\text{empty} \vdash [x:=t2] t12 : T2$  as desired.

- If the final step of the derivation uses rule  $\text{T\_If}$ , then there are terms  $t1, t2$ , and  $t3$  such that  $t = \text{if } t1 \text{ then } t2 \text{ else } t3$ , with  $\text{empty} \vdash t1 : \text{Bool}$  and with  $\text{empty} \vdash t2 : T$  and  $\text{empty} \vdash t3 : T$ . Moreover, by the induction hypothesis, if  $t1$  steps to  $t1'$  then  $\text{empty} \vdash t1' : \text{Bool}$ . There are three cases to consider, depending on which rule was used to show  $t \Rightarrow t'$ .

- \* If  $t \Rightarrow t'$  by rule `ST_Lf`, then  $t' = \text{if } t1' \text{ then } t2 \text{ else } t3$  with  $t1 \Rightarrow t1'$ . By the induction hypothesis,  $\text{empty} \vdash t1' : \text{Bool}$ , and so  $\text{empty} \vdash t' : T$  by `T_Lf`.
- \* If  $t \Rightarrow t'$  by rule `ST_LfTrue` or `ST_LfFalse`, then either  $t' = t2$  or  $t' = t3$ , and  $\text{empty} \vdash t' : T$  follows by assumption.
- If the final step of the derivation is by `T_Sub`, then there is a type  $S$  such that  $S <: T$  and  $\text{empty} \vdash t : S$ . The result is immediate by the induction hypothesis for the typing subderivation and an application of `T_Sub`.  $\square$

**Theorem preservation :**  $\forall t t' T,$   
 $\text{empty} \vdash t \text{ \texttt{in} } T \rightarrow$   
 $t \Rightarrow t' \rightarrow$   
 $\text{empty} \vdash t' \text{ \texttt{in} } T.$

**Proof with eauto.**

```
intros t t' T HT.
remember empty as Gamma. generalize dependent HeqGamma.
generalize dependent t'.
induction HT;
  intros t' HeqGamma HE; subst; inversion HE; subst...
-
  inversion HE; subst...
+
  destruct (abs_arrow _ _ _ _ HT1) as [HA1 HA2].
  apply substitution_preserves_typing with T...
```

**Qed.**

## 28.4.8 Records, via Products and Top

This formalization of the STLC with subtyping has omitted record types, for brevity. If we want to deal with them more seriously, we have two choices.

First, we can treat them as part of the core language, writing down proper syntax, typing, and subtyping rules for them. Chapter `RecordSub` shows how this extension works.

On the other hand, if we are treating them as a derived form that is desugared in the parser, then we shouldn't need any new rules: we should just check that the existing rules for subtyping product and *Unit* types give rise to reasonable rules for record subtyping via this encoding. To do this, we just need to make one small change to the encoding described earlier: instead of using *Unit* as the base case in the encoding of tuples and the "don't care" placeholder in the encoding of records, we use *Top*. So:

$\{a:\text{Nat}, b:\text{Nat}\} \text{ ----> } \{\text{Nat}, \text{Nat}\}$	$\text{i.e., } (\text{Nat}, (\text{Nat}, \text{Top}))$
$\{c:\text{Nat}, a:\text{Nat}\} \text{ ----> } \{\text{Nat}, \text{Top}, \text{Nat}\}$	$\text{i.e., } (\text{Nat}, (\text{Top}, (\text{Nat}, \text{Top})))$

The encoding of record values doesn't change at all. It is easy (and instructive) to check that the subtyping rules above are validated by the encoding. For the rest of this chapter, we'll follow this encoding-based approach.

## 28.4.9 Exercises

**Exercise: 2 stars (variations)** Each part of this problem suggests a different way of changing the definition of the STLC with Unit and subtyping. (These changes are not cumulative: each part starts from the original language.) In each part, list which properties (Progress, Preservation, both, or neither) become false. If a property becomes false, give a counterexample.

- Suppose we add the following typing rule:  $\Gamma \vdash t : S1 \multimap S2 \quad S1 <: T1 \quad T1 <: S1 \quad S2 <: T2$

– ————— (T\_Funny1)  $\Gamma \vdash t : T1 \multimap T2$

- Suppose we add the following reduction rule:

– ————— (ST\_Funny21)

$\text{unit} \implies (\backslash x:\text{Top}. x)$

- Suppose we add the following subtyping rule:

– ————— (S\_Funny3)

$\text{Unit} <: \text{Top} \multimap \text{Top}$

- Suppose we add the following subtyping rule:

– ————— (S\_Funny4)

$\text{Top} \multimap \text{Top} <: \text{Unit}$

- Suppose we add the following evaluation rule:

– ————— (ST\_Funny5)

$(\text{unit } t) \implies (t \text{ unit})$

- Suppose we add the same evaluation rule *and* a new typing rule:

– ————— (ST\_Funny5)

$(\text{unit } t) \implies (t \text{ unit})$

– ————— (T\_Funny6)

empty |- Unit : Top->Top

- Suppose we *change* the arrow subtyping rule to:  $S1 <: T1 \quad S2 <: T2$

– ————— (S\_Arrow')  $S1 \rightarrow S2 <: T1 \rightarrow T2$

□

## 28.5 Exercise: Adding Products

**Exercise: 4 stars (products)** Adding pairs, projections, and product types to the system we have defined is a relatively straightforward matter. Carry out this extension:

- Add constructors for pairs, first and second projections, and product types to the definitions of **ty** and **tm**. (Don't forget to add corresponding cases to *T\_cases* and *t\_cases*.)
- Extend the substitution function and value relation as in MoreSTLC.
- Extend the operational semantics with the same reduction rules as in MoreSTLC.
- Extend the subtyping relation with this rule:

$S1 <: T1 \quad S2 <: T2$

– ————— (Sub\_Prod)  $S1 * S2 <: T1 * T2$

- Extend the typing relation with the same rules for pairs and projections as in MoreSTLC.
- Extend the proofs of progress, preservation, and all their supporting lemmas to deal with the new constructs. (You'll also need to add some completely new lemmas.)

□

*Date* : 2016 – 03 – 04 09 : 33 : 20 – 0500(*Fri*, 04*Mar*2016)



# Chapter 29

## Library Typechecking

### 29.1 MoreStlc: A Typechecker for STLC

```
Require Import Coq.Bool.Bool.
```

```
Require Import Sflib.
```

```
Require Import Maps.
```

```
Require Import Stlc.
```

The **has\_type** relation of the STLC defines what it means for a term to belong to a type (in some context). But it doesn't, by itself, tell us how to *check* whether or not a term is well typed.

Fortunately, the rules defining **has\_type** are *syntax directed* – they exactly follow the shape of the term. This makes it straightforward to translate the typing rules into clauses of a typechecking *function* that takes a term and a context and either returns the term's type or else signals that the term is not typable.

```
Module STLCCHECKER.
```

```
Import STLC.
```

#### 29.1.1 Comparing Types

First, we need a function to compare two types for equality...

```
Fixpoint beq_ty (T1 T2:ty) : bool :=
  match T1,T2 with
  | TBool, TBool =>
    true
  | TArrow T11 T12, TArrow T21 T22 =>
    andb (beq_ty T11 T21) (beq_ty T12 T22)
  | _,_ =>
    false
  end.
```

... and we need to establish the usual two-way connection between the boolean result returned by `beq_ty` and the logical proposition that its inputs are equal.

Lemma `beq_ty_refl` :  $\forall T1$ ,

`beq_ty T1 T1 = true`.

Proof.

`intros T1. induction T1; simpl.`

`reflexivity.`

`rewrite IHT1_1. rewrite IHT1_2. reflexivity. Qed.`

Lemma `beq_ty__eq` :  $\forall T1 T2$ ,

`beq_ty T1 T2 = true`  $\rightarrow T1 = T2$ .

Proof with `auto`.

`intros T1. induction T1; intros T2 Hbeq; destruct T2; inversion Hbeq.`

`-`

`reflexivity.`

`-`

`rewrite andb_true_iff in H0. inversion H0 as [Hbeq1 Hbeq2].`

`apply IHT1_1 in Hbeq1. apply IHT1_2 in Hbeq2. subst... Qed.`

## 29.1.2 The Typechecker

Now here's the typechecker. It works by walking over the structure of the given term, returning either `Some T` or `None`. Each time we make a recursive call to find out the types of the subterms, we need to pattern-match on the results to make sure that they are not `None`. Also, in the `tapp` case, we use pattern matching to extract the left- and right-hand sides of the function's arrow type (and fail if the type of the function is not `TArrow T11 T12` for some `T11` and `T12`).

Fixpoint `type_check` (*Gamma*:context) (*t*:tm) : `option ty` :=

`match t with`

`| tvar x  $\Rightarrow$  Gamma x`

`| tabs x T11 t12  $\Rightarrow$  match type_check (update Gamma x T11) t12 with`

`| Some T12  $\Rightarrow$  Some (TArrow T11 T12)`

`| _  $\Rightarrow$  None`

`end`

`| tapp t1 t2  $\Rightarrow$  match type_check Gamma t1, type_check Gamma t2 with`

`| Some (TArrow T11 T12), Some T2  $\Rightarrow$`

`if beq_ty T11 T2 then Some T12 else None`

`| _,-  $\Rightarrow$  None`

`end`

`| ttrue  $\Rightarrow$  Some TBool`

`| tfalse  $\Rightarrow$  Some TBool`

`| tif x t f  $\Rightarrow$  match type_check Gamma x with`

`| Some TBool  $\Rightarrow$`

```

      match type_check Gamma t, type_check Gamma f with
      | Some T1, Some T2 =>
        if beq_ty T1 T2 then Some T1 else None
      | _,- => None
      end
    | _ => None
  end
end.

```

### 29.1.3 Properties

To verify that this typechecking algorithm is the correct one, we show that it is *sound* and *complete* for the original **has\_type** relation – that is, `type_check` and **has\_type** define the same partial function.

Theorem `type_checking_sound` :  $\forall \text{Gamma } t \text{ } T,$   
`type_check Gamma t = Some T  $\rightarrow$  has_type Gamma t T.`

Proof with `eauto`.

```

  intros Gamma t. generalize dependent Gamma.
  induction t; intros Gamma T Htc; inversion Htc.
- eauto.
-
  remember (type_check Gamma t1) as TO1.
  remember (type_check Gamma t2) as TO2.
  destruct TO1 as [T1]; try solve by inversion;
  destruct T1 as [| T11 T12]; try solve by inversion.
  destruct TO2 as [T2]; try solve by inversion.
  destruct (beq_ty T11 T2) eqn: Heqb;
  try solve by inversion.
  apply beq_ty_eq in Heqb.
  inversion H0; subst...
-
  rename i into y. rename t into T1.
  remember (update Gamma y T1) as G'.
  remember (type_check G' t0) as TO2.
  destruct TO2; try solve by inversion.
  inversion H0; subst...
- eauto.
- eauto.
-
  remember (type_check Gamma t1) as TOc.
  remember (type_check Gamma t2) as TO1.
  remember (type_check Gamma t3) as TO2.

```

```

destruct  $TOc$  as [ $Tc$ ]; try solve by inversion.
destruct  $Tc$ ; try solve by inversion.
destruct  $TO1$  as [ $T1$ ]; try solve by inversion.
destruct  $TO2$  as [ $T2$ ]; try solve by inversion.
destruct (beq_ty  $T1$   $T2$ ) eqn:Heqb;
try solve by inversion.
apply beq_ty_eq in Heqb.
inversion  $H0$ . subst. subst...

```

Qed.

Theorem type\_checking\_complete :  $\forall$   $\Gamma$   $t$   $T$ ,  
**has\_type**  $\Gamma$   $t$   $T \rightarrow$  type\_check  $\Gamma$   $t$  = **Some**  $T$ .

Proof with auto.

```

intros  $\Gamma$   $t$   $T$  Hty.
induction Hty; simpl.
- eauto.
- rewrite IHHTy...
-
  rewrite IHHTy1. rewrite IHHTy2.
  rewrite (beq_ty_refl  $T11$ )...
- eauto.
- eauto.
- rewrite IHHTy1. rewrite IHHTy2.
  rewrite IHHTy3. rewrite (beq_ty_refl  $T$ )...

```

Qed.

End STLCChecker.

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500 (Wed, 17 Feb 2016)

# Chapter 30

## Library Records

### 30.1 Records: Adding Records to STLC

```
Require Import SfLib.  
Require Import Maps.  
Require Import Imp.  
Require Import Smallstep.  
Require Import Stlc.
```

### 30.2 Adding Records

We saw in chapter `MoreStlc` how records can be treated as syntactic sugar for nested uses of products. This is fine for simple examples, but the encoding is informal (in reality, if we really treated records this way, it would be carried out in the parser, which we are eliding here), and anyway it is not very efficient. So it is also interesting to see how records can be treated as first-class citizens of the language.

Recall the informal definitions we gave before:

Syntax:

$t ::=$	Terms:
...	
$\{i1=t1, \dots, in=tn\}$	record
$t.i$	projection
$v ::=$	Values:
...	
$\{i1=v1, \dots, in=vn\}$	record value
$T ::=$	Types:
...	

| {i1:T1, ..., in:Tn}                      record type

Reduction:  $t_i \Rightarrow t_i'$  (ST\_Rcd)

---

{i1=v1, ..., im=vm, in=tn, ...}  $\Rightarrow$  {i1=v1, ..., im=vm, in=tn', ...}  
 $t_1 \Rightarrow t_1'$

---

(ST\_Proj1)  $t_1.i \Rightarrow t_1'.i$

---

(ST\_ProjRcd) {..., i=vi, ...}.i  $\Rightarrow$  vi    Typing:  $\Gamma \vdash t_1 : T_1 \dots \Gamma \vdash t_n : T_n$

---

(T\_Rcd)  $\Gamma \vdash \{i_1=t_1, \dots, i_n=t_n\} : \{i_1:T_1, \dots, i_n:T_n\}$   
 $\Gamma \vdash t : \{..., i:T_i, \dots\}$

---

(T\_Proj)  $\Gamma \vdash t.i : T_i$

---

### 30.3 Formalizing Records

Module STLCExtendedRecords.

#### Syntax and Operational Semantics

The most obvious way to formalize the syntax of record types would be this:

Module FIRSTTRY.

Definition alist ( $X : \text{Type}$ ) := **list** (**id**  $\times$   $X$ ).

Inductive **ty** : Type :=  
| TBase : **id**  $\rightarrow$  **ty**  
| TArrow : **ty**  $\rightarrow$  **ty**  $\rightarrow$  **ty**  
| TRcd : (alist **ty**)  $\rightarrow$  **ty**.

Unfortunately, we encounter here a limitation in Coq: this type does not automatically give us the induction principle we expect the induction hypothesis in the TRcd case doesn't give us any information about the **ty** elements of the list, making it useless for the proofs we want to do.

End FIRSTTRY.

It is possible to get a better induction principle out of Coq, but the details of how this is done are not very pretty, and it is not as intuitive to use as the ones Coq generates automatically for simple Inductive definitions.

Fortunately, there is a different way of formalizing records that is, in some ways, even simpler and more natural: instead of using the existing **list** type, we can essentially include its constructors ("nil" and "cons") in the syntax of types.

```

Inductive ty : Type :=
| TBase : id → ty
| TArrow : ty → ty → ty
| TRNil : ty
| TRCons : id → ty → ty → ty.

```

Similarly, at the level of terms, we have constructors `trnil` the empty record – and `trcons`, which adds a single field to the front of a list of fields.

```

Inductive tm : Type :=
| tvar : id → tm
| tapp : tm → tm → tm
| tabs : id → ty → tm → tm

| tproj : tm → id → tm
| trnil : tm
| trcons : id → tm → tm → tm.

```

Some variables, for examples...

```

Notation a := (ld 0).
Notation f := (ld 1).
Notation g := (ld 2).
Notation l := (ld 3).
Notation A := (TBase (ld 4)).
Notation B := (TBase (ld 5)).
Notation k := (ld 6).
Notation i1 := (ld 7).
Notation i2 := (ld 8).

```

```
{ i1:A }
```

```
{ i1:A→B, i2:A }
```

## Well-Formedness

Generalizing our abstract syntax for records (from lists to the nil/cons presentation) introduces the possibility of writing strange types like this

```
Definition weird_type := TRCons X A B.
```

where the "tail" of a record type is not actually a record type!

We'll structure our typing judgement so that no ill-formed types like `weird_type` are assigned to terms. To support this, we define **record\_ty** and **record\_tm**, which identify record types and terms, and **well\_formed\_ty** which rules out the ill-formed types.

First, a type is a record type if it is built with just `TRNil` and `TRCons` at the outermost level.

```
Inductive record_ty : ty → Prop :=
```

```

| RTnil :
  record_ty TRNil
| RTcons : ∀ i T1 T2,
  record_ty (TRCons i T1 T2).

```

Similarly, a term is a record term if it is built with `trnil` and `trcons`

```

Inductive record_tm : tm → Prop :=
| rtnil :
  record_tm trnil
| rtcons : ∀ i t1 t2,
  record_tm (trcons i t1 t2).

```

Note that `record_ty` and `record_tm` are not recursive – they just check the outermost constructor. The `well_formed_ty` property, on the other hand, verifies that the whole type is well formed in the sense that the tail of every record (the second argument to `TRCons`) is a record.

Of course, we should also be concerned about ill-formed terms, not just types; but type-checking can rule those out without the help of an extra `well_formed_tm` definition because it already examines the structure of terms. LATER : should they fill in part of this as an exercise? We didn't give rules for it above

```

Inductive well_formed_ty : ty → Prop :=
| wfTBase : ∀ i,
  well_formed_ty (TBase i)
| wfTArrow : ∀ T1 T2,
  well_formed_ty T1 →
  well_formed_ty T2 →
  well_formed_ty (TArrow T1 T2)
| wfTRNil :
  well_formed_ty TRNil
| wfTRCons : ∀ i T1 T2,
  well_formed_ty T1 →
  well_formed_ty T2 →
  record_ty T2 →
  well_formed_ty (TRCons i T1 T2).

```

Hint Constructors `record_ty record_tm well_formed_ty`.

## Substitution

```

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒ if beq_id x y then s else t
  | tabs y T t1 ⇒ tabs y T (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)

```



```

| tproj t1 i ⇒ tproj (subst x s t1) i
| trnil ⇒ trnil
| trcons i t1 tr1 ⇒ trcons i (subst x s t1) (subst x s tr1)
end.

```

Notation "'[ x := s ]' t" := (subst x s t) (at level 20).

## Reduction

Next we define the values of our language. A record is a value if all of its fields are.

```

Inductive value : tm → Prop :=
| v_abs : ∀ x T11 t12,
  value (tabs x T11 t12)
| v_rnil : value trnil
| v_rcons : ∀ i v1 vr,
  value v1 →
  value vr →
  value (trcons i v1 vr).

```

Hint Constructors value.

Utility functions for extracting one field from record type or term:

```

Fixpoint Tlookup (i:id) (Tr:ty) : option ty :=
match Tr with
| TRCons i' T Tr' ⇒ if beq_id i i' then Some T else Tlookup i Tr'
| _ ⇒ None
end.

```

```

Fixpoint tlookup (i:id) (tr:tm) : option tm :=
match tr with
| trcons i' t tr' ⇒ if beq_id i i' then Some t else tlookup i tr'
| _ ⇒ None
end.

```

The **step** function uses the term-level lookup function (for the projection rule), while the type-level lookup is needed for **has\_type**.

Reserved Notation "t1 '==>' t2" (at level 40).

```

Inductive step : tm → tm → Prop :=
| ST_AppAbs : ∀ x T11 t12 v2,
  value v2 →
  (tapp (tabs x T11 t12) v2) ==> ([x:=v2] t12)
| ST_App1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  (tapp t1 t2) ==> (tapp t1' t2)
| ST_App2 : ∀ v1 t2 t2',

```

```

      value v1 →
      t2 ==> t2' →
      (tapp v1 t2) ==> (tapp v1 t2')
| ST_Proj1 : ∀ t1 t1' i,
  t1 ==> t1' →
  (tproj t1 i) ==> (tproj t1' i)
| ST_ProjRcd : ∀ tr i vi,
  value tr →
  tlookup i tr = Some vi →
  (tproj tr i) ==> vi
| ST_Rcd_Head : ∀ i t1 t1' tr2,
  t1 ==> t1' →
  (trcons i t1 tr2) ==> (trcons i t1' tr2)
| ST_Rcd_Tail : ∀ i v1 tr2 tr2',
  value v1 →
  tr2 ==> tr2' →
  (trcons i v1 tr2) ==> (trcons i v1 tr2')

```

where "t1 '==>' t2" := (**step** t1 t2).

Notation multistep := (**multi step**).

Notation "t1 '==>\*' t2" := (multistep t1 t2) (at level 40).

Hint Constructors **step**.

## Typing

Definition context := partial\_map ty.

Next we define the typing rules. These are nearly direct transcriptions of the inference rules shown above. The only major difference is the use of **well\_formed\_ty**. In the informal presentation we used a grammar that only allowed well formed record types, so we didn't have to add a separate check.

We'd like to set things up so that that whenever **has\_type** *Gamma* *t* *T* holds, we also have **well\_formed\_ty** *T*. That is, **has\_type** never assigns ill-formed types to terms. In fact, we prove this theorem below.

However, we don't want to clutter the definition of **has\_type** with unnecessary uses of **well\_formed\_ty**. Instead, we place **well\_formed\_ty** checks only where needed - where an inductive call to **has\_type** won't already be checking the well-formedness of a type.

For example, we check **well\_formed\_ty** *T* in the **T\_Var** case, because there is no inductive **has\_type** call that would enforce this. Similarly, in the **T\_Abs** case, we require a proof of **well\_formed\_ty** *T11* because the inductive call to **has\_type** only guarantees that *T12* is well-formed.

In the rules you must write, the only necessary **well\_formed\_ty** check comes in the **tnil** case.

Reserved Notation " $\Gamma \vdash t \text{ in } T$ " (at level 40).

Inductive **has\_type** : context  $\rightarrow$  tm  $\rightarrow$  ty  $\rightarrow$  Prop :=

```

| T_Var :  $\forall \Gamma x T$ ,
   $\Gamma x = \text{Some } T \rightarrow$ 
  well_formed_ty  $T \rightarrow$ 
   $\Gamma \vdash (\text{tvar } x) \text{ in } T$ 
| T_Abs :  $\forall \Gamma x T11 T12 t12$ ,
  well_formed_ty  $T11 \rightarrow$ 
  (update  $\Gamma x T11$ )  $\vdash t12 \text{ in } T12 \rightarrow$ 
   $\Gamma \vdash (\text{tabs } x T11 t12) \text{ in } (\text{TArrow } T11 T12)$ 
| T_App :  $\forall T1 T2 \Gamma t1 t2$ ,
   $\Gamma \vdash t1 \text{ in } (\text{TArrow } T1 T2) \rightarrow$ 
   $\Gamma \vdash t2 \text{ in } T1 \rightarrow$ 
   $\Gamma \vdash (\text{tapp } t1 t2) \text{ in } T2$ 

| T_Proj :  $\forall \Gamma i t Ti Tr$ ,
   $\Gamma \vdash t \text{ in } Tr \rightarrow$ 
  Tlookup  $i Tr = \text{Some } Ti \rightarrow$ 
   $\Gamma \vdash (\text{tproj } t i) \text{ in } Ti$ 
| T_RNil :  $\forall \Gamma$ ,
   $\Gamma \vdash \text{trnil} \text{ in } \text{TRNil}$ 
| T_RCons :  $\forall \Gamma i t T tr Tr$ ,
   $\Gamma \vdash t \text{ in } T \rightarrow$ 
   $\Gamma \vdash tr \text{ in } Tr \rightarrow$ 
  record_ty  $Tr \rightarrow$ 
  record_tm  $tr \rightarrow$ 
   $\Gamma \vdash (\text{trcons } i t tr) \text{ in } (\text{TRCons } i T Tr)$ 

```

where " $\Gamma \vdash t \text{ in } T$ " := (**has\_type**  $\Gamma t T$ ).

Hint Constructors **has\_type**.

### 30.3.1 Examples

**Exercise: 2 stars (examples)** Finish the proofs.

Feel free to use Coq's automation features in this proof. However, if you are not confident about how the type system works, you may want to carry out the proof first using the basic features (apply instead of eapply, in particular) and then perhaps compress it using automation.

Lemma typing\_example\_2 :

```

empty  $\vdash$ 
  (tapp (tabs a (TRCons i1 (TArrow A A))
        (TRCons i2 (TArrow B B))

```

```

      TRNil))
    (tproj (tvar a) i2))
  (trcons i1 (tabs a A (tvar a))
    (trcons i2 (tabs a B (tvar a))
      trnil))) \in
  (TArrow B B).

```

Proof.

*Admitted.*

Before starting to prove this fact (or the one above!), make sure you understand what it is saying.

Example typing\_nonexample :

```

¬ ∃ T,
  (update empty a (TRCons i2 (TArrow A A)
    TRNil)) ⊢
    (trcons i1 (tabs a B (tvar a)) (tvar a)) \in
    T.

```

Proof.

*Admitted.*

Example typing\_nonexample\_2 : ∀ y,

```

¬ ∃ T,
  (update empty y A) ⊢
    (tapp (tabs a (TRCons i1 A TRNil)
      (tproj (tvar a) i1))
      (trcons i1 (tvar y) (trcons i2 (tvar y) trnil))) \in
      T.

```

Proof.

*Admitted.*

### 30.3.2 Properties of Typing

The proofs of progress and preservation for this system are essentially the same as for the pure simply typed lambda-calculus, but we need to add some technical lemmas involving records.

#### Well-Formedness

```

Lemma wf_rcd_lookup : ∀ i T Ti,
  well_formed_ty T →
  Tlookup i T = Some Ti →
  well_formed_ty Ti.

```

Proof with eauto.

```

intros i T.

```

induction  $T$ ; intros; try solve by inversion.

-  
 inversion  $H$ . subst. unfold Tlookup in  $H0$ .  
 destruct (beq\_id  $i$   $i0$ )...  
 inversion  $H0$ . subst... Qed.

Lemma step\_preserves\_record\_tm :  $\forall tr\ tr'$ ,  
**record\_tm**  $tr \rightarrow$   
 $tr ==> tr' \rightarrow$   
**record\_tm**  $tr'$ .

Proof.

intros  $tr\ tr'\ Hrt\ Hstp$ .  
 inversion  $Hrt$ ; subst; inversion  $Hstp$ ; subst; auto.

Qed.

Lemma has\_type\_wf :  $\forall Gamma\ t\ T$ ,  
 $Gamma \vdash t \text{ \texttt{in} } T \rightarrow \text{well\_formed\_ty } T$ .

Proof with eauto.

intros  $Gamma\ t\ T\ Htyp$ .  
 induction  $Htyp$ ...

-  
 inversion  $IHHtyp1$ ...  
 -  
 eapply wf\_rcd\_lookup...

Qed.

## Field Lookup

Lemma: If  $\text{empty} \vdash v : T$  and Tlookup  $i\ T$  returns Some  $Ti$ , then tlookup  $i\ v$  returns Some  $ti$  for some term  $ti$  such that  $\text{empty} \vdash ti \text{ \texttt{in} } Ti$ .

Proof: By induction on the typing derivation  $Htyp$ . Since Tlookup  $i\ T = \text{Some } Ti$ ,  $T$  must be a record type, this and the fact that  $v$  is a value eliminate most cases by inspection, leaving only the T\_RCons case.

If the last step in the typing derivation is by T\_RCons, then  $t = \text{trcons } i0\ t\ tr$  and  $T = \text{TRCons } i0\ T\ Tr$  for some  $i0, t, tr, T$  and  $Tr$ .

This leaves two possibilities to consider - either  $i0 = i$  or not.

- If  $i = i0$ , then since Tlookup  $i\ (\text{TRCons } i0\ T\ Tr) = \text{Some } Ti$  we have  $T = Ti$ . It follows that  $t$  itself satisfies the theorem.
- On the other hand, suppose  $i \neq i0$ . Then Tlookup  $i\ T = \text{Tlookup } i\ Tr$  and tlookup  $i\ t = \text{tlookup } i\ tr$ , so the result follows from the induction hypothesis.  $\square$

Lemma lookup\_field\_in\_value :  $\forall v\ T\ i\ Ti$ ,  
**value**  $v \rightarrow$

```

empty ⊢ v \in T →
Tlookup i T = Some Ti →
∃ ti, tlookup i v = Some ti ∧ empty ⊢ ti \in Ti.
Proof with eauto.
intros v T i Ti Hval Htyp Hget.
remember (@empty ty) as Gamma.
induction Htyp; subst; try solve by inversion...
-
  simpl in Hget. simpl. destruct (beq_id i i0).
  +
    simpl. inversion Hget. subst.
    ∃ t...
  +
    destruct IHHtyp2 as [vi [Hgeti Htypi]]...
    inversion Hval... Qed.

```

## Progress

Theorem progress :  $\forall t T,$   
 $\text{empty} \vdash t \in T \rightarrow$   
**value**  $t \vee \exists t', t \Rightarrow t'.$

```

Proof with eauto.
intros t T Ht.
remember (@empty ty) as Gamma.
generalize dependent HeqGamma.
induction Ht; intros HeqGamma; subst.
-
  inversion H.
-
  left...
-
  right.
  destruct IHHt1; subst...
  +
    destruct IHHt2; subst...
    ×

    inversion H; subst; try (solve by inversion).
    ∃ ([x:=t2] t12)...
    ×

```

```

    destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
+
    destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
-

right. destruct IHHt...
+

    destruct (lookup_field_in_value _ _ _ H0 Ht H) as [ti [Hlkup _]].
    ∃ ti...
+

    destruct H0 as [t' Hstp]. ∃ (tproj t' i)...
-

left...
-

destruct IHHt1...
+
    destruct IHHt2; try reflexivity.
    ×

    left...
    ×

    right. destruct H2 as [tr' Hstp].
    ∃ (trcons i t tr')...
+

    right. destruct H1 as [t' Hstp].
    ∃ (trcons i t' tr)... Qed.

```

## Context Invariance

```

Inductive appears_free_in : id → tm → Prop :=
| afi_var : ∀ x,
  appears_free_in x (tvar x)
| afi_app1 : ∀ x t1 t2,
  appears_free_in x t1 → appears_free_in x (tapp t1 t2)
| afi_app2 : ∀ x t1 t2,

```

```

    appears_free_in x t2 → appears_free_in x (tapp t1 t2)
| afi_abs : ∀ x y T11 t12,
    y ≠ x →
    appears_free_in x t12 →
    appears_free_in x (tabs y T11 t12)
| afi_proj : ∀ x t i,
    appears_free_in x t →
    appears_free_in x (tproj t i)
| afi_rhead : ∀ x i ti tr,
    appears_free_in x ti →
    appears_free_in x (trcons i ti tr)
| afi_rtail : ∀ x i ti tr,
    appears_free_in x tr →
    appears_free_in x (trcons i ti tr).

```

Hint Constructors **appears\_free\_in**.

Lemma context\_invariance : ∀ *Gamma Gamma' t S*,  
     *Gamma* ⊢ *t* \in *S* →  
     (∀ *x*, **appears\_free\_in** *x t* → *Gamma x* = *Gamma' x*) →  
     *Gamma'* ⊢ *t* \in *S*.

Proof with eauto.

```

intros. generalize dependent Gamma'.
induction H;
  intros Gamma' Heqv...
-
  apply T_Var... rewrite ← Heqv...
-
  apply T_Abs... apply IHhas_type. intros y Hafi.
  unfold update, t_update. destruct (beq_idP x y)...
-
  apply T_App with T1...
-
  apply T_RCons... Qed.

```

Lemma free\_in\_context : ∀ *x t T Gamma*,  
     **appears\_free\_in** *x t* →  
     *Gamma* ⊢ *t* \in *T* →  
     ∃ *T'*, *Gamma x* = **Some** *T'*.

Proof with eauto.

```

intros x t T Gamma Hafi Htyp.
induction Htyp; inversion Hafi; subst...
-
  destruct IHHtyp as [T' Hctx]... ∃ T'.
  unfold update, t_update in Hctx.

```



rewrite false\_beq\_id in *Hctx*...  
 Qed.

## Preservation

Lemma substitution\_preserves\_typing :  $\forall \text{ Gamma } x \text{ U } v \text{ t } S,$   
 (update *Gamma* *x* *U*)  $\vdash t \setminus \text{in } S \rightarrow$   
 empty  $\vdash v \setminus \text{in } U \rightarrow$   
*Gamma*  $\vdash ([x:=v]t) \setminus \text{in } S.$

Proof with eauto.

```

intros Gamma x U v t S Htypt Htypv.
generalize dependent Gamma. generalize dependent S.
induction t;
  intros S Gamma Htypt; simpl; inversion Htypt; subst...
-
  simpl. rename i into y.
  unfold update, t_update in H0. destruct (beq_idP x y) as [Hxy|Hxy].
  +

    subst.
    inversion H0; subst. clear H0.
    eapply context_invariance...
    intros x Hcontra.
    destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
    inversion HT'.
  +

    apply T_Var...
  -
    rename i into y. rename t into T11.
    apply T_Abs...
    destruct (beq_idP x y) as [Hxy|Hxy].
  +

    eapply context_invariance...
    subst.
    intros x Hafi. unfold update, t_update.
    destruct (beq_id y x)...
  +

    apply IHt. eapply context_invariance...
    intros z Hafi. unfold update, t_update.
    destruct (beq_idP y z)...
```

```

      subst. rewrite false_beq_id...
-
    apply T_RCons... inversion  $H7$ ; subst; simpl...
Qed.
Theorem preservation :  $\forall t t' T,$ 
  empty  $\vdash t \setminus \text{in } T \rightarrow$ 
   $t ==> t' \rightarrow$ 
  empty  $\vdash t' \setminus \text{in } T.$ 
Proof with eauto.
  intros  $t t' T HT.$ 
  remember (@empty ty) as  $\Gamma$ . generalize dependent  $\text{Heq}\Gamma$ .
  generalize dependent  $t'$ .
  induction  $HT$ ;
    intros  $t' \text{Heq}\Gamma HE$ ; subst; inversion  $HE$ ; subst...
-

  inversion  $HE$ ; subst...
+

  apply substitution_preserves_typing with  $T1$ ...
  inversion  $HT1$ ...
-

  destruct (lookup_field_in_value _ _ _  $H2 HT H$ )
    as [ $vi$  [ $Hget Htyp$ ]].
  rewrite  $H4$  in  $Hget$ . inversion  $Hget$ . subst...
-

  apply T_RCons... eapply step_preserves_record_tm...
Qed.
□

```

End STLCExtendedRecords.

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500 (*Wed, 17 Feb 2016*)

# Chapter 31

## Library References

### 31.1 References: Typing Mutable References

```
Require Import Coq.Arith.Arith.  
Require Import Coq.omega.Omega.  
Require Import Coq.Lists.List.  
Import ListNotations.  
  
Require Import Sflib.  
Require Import Maps.  
Require Import Smallstep.
```

So far, we have considered a variety of *pure* language features, including functional abstraction, basic types such as numbers and booleans, and structured types such as records and variants. These features form the backbone of most programming languages – including purely functional languages such as Haskell, “mostly functional” languages such as ML, imperative languages such as C, and object-oriented languages such as Java.

Most practical programming languages also include various *impure* features that cannot be described in the simple semantic framework we have used so far. In particular, besides just yielding results, evaluation of terms in these languages may assign to mutable variables (reference cells, arrays, mutable record fields, etc.), perform input and output to files, displays, or network connections, make non-local transfers of control via exceptions, jumps, or continuations, engage in inter-process synchronization and communication, and so on. In the literature on programming languages, such “side effects” of computation are more generally referred to as *computational effects*.

In this chapter, we’ll see how one sort of computational effect – mutable references – can be added to the calculi we have studied. The main extension will be dealing explicitly with a *store* (or *heap*). This extension is straightforward to define; the most interesting part is the refinement we need to make to the statement of the type preservation theorem.

## 31.2 Definitions

Pretty much every programming language provides some form of assignment operation that changes the contents of a previously allocated piece of storage. (Coq’s internal language is a rare exception!)

In some languages – notably ML and its relatives – the mechanisms for name-binding and those for assignment are kept separate. We can have a variable  $x$  whose *value* is the number 5, or we can have a variable  $y$  whose value is a *reference* (or *pointer*) to a mutable cell whose current contents is 5. These are different things, and the difference is visible to the programmer. We can add  $x$  to another number, but not assign to it. We can use  $y$  directly to assign a new value to the cell that it points to (by writing  $y:=84$ ), but we cannot use it directly as an argument to an operation like  $+$ . Instead, we must explicitly *dereference* it, writing  $!y$  to obtain its current contents.

In most other languages – in particular, in all members of the C family, including Java – *every* variable name refers to a mutable cell, and the operation of dereferencing a variable to obtain its current contents is implicit.

For purposes of formal study, it is useful to keep these mechanisms separate. The development in this chapter will closely follow ML’s model. Applying the lessons learned here to C-like languages is a straightforward matter of collapsing some distinctions and rendering some operations such as dereferencing implicit instead of explicit.

In this chapter, we study adding mutable references to the simply-typed lambda calculus with natural numbers.

## 31.3 Syntax

Module STLCREF.

The basic operations on references are *allocation*, *dereferencing*, and *assignment*.

- To allocate a reference, we use the *ref* operator, providing an initial value for the new cell. For example, *ref* 5 creates a new cell containing the value 5, and evaluates to a reference to that cell.
- To read the current value of this cell, we use the dereferencing operator  $!$ ; for example,  $!(ref\ 5)$  evaluates to 5.
- To change the value stored in a cell, we use the assignment operator. If  $r$  is a reference,  $r := 7$  will store the value 7 in the cell referenced by  $r$ . However,  $r := 7$  evaluates to the trivial value **unit**; it exists only to have the *side effect* of modifying the contents of a cell.

## Types

We start with the simply typed lambda calculus over the natural numbers. To the base natural number type and arrow types we need to add two more types to deal with references. First, we need the *unit type*, which we will use as the result type of an assignment operation. We then add *reference types*. If  $T$  is a type, then  $\text{Ref } T$  is the type of references which point to a cell holding values of type  $T$ .  $T ::= \text{Nat} \mid \text{Unit} \mid T \rightarrow T \mid \text{Ref } T$

```
Inductive ty : Type :=
| TNat : ty
| TUnit : ty
| TArrow : ty → ty → ty
| TRef : ty → ty.
```

## Terms

Besides variables, abstractions, applications, natural-number-related terms, and **unit**, we need four more sorts of terms in order to handle mutable references:

$t ::= \dots$	Terms
$\text{ref } t$	allocation
$!t$	dereference
$t := t$	assignment
$l$	location

```
Inductive tm : Type :=
```

```
| tvar : id → tm
| tapp : tm → tm → tm
| tabs : id → ty → tm → tm
| tnat : nat → tm
| tsucc : tm → tm
| tpred : tm → tm
| tmult : tm → tm → tm
| tif0 : tm → tm → tm → tm

| tunit : tm
| tref : tm → tm
| tderef : tm → tm
| tassign : tm → tm → tm
| tloc : nat → tm.
```

Intuitively...

- $\text{ref } t$  (formally,  $\text{tref } t$ ) allocates a new reference cell with the value  $t$  and evaluates to the location of the newly allocated cell;

- $!t$  (formally,  $\text{tderef } t$ ) evaluates to the contents of the cell referenced by  $t$ ;
- $t1 := t2$  (formally,  $\text{tassign } t1 \ t2$ ) assigns  $t2$  to the cell referenced by  $t1$ ; and
- $l$  (formally,  $\text{tloc } l$ ) is a reference to the cell at location  $l$ . We'll discuss locations later.

In informal examples, we'll also freely use the extensions of the STLC developed in the `MoreStlc` chapter; however, to keep the proofs small, we won't bother formalizing them again here. It would be easy to do so, since there are no very interesting interactions between those features and references.

Module `EXAMPLEVARIABLES`.

Definition `x` := `ld 0`.

Definition `y` := `ld 1`.

Definition `r` := `ld 2`.

Definition `s` := `ld 3`.

End `EXAMPLEVARIABLES`.

## Typing (Preview)

Informally, the typing rules for allocation, dereferencing, and assignment will look like this:

$\Gamma \vdash t1 : T1$

---

(T\_Ref)  $\Gamma \vdash \text{ref } t1 : \text{Ref } T1$   
 $\Gamma \vdash t1 : \text{Ref } T1$

---

(T\_Deref)  $\Gamma \vdash !t1 : T1$   
 $\Gamma \vdash t1 : \text{Ref } T1 \quad \Gamma \vdash t2 : T1$

---

(T\_Assign)  $\Gamma \vdash t1 := t2 : \text{Unit}$  The rule for locations will require a bit more machinery, and this will motivate some changes to the other rules; we'll come back to this later.

## Values and Substitution

Besides abstractions and numbers, we have two new types of values: the unit value, and locations.

Inductive **value** : **tm**  $\rightarrow$  Prop :=

| **v\_abs** :  $\forall x \ T \ t,$   
     **value** (**tabs**  $x \ T \ t$ )  
 | **v\_nat** :  $\forall n,$   
     **value** (**tnat**  $n$ )  
 | **v\_unit** :  
     **value** **tunit**  
 | **v\_loc** :  $\forall l,$

**value** (tloc *l*).

Hint Constructors **value**.

Extending substitution to handle the new syntax of terms is straightforward.

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' =>
    if beq_id x x' then s else t
  | tapp t1 t2 =>
    tapp (subst x s t1) (subst x s t2)
  | tabs x' T t1 =>
    if beq_id x x' then t else tabs x' T (subst x s t1)
  | tnat n =>
    t
  | tsucc t1 =>
    tsucc (subst x s t1)
  | tpred t1 =>
    tpred (subst x s t1)
  | tmult t1 t2 =>
    tmult (subst x s t1) (subst x s t2)
  | tif0 t1 t2 t3 =>
    tif0 (subst x s t1) (subst x s t2) (subst x s t3)
  | tunit =>
    t
  | tref t1 =>
    tref (subst x s t1)
  | tderef t1 =>
    tderef (subst x s t1)
  | tassign t1 t2 =>
    tassign (subst x s t1) (subst x s t2)
  | tloc _ =>
    t
  end.
```

Notation "[x := s] t" := (subst x s t) (at level 20).

## 31.4 Pragmatics

### 31.4.1 Side Effects and Sequencing

The fact that the result of an assignment expression is the trivial value **unit** allows us to use a nice abbreviation for *sequencing*. For example, we can write

```
r:=succ(!r); !r
```

as an abbreviation for

```
(\x:Unit. !r) (r := succ(!r)).
```

This has the effect of evaluating two expressions in order and returning the value of the second. Restricting the type of the first expression to *Unit* helps the typechecker to catch some silly errors by permitting us to throw away the first value only if it is really guaranteed to be trivial.

Notice that, if the second expression is also an assignment, then the type of the whole sequence will be *Unit*, so we can validly place it to the left of another `;` to build longer sequences of assignments:

```
r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r
```

Formally, we introduce sequencing as a "derived form" `tseq` that expands into an abstraction and an application.

**Definition** `tseq t1 t2 :=`  
`tapp (tabs (ld 0) TUnit t2) t1.`

### 31.4.2 References and Aliasing

It is important to bear in mind the difference between the *reference* that is bound to `r` and the *cell* in the store that is pointed to by this reference.

If we make a copy of `r`, for example by binding its value to another variable `s`, what gets copied is only the *reference*, not the contents of the cell itself.

For example, after evaluating

```
let r = ref 5 in
let s = r in
s := 82;
(!r)+1
```

the cell referenced by `r` will contain the value 82, while the result of the whole expression will be 83. The references `r` and `s` are said to be *aliases* for the same cell.

The possibility of aliasing can make programs with references quite tricky to reason about. For example, the expression

```
r := 5; r := !s
```

assigns 5 to `r` and then immediately overwrites it with `s`'s current value; this has exactly the same effect as the single assignment

```
r := !s
```

*unless* we happen to do it in a context where `r` and `s` are aliases for the same cell!



### 31.4.3 Shared State

Of course, aliasing is also a large part of what makes references useful. In particular, it allows us to set up “implicit communication channels” – shared state – between different parts of a program. For example, suppose we define a reference cell and two functions that manipulate its contents:

```
let c = ref 0 in
let incc = \_:Unit. (c := succ (!c); !c) in
let decc = \_:Unit. (c := pred (!c); !c) in
...
```

Note that, since their argument types are *Unit*, the abstractions in the definitions of *incc* and *decc* are not providing any useful information to the bodies of the functions (using the wildcard `_` as the name of the bound variable is a reminder of this). Instead, their purpose is to “slow down” the execution of the function bodies: since function abstractions are values, the two `lets` are executed simply by binding these functions to the names *incc* and *decc*, rather than by actually incrementing or decrementing *c*. Later, each call to one of these functions results in its body being executed once and performing the appropriate mutation on *c*. Such functions are often called *thunks*.

In the context of these declarations, calling *incc* results in changes to *c* that can be observed by calling *decc*. For example, if we replace the ... with *(incc unit; incc unit; decc unit)*, the result of the whole program will be 1.

### 31.4.4 Objects

We can go a step further and write a *function* that creates *c*, *incc*, and *decc*, packages *incc* and *decc* together into a record, and returns this record:

```
newcounter =
  \_:Unit.
    let c = ref 0 in
    let incc = \_:Unit. (c := succ (!c); !c) in
    let decc = \_:Unit. (c := pred (!c); !c) in
    {i=incc, d=decc}
```

Now, each time we call *newcounter*, we get a new record of functions that share access to the same storage cell *c*. The caller of *newcounter* can’t get at this storage cell directly, but can affect it indirectly by calling the two functions. In other words, we’ve created a simple form of *object*.

```
let c1 = newcounter unit in
let c2 = newcounter unit in
// Note that we’ve allocated two separate storage cells now!
let r1 = c1.i unit in
let r2 = c2.i unit in
r2 // yields 1, not 2!
```

**Exercise: 1 star (store\_draw)** Draw (on paper) the contents of the store at the point in execution where the first two `lets` have finished and the third one is about to begin.

□

### 31.4.5 References to Compound Types

A reference cell need not contain just a number: the primitives we've defined above allow us to create references to values of any type, including functions. For example, we can use references to functions to give a (not very efficient) implementation of arrays of numbers, as follows. Write *NatArray* for the type *Ref (Nat → Nat)*.

Recall the `equal` function from the `MoreStlc` chapter:

```
equal =
  fix
    (\eq:Nat->Nat->Bool.
      \m:Nat. \n:Nat.
        if m=0 then iszero n
        else if n=0 then false
        else eq (pred m) (pred n))
```

Now, to build a new array, we allocate a reference cell and fill it with a function that, when given an index, always returns 0.

```
newarray = \_:Unit. ref (\n:Nat.0)
```

To look up an element of an array, we simply apply the function to the desired index.

```
lookup = \a:NatArray. \n:Nat. (!a) n
```

The interesting part of the encoding is the `update` function. It takes an array, an index, and a new value to be stored at that index, and does its job by creating (and storing in the reference) a new function that, when it is asked for the value at this very index, returns the new value that was given to `update`, and on all other indices passes the lookup to the function that was previously stored in the reference.

```
update = \a:NatArray. \m:Nat. \v:Nat.
  let oldf = !a in
  a := (\n:Nat. if equal m n then v else oldf n);
```

References to values containing other references can also be very useful, allowing us to define data structures such as mutable lists and trees.

**Exercise: 2 stars (compact\_update)** If we defined `update` more compactly like this

```
update = \a:NatArray. \m:Nat. \v:Nat.
  a := (\n:Nat. if equal m n then v else (!a) n)
```

would it behave the same?

□

### 31.4.6 Null References

There is one more difference between our references and C-style mutable variables: in C-like languages, variables holding pointers into the heap may sometimes have the value *NULL*. Dereferencing such a "null pointer" is an error, and results in an exception (Java) or in termination of the program (C).

Null pointers cause significant trouble in C-like languages: the fact that any pointer might be null means that any dereference operation in the program can potentially fail. However, even in ML-like languages, there are occasionally situations where we may or may not have a valid pointer in our hands. Fortunately, there is no need to extend the basic mechanisms of references to achieve this: the sum types introduced in the **MoreStlc** chapter already give us what we need.

First, we can use sums to build an analog of the **option** types introduced in the **Lists** chapter. Define *Option T* to be an abbreviation for *Unit + T*.

Then a "nullable reference to a *T*" is simply an element of the type *Option (Ref T)*.

### 31.4.7 Garbage Collection

A last issue that we should mention before we move on with formalizing references is storage *de*-allocation. We have not provided any primitives for freeing reference cells when they are no longer needed. Instead, like many modern languages (including ML and Java) we rely on the run-time system to perform *garbage collection*, collecting and reusing cells that can no longer be reached by the program.

This is *not* just a question of taste in language design: it is extremely difficult to achieve type safety in the presence of an explicit deallocation operation. The reason for this is the familiar *dangling reference* problem: we allocate a cell holding a number, save a reference to it in some data structure, use it for a while, then deallocate it and allocate a new cell holding a boolean, possibly reusing the same storage. Now we can have two names for the same storage cell – one with type *Ref Nat* and the other with type *Ref Bool*.

**Exercise: 1 star (type\_safety\_violation)** Show how this can lead to a violation of type safety.

□

## 31.5 Operational Semantics

### 31.5.1 Locations

The most subtle aspect of the treatment of references appears when we consider how to formalize their operational behavior. One way to see why is to ask, "What should be the *values* of type *Ref T<sub>i</sub>*?" The crucial observation that we need to take into account is that

evaluating a *ref* operator should *do* something – namely, allocate some storage – and the result of the operation should be a reference to this storage.

What, then, is a reference?

The run-time store in most programming language implementations is essentially just a big array of bytes. The run-time system keeps track of which parts of this array are currently in use; when we need to allocate a new reference cell, we allocate a large enough segment from the free region of the store (4 bytes for integer cells, 8 bytes for cells storing *Floats*, etc.), mark it as being used, and return the index (typically, a 32- or 64-bit integer) of the start of the newly allocated region. These indices are references.

For present purposes, there is no need to be quite so concrete. We can think of the store as an array of *values*, rather than an array of bytes, abstracting away from the different sizes of the run-time representations of different values. A reference, then, is simply an index into the store. (If we like, we can even abstract away from the fact that these indices are numbers, but for purposes of formalization in Coq it is a bit more convenient to use numbers.) We'll use the word *location* instead of *reference* or *pointer* from now on to emphasize this abstract quality.

Treating locations abstractly in this way will prevent us from modeling the *pointer arithmetic* found in low-level languages such as C. This limitation is intentional. While pointer arithmetic is occasionally very useful, especially for implementing low-level services such as garbage collectors, it cannot be tracked by most type systems: knowing that location *n* in the store contains a *float* doesn't tell us anything useful about the type of location *n*+4. In C, pointer arithmetic is a notorious source of type safety violations.

### 31.5.2 Stores

Recall that, in the small-step operational semantics for IMP, the step relation needed to carry along an auxiliary state in addition to the program being executed. In the same way, once we have added reference cells to the STLC, our step relation must carry along a store to keep track of the contents of reference cells.

We could re-use the same functional representation we used for states in IMP, but for carrying out the proofs in this chapter it is actually more convenient to represent a store simply as a *list* of values. (The reason we couldn't use this representation before is that, in IMP, a program could modify any location at any time, so states had to be ready to map *any* variable to a value. However, in the STLC with references, the only way to create a reference cell is with **tr<sub>ref</sub>** *t1*, which puts the value of *t1* in a new reference cell and evaluates to the location of the newly created reference cell. When evaluating such an expression, we can just add a new reference cell to the end of the list representing the store.)

**Definition** *store* := **list** **tm**.

We use **store\_lookup** *n st* to retrieve the value of the reference cell at location *n* in the store *st*. Note that we must give a default value to **nth** in case we try looking up an index which is too large. (In fact, we will never actually do this, but proving it will of course require some work!)

Definition store\_lookup (n:nat) (st:store) :=  
 nth n st tunit.

To update the store, we use the `replace` function, which replaces the contents of a cell at a particular index.

```
Fixpoint replace {A:Type} (n:nat) (x:A) (l:list A) : list A :=
  match l with
  | nil => nil
  | h :: t =>
    match n with
    | 0 => x :: t
    | S n' => h :: replace n' x t
    end
  end.
```

Of course, we also need some boring lemmas about `replace`, which are also fairly straightforward to prove.

Lemma replace\_nil :  $\forall A n (x:A)$ ,  
 replace n x nil = nil.

Proof.

destruct n; auto.

Qed.

Lemma length\_replace :  $\forall A n x (l:list A)$ ,  
 length (replace n x l) = length l.

Proof with auto.

intros A n x l. generalize dependent n.

induction l; intros n.

destruct n...

destruct n...

simpl. rewrite IHl...

Qed.

Lemma lookup\_replace\_eq :  $\forall l t st$ ,  
 l < length st  $\rightarrow$   
 store\_lookup l (replace l t st) = t.

Proof with auto.

intros l t st.

unfold store\_lookup.

generalize dependent l.

induction st as [|t' st']; intros l Hlen.

-

inversion Hlen.

-

destruct l; simpl...

```

    apply IHst'. simpl in Hlen. omega.
Qed.
Lemma lookup_replace_neq : ∀ l1 l2 t st,
  l1 ≠ l2 →
  store_lookup l1 (replace l2 t st) = store_lookup l1 st.
Proof with auto.
  unfold store_lookup.
  induction l1 as [|l1']; intros l2 t st Hneq.
  -
    destruct st.
    + rewrite replace_nil...
    + destruct l2... contradict Hneq...
  -
    destruct st as [|t2 st2].
    + destruct l2...
    +
      destruct l2...
      simpl; apply IHl1'...
Qed.

```

### 31.5.3 Reduction

Next, we need to extend our operational semantics to take stores into account. Since the result of evaluating an expression will in general depend on the contents of the store in which it is evaluated, the evaluation rules should take not just a term but also a store as argument. Furthermore, since the evaluation of a term may cause side effects on the store that may affect the evaluation of other terms in the future, the evaluation rules need to return a new store. Thus, the shape of the single-step evaluation relation changes from  $t ==> t'$  to  $t / st ==> t' / st'$ , where  $st$  and  $st'$  are the starting and ending states of the store.

To carry through this change, we first need to augment all of our existing evaluation rules with stores: value v2

---


$$\text{(ST\_AppAbs)} \quad (\backslash x:T.t12) \ v2 / st ==> x:=v2t12 / st$$

$$t1 / st ==> t1' / st'$$


---

$$\text{(ST\_App1)} \quad t1 \ t2 / st ==> t1' \ t2 / st'$$

$$\text{value } v1 \ t2 / st ==> t2' / st'$$


---

(ST\_App2)  $v1 \ t2 / st ==> v1 \ t2' / st'$  Note that the first rule here returns the store unchanged: function application, in itself, has no side effects. The other two rules simply propagate side effects from premise to conclusion.

Now, the result of evaluating a *ref* expression will be a fresh location; this is why we included locations in the syntax of terms and in the set of values.

It is crucial to note that making this extension to the syntax of terms does not mean that we intend *programmers* to write terms involving explicit, concrete locations: such terms will arise only as intermediate results of evaluation. This may initially seem odd, but really it follows naturally from our design decision to represent the result of every evaluation step by a modified term. If we had chosen a more "machine-like" model for evaluation, e.g., with an explicit stack to contain values of bound identifiers, then the idea of adding locations to the set of allowed values would probably seem more obvious.

In terms of this expanded syntax, we can state evaluation rules for the new constructs that manipulate locations and the store. First, to evaluate a dereferencing expression  $!t1$ , we must first reduce  $t1$  until it becomes a value:  $t1 / st \implies t1' / st'$

---

(ST\_Deref)  $!t1 / st \implies !t1' / st'$  Once  $t1$  has finished reducing, we should have an expression of the form  $!l$ , where  $l$  is some location. (A term that attempts to dereference any other sort of value, such as a function or **unit**, is erroneous, as is a term that tries to dereference a location that is larger than the size  $|st|$  of the currently allocated store; the evaluation rules simply get stuck in this case. The type safety properties that we'll establish below assure us that well-typed terms will never misbehave in this way.)  $l < |st|$

---

(ST\_DerefLoc)  $!(loc\ l) / st \implies lookup\ l\ st / st$

Next, to evaluate an assignment expression  $t1 := t2$ , we must first evaluate  $t1$  until it becomes a value (a location), and then evaluate  $t2$  until it becomes a value (of any sort):  $t1 / st \implies t1' / st'$

---

(ST\_Assign1)  $t1 := t2 / st \implies t1' := t2 / st'$   
 $t2 / st \implies t2' / st'$

---

(ST\_Assign2)  $v1 := t2 / st \implies v1 := t2' / st'$  Once we have finished with  $t1$  and  $t2$ , we have an expression of the form  $l := v2$ , which we execute by updating the store to make location  $l$  contain  $v2$ :  $l < |st|$

---

(ST\_Assign)  $loc\ l := v2 / st \implies unit / l := v2\ st$  The notation  $[l := v2]st$  means "the store that maps  $l$  to  $v2$  and maps all other locations to the same thing as  $st$ ." Note that the term resulting from this evaluation step is just **unit**; the interesting result is the updated store.)

---

Finally, to evaluate an expression of the form *ref*  $t1$ , we first evaluate  $t1$  until it becomes a value:  $t1 / st \implies t1' / st'$

---

(ST\_Ref)  $ref\ t1 / st \implies ref\ t1' / st'$  Then, to evaluate the *ref* itself, we choose a fresh location at the end of the current store – i.e., location  $|st|$  – and yield a new store that extends  $st$  with the new value  $v1$ .

---

(ST\_RefValue)  $\text{ref } v1 / st \implies \text{loc } |st| / st, v1$  The value resulting from this step is the newly allocated location itself. (Formally,  $st, v1$  means  $st ++ v1 :: \text{nil}$  – i.e., to add a new reference cell to the store, we append it to the end.)

Note that these evaluation rules do not perform any kind of garbage collection: we simply allow the store to keep growing without bound as evaluation proceeds. This does not affect the correctness of the results of evaluation (after all, the definition of "garbage" is precisely parts of the store that are no longer reachable and so cannot play any further role in evaluation), but it means that a naive implementation of our evaluator might sometimes run out of memory where a more sophisticated evaluator would be able to continue by reusing locations whose contents have become garbage.

Formally...

Reserved Notation " $t1 \text{ '/' } st \implies t2 \text{ '/' } st$ "  
(at level 40,  $st1$  at level 39,  $t2$  at level 39).

Import *ListNotations*.

Inductive **step** : **tm**  $\times$  store  $\rightarrow$  **tm**  $\times$  store  $\rightarrow$  Prop :=  
| ST\_AppAbs :  $\forall x \ T \ t12 \ v2 \ st,$   
  **value**  $v2 \rightarrow$   
   $\text{tapp} (\text{tabs } x \ T \ t12) \ v2 / st \implies [x:=v2] t12 / st$   
| ST\_App1 :  $\forall t1 \ t1' \ t2 \ st \ st',$   
   $t1 / st \implies t1' / st' \rightarrow$   
   $\text{tapp } t1 \ t2 / st \implies \text{tapp } t1' \ t2 / st'$   
| ST\_App2 :  $\forall v1 \ t2 \ t2' \ st \ st',$   
  **value**  $v1 \rightarrow$   
   $t2 / st \implies t2' / st' \rightarrow$   
   $\text{tapp } v1 \ t2 / st \implies \text{tapp } v1 \ t2' / st'$   
| ST\_SuccNat :  $\forall n \ st,$   
   $\text{tsucc} (\text{tnat } n) / st \implies \text{tnat } (\text{S } n) / st$   
| ST\_Succ :  $\forall t1 \ t1' \ st \ st',$   
   $t1 / st \implies t1' / st' \rightarrow$   
   $\text{tsucc } t1 / st \implies \text{tsucc } t1' / st'$   
| ST\_PredNat :  $\forall n \ st,$   
   $\text{tpred} (\text{tnat } n) / st \implies \text{tnat } (\text{pred } n) / st$   
| ST\_Pred :  $\forall t1 \ t1' \ st \ st',$   
   $t1 / st \implies t1' / st' \rightarrow$   
   $\text{tpred } t1 / st \implies \text{tpred } t1' / st'$   
| ST\_MultNats :  $\forall n1 \ n2 \ st,$   
   $\text{tmult} (\text{tnat } n1) (\text{tnat } n2) / st \implies \text{tnat } (\text{mult } n1 \ n2) / st$   
| ST\_Mult1 :  $\forall t1 \ t2 \ t1' \ st \ st',$   
   $t1 / st \implies t1' / st' \rightarrow$   
   $\text{tmult } t1 \ t2 / st \implies \text{tmult } t1' \ t2 / st'$   
| ST\_Mult2 :  $\forall v1 \ t2 \ t2' \ st \ st',$   
  **value**  $v1 \rightarrow$



```

      t2 / st ==> t2' / st' →
      tmult v1 t2 / st ==> tmult v1 t2' / st'
| ST_If0 : ∀ t1 t1' t2 t3 st st',
  t1 / st ==> t1' / st' →
  tif0 t1 t2 t3 / st ==> tif0 t1' t2 t3 / st'
| ST_If0_Zero : ∀ t2 t3 st,
  tif0 (tnat 0) t2 t3 / st ==> t2 / st
| ST_If0_Nonzero : ∀ n t2 t3 st,
  tif0 (tnat (S n)) t2 t3 / st ==> t3 / st
| ST_RefValue : ∀ v1 st,
  value v1 →
  tref v1 / st ==> tloc (length st) / (st ++ v1 :: nil)
| ST_Ref : ∀ t1 t1' st st',
  t1 / st ==> t1' / st' →
  tref t1 / st ==> tref t1' / st'
| ST_DerefLoc : ∀ st l,
  l < length st →
  tderef (tloc l) / st ==> store_lookup l st / st
| ST_Deref : ∀ t1 t1' st st',
  t1 / st ==> t1' / st' →
  tderef t1 / st ==> tderef t1' / st'
| ST_Assign : ∀ v2 l st,
  value v2 →
  l < length st →
  tassign (tloc l) v2 / st ==> tunit / replace l v2 st
| ST_Assign1 : ∀ t1 t1' t2 st st',
  t1 / st ==> t1' / st' →
  tassign t1 t2 / st ==> tassign t1' t2 / st'
| ST_Assign2 : ∀ v1 t2 t2' st st',
  value v1 →
  t2 / st ==> t2' / st' →
  tassign v1 t2 / st ==> tassign v1 t2' / st'

```

where "t1 '/' st1 '==>' t2 '/' st2" := (step (t1, st1) (t2, st2)).

One slightly ugly point should be noted here: In the ST\_RefValue rule, we extend the state by writing  $st ++ v1 :: \text{nil}$  rather than the more natural  $st ++ [v1]$ . The reason for this is that the notation we've defined for substitution uses square brackets, which clash with the standard library's notation for lists.

Hint Constructors step.

Definition multistep := (multi step).

Notation "t1 '/' st '==>\*' t2 '/' st'" := (multistep (t1, st) (t2, st'))  
 (at level 40, st at level 39, t2 at level 39).

## 31.6 Typing

Our contexts for free variables will be exactly the same as for the STLC, partial maps from identifiers to types.

Definition  $\text{context} := \text{partial\_map } \mathbf{ty}$ .

### 31.6.1 Store typings

Having extended our syntax and evaluation rules to accommodate references, our last job is to write down typing rules for the new constructs – and, of course, to check that they are sound. Naturally, the key question is, “What is the type of a location?”

First of all, notice that we do *not* need to answer this question for purposes of type-checking the terms that programmers actually write. Concrete location constants arise only in terms that are the intermediate results of evaluation; they are not in the language that programmers write. So we only need to determine the type of a location when we’re in the middle of an evaluation sequence, e.g., trying to apply the progress or preservation lemmas. Thus, even though we normally think of typing as a *static* program property, it makes sense for the typing of locations to depend on the *dynamic* progress of the program too.

As a first try, note that when we evaluate a term containing concrete locations, the type of the result depends on the contents of the store that we start with. For example, if we evaluate the term  $!(\text{loc } 1)$  in the store  $[\mathbf{unit}, \mathbf{unit}]$ , the result is  $\mathbf{unit}$ ; if we evaluate the same term in the store  $[\mathbf{unit}, \backslash x:\text{Unit}.x]$ , the result is  $\backslash x:\text{Unit}.x$ . With respect to the former store, the location 1 has type  $\text{Unit}$ , and with respect to the latter it has type  $\text{Unit} \rightarrow \text{Unit}$ . This observation leads us immediately to a first attempt at a typing rule for locations:  $\Gamma \vdash \text{lookup } l \text{ st} : T1$

---

$\Gamma \vdash \text{loc } l : \text{Ref } T1$  That is, to find the type of a location  $l$ , we look up the current contents of  $l$  in the store and calculate the type  $T1$  of the contents. The type of the location is then  $\text{Ref } T1$ .

Having begun in this way, we need to go a little further to reach a consistent state. In effect, by making the type of a term depend on the store, we have changed the typing relation from a three-place relation (between contexts, terms, and types) to a four-place relation (between contexts, *stores*, terms, and types). Since the store is, intuitively, part of the context in which we calculate the type of a term, let’s write this four-place relation with the store to the left of the turnstile:  $\Gamma; st \vdash t : T$ . Our rule for typing references now has the form  $\Gamma; st \vdash \text{lookup } l \text{ st} : T1$

---

$\Gamma; st \vdash \text{loc } l : \text{Ref } T1$  and all the rest of the typing rules in the system are extended similarly with stores. The other rules do not need to do anything interesting with their stores – just pass them from premise to conclusion.

However, there are two problems with this rule. First, typechecking is rather inefficient, since calculating the type of a location  $l$  involves calculating the type of the current contents

$v$  of  $l$ . If  $l$  appears many times in a term  $t$ , we will re-calculate the type of  $v$  many times in the course of constructing a typing derivation for  $t$ . Worse, if  $v$  itself contains locations, then we will have to recalculate *their* types each time they appear.

Second, the proposed typing rule for locations may not allow us to derive anything at all, if the store contains a *cycle*. For example, there is no finite typing derivation for the location 0 with respect to this store:

$[\backslash x:\text{Nat}. (!(\text{loc } 1)) \ x, \backslash x:\text{Nat}. (!(\text{loc } 0)) \ x]$

**Exercise: 2 stars (cyclic\_store)** Can you find a term whose evaluation will create this particular cyclic store?

□

Both of these problems arise from the fact that our proposed typing rule for locations requires us to recalculate the type of a location every time we mention it in a term. But this, intuitively, should not be necessary. After all, when a location is first created, we know the type of the initial value that we are storing into it. Suppose we are willing to enforce the invariant that the type of the value contained in a given location *never changes*; that is, although we may later store other values into this location, those other values will always have the same type as the initial one. In other words, we always have in mind a single, definite type for every location in the store, which is fixed when the location is allocated. Then these intended types can be collected together as a *store typing* —a finite function mapping locations to types.

As usual, this *conservative* typing restriction on allowed updates means that we will rule out as ill-typed some programs that could evaluate perfectly well without getting stuck.

Just like we did for stores, we will represent a store type simply as a list of types: the type at index  $i$  records the type of the value stored in cell  $i$ .

**Definition**  $\text{store\_ty} := \text{list } \text{ty}$ .

The  $\text{store\_Tlookup}$  function retrieves the type at a particular index.

**Definition**  $\text{store\_Tlookup } (n:\text{nat}) \ (ST:\text{store\_ty}) :=$   
 $\text{nth } n \ ST \ \text{TUnit}$ .

Suppose we are *given* a store typing  $ST$  describing the store  $st$  in which some term  $t$  will be evaluated. Then we can use  $ST$  to calculate the type of the result of  $t$  without ever looking directly at  $st$ . For example, if  $ST$  is  $[Unit, Unit \rightarrow Unit]$ , then we may immediately infer that  $!(\text{loc } 1)$  has type  $Unit \rightarrow Unit$ . More generally, the typing rule for locations can be reformulated in terms of store typings like this:  $l < |ST|$

---

$\Gamma; ST \vdash \text{loc } l : \text{Ref } (\text{lookup } l \ ST)$

That is, as long as  $l$  is a valid location (it is less than the length of  $ST$ ), we can compute the type of  $l$  just by looking it up in  $ST$ . Typing is again a four-place relation, but it is parameterized on a store *typing* rather than a concrete store. The rest of the typing rules are analogously augmented with store typings.

### 31.6.2 The Typing Relation

We can now give the typing relation for the STLC with references. Here, again, are the rules we're adding to the base STLC (with numbers and *Unit*):

$$1 < |\text{ST}|$$

---

(T\_Loc)  $\Gamma; \text{ST} \vdash \text{loc } l : \text{Ref} \text{ (lookup } l \text{ ST)}$   
 $\Gamma; \text{ST} \vdash t1 : T1$

---

(T\_Ref)  $\Gamma; \text{ST} \vdash \text{ref } t1 : \text{Ref } T1$   
 $\Gamma; \text{ST} \vdash t1 : \text{Ref } T11$

---

(T\_Deref)  $\Gamma; \text{ST} \vdash !t1 : T11$   
 $\Gamma; \text{ST} \vdash t1 : \text{Ref } T11 \quad \Gamma; \text{ST} \vdash t2 : T11$

---

(T\_Assign)  $\Gamma; \text{ST} \vdash t1 := t2 : \text{Unit}$

---

Reserved Notation " $\Gamma; \text{ST} \vdash t \text{ in } T$ " (at level 40).

Inductive **has\_type** : context  $\rightarrow$  store\_ty  $\rightarrow$  tm  $\rightarrow$  ty  $\rightarrow$  Prop :=

```
| T_Var :  $\forall \Gamma \text{ ST } x \ T,$ 
     $\Gamma \ x = \text{Some } T \rightarrow$ 
     $\Gamma; \text{ST} \vdash (\text{tvar } x) \text{ in } T$ 
| T_Abs :  $\forall \Gamma \text{ ST } x \ T11 \ T12 \ t12,$ 
     $(\text{update } \Gamma \ x \ T11); \text{ST} \vdash t12 \text{ in } T12 \rightarrow$ 
     $\Gamma; \text{ST} \vdash (\text{tabs } x \ T11 \ t12) \text{ in } (\text{TArrow } T11 \ T12)$ 
| T_App :  $\forall T1 \ T2 \ \Gamma \text{ ST } t1 \ t2,$ 
     $\Gamma; \text{ST} \vdash t1 \text{ in } (\text{TArrow } T1 \ T2) \rightarrow$ 
     $\Gamma; \text{ST} \vdash t2 \text{ in } T1 \rightarrow$ 
     $\Gamma; \text{ST} \vdash (\text{tapp } t1 \ t2) \text{ in } T2$ 
| T_Nat :  $\forall \Gamma \text{ ST } n,$ 
     $\Gamma; \text{ST} \vdash (\text{tnat } n) \text{ in } \text{TNat}$ 
| T_Succ :  $\forall \Gamma \text{ ST } t1,$ 
     $\Gamma; \text{ST} \vdash t1 \text{ in } \text{TNat} \rightarrow$ 
     $\Gamma; \text{ST} \vdash (\text{tsucc } t1) \text{ in } \text{TNat}$ 
| T_Pred :  $\forall \Gamma \text{ ST } t1,$ 
     $\Gamma; \text{ST} \vdash t1 \text{ in } \text{TNat} \rightarrow$ 
     $\Gamma; \text{ST} \vdash (\text{tpred } t1) \text{ in } \text{TNat}$ 
| T_Mult :  $\forall \Gamma \text{ ST } t1 \ t2,$ 
     $\Gamma; \text{ST} \vdash t1 \text{ in } \text{TNat} \rightarrow$ 
     $\Gamma; \text{ST} \vdash t2 \text{ in } \text{TNat} \rightarrow$ 
     $\Gamma; \text{ST} \vdash (\text{tmult } t1 \ t2) \text{ in } \text{TNat}$ 
| T_If0 :  $\forall \Gamma \text{ ST } t1 \ t2 \ t3 \ T,$ 
     $\Gamma; \text{ST} \vdash t1 \text{ in } \text{TNat} \rightarrow$ 
```

$$\begin{array}{l}
\text{Gamma}; ST \vdash t2 \text{ \texttt{\textbackslash in} } T \rightarrow \\
\text{Gamma}; ST \vdash t3 \text{ \texttt{\textbackslash in} } T \rightarrow \\
\text{Gamma}; ST \vdash (\text{tif0 } t1 \ t2 \ t3) \text{ \texttt{\textbackslash in} } T \\
| \text{ T\_Unit} : \forall \text{ Gamma } ST, \\
\quad \text{Gamma}; ST \vdash \text{tunit} \text{ \texttt{\textbackslash in} } \text{TUnit} \\
| \text{ T\_Loc} : \forall \text{ Gamma } ST \ l, \\
\quad l < \text{length } ST \rightarrow \\
\quad \text{Gamma}; ST \vdash (\text{tloc } l) \text{ \texttt{\textbackslash in} } (\text{TRef } (\text{store\_Tlookup } l \ ST)) \\
| \text{ T\_Ref} : \forall \text{ Gamma } ST \ t1 \ T1, \\
\quad \text{Gamma}; ST \vdash t1 \text{ \texttt{\textbackslash in} } T1 \rightarrow \\
\quad \text{Gamma}; ST \vdash (\text{tref } t1) \text{ \texttt{\textbackslash in} } (\text{TRef } T1) \\
| \text{ T\_Deref} : \forall \text{ Gamma } ST \ t1 \ T11, \\
\quad \text{Gamma}; ST \vdash t1 \text{ \texttt{\textbackslash in} } (\text{TRef } T11) \rightarrow \\
\quad \text{Gamma}; ST \vdash (\text{tderef } t1) \text{ \texttt{\textbackslash in} } T11 \\
| \text{ T\_Assign} : \forall \text{ Gamma } ST \ t1 \ t2 \ T11, \\
\quad \text{Gamma}; ST \vdash t1 \text{ \texttt{\textbackslash in} } (\text{TRef } T11) \rightarrow \\
\quad \text{Gamma}; ST \vdash t2 \text{ \texttt{\textbackslash in} } T11 \rightarrow \\
\quad \text{Gamma}; ST \vdash (\text{tassign } t1 \ t2) \text{ \texttt{\textbackslash in} } \text{TUnit}
\end{array}$$

where "Gamma ',' ST '-' t '\texttt{\textbackslash in} T" := (**has\_type** Gamma ST t T).

Hint Constructors **has\_type**.

Of course, these typing rules will accurately predict the results of evaluation only if the concrete store used during evaluation actually conforms to the store typing that we assume for purposes of typechecking. This proviso exactly parallels the situation with free variables in the STLC: the substitution lemma promises us that, if  $\text{Gamma} \vdash t : T$ , then we can replace the free variables in  $t$  with values of the types listed in  $\text{Gamma}$  to obtain a closed term of type  $T$ , which, by the type preservation theorem will evaluate to a final result of type  $T$  if it yields any result at all. (We will see later how to formalize an analogous intuition for stores and store typings.)

However, for purposes of typechecking the terms that programmers actually write, we do not need to do anything tricky to guess what store typing we should use. Recall that concrete location constants arise only in terms that are the intermediate results of evaluation; they are not in the language that programmers write. Thus, we can simply typecheck the programmer's terms with respect to the *empty* store typing. As evaluation proceeds and new locations are created, we will always be able to see how to extend the store typing by looking at the type of the initial values being placed in newly allocated cells; this intuition is formalized in the statement of the type preservation theorem below.

## 31.7 Properties

Our final task is to check that standard type safety properties continue to hold for the STLC with references. The progress theorem (“well-typed terms are not stuck”) can be stated and proved almost as for the STLC; we just need to add a few straightforward cases to the proof, dealing with the new constructs. The preservation theorem is a bit more interesting, so let’s look at it first.

### 31.7.1 Well-Typed Stores

Since we have extended both the evaluation relation (with initial and final stores) and the typing relation (with a store typing), we need to change the statement of preservation to include these parameters. Clearly, though, we cannot just add stores and store typings without saying anything about how they are related:

**Theorem** `preservation_wrong1` :  $\forall ST\ T\ t\ st\ t'\ st',$   
 $\text{empty}; ST \vdash t \text{ \texttt{\textbackslash in} } T \rightarrow$   
 $t / st \implies t' / st' \rightarrow$   
 $\text{empty}; ST \vdash t' \text{ \texttt{\textbackslash in} } T.$

**Abort.**

If we typecheck with respect to some set of assumptions about the types of the values in the store and then evaluate with respect to a store that violates these assumptions, the result will be disaster. We say that a store  $st$  is *well typed* with respect a store typing  $ST$  if the term at each location  $l$  in  $st$  has the type at location  $l$  in  $ST$ . Since only closed terms ever get stored in locations (why?), it suffices to type them in the empty context. The following definition of `store_well_typed` formalizes this.

**Definition** `store_well_typed` ( $ST$ :store\_ty) ( $st$ :store) :=  
 $\text{length } ST = \text{length } st \wedge$   
 $(\forall l, l < \text{length } st \rightarrow$   
 $\text{empty}; ST \vdash (\text{store\_lookup } l\ st) \text{ \texttt{\textbackslash in} } (\text{store\_Tlookup } l\ ST)).$

Informally, we will write  $ST \vdash st$  for `store_well_typed`  $ST\ st$ .

Intuitively, a store  $st$  is consistent with a store typing  $ST$  if every value in the store has the type predicted by the store typing. (The only subtle point is the fact that, when typing the values in the store, we supply the very same store typing to the typing relation! This allows us to type circular stores.)

**Exercise: 2 stars (store\_not\_unique)** Can you find a store  $st$ , and two different store typings  $ST1$  and  $ST2$  such that both  $ST1 \vdash st$  and  $ST2 \vdash st$ ?

□

We can now state something closer to the desired preservation property:

**Theorem** `preservation_wrong2` :  $\forall ST\ T\ t\ st\ t'\ st',$   
 $\text{empty}; ST \vdash t \text{ \texttt{\textbackslash in} } T \rightarrow$

$t / st \implies t' / st' \rightarrow$   
 $\text{store\_well\_typed } ST \ st \rightarrow$   
 $\text{empty}; ST \vdash t' \text{ \texttt{in} } T.$   
 Abort.

This statement is fine for all of the evaluation rules except the allocation rule `ST_RefValue`. The problem is that this rule yields a store with a larger domain than the initial store, which falsifies the conclusion of the above statement: if  $st'$  includes a binding for a fresh location  $l$ , then  $l$  cannot be in the domain of  $ST$ , and it will not be the case that  $t'$  (which definitely mentions  $l$ ) is typable under  $ST$ .

### 31.7.2 Extending Store Typings

Evidently, since the store can increase in size during evaluation, we need to allow the store typing to grow as well. This motivates the following definition. We say that the store type  $ST'$  *extends*  $ST$  if  $ST'$  is just  $ST$  with some new types added to the end.

Inductive **extends** : store\_ty  $\rightarrow$  store\_ty  $\rightarrow$  Prop :=  
 | extends\_nil :  $\forall ST'$ ,  
   **extends**  $ST'$  **nil**  
 | extends\_cons :  $\forall x ST' ST$ ,  
   **extends**  $ST' ST \rightarrow$   
   **extends**  $(x :: ST') (x :: ST).$

Hint Constructors **extends**.

We'll need a few technical lemmas about extended contexts.

First, looking up a type in an extended store typing yields the same result as in the original:

Lemma extends\_lookup :  $\forall l ST ST'$ ,  
 $l < \text{length } ST \rightarrow$   
**extends**  $ST' ST \rightarrow$   
 $\text{store\_Tlookup } l ST' = \text{store\_Tlookup } l ST.$

Proof with auto.

```

intros l ST ST' Hlen H.
generalize dependent ST'. generalize dependent l.
induction ST as [|a ST2]; intros l Hlen ST' HST'.
- inversion Hlen.
- unfold store_Tlookup in *.
  destruct ST'.
  + inversion HST'.
  +
    inversion HST'; subst.
    destruct l as [|l'].
    × auto.

```

```

    × simpl. apply IHST2...
      simpl in Hlen; omega.

```

Qed.

Next, if  $ST'$  extends  $ST$ , the length of  $ST'$  is at least that of  $ST$ .

```

Lemma length_extends : ∀ l ST ST',
  l < length ST →
  extends ST' ST →
  l < length ST'.

```

Proof with eauto.

```

  intros. generalize dependent l. induction H0; intros l Hlen.
    inversion Hlen.
    simpl in *.
    destruct l; try omega.
    apply lt_n_S. apply IHextends. omega.

```

Qed.

Finally,  $ST \mathrel{++} T$  extends  $ST$ , and **extends** is reflexive.

```

Lemma extends_app : ∀ ST T,
  extends (ST ++ T) ST.

```

Proof with auto.

```

  induction ST; intros T...
  simpl...

```

Qed.

```

Lemma extends_refl : ∀ ST,
  extends ST ST.

```

Proof.

```

  induction ST; auto.

```

Qed.

### 31.7.3 Preservation, Finally

We can now give the final, correct statement of the type preservation property:

```

Definition preservation_theorem := ∀ ST t t' T st st',
  empty; ST ⊢ t \in T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     empty; ST' ⊢ t' \in T ∧
     store_well_typed ST' st').

```

Note that the preservation theorem merely asserts that there is *some* store typing  $ST'$  extending  $ST$  (i.e., agreeing with  $ST$  on the values of all the old locations) such that the



new term  $t'$  is well typed with respect to  $ST'$ ; it does not tell us exactly what  $ST'$  is. It is intuitively clear, of course, that  $ST'$  is either  $ST$  or else it is exactly  $ST \mathrel{++} T1::\text{nil}$ , where  $T1$  is the type of the value  $v1$  in the extended store  $st \mathrel{++} v1::\text{nil}$ , but stating this explicitly would complicate the statement of the theorem without actually making it any more useful: the weaker version above is already in the right form (because its conclusion implies its hypothesis) to "turn the crank" repeatedly and conclude that every *sequence* of evaluation steps preserves well-typedness. Combining this with the progress property, we obtain the usual guarantee that "well-typed programs never go wrong."

In order to prove this, we'll need a few lemmas, as usual.

### 31.7.4 Substitution lemma

First, we need an easy extension of the standard substitution lemma, along with the same machinery about context invariance that we used in the proof of the substitution lemma for the STLC.

```

Inductive appears_free_in : id → tm → Prop :=
| afi_var : ∀ x,
  appears_free_in x (tvar x)
| afi_app1 : ∀ x t1 t2,
  appears_free_in x t1 → appears_free_in x (tapp t1 t2)
| afi_app2 : ∀ x t1 t2,
  appears_free_in x t2 → appears_free_in x (tapp t1 t2)
| afi_abs : ∀ x y T11 t12,
  y ≠ x →
  appears_free_in x t12 →
  appears_free_in x (tabs y T11 t12)
| afi_succ : ∀ x t1,
  appears_free_in x t1 →
  appears_free_in x (tsucc t1)
| afi_pred : ∀ x t1,
  appears_free_in x t1 →
  appears_free_in x (tpred t1)
| afi_mult1 : ∀ x t1 t2,
  appears_free_in x t1 →
  appears_free_in x (tmult t1 t2)
| afi_mult2 : ∀ x t1 t2,
  appears_free_in x t2 →
  appears_free_in x (tmult t1 t2)
| afi_if0_1 : ∀ x t1 t2 t3,
  appears_free_in x t1 →
  appears_free_in x (tif0 t1 t2 t3)
| afi_if0_2 : ∀ x t1 t2 t3,

```

```

    appears_free_in x t2 →
    appears_free_in x (tif0 t1 t2 t3)
| afi_if0_3 : ∀ x t1 t2 t3,
    appears_free_in x t3 →
    appears_free_in x (tif0 t1 t2 t3)
| afi_ref : ∀ x t1,
    appears_free_in x t1 → appears_free_in x (tref t1)
| afi_deref : ∀ x t1,
    appears_free_in x t1 → appears_free_in x (tderef t1)
| afi_assign1 : ∀ x t1 t2,
    appears_free_in x t1 → appears_free_in x (tassign t1 t2)
| afi_assign2 : ∀ x t1 t2,
    appears_free_in x t2 → appears_free_in x (tassign t1 t2).

```

Hint Constructors **appears\_free\_in**.

```

Lemma free_in_context : ∀ x t T Gamma ST,
    appears_free_in x t →
    Gamma; ST ⊢ t \in T →
    ∃ T', Gamma x = Some T'.

```

Proof with eauto.

```

intros. generalize dependent Gamma. generalize dependent T.
induction H;
    intros; (try solve [ inversion H0; subst; eauto ]).
-
    inversion H1; subst.
    apply IHappears_free_in in H8.
    rewrite update_neq in H8; assumption.

```

Qed.

```

Lemma context_invariance : ∀ Gamma Gamma' ST t T,
    Gamma; ST ⊢ t \in T →
    (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
    Gamma'; ST ⊢ t \in T.

```

Proof with eauto.

```

intros.
generalize dependent Gamma'.
induction H; intros...
-
    apply T_Var. symmetry. rewrite ← H...
-
    apply T_Abs. apply IHhas_type; intros.
    unfold update, t_update.
    destruct (beq_idP x x0)...
-

```

```

    eapply T_App.
      apply IHhas_type1...
      apply IHhas_type2...
-
    eapply T_Mult.
      apply IHhas_type1...
      apply IHhas_type2...
-
    eapply T_If0.
      apply IHhas_type1...
      apply IHhas_type2...
      apply IHhas_type3...
-
    eapply T_Assign.
      apply IHhas_type1...
      apply IHhas_type2...
Qed.

Lemma substitution_preserves_typing :  $\forall$  Gamma ST x s S t T,
  empty; ST  $\vdash$  s \in S  $\rightarrow$ 
  (update Gamma x S); ST  $\vdash$  t \in T  $\rightarrow$ 
  Gamma; ST  $\vdash$  ([x:=s]t) \in T.
Proof with eauto.
  intros Gamma ST x s S t T Hs Ht.
  generalize dependent Gamma. generalize dependent T.
  induction t; intros T Gamma H;
    inversion H; subst; simpl...
-
  rename i into y.
  destruct (beq_idP x y).
+
  subst.
  rewrite update_eq in H3.
  inversion H3; subst.
  eapply context_invariance...
  intros x Hcontra.
  destruct (free_in_context _ _ _ _ Hcontra Hs) as [T' HT'].
  inversion HT'.
+
  apply T_Var.
  rewrite update_neq in H3...
- subst.
  rename i into y.

```

```

destruct (beq_idP x y).
+
  subst.
  apply T_Abs. eapply context_invariance...
  intros. rewrite update_shadow. reflexivity.
+
  apply T_Abs. apply IHt.
  eapply context_invariance...
  intros. unfold update, t_update.
  destruct (beq_idP y x0)...
  subst.
  rewrite false_beq_id...
Qed.

```

### 31.7.5 Assignment Preserves Store Typing

Next, we must show that replacing the contents of a cell in the store with a new value of appropriate type does not change the overall type of the store. (This is needed for the ST\_Assign rule.)

Lemma assign\_pres\_store\_typing :  $\forall ST\ st\ l\ t,$   
 $l < \text{length}\ st \rightarrow$   
 $\text{store\_well\_typed}\ ST\ st \rightarrow$   
 $\text{empty}; ST \vdash t \text{ \texttt{in} } (\text{store\_Tlookup}\ l\ ST) \rightarrow$   
 $\text{store\_well\_typed}\ ST\ (\text{replace}\ l\ t\ st).$

Proof with auto.

```

intros ST st l t Hlen HST Ht.
inversion HST; subst.
split. rewrite length_replace...
intros l' Hl'.
destruct (beq_nat l' l) eqn: Heqll'.
-
  apply beq_nat_true in Heqll'; subst.
  rewrite lookup_replace_eq...
-
  apply beq_nat_false in Heqll'.
  rewrite lookup_replace_neq...
  rewrite length_replace in Hl'.
  apply H0...

```

Qed.

### 31.7.6 Weakening for Stores

Finally, we need a lemma on store typings, stating that, if a store typing is extended with a new location, the extended one still allows us to assign the same types to the same terms as the original.

(The lemma is called `store_weakening` because it resembles the "weakening" lemmas found in proof theory, which show that adding a new assumption to some logical theory does not decrease the set of provable theorems.)

Lemma `store_weakening` :  $\forall \text{ Gamma } ST \text{ } ST' \text{ } t \text{ } T,$

**extends**  $ST' \text{ } ST \rightarrow$   
 $\text{Gamma}; ST \vdash t \text{ \texttt{in} } T \rightarrow$   
 $\text{Gamma}; ST' \vdash t \text{ \texttt{in} } T.$

Proof with `eauto`.

```
intros. induction H0; eauto.
-
  erewrite ← extends_lookup...
  apply T_Loc.
  eapply length_extends...
```

Qed.

We can use the `store_weakening` lemma to prove that if a store is well typed with respect to a store typing, then the store extended with a new term  $t$  will still be well typed with respect to the store typing extended with  $t$ 's type.

Lemma `store_well_typed_app` :  $\forall ST \text{ } st \text{ } t1 \text{ } T1,$   
`store_well_typed`  $ST \text{ } st \rightarrow$   
`empty`;  $ST \vdash t1 \text{ \texttt{in} } T1 \rightarrow$   
`store_well_typed`  $(ST \text{ ++ } T1 :: \text{nil}) (st \text{ ++ } t1 :: \text{nil}).$

Proof with `auto`.

```
intros.
unfold store_well_typed in *.
inversion H as [Hlen Hmatch]; clear H.
rewrite app_length, plus_comm. simpl.
rewrite app_length, plus_comm. simpl.
split...
-
  intros l Hl.
  unfold store_lookup, store_Tlookup.
  apply le_lt_eq_dec in Hl; inversion Hl as [Hlt | Heq].
  +
    apply lt_S_n in Hlt.
    rewrite !app_nth1...
    × apply store_weakening with ST. apply extends_app.
      apply Hmatch...
```

```

    × rewrite Hlen...
+
  inversion Heq.
  rewrite app_nth2; try omega.
  rewrite ← Hlen.
  rewrite minus_diag. simpl.
  apply store_weakening with ST...
  { apply extends_app. }
  rewrite app_nth2; try omega.
  rewrite minus_diag. simpl. trivial.
Qed.

```

### 31.7.7 Preservation!

Now that we've got everything set up right, the proof of preservation is actually quite straightforward.

Begin with one technical lemma `Lemma nth_eq_last :  $\forall A (l:\text{list } A) x d,$`   
`nth (length l) (l ++ x :: nil) d = x.`

Proof.

```
induction l; intros; [ auto | simpl; rewrite IHl; auto ].
```

Qed.

Theorem `preservation :  $\forall ST t t' T st st',$`   
`empty; ST ⊢ t \in T →`  
`store_well_typed ST st →`  
`t / st ==> t' / st' →`  
`∃ ST',`  
`(extends ST' ST ∧`  
`empty; ST' ⊢ t' \in T ∧`  
`store_well_typed ST' st').`

Proof with eauto using store\_weakening, extends\_refl.

```

  remember (@empty ty) as Gamma.
  intros ST t t' T st st' Ht.
  generalize dependent t'.
  induction Ht; intros t' HST Hstep;
    subst; try (solve by inversion); inversion Hstep; subst;
    try (eauto using store_weakening, extends_refl).
- ∃ ST.
  inversion Ht1; subst.
  split; try split... eapply substitution_preserves_typing...
-
  eapply IHHt1 in H0...
  inversion H0 as [ST' [Hext [Hty Hsty]]].

```

```

  ∃ ST'...
-
  eapply IHHt2 in H5...
  inversion H5 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  +
    eapply IHHt in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
-
  +
    eapply IHHt in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
-
  eapply IHHt1 in H0...
  inversion H0 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  eapply IHHt2 in H5...
  inversion H5 as [ST' [Hext [Hty Hsty]]].
  ∃ ST'...
-
  +
    eapply IHHt1 in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'... split...
-
  ∃ (ST ++ T1 :: nil).
  inversion HST; subst.
  split.
  apply extends_app.
  split.
  replace (TRef T1)
    with (TRef (store_Tlookup (length st) (ST ++ T1 :: nil))).
  apply T_Loc.
  rewrite ← H. rewrite app_length, plus_comm. simpl. omega.
  unfold store_Tlookup. rewrite ← H. rewrite nth_eq_last.
  reflexivity.
  apply store_well_typed_app; assumption.
-

```

```

    eapply IHht in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    ∃ ST. split; try split...
    inversion HST as [_ Hsty].
    replace T11 with (store_Tlookup l ST).
    apply Hsty...
    inversion Ht; subst...
  -
    eapply IHht in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    ∃ ST. split; try split...
    eapply assign_pres_store_typing...
    inversion Ht1; subst...
  -
    eapply IHht1 in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  -
    eapply IHht2 in H5...
    inversion H5 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
Qed.

```

**Exercise: 3 stars (preservation\_informal)** Write a careful informal proof of the preservation theorem, concentrating on the T\_App, T\_Deref, T\_Assign, and T\_Ref cases.

□

### 31.7.8 Progress

Fortunately, progress for this system is pretty easy to prove; the proof is very similar to the proof of progress for the STLC, with a few new cases for the new syntactic constructs.

Theorem progress :  $\forall ST\ t\ T\ st,$   
 empty;  $ST \vdash t \text{ in } T \rightarrow$   
 store\_well\_typed  $ST\ st \rightarrow$   
 (value  $t \vee \exists t', \exists st', t / st \Rightarrow t' / st'$ ).

Proof with eauto.

```

  intros ST t T st Ht HST. remember (@empty ty) as Gamma.
  induction Ht; subst; try solve by inversion...

```



```

-
right. destruct IHHt1 as [Ht1p | Ht1p]...
+
  inversion Ht1p; subst; try solve by inversion.
  destruct IHHt2 as [Ht2p | Ht2p]...
  ×
    inversion Ht2p as [t2' [st' Hstep]].
    ∃ (tapp (tabs x T t) t2'). ∃ st'...
+
  inversion Ht1p as [t1' [st' Hstep]].
  ∃ (tapp t1' t2). ∃ st'...
-
right. destruct IHHt as [Ht1p | Ht1p]...
+
  inversion Ht1p; subst; try solve [ inversion Ht ].
  ×
    ∃ (tnat (S n)). ∃ st...
+
  inversion Ht1p as [t1' [st' Hstep]].
  ∃ (tsucc t1'). ∃ st'...
-
right. destruct IHHt as [Ht1p | Ht1p]...
+
  inversion Ht1p; subst; try solve [inversion Ht ].
  ×
    ∃ (tnat (pred n)). ∃ st...
+
  inversion Ht1p as [t1' [st' Hstep]].
  ∃ (tpred t1'). ∃ st'...
-
right. destruct IHHt1 as [Ht1p | Ht1p]...
+
  inversion Ht1p; subst; try solve [inversion Ht1].
  destruct IHHt2 as [Ht2p | Ht2p]...
  ×
    inversion Ht2p; subst; try solve [inversion Ht2].
    ∃ (tnat (mult n n0)). ∃ st...
  ×
    inversion Ht2p as [t2' [st' Hstep]].
    ∃ (tmult (tnat n) t2'). ∃ st'...
+
  inversion Ht1p as [t1' [st' Hstep]].

```

```

       $\exists$  (tmult  $t1'$   $t2$ ).  $\exists$   $st'$ ...
-
right. destruct  $IHHt1$  as [ $Ht1p$  |  $Ht1p$ ]...
+
  inversion  $Ht1p$ ; subst; try solve [inversion  $Ht1$ ].
  destruct  $n$ .
   $\times$   $\exists$   $t2$ .  $\exists$   $st'$ ...
   $\times$   $\exists$   $t3$ .  $\exists$   $st'$ ...
+
  inversion  $Ht1p$  as [ $t1'$  [ $st'$   $Hstep$ ]].
   $\exists$  (tif0  $t1'$   $t2$   $t3$ ).  $\exists$   $st'$ ...
-
right. destruct  $IHHt$  as [ $Ht1p$  |  $Ht1p$ ]...
+
  inversion  $Ht1p$  as [ $t1'$  [ $st'$   $Hstep$ ]].
   $\exists$  (tref  $t1'$ ).  $\exists$   $st'$ ...
-
right. destruct  $IHHt$  as [ $Ht1p$  |  $Ht1p$ ]...
+
  inversion  $Ht1p$ ; subst; try solve by inversion.
  eexists. eexists. apply ST_DerefLoc...
  inversion  $Ht$ ; subst. inversion  $HST$ ; subst.
  rewrite  $\leftarrow H$ ...
+
  inversion  $Ht1p$  as [ $t1'$  [ $st'$   $Hstep$ ]].
   $\exists$  (tderef  $t1'$ ).  $\exists$   $st'$ ...
-
right. destruct  $IHHt1$  as [ $Ht1p$ | $Ht1p$ ]...
+
  destruct  $IHHt2$  as [ $Ht2p$ | $Ht2p$ ]...
   $\times$ 
    inversion  $Ht1p$ ; subst; try solve by inversion.
    eexists. eexists. apply ST_Assign...
    inversion  $HST$ ; subst. inversion  $Ht1$ ; subst.
    rewrite  $H$  in  $H5$ ...
   $\times$ 
    inversion  $Ht2p$  as [ $t2'$  [ $st'$   $Hstep$ ]].
     $\exists$  (tassign  $t1$   $t2'$ ).  $\exists$   $st'$ ...
+
  inversion  $Ht1p$  as [ $t1'$  [ $st'$   $Hstep$ ]].
   $\exists$  (tassign  $t1'$   $t2$ ).  $\exists$   $st'$ ...
Qed.

```

## 31.8 References and Nontermination

Section RefsAndNontermination.

Import *ExampleVariables*.

We know that the simply typed lambda calculus is *normalizing*, that is, every well-typed term can be reduced to a value in a finite number of steps. What about STLC + references? Surprisingly, adding references causes us to lose the normalization property: there exist well-typed terms in the STLC + references which can continue to reduce forever, without ever reaching a normal form!

How can we construct such a term? The main idea is to make a function which calls itself. We first make a function which calls another function stored in a reference cell; the trick is that we then smuggle in a reference to itself!

```
(\r:Ref (Unit -> Unit).
  r := (\x:Unit.(!r) unit); (!r) unit)
(ref (\x:Unit.unit))
```

First, *ref*  $(\lambda x:Unit.unit)$  creates a reference to a cell of type  $Unit \rightarrow Unit$ . We then pass this reference as the argument to a function which binds it to the name *r*, and assigns to it the function  $(\lambda x:Unit.(!r) unit)$  – that is, the function which ignores its argument and calls the function stored in *r* on the argument **unit**; but of course, that function is itself! To get the ball rolling we finally execute this function with  $(!r)$  **unit**.

Definition loop\_fun :=

```
tabs x TUnit (tapp (tderef (tvar r)) tunit).
```

Definition loop :=

```
tapp
(tabs r (TRef (TArrow TUnit TUnit))
  (tseq (tassign (tvar r) loop_fun)
        (tapp (tderef (tvar r)) tunit)))
(tref (tabs x TUnit tunit)).
```

This term is well typed:

Lemma loop\_typeable :  $\exists T, \text{empty}; \text{nil} \vdash \text{loop} \text{ in } T$ .

Proof with eauto.

```
eexists. unfold loop. unfold loop_fun.
eapply T_App...
eapply T_Abs...
eapply T_App...
  eapply T_Abs. eapply T_App. eapply T_Deref. eapply T_Var.
  unfold update, t_update. simpl. reflexivity. auto.
eapply T_Assign.
  eapply T_Var. unfold update, t_update. simpl. reflexivity.
eapply T_Abs.
```

```

    eapply T_App...
    eapply T_Deref. eapply T_Var. reflexivity.
Qed.

```

To show formally that the term diverges, we first define the **step\_closure** of the single-step reduction relation, written  $\Rightarrow^+$ . This is just like the reflexive step closure of single-step reduction (which we've been writing  $\Rightarrow^*$ ), except that it is not reflexive:  $t \Rightarrow^+ t'$  means that  $t$  can reach  $t'$  by *one or more* steps of reduction.

```

Inductive step_closure {X:Type} (R: relation X) : X → X → Prop :=
| sc_one : ∀ (x y : X),
    R x y → step_closure R x y
| sc_step : ∀ (x y z : X),
    R x y →
    step_closure R y z →
    step_closure R x z.

```

Definition multistep1 := (step\_closure step).

```

Notation "t1 '/' st '==>+' t2 '/' st'" := (multistep1 (t1, st) (t2, st'))
(at level 40, st at level 39, t2 at level 39).

```

Now, we can show that the expression `loop` reduces to the expression `!(loc 0) unit` and the size-one store `[r:=(loc 0)] loop_fun`.

As a convenience, we introduce a slight variant of the *normalize* tactic, called *reduce*, which tries solving the goal with `multi_refl` at each step, instead of waiting until the goal can't be reduced any more. Of course, the whole point is that `loop` doesn't normalize, so the old *normalize* tactic would just go into an infinite loop reducing it forever!

```

Ltac print_goal := match goal with ⊢ ?x ⇒ idtac x end.

```

```

Ltac reduce :=
  repeat (print_goal; eapply multi_step ;
    [ (eauto 10; fail) | (instantiate; compute)] ;
    try solve [apply multi_refl]).

```

Lemma loop\_steps\_to\_loop\_fun :

```

  loop / nil ==>*
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil.

```

Proof with eauto.

```

  unfold loop.

```

```

  reduce.

```

Qed.

Finally, the latter expression reduces in two steps to itself!

Lemma loop\_fun\_step\_self :

```

  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil ==>+
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil.

```

Proof with eauto.

```

unfold loop_fun; simpl.
eapply sc_step. apply ST_App1...
eapply sc_one. compute. apply ST_AppAbs...
Qed.

```

**Exercise: 4 stars (factorial\_ref)** Use the above ideas to implement a factorial function in STLC with references. (There is no need to prove formally that it really behaves like the factorial. Just use the example below to make sure it gives the correct result when applied to the argument 4.)

Definition factorial : tm :=

*admit.*

Lemma factorial\_type : empty; nil ⊢ factorial \in (TArrow TNat TNat).

Proof with eauto.

*Admitted.*

If your definition is correct, you should be able to just uncomment the example below; the proof should be fully automatic using the *reduce* tactic.

□

## 31.9 Additional Exercises

**Exercise: 5 stars, optional (garabage\_collector)** Challenge problem: modify our formalization to include an account of garbage collection, and prove that it satisfies whatever nice properties you can think to prove about it.

□

End RefsAndNontermination.

End STLCREF.

*Date : 2016 – 03 – 0409 : 33 : 20 – 0500(Fri, 04Mar2016)*

# Chapter 32

## Library RecordSub

### 32.1 RecordSub: Subtyping with Records

```
Require Import Sflib.  
Require Import Maps.  
Require Import MoreStlc.
```

### 32.2 Core Definitions

#### Syntax

Inductive **ty** : Type :=

```
| TTop : ty  
| TBase : id → ty  
| TArrow : ty → ty → ty  
  
| TRNil : ty  
| TRCons : id → ty → ty → ty.
```

Inductive **tm** : Type :=

```
| tvar : id → tm  
| tapp : tm → tm → tm  
| tabs : id → ty → tm → tm  
| tproj : tm → id → tm  
  
| trnil : tm  
| trcons : id → tm → tm → tm.
```

## Well-Formedness

Inductive **record\_ty** : **ty** → Prop :=

| RTnil :  
    **record\_ty** TRNil  
| RTcons : ∀ i T1 T2,  
    **record\_ty** (TRCons i T1 T2).

Inductive **record\_tm** : **tm** → Prop :=

| rtnil :  
    **record\_tm** trnil  
| rtcons : ∀ i t1 t2,  
    **record\_tm** (trcons i t1 t2).

Inductive **well\_formed\_ty** : **ty** → Prop :=

| wfTTop :  
    **well\_formed\_ty** TTop  
| wfTBase : ∀ i,  
    **well\_formed\_ty** (TBase i)  
| wfTArrow : ∀ T1 T2,  
    **well\_formed\_ty** T1 →  
    **well\_formed\_ty** T2 →  
    **well\_formed\_ty** (TArrow T1 T2)  
| wfTRNil :  
    **well\_formed\_ty** TRNil  
| wfTRCons : ∀ i T1 T2,  
    **well\_formed\_ty** T1 →  
    **well\_formed\_ty** T2 →  
    **record\_ty** T2 →  
    **well\_formed\_ty** (TRCons i T1 T2).

Hint Constructors **record\_ty record\_tm well\_formed\_ty**.

## Substitution

Fixpoint subst (x:id) (s:tm) (t:tm) : **tm** :=

match t with  
| tvar y ⇒ if beq\_id x y then s else t  
| tabs y T t1 ⇒ tabs y T (if beq\_id x y then t1 else (subst x s t1))  
| tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)  
| tproj t1 i ⇒ tproj (subst x s t1) i  
| trnil ⇒ trnil  
| trcons i t1 tr2 ⇒ trcons i (subst x s t1) (subst x s tr2)  
end.

Notation "'[ x ' := ' s ' ]' t" := (subst x s t) (at level 20).

## Reduction

Inductive **value** : **tm** → Prop :=

```
| v_abs : ∀ x T t,
  value (tabs x T t)
| v_rnil : value trnil
| v_rcons : ∀ i v vr,
  value v →
  value vr →
  value (trcons i v vr).
```

Hint Constructors **value**.

Fixpoint Tlookup (i:id) (Tr:ty) : **option** ty :=

```
match Tr with
| TRCons i' T Tr' => if beq_id i i' then Some T else Tlookup i Tr'
| _ => None
end.
```

Fixpoint tlookup (i:id) (tr:tm) : **option** tm :=

```
match tr with
| trcons i' t tr' => if beq_id i i' then Some t else tlookup i tr'
| _ => None
end.
```

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

```
| ST_AppAbs : ∀ x T t12 v2,
  value v2 →
  (tapp (tabs x T t12) v2) ==> [x:=v2] t12
| ST_App1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  (tapp t1 t2) ==> (tapp t1' t2)
| ST_App2 : ∀ v1 t2 t2',
  value v1 →
  t2 ==> t2' →
  (tapp v1 t2) ==> (tapp v1 t2')
| ST_Proj1 : ∀ tr tr' i,
  tr ==> tr' →
  (tproj tr i) ==> (tproj tr' i)
| ST_ProjRcd : ∀ tr i vi,
  value tr →
  tlookup i tr = Some vi →
  (tproj tr i) ==> vi
| ST_Rcd_Head : ∀ i t1 t1' tr2,
```



```

    t1 ==> t1' →
    (trcons i t1 tr2) ==> (trcons i t1' tr2)
| ST_Rcd_Tail : ∀ i v1 tr2 tr2',
  value v1 →
  tr2 ==> tr2' →
  (trcons i v1 tr2) ==> (trcons i v1 tr2')

```

where "t1 '==>' t2" := (**step** t1 t2).

Hint Constructors **step**.

## 32.3 Subtyping

Now we come to the interesting part. We begin by defining the subtyping relation and developing some of its important technical properties.

### 32.3.1 Definition

The definition of subtyping is essentially just what we sketched in the motivating discussion, but we need to add well-formedness side conditions to some of the rules.

Inductive **subtype** : **ty** → **ty** → Prop :=

```

| S_Refl : ∀ T,
  well_formed_ty T →
  subtype T T
| S_Trans : ∀ S U T,
  subtype S U →
  subtype U T →
  subtype S T
| S_Top : ∀ S,
  well_formed_ty S →
  subtype S TTop
| S_Arrow : ∀ S1 S2 T1 T2,
  subtype T1 S1 →
  subtype S2 T2 →
  subtype (TArrow S1 S2) (TArrow T1 T2)

| S_RcdWidth : ∀ i T1 T2,
  well_formed_ty (TRCons i T1 T2) →
  subtype (TRCons i T1 T2) TRNil
| S_RcdDepth : ∀ i S1 T1 Sr2 Tr2,
  subtype S1 T1 →

```

```

subtype Sr2 Tr2 →
record_ty Sr2 →
record_ty Tr2 →
subtype (TRCons i S1 Sr2) (TRCons i T1 Tr2)
| S_RcdPerm : ∀ i1 i2 T1 T2 Tr3,
well_formed_ty (TRCons i1 T1 (TRCons i2 T2 Tr3)) →
i1 ≠ i2 →
subtype (TRCons i1 T1 (TRCons i2 T2 Tr3))
          (TRCons i2 T2 (TRCons i1 T1 Tr3)).

```

Hint Constructors **subtype**.

### 32.3.2 Subtyping Examples and Exercises

Module EXAMPLES.

```

Notation x := (ld 0).
Notation y := (ld 1).
Notation z := (ld 2).
Notation j := (ld 3).
Notation k := (ld 4).
Notation i := (ld 5).
Notation A := (TBase (ld 6)).
Notation B := (TBase (ld 7)).
Notation C := (TBase (ld 8)).

```

```

Definition TRcd_j :=
  (TRCons j (TArrow B B) TRNil). Definition TRcd_kj :=
  TRCons k (TArrow A A) TRcd_j.

```

```

Example subtyping_example_0 :
  subtype (TArrow C TRcd_kj)
    (TArrow C TRNil).

```

Proof.

```

  apply S_Arrow.
  apply S_Refl. auto.
  unfold TRcd_kj, TRcd_j. apply S_RcdWidth; auto.

```

Qed.

The following facts are mostly easy to prove in Coq. To get full benefit from the exercises, make sure you also understand how to prove them on paper!

```

Exercise: 2 stars Example subtyping_example_1 :
  subtype TRcd_kj TRcd_j.

```

Proof with eauto.

*Admitted.*

□

**Exercise: 1 star** Example subtyping\_example\_2 :

```
subtype (TArrow TTop TRcd_kj)
  (TArrow (TArrow C C) TRcd_j).
```

Proof with eauto.

*Admitted.*

□

**Exercise: 1 star** Example subtyping\_example\_3 :

```
subtype (TArrow TRNil (TRCons j A TRNil))
  (TArrow (TRCons k B TRNil) TRNil).
```

Proof with eauto.

*Admitted.*

□

**Exercise: 2 stars** Example subtyping\_example\_4 :

```
subtype (TRCons x A (TRCons y B (TRCons z C TRNil)))
  (TRCons z C (TRCons y B (TRCons x A TRNil))).
```

Proof with eauto.

*Admitted.*

□

Definition trcd\_kj :=

```
(trcons k (tabs z A (tvar z))
  (trcons j (tabs z B (tvar z))
    trnil)).
```

End EXAMPLES.

### 32.3.3 Properties of Subtyping

#### Well-Formedness

Lemma subtype\_wf :  $\forall S T$ ,

```
subtype S T  $\rightarrow$ 
well_formed_ty T  $\wedge$  well_formed_ty S.
```

Proof with eauto.

```
intros S T Hsub.
```

```
induction Hsub;
```

```
  intros; try (destruct IHsub1; destruct IHsub2)...
```

```
-
```

```
  split... inversion H. subst. inversion H5... Qed.
```

```

Lemma wf_rcd_lookup : ∀ i T Ti,
  well_formed_ty T →
  Tlookup i T = Some Ti →
  well_formed_ty Ti.
Proof with eauto.
  intros i T.
  induction T; intros; try solve by inversion.
-
  inversion H. subst. unfold Tlookup in H0.
  destruct (beq_id i i0)... inversion H0; subst... Qed.

```

## Field Lookup

Our record matching lemmas get a little more complicated in the presence of subtyping for two reasons: First, record types no longer necessarily describe the exact structure of corresponding terms. Second, reasoning by induction on **has\_type** derivations becomes harder in general, because **has\_type** is no longer syntax directed.

```

Lemma rcd_types_match : ∀ S T i Ti,
  subtype S T →
  Tlookup i T = Some Ti →
  ∃ Si, Tlookup i S = Some Si ∧ subtype Si Ti.
Proof with (eauto using wf_rcd_lookup).
  intros S T i Ti Hsub Hget. generalize dependent Ti.
  induction Hsub; intros Ti Hget;
  try solve by inversion.
-
  ∃ Ti...
-
  destruct (IHsub2 Ti) as [Ui Hui]... destruct Hui.
  destruct (IHsub1 Ui) as [Si Hsi]... destruct Hsi.
  ∃ Si...
-
  rename i0 into k.
  unfold Tlookup. unfold Tlookup in Hget.
  destruct (beq_id i k)...
+
  inversion Hget. subst. ∃ S1...
-
  ∃ Ti. split.
+
  unfold Tlookup. unfold Tlookup in Hget.
  destruct (beq_idP i i1)...
  ×

```

```

destruct (beq_idP i i2)...
destruct H0.
subst...
+
inversion H. subst. inversion H5. subst... Qed.

```

**Exercise: 3 stars (rcd\_types\_match\_informal)** Write a careful informal proof of the `rcd_types_match` lemma.

□

## Inversion Lemmas

**Exercise: 3 stars, optional (sub\_inversion\_arrow)** Lemma `sub_inversion_arrow` :  $\forall U V1 V2$ ,

```

subtype U (TArrow V1 V2) →
  ∃ U1, ∃ U2,
    (U=(TArrow U1 U2)) ∧ (subtype V1 U1) ∧ (subtype U2 V2).

```

Proof with `eauto`.

```

intros U V1 V2 Hs.
remember (TArrow V1 V2) as V.
generalize dependent V2. generalize dependent V1.
Admitted.
□

```

## 32.4 Typing

Definition `context` := `partial_map ty`.

Reserved Notation " $\Gamma \vdash t \text{ in } T$ " (at level 40).

Inductive **has\_type** : `context` → **tm** → **ty** → Prop :=

```

| T_Var : ∀ Gamma x T,
  Gamma x = Some T →
  well_formed_ty T →
  has_type Gamma (tvar x) T
| T_Abs : ∀ Gamma x T11 T12 t12,
  well_formed_ty T11 →
  has_type (update Gamma x T11) t12 T12 →
  has_type Gamma (tabs x T11 t12) (TArrow T11 T12)
| T_App : ∀ T1 T2 Gamma t1 t2,
  has_type Gamma t1 (TArrow T1 T2) →
  has_type Gamma t2 T1 →
  has_type Gamma (tapp t1 t2) T2

```

```

| T_Proj : ∀ Gamma i t T Ti,
  has_type Gamma t T →
  Tlookup i T = Some Ti →
  has_type Gamma (tproj t i) Ti

| T_Sub : ∀ Gamma t S T,
  has_type Gamma t S →
  subtype S T →
  has_type Gamma t T

| T_RNil : ∀ Gamma,
  has_type Gamma trnil TRNil
| T_RCons : ∀ Gamma i t T tr Tr,
  has_type Gamma t T →
  has_type Gamma tr Tr →
  record_ty Tr →
  record_tm tr →
  has_type Gamma (trcons i t tr) (TRCons i T Tr)

```

where "Gamma |- t \in T" := (has\_type Gamma t T).

Hint Constructors has\_type.

### 32.4.1 Typing Examples

Module EXAMPLES2.

Import Examples.

**Exercise: 1 star** Example typing\_example\_0 :

```

has_type empty
  (trcons k (tabs z A (tvar z))
    (trcons j (tabs z B (tvar z))
      trnil))
  TRcd_kj.

```

Proof.

*Admitted.*

□

**Exercise: 2 stars** Example typing\_example\_1 :

```

has_type empty
  (tapp (tabs x TRcd_j (tproj (tvar x) j))
    (trcd_kj))
  (TArrow B B).

```

Proof with eauto.

*Admitted.*

□

**Exercise: 2 stars, optional** Example typing\_example\_2 :

```
has_type empty
  (tapp (tabs z (TArrow (TArrow C C) TRcd_j)
        (tproj (tapp (tvar z)
                     (tabs x C (tvar x)))
              j))
        (tabs z (TArrow C C) trcd_kj))
  (TArrow B B).
```

Proof with eauto.

*Admitted.*

□

End EXAMPLES2.

## 32.4.2 Properties of Typing

### Well-Formedness

Lemma has\_type\_\_wf :  $\forall \text{Gamma } t \text{ } T$ ,  
**has\_type** *Gamma* *t* *T*  $\rightarrow$  **well\_formed\_ty** *T*.

Proof with eauto.

```
intros Gamma t T Htyp.
induction Htyp...
-
  inversion IHHtyp1...
-
  eapply wf_rcd_lookup...
-
  apply subtype__wf in H.
  destruct H...
```

Qed.

Lemma step\_preserves\_record\_tm :  $\forall tr \ tr'$ ,  
**record\_tm** *tr*  $\rightarrow$   
*tr*  $\Rightarrow$  *tr'*  $\rightarrow$   
**record\_tm** *tr'*.

Proof.

```
intros tr tr' Hrt Hstp.
inversion Hrt; subst; inversion Hstp; subst; eauto.
```

Qed.

## Field Lookup

Lemma lookup\_field\_in\_value :  $\forall v \ T \ i \ Ti$ ,

```

value  $v \rightarrow$ 
has_type empty  $v \ T \rightarrow$ 
Tlookup  $i \ T = \text{Some } Ti \rightarrow$ 
 $\exists vi$ , tlookup  $i \ v = \text{Some } vi \wedge$  has_type empty  $vi \ Ti$ .

```

Proof with eauto.

```

remember empty as Gamma.
intros  $t \ T \ i \ Ti \ Hval \ Htyp$ . revert  $Ti \ HeqGamma \ Hval$ .
induction  $Htyp$ ; intros; subst; try solve by inversion.
-
  apply (rcd_types_match  $S$ ) in  $H0$ ... destruct  $H0$  as [ $Si \ [HgetSi \ Hsub]$ ].
  destruct ( $IHHtyp \ Si$ ) as [ $vi \ [Hget \ Htyvi]$ ]...
-
  simpl in  $H0$ . simpl. simpl in  $H1$ .
  destruct (beq_id  $i \ i0$ ).
+
  inversion  $H1$ . subst.  $\exists t$ ...
+
  destruct ( $IHHtyp2 \ Ti$ ) as [ $vi \ [get \ Htyvi]$ ]...
  inversion  $Hval$ ... Qed.

```

## Progress

**Exercise: 3 stars (canonical\_forms\_of\_arrow\_types)** Lemma canonical\_forms\_of\_arrow\_types  
:  $\forall \Gamma s \ T1 \ T2$ ,

```

has_type  $\Gamma s \ (TArrow \ T1 \ T2) \rightarrow$ 
value  $s \rightarrow$ 
 $\exists x, \exists S1, \exists s2$ ,
 $s = \text{tabs } x \ S1 \ s2$ .

```

Proof with eauto.

*Admitted.*

□

Theorem progress :  $\forall t \ T$ ,

```

has_type empty  $t \ T \rightarrow$ 
value  $t \vee \exists t', t ==> t'$ .

```

Proof with eauto.

```

intros  $t \ T \ Ht$ .
remember empty as Gamma.
revert  $HeqGamma$ .
induction  $Ht$ ;
  intros  $HeqGamma$ ; subst...

```



```

-
  inversion  $H$ .
-
  right.
  destruct  $IHHt1$ ; subst...
+
  destruct  $IHHt2$ ; subst...
  ×
  destruct (canonical_forms_of_arrow_types empty  $t1$   $T1$   $T2$ )
    as [ $x$  [ $S1$  [ $t12$   $Hegt1$ ]]]...
  subst.  $\exists$  ( $[x:=t2]$   $t12$ )...
  ×
  destruct  $H0$  as [ $t2'$   $Hstp$ ].  $\exists$  ( $\text{tapp } t1 \ t2'$ )...
+
  destruct  $H$  as [ $t1'$   $Hstp$ ].  $\exists$  ( $\text{tapp } t1' \ t2$ )...
-
  right. destruct  $IHHt$ ...
+
  destruct (lookup_field_in_value  $t$   $T$   $i$   $Ti$ ) as [ $t'$  [ $Hget$   $Ht'$ ]]...
+
  destruct  $H0$  as [ $t'$   $Hstp$ ].  $\exists$  ( $\text{tproj } t' \ i$ )...
-
  destruct  $IHHt1$ ...
+
  destruct  $IHHt2$ ...
  ×
  right. destruct  $H2$  as [ $tr'$   $Hstp$ ].
   $\exists$  ( $\text{trcons } i \ t \ tr'$ )...
+
  right. destruct  $H1$  as [ $t'$   $Hstp$ ].
   $\exists$  ( $\text{trcons } i \ t' \ tr$ )... Qed.

```

Informal proof of progress:

Theorem : For any term  $t$  and type  $T$ , if  $\text{empty} \vdash t : T$  then  $t$  is a value or  $t \implies t'$  for some term  $t'$ .

Proof : Let  $t$  and  $T$  be given such that  $\text{empty} \vdash t : T$ . We go by induction on the typing derivation. Cases  $\text{T\_Abs}$  and  $\text{T\_RNil}$  are immediate because abstractions and  $\{\}$  are always values. Case  $\text{T\_Var}$  is vacuous because variables cannot be typed in the empty context.

- If the last step in the typing derivation is by  $\text{T\_App}$ , then there are terms  $t1 \ t2$  and types  $T1 \ T2$  such that  $t = t1 \ t2$ ,  $T = T2$ ,  $\text{empty} \vdash t1 : T1 \rightarrow T2$  and  $\text{empty} \vdash t2 : T1$ .

The induction hypotheses for these typing derivations yield that  $t1$  is a value or steps, and that  $t2$  is a value or steps. We consider each case:

- Suppose  $t1 \implies t1'$  for some term  $t1'$ . Then  $t1\ t2 \implies t1'\ t2$  by **ST\_App1**.
- Otherwise  $t1$  is a value.
  - \* Suppose  $t2 \implies t2'$  for some term  $t2'$ . Then  $t1\ t2 \implies t1\ t2'$  by rule **ST\_App2** because  $t1$  is a value.
  - \* Otherwise,  $t2$  is a value. By lemma *canonical\_forms\_for\_arrow\_types*,  $t1 = \backslash x:S1.s2$  for some  $x$ ,  $S1$ , and  $s2$ . And  $(\backslash x:S1.s2)\ t2 \implies [x:=t2]s2$  by **ST\_AppAbs**, since  $t2$  is a value.
- If the last step of the derivation is by **T\_Proj**, then there is a term  $tr$ , type  $Tr$  and label  $i$  such that  $t = tr.i$ ,  $\text{empty} \vdash tr : Tr$ , and  $\text{Tlookup } i\ Tr = \text{Some } T$ .

The IH for the typing subderivation gives us that either  $tr$  is a value or it steps. If  $tr \implies tr'$  for some term  $tr'$ , then  $tr.i \implies tr'.i$  by rule **ST\_Proj1**.

Otherwise,  $tr$  is a value. In this case, lemma *lookup\_field\_in\_value* yields that there is a term  $ti$  such that  $\text{tlookup } i\ tr = \text{Some } ti$ . It follows that  $tr.i \implies ti$  by rule **ST\_ProjRcd**.

- If the final step of the derivation is by **T\_Sub**, then there is a type  $S$  such that  $S <: T$  and  $\text{empty} \vdash t : S$ . The desired result is exactly the induction hypothesis for the typing subderivation.
- If the final step of the derivation is by **T\_RCons**, then there exist some terms  $t1\ tr$ , types  $T1\ Tr$  and a label  $t$  such that  $t = \{i=t1, tr\}$ ,  $T = \{i:T1, Tr\}$ , **record\_tm**  $tr$ , **record\_tm**  $Tr$ ,  $\text{empty} \vdash t1 : T1$  and  $\text{empty} \vdash tr : Tr$ .

The induction hypotheses for these typing derivations yield that  $t1$  is a value or steps, and that  $tr$  is a value or steps. We consider each case:

- Suppose  $t1 \implies t1'$  for some term  $t1'$ . Then  $\{i=t1, tr\} \implies \{i=t1', tr\}$  by rule **ST\_Rcd\_Head**.
- Otherwise  $t1$  is a value.
  - \* Suppose  $tr \implies tr'$  for some term  $tr'$ . Then  $\{i=t1, tr\} \implies \{i=t1, tr'\}$  by rule **ST\_Rcd\_Tail**, since  $t1$  is a value.
  - \* Otherwise,  $tr$  is also a value. So,  $\{i=t1, tr\}$  is a value by **v\_rcons**.

## Inversion Lemmas

Lemma *typing\_inversion\_var* :  $\forall\ Gamma\ x\ T,$

**has\_type**  $\Gamma$  (tvar  $x$ )  $T \rightarrow$

$\exists\ S,$

$\Gamma$   $x = \text{Some } S \wedge \text{subtype } S\ T.$

Proof with *eauto*.

```

intros Gamma x T Hty.
remember (tvar x) as t.
induction Hty; intros;
  inversion Heqt; subst; try solve by inversion.
-
  ∃ T...
-
  destruct IHHty as [U [Hctx HsubU]]... Qed.
Lemma typing_inversion_app : ∀ Gamma t1 t2 T2,
  has_type Gamma (tapp t1 t2) T2 →
  ∃ T1,
    has_type Gamma t1 (TArrow T1 T2) ∧
    has_type Gamma t2 T1.
Proof with eauto.
  intros Gamma t1 t2 T2 Hty.
  remember (tapp t1 t2) as t.
  induction Hty; intros;
    inversion Heqt; subst; try solve by inversion.
  -
    ∃ T1...
  -
    destruct IHHty as [U1 [Hty1 Hty2]]...
    assert (Hwf := has_type__wf _ _ _ Hty2).
    ∃ U1... Qed.
Lemma typing_inversion_abs : ∀ Gamma x S1 t2 T,
  has_type Gamma (tabs x S1 t2) T →
  (∃ S2, subtype (TArrow S1 S2) T
    ∧ has_type (update Gamma x S1) t2 S2).
Proof with eauto.
  intros Gamma x S1 t2 T H.
  remember (tabs x S1 t2) as t.
  induction H;
    inversion Heqt; subst; intros; try solve by inversion.
  -
    assert (Hwf := has_type__wf _ _ _ H0).
    ∃ T12...
  -
    destruct IHhas_type as [S2 [Hsub Hty]]...
    Qed.
Lemma typing_inversion_proj : ∀ Gamma i t1 Ti,
  has_type Gamma (tproj t1 i) Ti →
  ∃ T, ∃ Si,

```

$\text{Tlookup } i \ T = \text{Some } Si \wedge \text{subtype } Si \ Ti \wedge \text{has\_type } \Gamma t1 \ T.$   
 Proof with eauto.

```

intros  $\Gamma i \ t1 \ Ti \ H.$ 
remember (tproj t1 i) as t.
induction H;
  inversion Heqt; subst; intros; try solve by inversion.
-
  assert (well_formed_ty Ti) as Hwf.
  {
    apply (wf_rcd_lookup i T Ti)...
    apply has_type__wf in H... }
   $\exists T. \exists Ti...$ 
-
  destruct IHhas_type as [U [Ui [Hget [Hsub Hty]]]]...
   $\exists U. \exists Ui... \text{Qed.}$ 

```

Lemma typing\_inversion\_rcons :  $\forall \Gamma i \ ti \ tr \ T,$   
 $\text{has\_type } \Gamma (\text{trcons } i \ ti \ tr) \ T \rightarrow$   
 $\exists Si, \exists Sr,$   
 $\text{subtype } (\text{TRCons } i \ Si \ Sr) \ T \wedge \text{has\_type } \Gamma ti \ Si \wedge$   
 $\text{record\_tm } tr \wedge \text{has\_type } \Gamma tr \ Sr.$

Proof with eauto.

```

intros  $\Gamma i \ ti \ tr \ T \ Hty.$ 
remember (trcons i ti tr) as t.
induction Hty;
  inversion Heqt; subst...
-
  apply IHHty in H0.
  destruct H0 as [Ri [Rr [HsubRS [HtypRi HtypRr]]]].
   $\exists Ri. \exists Rr...$ 
-
  assert (well_formed_ty (TRCons i T Tr)) as Hwf.
  {
    apply has_type__wf in Hty1.
    apply has_type__wf in Hty2... }
   $\exists T. \exists Tr... \text{Qed.}$ 

```

Lemma abs\_arrow :  $\forall x \ S1 \ s2 \ T1 \ T2,$   
 $\text{has\_type empty (tabs } x \ S1 \ s2) (\text{TArrow } T1 \ T2) \rightarrow$   
 $\text{subtype } T1 \ S1$   
 $\wedge \text{has\_type (update empty } x \ S1) \ s2 \ T2.$

Proof with eauto.

```

intros  $x \ S1 \ s2 \ T1 \ T2 \ Hty.$ 
apply typing_inversion_abs in Hty.

```

```

destruct Hty as [S2 [Hsub Hty]].
apply sub_inversion_arrow in Hsub.
destruct Hsub as [U1 [U2 [Heq [Hsub1 Hsub2]]]].
inversion Heq; subst... Qed.

```

## Context Invariance

```

Inductive appears_free_in : id → tm → Prop :=
| afi_var : ∀ x,
  appears_free_in x (tvar x)
| afi_app1 : ∀ x t1 t2,
  appears_free_in x t1 → appears_free_in x (tapp t1 t2)
| afi_app2 : ∀ x t1 t2,
  appears_free_in x t2 → appears_free_in x (tapp t1 t2)
| afi_abs : ∀ x y T11 t12,
  y ≠ x →
  appears_free_in x t12 →
  appears_free_in x (tabs y T11 t12)
| afi_proj : ∀ x t i,
  appears_free_in x t →
  appears_free_in x (tproj t i)
| afi_rhead : ∀ x i t tr,
  appears_free_in x t →
  appears_free_in x (trcons i t tr)
| afi_rtail : ∀ x i t tr,
  appears_free_in x tr →
  appears_free_in x (trcons i t tr).

```

Hint Constructors **appears\_free\_in**.

```

Lemma context_invariance : ∀ Gamma Gamma' t S,
  has_type Gamma t S →
  (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
  has_type Gamma' t S.

```

Proof with eauto.

```

intros. generalize dependent Gamma'.
induction H;
  intros Gamma' Heqv...
-
  apply T_Var... rewrite ← Heqv...
-
  apply T_Abs... apply IHhas_type. intros x0 Hafi.
  unfold update, t_update. destruct (beq_idP x x0)...
-

```

```

    apply T_App with T1...
  -
    apply T_RCons... Qed.
Lemma free_in_context :  $\forall x t T \text{ Gamma},$ 
  appears_free_in  $x t \rightarrow$ 
  has_type  $\text{Gamma } t T \rightarrow$ 
   $\exists T', \text{Gamma } x = \text{Some } T'.$ 
Proof with eauto.
  intros  $x t T \text{ Gamma Hafi Htyp}.$ 
  induction Htyp; subst; inversion Hafi; subst...
  -
    destruct (IH Htyp H5) as [ $T \text{ Hctx}$ ].  $\exists T.$ 
    unfold update, t_update in Hctx. rewrite false_beq_id in Hctx... Qed.

```

## Preservation

```

Lemma substitution_preserves_typing :  $\forall \text{Gamma } x U v t S,$ 
  has_type (update  $\text{Gamma } x U$ )  $t S \rightarrow$ 
  has_type empty  $v U \rightarrow$ 
  has_type  $\text{Gamma } ([x:=v] t) S.$ 
Proof with eauto.
  intros  $\text{Gamma } x U v t S \text{ Htypt Htypv}.$ 
  generalize dependent  $S.$  generalize dependent  $\text{Gamma}.$ 
  induction t; intros; simpl.
  -
    rename  $i$  into  $y.$ 
    destruct (typing_inversion_var _ _ _ Htypt) as [ $T \text{ [Hctx Hsub]}$ ].
    unfold update, t_update in Hctx.
    destruct (beq_idP  $x y$ )...
    +
      subst.
      inversion Hctx; subst. clear Hctx.
      apply context_invariance with empty...
      intros  $x \text{ Hcontra}.$ 
      destruct (free_in_context _ _  $S$  empty Hcontra) as [ $T' \text{ HT'}$ ]...
      inversion HT'.
    +
      destruct (subtype_wf _ _ Hsub)...
  -
    destruct (typing_inversion_app _ _ _ _ Htypt) as [ $T1 \text{ [Htypt1 Htypt2]}$ ].
    eapply T_App...
  -

```

```

rename i into y. rename t into T1.
destruct (typing_inversion_abs _ _ _ _ Htypt)
  as [T2 [Hsub Htypt2]].
destruct (subtype_wf _ _ Hsub) as [Hwf1 Hwf2].
inversion Hwf2. subst.
apply T_Sub with (TArrow T1 T2)... apply T_Abs...
destruct (beq_idP x y).
+
  eapply context_invariance...
  subst.
  intros x Hafi. unfold update, t_update.
  destruct (beq_id y x)...
+
  apply IHt. eapply context_invariance...
  intros z Hafi. unfold update, t_update.
  destruct (beq_idP y z)...
  subst. rewrite false_beq_id...
-
destruct (typing_inversion_proj _ _ _ _ Htypt)
  as [T [Ti [Hget [Hsub Htypt1]]]]...
-
eapply context_invariance...
intros y Hcontra. inversion Hcontra.
-
destruct (typing_inversion_rcons _ _ _ _ Htypt) as
  [Ti [Tr [Hsub [HtypTi [Hrcdt2 HtypTr]]]]].
apply T_Sub with (TRCons i Ti Tr)...
apply T_RCons...
+
  apply subtype_wf in Hsub. destruct Hsub. inversion H0...
+
  inversion Hrcdt2; subst; simpl... Qed.

```

Theorem preservation :  $\forall t t' T,$   
**has\_type** empty *t T*  $\rightarrow$   
*t*  $\Rightarrow$  *t'*  $\rightarrow$   
**has\_type** empty *t' T*.

Proof with eauto.  
 intros *t t' T HT*.  
 remember empty as *Gamma*. generalize dependent *HeqGamma*.  
 generalize dependent *t'*.  
 induction *HT*;  
 intros *t' HeqGamma HE*; subst; inversion *HE*; subst...

```

- inversion HE; subst...
+
  destruct (abs_arrow _ _ _ _ HT1) as [HA1 HA2].
  apply substitution_preserves_typing with T...
-
  destruct (lookup_field_in_value _ _ _ _ H2 HT H)
    as [vi [Hget Hty]].
  rewrite H4 in Hget. inversion Hget. subst...
-
  eauto using step_preserves_record_tm. Qed.

```

Informal proof of preservation:

Theorem: If  $t, t'$  are terms and  $T$  is a type such that  $\text{empty} \vdash t : T$  and  $t \Rightarrow t'$ , then  $\text{empty} \vdash t' : T$ .

Proof: Let  $t$  and  $T$  be given such that  $\text{empty} \vdash t : T$ . We go by induction on the structure of this typing derivation, leaving  $t'$  general. Cases  $T\_Abs$  and  $T\_RNil$  are vacuous because abstractions and  $\{\}$  don't step. Case  $T\_Var$  is vacuous as well, since the context is empty.

- If the final step of the derivation is by  $T\_App$ , then there are terms  $t1\ t2$  and types  $T1\ T2$  such that  $t = t1\ t2$ ,  $T = T2$ ,  $\text{empty} \vdash t1 : T1 \rightarrow T2$  and  $\text{empty} \vdash t2 : T1$ .

By inspection of the definition of the step relation, there are three ways  $t1\ t2$  can step. Cases  $ST\_App1$  and  $ST\_App2$  follow immediately by the induction hypotheses for the typing subderivations and a use of  $T\_App$ .

Suppose instead  $t1\ t2$  steps by  $ST\_AppAbs$ . Then  $t1 = \backslash x:S.t12$  for some type  $S$  and term  $t12$ , and  $t' = [x:=t2]t12$ .

By Lemma `abs_arrow`, we have  $T1 <: S$  and  $x:S1 \vdash s2 : T2$ . It then follows by lemma `substitution_preserves_typing` that  $\text{empty} \vdash [x:=t2] t12 : T2$  as desired.

- If the final step of the derivation is by  $T\_Proj$ , then there is a term  $tr$ , type  $Tr$  and label  $i$  such that  $t = tr.i$ ,  $\text{empty} \vdash tr : Tr$ , and  $Tlookup\ i\ Tr = \text{Some } T$ .

The IH for the typing derivation gives us that, for any term  $tr'$ , if  $tr \Rightarrow tr'$  then  $\text{empty} \vdash tr' : Tr$ . Inspection of the definition of the step relation reveals that there are two ways a projection can step. Case  $ST\_Proj1$  follows immediately by the IH.

Instead suppose  $tr.i$  steps by  $ST\_ProjRcd$ . Then  $tr$  is a value and there is some term  $vi$  such that  $tlookup\ i\ tr = \text{Some } vi$  and  $t' = vi$ . But by lemma `lookup_field_in_value`,  $\text{empty} \vdash vi : Ti$  as desired.

- If the final step of the derivation is by  $T\_Sub$ , then there is a type  $S$  such that  $S <: T$  and  $\text{empty} \vdash t : S$ . The result is immediate by the induction hypothesis for the typing subderivation and an application of  $T\_Sub$ .



- If the final step of the derivation is by  $T\_RCons$ , then there exist some terms  $t1\ tr$ , types  $T1\ Tr$  and a label  $t$  such that  $t = \{i=t1, tr\}$ ,  $T = \{i:T1, Tr\}$ , **record\_tm**  $tr$ , **record\_tm**  $Tr$ ,  $\text{empty} \vdash t1 : T1$  and  $\text{empty} \vdash tr : Tr$ .

By the definition of the step relation,  $t$  must have stepped by  $ST\_Rcd\_Head$  or  $ST\_Rcd\_Tail$ . In the first case, the result follows by the IH for  $t1$ 's typing derivation and  $T\_RCons$ . In the second case, the result follows by the IH for  $tr$ 's typing derivation,  $T\_RCons$ , and a use of the **step\_preserves\_record\_tm** lemma.

### 32.4.3 Exercises on Typing

**Exercise: 2 stars, optional (variations)** Each part of this problem suggests a different way of changing the definition of the STLC with records and subtyping. (These changes are not cumulative: each part starts from the original language.) In each part, list which properties (Progress, Preservation, both, or neither) become false. If a property becomes false, give a counterexample.

- Suppose we add the following typing rule:  $\Gamma \vdash t : S1 \multimap S2 \quad S1 <: T1 \quad T1 <: S1 \quad S2 <: T2$

– ————— (T\_Funny1)  $\Gamma \vdash t : T1 \multimap T2$

- Suppose we add the following reduction rule:

– ————— (ST\_Funny21)

$\{\}$   $\implies (\backslash x:\text{Top}. x)$

- Suppose we add the following subtyping rule:

– ————— (S\_Funny3)

$\{\} <: \text{Top} \multimap \text{Top}$

- Suppose we add the following subtyping rule:

– ————— (S\_Funny4)

$\text{Top} \multimap \text{Top} <: \{\}$

- Suppose we add the following evaluation rule:

– ————— (ST\_Funny5)

$(\{\} t) \implies (t \{\})$

- Suppose we add the same evaluation rule *\*and\** a new typing rule:

– ————— (ST\_Funny5)

$(\{\} t) ==> (t \{\})$

– ————— (T\_Funny6)

$\text{empty} \vdash \{\} : \text{Top} \rightarrow \text{Top}$

- Suppose we *\*change\** the arrow subtyping rule to:  $S1 <: T1 \ S2 <: T2$

– ————— (S\_Arrow')  $S1 \rightarrow S2 <: T1 \rightarrow T2$

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500 (Wed, 17 Feb 2016)

# Chapter 33

## Library Norm

### 33.1 Norm: Normalization of STLC

```
Require Import Coq.Lists.List.  
Import ListNotations.  
  
Require Import Sflib.  
Require Import Maps.  
Require Import Smallstep.  
Hint Constructors multi.
```

(This chapter is optional.)

In this chapter, we consider another fundamental theoretical property of the simply typed lambda-calculus: the fact that the evaluation of a well-typed program is guaranteed to halt in a finite number of steps—i.e., every well-typed term is *normalizable*.

Unlike the type-safety properties we have considered so far, the normalization property does not extend to full-blown programming languages, because these languages nearly always extend the simply typed lambda-calculus with constructs, such as general recursion (as we discussed in the `MoreStlc` chapter) or recursive types, that can be used to write non-terminating programs. However, the issue of normalization reappears at the level of *types* when we consider the metatheory of polymorphic versions of the lambda calculus such as `F_omega`: in this system, the language of types effectively contains a copy of the simply typed lambda-calculus, and the termination of the typechecking algorithm will hinge on the fact that a “normalization” operation on type expressions is guaranteed to terminate.

Another reason for studying normalization proofs is that they are some of the most beautiful—and mind-blowing—mathematics to be found in the type theory literature, often (as here) involving the fundamental proof technique of *logical relations*.

The calculus we shall consider here is the simply typed lambda-calculus over a single base type **bool** and with pairs. We’ll give full details of the development for the basic lambda-calculus terms treating **bool** as an uninterpreted base type, and leave the extension to the boolean operators and pairs to the reader. Even for the base calculus, normalization is not

entirely trivial to prove, since each reduction of a term can duplicate redexes in subterms.

**Exercise: 1 star** Where do we fail if we attempt to prove normalization by a straightforward induction on the size of a well-typed term?

□

## 33.2 Language

We begin by repeating the relevant language definition, which is similar to those in the MoreStlc chapter, and supporting results including type preservation and step determinism. (We won't need progress.) You may just wish to skip down to the Normalization section...

### Syntax and Operational Semantics

```
Inductive ty : Type :=
| TBool : ty
| TArrow : ty → ty → ty
| TProd : ty → ty → ty
.
```

```
Inductive tm : Type :=

| tvar : id → tm
| tapp : tm → tm → tm
| tabs : id → ty → tm → tm

| tpair : tm → tm → tm
| tfst : tm → tm
| tsnd : tm → tm

| ttrue : tm
| tfalse : tm
| tif : tm → tm → tm → tm.
```

### Substitution

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒ if beq_id x y then s else t
  | tabs y T t1 ⇒ tabs y T (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)
  | tpair t1 t2 ⇒ tpair (subst x s t1) (subst x s t2)
```

```

| tfst t1 ⇒ tfst (subst x s t1)
| tsnd t1 ⇒ tsnd (subst x s t1)
| ttrue ⇒ ttrue
| tfalse ⇒ tfalse
| tif t0 t1 t2 ⇒ tif (subst x s t0) (subst x s t1) (subst x s t2)
end.

```

Notation "'[ x ' := ' s ' ]' t" := (subst x s t) (at level 20).

## Reduction

Inductive **value** : **tm** → Prop :=

```

| v_abs : ∀ x T11 t12,
  value (tabs x T11 t12)
| v_pair : ∀ v1 v2,
  value v1 →
  value v2 →
  value (tpair v1 v2)
| v_true : value ttrue
| v_false : value tfalse

```

.

Hint Constructors **value**.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive **step** : **tm** → **tm** → Prop :=

```

| ST_AppAbs : ∀ x T11 t12 v2,
  value v2 →
  (tapp (tabs x T11 t12) v2) ==> [x:=v2] t12
| ST_App1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  (tapp t1 t2) ==> (tapp t1' t2)
| ST_App2 : ∀ v1 t2 t2',
  value v1 →
  t2 ==> t2' →
  (tapp v1 t2) ==> (tapp v1 t2')

| ST_Pair1 : ∀ t1 t1' t2,
  t1 ==> t1' →
  (tpair t1 t2) ==> (tpair t1' t2)
| ST_Pair2 : ∀ v1 t2 t2',
  value v1 →
  t2 ==> t2' →
  (tpair v1 t2) ==> (tpair v1 t2')

```

```

| ST_Fst : ∀ t1 t1',
  t1 ==> t1' →
  (tfst t1) ==> (tfst t1')
| ST_FstPair : ∀ v1 v2,
  value v1 →
  value v2 →
  (tfst (tpair v1 v2)) ==> v1
| ST_Snd : ∀ t1 t1',
  t1 ==> t1' →
  (tsnd t1) ==> (tsnd t1')
| ST_SndPair : ∀ v1 v2,
  value v1 →
  value v2 →
  (tsnd (tpair v1 v2)) ==> v2

| ST_IfTrue : ∀ t1 t2,
  (tif ttrue t1 t2) ==> t1
| ST_IfFalse : ∀ t1 t2,
  (tif tfalse t1 t2) ==> t2
| ST_If : ∀ t0 t0' t1 t2,
  t0 ==> t0' →
  (tif t0 t1 t2) ==> (tif t0' t1 t2)

```

where "t1 '==>' t2" := (**step** t1 t2).

Notation multistep := (**multi step**).

Notation "t1 '==>\*' t2" := (multistep t1 t2) (at level 40).

Hint Constructors **step**.

Notation step\_normal\_form := (normal\_form **step**).

Lemma value\_\_normal : ∀ t, **value** t → step\_normal\_form t.

Proof with eauto.

intros t H; induction H; intros [t' ST]; inversion ST...

Qed.

## Typing

Definition context := partial\_map ty.

Inductive **has\_type** : context → **tm** → **ty** → Prop :=

```

| T_Var : ∀ Gamma x T,
  Gamma x = Some T →
  has_type Gamma (tvar x) T

```

```

| T_Abs : ∀ Gamma x T11 T12 t12,
  has_type (update Gamma x T11) t12 T12 →
  has_type Gamma (tabs x T11 t12) (TArrow T11 T12)
| T_App : ∀ T1 T2 Gamma t1 t2,
  has_type Gamma t1 (TArrow T1 T2) →
  has_type Gamma t2 T1 →
  has_type Gamma (tapp t1 t2) T2

| T_Pair : ∀ Gamma t1 t2 T1 T2,
  has_type Gamma t1 T1 →
  has_type Gamma t2 T2 →
  has_type Gamma (tpair t1 t2) (TProd T1 T2)
| T_Fst : ∀ Gamma t T1 T2,
  has_type Gamma t (TProd T1 T2) →
  has_type Gamma (tfst t) T1
| T_Snd : ∀ Gamma t T1 T2,
  has_type Gamma t (TProd T1 T2) →
  has_type Gamma (tsnd t) T2

| T_True : ∀ Gamma,
  has_type Gamma ttrue TBool
| T_False : ∀ Gamma,
  has_type Gamma tfalse TBool
| T_If : ∀ Gamma t0 t1 t2 T,
  has_type Gamma t0 TBool →
  has_type Gamma t1 T →
  has_type Gamma t2 T →
  has_type Gamma (tif t0 t1 t2) T

```

Hint Constructors **has\_type**.

Hint Extern 2 (**has\_type** - (tapp - -) -) ⇒ eapply T\_App; auto.

Hint Extern 2 (- = -) ⇒ compute; reflexivity.

## Context Invariance

Inductive **appears\_free\_in** : **id** → **tm** → Prop :=

```

| afi_var : ∀ x,
  appears_free_in x (tvar x)
| afi_app1 : ∀ x t1 t2,
  appears_free_in x t1 → appears_free_in x (tapp t1 t2)
| afi_app2 : ∀ x t1 t2,
  appears_free_in x t2 → appears_free_in x (tapp t1 t2)

```

```

| afi_abs : ∀ x y T11 t12,
  y ≠ x →
  appears_free_in x t12 →
  appears_free_in x (tabs y T11 t12)

| afi_pair1 : ∀ x t1 t2,
  appears_free_in x t1 →
  appears_free_in x (tpair t1 t2)
| afi_pair2 : ∀ x t1 t2,
  appears_free_in x t2 →
  appears_free_in x (tpair t1 t2)
| afi_fst : ∀ x t,
  appears_free_in x t →
  appears_free_in x (tfst t)
| afi_snd : ∀ x t,
  appears_free_in x t →
  appears_free_in x (tsnd t)

| afi_if0 : ∀ x t0 t1 t2,
  appears_free_in x t0 →
  appears_free_in x (tif t0 t1 t2)
| afi_if1 : ∀ x t0 t1 t2,
  appears_free_in x t1 →
  appears_free_in x (tif t0 t1 t2)
| afi_if2 : ∀ x t0 t1 t2,
  appears_free_in x t2 →
  appears_free_in x (tif t0 t1 t2)

```

Hint Constructors **appears\_free\_in**.

Definition closed ( $t:tm$ ) :=  
 $\forall x, \neg \text{appears\_free\_in } x \ t.$

Lemma context\_invariance :  $\forall \text{Gamma Gamma}' t S,$   
 $\text{has\_type Gamma } t \ S \rightarrow$   
 $(\forall x, \text{appears\_free\_in } x \ t \rightarrow \text{Gamma } x = \text{Gamma}' x) \rightarrow$   
 $\text{has\_type Gamma}' t \ S.$

Proof with eauto.

```

intros. generalize dependent Gamma'.
induction H;
  intros Gamma' Heqv...
-
  apply T_Var... rewrite ← Heqv...
-

```



```

    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold update, t_update. destruct (beq_idP x y)...
  -
    apply T_Pair...
  -
    eapply T_If...
Qed.

```

Lemma free\_in\_context :  $\forall x t T \text{ Gamma},$   
**appears\_free\_in**  $x t \rightarrow$   
**has\_type**  $\text{Gamma } t T \rightarrow$   
 $\exists T', \text{Gamma } x = \text{Some } T'.$

Proof with eauto.

```

  intros x t T Gamma Hafi Htyp.
  induction Htyp; inversion Hafi; subst...
  -
    destruct IHHtyp as [T' Hctx]...  $\exists T'.$ 
    unfold update, t_update in Hctx.
    rewrite false_beq_id in Hctx...

```

Qed.

Corollary typable\_empty\_\_closed :  $\forall t T,$   
**has\_type** empty  $t T \rightarrow$   
 closed  $t.$

Proof.

```

  intros. unfold closed. intros x H1.
  destruct (free_in_context _ _ _ H1 H) as [T' C].
  inversion C. Qed.

```

## Preservation

Lemma substitution\_preserves\_typing :  $\forall \text{Gamma } x U v t S,$   
**has\_type** (update  $\text{Gamma } x U$ )  $t S \rightarrow$   
**has\_type** empty  $v U \rightarrow$   
**has\_type**  $\text{Gamma } ([x:=v] t) S.$

Proof with eauto.

```

  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  induction t;
  -
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  -
    simpl. rename i into y.
    unfold update, t_update in H1.
    destruct (beq_idP x y).

```

+

```
subst.
inversion H1; subst. clear H1.
eapply context_invariance...
intros x Hcontra.
destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
inversion HT'.
```

+

```
apply T_Var...
```

-

```
rename i into y. rename t into T11.
apply T_Abs...
destruct (beq_idP x y).
```

+

```
eapply context_invariance...
subst.
intros x Hafi. unfold update, t_update.
destruct (beq_id y x)...
```

+

```
apply IHt. eapply context_invariance...
intros z Hafi. unfold update, t_update.
destruct (beq_idP y z)...
subst. rewrite false_beq_id...
```

Qed.

Theorem preservation :  $\forall t t' T,$

**has\_type** empty *t T*  $\rightarrow$

*t*  $\Rightarrow$  *t'*  $\rightarrow$

**has\_type** empty *t' T*.

Proof with eauto.

```
intros t t' T HT.
remember (@empty ty) as Gamma. generalize dependent HeqGamma.
generalize dependent t'.
induction HT;
  intros t' HeqGamma HE; subst; inversion HE; subst...
```

-

```
inversion HE; subst...
```

+

```

    apply substitution_preserves_typing with T1...
    inversion HT1...
-
    inversion HT...
-
    inversion HT...
Qed.
□

```

## Determinism

```

Lemma step_deterministic :
  deterministic step.
Proof with eauto.
  unfold deterministic.
  Admitted.

```

## 33.3 Normalization

Now for the actual normalization proof.

Our goal is to prove that every well-typed term evaluates to a normal form. In fact, it turns out to be convenient to prove something slightly stronger, namely that every well-typed term evaluates to a *value*. This follows from the weaker property anyway via the Progress lemma (why?) but otherwise we don't need Progress, and we didn't bother re-proving it above.

Here's the key definition:

**Definition** `halts (t:tm) : Prop := ∃ t', t ==>* t' ∧ value t'.`

A trivial fact:

**Lemma** `value_halts : ∀ v, value v → halts v.`

**Proof.**

```

  intros v H. unfold halts.
  ∃ v. split.
  apply multi_refl.
  assumption.

```

**Qed.**

The key issue in the normalization proof (as in many proofs by induction) is finding a strong enough induction hypothesis. To this end, we begin by defining, for each type  $T$ , a set  $R\_T$  of closed terms of type  $T$ . We will specify these sets using a relation  $R$  and write  $R\ T\ t$  when  $t$  is in  $R\_T$ . (The sets  $R\_T$  are sometimes called *saturated sets* or *reducibility candidates*.)

Here is the definition of  $R$  for the base language:

- $R \text{ bool } t$  iff  $t$  is a closed term of type **bool** and  $t$  halts in a value
- $R (T1 \rightarrow T2) t$  iff  $t$  is a closed term of type  $T1 \rightarrow T2$  and  $t$  halts in a value *and* for any term  $s$  such that  $R T1 s$ , we have  $R T2 (t s)$ .

This definition gives us the strengthened induction hypothesis that we need. Our primary goal is to show that all *programs* —i.e., all closed terms of base type—halt. But closed terms of base type can contain subterms of functional type, so we need to know something about these as well. Moreover, it is not enough to know that these subterms halt, because the application of a normalized function to a normalized argument involves a substitution, which may enable more evaluation steps. So we need a stronger condition for terms of functional type: not only should they halt themselves, but, when applied to halting arguments, they should yield halting results.

The form of  $R$  is characteristic of the *logical relations* proof technique. (Since we are just dealing with unary relations here, we could perhaps more properly say *logical properties*.) If we want to prove some property  $P$  of all closed terms of type  $A$ , we proceed by proving, by induction on types, that all terms of type  $A$  *possess* property  $P$ , all terms of type  $A \rightarrow A$  *preserve* property  $P$ , all terms of type  $(A \rightarrow A) \rightarrow (A \rightarrow A)$  *preserve the property of preserving* property  $P$ , and so on. We do this by defining a family of properties, indexed by types. For the base type  $A$ , the property is just  $P$ . For functional types, it says that the function should map values satisfying the property at the input type to values satisfying the property at the output type.

When we come to formalize the definition of  $R$  in Coq, we hit a problem. The most obvious formulation would be as a parameterized Inductive proposition like this:

```
Inductive R : ty -> tm -> Prop := | R_bool : forall b t, has_type empty t TBool ->
halts t -> R TBool t | R_arrow : forall T1 T2 t, has_type empty t (TArrow T1 T2) ->
halts t -> (forall s, R T1 s -> R T2 (tapp t s)) -> R (TArrow T1 T2) t.
```

Unfortunately, Coq rejects this definition because it violates the *strict positivity requirement* for inductive definitions, which says that the type being defined must not occur to the left of an arrow in the type of a constructor argument. Here, it is the third argument to  $R\_arrow$ , namely  $(\forall s, R T1 s \rightarrow R TS (\text{tapp } t \text{ } s))$ , and specifically the  $R T1 s$  part, that violates this rule. (The outermost arrows separating the constructor arguments don't count when applying this rule; otherwise we could never have genuinely inductive properties at all!) The reason for the rule is that types defined with non-positive recursion can be used to build non-terminating functions, which as we know would be a disaster for Coq's logical soundness. Even though the relation we want in this case might be perfectly innocent, Coq still rejects it because it fails the positivity test.

Fortunately, it turns out that we *can* define  $R$  using a Fixpoint:

```
Fixpoint R (T:ty) (t:tm) {struct T} : Prop :=
  has_type empty t T ^ halts t ^
  (match T with
```

```

| TBool  $\Rightarrow$  True
| TArrow  $T1\ T2 \Rightarrow (\forall\ s,\ R\ T1\ s \rightarrow R\ T2\ (\text{tapp } t\ s))$ 

| TProd  $T1\ T2 \Rightarrow$  False
end).

```

As immediate consequences of this definition, we have that every element of every set  $R\_T$  halts in a value and is closed with type  $t$  :

Lemma `R_halts` :  $\forall\ \{T\}\ \{t\},\ R\ T\ t \rightarrow \text{halts } t$ .

Proof.

```

intros. destruct  $T$ ; unfold  $R$  in  $H$ ; inversion  $H$ ; inversion  $H1$ ; assumption.

```

Qed.

Lemma `R_typable_empty` :  $\forall\ \{T\}\ \{t\},\ R\ T\ t \rightarrow \text{has\_type empty } t\ T$ .

Proof.

```

intros. destruct  $T$ ; unfold  $R$  in  $H$ ; inversion  $H$ ; inversion  $H1$ ; assumption.

```

Qed.

Now we proceed to show the main result, which is that every well-typed term of type  $T$  is an element of  $R\_T$ . Together with `R_halts`, that will show that every well-typed term halts in a value.

### 33.3.1 Membership in $R\_T$ is invariant under evaluation

We start with a preliminary lemma that shows a kind of strong preservation property, namely that membership in  $R\_T$  is *invariant* under evaluation. We will need this property in both directions, i.e., both to show that a term in  $R\_T$  stays in  $R\_T$  when it takes a forward step, and to show that any term that ends up in  $R\_T$  after a step must have been in  $R\_T$  to begin with.

First of all, an easy preliminary lemma. Note that in the forward direction the proof depends on the fact that our language is deterministic. This lemma might still be true for nondeterministic languages, but the proof would be harder!

Lemma `step_preserves_halting` :  $\forall\ t\ t',\ (t \Rightarrow t') \rightarrow (\text{halts } t \leftrightarrow \text{halts } t')$ .

Proof.

```

intros  $t\ t'\ ST$ . unfold halts.

```

```

split.

```

```

-

```

```

  intros [ $t''$ ] [ $STM\ V$ ].

```

```

  inversion  $STM$ ; subst.

```

```

    exfalso. apply value__normal in  $V$ . unfold normal_form in  $V$ . apply  $V$ .  $\exists\ t'$ . auto.

```

```

    rewrite (step_deterministic - - -  $ST\ H$ ).  $\exists\ t''$ . split; assumption.

```

```

-

```

```

  intros [ $t'0$ ] [ $STM\ V$ ].

```

```

   $\exists\ t'0$ . split; eauto.

```

Qed.

Now the main lemma, which comes in two parts, one for each direction. Each proceeds by induction on the structure of the type  $T$ . In fact, this is where we make fundamental use of the structure of types.

One requirement for staying in  $R\_T$  is to stay in type  $T$ . In the forward direction, we get this from ordinary type Preservation.

Lemma `step_preserves_R` :  $\forall T\ t\ t', (t \Rightarrow t') \rightarrow R\ T\ t \rightarrow R\ T\ t'$ .

Proof.

```
induction T; intros t t' E Rt; unfold R; fold R; unfold R in Rt; fold R in Rt;
  destruct Rt as [typable_empty_t [halts_t RRt]].
  split. eapply preservation; eauto.
  split. apply (step_preserves_halting _ _ E); eauto.
  auto.
  split. eapply preservation; eauto.
  split. apply (step_preserves_halting _ _ E); eauto.
  intros.
  eapply IHT2.
  apply ST_App1. apply E.
  apply RRt; auto.
  Admitted.
```

The generalization to multiple steps is trivial:

Lemma `multistep_preserves_R` :  $\forall T\ t\ t', (t \Rightarrow^* t') \rightarrow R\ T\ t \rightarrow R\ T\ t'$ .

Proof.

```
intros T t t' STM; induction STM; intros.
  assumption.
  apply IHSTM. eapply step_preserves_R. apply H. assumption.
```

Qed.

In the reverse direction, we must add the fact that  $t$  has type  $T$  before stepping as an additional hypothesis.

Lemma `step_preserves_R'` :  $\forall T\ t\ t', \text{has\_type empty } t\ T \rightarrow (t \Rightarrow t') \rightarrow R\ T\ t' \rightarrow R\ T\ t$ .

Proof.

*Admitted.*

Lemma `multistep_preserves_R'` :  $\forall T\ t\ t', \text{has\_type empty } t\ T \rightarrow (t \Rightarrow^* t') \rightarrow R\ T\ t' \rightarrow R\ T\ t$ .

Proof.

```
intros T t t' HT STM.
  induction STM; intros.
  assumption.
  eapply step_preserves_R'. assumption. apply H. apply IHSTM.
```

eapply preservation; eauto. auto.

Qed.

### 33.3.2 Closed instances of terms of type $T$ belong to $R\_T$

Now we proceed to show that every term of type  $T$  belongs to  $R\_T$ . Here, the induction will be on typing derivations (it would be surprising to see a proof about well-typed terms that did not somewhere involve induction on typing derivations!). The only technical difficulty here is in dealing with the abstraction case. Since we are arguing by induction, the demonstration that a term  $\text{tabs } x \ T1 \ t2$  belongs to  $R\_ (T1 \rightarrow T2)$  should involve applying the induction hypothesis to show that  $t2$  belongs to  $R\_ (T2)$ . But  $R\_ (T2)$  is defined to be a set of *closed* terms, while  $t2$  may contain  $x$  free, so this does not make sense.

This problem is resolved by using a standard trick to suitably generalize the induction hypothesis: instead of proving a statement involving a closed term, we generalize it to cover all closed *instances* of an open term  $t$ . Informally, the statement of the lemma will look like this:

If  $x1:T1, \dots, xn:Tn \vdash t : T$  and  $v1, \dots, vn$  are values such that  $R \ T1 \ v1, R \ T2 \ v2, \dots, R \ Tn \ vn$ , then  $R \ T \ ([x1:=v1][x2:=v2] \dots [xn:=vn]t)$ .

The proof will proceed by induction on the typing derivation  $x1:T1, \dots, xn:Tn \vdash t : T$ ; the most interesting case will be the one for abstraction.

### Multisubstitutions, multi-extensions, and instantiations

However, before we can proceed to formalize the statement and proof of the lemma, we'll need to build some (rather tedious) machinery to deal with the fact that we are performing *multiple* substitutions on term  $t$  and *multiple* extensions of the typing context. In particular, we must be precise about the order in which the substitutions occur and how they act on each other. Often these details are simply elided in informal paper proofs, but of course Coq won't let us do that. Since here we are substituting closed terms, we don't need to worry about how one substitution might affect the term put in place by another. But we still do need to worry about the *order* of substitutions, because it is quite possible for the same identifier to appear multiple times among the  $x1, \dots, xn$  with different associated  $vi$  and  $Ti$ .

To make everything precise, we will assume that environments are extended from left to right, and multiple substitutions are performed from right to left. To see that this is consistent, suppose we have an environment written as  $\dots, y:\text{bool}, \dots, y:\text{nat}, \dots$  and a corresponding term substitution written as  $\dots[y:=(t\text{bool true})] \dots [y:=(t\text{nat } 3)] \dots t$ . Since environments are extended from left to right, the binding  $y:\text{nat}$  hides the binding  $y:\text{bool}$ ; since substitutions are performed right to left, we do the substitution  $y:=(t\text{nat } 3)$  first, so that the substitution  $y:=(t\text{bool true})$  has no effect. Substitution thus correctly preserves the type of the term.

With these points in mind, the following definitions should make sense.

A *multisubstitution* is the result of applying a list of substitutions, which we call an *environment*.

**Definition** env := **list** (id  $\times$  tm).

```

Fixpoint msubst (ss:env) (t:tm) {struct ss} : tm :=
match ss with
| nil ⇒ t
| ((x,s)::ss') ⇒ msubst ss' ([x:=s] t)
end.

```

We need similar machinery to talk about repeated extension of a typing context using a list of (identifier, type) pairs, which we call a *type assignment*.

Definition tass := **list** (id × ty).

```

Fixpoint mupdate (Gamma : context) (xts : tass) :=
  match xts with
  | nil ⇒ Gamma
  | ((x,v)::xts') ⇒ update (mupdate Gamma xts') x v
  end.

```

We will need some simple operations that work uniformly on environments and type assignments

```

Fixpoint lookup {X:Set} (k : id) (l : list (id × X)) {struct l} : option X :=
  match l with
  | nil ⇒ None
  | (j,x) :: l' ⇒
    if beq_id j k then Some x else lookup k l'
  end.

```

```

Fixpoint drop {X:Set} (n:id) (nxs:list (id × X)) {struct nxes} : list (id × X) :=
  match nxes with
  | nil ⇒ nil
  | ((n',x)::nxs') ⇒ if beq_id n' n then drop n nxes' else (n',x)::(drop n nxes')
  end.

```

An *instantiation* combines a type assignment and a value environment with the same domains, where corresponding elements are in R

```

Inductive instantiation : tass → env → Prop :=
| V_nil : instantiation nil nil
| V_cons : ∀ x T v c e, value v → R T v → instantiation c e → instantiation ((x,T)::c) ((x,v)::e).

```

We now proceed to prove various properties of these definitions.

## More Substitution Facts

First we need some additional lemmas on (ordinary) substitution.

```

Lemma vacuous_substitution : ∀ t x,
  ¬ appears_free_in x t →
  ∀ t', [x:=t'] t = t.

```



Proof with eauto.

*Admitted.*

Lemma subst\_closed:  $\forall t,$   
    closed  $t \rightarrow$   
     $\forall x t', [x:=t'] t = t.$

Proof.

intros. apply *vacuous\_substitution*. apply *H*. Qed.

Lemma subst\_not\_afi :  $\forall t x v, \text{closed } v \rightarrow \neg \text{appears\_free\_in } x ([x:=v] t).$

Proof with eauto.    unfold closed, **not**.

induction  $t$ ; intros  $x v P A$ ; simpl in  $A$ .

-  
    destruct (*beq\_idP*  $x i$ )...  
    inversion  $A$ ; subst. auto.

-  
    inversion  $A$ ; subst...

-  
    destruct (*beq\_idP*  $x i$ )...  
    + inversion  $A$ ; subst...  
    + inversion  $A$ ; subst...

-  
    inversion  $A$ ; subst...

-  
    inversion  $A$ ; subst...

-  
    inversion  $A$ ; subst...

-  
    inversion  $A$ .

-  
    inversion  $A$ .

-  
    inversion  $A$ ; subst...

Qed.

Lemma duplicate\_subst :  $\forall t' x t v,$   
    closed  $v \rightarrow [x:=t] ([x:=v] t') = [x:=v] t'.$

Proof.

intros. eapply *vacuous\_substitution*. apply subst\_not\_afi. auto.

Qed.

Lemma swap\_subst :  $\forall t x x1 v v1, x \neq x1 \rightarrow \text{closed } v \rightarrow \text{closed } v1 \rightarrow$   
     $[x1:=v1] ([x:=v] t) = [x:=v] ([x1:=v1] t).$

Proof with eauto.

induction  $t$ ; intros; simpl.

```

-
  destruct (beq_idP x i); destruct (beq_idP x1 i).
+ subst. exfalse...
+ subst. simpl. rewrite <- beq_id_refl. apply subst_closed...
+ subst. simpl. rewrite <- beq_id_refl. rewrite subst_closed...
+ simpl. rewrite false_beq_id... rewrite false_beq_id...
Admitted.

```

## Properties of multi-substitutions

Lemma msubst\_closed:  $\forall t, \text{closed } t \rightarrow \forall ss, \text{msubst } ss \ t = t$ .

Proof.

```

  induction ss.
  reflexivity.
  destruct a. simpl. rewrite subst_closed; assumption.

```

Qed.

Closed environments are those that contain only closed terms.

Fixpoint closed\_env (env:env) {struct env} :=

match env with

```

| nil => True
| (x, t)::env' => closed t & closed_env env'

```

end.

Next come a series of lemmas characterizing how `msubst` of closed terms distributes over `subst` and over each term form

Lemma subst\_msubst:  $\forall env \ x \ v \ t, \text{closed } v \rightarrow \text{closed\_env } env \rightarrow$

$\text{msubst } env \ ([x:=v]t) = [x:=v](\text{msubst } (\text{drop } x \ env) \ t)$ .

Proof.

```

  induction env0; intros; auto.
  destruct a. simpl.
  inversion H0. fold closed_env in H2.
  destruct (beq_idP i x).
  - subst. rewrite duplicate_subst; auto.
  - simpl. rewrite swap_subst; eauto.

```

Qed.

Lemma msubst\_var:  $\forall ss \ x, \text{closed\_env } ss \rightarrow$

```

  msubst ss (tvar x) =
  match lookup x ss with
  | Some t => t
  | None => tvar x

```

end.

Proof.

```

induction ss; intros.
  reflexivity.
  destruct a.
    simpl. destruct (beq_id i x).
    apply msubst_closed. inversion H; auto.
    apply IHss. inversion H; auto.
Qed.

Lemma msubst_abs:  $\forall ss\ x\ T\ t,$ 
  msubst ss (tabs x T t) = tabs x T (msubst (drop x ss) t).
Proof.
  induction ss; intros.
    reflexivity.
    destruct a.
      simpl. destruct (beq_id i x); simpl; auto.
Qed.

Lemma msubst_app :  $\forall ss\ t1\ t2,$  msubst ss (tapp t1 t2) = tapp (msubst ss t1) (msubst ss t2).
Proof.
  induction ss; intros.
    reflexivity.
    destruct a.
      simpl. rewrite  $\leftarrow$  IHss. auto.
Qed.

```

You'll need similar functions for the other term constructors.

## Properties of multi-extensions

We need to connect the behavior of type assignments with that of their corresponding contexts.

Lemma mupdate\_lookup :  $\forall (c : \text{tass})\ (x:\text{id}),$  lookup x c = (mupdate empty c) x.

Proof.

```

induction c; intros.
  auto.
  destruct a. unfold lookup, mupdate, update, t_update. destruct (beq_id i x); auto.
Qed.

```

Lemma mupdate\_drop :  $\forall (c : \text{tass})\ \text{Gamma}\ x\ x',$   
 mupdate Gamma (drop x c) x' = if beq\_id x x' then Gamma x' else mupdate  
 Gamma c x'.

Proof.

```

induction c; intros.
- destruct (beq_idP x x'); auto.
- destruct a. simpl.

```

```

destruct (beq_idP i x).
+ subst. rewrite IHc.
  unfold update, t_update. destruct (beq_idP x x'); auto.
+ simpl. unfold update, t_update. destruct (beq_idP i x'); auto.
  subst. rewrite false_beq_id; congruence.

```

Qed.

## Properties of Instantiations

These are straightforward.

Lemma instantiation\_domains\_match:  $\forall \{c\} \{e\}$ ,

**instantiation**  $c\ e \rightarrow \forall \{x\} \{T\}$ ,  $\text{lookup } x\ c = \text{Some } T \rightarrow \exists t$ ,  $\text{lookup } x\ e = \text{Some } t$ .

Proof.

```

intros c e V. induction V; intros x0 T0 C.
  solve by inversion.
  simpl in *.
  destruct (beq_id x x0); eauto.

```

Qed.

Lemma instantiation\_env\_closed :  $\forall c\ e$ , **instantiation**  $c\ e \rightarrow \text{closed\_env } e$ .

Proof.

```

intros c e V; induction V; intros.
  econstructor.
  unfold closed_env. fold closed_env.
  split. eapply typable_empty_closed. eapply R_typable_empty. eauto.
  auto.

```

Qed.

Lemma instantiation\_R :  $\forall c\ e$ , **instantiation**  $c\ e \rightarrow$   
 $\forall x\ t\ T$ ,  $\text{lookup } x\ c = \text{Some } T \rightarrow$   
 $\text{lookup } x\ e = \text{Some } t \rightarrow R\ T\ t$ .

Proof.

```

intros c e V. induction V; intros x' t' T' G E.
  solve by inversion.
  unfold lookup in *. destruct (beq_id x x').
  inversion G; inversion E; subst. auto.
  eauto.

```

Qed.

Lemma instantiation\_drop :  $\forall c\ env$ ,

**instantiation**  $c\ env \rightarrow \forall x$ , **instantiation**  $(\text{drop } x\ c)\ (\text{drop } x\ env)$ .

Proof.

```

intros c e V. induction V.
  intros. simpl. constructor.
  intros. unfold drop. destruct (beq_id x x0); auto. constructor; eauto.

```

Qed.

### Congruence lemmas on multistep

We'll need just a few of these; add them as the demand arises.

Lemma multistep\_App2 :  $\forall v t t'$ ,  
  **value**  $v \rightarrow (t ==>* t') \rightarrow (\text{tapp } v t) ==>* (\text{tapp } v t')$ .

Proof.

```
intros v t t' V STM. induction STM.  
  apply multi_refl.  
  eapply multi_step.  
  apply ST_App2; eauto. auto.
```

Qed.

### The R Lemma.

We finally put everything together.

The key lemma about preservation of typing under substitution can be lifted to multi-substitutions:

Lemma msubst\_preserves\_typing :  $\forall c e$ ,  
  **instantiation**  $c e \rightarrow$   
   $\forall \text{Gamma } t S, \text{has\_type } (\text{mupdate } \text{Gamma } c) t S \rightarrow$   
  **has\\_type**  $\text{Gamma } (\text{msubst } e t) S$ .

Proof.

```
induction 1; intros.  
  simpl in H. simpl. auto.  
  simpl in H2. simpl.  
  apply IHinstantiation.  
  eapply substitution_preserves_typing; eauto.  
  apply (R_typable_empty H0).
```

Qed.

And at long last, the main lemma.

Lemma msubst\_R :  $\forall c \text{ env } t T$ ,  
  **has\\_type**  $(\text{mupdate empty } c) t T \rightarrow \text{instantiation } c \text{ env} \rightarrow \text{R } T (\text{msubst } \text{env } t)$ .

Proof.

```
intros c env0 t T HT V.  
generalize dependent env0.  
remember (mupdate empty c) as Gamma.  
assert ( $\forall x, \text{Gamma } x = \text{lookup } x c$ ).  
  intros. rewrite HeqGamma. rewrite mupdate_lookup. auto.  
clear HeqGamma.  
generalize dependent c.
```

```

induction HT; intros.

-
  rewrite H0 in H. destruct (instantiation_domains_match V H) as [t P].
  eapply instantiation_R; eauto.
  rewrite msubst_var. rewrite P. auto. eapply instantiation_env_closed; eauto.

-
  rewrite msubst_abs.
  assert (WT: has_type empty (tabs x T11 (msubst (drop x env0) t12)) (TArrow T11
T12)).
  { eapply T_Abs. eapply msubst_preserves_typing.
    { eapply instantiation_drop; eauto. }
    eapply context_invariance.
    { apply HT. }
    intros.
    unfold update, t_update. rewrite mupdate_drop. destruct (beq_idP x x0).
    + auto.
    + rewrite H.
      clear - c n. induction c.
      simpl. rewrite false_beq_id; auto.
      simpl. destruct a. unfold update, t_update.
      destruct (beq_id i x0); auto. }
  unfold R. fold R. split.
  auto.
  split. apply value_halts. apply v_abs.
  intros.
  destruct (R_halts H0) as [v [P Q]].
  pose proof (multistep_preserves_R _ _ _ P H0).
  apply multistep_preserves_R' with (msubst ((x, v) :: env0) t12).
  eapply T_App. eauto.
  apply R_typable_empty; auto.
  eapply multi_trans. eapply multistep_App2; eauto.
  eapply multi_R.
  simpl. rewrite subst_msubst.
  eapply ST_AppAbs; eauto.
  eapply typable_empty__closed.
  apply (R_typable_empty H1).
  eapply instantiation_env_closed; eauto.
  eapply (IHHT ((x, T11) :: c)).
    intros. unfold update, t_update, lookup. destruct (beq_id x x0); auto.
    constructor; auto.

-
  rewrite msubst_app.

```

```

destruct (IHHT1 c H env0 V) as [- [- P1]].
pose proof (IHHT2 c H env0 V) as P2. fold R in P1. auto.
Admitted.

```

## Normalization Theorem

Theorem normalization :  $\forall t \ T, \text{has\_type empty } t \ T \rightarrow \text{halts } t$ .

Proof.

```

intros.
replace t with (msubst nil t) by reflexivity.
apply (@R_halts T).
apply (msubst_R nil); eauto.
eapply V_nil.

```

Qed.

*Date* : 2016 – 03 – 0409 : 33 : 20 – 0500(*Fri*, 04*Mar*2016)

# Chapter 34

## Library LibTactics

### 34.1 LibTactics: A Collection of Handy General-Purpose Tactics

This file contains a set of tactics that extends the set of builtin tactics provided with the standard distribution of Coq. 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, Coq tactics will be improved as time goes by, and this file should ultimately be useless. In the meanwhile, serious Coq users will probably find it very useful.

The present file contains the implementation and the detailed documentation of those tactics. The SF reader need not read this file; instead, he/she is encouraged to read the chapter named UseTactics.v, which is gentle introduction to the most useful tactics from the LibTactic library.

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: tactic followed with the symbol `”~“` or `”*“` will call automation on the generated subgoals. 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 *rapPLY*,

Set Implicit Arguments.

Require Import [List](#).

Remove Hints Bool.trans\_eq\_bool.

## 34.2 Tools for programming with Ltac

### 34.2.1 Identity continuation

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

### 34.2.2 Untyped arguments for tactics

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

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

### 34.2.3 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.
```

### 34.2.4 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*.

### 34.2.5 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.
```

`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.
```

### 34.2.6 List of arguments for tactics

A datatype of type `list Boxer` is used to manipulate list of Coq 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.
```

### 34.2.7 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 ].
```

### 34.2.8 On-the-fly removal of hypotheses

In a list of arguments »  $H1\ H2\ \dots\ HN$  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.
```

### 34.2.9 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.

Require **BinPos** **Coq.ZArith.BinInt**.

Definition **ltac\_nat\_from\_int** ( $x$ :**BinInt.Z**) : **nat** :=

```
match x with
| BinInt.Z0  $\Rightarrow$  0%nat
| BinInt.Zpos  $p \Rightarrow$  BinPos.nat_of_P  $p$ 
| BinInt.Zneg  $p \Rightarrow$  0%nat
end.
```

Ltac *nat\_from\_number*  $N$  :=

```
match type of  $N$  with
| nat  $\Rightarrow$  constr:( $N$ )
| BinInt.Z  $\Rightarrow$  let  $N' :=$  constr:(ltac_nat_from_int  $N$ ) in eval compute in  $N'$ 
end.
```

*ltac\_pattern*  $E$  at  $K$  is the same as *pattern*  $E$  at  $K$  except that  $K$  is a Coq natural 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 nat_from_number  $K$  with
| 1  $\Rightarrow$  pattern  $E$  at 1
| 2  $\Rightarrow$  pattern  $E$  at 2
| 3  $\Rightarrow$  pattern  $E$  at 3
| 4  $\Rightarrow$  pattern  $E$  at 4
| 5  $\Rightarrow$  pattern  $E$  at 5
| 6  $\Rightarrow$  pattern  $E$  at 6
| 7  $\Rightarrow$  pattern  $E$  at 7
| 8  $\Rightarrow$  pattern  $E$  at 8
end.
```

Tactic Notation "**ltac\_pattern**" **constr**( $E$ ) "at" **constr**( $K$ ) "in" *hyp*( $H$ ) :=

```
match nat_from_number  $K$  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  $H$ 
end.
```

### 34.2.10 Testing tactics

*show tac* executes a tactic *tac* that produces a result, and then display its result.

Tactic Notation "show" *tactic(tac)* :=  
  let *R* := *tac* in pose *R*.

*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 *dup 2*.

Lemma *dup\_lemma* :  $\forall P, P \rightarrow P \rightarrow P$ .

Proof using. auto. Qed.

Ltac *dup\_tactic N* :=  
  match *nat\_from\_number N* with  
  | 0  $\Rightarrow$  idtac  
  | *S* 0  $\Rightarrow$  idtac  
  | *S* ?*N'*  $\Rightarrow$  apply *dup\_lemma*; [ | *dup\_tactic N'* ]  
  end.

Tactic Notation "dup" *constr(N)* :=  
  *dup\_tactic N*.

Tactic Notation "dup" :=  
  *dup 2*.

### 34.2.11 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  $\vdash ?G \Rightarrow$  *check\_noevar G* end.

### 34.2.12 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 in evar (*x:T*); *cont x*; subst *x*.

Tactic Notation "with\_evar" *constr(T) tactic(cont)* :=  
  *with\_evar\_base T cont*.



### 34.2.13 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.
```

### 34.2.14 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.
```

### 34.2.15 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 x1 ... xN` 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
  | ?P _ _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
```

```

| ?P _  $\Rightarrow$  constr:(P)
| ?P  $\Rightarrow$  constr:(P)
end.

```

*get\_fun\_arg E* is a tactic that decomposes an application term *E*, ie, when applied to a term of the form *X1 ... XN* it returns a pair made of *X1 .. X(N-1)* and *XN*.

```

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

```

### 34.2.16 Action at occurence and action not at occurence

*ltac\_action\_at K of E do Tac* isolates the *K*-th occurence 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 in *ltac\_pattern E* at *K*;  
 match goal with  $\vdash$  ?*P* \_  $\Rightarrow$  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 in *ltac\_pattern E* at *K* in *H*;  
 match type of *H* with ?*P* \_  $\Rightarrow$  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*.

### 34.2.17 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`.

Hint Unfold `eq'`.

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

## 34.3 Common tactics for simplifying goals like intuition

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

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

```
Ltac jauto_set :=  
  intros; jauto_set_hyps;  
  intros; jauto_set_goal;  
  unfold not in *.
```

## 34.4 Backward and forward chaining

### 34.4.1 Application

```
Ltac old_refine f :=  
  refine f.
```

*rappl*y is a tactic similar to `eapply` 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" 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 _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
  ].

```

The tactics *applies<sub>N</sub> 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 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*  $H1, \dots, HN$   $E$  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'.

```

### 34.4.2 Assertions

*asserts*  $H$ :  $T$  is another syntax for **assert** ( $H : T$ ), which also works with introduction patterns. For instance, one can write: *asserts*  $\backslash[x P \backslash] (\exists n, n = 3)$ , or *asserts*  $\backslash[H|H \backslash] (n = 0 \vee n = 1)$ .

Tactic Notation "asserts" *simple\_intropattern*( $I$ ) ":" constr( $T$ ) :=

```

    let H := fresh in assert (H : T);

```

[ | generalize  $H$ ; clear  $H$ ; intros  $I$  ].

$asserts\ H1 \dots HN: T$  is a shorthand for  $asserts\ \backslash[H1\ \backslash[H2\ \backslash[ \dots HN\backslash]\backslash]\backslash: T$ .

Tactic Notation "asserts"  $simple\_intropattern(I1)$   
 $simple\_intropattern(I2) \text{ ":" } constr(T) :=$   
 $asserts\ [I1\ I2]: T$ .

Tactic Notation "asserts"  $simple\_intropattern(I1)$   
 $simple\_intropattern(I2)\ simple\_intropattern(I3) \text{ ":" } constr(T) :=$   
 $asserts\ [I1\ [I2\ I3]]: T$ .

Tactic Notation "asserts"  $simple\_intropattern(I1)$   
 $simple\_intropattern(I2)\ simple\_intropattern(I3)$   
 $simple\_intropattern(I4) \text{ ":" } constr(T) :=$   
 $asserts\ [I1\ [I2\ [I3\ I4]]]: T$ .

Tactic Notation "asserts"  $simple\_intropattern(I1)$   
 $simple\_intropattern(I2)\ simple\_intropattern(I3)$   
 $simple\_intropattern(I4)\ simple\_intropattern(I5) \text{ ":" } constr(T) :=$   
 $asserts\ [I1\ [I2\ [I3\ [I4\ I5]]]]: T$ .

Tactic Notation "asserts"  $simple\_intropattern(I1)$   
 $simple\_intropattern(I2)\ simple\_intropattern(I3)$   
 $simple\_intropattern(I4)\ simple\_intropattern(I5)$   
 $simple\_intropattern(I6) \text{ ":" } constr(T) :=$   
 $asserts\ [I1\ [I2\ [I3\ [I4\ [I5\ I6]]]]]: T$ .

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

Tactic Notation "asserts"  $\text{ ":" } constr(T) :=$   
 $let\ H := fresh\ in\ asserts\ H : T$ .

$cuts\ H: T$  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 hypothesis.

Tactic Notation "cuts"  $simple\_intropattern(I) \text{ ":" } constr(T) :=$   
 $cut\ (T); [ intros\ I | idtac ]$ .

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

Tactic Notation "cuts"  $\text{ ":" } constr(T) :=$   
 $let\ H := fresh\ in\ cuts\ H: T$ .

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

Tactic Notation "cuts"  $simple\_intropattern(I1)$   
 $simple\_intropattern(I2) \text{ ":" } constr(T) :=$   
 $cuts\ [I1\ I2]: T$ .

Tactic Notation "cuts"  $simple\_intropattern(I1)$   
 $simple\_intropattern(I2)\ simple\_intropattern(I3) \text{ ":" } constr(T) :=$   
 $cuts\ [I1\ [I2\ I3]]: T$ .

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.

```

### 34.4.3 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 inserted. Underscore arguments  $--$  are interpreted 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_all 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

```



```

      |  $U \rightarrow ?Q \Rightarrow$  first [ app_assert t U cont' | fail 3 ]
      |  $\forall \_ : U, \_ \Rightarrow$  first [ app_evar t U cont' | fail 3 ]
      |  $?P \rightarrow ?Q \Rightarrow$  first [ app_assert t P cont | fail 3 ]
      |  $\forall \_ : ?A, \_ \Rightarrow$  first [ app_evar t A cont | fail 3 ]
    end
  end
| fail 2 ]
| -  $\Rightarrow$ 
  match T with
  |  $?P \rightarrow ?Q \Rightarrow$  first [ app_arg t P v cont'
                                | app_assert t P cont
                                | fail 3 ]
  |  $\forall \_ : \text{Type}, \_ \Rightarrow$ 
    match type of v with
    | Type  $\Rightarrow$  first [ cont' (t v)
                        | app_evar t Type cont
                        | fail 3 ]
    | -  $\Rightarrow$  first [ app_evar t Type cont
                    | fail 3 ]
    end
  |  $\forall \_ : ?A, \_ \Rightarrow$ 
    let V := type of v in
    match type of V with
    | Prop  $\Rightarrow$  first [ app_evar t A cont
                        | fail 3 ]
    | -  $\Rightarrow$  first [ cont' (t v)
                    | app_evar t A cont
                    | fail 3 ]
    end
  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_all t final ::=
  let rec go t :=
    match type of t with
    |  $?P \rightarrow ?Q \Rightarrow$  app_assert t P go
    |  $\forall \_ : ?A, \_ \Rightarrow$ 

```

```

      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_all 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 ]
                | ∀ _ : ?A, _ ⇒ 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
  | ∀ _:?A, - =>
    let V := type of v in
    match type of V with
    | Prop => first [ app_typeclass t cont
                    | app_evar t A cont
                    | fail 3 ]
    | - => first [ cont' (t v)
                  | app_typeclass t cont
                  | app_evar 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 =>
      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 =>
    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 =>
    args_unfold_head_if_not_product args
  | _ => 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: ( $\gg E0 A1$ ).

```

```

Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) :=
  lets: ( $\gg E0 A1 A2$ ).

```

```

Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) :=
  lets: ( $\gg E0 A1 A2 A3$ ).

```

```

Tactic Notation "lets" ":" constr(E0)
  constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: ( $\gg E0 A1 A2 A3 A4$ ).

```

```

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

```

```

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).

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).

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 [ 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 imple-
mentation is restricted to the case where H is an existing hypothesis – TODO: generalize.

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.

```

#### 34.4.4 Experimental tactics for application

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

Tactic Notation "fapply" constr( $E$ ) :=  
 let  $H$  := fresh 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  $H$  ].

### 34.4.5 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 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 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* ( $X$  :=  $E$ ).

### 34.4.6 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  $[A \wedge B]$  by showing first  $[A]$  **and** then  $[A \rightarrow B]$ , through the command [*logic* (foral  $A$   $B$ ,  $A \rightarrow (A \rightarrow B) \rightarrow A \wedge B$ )]

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`).

### 34.4.7 Application modulo equalities

The tactic *equates* replaces a goal of the form  $P \times y \ z$  with a goal of the form  $P \times ?a \ z$  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  $\forall n \ m, P \ n \ n \ m$ , 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. intros. subst. auto. Qed.

Lemma *equates\_1* :

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

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

Proof. 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. 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. 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. 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. 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 → x1 = y1 → P x1 x2 x3 x4 x5 x6.
Proof. intros. subst. auto. Qed.
End equatesLemma.

Ltac equates_lemma n :=
  match nat_from_number 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)
  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; [ instantiate; 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).

  applys_eq H i1 .. iK is the same as equates i1 .. iK followed by apply H on the first
  subgoal.

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

```

```

    applys_eq H (» n1 n2).
Tactic Notation "applys_eq" constr(H) constr(n1) constr(n2) constr(n3) :=
    applys_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).

```

### 34.4.8 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" :=
    elimtype False.

```

*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* (» *E1* .. *EN*), which tries to apply *applys* (» *E1* .. *EN*) and if it does not work then tries *forwards H*: (» *E1* .. *EN*) followed with *false*

```

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

```

```

Tactic Notation "false" constr(E) :=
    false_then E ltac:(fun _ => idtac).

```

```

Tactic Notation "false" constr(E) constr(E1) :=
    false (» E E1).

```

```

Tactic Notation "false" constr(E) constr(E1) constr(E2) :=
    false (» E E1 E2).

```

```

Tactic Notation "false" constr(E) constr(E1) constr(E2) constr(E3) :=
  false (» E E1 E2 E3).
Tactic Notation "false" constr(E) constr(E1) constr(E2) constr(E3) constr(E4) :=
  false (» 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 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: _ ⊢ _ ⇒
    solve [ inversion H; false
           | clear H; false_invert_iter
           | fail 2 ] 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  $E \neq E$ . 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  $\neq$  ?x ⊢ _ ⇒
    false_goal; apply H; reflexivity end.

```

## 34.5 Introduction and generalization

### 34.5.1 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 produce 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
  | ⊢ _ ⇒ idtac
  end.
```

```
Ltac introv_noarg :=
  match goal with
  | ⊢ ?P → ?Q ⇒ idtac
  | ⊢ ∀ _, _ ⇒ introv_rec
  | ⊢ ?G ⇒ hnf;
    match goal with
    | ⊢ ?P → ?Q ⇒ idtac
    | ⊢ ∀ _, _ ⇒ introv_rec
    end
  | ⊢ _ ⇒ idtac
  end.
```

```
Ltac introv_noarg_not_optimized :=
  intro; match goal with H:⊢_ ⇒ revert H end; introv_rec.
```

```
Ltac introv_arg H :=
  hnf; match goal with
  | ⊢ ?P → ?Q ⇒ intros H
  | ⊢ ∀ _, _ ⇒ 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 *introz* 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.

```



### 34.5.2 Generalization

*gen X1 .. XN* is a shorthand for calling **generalize dependent** successively on variables *XN...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*.

### 34.5.3 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  $\leftarrow$  X HX: E.`

Tactic Notation "set\_eq" "<-" ":" *constr*( $E$ ) :=  
`let X := fresh "X" in set_eq  $\leftarrow$  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  $\leftarrow$  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 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) → C. *gen\_eq: E* and *gen\_eq: E as X* are also accepted.

```

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

## 34.6 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 *).

```

*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 hypothese 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 hypothese is similar to the one of *rewrite*. Note: the tactic *replaces* plays a similar role.

```

Ltac asserts_rewrite_tactic E action :=
  let EQ := fresh 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 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 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.
```

### 34.6.1 Replace

*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 := fresh in assert (T: E = F); [ | replace E with F; clear T ].
```

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

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

### 34.6.2 Change

*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"  $\text{constr}(E1)$  "with"  $\text{constr}(E2)$  "in"  $\text{hyp}(H)$  :=  
`asserts_rewrite ( $E1 = E2$ ) in  $H$ ; [ reflexivity | ].`

Tactic Notation "changes"  $\text{constr}(E1)$  "with"  $\text{constr}(E2)$  :=  
`asserts_rewrite ( $E1 = E2$ ); [ reflexivity | ].`

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

### 34.6.3 Renaming

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

Tactic Notation "renames"  $\text{ident}(X1)$  "to"  $\text{ident}(Y1)$  :=  
`rename  $X1$  into  $Y1$ .`

Tactic Notation "renames"  $\text{ident}(X1)$  "to"  $\text{ident}(Y1)$  ", "  
 $\text{ident}(X2)$  "to"  $\text{ident}(Y2)$  :=  
`renames  $X1$  to  $Y1$ ; renames  $X2$  to  $Y2$ .`

Tactic Notation "renames"  $\text{ident}(X1)$  "to"  $\text{ident}(Y1)$  ", "  
 $\text{ident}(X2)$  "to"  $\text{ident}(Y2)$  ", "  $\text{ident}(X3)$  "to"  $\text{ident}(Y3)$  :=  
`renames  $X1$  to  $Y1$ ; renames  $X2$  to  $Y2$ ,  $X3$  to  $Y3$ .`

Tactic Notation "renames"  $\text{ident}(X1)$  "to"  $\text{ident}(Y1)$  ", "  
 $\text{ident}(X2)$  "to"  $\text{ident}(Y2)$  ", "  $\text{ident}(X3)$  "to"  $\text{ident}(Y3)$  ", "  
 $\text{ident}(X4)$  "to"  $\text{ident}(Y4)$  :=  
`renames  $X1$  to  $Y1$ ; renames  $X2$  to  $Y2$ ,  $X3$  to  $Y3$ ,  $X4$  to  $Y4$ .`

Tactic Notation "renames"  $\text{ident}(X1)$  "to"  $\text{ident}(Y1)$  ", "  
 $\text{ident}(X2)$  "to"  $\text{ident}(Y2)$  ", "  $\text{ident}(X3)$  "to"  $\text{ident}(Y3)$  ", "  
 $\text{ident}(X4)$  "to"  $\text{ident}(Y4)$  ", "  $\text{ident}(X5)$  "to"  $\text{ident}(Y5)$  :=  
`renames  $X1$  to  $Y1$ ; renames  $X2$  to  $Y2$ ,  $X3$  to  $Y3$ ,  $X4$  to  $Y4$ ,  $X5$  to  $Y5$ .`

*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.
```

### 34.6.4 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.
```



*unfolds*  $P1, \dots, PN$  is a shortcut for `unfold  $P1, \dots, PN$  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( $F6$ ) :=  
unfold  $F1, F2, F3, F4, F5, F6$  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*  $P1, \dots, PN$  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  $H5$ .`

### 34.6.5 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 Coq 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).

### 34.6.6 Evaluation

Tactic Notation "hnfs" := hnf in \*.

### 34.6.7 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  $H: ?x = ?y \vdash \_ \Rightarrow$ 
      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  $H: ?T \vdash \_ \Rightarrow$ 
  match  $T$  with
  | limit  $\Rightarrow$  idtac
  |  $?x = ?y \Rightarrow$ 
    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 ? $M \Rightarrow$  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
  | ( $_,_,_,_,_$ ) = ( $_,_,_,_,_$ )  $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 4 intro_subst_hyp
  | ( $_,_,_,_$ ) = ( $_,_,_,_$ )  $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 4 intro_subst_hyp
  | ( $_,_,_$ ) = ( $_,_,_$ )  $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 3 intro_subst_hyp
  | ( $_,_$ ) = ( $_,_$ )  $\Rightarrow$  injection  $H$ ; clear  $H$ ; do 2 intro_subst_hyp
  |  $?x = ?x \Rightarrow$  clear  $H$ 
  |  $?x = ?y \Rightarrow$  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.
```

### 34.6.8 Tactics to work with proof irrelevance

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 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).
```

### 34.6.9 Proving equalities

Note: current implementation only supports up to arity 5

*fequal* is a variation on `f_equal` which has a better behaviour on equalities between n-ary tuples.

```
Ltac fequal_base :=
  let go := f_equal; [ fequal_base | ] in
  match goal with
  | ⊢ ( _ , _ , _ ) = ( _ , _ , _ ) ⇒ go
  | ⊢ ( _ , _ , _ , _ ) = ( _ , _ , _ , _ ) ⇒ go
  | ⊢ ( _ , _ , _ , _ , _ ) = ( _ , _ , _ , _ , _ ) ⇒ go
  | ⊢ ( _ , _ , _ , _ , _ , _ ) = ( _ , _ , _ , _ , _ , _ ) ⇒ go
```

```
| ⊢ _ ⇒ f_equal
end.
```

Tactic Notation "fequal" :=  
*fequal\_base*.

*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  $f\ x1 \dots xN = f\ y1 \dots yN$  and produces some subgoals of the form  $xi = yi$ .

Ltac *fequal\_post* :=  
 first [ reflexivity | congruence | apply *proof\_irrelevance* | idtac ].

Tactic Notation "fequals" :=  
*fequal*; *fequal\_post*.

*fequals\_rec* calls *fequals* recursively. It is equivalent to **repeat** (**progress** *fequals*).

Tactic Notation "fequals\_rec" :=  
**repeat** (**progress** *fequals*).

## 34.7 Inversion

### 34.7.1 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 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).
```

### 34.7.2 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 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 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; 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 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*. TODO: 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 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).

```

### 34.7.3 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 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  $a_i = b_i$  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*.

#### 34.7.4 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.

*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**; **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 **intros** and **clear** *H* 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.

### 34.7.5 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* is to be defined later as a tactic to clean up goals. By defaults, it looks for obvious contradictions. Currently, this tactic is extended in LibReflect to clean up boolean propositions.

Ltac *case\_if\_post* := *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 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*.

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 in case_if as Eq.

Tactic Notation "case_if" "in" hyp(H) :=
  let Eq := fresh 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 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.

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 in cases_if as Eq.

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

BROKEN since v8.5beta2.

*destruct\_head\_match* performs a case analysis on the argument of the head pattern matching when the goal has the form `match ?E with ...` or `match ?E with ... = _` or `_ = match ?E with ...`. Due to the limits of Ltac, this tactic will not fail if a match does not occur. Instead, it might perform a case analysis on an unspecified subterm from the goal. Warning: experimental.

`Ltac find_head_match T :=`

```
  match T with context [?E] ⇒
    match T with
    | E ⇒ fail 1
    | _ ⇒ constr:(E)
    end
  end.
```

`Ltac destruct_head_match_core cont :=`

```
  match goal with
  | ⊢ ?T1 = ?T2 ⇒ first [ let E := find_head_match T1 in cont E
```

```

| let E := find_head_match T2 in cont E ]
| ⊢ ?T1 ⇒ let E := find_head_match T1 in cont E
end;
destruct_if_post.

Tactic Notation "destruct_head_match" "as" simple_intropattern(I) :=
  destruct_head_match_core ltac:(fun E ⇒ destruct E as I).

Tactic Notation "destruct_head_match" :=
  destruct_head_match_core ltac:(fun E ⇒ destruct E).

cases' E is similar to case_eq E except that it generates the equality in the context and
not in the goal. 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 X H E;
  destruct X.

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

cases_if' is similar to cases_if except that it generates the symmetric equality.

Ltac cases_if_on' E Eq :=
  match type of E with
  | { _ } + { _ } ⇒ destruct E as [Eq|Eq]; try subst_hyp Eq
  | _ ⇒ let X := fresh in
        sets_eq X Eq: E;
        destruct X
  end; case_if_post.

Tactic Notation "cases_if" "as" simple_intropattern(Eq) :=
  match goal with
  | ⊢ context [if ?B then _ else _] ⇒ cases_if_on' B Eq
  | K: context [if ?B then _ else _] ⊢ _ ⇒ cases_if_on' B Eq
  end.

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

```

## 34.8 Induction

*inductions E* is a shorthand for dependent induction *E*. *inductions E gen X1 .. XN* is a shorthand for dependent induction *E* generalizing *X1 .. XN*.

Require Import [Coq.Program.Equality](#).

Ltac *inductions\_post* :=

```

    unfold eq' in *.
Tactic Notation "inductions" ident(E) :=
    dependent induction E; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) :=
    dependent induction E generalizing X1; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2) :=
    dependent induction E generalizing X1 X2; 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  $(\text{fun } a \Rightarrow P\ a)\ X$ , using `pattern X`, and then it applies the well-founded induction principle instantiated on *E*, where *E* is a term of type *well\_founded* *R*, and *R* is a binary relation. Syntaxes *induction\_wf: E X* and *induction\_wf E X*.

```

Tactic Notation "induction_wf" ident(IH) ":" constr(E) ident(X) :=
    pattern X; apply (well_founded_ind E); clear X; intros X IH.
Tactic Notation "induction_wf" ":" constr(E) ident(X) :=
    let IH := fresh "IH" in induction_wf IH: E X.
Tactic Notation "induction_wf" ":" constr(E) ident(X) :=
    induction_wf: 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)

```
Require Import Compare_dec Omega.
```

```
Lemma induct_height_max2 :  $\forall\ n1\ n2 : \text{nat},$ 
```

```

 $\exists n, n1 < n \wedge n2 < n.$ 
Proof using.
  intros. destruct (lt_dec n1 n2).
   $\exists (S\ n2).$  omega.
   $\exists (S\ n1).$  omega.
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
  | _  $\Rightarrow \exists (S\ x);$  eauto
  end.

Ltac induct_height := induct_height_step O.

```

## 34.9 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*  $\Rightarrow$  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 _  $\vdash$  _  $\Rightarrow$  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 omega* to obtain the same behavior as *omega* but with an auxiliary lemma being generated.

Tactic Notation "abstracts" *tactic*(*tac*) :=  
 clear\_coind; *tac*.



## 34.10 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.
```

## 34.11 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*  $\leftarrow H$  swaps the two subgoals, i.e., produces  $(Q \rightarrow P)$  as first subgoal.

```
Lemma iff_intro_swap :  $\forall (P Q : \text{Prop}),$ 
   $(Q \rightarrow P) \rightarrow (P \rightarrow Q) \rightarrow (P \leftrightarrow Q).$ 
```

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  $\leftarrow$  H H.
```

```
Tactic Notation "iff" "<-" :=
  let H := fresh "H" in iff  $\leftarrow$  H.
```

## 34.12 N-ary Conjunctions and Disjunctions

N-ary Conjunctions Splitting in Goals

Underlying implementation of *splits*.

```
Ltac splits_tactic N :=
  match N with
  | 0  $\Rightarrow$  fail
  | S 0  $\Rightarrow$  idtac
  | S ?N'  $\Rightarrow$  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.

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 := nat_from_number N in
  splits_tactic N.

splits_all will recursively split any conjunction, unfolding definitions when necessary.
Warning: this tactic will loop on goals of the form well_founded R. Todo: fix this

Ltac splits_all_base := repeat split.

```

Tactic Notation "splits\_all" :=  
*splits\_all\_base*.

N-ary Conjunctions Deconstruction  
 Underlying implementation of *destructs*.

```
Ltac destructs_conjunction_tactic N T :=
  match N with
  | 2 ⇒ destruct T as [? ?]
  | 3 ⇒ destruct T as [? [? ?]]
  | 4 ⇒ destruct T as [? [? [? ?]]]
  | 5 ⇒ destruct T as [? [? [? [? ?]]]]
  | 6 ⇒ destruct T as [? [? [? [? [? ?]]]]]
  | 7 ⇒ 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 := nat_from_number N in
  destructs_conjunction_tactic N T.
```

Proving goals which are N-ary disjunctions  
 Underlying implementation of *branch*.

```
Ltac branch_tactic K N :=
  match constr:(K,N) with
  | (_,0) ⇒ fail 1
  | (0,_) ⇒ fail 1
  | (1,1) ⇒ idtac
  | (1,_) ⇒ 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
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 := nat_from_number 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 := nat_from_number N in
  let K := nat_from_number K in
  branch_tactic K N.

```

N-ary Disjunction Deconstruction  
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 := nat_from_number N in
  destructs_disjunction_tactic N T.

```

N-ary Existentials

```

Ltac get_term_existential_arity T :=
  match T with
  | [ x1 x2 x3 x4 x5 x6 x7 x8 , _ => constr:(8)
  | [ x1 x2 x3 x4 x5 x6 x7 , _ => constr:(7)
  | [ x1 x2 x3 x4 x5 x6 , _ => constr:(6)
  | [ x1 x2 x3 x4 x5 , _ => constr:(5)
  | [ x1 x2 x3 x4 , _ => constr:(4)
  | [ x1 x2 x3 , _ => constr:(3)
  | [ x1 x2 , _ => constr:(2)
  | [ x1 , _ => constr:(1)
  | _ -> ?T' => get_term_existential_arity T'
  | _ => let P := get_head T in
    let T' := eval unfold P in T in
    match T' with
    | T => fail 1
    | _ => get_term_existential_arity T'
  end
end.

```

```

Ltac get_goal_existential_arity :=
  match goal with | _ => ?T => get_term_existential_arity T end.

```

$\exists T1 \dots TN$  is a shorthand for  $\exists T1; \dots; \exists TN$ . It is intended to prove goals of the form *exist*  $X1 \dots XN, P$ . If an argument provided is `--` (double underscore), then an `ev` is

is introduced.  $\exists T1 .. TN \text{ ---}$  is equivalent to  $\exists T1 .. TN \text{ -- -- --}$  with as many  $\text{--}$  as possible.

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

Tactic Notation "exists" constr( $T1$ ) :=  
 match  $T1$  with  
 | ltac\_wild  $\Rightarrow$  *esplit*  
 | ltac\_wilds  $\Rightarrow$  repeat *esplit*  
 |  $\_ \Rightarrow \exists T1$   
 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( $T4$ ) :=  
 $\exists T1; \exists T2; \exists T3; \exists T4$ .

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

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

Tactic Notation "exists\_---" constr( $N$ ) :=  
 let rec aux  $N$  :=  
 match  $N$  with  
 | 0  $\Rightarrow$  idtac  
 |  $\textcolor{violet}{S} ?N' \Rightarrow$  *esplit*; aux  $N'$   
 end in  
 let  $N :=$  nat\_from\_number  $N$  in aux  $N$ .

Tactic Notation "exists\_---" :=  
 let  $N :=$  get\_goal\_existential\_arity in  
 exists\_---  $N$ .

Tactic Notation "exists" :=  
 exists\_---.

Tactic Notation "exists\_all" := exists\_---.

Existentials and conjunctions in hypotheses

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

Ltac *unpack\_core* :=  
 repeat match goal with  
 |  $H: \_ \textcolor{violet}{\wedge} \_ \vdash \_ \Rightarrow$  destruct  $H$

```

| H:  $\exists a, \_ \vdash \_ \Rightarrow$  destruct H
end.

Ltac unpack_from H :=
  first [ progress (unpack_core)
        | destruct H; unpack_core ].

Tactic Notation "unpack" :=
  unpack_core.

Tactic Notation "unpack" constr(H) :=
  unpack_from H.

```

## 34.13 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 ].

```

## 34.14 Tactics to invoke automation

### 34.14.1 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.

```

### 34.14.2 *hint* to add hints local to a lemma

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

Tactic Notation "hint" constr( $E$ ) :=

let  $H := \text{fresh "Hint" in lets } H: E.$

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* ( $E4$ ).

### 34.14.3 *jauto*, a new automation tactics

*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].

### 34.14.4 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 [ jauto ].`

`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; lets: E2; auto\_tilde*.

Tactic Notation "autos" "~" constr(*E1*) constr(*E2*) constr(*E3*) :=

*lets: E1; lets: E2; lets: E3; auto\_tilde*.

$autos\times$  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; lets: E2; auto\_star*.

Tactic Notation "autos" "\*" constr(*E1*) constr(*E2*) constr(*E3*) :=

*lets: E1; lets: E2; lets: E3; auto\_star*.

*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*).

### 34.14.5 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) 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 "applies" "~" constr(*H*) :=  
*supply H*; *auto\_tilde*. Tactic Notation "applies" "~" constr(*E0*) constr(*A1*) :=  
*applies 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 "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 "destruct\_head\_match" "~" :=  
*destruct\_head\_match; 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 "splits" "~" :=  
*splits; auto\_tilde.*  
Tactic Notation "splits" "~" *constr*(*N*) :=  
*splits N; auto\_tilde.*  
Tactic Notation "splits\_all" "~" :=  
*splits\_all; 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" "~" *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\_\_\_" "~" :=  
*exists\_\_\_; auto\_tilde.*  
Tactic Notation "exists" "~" *constr*(*T1*) :=  
 $\exists T1$ ; *auto\_tilde.*

```

Tactic Notation "exists" "~" constr(T1) constr(T2) :=
  ∃ 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.

```

### 34.14.6 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 *Coq.Classes.Equivalence*.

```

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) 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 "applies" "\*" *constr(H)* :=  
*apply H; auto\_star.* Tactic Notation "applies" "\*" *constr(E0) constr(A1) :=*  
*applies E0 A1; auto\_star.*  
Tactic Notation "applies" "\*" *constr(E0) constr(A1) :=*  
*applies E0 A1; auto\_star.*  
Tactic Notation "applies" "\*" *constr(E0) constr(A1) constr(A2) :=*  
*applies E0 A1 A2; auto\_star.*  
Tactic Notation "applies" "\*" *constr(E0) constr(A1) constr(A2) constr(A3) :=*  
*applies E0 A1 A2 A3; auto\_star.*  
Tactic Notation "applies" "\*" *constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)*  
:=  
*applies E0 A1 A2 A3 A4; auto\_star.*  
Tactic Notation "applies" "\*" *constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)*  
*constr(A5) :=*  
*applies 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"  $\text{hyp}(H) :=$   
      $\text{hnf in } H; \text{auto\_star}.$   
 Tactic Notation "subst" "\*" :=  
      $\text{subst}; \text{auto\_star}.$   
 Tactic Notation "intro\_hyp" "\*"  $\text{hyp}(H) :=$   
      $\text{subst\_hyp } H; \text{auto\_star}.$   
 Tactic Notation "intro\_subst" "\*" :=  
      $\text{intro\_subst}; \text{auto\_star}.$   
 Tactic Notation "subst\_eq" "\*"  $\text{constr}(E) :=$   
      $\text{subst\_eq } E; \text{auto\_star}.$   
 Tactic Notation "rewrite" "\*"  $\text{constr}(E) :=$   
      $\text{rewrite } E; \text{auto\_star}.$   
 Tactic Notation "rewrite" "\*" "<-"  $\text{constr}(E) :=$   
      $\text{rewrite } \leftarrow E; \text{auto\_star}.$   
 Tactic Notation "rewrite" "\*"  $\text{constr}(E)$  "in"  $\text{hyp}(H) :=$   
      $\text{rewrite } E \text{ in } H; \text{auto\_star}.$   
 Tactic Notation "rewrite" "\*" "<-"  $\text{constr}(E)$  "in"  $\text{hyp}(H) :=$   
      $\text{rewrite } \leftarrow E \text{ in } H; \text{auto\_star}.$   
 Tactic Notation "rewrites" "\*"  $\text{constr}(E) :=$   
      $\text{rewrites } E; \text{auto\_star}.$   
 Tactic Notation "rewrites" "\*"  $\text{constr}(E)$  "in"  $\text{hyp}(H) :=$   
      $\text{rewrites } E \text{ in } H; \text{auto\_star}.$   
 Tactic Notation "rewrites" "\*"  $\text{constr}(E)$  "in" "\*" :=  
      $\text{rewrites } E \text{ in } *; \text{auto\_star}.$   
 Tactic Notation "rewrites" "\*" "<-"  $\text{constr}(E) :=$   
      $\text{rewrites } \leftarrow E; \text{auto\_star}.$   
 Tactic Notation "rewrites" "\*" "<-"  $\text{constr}(E)$  "in"  $\text{hyp}(H) :=$   
      $\text{rewrites } \leftarrow E \text{ in } H; \text{auto\_star}.$   
 Tactic Notation "rewrites" "\*" "<-"  $\text{constr}(E)$  "in" "\*" :=  
      $\text{rewrites } \leftarrow E \text{ in } *; \text{auto\_star}.$   
 Tactic Notation "rewrite\_all" "\*"  $\text{constr}(E) :=$   
      $\text{rewrite\_all } E; \text{auto\_star}.$   
 Tactic Notation "rewrite\_all" "\*" "<-"  $\text{constr}(E) :=$   
      $\text{rewrite\_all } \leftarrow E; \text{auto\_star}.$   
 Tactic Notation "rewrite\_all" "\*"  $\text{constr}(E)$  "in"  $\text{ident}(H) :=$   
      $\text{rewrite\_all } E \text{ in } H; \text{auto\_star}.$   
 Tactic Notation "rewrite\_all" "\*" "<-"  $\text{constr}(E)$  "in"  $\text{ident}(H) :=$   
      $\text{rewrite\_all } \leftarrow E \text{ in } H; \text{auto\_star}.$   
 Tactic Notation "rewrite\_all" "\*"  $\text{constr}(E)$  "in" "\*" :=  
      $\text{rewrite\_all } E \text{ in } *; \text{auto\_star}.$   
 Tactic Notation "rewrite\_all" "\*" "<-"  $\text{constr}(E)$  "in" "\*" :=  
      $\text{rewrite\_all } \leftarrow E \text{ in } *; \text{auto\_star}.$

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"  $\text{hyp}(H)$  :=  
     *asserts\_rewrite E; auto\_star.*  
 Tactic Notation "asserts\_rewrite" "\*" "<-" constr( $E$ ) "in"  $\text{hyp}(H)$  :=  
     *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"  $\text{hyp}(H)$  :=  
     *cuts\_rewrite E in H; auto\_star.*  
 Tactic Notation "cuts\_rewrite" "\*" "<-" constr( $E$ ) "in"  $\text{hyp}(H)$  :=  
     *cuts\_rewrite ← E in H; auto\_star.*  
 Tactic Notation "erewrite" "\*" constr( $E$ ) :=  
     *erewrite 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"  $\text{hyp}(H)$  :=  
     *pi\_rewrite E in H; auto\_star.*  
 Tactic Notation "invert" "\*"  $\text{hyp}(H)$  :=  
     *invert H; auto\_star.*  
 Tactic Notation "inverts" "\*"  $\text{hyp}(H)$  :=  
     *inverts H; auto\_star.*  
 Tactic Notation "inverts" "\*"  $\text{hyp}(E)$  "as" :=  
     *inverts E as; auto\_star.*  
 Tactic Notation "injects" "\*"  $\text{hyp}(H)$  :=  
     *injects H; auto\_star.*  
 Tactic Notation "inversions" "\*"  $\text{hyp}(H)$  :=  
     *inversions H; auto\_star.*  
 Tactic Notation "cases" "\*" constr( $E$ ) "as"  $\text{ident}(H)$  :=



```

    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 "destruct_head_match" "*" :=
    destruct_head_match; 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 "splits_all" "*" :=
    splits_all; 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\_\_\_" "\*" :=  
*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.*

## 34.15 Tactics to sort out the proof context

### 34.15.1 Hiding hypotheses

Definition ltac\_something ( $P$ :Type) ( $e:P$ ) :=  $e$ .

Notation "'Something'" :=  
( $\text{@ltac\_something}$  - -).

Lemma ltac\_something\_eq :  $\forall$  ( $e$ :Type),  
 $e = (\text{@ltac\_something} - e)$ .

Proof using. auto. Qed.

Lemma ltac\_something\_hide :  $\forall$  ( $e$ :Type),

```

    e → (@ltac_something _ e).
Proof using. auto. Qed.

Lemma ltac_something_show : ∀ (e:Type),
  (@ltac_something _ e) → 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 ⇒
    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 ⊢ _ ⇒
    match T with
    | @ltac_something _ _ ⇒ fail 1
    | _ ⇒ 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  
  $H: @ltac.something \_ \vdash \_ \Rightarrow show\_hyp\ H$  end.

Tactic Notation "hide\_hyps" :=  
 repeat match goal with  $H: ?T \vdash \_ \Rightarrow$   
 match type of  $T$  with  
 | Prop  $\Rightarrow$   
 match  $T$  with  
 |  $@ltac.something \_ \Rightarrow fail\ 2$   
 |  $\_ \Rightarrow hide\_hyp\ H$   
 end  
 |  $\_ \Rightarrow fail\ 1$   
 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.`

### 34.15.2 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.`

### 34.15.3 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 :

```
  ∀ x y: nat, y < 2 → x = x → x ≥ 2 → x < 3 → 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 nat_from_number N with
```

```
  | 0 => idtac
```

```
  | S ?p => clears_last; clears_last_base p
```

```
  end.
```

Tactic Notation "clears\_last" constr(*N*) :=

```
  clears_last_base N.
```

## 34.16 Tactics for development purposes

### 34.16.1 Skipping subgoals

DEPRECATED: the new "admit" tactics now works fine.

The *skip* tactic can be used at any time to admit the current goal. Using *skip* is much more efficient than using the *Focus* top-level command to reach a particular subgoal.

There are two possible implementations of *skip*. The first one relies on the use of an existential variable. The second one relies on an axiom of type **False**. Remark that the builtin tactic **admit** is not applicable if the current goal contains uninstantiated variables.

The advantage of the first technique is that a proof using *skip* must end with *Admitted*, since **Qed** will be rejected with the message "uninstantiated existential variables". It is thereafter clear that the development is incomplete.

The advantage of the second technique is exactly the converse: one may conclude the proof using **Qed**, and thus one saves the pain from renaming **Qed** into *Admitted* and vice-versa all the time. Note however, that it is still necessary to instantiate all the existential variables introduced by other tactics in order for **Qed** to be accepted.

The two implementation are provided, so that you can select the one that suits you best. By default *skip'* uses the first implementation, and *skip* uses the second implementation.

```
Ltac skip_with_existential :=
  match goal with  $\vdash ?G \Rightarrow$ 
    let  $H := \text{fresh}$  in evar( $H:G$ ); eexact  $H$  end.
```

Variable *skip\_axiom* : **False**.

```
Ltac skip_with_axiom :=
  elimtype False; apply skip_axiom.
```

```
Tactic Notation "skip" :=
  skip_with_axiom.
```

```
Tactic Notation "skip'" :=
  skip_with_existential.
```

*demo* is like *admit* but it documents the fact that *admit* is intended Tactic Notation "demo" :=  
*skip*.

*skip H: T* adds an assumption named *H* of type *T* to the current context, blindly assuming that it is true. *skip: T* and *skip H\_asserts: T* and *skip\_asserts: T* are other possible syntax. Note that *H* may be an intro pattern. The syntax *skip H1 .. HN: T* can be used when *T* is a conjunction of *N* items.

```
Tactic Notation "skip" simple_intropattern( $I$ ) ":" constr( $T$ ) :=
  asserts  $I: T$ ; [ skip | ].
```

```
Tactic Notation "skip" ":" constr( $T$ ) :=
  let  $H := \text{fresh}$  in skip  $H: T$ .
```

```
Tactic Notation "skip" "~" ":" constr( $T$ ) :=
  skip:  $T$ ; auto_tilde.
```

```
Tactic Notation "skip" "*" ":" constr( $T$ ) :=
  skip:  $T$ ; auto_star.
```

```
Tactic Notation "skip" simple_intropattern( $I1$ )
  simple_intropattern( $I2$ ) ":" constr( $T$ ) :=
  skip [ $I1 I2$ ]:  $T$ .
```

```
Tactic Notation "skip" simple_intropattern( $I1$ )
  simple_intropattern( $I2$ ) simple_intropattern( $I3$ ) ":" constr( $T$ ) :=
  skip [ $I1 [I2 I3]$ ]:  $T$ .
```

```
Tactic Notation "skip" simple_intropattern( $I1$ )
  simple_intropattern( $I2$ ) simple_intropattern( $I3$ )
  simple_intropattern( $I4$ ) ":" constr( $T$ ) :=
  skip [ $I1 [I2 [I3 I4]]$ ]:  $T$ .
```

```
Tactic Notation "skip" simple_intropattern( $I1$ )
  simple_intropattern( $I2$ ) simple_intropattern( $I3$ )
  simple_intropattern( $I4$ ) simple_intropattern( $I5$ ) ":" constr( $T$ ) :=
```



*skip* [*I1* [*I2* [*I3* [*I4* *I5*]]]]: *T*.

Tactic Notation "skip" *simple\_intropattern*(*I1*)  
*simple\_intropattern*(*I2*) *simple\_intropattern*(*I3*)  
*simple\_intropattern*(*I4*) *simple\_intropattern*(*I5*)  
*simple\_intropattern*(*I6*) ":" **constr**(*T*) :=  
*skip* [*I1* [*I2* [*I3* [*I4* [*I5* *I6*]]]]]: *T*.

Tactic Notation "skip\_asserts" *simple\_intropattern*(*I*) ":" **constr**(*T*) :=  
*skip* *I*: *T*.

Tactic Notation "skip\_asserts" ":" **constr**(*T*) :=  
*skip*: *T*.

*skip\_cuts* *T* simply replaces the current goal with *T*.

Tactic Notation "skip\_cuts" **constr**(*T*) :=  
*cuts*: *T*; [ *skip* | ].

*skip\_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 *skip\_goal* is also accepted.

Tactic Notation "skip\_goal" *ident*(*H*) :=  
match goal with  $\vdash ?G \Rightarrow$  *skip* *H*: *G* end.

Tactic Notation "skip\_goal" :=  
let *IH* := fresh "IH" in *skip\_goal* *IH*.

*skip\_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 "skip\_rewrite" **constr**(*T*) :=  
let *M* := fresh in *skip\_asserts* *M*: *T*; **rewrite** *M*; **clear** *M*.

*skip\_rewrite* *T* in *H* is similar as *rewrite\_skip*, except that it rewrites in hypothesis *H*.

Tactic Notation "skip\_rewrite" **constr**(*T*) "in" *hyp*(*H*) :=  
let *M* := fresh in *skip\_asserts* *M*: *T*; **rewrite** *M* in *H*; **clear** *M*.

*skip\_rewrites\_all* *T* is similar as *rewrite\_skip*, except that it rewrites everywhere (goal and all hypotheses).

Tactic Notation "skip\_rewrite\_all" **constr**(*T*) :=  
let *M* := fresh in *skip\_asserts* *M*: *T*; **rewrite\_all** *M*; **clear** *M*.

*skip\_induction* *E* applies to any goal. It simply assumes the current goal to be true (the assumption is named "IH" by default), and call **destruct** *E* instead of **induction** *E*. It is useful to try and set up a proof by induction first, and fix the applications of the induction hypotheses during a second pass on the Proof using.

Tactic Notation "skip\_induction" **constr**(*E*) :=  
let *IH* := fresh "IH" in *skip\_goal* *IH*; **destruct** *E*.

Tactic Notation "skip\_induction" **constr**(*E*) "as" *simple\_intropattern*(*I*) :=  
let *IH* := fresh "IH" in *skip\_goal* *IH*; **destruct** *E* as *I*.

## 34.17 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) :=  
    apply H; auto_star.  
  Tactic Notation "subst" "*" :=  
    subst; auto_star.  
End LIBTACTICSCOMPATIBILITY.  
Open Scope nat_scope.
```

## 34.18 Additional notations for Coq

### 34.18.1 N-ary Existentials –TODO: DEPRECATED, Coq now supports it.

$\exists T1 \dots TN, P$  is a shorthand for  $\exists T1, \dots, \exists TN, P$ . Note that `Coq.Program.Syntax` already defines `exists` for arity up to 4.

### 34.18.2 '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`

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 ident*, 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 ident*, right associativity,  
format "[v' ']' 'let' x ':' A ':= v 'in' ']' '/' '[ e ']' ']'")  
: *let\_scope*.

Global Open Scope *let\_scope*.

Lemma `let_binding_unfold` :  $\forall (A B:\text{Type}) (v:A) (K:A \rightarrow B)$ ,

`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`  
`| ⊢ context [ let_binding v ?K ] =>`  
`changes (let_binding v K) with (K v)`  
`| H: context [ let_binding v ?K ] ⊢ _ =>`  
`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 ] =>`  
`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 ] =>`  
`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 ]  $\Rightarrow$ 
    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 ]  $\Rightarrow$ 
    set_eq x: v in H;
    let_simpl x in H
  end.

```

```

Tactic Notation "let_name_all" :=
  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 x
  | H: context [ let_binding ?v ?K ]  $\vdash$  _  $\Rightarrow$ 
    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.

```

Section FuncEq.

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

Lemma func\_eq\_1 :  $\forall (f:A1 \rightarrow B)\ x1\ y1,$   
 $x1 = y1 \rightarrow$   
 $f\ x1 = f\ y1.$

Proof. intros. subst $\neg$ . Qed.

Lemma func\_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. intros. subst $\neg$ . Qed.

Lemma func\_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. intros. subst $\neg$ . Qed.

Lemma func\_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. intros. subst $\neg$ . Qed.

Lemma func\_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. intros. subst $\neg$ . Qed.

End FuncEq.

```

Ltac f_equal_fixed :=
  try (
    first
    | apply func_eq_1
    | apply func_eq_2
    | apply func_eq_3
    | apply func_eq_4
    | apply func_eq_5 ];
  try reflexivity).

```

Ltac fequal\_base ::=

```

let go := f_equal_fixed; [ fequal_base | ] in
match goal with
| ⊢ (·, ·, ·) = (·, ·, ·) ⇒ go
| ⊢ (·, ·, ·, ·) = (·, ·, ·, ·) ⇒ go
| ⊢ (·, ·, ·, ·, ·) = (·, ·, ·, ·, ·) ⇒ go
| ⊢ (·, ·, ·, ·, ·, ·) = (·, ·, ·, ·, ·, ·) ⇒ go
| ⊢ _ ⇒ f_equal_fixed
end.

```

Generalize all propositions into the goal

```

Ltac generalize_all_prop :=
  repeat match goal with H: ?T ⊢ _ ⇒
    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 35

## Library UseTactics

### 35.1 UseTactics: Tactic Library for Coq: A Gentle Introduction

Coq comes with a set of builtin tactics, such as `reflexivity`, `intros`, `inversion` and so on. While it is possible to conduct proofs using only those tactics, you can significantly increase your productivity by working with a set of more powerful tactics. This chapter describes a number of such very useful tactics, which, for various reasons, are not yet available by default in Coq. These tactics are defined in the *LibTactics.v* file.

```
Require Import Coq.Arith.Arith.
```

```
Require Import Maps.
```

```
Require Import Imp.
```

```
Require Import Types.
```

```
Require Import Smallstep.
```

```
Require Import LibTactics.
```

Remark: `SSReflect` is another package providing powerful tactics. The library `"LibTactics"` differs from `"SSReflect"` in two respects:

- `"SSReflect"` was primarily developed for proving mathematical theorems, whereas `"LibTactics"` was primarily developed for proving theorems on programming languages. In particular, `"LibTactics"` provides a number of useful tactics that have no counterpart in the `"SSReflect"` package.
- `"SSReflect"` entirely rethinks the presentation of tactics, whereas `"LibTactics"` mostly stick to the traditional presentation of Coq tactics, simply providing a number of additional tactics. For this reason, `"LibTactics"` is probably easier to get started with than `"SSReflect"`.

This chapter is a tutorial focusing on the most useful features from the `"LibTactics"` library. It does not aim at presenting all the features of `"LibTactics"`. The detailed specifi-

cation of tactics can be found in the source file *LibTactics.v*. Further documentation as well as demos can be found at <http://www.chargueraud.org/softs/tlc/>.

In this tutorial, tactics are presented using examples taken from the core chapters of the “Software Foundations” course. To illustrate the various ways in which a given tactic can be used, we use a tactic that duplicates a given goal. More precisely, *dup* produces two copies of the current goal, and *dup n* produces *n* copies of it.

## 35.2 Tactics for introduction and case analysis

This section presents the following tactics:

- *introv*, for naming hypotheses more efficiently,
- *inverts*, for improving the *inversion* tactic,
- *cases*, for performing a case analysis without losing information,
- *cases\_if*, for automating case analysis on the argument of *if*.

### 35.2.1 The tactic *introv*

Module INTROVEXAMPLES.

Require Import Stlc.

Import Imp STLC.

The tactic *introv* allows to automatically introduce the variables of a theorem and explicitly name the hypotheses involved. In the example shown next, the variables *c*, *st*, *st1* and *st2* involved in the statement of determinism need not be named explicitly, because their name were already given in the statement of the lemma. On the contrary, it is useful to provide names for the two hypotheses, which we name *E1* and *E2*, respectively.

Theorem ceval\_deterministic:  $\forall c \ st \ st1 \ st2,$

$c / st \ \backslash \backslash \ st1 \rightarrow$

$c / st \ \backslash \backslash \ st2 \rightarrow$

$st1 = st2.$

Proof.

*introv E1 E2. Abort.*

When there is no hypothesis to be named, one can call *introv* without any argument.

Theorem dist\_exists\_or :  $\forall (X:\text{Type}) (P \ Q : X \rightarrow \text{Prop}),$

$(\exists x, P \ x \ \vee \ Q \ x) \leftrightarrow (\exists x, P \ x) \vee (\exists x, Q \ x).$

Proof.

*introv. Abort.*

The tactic *introv* also applies to statements in which  $\forall$  and  $\rightarrow$  are interleaved.



Theorem `ceval_deterministic'`:  $\forall c \ st \ st1,$   
 $(c / st \setminus \setminus st1) \rightarrow \forall st2, (c / st \setminus \setminus st2) \rightarrow st1 = st2.$

Proof.

*introv* *E1 E2*. Abort.

Like the arguments of `intros`, the arguments of *introv* can be structured patterns.

Theorem `exists_impl`:  $\forall X \ (P : X \rightarrow \text{Prop}) \ (Q : \text{Prop}) \ (R : \text{Prop}),$   
 $(\forall x, P \ x \rightarrow Q) \rightarrow$   
 $((\exists x, P \ x) \rightarrow Q).$

Proof.

*introv* [*x H2*]. `eauto`.

Qed.

Remark: the tactic *introv* works even when definitions need to be unfolded in order to reveal hypotheses.

End INTROVEXAMPLES.

### 35.2.2 The tactic *inverts*

Module `INVERTSEXAMPLES`.

Require Import `Stlc Equiv Imp`.

Import *STLC*.

The `inversion` tactic of Coq is not very satisfying for three reasons. First, it produces a bunch of equalities which one typically wants to substitute away, using `subst`. Second, it introduces meaningless names for hypotheses. Third, a call to `inversion H` does not remove *H* from the context, even though in most cases an hypothesis is no longer needed after being inverted. The tactic *inverts* address all of these three issues. It is intended to be used in place of the tactic `inversion`.

The following example illustrates how the tactic *inverts H* behaves mostly like `inversion H` except that it performs some substitutions in order to eliminate the trivial equalities that are being produced by `inversion`.

Theorem `skip_left`:  $\forall c,$   
 $\text{cequiv} (\text{SKIP};; c) \ c.$

Proof.

*introv*. `split`; `intros H`.

*dup*. - `inversion H`. `subst`. `inversion H2`. `subst`. `assumption`.

- *inverts H*. *inverts H2*. `assumption`.

Abort.

A slightly more interesting example appears next.

Theorem `ceval_deterministic`:  $\forall c \ st \ st1 \ st2,$   
 $c / st \setminus \setminus st1 \rightarrow$   
 $c / st \setminus \setminus st2 \rightarrow$

*st1* = *st2*.

Proof.

```

intros E1 E2. generalize dependent st2.
induction E1; intros st2 E2.
admit. admit.    dup.    - inversion E2. subst. admit.
- inverts E2. admit.

```

Abort.

The tactic *inverts H as*. is like *inverts H* except that the variables and hypotheses being produced are placed in the goal rather than in the context. This strategy allows naming those new variables and hypotheses explicitly, using either *intros* or *intros*.

Theorem *ceval\_deterministic*:  $\forall c \ st \ st1 \ st2$ ,

```

c / st \ \ st1  $\rightarrow$ 
c / st \ \ st2  $\rightarrow$ 
st1 = st2.

```

Proof.

```

intros E1 E2. generalize dependent st2.
(induction E1); intros st2 E2;
  inverts E2 as.
- reflexivity.
-

```

```

  subst n.
  reflexivity.
-

```

```

  intros st3 Red1 Red2.
  assert (st' = st3) as EQ1.
  { apply IHE1_1; assumption. }
  subst st3.
  apply IHE1_2. assumption.
-

```

```

  intros.
  apply IHE1. assumption.
-

```

```

  intros.
  rewrite H in H5. inversion H5.

```

Abort.

In the particular case where a call to *inversion* produces a single subgoal, one can use the syntax *inverts H as H1 H2 H3* for calling *inverts* and naming the new hypotheses *H1*, *H2* and *H3*. In other words, the tactic *inverts H as H1 H2 H3* is equivalent to *inverts H as; intros H1 H2 H3*. An example follows.

Theorem skip\_left':  $\forall c,$   
 cequiv (SKIP;;  $c$ )  $c$ .

Proof.

*intros*. split; *intros*  $H$ .  
*inverts*  $H$  as  $U$   $V$ . *inverts*  $U$ . *assumption*.

Abort.

A more involved example appears next. In particular, this example shows that the name of the hypothesis being inverted can be reused.

Example typing\_nonexample\_1 :

$\neg \exists T,$   
**has\_type** empty  
 (tabs x TBool  
   (tabs y TBool  
     (tapp (tvar x) (tvar y))))  
 $T$ .

Proof.

*dup* 3.  
 - *intros*  $C$ . *destruct*  $C$ .  
*inversion*  $H$ . *subst*. *clear*  $H$ .  
*inversion*  $H5$ . *subst*. *clear*  $H5$ .  
*inversion*  $H4$ . *subst*. *clear*  $H4$ .  
*inversion*  $H2$ . *subst*. *clear*  $H2$ .  
*inversion*  $H5$ . *subst*. *clear*  $H5$ .  
*inversion*  $H1$ .  
 - *intros*  $C$ . *destruct*  $C$ .  
*inverts*  $H$  as  $H1$ .  
*inverts*  $H1$  as  $H2$ .  
*inverts*  $H2$  as  $H3$ .  
*inverts*  $H3$  as  $H4$ .  
*inverts*  $H4$ .  
 - *intros*  $C$ . *destruct*  $C$ .  
*inverts*  $H$  as  $H$ .  
*inverts*  $H$  as  $H$ .  
*inverts*  $H$  as  $H$ .  
*inverts*  $H$  as  $H$ .  
*inverts*  $H$ .

Qed.

End INVERTSEXAMPLES.

Note: in the rare cases where one needs to perform an inversion on an hypothesis  $H$  without clearing  $H$  from the context, one can use the tactic *inverts keep H*, where the keyword *keep* indicates that the hypothesis should be kept in the context.

## 35.3 Tactics for n-ary connectives

Because Coq encodes conjunctions and disjunctions using binary constructors  $\wedge$  and  $\vee$ , working with a conjunction or a disjunction of  $N$  facts can sometimes be quite cumbersome. For this reason, "LibTactics" provides tactics offering direct support for n-ary conjunctions and disjunctions. It also provides direct support for n-ary existentials.

This section presents the following tactics:

- *splits* for decomposing n-ary conjunctions,
- *branch* for decomposing n-ary disjunctions,
- $\exists$  for proving n-ary existentials.

Module NARYEXAMPLES.

Require Import References Sflib.

Import STLCTRef.

### 35.3.1 The tactic *splits*

The tactic *splits* applies to a goal made of a conjunction of  $n$  propositions and it produces  $n$  subgoals. For example, it decomposes the goal  $G1 \wedge G2 \wedge G3$  into the three subgoals  $G1$ ,  $G2$  and  $G3$ .

Lemma demo\_splits :  $\forall n m,$   
 $n > 0 \wedge n < m \wedge m < n+10 \wedge m \neq 3.$

Proof.

intros. *splits*.

Abort.

### 35.3.2 The tactic *branch*

The tactic *branch*  $k$  can be used to prove a n-ary disjunction. For example, if the goal takes the form  $G1 \vee G2 \vee G3$ , the tactic *branch* 2 leaves only  $G2$  as subgoal. The following example illustrates the behavior of the *branch* tactic.

Lemma demo\_branch :  $\forall n m,$   
 $n < m \vee n = m \vee m < n.$

Proof.

intros.

destruct (lt\_eq\_lt\_dec  $n m$ ) as  $[[H1|H2]|H3]$ .

- *branch* 1. apply  $H1$ .

- *branch* 2. apply  $H2$ .

- *branch* 3. apply  $H3$ .

Qed.

### 35.3.3 The tactic $\exists$

The library "LibTactics" introduces a notation for n-ary existentials. For example, one can write  $\exists x y z, H$  instead of  $\exists x, \exists y, \exists z, H$ . Similarly, the library provides a n-ary tactic  $\exists a b c$ , which is a shorthand for  $\exists a; \exists b; \exists c$ . The following example illustrates both the notation and the tactic for dealing with n-ary existentials.

```
Theorem progress :  $\forall ST\ t\ T\ st$ ,
  has_type empty  $ST\ t\ T \rightarrow$ 
  store_well_typed  $ST\ st \rightarrow$ 
  value  $t \vee \exists t'\ st', t / st ==> t' / st'$ .
Proof with eauto.
  intros  $ST\ t\ T\ st\ Ht\ HST$ . remember (@empty ty) as Gamma.
  (induction Ht); subst; try solve by inversion...
-
  right. destruct IH $Ht1$  as [ $Ht1p \mid Ht1p$ ]...
  +
    inversion  $Ht1p$ ; subst; try solve by inversion.
    destruct IH $Ht2$  as [ $Ht2p \mid Ht2p$ ]...
    inversion  $Ht2p$  as [ $t2' \mid st'\ Hstep$ ].
     $\exists$  (tapp (tabs  $x\ T\ t$ )  $t2'$ )  $st'$ ...
```

Abort.

Remark: a similar facility for n-ary existentials is provided by the module *Coq.Program.Syntax* from the standard library. (*Coq.Program.Syntax* supports existentials up to arity 4; *LibTactics* supports them up to arity 10.

End NARYEXAMPLES.

## 35.4 Tactics for working with equality

One of the major weakness of Coq compared with other interactive proof assistants is its relatively poor support for reasoning with equalities. The tactics described next aims at simplifying pieces of proof scripts manipulating equalities.

This section presents the following tactics:

- *asserts\_rewrite* for introducing an equality to rewrite with,
- *cuts\_rewrite*, which is similar except that its subgoals are swapped,
- *subst* for improving the **subst** tactic,
- *fequals* for improving the **f\_equal** tactic,
- *applies\_eq* for proving  $P \times y$  using an hypothesis  $P \times z$ , automatically producing an equality  $y = z$  as subgoal.

Module EQUALITYEXAMPLES.

### 35.4.1 The tactics *asserts\_rewrite* and *cuts\_rewrite*

The tactic *asserts\_rewrite* ( $E1 = E2$ ) replaces  $E1$  with  $E2$  in the goal, and produces the goal  $E1 = E2$ .

Theorem `mult_0_plus` :  $\forall n m : \text{nat},$   
 $(0 + n) \times m = n \times m.$

Proof.

```
dup.
intros n m.
assert (H: 0 + n = n). reflexivity. rewrite → H.
reflexivity.

intros n m.
asserts_rewrite (0 + n = n).
reflexivity.      reflexivity. Qed.
```

The tactic *cuts\_rewrite* ( $E1 = E2$ ) is like *asserts\_rewrite* ( $E1 = E2$ ), except that the equality  $E1 = E2$  appears as first subgoal.

Theorem `mult_0_plus'` :  $\forall n m : \text{nat},$   
 $(0 + n) \times m = n \times m.$

Proof.

```
intros n m.
cuts_rewrite (0 + n = n).
reflexivity.      reflexivity. Qed.
```

More generally, the tactics *asserts\_rewrite* and *cuts\_rewrite* can be provided a lemma as argument. For example, one can write *asserts\_rewrite* ( $\forall a b, a*(S b) = a \times b + a$ ). This formulation is useful when  $a$  and  $b$  are big terms, since there is no need to repeat their statements.

Theorem `mult_0_plus''` :  $\forall u v w x y z : \text{nat},$   
 $(u + v) \times (S (w \times x + y)) = z.$

Proof.

```
intros. asserts_rewrite ( $\forall a b, a*(S b) = a \times b + a$ ).
Abort.
```

### 35.4.2 The tactic *substs*

The tactic *substs* is similar to *subst* except that it does not fail when the goal contains "circular equalities", such as  $x = f x$ .

Lemma `demo_substs` :  $\forall x y (f : \text{nat} \rightarrow \text{nat}),$   
 $x = f x \rightarrow y = x \rightarrow y = f x.$

Proof.

```
intros. substs.      assumption.
Qed.
```

### 35.4.3 The tactic *fequals*

The tactic *fequals* is similar to `f_equal` except that it directly discharges all the trivial subgoals produced. Moreover, the tactic *fequals* features an enhanced treatment of equalities between tuples.

```
Lemma demo_fequals : ∀ (a b c d e : nat) (f : nat → nat → nat → nat → nat),
  a = 1 → b = e → e = 2 →
  f a b c d = f 1 2 c 4.
```

Proof.

```
  intros. fequals.
```

Abort.

### 35.4.4 The tactic *applies\_eq*

The tactic *applies\_eq* is a variant of `eapply` that introduces equalities for subterms that do not unify. For example, assume the goal is the proposition  $P \times y$  and assume we have the assumption  $H$  asserting that  $P \times z$  holds. We know that we can prove  $y$  to be equal to  $z$ . So, we could call the tactic *assert\_rewrite* ( $y = z$ ) and change the goal to  $P \times z$ , but this would require copy-pasting the values of  $y$  and  $z$ . With the tactic *applies\_eq*, we can call *applies\_eq*  $H$  1, which proves the goal and leaves only the subgoal  $y = z$ . The value 1 given as argument to *applies\_eq* indicates that we want an equality to be introduced for the first argument of  $P \times y$  counting from the right. The three following examples illustrate the behavior of a call to *applies\_eq*  $H$  1, a call to *applies\_eq*  $H$  2, and a call to *applies\_eq*  $H$  1 2.

```
Axiom big_expression_using : nat → nat.
```

```
Lemma demo_applies_eq_1 : ∀ (P : nat → nat → Prop) x y z,
  P x (big_expression_using z) →
  P x (big_expression_using y).
```

Proof.

```
  introv H. dup.
```

```
  assert (Eq : big_expression_using y = big_expression_using z).
```

```
    admit. rewrite Eq. apply H.
```

```
  applies_eq H 1.
```

```
    admit. Abort.
```

If the mismatch was on the first argument of  $P$  instead of the second, we would have written *applies\_eq*  $H$  2. Recall that the occurrences are counted from the right.

```
Lemma demo_applies_eq_2 : ∀ (P : nat → nat → Prop) x y z,
  P (big_expression_using z) x →
  P (big_expression_using y) x.
```

Proof.

```
  introv H. applies_eq H 2.
```

Abort.

When we have a mismatch on two arguments, we want to produce two equalities. To achieve this, we may call *applies\_eq* *H* 1 2. More generally, the tactic *applies\_eq* expects a lemma and a sequence of natural numbers as arguments.

```
Lemma demo_applies_eq_3 : ∀ (P: nat → nat → Prop) x1 x2 y1 y2,
  P (big_expression_using x2) (big_expression_using y2) →
  P (big_expression_using x1) (big_expression_using y1).
```

Proof.

```
  introv H. applies_eq H 1 2.
```

Abort.

End EQUALITYEXAMPLES.

## 35.5 Some convenient shorthands

This section of the tutorial introduces a few tactics that help make proof scripts shorter and more readable:

- *unfolds* (without argument) for unfolding the head definition,
- *false* for replacing the goal with **False**,
- *gen* as a shorthand for **dependent generalize**,
- *skip* for skipping a subgoal even if it contains existential variables,
- *sort* for re-ordering the proof context by moving moving all propositions at the bottom.

### 35.5.1 The tactic *unfolds*

Module UNFOLDSEXAMPLE.

```
  Require Import Hoare.
```

The tactic *unfolds* (without any argument) unfolds the head constant of the goal. This tactic saves the need to name the constant explicitly.

```
Lemma bexp_eval_true : ∀ b st,
  beval st b = true → (bassn b) st.
```

Proof.

```
  intros b st Hbe. dup.
```

```
  unfold bassn. assumption.
```

```
  unfolds. assumption.
```

Qed.

Remark: contrary to the tactic *hnf*, which may unfold several constants, *unfolds* performs only a single step of unfolding.



Remark: the tactic *unfolds* in *H* can be used to unfold the head definition of the hypothesis *H*.

End UNFOLDEXAMPLE.

### 35.5.2 The tactics *false* and *tryfalse*

The tactic *false* can be used to replace any goal with **False**. In short, it is a shorthand for *exfalso*. Moreover, *false* proves the goal if it contains an absurd assumption, such as **False** or  $0 = S\ n$ , or if it contains contradictory assumptions, such as  $x = \text{true}$  and  $x = \text{false}$ .

Lemma *demo\_false* :

$\forall n, S\ n = 1 \rightarrow n = 0.$

Proof.

*intros. destruct n. reflexivity. false.*

Qed.

The tactic *false* can be given an argument: *false H* replace the goals with **False** and then applies *H*.

Lemma *demo\_false\_arg* :

$(\forall n, n < 0 \rightarrow \text{False}) \rightarrow (3 < 0) \rightarrow 4 < 0.$

Proof.

*intros H L. false H. apply L.*

Qed.

The tactic *tryfalse* is a shorthand for *try solve [false]*: it tries to find a contradiction in the goal. The tactic *tryfalse* is generally called after a case analysis.

Lemma *demo\_tryfalse* :

$\forall n, S\ n = 1 \rightarrow n = 0.$

Proof.

*intros. destruct n; tryfalse. reflexivity.*

Qed.

### 35.5.3 The tactic *gen*

The tactic *gen* is a shorthand for *generalize dependent* that accepts several arguments at once. An invocation of this tactic takes the form *gen x y z*.

Module GENEXAMPLE.

Require Import Stlc.

Import STLC.

Lemma *substitution\_preserves\_typing* :  $\forall\ Gamma\ x\ U\ v\ t\ S,$

**has\_type** (update *Gamma* *x* *U*) *t* *S*  $\rightarrow$

**has\_type** empty *v* *U*  $\rightarrow$

**has\_type** *Gamma* ( $[x := v]\ t$ ) *S*.

Proof.

*dup.*

intros *Gamma x U v t S Htypt Htypv.*

generalize dependent *S*. generalize dependent *Gamma*.

induction *t*; intros; simpl.

*admit. admit. admit. admit. admit. admit.*

*introv Htypt Htypv. gen S Gamma.*

induction *t*; intros; simpl.

*admit. admit. admit. admit. admit. admit.*

Abort.

End GENEXAMPLE.

### 35.5.4 The tactics *skip*, *skip\_rewrite* and *skip\_goal*

Temporarily admitting a given subgoal is very useful when constructing proofs. It gives the ability to focus first on the most interesting cases of a proof. The tactic *skip* is like *admit* except that it also works when the proof includes existential variables. Recall that existential variables are those whose name starts with a question mark, (e.g., ?24), and which are typically introduced by *eapply*.

Module SKIPEXAMPLE.

Require Import Stlc.

Import STLC.

Example astep\_example1 :

(APlus (ANum 3) (AMult (ANum 3) (ANum 4))) / empty\_state ==>a× (ANum 15).

Proof.

*eapply multi\_step. skip. eapply multi\_step. skip. skip.*

Abort.

The tactic *skip H: P* adds the hypothesis *H: P* to the context, without checking whether the proposition *P* is true. It is useful for exploiting a fact and postponing its proof. Note: *skip H: P* is simply a shorthand for *assert (H:P). skip*.

Theorem demo\_skipH : **True**.

Proof.

*skip H: (∀ n m : nat, (0 + n) × m = n × m).*

Abort.

The tactic *skip\_rewrite (E1 = E2)* replaces *E1* with *E2* in the goal, without checking that *E1* is actually equal to *E2*.

Theorem mult\_0\_plus : ∀ n m : **nat**,

**(0 + n) × m = n × m.**

Proof.

*dup.*

```

intros n m.
assert (H: 0 + n = n). skip. rewrite → H.
reflexivity.

intros n m.
skip_rewrite (0 + n = n).
reflexivity.
Qed.

```

Remark: the tactic *skip\_rewrite* can in fact be given a lemma statement as argument, in the same way as *asserts\_rewrite*.

The tactic *skip\_goal* adds the current goal as hypothesis. This cheat is useful to set up the structure of a proof by induction without having to worry about the induction hypothesis being applied only to smaller arguments. Using *skip\_goal*, one can construct a proof in two steps: first, check that the main arguments go through without waisting time on fixing the details of the induction hypotheses; then, focus on fixing the invocations of the induction hypothesis.

```

Theorem ceval_deterministic: ∀ c st st1 st2,
  c / st \\\ st1 →
  c / st \\\ st2 →
  st1 = st2.

```

Proof.

```

skip_goal.
introv E1 E2. gen st2.
(induction E1); introv E2; inverts E2 as.
- reflexivity.
-
  subst n.
  reflexivity.
-
  intros st3 Red1 Red2.
  assert (st' = st3) as EQ1.
  {

    eapply IH. eapply E1_1. eapply Red1. }
  subst st3.
  eapply IH. eapply E1_2. eapply Red2.
Abort.
End SKIPEXAMPLE.

```

### 35.5.5 The tactic *sort*

Module SORTEXAMPLES.

Require Import Imp.

The tactic *sort* reorganizes the proof context by placing all the variables at the top and all the hypotheses at the bottom, thereby making the proof context more readable.

Theorem *ceval\_deterministic*:  $\forall c\ st\ st1\ st2,$

$c / st \setminus\setminus st1 \rightarrow$

$c / st \setminus\setminus st2 \rightarrow$

$st1 = st2.$

Proof.

intros *c st st1 st2 E1 E2*.

generalize dependent *st2*.

(induction *E1*); intros *st2 E2*; inverts *E2*.

admit. admit. sort. Abort.

End SORTEXAMPLES.

## 35.6 Tactics for advanced lemma instantiation

This last section describes a mechanism for instantiating a lemma by providing some of its arguments and leaving other implicit. Variables whose instantiation is not provided are turned into existential variables, and facts whose instantiation is not provided are turned into subgoals.

Remark: this instantiation mechanism goes far beyond the abilities of the "Implicit Arguments" mechanism. The point of the instantiation mechanism described in this section is that you will no longer need to spend time figuring out how many underscore symbols you need to write.

In this section, we'll use a useful feature of Coq for decomposing conjunctions and existentials. In short, a tactic like *intros* or *destruct* can be provided with a pattern (*H1* & *H2* & *H3* & *H4* & *H5*), which is a shorthand for [*H1* [*H2* [*H3* [*H4* *H5*]]]]]. For example, *destruct* (*H* \_ \_ \_ *Htypt*) as [*T* [*Hctx* *Hsub*]]. can be rewritten in the form *destruct* (*H* \_ \_ \_ *Htypt*) as (*T* & *Hctx* & *Hsub*).

### 35.6.1 Working of *lets*

When we have a lemma (or an assumption) that we want to exploit, we often need to explicitly provide arguments to this lemma, writing something like: *destruct* (*typing\_inversion\_var* \_ \_ \_ *Htypt*) as (*T* & *Hctx* & *Hsub*). The need to write several times the "underscore" symbol is tedious. Not only we need to figure out how many of them to write down, but it also makes the proof scripts look prettly ugly. With the tactic *lets*, one can simply write: *lets* (*T* & *Hctx* & *Hsub*): *typing\_inversion\_var Htypt*.

In short, this tactic *lets* allows to specialize a lemma on a bunch of variables and hypotheses. The syntax is *lets* !: *E0 E1 .. EN*, for building an hypothesis named ! by applying the fact *E0* to the arguments *E1* to *EN*. Not all the arguments need to be provided, however

the arguments that are provided need to be provided in the correct order. The tactic relies on a first-match algorithm based on types in order to figure out how to instantiate the lemma with the arguments provided.

Module EXAMPLESLETS.

Require Import Sub.

Axiom *typing\_inversion\_var* :  $\forall (G:\text{context}) (x:\text{id}) (T:\text{ty}),$   
**has\_type** *G* (tvar *x*) *T*  $\rightarrow$   
 $\exists S, G\ x = \text{Some } S \wedge \text{subtype } S\ T.$

First, assume we have an assumption *H* with the type of the form **has\_type** *G* (tvar *x*) *T*. We can obtain the conclusion of the lemma *typing\_inversion\_var* by invoking the tactics *lets K: typing\_inversion\_var H*, as shown next.

Lemma demo\_lets\_1 :  $\forall (G:\text{context}) (x:\text{id}) (T:\text{ty}),$   
**has\_type** *G* (tvar *x*) *T*  $\rightarrow$  **True**.

Proof.

intros *G x T H*. *dup*.

*lets K: typing\_inversion\_var H*.

destruct *K* as (*S & Eq & Sub*).

*admit*.

*lets (S & Eq & Sub): typing\_inversion\_var H*.

*admit*.

Abort.

Assume now that we know the values of *G*, *x* and *T* and we want to obtain *S*, and have **has\_type** *G* (tvar *x*) *T* be produced as a subgoal. To indicate that we want all the remaining arguments of *typing\_inversion\_var* to be produced as subgoals, we use a triple-underscore symbol *----*. (We'll later introduce a shorthand tactic called *forwards* to avoid writing triple underscores.)

Lemma demo\_lets\_2 :  $\forall (G:\text{context}) (x:\text{id}) (T:\text{ty}),$  **True**.

Proof.

intros *G x T*.

*lets (S & Eq & Sub): typing\_inversion\_var G x T ----*.

Abort.

Usually, there is only one context *G* and one type *T* that are going to be suitable for proving **has\_type** *G* (tvar *x*) *T*, so we don't really need to bother giving *G* and *T* explicitly. It suffices to call *lets (S & Eq & Sub): typing\_inversion\_var x*. The variables *G* and *T* are then instantiated using existential variables.

Lemma demo\_lets\_3 :  $\forall (x:\text{id}),$  **True**.

Proof.

intros *x*.

*lets (S & Eq & Sub): typing\_inversion\_var x ----*.

Abort.

We may go even further by not giving any argument to instantiate *typing\_inversion\_var*. In this case, three unification variables are introduced.

Lemma demo\_lets\_4 : **True**.

Proof.

*lets* (*S* & *Eq* & *Sub*): *typing\_inversion\_var* ---.

Abort.

Note: if we provide *lets* with only the name of the lemma as argument, it simply adds this lemma in the proof context, without trying to instantiate any of its arguments.

Lemma demo\_lets\_5 : **True**.

Proof.

*lets* *H*: *typing\_inversion\_var*.

Abort.

A last useful feature of *lets* is the double-underscore symbol, which allows skipping an argument when several arguments have the same type. In the following example, our assumption quantifies over two variables *n* and *m*, both of type *nat*. We would like *m* to be instantiated as the value 3, but without specifying a value for *n*. This can be achieved by writing *lets* *K*: *H* \_\_ 3.

Lemma demo\_lets\_underscore :

$(\forall n\ m, n \leq m \rightarrow n < m+1) \rightarrow$  **True**.

Proof.

*intros* *H*.

*lets* *K*: *H* 3.      *clear* *K*.

*lets* *K*: *H* \_\_ 3.      *clear* *K*.

Abort.

Note: one can write *lets*: *E0 E1 E2* in place of *lets* *H*: *E0 E1 E2*. In this case, the name *H* is chosen arbitrarily.

Note: the tactics *lets* accepts up to five arguments. Another syntax is available for providing more than five arguments. It consists in using a list introduced with the special symbol *»*, for example *lets* *H*: (*» E0 E1 E2 E3 E4 E5 E6 E7 E8 E9* 10).

End EXAMPLESLETS.

## 35.6.2 Working of *applys*, *forwards* and *specializes*

The tactics *applys*, *forwards* and *specializes* are shorthand that may be used in place of *lets* to perform specific tasks.

- *forwards* is a shorthand for instantiating all the arguments

of a lemma. More precisely, *forwards* *H*: *E0 E1 E2 E3* is the same as *lets* *H*: *E0 E1 E2 E3* ---, where the triple-underscore has the same meaning as explained earlier on.

- *applies* allows building a lemma using the advanced instantiation

mode of *lets*, and then apply that lemma right away. So, *applies E0 E1 E2 E3* is the same as *lets H: E0 E1 E2 E3* followed with *eapply H* and then *clear H*.

- *specializes* is a shorthand for instantiating in-place

an assumption from the context with particular arguments. More precisely, *specializes H E0 E1* is the same as *lets H': H E0 E1* followed with *clear H* and *rename H' into H*.

Examples of use of *applies* appear further on. Several examples of use of *forwards* can be found in the tutorial chapter *UseAuto*.

### 35.6.3 Example of instantiations

Module EXAMPLESINSTANTIATIONS.

Require Import Sub.

The following proof shows several examples where *lets* is used instead of **destruct**, as well as examples where *applies* is used instead of **apply**. The proof also contains some holes that you need to fill in as an exercise.

Lemma substitution\_preserves\_typing :  $\forall \text{Gamma } x \text{ } U \text{ } v \text{ } t \text{ } S$ ,

**has\_type** (update Gamma x U) t S  $\rightarrow$

**has\_type** empty v U  $\rightarrow$

**has\_type** Gamma ([x:=v] t) S.

Proof with eauto.

intros Gamma x U v t S Htypt Htypv.

generalize dependent S. generalize dependent Gamma.

(induction t); intros; simpl.

-

rename i into y.

*lets* (T&Hctx&Hsub): typing\_inversion\_var Htypt.

unfold update, t\_update in Hctx.

destruct (beq\_idP x y)...

+

subst.

inversion Hctx; subst. clear Hctx.

apply context\_invariance with empty...

intros x Hcontra.

*lets* [T' HT']: free\_in\_context S (@empty ty) Hcontra...

inversion HT'.

-

```

    admit.
-
    rename i into y. rename t into T1.
lets (T2&Hsub&Htypt2): typing_inversion_abs Htypt.
applies T_Sub (TArrow T1 T2)...
  apply T_Abs...
  destruct (beq_idP x y).
+
  eapply context_invariance...
  subst.
  intros x Hafi. unfold update, t_update.
  destruct (beq_idP y x)...
+
  apply IHt. eapply context_invariance...
  intros z Hafi. unfold update, t_update.
  destruct (beq_idP y z)...
  subst. rewrite false_beq_id...
-
lets: typing_inversion_true Htypt...
-
lets: typing_inversion_false Htypt...
-
lets (Htyp1&Htyp2&Htyp3): typing_inversion_if Htypt...
-

lets: typing_inversion_unit Htypt...
Admitted.
End EXAMPLESINSTANTIATIONS.

```

## 35.7 Summary

In this chapter we have presented a number of tactics that help make proof script more concise and more robust on change.

- *introv* and *inverts* improve naming and inversions.
- *false* and *tryfalse* help discarding absurd goals.
- *unfolds* automatically calls `unfold` on the head definition.



- *gen* helps setting up goals for induction.
- *cases* and *cases\_if* help with case analysis.
- *splits*, *branch* and  $\exists$  to deal with n-ary constructs.
- *asserts\_rewrite*, *cuts\_rewrite*, *subst* and *fequals* help working with equalities.
- *lets*, *forwards*, *specializes* and *applies* provide means of very conveniently instantiating lemmas.
- *applies\_eq* can save the need to perform manual rewriting steps before being able to apply lemma.
- *skip*, *skip\_rewrite* and *skip\_goal* give the flexibility to choose which subgoals to try and discharge first.

Making use of these tactics can boost one's productivity in Coq proofs.

If you are interested in using *LibTactics.v* in your own developments, make sure you get the latest version from: <http://www.chargueraud.org/softs/tlc/> .

*Date* : 2016 – 03 – 04 09 : 33 : 20 – 0500(*Fri*, 04*Mar*2016)

# Chapter 36

## Library UseAuto

### 36.1 UseAuto: Theory and Practice of Automation in Coq Proofs

In a machine-checked proof, every single detail has to be justified. This can result in huge proof scripts. Fortunately, Coq comes with a proof-search mechanism and with several decision procedures that enable the system to automatically synthesize simple pieces of proof. Automation is very powerful when set up appropriately. The purpose of this chapter is to explain the basics of working of automation.

The chapter is organized in two parts. The first part focuses on a general mechanism called "proof search." In short, proof search consists in naively trying to apply lemmas and assumptions in all possible ways. The second part describes "decision procedures", which are tactics that are very good at solving proof obligations that fall in some particular fragment of the logic of Coq.

Many of the examples used in this chapter consist of small lemmas that have been made up to illustrate particular aspects of automation. These examples are completely independent from the rest of the Software Foundations course. This chapter also contains some bigger examples which are used to explain how to use automation in realistic proofs. These examples are taken from other chapters of the course (mostly from STLC), and the proofs that we present make use of the tactics from the library *LibTactics.v*, which is presented in the chapter UseTactics.

```
Require Import Coq.Arith.Arith.  
Require Import Coq.Lists.List.  
Import ListNotations.  
  
Require Import Sflib.  
Require Import Maps.  
Require Import Stlc.  
Require Import LibTactics.
```

## 36.2 Basic Features of Proof Search

The idea of proof search is to replace a sequence of tactics applying lemmas and assumptions with a call to a single tactic, for example `auto`. This form of proof automation saves a lot of effort. It typically leads to much shorter proof scripts, and to scripts that are typically more robust to change. If one makes a little change to a definition, a proof that exploits automation probably won't need to be modified at all. Of course, using too much automation is a bad idea. When a proof script no longer records the main arguments of a proof, it becomes difficult to fix it when it gets broken after a change in a definition. Overall, a reasonable use of automation is generally a big win, as it saves a lot of time both in building proof scripts and in subsequently maintaining those proof scripts.

### 36.2.1 Strength of Proof Search

We are going to study four proof-search tactics: `auto`, `eauto`, `iauto` and `jauto`. The tactics `auto` and `eauto` are builtin in Coq. The tactic `iauto` is a shorthand for the builtin tactic `try solve [intuition eauto]`. The tactic `jauto` is defined in the library `LibTactics`, and simply performs some preprocessing of the goal before calling `eauto`. The goal of this chapter is to explain the general principles of proof search and to give rule of thumbs for guessing which of the four tactics mentioned above is best suited for solving a given goal.

Proof search is a compromise between efficiency and expressiveness, that is, a tradeoff between how complex goals the tactic can solve and how much time the tactic requires for terminating. The tactic `auto` builds proofs only by using the basic tactics `reflexivity`, `assumption`, and `apply`. The tactic `eauto` can also exploit `eapply`. The tactic `jauto` extends `eauto` by being able to open conjunctions and existentials that occur in the context. The tactic `iauto` is able to deal with conjunctions, disjunctions, and negation in a quite clever way; however it is not able to open existentials from the context. Also, `iauto` usually becomes very slow when the goal involves several disjunctions.

Note that proof search tactics never perform any rewriting step (tactics `rewrite`, `subst`), nor any case analysis on an arbitrary data structure or property (tactics `destruct` and `inversion`), nor any proof by induction (tactic `induction`). So, proof search is really intended to automate the final steps from the various branches of a proof. It is not able to discover the overall structure of a proof.

### 36.2.2 Basics

The tactic `auto` is able to solve a goal that can be proved using a sequence of `intros`, `apply`, `assumption`, and `reflexivity`. Two examples follow. The first one shows the ability for `auto` to call `reflexivity` at any time. In fact, calling `reflexivity` is always the first thing that `auto` tries to do.

Lemma `solving_by_reflexivity` :

$2 + 3 = 5$ .

Proof. `auto`. Qed.

The second example illustrates a proof where a sequence of two calls to `apply` are needed. The goal is to prove that if  $Q\ n$  implies  $P\ n$  for any  $n$  and if  $Q\ n$  holds for any  $n$ , then  $P\ 2$  holds.

```
Lemma solving_by_apply : ∀ (P Q : nat → Prop),
  (∀ n, Q n → P n) →
  (∀ n, Q n) →
  P 2.
```

Proof. `auto. Qed.`

If we are interested to see which proof `auto` came up with, one possibility is to look at the generated proof-term, using the command:

```
Print solving_by_apply.
```

The proof term is:

```
fun (P Q : nat → Prop) (H : ∀ n : nat, Q n → P n) (H0 : ∀ n : nat, Q n) => H 2 (H0 2)
```

This essentially means that `auto` applied the hypothesis  $H$  (the first one), and then applied the hypothesis  $H0$  (the second one).

The tactic `auto` can invoke `apply` but not `eapply`. So, `auto` cannot exploit lemmas whose instantiation cannot be directly deduced from the proof goal. To exploit such lemmas, one needs to invoke the tactic `eauto`, which is able to call `eapply`.

In the following example, the first hypothesis asserts that  $P\ n$  is true when  $Q\ m$  is true for some  $m$ , and the goal is to prove that  $Q\ 1$  implies  $P\ 2$ . This implication follows direction from the hypothesis by instantiating  $m$  as the value 1. The following proof script shows that `eauto` successfully solves the goal, whereas `auto` is not able to do so.

```
Lemma solving_by_eapply : ∀ (P Q : nat → Prop),
  (∀ n m, Q m → P n) →
  Q 1 → P 2.
```

Proof. `auto. eauto. Qed.`

### 36.2.3 Conjunctions

So far, we've seen that `eauto` is stronger than `auto` in the sense that it can deal with `eapply`. In the same way, we are going to see how `jauto` and `iauto` are stronger than `auto` and `eauto` in the sense that they provide better support for conjunctions.

The tactics `auto` and `eauto` can prove a goal of the form  $F \wedge F'$ , where  $F$  and  $F'$  are two propositions, as soon as both  $F$  and  $F'$  can be proved in the current context. An example follows.

```
Lemma solving_conj_goal : ∀ (P : nat → Prop) (F : Prop),
  (∀ n, P n) → F → F ∧ P 2.
```

Proof. `auto. Qed.`

However, when an assumption is a conjunction, `auto` and `eauto` are not able to exploit this conjunction. It can be quite surprising at first that `eauto` can prove very complex goals

but that it fails to prove that  $F \wedge F'$  implies  $F$ . The tactics *iauto* and *jauto* are able to decompose conjunctions from the context. Here is an example.

Lemma solving\_conj\_hyp :  $\forall (F F' : \text{Prop}),$   
 $F \wedge F' \rightarrow F.$

Proof. auto. eauto. jauto. Qed.

The tactic *jauto* is implemented by first calling a pre-processing tactic called *jauto\_set*, and then calling *eauto*. So, to understand how *jauto* works, one can directly call the tactic *jauto\_set*.

Lemma solving\_conj\_hyp' :  $\forall (F F' : \text{Prop}),$   
 $F \wedge F' \rightarrow F.$

Proof. intros. jauto\_set. eauto. Qed.

Next is a more involved goal that can be solved by *iauto* and *jauto*.

Lemma solving\_conj\_more :  $\forall (P Q R : \text{nat} \rightarrow \text{Prop}) (F : \text{Prop}),$   
 $(F \wedge (\forall n m, (Q m \wedge R n) \rightarrow P n)) \rightarrow$   
 $(F \rightarrow R 2) \rightarrow$   
 $Q 1 \rightarrow$   
 $P 2 \wedge F.$

Proof. jauto. Qed.

The strategy of *iauto* and *jauto* is to run a global analysis of the top-level conjunctions, and then call *eauto*. For this reason, those tactics are not good at dealing with conjunctions that occur as the conclusion of some universally quantified hypothesis. The following example illustrates a general weakness of Coq proof search mechanisms.

Lemma solving\_conj\_hyp\_forall :  $\forall (P Q : \text{nat} \rightarrow \text{Prop}),$   
 $(\forall n, P n \wedge Q n) \rightarrow P 2.$

Proof.

auto. eauto. iauto. jauto.  
 intros. destruct (H 2). auto.

Qed.

This situation is slightly disappointing, since automation is able to prove the following goal, which is very similar. The only difference is that the universal quantification has been distributed over the conjunction.

Lemma solved\_by\_jauto :  $\forall (P Q : \text{nat} \rightarrow \text{Prop}) (F : \text{Prop}),$   
 $(\forall n, P n) \wedge (\forall n, Q n) \rightarrow P 2.$

Proof. jauto. Qed.

### 36.2.4 Disjunctions

The tactics *auto* and *eauto* can handle disjunctions that occur in the goal.

Lemma solving\_disj\_goal :  $\forall (F F' : \text{Prop}),$   
 $F \rightarrow F \vee F'.$

Proof. auto. Qed.

However, only *iauto* is able to automate reasoning on the disjunctions that appear in the context. For example, *iauto* can prove that  $F \vee F'$  entails  $F' \vee F$ .

Lemma solving\_disj\_hyp :  $\forall (F F' : \text{Prop}),$   
 $F \vee F' \rightarrow F' \vee F.$

Proof. auto. eauto. jauto. iauto. Qed.

More generally, *iauto* can deal with complex combinations of conjunctions, disjunctions, and negations. Here is an example.

Lemma solving\_tauto :  $\forall (F1 F2 F3 : \text{Prop}),$   
 $((\neg F1 \wedge F3) \vee (F2 \wedge \neg F3)) \rightarrow$   
 $(F2 \rightarrow F1) \rightarrow$   
 $(F2 \rightarrow F3) \rightarrow$   
 $\neg F2.$

Proof. iauto. Qed.

However, the ability of *iauto* to automatically perform a case analysis on disjunctions comes with a downside: *iauto* may be very slow. If the context involves several hypotheses with disjunctions, *iauto* typically generates an exponential number of subgoals on which *eauto* is called. One major advantage of *jauto* compared with *iauto* is that it never spends time performing this kind of case analyses.

### 36.2.5 Existentials

The tactics *eauto*, *iauto*, and *jauto* can prove goals whose conclusion is an existential. For example, if the goal is  $\exists x, f x$ , the tactic *eauto* introduces an existential variable, say ?25, in place of  $x$ . The remaining goal is  $f ?25$ , and *eauto* tries to solve this goal, allowing itself to instantiate ?25 with any appropriate value. For example, if an assumption  $f 2$  is available, then the variable ?25 gets instantiated with 2 and the goal is solved, as shown below.

Lemma solving\_exists\_goal :  $\forall (f : \text{nat} \rightarrow \text{Prop}),$   
 $f 2 \rightarrow \exists x, f x.$

Proof.

auto. eauto. Qed.

A major strength of *jauto* over the other proof search tactics is that it is able to exploit the existentially-quantified hypotheses, i.e., those of the form  $\exists x, P$ .

Lemma solving\_exists\_hyp :  $\forall (f g : \text{nat} \rightarrow \text{Prop}),$   
 $(\forall x, f x \rightarrow g x) \rightarrow$   
 $(\exists a, f a) \rightarrow$   
 $(\exists a, g a).$

Proof.

auto. eauto. iauto. jauto. Qed.

### 36.2.6 Negation

The tactics `auto` and `eauto` suffer from some limitations with respect to the manipulation of negations, mostly related to the fact that negation, written  $\neg P$ , is defined as  $P \rightarrow \mathbf{False}$  but that the unfolding of this definition is not performed automatically. Consider the following example.

```
Lemma negation_study_1 :  $\forall (P : \mathbf{nat} \rightarrow \mathbf{Prop}),$   
   $P\ 0 \rightarrow (\forall x, \neg P\ x) \rightarrow \mathbf{False}.$ 
```

Proof.

```
  intros P H0 HX.
```

```
  eauto.    unfold not in *. eauto.
```

Qed.

For this reason, the tactics `iauto` and `jauto` systematically invoke `unfold not in *` as part of their pre-processing. So, they are able to solve the previous goal right away.

```
Lemma negation_study_2 :  $\forall (P : \mathbf{nat} \rightarrow \mathbf{Prop}),$   
   $P\ 0 \rightarrow (\forall x, \neg P\ x) \rightarrow \mathbf{False}.$ 
```

Proof. `jauto`. Qed.

We will come back later on to the behavior of proof search with respect to the unfolding of definitions.

### 36.2.7 Equalities

Coq's proof-search feature is not good at exploiting equalities. It can do very basic operations, like exploiting reflexivity and symmetry, but that's about it. Here is a simple example that `auto` can solve, by first calling `symmetry` and then applying the hypothesis.

```
Lemma equality_by_auto :  $\forall (f\ g : \mathbf{nat} \rightarrow \mathbf{Prop}),$   
   $(\forall x, f\ x = g\ x) \rightarrow g\ 2 = f\ 2.$ 
```

Proof. `auto`. Qed.

To automate more advanced reasoning on equalities, one should rather try to use the tactic `congruence`, which is presented at the end of this chapter in the "Decision Procedures" section.

## 36.3 How Proof Search Works

### 36.3.1 Search Depth

The tactic `auto` works as follows. It first tries to call `reflexivity` and `assumption`. If one of these calls solves the goal, the job is done. Otherwise `auto` tries to apply the most recently introduced assumption that can be applied to the goal without producing an error. This application produces subgoals. There are two possible cases. If the subgoals produced can be solved by a recursive call to `auto`, then the job is done. Otherwise, if this application

produces at least one subgoal that `auto` cannot solve, then `auto` starts over by trying to apply the second most recently introduced assumption. It continues in a similar fashion until it finds a proof or until no assumption remains to be tried.

It is very important to have a clear idea of the backtracking process involved in the execution of the `auto` tactic; otherwise its behavior can be quite puzzling. For example, `auto` is not able to solve the following triviality.

Lemma `search_depth_0` :

**`True ∧ True ∧ True ∧ True ∧ True ∧ True.`**

Proof.

`auto.`

Abort.

The reason `auto` fails to solve the goal is because there are too many conjunctions. If there had been only five of them, `auto` would have successfully solved the proof, but six is too many. The tactic `auto` limits the number of lemmas and hypotheses that can be applied in a proof, so as to ensure that the proof search eventually terminates. By default, the maximal number of steps is five. One can specify a different bound, writing for example `auto 6` to search for a proof involving at most six steps. For example, `auto 6` would solve the previous lemma. (Similarly, one can invoke `eauto 6` or `intuition eauto 6`.) The argument `n` of `auto n` is called the "search depth." The tactic `auto` is simply defined as a shorthand for `auto 5`.

The behavior of `auto n` can be summarized as follows. It first tries to solve the goal using `reflexivity` and `assumption`. If this fails, it tries to apply a hypothesis (or a lemma that has been registered in the hint database), and this application produces a number of subgoals. The tactic `auto (n-1)` is then called on each of those subgoals. If all the subgoals are solved, the job is completed, otherwise `auto n` tries to apply a different hypothesis.

During the process, `auto n` calls `auto (n-1)`, which in turn might call `auto (n-2)`, and so on. The tactic `auto 0` only tries `reflexivity` and `assumption`, and does not try to apply any lemma. Overall, this means that when the maximal number of steps allowed has been exceeded, the `auto` tactic stops searching and backtracks to try and investigate other paths.

The following lemma admits a unique proof that involves exactly three steps. So, `auto n` proves this goal iff `n` is greater than three.

Lemma `search_depth_1` :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$

$P\ 0 \rightarrow$   
 $(P\ 0 \rightarrow P\ 1) \rightarrow$   
 $(P\ 1 \rightarrow P\ 2) \rightarrow$   
 $(P\ 2).$

Proof.

`auto 0. auto 1. auto 2. auto 3. Qed.`

We can generalize the example by introducing an assumption asserting that  $P\ k$  is derivable from  $P\ (k-1)$  for all  $k$ , and keep the assumption  $P\ 0$ . The tactic `auto`, which is the same as `auto 5`, is able to derive  $P\ k$  for all values of  $k$  less than 5. For example, it can prove  $P\ 4$ .



```

Lemma search_depth_3 :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$ 
  ( $P\ 0$ )  $\rightarrow$ 
  ( $\forall k, P\ (k-1) \rightarrow P\ k$ )  $\rightarrow$ 
  ( $P\ 4$ ).

```

Proof. auto. Qed.

However, to prove  $P\ 5$ , one needs to call at least auto 6.

```

Lemma search_depth_4 :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$ 
  ( $P\ 0$ )  $\rightarrow$ 
  ( $\forall k, P\ (k-1) \rightarrow P\ k$ )  $\rightarrow$ 
  ( $P\ 5$ ).

```

Proof. auto. auto 6. Qed.

Because auto looks for proofs at a limited depth, there are cases where auto can prove a goal  $F$  and can prove a goal  $F'$  but cannot prove  $F \wedge F'$ . In the following example, auto can prove  $P\ 4$  but it is not able to prove  $P\ 4 \wedge P\ 4$ , because the splitting of the conjunction consumes one proof step. To prove the conjunction, one needs to increase the search depth, using at least auto 6.

```

Lemma search_depth_5 :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$ 
  ( $P\ 0$ )  $\rightarrow$ 
  ( $\forall k, P\ (k-1) \rightarrow P\ k$ )  $\rightarrow$ 
  ( $P\ 4 \wedge P\ 4$ ).

```

Proof. auto. auto 6. Qed.

### 36.3.2 Backtracking

In the previous section, we have considered proofs where at each step there was a unique assumption that auto could apply. In general, auto can have several choices at every step. The strategy of auto consists of trying all of the possibilities (using a depth-first search exploration).

To illustrate how automation works, we are going to extend the previous example with an additional assumption asserting that  $P\ k$  is also derivable from  $P\ (k+1)$ . Adding this hypothesis offers a new possibility that auto could consider at every step.

There exists a special command that one can use for tracing all the steps that proof-search considers. To view such a trace, one should write `debug eauto`. (For some reason, the command `debug auto` does not exist, so we have to use the command `debug eauto` instead.)

```

Lemma working_of_auto_1 :  $\forall (P : \text{nat} \rightarrow \text{Prop}),$ 
  ( $P\ 0$ )  $\rightarrow$ 
  ( $\forall k, P\ (k-1) \rightarrow P\ k$ )  $\rightarrow$ 
  ( $\forall k, P\ (k+1) \rightarrow P\ k$ )  $\rightarrow$ 
  ( $P\ 2$ ).

```

Proof. intros  $P\ H1\ H2\ H3$ . eauto. Qed.

The output message produced by `debug eauto` is as follows.

```

depth=5
depth=4 apply H2
depth=3 apply H2
depth=3 exact H1

```

The depth indicates the value of `n` with which `eauto n` is called. The tactics shown in the message indicate that the first thing that `eauto` has tried to do is to apply `H2`. The effect of applying `H2` is to replace the goal `P 2` with the goal `P 1`. Then, again, `H2` has been applied, changing the goal `P 1` into `P 0`. At that point, the goal was exactly the hypothesis `H1`.

It seems that `eauto` was quite lucky there, as it never even tried to use the hypothesis `H3` at any time. The reason is that `auto` always tried to use the `H2` first. So, let's permute the hypotheses `H2` and `H3` and see what happens.

Lemma `working_of_auto_2` :  $\forall (P : \text{nat} \rightarrow \text{Prop})$ ,

```

(P 0) →
(∀ k, P (k+1) → P k) →
(∀ k, P (k-1) → P k) →
(P 2).

```

Proof. `intros P H1 H3 H2. eauto. Qed.`

This time, the output message suggests that the proof search investigates many possibilities. If we print the proof term:

`Print working_of_auto_2.`

we observe that the proof term refers to `H3`. Thus the proof is not the simplest one, since only `H2` and `H1` are needed.

It turns out that the proof goes through the proof obligation `P 3`, even though it is not required to do so. The following tree drawing describes all the goals that `eauto` has been going through.

|5||4||3||2||1||0| -- below, tabulation indicates the depth

```

[P 2]
-> [P 3]
    -> [P 4]
        -> [P 5]
            -> [P 6]
                -> [P 7]
                    -> [P 5]
                        -> [P 4]
                            -> [P 5]
                                -> [P 3]
                                    --> [P 3]
                                        -> [P 4]
                                            -> [P 5]
                                                -> [P 3]

```

```

-> [P 2]
  -> [P 3]
    -> [P 1]
-> [P 2]
  -> [P 3]
    -> [P 4]
      -> [P 5]
        -> [P 3]
          -> [P 2]
            -> [P 3]
              -> [P 1]
-> [P 1]
  -> [P 2]
    -> [P 3]
      -> [P 1]
-> [P 0]
  -> !! Done !!

```

The first few lines read as follows. To prove  $P\ 2$ , `eauto` 5 has first tried to apply  $H3$ , producing the subgoal  $P\ 3$ . To solve it, `eauto` 4 has tried again to apply  $H3$ , producing the goal  $P\ 4$ . Similarly, the search goes through  $P\ 5$ ,  $P\ 6$  and  $P\ 7$ . When reaching  $P\ 7$ , the tactic `eauto` 0 is called but as it is not allowed to try and apply any lemma, it fails. So, we come back to the goal  $P\ 6$ , and try this time to apply hypothesis  $H2$ , producing the subgoal  $P\ 5$ . Here again, `eauto` 0 fails to solve this goal.

The process goes on and on, until backtracking to  $P\ 3$  and trying to apply  $H3$  three times in a row, going through  $P\ 2$  and  $P\ 1$  and  $P\ 0$ . This search tree explains why `eauto` came up with a proof term starting with an application of  $H3$ .

### 36.3.3 Adding Hints

By default, `auto` (and `eauto`) only tries to apply the hypotheses that appear in the proof context. There are two possibilities for telling `auto` to exploit a lemma that have been proved previously: either adding the lemma as an assumption just before calling `auto`, or adding the lemma as a hint, so that it can be used by every calls to `auto`.

The first possibility is useful to have `auto` exploit a lemma that only serves at this particular point. To add the lemma as hypothesis, one can type `generalize mylemma`; `intros`, or simply `lets: mylemma` (the latter requires *LibTactics.v*).

The second possibility is useful for lemmas that need to be exploited several times. The syntax for adding a lemma as a hint is `Hint Resolve mylemma`. For example, the lemma asserting that any number is less than or equal to itself,  $\forall x, x \leq x$ , called *Le.le\_refl* in the Coq standard library, can be added as a hint as follows.

```
Hint Resolve Le.le_refl.
```

A convenient shorthand for adding all the constructors of an inductive datatype as hints is the command `Hint Constructors mydatatype`.

Warning: some lemmas, such as transitivity results, should not be added as hints as they would very badly affect the performance of proof search. The description of this problem and the presentation of a general work-around for transitivity lemmas appear further on.

### 36.3.4 Integration of Automation in Tactics

The library "LibTactics" introduces a convenient feature for invoking automation after calling a tactic. In short, it suffices to add the symbol star ( $\times$ ) to the name of a tactic. For example, `apply $\times$  H` is equivalent to `apply H; auto_star`, where `auto_star` is a tactic that can be defined as needed.

The definition of `auto_star`, which determines the meaning of the star symbol, can be modified whenever needed. Simply write: `Ltac auto_star ::= a_new_definition`. Observe the use of `::=` instead of `:=`, which indicates that the tactic is being rebound to a new definition. So, the default definition is as follows.

```
Ltac auto_star ::= try solve [ jauto ].
```

Nearly all standard Coq tactics and all the tactics from "LibTactics" can be called with a star symbol. For example, one can invoke `subst $\times$` , `destruct $\times$  H`, `inverts $\times$  H`, `lets $\times$  l: H x`, `specializes $\times$  H x`, and so on... There are two notable exceptions. The tactic `auto $\times$`  is just another name for the tactic `auto_star`. And the tactic `apply $\times$  H` calls `eapply H` (or the more powerful `applys H` if needed), and then calls `auto_star`. Note that there is no `eapply $\times$  H` tactic, use `apply $\times$  H` instead.

In large developments, it can be convenient to use two degrees of automation. Typically, one would use a fast tactic, like `auto`, and a slower but more powerful tactic, like `jauto`. To allow for a smooth coexistence of the two form of automation, *LibTactics.v* also defines a "tilde" version of tactics, like `apply $\neg$  H`, `destruct $\neg$  H`, `subst $\neg$` , `auto $\neg$`  and so on. The meaning of the tilde symbol is described by the `auto_tilde` tactic, whose default implementation is `auto`.

```
Ltac auto_tilde ::= auto.
```

In the examples that follow, only `auto_star` is needed.

An alternative, possibly more efficient version of `auto_star` is the following":

```
Ltac auto_star ::= try solve eassumption | auto | jauto .
```

With the above definition, `auto_star` first tries to solve the goal using the assumptions; if it fails, it tries using `auto`, and if this still fails, then it calls `jauto`. Even though `jauto` is strictly stronger than `eassumption` and `auto`, it makes sense to call these tactics first, because, when they succeed, they save a lot of time, and when they fail to prove the goal, they fail very quickly."

## 36.4 Examples of Use of Automation

Let's see how to use proof search in practice on the main theorems of the "Software Foundations" course, proving in particular results such as determinism, preservation and progress.

### 36.4.1 Determinism

Module DETERMINISTICIMP.

Require Import Imp.

Recall the original proof of the determinism lemma for the IMP language, shown below.

Theorem ceval\_deterministic:  $\forall c \ st \ st1 \ st2$ ,

$c / st \ \backslash \ st1 \rightarrow$

$c / st \ \backslash \ st2 \rightarrow$

$st1 = st2$ .

Proof.

intros  $c \ st \ st1 \ st2 \ E1 \ E2$ .

generalize dependent  $st2$ .

(induction  $E1$ ); intros  $st2 \ E2$ ; inversion  $E2$ ; subst.

- reflexivity.

- reflexivity.

-

assert ( $st' = st'0$ ) as  $EQ1$ .

{ apply  $IHE1\_1$ ; assumption. }

subst  $st'0$ .

apply  $IHE1\_2$ . assumption.

-

apply  $IHE1$ . assumption.

-

rewrite  $H$  in  $H5$ . inversion  $H5$ .

-

rewrite  $H$  in  $H5$ . inversion  $H5$ .

-

apply  $IHE1$ . assumption.

-

reflexivity.

-

rewrite  $H$  in  $H2$ . inversion  $H2$ .

-

rewrite  $H$  in  $H4$ . inversion  $H4$ .

-

assert ( $st' = st'0$ ) as  $EQ1$ .

{ apply  $IHE1\_1$ ; assumption. }

```

      subst st'0.
      apply IHE1_2. assumption.
Qed.

```

Exercise: rewrite this proof using `auto` whenever possible. (The solution uses `auto` 9 times.)

```

Theorem ceval_deterministic':  $\forall c\ st\ st1\ st2,$ 
   $c / st \setminus\setminus st1 \rightarrow$ 
   $c / st \setminus\setminus st2 \rightarrow$ 
   $st1 = st2.$ 

```

Proof.

```

  admit.

```

Admitted.

In fact, using automation is not just a matter of calling `auto` in place of one or two other tactics. Using automation is about rethinking the organization of sequences of tactics so as to minimize the effort involved in writing and maintaining the proof. This process is eased by the use of the tactics from *LibTactics.v*. So, before trying to optimize the way automation is used, let's first rewrite the proof of determinism:

- use *introv* *H* instead of `intros x H`,
- use *gen* *x* instead of `generalize dependent x`,
- use *inverts* *H* instead of `inversion H`; `subst`,
- use *tryfalse* to handle contradictions, and get rid of the cases where `beval st b1 = true` and `beval st b1 = false` both appear in the context,
- stop using *ceval\_cases* to label subcases.

```

Theorem ceval_deterministic'':  $\forall c\ st\ st1\ st2,$ 
   $c / st \setminus\setminus st1 \rightarrow$ 
   $c / st \setminus\setminus st2 \rightarrow$ 
   $st1 = st2.$ 

```

Proof.

```

  introv E1 E2. gen st2.
  induction E1; intros; inverts E2; tryfalse.
- auto.
- auto.
- assert (st' = st'0). auto. subst. auto.
- auto.
- auto.
- auto.
- assert (st' = st'0). auto. subst. auto.

```

Qed.

To obtain a nice clean proof script, we have to remove the calls `assert (st' = st'0)`. Such a tactic invocation is not nice because it refers to some variables whose name has been automatically generated. This kind of tactics tend to be very brittle. The tactic `assert (st' = st'0)` is used to assert the conclusion that we want to derive from the induction hypothesis. So, rather than stating this conclusion explicitly, we are going to ask Coq to instantiate the induction hypothesis, using automation to figure out how to instantiate it. The tactic `forwards`, described in *LibTactics.v* precisely helps with instantiating a fact. So, let's see how it works out on our example.

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
  c / st \\\ st1 →
  c / st \\\ st2 →
  st1 = st2.
```

Proof.

```
  introv E1 E2. gen st2.
  induction E1; intros; inverts E2; tryfalse.
- auto.
- auto.
- dup 4.

+ assert (st' = st'0). apply IHE1_1. apply H1.
  skip.

+ forwards: IHE1_1. apply H1.
  skip.

+ forwards: IHE1_1. eauto.
  skip.

+ forwards*: IHE1_1.
  skip.
```

Abort.

To polish the proof script, it remains to factorize the calls to `auto`, using the star symbol. The proof of determinism can then be rewritten in only four lines, including no more than 10 tactics.

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
  c / st \\\ st1 →
  c / st \\\ st2 →
  st1 = st2.
```

Proof.

```
  introv E1 E2. gen st2.
  induction E1; intros; inverts× E2; tryfalse.
- forwards*: IHE1_1. subst×.
- forwards*: IHE1_1. subst×.
```

Qed.

End DETERMINISTICIMP.

### 36.4.2 Preservation for STLC

Module PRESERVATIONPROGRESSSTLC.

Require Import StlcProp.

Import *STLC*.

Import *STLCProp*.

Consider the proof of perservation of STLC, shown below. This proof already uses `eauto` through the triple-dot mechanism.

Theorem preservation :  $\forall t t' T,$

**has\_type** empty  $t T \rightarrow$

$t ==> t' \rightarrow$

**has\_type** empty  $t' T$ .

Proof with eauto.

*remember* (@empty **ty**) as *Gamma*.

intros  $t t' T HT$ . generalize dependent  $t'$ .

(induction  $HT$ ); intros  $t' HE$ ; subst *Gamma*.

-

inversion  $HE$ .

-

inversion  $HE$ .

-

inversion  $HE$ ; subst...

+

apply substitution\_preserves\_typing with  $T11...$

inversion  $HT1...$

-

inversion  $HE$ .

-

inversion  $HE$ .

-

inversion  $HE$ ; subst...

Qed.

Exercise: rewrite this proof using tactics from `LibTactics` and calling automation using the star symbol rather than the triple-dot notation. More precisely, make use of the tactics *inverts* $\times$  and *applies* $\times$  to call `auto` $\times$  after a call to *inverts* or to *applies*. The solution is three lines long.

Theorem preservation' :  $\forall t t' T,$

**has\_type** empty  $t T \rightarrow$



$t \Rightarrow t' \rightarrow$   
**has\_type** empty  $t'$   $T$ .

Proof.

*admit.*

*Admitted.*

### 36.4.3 Progress for STLC

Consider the proof of the progress theorem.

Theorem progress :  $\forall t T$ ,  
**has\_type** empty  $t T \rightarrow$   
**value**  $t \vee \exists t', t \Rightarrow t'$ .

Proof with eauto.

```

intros t T Ht.
remember (@empty ty) as Gamma.
(induction Ht); subst Gamma...
-
  inversion H.
-
  right. destruct IHHt1...
+
  destruct IHHt2...
  ×
    inversion H; subst; try solve by inversion.
    ∃ ([x0:=t2] t)...
  ×
    destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
+
  destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
-
  right. destruct IHHt1...
  destruct t1; try solve by inversion...
  inversion H. ∃ (tif x0 t2 t3)...
```

Qed.

Exercise: optimize the above proof. Hint: make use of `destruct×` and `inverts×`. The solution consists of 10 short lines.

Theorem progress' :  $\forall t T$ ,  
**has\_type** empty  $t T \rightarrow$   
**value**  $t \vee \exists t', t \Rightarrow t'$ .

Proof.

*admit.*

*Admitted.*

End PRESERVATIONPROGRESSSTLC.

### 36.4.4 BigStep and SmallStep

Module SEMANTICS.

Require Import Smallstep.

Consider the proof relating a small-step reduction judgment to a big-step reduction judgment.

Theorem multistep\_\_eval :  $\forall t v,$   
 $\text{normal\_form\_of } t v \rightarrow \exists n, v = \mathbb{C} n \wedge t \Downarrow n.$

Proof.

```

intros t v Hnorm.
unfold normal_form_of in Hnorm.
inversion Hnorm as [Hs Hnf]; clear Hnorm.
rewrite nf_same_as_value in Hnf. inversion Hnf. clear Hnf.
 $\exists n.$  split. reflexivity.
induction Hs; subst.
-
  apply E_Const.
-
  eapply step__eval. eassumption. apply IHHS. reflexivity.

```

Qed.

Our goal is to optimize the above proof. It is generally easier to isolate inductions into separate lemmas. So, we are going to first prove an intermediate result that consists of the judgment over which the induction is being performed.

Exercise: prove the following result, using tactics *introv*, *induction* and *subst*, and *apply*×. The solution is 3 lines long.

Theorem multistep\_eval\_ind :  $\forall t v,$   
 $t ==>^* v \rightarrow \forall n, \mathbb{C} n = v \rightarrow t \Downarrow n.$

Proof.

*admit.*

*Admitted.*

Exercise: using the lemma above, simplify the proof of the result *multistep\_\_eval*. You should use the tactics *introv*, *inverts*, *split*× and *apply*×. The solution is 2 lines long.

Theorem multistep\_\_eval' :  $\forall t v,$   
 $\text{normal\_form\_of } t v \rightarrow \exists n, v = \mathbb{C} n \wedge t \Downarrow n.$

Proof.

*admit.*

*Admitted.*

If we try to combine the two proofs into a single one, we will likely fail, because of a limitation of the *induction* tactic. Indeed, this tactic loses information when applied to a

property whose arguments are not reduced to variables, such as  $t \Rightarrow^* (C\ n)$ . You will thus need to use the more powerful tactic called **dependent induction**. This tactic is available only after importing the **Program** library, as shown below.

Require Import **Program**.

Exercise: prove the lemma **multistep\_\_eval** without invoking the lemma **multistep\_eval\_ind**, that is, by inlining the proof by induction involved in **multistep\_eval\_ind**, using the tactic **dependent induction** instead of **induction**. The solution is 5 lines long.

Theorem **multistep\_\_eval''** :  $\forall\ t\ v,$   
 $\text{normal\_form\_of}\ t\ v \rightarrow \exists\ n, v = C\ n \wedge t \setminus\setminus n.$

Proof.

*admit.*

*Admitted.*

End SEMANTICS.

### 36.4.5 Preservation for STLCRef

Module PRESERVATIONPROGRESSREFERENCES.

Require Import **Coq.omega.Omega**.

Require Import References.

Import *STLCRef*.

Hint Resolve store\_weakening extends\_refl.

The proof of preservation for STLCREF can be found in chapter References. The optimized proof script is more than twice shorter. The following material explains how to build the optimized proof script. The resulting optimized proof script for the preservation theorem appears afterwards.

Theorem **preservation** :  $\forall\ ST\ t\ t'\ T\ st\ st',$   
 $\text{has\_type empty}\ ST\ t\ T \rightarrow$   
 $\text{store\_well\_typed}\ ST\ st \rightarrow$   
 $t / st \Rightarrow t' / st' \rightarrow$   
 $\exists\ ST',$   
 $(\text{extends}\ ST'\ ST \wedge$   
 $\text{has\_type empty}\ ST'\ t'\ T \wedge$   
 $\text{store\_well\_typed}\ ST'\ st').$

Proof.

*remember (@empty ty) as Gamma. introv Ht. gen t'.*  
*(induction Ht); introv HST Hstep;*

*subst Gamma; inverts Hstep; eauto.*

-

$\exists ST. \text{inverts } Ht1. \text{ splits} \times. \text{ applys} \times \text{ substitution\_preserves\_typing.}$

-

*forwards*: *IHHt1*. *eauto*. *eauto*. *eauto*.  
*jauto\_set\_hyps*; *intros*.  
*jauto\_set\_goal*; *intros*.  
*eauto*. *eauto*. *eauto*.

-

*forwards\**: *IHHt2*.  
- *forwards\**: *IHHt*.  
- *forwards\**: *IHHt*.  
- *forwards\**: *IHHt1*.  
- *forwards\**: *IHHt2*.  
- *forwards\**: *IHHt1*.

-

+

$\exists (ST \text{ ++ } T1 : : \text{nil}). \text{inverts keep } HST. \text{ splits.}$   
*apply extends\_app*.  
*applys\_eq* *T\_Loc* 1.  
*rewrite app\_length*. *simpl*. *omega*.  
*unfold store\_Tlookup*. *rewrite*  $\leftarrow H$ . *rewrite*  $\times$  *app\_nth2*.  
*rewrite minus\_diag*. *simpl*. *reflexivity*.  
*apply*  $\times$  *store\_well\_typed\_app*.

- *forwards\**: *IHHt*.

-

+

$\exists ST. \text{ splits} \times.$   
*lets* [*- Hsty*]: *HST*.  
*applys\_eq*  $\times$  *Hsty* 1.  
*inverts*  $\times$  *Ht*.

- *forwards\**: *IHHt*.

-

+

$\exists ST. splits \times. applies \times assign\_pres\_store\_typing. inverts \times Ht1.$

- *forwards\**: *IHHt1*.

- *forwards\**: *IHHt2*.

**Qed.**

Let's come back to the proof case that was hard to optimize. The difficulty comes from the statement of `nth_eq_last`, which takes the form `nth (length l) (l ++ x::nil) d = x`. This lemma is hard to exploit because its first argument, `length l`, mentions a list `l` that has to be exactly the same as the `l` occurring in `snoc l x`. In practice, the first argument is often a natural number `n` that is provably equal to `length l` yet that is not syntactically equal to `length l`. There is a simple fix for making `nth_eq_last` easy to apply: introduce the intermediate variable `n` explicitly, so that the goal becomes `nth n (snoc l x) d = x`, with a premise asserting `n = length l`.

Lemma `nth_eq_last'` :  $\forall (A : \text{Type}) (l : \text{list } A) (x \ d : A) (n : \text{nat}),$   
 $n = \text{length } l \rightarrow \text{nth } n (l ++ x :: \text{nil}) \ d = x.$

**Proof.** `intros. subst. apply nth_eq_last. Qed.`

The proof case for *ref* from the preservation theorem then becomes much easier to prove, because `rewrite nth_eq_last'` now succeeds.

Lemma `preservation_ref` :  $\forall (st : \text{store}) (ST : \text{store\_ty}) \ T1,$   
 $\text{length } ST = \text{length } st \rightarrow$   
 $\text{TRef } T1 = \text{TRef } (\text{store\_Tlookup } (\text{length } st) (ST ++ T1 :: \text{nil})).$

**Proof.**

`intros. dup.`

`unfold store_Tlookup. rewrite  $\times$  nth_eq_last'.`

`fequal. symmetry. apply  $\times$  nth_eq_last'.`

**Qed.**

The optimized proof of preservation is summarized next.

Theorem `preservation'` :  $\forall ST \ t \ t' \ T \ st \ st',$

`has_type empty ST t T  $\rightarrow$`

`store_well_typed ST st  $\rightarrow$`

`t / st ==> t' / st'  $\rightarrow$`

$\exists ST',$

`(extends ST' ST  $\wedge$`

`has_type empty ST' t' T  $\wedge$`

`store_well_typed ST' st').`

**Proof.**

`remember (@empty ty) as Gamma. introv Ht. gen t'.`

`induction Ht; introv HST Hstep; subst Gamma; inverts Hstep; eauto.`

```

-  $\exists ST. \text{inverts } Ht1. \text{ splits} \times. \text{ applys} \times \text{ substitution\_preserves\_typing}.$ 
- forwards*: IHHt1.
- forwards*: IHHt2.
- forwards*: IHHt.
- forwards*: IHHt.
- forwards*: IHHt1.
- forwards*: IHHt2.
- forwards*: IHHt1.
-  $\exists (ST \text{ ++ } T1 :: \text{nil}). \text{inverts keep } HST. \text{ splits}.$ 
  apply extends_app.
  applys_eq T_Loc 1.
  rewrite app_length. simpl. omega.
  unfold store_Tlookup. rewrite  $\times$  nth_eq_last'.
  apply  $\times$  store_well_typed_app.
- forwards*: IHHt.
-  $\exists ST. \text{ splits} \times. \text{ lets } [_ \text{ Hsty}]: HST.$ 
  applys_eq  $\times$  Hsty 1. inverts  $\times$  Ht.
- forwards*: IHHt.
-  $\exists ST. \text{ splits} \times. \text{ applys} \times \text{ assign\_pres\_store\_typing}. \text{inverts} \times Ht1.$ 
- forwards*: IHHt1.
- forwards*: IHHt2.

```

Qed.

### 36.4.6 Progress for STLCRef

The proof of progress for STLCREF can be found in chapter References. The optimized proof script is, here again, about half the length.

Theorem progress :  $\forall ST \ t \ T \ st,$   
**has\_type** empty *ST t T*  $\rightarrow$   
 store\_well\_typed *ST st*  $\rightarrow$   
 (value *t*  $\vee \exists t', \exists st', t / st ==> t' / st'$ ).

Proof.

```

introv Ht HST. remember (@empty ty) as Gamma.
induction Ht; subst Gamma; tryfalse; try solve [left*].
- right. destruct  $\times$  IHHt1 as [K].
  inverts K; inverts Ht1.
  destruct  $\times$  IHHt2.
- right. destruct  $\times$  IHHt as [K].
  inverts K; try solve [inverts Ht]. eauto.
- right. destruct  $\times$  IHHt as [K].
  inverts K; try solve [inverts Ht]. eauto.
- right. destruct  $\times$  IHHt1 as [K].

```

```

    inverts K; try solve [inverts Ht1].
    destruct× IHHt2 as [M]].
    inverts M; try solve [inverts Ht2]. eauto.
- right. destruct× IHHt1 as [K]].
  inverts K; try solve [inverts Ht1]. destruct× n.
- right. destruct× IHHt.
- right. destruct× IHHt as [K]].
  inverts K; inverts Ht as M.
  inverts HST as N. rewrite× N in M.
- right. destruct× IHHt1 as [K]].
  destruct× IHHt2.
  inverts K; inverts Ht1 as M.
  inverts HST as N. rewrite× N in M.

```

Qed.

End PRESERVATIONPROGRESSREFERENCES.

### 36.4.7 Subtyping

Module SUBTYPINGINVERSION.

Require Import Sub.

Consider the inversion lemma for typing judgment of abstractions in a type system with subtyping.

```

Lemma abs_arrow : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
    subtype T1 S1
  ∧ has_type (update empty x S1) s2 T2.

```

Proof with eauto.

```

  intros x S1 s2 T1 T2 Hty.
  apply typing_inversion_abs in Hty.
  destruct Hty as [S2 [Hsub Hty]].
  apply sub_inversion_arrow in Hsub.
  destruct Hsub as [U1 [U2 [Heq [Hsub1 Hsub2]]]].
  inversion Heq; subst...

```

Qed.

Exercise: optimize the proof script, using *introv*, *lets* and *inverts×*. In particular, you will find it useful to replace the pattern `apply K in H. destruct H as l with lets l: K H`. The solution is 4 lines.

```

Lemma abs_arrow' : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
    subtype T1 S1
  ∧ has_type (update empty x S1) s2 T2.

```

Proof.

*admit.*

*Admitted.*

The lemma `substitution_preserves_typing` has already been used to illustrate the working of *lets* and *applies* in chapter `UseTactics`. Optimize further this proof using automation (with the star symbol), and using the tactic *cases\_if*'. The solution is 33 lines).

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,  
  has_type (update Gamma x U) t S →  
  has_type empty v U →  
  has_type Gamma ([x:=v] t) S.
```

Proof.

*admit.*

*Admitted.*

End SUBTYPINGINVERSION.

## 36.5 Advanced Topics in Proof Search

### 36.5.1 Stating Lemmas in the Right Way

Due to its depth-first strategy, `eauto` can get exponentially slower as the depth search increases, even when a short proof exists. In general, to make proof search run reasonably fast, one should avoid using a depth search greater than 5 or 6. Moreover, one should try to minimize the number of applicable lemmas, and usually put first the hypotheses whose proof usefully instantiates the existential variables.

In fact, the ability for `eauto` to solve certain goals actually depends on the order in which the hypotheses are stated. This point is illustrated through the following example, in which  $P$  is a property of natural numbers. This property is such that  $P\ n$  holds for any  $n$  as soon as  $P\ m$  holds for at least one  $m$  different from zero. The goal is to prove that  $P\ 2$  implies  $P\ 1$ . When the hypothesis about  $P$  is stated in the form  $\forall n\ m, P\ m \rightarrow m \neq 0 \rightarrow P\ n$ , then `eauto` works. However, with  $\forall n\ m, m \neq 0 \rightarrow P\ m \rightarrow P\ n$ , the tactic `eauto` fails.

```
Lemma order_matters_1 : ∀ (P : nat → Prop),  
  (∀ n m, P m → m ≠ 0 → P n) → P 2 → P 1.
```

Proof.

`eauto.` `Qed.`

```
Lemma order_matters_2 : ∀ (P : nat → Prop),  
  (∀ n m, m ≠ 0 → P m → P n) → P 5 → P 1.
```

Proof.

`eauto.`

`intros P H K.`

`eapply H.`



```
eauto.
Abort.
```

It is very important to understand that the hypothesis  $\forall n\ m, P\ m \rightarrow m \neq 0 \rightarrow P\ n$  is eauto-friendly, whereas  $\forall n\ m, m \neq 0 \rightarrow P\ m \rightarrow P\ n$  really isn't. Guessing a value of  $m$  for which  $P\ m$  holds and then checking that  $m \neq 0$  holds works well because there are few values of  $m$  for which  $P\ m$  holds. So, it is likely that `eauto` comes up with the right one. On the other hand, guessing a value of  $m$  for which  $m \neq 0$  and then checking that  $P\ m$  holds does not work well, because there are many values of  $m$  that satisfy  $m \neq 0$  but not  $P\ m$ .

### 36.5.2 Unfolding of Definitions During Proof-Search

The use of intermediate definitions is generally encouraged in a formal development as it usually leads to more concise and more readable statements. Yet, definitions can make it a little harder to automate proofs. The problem is that it is not obvious for a proof search mechanism to know when definitions need to be unfolded. Note that a naive strategy that consists in unfolding all definitions before calling proof search does not scale up to large proofs, so we avoid it. This section introduces a few techniques for avoiding to manually unfold definitions before calling proof search.

To illustrate the treatment of definitions, let  $P$  be an abstract property on natural numbers, and let `myFact` be a definition denoting the proposition  $P\ x$  holds for any  $x$  less than or equal to 3.

```
Axiom P : nat → Prop.
```

```
Definition myFact := ∀ x, x ≤ 3 → P x.
```

Proving that `myFact` under the assumption that  $P\ x$  holds for any  $x$  should be trivial. Yet, `auto` fails to prove it unless we unfold the definition of `myFact` explicitly.

```
Lemma demo_hint_unfold_goal_1 :
```

```
(∀ x, P x) → myFact.
```

```
Proof.
```

```
auto. unfold myFact. auto. Qed.
```

To automate the unfolding of definitions that appear as proof obligation, one can use the command `Hint Unfold myFact` to tell Coq that it should always try to unfold `myFact` when `myFact` appears in the goal.

```
Hint Unfold myFact.
```

This time, automation is able to see through the definition of `myFact`.

```
Lemma demo_hint_unfold_goal_2 :
```

```
(∀ x, P x) → myFact.
```

```
Proof. auto. Qed.
```

However, the `Hint Unfold` mechanism only works for unfolding definitions that appear in the goal. In general, proof search does not unfold definitions from the context. For example, assume we want to prove that  $P\ 3$  holds under the assumption that `True`  $\rightarrow$  `myFact`.

Lemma demo\_hint\_unfold\_context\_1 :

(**True**  $\rightarrow$  myFact)  $\rightarrow P$  3.

Proof.

intros.

auto. unfold myFact in \*. auto. Qed.

There is actually one exception to the previous rule: a constant occurring in an hypothesis is automatically unfolded if the hypothesis can be directly applied to the current goal. For example, auto can prove myFact  $\rightarrow P$  3, as illustrated below.

Lemma demo\_hint\_unfold\_context\_2 :

myFact  $\rightarrow P$  3.

Proof. auto. Qed.

### 36.5.3 Automation for Proving Absurd Goals

In this section, we'll see that lemmas concluding on a negation are generally not useful as hints, and that lemmas whose conclusion is **False** can be useful hints but having too many of them makes proof search inefficient. We'll also see a practical work-around to the efficiency issue.

Consider the following lemma, which asserts that a number less than or equal to 3 is not greater than 3.

Parameter le\_not\_gt :  $\forall x$ ,

$(x \leq 3) \rightarrow \neg (x > 3)$ .

Equivalently, one could state that a number greater than three is not less than or equal to 3.

Parameter gt\_not\_le :  $\forall x$ ,

$(x > 3) \rightarrow \neg (x \leq 3)$ .

In fact, both statements are equivalent to a third one stating that  $x \leq 3$  and  $x > 3$  are contradictory, in the sense that they imply **False**.

Parameter le\_gt\_false :  $\forall x$ ,

$(x \leq 3) \rightarrow (x > 3) \rightarrow \text{False}$ .

The following investigation aim at figuring out which of the three statements is the most convenient with respect to proof automation. The following material is enclosed inside a **Section**, so as to restrict the scope of the hints that we are adding. In other words, after the end of the section, the hints added within the section will no longer be active.

Section DemoAbsurd1.

Let's try to add the first lemma, *le\_not\_gt*, as hint, and see whether we can prove that the proposition  $\exists x, x \leq 3 \wedge x > 3$  is absurd.

Hint Resolve le\_not\_gt.

Lemma demo\_auto\_absurd\_1 :

$(\exists x, x \leq 3 \wedge x > 3) \rightarrow \text{False}.$

Proof.

intros. *jauto\_set*. eauto. eapply *le\_not\_gt*. eauto. eauto.

Qed.

The lemma *gt\_not\_le* is symmetric to *le\_not\_gt*, so it will not be any better. The third lemma, *le\_gt\_false*, is a more useful hint, because it concludes on **False**, so proof search will try to apply it when the current goal is **False**.

Hint Resolve *le\_gt\_false*.

Lemma demo\_auto\_absurd\_2 :

$(\exists x, x \leq 3 \wedge x > 3) \rightarrow \text{False}.$

Proof.

*dup*.

intros. *jauto\_set*. eauto.

*jauto*.

Qed.

In summary, a lemma of the form  $H1 \rightarrow H2 \rightarrow \text{False}$  is a much more effective hint than  $H1 \rightarrow \neg H2$ , even though the two statements are equivalent up to the definition of the negation symbol  $\neg$ .

That said, one should be careful with adding lemmas whose conclusion is **False** as hint. The reason is that whenever reaching the goal **False**, the proof search mechanism will potentially try to apply all the hints whose conclusion is **False** before applying the appropriate one.

End DemoAbsurd1.

Adding lemmas whose conclusion is **False** as hint can be, locally, a very effective solution. However, this approach does not scale up for global hints. For most practical applications, it is reasonable to give the name of the lemmas to be exploited for deriving a contradiction. The tactic *false H*, provided by *LibTactics* serves that purpose: *false H* replaces the goal with **False** and calls *eapply H*. Its behavior is described next. Observe that any of the three statements *le\_not\_gt*, *gt\_not\_le* or *le\_gt\_false* can be used.

Lemma demo\_false :  $\forall x,$

$(x \leq 3) \rightarrow (x > 3) \rightarrow 4 = 5.$

Proof.

intros. *dup* 4.

- *false*. *eapply le\_gt\_false*.

+ *auto*. + *skip*.

- *false*. *eapply le\_gt\_false*.

+ *eauto*. + *eauto*.

- *false le\_gt\_false*. *eauto*. *eauto*.

- *false le\_not\_gt*. *eauto*. *eauto*.

Qed.

In the above example, `false le_gt_false; eauto` proves the goal, but `false le_gt_false; auto` does not, because `auto` does not correctly instantiate the existential variable. Note that `false × le_gt_false` would not work either, because the star symbol tries to call `auto` first. So, there are two possibilities for completing the proof: either call `false le_gt_false; eauto`, or call `false × (le_gt_false 3)`.

### 36.5.4 Automation for Transitivity Lemmas

Some lemmas should never be added as hints, because they would very badly slow down proof search. The typical example is that of transitivity results. This section describes the problem and presents a general workaround.

Consider a subtyping relation, written *subtype* *S T*, that relates two object *S* and *T* of type *typ*. Assume that this relation has been proved reflexive and transitive. The corresponding lemmas are named *subtype\_refl* and *subtype\_trans*.

Parameter *typ* : Type.

Parameter *subtype* : *typ* → *typ* → Prop.

Parameter *subtype\_refl* : ∀ *T*,  
  *subtype T T*.

Parameter *subtype\_trans* : ∀ *S T U*,  
  *subtype S T* → *subtype T U* → *subtype S U*.

Adding reflexivity as hint is generally a good idea, so let's add reflexivity of subtyping as hint.

Hint Resolve *subtype\_refl*.

Adding transitivity as hint is generally a bad idea. To understand why, let's add it as hint and see what happens. Because we cannot remove hints once we've added them, we are going to open a "Section," so as to restrict the scope of the transitivity hint to that section.

Section HintsTransitivity.

Hint Resolve *subtype\_trans*.

Now, consider the goal  $\forall S T, \text{subtype } S T$ , which clearly has no hope of being solved. Let's call `eauto` on this goal.

Lemma *transitivity\_bad\_hint\_1* : ∀ *S T*,  
  *subtype S T*.

Proof.

  intros. eauto. Abort.

Note that after closing the section, the hint *subtype\_trans* is no longer active.

End HintsTransitivity.

In the previous example, the proof search has spent a lot of time trying to apply transitivity and reflexivity in every possible way. Its process can be summarized as follows. The first goal is *subtype*  $S\ T$ . Since reflexivity does not apply, *eauto* invokes transitivity, which produces two subgoals, *subtype*  $S\ ?X$  and *subtype*  $?X\ T$ . Solving the first subgoal, *subtype*  $S\ ?X$ , is straightforward, it suffices to apply reflexivity. This unifies  $?X$  with  $S$ . So, the second subgoal, *subtype*  $?X\ T$ , becomes *subtype*  $S\ T$ , which is exactly what we started from...

The problem with the transitivity lemma is that it is applicable to any goal concluding on a subtyping relation. Because of this, *eauto* keeps trying to apply it even though it most often doesn't help to solve the goal. So, one should never add a transitivity lemma as a hint for proof search.

There is a general workaround for having automation to exploit transitivity lemmas without giving up on efficiency. This workaround relies on a powerful mechanism called "external hint." This mechanism allows to manually describe the condition under which a particular lemma should be tried out during proof search.

For the case of transitivity of subtyping, we are going to tell Coq to try and apply the transitivity lemma on a goal of the form *subtype*  $S\ U$  only when the proof context already contains an assumption either of the form *subtype*  $S\ T$  or of the form *subtype*  $T\ U$ . In other words, we only apply the transitivity lemma when there is some evidence that this application might help. To set up this "external hint," one has to write the following.

```
Hint Extern 1 (subtype ?S ?U) =>
  match goal with
  | H: subtype S ?T ⊢ _ => apply (@subtype_trans S T U)
  | H: subtype ?T U ⊢ _ => apply (@subtype_trans S T U)
end.
```

This hint declaration can be understood as follows.

- "Hint Extern" introduces the hint.
- The number "1" corresponds to a priority for proof search. It doesn't matter so much what priority is used in practice.
- The pattern *subtype*  $?S\ ?U$  describes the kind of goal on which the pattern should apply. The question marks are used to indicate that the variables  $?S$  and  $?U$  should be bound to some value in the rest of the hint description.
- The construction *match goal with* ... *end* tries to recognize patterns in the goal, or in the proof context, or both.
- The first pattern is  $H: \textit{subtype}\ S\ ?T \vdash \_$ . It indicates that the context should contain an hypothesis  $H$  of type *subtype*  $S\ ?T$ , where  $S$  has to be the same as in the goal, and where  $?T$  can have any value.
- The symbol  $\vdash \_$  at the end of  $H: \textit{subtype}\ S\ ?T \vdash \_$  indicates that we do not impose further condition on how the proof obligation has to look like.

- The branch `⇒ apply (@subtype.trans S T U)` that follows indicates that if the goal has the form `subtype S U` and if there exists an hypothesis of the form `subtype S T`, then we should try and apply transitivity lemma instantiated on the arguments `S`, `T` and `U`. (Note: the symbol `@` in front of `subtype.trans` is only actually needed when the “Implicit Arguments” feature is activated.)
- The other branch, which corresponds to an hypothesis of the form `H: subtype ?T U` is symmetrical.

Note: the same external hint can be reused for any other transitive relation, simply by renaming `subtype` into the name of that relation.

Let us see an example illustrating how the hint works.

```
Lemma transitivity_workaround_1 : ∀ T1 T2 T3 T4,
  subtype T1 T2 → subtype T2 T3 → subtype T3 T4 → subtype T1 T4.
```

Proof.

```
intros. eauto. Qed.
```

We may also check that the new external hint does not suffer from the complexity blow up.

```
Lemma transitivity_workaround_2 : ∀ S T,
  subtype S T.
```

Proof.

```
intros. eauto. Abort.
```

## 36.6 Decision Procedures

A decision procedure is able to solve proof obligations whose statement admits a particular form. This section describes three useful decision procedures. The tactic `omega` handles goals involving arithmetic and inequalities, but not general multiplications. The tactic `ring` handles goals involving arithmetic, including multiplications, but does not support inequalities. The tactic `congruence` is able to prove equalities and inequalities by exploiting equalities available in the proof context.

### 36.6.1 Omega

The tactic `omega` supports natural numbers (type `nat`) as well as integers (type `Z`, available by including the module `ZArith`). It supports addition, subtraction, equalities and inequalities. Before using `omega`, one needs to import the module `Omega`, as follows.

```
Require Import Omega.
```

Here is an example. Let `x` and `y` be two natural numbers (they cannot be negative). Assume `y` is less than 4, assume `x+x+1` is less than `y`, and assume `x` is not zero. Then, it must be the case that `x` is equal to one.

```

Lemma omega_demo_1 : ∀ (x y : nat),
  (y ≤ 4) → (x + x + 1 ≤ y) → (x ≠ 0) → (x = 1).
Proof. intros. omega. Qed.

```

Another example: if  $z$  is the mean of  $x$  and  $y$ , and if the difference between  $x$  and  $y$  is at most 4, then the difference between  $x$  and  $z$  is at most 2.

```

Lemma omega_demo_2 : ∀ (x y z : nat),
  (x + y = z + z) → (x - y ≤ 4) → (x - z ≤ 2).
Proof. intros. omega. Qed.

```

One can prove **False** using **omega** if the mathematical facts from the context are contradictory. In the following example, the constraints on the values  $x$  and  $y$  cannot be all satisfied in the same time.

```

Lemma omega_demo_3 : ∀ (x y : nat),
  (x + 5 ≤ y) → (y - x < 3) → False.
Proof. intros. omega. Qed.

```

Note: **omega** can prove a goal by contradiction only if its conclusion is reduced **False**. The tactic **omega** always fails when the conclusion is an arbitrary proposition  $P$ , even though **False** implies any proposition  $P$  (by `ex_falso_quodlibet`).

```

Lemma omega_demo_4 : ∀ (x y : nat) (P : Prop),
  (x + 5 ≤ y) → (y - x < 3) → P.
Proof.
  intros.
  false. omega.
Qed.

```

## 36.6.2 Ring

Compared with **omega**, the tactic **ring** adds support for multiplications, however it gives up the ability to reason on inequations. Moreover, it supports only integers (type  $\mathbb{Z}$ ) and not natural numbers (type  $\mathbb{N}$ ). Here is an example showing how to use **ring**.

```

Module RINGDEMO.
  Require Import ZArith.
  Open Scope Z_scope.

  Lemma ring_demo : ∀ (x y z : Z),
    x × (y + z) - z × 3 × x
    = x × y - 2 × x × z.
Proof. intros. ring. Qed.
End RINGDEMO.

```

### 36.6.3 Congruence

The tactic `congruence` is able to exploit equalities from the proof context in order to automatically perform the rewriting operations necessary to establish a goal. It is slightly more powerful than the tactic `subst`, which can only handle equalities of the form  $x = e$  where  $x$  is a variable and  $e$  an expression.

Lemma `congruence_demo_1` :

$$\begin{aligned} & \forall (f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) (g h : \text{nat} \rightarrow \text{nat}) (x y z : \text{nat}), \\ & f (g x) (g y) = z \rightarrow \\ & 2 = g x \rightarrow \\ & g y = h z \rightarrow \\ & f 2 (h z) = z. \end{aligned}$$

Proof. `intros. congruence. Qed.`

Moreover, `congruence` is able to exploit universally quantified equalities, for example  $\forall a, g a = h a$ .

Lemma `congruence_demo_2` :

$$\begin{aligned} & \forall (f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) (g h : \text{nat} \rightarrow \text{nat}) (x y z : \text{nat}), \\ & (\forall a, g a = h a) \rightarrow \\ & f (g x) (g y) = z \rightarrow \\ & g x = 2 \rightarrow \\ & f 2 (h y) = z. \end{aligned}$$

Proof. `congruence. Qed.`

Next is an example where `congruence` is very useful.

Lemma `congruence_demo_4` :  $\forall (f g : \text{nat} \rightarrow \text{nat})$ ,

$$\begin{aligned} & (\forall a, f a = g a) \rightarrow \\ & f (g (g 2)) = g (f (f 2)). \end{aligned}$$

Proof. `congruence. Qed.`

The tactic `congruence` is able to prove a contradiction if the goal entails an equality that contradicts an inequality available in the proof context.

Lemma `congruence_demo_3` :

$$\begin{aligned} & \forall (f g h : \text{nat} \rightarrow \text{nat}) (x : \text{nat}), \\ & (\forall a, f a = h a) \rightarrow \\ & g x = f x \rightarrow \\ & g x \neq h x \rightarrow \\ & \text{False}. \end{aligned}$$

Proof. `congruence. Qed.`

One of the strengths of `congruence` is that it is a very fast tactic. So, one should not hesitate to invoke it wherever it might help.



## 36.7 Summary

Let us summarize the main automation tactics available.

- `auto` automatically applies `reflexivity`, `assumption`, and `apply`.
- `eauto` moreover tries `eapply`, and in particular can instantiate existentials in the conclusion.
- `iauto` extends `eauto` with support for negation, conjunctions, and disjunctions. However, its support for disjunction can make it exponentially slow.
- `jauto` extends `eauto` with support for negation, conjunctions, and existential at the head of hypothesis.
- `congruence` helps reasoning about equalities and inequalities.
- `omega` proves arithmetic goals with equalities and inequalities, but it does not support multiplication.
- `ring` proves arithmetic goals with multiplications, but does not support inequalities.

In order to set up automation appropriately, keep in mind the following rule of thumbs:

- automation is all about balance: not enough automation makes proofs not very robust on change, whereas too much automation makes proofs very hard to fix when they break.
- if a lemma is not goal directed (i.e., some of its variables do not occur in its conclusion), then the premises need to be ordered in such a way that proving the first premises maximizes the chances of correctly instantiating the variables that do not occur in the conclusion.
- a lemma whose conclusion is **False** should only be added as a local hint, i.e., as a hint within the current section.
- a transitivity lemma should never be considered as hint; if automation of transitivity reasoning is really necessary, an **Extern Hint** needs to be set up.
- a definition usually needs to be accompanied with a **Hint Unfold**.

Becoming a master in the black art of automation certainly requires some investment, however this investment will pay off very quickly.

*Date* : 2016 – 02 – 17 17 : 39 : 13 – 0500(*Wed, 17Feb2016*)

# Chapter 37

## Library PE

### 37.1 PE: Partial Evaluation

The *Equiv* chapter introduced constant folding as an example of a program transformation and proved that it preserves the meaning of programs. Constant folding operates on manifest constants such as `ANum` expressions. For example, it simplifies the command `Y ::= APlus (ANum 3) (ANum 1)` to the command `Y ::= ANum 4`. However, it does not propagate known constants along data flow. For example, it does not simplify the sequence `X ::= ANum 3;; Y ::= APlus (AId X) (ANum 1)` to `X ::= ANum 3;; Y ::= ANum 4` because it forgets that `X` is 3 by the time it gets to `Y`.

We might naturally want to enhance constant folding so that it propagates known constants and uses them to simplify programs. Doing so constitutes a rudimentary form of *partial evaluation*. As we will see, partial evaluation is so called because it is like running a program, except only part of the program can be evaluated because only part of the input to the program is known. For example, we can only simplify the program `X ::= ANum 3;; Y ::= AMinus (APlus (AId X) (ANum 1)) (AId Y)` to `X ::= ANum 3;; Y ::= AMinus (ANum 4) (AId Y)` without knowing the initial value of `Y`.

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Logic.FunctionalExtensionality.
Require Import Coq.Lists.List.
Import ListNotations.

Require Import Sflib.
Require Import Maps.
Require Import Imp.
```

## 37.2 Generalizing Constant Folding

The starting point of partial evaluation is to represent our partial knowledge about the state. For example, between the two assignments above, the partial evaluator may know only that `X` is 3 and nothing about any other variable.

### 37.2.1 Partial States

Conceptually speaking, we can think of such partial states as the type `id → option nat` (as opposed to the type `id → nat` of concrete, full states). However, in addition to looking up and updating the values of individual variables in a partial state, we may also want to compare two partial states to see if and where they differ, to handle conditional control flow. It is not possible to compare two arbitrary functions in this way, so we represent partial states in a more concrete format: as a list of `id × nat` pairs.

Definition `pe_state := list (id × nat)`.

The idea is that a variable `id` appears in the list if and only if we know its current `nat` value. The `pe_lookup` function thus interprets this concrete representation. (If the same variable `id` appears multiple times in the list, the first occurrence wins, but we will define our partial evaluator to never construct such a `pe_state`.)

```
Fixpoint pe_lookup (pe_st : pe_state) (V : id) : option nat :=
  match pe_st with
  | [] ⇒ None
  | (V', n') :: pe_st ⇒ if beq_id V V' then Some n'
                        else pe_lookup pe_st V
  end.
```

For example, `empty_pe_state` represents complete ignorance about every variable – the function that maps every `id` to `None`.

Definition `empty_pe_state : pe_state := []`.

More generally, if the `list` representing a `pe_state` does not contain some `id`, then that `pe_state` must map that `id` to `None`. Before we prove this fact, we first define a useful tactic for reasoning with `id` equality. The tactic `compare V V'` means to reason by cases over `beq_id V V'`. In the case where `V = V'`, the tactic substitutes `V` for `V'` throughout.

```
Tactic Notation "compare" ident(i) ident(j) :=
  let H := fresh "Heq" i j in
  destruct (beq_idP i j);
  [ subst j | ].
```

Theorem `pe_domain`:  $\forall pe\_st\ V\ n,$   
`pe_lookup pe_st V = Some n →`  
`in V (map (@fst _ _) pe_st).`

Proof. `intros pe_st V n H. induction pe_st as [| [V' n'] pe_st].`

```
- inversion H.
- simpl in H. simpl. compare V V'; auto. Qed.
```

In what follows, we will make heavy use of the `ln` property from the standard library, also defined in *Logic.v*:

Print *In*.

Besides the various lemmas about `ln` that we've already come across, the following one (taken from the standard library) will also be useful:

Check `filter_ln`.

If a type `A` has an operator `beq` for testing equality of its elements, we can compute a boolean `inb beq a l` for testing whether `ln a l` holds or not.

```
Fixpoint inb {A : Type} (beq : A → A → bool) (a : A) (l : list A) :=
  match l with
  | [] ⇒ false
  | a'::l' ⇒ beq a a' || inb beq a l'
  end.
```

It is easy to relate `inb` to `ln` with the `reflect` property:

```
Lemma inbP : ∀ A : Type, ∀ beq : A → A → bool,
  (∀ a1 a2, reflect (a1 = a2) (beq a1 a2)) →
  ∀ a l, reflect (ln a l) (inb beq a l).
```

Proof.

```
intros A beq beqP a l.
induction l as [|a' l' IH].
- constructor. intros [].
- simpl. destruct (beqP a a').
  + subst. constructor. left. reflexivity.
  + simpl. destruct IH; constructor.
    × right. trivial.
    × intros [H1 | H2]; congruence.
```

Qed.

## 37.2.2 Arithmetic Expressions

Partial evaluation of `aexp` is straightforward – it is basically the same as constant folding, `fold_constants_aexp`, except that sometimes the partial state tells us the current value of a variable and we can replace it by a constant expression.

```
Fixpoint pe_aexp (pe_st : pe_state) (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | Ald i ⇒ match pe_lookup pe_st i with
    | Some n ⇒ ANum n
```

```

      | None  $\Rightarrow$  Ald  $i$ 
    end
  | APlus  $a1\ a2 \Rightarrow$ 
    match (pe_aexp pe_st  $a1$ , pe_aexp pe_st  $a2$ ) with
    | (ANum  $n1$ , ANum  $n2$ )  $\Rightarrow$  ANum ( $n1 + n2$ )
    | ( $a1'$ ,  $a2'$ )  $\Rightarrow$  APlus  $a1'\ a2'$ 
    end
  | AMinus  $a1\ a2 \Rightarrow$ 
    match (pe_aexp pe_st  $a1$ , pe_aexp pe_st  $a2$ ) with
    | (ANum  $n1$ , ANum  $n2$ )  $\Rightarrow$  ANum ( $n1 - n2$ )
    | ( $a1'$ ,  $a2'$ )  $\Rightarrow$  AMinus  $a1'\ a2'$ 
    end
  | AMult  $a1\ a2 \Rightarrow$ 
    match (pe_aexp pe_st  $a1$ , pe_aexp pe_st  $a2$ ) with
    | (ANum  $n1$ , ANum  $n2$ )  $\Rightarrow$  ANum ( $n1 \times n2$ )
    | ( $a1'$ ,  $a2'$ )  $\Rightarrow$  AMult  $a1'\ a2'$ 
    end
end.

```

This partial evaluator folds constants but does not apply the associativity of addition.

Example test\_pe\_aexp1:

```

pe_aexp [(X,3)] (APlus (APlus (Ald X) (ANum 1)) (Ald Y))
= APlus (ANum 4) (Ald Y).

```

Proof. reflexivity. Qed.

Example test\_pe\_aexp2:

```

pe_aexp [(Y,3)] (APlus (APlus (Ald X) (ANum 1)) (Ald Y))
= APlus (APlus (Ald X) (ANum 1)) (ANum 3).

```

Proof. reflexivity. Qed.

Now, in what sense is `pe_aexp` correct? It is reasonable to define the correctness of `pe_aexp` as follows: whenever a full state  $st:state$  is *consistent* with a partial state  $pe\_st:pe\_state$  (in other words, every variable to which  $pe\_st$  assigns a value is assigned the same value by  $st$ ), evaluating  $a$  and evaluating `pe_aexp pe_st a` in  $st$  yields the same result. This statement is indeed true.

Definition pe\_consistent ( $st:state$ ) ( $pe\_st:pe\_state$ ) :=

```

 $\forall V\ n,$  Some  $n = pe\_lookup\ pe\_st\ V \rightarrow st\ V = n.$ 

```

Theorem pe\_aexp\_correct\_weak:  $\forall\ st\ pe\_st,$  pe\_consistent  $st\ pe\_st \rightarrow$

```

 $\forall\ a,$  aeval  $st\ a = aeval\ st\ (pe\_aexp\ pe\_st\ a).$ 

```

Proof. unfold pe\_consistent. intros  $st\ pe\_st\ H\ a.$

```

induction a; simpl;
try reflexivity;
try (destruct (pe_aexp pe_st a1);
    destruct (pe_aexp pe_st a2);

```

```

      rewrite IHa1; rewrite IHa2; reflexivity).
-
  remember (pe_lookup pe_st i) as l. destruct l.
+ rewrite H with (n:=n) by apply Heql. reflexivity.
+ reflexivity.
Qed.

```

However, we will soon want our partial evaluator to remove assignments. For example, it will simplify  $X ::= \text{ANum } 3;; Y ::= \text{AMinus (AId } X) (\text{AId } Y);; X ::= \text{ANum } 4$  to just  $Y ::= \text{AMinus (ANum } 3) (\text{AId } Y);; X ::= \text{ANum } 4$  by delaying the assignment to  $X$  until the end. To accomplish this simplification, we need the result of partial evaluating  $\text{pe\_aexp } (X, 3)$  ( $\text{AMinus (AId } X) (\text{AId } Y)$ ) to be equal to  $\text{AMinus (ANum } 3) (\text{AId } Y)$  and *not* the original expression  $\text{AMinus (AId } X) (\text{AId } Y)$ . After all, it would be incorrect, not just inefficient, to transform  $X ::= \text{ANum } 3;; Y ::= \text{AMinus (AId } X) (\text{AId } Y);; X ::= \text{ANum } 4$  to  $Y ::= \text{AMinus (AId } X) (\text{AId } Y);; X ::= \text{ANum } 4$  even though the output expressions  $\text{AMinus (ANum } 3) (\text{AId } Y)$  and  $\text{AMinus (AId } X) (\text{AId } Y)$  both satisfy the correctness criterion that we just proved. Indeed, if we were to just define  $\text{pe\_aexp } pe\_st \ a = a$  then the theorem *pe\_aexp\_correct*' would already trivially hold.

Instead, we want to prove that the  $\text{pe\_aexp}$  is correct in a stronger sense: evaluating the expression produced by partial evaluation ( $\text{aeval } st \ (\text{pe\_aexp } pe\_st \ a)$ ) must not depend on those parts of the full state  $st$  that are already specified in the partial state  $pe\_st$ . To be more precise, let us define a function *pe\_override*, which updates  $st$  with the contents of  $pe\_st$ . In other words, *pe\_override* carries out the assignments listed in  $pe\_st$  on top of  $st$ .

```

Fixpoint pe_update (st:state) (pe_st:pe_state) : state :=
  match pe_st with
  | [] => st
  | (V,n)::pe_st => t_update (pe_update st pe_st) V n
  end.

```

Example test\_pe\_update:

```

pe_update (t_update empty_state Y 1) [(X,3);(Z,2)]
= t_update (t_update (t_update empty_state Y 1) Z 2) X 3.

```

Proof. reflexivity. Qed.

Although  $\text{pe\_update}$  operates on a concrete **list** representing a  $\text{pe\_state}$ , its behavior is defined entirely by the  $\text{pe\_lookup}$  interpretation of the  $\text{pe\_state}$ .

Theorem *pe\_update\_correct*:  $\forall st \ pe\_st \ V0,$

```

  pe_update st pe_st V0 =
  match pe_lookup pe_st V0 with
  | Some n => n
  | None => st V0
  end.

```

Proof. intros. induction  $pe\_st$  as [| [V n] pe\_st]. reflexivity.  
 simpl in \*. unfold t\_update.

`compare V0 V; auto. rewrite ← beq_id_refl; auto. rewrite false_beq_id; auto. Qed.`

We can relate `pe_consistent` to `pe_update` in two ways. First, overriding a state with a partial state always gives a state that is consistent with the partial state. Second, if a state is already consistent with a partial state, then overriding the state with the partial state gives the same state.

Theorem `pe_update_consistent`:  $\forall st\ pe\_st,$   
 $pe\_consistent\ (pe\_update\ st\ pe\_st)\ pe\_st.$

Proof. `intros st pe_st V n H. rewrite pe_update_correct.`  
`destruct (pe_lookup pe_st V); inversion H. reflexivity. Qed.`

Theorem `pe_consistent_update`:  $\forall st\ pe\_st,$   
 $pe\_consistent\ st\ pe\_st \rightarrow \forall V, st\ V = pe\_update\ st\ pe\_st\ V.$

Proof. `intros st pe_st H V. rewrite pe_update_correct.`  
`remember (pe_lookup pe_st V) as l. destruct l; auto. Qed.`

Now we can state and prove that `pe_aexp` is correct in the stronger sense that will help us define the rest of the partial evaluator.

Intuitively, running a program using partial evaluation is a two-stage process. In the first, *static* stage, we partially evaluate the given program with respect to some partial state to get a *residual* program. In the second, *dynamic* stage, we evaluate the residual program with respect to the rest of the state. This dynamic state provides values for those variables that are unknown in the static (partial) state. Thus, the residual program should be equivalent to *prepending* the assignments listed in the partial state to the original program.

Theorem `pe_aexp_correct`:  $\forall (pe\_st:pe\_state)\ (a:aexp)\ (st:state),$   
 $aeval\ (pe\_update\ st\ pe\_st)\ a = aeval\ st\ (pe\_aexp\ pe\_st\ a).$

Proof.  
`intros pe_st a st.`  
`induction a; simpl;`  
`try reflexivity;`  
`try (destruct (pe_aexp pe_st a1);`  
`destruct (pe_aexp pe_st a2);`  
`rewrite IHa1; rewrite IHa2; reflexivity).`  
`rewrite pe_update_correct. destruct (pe_lookup pe_st i); reflexivity.`  
`Qed.`

### 37.2.3 Boolean Expressions

The partial evaluation of boolean expressions is similar. In fact, it is entirely analogous to the constant folding of boolean expressions, because our language has no boolean variables.

Fixpoint `pe_bexp` (`pe_st : pe_state`) (`b : bexp`) : `bexp` :=  
`match b with`  
`| BTrue ⇒ BTrue`  
`| BFalse ⇒ BFalse`

```

| BEq a1 a2 ⇒
  match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
  | (ANum n1, ANum n2) ⇒ if beq_nat n1 n2 then BTrue else BFalse
  | (a1', a2') ⇒ BEq a1' a2'
  end
| BLe a1 a2 ⇒
  match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
  | (ANum n1, ANum n2) ⇒ if leb n1 n2 then BTrue else BFalse
  | (a1', a2') ⇒ BLe a1' a2'
  end
| BNot b1 ⇒
  match (pe_bexp pe_st b1) with
  | BTrue ⇒ BFalse
  | BFalse ⇒ BTrue
  | b1' ⇒ BNot b1'
  end
| BAnd b1 b2 ⇒
  match (pe_bexp pe_st b1, pe_bexp pe_st b2) with
  | (BTrue, BTrue) ⇒ BTrue
  | (BTrue, BFalse) ⇒ BFalse
  | (BFalse, BTrue) ⇒ BFalse
  | (BFalse, BFalse) ⇒ BFalse
  | (b1', b2') ⇒ BAnd b1' b2'
  end
end.

```

Example test\_pe\_bexp1:

```

pe_bexp [(X,3)] (BNot (BLe (Ald X) (ANum 3)))
= BFalse.

```

Proof. reflexivity. Qed.

Example test\_pe\_bexp2:  $\forall b$ ,

```

b = BNot (BLe (Ald X) (APlus (Ald X) (ANum 1))) →
pe_bexp [] b = b.

```

Proof. intros b H. rewrite → H. reflexivity. Qed.

The correctness of `pe_bexp` is analogous to the correctness of `pe_aexp` above.

Theorem `pe_bexp_correct`:  $\forall (pe\_st:pe\_state) (b:bexp) (st:state)$ ,  
`beval (pe_update st pe_st) b = beval st (pe_bexp pe_st b)`.

Proof.

```

intros pe_st b st.
induction b; simpl;
try reflexivity;
try (remember (pe_aexp pe_st a) as a');

```



```

remember (pe_aexp pe_st a0) as a0';
assert (Ha: aeval (pe_update st pe_st) a = aeval st a');
assert (Ha0: aeval (pe_update st pe_st) a0 = aeval st a0');
  try (subst; apply pe_aexp_correct);
destruct a'; destruct a0'; rewrite Ha; rewrite Ha0;
simpl; try destruct (beq_nat n n0); try destruct (leb n n0);
reflexivity);
try (destruct (pe_bexp pe_st b); rewrite IHb; reflexivity);
try (destruct (pe_bexp pe_st b1);
  destruct (pe_bexp pe_st b2);
  rewrite IHb1; rewrite IHb2; reflexivity).

```

Qed.

### 37.3 Partial Evaluation of Commands, Without Loops

What about the partial evaluation of commands? The analogy between partial evaluation and full evaluation continues: Just as full evaluation of a command turns an initial state into a final state, partial evaluation of a command turns an initial partial state into a final partial state. The difference is that, because the state is partial, some parts of the command may not be executable at the static stage. Therefore, just as `pe_aexp` returns a residual **aexp** and `pe_bexp` returns a residual **bexp** above, partially evaluating a command yields a residual command.

Another way in which our partial evaluator is similar to a full evaluator is that it does not terminate on all commands. It is not hard to build a partial evaluator that terminates on all commands; what is hard is building a partial evaluator that terminates on all commands yet automatically performs desired optimizations such as unrolling loops. Often a partial evaluator can be coaxed into terminating more often and performing more optimizations by writing the source program differently so that the separation between static and dynamic information becomes more apparent. Such coaxing is the art of *binding-time improvement*. The binding time of a variable tells when its value is known – either “static“, or “dynamic.“

Anyway, for now we will just live with the fact that our partial evaluator is not a total function from the source command and the initial partial state to the residual command and the final partial state. To model this non-termination, just as with the full evaluation of commands, we use an inductively defined relation. We write  $c1 / st \searrow c1' / st'$  to mean that partially evaluating the source command  $c1$  in the initial partial state  $st$  yields the residual command  $c1'$  and the final partial state  $st'$ . For example, we want something like  $(X ::= \text{ANum } 3 ;; Y ::= \text{AMult } (\text{AId } Z) (\text{APlus } (\text{AId } X) (\text{AId } X))) / \square \searrow (Y ::= \text{AMult } (\text{AId } Z) (\text{ANum } 6)) / (X, 3)$  to hold. The assignment to  $X$  appears in the final partial state, not the residual command.

### 37.3.1 Assignment

Let's start by considering how to partially evaluate an assignment. The two assignments in the source program above needs to be treated differently. The first assignment  $X ::= \text{ANum } 3$ , is *static*: its right-hand-side is a constant (more generally, simplifies to a constant), so we should update our partial state at  $X$  to 3 and produce no residual code. (Actually, we produce a residual *SKIP*.) The second assignment  $Y ::= \text{AMult (Ald Z) (APlus (Ald X) (Ald X))}$  is *dynamic*: its right-hand-side does not simplify to a constant, so we should leave it in the residual code and remove  $Y$ , if present, from our partial state. To implement these two cases, we define the functions `pe_add` and `pe_remove`. Like `pe_update` above, these functions operate on a concrete **list** representing a `pe_state`, but the theorems `pe_add_correct` and `pe_remove_correct` specify their behavior by the `pe_lookup` interpretation of the `pe_state`.

```
Fixpoint pe_remove (pe_st:pe_state) (V:id) : pe_state :=
  match pe_st with
  | [] => []
  | (V',n')::pe_st => if beq_id V V' then pe_remove pe_st V
                      else (V',n') :: pe_remove pe_st V
  end.
```

```
Theorem pe_remove_correct: ∀ pe_st V V0,
  pe_lookup (pe_remove pe_st V) V0
  = if beq_id V V0 then None else pe_lookup pe_st V0.
```

```
Proof. intros pe_st V V0. induction pe_st as [| [V' n'] pe_st].
- destruct (beq_id V V0); reflexivity.
- simpl. compare V V'.
  + rewrite IHpe_st.
    destruct (beq_idP V V0). reflexivity. rewrite false_beq_id; auto.
  + simpl. compare V0 V'.
    × rewrite false_beq_id; auto.
    × rewrite IHpe_st. reflexivity.
```

Qed.

```
Definition pe_add (pe_st:pe_state) (V:id) (n:nat) : pe_state :=
  (V,n) :: pe_remove pe_st V.
```

```
Theorem pe_add_correct: ∀ pe_st V n V0,
  pe_lookup (pe_add pe_st V n) V0
  = if beq_id V V0 then Some n else pe_lookup pe_st V0.
```

```
Proof. intros pe_st V n V0. unfold pe_add. simpl.
  compare V V0.
- rewrite ← beq_id_refl; auto.
- rewrite pe_remove_correct. repeat rewrite false_beq_id; auto.
```

Qed.

We will use the two theorems below to show that our partial evaluator correctly deals with dynamic assignments and static assignments, respectively.

Theorem `pe_update_update_remove`:  $\forall st\ pe\_st\ V\ n,$   
`t_update (pe_update st pe_st) V n =`  
`pe_update (t_update st V n) (pe_remove pe_st V).`

Proof. `intros st pe_st V n. apply functional_extensibility. intros V0.`  
`unfold t_update. rewrite !pe_update_correct. rewrite pe_remove_correct.`  
`destruct (beq_id V V0); reflexivity. Qed.`

Theorem `pe_update_update_add`:  $\forall st\ pe\_st\ V\ n,$   
`t_update (pe_update st pe_st) V n =`  
`pe_update st (pe_add pe_st V n).`

Proof. `intros st pe_st V n. apply functional_extensibility. intros V0.`  
`unfold t_update. rewrite !pe_update_correct. rewrite pe_add_correct.`  
`destruct (beq_id V V0); reflexivity. Qed.`

### 37.3.2 Conditional

Trickier than assignments to partially evaluate is the conditional, *IFB b1 THEN c1 ELSE c2 FI*. If *b1* simplifies to `BTrue` or `BFalse` then it's easy: we know which branch will be taken, so just take that branch. If *b1* does not simplify to a constant, then we need to take both branches, and the final partial state may differ between the two branches!

The following program illustrates the difficulty: `X ::= ANum 3;; IFB BLe (AId Y) (ANum 4) THEN Y ::= ANum 4;; IFB BEq (AId X) (AId Y) THEN Y ::= ANum 999 ELSE SKIP FI ELSE SKIP FI` Suppose the initial partial state is empty. We don't know statically how *Y* compares to 4, so we must partially evaluate both branches of the (outer) conditional. On the *THEN* branch, we know that *Y* is set to 4 and can even use that knowledge to simplify the code somewhat. On the *ELSE* branch, we still don't know the exact value of *Y* at the end. What should the final partial state and residual program be?

One way to handle such a dynamic conditional is to take the intersection of the final partial states of the two branches. In this example, we take the intersection of  $(Y,4),(X,3)$  and  $(X,3)$ , so the overall final partial state is  $(X,3)$ . To compensate for forgetting that *Y* is 4, we need to add an assignment `Y ::= ANum 4` to the end of the *THEN* branch. So, the residual program will be something like `SKIP;; IFB BLe (AId Y) (ANum 4) THEN SKIP;; SKIP;; Y ::= ANum 4 ELSE SKIP FI`

Programming this case in Coq calls for several auxiliary functions: we need to compute the intersection of two `pe_states` and turn their difference into sequences of assignments.

First, we show how to compute whether two `pe_states` to disagree at a given variable. In the theorem `pe_disagree_domain`, we prove that two `pe_states` can only disagree at variables that appear in at least one of them.

Definition `pe_disagree_at (pe_st1 pe_st2 : pe_state) (V:id) : bool :=`  
`match pe_lookup pe_st1 V, pe_lookup pe_st2 V with`  
`| Some x, Some y => negb (beq_nat x y)`  
`| None, None => false`  
`| -, - => true`

end.

Theorem `pe_disagree_domain`:  $\forall (pe\_st1\ pe\_st2 : pe\_state) (V:id),$   
`true = pe_disagree_at pe_st1 pe_st2 V  $\rightarrow$`   
`In V (map (@fst -) pe_st1 ++ map (@fst -) pe_st2).`

Proof. `unfold pe_disagree_at. intros pe_st1 pe_st2 V H.`

`apply in_app_iff.`

`remember (pe_lookup pe_st1 V) as lookup1.`

`destruct lookup1 as [n1]. left. apply pe_domain with n1. auto.`

`remember (pe_lookup pe_st2 V) as lookup2.`

`destruct lookup2 as [n2]. right. apply pe_domain with n2. auto.`

`inversion H. Qed.`

We define the `pe_compare` function to list the variables where two given `pe_states` disagree. This list is exact, according to the theorem `pe_compare_correct`: a variable appears on the list if and only if the two given `pe_states` disagree at that variable. Furthermore, we use the `pe_unique` function to eliminate duplicates from the list.

Fixpoint `pe_unique (l : list id) : list id :=`

`match l with`

`| []  $\Rightarrow$  []`

`| x::l  $\Rightarrow$  x :: filter (fun y  $\Rightarrow$  if beq_id x y then false else true) (pe_unique l)`

`end.`

Theorem `pe_unique_correct`:  $\forall l\ x,$

`In x l  $\leftrightarrow$  In x (pe_unique l).`

Proof. `intros l x. induction l as [| h t]. reflexivity.`

`simpl in *. split.`

-

`intros. inversion H; clear H.`

`left. assumption.`

`destruct (beq_idP h x).`

`left. assumption.`

`right. apply filter_In. split.`

`apply IHt. assumption.`

`rewrite false_beq_id; auto.`

-

`intros. inversion H; clear H.`

`left. assumption.`

`apply filter_In in H0. inversion H0. right. apply IHt. assumption.`

Qed.

Definition `pe_compare (pe_st1 pe_st2 : pe_state) : list id :=`

`pe_unique (filter (pe_disagree_at pe_st1 pe_st2)`

`(map (@fst -) pe_st1 ++ map (@fst -) pe_st2)).`

Theorem `pe_compare_correct`:  $\forall pe\_st1\ pe\_st2\ V,$

```

pe_lookup pe_st1 V = pe_lookup pe_st2 V ↔
  → In V (pe_compare pe_st1 pe_st2).
Proof. intros pe_st1 pe_st2 V.
unfold pe_compare. rewrite ← pe_unique_correct. rewrite filter_In.
split; intros Heq.
-
  intro. destruct H. unfold pe_disagree_at in H0. rewrite Heq in H0.
  destruct (pe_lookup pe_st2 V).
  rewrite ← beq_nat_refl in H0. inversion H0.
  inversion H0.
-
  assert (Hagree: pe_disagree_at pe_st1 pe_st2 V = false).
  {
    remember (pe_disagree_at pe_st1 pe_st2 V) as disagree.
    destruct disagree; [| reflexivity].
    apply pe_disagree_domain in Heqdisagree.
    exfalso. apply Heq. split. assumption. reflexivity. }
  unfold pe_disagree_at in Hagree.
  destruct (pe_lookup pe_st1 V) as [n1|];
  destruct (pe_lookup pe_st2 V) as [n2|];
  try reflexivity; try solve by inversion.
  rewrite negb_false_iff in Hagree.
  apply beq_nat_true in Hagree. subst. reflexivity. Qed.

```

The intersection of two partial states is the result of removing from one of them all the variables where the two disagree. We define the function `pe_removes`, in terms of `pe_remove` above, to perform such a removal of a whole list of variables at once.

The theorem `pe_compare_removes` testifies that the `pe_lookup` interpretation of the result of this intersection operation is the same no matter which of the two partial states we remove the variables from. Because `pe_update` only depends on the `pe_lookup` interpretation of partial states, `pe_update` also does not care which of the two partial states we remove the variables from; that theorem `pe_compare_update` is used in the correctness proof shortly.

```

Fixpoint pe_removes (pe_st:pe_state) (ids : list id) : pe_state :=
  match ids with
  | [] ⇒ pe_st
  | V::ids ⇒ pe_remove (pe_removes pe_st ids) V
  end.

```

Theorem `pe_removes_correct`:  $\forall pe\_st\ ids\ V,$   
`pe_lookup (pe_removes pe_st ids) V =`  
 if `inb beq_id V ids` then `None` else `pe_lookup pe_st V`.

Proof. intros `pe_st ids V`. induction `ids` as [| V' ids]. reflexivity.  
 simpl. rewrite `pe_remove_correct`. rewrite `IHids`.  
 compare V' V.

```

- rewrite ← beq_id_refl. reflexivity.
- rewrite false_beq_id; try congruence. reflexivity.

```

Qed.

Theorem `pe_compare_removes`:  $\forall pe\_st1\ pe\_st2\ V,$   
`pe_lookup (pe_removes pe_st1 (pe_compare pe_st1 pe_st2)) V =`  
`pe_lookup (pe_removes pe_st2 (pe_compare pe_st1 pe_st2)) V.`

Proof.

```

intros pe_st1 pe_st2 V. rewrite !pe_removes_correct.
destruct (inbP _ _ beq_idP V (pe_compare pe_st1 pe_st2)).
- reflexivity.
- apply pe_compare_correct. auto. Qed.

```

Theorem `pe_compare_update`:  $\forall pe\_st1\ pe\_st2\ st,$   
`pe_update st (pe_removes pe_st1 (pe_compare pe_st1 pe_st2)) =`  
`pe_update st (pe_removes pe_st2 (pe_compare pe_st1 pe_st2)).`

Proof. intros. apply **functional\_extensionality**. intros V.

```

rewrite !pe_update_correct. rewrite pe_compare_removes. reflexivity.

```

Qed.

Finally, we define an `assign` function to turn the difference between two partial states into a sequence of assignment commands. More precisely, `assign pe_st ids` generates an assignment command for each variable listed in *ids*.

```

Fixpoint assign (pe_st : pe_state) (ids : list id) : com :=
  match ids with
  | [] ⇒ SKIP
  | V :: ids ⇒ match pe_lookup pe_st V with
    | Some n ⇒ (assign pe_st ids ;; V ::= ANum n)
    | None ⇒ assign pe_st ids
  end
end.

```

The command generated by `assign` always terminates, because it is just a sequence of assignments. The (total) function `assigned` below computes the effect of the command on the (dynamic state). The theorem `assign_removes` then confirms that the generated assignments perfectly compensate for removing the variables from the partial state.

Definition `assigned` (*pe\_st*:pe\_state) (*ids* : list id) (*st*:state) : state :=

```

fun V ⇒ if inb beq_id V ids then
  match pe_lookup pe_st V with
  | Some n ⇒ n
  | None ⇒ st V
  end
else st V.

```

Theorem `assign_removes`:  $\forall pe\_st\ ids\ st,$   
`pe_update st pe_st =`

```

    pe_update (assigned pe_st ids st) (pe_removes pe_st ids).
Proof. intros pe_st ids st. apply functional_extensionality. intros V.
    rewrite !pe_update_correct. rewrite pe_removes_correct. unfold assigned.
    destruct (inbP _ _ beq_idP V ids); destruct (pe_lookup pe_st V); reflexivity.
Qed.

Lemma ceval_extensionality:  $\forall c \ st \ st1 \ st2,$ 
     $c / st \setminus st1 \rightarrow (\forall V, st1 \ V = st2 \ V) \rightarrow c / st \setminus st2.$ 
Proof. intros c st st1 st2 H Heq.
    apply functional_extensionality in Heq. rewrite  $\leftarrow$  Heq. apply H. Qed.

Theorem eval_assign:  $\forall pe\_st \ ids \ st,$ 
    assign pe_st ids / st  $\setminus$  assigned pe_st ids st.
Proof. intros pe_st ids st. induction ids as [| V ids]; simpl.
- eapply ceval_extensionality. apply E_Skip. reflexivity.
-
    remember (pe_lookup pe_st V) as lookup. destruct lookup.
+ eapply E_Seq. apply IHids. unfold assigned. simpl.
    eapply ceval_extensionality. apply E_Ass. simpl. reflexivity.
    intros V0. unfold t_update. compare V V0.
     $\times$  rewrite  $\leftarrow$  Heqlookup. rewrite  $\leftarrow$  beq_id_refl. reflexivity.
     $\times$  rewrite false_beq_id; simpl; congruence.
+ eapply ceval_extensionality. apply IHids.
    unfold assigned. intros V0. simpl. compare V V0.
     $\times$  rewrite  $\leftarrow$  Heqlookup.
    rewrite  $\leftarrow$  beq_id_refl.
    destruct (inbP _ _ beq_idP V ids); reflexivity.
     $\times$  rewrite false_beq_id; simpl; congruence.
Qed.

```

### 37.3.3 The Partial Evaluation Relation

At long last, we can define a partial evaluator for commands without loops, as an inductive relation! The inequality conditions in PE\_AssDynamic and PE\_If are just to keep the partial evaluator deterministic; they are not required for correctness.

Reserved Notation " $c1 \ '/' \ st \ '\setminus' \ c1' \ '/' \ st'$ "  
 (at level 40,  $st$  at level 39,  $c1'$  at level 39).

Inductive **pe\_com** : **com**  $\rightarrow$  pe\_state  $\rightarrow$  **com**  $\rightarrow$  pe\_state  $\rightarrow$  Prop :=  
 | PE\_Skip :  $\forall pe\_st,$   
     SKIP / pe\_st  $\setminus$  SKIP / pe\_st  
 | PE\_AssStatic :  $\forall pe\_st \ a1 \ n1 \ l,$   
     pe\_aexp pe\_st a1 = ANum n1  $\rightarrow$   
     ( $l ::= a1$ ) / pe\_st  $\setminus$  SKIP / pe\_add pe\_st l n1  
 | PE\_AssDynamic :  $\forall pe\_st \ a1 \ a1' \ l,$

```

    pe_aexp pe_st a1 = a1' →
    (∀ n, a1' ≠ ANum n) →
    (l ::= a1) / pe_st \\\ (l ::= a1') / pe_remove pe_st l
| PE_Seq : ∀ pe_st pe_st' pe_st'' c1 c2 c1' c2',
    c1 / pe_st \\\ c1' / pe_st' →
    c2 / pe_st' \\\ c2' / pe_st'' →
    (c1 ;; c2) / pe_st \\\ (c1' ;; c2') / pe_st''
| PE_IfTrue : ∀ pe_st pe_st' b1 c1 c2 c1',
    pe_bexp pe_st b1 = BTrue →
    c1 / pe_st \\\ c1' / pe_st' →
    (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\\ c1' / pe_st'
| PE_IfFalse : ∀ pe_st pe_st' b1 c1 c2 c2',
    pe_bexp pe_st b1 = BFalse →
    c2 / pe_st \\\ c2' / pe_st' →
    (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\\ c2' / pe_st'
| PE_If : ∀ pe_st pe_st1 pe_st2 b1 c1 c2 c1' c2',
    pe_bexp pe_st b1 ≠ BTrue →
    pe_bexp pe_st b1 ≠ BFalse →
    c1 / pe_st \\\ c1' / pe_st1 →
    c2 / pe_st \\\ c2' / pe_st2 →
    (IFB b1 THEN c1 ELSE c2 FI) / pe_st
    \\\ (IFB pe_bexp pe_st b1
        THEN c1' ;; assign pe_st1 (pe_compare pe_st1 pe_st2)
        ELSE c2' ;; assign pe_st2 (pe_compare pe_st1 pe_st2) FI)
    / pe_removes pe_st1 (pe_compare pe_st1 pe_st2)

```

where "c1 '/' st '\\\' c1' '/' st'" := (**pe\_com** c1 st c1' st').

Hint Constructors **pe\_com**.

Hint Constructors **ceval**.

### 37.3.4 Examples

Below are some examples of using the partial evaluator. To make the **pe\_com** relation actually usable for automatic partial evaluation, we would need to define more automation tactics in Coq. That is not hard to do, but it is not needed here.

Example pe\_example1:

```

(X ::= ANum 3 ;; Y ::= AMult (Ald Z) (APlus (Ald X) (Ald X)))
/ [] \\\ (SKIP ;; Y ::= AMult (Ald Z) (ANum 6)) / [(X,3)].

```

Proof. eapply PE\_Seq. eapply PE\_AssStatic. reflexivity.

eapply PE\_AssDynamic. reflexivity. intros n H. inversion H. Qed.

Example pe\_example2:

```

(X ::= ANum 3 ;; IFB BLe (Ald X) (ANum 4) THEN X ::= ANum 4 ELSE SKIP FI)

```



```

/ [] \\ (SKIP;; SKIP) / [(X,4)].
Proof. eapply PE_Seq. eapply PE_AssStatic. reflexivity.
      eapply PE_IfTrue. reflexivity.
      eapply PE_AssStatic. reflexivity. Qed.

Example pe_example3:
  (X ::= ANum 3;;
   IFB BLe (Ald Y) (ANum 4) THEN
     Y ::= ANum 4;;
     IFB BEq (Ald X) (Ald Y) THEN Y ::= ANum 999 ELSE SKIP FI
   ELSE SKIP FI) / []
  \\ (SKIP;;
      IFB BLe (Ald Y) (ANum 4) THEN
        (SKIP;; SKIP);; (SKIP;; Y ::= ANum 4)
      ELSE SKIP;; SKIP FI)
  / [(X,3)].
Proof. erewrite f_equal2 with (f := fun c st => _ / _ \\ c / st).
      eapply PE_Seq. eapply PE_AssStatic. reflexivity.
      eapply PE_If; intuition eauto; try solve by inversion.
      econstructor. eapply PE_AssStatic. reflexivity.
      eapply PE_IfFalse. reflexivity. econstructor.
      reflexivity. reflexivity. Qed.

```

### 37.3.5 Correctness of Partial Evaluation

Finally let's prove that this partial evaluator is correct!

Reserved Notation "c' '/' pe\_st' '/' st '\\\\' st'"  
 (at level 40, *pe\_st'* at level 39, *st* at level 39).

Inductive **pe\_ceval**

```

(c':com) (pe_st':pe_state) (st:state) (st'':state) : Prop :=
| pe_ceval_intro : ∀ st',
  c' / st \\ st' →
  pe_update st' pe_st' = st'' →
  c' / pe_st' / st \\ st''
where "c' '/' pe_st' '/' st '\\\\' st'" := (pe_ceval c' pe_st' st st'').

```

Hint Constructors **pe\_ceval**.

Theorem **pe\_com\_complete**:

```

∀ c pe_st pe_st' c', c / pe_st \\ c' / pe_st' →
∀ st st'',
(c / pe_update st pe_st \\ st'') →
(c' / pe_st' / st \\ st'').

```

Proof. intros c pe\_st pe\_st' c' Hpe.

```

induction Hpe; intros st st'' Heval;
try (inversion Heval; subst;
    try (rewrite → pe_bexp_correct, → H in *; solve by inversion);
    []);
eauto.
- econstructor. econstructor.
  rewrite → pe_aexp_correct. rewrite ← pe_update_update_add.
  rewrite → H. reflexivity.
- econstructor. econstructor. reflexivity.
  rewrite → pe_aexp_correct. rewrite ← pe_update_update_remove.
  reflexivity.
-
  edestruct IHHpe1. eassumption. subst.
  edestruct IHHpe2. eassumption.
  eauto.
- inversion Heval; subst.
  + edestruct IHHpe1. eassumption.
    econstructor. apply E_lfTrue. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite ← assign_removes. eassumption.
  + edestruct IHHpe2. eassumption.
    econstructor. apply E_lfFalse. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite → pe_compare_update.
    rewrite ← assign_removes. eassumption.
Qed.

```

Theorem pe\_com\_sound:

$$\begin{aligned}
& \forall c \text{ pe\_st } \text{pe\_st}' \ c', \ c \ / \ \text{pe\_st} \ \backslash \backslash \ c' \ / \ \text{pe\_st}' \rightarrow \\
& \forall st \ st'', \\
& (c' \ / \ \text{pe\_st}' \ / \ st \ \backslash \backslash \ st'') \rightarrow \\
& (c \ / \ \text{pe\_update } st \ \text{pe\_st} \ \backslash \backslash \ st'').
\end{aligned}$$

Proof. intros c pe\_st pe\_st' c' Hpe.

```

induction Hpe;
  intros st st'' [st' Heval Heq];
  try (inversion Heval; []; subst); auto.
- rewrite ← pe_update_update_add. apply E_Ass.
  rewrite → pe_aexp_correct. rewrite → H. reflexivity.
- rewrite ← pe_update_update_remove. apply E_Ass.
  rewrite ← pe_aexp_correct. reflexivity.
- eapply E_Seq; eauto.
- apply E_lfTrue.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.

```

```

- apply E_IfFalse.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.
-
  inversion Heval; subst; inversion H7;
  (eapply ceval_deterministic in H8; [| apply eval_assign]); subst.
+
  apply E_IfTrue. rewrite → pe_bexp_correct. assumption.
  rewrite ← assign_removes. eauto.
+
  rewrite → pe_compare_update.
  apply E_IfFalse. rewrite → pe_bexp_correct. assumption.
  rewrite ← assign_removes. eauto.

```

Qed.

The main theorem. Thanks to David Menendez for this formulation!

Corollary `pe_com_correct`:

$$\begin{aligned}
& \forall c \text{ pe\_st } pe\_st' \ c', \ c \ / \ pe\_st \ \backslash\backslash \ c' \ / \ pe\_st' \rightarrow \\
& \forall st \ st'', \\
& (c \ / \ pe\_update \ st \ pe\_st \ \backslash\backslash \ st'') \leftrightarrow \\
& (c' \ / \ pe\_st' \ / \ st \ \backslash\backslash \ st'').
\end{aligned}$$

Proof. `intros c pe_st pe_st' c' H st st''. split.`

```

- apply pe_com_complete. apply H.
- apply pe_com_sound. apply H.

```

Qed.

## 37.4 Partial Evaluation of Loops

It may seem straightforward at first glance to extend the partial evaluation relation `pe_com` above to loops. Indeed, many loops are easy to deal with. Considered this repeated-squaring loop, for example: `WHILE BLe (ANum 1) (AId X) DO Y ::= AMult (AId Y) (AId Y);; X ::= AMinus (AId X) (ANum 1) END` If we know neither `X` nor `Y` statically, then the entire loop is dynamic and the residual command should be the same. If we know `X` but not `Y`, then the loop can be unrolled all the way and the residual command should be, for example, `Y ::= AMult (AId Y) (AId Y);; Y ::= AMult (AId Y) (AId Y);; Y ::= AMult (AId Y) (AId Y)` if `X` is initially 3 (and finally 0). In general, a loop is easy to partially evaluate if the final partial state of the loop body is equal to the initial state, or if its guard condition is static.

But there are other loops for which it is hard to express the residual program we want in `Imp`. For example, take this program for checking whether `Y` is even or odd: `X ::= ANum 0;; WHILE BLe (ANum 1) (AId Y) DO Y ::= AMinus (AId Y) (ANum 1);; X ::= AMinus (ANum 1) (AId X) END` The value of `X` alternates between 0 and 1 during the loop. Ideally, we would like to unroll this loop, not all the way but *two-fold*, into something like `WHILE BLe (ANum 1) (AId Y) DO Y ::= AMinus (AId Y) (ANum 1);; IF BLe (ANum 1) (AId`

Y) THEN Y ::= AMinus (AId Y) (ANum 1) ELSE X ::= ANum 1;; EXIT FI END;; X ::= ANum 0 Unfortunately, there is no *EXIT* command in Imp. Without extending the range of control structures available in our language, the best we can do is to repeat loop-guard tests or add flag variables. Neither option is terribly attractive.

Still, as a digression, below is an attempt at performing partial evaluation on Imp commands. We add one more command argument  $c''$  to the **pe\_com** relation, which keeps track of a loop to roll up.

Module LOOP.

Reserved Notation " $c1 \text{ '/' } st \text{ '\\ } c1' \text{ '/' } st' \text{ '/' } c''$ "

(at level 40,  $st$  at level 39,  $c1'$  at level 39,  $st'$  at level 39).

Inductive **pe\_com** : **com** → pe\_state → **com** → pe\_state → **com** → Prop :=

- | PE\_Skip :  $\forall pe\_st,$   
SKIP /  $pe\_st$  \\ SKIP /  $pe\_st$  / SKIP
- | PE\_AssStatic :  $\forall pe\_st \ a1 \ n1 \ l,$   
pe\_aexp  $pe\_st \ a1 = \text{ANum } n1 \rightarrow$   
( $l ::= a1$ ) /  $pe\_st$  \\ SKIP / pe\_add  $pe\_st \ l \ n1$  / SKIP
- | PE\_AssDynamic :  $\forall pe\_st \ a1 \ a1' \ l,$   
pe\_aexp  $pe\_st \ a1 = a1' \rightarrow$   
( $\forall n, a1' \neq \text{ANum } n$ )  $\rightarrow$   
( $l ::= a1$ ) /  $pe\_st$  \\ ( $l ::= a1'$ ) / pe\_remove  $pe\_st \ l$  / SKIP
- | PE\_Seq :  $\forall pe\_st \ pe\_st' \ pe\_st'' \ c1 \ c2 \ c1' \ c2' \ c'',$   
 $c1$  /  $pe\_st$  \\  $c1'$  /  $pe\_st'$  / SKIP  $\rightarrow$   
 $c2$  /  $pe\_st'$  \\  $c2'$  /  $pe\_st''$  /  $c'' \rightarrow$   
( $c1 ; ; c2$ ) /  $pe\_st$  \\ ( $c1' ; ; c2'$ ) /  $pe\_st''$  /  $c''$
- | PE\_IfTrue :  $\forall pe\_st \ pe\_st' \ b1 \ c1 \ c2 \ c1' \ c'',$   
pe\_bexp  $pe\_st \ b1 = \text{BTrue} \rightarrow$   
 $c1$  /  $pe\_st$  \\  $c1'$  /  $pe\_st'$  /  $c'' \rightarrow$   
(IFB  $b1$  THEN  $c1$  ELSE  $c2$  FI) /  $pe\_st$  \\  $c1'$  /  $pe\_st'$  /  $c''$
- | PE\_IfFalse :  $\forall pe\_st \ pe\_st' \ b1 \ c1 \ c2 \ c2' \ c'',$   
pe\_bexp  $pe\_st \ b1 = \text{BFalse} \rightarrow$   
 $c2$  /  $pe\_st$  \\  $c2'$  /  $pe\_st'$  /  $c'' \rightarrow$   
(IFB  $b1$  THEN  $c1$  ELSE  $c2$  FI) /  $pe\_st$  \\  $c2'$  /  $pe\_st'$  /  $c''$
- | PE\_If :  $\forall pe\_st \ pe\_st1 \ pe\_st2 \ b1 \ c1 \ c2 \ c1' \ c2' \ c'',$   
pe\_bexp  $pe\_st \ b1 \neq \text{BTrue} \rightarrow$   
pe\_bexp  $pe\_st \ b1 \neq \text{BFalse} \rightarrow$   
 $c1$  /  $pe\_st$  \\  $c1'$  /  $pe\_st1$  /  $c'' \rightarrow$   
 $c2$  /  $pe\_st$  \\  $c2'$  /  $pe\_st2$  /  $c'' \rightarrow$   
(IFB  $b1$  THEN  $c1$  ELSE  $c2$  FI) /  $pe\_st$   
\\ (IFB pe\_bexp  $pe\_st \ b1$   
THEN  $c1' ; ; \text{assign } pe\_st1 \ (\text{pe\_compare } pe\_st1 \ pe\_st2)$   
ELSE  $c2' ; ; \text{assign } pe\_st2 \ (\text{pe\_compare } pe\_st1 \ pe\_st2)$  FI)  
/ pe\_removes  $pe\_st1 \ (\text{pe\_compare } pe\_st1 \ pe\_st2)$

/ c''

| PE\_WhileEnd :  $\forall pe\_st\ b1\ c1,$   
 $pe\_bexp\ pe\_st\ b1 = BFalse \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st \ \backslash\ \backslash\ SKIP / pe\_st / SKIP$

| PE\_WhileLoop :  $\forall pe\_st\ pe\_st'\ pe\_st''\ b1\ c1\ c1'\ c2'\ c2'',$   
 $pe\_bexp\ pe\_st\ b1 = BTrue \rightarrow$   
 $c1 / pe\_st \ \backslash\ \backslash\ c1' / pe\_st' / SKIP \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st' \ \backslash\ \backslash\ c2' / pe\_st'' / c2'' \rightarrow$   
 $pe\_compare\ pe\_st\ pe\_st'' \neq \square \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st \ \backslash\ \backslash\ (c1'; ; c2') / pe\_st'' / c2''$

| PE\_While :  $\forall pe\_st\ pe\_st'\ pe\_st''\ b1\ c1\ c1'\ c2'\ c2'',$   
 $pe\_bexp\ pe\_st\ b1 \neq BFalse \rightarrow$   
 $pe\_bexp\ pe\_st\ b1 \neq BTrue \rightarrow$   
 $c1 / pe\_st \ \backslash\ \backslash\ c1' / pe\_st' / SKIP \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st' \ \backslash\ \backslash\ c2' / pe\_st'' / c2'' \rightarrow$   
 $pe\_compare\ pe\_st\ pe\_st'' \neq \square \rightarrow$   
 $(c2'' = SKIP \vee c2'' = WHILE\ b1\ DO\ c1\ END) \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st$   
 $\ \backslash\ \backslash\ (IFB\ pe\_bexp\ pe\_st\ b1$   
 $\ \ \ \ THEN\ c1'; ; c2'; ; assign\ pe\_st''\ (pe\_compare\ pe\_st\ pe\_st'')$   
 $\ \ \ \ ELSE\ assign\ pe\_st\ (pe\_compare\ pe\_st\ pe\_st'')\ FI)$   
 $\ / \ pe\_removes\ pe\_st\ (pe\_compare\ pe\_st\ pe\_st'')$   
 $\ / \ c2''$

| PE\_WhileFixedEnd :  $\forall pe\_st\ b1\ c1,$   
 $pe\_bexp\ pe\_st\ b1 \neq BFalse \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st \ \backslash\ \backslash\ SKIP / pe\_st / (WHILE\ b1\ DO\ c1\ END)$

| PE\_WhileFixedLoop :  $\forall pe\_st\ pe\_st'\ pe\_st''\ b1\ c1\ c1'\ c2',$   
 $pe\_bexp\ pe\_st\ b1 = BTrue \rightarrow$   
 $c1 / pe\_st \ \backslash\ \backslash\ c1' / pe\_st' / SKIP \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st'$   
 $\ \backslash\ \backslash\ c2' / pe\_st'' / (WHILE\ b1\ DO\ c1\ END) \rightarrow$   
 $pe\_compare\ pe\_st\ pe\_st'' = \square \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st$   
 $\ \backslash\ \backslash\ (WHILE\ BTrue\ DO\ SKIP\ END) / pe\_st / SKIP$

| PE\_WhileFixed :  $\forall pe\_st\ pe\_st'\ pe\_st''\ b1\ c1\ c1'\ c2',$   
 $pe\_bexp\ pe\_st\ b1 \neq BFalse \rightarrow$   
 $pe\_bexp\ pe\_st\ b1 \neq BTrue \rightarrow$   
 $c1 / pe\_st \ \backslash\ \backslash\ c1' / pe\_st' / SKIP \rightarrow$   
 $(WHILE\ b1\ DO\ c1\ END) / pe\_st'$   
 $\ \backslash\ \backslash\ c2' / pe\_st'' / (WHILE\ b1\ DO\ c1\ END) \rightarrow$   
 $pe\_compare\ pe\_st\ pe\_st'' = \square \rightarrow$

```
(WHILE b1 DO c1 END) / pe_st
  \ (WHILE pe_bexp pe_st b1 DO c1';; c2' END) / pe_st / SKIP
```

where "c1 '/' st '\ ' c1' '/' st' '/' c'" := (**pe\_com** c1 st c1' st' c').

Hint Constructors **pe\_com**.

### 37.4.1 Examples

```
Ltac step i :=
  (eapply i; intuition eauto; try solve by inversion);
  repeat (try eapply PE_Seq;
    try (eapply PE_AssStatic; simpl; reflexivity);
    try (eapply PE_AssDynamic;
      | simpl; reflexivity
      | intuition eauto; solve by inversion ])).
```

```
Definition square_loop: com :=
  WHILE BLe (ANum 1) (Ald X) DO
    Y ::= AMult (Ald Y) (Ald Y);;
    X ::= AMinus (Ald X) (ANum 1)
  END.
```

```
Example pe_loop_example1:
  square_loop / □
  \ (WHILE BLe (ANum 1) (Ald X) DO
    (Y ::= AMult (Ald Y) (Ald Y);;
    X ::= AMinus (Ald X) (ANum 1));; SKIP
  END) / □ / SKIP.
```

Proof. *erewrite* f\_equal2 with (f := fun c st => \_ / \_ \ c / st / SKIP).  
*step* PE\_WhileFixed. *step* PE\_WhileFixedEnd. reflexivity.  
 reflexivity. reflexivity. Qed.

```
Example pe_loop_example2:
  (X ::= ANum 3;; square_loop) / □
  \ (SKIP;;
    (Y ::= AMult (Ald Y) (Ald Y);; SKIP);;
    (Y ::= AMult (Ald Y) (Ald Y);; SKIP);;
    (Y ::= AMult (Ald Y) (Ald Y);; SKIP);;
    SKIP) / [(X,0)] / SKIP.
```

Proof. *erewrite* f\_equal2 with (f := fun c st => \_ / \_ \ c / st / SKIP).  
 eapply PE\_Seq. eapply PE\_AssStatic. reflexivity.  
*step* PE\_WhileLoop.  
*step* PE\_WhileLoop.  
*step* PE\_WhileLoop.

*step* PE\_WhileEnd.  
 inversion H. inversion H. inversion H.  
 reflexivity. reflexivity. Qed.

Example pe\_loop\_example3:

```

(Z ::= ANum 3;; subtract_slowly) / □
\\ (SKIP;;
  IFB BNot (BEq (Ald X) (ANum 0)) THEN
    (SKIP;; X ::= AMinus (Ald X) (ANum 1));;
  IFB BNot (BEq (Ald X) (ANum 0)) THEN
    (SKIP;; X ::= AMinus (Ald X) (ANum 1));;
  IFB BNot (BEq (Ald X) (ANum 0)) THEN
    (SKIP;; X ::= AMinus (Ald X) (ANum 1));;
  WHILE BNot (BEq (Ald X) (ANum 0)) DO
    (SKIP;; X ::= AMinus (Ald X) (ANum 1));; SKIP
  END;;
  SKIP;; Z ::= ANum 0
  ELSE SKIP;; Z ::= ANum 1 FI;; SKIP
  ELSE SKIP;; Z ::= ANum 2 FI;; SKIP
  ELSE SKIP;; Z ::= ANum 3 FI) / □ / SKIP.

```

Proof. *erewrite* f\_equal2 with (f := fun c st ⇒ \_ / \_ \\ c / st / SKIP).

eapply PE\_Seq. eapply PE\_AssStatic. reflexivity.  
*step* PE\_While.  
*step* PE\_While.  
*step* PE\_While.  
*step* PE\_WhileFixed.  
*step* PE\_WhileFixedEnd.  
 reflexivity. inversion H. inversion H. inversion H.  
 reflexivity. reflexivity. Qed.

Example pe\_loop\_example4:

```

(X ::= ANum 0;;
  WHILE BLe (Ald X) (ANum 2) DO
    X ::= AMinus (ANum 1) (Ald X)
  END) / □ \\ (SKIP;; WHILE BTrue DO SKIP END) / [(X,0)] / SKIP.

```

Proof. *erewrite* f\_equal2 with (f := fun c st ⇒ \_ / \_ \\ c / st / SKIP).

eapply PE\_Seq. eapply PE\_AssStatic. reflexivity.  
*step* PE\_WhileFixedLoop.  
*step* PE\_WhileLoop.  
*step* PE\_WhileFixedEnd.  
 inversion H. reflexivity. reflexivity. reflexivity. Qed.

### 37.4.2 Correctness

Because this partial evaluator can unroll a loop  $n$ -fold where  $n$  is a (finite) integer greater than one, in order to show it correct we need to perform induction not structurally on dynamic evaluation but on the number of times dynamic evaluation enters a loop body.

Reserved Notation " $c1 \text{ '}' st \text{ '}' \setminus \setminus st' \text{ '}' \# n$ "  
(at level 40,  $st$  at level 39,  $st'$  at level 39).

Inductive **ceval\_count** : **com**  $\rightarrow$  state  $\rightarrow$  state  $\rightarrow$  **nat**  $\rightarrow$  Prop :=

```

| E'Skip :  $\forall st,$ 
  SKIP /  $st \setminus \setminus st \# 0$ 
| E'Ass :  $\forall st a1 n l,$ 
  aeval  $st a1 = n \rightarrow$ 
  ( $l ::= a1$ ) /  $st \setminus \setminus (t\_update\ st\ l\ n) \# 0$ 
| E'Seq :  $\forall c1 c2 st st' st'' n1 n2,$ 
   $c1 / st \setminus \setminus st' \# n1 \rightarrow$ 
   $c2 / st' \setminus \setminus st'' \# n2 \rightarrow$ 
   $(c1 ;; c2) / st \setminus \setminus st'' \# (n1 + n2)$ 
| E'IfTrue :  $\forall st st' b1 c1 c2 n,$ 
  beval  $st b1 = \text{true} \rightarrow$ 
   $c1 / st \setminus \setminus st' \# n \rightarrow$ 
  (IFB  $b1$  THEN  $c1$  ELSE  $c2$  FI) /  $st \setminus \setminus st' \# n$ 
| E'IfFalse :  $\forall st st' b1 c1 c2 n,$ 
  beval  $st b1 = \text{false} \rightarrow$ 
   $c2 / st \setminus \setminus st' \# n \rightarrow$ 
  (IFB  $b1$  THEN  $c1$  ELSE  $c2$  FI) /  $st \setminus \setminus st' \# n$ 
| E'WhileEnd :  $\forall b1 st c1,$ 
  beval  $st b1 = \text{false} \rightarrow$ 
  (WHILE  $b1$  DO  $c1$  END) /  $st \setminus \setminus st \# 0$ 
| E'WhileLoop :  $\forall st st' st'' b1 c1 n1 n2,$ 
  beval  $st b1 = \text{true} \rightarrow$ 
   $c1 / st \setminus \setminus st' \# n1 \rightarrow$ 
  (WHILE  $b1$  DO  $c1$  END) /  $st' \setminus \setminus st'' \# n2 \rightarrow$ 
  (WHILE  $b1$  DO  $c1$  END) /  $st \setminus \setminus st'' \# S(n1 + n2)$ 

```

where " $c1 \text{ '}' st \text{ '}' \setminus \setminus st' \text{ '}' \# n$ " := (**ceval\_count**  $c1\ st\ st'\ n$ ).

Hint Constructors **ceval\_count**.

Theorem **ceval\_count\_complete**:  $\forall c\ st\ st',$

$c / st \setminus \setminus st' \rightarrow \exists n, c / st \setminus \setminus st' \# n.$

Proof. intros  $c\ st\ st'\ Heval.$

```

induction Heval;
  try inversion IHHeval1;
  try inversion IHHeval2;

```



try inversion *IHHeval*;  
eauto. Qed.

Theorem *ceval\_count\_sound*:  $\forall c \ st \ st' \ n,$   
 $c / st \ \backslash \backslash \ st' \ \# \ n \rightarrow c / st \ \backslash \backslash \ st'.$

Proof. intros *c st st' n Heval*. induction *Heval*; eauto. Qed.

Theorem *pe\_compare\_nil\_lookup*:  $\forall pe\_st1 \ pe\_st2,$   
 $pe\_compare \ pe\_st1 \ pe\_st2 = [] \rightarrow$   
 $\forall V, pe\_lookup \ pe\_st1 \ V = pe\_lookup \ pe\_st2 \ V.$

Proof. intros *pe\_st1 pe\_st2 H V*.  
apply (*pe\_compare\_correct pe\_st1 pe\_st2 V*).  
rewrite *H*. intro. inversion *H0*. Qed.

Theorem *pe\_compare\_nil\_update*:  $\forall pe\_st1 \ pe\_st2,$   
 $pe\_compare \ pe\_st1 \ pe\_st2 = [] \rightarrow$   
 $\forall st, pe\_update \ st \ pe\_st1 = pe\_update \ st \ pe\_st2.$

Proof. intros *pe\_st1 pe\_st2 H st*.  
apply *functional\_extensionality*. intros *V*.  
rewrite !*pe\_update\_correct*.  
apply *pe\_compare\_nil\_lookup* with (*V:=V*) in *H*.  
rewrite *H*. reflexivity. Qed.

Reserved Notation "*c' '/' pe\_st' '/' c'' '/' st '\ \ ' st'' '# ' n*"  
(at level 40, *pe\_st'* at level 39, *c''* at level 39,  
*st* at level 39, *st''* at level 39).

Inductive *pe\_ceval\_count* (*c':com*) (*pe\_st':pe\_state*) (*c'':com*)  
(*st:state*) (*st'':state*) (*n:nat*) : Prop :=

| *pe\_ceval\_count\_intro* :  $\forall st' \ n',$   
 $c' / st \ \backslash \backslash \ st' \rightarrow$   
 $c'' / pe\_update \ st' \ pe\_st' \ \backslash \backslash \ st'' \ \# \ n' \rightarrow$   
 $n' \leq n \rightarrow$   
 $c' / pe\_st' / c'' / st \ \backslash \backslash \ st'' \ \# \ n$   
where "*c' '/' pe\_st' '/' c'' '/' st '\ \ ' st'' '# ' n*" :=  
(*pe\_ceval\_count c' pe\_st' c'' st st'' n*).

Hint Constructors *pe\_ceval\_count*.

Lemma *pe\_ceval\_count\_le*:  $\forall c' \ pe\_st' \ c'' \ st \ st'' \ n \ n',$   
 $n' \leq n \rightarrow$   
 $c' / pe\_st' / c'' / st \ \backslash \backslash \ st'' \ \# \ n' \rightarrow$   
 $c' / pe\_st' / c'' / st \ \backslash \backslash \ st'' \ \# \ n.$

Proof. intros *c' pe\_st' c'' st st'' n n' Hle H*. inversion *H*.  
econstructor; try *eassumption*. omega. Qed.

Theorem *pe\_com\_complete*:

$\forall c \ pe\_st \ pe\_st' \ c' \ c'', c / pe\_st \ \backslash \backslash \ c' / pe\_st' / c'' \rightarrow$

```

  ∀ st st'' n,
  (c / pe_update st pe_st \\\ st'' # n) →
  (c' / pe_st' / c'' / st \\\ st'' # n).
Proof. intros c pe_st pe_st' c' c'' Hpe.
  induction Hpe; intros st st'' n Heval;
  try (inversion Heval; subst;
    try (rewrite → pe_bexp_correct, → H in *; solve by inversion);
    []);
  eauto.
- econstructor. econstructor.
  rewrite → pe_aexp_correct. rewrite ← pe_update_update_add.
  rewrite → H. apply E'Skip. auto.
- econstructor. econstructor. reflexivity.
  rewrite → pe_aexp_correct. rewrite ← pe_update_update_remove.
  apply E'Skip. auto.
-
  edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
  inversion Hskip. subst.
  edestruct IHHpe2. eassumption.
  econstructor; eauto. omega.
- inversion Heval; subst.
  + edestruct IHHpe1. eassumption.
    econstructor. apply E_lfTrue. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite ← assign_removes. eassumption. eassumption.
  + edestruct IHHpe2. eassumption.
    econstructor. apply E_lfFalse. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite → pe_compare_update.
    rewrite ← assign_removes. eassumption. eassumption.
-
  edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
  inversion Hskip. subst.
  edestruct IHHpe2. eassumption.
  econstructor; eauto. omega.
- inversion Heval; subst.
  + econstructor. apply E_lfFalse.
    rewrite ← pe_bexp_correct. assumption.
    apply eval_assign.
    rewrite ← assign_removes. inversion H2; subst; auto.
    auto.
  +

```

```

    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    econstructor. apply E_IfTrue.
    rewrite ← pe_bexp_correct. assumption.
    repeat eapply E_Seq; eauto. apply eval_assign.
    rewrite → pe_compare_update, ← assign_removes. eassumption.
    omega.
- exfalso.
  generalize dependent (S (n1 + n2)). intros n.
  clear - H H0 IHHpe1 IHHpe2. generalize dependent st.
  induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
  + rewrite pe_bexp_correct, H in H7. inversion H7.
  +
    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    rewrite ← (pe_compare_nil_update _ _ H0) in H7.
    apply H1 in H7; [| omega]. inversion H7.
- generalize dependent st.
  induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
  + rewrite pe_bexp_correct in H8. eauto.
  + rewrite pe_bexp_correct in H5.
    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    rewrite ← (pe_compare_nil_update _ _ H1) in H8.
    apply H2 in H8; [| omega]. inversion H8.
    econstructor; [ eapply E_WhileLoop; eauto | eassumption | omega].
Qed.

```

Theorem pe\_com\_sound:

$$\begin{aligned}
& \forall c \text{ pe\_st } \text{pe\_st}' \ c' \ c'', c / \text{pe\_st} \setminus\setminus c' / \text{pe\_st}' / c'' \rightarrow \\
& \forall st \ st'' \ n, \\
& (c' / \text{pe\_st}' / c'' / st \setminus\setminus st'' \# n) \rightarrow \\
& (c / \text{pe\_update } st \text{ pe\_st} \setminus\setminus st'').
\end{aligned}$$

Proof. intros c pe\_st pe\_st' c' c'' Hpe.

```

induction Hpe;
  intros st st'' n [st' n' Heval Heval' Hle];
  try (inversion Heval; [|; subst]);
  try (inversion Heval'; [|; subst]); eauto.
- rewrite ← pe_update_update_add. apply E_Ass.
  rewrite → pe_aexp_correct. rewrite → H. reflexivity.

```

```

- rewrite ← pe_update_update_remove. apply E_Ass.
  rewrite ← pe_aexp_correct. reflexivity.
- eapply E_Seq; eauto.
- apply E_IfTrue.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
  eapply IHHpe. eauto.
- apply E_IfFalse.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
  eapply IHHpe. eauto.
- inversion Heval; subst; inversion H7; subst; clear H7.
+
  eapply ceval_deterministic in H8; [| apply eval_assign]. subst.
  rewrite ← assign_removes in Heval'.
  apply E_IfTrue. rewrite → pe_bexp_correct. assumption.
  eapply IHHpe1. eauto.
+
  eapply ceval_deterministic in H8; [| apply eval_assign]. subst.
  rewrite → pe_compare_update in Heval'.
  rewrite ← assign_removes in Heval'.
  apply E_IfFalse. rewrite → pe_bexp_correct. assumption.
  eapply IHHpe2. eauto.
- apply E_WhileEnd.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
- apply E_WhileLoop.
  rewrite → pe_bexp_correct. rewrite → H. reflexivity.
  eapply IHHpe1. eauto. eapply IHHpe2. eauto.
- inversion Heval; subst.
+
  inversion H9. subst. clear H9.
  inversion H10. subst. clear H10.
  eapply ceval_deterministic in H11; [| apply eval_assign]. subst.
  rewrite → pe_compare_update in Heval'.
  rewrite ← assign_removes in Heval'.
  eapply E_WhileLoop. rewrite → pe_bexp_correct. assumption.
  eapply IHHpe1. eauto.
  eapply IHHpe2. eauto.
+ apply ceval_count_sound in Heval'.
  eapply ceval_deterministic in H9; [| apply eval_assign]. subst.
  rewrite ← assign_removes in Heval'.
  inversion H2; subst.
  × inversion Heval'. subst. apply E_WhileEnd.
    rewrite → pe_bexp_correct. assumption.

```

```

      × assumption.
- eapply ceval_count_sound. apply Heval'.
-
  apply loop_never_stops in Heval. inversion Heval.
-
  clear - H1 IHHpe1 IHHpe2 Heval.
  remember (WHILE pe_bexp pe_st b1 DO c1';; c2' END) as c'.
  induction Heval;
    inversion Heqc'; subst; clear Heqc'.
+ apply E_WhileEnd.
  rewrite pe_bexp_correct. assumption.
+
  assert (IHHeval2' := IHHeval2 (refl_equal _)).
  apply ceval_count_complete in IHHeval2'. inversion IHHeval2'.
  clear IHHeval1 IHHeval2 IHHeval2'.
  inversion Heval1. subst.
  eapply E_WhileLoop. rewrite pe_bexp_correct. assumption. eauto.
  eapply IHHpe2. econstructor. eassumption.
  rewrite ← (pe_compare_nil_update _ _ H1). eassumption. apply le_n.
Qed.

```

Corollary pe\_com\_correct:

$$\begin{aligned}
& \forall c \text{ pe\_st } \text{pe\_st}' \text{ c}', c / \text{pe\_st} \setminus \setminus c' / \text{pe\_st}' / \text{SKIP} \rightarrow \\
& \forall st \text{ st}'', \\
& (c / \text{pe\_update } st \text{ pe\_st} \setminus \setminus st'') \leftrightarrow \\
& (\exists st', c' / st \setminus \setminus st' \wedge \text{pe\_update } st' \text{ pe\_st}' = st'').
\end{aligned}$$

Proof. intros c pe\_st pe\_st' c' H st st''. split.

```

- intros Heval.
  apply ceval_count_complete in Heval. inversion Heval as [n Heval'].
  apply pe_com_complete with (st:=st) (st'':=st'') (n:=n) in H.
  inversion H as [? ? ? Hskip ?]. inversion Hskip. subst. eauto.
  assumption.
- intros [st' [Heval Heq]]. subst st''.
  eapply pe_com_sound in H. apply H.
  econstructor. apply Heval. apply E'Skip. apply le_n.

```

Qed.

End LOOP.

## 37.5 Partial Evaluation of Flowchart Programs

Instead of partially evaluating *WHILE* loops directly, the standard approach to partially evaluating imperative programs is to convert them into *flowcharts*. In other words, it turns

out that adding labels and jumps to our language makes it much easier to partially evaluate. The result of partially evaluating a flowchart is a residual flowchart. If we are lucky, the jumps in the residual flowchart can be converted back to *WHILE* loops, but that is not possible in general; we do not pursue it here.

### 37.5.1 Basic blocks

A flowchart is made of *basic blocks*, which we represent with the inductive type **block**. A basic block is a sequence of assignments (the constructor **Assign**), concluding with a conditional jump (the constructor **If**) or an unconditional jump (the constructor **Goto**). The destinations of the jumps are specified by *labels*, which can be of any type. Therefore, we parameterize the **block** type by the type of labels.

```
Inductive block (Label:Type) : Type :=
| Goto : Label → block Label
| If : bexp → Label → Label → block Label
| Assign : id → aexp → block Label → block Label.
```

Arguments Goto {Label} ..

Arguments If {Label} - - ..

Arguments Assign {Label} - - ..

We use the "even or odd" program, expressed above in Imp, as our running example. Converting this program into a flowchart turns out to require 4 labels, so we define the following type.

```
Inductive parity_label : Type :=
| entry : parity_label
| loop : parity_label
| body : parity_label
| done : parity_label.
```

The following **block** is the basic block found at the **body** label of the example program.

```
Definition parity_body : block parity_label :=
  Assign Y (AMinus (Ald Y) (ANum 1))
    (Assign X (AMinus (ANum 1) (Ald X))
      (Goto loop)).
```

To evaluate a basic block, given an initial state, is to compute the final state and the label to jump to next. Because basic blocks do not *contain* loops or other control structures, evaluation of basic blocks is a total function – we don't need to worry about non-termination.

```
Fixpoint keval {L:Type} (st:state) (k : block L) : state × L :=
  match k with
  | Goto l ⇒ (st, l)
  | If b l1 l2 ⇒ (st, if beval st b then l1 else l2)
  | Assign i a k ⇒ keval (t_update st i (aeval st a)) k
```

end.

Example keval\_example:

```
keval empty_state parity_body
= (t_update (t_update empty_state Y 0) X 1, loop).
```

Proof. reflexivity. Qed.

### 37.5.2 Flowchart programs

A flowchart program is simply a lookup function that maps labels to basic blocks. Actually, some labels are *halting states* and do not map to any basic block. So, more precisely, a flowchart program whose labels are of type  $L$  is a function from  $L$  to **option** (**block**  $L$ ).

Definition program ( $L$ :Type) : Type :=  $L \rightarrow$  **option** (**block**  $L$ ).

Definition parity : program **parity\_label** := fun  $l \Rightarrow$

```
match  $l$  with
| entry  $\Rightarrow$  Some (Assign X (ANum 0) (Goto loop))
| loop  $\Rightarrow$  Some (If (BLe (ANum 1) (Ald Y)) body done)
| body  $\Rightarrow$  Some parity_body
| done  $\Rightarrow$  None
end.
```

Unlike a basic block, a program may not terminate, so we model the evaluation of programs by an inductive relation **peval** rather than a recursive function.

Inductive **peval** { $L$ :Type} ( $p$  : program  $L$ )

: state  $\rightarrow L \rightarrow$  state  $\rightarrow L \rightarrow$  Prop :=

```
| E_None:  $\forall$   $st$   $l$ ,
   $p$   $l$  = None  $\rightarrow$ 
  peval  $p$   $st$   $l$   $st$   $l$ 
| E_Some:  $\forall$   $st$   $l$   $k$   $st'$   $l'$   $st''$   $l''$ ,
   $p$   $l$  = Some  $k$   $\rightarrow$ 
  keval  $st$   $k$  = ( $st'$ ,  $l'$ )  $\rightarrow$ 
  peval  $p$   $st'$   $l'$   $st''$   $l''$   $\rightarrow$ 
  peval  $p$   $st$   $l$   $st''$   $l''$ .
```

Example parity\_eval: **peval** parity empty\_state entry empty\_state done.

Proof. *erewrite* **f\_equal** with ( $f :=$  fun  $st \Rightarrow$  **peval** \_ \_  $st$  \_).

eapply E\_Some. reflexivity. reflexivity.

eapply E\_Some. reflexivity. reflexivity.

apply E\_None. reflexivity.

apply **functional\_extensionality**. intros  $i$ . rewrite  $t\_update\_same$ ; auto.

Qed.

### 37.5.3 Partial evaluation of basic blocks and flowchart programs

Partial evaluation changes the label type in a systematic way: if the label type used to be  $L$ , it becomes  $\text{pe\_state} \times L$ . So the same label in the original program may be unfolded, or blown up, into multiple labels by being paired with different partial states. For example, the label `loop` in the parity program will become two labels:  $([(X,0)], \text{loop})$  and  $([(X,1)], \text{loop})$ . This change of label type is reflected in the types of `pe_block` and `pe_program` defined presently.

```

Fixpoint pe_block {L:Type} (pe_st:pe_state) (k : block L)
: block (pe_state  $\times$  L) :=
match k with
| Goto l  $\Rightarrow$  Goto (pe_st, l)
| If b l1 l2  $\Rightarrow$ 
  match pe_bexp pe_st b with
  | BTrue  $\Rightarrow$  Goto (pe_st, l1)
  | BFalse  $\Rightarrow$  Goto (pe_st, l2)
  | b'  $\Rightarrow$  If b' (pe_st, l1) (pe_st, l2)
  end
| Assign i a k  $\Rightarrow$ 
  match pe_aexp pe_st a with
  | ANum n  $\Rightarrow$  pe_block (pe_add pe_st i n) k
  | a'  $\Rightarrow$  Assign i a' (pe_block (pe_remove pe_st i) k)
  end
end.

```

Example `pe_block_example`:

```

pe_block [(X,0)] parity_body
= Assign Y (AMinus (Ald Y) (ANum 1)) (Goto [(X,1)], loop)).

```

Proof. reflexivity. Qed.

Theorem `pe_block_correct`:  $\forall (L:\text{Type}) \text{ st } pe\_st \text{ k } st' \text{ pe\_st}' (l':L),$   
 $\text{keval st (pe\_block pe\_st k)} = (st', (pe\_st', l')) \rightarrow$   
 $\text{keval (pe\_update st pe\_st) k} = (\text{pe\_update st' pe\_st}', l').$

Proof. intros. generalize dependent `pe_st`. generalize dependent `st`.

```

induction k as [l | b l1 l2 | i a k];
  intros st pe_st H.
- inversion H; reflexivity.
-
  replace (keval st (pe_block pe_st (If b l1 l2)))
    with (keval st (If (pe_bexp pe_st b) (pe_st, l1) (pe_st, l2)))
    in H by (simpl; destruct (pe_bexp pe_st b); reflexivity).
  simpl in *. rewrite pe_bexp_correct.
  destruct (beval st (pe_bexp pe_st b)); inversion H; reflexivity.
-
  simpl in *. rewrite pe_aexp_correct.

```



```

destruct (pe_aexp pe_st a); simpl;
  try solve [rewrite pe_update_update_add; apply IHk; apply H];
  solve [rewrite pe_update_update_remove; apply IHk; apply H].
Qed.

Definition pe_program {L:Type} (p : program L)
: program (pe_state × L) :=
  fun pe_l ⇒ match pe_l with (pe_st, l) ⇒
    option_map (pe_block pe_st) (p l)
  end.

Inductive pe_peval {L:Type} (p : program L)
(st:state) (pe_st:pe_state) (l:L) (st'o:state) (l':L) : Prop :=
| pe_peval_intro : ∀ st' pe_st',
  peval (pe_program p) st (pe_st, l) st' (pe_st', l') →
  pe_update st' pe_st' = st'o →
  pe_peval p st pe_st l st'o l'.

Theorem pe_program_correct:
  ∀ (L:Type) (p : program L) st pe_st l st'o l',
  peval p (pe_update st pe_st) l st'o l' ↔
  pe_peval p st pe_st l st'o l'.

Proof. intros.
  split.
- intros Heval.
  remember (pe_update st pe_st) as sto.
  generalize dependent pe_st. generalize dependent st.
  induction Heval as
  [ sto l Hlookup | sto l k st'o l' st''o l'' Hlookup Hkeval Heval ];
  intros st pe_st Heqsto; subst sto.
+ eapply pe_peval_intro. apply E_None.
  simpl. rewrite Hlookup. reflexivity. reflexivity.
+
  remember (keval st (pe_block pe_st k)) as x.
  destruct x as [st' [pe_st' l'_]].
  symmetry in Heqx. erewrite pe_block_correct in Hkeval by apply Heqx.
  inversion Hkeval. subst st'o l'_ . clear Hkeval.
  edestruct IHHeval. reflexivity. subst st''o. clear IHHeval.
  eapply pe_peval_intro; [| reflexivity]. eapply E_Some; eauto.
  simpl. rewrite Hlookup. reflexivity.
- intros [st' pe_st' Heval Heqst'o].
  remember (pe_st, l) as pe_st_l.
  remember (pe_st', l') as pe_st'_l'.
  generalize dependent pe_st. generalize dependent l.
  induction Heval as

```

```

[ st [pe_st_ l_] Hlookup
| st [pe_st_ l_] pe_k st' [pe_st'_ l'_] st'' [pe_st'' l'']
  Hlookup Hkeval Heval ];
intros l pe_st Heqpe_st_l;
inversion Heqpe_st_l; inversion Heqpe_st'_l'; repeat subst.
+ apply E_None. simpl in Hlookup.
  destruct (p l'); [ solve [ inversion Hlookup ] | reflexivity ].
+
  simpl in Hlookup. remember (p l) as k.
  destruct k as [k|]; inversion Hlookup; subst.
  eapply E_Some; eauto. apply pe_block_correct. apply Hkeval.
Qed.

```

*Date* : 2016 – 03 – 07 14 : 18 : 04 – 0500 (*Mon, 07 Mar 2016*)

# Chapter 38

## Library Postscript

### 38.1 Postscript

### 38.2 Looking back...

- *Functional programming* - "declarative" programming (recursion over persistent data structures)
  - higher-order functions
  - polymorphism
- *Logic*, the mathematical basis for software engineering:

$$\begin{array}{ccc} \text{logic} & & \text{calculus} \\ \hline \text{software engineering} & = & \text{mechanical/civil engineering} \end{array}$$

- inductively defined sets and relations
  - inductive proofs
  - proof objects
- *Coq*, an industrial-strength proof assistant
  - functional core language
  - core tactics
  - automation
- *Foundations of programming languages*
  - notations and definitional techniques for precisely specifying

- \* abstract syntax
- \* operational semantics
  - big-step style
  - small-step style
- \* type systems
- program equivalence
- Hoare logic
- fundamental metatheory of type systems
  - \* progress and preservation
- theory of subtyping

## 38.3 Looking forward...

Some good places to go for more...

- Several optional chapters of *Software Foundations*
- Cutting-edge conferences on programming languages and formal verification:
  - POPL
  - PLDI
  - OOPSLA
  - ICFP
  - CAV
  - (and many others)
- More on functional programming
  - Learn You a Haskell for Great Good, by Miran Lipovaca (ebook)
  - and many other texts on Haskell, OCaml, Scheme, Scala, ...
- More on Hoare logic and program verification
  - The Formal Semantics of Programming Languages: An Introduction, by Glynn Winskel. MIT Press, 1993.
  - Many practical verification tools, e.g. Microsoft’s Boogie system, Java Extended Static Checking, etc.

- More on the foundations of programming languages:
  - Types and Programming Languages, by Benjamin C. Pierce. MIT Press, 2002.
  - Practical Foundations for Programming Languages, by Robert Harper. Forthcoming from MIT Press. Manuscript available from his web page.
  - Foundations for Programming Languages, by John C. Mitchell. MIT Press, 1996.
- More on Coq:
  - Certified Programming with Dependent Types, by Adam Chlipala. A draft textbook on practical proof engineering with Coq, available from his web page.
  - Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran. Springer-Verlag, 2004.
  - Iron Lambda (<http://iron.ouroborus.net/>) is a collection of Coq formalisations for functional languages of increasing complexity. It fills part of the gap between the end of the Software Foundations course and what appears in current research papers. The collection has at least Progress and Preservation theorems for a number of variants of STLC and the polymorphic lambda-calculus (System F)